Am486® Microprocessor Software User's Manual

Rev. 1, 1994

ADVANCED MICRO DEVICES

Advanced Micro Devices reserves the right to make changes in its products
without notice in order to improve design or performance characteristics.

This publication neither states nor implies any warranty of any kind, including but not limited to implied warrants of merchantability or fitness for a particular application. AMD® assumes no responsibility for the use of any circuitry other than the circuitry in an AMD product.

The information in this publication is believed to be accurate in all respects at the time of publication, but is subject to change without notice. AMD assumes no responsibility for any errors or omissions, and disclaims responsibility for any consequences resulting from the use of the information included herein. Additionally, AMD assumes no responsibility for the functioning of undescribed features or parameters.

Trademarks

INTRODUCTION

The Am486® Microprocessor Software User’s Manual is designed to support system software engineers developing BIOS and application software for use with products from the Am486 microprocessor family. Because, typically, such engineers are already familiar with basic personal computer system programming requirements, this book focuses on providing information about the basic processor instruction set and the programmable registers.

Each chapter begins with an overview diagram of registers or instructions organized by operational category with cross-references to the detailed description page for each item. The detailed descriptions are listed alphabetically on the subsequent pages in the chapter.

Supplementary information is provided in Appendices A through J. A glossary of terms is included after the appendices. For convenience, a basic ASCII cross-reference is on the inside back cover of this manual.

TABLE OF CONTENTS

Appendices

A General Guidelines for Programming

B Opcode Map

C Flag Cross-Reference

D Condition Codes

E Instruction Format and Timing

F Numeric Exception Summary

G Code Optimization

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

The Am486® Microprocessor Software User's Manual is designed to support system software engineers developing BIOS and application software for use with products from the Am486 microprocessor family. Because, typically, such engineers are already familiar with basic personal computer system programming requirements, this book focuses on providing information about the basic processor instruction set and the programmable registers.

Each chapter begins with an overview diagram of registers or instructions organized by operational category with cross-references to the detailed description page for each item. The detailed descriptions are listed alphabetically on the subsequent pages in the chapter.

Supplementary information is provided in Appendices A through J. A glossary of terms is included after the appendices. For convenience, a basic ASCII cross-reference is on the inside back cover of this manual.

LIST OF FIGURES

LIST OF TABLES

CHAPTER

1 Am486 MICROPROCESSOR REGISTER SET

1.1 OVERVIEW

The Am486 Microprocessor Register Set includes the same basic system architecture as other 486-based microprocessors. Page 1-2 provides a roadmap to these registers using functional categories. For each register, the roadmap lists the page on which the detailed register description appears. In the detailed description section that follows the Am486 microprocessor register roadmap, the registers appear in alphabetical order using the name listed in the roadmap.

1.2 DETAILED REGISTER DESCRIPTIONS

Register descriptions begin on page 1-3, using the following format:

Register Name/s **General Description** **Bit Size**

Bit(s)	Bit Set Name	Description
nn xx	XXX	Function

Addressing

Description of register addressing method.

Default Value

Factory/default register setting.

Functional Description

Verbal description of register function by bit or bit set.

Note: *Standard compiler programs convert the register names into opcode. This chapter references the registers by name. Appendix E includes the opcodes used to address the registers as part of the 'Instruction Format and Timing' descriptions.*

Am486 Microprocessor Register Roadmap

General

AH	1-3
AL	1-4
AX	1-5
BH	1-6
BL	1-7
BP	1-8
BX	1-9
CH	1-10
CL	1-11
CX	1-18
DH	1-19
DI	1-20
DL	1-21
DX	1-32
EAX	1-33
EBP	1-34
EBX	1-35
ECX	1-36
EDI	1-37
EDX	1-38
ESI	1-42
ESP	1-43
SI	1-58
SP	1-59

Segment

CS	1-17
DS	1-31
ES	1-41
FS	1-51
GS	1-53
SS	1-60

Status and Control

EFLAGS	1-39
EIP	1-40
FLAGS	1-44
IP	1-55

Memory Management

GDTR	1-52
IDTR	1-54
LDTR	1-56
TR	1-61

Debug

DR0	1-22
DR1	1-23
DR2	1-24
DR3	1-25
DR4	1-26
DR5	1-27
DR6	1-28
DR7	1-29

Test

TR3	1-62
TR4	1-63
TR5	1-64
TR6	1-65
TR7	1-66

FPU

CR0	1-12
CR1	1-14
CR2	1-15
CR3	1-16
FPUCR	1-45
FPUDP	1-46
FPUIP	1-47
FPUSR	1-48
FPUTWR	1-50
R0	1-57
R1	1-57
R2	1-57
R3	1-57
R4	1-57
R5	1-57
R6	1-57
R7	1-57

1.3 AH Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	AH Register	Processor general register, High byte of AX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.4 AL Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	AL Register	Processor general register, Low byte of AX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.5 AX Processor General Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	AX Register	Processor general register, Low word of EAX; see also AL, AH.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations.

1.6 BH Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	BH Register	Processor general register, High byte of BX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.7 BL Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	BL Register	Processor general register, Low byte of BX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.8 BP Processor General Register/Base Pointer 16 bits

Bit(s)	Bit Set Name	Description
15–0	BP Register	Processor general register, base pointer register.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations. When using 16-bit addressing, you can copy the stack pointer (SP — see page 1-59) into BP before pushing anything onto the stack, and access data structures using fixed offsets from the BP value.

1.9 BX Processor General Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	BX Register	Processor general register, Low word of EBX; see also BH and BL.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations.

1.10 CH Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	CH Register	Processor general register, High byte of CX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.11 CL Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	CL Register	Processor general register, Low byte of CX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.12 CR0 Control Register 0 32 bits

Bit(s)	Bit Set Name	Description
31	PG	0 = Paging disabled. 1 = Paging enabled.
30	CD	0 = Internal cache enabled. 1 = Internal cache disabled.
29	NW	0 = Enables write-throughs and invalidation cycles. 1 = Disables write-throughs and invalidation cycles.
28–19	N/A	Reserved
18	AM	0 = Alignment checking disabled. 1 = Alignment checking allowed.
17	N/A	Reserved
16	WP	0 = Supervising process can write read-only user-level pages. 1 = User-level pages protected against supervisor mode access.
15–6	N/A	Reserved
5	NE	0 = No error since last clear. 1 = Numeric error occurred.
4	ET	0 = No 387 coprocessor support. 1 = 387 coprocessor support.
3	TS	0 = No task switch since last clear. 1 = Task switched.
2	EM	0 = No emulation. 1 = Numeric emulation.
1	MP	0 = No coprocessor. 1 = Coprocessor present.
0	PE	0 = No protection. 1 = Segment level protection.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

CR0 configures several system level controls, as follows:

- **PG** (bit 31) enables paging when set and disables paging when clear.
- **CD** (bit 30) enables the internal cache when clear and disables the cache when set. Cache misses do not cause cache line fills when the bit is set. Cache hits are not disabled; you must flush the cache to disable it completely.
- **NW** (bit 29) enables write-throughs and cache invalidation cycles when clear and disables invalidation cycles and write-throughs when hit in the cache when set. Disabling write-throughs can allow stale data to appear in the cache.
- **AM** (bit 18) allows alignment checking when set and disables alignment checking when clear. Alignment checking occurs only when this bit is set, the AC flag is set, and CPL is 3 (user mode).

- **WP** (bit 16) protects user-level pages against supervisor-mode access when set. When clear, a supervisor process can write read-only user-level pages. This feature is useful for implementing the copy-on-write method of creating a new process (forking) used by some operating systems, such as UNIX.
- **NE** (bit 5) enables the standard mechanism for reporting floating-point errors when set. When NE is clear and the $\overline{\text{IGNNE}}$ input is active, numeric errors are ignored. When NE is set and $\overline{\text{IGNNE}}$ is inactive, a numeric error causes the processor to stop and wait for an interrupt from the $\overline{\text{FERR}}$ pin.
- **ET** (bit 4) is set to support 387 coprocessor functions.
- **TS** (bit 3) is set whenever a task switch occurs. The processor checks this bit when interpreting floating-point arithmetic instructions to allow delaying save/restore of numeric content until the numeric data is actually used. The CLTS instruction clears this bit.
- **EM** (bit 2) is used when set (along with TS) to generate a coprocessor-not-available exception when a WAIT or numeric instruction is executed. EM can be set to cause exception 7 on any WAIT or numeric instruction. When clear, the bit does not cause the exception.
- **MP** (bit 1) indicates, when set, that a coprocessor is present. When clear, the floating-point capability is not present.
- **PE** (bit 0) enables segment-level protection when set. Clearing this bit removes the protection.

The remaining bits are undefined and reserved.

1.13 CR1 Control Register 1 32 bits

Bit(s)	Bit Set Name	Description
31–0	CR1	Reserved

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

Register reserved

1.14 CR2 Control Register 2 32 bits

Bit(s)	Bit Set Name	Description
31–0	CR2	Page fault linear address

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

When an exception occurs during paging, CR2 stores the 32-bit linear address that caused the exception.

1.15 CR3 Control Register 3 32 bits

Bit(s)	Bit Set Name	Description
31–12	PDBR	Page directory base register contains the 20 most significant bits of the page directory (first-level page table) address.
11–5	N/A	Reserved
4	PCD	Page-level cache disable bit. 1=Paging disabled; 0=Paging enabled.
3	PWT	Page-level writes transparent. 1=Write-through to external cache enabled; 2=Write-through disabled.
2–0	N/A	Reserved

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

CR3 configures some of the page-level controls, as follows:

- **PDBR** (bits 31–12) is the page directory table address system control register. It contains the 20 most significant bits of the page directory (first-level page table) address. Because the page directory must be aligned to a page boundary, the lower 12 address bits are ignored.
- **PCD** (bit 4) is driven on the PCD pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. It is driven on all bus cycles when paging is not enabled. The PCD pin is one of the write-through cache controls for external cache and is used on a cycle-by-cycle basis.
- **PWT** (bit 3) is driven on the PWT pin during bus cycles that are not paged, such as interrupt acknowledge cycles, when paging is enabled. It is driven on all bus cycles when paging is not enabled. The PWT pin is one of the write-through cache controls for external cache and is used on a cycle-by-cycle basis.

Bits 11–5 and 2–0 are undefined and are reserved.

1.16 CS Code Segment Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	CS	Code segment register holds the base address for the code segment of memory, that area containing the instructions being executed.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The processor organizes memory into segments as one of the possible ways to access memory. There are six segments (tables within memory) accessed through the segment registers. Each register stores the base address for its segment. The segment containing the instructions being executed is called the code segment. Its segment selector (base address) is stored in the CS register. The processor fetches instructions from the code segment, using the contents of the EIP or IP register as an offset into the segment. The CS register value changes as a result of interrupts, exceptions, and instructions that transfer control between segments (see CALL, IRET, and JMP instructions).

1.17 CX Processor General Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	CX Register	Processor general register, Low word of ECX; see also CL, CH.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations.

1.18 DH Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	DH Register	Processor general register, High byte of DX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.19 DI Processor General Register — Data Index 16 bits

Bit(s)	Bit Set Name	Description
15–0	DI	Processor general register; used as a destination index for string operations.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Descriptions

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations. For string operations the DI register points to destination operands and increments or decrements between operations, depending on the DF setting in the EFLAGS register (see page 1-39). The DI register can only point to operands in the memory space specified by the ES segment register.

1.20 DL Processor General Register 8 bits

Bit(s)	Bit Set Name	Description
7–0	AL Register	Processor general register, Low byte of DX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 8-bit general processor registers used to hold 8-bit operands for logical and arithmetic operations.

1.21 DR0 Linear Breakpoint Address 0 Debug Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	DR0	Stores the address of a debug breakpoint.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

Access the built-in debugging features of the microprocessor through the eight Debug Registers. The linear breakpoint address registers (DR0 to DR3) store addresses for as many as four breakpoints. These breakpoints invoke debugging software. Whenever an operation accesses one of these addresses, it generates an exception that initiates the referenced debugging subroutine. You must specify the form of memory access that triggers the breakpoint; for example, select an instruction fetch or a doubleword write operation. The debug registers support instruction breakpoints and data breakpoints.

1.22 DR1 Linear Breakpoint Address 1 Debug Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	DR1	Stores the address of a debug breakpoint.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

Access the built-in debugging features of the microprocessor through the eight Debug Registers. The linear breakpoint address registers (DR0 to DR3) store addresses for as many as four breakpoints. These breakpoints invoke debugging software. Whenever an operation accesses one of these addresses, it generates an exception that initiates the referenced debugging subroutine. You must specify the form of memory access that triggers the breakpoint; for example, select an instruction fetch or a doubleword write operation. The debug registers support instruction breakpoints and data breakpoints.

1.23 DR2 Linear Breakpoint Address 2 Debug Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	DR2	Stores the address of a debug breakpoint.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

Access the built-in debugging features of the microprocessor through the eight Debug Registers. The linear breakpoint address registers (DR0 to DR3) store addresses for as many as four breakpoints. These breakpoints invoke debugging software. Whenever an operation accesses one of these addresses, it generates an exception that initiates the referenced debugging subroutine. You must specify the form of memory access that triggers the breakpoint; for example, select an instruction fetch or a doubleword write operation. The debug registers support instruction breakpoints and data breakpoints.

1.24 DR3 Linear Breakpoint Address 3 Debug Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	DR3	Stores the address of a debug breakpoint.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

Access the built-in debugging features of the microprocessor through the eight Debug Registers. The linear breakpoint address registers (DR0 to DR3) store addresses for as many as four breakpoints. These breakpoints invoke debugging software. Whenever an operation accesses one of these addresses, it generates an exception that initiates the referenced debugging subroutine. You must specify the form of memory access that triggers the breakpoint; for example, select an instruction fetch or a doubleword write operation. The debug registers support instruction breakpoints and data breakpoints.

1.25 DR4 Debug Register 4 32 bits

Bit(s)	Bit Set Name	Description
31–0	DR4	Reserved

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

Not currently used.

1.26 DR5 Debug Register 5 32 bits

Bit(s)	Bit Set Name	Description
31–0	DR5	Reserved

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

Not currently used.

1.27 DR6 Breakpoint Status Debug Register 32 bits

Bit(s)	Bit Set Name	Description
31–16	N/A	Reserved, always 0000 0000 0000 0000
15	BT	0 = Default, no setting condition detected. 1 = Switch to task with TSS that has debug trap bit (T) set.
14	BS	0 = Default, no setting condition detected. 1 = Trap flag (TF) set.
13	BD	0 = Default, no setting condition detected. 1 = Next instruction reads or writes a debug register that is in use by in-circuit emulation.
12–4	N/A	Reserved, always 0 0000 0000
3	B3	0 = No debug exception generated for breakpoint 3. 1 = Debug exception generated for breakpoint 3.
2	B2	0 = No debug exception generated for breakpoint 2. 1 = Debug exception generated for breakpoint 2.
1	B1	0 = No debug exception generated for breakpoint 1. 1 = Debug exception generated for breakpoint 1.
0	B0	0 = No debug exception generated for breakpoint 0. 1 = Debug exception generated for breakpoint 0.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The Breakpoint Status Debug Register stores the current breakpoint exception status. If an exception occurs, read this register to determine which breakpoint caused the exception, or whether one of three other possible triggering events occurred.

- The BT bit (15) indicates when the exception was generated by switching to a task for which the TSS had the T bit (debug trap) set.
- The BS bit (14) indicates whether the trap flag (TF) in the EFLAGS register is set.
- The BD bit (13) indicates that the debug registers are in use by in-circuit emulation and that the next instruction writes to or reads from one of the registers.
- B3 (bit 3), B2 (bit 2), B1 (bit 1), and B0 (bit 0) specify, when set, that the specified breakpoint exception occurred.

Note: *The processor never clears the contents of TR6. When writing a debug handler routine, always make sure that the program clears TR6 before returning,*

1.28 DR7 Breakpoint Control Debug Register 32 bits

Bit(s)	Bit Set Name	Description
31–30	LEN3	00 = Breakpoint 3 is one byte. 01 = Breakpoint 3 is word (two bytes). 10 = Reserved, undefined. 11 = Breakpoint 3 is doubleword (four bytes).
29–28	R/W3	00 = Breakpoint 3 breaks on instruction execution only. 01 = Breakpoint 3 breaks on data writes only. 10 = Reserved, undefined. 11 = Breakpoint 3 breaks on data reads or writes, but not instructions.
27–26	LEN2	00 = Breakpoint 2 is one byte. 01 = Breakpoint 2 is word (two bytes). 10 = Reserved, undefined. 11 = Breakpoint 2 is doubleword (four bytes).
25–24	R/W2	00 = Breakpoint 2 breaks on instruction execution only. 01 = Breakpoint 2 breaks on data writes only. 10 = Reserved, undefined. 11 = Breakpoint 2 breaks on data reads or writes, but not instructions.
23–22	LEN1	00 = Breakpoint 1 is one byte. 01 = Breakpoint 1 is word (two bytes). 10 = Reserved, undefined. 11 = Breakpoint 1 is doubleword (four bytes).
21–20	R/W1	00 = Breakpoint 1 breaks on instruction execution only. 01 = Breakpoint 1 breaks on data writes only. 10 = Reserved, undefined. 11 = Breakpoint 1 breaks on data reads or writes, but not instructions.
19–18	LEN0	00 = Breakpoint 0 is one byte. 01 = Breakpoint 0 is word (two bytes). 10 = Reserved, undefined. 11 = Breakpoint 0 is doubleword (four bytes).
17–16	R/W0	00 = Breakpoint 0 breaks on instruction execution only. 01 = Breakpoint 0 breaks on data writes only. 10 = Reserved, undefined. 11 = Breakpoint 0 breaks on data reads or writes, but not instructions.
15–10	N/A	Reserved, always 0000 00
9	GE	Global enable, not used.
8	LE	Local enable, not used.
7	G3	0 = Global disable of breakpoint 3. 1 = Global enable of breakpoint 3.
6	L3	0 = Local disable of breakpoint 3. 1 = Local enable of breakpoint 3.
5	G2	0 = Global disable of breakpoint 2. 1 = Global enable of breakpoint 2.
4	L2	0 = Local disable of breakpoint 2. 1 = Local enable of breakpoint 2.
3	G1	0 = Global disable of breakpoint 1. 1 = Global enable of breakpoint 1.
2	L1	0 = Local disable of breakpoint 1. 1 = Local enable of breakpoint 1.
1	G0	0 = Global disable of breakpoint 0. 1 = Global enable of breakpoint 0.
0	L0	0 = Local disable of breakpoint 0. 1 = Local enable of breakpoint 0.

Addressing

Specify by name as instruction operand.

Default Value

00000000h

Functional Description

The Debug Control Register (DR7) configures the breakpoints. The High word of the register defines for each breakpoint, the type of breakpoint it is (R/W3, R/W2, R/W1, and R/W0) and the length of each field (LEN3, LEN2, LEN1, and LEN0).

Note: *For each LENn and R/Wn pair, if the breakpoint is defined as an instruction breakpoint (R/Wn = 00), set LENn = 00. The instruction break is only defined for byte lengths; the operation of an instruction break with any other length is undefined.*

The lowest byte of DR7 allows enabling or disabling of the breakpoints at one or two levels: global (G3, G2, G1, G0) or local (L3, L2, L1, L0). If a breakpoint is enabled at the global level (Gn=1), it is enabled for all operations. If a breakpoint is disabled at the global level, it can still be enabled for a single task with the local enable bit Ln. This acts as a temporary enable that exists while the specified task runs. When a task switch occurs, it resets the Ln enable bit for the associated breakpoint.

1.29 DS Data Segment Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	DS	A segment register that holds the base address for one of the four data segments of memory, available to the program currently executing.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The processor organizes memory into segments as one of the possible ways to access memory. There are six segments (tables within memory) accessed through the segment registers. Each register stores the base address for its segment. There are four data segments that can contain data used by a program being executed. The segment selectors (base addresses) for these segments are stored in the DS, ES, FS, and GS registers. The processor fetches data from a data segment, using an offset into the segment. The data segment register value changes as a result of interrupts, exceptions, and instructions that transfer control between segments (see CALL, IRET, and JMP instructions).

1.30 DX Processor General Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	DX Register	Processor general register, Low word of EDX; see also DL, DH.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations.

1.31 EAX Processor General Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	EAX	Processor general register; see also AX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations.

1.32 EBP Processor General Register — Base Pointer 32 bits

Bit(s)	Bit Set Name	Description
31–0	EBP	Processor general register; base pointer register; see also BP.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations. When using 32-bit addressing, copy the stack pointer (ESP — see page 1-43) into EBP before pushing anything onto the stack, and access data structures using fixed offsets from the EBP value.

1.33 EBX Processor General Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	EBX	Processor general register; see also BX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations.

1.34 ECX Processor General Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	ECX	Processor general register; see also CX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations.

1.35 EDI Processor General Register — Data Index 32 bits

Bit(s)	Bit Set Name	Description
31–0	EDI	Processor general register; data index register, used as a 32-bit destination index for string operations; see also DI.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations. For string operations, the EDI register points to destination operands and increments or decrements between operations, depending on the DF setting in the EFLAGS register (see page 1-39). The EDI register can only point to operands in the memory space specified by the ES segment register.

1.36 EDX Processor General Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	EDX	Processor general register; see also DX.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations.

1.37 EFLAGS Extended Flags Register 32 bits

Bit(s)	Bit Set Name	Description
31–19	N/A	Reserved, always 0000 0000 0000 0
18	AC	0 = Alignment Check mode not enabled. 1 = Alignment Check mode enabled.
17	VM	0 = Normal processing mode. 1 = Virtual-8086 mode.
16	RF	0 = Normal operation. 1 = Debug exceptions disabled to allow debugger program to run without causing another exception, Resume Flag set.
15	N/A	Reserved, always 0
14	NT	0 = Current task is not nested below another task. 1 = Current task is nested below another task.
13–12	IOPL	00 = Highest I/O access privilege level; typically operating system. 01 = Second highest I/O access privilege level; system services. 10 = Third highest I/O access privilege level; system services. 11 = Lowest I/O access privilege level; application software.
11	OF	0 = Arithmetic result within limits. 1 = Arithmetic result not in positive/negative range, Overflow Flag set.
10	DF	0 = Forward direction, addressing increments. 1 = Backward direction, addressing decrements, Direction Flag set.
9	IF	0 = Maskable interrupts disabled. 1 = Maskable interrupts enabled, Interrupt Flag set.
8	TF	0 = Normal operation. 1 = Trap Flag set, processor enters single-step mode for debugging; each instruction generates a debug exception.
7	SF	0 = Arithmetic result is not negative (≥0); sign is +. 1 = Arithmetic result is negative (<0); Sign Flag set, sign is –.
6	ZF	0 = Arithmetic result is not zero. 1 = Arithmetic result is zero, Zero Flag set.
5	N/A	Reserved, always 0
4	AF	0 = No BCD carry. 1 = BCD carry from bit position 3, Auxiliary Flag set.
3	N/A	Reserved, always 0
2	PF	0 = Result Low byte has odd parity. 1 = Result Low byte has even parity, Parity Flag set.
1	N/A	Reserved, always 1
0	CF	0 = No carry from MSB of result. 1 = Carry from MSB of result, Carry Flag set.

Addressing

Specify by bit/set names or by using the special flag instructions (BT, BTR, BTS, CLC, CLD, LAHF, POPF, POPFD, PUSHF, PUSHFD, SAHF, STC, STD, STI) described in Chapter 2.

Default Value

00000002h

Functional Description

The 32-bit EFLAGS register has system flags, status flags, and a control flag.

1.38 EIP Extended Instruction Pointer Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	EIP	Extended Instruction Pointer, the offset that points to the next instruction within the current code segment.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The EIP register contains the 32-bit offset that points to the next instruction within the current code segment. The control-transfer instructions, such as JUMP or RET, and interrupts and exceptions control the contents of this register implicitly. The contents of this register advance from one instruction boundary to the next. Because of instruction prefetching, its value is only an approximate indication of the bus activity loading instructions into the processor. The IP register (see page 1-55) is the lower word of the EIP register.

1.39 ES Data Segment Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	ES	A segment register that holds the base address for one of the four data segments of memory, available to the program currently executing.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The processor organizes memory into segments as one of the possible ways to access memory. There are six segments (tables within memory) accessed through the segment registers. Each register stores the base address for its segment. There are four data segments that can contain data used by a program being executed. The segment selectors (base addresses) for these segments are stored in the DS, ES, FS, and GS registers. The processor fetches data from a data segment, using an offset into the segment. The data segment register value changes as a result of interrupts, exceptions, and instructions that transfer control between segments (see CALL, IRET, and JMP instructions).

1.40 ESI Processor General Register — Stack Index 32 bits

Bit(s)	Bit Set Name	Description
31–0	ESI	Processor 32-bit general register, also used as a 32-bit stack index register.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations. String operations can use ESI as the source index register. The value in ESI represents the offset into a memory space defined by one of the segment registers. The default segment register is DS, but a segment override prefix allows a string instruction to use CS, SS, ES, FS, or GS. When used by string instructions, ESI automatically increments or decrements (based on the value of DF in the EFLAGS register — see page 1-39). This feature allows sequential string operations to operate on a set of string values without having to specify a new ESI value for each instruction.

1.41 ESP Processor General Register — Stack Pointer 32 bits

Bit(s)	Bit Set Name	Description
31–0	ESP	Processor general 32-bit register; also used as the Stack Pointer register.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 32-bit general processor registers used to hold 32-bit operands for logical and arithmetic operations. When used as the Stack Pointer, the register holds the offset value that points to the current top-of-stack (TOS) location within the memory segment specified by the Stack Segment (SS) register (see page 1-60). When a program PUSHes a value onto the stack, the processor decrements the value in the ESP register, and then writes the value to the new TOS specified by ESP. To POP a value, the processor copies it from the current address specified by ESP and then increments the ESP value.

1.42 FLAGS Flags Register 16 bits

Bit(s)	Bit Set Name	Description
15	N/A	Reserved, always 0
14	NT	0 = Current task is not nested below another task. 1 = Current task is nested below another task.
13–12	IOPL	00 = Highest I/O access privilege level; typically operating system. 01 = Second highest I/O access privilege level; system services. 10 = Third highest I/O access privilege level; system services. 11 = Lowest I/O access privilege level; application software.
11	OF	0 = Arithmetic result within limits. 1 = Arithmetic result not in positive/negative range, Overflow Flag set.
10	DF	0 = Forward direction, addressing increments. 1 = Backward direction, addressing decrements, Direction Flag set.
9	IF	0 = Maskable interrupts disabled. 1 = Maskable interrupts enabled, Interrupt Flag set.
8	TF	0 = Normal operation. 1 = Trap Flag set, processor enters single-step mode for debugging; each instruction generates a debug exception.
7	SF	0 = Arithmetic result is not negative (≥0); sign is +. 1 = Arithmetic result is negative (<0); Sign Flag set, sign is –.
6	ZF	0 = Arithmetic result is not zero. 1 = Arithmetic result is zero, Zero Flag set.
5	N/A	Reserved, always 0
4	AF	0 = No BCD carry. 1 = BCD carry from bit position 3, Auxiliary Flag set.
3	N/A	Reserved, always 0
2	PF	0 = Result Low byte has odd parity. 1 = Result Low byte has even parity, Parity Flag set.
1	N/A	Reserved, always 1
0	CF	0 = No carry from MSB of result. 1 = Carry from MSB of result, Carry Flag set.

Addressing

Specify by bit/set names or by using the special flag instructions (BT, BTR, BTS, CLC, CLD, LAHF, POPF, POPFD, PUSHF, PUSHFD, SAHF, STC, STD, STI) described in Chapter 2.

Default Value

00000002h

Functional Description

The 16-bit FLAGS register has system flags, status flags, and a control flag, described above. FLAGS is the lower word of EFLAGS (see page 1-39).

1.43 FPUCR FPU Control Register 16 bits

Bit(s)	Bit Set Name	Description
15–13	N/A	Reserved, undefined.
12	Infinity Control	Not used.
11–10	Rounding Control (RC)	00 = Round to nearest or even value. 01 = Round down (toward – ∞). 10 = Round up (toward + ∞). 11 = Chop (truncate toward 0).
9–8	Precision Control (PC)	00 = 24 bits (single precision). 01 = Not used/reserved. 10 = 53 bits (double precision). 11 = 64 bits (extended precision).
7–6	N/A	Reserved, undefined.
5	Precision Exception Mask	0 = Exception not masked. 1 = Exception masked.
4	Underflow Exception Mask	0 = Exception not masked. 1 = Exception masked.
3	Overflow Exception Mask	0 = Exception not masked. 1 = Exception masked.
2	Zero Divide Exception Mask	0 = Exception not masked. 1 = Exception masked.
1	Denormalized Operand Exception Mask	0 = Exception not masked. 1 = Exception masked.
0	Invalid Operation Exception Mask	0 = Exception not masked. 1 = Exception masked.

Addressing

Use the appropriate Instruction (FLDCW, FNSTCW, or FSTCW) to address the contents of this register. See Chapter 2 for a description of these instructions.

Default Value

Undefined

Functional Description

The FPUCR stores the current FPU Control Word value. The Control Word allows configuration of Rounding and Precision Control values and masking of the six exception types described above. No direct writing to or reading from this register is possible. Load a value from memory using the FLDCW instruction to write to the register. Load a copy to memory using the FNSTCW or FSTCW instruction to read the register contents.

1.44 FPUDP FPU Data Pointer 32 or 64 bits

Bit(s)	Bit Set Name	Description
32-bit Format in Protected Mode (64-bit field):		
64–49	N/A	Reserved
48–32	Operand Selector	Stores the value loaded into the segment register to select the data segment.
31–0	Data Operand Offset	Stores the data offset value within the specified segment.
32-bit Format in Real or Virtual-8086 Mode (64-bit field):		
64–61	N/A	Reserved, always 0000
60–45	Operand Pointer (bits 31–16)	Upper word of the operand address.
44–32	N/A	Reserved, always 0000 0000 0000
31–16	N/A	Reserved, undefined.
15–0	Operand Pointer (bits 15–0)	Lower word of the operand address.
16-bit Format in Protected Mode (32-bit field):		
31–16	Operand Selector	Stores the value loaded into the segment register to select the data segment.
15–0	Data Operand Offset	Stores the data offset value within the specified segment.
16-bit Format in Real or Virtual-8086 Mode (32-bit field):		
31–28	Operand Pointer (bits 19–16)	Upper four bits of the operand address.
27–16	N/A	Reserved, always 0000 0000 0000
15–0	Operand Pointer (bits 15–0)	Lower 16 bits of the operand address.

Addressing

Direct addressing of the register contents is not possible. Use the instructions FLDENV, FNSAVE, FNSTENV, FRSTOR, FSAVE, and FSTENV to write to or read from the register. The save (FNSAVE, FSAVE) and store (FNSTENV and FSTENV) instructions write the contents of all the FPU registers to memory. The FPU Data Pointer starts at offset 14h from the base address in 32-bit format, or offset Ah from the base address in 16-bit format, within the stored ENVironment data.

Default Value

Undefined

Functional Description

The data pointer stores the address of the last data operand that caused a floating-point exception. The format of the pointer varies depending on the addressing format (32-bit or 16-bit) and mode (Protected or Real/Virtual), as described above.

1.45 FPUIP FPU Instruction Pointer 32 or 64 bits

Bit(s)	Bit Set Name	Description
32-bit Format in Protected Mode (64-bit field):		
64–49	N/A	Reserved
48–32	CS Selector	Stores the value loaded into the code segment register .
31–0	IP Offset	Stores the instruction pointer offset value.
32-bit Format in Real or Virtual-8086 Mode (64-bit field):		
64–61	N/A	Reserved, always 0000
60–45	Instruction Pointer (bits 31–16)	Upper word of the instruction pointer address.
44–32	N/A	Reserved, always 0
43–32	Opcode	Stores the 11-bit opcode value.
31–16	N/A	Reserved, undefined.
15–0	Instruction Pointer (bits 15–0)	Lower word of the instruction pointer address.
16-bit Format in Protected Mode (32-bit field):		
31–16	CS Selector	Stores the value loaded into the code segment register.
15–0	IP Offset	Stores the instruction pointer offset value.
16-bit Format in Real or Virtual-8086 Mode (32-bit field):		
31–28	Instruction Pointer (bits 19–16)	Upper four bits of the instruction pointer address.
27	N/A	Reserved, always 0
26–16	Opcode	Stores the 11-bit opcode value.
15–0	Instruction Pointer (bits 15–0)	Lower 16 bits of the instruction pointer address.

Addressing

Direct addressing of the register contents is not possible. Use the instructions FLDENV, FNSAVE, FNSTENV, FRSTOR, FSAVE, and FSTENV to write to or read from the register. The save (FNSAVE, FSAVE) and store (FNSTENV and FSTENV) instructions write the contents of all the FPU registers to memory. The FPU Instruction Pointer starts at offset Ch from the base address in 32-bit format, or offset 6h from the base address in 16-bit format, within the stored ENVironment data.

Default Value

Undefined

Functional Description

The data pointer stores the address of the last instruction that caused a floating-point exception. In Real or Virtual-8086 mode, the opcode field stores the opcode value for the last non-control FPU instruction. The format of the pointer varies depending on the addressing format (32-bit or 16-bit) and mode (Protected or Real/Virtual), as described above.

1.46 FPUSR FPU Status Register 16 bits

Bit(s)	Bit Set Name	Description
15	B	0 = FPU not busy 1 = FPU busy
14	C3	Condition flag C3, value varies depending on floating-point instruction. For compare and test instructions, 0 = result not zero and 1 = result zero. FXAM uses C3, C2, and C0 to generate a result code (see FXAM). For FPREM and FPREM1, C3 is the least significant bit of the result.
13–11	TOP	000 = R0 is top of stack. 001 = R1 is top of stack. 010 = R2 is top of stack. 011 = R3 is top of stack. 100 = R4 is top of stack. 101 = R5 is top of stack. 110 = R6 is top of stack. 111 = R7 is top of stack.
10	C2	Condition flag C2, value varies depending on floating-point instruction. For compare and test instructions: 0 = operand is comparable and 1 = operand is not comparable. FXAM uses C3, C2, and C0 to generate a result code (see FXAM). For FPREM and FPREM1: 0 = reduction complete and 1 = reduction incomplete.
9	C1	Condition flag C1, value varies depending on floating point instruction. If the instruction generates an exception: 0 = underflow error and 1 = overflow error. If there is no exception: For FXAM, 0 = value is ≥0, sign is +; 1 = value is < 0, sign is –. For FPREM and FPREM1, C1 is the second least significant result bit. For arithmetic instructions: 0 = last rounding down and 1 = last rounding up.
8	C0	Condition flag C0, value varies depending on floating point instruction. For compare and test instructions: 0 = result did not generate carry and 1 = result generated carry. FXAM uses C3, C2, and C0 to generate a result code (see FXAM). For FPREM and FPREM1, C0 is the third least significant result bit.
7	ES	0 = No exception generated. 1 = Exception generated.
6	SF	0 = No exception generated. 1 = Stack fault exception generated.
5	PE	0 = No exception generated. 1 = Precision exception generated.
4	UE	0 = No exception generated. 1 = Underflow exception generated.
3	OE	0 = No exception generated. 1 = Overflow exception generated.
2	ZE	0 = No exception generated. 1 = Divide by zero exception generated.
1	DE	0 = No exception generated. 1 = Denormalized operand exception generated.
0	IE	0 = No exception generated. 1 = Invalid operation exception generated.

Addressing

Use the appropriate instruction (FLDSW, FNSTSW, or FSTSW) to address the contents of this register. See Chapter 2 for a description of these instructions.

Default Value

Undefined

Functional Description

The FPUSR stores the current FPU Status Word value. The Status Word allows monitoring of the current status of the FPU. Direct addressing of the register contents is not possible. Load a value from memory using the FLDSW instruction to write to the register. Read the current value by loading a copy to memory using the FNSTSW or FSTSW instruction. The interaction between the FPU instructions and the Status Word is discussed in detail in Chapter 2 as part of the individual instruction descriptions.

1.47 FPUTWR FPU Tag Word Register 16 bits

Bit(s)	Bit Set Name	Description
15–14	TAG(7)	00 = R7 contents valid. 01 = R7 contents are zero. 10 = R7 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R7 empty.
13–12	TAG(6)	00 = R6 contents valid. 01 = R6 contents are zero 10 = R6 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R6 empty.
11–10	TAG(5)	00 = R5 contents valid. 01 = R5 contents are zero. 10 = R5 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R5 empty.
9–8	TAG(4)	00 = R4 contents valid. 01 = R4 contents are zero. 10 = R4 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R4 empty.
7–6	TAG(3)	00 = R3 contents valid. 01 = R3 contents are zero. 10 = R3 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R3 empty.
5–4	TAG(2)	00 = R2 contents valid. 01 = R2 contents are zero. 10 = R2 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R2 empty.
3–2	TAG(1)	00 = R1 contents valid. 01 = R1 contents are zero. 10 = R1 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R1 empty.
1–0	TAG(0)	00 = R0 contents valid. 01 = R0 contents are zero. 10 = R0 contents special: invalid (NaN or unsupported), infinity, or denormal. 11 = R0 empty.

Addressing

Direct addressing of the register contents is not possible. Use the instructions FLDENV, FNSAVE, FNSTENV, FRSTOR, FSAVE, and FSTENV to write to or read from the register. The save (FNSAVE, FSAVE) and store (FNSTENV and FSTENV) instructions write the contents of all the FPU registers to memory. The FPU Tag Word starts at offset 8h from the base address in 32-bit format, or offset 4h from the base address in 16-bit format, within the stored ENVironment data.

Default Value

Undefined

Functional Description

The FPUTWR stores the Tag Word for the eight FPU data registers (R0–R7). The Tag Word describes the current status for each of these register, as described above. Because the FPU instructions refer to the registers indirectly through the stack register notation as ST(0) through ST(7) and the actual associated registers change as the stack pointer changes, use the value for TOP (bits 13–11 in the FPU Status Word — see page 1-48) to associate the tag values with the relative stack registers.

1.48 FS Data Segment Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	FS	A segment register that holds the base address for one of the four data segments of memory, available to the program currently executing.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The processor organizes memory into segments as one of the possible ways to access memory. There are six segments (tables within memory) accessed through the segment registers. Each register stores the base address for its segment. There are four data segments that can contain data used by a program being executed. The segment selectors (base addresses) for these segments are stored in the DS, ES, FS, and GS registers. The processor fetches data from a data segment, using an offset into the segment. The data segment register value changes as a result of interrupts, exceptions, and instructions that transfer control between segments (see CALL, IRET, and JMP instructions).

1.49 GDTR Global Descriptor Table Register 48 bits

Bit(s)	Bit Set Name	Description
47–16	GDT Base Address	Stores the base address for the Global Descriptor Table location.
15–0	GDT Segment Limit	Stores the limit for the Global Descriptor Table segment.

Addressing

Direct addressing of the register contents is not possible. Write to the register using the LGDT instruction. Read the contents of the register into memory using the SGDT instruction. Both instructions require the highest privilege level generally accorded only to operating system software.

Default Value

Undefined; BIOS and operating system software define the contents of this register.

Functional Description

The register holds the 32-bit base address and 16-bit segment limit for the Global Descriptor Table (GDT). The referenced GDT contains the segment descriptors for the memory available to any general operation. The table can vary in size from a minimum of 8 bytes to a maximum of 64K bytes. Each memory segment descriptor requires 8 bytes, so the GDT can store as many as 8192 segment descriptors. The first 8 bytes of the GDT are, however, reserved as the null descriptor to define a null pointer value. Load the null value into unused segment registers to initialize them.

The GDT contains selectors for all of the defined Local Descriptor Tables (LDTs) but should exclude segments defined for use by the system services (interrupts and traps). The system services segments are included as part of the Interrupt Descriptor Table (IDT). A detailed description of the descriptor tables is included as part of Appendix A.

1.50 GS Data Segment Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	CS	A segment register that holds the base address for one of the four data segments of memory, available to the program currently executing.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The processor organizes memory into segments as one of the possible ways to access memory. There are six segments (tables within memory) accessed through the segment registers. Each register stores the base address for its segment. There are four data segments that can contain data used by a program being executed. The segment selectors (base addresses) for these segments are stored in the DS, ES, FS, and GS registers. The processor fetches data from a data segment, using an offset into the segment. The data segment register value changes as a result of interrupts, exceptions, and instructions that transfer control between segments (see CALL, IRET, and JMP instructions).

1.51 IDTR Interrupt Descriptor Table Register 48 bits

Bit(s)	Bit Set Name	Description
47–16	IDT Base Address	Stores the base address for the Interrupt Descriptor Table location.
15–0	IDT Segment Limit	Stores the limit for the Interrupt Descriptor Table segment.

Addressing

Direct addressing of the register contents is not possible. Write to the register using the LIDT instruction. Read the contents of the register into memory using the SIDT instruction. Both instructions require the highest privilege level generally accorded only to operating system software.

Default Value

Undefined; BIOS and operating system software define the contents of this register.

Functional Description

The register holds the 32-bit base address and 16-bit segment limit for the Interrupt Descriptor Table (IDT). The referenced IDT contains the segment descriptors for the memory available to system service (interrupt and trap) operations. The table can vary in size from a minimum of 8 bytes to a maximum of 64K bytes. Each memory segment descriptor requires 8 bytes, so the IDT can store as many as 8192 segment descriptors. To protect them from use by other tasks, exclude the system services segments from the General Descriptor Table (GDT). A detailed description of the descriptor table is included as part of Appendix A.

1.52 IP Instruction Pointer 16 bits

Bit(s)	Bit Set Name	Description
15–0	IP	Instruction Pointer, contains the 16-bit offset into the current code segment for the next instruction.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The IP register contains the 16-bit offset that points to the next instruction within the current code segment, when operating using 16-bit addressing. The control-transfer instructions, such as JUMP or RET, and interrupts and exceptions control the contents of this register implicitly. The contents of this register advance from one instruction boundary to the next. Because of instruction prefetching, its value is only an approximate indication of the bus activity loading instructions into the processor. The IP register is the lower word of the EIP register (see page 1-40).

1.53 LDTR Local Descriptor Table Register 48 bits

Bit(s)	Bit Set Name	Description
47–16	LDT Base Address	Stores the base address for the Local Descriptor Table location.
15–0	LDT Segment Limit	Stores the limit for the Local Descriptor Table segment.

Addressing

Direct addressing of the register contents is not possible. Write to the register using the LLDT instruction. Read the contents of the register into memory using the SLDT instruction. Both instructions require the highest privilege level generally accorded only to operating system software.

Default Value

Undefined; BIOS and operating system software define the contents of this register.

Functional Description

The register holds the 32-bit base address and 16-bit segment limit for the current Local Descriptor Table (LDT) used by a referenced segment register (CS, DS, ES, FS, GS, or SS). By using the segment registers, a task can access as many as six different memory segments simultaneously. The referenced LDT contains the segment descriptors for the memory available to a specific task. The table can vary in size from a minimum of 8 bytes to a maximum of 64 Kbytes. Each memory segment descriptor requires 8 bytes, so each LDT can store as many as 8192 segment descriptors.

The LDT should exclude segments defined for use by the system services (interrupts and traps). The system services segments are included as part of the Interrupt Descriptor Table (IDT). A detailed description of the descriptor table is included as part of Appendix A.

1.54 R0–R7 FPU Data Registers 0–7 80 bits each

Bit(s)	Bit Set Name	Description
79	Sign	0 = + 1 = –
78–64	Exponent	Exponent value
63–0	Significand	Significand value

Addressing

Address the registers through the stack address ST(n). ST(0) is the top of the FPU stack. Bits 13–11 in the FPU Status Word indicate which data register is at the top of the stack (see page 1-48).

Default Value

00000000000000000000h

Functional Description

The FPU data registers store data for processing by the FPU. Numeric instructions address the data registers relative to the register at the top of the FPU stack. At any point in time, the register at the top of the stack (R0–R7) is indicated by the TOP field in the FPU status word. Load or push operations decrement TOP by one and load a value into the new TOP register. A store-and-pop operation stores the value from the current TOP register and then increments TOP by 1. The FPU register stack, similar to stack operations in memory, grows down toward lower-numbered registers. Some numeric operations allow operating on registers as an offset of the stack top.

1.55 SI Processor General Register — Stack Index 16 bits

Bit(s)	Bit Set Name	Description
15–0	SI	Processor general 16-bit register; also used as the Stack Index register.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations. String operations can use SI as the source index register for 16-bit addressing. The value in SI represents the offset into a memory space defined by one of the segment registers. The default segment register is DS, but a segment override prefix allows a string instruction to use CS, SS, ES, FS, or GS. When used by string instructions, SI automatically increments or decrements (based on the value of DF in the EFLAGS register — see page 1-39). This feature allows sequential string operations to operate on a set of string values without having to specify a new SI value for each instruction.

1.56 SP Processor General Register — Stack Pointer 16 bits

Bit(s)	Bit Set Name	Description
15–0	SP	Processor general 16-bit register, also used as Stack Pointer register for 16-bit addressing modes.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

One of the eight, 16-bit general processor registers used to hold 16-bit operands for logical and arithmetic operations. When used as the Stack Pointer in 16-bit addressing mode, the register holds the offset value that points to the current top-of-stack (TOS) location within the memory segment specified by the Stack Segment (SS) register (see page 1-60). When a program PUSHes a value onto the stack, the processor decrements the value in the SP register, and then writes the value to the new TOS specified by SP. To POP a value, the processor copies it from the current address specified by SP and then increments the SP value.

1.57 SS Stack Segment Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	SS	Stack segment register holds the base address for the stack segment in memory.

Addressing

Specify by name as instruction operand.

Default Value

Undefined

Functional Description

The processor organizes memory into segments as one of the possible ways to access memory. There are six segments (tables within memory) accessed through the segment registers. Each register stores the base address for its segment. The segment containing the temporary user space for the program being executed is called the stack segment. Its segment selector (base address) is stored in the SS register. The processor writes to and fetches dynamically stored information from the stack segment, using the contents of the SP register as a top-of-stack pointer and the SI register as offset into the segment. The SS register value changes as a result of interrupts, exceptions, and instructions that transfer control between tasks (see CALL, IRET, and JMP instructions).

1.58 TR Task Register 16 bits

Bit(s)	Bit Set Name	Description
15–0	Selector	The selector value used to access/index a TSS descriptor in the GDT.

Addressing

Access by using the load instruction LTR or the store instruction STR.

Default Value

Design dependent; loaded during system initialization and then modified by task switching.

Functional Description

The Task Register points to the current TSS. This register consists of a visible 16-bit selector that points to the TSS descriptor in the GDT for the current task, and the invisible base address and segment limit maintained by the TSS. The processor maintains the invisible part of the TR to make execution more efficient by addressing the Task State Segment directly through the register. The LTR instruction requires the highest privilege level (CPL = 0) because changing the register must be restricted to initialization and operating software task switches to prevent unpredictable results. The STR instruction has no privilege restriction.

1.59 TR3 Cache Test Data Register 32 bits

Bit(s)	Bit Set Name	Description
31–0	Data	Data storage for internal cache testing.

Addressing

Specify by name as instruction operand.

Default Value

00000000h

Functional Description

TR3 is the cache test data register. This register contains a doubleword used to write to the cache, or, a doubleword read from the cache read buffer. The fill and read buffers each store four doublewords that pass through TR3 one at a time. Select a specific doubleword in either buffer by using the 2-bit Entry Select field (bits 2 and 3) of TR5 (see page 1-64).

1.60 TR4 Cache Test Status Register 32 bits

Bit(s)	Bit Set Name	Description
31–11	TAG	The address that becomes the tag on a cache write.
10	VALID	0 = Not valid. 1 = Valid bit on a cache lookup, this is a copy of one of the bits 6–3; on a write it is a new bit.
9–7	LRU	On a cache lookup, this is the three LRU bits of the accessed set; the LRU bits in the cache are updated by the pseudo-LRU cache replacement algorithm. On a cache write, these bits are ignored.
6–3	VALID	On a cache lookup, these are the four Valid bits of the accessed set.
2–0	N/A	Reserved; always 000.

Addressing (I/O)

Specify by name as instruction operand.

Default Value

00000000h

Functional Description

TR4 contains the Cache Test Status Register. This includes the Valid bits, LRU bits, and a tag.

1.61 TR5 Cache Test Control Register 32 bits

Bit(s)	Bit Set Name	Description
31–11	N/A	Not used.
10–4	SET SELECT	Selects one of the 128 available sets.
3–2	ENTRY SELECT	During a cache read or write, selects one of four entries in the set addressed by the Set Select; during cache-fill-buffer writes or read-buffer reads, selects one of the four doublewords in a line.
1–0	CONTROL	00 = Write to cache fill buffer, or read from cache read buffer. 01 = Perform cache write. 10 = Perform cache read. 11 = Flush the cache (mark all entries as invalid).

Addressing

Specify by name as instruction operand.

Default Value

00000000h

Functional Description

TR5 is the Cache Test Control Register. The register defines the section (set and entry) of the cache to test and the operation to perform.

1.62 TR6 TLB Test Control Register 32 bits

Bit(s)	Bit Set Name	Description
31–12	Linear Address	On a write, the TLB entry is allocated to this linear address. On a TLB lookup, the TLB is interrogated with this value.
11	V	0 = TLB not valid. 1 = TLB data valid.
10–9	D, $\overline{D}$	00 = Undefined 01 = Match on lookup; clear D on write. 10 = Match on lookup; set D on write. 11 = Undefined
8–7	U, $\overline{U}$	00 = Undefined 01 = Match on lookup; clear U on write. 10 = Match on lookup; set U on write. 11 = Undefined
6–5	W, $\overline{W}$	00 = Undefined 01 = Match on lookup; clear W on write. 10 = Match on lookup; set W on write. 11 = Undefined
4–1	N/A	Reserved, always 0000
0	C	0 = TLB write enabled. 1 = TLB lookup enabled.

Addressing

Specify by name as instruction operand.

Default Value

00000000h

Functional Description

The Am486 processor uses a translation lookaside buffer (TLB) to translate linear address to physical address in the cache. The TLB contains the 20 high-order bits of a physical address used as a base address for a memory page. The 12 low-order bits (the offset into the page) are the same in both a linear and physical address. Corresponding to the block of data entries is a block of valid, attribute, and tag entries. The entry consists of the 17 high-order bits of the linear address (31–15). The processor uses the middle-order bits (14–12) to address eight sets and then checks the four tags of a selected set for a match with the high-order bits. If a match is found among the tags of the selected set, the corresponding valid bit is set to 1 and the linear address is translated by replacing its high-order 20 bits with the 20 bits of the corresponding data entry. Three LRU bits are included in each set to track the use of data in each set. The LRU bits are checked when a new entry is needed and none of the entries in the set is invalid; a pseudo-LRU replacement algorithm modifies the LRU when required.

Testing of the TLB uses two registers, TR6 and TR7. TR6 is the Test Control Register. TR7 contains test data read from or written to the TLB.

1.63 TR7 TLB Test Status Register 32 bits

Bit(s)	Bit Set Name	Description
31–12	Physical Address	This is the data field of the TLB. On a write to the TLB, the Linear Address in TR6 is set to this value. On a TLB Lookup, the physical address is loaded from the TLB to this field.
11	PCD	The page-level cache-disable (PCD) bit of a page table entry.
10	PWT	The page-level write-through (PWT) bit of a page table entry.
9–7	LRU	The LRU values before a TLB lookup. TLB lookups that result in hits and TLB writes change the value of these bits.
6–5	N/A	Reserved, always 00
4	PL	0 = On a write, the internal pointer of the paging unit selects the TLB block to load. On a TLB lookup, this value indicates a miss. 1 = On a write, the REP field selects which associative block of the TLB to load. On a TLB lookup, this value indicates a hit.
3–2	REP	If TLB = 0, REP is undefined. If TLB = 1, then, For a TLB write, REP indicates which block to write. For a TLB lookup, REP reports in which of the associative blocks, the tag was found.
1–0	N/A	Reserved, always 00

Addressing

Specify by name as instruction operand.

Default Value

00000000h

Functional Description

The Am486 processor uses a translation lookaside buffer (TLB) to translate linear address to physical address in the cache. The TLB contains the 20 high-order bits of a physical address used as a base address for a memory page. The 12 low-order bits (the offset into the page) are the same in both a linear and physical address. Corresponding to the block of data entries is a block of valid, attribute, and tag entries. The entry consists of the 17 high-order bits of the linear address (31–15). The processor uses the middle-order bits (14–12) to address eight sets and then checks the four tags of a selected set for a match with the high-order bits. If a match is found among the tags of the selected set, the corresponding valid bit is set to 1 and the linear address is translated by replacing its high-order 20 bits with the 20 bits of the corresponding data entry. Three LRU bits are included in each set to track the use of data in each set. The LRU bits are checked when a new entry is needed and none of the entries in the set is invalid; a pseudo-LRU replacement algorithm modifies the LRU when required.

Testing of the TLB uses two registers, TR6 and TR7. TR6 is the Test Control Register. TR7 contains test data read from or written to the TLB.

CHAPTER

2 Am486 MICROPROCESSOR INSTRUCTION SET

2.1 OVERVIEW

The Am486 microprocessor instruction set uses the same basic instructions as other 486-based microprocessors. Pages 2-2 and 2-3 provide a roadmap to these instructions using functional categories. For each instruction, the roadmap lists the page on which the detailed instruction description appears. In the detailed description section that follows the instruction roadmap, the instructions appear in alphabetical order using the roadmap name.

2.2 DETAILED INSTRUCTION DESCRIPTIONS

Note: *If you are unfamiliar with the instruction notation used in this chapter, refer to Appendix A for a detailed explanation of instructions and their use in application programming.*

Instruction descriptions begin on page 2-4, using the following format:

INSTRUCTION NAME/S   General Description

Opcode	Instruction	Clocks	Concurrent Execution*	Description
nn xx	XXX	nn	nn Some FPU instructions	Function

Operation

Algorithmic description using a notation similar to Algol or Pascal language.

Description

Verbal description of code operation.

[FPU] Flags Affected

Description of changes made to system flags (or FPU flags C0, C1, C2, and C3).

Numeric Exceptions (floating-point operations only)

List of possible FPU exceptions.

Protected Mode Exceptions

Description of exceptions generated in Protected Mode.

Real Address Mode Exceptions

Description of exceptions generated in Real Address Mode.

Virtual 8086 Mode Exceptions

Description of exceptions generated in Virtual 8086 Mode.

**shaded column not included for all instructions.*

Instruction Roadmap

Binary Arithmetic

AAA	2-4
AAD	2-5
AAM	2-6
AAS	2-7
ADC	2-8
ADD	2-9
CMP	2-32
DAA	2-38
DAS	2-39
DEC	2-40
DIV	2-41
IDIV	2-124
IMUL	2-125
INC	2-127
MUL	2-202
NEG	2-203
SBB	2-233
SUB	2-253

Block Structured Language

ENTER	2-42
LEAVE	2-180

Data Movement

CBW	2-25
CDQ	2-26
CWD	2-36
CWDE	2-37
MOV	2-195
POP	2-210
POPA	2-212
POPAD	2-213
PUSH	2-215
PUSHA	2-217
PUSHAD	2-218
XCHG	2-259

Data Pointer

LDS	2-178
LES	2-181
LFS	2-182
LGS	2-184
LSS	2-193

Control Transfer

CALL	2-20
IRET	2-136
IRETD	2-136
JA	2-140
JAE	2-141
JB	2-142
JBE	2-143
JC	2-144
JCXZ	2-145
JE	2-146
JECXZ	2-147
JG	2-148
JGE	2-149
JL	2-150
JLE	2-151
JMP	2-152
JNA	2-156
JNAE	2-157
JNB	2-158
JNBE	2-159
JNC	2-160
JNE	2-161
JNG	2-162
JNGE	2-163
JNL	2-164
JNLE	2-165
JNO	2-166
JNP	2-167
JNS	2-168
JNZ	2-169
JO	2-170
JP	2-171
JPE	2-172
JPO	2-173
JS	2-174
JZ	2-175
LOOP	2-191
LOOPE	2-191
LOOPNE	2-191
LOOPNZ	2-191
LOOPZ	2-191
RET	2-224

Interrupt Control

BOUND	2-12
INT	2-130
INTO	2-130

Flag Control

CLC	2-27
CLD	2-28
CLI	2-29
CLTS	2-30
CMC	2-31
LAHF	2-176
POPF	2-214
POPFD	2-214
PUSHF	2-219
PUSHFD	2-219
SAHF	2-230
STC	2-247
STD	2-248
STI	2-249

Logical Operation

AND	2-10
BSF	2-13
BSR	2-14
BT	2-16
BTC	2-17
BTR	2-18
BTS	2-19
NOT	2-205
OR	2-206
XOR	2-261

Input/Output (I/O)

IN	2-126
OUT	2-207

Protection Control

ARPL	2-11
LAR	2-177
LGDT	2-183
LIDT	2-185
LLDT	2-186
LMSW	2-187
LOCK	2-188
LSL	2-192
LTR	2-194
SGDT	2-237
SIDT	2-244
SLDT	2-245
SMSW	2-246
STR	2-252
VERR	2-255
VERW	2-255

Process Control

HLT	2-123
INVD	2-134
INVLPG	2-135
WAIT	2-256
WBINVD	2-257

Shift and Rotate

RCL	2-220
RCR	2-221
ROL	2-228
ROR	2-229
SAL	2-231
SAR	2-232
SHL	2-238
SHLD	2-239
SHR	2-241
SHRD	2-242

Miscellaneous

BSWAP	2-15
CMPXCHG	2-35
LEA	2-179
NOP	2-204
TEST	2-254
XADD	2-258
XLAT	2-260
XLATB	2-260

Set Register

SETA	2-236
SETAE	2-236
SETB	2-236
SETBE	2-236
SETC	2-236
SETE	2-236
SETG	2-236
SETGE	2-236
SETL	2-236
SETLE	2-236
SETNA	2-236
SETNAE	2-236
SETNB	2-236
SETNBE	2-236
SETNC	2-236
SETNE	2-236
SETNG	2-236
SETNGE	2-236
SETNL	2-236
SETNLE	2-236
SETNO	2-236
SETNP	2-236
SETNS	2-236
SETNZ	2-236
SETO	2-236
SETP	2-236
SETPE	2-236
SETPO	2-236
SETS	2-236
SETZ	2-236

String Operations

CMPS	2-33
CMPSB	2-33
CMPSD	2-33
CMPSW	2-33
INS	2-128
INSB	2-128
INSD	2-128
INSW	2-128
LODS	2-189
LODSB	2-189
LODSD	2-189
LODSW	2-189
MOVS	2-198
MOVSB	2-198
MOVSD	2-198
MOVSW	2-198
MOVSX	2-200
MOVZX	2-201
OUTS	2-208
OUTSB	2-208
OUTSD	2-208
OUTSW	2-208
REP	2-222
REPE	2-222
REPNE	2-222
REPNZ	2-222
REPZ	2-222
SCAS	2-234
SCASB	2-234
SCASD	2-234
SCASW	2-234
STOS	2-250
STOSB	2-250
STOSD	2-250
STOSW	2-250

Floating-Point Operations

F2XM1	2-43
FABS	2-44
FADD	2-45
FADDP	2-46
FBLD	2-47
FBSTP	2-48
FCHS	2-49
FCLEX	2-50
FCOM	2-51
FCOMP	2-52
FCOMPP	2-53
FCOS	2-54
FDECSTP	2-55
FDIV	2-56
FDIVP	2-57
FDIVR	2-58
FDIVRP	2-59
FFREE	2-60
FIADD	2-61
FICOM	2-62
FICOMP	2-63
FIDIV	2-64
FIDIVR	2-65
FILD	2-66
FIMUL	2-67
FINCSTP	2-68
FINIT	2-69
FIST	2-70
FISTP	2-71
FISUB	2-72
FISUBR	2-73
FLD1	2-75
FLD	2-74
FLDCW	2-76
FLDENV	2-77
FLDL2E	2-78
FLDL2T	2-79
FLDLG2	2-80
FLDLN2	2-81
FLDPI	2-82
FLDZ	2-83
FMUL	2-84
FMULP	2-85
FNCLEX	2-86
FNINIT	2-87
FNOP	2-88
FNSAVE	2-89
FNSTCW	2-90
FNSTENV	2-91
FNSTSW	2-92
FPATAN	2-93
FPREM	2-94
FPREM1	2-95
FPTAN	2-96
FRNDINT	2-97
FRSTOR	2-98
FSAVE	2-99
FSCALE	2-100
FSIN	2-101
FSINCOS	2-102
FSQRT	2-103
FST	2-104
FSTCW	2-105
FSTENV	2-106
FSTP	2-107
FSTSW	2-108
FSUB	2-109
FSUBP	2-110
FSUBR	2-111
FSUBRP	2-112
FTST	2-113
FUCOM	2-114
FUCOMP	2-115
FUCOMPP	2-116
FWAIT	2-117
FXAM	2-118
FXCH	2-119
FXTRACT	2-120
FYL2X	2-121
FYL2XP1	2-122

2.3 AAA ASCII Adjusts AL after Addition

Opcode	Instruction	Clocks	Description
37	AAA	3	ASCII adjusts after addition.

Operation

```
IF ((AL and 0Fh) > 9) OR (AF = 1)
THEN
   AL ← (AL + 6) and 0Fh;
   AH ← AH + 1;
   AF ← 1;
   CF ← 1;
ELSE
   CF ← 0;
   AF ← 0;
FI
```

Description

Use the AAA instruction after an ADD instruction that leaves a byte result in the AL register. The lower nibbles of the operands of the ADD instruction should be in the range 0–9 (BCD digits). The AAA instruction adjusts the AL register to contain the correct decimal digit result. If the addition produced a decimal carry, AAA increments the AH register and sets the Carry and Auxiliary-carry Flags (CF and AF). If there is no decimal carry, AAA clears CF and AF and leaves the AH register unchanged. AAA sets the top nibble of the AL register to 0. To convert the AL register to an ASCII result, use an OR AL, 30h instruction after the AAA instruction.

Flags Affected

For a decimal carry, AAA sets AF and CF. AAA clears AF and CF when there is no carry. OF, SF, ZF, and PF are not affected by this instruction.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.4 AAD ASCII Adjusts AX before Division

Opcode	Instruction	Clocks	Description
D5 0A	AAD	14	ASCII adjusts AX before division.

Operation

```
AH ← AH · 10 + AL ; 10 is decimal
AH ← 0
```

Description

AAD prepares two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) for a division operation that yields an unpacked result. The instruction sets the AL register to AL + (10 • AH) and then clears the AH register. The AX register then equals the binary equivalent of the original unpacked two digit number.

Flags Affected

The result determines the SF, ZF, and PF settings. This instruction does not affect OF, AF, and CF.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.5 AAM ASCII Adjusts AX after Multiply

Opcode	Instruction	Clocks	Description
D4 0A	AAM	15	ASCII adjusts AX after multiply.

Operation

```
AH ← AL / 10
AL ← AL MOD 10
```

Description

Use AAM only after executing the MUL instruction between two unpacked BCD operands with the result in the AX register. Because the result is less than 100, it resides entirely in the AL register. AAM unpacks the AL result by dividing AL by 10, leaving the quotient (most-significant digit) in AH and the remainder (least-significant digit) in AL.

Flags Affected

The result determines the SF, ZF, and PF settings. This instruction does not affect OF, AF, and CF.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.6 AAS ASCII Adjusts AL after Subtraction

Opcode	Instruction	Clocks	Description
3F	AAS	3	ASCII adjusts AL after subtract.

Operation

```
IF ((AL and 0Fh) > 9) OR (AF = 1)
THEN
   AL ← AL – 6;
   AL ← AL and 0Fh;
   AH ← AH – 1;
   AF ← 1;
   CF ← 1;
ELSE
   CF ← 0;
   AF ← 0;
FI
```

Description

Use AAS only after a SUB instruction that leaves the byte result in AL. The lower nibbles of the SUB instruction must be in the range 0–9 (BCD). AAS adjusts AL so that it contains the correct decimal result. If the subtraction produced a decimal carry, AAS decrements AH and sets CF and AF. If there is no decimal carry, AAS clears CF and AF and leaves AH unchanged. AAS sets the top nibble set in AL to 0. Use OR AL, 30h after AAS to convert AL to an ASCII result.

Flags Affected

For a decimal carry, AAS sets AF and CF. AAS clears AF and CF when there is no carry. OF, SF, ZF, and PF are not affected by this instruction.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.7 ADC Adds Integers with Carry

Opcode	Instruction	Clocks	Description
14 ib	ADC AL, imm8	1	Adds immediate byte to AL with carry.
15 iw	ADC AX, imm16	1	Adds immediate word to AX with carry.
15 id	ADC EAX, imm32	1	Adds immediate doubleword to EAX with carry.
80 /2 ib	ADC r/m8, imm8	1/3	Adds immediate byte to r/m byte with carry.
81 /2 iw	ADC r/m16, imm16	1/3	Adds immediate word to r/m word with carry.
81 /2 id	ADC r/m32, imm32	1/3	Adds immediate doubleword to r/m doubleword with carry.
83 /2 ib	ADC r/m16, imm8	1/3	Adds sign-extended immediate byte to r/m word with carry
83 /2 ib	ADC r/m32, imm8	1/3	Adds sign-extended immediate byte into r/m doubleword with carry.
10 /r	ADC r/m8, r8	1/3	Adds byte register to r/m byte with carry.
11 /r	ADC r/m16, r16	1/3	Adds word register to r/m word with carry.
11 /r	ADC r/m32, r32	1/3	Adds doubleword register to r/m doubleword with carry.
12 /r	ADC r8, r/m8	1/2	Adds r/m byte to byte register with carry.
13 /r	ADC r16, r/m16	1/2	Adds r/m word to word register with carry.
13 /r	ADC r32, r/m32	1/2	Adds r/m doubleword to doubleword register with carry.

Operation

```
DEST ← DEST + SRC + CF
```

Description

ADC performs an integer addition of the two operands DEST and SRC and sets the Carry Flag (CF) as required. ADC assigns the result to DEST and sets the flags accordingly. ADC is typically part of a multibyte or multiword addition operation. ADC sign-extends immediate byte values to the appropriate size before adding to a word or doubleword operand.

Flags Affected

The result determines the OF, SF, ZF, CF, and PF settings.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.8 ADD Adds Integers

Opcode	Instruction	Clocks	Description
04 ib	ADD AL, imm8	1	Adds immediate byte to AL.
05 iw	ADD AX, imm16	1	Adds immediate word to AX.
05 id	ADD EAX, imm32	1	Adds immediate doubleword to EAX.
80 /0 ib	ADD r/m8, imm8	1/3	Adds immediate byte to r/m byte.
81 /0 iw	ADD r/m16, imm16	1/3	Adds immediate word to r/m word.
81 /0 id	ADD r/m32, imm32	1/3	Adds immediate doubleword to r/m doubleword.
83 /0 ib	ADD r/m16, imm8	1/3	Adds sign-extended immediate byte to r/m word.
83 /0 ib	ADD r/m32, imm8	1/3	Adds sign-extended immediate byte into r/m doubleword.
00 /r	ADD r/m8, r8	1/3	Adds byte register to r/m byte.
01 /r	ADD r/m16, r16	1/3	Adds word register to r/m word.
01 /r	ADD r/m32, r32	1/3	Adds doubleword register to r/m doubleword.
02 /r	ADD r8, r/m8	1/2	Adds r/m byte to byte register.
03 /r	ADD r16, r/m16	1/2	Adds r/m word to word register.
03 /r	ADD r32, r/m32	1/2	Adds r/m doubleword to doubleword register.

Operation

```
DEST ← DEST + SRC
```

Description

ADD performs an integer addition of the two operands DEST and SRC. ADD assigns the result to DEST and sets the flags accordingly. ADC sign-extends immediate byte values to the appropriate size before adding to a word or doubleword operand.

Flags Affected

The result determines the OF, SF, ZF, CF, and PF settings.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.9 AND Logical AND Function

Opcode	Instruction	Clocks	Description
24 ib	AND AL, imm8	1	ANDs immediate byte to AL.
25 iw	AND AX, imm16	1	ANDs immediate word to AX.
25 id	AND EAX, imm32	1	ANDs immediate doubleword to EAX.
80 /4 ib	AND r/m8, imm8	1/3	ANDs immediate byte to r/m byte.
81 /4 iw	AND r/m16, imm16	1/3	ANDs immediate word to r/m word.
81 /4 id	AND r/m32, imm32	1/3	ANDs immediate doubleword to r/m doubleword.
83 /4 ib	AND r/m16, imm8	1/3	ANDs sign-extended immediate bye to r/m word.
83 /4 ib	AND r/m32, imm8	1/3	ANDs sign-extended immediate byte into r/m doubleword.
20 /r	AND r/m8, r8	1/3	ANDs byte register to r/m byte.
21 /r	AND r/m16, r16	1/3	ANDs word register to r/m word.
21 /r	AND r/m32, r32	1/3	ANDs doubleword register to r/m doubleword.
22 /r	AND r8, r/m8	1/2	ANDs r/m byte to byte register.
23 /r	AND r16, r/m16	1/2	ANDs r/m word to word register.
23 /r	AND r32, r/m32	1/2	ANDs r/m doubleword to doubleword register.

Operation

```
DEST ← DEST AND SRC
CF ← 0
OF ← 0
```

Description

AND computes the logical AND of the two operands. If corresponding bits of the operands are 1, the resulting bit is 1. If the bits are not the same or are both 0, the result is 0. The answer replaces the first operand.

Flags Affected

AND clears CF and OF. The result determines the ZF, CF, and PF settings.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.10 ARPL Adjusts RPL Field of Selector

Opcode	Instruction	Clocks	Description
63 /r	ARPL r/m16, r16	9/9	Adjusts RPL of r/m16 to no less than the RPL of r16.

Operation

```
IF RPL bits(0,1) of DEST < RPL bits(0,1) of SRC
THEN
   ZF ← 1;
   RPL bits(0,1) of DEST ← RPL bits(0,1) of SRC;
ELSE
   ZF ← 0;
FI
```

Description

ARPL has two operands. The first (r/m16) is a 16-bit memory variable or word register that contains the selector value. The second (r16) is a word register. If the RPL field (“requested privilege level” — bits 0 and 1) of the first operand is less than the RPL field of the second operand, ARPL sets ZF and increases the RPL field of the first operand to equal the RPL field of the second operand. If the first operand RPL field is equal to or greater than the second operand RPL field, ARPL clears ZF and does not change the first operand.

Typically, ARPL appears in operating system software and not application programs. Its use guarantees that a selector parameter to a subroutine does not request a higher privilege level than allowed to the caller. The second operand used by ARPL is normally a register that contains the CS selector value of the caller.

Flags Affected

ARPL sets ZF to 1 if the first operand RPL field is less than the second operand RPL field. ARPL resets the ZF to 0 if the first operand RPL field is greater than or equal to the second operand RPL field.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Invalid Opcode (6); Real Address Mode does not recognize ARPL.

Virtual 8086 Mode Exceptions

Invalid Opcode (6); Virtual 8086 Mode does not recognize ARPL. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.11 BOUND Checks Array Index Against Bounds

Opcode	Instruction	Clocks	Description
62 /r	BOUND r16,m16&16	7	Checks to see if r16 is within bounds (passes test).
62 /r	BOUND r32,m32&32	7	Checks to see if r32 is within bounds (passes test).

Operation

```
IF (LeftSRC < [RightSRC] OR LeftSRC > [RightSRC + OperandSize/8])
THEN BOUND Range Exceeded Exception;
FI
```

Description

BOUND ensures that a signed array index is within the limits specified by a block of memory between an upper and lower bound. The register size determines whether the operation uses words or doublewords. The first operand (from the specified register) must be greater than or equal to the lower bound value, but not greater than the upper bound. The lower bound value is stored at the address specified by the second operand. The upper bound value is stored at a consecutive higher memory address (+2 for word operations; +4 for doubleword operations). If the first operand is out of the specified bounds, BOUND returns an Interrupt 15. The return EIP points to the BOUND instruction.

Flags Affected

None

Protected Mode Exceptions

If the test fails, BOUND generates a BOUND Range Exceeded (5) exception. General Protection Fault (13) indicates an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference. Invalid Opcode (6) occurs if BOUND uses a register as the second operand.

Real Address Mode Exceptions

BOUND Range Exceeded (5) indicates the test failed. Invalid Opcode (6) indicates the second operand is a register. General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

BOUND Range Exceeded (5) indicates the test failed. Invalid Opcode (6) indicates the second operand is a register. General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.12 BSF Bit Scan Forward

Opcode	Instruction	Clocks	Description
0F BC	BSF r16, r/m16	6–42/7–43	Performs a forward bit scan on r/m word.
0F BC	BSF r32, r/m32	6–42/7–43	Performs a forward bit scan on r/m doubleword.

Operation

```
IF r/m = 0
THEN
   ZF ← 1;
   register ← UNDEFINED;
ELSE
   temp ← 0;
   ZF ← 0;
   WHILE BIT[r/m, temp = 0];
   DO;
      temp ← temp + 1;
      register ← temp;
   OD;
FI
```

Description

BSF scans the bits in the second word or doubleword operand starting with bit 0. If all the bits are 0, BSF sets ZF. If any bit is not 0, BSF clears ZF and loads the destination register with the bit index of the first set bit.

Flags Affected

ZF is set if all bits are 0. If any bit is 1, BSF clears ZF.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.13 BSR Bit Scan Reverse

Opcode	Instruction	Clocks	Description
0F BD	BSR r16, r/m16	6–103/7–104	Performs a reverse bit scan on r/m word.
0F BD	BSR r32, r/m32	6–103/7–104	Performs a reverse bit scan on r/m doubleword.

Operation

```
IF r/m = 0
THEN
   ZF ← 1;
   register ← UNDEFINED;
ELSE
   temp ← OperandSize –1;
   ZF ← 0;
   WHILE BIT[r/m, temp = 0];
   DO;
      temp ← temp + 1;
      register ← temp;
   OD;
FI
```

Description

BSR scans the bits in the second word or doubleword operand from the most-significant bit to the least-significant bit. If all the bits are 0, BSR sets ZF. If any bit is not 0, BSR clears ZF and loads the destination register with the bit index of the first set bit found when scanning in the reverse direction.

Flags Affected

ZF is set if all bits are 0. If any bit is 1, BSR clears ZF.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.14 BSWAP Byte Swap

Opcode	Instruction	Clocks	Description
0F C8/r	BSWAP r32	1	Swaps bytes to convert little/big endian data in a 32-bit register to big/little endian form.

Operation

```
TEMP ← r32
r32(7..0) ← TEMP(31..24)
r32(15..8) ← TEMP(23..16)
r32(23..16) ← TEMP(15..8)
r32(31..24) ← TEMP(7..0)
```

Description

BSWAP reverses the byte order of a 32-bit register, converting a value in little/big endian form to big/little endian form. Applying BSWAP to a 16-bit operand leaves an undefined result in the destination register.

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The BSWAP instruction is not available on 386DX or SX microprocessors. If you are writing code that must be compatible with these systems, you must use 386 functionally-equivalent code to perform this operation.*

2.15 BT Bit Test

Opcode	Instruction	Clocks	Description
0F A3	BT r/m16, r16	3/8	Saves bit in Carry Flag.
0F A3	BT r/m32, r32	3/8	Saves bit in Carry Flag.
0F BA /4 /ib	BT r/m16, imm8	3/8	Saves bit in Carry Flag.
0F BA /4 /ib	BT r/m32, imm8	3/8	Saves bit in Carry Flag.

Operation

```
CF ← BIT[LeftSRC, RightSRC]
```

Description

BT saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into CF.

Flags Affected

CF contains the value of the selected bit.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *You can indicate the bit index by using a general register value or an immediate 8-bit constant. The operand is taken modulo 32, so the range of immediate bit offsets is 0–31. This allows you to select any bit in a word or doubleword register. For memory bit strings, you can support longer fields by using the immediate bit offset field in combination with the memory-operand displacement field. The Low order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the High order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode. When accessing a bit in memory, you can make the processor access two (16-bit operand) or four (32-bit operand) bytes from the starting address using:*

Effective Address + ([2 or 4] · (BitOffset DIV [16 or 32]))

You may use this form even if the processor only needs to access one byte to reach the given bit. When using this form, avoid referencing areas close to address space holes, and in particular, avoid references to memory-mapped I/O registers. Use MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

2.16 BTC Bit Test and Complement

Opcode	Instruction	Clocks	Description
0F BB	BTC r/m16, r16	6/13	Saves bit in Carry Flag and complement.
0F BB	BTC r/m32, r32	6/13	Saves bit in Carry Flag and complement.
0F BA /7 ib	BTC r/m16, imm8	6/8	Saves bit in Carry Flag and complement.
0F BA /7 ib	BTC r/m32, imm8	6/8	Saves bit in Carry Flag and complement.

Operation

```
CF ← BIT[LeftSRC, RightSRC]
BIT[LeftSRC, RightSRC] ← NOT BIT[LeftSRC, RightSRC]
```

Description

BTC saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into CF and complements the bit.

Flags Affected

CF contains the value of the selected bit.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *You can indicate the bit index by using a general register value or an immediate 8-bit constant. The operand is taken modulo 32, so the range of immediate bit offsets is 0–31. This allows you to select any bit in a word or doubleword register. For memory bit strings, you can support longer fields by using the immediate bit offset field in combination with the memory-operand displacement field. The Low order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the High order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode. When accessing a bit in memory, you can make the processor access two (16-bit operand) or four (32-bit operand) bytes from the starting address using:*

Effective Address + ([2 or 4] · (BitOffset DIV [16 or 32]))

You may use this form even if the processor only needs to access one byte to reach the given bit. When using this form, avoid referencing areas close to address space holes, and in particular, avoid references to memory-mapped I/O registers. Use MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

2.17 BTR Bit Test And Reset

Opcode	Instruction	Clocks	Description
0F B3	BTR r/m16, r16	6/13	Saves bit in Carry Flag and reset.
0F B3	BTR r/m32, r32	6/13	Saves bit in Carry Flag and reset.
0F BA /6 ib	BTR r/m16, imm8	6/8	Saves bit in Carry Flag and reset.
0F BA /6 ib	BTR r/m32, imm8	6/8	Saves bit in Carry Flag and reset.

Operation

```
CF← BIT[LeftSRC, RightSRC]
BIT[LeftSRC, RightSRC]← 0
```

Description

BTR saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into CF and resets the bit to 0.

Flags Affected

CF contains the value of the selected bit.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *You can indicate the bit index by using a general register value or an immediate 8-bit constant. The operand is taken modulo 32, so the range of immediate bit offsets is 0–31. This allows you to select any bit in a word or doubleword register. For memory bit strings, you can support longer fields by using the immediate bit offset field in combination with the memory-operand displacement field. The Low order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the High order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode. When accessing a bit in memory, you can make the processor access two (16-bit operand) or four (32-bit operand) bytes from the starting address using:*

Effective Address + ([2 or 4] · (BitOffset DIV [16 or 32]))

You may use this form even if the processor only needs to access one byte to reach the given bit. When using this form, avoid referencing areas close to address space holes, and in particular, avoid references to memory-mapped I/O registers. Use MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

2.18 BTS Bit Test And Set

Opcode	Instruction	Clocks	Description
0F AB	BTS r/m16, r16	6/13	Saves bit in Carry Flag and sets it to a 1.
0F AB	BTS r/m32, r32	6/13	Saves bit in Carry Flag and sets it to a 1.
0F BA /5 ib	BTS r/m16, imm8	6/8	Saves bit in Carry Flag and sets it to a 1.
0F BA /5 ib	BTS r/m32, imm8	6/8	Saves bit in Carry Flag and sets it to a 1.

Operation

```
CF ← BIT[LeftSRC, RightSRC]
BIT[LeftSRC, RightSRC]← 1
```

Description

BTS saves the value of the bit indicated by the base (first operand) and the bit offset (second operand) into CF and sets the bit to 1.

Flags Affected

CF contains the value of the selected bit.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *You can indicate the bit index by using a general register value or an immediate 8-bit constant. The operand is taken modulo 32, so the range of immediate bit offsets is 0–31. This allows you to select any bit in a word or doubleword register. For memory bit strings, you can support longer fields by using the immediate bit offset field in combination with the memory-operand displacement field. The Low order 3 to 5 bits of the immediate bit offset are stored in the immediate bit offset field, and the High order 27 to 29 bits are shifted and combined with the byte displacement in the addressing mode. When accessing a bit in memory, you can make the processor access two (16-bit operand) or four (32-bit operand) bytes from the starting address using:*

Effective Address + ([2 or 4] · (BitOffset DIV [16 or 32]))

You may use this form even if the processor only needs to access one byte to reach the given bit. When using this form, avoid referencing areas close to address space holes, and in particular, avoid references to memory-mapped I/O registers. Use MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

2.19 CALL Calls Procedure

Opcode	Instruction	Clocks	Description
E8 cw	CALL rel16	3	Calls near, displacement relative to next instruction.
FF /2	CALL r/m16	5/5	Calls near, register indirect/memory indirect.
9A cd	CALL ptr16:16	18, pm = 20	Calls far to full pointer given.
9A cd	CALL ptr16:16	pm = 35	Calls gate, same privilege.
9A cd	CALL ptr16:16	pm = 69	Calls gate, more privilege, no parameters.
9A cd	CALL ptr16:16	pm = 77+4x	Calls gate, more privilege, x parameters.
9A cd	CALL ptr16:16	pm = 37+ts*	Calls to task.
FF /3	CALL m16:16	17,pm = 20	Calls far to address at r/m word.
FF /3	CALL m16:16	pm = 35	Calls gate, same privilege.
FF /3	CALL m16:16	pm = 69	Calls gate, more privilege, no parameters.
FF /3	CALL m16:16	pm = 77+4x	Calls gate, more privilege, x parameters.
FF /3	CALL m16:16	pm = 37+ts*	Calls to task.
E8 cd	CALL rel32	3	Calls near, displacement relative to next instruction.
FF /2	CALL r/m32	5/5	Calls near, register indirect/memory indirect.
9A cp	CALL ptr16:32	18, pm = 20	Calls far to full pointer given.
9A cp	CALL ptr16:32	pm = 35	Calls gate, same privilege.
9A cp	CALL ptr16:32	pm = 69	Calls gate, more privilege, no parameters.
9A cp	CALL ptr16:32	pm = 77+4x	Calls gate, more privilege, x parameters.
9A cp	CALL ptr16:32	pm = 37+ts*	Calls to task.
FF /3	CALL m16:32	17,pm = 20	Calls far to address at r/m doubleword.
FF /3	CALL m16:32	pm = 35	Calls gate, same privilege.
FF /3	CALL m16:32	pm = 69	Calls gate, more privilege, no parameters.
FF /3	CALL m16:32	pm = 77+4x	Calls gate, more privilege, x parameters.
FF /3	CALL m16:32	pm = 37+ts*	Calls to task.

*ts = 199 for 486TSS, 180 for 286TSS, or 177 for VM TSS.

Operation

```
IF rel16 or rel32 type of call
THEN (* near relative call *)
   IF OperandSize = 16
   THEN
      Push(IP);
      EIP ← (EIP + rel16) AND 0000FFFFh;
   ELSE (* OperandSize = 32 *)
      Push(EIP);
      EIP ← EIP + rel32; FI; FI;

IF r/m16 or r/m32 type of call
THEN (* near absolute call *)
   OperandSize = 16
   THEN
      Push(IP);
      EIP ← [r/m16] AND 0000FFFFh;
   ELSE (* OperandSize = 32 *)
      Push(EIP)
      EIP ← [r/m32];FI; FI;

IF (PE = 0 OR (PE = 1 and VM = 1) [* real or Virtual 8086 Mode *]
   AND operand type = [m16:16, m16:32, ptr16:16, or ptr16:32]
THEN
   IF OperandSize = 16
   THEN
      Push(CS);
      Push(IP)
   ELSE
      Push(CS);
      Push(EIP)
   FI;
   IF operand type is m16:16 or m16:32
   THEN (* indirect far call *)
      IF OperandSize = 16
      THEN
         CS:IP ← [m16:16];
         EIP ← EIP AND 0000FFFFh; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *);
         CS:IP ← [m16:32]; FI;
   IF operand type is ptr16:16 or ptr16:32
   THEN (* direct far call *)
      IF OperandSize = 16
      THEN
         CS:IP ← ptr16:16;
         EIP ← EIP AND 0000FFFFh; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *);
         CS:IP ← ptr16:32; FI; FI

IF (PE = 1 AND VM = 0)(* Protected Mode, not V86 Mode *)
   AND instruction = far CALL
THEN
   If indirect, then check access of EA doubleword;
      General Protection Fault if limit violation;
   New CS selector must not be null else General Protection Fault;
   Check that new CS selector index is within its descriptor limits;
      else General Protection Fault(new CS selector);
   Examine AR byte of selected descriptor for various legal values;
      depending on value:
         go to CONFORMING-CODE-SEGMENT;
         go to NONCONFORMING-CODE-SEGMENT;
         go to CALL-GATE;
         go to TASK-GATE;
         go to TASK-GATE-SEGMENT;
   ELSE General Protection Fault(code segment selector); FI

CONFORMING-CODE-SEGMENT
   DPL must be ≤ CPL ELSE General Protection Fault(code segment selector);
   Segment must be present
      ELSE Segment Not Present Exception(code segment selector);
   Stack must be big enough for return address ELSE Stack Fault (12);
   Instruction pointer must be in code segment limit
      ELSE General Protection Fault;
   Load code segment descriptor into CS register;
   Load CS with new code segment selector;
   Load EIP with zero-extend(new offset);
   IF OperandSize = 16 THEN EIP ← EIP AND 0000FFFFh; FI;

NONCONFORMING-CODE-SEGMENT
   RPL must be ≤CPL ELSE General Protection Fault(code segment selector)
   DPL must be = CPL ELSE General Protection Fault(code segment selector)
   Segment must be present
      ELSE Segment Not Present Exception (code segment selector)
   Stack must be big enough for return address ELSE Stack Fault(0)
   Instruction pointer must be in code segment limit
      ELSE General Protection Fault
   Load code segment descriptor into CS register
   Load CS with new code segment selector
   Set RPL of CS to CPL
```

```
         Load EIP with zero-extend (new offset);
         IF OperandSize = 16 THEN EIP ← EIP AND 0000FFFFh; FI;

      CALL-GATE
         Call gate DPL must be ≥ CPL
            ELSE General Protection Fault(call gate selector)
         Call gate DPL must be ≥ RPL
            ELSE General Protection Fault(call gate selector)
         Call gate must be present
            ELSE Segment Not Present (11)(call gate selector)
         Examine code segment selector in call gate descriptor:
            Selector must not be null ELSE General Protection Fault
            Selector must be within its descriptor table limits
               ELSE General Protection Fault(code segment selector)
            AR byte of selected descriptor must indicate code
               segment ELSE General Protection Fault(code segment selector)
            DPL of selected descriptor must be ≤ CPL
               ELSE General Protection Fault(code segment selector)
            IF non-conforming code segment AND DPL < CPL
               THEN go to MORE-PRIVILEGE
               ELSE go to SAME-PRIVILEGE; FI;

      MORE-PRIVILEGE:
         Get new SS selector for new privilege level from TSS
            Check selector and descriptor for new SS:
               Selector must not be null ELSE Invalid TSS Exception(0)
               Selector index must be within its descriptor
                        table limits ELSE Invalid TSS Exception(SS selector)
               Selector’s RPL must equal DPL of code segment
                        ELSE Invalid TSS Exception(SS selector)
            Stack segment DPL must equal DPL of code
               segment ELSE Invalid TSS Exception(SS selector)
            Descriptor must indicate writable data segment
               ELSE Invalid TSS Exception(SS selector)
            Segment present ELSE Stack Fault(SS selector)
         IF OperandSize = 32
         THEN
            New stack must have room for parameters plus 16 bytes
               ELSE Invalid TSS Exception(SS selector)
            EIP must be in code segment limit ELSE General Protection Fault
            Load new SS:eSP value from TSS
            Load new CS: EIP value from gate
         ELSE
            New stack must have room for parameters plus 8 bytes
               ELSE Stack Fault (12)(SS selector)
            IP must be in code segment limit ELSE General Protection Fault
            Load new SS:eSP value from TSS
            Load new CS:IP value from gate;FI;
         Load CS descriptor
         Load SS descriptor
         Push long pointer of old stack onto new stack
         Get word count from call gate, mask to 5 bits
         Copy parameters from old stack onto new stack
         Push return address onto new stack
         Set CPL to stack segment DPL
         Set RPL of CS to CPL

      SAME-PRIVILEGE:
         IF OperandSize = 32
```

```
   THEN
      Stack must have room for 6-byte return address (padded to 8 bytes)
         ELSE Stack Fault
      EIP must be within code segment limit ELSE General Protection Fault
      Load CS:EIP from gate
   ELSE
      Stack must have room for 4-byte return address ELSE Stack Fault
      IP must be within code segment limit ELSE General Protection Fault
      Load CS:IP from gate
   FI;
      Push return address onto stack
      Load code segment descriptor into CS register
      Set RPL of CS to CPL

TASK-GATE
   Task gate DPL must be ≥ CPL ELSE Invalid TSS Exception(gate selector)
   Task gate DPL must be ≥ RPL ELSE Invalid TSS Exception(gate selector)
   Task Gate must be present
      ELSE Segment Not Present Exception(gate selector)
   Examine selector to TSS, given in Task Gate descriptor:
      Must specify global in the local/global bit
         ELSE Invalid TSS Exception (TSS selector)
      Index must be within GDT limits
         ELSE Invalid TSS Exception (TSS selector)
      TSS descriptor AR byte must specify nonbusy TSS
         ELSE Invalid TSS Exception(TSS selector)
      Task State Segment must be present
         ELSE Segment Not Present (11)(TSS selector)
   SWITCH-TASKS (with nesting) to TSS
   IP must be in code segment limit ELSE Invalid TSS Exception(0)

TO TASK-STATE-SEGMENT
   TSS DPL must be ≥ CPL ELSE Invalid TSS Exception(TSS selector)
   TSS DPL must be ≥ RPL ELSE Invalid TSS Exception(TSS selector)
   TSS descriptor AR byte must specify available TSS
      ELSE Invalid TSS Exception(TSS selector)
   Task State Segment must be present
      ELSE Segment Not Present (11)(TSS selector)
   SWITCH-TASKS (with nesting) to TSS
   IP must be in code segment limit ELSE Invalid TSS Exception(0)
```

Description

CALL exits the current instruction sequence and executes the procedure named in the operand. A return at the end of the CALLed procedure exits the procedure and starts execution at the instruction following the CALL instruction.

A CALL with a destination of r/m16, r/m32, rel16, or rel32 is a near CALL. It uses the current segment register value. The CALL rel16 and CALL rel32 forms add a signed offset to the address of the next instruction to determine the destination. Use the rel16 form if the next instruction uses a 16-bit (word) operand. Use the rel32 form if the next instruction uses a 32-bit (doubleword) operand. CALL stores the result in the 32-bit EIP register. With rel16, CALL clears the upper word of the EIP register, resulting in an offset whose value does not exceed 16 bits. CALL r/m16 and CALL r/m32 specify a register or memory location from which the absolute segment offset is fetched. CALL r/m16 fetches a 16-bit offset for a word operand; CALL r/m32 fetches a 32-bit offset for a doubleword operand. CALL pushes the offset of the next instruction in sequence onto the stack. The near RET instruction in the procedure pops the instruction offset when it returns control.

The far calls, CALL ptr16:16 and CALL ptr16:32, use a 4-byte or 6-byte operand as a long pointer to the called procedure. The CALL m16:16 and m16:32 forms fetch the long pointer from the memory location specified (indirection). In Real Address Mode or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on the operand-size attribute). These forms of the instruction push both the CS and IP or EIP registers as a return address.

In Protected Mode, both long pointer forms consult the AR byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the call will perform one of the following types of control transfers:

- A far call to the same protection level
- An inter-protection level far call
- A task switch

A CALL-indirect-through-memory, using the stack pointer (ESP) as a base register, references memory before the CALL. The base is the value of the ESP before the instruction executes.

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

For far calls: General Protection Fault (13), Segment Not Present (11), Stack Fault (12), and Invalid TSS (10), as indicated in Appendix A.

For near direct calls: General Protection Fault (13) if procedure location is beyond the code segment limits; Stack Fault (12) if pushing the return address exceeds the bounds of the stack segment; Page Fault Exception (14) for a page fault; Alignment Check (17) for unaligned memory reference if the current privilege level is 3.

For a near indirect call: General Protection Fault (13) for an illegal memory-operand effective address in the code or data segments; Stack Fault (12) for an illegal SS segment address; General Protection Fault (13) if the indirect offset obtained is beyond the code segment limits; Page Fault Exception (14) for a page fault; Alignment Check (17) for unaligned memory reference if the current privilege level is 3.

Real Address Mode Exceptions

General Protection Fault (13) if any part of the operand would lie outside of the effective address space from 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) if any part of the operand would lie outside of the effective address space from 0 to 0FFFFh. Page Fault Exception (14) for a page fault; Alignment Check (17) for aligned memory reference if the current privilege level is 3.

Note: *Any far call from a 32-bit code segment to a 16-bit code segment should be made from the first 64 Kbytes of the 32-bit code segment, because the operand-size attribute of the instruction is set to 16, allowing only a 16-bit return address offset to be saved.*

2.20 CBW Converts Byte to Word

Opcode	Instruction	Clocks	Description
98	CBW	3	AX ← sign-extend of AL

Operation

```
IF OperandSize = 16
THEN AX ← SignExtend (AL)
```

Description

The CBW instruction converts the signed byte in the AL register to a signed word in the AX register by extending the most-significant bit of the AL register (the sign bit) into all of the bits of the AH register.

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.21 CDQ Converts Doubleword to Quadword

Opcode	Instruction	Clocks	Description
99	CDQ	3	EDX:EAX ← sign-extend of EAX

Operation

```
IF OperandSize = 32
THEN
   IF EAX < 0
   THEN EDX ← 0FFFFFFFFh;
   ELSE EDX ← 0;
FI
```

Description

The CDQ instruction converts the signed doubleword in the EAX register to a signed 64-bit integer in the register pair EDX:EAX by extending the most-significant bit of the EAX register (the sign bit) into all the bits of the EDX register.

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.22 CLC Clears Carry Flag

Opcode	Instruction	Clocks	Description
F8	CLC	2	Clears Carry Flag.

Operation

```
CF ← 0
```

Description

CLC clears CF. It does not affect other flags or registers.

Flags Affected

CF is cleared.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.23 CLD Clears Direction Flag

Opcode	Instruction	Clocks	Description
FC	CLD	2	Clears Direction Flag to make the Stack Index (SI or ESI) and/or the Data Index (DI or EDI) Registers increment.

Operation

```
DF ← 0
```

Description

The CLD instruction clears the Direction Flag, causing all subsequent string operations to increment the index registers on which they operate: SI (8-bit or 16-bit operation) or ESI (32-bit operation), and/or DI (8-bit or 16-bit operation) or EDI (32-bit operation).

Flags Affected

DF is cleared. No other flags or registers are affected.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.24 CLI Clears Interrupt-Enable Flag

Opcode	Instruction	Clocks	Description
FA	CLI	5	Clears Interrupt-enable Flag: maskable interrupts disabled.

Operation

```
IF ← 0
```

Description

The CLI instruction clears IF if the current privilege level is at least as privileged as IOPL. No other flags are affected. External interrupts are not recognized at the end of the CLI instruction or from that point on until the IF flag is set.

Flags Affected

IF is cleared.

Protected Mode Exceptions

General Protection Fault (13) if the current privilege level is greater (has less privilege) than the I/O privilege level in the FLAGS register. The I/O privilege level specifies the least privileged level at which I/O can be performed.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

General Protection Fault (13) if the current privilege level is greater (has less privilege) than the I/O privilege level in the FLAGS register. The I/O privilege level specifies the least privileged level at which I/O can be performed.

2.25 CLTS Clears Task-Switched Flag in CR0

Opcode	Instruction	Clocks	Description
0F 06	CLTS	7	Clears Task-Switched flag.

Operation

```
TS Flag in CR0 ← 0
```

Description

The CLTS instruction clears the Task-Switched (TS) flag in the CR0 register. This flag is set by the microprocessor every time a task switch occurs. The TS flag is used to manage microprocessor extensions as follows:

- Every execution of an ESC instruction is trapped if the TS flag is set.
- Execution of a WAIT instruction is trapped if the MP flag and the TS flag are both set.

If a task switch occurs after an ESC instruction begins execution, you may need to save the floating-point unit's context before issuing a new ESC instruction. The fault handler saves the context and clears the IS flag.

The CLTS instruction appears in operating system software, not in application programs. It is a privileged instruction that only executes at privilege level 0.

Flags Affected

The TS flag is cleared (the TS flag is in the CR0 register, not the FLAGS or EFLAGS register).

Protected Mode Exceptions

General Protection Fault (13) if the CLTS instruction is executed with a current privilege level other than 0.

Real Address Mode Exceptions

None (valid in Real Address Mode to allow initialization for Protected Mode).

Virtual 8086 Mode Exceptions

General Protection Fault (13) if the CLTS instruction is executed with a current privilege level other than 0.

2.26 CMC Complements Carry Flag

Opcode	Instruction	Clocks	Description
F5	CMC	2	Complements the Carry Flag.

Operation

```
CF ← NOT CF
```

Description

The CMC instruction reverses the setting of CF. No other flags are affected.

Flags Affected

CF contains the complement of its original value.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.27 CMP Compares Two Operands

Opcode	Instruction	Clocks	Description
3C ib	CMP AL,imm8	1	Compares immediate byte to AL.
3D iw	CMP AX,imm16	1	Compares immediate word to AX.
3D id	CMP EAX,imm32	1	Compares immediate doubleword to EAX.
80 /7 ib	CMP r/m8,imm8	1/2	Compares immediate byte to r/m byte.
81 /7 iw	CMP r/m16,imm16	1/2	Compares immediate word to r/m word.
81 /7 id	CMP r/m32,imm32	1/2	Compares immediate doubleword to r/m doubleword.
83 /7 ib	CMP r/m16,imm8	1/2	Compares sign extended immediate byte to r/m word.
83 /7 ib	CMP r/m32,imm8	1/2	Compares sign extended immediate word to r/m doubleword.
38 /r	CMP r/m8,48	1/2	Compares byte register to r/m byte.
39 /r	CMP r/m16,r16	1/2	Compares word register to r/m word.
39 /r	CMP r/m32,r32	1/2	Compares doubleword register to r/m doubleword.
3A /r	CMP 48,4/m8	1/2	Compares r/m byte to byte register.
3B /r	CMP r16,r/m16	1/2	Compares r/m word to word register.
3B /r	CMP r32,r/m32	1/2	Compares r/m doubleword to doubleword register.

Operation

```
LeftSRC – SignExtend(RightSRC);
(* CMP does not store a result; its purpose is to set the flags *)
```

Description

CMP subtracts the second operand from the first, but does not store the result; CMP only changes the flag settings. The CMP instruction is typically used in conjunction with conditional jumps and the conditional SET instructions. (Refer to Appendix D for the list of signed and unsigned flag tests provided.) If an operand greater than one byte is compared to an immediate byte, the byte value is first sign-extended.

Flags Affected

The result determines the OF, SF, ZF, AF, PF, and CF settings.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.28 CMPS/CMPSB/CMPSD/CMPSW Compares Two String Operands

Opcode	Instruction	Clocks	Description
A6	CMPS m8,m8	8	Compares bytes ES:DI (second operand) with SI (first operand).
A7	CMPS m16,m16	8	Compares words ES:DI (second operand) with SI (first operand).
A7	CMPS m32,m32	8	Compares doublewords ES:EDI (second operand) with ESI (first operand).
A6	CMPSB	8	Compares bytes ES:DI with DS:SI.
A7	CMPSD	8	Compares doublewords ES:EDI with DS:SI.
A7	CMPSW	8	Compares words ES:DI with DS:SI.

Operation

```
IF OperandSize = 8 (* byte *)
THEN
   SI – DI
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← –1; FI;
IF OperandSize = 16 (* word *)
THEN
   SI – DI
   IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← –2; FI;
IF OperandSize = 32 (* doubleword *)
THEN
   ESI – EDI
   IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← –4; FI;
FI;
source-index = source-index + IncDec;
destination-index = destination-index + IncDec
```

Note: *If AddressSize = 16, SI = Source Index and DI = Destination Index.*
If AddressSize = 32, ESI = Source Index and EDI = Destination Index.

Description

CMPS compares the byte, word, or doubleword pointed to by the SI (8- or 16-bit operation) or ESI (32-bit operation) register with the byte, word, or doubleword pointed to by the DI (8- or 16-bit operation) or EDI (32-bit operation) register. You must preload the registers before executing CMPS.

CMPS subtracts the (E)DI indexed operand from the (E)SI indexed operand. This is the reverse of the usual AMD convention in which the left operand is the destination and the right operand is the source. No result is stored; only the flags reflect the change. The operand size determines whether bytes, words, or doublewords are compared. The first operand (SI or ESI) uses the DS register unless a segment override byte is present. The second operand (DI or EDI) must be addressable from the ES register; no segment override is possible. After the comparison, both the source-index register and the destination-index register are automatically advanced. If DF is 0, the registers increment according to the operand size (byte = 1; word = 2; doubleword = 4); if DF is 1, the registers decrement.

CMPSB, CMPSD, and CMPSW instructions are synonymous with the byte, doubleword, and word CMPS instructions, respectively.

The CMPS instruction can be preceded by the REPE or REPNE prefix for block comparison of CX or ECX bytes, words, or doublewords. Refer to the description of the REP instruction for more information on this operation.

Flags Affected

OF, SF, ZF, AF, PF, and CF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.29 CMPXCHG Compares And Exchanges

Opcode	Instruction	Clocks	Description
0F B0 /r	CMPXCHG r/m8,r8	6/7 if equal; 6/10 if not	Compares AL with r/m byte. If equal, sets ZF and loads byte register into r/m byte; otherwise, clears ZF and loads r/m byte into AL.
0F B1 /r	CMPXCHG r/m16,r16	6/7if equal; 6/10 if not	Compares AX with r/m word. If equal, sets ZF and loads word register into r/m word; otherwise, clears ZF and loads r/m word into AX.
0F B1/r	CMPXCHG r/m32,r32	6/7 if equal; 6/10 if not	Compares EAX with r/m doubleword. If equal, sets ZF and loads doubleword register into r/m doubleword; otherwise, clears ZF and loads r/m doubleword into EAX.

Operation

```
IF accumulator = DEST
   ZF ← 1
   DEST ← SRC
ELSE
   ZF ← 0
   accumulator ← DEST
```

Description

CMPXCHG compares the accumulator (AL, AX, or EAX register) with DEST. If they are equal, SRC is loaded into DEST. Otherwise, DEST is loaded into the accumulator.

Flags Affected

CF, PF, AF, SF, and OF are affected as if a CMP instruction had been executed with DEST and the accumulator as operands. ZF is set if the destination operand and the accumulator are equal; otherwise it is cleared.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *This instruction can be used with a LOCK prefix. In order to simplify the interface to the microprocessor's bus, the destination operand receives a write cycle without regard to the result of the comparison. DEST is written back if the comparison fails, and SRC is written into the destination otherwise. (The microprocessor never produces a locked read without also producing a locked write.) This instruction is not supported by 386 processors.*

2.30 CWD Converts Word to Doubleword Using DX:AX Register Pair

Opcode	Instruction	Clocks	Description
99	CWD	3	DX:AX ← sign-extend of AX

Operation

```
IF OperandSize = 16
THEN
   IF AX < 0
   THEN DX ← 0FFFFh;
   ELSE DX ← 0;
FI
```

Description

The CWD instruction converts the signed word in the AX register to a signed doubleword in the DX:AX register pair by extending the most-significant bit of the AX register into all the bits of the DX register.

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.31 CWDE Converts Word to Doubleword Using EAX Register

Opcode	Instruction	Clocks	Description
98	CWDE	3	EAX ← sign-extend of AX

Operation

```
IF OperandSize = 32
THEN EAX ← SignExtend (AX)
```

Description

The CWDE instruction converts the signed word in the AX register to a doubleword in the EAX register by extending the most-significant bit of the AX register into the two most-significant bytes of the EAX register.

Note: *The CWDE instruction is different from the CWD instruction. The CWD instruction uses the DX:AX register pair rather than the EAX register as a destination.*

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.32 DAA Decimal Adjusts AL after Addition

Opcode	Instruction	Clocks	Description
27	DAA	2	Decimal adjusts AL after addition.

Operation

```
IF ((AL AND 0Fh) > 9) OR (AF = 1)
THEN
   AL ← AL + 6;
   AF ← 1;
ELSE
   AF ← 0;
FI;
IF (AL > 9Fh) On (CF = 1)
THEN
   AL ← AL + 60h;
   CF ← 1;
ELSE CF ← 0;
FI
```

Description

Execute the DAA instruction only after executing an ADD instruction that leaves a two-BCD-digit byte result in the AL register. The ADD operands should consist of two packed BCD digits. The DAA instruction adjusts the AL register to contain the correct two-digit packed decimal result.

Flags Affected

AF and CF are set if there is a decimal carry, cleared if there is no decimal carry; SF, ZF and PF are set according to the result. OF is undefined.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.33 DAS Decimal Adjusts AL after Subtraction

Opcode	Instruction	Clocks	Description
2F	DAS	2	Decimal adjusts after subtraction.

Operation

```
IF (AL AND 0Fh) > 9 OR AF = 1
THEN
   AL ← AL – 6;
   AF ← 1;
ELSE
   AF ← 0;
FI;
IF (AL > 9Fh) OR (CF = 1)
THEN
   AL ← AL – 60h;
   CF ← 1;
ELSE CF ← 0;
FI
```

Description

Execute the DAS instruction only after a subtraction instruction that leaves a two-BCD digit byte result in the AL register. The operands should consist of two packed BCD digits. The DAS instruction adjusts the AL register to contain the correct packed two-digit decimal result.

Flags Affected

AF and CF are set if there is a decimal carry, cleared if there is no decimal carry; SF, ZF and PF are set according to the result. OF is undefined.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.34 DEC Decrements by 1

Opcode	Instruction	Clocks	Description
FE /1	DEC r/m8	1/3	Decrements r/m byte by 1.
FF /1	DEC r/m16	1/3	Decrements r/m word by 1.
FF /1	DEC r/m32	1/3	Decrements r/m doubleword by 1.
48 + rw	DEC r16	1	Decrements word register by 1.
48 + rw	DEC r32	1	Decrements doubleword register by 1.

Operation

```
DEST ← DEST – 1
```

Description

The DEC instruction subtracts 1 from the operand. The DEC instruction does not change CF. To affect CF, use the SUB instruction with an immediate operand of 1.

Flags Affected

OF, SF, ZF, AF, and PF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.35 DIV Unsigned Divide

Opcode	Instruction	Clocks	Description
F6 /6	DIV AL, r/m8	16/16	Unsigned division of AX by r/m byte (AL = Quo, AH = Rem).
F7 /6	DIV AX,r/m16	24/24	Unsigned division of DX:AX by r/m word (AX = Quo, DX = Rem).
F7 /6	DIV EAX,r/m32	40/40	Unsigned division of EDS:EAX by r/m doubleword (EAX = Quo, EDX = Rem).

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Divide By Zero Exception 0;
ELSE
   quotient ← temp;
   remainder ← dividend MOD (r/m);
FI
```

Note: *Divisions are unsigned. The divisor is given by the r/m operand. The dividend, quotient, and remainder use implicit registers. Refer to the table under 'Description.'*

Description

The DIV instruction performs an unsigned division. The dividend is implicit; only the divisor is given as an operand. The remainder is always less than the divisor. The type of the divisor determines which registers to use as follows:

Size	Divisor	Quotient	Remainder	Dividend
byte	AX	r/m8	AL	AH
word	DX:AX	r/m16	AX	DX
doubleword	EDX:EAX	r/m32	EAX	EDX

Flags Affected

OF, SF, ZF, AF, PF, and CF are undefined.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Divide By Zero Exception 0 if the quotient is too big to fit in the designated register (AL, AX, or EAX), or if the divisor is 0. General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

Divide By Zero Exception 0 if the quotient is too big to fit in the designated register (AL, AX, or EAX), or if the divisor is 0. General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.36 ENTER Makes Stack Frame for Procedure

Opcode	Instruction	Clocks	Description
C8 /w 00	ENTER imm16,0	14	Makes procedure stack frame.
C8 /w 01	ENTER imm16,1	17	Makes stack frame for procedure parameters.
C8 /w ib	ENTER imm16,imm8	17 + 3n	Makes stack frame for procedure parameters.

Operation

```
level ← level MOD 32
IF OperandSize = 16 THEN Push(BP) ELSE Push (EBP) FI;
   (* Save stack pointer *)
frame-ptr ← eSP
IF level > 0
THEN (* level is rightmost parameter *)
   FOR i ← 1 TO level – 1
   DO
      IF OperandSize = 16
      THEN
         BP ← BP – 2;
         Push [BP]
      ELSE (* OperandSize = 32 *)
         EBP ← EBP – 4;
         Push[EBP]; FI;
   OD;
   Push(frame-ptr)
FI;
IF OperandSize = 16 THEN BP ← frame-ptr ELSE EBP ← frame-ptr; FI;
IF StackAddrSize = 16
THEN SP ← SP – First operand;
ELSE ESP ← ESP – ZeroExtend (First operand); FI
```

Description

ENTER creates the stack frame required by most block-structured high-level languages. The first operand specifies the number of allocated dynamic storage bytes. The second operand gives the lexical nesting level (0–31) of the routine within the high-level language source code and determines the number of stack frame pointers copied into the new stack frame from the preceding frame. The processor uses the BP (word) or EBP (doubleword) register as the frame pointer and the SP (word) or ESP (doubleword) register as the stack pointer. If the second operand is 0, ENTER pushes the frame pointer onto the stack, subtracts the first operand from the stack pointer, and sets the frame pointer to the current stack-pointer value.

Flags Affected

None

Protected Mode Exceptions

Stack Fault (12) if SP or ESP exceeds the stack limit. Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.37 F2XM1 Computes 2^x–1

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F0	F2XM1	242 (140–279)	2	Replaces ST with (2^{ST} – 1).

Operation

```
ST ← (2^ST – 1)
```

Description

F2XM1 replaces the contents of ST with (2^{ST}–1). ST must lie in the range –1 < ST < 1.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If the operand is outside the acceptable range, the result of F2XM1 is undefined. The F2XM1 instruction is designed to produce a very accurate result even when the operand is close to zero. Larger errors are incurred for operands with magnitudes very close to 1. Values other than 2 can be exponentiated using the formula:*

$$x^y = 2^{(y \cdot \log_2 x)}$$

The instructions FLDL2T and FLDL2E load the constants $\log_2 10$ *and* $\log_2 e$*, respectively. FYL2X can be used to calculate* $y \cdot \log_2 x$ *for arbitrary positive x.*

2.38 FABS Absolute Value

Opcode	Instruction	Clocks	Description
D9 E1	FABS	3	Replaces ST with its absolute value.

Operation

```
sign bit of ST ← 0
```

Description

The absolute value instruction clears the sign bit of ST. This operation leaves a positive value unchanged, or replaces a negative value with a positive value of equal magnitude.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *The invalid-operation exception is raised only on stack underflow, even if the operand is signaling NaN or is in an unsupported format.*

2.39 FADD Adds Floating Point

Opcode	Instruction	Clocks	Concurrent Execution	Description
D8 /0	FADD m32 real	10 (8–20)	7 (5–17)	Adds m32real to ST.
DC /0	FADD m64 real	10 (8–20)	7 (5–17)	Adds m64real to ST.
D8 C0+i	FADD ST,ST(i)	10 (8–20)	7 (5–17)	Adds ST(i) to ST.
DC C0+i	FADD ST(i),ST	10 (8–20)	7 (5–17)	Adds ST to ST(i).

Operation

```
DEST ← DEST +SRC
```

Description

The addition instructions add the source and destination operands and return the sum to the destination. The operand at the stack top can be doubled by coding:

```
FADD ST, ST(0)
```

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.40 FADDP Adds Floating Point and Pops FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE C0+i	FADDP ST(i),ST	10 (8–20)	7 (5–17)	Adds ST to ST(i) and pops ST.
DE C1	FADDP	10 (8–20)	7 (5–17)	Adds ST to ST(1) and pops ST.

Operation

```
DEST ← DEST +SRC;
pop ST;
FI
```

Description

The addition instructions add the source and destination operands, return the sum to the destination, and pop the stack.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.41 FBLD Loads Binary Coded Decimal

Opcode	Instruction	Clocks	Concurrent Execution	Description
D8 /4	FBLD m80 dec	75 (70–103)	7.7 (2–8)	Pushes m80dec onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← SRC
```

Description

FBLD converts the BCD source operand into extended-real format and pushes it onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *The source is loaded without rounding error. The sign of the source is preserved, including the case where the value is negative zero. The packed decimal digits are assumed to be in the range 0–9. The instruction does not check for invalid digits (A–Fh) and the result of attempting to load an invalid encoding is undefined. ST(7) must be empty to avoid causing an invalid-operation exception.*

2.42 FBSTP Stores Binary Coded Decimal and Pops FPU Stack Top

Opcode	Instruction	Clocks	Description
DF /6	FBSTP m80dec	175 (172–176)	Stores ST in m80dec and pops ST.

Operation

```
DEST ← ST(0);
pop ST FI
```

Description

FBSTP converts the value in ST into a packed decimal integer, stores the result at the destination in memory, and pops ST. Non-integral values are first rounded according to the RC field of the control word.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.43 FCHS Changes Sign

Opcode	Instruction	Clocks	Description
D9 E0	FCHS	6	Replaces ST with a value of opposite sign.

Operation

```
sign bit of ← ST NOT (sign bit of ST)
```

Description

The FCHS instruction inverts the sign bit of ST. This operation replaces a positive value with a negative value of equal magnitude, or vice versa.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *The invalid-operation exception is raised only on stack underflow, even if the operand is a signaling NaN or is in an unsupported format.*

2.44 FCLEX Clears Exceptions after Checking for FPU Error

Opcode	Instruction	Clocks	Description
9B DB E2	FCLEX	7 + 3+ for FWAIT	Clears floating-point exception flags after checking for floating-point error conditions.

Operation

```
SW[0–7] ← 0;
SW[15] ← 0
```

Description

FCLEX clears the exception flags, the exception status flag, and the busy flag of the FPU status word after checking for floating-point error conditions.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

2.45 FCOM Compares Real

Opcode	Instruction	Clocks	Description
C8 /2	FCOM m32real	4	Compares ST with m32real.
DC /2	FCOM m64real	4	Compares ST with m64real.
D8 D0+i	FCOM st(i)	4	Compares ST with ST(i).
D8 D1	FCOM	4	Compares ST with ST(1).

Operation

```
CASE (relation of operands) OF
   Not comparable:C3, C2, C0 ← 111;
   ST > SRC:      C3, C2, C0 ← 000;
   ST < SRC:      C3, C2, C0 ← 001;
   ST = SRC:      C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
FI
```

Description

FCOM compares the stack top to the source, which can be a register or a 32-bit or 64-bit real memory operand. If no operand is encoded, ST is compared to ST(1). Following the instruction, the condition codes reflect the relation between ST and the source operand.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are set as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised and the condition bits are set to “unordered.” The sign of zero is ignored, so that – 0.0 = + 0.0.*

2.46 FCOMP Compares Real and Pops FPU Stack Top

Opcode	Instruction	Clocks	Description
D8 /3	FCOMP m32real	4	Compares ST with m32real and pops ST.
DC /3	FCOMP m64real	4	Compares ST with m64real and pops ST.
D8 D8+i	FCOMP ST(i)	4	Compares ST with ST(i) and pops ST.
D8 D9	FCOMP	4	Compares ST with ST(1) and pops ST.

Operation

```
CASE (relation of operands) OF
   Not comparable:C3, C2, C0 ← 111;
   ST > SRC:      C3, C2, C0 ← 000;
   ST < SRC:      C3, C2, C0 ← 001;
   ST = SRC:      C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
pop ST; FI
```

Description

FCOMP compares the stack top to the source, which can be a register or a single or double-real memory-operand, and then pops the stack. If no operand is encoded, ST is compared to ST(1). Following the instruction, the condition codes reflect the relation between ST and the source operand.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are set as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised, and the condition bits are set to “unordered.” The sign of zero is ignored, so that – 0.0 = + 0.0.*

2.47 FCOMPP Compares Real and Pops FPU Stack Top Twice

Opcode	Instruction	Clocks	Description
DE D9	FCOMPP	5	Compares ST with ST(1) and pops ST twice.

Operation

```
CASE (relation of operands) OF
   Not comparable:C3, C2, C0 ← 111;
   ST > ST(1):    C3, C2, C0 ← 000;
   ST < ST(1):    C3, C2, C0 ← 001;
   ST = ST(1):    C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
pop ST; pop ST; FI
```

Description

FCOMPP compares the stack top to ST(1). Following the instruction, the condition codes reflect the relation between ST and ST(1).

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are set as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised, and the condition bits are set to “unordered.” The sign of zero is ignored, so that – 0.0 = + 0.0.*

2.48 FCOS Cosine

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 FF	FCOS	241 (193–279)	2	Replaces ST with its cosine.

Operation

```
IF operand is in range
THEN
   C2 ← 0;
   ST ← cos (ST);
ELSE
   C2 ← 1;
FI
```

Description

The cosine instruction replaces the contents of ST with cos (ST). ST, expressed in radians, must lie in the range $|\theta| < 2^{63}$.

FPU Flags Affected

If C2 = 0 (reduction complete), the result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); if PE is set, C1 indicates whether the last rounding was upward. If C2 = 1 (reduction incomplete), C1 is undefined. C0 and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If the operand is outside the acceptable range, the C2 flag is set and ST remains unchanged. Reduce the operand to an absolute value smaller than 2^{63} by subtracting an appropriate integer multiple of 2π. For π, use the full 66-bit internal π used by the FPU: $4 \cdot$ 0.C90FDAA22168C234Ch. This ensures that the results are consistent with argument reduction used by the FPU for trigonometric functions.You cannot represent this number as an extended-real value, however. A suggested solution is to represent π as the sum of a highπ (the 33 most-significant bits) and a lowπ (the 33 least-significant bits). The Am486 processor checks for interrupts while performing this instruction. It aborts execution to service an interrupt.*

If you need to compute sine and cosine, use FSINCOS for faster execution.

2.49 FDECSTP Decrements Top-of-Stack Pointer

Opcode	Instruction	Clocks	Description
D9 F6	FDECSTP	3	Decrements top-of-stack pointer for FPU register stack.

Operation

```
IF TOP = 0
THEN TOP ← 7;
ELSE TOP ← TOP – 1;
FI
```

Description

FDECSTP subtracts one (without carry) from the 3-bit TOP field of the FPU status word.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *The effect of FDECSTP is to rotate the stack. It does not alter register tags or contents, nor does it transfer data.*

2.50 FDIV Divides Real

Opcode	Instruction	Clocks	Concurrent Execution	Description
D8 /6	FDIV m32real	73	70	Divides ST by m32real.
DC /6	FDIV m64real	73	70	Divides ST by m64real.
D8 F0+i	FDIV ST,ST(i)	73	70	Divides ST by ST(i).
DC F8+i	FDIV ST(i),ST	73	70	Replaces ST(i) with ST ÷ ST(i).

Operation

```
DEST ← ST ÷ other operand
```

Description

The division instructions divide the stack top by the other operand and return the quotient to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it automatically converts to the extended-real format. The performance of division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the division instruction executes in 62 clocks. If the specified precision is 24 bits, the division instruction takes only 35 clocks.*

2.51 FDIVP Divides Real and Pops FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE F8+i	FDIVP ST(i),ST	73	70	Replaces ST(i) with ST ÷ ST(i); pops ST.
DE F9	FDIVP	73	70	Replaces ST(1) with ST ÷ ST(1); pops ST.

Operation

```
DEST ← ST÷ other operand;
pop ST FI
```

Description

The division instructions divide the stack top by the other operand and return the quotient to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

If the source operand is in memory, it automatically converts to the extended-real format. The performance of division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the division instruction executes in 62 clocks. If the specified precision is 24 bits, the division instruction takes only 35 clocks.

2.52 FDIVR Reverse Divides Real

Opcode	Instruction	Clocks	Concurrent Execution	Description
D8 /7	FDIVR m32real	73	70	Replaces ST with m32real ÷ ST.
DC /7	FDIVR m64real	73	70	Replaces ST with m64real ÷ ST.
D8 F8+i	FDIVR ST,ST(i)	73	70	Replaces ST with ST(i) ÷ ST.
DC F0+i	FDIVR ST(i),ST	73	70	Divides ST(i) by ST.

Operation

```
DEST ← other operand ÷ ST
```

Description

The division instructions divide the other operand by the stack top and return the quotient to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it automatically converts to the extended-real format. The performance of division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the division instruction executes in 62 clocks. If the specified precision is 24 bits, the division instruction takes only 35 clocks.*

2.53 FDIVRP Reverse Divides Real and Pops FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE F0+i	FDIVRP ST(i),ST	73	70	Divides ST(i) by ST and pops ST
DE F1	FDIVRP	73	70	Divides ST(1) by ST and pops ST

Operation

```
DEST ← other operand ÷ ST;
pop ST FI
```

Description

The division instructions divide the other operand by the stack top and return the quotient to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it automatically converts to the extended-real format. The performance of division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the division instruction executes in 62 clocks. If the specified precision is 24 bits, the division instruction takes only 35 clocks.*

2.54 FFREE Free Floating-Point Register

Opcode	Instruction	Clocks	Description
DD C0+i	FFREE ST(i)	3	Tags ST(i) as empty.

Operation

```
TAG(i) ← 11B
```

Description

FFREE tags the destination register as empty.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *FFREE does not affect the contents of the destination register. The floating-point top-of-stack pointer (TOP) is also unaffected.*

2.55 FIADD Adds Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
DA /0	FIADD m32int	22.5 (19–32)	7 (5–17)	Adds m32int to ST.
DE /0	FIADD m16int	24 (20–35)	7 (5–17)	Adds m16int to ST.

Operation

```
DEST ← DEST + SRC
```

Description

The addition instructions add the source and destination operands and return the sum to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.56 FICOM Compares Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE /2	FICOM m16intl	18 (16–20)	1	Compares ST with m16int.
DA /2	FICOM m32intl	16.5 (15–17)	1	Compares ST with m32int.

Operation

```
CASE (relation of operands) OF
   Not comparable:C3, C2, C0 ← 111;
   ST > SRC:      C3, C2, C0 ← 000;
   ST < SRC:      C3, C2, C0 ← 001;
   ST = SRC:      C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
FI
```

Description

FICOM compares the stack top to the source. Following the instruction, the condition codes reflect the relation between ST and the source operand.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. The values of C0, C2, and C3 are as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *The memory operand is converted to extended-real format before the comparison is performed. If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised and the condition bits are set to “unordered.”*

2.57 FICOMP Compares Integer and Pops FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE /3	FICOMP m16int	18 (16–20)	1	Compares ST with m16int and pops ST.
DA /3	FICOMP m32int	16.5 (15–17)	1	Compares ST with m32int and pops ST.

Operation

```
CASE (relation of operands) OF
   Not comparable:C3, C2, C0 ← 111;
   ST > SRC:      C3, C2, C0 ← 000;
   ST < SRC:      C3, C2, C0 ← 001;
   ST = SRC:      C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
pop ST FI
```

Description

FICOMP compares the stack top to the source, then pops the stack top. Following the instruction, the condition codes reflect the relation between ST and the source operand.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. The values of C0, C2, and C3 are as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *The memory operand is converted to extended-real format before the comparison is performed. If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised and the condition bits are set to “unordered.”*

2.58 FIDIV Divides Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
DA /6	FIDIV m32int	73	70	Divides ST by m32int.
DE /6	FIDIV m16int	73	70	Divides ST by m16int.

Operation

```
DEST ← ST÷ other operand
```

Description

The division instructions divide the stack top by the other operand and return the quotient to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it automatically converts to the extended-real format. The performance of division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the division instruction executes in 62 clocks. If the specified precision is 24 bits, the division instruction takes only 35 clocks.*

2.59 FIDIVR Reverse Divides Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
DA /7	FIDIVR m32int	73	70	Replaces ST with m32int ÷ ST.
DE /7	FIDIVR m16int	73	70	Replaces ST with m16int ÷ ST.

Operation

```
DEST ← other operand ÷ ST
```

Description

The division instructions divide the other operand by the stack top and return the quotient to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it automatically converts to the extended-real format. The performance of division instructions depends on the PC (Precision Control) field of the FPU control word. If PC specifies a precision of 53 bits, the division instruction executes in 62 clocks. If the specified precision is 24 bits, the division instruction takes only 35 clocks.*

2.60 FILD Loads Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
DF /0	FILD m16int	14.5 (13–16)	4	Pushes m16int onto FPU stack.
DB /0	FILD m32int	11.5 (9–12)	4 (2–4)	Pushes m32int onto FPU stack.
DF /5	FILD m64int	16.8 (10–18)	7.8 (2–8)	Pushes m64int onto FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← SRC
```

Description

FILD converts the source signed integer operand into extended-real format and pushes it onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *The source is loaded without rounding error. ST(7) must be empty to avoid causing an invalid-operation exception.*

2.61 FIMUL Multiplies Integer

Opcode	Instruction	Clocks	Description
DA /1	FIMUL m32int	8	Multiplies ST by m32int.
DE /1	FIMUL m16int	8	Multiplies ST by m16int.

Operation

```
DEST ← DEST · SRC
```

Description

The multiplication instructions multiply the destination operand by the source operand and return the product to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.62 FINCSTP Increments Top-of-Stack Pointer

Opcode	Instruction	Clocks	Description
D9 F7	FINCSTP	3	Increments top-of-stack pointer for FPU register stack.

Operation

```
IF TOP = 7
THEN TOP ← 0;
ELSE TOP ← TOP + 1;
FI
```

Description

FINCSTP adds one (without carry) to the 3-bit TOP field of the FPU status word.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) if either EM or TS in CR0 is set.

Note: *The effect of FINCSTP is to rotate the stack. It does not alter register tags or contents, nor does it transfer data. It is not equivalent to popping the stack because it does not set the tag of the old stack-top to empty.*

2.63 FINIT Initializes FPU after Checking for Unmasked FPU Error

Opcode	Instruction	Clocks	Description
DB E3	FINIT	17 + 3+ for FWAIT	Initializes FPU after checking for unmasked floating-point error condition.

Operation

```
CW ← 037Fh;                             (* Control word *)
SW ← 0;                                 (* Status word *)
TW ← FFFFh;                             (* Tag word *)
FEA ← 0; FDS ← 0;                       (* Data pointer *)
FIP ← 0; FOP ← 0; FCS ← 0               (* Instruction pointer *)
```

Description

The initialization instructions set the FPU into a known state, unaffected by any previous activity.

The FPU control word is set to 037Fh (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, stack register R0 = stack top). The stack registers are all tagged as empty. The error pointers (both instruction and data) are cleared.

FPU Flags Affected

C0, C1, C2, C3 cleared

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *FINIT leaves the FPU in the same state as that which results from a hardware RESET signal. Unlike the Intel 387 math coprocessor, FINIT clears the error pointers in the Am486 processor.*

2.64 FIST Stores Integer

Opcode	Instruction	Clocks	Description
DF /2	FIST m16int	33.4 (29–34)	Stores ST in m16int.
DB /2	FIST m32int	32.4 (28–34)	Stores ST in m32int.

Operation

```
DEST ← ST(0)
```

Description

FIST converts the value in ST into a signed integer according to the RC field of the control word and transfers the result to the destination. ST remains unchanged. FIST accepts word and short integer destinations.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** Negative zero is stored with the same encoding (00...00) as positive zero. If the value is too large to represent as an integer, an exception is raised. The masked response is to write the most negative integer to memory.*

2.65 FISTP Stores Integer and Pops FPU Stack Top

Opcode	Instruction	Clocks	Description
DF /3	FISTP m16int	33.4 (29–34	Stores ST in m16int and pops ST.
DB /3	FISTP m32int	33.4 (29–34)	Stores ST in m32int and pops ST.
DF /7	FISTP m64int	33.4 (29–34)	Stores ST in m64int and pops ST.

Operation

```
DEST ← ST(0);
pop ST FI
```

Description

FISTP converts the value in ST into a signed integer according to the RC field of the control word and transfers the result to the destination. ST remains unchanged. FISTP accepts word, short integer, and long integer destinations.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *Negative zero is stored with the same encoding (00...00) as positive zero. If the value is too large to represent as an integer, an exception is raised. The masked response is to write the most negative integer to memory.*

2.66 FISUB Subtracts Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
DA /4	FISUB m32int	22.5 (19–32)	7 (5–17)	Subtracts m32int from ST.
DE /4	FISUB m16int	24 (20–35)	7 (5–17)	Subtracts m16int from ST.

Operation

```
DEST ← ST – Other Operand
```

Description

The subtraction instructions subtract the other operand from the stack top and return the difference to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.67 FISUBR Reverse Subtracts Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
DA /5	FISUBR m32int	22.5 (19–32)	7 (5–17)	Replaces ST with m32int – ST.
DE /5	FISUBR m16int	24 (20–35)	7 (5–17)	Replaces ST with m16int – ST.

Operation

```
DEST ← Other Operand – ST
```

Description

The reverse subtraction instructions subtract the stack top from the other operand and return the difference to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.68 FLD Loads Real

Opcode	Instruction	Clocks	Description
D9 /0	FLD m32real	3	Pushes m32real onto the FPU stack.
DD /0	FLD m64real	3	Pushes m64real onto the FPU stack.
DB /5	FLD m80real	6	Pushes m80real onto the FPU stack.
D9 C0+i	FLD ST(i)	4	Pushes ST(i) onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← SRC
```

Description

FLD pushes the source operand onto the FPU stack. If the source is an FPU stack register, the register number is computed from the top-of-stack pointer before it is decremented. Because of this instruction characteristic, the following coding duplicates the stack top:

```
FLD ST(0)
```

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in single or double-real format, it is automatically converted to the extended-real format. Loading an extended-real operand does not require conversion, so the I and D exceptions will not occur in this case. ST(7) must be empty to avoid causing an invalid-operation exception.*

2.69 FLD1 Loads Constant +1.0

Opcode	Instruction	Clocks	Description
D9 E8	FLD1	4	Pushes +1.0 onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← +1.0
```

Description

FLD1 pushes a +1.0 (in extended-real format) onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *ST(7) must be empty to avoid an invalid exception. An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.*

2.70 FLDCW Loads Control Word

Opcode	Instruction	Clocks	Description
D9 /5	FNLDCW m2byte	4	Loads the FPU control word from m2byte.

Operation

```
CW ← SRC
```

Description

FLDCW replaces the current value of the FPU control word with the value contained in the specified memory word.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None, except for unmasking an existing exception.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** FLDCW is typically used to establish or change the FPU's mode of operation. If an exception bit in the status word is set, loading a new control word that unmasks that exception will result in a floating-point error condition. When changing modes, the recommended procedure is to clear any pending exceptions before loading the new control word.*

2.71 FLDENV Loads FPU Environment

Opcode	Instruction	Clocks	Description
D9 /4	FLDENV m14/28byte	44 real or virtual/ 34 protected	Loads FPU environment from m14byte or m28byte.

Operation

```
FPU environment ← SRC
```

Description

FLDENV reloads the FPU environment from the memory area defined by the source operand. This data should be written by previous FSTENV or FNSTENV instruction. The FPU environment consists of the FPU control word, status word, tag word, and error pointers (both data and instruction). The environment layout in memory depends on both the operand size and the current operating mode of the microprocessor. The USE attribute of the current code segment determines the operand size: the 14-byte operand applies to a USE16 segment, and the 28-byte operand applies to a USE32 segment. FLDENV should be executed in the same operating mode as the corresponding FSTENV or FNSTENV.

FPU Flags Affected

C0, C1, C2, C3 as loaded

Numeric Exceptions

None, except for loading an unmasked exception.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** If the environment image contains an unmasked exception, loading it will result in a floating-point error condition.*

2.72 FLDL2E Loads Constant $\log_2 e$

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 EA	FLDL2E	8	2	Pushes $\log_2 e$ onto the FPU Stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← $\log_2 e$
```

Description

FLDL2E pushes $\log_2 e$ (in extended-real format) onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *ST(7) must be empty to avoid an invalid exception. An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.*

2.73 FLDL2T Loads Constant $\log_2 10$

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 E9	FLDL2T	8	2	Pushes $\log_2 10$ onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← log₂10
```

Description

FLDL2T pushes $\log_2 10$ (in extended-real format) onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *ST(7) must be empty to avoid an invalid exception. An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.*

2.74 FLDLG2 Loads Constant $\log_{10}2$

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 EC	FLDLG2	8		Pushes $\log_{10}2$ onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← log₁₀2
```

Description

FLDLG2 pushes $\log_{10}2$ (in extended-real format) onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *ST(7) must be empty to avoid an invalid exception. An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.*

2.75 FLDLN2 Loads Constant $\log_e 2$

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 ED	FLDLN2	8	2	Pushes $\log_e 2$ onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← log_e2
```

Description

FLDLN2 pushes $\log_e 2$ (in extended-real format) onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *ST(7) must be empty to avoid an invalid exception. An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.*

2.76 FLDPI Loads Constant π

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 EB	FLDPI	8	2	Pushes π onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← π
```

Description

FLDPI pushes π (in extended-real format) onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *ST(7) must be empty to avoid an invalid exception. An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.*

2.77 FLDZ Loads Constant +0.0

Opcode	Instruction	Clocks	Description
D9 EE	FLDZ	4	Pushes +0.0 onto the FPU stack.

Operation

```
Decrement FPU top-of-stack pointer;
ST(0) ← +0.0
```

Description

FLDZ pushes +0.00 (in extended-real format) onto the FPU stack.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *ST(7) must be empty to avoid an invalid exception. An internal 66-bit constant is used and rounded to external-real format (as specified by the RC bit of the control words). The precision exception is not raised.*

2.78 FMUL Multiplies Real

Opcode	Instruction	Clocks	Concurrent Execution	Description
D8 /1	FMUL m32real	11	8	Multiplies ST by m32real.
DC /1	FMUL m64real	14	11	Multiplies ST by m64real.
D8 C8+i	FMUL ST,ST(i)	16	13	Multiplies ST by ST(i).
DC C8+i	FMUL ST(i),ST	16	13	Multiplies ST(i) by ST.

Operation

```
DEST ← DEST · SRC
```

Description

The multiplication instructions multiply the destination operand by the source operand and return the product to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.79 FMULP Multiplies Real and Pops FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE C8+i	FMULP ST(i),ST	16	13	Multiplies ST(i) by ST and pops ST.
DE C9	FMULP	16	13	Multiplies ST(1) by ST and pops ST.

Operation

```
DEST ← DEST · SRC;
pop ST FI
```

Description

The multiplication instructions multiply the destination operand by the source operand and return the product to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.80 FNCLEX Clears Exceptions without Checking for FPU Error

Opcode	Instruction	Clocks	Description
D8 E2	FNCLEX	7	Clears floating-point exception flag without checking for floating-point error conditions.

Operation

```
SW[0–7] ← 0;
SW[15] ← 0
```

Description

FNCLEX clears the exception flags, the exception status flag, and the busy flag of the FPU status word.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

2.81 FNINIT Initializes FPU without Checking for Unmasked FPU Error

Opcode	Instruction	Clocks	Description
DB E3	FNINIT	17	Initializes FPU without checking for unmasked floating-point error condition.

Operation

```
CW ← 037Fh;                           (* Control word *)
SW ← 0;                               (* Status word *)
TW ← FFFFh;                           (* Tag word *)
FEA ← 0; FDS ← 0;                     (* Data pointer *)
FIP ← 0; FOP ← 0; FCS ← 0;            (* Instruction pointer *)
```

Description

The initialization instructions set the FPU into a known state, unaffected by any previous activity.

The FPU control word is set to 037Fh (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, stack register R0 = stack top). The stack registers are all tagged as empty. The error pointers (both instruction and data) are cleared.

FPU Flags Affected

C0, C1, C2, C3 cleared

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

***Note:** FNINIT leaves the FPU in the same state as that which results from a hardware RESET signal. Unlike the Intel 387 math coprocessor, FNINIT clears the error pointers in the Am486 processor.*

2.82 FNOP No Operation

Opcode	Instruction	Clocks	Description
D9 D0	FNOP	3	No operation is performed.

Description

FNOP performs no operation. If affects only the instruction pointers.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

2.83 FNSAVE Stores FPU State w/o Checking for Unmasked FPU Error

Opcode	Instruction	Clocks	Description
DD /6	FNSAVE m94/108byte	154 real or virtual/ 143 protected	Stores FPU environment to m94byte or m108byte without checking for unmasked floating-point error condition, and then reinitializes the FPU.

Operation

```
DEST ← FPU state;
initialize FPU; (* Equivalent to FNINIT *)
```

Description

FNSAVE writes the current FPU state (environment and register stack) to the specified destination, and then reinitializes the FPU, without checking for unmasked floating-point error conditions. The environment consists of the FPU control word, status word, tag word, and error pointers (both data and instruction). The state layout in memory depends on both the operand size and the current operating mode of the microprocessor. The USE attribute of the current code segment determines the operand size: the 94-byte operand applies to USE16 segment, and the 108-byte operand applies to a USE32 segment. The stack registers, ST(0) to ST(7), are in the 80 bytes immediately following the environment image.

FPU Flags Affected

C0, C1, C2, C3 cleared

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** FNSAVE does not store the FPU state until all FPU activity is complete; the saved image reflects the state of the FPU after any previously decoded instruction is executed. If a program must read from the memory image of the state after a save instruction, it must issue an FWAIT instruction to ensure that the storage is complete. The save instructions are typically used when an operating system needs to perform a context switch, or an exception handler needs to use the FPU, or an application program wants to pass a “clean” FPU to a subroutine.*

2.84 FNSTCW Stores Control Word without Checking for FPU Error

Opcode	Instruction	Clocks	Description
D9 /7	FNSTCW m2byte	3	Stores FPU control work to m2byte without checking for unmasked floating-point error condition.

Operation

```
DEST ← CW
```

Description

FNSTCW writes the current value of the FPU control word to the specified destination without checking for an unmasked floating-point error condition.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.85 FNSTENV Stores FPU Environment w/o Checking for FPU Error

Opcode	Instruction	Clocks	Description
D9 /6	FNSTENV m14/28byte	67 real or virtual/ 56 protected	Stores FPU environment to m14byte or m28byte without checking for unmasked floating-point error condition. Then masks all floating-point exceptions.

Operation

```
DEST ← FPU environment;
CW[0–5] ← 111111
```

Description

FNSTENV writes the current FPU environment to the specified destination, and then masks all floating-point exceptions without checking for unmasked floating-point error conditions. The FPU environment consists of the FPU control word, status word, tag word, and error pointer (both data and instruction). The environment layout in memory depends on both the operand size and the current operating mode of the microprocessor. The USE attribute of the current code segment determines the operand size: the 14-byte operand applies to a USE16 segment, and the 28-byte operand applies to a USE32 segment.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *FNSTENV does not store the environment until all FPU activity is complete; the saved environment reflects the state of the FPU after any previously decoded instruction has been executed. The store environment instructions are often used by exception handlers because they provide access to the FPU error pointers. The environment is typically saved onto the memory stack. After saving the environment, FNSTENV sets all the exception masks in the FPU control word. This prevents floating-point errors from interrupting the exception handler.*

2.86 FNSTSW Stores Status Word w/o Checking for Unmasked FPU Error

Opcode	Instruction	Clocks	Description
DF /7	FNSTSW m2byte	3	Stores FPU status word to m2byte without checking for unmasked floating-point error condition.
DF E0	FNSTSW AX	3	Stores FPU status word to AX register without checking for unmasked floating-point error condition.

Operation

```
DEST ← SW
```

Description

FNSTSW writes the current value of the FPU status word to the specified destination, which can be either a 2-byte location in memory or the AX register, without checking for an unmasked floating-point error condition.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *FNSTSW is used primarily in conditional branching (after a comparison, FPREM, FPREM1, or FXAM instruction). It can also invoke exception handlers (by polling the exception bits) in environments that do not use interrupts. When FNSTSW AX is executed, the AX register is updated before the Am486 microprocessor executes any further instructions. The status stored is that from the completion of the prior ESC instruction.*

2.87 FPATAN Partial Arctangent

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F3	FPATAN	289 (218–303)	5 (2–17)	Replaces ST(1) with arctan (ST(1) ÷ ST) and pops ST.

Operation

```
ST(1) ← arctan (ST(1) ÷ ST);
pop ST FI
```

Description

The partial arctangent instruction computes the arctangent of ST(1) ÷ ST and returns the computed value, expressed in radians, to ST(1). It then pops ST. The result has the same sign as the operand from ST(1) and a magnitude less than π.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *There is no restriction on the range of arguments that FPATAN can accept. The fact that FPATAN takes two arguments and computes the arctangent of their ratio simplifies the calculation of other trigonometric functions. For instance, arcsin (x) (which is the arctangent of* $x \div \sqrt{(1 - x2)}$*) can be computed using the following sequence of operations: Push x onto the FPU stack; compute* $\sqrt{(1 - x2)}$ *and push the resulting value onto the stack; execute FPATAN. The Am486 processor checks for interrupts while performing this instruction. It will abort this instruction to serve an interrupt.*

2.88 FPREM Partial Remainder (Non-IEEE 754 compliant)

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F8	FPREM	84 (70–138)	2 (2–8)	Replaces ST with the remainder obtained when dividing ST by ST(1).

Operation

```
EXPDIF ← exponent(ST) – exponent(ST(1));
IF EXPDIF < 64
THEN
   Q ← integer obtained by chopping ST ÷ ST(1) toward zero;
   ST ← ST – (ST(1) · Q);
   C2 ← 0;
   C0, C1, C3 ← three least-significant bits of Q; (* Q2, Q1, Q0 *)
ELSE
   C2 ← 1;
   N ← a number between 32 and 63
   QQ ← integer obtained by chopping (ST ÷ ST(1)) ÷ 2^(EXPDIF–N) toward zero;
   ST ← ST – (ST(1) · QQ · 2^(EXPDIF–N);
FI;
```

Description

FPREM computes the remainder of dividing ST by ST(1) using iterative subtraction and leaves the result in ST. The remainder’s sign is the same as the sign of the original dividend in ST. The magnitude of the remainder is less than that of the modulus.

FPU Flags Affected

If the IE and SF status word bits are set (stack exception), C1 indicates whether it is an overflow (C1 = 1) or underflow (C1 = 0); otherwise, C3 = Q0, C1 = Q1, and C0 = Q2 least-significant quotient bits. C2 indicates the reduction status: 0 = complete; 1 = incomplete.

Numeric Exceptions

Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *FPREM produces an exact result with no precision (inexact) exception and no rounding. FPREM does not comply with IEEE Std 754 (see FPREM1), but is compatible with 8087 and 80287 coprocessors. A higher-priority interrupting routine can force the FPU to switch context between the instructions in the remainder loop.*

FPREM can reduce periodic function arguments. C3, C1, and C0 represent the three least-significant quotient bits when execution is complete. This is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle within the correct sector of the unit circle.

2.89 FPREM1 Partial Remainder (IEEE 754 compliant)

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F5	FPREM1	94.5 (72–167)	5.5 (2–18)	Replaces ST with the remainder obtained when dividing ST by ST(1).

Operation

```
EXPDIF ← exponent(ST) – exponent(ST(1));
IF EXPDIF < 64
THEN
   Q ← integer obtained by chopping ST ÷ ST(1) toward zero;
   ST ← ST – (ST(1) · Q);
   C2 ← 0;
   C0, C1, C3 ← three least-significant bits of Q; (* Q2, Q1, Q0 *)
ELSE
   C2 ← 1;
   N ← a number between 32 and 63
   QQ ← integer obtained by chopping (ST ÷ ST(1)) ÷ 2^(EXPDIF–N) toward zero;
   ST ← ST – (ST(1) · QQ · 2^(EXPDIF–N);
FI;
```

Description

FPREM1 computes the remainder of dividing ST by ST(1) using iterative subtraction, and leaves the result in ST. The magnitude of the remainder is less than half that of the modulus.

FPU Flags Affected

If the IE and SF status word bits are set (stack exception), C1 indicates whether it is an overflow (C1 = 1) or underflow (C1 = 0); otherwise, C3 = Q0, C1 = Q1, and C0 = Q2 least-significant quotient bits. C2 indicates the reduction status: 0 = complete; 1 = incomplete.

Numeric Exceptions

Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *FPREM1 produces an exact result with no precision (inexact) exception and no rounding. FPREM1 complies with IEEE Std 754 (see also FPREM). A higher-priority interrupting routine can force the FPU to switch context between the instructions in the remainder loop.*

FPREM1 can reduce periodic function arguments. C3, C1, and C0 represent the three least-significant quotient bits when execution is complete. This is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle within the correct sector of the unit circle.

2.90 FPTAN Partial Tangent

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F2	FPTAN	244 (200–273)	70	Replaces ST with its tangent and push 1 onto the FPU stack.

Operation

```
IF operand is in range
THEN
   C2 ← 0;
   ST ← tan (ST);
   Decrement top-of-stack pointer;
   ST ← 1.0;
ELSE
   C2 ← 1;
FI
```

Description

FPTAN replaces the contents of ST with tan (ST), and then pushes 1.0 onto the FPU stack to maintain 8087 and 80287 compatibility. ST, expressed in radians, must lie in the range $| \theta | < 2^{63}$.

FPU Flags Affected

If C2 = 0 (reduction complete), the result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); if PE is set, C1 indicates whether the last rounding was upward. If C2 = 1 (reduction incomplete), C1 is undefined. C0 and C3 are always undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If the operand is outside the acceptable range, the C2 flag is set, and ST remains unchanged. Reduce the operand to an absolute value smaller than 2^{63} by subtracting an appropriate integer multiple of 2π. For π, use the value used as the full 66-bit internal π used by the FPU: $4 \cdot$ 0.C90FDAA22168C234Ch. This ensures that the results are consistent with argument reduction used by the FPU for trigonometric functions. You cannot represent this number as an extended-real value, however. A suggested solution is to represent π as the sum of a highπ (the 33 most-significant bits) and a lowπ (the 33 least-significant bits). The Am486 processor can abort this instruction to service an interrupt. ST(7) must be empty to avoid an invalid-operation exception.*

2.91 FRNDINT Rounds to Integer

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 FC	FRNDINT	29.1 (21–30)	7.4 (2–8)	Rounds ST to an integer.

Operation

```
ST ← rounded ST
```

Description

FRNDINT rounds the value in ST to an integer according to the RC field of the FPU control word.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

2.92 FRSTOR Restores FPU State

Opcode	Instruction	Clocks	Description
DB /4	FRSTOR m94/108byte	131 real or virtual/ 120 protected	Loads FPU state from m94byte or m108byte.

Operation

```
FPU state ← SRC;
```

Description

FRSTOR reloads the FPU state (environment and register stack) from the memory area defined by the source operand. This data should have been written by a previous FSAVE or FNSAVE instruction.

The FPU environment consists of the FPU control word, status word, tag word, and error pointers (both data and instruction). The environment layout in memory depends on both the operand size and the current operating mode of the microprocessor. The USE attribute of the current code segment determines the operand size: the 14-byte operand applies to a USE16 segment, and the 28-byte operand applies to a USE32 segment.

Figures 15-5 through 15-8 show the environment layouts for both operand sizes in both Real Mode and Protected Mode. (In Virtual 8086 Mode, the Real Mode layout is used.) The stack registers, beginning with ST and ending with ST(7), are in the 80 bytes that immediately follow the environment image. FRSTOR should be executed in the same operating mode as the corresponding FSAVE or FNSAVE.

FPU Flags Affected

C0, C1, C2, C3 as loaded

Numeric Exceptions

None, except for loading an unmasked exception.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** If the state image contains an unmasked exception, loading it generates a floating-point error condition.*

2.93 FSAVE Stores FPU State after Checking for Unmasked FPU Error

Opcode	Instruction	Clocks	Description
9B DD /6	FSAVE m94/108byte	154 real or virtual/ 143 protected + 3+ for FWAIT	Stores FPU environment to m94byte or m108byte after checking for unmasked floating-point error condition. Reinitializes FPU.

Operation

```
DEST ← FPU state;
initialize FPU; (* Equivalent to FNINIT *)
```

Description

FSAVE writes the current FPU state (environment and register stack) to the specified destination and then reinitializes the FPU, without checking for unmasked floating-point error conditions. The environment consists of the FPU control word, status word, tag word, and error pointers (both data and instruction). The state layout in memory depends on both the operand size and the current operating mode of the microprocessor. The USE attribute of the current code segment determines the operand size: the 94-byte operand applies to USE16 segment, and the 108-byte operand applies to a USE32 segment. The stack registers, ST(0) to ST(7), are in the 80 bytes immediately following the environment image.

FPU Flags Affected

C0, C1, C2, C3 cleared

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** FSAVE does not store the FPU state until all FPU activity is complete. The saved image reflects the state of the FPU after any previously decoded instruction is executed. If a program must read from the memory image of the state after a save instruction, it must issue an FWAIT instruction to ensure that the storage is complete. The save instructions are typically used when an operating system needs to perform a context switch, or an exception handler needs to use the FPU, or an application program wants to pass a “clean” FPU to a subroutine.*

2.94 FSCALE Scales

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 FD	FSCALE	31 (30–32)	2	Scales ST by ST(1).

Operation

ST ← ST · $2^{ST(1)}$

Description

The scale instruction interprets the value in ST(1) as an integer, and adds this integer to the exponent of ST. Thus, FSCALE provides rapid multiplication or division by integral powers of 2.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *FSCALE can be used as an inverse to FXTRACT. Since FSCALE does not pop the exponent part, however, FSCALE must be followed by FSTP ST(1) in order to completely undo the effect of a preceding FXTRACT. There is no limit on the range of the scale factor in ST(1). If the value is not integral, FSCALE uses the nearest integer smaller in magnitude (i.e., it chops the value toward 0). If the resulting integer is zero, the value in ST is not changed.*

2.95 FSIN Sine

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 FE	FSIN	241 (193–279)	2	Replaces ST with its sine.

Operation

```
IF operand is in range
THEN
   C2 ← 0;
   ST ← sin (ST);
ELSE
   C2 ← 1;
FI;
```

Description

The sine instruction replaces the contents of ST with sin (ST). ST, expressed in radians, must lie in the range $|\theta| < 2^{63}$.

FPU Flags Affected

If C2 = 0 (reduction complete), the result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); if PE is set, C1 indicates whether the last rounding was upward. If C2 = 1 (reduction incomplete), C1 is undefined. C0 and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If the operand is outside the acceptable range, the C2 flag is set and ST remains unchanged. Reduce the operand to an absolute value smaller than 2^{63} by subtracting an appropriate integer multiple of 2π. For π, use the full 66-bit internal π used by the FPU: $4 \cdot$ 0.C90FDAA22168C234Ch. This ensures that the results are consistent with the argument reduction used by the FPU for trigonometric functions.You cannot represent this number as an extended-real value, however. A suggested solution is to represent π as the sum of a highπ (the 33 most-significant bits) and a lowπ (the 33 least-significant bits). The Am486 processor can abort this instruction to service an interrupt.*

If you need to compute sine and cosine, use FSINCOS for faster execution.

2.96 FSINCOS Sine and Cosine

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 FB	FSINCOS	291 (243–329)	2	Computes the sine and cosine of ST; replaces ST with the sine, and then pushes the cosine onto the FPU stack.

Operation

```
IF operand is in range
THEN
   C2 ← 0;
   TEMP ← cos (ST);
   ST ← sin (ST);
   Decrement FPU top-of-stack pointer;
   ST ← TEMP;
ELSE
   C2 ← 1;
FI:
```

Description

FSINCOS computes both sine (ST) and cosine (ST), replaces ST with the sine, and then pushes the cosine onto the FPU stack. ST, expressed in radians, must lie in the range $| \theta | < 2^{63}$.

FPU Flags Affected

If C2 = 0 (reduction complete), the result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); if PE is set, C1 indicates whether the last rounding was upward. If C2 = 1 (reduction incomplete), C1 is undefined. C0 and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If the operand is outside the acceptable range, the C2 flag is set, and ST remains unchanged. Reduce the operand to an absolute value smaller than 2^{63} by subtracting an appropriate integer multiple of 2π. For π, use the full 66-bit internal π used by the FPU: 4 · 0.C90FDAA22168C234Ch. This ensures that the results are consistent with the argument reduction used by the FPU for trigonometric functions. You cannot represent this number as an extended-real value, however. A suggested solution is to represent π as the sum of a highπ (the 33 most-significant bits) and a lowπ (the 33 least-significant bits). The Am486 processor can abort this instruction to service an interrupt.*

2.97 FSQRT Square Root

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 FA	FSQRT	85.5 (83–87)	70	Replaces ST with its square root.

Operation

```
ST ← square root of ST;
```

Description

The square root instruction replaces the value in ST with its square root.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *The square root of –0 is –0.*

2.98 FST Stores Real

Opcode	Instruction	Clocks	Description
D9 /2	FST m32real	7	Copies ST to m32real.
DD /2	FST m64real	8	Copies ST to m64real.
DD D0+i	FST ST(i)	3	Copies ST to ST(i).

Operation

```
DEST ← ST(0)
```

Description

FST copies the current value in the ST register to the destination, which can be another register or a single or double real-memory operand.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Register destinations: Stack Fault
Single or double real destinations: Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the destination is 32 bits or 64 bits, the significand is rounded to the width of the destination according to the RC field of the control word, and the exponent is converted to the width and bias of the destination format. The over/underflow condition is checked as well. If ST contains zero, ±∞, or a NaN, then the significand is not rounded but chopped (on the right) to fit the destination. The exponent of such a value is not converted; it too is chopped on the right. These operations preserve the value's identity as∞ or NaN (exponent all ones). The invalid-operation exception is not raised when the destination is a nonempty stack element.*

2.99 FSTCW Stores Control Word after Checking for FPU Error

Opcode	Instruction	Clocks	Description
9B D9 /7	FSTCW m2byte	3 + 3+ for FWAIT	Stores FPU control word to m2byte after checking for unmasked floating-point error condition.

Operation

```
DEST ← CW
```

Description

FSTCW writes the current value of the FPU control word to the specified destination, after checking for an unmasked floating-point error condition.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.100 FSTENV Stores FPU Environment after Checking for FPU Error

Opcode	Instruction	Clocks	Description
9B D9 /6	FSTENV m14/28byte	67 real or virtual/ 56 protected + 3+ for FWAIT	Stores FPU environment to m14byte or m28byte after checking for unmasked floating-point error condition; then masks all floating-point exceptions.

Operation

```
DEST ← FPU environment;
CW[O–5] ← 111111
```

Description

FSTENV writes the current FPU environment to the specified destination, and then masks all floating-point exceptions, after checking for unmasked floating-point error conditions. The FPU environment consists of the FPU control word, status word, tag word, and error pointer (both data and instruction). The environment layout in memory depends on both the operand size and the current operating mode of the microprocessor. The USE attribute of the current code segment determines the operand size: the 14-byte operand applies to a USE16 segment, and the 28-byte operand applies to a USE32 segment.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** FSTENV does not store the FPU environment until all FPU activity is complete. The saved environment reflects the state of the FPU after any previously decoded instruction has been executed. The stored environment instructions are often used by exception handlers because they provide access to the FPU error pointers. The FPU environment is typically saved onto the memory stack. After saving the FPU environment, FSTENV sets all the exception masks in the FPU control word. This prevents floating-point errors from interrupting the exception handler.*

2.101 FSTP Stores Real and Pops the FPU Stack Top

Opcode	Instruction	Clocks	Description
D9 /3	FSTP m32real	7	Copies ST to m32real, then pops ST.
DD /3	FSTP m64real	8	Copies ST to m64real, then pops ST.
DB /7	FSTPm80real	6	Copies ST to m80real, then pops ST.
DD D8+i	FSTP ST(i)	3	Copies ST to ST(i), then pops ST.

Operation

```
DEST ← ST(0);
pop ST FI
```

Description

FSTP copies the current ST register value to the destination, which can be another register or a single-, double-, or extended-real memory operand, and then pops ST. If the source is a register, the number is used before the stack is popped.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Register or extended-real destinations: Stack Fault
Single or double real destinations: Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the destination is 32 bits or 64 bits, the significand is rounded to the width of the destination according to the RC field of the control word, and the exponent is converted to the width and bias of the destination format. The over/underflow condition is checked for as well. If ST contains zero, ±∞, or a NaN, then the significand is not rounded but chopped (on the right) to fit the destination. The exponent of such a value is not converted; it too is chopped on the right. These operations preserve the value's identity as∞ or NaN (exponent all ones). The invalid-operation exception is not raised when the destination is a nonempty stack element.*

2.102 FSTSW Stores Status Word after Checking for Unmasked FPU Error

Opcode	Instruction	Clocks	Description
9B DF /7	FSTSW m2byte	3 + 3+ for FWAIT	Stores FPU status word to m2byte after checking for unmasked floating-point error condition.
9B DF E0	FSTSW AX	3 + 3+ for FWAIT	Stores FPU status word to AX register after checking for unmasked floating-point error condition.

Operation

```
DEST ← SW
```

Description

FSTSW writes the current value of the FPU status word to the specified destination, which can be either a 2-byte location in memory or the AX register, after checking for an unmasked floating-point error condition.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** FSTSW is used primarily in conditional branching (after a comparison, FPREM, FPREM1, or FXAM instruction). It can also invoke exception handlers (by polling the exception bits) in environments that do not use interrupts.*

2.103 FSUB Subtracts Real

Opcode	Instruction	Clocks	Concurrent Execution	Description
D8 /4	FSUB m32rea;	10 (8–20)	7 (5–17)	Subtracts m32real from ST.
DC /4	FSUB m64real	10 (8–20)	7 (5–17)	Subtracts m64real from ST.
D8 E0+i	FSUB ST,ST(i)	10 (8–20)	7 (5–17)	Subtracts ST(i) from ST.
DC E8+i	FSUB ST(i),ST	10 (8–20)	7 (5–17)	Replaces ST(i) with ST–ST(i).

Operation

```
DEST ← ST – Other Operand
```

Description

The subtraction instructions subtract the other operand from the stack top and return the difference to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.104 FSUBP Subtracts Real and Pops FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE E8+i	FSUBP ST(i),ST	10 (8–20)	7 (5–17)	Replaces ST(i) with ST–ST(i); pops ST.
DE E9	FSUBP	10 (8–20)	7 (5–17)	Replaces ST(1) with ST–ST(1); pops ST.

Operation

```
DEST ← ST – Other Operand;
pop ST FI
```

Description

FSUBP subtracts the other operand from the stack top, returns the difference to the destination, and pops ST.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.105 FSUBR Reverse Subtracts Real

Opcode	Instruction	Clocks	Concurrent Execution	Description
D8 /5	FSUBR m32real	10 (8–20)	7 (5–17)	Replaces ST with m32real – ST.
DC /5	FSUBR m64real	10 (8–20)	7 (5–17)	Replaces ST with m64real – ST.
D8 E8+i	FSUBR ST,ST(i)	10 (8–20)	7 (5–17)	Replaces ST with ST(i) – ST.
DC E0+i	FSUBR ST(i),ST	10 (8–20)	7 (5–17)	Subtracts ST from ST(i).

Operation

```
DEST ← Other Operand – ST
```

Description

The reverse subtraction instructions subtract the stack top from the other operand and return the difference to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.106 FSUBRP Reverse Subtracts and Pops FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DE E0+i	FSUBRP ST(i),ST	10 (8–20)	7 (5–17)	Subtracts ST from ST(i); pops ST.
DE E1	FSUBRP	10 (8–20)	7 (5–17)	Subtracts ST from ST(1); pops ST.

Operation

```
DEST ← Other Operand – ST;
pop ST FI
```

Description

The reverse subtraction instructions subtract the stack top from the other operand and return the difference to the destination.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *If the source operand is in memory, it is automatically converted to the extended-real format.*

2.107 FTST Test

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 E4	FTST	4	1	Compares ST with 0.0.

Operation

```
CASE (relation of operands) OF
   Not comparable:         C3, C2, C0 ← 111;
   ST > SRC:               C3, C2, C0 ← 000;
   ST < SRC:               C3, C2, C0 ← 001;
   ST = SRC:               C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
FI
```

Description

FTST compares the stack top to 0.0. Following the instruction, the condition codes reflect the result of the comparison.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are set as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

***Note:** If ST contains a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised, and the condition bits are set to “unordered.”*

The sign of zero is ignored, so that –0.0 = +0.0.

2.108 FUCOM Unordered Compare Real

Opcode	Instruction	Clocks	Concurrent Execution	Description
DD E0+1	FUCOM ST(i)	4		Compares ST with ST(i).
DD E1	FUCOM	4		Compares ST with ST(1).

Operation

```
CASE (relation of operands) OF
   Not comparable:         C3, C2, C0 ← 111;
   ST > SRC:               C3, C2, C0 ← 000;
   ST < SRC:               C3, C2, C0 ← 001;
   ST = SRC:               C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
FI
```

Description

FUCOM compares the stack top to the source, which must be a register. If no operand is encoded, ST is compared to ST(1). Following the instruction, the condition codes reflect the relation between ST and the source operand.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are set as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised, and the condition bits are set to “unordered.”*

If either operand is a QNaN, the condition bits are set to “unordered.” Unlike the ordinary compare instructions (FCOM, etc.), the unordered compare instructions do not raise the invalid-operation exception if there is a QNaN operand.

The sign of zero is ignored, so that –0.0 = +0.0.

2.109 FUCOMP Unordered Compare Real and Pop FPU Stack Top

Opcode	Instruction	Clocks	Concurrent Execution	Description
DD E8+i	FUCOMP ST(i)	4	1	Compares ST with ST(i) and pops ST.
DD E9	FUCOMP	4	1	Compares ST with ST(1) and pops ST.

Operation

```
CASE (relation of operands) OF
   Not comparable:         C3, C2, C0 ← 111;
   ST > SRC:               C3, C2, C0 ← 000;
   ST < SRC:               C3, C2, C0 ← 001;
   ST = SRC:               C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
pop ST FI
```

Description

FUCOMP compares the stack top to the source, which must be a register, then pops ST. If no operand is encoded, ST is compared to ST(1). Following the instruction, the condition codes reflect the relation between ST and the source operand.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are set as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised and the condition bits are set to “unordered.”*

If either operand is a QNaN, the condition bits are set to “unordered.” Unlike the ordinary compare instructions (FCOM, etc.), the unordered compare instructions do not raise the invalid-operation exception if there is a QNaN operand.

The sign of zero is ignored, so that –0.0 = +0.0.

2.110 FUCOMPP Unordered Compare Real and Pop FPU Stack Top Twice

Opcode	Instruction	Clocks	Concurrent Execution	Description
DA E9	FUCOMPP	5	1	Compares ST with ST(1) and pops ST twice.

Operation

```
CASE (relation of operands) OF
   Not comparable:         C3, C2, C0 ← 111;
   ST > SRC:               C3, C2, C0 ← 000;
   ST < SRC:               C3, C2, C0 ← 001;
   ST = SRC:               C3, C2, C0 ← 100;
CF ← C0;
PF ← C2;
ZF ← C3;
pop ST; pop ST; FI
```

Description

FUCOMPP compares the stack top to ST(1) and pops ST twice. Following the instruction, the condition codes reflect the relation between ST and the source operand.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are set as specified above.

Numeric Exceptions

Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If either operand is a NaN or is in an undefined format, or if a stack fault occurs, the invalid-operation exception is raised and the condition bits are set to “unordered.”*

If either operand is a QNaN, the condition bits are set to “unordered.” Unlike the ordinary compare instructions (FCOM, etc.), the unordered compare instructions do not raise the invalid-operation exception if there is a QNaN operand.

The sign of zero is ignored, so that –0.0 = +0.0.

2.111 FWAIT Wait

Opcode	Instruction	Clocks	Description
9B	FWAIT	(1–3)	Alias for WAIT.

Description

FWAIT causes the microprocessor to check for pending unmasked numeric exceptions before proceeding.

FPU Flags Affected

C0, C1, C2, C3 undefined

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if both MP and TS in CR0 are set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if both MP and TS in CR0 are set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if both MP and TS in CR0 are set.

***Note:** As its opcode shows, FWAIT is not actually an ESC instruction but an alternate mnemonic for WAIT. Coding FWAIT after an ESC instruction ensures that any unmasked floating-point exceptions caused by the instruction are handled before the processor modifies the instruction's results.*

2.112 FXAM Examine

Opcode	Instruction	Clocks	Description
D9 E5	FXAM	8	Reports the type of object in the ST register.

Operation

```
Cl ← sign bit of ST; (* 0 for positive, 1 for negative *)

CASE (type of object in ST) OF
   Unsupported:   C3, C2, C0 ← 000;
   NaN:           C3, C2, C0 ← 001;
   Normal:        C3, C2, C0 ← 010;
   Infinity:      C3, C2, C0 ← 011;
   Zero:          C3, C2, C0 ← 100;
   Empty:         C3, C2, C0 ← 101;
   Denormal:      C3, C2, C0 ← 110;
CF ← C0;
PF ← C2;
ZF ← C3;
FI
```

Description

The examine instruction reports the type of object contained in the ST register by setting the FPU Flags.

FPU Flags Affected

C0, C1, C2, C3 are set as shown above.

Numeric Exceptions

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

2.113 FXCH Exchanges Stack Register Contents

Opcode	Instruction	Clocks	Description
D9 C8+i	FXCH ST(i)	4	Exchanges the contents of ST and ST(i).
D9 C9	FXCH	4	Exchanges the contents of ST and ST(1).

Operation

```
TEMP ← ST;
ST ← DEST;
DEST ← TEMP
```

Description

FXCH swaps the contents of the destination and stack top registers. If the destination is not coded explicitly, ST(1) is used.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *Many numeric instructions operate only on the stack top; FXCH provides a simple means for using these instructions on lower stack elements. For example, the following sequence takes the square root of the third register from the top (assuming that ST is not empty):*

```
FXCH ST(3)
FSQRT
FXCH ST(3)
```

2.114 FXTRACT Extracts Exponent and Significand

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F4	FXTRACT	19 (16–20)	4 (2–4)	Separates ST into its exponent and significand; replaces ST with the exponent and then pushes the significand onto the FPU stack.

Operation

```
TEMP ← significand of ST;
ST ← exponent of ST;
Decrement FPU top-of-stack pointer;
ST ← TEMP
```

Description

FXTRACT splits the value in ST into its exponent and significand. The exponent replaces the original operand on the stack and the significand is pushed onto the stack. ST (the new stack top) contains the value of the significand as a real number with the same sign, a 0 true (16,383 or 3FFFh biased) exponent, and identical significand as the original operand. ST(1) contains the original operand's true (unbiased) exponent expressed as a real number.

FPU Flags Affected

The result determines the C1 setting. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0); otherwise, C1 = 0. C0, C2, and C3 are undefined.

Numeric Exceptions

Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

***Note:** FXTRACT (extract exponent and significand) performs a superset of the IEEE recommended logb(x) function. It is useful for power and range scaling operations. Both FXTRACT and F2XM1 are needed to perform a general power operation. You must use FXTRACT with FBSTP when converting extend-format real numbers to decimal representations to allow scaling that does not overflow the extended format range. FXTRACT is also useful for debugging because it allows separate examination of a real number's exponent and significand.*

If the original operand is zero, FXTRACT leaves $-\infty$ in ST(1) (the exponent), assigns a zero value with the same sign as the original operand to ST, and generates a zero divide exception. ST(7) must be empty to avoid the invalid-operation exception.

2.115 FYL2X Computes y · $\log_2 x$

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F1	FYL2X	311 (196–329)	13	Replaces ST(1) with ST(1) · $\log_2$ST and pops ST.

Operation

```
ST(l) ← ST(l) · log₂ST;
pop ST
```

Description

FYL2X computes the base-2 logarithm of ST, multiplies the logarithm by ST(1), and returns the resulting value to ST(1). It then pops ST. The operand in ST cannot be negative.

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Overflow, Divide By Zero, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *The Am486 processor can abort this instruction to service an interrupt.*

If the operand in ST is negative, the invalid-operation exception is raised. FYL2X has built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base:

$$\log_b x = (\log_2 b)^{-1} \cdot \log_2 x$$

The instructions FLDL2T and FLDL2E load the constants $\log_2 10$ and $\log_2 e$, respectively.

2.116 FYL2XP1 Computes y · $\log_2$(x+1)

Opcode	Instruction	Clocks	Concurrent Execution	Description
D9 F9	FYL2XP1	313 (171–326)	13	Replaces ST(1) with ST(1) · $\log_2$(ST+1.0) and pops ST.

Operation

```
ST(1) ← ST(1) · log₂(ST + 1.0);
pop ST
```

Description

FYL2XP1 computes the base-2 logarithm of (ST + 1.0), multiplies the logarithm by ST(1), and returns the resulting value to ST(1). It then pops ST. The operand in ST must be in the range:

$-(1-(\sqrt{2}/2)) \leq ST \leq \sqrt{2}-1$

FPU Flags Affected

The result determines the C1 setting. If the PE bit of the status word is set, C1 represents whether the last rounding in the instruction was upward or not. If both the IE and SF bits of the status word are set (indicating a stack exception), C0 distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0). C0, C2, and C3 are undefined.

Numeric Exceptions

Precision (Inexact Result), Underflow, Denormalized Operand, Invalid Operation, Stack Fault

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if either EM or TS in CR0 is set.

Note: *If the operand in ST is outside the acceptable range, the result of FYL2XP1 is undefined.*

The FYL2XP1 instruction provides improved accuracy over FYL2X when computing the logarithms of numbers very close to 1. When ε is small, more significant digits can be retained by providing ε as an argument to FYL2XP1 than by providing $1+\varepsilon$ as an argument to FYL2X.

The Am486 processor can abort this instruction to service an interrupt.

2.117 HLT Halt

Opcode	Instruction	Clocks	Description
F4	HLT	4	Halt

Operation

```
Enter Halt state
```

Description

The HLT instruction stops instruction execution and places the microprocessor in a HALT state. An enabled interrupt, an NMI, or a reset resumes execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved CS:IP (or CS:EIP) value points to the instruction following the HLT instruction.

Flags Affected

None

Protected Mode Exceptions

The HLT instruction is a privileged instruction; General Protection Fault (13) indicates the current privilege level is not 0.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

General Protection Fault (13); the HLT instruction is a privileged instruction.

2.118 IDIV Signed Divide

Opcode	Instruction	Clocks	Description
F6 /7	IDIV r/m8	19/20	Performs a signed divide AX by r/m byte (AL = Quo, AH = Rem).
F7 /7	IDIV AX,r/m16	27/28	Performs a signed divide DX:AX by r/m word (AX = Quy, DX = Rem).
F7 /7	IDIV EAX,r/m32	43/44	Performs a signed divide EDX:EAX by r/m doubleword (EAX = Quo, EDX = Rem).

Operation

```
temp ← dividend / divisor;
IF temp does not fit in quotient
THEN Divide By Zero Exception 0;
ELSE
   quotient ← temp;
   remainder ← dividend MOD (r/m);
FI
```

Note: *Divisions are signed.*

Description

IDIV performs a signed division. The dividend, quotient, and remainder are implicitly allocated to fixed registers. The divisor is an explicit r/m operand. The divisor type determines which registers to use as follows:

Size	Divisor	Quotient	Remainder	Dividend
byte	r/m8	AL	AH	AX
word	r/m16	AX	DX	DX:AX
doubleword	r/m32	EAX	EDX	EDX:EAX

Non-integral quotients are truncated toward 0. The remainder has the same sign as the dividend and the remainder absolute value is always less than the divisor absolute value.

Flags Affected

OF, SF, ZF, AF, PF, CF are undefined.

Protected Mode Exceptions

Divide By Zero (0) indicates a quotient too large for the designated register (AL or AX), or a divisor of 0. General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. For CPL = 3, Alignment Check (17) indicates an unaligned memory reference.

Real Address Mode Exceptions

Divide By Zero (0) indicates a quotient too large for the designated register (AL or AX), or a divisor of 0. General Protection Fault (13) indicates that part of the operand is outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

Divide By Zero (0) indicates a quotient too large for the designated register (AL or AX), or a divisor of 0. General Protection Fault (13) indicates that part of the operand is outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.119 IMUL Signed Multiply

Opcode	Instruction	Clocks	Description
F6 /5	IMUL r/8	13–18	AX ← AL · r/m byte
F7 /5	IMUL r/16	13–26	DX:AX ← AX · r/m word
F7 /5	IMUL r/m32	13–42	EDS:EAX ← EAX · r/m doubleword
0F AF /r	IMUL r16,r/m16	13–26	word reg ← word reg · r/m word
0F AF /r	IMUL r32,r/m32	13–42	doubleword reg ← doubleword reg · r/m doubleword
6B /r ib	IMUL r16,r/m16,imm8	13–26	word reg ← r/m16 · sign-extended immediate byte
6B /r ib	IMUL r32,r/m32,imm8	13–42	doubleword reg ← r/m32 · sign-extended immediate byte
6B /r ib	IMUL r16,imm8	13–26	word reg ← word reg · sign-ext. immediate byte
6B /r ib	IMUL r32,imm8	13–42	doubleword reg ← doubleword reg · sign-ext. immediate byte
69 /r iw	IMUL r16,r/m16,imm16	13–26	word reg ← r/m16 · immediate word
69 /r id	IMUL r32,r/m32,imm32	13–42	doubleword reg ← r/m32 · immediate doubleword
69 /r iw	IMUL r16,imm16	13–26	word reg ← r/m16 · immediate word
69 /r id	IMUL r32,imm32	13–42	doubleword reg ← r/m32 · immediate doubleword

Actual clock count depends on the most-significant bit location in the optimizing multiplier. If the multipler (*m*) is 0, the clock count is 9; otherwise clock = max (ceiling($\log_2 |m|$), 3) + 6. If *m* is a memory operand, add 3.

Operation

```
result ← multiplicand · multiplier
```

Description

IMUL performs signed multiplication. Some forms of the instruction use implicit register operands.

Flags Affected

SF, ZF, AF, and PF are undefined. IMUL clears CF and OF under certain conditions. If you use the accumulator form (IMUL r/m8, IMUL r/m16, or IMUL r/32), IMUL clears the flags if the result equals the sign-extended value of the source register (AL, AX, or EAX respectively). For IMUL r16,r/m16; IMUL r/32,r/m32; IMUL r16,r/m16,imm16; or IMUL r32,r/m32, imm32; IMUL clears the flags if the result fits exactly in the destination register.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** For the accumulator forms (IMUL r/m8, IMUL r/m16, or IMUL r/m32), the result of the multiplication is available even if the Overflow Flag is set because the result is twice the size of the multiplicand and multiplier. This is large enough to handle any possible result.*

2.120 IN Inputs Data from Port

Opcode	Instruction	Clocks	Description
E4 ib	IN AL,imm8	*All forms:*	Inputs byte from immediate port into AL.
E5 ib	IN AX,imm8	rm = 14, vm = 27	Inputs word from immediate port into AX.
E5 ib	IN EAX,imm8	If CPL ≤ IOPL,	Inputs doubleword from immediate port into EAX.
EC	IN AL,DX	pm = 8	Inputs byte from port DX into AL.
ED	IN AX,DX	If CPL>IOPL,	Inputs word from port DX into AX.
ED	IN EAX,DX	pm = 28	Inputs doubleword from port DX into EAX.

Operation

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 Mode, or Protected Mode with CPL > IOPL *)
   IF NOT I/O-Permission (SRC, width (SRC))
   THEN General Protection Fault (13);
   FI;
FI;
DEST ← [SRC]; (* Reads from I/O address space *)
```

Description

The IN instruction transfers a data byte, word, or doubleword from the port numbered by the second operand into the register (AL, AX, or EAX) specified by the first operand. Access any port from 0 to 65535 by placing the port number in the DX register and using an IN instruction with the DX register as the second parameter. These I/O instructions can be shortened by using an 8-bit port I/O in the instruction. The upper eight bits of the port address will be 0 when 8-bit port I/O is used.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the current privilege level is larger (has less privilege) than the I/O privilege level and any of the corresponding I/O permission bits in TSS equals 1.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that at least one of the corresponding I/O permission bits in TSS equals 1.

2.121 INC Increments by One

Opcode	Instruction	Clocks	Description
FE /0	INC r/m8	1/3	Increments r/m byte by 1.
FF /0	INC r/m16	1/3	Increments r/m word by 1.
FF /6	INC r/m32	1/3	Increments r/m doubleword by 1.
40 + rw	INC r16	1	Increments word register by 1.
40 + rd	INC r32	1	Increments doubleword register by 1.

Operation

```
DEST ← DEST + 1
```

Description

The INC instruction adds 1 to the operand. It does not change CF. To affect CF, use the ADD instruction with a second operand of 1.

Flags Affected

OF, SF, ZF, AF, and PF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.122 INS/INSB/INSD/INSW Inputs Data from Port to String

Opcode	Instruction	Clocks	Description
6C	INS r/m8,DX	*All forms:*	Inputs byte from port DX into ES:DI.
6D	INS r/m16,DX	If CPL ≤ IOPL,	Inputs word from port DX into ES:DI.
6D	INS r/m32,DX	17, pm = 10	Inputs doubleword from port DX into ES:EDI.
6C	INSB	If CPL>IOPL,	Inputs byte from port DX into ES:DI.
6D	INSD	32, vm = 30	Inputs doubleword from port DX into ES:EDI.
6D	INSW		Inputs word from port DX into ES:DI.

Operation

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 Mode, or Protected Mode with CPL > IOPL *)
   IF NOT I/O-Permission (SRC, width(SRC))
   THEN General Protection Fault (13);
   FI;
FI;
IF OperandSize = 8 (* byte *)
THEN
   ES:DI ← [DX]; (* Reads byte at DX from I/O address space *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← –1; FI;
IF OperandSize = 16 (* word *)
THEN
   ES:DI ← [DX]; (* Reads word at DX from I/O address space *)
   IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← –2; FI;
IF OperandSize = 32 (* doubleword *)
THEN
   ES:EDI ← [DX]; (* Reads doubleword at DX from I/O address space *)
   IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← –4; FI;
FI;
source-index = source-index + IncDec;
destination-index = destination-index + IncDec
```

Description

INS transfers data from the input port numbered by the DX register to the memory byte, word, or doubleword at ES:(E)DI. The memory operand must be addressable from the ES register; no segment override is possible. The destination register is the DI register if the address-size attribute of the instruction is 16 bits, or the EDI register if the address-size attribute is 32 bits.

The INS instruction does not allow the specification of the port number as an immediate value. You must address the port through the DX register value. Similarly, the destination index register determines the destination address. You must preload the DX register value into the DX register and the correct index into the destination index register before executing the INS instruction.

After the transfer is made, the DI or EDI register advances automatically. If DF is 0 (a CLD instruction was executed), the DI or EDI register increments; if DF is 1 (an STD instruction was executed), the DI or EDI register decrements. The DI register increments or decrements by 1 if the input is a byte, by 2 if it is a word, or by 4 if it is a doubleword.

The INSB, INSW, and INSD instructions are synonyms of the byte, word, and doubleword INS instructions. INS instructions can use the REP prefix for block input of CX bytes or words. Refer to the REP instruction for details of this operation.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the current privilege level is numerically greater than the I/O privilege level and any of the corresponding I/O permission bits in TSS equals 1. General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates either that one of the corresponding I/O permission bits in TSS equals 1, or that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.123 INT/INTO Call to Interrupt Procedure

Opcode	Instruction	Clocks	Description
CC	INT 3	26	Interrupt 3 — trap to debugger
CC	INT 3	44	Interrupt 3 — Protected Mode, same privilege
CC	INT 3	71	Interrupt 3 — Protected Mode, more privilege
CC	INT 3	82	Interrupt 3 — from V86 Mode to PL 0
CC	INT 3	37 + ts*	Interrupt 3 — Protected Mode via task gate
CD ib	INT imm8	30	Interrupt numbered by immediate byte
CD ib	INT imm8	44	Interrupt — Protected Mode, same privilege
CD ib	INT imm8	71	Interrupt — Protected Mode, more privilege
CD ib	INT imm8	86	Interrupt — from V86 Mode to PL 0
CD ib	INT imm8	37 + ts*	Interrupt — Protected Mode, via task gate
CE	INTO	Pass = 28; Fail = 3	Interrupt 4 — if Overflow Flag is 1
CE	INTO	46	Interrupt 4 — Protected Mode, same privilege
CE	INTO	73	Interrupt 4 — Protected Mode, more privilege
CE	INTO	84	Interrupt 4 — from V86 Mode to PL 0
CE	INTO	39 + ts*	Interrupt 4 — Protected Mode, via task gate

*ts = 199 for 486 TSS, 180 for 286 TSS, or 177 for VM TSS

Operation

Note: *The following operational description applies not only to the above instructions but also to external interrupts and exceptions.*

```
IF PE = 0
THEN GOTO REAL-ADDRESS-MODE;
ELSE GOTO PROTECTED-MODE;
FI;

REAL-ADDRESS-MODE:
   Push (FLAGS);
   IF ← 0; (* C1ear Interrupt Flag *)
   TF ← 0; (* Clear Trap Flag *)
   Push(CS);
   Push(IP);
   (* No error codes are pushed *)
   CS ← IDT[interrupt number · 4].selector;
   IP ← IDT[Interrupt number · 4].offset;

(* Start execution in Real Address Mode *)
PROTECTED-MODE:
   Interrupt vector must be within IDT table limits,
      else General Protection Fault(vector number · 8 + 2 + EXT);
   Descriptor AR byte must indicate interrupt gate, trap gate, or task gate,
      else General Protection Fault(vector number · 8 + 2 + EXT);
   IF software interrupt (* i.e. caused by INT n, INT 3, or INTO *)
   THEN
      IF gate descriptor DPL < CPL
      THEN General Protection Fault(vector number · 8 + 2 + EXT);
      FI;
   FI;
   Gate must be present,
      ELSE Segment Not Present(vector number · 8 + 2 + EXT);
   IF trap gate OR interrupt gate
   THEN GOTO TRAP-GATE-OR-INTERRUPT-GATE;
   ELSE GOTO TASK-GATE;
   FI;
```

```
TRAP-GATE-OR-INTERRUPT-GATE:
   Examine CS selector and descriptor given in the gate descriptor;
   Selector must be non-null, else General Protection Fault(EXT);
   Selector must be within its descriptor table limits
      ELSE General Protection Fault(selector + EXT);
   Descriptor AR byte must indicate code segment
      ELSE General Protection Fault(selector + EXT);
   Segment must be present, else Segment Not Present (11)(selector + EXT);
   IF code segment is non-conforming AND DPL < CPL
   THEN GOTO INTERRUPT-TO-INNER-PRIVILEGE;
   ELSE
      IF code segment is conforming OR code segment DPL = CPL
      THEN GOTO INTERRUPT-TO-SAME-PRIVILEGE-LEVEL;
      ELSE General Protection Fault(CS selector + EXT);
      FI;
   FI;

INTERRUPT-TO-INNER-PRIVILEGE:
   Check selector and descriptor for new stack in current TSS;
      Selector must be non-null, ELSE Invalid TSS(EXT);
      Selector index must be within its descriptor table limits
         ELSE Invalid TSS(SS selector+ EXT);
      Selector’s RPL must equal DPL of code segment,
         ELSE Invalid TSS(SS selector+ EXT);
      Stack segment DPL must equal DPL of code segment,
         ELSE Invalid TSS(SS selector+ EXT);
      Descriptor must indicate writable data segment,
         ELSE Invalid TSS(SS selector + EXT);
      Segment must be present, else Stack Fault(SS selector+ EXT);
   IF 32-bit gate
   THEN New stack must have room for 20 bytes else Stack Fault
   ELSE New stack must have room for 10 bytes else Stack Fault
   FI;
   Instruction pointer must be within CS segment boundaries
      ELSE General Protection Fault;
   If VM = 1 in EFLAGS
   Then Goto INTERRUPT from V-86-MODE;
   Load new SS and eSP value from TSS;
   IF 32-bit gate
   THEN CS:EIP ← selector:offset from gate;
   ELSE CS:IP ← selector:offset from gate;
   FI;
   Load CS descriptor into invisible portion of CS register;
   Load SS descriptor into invisible portion of SS register;
   IF 32-bit gate
   THEN
      Push (long pointer to old stack) (* 3 words padded to 4 *);
      Push (EFLAGS);
      Push (long pointer to return location) (* 3 words padded to 4 *);
   ELSE
      Push (long pointer to old stack) (* 2 words *);
      Push (FLAGS);
      Push (long pointer to return location) (* 2 words *);
   FI;
   Set CPL to new code segment DPL;
   Set RPL of CS to CPL;
   IF interrupt gate THEN IF 0 (* Interrupt Flag to 0 (disabled) *); FI;
   TF ← 0;
   NT ← 0;

INTERRUPT-FROM-V86-MODE:
   TempEFlags ← EFLAGS;
   VM ← 0;
   TF ← 0;
   IF service through Interrupt Gate THEN IF ← 0;
   TempSS ← SS;
   TempESP ← ESP;
   SS ← TSS. SSO; (* Change to level 0 stack segment *)
   ESP ← TSS. ESPO; (* Change to level 0 stack pointer *)
   Push(GS); (* padded to two words *)
   Push(FS); (* padded to two words *)
   Push(DS); (* padded to two words *)
   Push(ES); (* padded to two words *)
   GS ;ID 0;
   FS ← 0;
   DS ← 0;
   ES ← 0;
   Push(TempSS); (* padded to two words *)
   Push(TempESP);
   Push(TempEFlags);
   Push(CS); (* padded to two words *)
   Push(EIP); CS:EIP <- selector:offset from interrupt gate;
   (* Starts execution of new routine in Protected Mode *)

INTERRUPT-TO-SAME-PRIVILEGE-LEVEL:
   IF 32-bit gate
   THEN Current stack limits must allow pushing 10 bytes, else Stack Fault
(12);
   ELSE Current stack limits must allow pushing 6 bytes, else Stack Fault
(12);
   FI;
   IF interrupt was caused by exception with error code
   THEN Stack limits must allow push of two more bytes;
   ELSE Stack Fault (12);
   FI;
   Instruction pointer must be in CS limit, else General Protection Fault
(13) (0);
   IF 32-bit gate
   THEN
      Push (EFLAGS);
      Push (long pointer to return location); (* 3 words padded to 4 *)
      CS: EIP ← selector:offset from gate;
   ELSE (* 16-bit gate *)
      Push (FLAGS);
      Push (long pointer to return location); (* 2 words *)
      CS:IP ← selector:offset from gate;
   FI;
   Load CS descriptor into invisible portion of CS register;
   Set the RPL field of CS to CPL;
   Push (error code); (* if any *)
   IF interrupt gate THEN IF ← 0; FI;
   TF ← 0;
   NT ← 0;

TASK-GATE:
   Examine selector to TSS, given in task gate descriptor;
```

```
      Must specify global in the local/global bit, else Invalid TSS (10)(TSS selector);
      Index must be within GDT limits, else Invalid TSS (10)(TSS selector);
      AR byte must specify available TSS (bottom bits 00001),
         else Invalid TSS (10)(TSS selector);
      TSS must be present, else Segment Not Present (11)(TSS selector);
   SWITCH-TASKS with nesting to TSS;
   IF interrupt was caused by fault with error code
   THEN
      Stack limits must allow push of two more bytes, else Stack Fault (12);
      Push error code onto stack;
   FI;
   Instruction pointer must be in CS limit, else General Protection Fault (13)
```

Description

The INT n instruction generates a call to an interrupt handler via software. The immediate operand, from 0 to 255, gives the index number into the Interrupt Descriptor Table (IDT) of the called interrupt routine. In Protected Mode, the IDT consists of an array of 8-byte descriptors; the invoked interrupt descriptor must indicate an interrupt, trap, or task pointer. In Real Address Mode, the IDT is an array of 4-byte pointers. In Protected and Real Address Modes, the base linear address of the IDT is defined by the contents of the IDTR.

The INTO conditional software instruction is identical to the INT n interrupt instruction except that the interrupt number is implicitly 4, and the interrupt is made only if the Am486 microprocessor Overflow Flag is set.

The first 32 interrupts are reserved for system use. Some of these interrupts are used for internally generated exceptions.

The INT n instruction generally behaves like a far call except that the contents of the FLAGS register are pushed onto the stack before the return address. Interrupt procedures return via the IRET instruction, which pops the flags and return address from the stack.

In Real Address Mode, the INT n instruction pushes the flags, the CS register, and the return IP onto the stack, in that order, then jumps to the long pointer indexed by the interrupt number.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13), Segment Not Present (11), Stack Fault (12), and Invalid TSS (10) can occur as indicated under ‘Operation’ above.

Real Address Mode Exceptions

None. However, if when INT or INTO starts executing, the SP or ESP register is 1, 3, or 5, the processor shuts down due to insufficient stack space.

Virtual 8086 Mode Exceptions

General Protection Fault (13) occurs if IOPL is less than 3, for the INT n instruction only, as part of the mode emulation; Interrupt 3 (0CCh) generates a breakpoint exception; the INTO instruction generates an overflow exception if OF is set.

2.124 INVD Invalidates Cache

Opcode	Instruction	Clocks	Description
0F 08	INVD	4	Invalidates entire cache.

Operation

```
FLUSH INTERNAL CACHE
SIGNAL EXTERNAL CACHE TO FLUSH
```

Description

The processor flushes the internal cache and issues a special-function bus cycle that indicates that the external cache should be flushed. Any data held in write-back external cache is discarded.

Flags Affected

None

Protected Mode Exceptions

The INVD instruction is a privileged instruction. General Protection Fault (13) indicates the current privilege level is not 0.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

General Protection Fault (13); the INVD instruction is a privileged instruction.

Note: *This instruction is implementation-dependent; its function may be implemented differently on future AMD microprocessors. It is the responsibility of the designer to ensure that the hardware responds to the external cache flush indication.*

This instruction is not supported by Am386® microprocessors.

2.125 INVLPG Invalidates TLB Entry

Opcode	Instruction	Clocks	Description
0F 01/7	INVLPG m	12 for hit	Invalidates TLB entry.

Operation

```
INVALIDATE TLB ENTRY
```

Description

INVLPG invalidates a single entry in the TLB (the cache used for page table entries). If the TLB contains a valid entry that maps the address of the memory operand, that TLB entry is marked invalid.

Flags Affected

None

Protected Mode Exceptions

INVLPG is a privileged instruction; General Protection Fault (13) indicates the current privilege level is not 0. An invalid-opcode exception is generated when used with a register operand.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

An invalid-opcode exception is generated when used with a register operand. General Protection Fault (13); the INVLPG instruction is a privileged instruction.

Note: *This instruction is not supported on Am386 microprocessors.*

2.126 IRET/IRETD Interrupt Return

Opcode	Instruction	Clocks	Description
CF	IRET	15	Interrupt return (far return and pop FLAGS)
CF	IRET	36	Interrupt return to lesser privilege
CF	IRET	32 + ts*	Interrupt return, different task (NT = 1)
CF	IRETD	15	Interrupt return, (far return and pop FLAGS)
CF	IRETD	36	Interrupt return to lesser privilege
CF	IRETD	15	Interrupt return to V86 Mode
CF	IRETD	32 + ts*	Interrupt return, different task (NT = 1)

*ts = 199 for 486 TSS, 180 for 286 TSS, or 177 for VM TSS

Operation

```
IF PE = 0
THEN (* Real Address Mode *)
   IF OperandSize = 32 (* Instruction = IRETD *)
   THEN EIP ← Pop();
   ELSE (* Instruction = IRET *)
      IP ← Pop();
   FI;
   CS ← Pop();
   IF OperandSize = 32 (* Instruction = IRETD *)
   THEN Pop(); EFLAGS ← Pop();
   ELSE (* Instruction = IRET *)
      FLAGS ← Pop();
   FI;
ELSE (* Protected Mode)
   IF VM = 1
   THEN General Protection Fault (13);
   ELSE
      IF NT = 1
      THEN GOTO TASK-RETURN;
      ELSE
         IF VM = 1 in FLAGS image on stack
         THEN GO TO STACK-RETURN-TO-V86;
         ELSE GOTO STACK-RETURN;
   FI;FI;FI;
FI;STACK-RETURN-TO-V86:(* Interrupted procedure was in Virtual 8086 Mode *)
   IF top 36 bytes of stack not within limits
   THEN Stack Fault (12);
   FI;
   IF instruction pointer not within code segment limit
      THEN General Protection Fault (13);
   FI;
EFLAGS ← SS:[ESP + 8]; (* Sets VM in interrupted routine *)
EIP ← Pop();
CS ← Pop(); (* CS behaves as in 8086, due to VM = 1 *)
throwaway ← Pop(); (* pop away EFLAGS already read *)
TempESP ← Pop();
TempSS ← Pop();
ES ← Pop(); (* pop 2 words; throw away high-order word *)
DS ← Pop(); (* pop 2 words; throw away high-order word *)
FS ← Pop(); (* pop 2 words; throw away high-order word *)
GS ← Pop(); (* pop 2 words; throw away high-order word *)
   SS:ESP ← TempSS:TempESP;

(* Resume execution in Virtual 8086 Mode *)
```

```
TASK-RETURN:
   Examine Back Link Selector in TSS addressed by the current task
      register:
      Must specify global in the local/global bit,
         ELSE Invalid TSS(new TSS selector);
      Index must be within GDT limits, else Invalid TSS(new TSS selector;
      AR byte must specify TSS, else Invalid TSS(new TSS selector);
      New TSS must be busy, else Invalid TSS(new TSS selector);
      TSS must be present, else Segment Not Present(new TSS selector);
   SWITCH-TASKS without nesting to TSS specified by back link selector;
   Mark the task just abandoned as NOT BUSY;
   Instruction pointer must be within code segment limit
      ELSE General Protection Fault);

STACK-RETURN:
   IF OperandSize = 32
   THEN Third word on stack must be within stack limits, else Stack Fault;
   ELSE Second word on stack must be within stack limits, else Stack Fault
   FI;
   Return CS selector RPL must be ≥CPL,
      ELSE General Protection Fault(Return selector);
   IF return selector RPL = CPL
   THEN GOTO RETURN-SAME-LEVEL;
   ELSE GOTO RETURN-OUTER-LEVEL;
   FI;

RETURN-SAME-LEVEL:
   IF OperandSize = 32
   THEN
      Top 12 bytes on stack must be within limits, else Stack Fault;
      Return CS selector (at eSP+ 4) must be non-null,
         ELSE General Protection Fault;
   ELSE
      Top 6 bytes on stack must be within limits, else Stack Fault;
      Return CS selector (at eS P + 2) must be non-null,
         ELSE General Protection Fault; FI;
   Selector index must be within its descriptor table limits,
         ELSE General Protection Fault
      Return selector; AR byte must indicate code segment,
         ELSE General Protection Fault(Return selector);
   IF non-conforming THEN code segment DPL must = CPL;
   ELSE General Protection Fault(Return selector); FI;
   IF conforming
   THEN code segment DPL must be ≤ CPL,
      ELSE General Protection Fault(Return selector);
   Segment must be present, else Segment Not Present (11)(Return selector);
   Instruction pointer must be within code segment boundaries,
      ELSE General Protection Fault; FI;
   IF OperandSize = 32
   THEN
      Load CS: EIP from stack;
      Load CS-register with new code segment descriptor;
      Load EFLAGS with third doubleword from stack;
      Increment eSP by 12;
   ELSE
      Load CS-register with new code segment descriptor;
      Load FLAGS with third word on stack;
      Increment eSP by 6; FI;
```

```
RETURN-OUTER-LEVEL:
   IF OperandSize = 32
   THEN Top 20 bytes on stack must be ithin limits, else Stack Fault;
   ELSE Top 10 bytes on stack must be within limits, else Stack Fault;
   FI;
   Examine return CS selector and associated descriptor:
      Selector must be non-null, else General Protection Fault;
      Selector index must be within its descriptor table limits;
         ELSE General Protection Fault(Return selector);
      AR byte must indicate code segment,
         ELSE General Protection Fault (Return selector);
      IF non-conforming
      THEN code segment DPL must = CS selector RPL;
      ELSE General Protection Fault(Return selector); FI;
      IF conforming
      THEN code segment DPL must be > CPL;
      ELSE General Protection Fault(Return selector); FI;
      Segment must be present,
         ELSE Segment Not Present(Return selector);
Examine return SS selector and associated descriptor:
   Selector must be non-null, ELSE General Protection Fault;
   Selector index must be within its descriptor table limits
      ELSE General Protection Fault(SS selector);
   Selector RPL must equal the RPL of the return CS selector
      ELSE General Protection Fault(SS selector);
   AR byte must indicate a writable data segment,
      ELSE General Protection Fault(SS selector);
   Stack segment DPL must equal the RPL of the return CS selector
      ELSE General Protection Fault(SS selector);
   SS must be present, else Segment Not Present(SS selector);
Instruction pointer must be within code segment limit
   ELSE General Protection Fault;
IF OperandSize = 32
THEN
   Load CS:EIP from stack;
   Load EFLAGS with values at (eSP + 8);
ELSE
   Load CS:IP from stack;
   Load FLAGS with values at (eSP + 4) ;
FI;
Load SS:eSP from stack;
Set CPL to the RPL of the return CS selector;
Load the CS register with the CS descriptor;
Load the SS register with the SS descriptor;
FOR each of ES, FS, GS, and DS
DO;
   IF the current value of the register is not valid for the outer level;
   THEN zero the register and clear the valid flag;
   FI;
   To be valid, the register setting must satisfy the following properties:
      Selector index must be within descriptor table limits;
      AR byte must indicate data or readable code segment;
      IF segment is data or non-conforming code,
      THEN DPL must be > CPL, or DPL must be < RPL;
OD
```

Description

In Real Address Mode, the IRET instruction pops the instruction pointer, the CS register, and the FLAGS register from the stack and resumes the interrupted routine.

In Protected Mode, the action of the IRET instruction depends on the setting of the Nested Task flag (NT) bit in the EFLAGS register. When the new flag image is popped from the stack, the IOPL bits in the EFLAGS register are changed only when CPL equals 0.

If the NT flag is cleared, the IRET instruction returns from an interrupt procedure without a task switch. The code returned to must be equally or less privileged than the interrupt routine (as indicated by the RPL bits of the CS selector popped from the stack). If the destination code is less privileged, the IRET instruction also pops the stack pointer and SS from the stack.

If the NT flag is set, the IRET instruction reverses the operation of a CALL or INT that caused a task switch. The updated state of the task executing the IRET instruction is saved in its task state segment. If the task is re-entered later, the code that follows the IRET instruction is executed.

Flags Affected

All flags are affected; the FLAGS or EFLAGS register is popped from stack.

Protected Mode Exceptions

General Protection Fault (13), Segment Not Present (11), or Stack Fault (12), occurs as indicated under ‘Operation’ above.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand being popped lies beyond address 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates the I/O privilege level is less than 3, as part of the emulation.

2.127 JA Jumps If Above (see also JNBE)

Opcode	Instruction	Clocks	Description
77 cb	JA rel8	3 (true),1 (false)	Jumps short if above (CF = 0 and ZF = 0).
0F 87 cw/cd	JA rel16/32	3 (true), 1 (false)	Jumps near if above (CF = 0 and ZF = 0).

Operation

```
IF CF = 0 AND ZF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFH;
   FI;
FI
```

Description

JA tests the flag set by a previous instruction. ‘Above’ indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.128 JAE Jumps If Above or Equal (see also JNB and JNC)

Opcode	Instruction	Clocks	Description
73 cb	JAE rel8	3 (true), 1 (false)	Jumps short if above or equal (CF = 0).
0F 83 cw/cd	JAE rel16/32	3 (true), 1 (false)	Jumps near if above or equal (CF = 0).

Operation

```
IF CF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JAE tests the flag set by a previous instruction. ‘Above’ indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.129 JB Jumps If Below (see also JC and JNAE)

Opcode	Instruction	Clocks	Description
72 cb	JB rel8	3 (true), 1 (false)	Jumps short if below (CF = 1).
0F 82 cw/cd	JB rel16/32	3 (true), 1 (false)	Jumps near if below (CF = 1).

Operation

```
IF CF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JB tests the flag set by a previous instruction. ‘Below’ indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.130 JBE Jumps If Below or Equal (see also JNA)

Opcode	Instruction	Clocks	Description
76 cb	JBE rel8	3 (true), 1 (false)	Jumps short if below or equal (CF = 1 or ZF = 1).
0F 86 cw/cd	JBE rel16/32	3 (true), 1 (false)	Jumps near if below or equal (CF = 1 or ZF = 1).

Operation

```
IF CF = 1 OR ZF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JBE tests the flag set by a previous instruction. ‘Below’ indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.131 JC Jumps If Carry (see also JB and JNAE)

Opcode	Instruction	Clocks	Description
72 cb	JC rel8	3 (true), 1 (false)	Jumps short if carry (CF = 1).
0F 86 cw/cd	JC rel16/32	3 (true), 1 (false)	Jumps near if carry (CF = 1).

Operation

```
IF CF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JC tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.132 JCXZ Jumps Short If CX Register is 0 (see also JECXZ)

Opcode	Instruction	Clocks	Description
E3 cb	JCXZ rel8	8 (true), 5 (false)	Jumps short if CX register is 0.

Operation

```
IF CX = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JCXZ tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

***Note:** JCXZ takes longer to execute than a two-instruction sequence that compares the count register to zero and jumps if the count is zero.*

The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.

2.133 JE Jumps Short If Equal (see also JZ)

Opcode	Instruction	Clocks	Description
74 cb	JE rel8	3 (true), 1 (false)	Jumps short if equal (ZF = 1).
0F 84 cw/cd	JE rel16/32	3 (true), 1 (false)	Jumps near if equal (ZF = 1).

Operation

```
IF ZF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JE tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.134 JECXZ Jumps Short If ECX Register is 0 (see also JCXZ)

Opcode	Instruction	Clocks	Description
E3 cb	JECXZ rel8	8 (true), 5 (false)	Jumps short if ECX register is 0.

Operation

```
IF ECX = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JECXZ tests the flag set by a previous instruction. ‘Above’ indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *JECXZ takes longer to execute than a two-instruction sequence that compares the count register to zero and jumps if the count is zero.*

The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.

2.135 JG Jumps If Greater (see also JNLE)

Opcode	Instruction	Clocks	Description
7F cb	JG rel8	3 (true), 1 (false)	Jumps short if greater (ZF = 0 and SF = OF).
0F 84 cw/cd	JG rel16/32	3 (true), 1 (false)	Jumps near if greater (ZF = 0 and SF = OF).

Operation

```
IF ZF = 0 AND SF = CF
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JG tests the flag set by a previous instruction. ‘Greater’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.136 JGE Jumps If Greater or Equal (see also JNL)

Opcode	Instruction	Clocks	Description
7D cb	JGE rel8	3 (true), 1 (false)	Jumps short if greater or equal (SF = OF).
0F 8D cw/cd	JGE rel16/32	3 (true), 1 (false)	Jumps near if greater or equal (SF = OF).

Operation

```
IF SF = OF
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JGE tests the flag set by a previous instruction. ‘Greater’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.137 JL Jumps If Less (see also JNGE)

Opcode	Instruction	Clocks	Description
7C cd	JL rel8	3 (true), 1 (false)	Jumps short if less (SF ≠ OF).
0F 8C cw/cd	JL rel16/32	3 (true), 1 (false)	Jumps near if less (SF ≠ OF).

Operation

```
IF SF ≠ OF
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JL tests the flag set by a previous instruction. ‘Less’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

***Note:** The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.138 JLE Jumps If Less or Equal (see also JNG)

Opcode	Instruction	Clocks	Description
7E cb	JLE rel8	3 (true), 1 (false)	Jumps short if less or equal (ZF = 1 and SF ≠ OF).
0F 8E cw/cd	JLE rel16/32	3 (true), 1 (false)	Jumps near if less or equal (ZF = 1 and SF ≠ OF).

Operation

```
IF ZF = 1 AND SF≠OF
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JLE tests the flag set by a previous instruction. ‘Less’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.139 JMP Jump

Opcode	Instruction	Clocks	Description
EB cb	JMP rel8	3	Jumps short.
E9 cw	JMP rel16	3	Jumps near, displacement relative to next instruction.
FF /4	JMP r/m16	5/5	Jumps near indirect.
EA cd	JMP ptr16:16	17,pm = 19	Jumps far to 4-byte intermediate address.
EA cd	JMP ptr 16:16	32	Jumps to call gate, same privilege.
EA cd	JMP ptr 16:16	42 + ts*	Jumps via task state segment.
EA cd	JMP ptr 16:16	43 + ts*	Jumps via task gate.
FF /5	JMP m16:16	13,pm = 18	Jumps r/m16:16 indirect and far.
FF /5	JMP m16:16	31	Jumps to call gate, same privilege.
FF /5	JMP m16:16	41 + ts*	Jumps via task state segment.
FF /5	JMP m16:16	42 + ts*	Jumps via task gate.
E9 cd	JMP rel32	3	Jumps near with displacement relative to next instruction.
FF /4	JMP r/m32	5/5	Jumps near, indirect.
EA cp	JMP ptr16:32	13,pm = 18	Jumps far to 6-byte immediate address.
EA cp	JMP ptr16:32	31	Jumps to call gate, same privilege.
EA cp	JMP ptr16:32	42 + ts*	Jumps via task state segment.
EA cp	JMP ptr16:32	43 + ts*	Jumps via task gate.
FF /5	JMP m16:32	13,pm = 18	Jumps far to address in r/m doubleword.
FF /5	JMP m16:32	31	Jumps to call gate, same privilege.
FF /5	JMP m16:32}	41 + ts*	Jumps via task state segment.
FF /5	JMP m16:32	42 + ts*	Jumps via task gate.

*ts = 199 for 486 TSS, 180 for 286 TSS, or 177 for VM TSS

Operation

```
IF instruction = relative JMP
   (* i.e. operand is rel8, rel16, or rel32 *)
THEN
   EIP ← EIP + rel8/16/32,
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI;

IF instruction = near indirect JMP
   (* i.e. operand is r/m16 or r/m32 *)
THEN
   IF OperandSize = 16
   THEN
      EIP ← [r/m16] AND OOO0FFFFh;
   ELSE (* OperandSize = 32 *)
      EIP ← [r/m32];
   FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1)) (* Real Mode or Virtual 8086 Mode *)
   AND instruction = far JMP
   (* i.e., operand type is m 16:16, m 16:32, ptr16:16, ptr16:32 *)
THEN GOTO REAL-OR-V86-MODE;
   IF operand type = m16:16 or m16:32
   THEN (* indirect *)
      IF OperandSize = 16
      THEN
         CS:IP ← [m16:16];
         EIP ← EIP AND 0000FFFFh; (* clear upper 16 bits *)
      ELSE (* OperandSize = 32 *)
         CS:EIP ← [m16:32];
      FI;
   FI;
   IF operand type = ptr16:16 or ptr16:32
   THEN
      IF OperandSize = 16
      THEN
         CS:IP ← ptr16:16,
         EIP ← EIP AND 000FFFFh; (* dear upper 16 bits *)
      ELSE (* OperandSize = 32 *)
         CS:EIP ← ptr16:32;
      FI;
   FI;
FI;

IF (PE = 1 AND VM = 0) (* Protected Mode, not Virtual 8086 Mode *)
   AND instruction = far JMP
THEN
   IF operand type = m16 or m16:32
   THEN (* indirect *)
      check access of EA doubleword;
      General Protection Fault or Stack Fault IF limit violation;
   FI;
   Destination selector is not null ELSE General Protection Fault
   Destination selector index is within its descriptor table limits ELSE
General Protection Fault(selector)
   Depending on AR byte of destination descriptor:
      GOTO C0NFORMING-CODE-SEGMENT;
      GOTO NONCONFORMING-CODE-SEGMENT;
      GOTO CALL-GATE;
      GOTO TASK-GATE;
      GOTO TASK-STATE-SEGMENT;
   ELSE General Protection Fault(selector); (* illegal AR in descriptor *)
FI;

CONFORMING-CODE-SEGMENT:
   Descriptor DPL must be ≤ CPL ELSE General Protection Fault(selector);
   Segment must be present ELSE Segment Not Present(selector);
   Instruction pointer must be within code-segment limit
      ELSE General Protection Fault;
   IF OperandSize = 32
   THEN Load CS:EIP from destination pointer;
   ELSE Load CS:IP from destination pointer;
   FI;
   Load CS register with new segment descriptor;

NONCONFORMING-CODE-SEGMENT:
   RPL of destination selector must be ≤ CPL
      ELSE General Protection Fault (selector);
   Descriptor DPL must = CPL ELSE General Protection Fault(selector);
   Segment must be present ELSE Segment Not Present (11)(selector);
   Instruction pointer must be within code-segment limit
      ELSE General Protection Fault;
   IF OperandSize = 32 THEN Load CS:EIP from destination pointer;
   ELSE Load CS:IP from destination pointer;
   FI;
   Load CS register with new segment descriptor;
   Set RPL field of CS register to CPL;
```

```
CALL-GATE:
   Descriptor DPL must be ≥ CPL
      ELSE General Protection Fault(gate selector);
   Descriptor DPL must be ≥ gate selector RPL
      ELSE General Protection Fault(gate selector);
   Gate must be present ELSE Segment Not Present(gate selector);
   Examine selector to code segment given in call gate descriptor:
      Selector must not be null ELSE General Protection Fault;
      Selector must be within its descriptor table limit
         ELSE General Protection Fault(CS selector);
      Descriptor AR byte must indicate code segment
         ELSE General Protection Fault(CS selector);
      IF non-conforming
      THEN code-segment descriptor DPL must = CPL
      ELSE General Protection Fault(CS selector); FI;
      IF conforming
      THEN code-segment descriptor DPL must be ≤ CPL;
      ELSE General Protection Fault (13)(CS selector;
      Code segment must be present ELSE Segment Not Present(CS selector);
      Instruction pointer must be within code-segment limit
         ELSE General Protection Fault;
      IF OperandSize = 32
      THEN Load CS:EIP from call gate;
      ELSE Load CS:IP from call gate; FI;
   Load CS register with new code-segment descriptor;
   Set RPL of CS to CPL

TASK-GATE:
   Gate descriptor DPL must be ≥ CPL
      ELSE General Protection Fault(gate selector);
   Gate descriptor DPL must be ≥ gate selector RPL
      ELSE General Protection Fault(gate selector);
   Task Gate must be present ELSE Segment Not Present(gate selector);
   Examine selector to TSS, given in Task Gate descriptor:
      Must specify global in the local/global bit
         ELSE General Protection Fault(TSS selector);
      Index must be within GDT limits
         ELSE General Protection Fault(TSS selector);
      Descriptor AR byte must specify available TSS (bottom bits 00001);
         ELSE General Protection Fault(TSS selector);
      Task State Segment must be present
         ELSE Segment Not Present(TSS selector);
   SWITCH-TASKS (without nesting) to TSS;
   Instruction pointer must be within code-segment limit
      ELSE General Protection Fault;

TASK-STATE-SEGMENT:
   TSS DPL must be ≥ CPL ELSE General Protection Fault(TSS selector);
   TSS DPL must be ≥ TSS selector RPL
      ELSE General Protection Fault(TSS selector);
   Descriptor AR byte must specify available TSS (bottom bits 00001)
      ELSE General Protection Fault(TSS selector);
   Task State Segment must be present
      ELSE Segment Not Present(TSS selector);
   SWITCH-TASKS (without nesting) to TSS;
   Instruction pointer must be within code-segment limit
      ELSE General Protection Fault
```

Description

JMP transfers control to a different point in the instruction stream without recording return information. The instruction has several different forms, as follows:

- *Near Direct Jumps:* The JMP r/m16 and JMP r/m32 forms specify a register or memory location from which the procedure absolute offset is fetched. The offset is 32 bits for r/m32, or 16 bits for r/m16.
- *Near Indirect Jumps:* To determine the destination, the JMP rel16 and JMP rel32 forms add an offset to the address of the instruction following the JMP. The rel16 form is used for 16-bit operand-size attributes (segment-size attribute 16 only); rel32 is used for 32-bit operand-size attributes (segment-size attribute 32 only). The result is stored in the 32-bit EIP register. With rel16, the upper 16 bits of the EIP register are cleared, which results in an offset that does not exceed 16 bits.
- *Far Jumps:* The JMP ptr16:16 and ptr16:32 forms use a 4-byte or 6-byte operand as a long pointer to the destination. The JMP m16:16 and m16:32 forms fetch the long pointer from the specified memory location (indirection). In Real or Virtual 8086 Mode, the long pointer provides 16 bits for the CS register and 16 or 32 bits for the EIP register (depending on operand-size). In Protected Mode, both forms consult the Access Rights (AR) byte in the descriptor indexed by the selector part of the long pointer. Depending on the value of the AR byte, the jump performs one of the following control transfer types:

  — A jump to a code segment at the same privilege level

  — A task switch

Flags Affected

All if a task switch occurs; none if no task switch occurs.

Protected Mode Exceptions

Near direct jumps: General Protection Fault (13) indicates the procedure is outside the code segment limits. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Near indirect jumps: General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. General Protection Fault (13) indicates the indirect offset is beyond the code segment limits. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Far jumps: General Protection Fault (13), Segment Not Present (11), Stack Fault (12), and Invalid TSS (10), as listed in the ‘Operations’ section starting on page 2-152.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand is outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

***Note:** All branches are converted into 16-byte code fetches regardless of jump address or cacheability.*

2.140 JNA Jumps If Not Above (see also JBE)

Opcode	Instruction	Clocks	Description
76 cb	JNA rel8	3 (true), 1 (false)	Jumps short if not above (CF = 1 or ZF = 1).
0F 86 cw/cd	JNA rel16/32	3 (true), 1 (false)	Jumps near if not above (CF = 1 or ZF = 1).

Operation

```
IF CF = 1 OR ZF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNA tests the flag set by a previous instruction. “Above” indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.141 JNAE Jumps If Not Above or Equal (see also JB and JC)

Opcode	Instruction	Clocks	Description
72 cb	JNAE rel8	3 (true), 1 (false)	Jumps short if not above or equal (CF = 1).
0F 82 cw/cd	JNAE rel16/32	3 (true), 1 (false)	Jumps near if not above or equal (CF = 1).

Operation

```
IF CF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNAE tests the flag set by a previous instruction. “Above” indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.142 JNB Jumps If Not Below (see also JAE and JNC)

Opcode	Instruction	Clocks	Description
73 cb	JNB rel8	3 (true), 1 (false)	Jumps short if not below (CF = 0).
0F 83 cw/cd	JNB rel16/32	3 (true), 1 (false)	Jumps near if not below (CF = 0).

Operation

```
IF CF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNB tests the flag set by a previous instruction. “Below” indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.143 JNBE Jumps If Not Below or Equal (see also JA)

Opcode	Instruction	Clocks	Description
77 cb	JNBE rel8	3 (true), 1 (false)	Jumps short if not below or equal (CF = 0 and ZF = 0).
0F 87 cw/cd	JNBE rel16/32	3 (true), 1 (false)	Jumps near if not below or equal (CF = 0 and ZF = 0).

Operation

```
IF CF = 0 AND ZF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNBE tests the flag set by a previous instruction. ‘Below’ indicates an unsigned integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

***Note:** The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.144 JNC Jumps If Not Carry (see also JAE and JNB)

Opcode	Instruction	Clocks	Description
73 cb	JNC rel8	3 (true), 1 (false)	Jumps short if not carry (CF = 0).
0F 83 cw/cd	JNC rel16/32	3 (true), 1 (false)	Jumps near if not carry (CF = 0).

Operation

```
IF CF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNC tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.145 JNE Jumps If Not Equal (see also JNZ)

Opcode	Instruction	Clocks	Description
75 cb	JNE rel8	3 (true), 1 (false)	Jumps short if not equal (ZF = 0).
0F 85 cw/cd	JNE rel16/32	3 (true), 1 (false)	Jumps near if not equal (ZF = 0).

Operation

```
IF ZF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNE tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.146 JNG Jumps If Not Greater (see also JLE)

Opcode	Instruction	Clocks	Description
7E cb	JNG rel8	3 (true), 1 (false)	Jumps short if not greater (ZF = 1 and SF ≠ OF).
0F 8E cw/cd	JNG rel16/32	3 (true), 1 (false)	Jumps near if not greater (ZF = 1 and SF ≠ OF).

Operation

```
IF ZF = 1 AND SF ≠ OF
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNG tests the flag set by a previous instruction. ‘Greater’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.147 JNGE Jumps If Not Greater or Equal (see also JL)

Opcode	Instruction	Clocks	Description
7C cb	JNGE rel8	3 (true), 1 (false)	Jumps short if not greater or equal (SF ≠ OF).
0F 8C cw/cd	JNGE rel16/32	3 (true), 1 (false)	Jumps near if not greater or equal (SF ≠ OF).

Operation

```
IF SF ≠ OF
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNGE tests the flag set by a previous instruction. ‘Greater’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.148 JNL Jumps If Not Less (see also JGE)

Opcode	Instruction	Clocks	Description
7D cb	JNL rel8	3 (true), 1 (false)	Jumps short if not less (SF = OF).
0F 8D cw/cd	JNL rel16/32	3 (true), 1 (false)	Jumps near if not less (SF = OF).

Operation

```
IF SF = OF
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNL tests the flag set by a previous instruction. ‘Less’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.149 JNLE Jumps If Not Less or Equal (see also JG)

Opcode	Instruction	Clocks	Description
7F cb	JNLE rel8	3 (true), 1 (false)	Jumps short if not less or equal (ZF = 0 and SF = 0).
0F 8F cw/cd	JNLE rel16/32	3 (true), 1 (false)	Jumps near if not less or equal (ZF = 0 and SF = 0).

Operation

```
IF ZF = 0 AND SF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNLE tests the flag set by a previous instruction. ‘Less’ indicates a signed integer comparison. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.150 JNO Jumps If Not Overflow

Opcode	Instruction	Clocks	Description
71 cb	JNO rel8	3 (true), 1 (false)	Jumps short if not overflow (OF = 0).
0F 81 cw/cd	JNO rel16/32	3 (true), 1 (false)	Jumps near if not overflow (OF = 0).

Operation

```
IF OF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNO tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.151 JNP Jumps If Not Parity (see also JPO)

Opcode	Instruction	Clocks	Description
7B cb	JNP rel8	3 (true), 1 (false)	Jumps short if not parity (PF = 0).
0F 8B cw/cd	JNP rel16/32	3 (true), 1 (false)	Jumps near if not parity (PF = 0).

Operation

```
IF PF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNP tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.152 JNS Jumps If Not Sign

Opcode	Instruction	Clocks	Description
79 cb	JNS rel8	3 (true), 1 (false)	Jumps short if not sign (SF = 0).
0F 89 cw/cd	JNS rel16/32	3 (true), 1 (false)	Jumps near if not sign (SF = 0).

Operation

```
IF SF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNS tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.153 JNZ Jumps If Not Zero (see also JNE)

Opcode	Instruction	Clocks	Description
75 cb	JNZ rel8	3 (true), 1 (false)	Jumps short if not zero (ZF = 0).
0F 85 cw/cd	JNZ rel16/32	3 (true), 1 (false)	Jumps near if not zero (ZF = 0).

Operation

```
IF ZF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JNZ tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.154 JO Jumps If Overflow

Opcode	Instruction	Clocks	Description
70 cb	JO rel8	3 (true), 1 (false)	Jumps short if overflow (OF = 1).
0F 80 cw/cd	JO rel16/32	3 (true), 1 (false)	Jumps near if overflow (OF = 1).

Operation

```
IF OF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JO tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.155 JP Jumps If Parity (see also JPE)

Opcode	Instruction	Clocks	Description
7A cb	JP rel8	3 (true), 1 (false)	Jumps short if parity (PF = 1).
0F 8A cw/cd	JP rel16/32	3 (true), 1 (false)	Jumps near if parity (PF = 1).

Operation

```
IF PF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JP tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.156 JPE Jumps If Parity Even (see also JP)

Opcode	Instruction	Clocks	Description
7A cb	JPE rel8	3 (true), 1 (false)	Jumps short if parity even (PF = 1).
0F 8A cw/cd	JPE rel16/32	3 (true), 1 (false)	Jumps near if parity even (PF = 1).

Operation

```
IF PF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JPE tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.157 JPO Jumps if Parity Odd (see also JNP)

Opcode	Instruction	Clocks	Description
7B cb	JPO rel8	3 (true), 1 (false)	Jumps short if parity odd (PF = 0).
0F 8B cw/cd	JPO rel16/32	3 (true), 1 (false)	Jumps near if parity odd (PF = 0).

Operation

```
IF PF = 0
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JPO tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.158 JS Jumps If Sign

Opcode	Instruction	Clocks	Description
78 cb	JS rel8	3 (true), 1 (false)	Jumps short if sign (SF = 1).
0F 88 cw/cd	JS rel16/32	3 (true), 1 (false)	Jumps near if sign (SF = 1).

Operation

```
IF SF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JS tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.159 JZ Jumps If 0 (see also JE)

Opcode	Instruction	Clocks	Description
74 cb	JZ rel8	3 (true), 1 (false)	Jumps short if 0 (ZF = 1).
0F 84 cw/cd	JZ rel16/32	3 (true), 1 (false)	Jumps near if 0 (ZF = 1).

Operation

```
IF ZF = 1
THEN
   EIP ← EIP + SignExtend(rel8/16/32)
   IF OperandSize = 16
   THEN EIP ← EIP AND 0000FFFFh;
   FI;
FI
```

Description

JZ tests the flag set by a previous instruction. If the given condition is true, a jump is made to the location provided as the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the limits of the code segment.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The instruction converts all branches into 16-byte code fetches regardless of jump address or cacheability.*

2.160 LAHF Loads Flags into AH

Opcode	Instruction	Clocks	Description
9F	LAHF	3	Loads the FLAGS register into AH.

Operation

```
AH ← SF:ZF:xx:AF:xx:PF:xx:CF
```

Description

The LAHF instruction transfers the FLAGS register (low byte of the EFLAGS register) to the AH register. After the transfer, the bits shadow the flags as follows:

- AH bit 0 = Carry Flag
- AH bit 2 = Parity Flag
- AH bit 4 = Auxiliary Flag
- AH bit 6 = Zero Flag
- AH bit 7 = Sign Flag

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.161 LAR Loads Access Rights Byte

Opcode	Instruction	Clocks	Description
0F 02 /r	LAR r16,r/m16	11/11	r16 ← r/m16 masked by FF00h
0F 02 /r	LAR r32,r/m32	11/11	r32 ← r/m32 masked by 00FxFF00h

Description

If the source selector is visible at the current privilege level (modified by the selector's RPL) and is a valid descriptor type within the descriptor limits, LAR stores the high-order doubleword of the descriptor masked by 00FxFF00 in the destination register, and sets ZF. The x indicates that the four bits corresponding to the upper four bits of the limit are undefined in the value loaded by the LAR instruction. If the selector is invisible or of the wrong type, LAR clears ZF. If the 32-bit operand size is specified, the entire 32-bit value is loaded into the 32-bit destination register. If the 16-bit operand size is specified, the lower 16 bits of this value are stored in the 16-bit destination register. All code and data segment descriptors are valid for LAR.

The valid special segment and gate descriptor types for the LAR instruction are given in the following table:

Type	Name	Valid/Invalid
0	Invalid	Invalid
1	Available 80286 TSS	Valid
2	LDT	Valid
3	Busy 80286 TSS	Valid
4	80286 call gate	Valid
5	80286/486 task gate	Valid
6	80286 trap gate	Valid
7	80286 interrupt gate	Valid
8	Invalid	Invalid
9	Available 486 TSS	Valid
A	Invalid	Invalid
B	Busy 486 TSS	Valid
C	486 call gate	Valid
D	Invalid	Invalid
E	486 trap gate	Valid
F	486 interrupt gate	Valid

Flags Affected

If the selector is invisible or of the wrong type, LAR clears ZF; otherwise, it sets ZF.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Invalid Opcode (6) occurs. LAR is unrecognized in Real Address Mode.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) occurs. LAR is unrecognized in Virtual 8086 Mode.

2.162 LDS Loads Pointer Using DS

Opcode	Instruction	Clocks	Description
C5 /r	LDS r16,m16:16	6/12	Loads DS:r16 with pointer from memory.
C5 /r	LDS r32,m16:32	6/12	Loads DS:r32 with pointer from memory.

Operation

```
IF (OperandSize = 16)
THEN
   r16 ← [Effective Address]; (* 16-bit transfer *)
   DS ← [Effective Address + 2]; (* 16-bit transfer *)
ELSE (* OperandSize = 32 *)
   r32 ← [Effective Address]; (* 32-bit transfer *)
   DS ← [Effective Address + 4]; (* 16-bit transfer *) FI;
IF Protected Mode and DS is loaded with a non-null selector:
   Index is within limits  ELSE General Protection Fault(selector);
   AR byte indicates data segment ELSE General Protection Fault(selector);
   IF data or non-conforming code
   THEN RPL and CPL are ≤ DPL in AR byte ELSE Gen.Protect.Fault(selector);
   Segment must be marked present ELSE Segment Not Present(selector);
Load segment register with selector and RPL bits;
Load segment register with descriptor;
IF Protected Mode and DS is loaded with a null selector:
   Load segment register with selector; Clear descriptor valid bit
```

Description

LDS reads a full pointer from memory and stores it in a register pair consisting of the DS register and a second operand-specified register. The first 16 bits are in DS and the remaining 16 or 32 bits (as specifed by the operand size) are placed into the register specified by the r16 or r32 register operand. The segment register descriptor comes from the selector descriptor table entry. Loading a null selector (values 0000–0003) into DS does not cause a protection exception, but any subsequent reference to a segment with a null selector causes a General Protection Fault (13) and no memory reference to the segment occurs.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Invalid Opcode (6) indicates the second operand is a register. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.163 LEA Loads Effective Address

Opcode	Instruction	Clocks	Description
8D /r	LEA r16,m[16-bit]	1	Stores effective address for m in 16-bit register.
8D /r	LEA r32,m[16-bit]	1	Stores effective address for m in 32-bit register.
8D /r	LEA r16,m[32-bit]	1	Stores effective address for m in 16-bit register.
8D /r	LEA r32,m[32-bit]	1	Stores effective address for m in 32-bit register.

Operation

```
IF OperandSize = 16 AND AddressSize = 16
   THEN r16 Addr(m);
IF OperandSize = 16 AND AddressSize = 32
   THEN r16 ← Truncate_to_16bits(Addr(m)); (* 32-bit address *)
IF OperandSize = 32 AND AddressSize = 16
   THEN  r32 ← Truncate_to_16bits(Addr(m));
IF OperandSize = 32 AND AddressSize = 32
   THEN r32 ← Addr(m);
FI
```

Description

LEA calculates the effective address (offset part) and stores it in the specified register. The operand-size attribute of the instruction (represented by OperandSize in ‘Operation’ above) is determined by the chosen register. The address-size attribute (represented by AddressSize) is determined by the USE attribute of the segment containing the second operand. The address-size and operand-size attributes affect the action performed by the LEA instruction, as follows:

- *16-bit operand, 16-bit address:* LEA calculates the effective 16-bit address and stores it in the 16-bit destination register.
- *16-bit operand, 32-bit address:* LEA calculates the effective 32-bit address and stores the lower 16 bits in the 16-bit destination register.
- *32-bit operand, 16-bit address:* LEA calculates the effective 16-bit address, zero extends it, and stores it in the 32-bit destination register.
- *32-bit operand, 32-bit address:* LEA calculates the effective 32-bit address and stores it in the 32-bit destination register.

Flags Affected

None

Protected Mode Exceptions

Invalid Opcode (6) indicates the second operand is a register.

Real Address Mode Exceptions

Invalid Opcode (6) indicates the second operand is a register.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) indicates the second operand is a register.

2.164 LEAVE High Level Procedure Exit

Opcode	Instruction	Clocks	Description
C9	LEAVE	5	Sets SP to BP, then pops BP.
C9	LEAVE	5	Sets ESP to EBP, then pops EBP.

Operation

```
IF StackAddrSize = 16
THEN
   SP ← BP;
   pop BP;
ELSE (* StackAddrSize = 32 *)
   ESP ← EBP;
   pop EBP;
FI
```

Description

The LEAVE instruction reverses the actions of the ENTER instruction. By copying the frame pointer to the stack pointer, the LEAVE instruction releases the stack space used by a procedure for its local variables. The old frame pointer is popped into the BP or EBP register, restoring the caller’s frame. A subsequent RET *nn* instruction removes any arguments pushed onto the stack of the exiting procedure.

Flags Affected

None

Protected Mode Exceptions

Stack Fault (12) indicates the BP or EBP register does not point to a location within the limits of the current stack segment.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

2.165 LES Loads Pointer Using ES

Opcode	Instruction	Clocks	Description
C4 /r	LES r16,m16:16	6/12	Loads ES:r16 with pointer from memory.
C4 /r	LES r32,m16:32	6/12	Loads ES:r32 with pointer from memory.

Operation

```
IF (OperandSize = 16)
THEN
   r16 ← [Effective Address]; (* 16-bit transfer *)
   ES ← [Effective Address + 2]; (* 16-bit transfer *)
ELSE (* OperandSize = 32 *)
   r32 ← [Effective Address]; (* 32-bit transfer *)
   ES ← [Effective Address + 4]; (* 16-bit transfer *)
FI;
IF Protected Mode and ES is loaded with a non-null selector:
   Index is within limits ELSE General Protection Fault(selector);
   AR byte indicates data segment ELSE General Protection Fault(selector);
   IF data or non-conforming code
   THEN RPL and CPL are ≤ DPL in AR byte ELSE Gen.Protect.Fault(selector);
   Segment is marked present ELSE Segment Not Present Fault(selector);
Load segment register with selector and RPL bits;
Load segment register with descriptor;
IF Protected Mode and ES is loaded with a null selector:
   Load segment register with selector; Clear descriptor valid bit
```

Description

LES reads a full pointer from memory and stores it in a register pair consisting of the ES register and a second operand-specified register. The first 16 bits are in ES and the remaining 16 or 32 bits (as specifed by the operand size) are placed into the register specified by the r16 or r32 register operand. The segment register descriptor comes from the selector descriptor table entry. Loading a null selector (values 0000–0003) into ES does not cause a protection exception, but any subsequent reference to a segment with a null selector causes a General Protection Fault (13) and no memory reference to the segment occurs.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Invalid Opcode (6) indicates the second operand is a register. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.166 LFS Loads Pointer Using FS

Opcode	Instruction	Clocks	Description
04 B4 /r	LFS r16,m16:16	6/12	Loads FS:r16 with pointer from memory.
04 B4 /r	LFS r32,m16:32	6/12	Loads FS:r32 with pointer from memory.

Operation

```
IF (OperandSize = 16)
THEN
   r16 ← [Effective Address]; (* 16-bit transfer *)
   FS ← [Effective Address + 2]; (* 16-bit transfer *)
ELSE (* OperandSize = 32 *)
   r32 ← [Effective Address]; (* 32-bit transfer *)
   FS ← [Effective Address + 4]; (* 16-bit transfer *)
FI;
IF Protected Mode and FS is loaded with a non-null selector:
   Index is within limits ELSE General Protection Fault(selector);
   AR byte indicates data segment ELSE Gen.Protect. Fault(selector);
   IF data or non-conforming code
   THEN RPL and CPL are ≤ DPL in AR byte ELSEGen.Protect.Fault(selector);
   Segment is marked present ELSE Segment Not Present(selector);
Load segment register with selector and RPL bits;
Load segment register with descriptor;
IF Protected Mode and FS is loaded with a null selector:
   Load segment register with selector; Clear descriptor valid bit;
```

Description

LFS reads a full pointer from memory and stores it in a register pair consisting of the FS register and a second operand-specified register. The first 16 bits are in FS and the remaining 16 or 32 bits (as specifed by the operand size) are placed into the register specified by the r16 or r32 register operand. The segment register descriptor comes from the selector descriptor table entry. Loading a null selector (values 0000–0003) into FS does not cause a protection exception, but any subsequent reference to a segment with a null selector causes a General Protection Fault (13) and no memory reference to the segment occurs.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Invalid Opcode (6) indicates the second operand is a register. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.167 LGDT Loads GDTR

Opcode	Instruction	Clocks	Description
0F 01 /2	LGDT m16&32	11	Loads m into GDTR.

Operation

```
IF OperandSize = 16
THEN GDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
ELSE GDTR.Limit:Base ← m16:32,
FI;
```

Description

LGDT loads a linear base address and limit value from a 6-byte data operand in memory into the GDTR. If a 16-bit operand is used with the LGDT instruction, the register is loaded with a 16-bit limit and a 24-bit base, and the high-order 8 bits of the 6-byte data operand are not used. If a 32-bit operand is used, a 16-bit limit and a 32-bit base are loaded; the high-order 8 bits of the 6-byte operand are used as high-order base address bits.

The SGDT instruction always stores into all 48 bits of the 6-byte data operand. With the 80286 microprocessor, the upper 8 bits are undefined after SGDT executes. With the Am386DX or Am486 microprocessors, the upper 8 bits are written with the high-order 8 address bits, for both a 16-bit operand and a 32-bit operand. If the LGDT instruction is used with a 16-bit operand to load the register stored by the SGDT instruction, the upper 8 bits are stored as zeros.

The LGDT instruction appears in operating system software. It is not used in application programs. LGDT and LIDT are the only instructions that load a linear address directly (i.e., not a segment relative address) in Protected Mode.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates one of three conditions: the current privilege level is not 0, the result destination is a non-writable segment, or the code or data segments have an illegal memory-operand effective address. Invalid Opcode (6) indicates the source operand is a register. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Invalid Opcode (6) indicates the source operand is a register.

***Note:** This instruction is valid in Real Address Mode to allow power-up initialization for Protected Mode.*

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Invalid Opcode (6) indicates the source operand is a register. Page Fault (14) indicates a page fault.

2.168 LGS Loads Pointer Using GS

Opcode	Instruction	Clocks	Description
0F B5 /r	LGS r16,m16:16	6/12	Loads GS:r16 with pointer from memory.
0F B5 /r	LGS r32,m16:32	6/12	Loads GS:r32 with pointer from memory.

Operation

```
IF (OperandSize = 16)
THEN
   r16 ← [Effective Address]; (* 16-bit transfer *)
   GS ← [Effective Address + 2]; (* 16-bit transfer *)
ELSE (* OperandSize = 32 *)
   r32 ← [Effective Address]; (* 32-bit transfer *)
   GS ← [Effective Address + 4]; (* 16-bit transfer *)
FI;
IF Protected Mode and GS is loaded with a non-null selector:
   Index must be within limits ELSE General Protection Fault(selector);
   AR byte indicates data segment ELSE Gen.Protect. Fault(selector);
   IF data or non-conforming code
   THEN RPL and CPL are ≤ DPL in AR byte ELSEGen.Protect.Fault(selector);
   Segment must be marked present ELSE Segment Not Present(selector);
Load segment register with selector and RPL bits;
Load segment register with descriptor;
IF Protected Mode and GS is loaded with a null selector:
   Load segment register with selector; Clear descriptor valid bit;
```

Description

LGS reads a full pointer from memory and stores it in a register pair consisting of the GS register and a second operand-specified register. The first 16 bits are in GS and the remaining 16 or 32 bits (as specifed by the operand size) are placed into the register specified by the r16 or r32 register operand. The segment register descriptor comes from the selector descriptor table entry. Loading a null selector (values 0000–0003) into GS does not cause a protection exception, but any subsequent reference to a segment with a null selector causes a General Protection Fault (13) and no memory reference to the segment occurs.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Invalid Opcode (6) indicates the second operand is a register. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.169 LIDT Loads IDTR

Opcode	Instruction	Clocks	Description
0F 01 /3	LIDT m16&32	11	Loads m into IDTR.

Operation

```
IF OperandSize = 16
THEN IDTR.Limit:Base ← m16:24 (* 24 bits of base loaded *)
ELSE IDTR.Limit:Base ← m16:32
FI;
```

Description

The LIDT instruction loads a linear base address and limit value from a 6-byte data operand in memory into the IDTR. If a 16-bit operand is used with the LIDT instruction, the register is loaded with a 16-bit limit and a 24-bit base, and the high-order 8 bits of the 6-byte data operand are not used. If a 32-bit operand is used, a 16-bit limit and a 32-bit base are loaded; the high-order 8 bits of the 6-byte operand are used as high-order base address bits.

The SIDT instruction always stores into all 48 bits of the 6-byte data operand. With the 80286 microprocessor, the upper 8 bits are undefined after SIDT executes. With the Am386DX or Am486 microprocessors, the upper 8 bits are written with the high-order 8 address bits, for both a 16-bit operand and a 32-bit operand. If the LIDT instruction is used with a 16-bit operand to load the register stored by the SIDT instruction, the upper 8 bits are stored as zeros.

The LIDT instruction appears in operating system software. It is not used in application programs. LGDT and LIDT are the only instructions that directly load a linear address (i.e., not a segment relative address) in Protected Mode.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates one of three conditions: the current privilege level is not 0, the result destination is a non-writable segment, or the code or data segments have an illegal memory-operand effective address. Invalid Opcode (6) indicates the source operand is a register. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Invalid Opcode (6) indicates the source operand is a register.

Note: *This instruction is valid in Real Address Mode to allow power-up initialization for Protected Mode.*

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Invalid Opcode (6) indicates the source operand is a register. Page Fault (14) indicates a page fault.

2.170 LLDT Loads LDTR

Opcode	Instruction	Clocks	Description
0F 00 /2	LLDT r/m16	11/11	Loads selector r/m16 into LDTR.

Operation

```
LDTR ← SRC
```

Description

The LLDT instruction loads the Local Descriptor Table register (LDTR). The word operand (memory or register) used with the LLDT instruction must contain a selector to the Global Descriptor Table (GDT). The GDT entry must be a Local Descriptor Table; the LDTR loads from the entry. The segment registers DS, ES, SS, FS, GS, and CS are not affected. The LDT field in the task state segment does not change. The selector operand can be 0; if so, the LDTR is marked invalid. All descriptor references (except by the LAR, VERR, VERW, or LSL instructions) cause a General Protection Fault (13).

Note: *The LLDT instruction is used in operating system software. It is not used in application programs.*

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates one of three conditions: the current privilege level is not 0, the result destination is a non-writable segment, or the code or data segments have an illegal memory-operand effective address. General Protection Fault (13) indicates the selector operand does not point into the Global Descriptor Table, or if the entry in the GDT is not a Local Descriptor Table. Segment Not Present (11) indicates the LDT descriptor is not present. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

Invalid Opcode (6) occurs because the LLDT instruction is not recognized in Real Address Mode.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) occurs because the LLDT instruction is not recognized in Virtual 8086 Mode.

Note: *The operand-size attribute has no effect on this instruction.*

2.171 LMSW Loads Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /6	LMSW r/m16	13/13	Loads r/m16 into the machine status word.

Operation

```
MSW ← r/m16; (* 16 bits is stored in the machine status word *)
```

Description

The LMSW instruction loads the machine status word (part of the CR0 register) from the source operand. This instruction can be used to switch to Protected Mode; if so, it must be followed by an intrasegment jump to flush the instruction queue. The LMSW instruction will not switch back to Real Address Mode.

***Note:** The LMSW instruction is used only in operating system software. It is not used in application programs.*

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates one of three conditions: the current privilege level is not 0, the result destination is a non-writable segment, or the code or data segments have an illegal memory-operand effective address. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault.

***Note:** The operand-size attribute has no effect on this instruction. This instruction is provided for compatibility with the 80286 microprocessor; programs for the Am486 microprocessor should use the MOV CR0, ... instruction instead. The LMSW instruction does not affect the PG or ET bits, and it cannot be used to clear the PE bit.*

2.172 LOCK Asserts $\overline{\text{LOCK}}$ Signal Prefix

Opcode	Instruction	Clocks	Description
F0	LOCK	1	Asserts $\overline{\text{LOCK}}$ signal for the next instruction.

Description

The LOCK prefix causes the processor to assert the $\overline{\text{LOCK}}$ signal during execution of the following instruction. In a multiprocessor environment, use of this signal ensures that the processor has exclusive use of any shared memory while $\overline{\text{LOCK}}$ is asserted. The read-modify-write sequence typically used to implement test and set on the processor is the BTS instruction.

LOCK functions only with the following instructions:

BTS, BTR, BTC	mem, reg/imm
XCHG	reg, mem
XCHG	mem, reg
ADD, OR, ADC, SBB, AND, SLTB, XOR	mem, reg/imm
NOT, NEG, INC, DEC	mem
CMPXCHG, XADD	reg/mem, reg

Using the LOCK prefix with any instruction not listed above generates an undefined opcode trap.

The XCHG instruction always asserts $\overline{\text{LOCK}}$ regardless of the presence or absence of the LOCK prefix. The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

Flags Affected

None

Protected Mode Exceptions

Invalid Opcode (6) indicates the LOCK prefix is used with an instruction not listed in the 'Description' section above; other exceptions can be generated by the subsequent (locked) instruction.

Real Address Mode Exceptions

Invalid Opcode (6) indicates the LOCK prefix is used with an instruction not listed in the 'Description' section above; other exceptions can be generated by the subsequent (locked) instruction.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) indicates the LOCK prefix is used with an instruction not listed in the 'Description' section above; exceptions can still be generated by the subsequent (locked) instruction.

2.173 LODS/LODSB/LODSD/LODSW Loads String Operand

Opcode	Instruction	Clocks	Description
AC	LODS m8	5	Loads byte (E)SI into AL.
AD	LODS m16	5	Loads word (E)SI into AX.
AD	LODS m32	5	Loads doubleword (E)SI into EAX.
AC	LODSB	5	Loads byte DS:(E)SI into AL.
AD	LODSD	5	Loads doubleword DS:(E)SI into EAX.
AD	LODSW	5	Loads word DS:(E)SI into AX.

Operation

```
AddressSize = 16
THEN use SI for source-index
ELSE (* AddressSize = 32 *)
   use ESI for source-index;
FI;
IF byte type of instruction
THEN
   AL ← [source-index); (* byte load *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← –1; FI;
ELSE
   IF OperandSize = 16
   THEN
      AX ← [source-index]; (* word load *)
      IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← –2; FI;
   ELSE (* OperandSize = 32 *)
      EAX ← [source-index]; (* doubleword load *)
      IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← –4; FI;
   FI;
FI;
source-index ← source-index + IncDec
```

Description

LODS loads the memory byte, word, or doubleword at the location pointed to by the source-index register into the AL, AX, or EAX register. After the transfer, the instruction automatically advances the source-index register. If DF = 0 (the CLD instruction was executed), the source index increments; if DF = 1 (the STD instruction was executed), it decrements. The increment/decrement rate is 1 for a byte, 2 for a word, or 4 for a doubleword. If the address-size attribute is 16 bits, the SI register is the source-index register; otherwise, the ESI register is used. The source data address is determined solely by the contents of the source-index register; load the correct index value into the register before executing LODS. LODSB, LODSW, and LODSD are synonyms for the byte, word, and doubleword LODS instructions.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.174 LOOP/LOOPE/LOOPNE/LOOPNZ/LOOPZ Loop Control CX Counter

Opcode	Instruction	Clocks	Description
E2	LOOP rel8	2,6	Decrements count; jumps short if CX ≠ 0.
E1 cb	LOOPE rel8	9,6	Decrements count; jumps short if CX ≠ 0 and ZF = 1.
E0 cb	LOOPNE rel8	9,6	Decrements count; jumps short if CX ≠ 0 and ZF = 0.
E0 cb	LOOPNZ rel8	9,6	Decrements count; jumps short if CX ≠ 0 and ZF = 0.
E1 cb	LOOPZ rel8	9,6	Decrements count; jumps short if CX ≠ 0 and ZF = 1.

Operation

```
IF AddressSize = 16 THEN CountReg is CX ELSE CountReg is ECX; FI;
CountReg ← CountReg –1;
IF instruction ≠ LOOP
THEN
   IF (instruction = LOOPE) OR (instruction = LOOPZ)
   THEN BranchCond ← (ZF = 1) AND (CountReg ≠ 0); FI;
   IF (instruction = LOOPNE) OR (instruction = LOOPNZ)
   THEN BranchCond ← (ZF = 0) AND (CountReg ≠ 0); FI; FI;

IF BranchCond
THEN
   IF OperandSize = 16
   THEN
      IP ← IP + SignExtend(re/8);
   ELSE (* OperandSize = 32 *)
      EIP ← EIP + SignExtend(re/8); FI; FI
```

Description

LOOP instructions provide iteration control, combining loop index management with conditional branching. Load an unsigned iteration count into the count register, then code the LOOP instruction at the end of the iterative instruction series. Make the LOOP destination the label at the beginning of the iteration. When executed, LOOP decrements the CX or ECX register without changing any flags. Then it checks the register and, if required, ZF. If the conditions are met, LOOP executes a short jump to the label. The address-size attribute determines whether to use the CX (16-bit) or ECX (32-bit) register as the count register. The LOOP operand must be in the range from 128 (decimal) bytes before the instruction to 127 bytes after the instruction.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the offset is beyond the current code segment limits.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *The unconditional LOOP instruction takes longer to execute than a 2-instruction sequence that decrements the count register and jumps if the count does not equal zero. All branches are converted into 16-byte code fetches regardless of jump address or cacheability.*

2.175 LSL Loads Segment Limit

Opcode	Instruction	Clocks	Description
0F 03 /r	LSL r16,r/m16	10/10	r16 ← segment limit, selector r/m16 (byte granular)
0F 03 /r	LSL r32,r/m32	10/10	r32 ← segment limit, selector r/m32 (byte granular)
0F 03 /r	LSL r16,r/m16	10/10	r16 ← segment limit, selector r/m16 (page granular)
0F 03 /r	LSL r32,r/m32	10/10	r32 ← segment limit, selector r/m32 (page granular)

Description

If the source selector within the descriptor table is visible at the CPL and RPL, and the descriptor is a type accepted by LSL, the instruction loads a register with an unscrambled segment limit and sets ZF. Otherwise, ZF is cleared and the destination register is unchanged. The segment limit loads as a byte-granular value. If the descriptor has a page-granular segment limit, LSL translates it to a byte limit before loading it into the destination register (shifts the 20-bit “raw” limit from descriptor 12 bits left, then ORs with 00000FFFh). The 32-bit forms of the LSL instruction store the 32-bit byte granular limit in the 32-bit destination register. Code and data segment descriptors are valid for the LSL instruction.

The valid special segment and gate descriptor types for LSL are in the following table:

Type	Name	Valid/Invalid
0	Invalid	Invalid
1	Available 80286 TSS	Valid
2	LDT	Valid
3	Busy 80286 TSS	Valid
4	80286 call gate	Invalid
5	80286/486 task gate	Invalid
6	80286 trap gate	Invalid
7	80286 interrupt gate	Invalid
8	Invalid	Valid
9	Available 486 TSS	Valid
A	Invalid	Invalid
B	Busy 486 TSS	Valid
C	486 call gate	Invalid
D	Invalid	Invalid
E	486 trap gate	Invalid
F	486 interrupt gate	Invalid

Flags Affected

If the selector is invisible or of the wrong type, LSL clears ZF; otherwise, it is set.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Invalid Opcode (6) occurs because LSL is not recognized in Real Address Mode.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) occurs because LSL is not recognized in Virtual 8086 Mode. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.176 LSS Loads Pointer Using SS

Opcode	Instruction	Clocks	Description
0F B2 /r	LSS r16,m16:16	6/12	Loads SS:r16 with pointer from memory.
0F B2 /r	LSS r32,m16:32	6/12	Loads SS:r32 with pointer from memory.

Operation

```
IF (OperandSize = 16)
THEN
   r16 ← [Effective Address]; (* 16-bit transfer *)
   SS ← [Effective Address + 2]; (* 16-bit transfer *)
   (* In Protected Mode, load the descriptor into the segment register *)
ELSE (* OperandSize = 32 *)
   r32 ← [Effective Address]; (* 32-bit transfer *)
   SS ← [Effective Address + 4]; (* 16-bit transfer *)
   (* In Protected Mode, load the descriptor into the segment register *)
FI;
IF selector is null THEN General Protection Fault; FI;
   Selector index is in limits ELSE General Protection Fault(selector);
   Selector’s RPL = CPL ELSE General Protection Fault(selector);
   AR byte indicates a writable data segment
      ELSE General Protection Fault(selector);
   DPL in the AR byte equals CPL ELSE General Protection Fault(selector);
   Segment is marked present ELSE Stack Fault(selector);
   Load SS with selector;
   Load SS with descriptor;
```

Description

LSS reads a full pointer from memory and stores it in a register pair consisting of the SS register and a second operand-specified register. The first 16 bits are in SS and the remaining 16 or 32 bits (as specifed by the operand size) are placed into the register specified by the r16 or r32 register operand. The segment register descriptor comes from the selector descriptor table entry.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Invalid Opcode (6) indicates the second operand is a register. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.177 LTR Loads Task Register

Opcode	Instruction	Clocks	Description
0F 00 /3	LTR r/m16	20/20	Loads EA word into task register.

Description

The LTR instruction loads the task register from the source register or memory location specified by the operand. The loaded TSS is marked busy. A task switch does not occur.

***Note:** The LTR instruction is used only in operating system software. It is not used in application programs.*

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the current privilege level is not 0 or that there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. General Protection Fault (13) with a selector indicates the object named by the source selector is not a TSS or is already busy. Segment Not Present (11) with a selector indicates the TSS is marked “not present.” Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

Invalid Opcode (6) occurs because the LTR instruction is not recognized in Real Address Mode.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) occurs because the LTR instruction is not recognized in Virtual 8086 Mode.

***Note:** The operand-size attribute has no effect on this instruction.*

2.178 MOV Moves Data/Registers

Opcode	Instruction	Clocks	Description
88 /r	MOV r/m8,r8	1	Moves byte register to r/m byte.
89 /r	MOV r/m16,r16	1	Moves word register to r/m word.
89 /r	MOV r/m32,r32	1	Moves doubleword register to r/m doubleword.
8A /r	MOV r8,r/m8	1	Moves r/m byte to byte register.
8B /r	MOV r16,r/m16	1	Moves r/m word to word register.
8B /r	MOV r32,r/m32	1	Moves r/m doubleword to doubleword register.
8C /r	MOV r/m16,Sreg	3/3	Moves segment register to r/m word.
8E /r	MOV Sreg,r/m16	3/9	Moves r/m word to segment register.
A0	MOV AL,moffs8	1	Moves byte at (seg:offset) to AL.
A1	MOV AX,moffs16	1	Moves word at (seg:offset) to AX.
A1	MOV EAX,moffs32	1	Moves doubleword at (seg:offset) to EAX.
A2	MOV moffs8,AL	1	Moves AL to (seg:offset).
A3	MOV moffs16,AX	1	Moves AX to (seg:offset).
A3	MOV moffs32,EAX	1	Moves EAX to (seg:offset).
B0 + rb	MOV reg8,imm8	1	Moves immediate byte to register.
B8 + rw	MOV reg16,imm16	1	Moves immediate word to register.
B8 + rd	MOV reg32,imm32	1	Moves immediate doubleword to register.
C6	MOV r/m8,imm8	1	Moves immediate byte to r/m byte.
C7	MOV r/m16,imm16	1	Moves immediate word to r/m word.
C7	MOV r/m32,imm32	1	Moves immediate doubleword to r/m doubleword.
	Special Registers:		
0F 22 /r	MOV CR0,r32	16	Moves (register) to (control register).
0F 20 /r	MOV r32,CR0/CR2/CR3	4	Moves (control register) to (register).
0F 22 /r	MOV CR2/CR3,r32	4	Moves (register) to (control register).
0F 21 /r	MOV r32,DR0/DR1/DR2/DR3	10	Moves (debug register) to (register).
0F 21 /r	MOV r32,DR6/DR7	10	Moves (debug register) to (register).
0F 23 /r	MOV DR0 -3,r32	11	Moves (register) to (debug register).
0F 23 /r	MOV DR6/DR7,r32	11	Moves (register) to (debug register).
0F 24 /r	MOV r32,TR4/TR5/TR6/TR7	4	Moves (test register) to (register).
0F 26 /r	MOV TR4/TR5/TR6/TR7,r32	4	Moves (register) to (test register).
0F 24 /r	MOV r32,TR3	3	Moves (test register3) to (register).
0F 26 /r	MOV TR3,r32	6	Moves (registers) to (test register3).

Note: *moffs8, moffs16, and moffs32 all consist of a simple offset relative to the segment base. The 8, 16, and 32 refer to the data size. The address-size attribute of the instruction determines the size of the offset, either 16 or 32 bits.*

Operation

```
DEST ← SRC
```

Description

The MOV instruction copies the second operand to the first operand. If the destination is a segment register (DS, ES, SS, etc.), then descriptor data is also loaded into the register. The data for the register is obtained from the descriptor table entry for the selector given. You can load a null selector (values 0000–0003) into the DS and ES registers without causing an exception; however, use of the DS or ES register causes a General Protection Fault (13) exception and no memory reference occurs. A MOV into SS instruction inhibits all interrupts until after the execution of the next instruction (which is presumably a MOV into ESP instruction).

Loading a segment register under Protected Mode results in special checks and actions, as described in the following listing:

```
IF SS is loaded;
THEN
   IF selector is null THEN General Protection Fault; FI;
Index must be within limits else General Protection Fault(selector);
Selector’s RPL equals CPL else General Protection Fault(selector);
AR byte indicates a writable data segment
   ELSE General Protection Fault(selector);
   DPL in the AR byte equals CPL ELSE General Protection Fault(selector);
   Segment is marked present ELSE Stack Fault(selector);
Load SS with selector;
Load SS with descriptor; FI;
IF DS, ES, FS or GS is loaded with non-null selector;
THEN
   Index is within limits ELSE General Protection Fault(selector);
   AR byte indicates data or readable code segment
      ELSE General Protection Fault(selector);
   IF data or non-conforming code segment
   THEN RPL and CPL are ≤ DPL in AR byte;
      ELSE Gen.Protect.Fault(selector);FI;
Segment is marked present ELSE Segment Not Present Fault(selector);
Load segment register with selector;
Load segment register with descriptor;FI;
IF DS, ES, FS or GS is loaded with a null selector;
THEN
   Load segment register with selector;
   Clear descriptor valid bit; FI
```

The last eleven listed forms of the MOV instruction store or load the following special registers in or from a general purpose register:

- Control registers CR0, CR2, and CR3
- Debug Registers DRO, DR1, DR2, DR3, DR6, and DR7
- Test Registers TR3, TR4, TR5, TR6, and TR7

Note: *32-bit operands are always used with these instructions, regardless of the operand-size attribute.*

Flags Affected

MOV data: None
MOV register: OF, SF, ZF, AF, PF, and CF are undefined.

Protected Mode Exceptions

MOV data: General Protection Fault (13), Stack Fault (12), and Segment Not Present (11) occur if a segment register is being loaded; otherwise, General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

MOV register: General Protection Fault (13) indicates the current privilege level is not 0.

Real Address Mode Exceptions

MOV data: General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

MOV register: None

Virtual 8086 Mode Exceptions

MOV data: General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

MOV register: General Protection Fault (13) occurs if instruction execution is attempted.

Note: *MOV register instructions must be executed at privilege level 0 or in Real Address Mode; otherwise, a protection exception will be raised. The reg field within the ModR/M byte specifies which of the special registers in each category is involved. The two bits in the mod field are always 11. The r/m field specifies the general register involved. Always set undefined or reserved bits to the value previously read.*

2.179 MOVS/MOVSB/MOVSD/MOVSW Moves Data from String to String

Opcode	Instruction	Clocks	Description
A4	MOVS m8,m8	7	Moves byte (E)SI to ES:(E)DI.
A5	MOVS m16,m16	7	Moves word (E)SI to ES:(E)DI.
A5	MOVS m32,m32	7	Moves doubleword (E)SI to ES:(E)DI.
A4	MOVSB	7	Moves byte (E)SI to ES:(E)DI.
A5	MOVSD	7	Moves doubleword (E)SI to ES:(E)DI.
A5	MOVSW	7	Moves word (E)SI to ES:(E)DI.

Operation

```
IF (instruction = MOVSD) OR (instruction has doubleword operands)
THEN OperandSize ← 32;
ELSE OperandSize ← 16;
IF AddressSize = 16
THEN use Sl for source-index and DI for destination-index;
ELSE (* AddressSize = 32 *)
   use ESI for source-index and EDI for destination-index; FI;
IF byte type of instruction
THEN
   [destination-index] ← [source-index); (* byte assignment *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← –1; FI;
ELSE
   IF OperandSize = 16
   THEN
      [destination-index] ← [source-index]; (* word assignment *)
      IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← –2; FI;
   ELSE (* OperandSize = 32 *)
      [destination-index] ← [source-index); (* doubleword assignment *)
      IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← –4; FI;
   FI;
FI;
source-index ← source-index + IncDec;
destination-index ← destination-index + IncDec
```

Description

MOVS copies the byte, word, or doubleword at SI or ESI to the byte, word, or doubleword at ES:DI or ES:EDI. The destination operand must be addressable from the ES register; no segment override is possible for the destination. You can use a segment override for the source operand; the default is the DS register. The contents of SI and DI (or ESI and EDI for 32-bit values) determine the source and destination addresses. Load the correct index values into the SI and DI (or ESI and EDI) registers before executing the MOVS instruction. After moving the data, MOVS advances the SI and DI (or ESI and EDI) registers automatically. If the Direction Flag (DF) is 0 (see STC), the registers increment; if DF is 1 (see STD), the registers decrement. The stepping is 1 for a byte, 2 for a word, or 4 for a doubleword operand.

MOVSB, MOVSW, and MOVSD are synonyms for the byte, word, and doubleword MOVS instructions.

You can use the REP prefix with MOVS for movement of CX bytes or words.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.180 MOVSX Moves with Sign Extension

Opcode	Instruction	Clocks	Description
0F BE /r	MOVSX r16,r/m8	3/3	Moves byte to word with sign-extend.
0F BE /r	MOVSX r32,r/m8	3/3	Moves byte to doubleword with sign-extend.
0F BF /r	MOVSX r32,r/m16	3/3	Moves word to doubleword with sign-extend.

Operation

```
DEST ← SignExtend(SRC)
```

Description

The MOVSX instruction reads the contents of the effective address or register as a byte or a word, sign-extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.181 MOVZX Moves with Zero Extension

Opcode	Instruction	Clocks	Description
0F B6 /r	MOVZX r16,r/m8	3/3	Moves byte to word with zero-extend.
0F B6 /r	MOVZX r32,r/m8	3/3	Moves byte to doubleword with zero-extend.
0F B7 /r	MOVZX r32,r/m16	3/3	Moves word to doubleword with zero-extend.

Operation

```
DEST ← ZeroExtend(SRC)
```

Description

The MOVZX instruction reads the contents of the effective address or register as a byte or a word, zero extends the value to the operand-size attribute of the instruction (16 or 32 bits), and stores the result in the destination register.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.182 MUL Unsigned Multiply

Opcode	Instruction	Clocks	Description
F6 /4	MUL AL,r/m8	13/18,13/18	Unsigned multiply (AX←AL · r/m byte)
F7 /4	MUL AX,r/m16	13/26,13/26	Unsigned multiply (DX:AX ← AX · r/m word)
F7 /4	MUL EAX,r/m32	13/42,13/42	Unsigned multiply (EDS:EAX ← EAX · r/m doubleword)

Actual clock count depends on the most-significant bit location in the optimizing multiplier. If the multipler (*m*) = 0, the clock count is 9. Otherwise clock = max (ceiling($\log_2 |m|$), 3) + 6.

Operation

```
IF byte-size operation
THEN AX ← AL · r/m8
ELSE (* word or doubleword operation *)
   IF OperandSize = 16
   THEN DX:AX ← AX · r/m16
   ELSE (* OperandSize = 32 *)
      EDX:EAX ← EAX · r/m32
   FI;
FI
```

Description

The MUL instruction performs unsigned multiplication. Its actions depend on the size of its operand, as follows:

- A byte operand is multiplied by the AL value; the result is left in the AX register. The CF and OF flap are cleared if the AH value is 0; otherwise, they are set.
- A word operand is multiplied by the AX value; the result is left in the DX:AX register pair. The DX register contains the high-order 16 bits of the product. CF and OF are cleared if the DX value is 0; otherwise, they are set.
- A doubleword operand is multiplied by the EAX value and the result is left in the EDX:EAX register. The EDX register contains the high-order 32 bits of the product. CF and OF are cleared if the EDX value is 0; otherwise, they are set.

Flags Affected

OF and CF are cleared if the upper half of the result is 0; otherwise they are set. SF, ZF, AF, and PF are undefined.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.183 NEG Two’s Complement Negation

Opcode	Instruction	Clocks	Description
F6 /3	NEG r/m8	1/3	Performs a two’s complement negation of r/m byte.
F7 /3	NEG r/m16	1/3	Performs a two’s complement negation of r/m word.
F7 /3	NEG r/m32	1/3	Performs a two’s complement negation of r/m doubleword.

Operation

```
IF r/m = 0 THEN CF ← 0 ELSE CF ← 1; FI;
r/m ← –r/m
```

Description

The NEG instruction replaces the value of a register or memory operand with its two’s complement. The operand is subtracted from zero and the result is placed in the operand.

NEG sets CF if the operand is not zero. If the operand is zero, NEG clears CF.

Flags Affected

CF is set unless the operand is zero. OF, SF, ZF, and PF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.184 NOP No Operation

Opcode	Instruction	Clocks	Description
90	NOP	1	No operation is performed.

Description

The NOP instruction performs no operation. The NOP instruction is a 1-byte instruction that takes up space but affects none of the machine context except the instruction pointer.

The NOP instruction is an alias mnemonic for the XCHG AX, AX or XCHG EAX, EAX instruction.

Flags Affected

None

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.185 NOT One's Complement Negation

Opcode	Instruction	Clocks	Description
F6 /2	NOT r/m8	1/3	Reverses each bit in r/m byte.
F7 /2	NOT r/m16	1/3	Reverses each bit in r/m word.
F7 /2	NOT r/m32	1/3	Reverses each bit in r/m doubleword.

Operation

```
r/m ← NOT r/m
```

Description

The NOT instruction inverts the operand; every 1 becomes a 0, and vice versa.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.186 OR Logical Inclusive OR

Opcode	Instruction	Clocks	Description
0C ib	OR AL,imm8	1	ORs immediate byte to AL.
0D iw	OR AX,imm16	1	ORs immediate word to AX.
0D id	OR EAX,imm32	1	ORs immediate doubleword to EAX.
80 /1 ib	OR r/m8,imm8	1/3	ORs immediate byte to r/m byte.
81 /1 iw	OR r/m16,imm16	1/3	ORs immediate word to r/m word.
81 /1 id	OR r/m32,imm32	1/3	ORs immediate word to r/m doubleword.
83 /1 ib	OR r/m16,imm8	1/3	ORs sign-extended immediate byte to r/m word.
83 /1 ib	OR r/m 32,imm8	1/3	ORs sign-ext. immediate byte to r/m doubleword.
08 /r	OR r/m8,r8	1/3	ORs byte register to r/m byte.
09 /r	OR r/m16,r16	1/3	ORs word register to r/m word.
09 /r	OR r/m32,r32	1/3	ORs doubleword register to r/m doubleword.
0A /r	OR r8,r/m8	1/2	ORs r/m byte to byte register.
0B /r	OR r16,r/m16	1/2	ORs r/m word to word register.
0B /r	OR r32,r/m32	1/2	ORs r/m doubleword to doubleword register.

Operation

```
DEST ← DEST OR SRC;
CF ← 0;
OF ← 0
```

Description

The OR instruction computes the inclusive OR of its two operands and places the result in the first operand. Each bit of the result is 0 if both corresponding bits of the operands are 0; otherwise, each bit is 1.

Flags Affected

OF and CF are cleared. SF, ZF, and PF are set according to the result. AF is undefined.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.187 OUT Outputs to Port

Opcode	Instruction	Clocks*	Description
E6 ib	OUT imm8,AL	*All forms:*	Outputs byte AL to immediate port number.
E7 ib	OUT imm8,AX	rm = 16, vm = 29	Outputs word AX to immediate port number.
E7 ib	OUT imm8,EAX	If CPL ≤ IOPL,	Outputs doubleword EAX to imm. port number.
EE	OUT DX,AL	pm = 11,10	Outputs byte AL to port number in DX.
EF	OUT DX,AX	If CPL>IOPL,	Outputs word AX to port number in DX.
EF	OUT DX,EAX	pm = 31,30	Outputs double EAX to port number in DX.

*rm is Real Mode, vm is Virtual 8086 Mode, pm is Protected Mode. For pm, the first number is the value for the imm8 form, and the second number is for the DX form of the port number.

Operation

```
IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 Mode, or Protected Mode with CPL > IOPL *)
   IF NOT I/O-Permission (DEST, width(DEST))
   THEN General Protection Fault (13);
   FI;
FI;
[DEST] ← SRC; (* I/O address space used *)
```

Description

The OUT instruction transfers a data byte or data word from the register (AL, AX, or EAX) given as the second operand to the output port numbered by the first operand. Output to any port from 0 to 65535 is performed by placing the port number in the DX register and then using an OUT instruction with the DX register as the first operand. If the instruction contains an 8-bit port ID, that value is zero-extended to 16 bits.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates the current privilege level is higher (has less privilege) than the I/O privilege level, and any of the corresponding I/O permission bits in the TSS equals 1.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that one of the corresponding I/O permission bits in the TSS equals 1.

2.188 OUTS/OUTSB/OUTSD/OUTSW Output String to Port

Opcode	Instruction	Clocks	Description
6E	OUTS DX,r/m8	*All forms:*	Outputs byte (E)SI to port in DX.
6F	OUTS DX,r/m16	rm = 17, vm = 30	Outputs word (E)SI to port in DX.
6F	OUTS DX,r/m32	If CPL ≤ IOPL,	Outputs doubleword (E)SI to port in DX.
6E	OUTSB	pm = 10	Outputs byte (E)SI to port in DX.
6F	OUTSD	If CPL>IOPL,	Outputs word (E)SI to port in DX.
6F	OUTSW	pm = 32	Outputs doubleword (E)SI to port in DX.

Operation

```
IF AddressSize = 16
THEN use Sl for source-index;
ELSE (* AddressSize = 32 *)
   use ESI for source-index;
FI;

IF (PE = 1) AND ((VM = 1) OR (CPL > IOPL))
THEN (* Virtual 8086 Mode, or Protected Mode with CPL > IOPL *)
   IF NOT I/O-Permission (DEST, width(DEST))
   THEN General Protection Fault (13);
   FI;
FI;
IF byte type of instruction
THEN
   [DX] ← [source-index]; (* Write byte at DX 1/0 address *)
   IF DF = 0 THEN IncDec ← 1 ELSE IncDec ← –1; FI;
FI;
IF OperandSize = 16
THEN
   [DX] ← [source-index]; (* Write word at DX I/O address *)
   IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← –2; FI;
FI;
IF OperandSize = 32
THEN
   [DX] ← [source-index]; (* Write doubleword at DX I/O address *)
   IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← –4; FI;
   FI;
FI;
source-index ← source-index + IncDec
```

Description

OUTS transfers data from the address indicated by the source-index register SI (16-bit addresses) or ESI (32-bit addresses) to the output port addressed by the DX register. OUTS does not allow specification of the port number as an immediate value. You must address the port through the DX register value. Load the correct values into the DX register and the source-index (SI or ESI) register before executing the OUTS instruction.

After the transfer, the source-index register advances automatically. If the Direction Flag (DF) is 0 (see CLD), the source-index register increments; if DF is 1 (see STD), it decrements. The increment/decrement rate is 1 for a byte, 2 for a word, or 4 for a doubleword.

OUTSB, OUTSW, and OUTSD are synonyms for the byte, word, and doubleword OUTS instructions.

You can use the REP prefix with the OUTS instruction for block output of CX bytes or words.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates one of three conditions: the current privilege level is greater than the I/O privilege level and at least one of the I/O permission bits in TSS equals 1, the result destination is a non-writable segment, or the code or data segments have an illegal memory-operand effective address. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that at least one of the corresponding I/O permission bits in TSS equals 1. General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.189 POP Pops Word from Stack

Opcode	Instruction	Clocks	Description
8F /0	POP m16	6	Pops top of stack into memory word.
8F /0	POP m32	6	Pops top of stack into memory doubleword.
58 + rw	POP r16	4	Pops top of stack into word register.
58 + rd	POP r32	4	Pops top of stack into doubleword register.
1F	POP DS	3	Pops top of stack into DS.
07	POP ES	3	Pops top of stack into ES.
17	POP SS	3	Pops top of stack into SS.
0F A1	POP FS	3	Pops top of stack into FS.
0F A9	POP GS	3	Pops top of stack into GS.

Operation

```
IF StackAddrSize = 16
THEN
   IF OperandSize = 16
   THEN
      DEST ← (SS:SP); (* copy a word *)
      SP ← SP + 2;
   ELSE (* OperandSize = 32 *)
      DEST ← (SS:SP); (* copy a doubleword *)
      SP ← SP + 4 FI;
ELSE (* StackAddrSize = 32 * )
   IF OperandSize = 16
   THEN
      DEST ← (SS: ESP); (* copy a word *)
      ESP ← ESP + 2;
   ELSE (* OperandSize = 32 *)
      DEST ← (SS:ESP); (* copy a doubleword *)
      ESP ← ESP + 4 FI;FI;
(* Protected Mode execution uses the following special checks and actions *)
IF SS is loaded:
   IF selector is null THEN General Protection Fault;
   Selector index is within its descriptor table limits
      ELSE General Protection Fault(selector);
   Selector’s RPL equals CPL ELSE General Protection Fault(selector);
   AR byte indicates writable data segment
      ELSE General Protection Fault(selector);
   DPL in the AR byte equals CPL ELSE General Protection Fault(selector);
   Segment must be marked present ELSE Stack Fault(selector);
   Load SS register with selector;
   Load SS register with descriptor;

IF DS, ES, FS or GS is loaded with non-null selector:
   AR byte must indicate data or readable code segment
      ELSE General Protection Fault(selector);
   IF data or non-conforming code
   THEN RPL and CPL must be less than or equal to DPL in
      AR byte
   ELSE General Protection Fault (13)(selector) FI;
   Segment must be marked present ELSE Segment Not Present (11)(selector);
   Load segment register with selector;
   Load segment register with descriptor;

IF DS, ES, FS, or GS is loaded with a null selector:
   Load segment register with selector
   Clear valid bit in invisible portion of register
```

Description

POP loads the word at the top of the processor stack into the destination specified by the operand. The top of the stack is specified by the contents of SS and either stack pointer register: SP for 16-bit addresses or ESP for 32-bit addresses. The stack pointer increments by 2 for a 16-bit operand or by 4 for a 32-bit operand to point to the new top of stack.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the value popped must be a selector. In Protected Mode, loading the selector initiates automatic loading of the descriptor information associated with that selector into the hidden part of the segment register; loading also initiates validation of both the selector and the descriptor information.

A null value (0000–0003) may be popped into the DS, ES, FS, or GS register without causing a protection exception. An attempt to reference a segment whose corresponding segment register is loaded with a null value causes a General Protection Fault (13) exception. No memory reference occurs. The saved value of the segment register is null.

A POP SS instruction inhibits all interrupts, including NMI, until after execution of the next instruction. This allows sequential execution of POP SS and POP SP (or POP ESP) instructions without danger of having an invalid stack during an interrupt. However, use of the LSS instruction is the preferred method of loading the SS and SP (or ESP) registers.

A POP-to-memory instruction that uses the stack pointer as a base register references memory after the POP. The base is the value of the stack pointer after the instruction executes.

Note: *POP CS is not a 486-processor instruction; use RET to pop from the stack into CS.*

Flags Affected

None

Protected Mode Exceptions

Segment Not Present (11) occurs if the segment descriptor indicates the segment is not present in memory; a Stack Fault (12) and a General Protection Fault (13) occur automatically with this error. By itself, a Stack Fault (12) indicates either that the current top of stack is not within the stack segment, or that the SS segment address is illegal. By itself, a General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *Back-to-back PUSH/POP instruction sequences are allowed without incurring an additional clock. The SSB bit determines the Stack Address Size. Pop ESP instructions increment the stack pointer (ESP) before data at the old top of stack is written into the destination.*

2.190 POPA Pops All 16-Bit General Registers

Opcode	Instruction	Clocks	Description
61	POPA	9	Pops DI, SI, BP, BX, DX, CX, and AX.

Operation

```
DI ← Pop();
Sl ← Pop();
BP ← Pop();
Increment SP by 2 (* skip next 2 bytes of stack *)
BX ← Pop();
DX ← Pop();
CX ← Pop();
AX ← Pop()
```

Description

POPA pops the eight 16-bit general registers, but it discards the SP value instead of loading it into the SP register. POPA reverses a previous PUSHA, restoring the general registers to their values before the PUSHA instruction was executed. POPA pops the DI register first.

Flags Affected

None

Protected Mode Exceptions

Stack Fault (12) indicates the starting or ending stack address is not within the stack segment. Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault.

2.191 POPAD Pops All 32-Bit General Registers

Opcode	Instruction	Clocks	Description
61	POPAD	9	Pops EDI, ESI, EBP, EDX, ECX, and EAX.

Operation

```
EDI ← Pop();
ESI ← Pop();
EBP ← Pop();
increment SP by 4 (* skip next 4 bytes of stack *)
EBX ← Pop();
EDX ← Pop();
ECX ← Pop();
EAX ← Pop()
```

Description

POPAD pops the eight 32-bit general registers, but discards the ESP value instead of loading it into the ESP register. POPAD reverses the previous PUSHAD instruction, restoring the general registers to their values before the PUSHAD instruction executed. POPAD pops the EDI register first.

Flags Affected

None

Protected Mode Exceptions

Stack Fault (12) indicates the starting or ending stack address is not within the stack segment. Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault.

2.192 POPF/POPFD Pops Stack into FLAGS or EFLAGS Register

Opcode	Instruction	Clocks	Description
9D	POPF	9, pm = 6	Pops word on top of stack into FLAGS.
9D	POPFD	9, pm = 6	Pops doubleword on top of stack into EFLAGS.

Operation

```
Flags ← Pop()
```

Description

POPF and POPFD instructions pop a word or doubleword on the top of the stack and store the value in the FLAGS or EFLAGS register. If the instruction operand-size attribute is 16 bits, a word is popped and stored in the FLAGS register. If the operand-size attribute is 32 bits, a doubleword is popped and stored in the EFLAGS register.

***Note:** Note that bits 16 and 17 of the EFLAGS register, called the VM and RF flags, respectively, are not affected by the POPF or POPFD instruction.*

The I/O privilege level is altered only when executing at privilege level 0. The Interrupt Flag is altered only when executing at a level at least as privileged as the I/O privilege level. (Real Address Mode is equivalent to privilege level 0.) If a POPF instruction is executed with insufficient privilege, an exception does not occur and the privileged bits do not change.

Flags Affected

All except the VM and RF flags are affected.

Protected Mode Exceptions

Stack Fault (12) indicates the top of stack is not within the stack segment.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

To maintain emulation, General Protection Fault (13) indicates the I/O privilege level is less than 3.

2.193 PUSH Pushes Operand onto Stack

Opcode	Instruction	Clocks	Description
FF /6	PUSH m16	4	Pushes memory word
FF /6	PUSH m32	4	Pushes memory doubleword
50 + /r	PUSH r16	1	Pushes register word
50 + /r	PUSH r32	1	Pushes register doubleword
6A	PUSH imm8	1	Pushes immediate byte
68	PUSH imm16	1	Pushes immediate word
68	PUSH imm32	1	Pushes immediate doubleword
0E	PUSH CS	3	Pushes CS
16	PUSH SS	3	Pushes SS
1E	PUSH DS	3	Pushes DS
06	PUSH ES	3	Pushes ES
0F A0	PUSH FS	3	Pushes FS
0F A8	PUSH GS	3	Pushes GS

Operation

```
IF StackAddrSize = 16
THEN
   IF OperandSize = 16 THEN
      SP ← SP 2;
      (SS:SP) ← (SOURCE); (* word assignment *)
   ELSE
      SP ← SP – 4;
      (SS:SP) ← (SOURCE); (* doubleword assignment *) FI;
ELSE (* StackAddrSize = 32 *)
   IF OperandSize = 16
   THEN
      ESP ← ESP – 2;
      (SS:ESP) ← (SOURCE); (* word assignment *)
   ELSE
      ESP ← ESP – 4;
      (SS:ESP) ← (SOURCE); (* doubleword assignment *) FI;
FI
```

Description

PUSH decrements the stack pointer by 2 (16-bit operands) or 4 (32-bit operands). Then PUSH places the operand on the new stack top, indicated by the stack pointer.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates either that the new value of SP or ESP register is outside the stack segment limit, or that there is an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

None, but if SP or ESP is 1, the processor shuts down due to a lack of stack space.

Virtual 8086 Mode Exceptions

Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference. If SP or ESP is 1, the processor shuts down due to a lack of stack space.

Note: *When used with a memory operand, PUSH takes longer to execute than a two-instruction sequence that moves the operand through a register. Back-to-back PUSH/POP instruction sequences are allowed without incurring an additional clock. Selective pushes write only to the top of the stack.*

2.194 PUSHA Pushes All 16-Bit General Registers

Opcode	Instruction	Clocks	Description
60	PUSHA	11	Pushes AX, CX, DX, BX, original SP, BP, SI, and DI.

Operation

```
Temp ← (SP);
Push(AX);
Push(CX);
Push(DX);
Push(BX);
Push(Temp);
Push(BP);
Push(SI);
Push(DI)
```

Description

PUSHA saves the 16-bit general registers on the processor stack. PUSHA decrements the stack pointer (SP) by 16 to accommodate the required 8-word field. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 16 new stack bytes in reverse order. The last register pushed is the DI register.

Flags Affected

None

Protected Mode Exceptions

Stack Fault (12) indicates the starting or ending stack address is outside the stack segment limit. Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

General Protection Fault (13) occurs if SP equals 7, 9, 11, 13, or 15. If the SP register equals 1, 3, or 5 before executing the PUSHA instruction, the processor shuts down.

Virtual 8086 Mode Exceptions

General Protection Fault (13) occurs if SP equals 7, 9, 11, 13, or 15. If the SP register equals 1, 3, or 5 before executing the PUSHA instruction, the processor shuts down. Page Fault (14) indicates a page fault.

2.195 PUSHAD Pushes All 32-Bit General Registers

Opcode	Instruction	Clocks	Description
60	PUSHAD	11	Pushes EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

Operation

```
Temp ← (ESP);
Push(EAX);
Push(ECX);
Push(EDX);
Push(EBX);
Push(Temp);
Push(EBP);
Push(ESI);
Push(EDI)
```

Description

PUSHAD saves the 32-bit general registers on the processor stack. PUSHAD decrements the stack pointer (ESP) by 32 to accommodate the eight doubleword values. Because the registers are pushed onto the stack in the order in which they were given, they appear in the 32 new stack bytes in reverse order. The last register pushed is the EDI register.

Flags Affected

None

Protected Mode Exceptions

Stack Fault (12) indicates the starting or ending stack address is outside the stack segment limit. Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

General Protection Fault (13) occurs if SP equals 7, 9, 11, 13, or 15. If the SP register equals 1, 3, or 5 before executing the PUSHAD instruction, the processor shuts down.

Virtual 8086 Mode Exceptions

General Protection Fault (13) occurs if SP equals 7, 9, 11, 13, or 15. If the SP register equals 1, 3, or 5 before executing the PUSHAD instruction, the processor shuts down. Page Fault (14) indicates a page fault.

2.196 PUSHF/PUSHFD Pushes FLAGS Register onto the Stack

Opcode	Instruction	Clocks	Description
9C	PUSHF	4, pm = 3	Pushes FLAGS.
9C	PUSHFD	4, pm = 3	Pushes EFLAGS.

Operation

```
IF OperandSize = 32
THEN push(EFLAGS);
ELSE push(FLAGS);
FI
```

Description

The PUSHF instruction decrements the stack pointer by 2 and copies the FLAGS register to the new top of stack; the PUSHFD instruction decrements the stack pointer by 4, and copies the EFLAGS register to the new stack top pointed to by SS:ESP.

Flags Affected

None

Protected Mode Exceptions

Stack Fault (12) indicates the new value of the ESP register is outside the stack segment boundaries.

Real Address Mode Exceptions

None; the processor shuts down due to a lack of stack space.

Virtual 8086 Mode Exceptions

To maintain emulation, General Protection Fault (13) indicates the I/O privilege level is less than 3.

2.197 RCL Rotates through Carry Left

Opcode	Instruction	Clocks	Description
D0 /2	RCL r/m8,1	3/4	Rotates 9 bits (CF,r/m byte) left once.
D2 /2	RCL r/8,CL	3–30/9–31	Rotates 9 bits (CF,r/m byte) left CL times.
C0 /2 ib	RCL r/m8,imm8	8–30/9–31	Rotates 9 bits (CF,r/m byte) left imm8 times.
D1 /2	RCL r/m16,1	3/4	Rotates 17 bits (CF,r/m word) left once.
D3 /2	RCL r/m16,CL	8–30/9–31	Rotates 17 bits (CF,r/m word) left CL times.
C1 /2 ib	RCL r/m16,imm8	8–30/9–31	Rotates 17 bits (CF,r/m word) left imm8 times.
D1 /2	RCL r/m32,1	3/4	Rotates 33 bits (CF,r/m doubleword) left once.
D3 /2	RCL r/m32,CL	8–30/9–31	Rotates 33 bits (CF,r/m doubleword) left CL times.
C1 /2 ib	RCL r/m32,imm8	8–30/9–31	Rotates 33 bits (CF,r/m doubleword) left imm8 times.

Operation

```
temp C0UNT;
WHILE (temp ≠ 0)
DO
   tmpcf ← high-order bit of (r/m);
   r/m ← r/m · 2 + (tmpcf);
   temp ← temp – 1;
OD;
IF C0UNT = 1
THEN
   IF high-order bit of r/m ≠ CF
   THEN OF ← 1;
   ELSE OF ← 0;
   FI;
ELSE OF ← undefined FI
```

Description

RCL shifts CF into the bottom bit and shifts the top bit into CF. The second operand indicates the number of rotations. The operand is either an immediate number or the CL register contents. The processor does not allow rotation counts greater than 31, using only the bottom five bits of the operand if it is greater than 31. Virtual 8086 Mode masks rotation counts.

Flags Affected

OF is affected only by single-bit rotations but is undefined otherwise. CF contains the value of the bit shifted into it. SF, ZF, AF, and PF are not affected.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.198 RCR Rotates through Carry Right

Opcode	Instruction	Clocks	Description
D0 /3	RCR r/m8,1	3/4	Rotates 9 bits (CF,r/m byte) right once.
D2 /3	RCR r/8,CL	3–30/9–31	Rotates 9 bits (CF,r/m byte) right CL times.
C0 /3 ib	RCR r/m8,imm8	8–30/9–31	Rotates 9 bits (CF,r/m byte) right imm8 times.
D1 /3	RCR r/m16,1	3/4	Rotates 17 bits (CF,r/m word) right once.
D3 /3	RCR r/m16,CL	8–30/9–31	Rotates 17 bits (CF,r/m word) right CL times.
C1 /3 ib	RCR r/m16,imm8	8–30/9–31	Rotates 17 bits (CF,r/m word) right imm8 times.
D1 /3	RCR r/m32,1	3/4	Rotates 33 bits (CF,r/m doubleword) right once.
D3 /3	RCR r/m32,CL	8–30/9–31	Rotates 33 bits (CF,r/m doubleword) right CL times.
C1 /3 ib	RCR r/m32,imm8	8–30/9–31	Rotates 33 bits (CF,r/m doubleword) right imm8 times.

Operation

```
temp ← C0UNT;
WHILE (temp ≠ 0 )
DO
   tmpcf ← low-order bit of (r/m);
   r/m ← r/m / 2 + (tmpcf $\cdot 2$ width(r/m));
   temp ← temp – 1;
OD;
IF C0UNT = 1
THEN
   IF (high-order bit of r/m) ≠ (bit next to high-order bit of r/m)
   THEN OF ← 1;
   ELSE OF ← 0;
   FI;
ELSE OF ← undefined FI
```

Description

RCR shifts CF into the top bit and shifts the bottom bit into CF. The second operand indicates the number of rotations. The operand is either an immediate number or the CL register contents. The processor does not allow rotation counts greater than 31, using only the bottom five bits of the operand if it is greater than 31. Virtual 8086 Mode masks rotation counts.

Flags Affected

OF is affected only by single-bit rotations but is undefined otherwise. CF contains the value of the bit shifted into it. SF, ZF, AF, and PF are not affected.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.199 REP/REPE/REPNE/REPNZ/REPZ Repeats Specified String Operation

Opcode	Instruction	Clocks*	Description

F3 6C	REP INS r/8,DX	rm = 16+8(E)CX
		If CPL≤IOPL,
		pm = 10+8(E)CX
		If CPL>IOPL,
		pm = 30+8(E)CX
		vm = 29+8(E)CX
			Inputs (E)CX bytes from port DX into ES:(E)DI.

F3 6D	REP INS r/m16,DX		Inputs (E)CX words from port DX into ES:(E)DI.
F3 6D	REP INS r/m32,DX		Inputs (E)CX doublewords from port DX into ES:(E)DI.
F3 A4	REP MOVS m8,m8	If (E)CX = 0 5 If (E)CX = 1 13 If (E)CX > 1 12+3(E)CX	Moves (E)CX bytes from (E)SI to ES:(E)DI.
F3 A5	REP MOVS m16,m16		Moves (E)CX words from (E)SI to ES:(E)DI.
F3 A5	REP MOVS m32,m32		Moves (E)CX doublewords from (E)SI to ES:(E)DI.

F3 6E	REP OUTS DX,r/m8	rm = 17+5(E)CX
		If CPL≤IOPL,
		pm = 11+5(E)CX
		If CPL>IOPL,
		pm = 31+5(E)CX
		vm = 30+5(E)CX
			Outputs (E)CX bytes to port DX from ES:(E)DI.

F3 6F	REP OUTS DX,r/m16		Outputs (E)CX words to port DX from ES:(E)DI.
F3 6F	REP OUTS DX,r/m32		Outputs (E)CX doublewords to port DX from ES:(E)DI.
F2 AC	REP LODS m8	If (E)CX = 0, 5 IF (E)CX > 0, 7+4(E)CX	Loads (E)CX bytes from (E)SI to AL.
F2 AD	REP LODS m16		Loads (E)CX words from (E)SI to AX.
F2 AD	REP LODS m32		Loads (E)CX doublewords from (E)SI to EAX.
F3 AA	REP STOS m8		Fills (E)CX bytes at ES:(E)DI with AL.
F3 AB	REP STOS m16		Fills (E)CX words at ES:(E)DI with AX.
F3 AB	REP STOS m32		Fills (E)CX doublewords at ES:(E)DI with EAX.
F3 A6	REPE CMPS m8,m8		Finds nonmatching bytes in ES:(E)DI and (E)SI.
F3 A7	REPE CMPS m16,m16		Finds nonmatching words in ES:(E)DI and (E)SI.
F3 A7	REPE CMPS m32,m32		Finds nonmatching doublewords in ES:(E)DI and (E)SI.
F3 AE	REPE SCAS m8		Finds non-AL byte starting at ES:(E)DI.
F3 AF	REPE SCAS m16		Finds non-AX word starting at ES:(E)DI.
F3 AF	REPE SCAS m32		Finds non-EAX doubleword starting at ES:(E)DI.
F2 A6	REPNE CMPS m8,m8		Finds matching bytes in ES:(E)DI and (E)SI.
F2 A7	REPNE CMPS m16,m16		Finds matching words in ES:(E)DI and (E)SI.
F2 A7	REPNE CMPS m32,m32		Finds matching doublewords in ES:(E)DI and (E)SI.
F2 AE	REPNE SCAS m8		Finds AL, starting at ES:(E)DI.
F2 AF	REPNE SCAS m16		Finds AX, starting at ES:(E)DI.
F2 AF	REPNE SCAS m32		Finds EAX, starting at ES:(E)DI.

*Clock data is grouped by category. The category applies to all instructions to the left of the enclosed cell. Modes: rm = Real, pm = Protected, vm = Virtual. If no Mode is indicated, values apply to all modes.

Operation

```
IF AddressSize = 16
THEN use CX for CountReg;
ELSE (* AddressSize = 32 *) use ECX for CountReg FI;
WHILE CountReg ≠ 0
DO
   service pending interrupts (if any);
   perform primitive string instruction;
   CountReg ← CountReg – 1;
   IF primitive operation is CMPSB, CMPSW, SCASB, or SCASW
   THEN
      IF (instruction is REP/REPE/REPZ) AND (ZF = 0)
      THEN exit WHILE loop
      ELSE
         IF (instruction is REPNZ or REPNE) AND (ZF = 1)
         THEN exit WHILE loop FI FI FI;
OD
```

Description

The REPeat string instructions are prefixes used with string instructions. The prefix causes the string instruction to repeat the number of times indicated in the count register (CX or ECX) or (for the REPE/REPZ and REPNE/REPNZ prefixes) until the indicated condition in ZF is no longer met. You can only apply a REP prefix to one string instruction at a time. To repeat an instruction block, use the LOOP instruction or another looping construct.

REP begins by checking the address size to select the correct count register: CX (16-bit) or ECX (32-bit). Then REP checks the count register. If it is zero, execution moves to the next instruction. REP then allows the processor to acknowledge any pending interrupts. After interrupt servicing, the processor performs the string operation and decrements the count register by one. REP checks ZF if the string operation is a SCAS or CMPS instruction. If the prefix is REPE or REPZ and ZF = 0 (last comparison was not equal), exit the interation and continue with the next instruction. If the prefix is REPNE or REPNZ and ZF = 1 (last comparison was equal), exit the iteration and continue with the next instruction. Otherwise REP checks the count register to start the next iteration. Repeated CMPS and SCAS instructions can be exited if either the count goes to 0 or if ZF fails the repeat condition. You can use either the JCXZ instruction or the conditional jumps that test ZF (the JZ, JNZ, and JNE instructions) to distinguish why iterations stopped.

Flags Affected

ZF is affected by the REP CMPS and REP SCAS as described above.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

Note: *Not all I/O ports can handle the rate at which REP INS and REP OUTS execute. Do not use REP with the LOOP instruction; it yields unpredictable results. The processor ignores REP when it is used with non-string instructions.*

2.200 RET Returns from Procedure

Opcode	Instruction	Clocks	Description
C3	RET	5	Returns near to caller.
CB	RET	13, pm = 18	Returns far to caller at same privilege.
CB	RET	13, pm = 18	Returns far at lesser privilege, switches stacks.
C2 iw	RET imm16	5	Returns near, pops imm16 bytes of parameters.
CA iw	RET imm16	14, pm = 17	Returns far to same privilege, pops imm16 bytes.
CA iw	RET imm16	14, pm = 17	Returns far to lesser privilege, pops imm16 bytes.

Operation

```
IF instruction = near RET
THEN;
   IF OperandSize = 16
   THEN
      lP ← Pop();
      EIP ← EIP AND 0000FFFFh;
   ELSE (* OperandSize = 32 *)
      EIP ← Pop();
   FI;
   IF instruction has immediate operand THEN eSP ← eSP + imm16;
   FI;
FI;

IF (PE = 0 OR (PE = 1 AND VM = 1)) (* Real Mode or Virtual 8086 Mode *)
   AND instruction = far RET
THEN;
   IF OperandSize = 16
   THEN
      lP ← Pop();
      EIP ← EIP AND 0000FFFFh;
      CS ← Pop(); (* 16-bit pop *)
   ELSE (* OperandSize = 32 *)
      EIP ← Pop();
      CS ← Pop(); (* 32-bit pop, high-order 16-bits discarded *)
   FI;
   IF instruction has immediate operand THEN eSP ← eSP + imm16;
   FI;
FI;

IF (PE = 1 AND VM = 0) (* Protected Mode, not V86 Mode *)
   AND instruction = far RET
THEN
   IF OperandSize = 32
   THEN Third word on stack must be within stack limits else Stack Fault;
   ELSE Second word on stack must be within stack limits else Stack Fault;
   FI;
   Return selector RPL is ≥ CPL ELSE Gen. Protection Fault(return selector)
   IF return selector RPL = CPL
   THEN GOTO SAME-LEVEL;
   ELSE GOTO OUTER-PRIVILEGE-LEVEL;
   FI;
FI;
```

```
SAME-LEVEL:
   Return selector must be non-null ELSE General Protection Fault
   Selector index is within limits ELSE General Protection Fault(selector)
   Descriptor AR byte indicates code segment
      ELSE General Protectection Fault(selector)
   IF non-conforming
   THEN code segment DPL must equal CPL;
   ELSE General Protection Fault(selector);
   FI;
   IF conforming
   THEN code segment DPL must be ≤ CPL;
   ELSE General Protection Fault(selector);
   FI;
   Code segment must be present ELSE Segment Not Present(selector);
   Top word on stack must be within stack limits ELSE Stack Fault;
   IP must be in code segment limit ELSE General Protection Fault;
   IF OperandSize = 32
   THEN
      Load CS: EIP from stack
      Load CS register with descriptor
      Increment eSP by 8 plus the immediate offset if it exists
   ELSE (* OperandSize = 16 *)
      Load CS:IP from stack
      Load CS register with descriptor
      Increment eSP by 4 plus the immediate offset if it exists
   FI;

OUTER-PRIVILEGE-LEVEL:
   IF OperandSize = 32
   THEN Top (16 + immediate) bytes on stack must be within stack limits
      ELSE Stack Fault;
   ELSE Top (8 +immediate) bytes on stack must be within stack limits
      ELSE Stack Fault;
   FI;
   Examine return CS selector and associated descriptor:
      Selector must be non-null ELSE General Protection Fault;
      Selector index is within limits ELSE Gen.Protection Fault(selector)
      Descriptor AR byte indicates code segment
         ELSE General Protection Fault(selector);
      IF non-conforming
      THEN code segment DPL must equal return selector RPL
      ELSE General Protection Fault(selector);
      FI;
      IF conforming
      THEN code segment DPL must be ≤ return selector RPL;
      ELSE General Protection Fault(selector);
      FI;
      Segment must be present ELSE Segment Not Present(selector)
   Examine return SS selector and associated descriptor:
      Selector must be non-null ELSE General Protection Fault;
      Selector index is within limits ELSE Gen.Protection Fault (selector);
      Selector RPL = RPL of the return CS selector
         ELSE General Protection Fault(selector);
      Descriptor AR byte indicates a writable data segment
         ELSE General Protection Fault(selector);
      Descriptor DPL = RPL of the return CS selector
         ELSE General Protection Fault(selector);
      Segment must be present ELSE Segment Not Present(selector);
```

```
IP must be in code segment limit ELSE General Protection Fault;
Set CPL to the RPL of the return CS selector;
IF OperandSize = 32
THEN
   Load CS: EIP from stack;
   Set CS RPL to CPL;
   Increment eSP by 8 plus the immediate offset if it exists;
   Load SS:eSP from stack;
ELSE (* OperandSize = 16 *)
   Load CS:IP from stack;
   Set CS RPL to CPL;
   Increment eSP by 4 plus the immediate offset if it exists;
   Load SS:eSP from stack;
FI;
Load the CS register with the return CS descriptor;
Load the SS register with the return SS descriptor;
For each of ES, FS, GS, and DS
DO
   IF the current register setting is not valid for the outer level,
      set the register to null (selector ← AR ← 0);
   To be valid, register setting must satisfy the following properties:
      Selector index must be within descriptor table limits;
      Descriptor AR byte must indicate data or readable code segment;
      IF segment is data or non-conforming code, THEN
               DPL must be ≥ CPL, or DPL must be ≥ RPL;
      FI;
   OD
```

Description

RET transfers control to a return address located on the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction. The optional numeric parameter to the RET instruction gives the number of stack bytes (OperandMode = 16) or words (OperandMode = 32) to be released after the return address is popped. These items are typically used as input parameters to the procedure called. For the intrasegment (near) return, the address on the stack is a segment offset, which is popped into the instruction pointer. The CS register is unchanged.

For the intersegment (far) return, the address on the stack is a long pointer. The offset is popped first, followed by the selector. In Real Mode, the CS and IP registers are loaded directly. In Protected Mode, an intersegment return causes the microprocessor to check the descriptor addressed by the return selector. The AR byte of the descriptor must indicate a code segment of equal or lesser privilege (or greater or equal numeric value) than the current privilege level. Returns to a lesser privilege level cause the stack to be reloaded from the value saved beyond the parameter block.

The DS, ES, FS, and GS segment registers can be cleared by the RET instruction during an interlevel transfer. If these registers refer to segments that cannot be used by the new privilege level, they are cleared to prevent unauthorized access from the new privilege level.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13), Segment Not Present (11), or Stack Fault (12) occur as described under ‘Operation.’ Page Fault (14) indicates a page fault.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault.

2.201 ROL Rotates Left

Opcode	Instruction	Clocks	Description
D0 /0	ROL r/m8,1	3/4	Rotates 8 bits r/m byte left once.
D2 /0	ROL r/8,CL	3/4	Rotates 8 bits r/m byte left CL times.
C0 /0 ib	ROL r/m8,imm8	2/4	Rotates 8 bits r/m byte left imm8 times.
D1 /0	ROL r/m16,1	3/4	Rotates 16 bits r/m word left once.
D3 /0	ROL r/m16,CL	3/4	Rotates 16 bits r/m word left CL times.
C1 /0 ib	ROL r/m16,imm8	2/4	Rotates 16 bits r/m word left imm8 times.
D1 /0	ROL r/m32,1	3/4	Rotates 32 bits r/m doubleword left once.
D3 /0	ROL r/m32,CL	3/4	Rotates 32 bits r/m doubleword left CL times.
C1 /0 ib	ROL r/m32,imm8	2/4	Rotates 32 bits r/m doubleword left imm8 times.

Operation

```
temp C0UNT;
WHILE (temp ≠ 0)
DO
   tmpcf ← high-order bit of (r/m);
   r/m ← r/m ·2 + (tmpcf);
   temp ← temp – 1;
OD;
IF C0UNT = 1
THEN
   IF high-order bit of r/m ≠ CF
   THEN OF ← 1;
   ELSE OF ← 0;FI;
ELSE OF ← undefined;FI
```

Description

ROL shifts the bits upward, except for the top bit, which becomes the bottom bit; ROL also copies the bit to CF. The second operand indicates the number of rotations. The operand is either an immediate number or the CL register contents. The processor does not allow rotation counts greater than 31, using only the bottom five bits of the operand if it is greater than 31. The 486 processor in Virtual 8086 Mode masks rotation counts.

Flags Affected

OF is only defined for single-bit rotations but is undefined otherwise. CF contains the value of the top bit copied into it. SF, ZF, AF, and PF are not affected.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.202 ROR Rotates Right

Opcode	Instruction	Clocks	Description
D0 /1	RCR r/m8,1	3/4	Rotates 8 bits r/m byte right once.
D2 /1	RCR r/8,CL	3/4	Rotates 8 bits r/m byte right CL times.
C0 /1 ib	RCR r/m8,imm8	2/4	Rotates 8 bits r/m byte right imm8 times.
D1 /1	RCR r/m16,1	3/4	Rotates 16 bits r/m word right once.
D3 /1	RCR r/m16,CL	3/4	Rotates 16 bits r/m word right CL times.
C1 /1 ib	RCR r/m16,imm8	2/4	Rotates 16 bits r/m word right imm8 times.
D1 /1	RCR r/m32,1	3/4	Rotates 32 bits r/m doubleword right once.
D3 /1	RCR r/m32,CL	3/4	Rotates 32 bits r/m doubleword right CL times.
C1 /1 ib	RCR r/m32,imm8	2/4	Rotates 32 bits r/m doubleword right imm8 times.

Operation

```
temp ← C0UNT;
WHILE (temp ≠ 0 )
DO
   tmpcf ← low-order bit of (r/m);
   r/m ← r/m / 2 + (tmpcf · 2 width(r/m));
   temp ← temp – 1;
OD;
IF C0UNT = 1
THEN
   IF (high-order bit of r/m) ≠ (bit next to high-order bit of r/m)
   THEN OF ← 1;
   ELSE OF ← 0;FI;
ELSE OF ← undefined FI
```

Description

ROR shifts the bits downward, except for the bottom bit, which becomes the top bit; ROR also copies the bit to CF. The second operand indicates the number of rotations to make. The operand is either an immediate number or the CL register contents. The processor does not allow rotation counts greater than 31, using only the bottom five bits of the operand if it is greater than 31. The 486 processor in Virtual 8086 Mode does mask rotation counts.

Flags Affected

OF is only defined for single-bit rotations but is undefined otherwise. CF contains the value of the top bit copied into it. SF, ZF, AF, and PF are not affected.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.203 SAHF Stores AH into Flags

Opcode	Instruction	Clocks	Description
9E	SAHF	2	Stores AH into EFLAGS bits SF, ZF, AF, PF, CF.

Operation

```
SF:ZF:xx:AF:xx:PF:xx:CF ← AH
```

Description

The SAHF instruction loads the SF, ZF, AF, PF, and CF bits in the EFLAGS register with values from the AH register, from bits 7, 6, 4, 2, and 0, respectively.

Flags Affected

SF, ZF, AF, PF, and CF are loaded with values from the AH register.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.204 SAL Shifts Arithmetic Left

Opcode	Instruction	Clocks	Description
D0 /4	SAL r/m8,1	3/4	Multiplies r/m byte by 2, once.
D2 /4	SAL r/m8,CL	3/4	Multiplies r/m byte by 2, CL times.
C0 /4 ib	SAL r/m8,imm8	2/4	Multiplies r/m byte by 2, imm8 times.
D1 /4	SAL r/m16,1	3/4	Multiplies r/m word by 2, once.
D3 /4	SAL r/m16,CL	3/4	Multiplies r/m word by 2, CL times.
C1 /4 ib	SAL r/m16,imm8	2/4	Multiplies r/m word by 2, imm8 times.
D1 /4	SAL r/m32,1	3/4	Multiplies r/m doubleword by 2, once.
D3 /4	SAL r/m32,CL	3/4	Multiplies r/m doubleword by 2, CL times.
C1 /4 ib	SAL r/m32,imm8	2/4	Multiplies r/m doubleword by 2, imm8 times.

Operation

```
(* C0UNT is the second parameter *)
(temp) ← C0UNT;
WHILE (temp ≠ 0)
DO
   CF ← high-order bit of r/m;
   r/m ← r/m · 2;
   temp ← temp 1 ;
OD;
IF C0UNT = 1
THEN
   OF ← high-order bit of r/m ≠ (CF);
FI
```

Description

SAL (or its synonym, SHL) shifts the bits of the operand upward. SAL shifts the high-order bit into CF and clears the Low order bit. The second operand indicates the number of shifts to make. The operand is either an immediate number or the CL register contents. The processor does not allow shift counts greater than 31; it uses only the bottom five bits of the operand if it is greater than 31.

Flags Affected

OF is defined for single-bit shifts; otherwise, it is undefined. The result determines the CF, ZF, PF, and SF settings.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.205 SAR Shifts Arithmetic Right

Opcode	Instruction	Clocks	Description
D0 /7	SAR r/m8,1	3/4	Performs a signed divide* r/m byte by 2 once.
D2 /7	SAR r/m8,CL	3/4	Performs a signed divide* r/m byte by 2 CL times.
C0 /7 ib	SAR r/m8,imm8	2/4	Performs a signed divide* r/m byte by 2 imm8 times.
D1 /7	SAR r/m16,1	3/4	Performs a signed divide* r/m word by 2 once.
D3 /7	SAR r/m16,CL	3/4	Performs a signed divide* r/m word by 2 CL times.
C1 /7 ib	SAR r/m16,imm8	2/4	Performs a signed divide* r/m word by 2 imm8 times.
D1 /7	SAR r/m32,1	3/4	Performs a signed divide* r/m doubleword by 2 once.
D3 /7	SAR r/m32,CL	3/4	Performs a signed divide* r/m doubleword by 2 CL times.
C1 /7 ib	SAR r/m32,imm8	2/4	Performs a signed divide* r/m doubleword by 2 imm8 times.

*Not the same division as IDIV; rounding is toward negative infinity.

Operation

```
(* C0UNT is the second parameter *)
(temp) ← C0UNT;
WHILE (temp ≠ 0)
DO
   CF ← low-order bit of r/m;
   r/m ← r/m / 2 (* Signed divide, rounding toward negative infinity *);
   temp ← temp 1 ;
OD;
IF C0UNT = 1
THEN
   OF ← 0 FI
```

Description

SAR shifts the bits of the operand downward. SAR shifts the Low order bit into CF. The effect is to divide the operand by two. SAR performs a signed divide with rounding toward negative infinity (not like IDIV); the high-order bit remains the same. The second operand indicates the number of shifts to make. The operand is either an immediate number or the Cl register contents. The processor does not allow shift counts greater than 31; it only uses the bottom five bits of the operand if it is greater than 31.

Flags Affected

OF is cleared for single shifts; otherwise, it is undefined. The result determines the CF, ZF, PF, and SF settings.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.206 SBB Integer Subtract with Borrow

Opcode	Instruction	Clocks	Description
1C ib	SBB AL,imm8	1	Subtracts immediate byte from AL with borrow.
1D iw	SBB AX,imm16	1	Subtracts immediate word from AX with borrow.
1D id	SBB EAX,imm32	1	Subtracts immediate doubleword from EAX with borrow.
80 /3 ib	SBB r/m8,imm8	1/3	Subtracts immediate byte from r/m byte with borrow.
81 /3 iw	SBB r/m16,imm16	1/3	Subtracts immediate word from r/m word with borrow.
81 /3 id	SBB r/m32,imm32	1/3	Subtracts imm. doubleword from r/m doubleword with borrow.
83 /3 ib	SBB r/m16,imm8	1/3	Subtracts sign-extended imm. byte from r/m word with borrow.
83 /3 ib	SBB r/m32,imm8	1/3	Subtracts sign-ext. imm. byte from r/m doubleword with borrow.
18 /r	SBB r/m8,r8	1/3	Subtracts byte register from r/m byte with borrow.
19 /r	SBB r/m16,r16	1/3	Subtracts word register from r/m word with borrow.
19 /r	SBB r/m32,r32	1/3	Subtracts doubleword register from r/m doubleword with borrow.
1A /r	SBB r8,r/m8	1/3	Subtracts r/m byte from byte register with borrow.
1B /r	SBB r16,r/m16	1/2	Subtracts r/m word from word register with borrow.
1B /r	SBB r32,r/m32	1/2	Subtracts r/m doubleword from doubleword register with borrow.

Operation

```
IF SRC is a byte and DEST is a word or doubleword
THEN DEST = DEST – (SignExtend(SRC) + CF)
ELSE DEST ← DEST – (SRC + CF)
```

Description

The SBB instruction adds the second operand (SRC) to CF and subtracts the result from the first operand (DEST). The result of the subtraction is assigned to the first operand (DEST) and the flags are set accordingly.

Note: *When an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended.*

Flags Affected

OF, SF, ZF, AF, PF, and CF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.207 SCAS/SCASB/SCASD/SCASW Compares String Data

Opcode	Instruction	Clocks	Description
AE	SCAS m8	6	Compares bytes AL–ES:DI, updates (E)DI.
AF	SCAS m16	6	Compares words AX–ES:DI, updates (E)DI.
AF	SCAS m32	6	Compares doublewords EAX–ES:DI, updates (E)DI.
AE	SCASB	6	Compares bytes AL–ES:DI, updates (E)DI.
AF	SCASD	6	Compares doublewords EAX–ES:DI, updates (E)DI.
AF	SCASW	6	Compares words AX–ES:DI, updates (E)DI.

Operation

```
IF AddressSize = 16
THEN use DI for dest-index;
ELSE (* AddressSize = 32 *) use EDI for dest-index;
FI;
IF byte type of instruction
THEN
   AL – [dest-index]; (* Compare byte in AL and dest *)
   IF DF = 0 THEN IndDec ← 1 ELSE ← IncDec – 1 ;
   FI;
ELSE
   IF OperandSize = 16
   THEN
      AX – [dest-index] ; (* compare word in AL and dest *)
      IF DF = 0 THEN IncDec ← 2 ELSE IncDec ← –2;
      FI;
   ELSE (* OperandSize = 32 *)
      EAX – [dest-index];(* compare doubleword in EAX & dest *)
      IF DF = 0 THEN IncDec ← 4 ELSE IncDec ← –4;
      FI;
   FI;
FI;
dest-index = dest-index + IncDec
```

Description

SCAS subtracts the memory byte, word, or doubleword at the destination register from the AL, AX, or EAX register. The result is discarded; only the flags are set. The operand must be addressable from the ES segment; no segment override is possible. The address size determines whether the index register is DI (16-bit address) or EDI (32-bit address). The contents of the destination register determine the address of the memory data being compared, not the SCAS instruction operand. The operand validates ES segment addressability and determines the data type. Load the correct index value into the DI or EDI register before executing the SCAS instruction.

After the comparison, the destination index register automatically updates. If the Direction Flag (DF) is 0 (see CLD), the destination index register increments; if DF is 1 (see STD), it decrements. The increment/decrement rate is 1 for bytes, 2 for words, or by 4 for doublewords.

The SCASB, SCASW, and SCASD instructions are synonyms for the byte, word, and doubleword SCAS instructions that do not require operands. They are simpler to code, but provide no type or segment checking.

You can precede SCAS with the REPE or REPNE prefix for a block search of CX or ECX bytes or words.

Flags Affected

OF, SF, ZF, AF, PF, and CF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates that there is an illegal memory-operand effective address in the ES segment. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.208 SETcc Sets Byte on Condition (see list below)

Opcode	Instruction	Clocks	Description
0F 97	SETA r/m8	4/3	Sets byte if above (CF = 0 and ZF = 0).
0F 93	SETAE r/m8	4/3	Sets byte if above or equal (CF = 0).
0F 92	SETB r/m8	4/3	Sets byte if below (CF = 1).
0F 96	SETBE r/m8	4/3	Sets byte if below or equal (CF = 1 or ZF = 1).
0F 92	SETC r/m8	4/3	Sets if carry (CF = 1).
0F 94	SETE r/m8	4/3	Sets byte if equal (ZF = 1).
0F 9F	SETG r/m8	4/3	Sets byte if greater (ZF = 0 and SF = OF).
0F 9D	SETGE r/m8	4/3	Sets byte if greater or equal (SF = OF).
0F 9C	SETL r/m8	4/3	Sets byte if less (SF≠OF).
0F 9E	SETLE r/m8	4/3	Sets byte if less or equal (ZF = 1 or SF≠OF).
0F 96	SETNA r/m8	4/3	Sets byte if not above (CF = 1 or ZF = 1).
0F 92	SETNAE r/m8	4/3	Sets byte if not above or equal (CF = 1).
0F 93	SETNB r/m 8	4/3	Sets byte if not below (CF = 0).
0F 97	SETNBE r/m8	4/3	Sets byte if not below or equal (CF = 0 and ZF = 0).
0F 93	SETNC r/m8	4/3	Sets byte if not carry (CF = 0).
0F 95	SETNE r/m8	4/3	Sets byte if not equal (ZF = 0).
0F 9E	SETNG r/m8	4/3	Sets byte if not greater (ZF = 1 or SF≠OF).
0F 9C	SETNEG r/m8	4/3	Sets byte if not greater or equal (SF≠OF).
0F 9D	SETNL r/m8	4/3	Sets byte if not less (SF = OF).
0F 9F	SETNLE r/m8	4/3	Sets byte if not less or equal (ZF = 0 and SF = OF).
0F 91	SETNO r/m8	4/3	Sets byte if not overflow (OF = 0).
0F 9B	SETNP r/m8	4/3	Sets byte if not parity (PF = 0).
0F 99	SETNS r/m8	4/3	Sets byte if not sign (SF = 0).
0F 95	SETNZ r/m8	4/3	Sets byte if not zero (ZF = 0).
0F 90	SETO r/m8	4/3	Sets byte if overflow (OF = 1).
0F 9A	SETP r/m8	4/3	Sets byte if parity (PF = 1).
0F 9A	SETPE r/m8	4/3	Sets byte if parity even (PF = 1).
0F 9B	SETPO r/m8	4/3	Sets byte if parity odd (PF = 0).
0F 98	SETS r/m8	4/3	Sets byte if sign (SF = 1).
0F 94	SETZ r/m8	4/3	Sets byte if zero (ZF = 1).

Operation

```
IF condition THEN r/m8 ← 1 ELSE r/m8 ← 0; FI
```

Description

SETcc loads a 1 (condition met) or a 0 (not met) into the r/m byte specified by the operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space.

Virtual 8086 Mode Exceptions

Same as Real Mode. Page Fault (14) indicates a page fault.

2.209 SGDT Store Global Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /0	SGDT m	10	Store GDTR to m

Operation

```
DEST ← 48-bit BASE/LIMIT register contents
```

Description

SGDT copies the contents of the descriptor table register to the six bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 16 bits, the next three bytes are assigned to the BASE field of the register and the fourth byte is undefined. Otherwise, if the operand-size attribute is 32 bits, the next four bytes are assigned to the 32-bit BASE field of the register.

Note: *The SGDT instruction is used only in operating system software. It is not used in application programs.*

Flags Affected

None

Protected Mode Exceptions

Invalid Opcode (6) indicates the destination operand is a register. General Protection Fault (13) indicates either that the destination is in a non-writable segment or there is an illegal memory-operand effective address in the CS, DS, ES, FS, or GS segments. Stack Fault (12) indicates an illegal address is in the SS segment. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Invalid Opcode (6) indicates the destination operand is a register. General Protection Fault (13) indicates that part of the operand lies outside of the effective address space from 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) indicates the destination operand is a register. General Protection Fault (13) indicates that part of the operand lies outside of the effective address space from 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *The 16-bit forms of the SGDT instructions are compatible with the 286 processor if the value in the upper eight bits is not referenced. The 286 processor stores a 1 in each of the upper bits, whereas Am386 and Am486 microprocessors store a 0 if the operand-size attribute is 16 bits.*

2.210 SHL Shift Left

Opcode	Instruction	Clocks	Description
D0 /4	SHL r/m8,1	3/4	Multiplies r/m byte by 2 once.
D2 /4	SHL r/m8,CL	3/4	Multiplies r/m byte by 2 CL times.
C0 /4 ib	SHL r/m8,imm8	2/4	Multiplies r/m byte by 2 imm8 times.
D1 /4	SHL r/m16,1	3/4	Multiplies r/m word by 2 once.
D3 /4	SHL r/m16,CL	3/4	Multiplies r/m word by 2 CL times.
C1 /4 ib	SHL r/m16,imm8	2/4	Multiplies r/m word by 2 imm8 times.
D1 /4	SHL r/m32,1	3/4	Multiplies r/m doubleword by 2 once.
D3 /4	SHL r/m32,CL	3/4	Multiplies r/m doubleword by 2 CL times.
C1 /4 ib	SHL r/m32,imm8	2/4	Multiplies r/m doubleword by 2 imm8 times.

Operation

```
(* C0UNT is the second parameter *)
(temp) ← C0UNT;
WHILE (temp ≠ 0)
DO
   CF ← high-order bit of r/m;
   r/m ← r/m · 2;
   temp ← temp 1;
OD;
IF C0UNT = 1
THEN
   OF ← high-order bit of r/m ≠ (CF);
FI
```

Description

SHL (or its synonym, SAL) shifts the bits of the operand upward. SHL shifts the high-order bit into CF and clears the Low order bit. The second operand indicates the number of shifts to make. The operand is either an immediate number or the CL register contents. The processor does not allow shift counts greater than 31; it uses only the bottom five bits of the operand if it is greater than 31.

Flags Affected

OF is defined for single-bit shifts; otherwise, it is undefined. The result determines the CF, ZF, PF, and SF settings. CF is undefined if the shift lengths are greater than the size of the shifted operand.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.211 SHLD Double Precision Shift Left

Opcode	Instruction	Clocks	Description
0F A4	SHLD r/m16,r16,imm8	2/3	r/m16 gets SHL of r/m16 concatenated with r16.
0F A4	SHLD r/m32,r32,imm8	2/3	r/m32 gets SHL of r/m32 concatenated with r32.
0F A5	SHLD r/m16,r16,CL	3/4	r/m16 gets SHL of r/m16 concatenated with r16.
0F A5	SHLD r/m32,r32,CL	3/4	r/m32 gets SHL of r/m32 concatenated with r32.

Operation

```
(* count is an unsigned integer corresponding to the last operand of the
instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt = 0
THEN no operation
ELSE
   IF ShiftAmt ≥ OperandSize
   THEN (* Bad parameters *)
      r/m ← UNDEFINED;
      CF, OF, SF, ZF, AF, PF ← UNDEFINED;
   ELSE (* Perform the shift *)
      CF ← BIT[Base, OperandSize – ShiftAmt];
         (* Last bit shifted out on exit *)
      FOR i ← OperandSize – 1 DOWNTO ShiftAmt
      DO
         BIT[Base, i] ← BIT[Base, i – ShiftAmt];
      OF;
      FOR i ← ShiftAmt – 1 DOWNTO 0
      DO
         BIT[Base, i] ← BIT[inBits, i – ShiftAmt + OperandSize];
      OD;
      Set SF, ZF, PF (r/m);
         (* SF, ZF, PF are set according to the value of the result *)
      AF ← UNDEFINED;
   FI;
FI
```

Description

SHLD shifts the r/m word/doubleword specified by the first operand to the left as many bits as indicated by the count operand, specified by an immediate byte or the CL register. The second operand word/doubleword register (r16/ r32) provides the bits to shift in from the right (starting with bit 0). SHLD then stores the result back into the r/m word/doubleword specified by the first operand. The register remains unaltered.

The count operand is taken modulo 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified registers, the operation is useful for multiprecision shifts (64 bits or more).

Flags Affected

SF, ZF, and PF are set according to the result. CF is set to the value of the last bit shifted out. OF is valid for a shift of one bit position only: 0 = no sign change occurred; 1 = sign change occurred; for a multibit shift, OF is undefined. AF is undefined.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.212 SHR Shift Right

Opcode	Instruction	Clocks	Description
D0 /5	SHR r/m8,1	3/4	Performs unsigned divide r/m byte by 2 once.
D2 /5	SHR r/m8,CL	3/4	Performs unsigned divide r/m byte by 2 CL times.
C0 /5 ib	SHR r/m8,imm8	2/4	Performs unsigned divide r/m byte by 2 imm8 times.
D1 /5	SHR r/m16,1	3/4	Performs unsigned divide r/m word by 2 once.
D3 /5	SHR r/m16,CL	3/4	Performs unsigned divide r/m word by 2 CL times.
C1 /5 ib	SHR r/m16,imm8	2/4	Performs unsigned divide r/m word by 2 imm8 times.
D1 /5	SHR r/m32,1	3/4	Performs unsigned divide r/m doubleword by 2 once.
D3 /5	SHR r/m32,CL	3/4	Performs unsigned divide r/m doubleword by 2 CL times.
C1 /5 ib	SHR r/m32,imm8	2/4	Performs unsigned divide r/m doubleword by 2 imm8 times.

Operation

```
(* C0UNT is the second parameter *)
(temp) ← C0UNT;
WHILE (temp ≠ 0)
DO
   CF ← low-order bit of r/m;
   r/m ← r/m / 2; (* Unsigned divide *);
   temp ← temp 1 ;
OD;
    OF ← high-order bit of operand;
FI
```

Description

SHR shifts the bits of the operand downward. SHR shifts the Low order bit into CF. The effect is to divide the operand by 2. SHR performs an unsigned divide and clears the high-order bit. The second operand indicates the number of shifts to make. The operand is either an immediate number or the CL register contents. The processor does not allow shift counts greater than 31; it only uses the bottom five bits of the operand if it is greater than 31.

Flags Affected

OF is set to the high-order bit of the original operand. The result determines the CF, ZF, PF, and SF settings. CF is undefined if the shift lengths are greater than the size of the shifted operand.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.213 SHRD Double Precision Shift Right

Opcode	Instruction	Clocks	Description
0F AC	SHRD r/m16,r16,imm8	2/3	r/m16 gets SHR of r/m16 concatenated with r16.
0F AC	SHRD r/m32,r32,imm8	2/3	r/m32 gets SHR of r/m32 concatenated wtih r32.
0F AD	SHRD r/m16,r16,CL	3/4	r/m16 gets SHR of r/m16 concatenated with r16.
0F AD	SHRD r/m32,r32,CL	3/4	r/m32 gets SHR of r/m32 concatenated with r32.

Operation

```
(* count is an unsigned integer corresponding to the last operand of the
instruction, either an immediate byte or the byte in register CL *)
ShiftAmt ← count MOD 32;
inBits ← register; (* Allow overlapped operands *)
IF ShiftAmt  =  0
THEN no operation
ELSE
   IF ShiftAmt _ OperandSize
   THEN (* Bad parameters *)
      r/m ← UNDEFINED;
      CF, OF, SF, ZF, AF, PF ← UNDEFINED;
   ELSE (* Perform the shift *)
      CF ← BIT[r/m, Shift – 1 ]; (* last bit shifted out on exit *)
      FOR i ← 0 TO OperandSize – 1 – ShiftAmt
      DO
         BIT[r/m, i] ← BIT[r/m, 1 – ShiftAmt];
      OD;
      FOR i ← OperandSize – ShiftAmt TO OperandSize – 1
      DO;
         BIT[r/m,i] ← BIT[inBits,i +ShiftAmt – OperandSize];
      OD;
         (* SF, ZF, PF are set according to the value of the result *)
      Set SF, ZF, PF (r/m);
      AF ← UNDEFINED;
   FI;
FI
```

Description

SHRD shifts the r/m word/doubleword specified by the first operand to the right as many bits as indicated by the count operand, specified by an immediate byte or the CL register. The second operand word/doubleword register (r16/ r32) provides the bits to shift in from the left (starting with bit 31). SHRD then stores the result back into the r/m word/doubleword specified by the first operand. The register remains unaltered.

The count operand is taken modulo 32 to provide a number between 0 and 31 by which to shift. Because the bits to shift are provided by the specified registers, the operation is useful for multiprecision shifts (64 bits or more).

Flags Affected

SF, ZF, and PF are set according to the result. CF is set to the value of the last bit shifted out. OF is valid for a shift of one bit position only: 0 = no sign change occurred; 1 = sign changed occurred; for a multibit shift, OF is undefined. AF is undefined.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.214 SIDT Stores Interrupt Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 01 /1	SIDT m	10	Stores IDTR to m.

Operation

```
DEST ← 48-bit BASE/LIMIT register contents
```

Description

The SIDT instruction copies the contents of the descriptor table register to the 6 bytes of memory indicated by the operand. The LIMIT field of the register is assigned to the first word at the effective address. If the operand-size attribute is 16 bits, the next 3 bytes are assigned the BASE field of the register and the fourth byte is undefined. Otherwise, if the operand-size attribute is 32 bits, the next 4 bytes are assigned the 32-bit BASE field of the register.

SIDT is only used in operating system software. It should not be used in application programs.

Flags Affected

None

Protected Mode Exceptions

Invalid Opcode (6) indicates the destination operand is a register. General Protection Fault (13) indicates the destination is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. Alignment Check (17) indicates an unaligned memory reference if the current privilege level is 3.

Real Address Mode Exceptions

Invalid Opcode (6) indicates the destination operand is a register. General Protection Fault (13) indicates that part of the operand is referenced outside the effective address space from 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) indicates the destination operand is a register. General Protection Fault (13) indicates that part of the operand is referenced outside the effective address space from 0 to 0FFFFh. Page Fault (14) indicates a page fault. Alignment Check (17) indicates an unaligned memory reference if the current privilege level is 3.

Note: *The 16-bit forms of the SIDT instructions are compatible with the 286 processor if the value in the upper eight bits is not referenced. The 286 processor stores a 1 in each of the upper bits, whereas Am386 and Am486 microprocessors store a 0 if the operand-size attribute is 16 bits.*

2.215 SLDT Stores Local Descriptor Table Register

Opcode	Instruction	Clocks	Description
0F 00 /0	SLDT r/m16	2/3	Stores LDTR to EA word.

Operation

```
r/m16 ← LDTR
```

Description

The SLDT instruction stores the Local Descriptor Table Register (LDTR) in the 2-byte register or memory location indicated by the effective address operand. This register is a selector that points into the Global Descriptor Table.

Note: *The SLDT instruction is used only in operating system software. It is not used in application programs.*

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Invalid Opcode (6) occurs. SLDT is not recognized in Real Address Mode.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) occurs. SLDT is not recognized in Virtual 8086 Mode. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *The operand-size attribute has no effect on the operation of the instruction.*

2.216 SMSW Stores Machine Status Word

Opcode	Instruction	Clocks	Description
0F 01 /4	SMSW r/m16	2/3	Stores machine status word to EA word.

Operation

```
r/m16 ← MSW
```

Description

The SMSW instruction stores the machine status word (part of the CR0 register) in the 2-byte register or memory location indicated by the effective address operand.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *This instruction is provided for compatibility with the 80286 microprocessor; programs for the Am486 microprocessor should use the MOV ..., CR0 instruction.*

2.217 STC Sets Carry Flag

Opcode	Instruction	Clocks	Description
F9	STC	2	Sets Carry Flag.

Operation

```
CF ← 1
```

Description

The STC instruction sets CF.

Flags Affected

CF is set.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.218 STD Sets Direction Flag

Opcode	Instruction	Clocks	Description
FD	STD	2	Sets Direction Flag to make the Stack Index (SI or ESI) and/or the Data Index (DI or EDI) Registers decrement.

Operation

```
DF ← 1
```

Description

The STD instruction sets the Direction Flag, causing all subsequent string operations to decrement the index registers on which they operate: SI (8-bit or 16-bit address) or ESI (32-bit address), and/or DI (8-bit or 16-bit address) or EDI (32-bit address).

Flags Affected

DF is set. No other flags or registers are affected.

Protected Mode Exceptions

None

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

None

2.219 STI Sets Interrupt-Enable Flag

Opcode	Instruction	Clocks	Description
FB	STI	5	SetsInterrupt-enable Flag to enable interrupts at the end of the next instruction.

Operation

```
IF ← 1
```

Description

STI sets the Interrupt-enable Flag (IF). The processor responds to external interrupts after executing the next instruction if that instruction does not clear IF. If external interrupts are disabled and the program executes STI before a RET instruction (such as at the end of a subroutine), RET executes before processing any external interrupts. If external interrupts are disabled and the program executes STI before a CLI instruction, no external interrupts are processed because CLI clears IF.

Flags Affected

IF is set.

Protected Mode Exceptions

General Protection Fault (13) indicates the current privilege level is greater (has less privilege) than the I/O privilege level.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates the current privilege level is greater (has less privilege) than the I/O privilege level.

Note: *If an NMI, trap, or fault occurs following STI, the interrupt will be processed before executing the next sequential instruction in the code.*

2.220 STOS/STOSB/STOSD/STOSW Stores String Data

Opcode	Instruction	Clocks	Description
AA	STOS m8	5	Stores AL in byte ES:(E)DI, update (E)DI.
AB	STOS m16	5	Stores AX in word ES:(E)DI, update (E)DI.
AB	STOS m32	5	Stores EAX in doubleword ES:(E)DI, update (E)DI.
AA	STOSB	5	Stores AL in byte ES:(E)DI, update (E)DI.
AB	STOSD	5	Stores EAX in doubleword ES:(E)DI, update (E)DI.
AB	STOSW	5	Stores AX in word ES:(E)DI, update (E)DI.

Operation

```
IF AddressSize = 16
THEN use ES:DI for DestReg
ELSE (* AddressSize = 32 *) use ES:EDI for DestReg;
FI;
IF byte type of instruction
THEN
   (ES:DestReg) ← AL,
   IF DF = 0
   THEN DestReg ← DestReg + 1;
   ELSE DestReg ← DestReg – 1;
   FI;
ELSE IF OperandSize = 16
   THEN (ES:DestReg) ← AX;
      IF DF = 0
      THEN DestReg ← DestReg + 2;
      ELSE DestReg ← DestReg – 2;
      FI;
   ELSE (* OperandSize = 32 *)
      (ES:DestReg) ← EAX;
      IF DF = 0
      THEN DestReg ← DestReg + 4;
      ELSE DestReg ← DestReg – 4;
      FI;
   FI;
FI
```

Description

STOS transfers the contents of the AL, AX, or EAX register to the memory byte, word, or doubleword given by the destination register (DI for 16-bit addresses, EDI for 32-bit addresses) relative to the ES segment. The destination operand must be addressable from the ES register. A segment override is not possible. The contents of the destination register determine the destination address. STOS does not use an explicit operand. This operand only validates ES segment addressability and determines the data type. You must load the correct index value into the destination register before executing the STOS instruction.

After the transfer, STOS automatically updates the Data Index (DI or EDI) register. If the Direction Flag (DF) is 0 (see CLD), the register increments; if DF is 1 (see STD), the register decrements. The increment/decrement rate is 1 for a byte, 2 for a word, or 4 for a doubleword.

STOSB, STOSW, and STOSD are synonyms for the byte, word, and doubleword STOS instructions. These forms do not require an operand and are simpler to use, but provide no type or segment checking.

You can precede STOS with the REP prefix for a block fill of CX or ECX bytes, words, or doublewords.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the ES segment. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.221 STR Stores Task Register

Opcode	Instruction	Clocks	Description
0F 00 /1	STR r/m16	2/3	Stores task register to EA word.

Operation

```
r/m ← task register
```

Description

The contents of the task register are copied to the 2-byte register or memory location indicated by the effective address operand.

Note: *The STR instruction is used only in operating system software. It is not used in application programs.*

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Invalid Opcode (6) occurs. STR is not recognized in Real Address Mode.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) occurs. STR is not recognized in Virtual 8086 Mode.

Note: *The operand-size attribute has no effect on this instruction.*

2.222 SUB Integer Subtraction

Opcode	Instruction	Clocks	Description
2C ib	SUB AL,imm8	1	Subtracts immediate byte from AL.
2D iw	SUB AX,imm16	1	Subtracts immediate word from AX.
2D id	SUB EAX,imm32	1	Subtracts immediate doubleword from EAX.
80 /5 ib	SUB r/m8,imm8	1/3	Subtracts immediate byte from r/m byte.
81 /5 iw	SUB r/m16,imm16	1/3	Subtracts immediate word from r/m word.
81 /5 id	SUB r/m32,imm32	1/3	Subtracts immediate doubleword from r/m doubleword.
83 /5 ib	SUB r/m16,imm8	1/3	Subtracts sign-ext. immediate byte from r/m word.
83 /5 ib	SUB r/m32,imm8	1/3	Subtracts sign-ext. immediate byte from r/m doubleword.
28 /r	SUB r/m8,r8	1/3	Subtracts byte register from r/m byte.
29 /r	SUB r/m16,r16	1/3	Subtracts word register from r/m word.
29 /r	SUB r/m32,r32	1/3	Subtracts doubleword register from r/m doubleword.
2A /r	SUB r8,r/m8	1/2	Subtracts r/m byte from byte register.
2B /r	SUB r16,r/m16	1/2	Subtracts r/m word from word register.
2B /r	SUB r32,r/m32	1/2	Subtracts r/m doubleword from doubleword register.

Operation

```
IF SRC is a byte and DEST is a word or doubleword
THEN DEST = DEST – SignExtend(SRC);
ELSE DEST ← DEST – SRC;
FI
```

Description

The SUB instruction subtracts the second operand (SRC) from the first operand (DEST). The first operand is assigned the result of the subtraction and the flags are set accordingly. If an immediate byte value is subtracted from a word operand, the immediate value is first sign-extended to the size of the destination operand.

Flags Affected

OF, SF, ZF, AF, PF, and CF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.223 TEST Logical Compare

Opcode	Instruction	Clocks	Description
A8 ib	TEST AL,imm8	1	AND immediate byte with AL
A9 iw	TEST AX,imm16	1	AND immediate word with AX
A9 id	TEST EAX,imm32	1	AND immediate doubleword with EAX
F6 /0 ib	TEST r/m8,imm8	1/2	AND immediate byte with r/m byte
F7 /0 iw	TEST r/m16,imm16	1/2	AND immediate word with r/m word
F7 /0 id	TEST r/m32,imm32	1/2	AND immediate doubleword with r/m doubleword
84 /r	TEST r/m8,r8	1/2	AND byte register with r/m byte
85 /r	TEST r/m16,r16	1/2	AND word register with r/m word
85 /r	TEST r/m32,r32	1/2	AND doubleword register with r/m doubleword

Operation

```
DEST : = LeftSRC AND RightSRC;
CF ← 0;
OF ← 0
```

Description

The TEST instruction computes the bit-wise logical AND of its two operands. Each bit of the result is 1 if both of the corresponding bits of the operands are 1; otherwise, each bit is 0. The result of the operation is discarded and only the flags are modified.

Flags Affected

OF and CF are cleared; SF, ZF, and PF are set according to the result.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.224 VERR/VERW Verifies Segment for Read/Write

Opcode	Instruction	Clocks	Description
0F 00 /4	VERR r/m16	11/11	Sets ZF = 1 if segment readable, selector in r/m16.
0F 00 /5	VERW r/m16	11/11	Sets ZF = 1 if segment writable, selector in r/m16.

Operation

```
IF segment with selector at (r/m) is accessible
   with current protection level
   AND ((segment is readable for VERR) OR
      (segment is writable for VERW))
THEN ZF ← 1;
ELSE ZF ← 0;
FI
```

Description

The VERR and VERW r/m word operand contains the selector value. The instructions determine whether the segment pointed to by the selector is accessible from the current privilege level, and, if it is readable (VERR) or writable (VERW). If the segment is accessible and usable, the processor sets the Zero Flag (ZF); if the segment is not accessible or usable, ZF is cleared. The following conditions must be met to set ZF:

- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT); the selector must be "defined."
- The selector must denote a code or data segment descriptor (not a task state segment, LDT, or gate).
- For VERR , the segment must be readable. For VERW, the segment must be a writable data segment.
- If the code segment is usable and conforming, the descriptor privilege level (DPL) can be any value for the VERR instruction. Otherwise, the DPL must be greater than or equal to (have less or the same privilege as) both the current privilege level and the selector's RPL.

Validation is the same as that used for reading/writing segments loaded into the DS, ES, FS, or GS register. ZF stores the validation result. The selector's value cannot cause a protection exception that would cause the software to anticipate segment access problems.

Flags Affected

ZF is set if the segment is accessible, and cleared if it is not.

Protected Mode Exceptions

No faults attributable to the selector operand are generated. General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

Invalid Opcode (6) occurs. VERR and VERW are not recognized in Real Address Mode.

Virtual 8086 Mode Exceptions

Invalid Opcode (6) occurs. VERR and VERW are not recognized in Virtual 8086 Mode.

2.225 WAIT Wait

Opcode	Instruction	Clocks	Description
9B	WAIT	1–3	Causes processor to check for numeric exceptions.

Description

WAIT causes the microprocessor to check for pending unmasked numeric exceptions before proceeding.

Flags Affected

None

Protected Mode Exceptions

Coprocessor Not Available (7) occurs if both MP and TS in CR0 are set.

Real Address Mode Exceptions

Coprocessor Not Available (7) occurs if both MP and TS in CR0 are set.

Virtual 8086 Mode Exceptions

Coprocessor Not Available (7) occurs if both MP and TS in CR0 are set.

Note: *Coding WAIT after an ESC instruction ensures that any unmasked floating-point exceptions the instruction may cause are handled before the microprocessor has a chance to modify the instruction’s results. FWAIT is an alternate mnemonic for WAIT.*

2.226 WBINVD Writes Back and Invalidates Cache

Opcode	Instruction	Clocks	Description
0F 09	WBINVD	5	Invalidates entire cache thereby causing the external cache to write its contents back to memory and then flush itself.

Operation

```
FLUSH INTERNAL CACHE
SIGNAL EXTERNAL CACHE TO WRITE-BACK
SIGNAL EXTERNAL CACHE TO FWSH
```

Description

The internal cache is flushed and a special-function bus cycle is issued to cause the external cache to write its contents to main memory. Another special-function bus cycle follows, directing the external cache to flush itself.

Flags Affected

None

Protected Mode Exceptions

The WBINVD instruction is a privileged instruction; General Protection Fault (13) indicates the current privilege level is not 0.

Real Address Mode Exceptions

None

Virtual 8086 Mode Exceptions

General Protection Fault (13) occurs. WBINVD instruction is a privileged instruction.

Note: *This instruction is implementation-dependent; its function may be implemented differently on future AMD microprocessors. Hardware designers should ensure that their systems respond to the external cache write-back and flush indications. This instruction is not supported by 386 microprocessors.*

2.227 XADD Exchanges and Adds

Opcode	Instruction	Clocks	Description
0F C0 /r	XADD r/m8,r8	4	Exchanges byte register and r/m byte; loads sum into r/m byte.
0F C1 /r	XADD r/m16,r16	4	Exchanges word register and r/m word; loads sum into r/m word.
0F C1 /r	XADD r/m32,r32	4	Exchanges doubleword register and r/m doubleword; loads sum into r/m doubleword.

Operation

```
TEMP ← SRC + DEST
SRC ← DEST
DEST ← TEMP
```

Description

The XADD instruction loads DEST into SRC and then loads the sum of DEST and the original value of SRC into DEST.

Flags Affected

CF, PF, AF, SF, ZF, and OF are affected as if an ADD instruction had been executed.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *You can use a LOCK prefix with this instruction. You cannot use this instruction with 386 microprocessors.*

2.228 XCHG Exchange

Opcode	Instruction	Clocks	Description
90 + r	XCHG AX,r16	3	Exchanges word register with AX.
90 + r	XCHG r16,AX	3	Exchanges AX with word register.
90 + r	XCHG EAX,r32	3	Exchanges doubleword register with EAX.
90 + r	XCHG r32,EAX	3	Exchanges EAX with doubleword register.
86 /r	XCHG r/m8,r8	3/5	Exchanges byte register with r/m byte.
86 /r	XCHG r8,r/m8	3/5	Exchanges r/m byte with byte register.
87 /r	XCHG r/m16,r16	3/5	Exchanges word register with r/m word.
87 /r	XCHG r16,r/m16	3/5	Exchanges r/m word with word register.
87 /r	XCHG r/m32,r32	3/5	Exchanges doubleword register with r/m doubleword.
87 /r	XCHG r32,r/m32	3/5	Exchanges r/m doubleword with doubleword register.

Operation

```
temp ← DEST
DEST ← SRC
SRC ← temp
```

Description

The XCHG instruction exchanges two operands. The operands can be in either order. If a memory operand is involved, the $\overline{\text{LOCK}}$ signal is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Note: *For 16-bit data, you can use XCHG instead of BSWAP.*

2.229 XLAT/XLATB Table Look-Up Translation

Opcode	Instruction	Clocks	Description
D7	XLAT m8	4	Sets AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	4	Sets AL to memory byte DS:[(E)BX + unsigned AL].

Operation

```
IF AddressSize = 16
THEN
   AL ← (BX + ZeroExtend (AL))
ELSE (* AddressSize = 32 *)
   AL ← (EBX + ZeroExtend (AL));
FI
```

Description

XLAT changes the AL register from the table index to the table entry. The AL register should be an unsigned index into a table addressed by the DS:BX register pair (for a 16-bit address) or the DS:EBX register pair (for a 32-bit address).

The XLAT operand allows for the possibility of a segment override, but the instruction uses the contents of the BX register even if they differ from the offset of the operand. Load the operand offset into the (E)BX register and the table index into AL before executing XLAT.

Use the no-operand form, XLATB, if the table referenced by (E)BX resides in the DS segment.

Flags Affected

None

Protected Mode Exceptions

General Protection Fault (13) indicates an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

2.230 XOR Logical Exclusive OR

Opcode	Instruction	Clocks	Description
34 ib	XOR AL, imm8	1	XOR immediate byte to AL
35 iw	XOR AX, imm16	1	XOR immediate word to AX
35 id	XOR EAX, imm32	1	XOR immediate doubleword to EAX
80 /6 ib	XOR r/m8, imm8	1/3	XOR immediate byte to r/m byte
81 /6 iw	XOR r/m16, imm16	1/3	XOR immediate word to r/m word
81 /6 id	XOR r/m32, imm32	1/3	XOR immediate doubleword to r/m doubleword
83 /6 ib	XOR r/m16, imm8	1/3	XOR sign-extended immediate bye with r/m word
83 /6 ib	XOR r/m32, imm8	1/3	XOR sign-extended immediate byte with r/m doubleword
30 /r	XOR r/m8, r8	1/3	XOR byte register to r/m byte
31 /r	XOR r/m16, r16	1/3	XOR word register to r/m word
31 /r	XOR r/m32, r32	1/3	XOR doubleword register to r/m doubleword
32 /r	XOR r8, r/m8	1/2	XOR r/m byte to byte register
33 /r	XOR r16, r/m16	1/2	XOR r/m word to word register
33 /r	XOR r32, r/m32	1/2	XOR r/m doubleword to doubleword register

Operation

```
DEST ← LeftSRC XOR RightSRC
CF ← 0
OF ← 0
```

Description

XOR computes the exclusive OR of the two operands. If corresponding bits of the operands are different, the resulting bit is 1. If the bits are the same, the result is 0. The answer replaces the first operand.

Flags Affected

XOR clears CF and OF. The result sets or resets SF, ZF, and PF as required. XOR does not affect AF.

Protected Mode Exceptions

General Protection Fault (13) indicates either that the result is in a non-writable segment or there is an illegal memory-operand effective address in the code or data segments. Stack Fault (12) indicates an illegal SS segment address. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

Real Address Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh.

Virtual 8086 Mode Exceptions

General Protection Fault (13) indicates that part of the operand lies outside the effective address space: 0 to 0FFFFh. Page Fault (14) indicates a page fault. If CPL is 3, Alignment Check (17) indicates there is an unaligned memory reference.

APPENDIX

A GENERAL GUIDELINES FOR PROGRAMMING

A.1 GENERAL

An Am486 microprocessor communicates with the outside world through programming. If you look at a description of its pinouts, you discover that the interface itself is not very complex. The major lines of two-way communication are the 32 data lines (D31–D0), the 30 address lines (A31–A2), the four byte enable lines ($\overline{\text{BE3}}$–$\overline{\text{BE0}}$), and the four parity lines (DP3–DP0). The CLK input provides the basic data timing signal—the heartbeat of the computer. The only lines that activate the processor are the RESET, INTR, and NMI lines. The RESET signal initializes the processor to a known state. The INTR and NMI lines come from system hardware to signal either that a peripheral device needs service or that an error or failure has occurred. The remaining input signals are control signals that tell the microprocessor when to access its bus ($\overline{\text{RDY}}$, HOLD, $\overline{\text{BRDY}}$, and $\overline{\text{BOFF}}$), manipulate the internal cache ($\overline{\text{KEN}}$, $\overline{\text{FLUSH}}$, AHOLD, and $\overline{\text{EADS}}$), use less than the 32 available data bits in a data transfer ($\overline{\text{BS16}}$ or $\overline{\text{BS8}}$), emulate Virtual 8086 Mode ($\overline{\text{A20M}}$), or ignore numeric errors ($\overline{\text{IGNNE}}$). So where does programming come in?

Programming defines the values (1s and 0s) that are placed on the data lines. Circuits inside the microprocessor define how these values are interpreted by the microprocessor. These circuits define the microprocessor “instruction code.” Corresponding data signals activate specific processing by the microprocessor.

There are three major types of programming that correspond to the basic requirements of a personal computer:

- Basic input/output system (BIOS) software
- Operating system (OS) software
- Application software

All of these types of software use the same instruction set to perform operations within a personal computer system. The major difference between them is the level at which they operate and the operations they perform.

A.1.1 BIOS Software

BIOS software is stored in a stable memory storage device (some type of ROM or FLASH RAM). This software usually performs at two levels: system initialization and peripheral interface (input/output). Initialization begins when the microprocessor receives a RESET signal. The signal starts an internal “hard-wired” program to test and initialize the internal registers (data transfer and storage locations) in the processor and load a test program into the system memory. This Power-On Self-Test program (POST) evaluates the operational status of the system components. When the tests are complete, the BIOS loads the lower part of memory with a set of address maps that reference the input/output (I/O) part of the BIOS software; and, if specified by stored system parameter values, loads the I/O programs themselves into locations in the system memory (BIOS shadowing). Finally, the BIOS turns over control to the operating system software by issuing an INT 19h instruction.

The actual location of the BIOS memory references in the lower part of memory is based on the original IBM standards and subsequent industry developments. See Appendix H for a description of the memory map. The I/O software referred to in the memory map can be read directly from in its source or from the system memory if the software is shadowed. Shadowing the BIOS software provides faster system response. The I/O software assumes further that the I/O devices themselves have a specific physical address through which they are addressed. See Appendix J for a list of the standard I/O addresses.

A.1.2 OS Software

Operating system software provides a more user-friendly level of operation. It provides a base set of programs that allow the user to access information retained on bulk storage devices (defining manageable sets of data using files and directories), adjust system information (such as system time and date), and invoke application programs. There is a variety of operating systems available, but the two most common among IBM-compatible personal computer users are the command-oriented DOS and the icon-oriented graphical interface Microsoft® Windows™. In addition to providing a basic interface between the user and the personal computer system, the OS software also integrates special programs called drivers to allow you to expand and customize the number and types of peripheral devices used with your system. These may include special video drivers to accommodate newer types of video cards and monitors, as well as user input devices (scanners, digitizers, mouse devices, trackballs, etc.), communication devices (fax/modems), network interfaces, and a myriad of emerging multimedia devices.

A.1.3 Application Software

Application software includes a variety of specific-function packages. With these packages, a personal computer can be a documentation production unit, an animation studio, a musical instrument, a tutor, a drafting tool, a communication base, an accounting division, a game arcade, or almost anything imaginable. More programs become available every day.

A.1.4 Software Overview

Regardless of the level of complexity, all of the types of programming share a common base. They all use the same concepts of program development, use the same instruction set, and have access to the same general registers. However, the microprocessor provides internal divisions of memory access (segmentation) and coding (priority levels) that allow segregation of the operation of the various types of programming.

A.2 BASIC PROGRAMMING MODEL

To create effective and efficient software, a programmer must have a good understanding of the following environmental elements established by the microprocessor architecture:

- Operating modes
- Memory organization
- Internal system protection
- Data types
- Registers
- Instruction format
- Operand selection
- Interrupts and exceptions
- I/O operations

A.2.1 Operating Modes

For user convenience, industry-wide compatibility, and general acceptance in the personal computer market, the Am486 microprocessor must support a variety of programs originally written for 8086, 8088, 286, 386, and 486 microprocessors. The Am486 microprocessor uses three operating modes to provide this level of compatibility:

- Protected Mode—the highest operating mode level. This mode supports the full 32-bit instruction set with all of its architectural features.
- Real Mode—the basic 8086 emulation mode. This mode limits the processor to real addresses (from 0 to 1 Mbyte) only with no translation. Some extensions of the 8086 mode are provided, such as the ability to break out of the mode.

Note: *Reset initialization always places the processor into Real Mode; the operating system needs to change bit 0 in CR0 to a 1 to go to Protected Mode.*

- Virtual Mode—a modified 8086 emulation mode. This mode is compatible with available protection and memory management. The processor can enter the Virtual Mode from Protected Mode to run programs written for the 8086 processor, and then return to Protected Mode to execute 32-bit instructions without having to undergo system reset and initialization.

Note: *Bit 17 of the EFLAGS register is the VM bit. Setting the bit to a 1 places the processor in Virtual Mode. Resetting the bit to 0 returns to Protected Mode.*

Whenever execution occurs, the current operating mode determines the extent to which a program implements a specific instruction. Chapter 2 includes for each instruction the exceptions that it may generate depending on the operating mode. In general, both Real Mode and Virtual Mode are limited to 8-bit operations and 1-Mbyte maximum addressing limits. Most memory management features, such as segmentation and paging, are not available to Real Mode or Virtual Mode operation. In these two modes, addressing is linear and direct. These two modes can access the instructions added by later processors with the restrictions described above.

A.2.2 Memory Organization

A microprocessor requires external memory to store the values (both data and programming code) that are loaded into and out of the microprocessor through the data lines. Although a personal computer system uses physical memory chips organized as a series of 8-bit bytes located at unique sequential physical addresses, the programmer has a variety of methods available to access a specific memory location. These optional memory access methods are controlled by the microprocessor memory management system.

The memory management system lets operating systems control the environments in which programs run. If several programs run at the same time, they each need an independent address space to avoid having to perform difficult and time-consuming checks to avoid interfering with each other. To accomplish this, the memory management system in Am486 processors uses two memory control mechanisms: *segmentation* and *paging*. Segmentation gives each program several independent and protected address spaces. Paging supports an environment where large address spaces are simulated using a small amount of RAM and some disk storage. System designers may choose to use either or both of these mechanisms.

A.2.2.1 Segmentation

Segmentation can allow memory to be completely unstructured and simple, like the memory model of an 8-bit microprocessor, or highly structured with address translation and protection. The microprocessor implements this concept by dividing memory into units called segments. Each segment is an independent, protected address space. Access to segments is controlled by a data set that describes its size, the privilege level required for access, the kinds of memory references allowed to it (instruction fetch, stack push or pop, read operation, write operation, etc.), and whether it is present in memory (this final feature allows segment contents to be swapped between memory and disk space).

In addition to controlling memory access, segmentation can also simplify the linkage of object code modules. There is no reason to write position dependent code when full use is made of the segmentation mechanism, because all memory references can be made relative to the base addresses of a module's code and data segments. Segmentation can be used to create ROM-based software modules in which fixed addresses (fixed, in the sense that they cannot be changed) are offsets from a segment's base address. Different software systems can have the ROM modules at different physical addresses because the segmentation mechanism will direct all memory references to the right place.

A.2.2.1.1 Simple Memory Architecture

In a simple memory architecture, all addresses refer to the same address space. This is the memory model used by 8-bit microprocessors such as the 8086 microprocessor where the logical address is the physical address. The Am486 microprocessor can be used in this way by mapping all segments into the same address space and keeping paging disabled. This might be done where an older design is being updated to 32-bit technology without also adopting the new architectural features.

A.2.2.1.2 Partial Segmentation Use

An application can also make partial use of segmentation. A common cause of software failures is the growth of the stack into the instruction code or data used by the program. Proper use of segmentation can prevent this. The stack can be put in an address space separate from the address space for both code and data. Stack addresses always refer to memory in the stack segment, while data addresses always refer to memory in the data segment. The stack segment has a hardware controlled maximum limit. Any attempt to exceed this limit generates an exception.

A.2.2.1.3 Full Segmentation Implementation

A complex system of programs may make full use of segmentation and have precise control of access to shared data. This creates an environment in which the programs can interact by manipulating data used throughout the system without creating exceptions or overwriting an operating code or data. Real Mode can implement full segmentation within the overall memory limits.

A.2.2.2 Paging

Paging simulates a large, unsegmented address space using a small, fragmented address space and some disk storage. Paging provides access to data structures larger than the available memory space by keeping them partly in memory and partly on disk. The microprocessor creates memory units of 4 Kbytes called pages. When a program attempts to access a page stored on disk, a special exception occurs. Unlike other exceptions and interrupts, an address translation exception restores the contents of the microprocessor registers to values that allow the exception generating instruction to reexecute. This special action is called instruction restart. It allows the operating system to read the page from disk, update the mapping of linear addresses to physical addresses for that page, and restart the program. This process is transparent to the program.

If an operating system or memory manager never sets bit 31 of the CR0 register (the PG bit), the paging mechanism is not enabled. Linear addresses are read as physical addresses directly. This might be desirable if you are updating a 16-bit processor design for use with a 32-bit microprocessor. The 16-bit processor operating system does not use paging because its address space is so small (64 Kbytes) and it is more efficient to swap entire segments between RAM and disk, rather than individual pages. Paging is enabled for operating systems that can support demand-paged virtual memory, such as UNIX. Paging is transparent to application software, so an operating system intended to support application programs written for 16-bit microprocessors may run those programs with paging enabled. Unlike paging, segmentation is not transparent to application programs. Programs that use relocatable codes (i.e., hard coded segments) must be run with the segments they were designed to use. Segmentation hardware translates a segmented (logical) address into an address for a continuous, unsegmented address space, called a linear address. If paging is enabled, paging hardware translates a linear address into a physical address. If paging is not enabled, the linear address is used as the physical address. The physical address appears on the address bus coming out of the microprocessor.

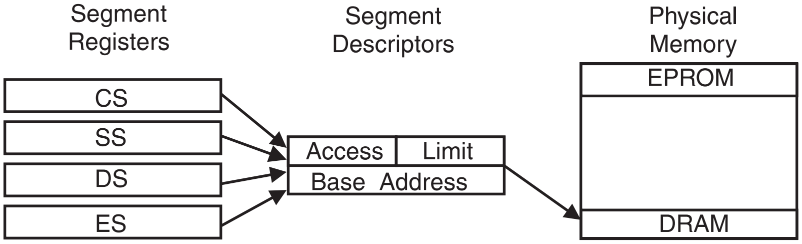
A.2.2.3 Selecting a Segmentation Model

A model for the segmentation of memory is chosen on the basis of reliability and performance. For example, a system that has several programs sharing data in real time would get maximum performance from a model that checks memory references in hardware. This would be a multisegment model. At the other extreme, a system that has just one program may get higher performance from an unsegmented or “flat” model. The elimination of “far” pointers and segment override prefixes reduces code size and increases execution speed. Context switching is faster because the contents of the segment registers no longer have to be saved or restored. Some of the benefits of segmentation also can be provided by paging. For example, data can be shared by mapping the same pages onto the address space of each program.

A.2.2.3.1 Flat Model

The simplest model is the flat model. In this model, all segments are mapped to the entire physical address space. A segment offset can refer to either code or data areas. To the greatest extent possible, this model removes the segmentation mechanism from the architecture seen by either the system designer or the application programmer. This might be done for a programming environment like UNIX, which supports paging but does not support segmentation. A segment is defined by a segment descriptor. At least two segment descriptors must be created for a flat model, one for code references and one for data references. Both descriptors have the same base address value. Whenever memory is accessed, the contents of one of the segment registers is used to select a segment descriptor. The segment descriptor provides the base address of the segment and its limit, as well as access control information (see Figure A-1).

Figure A-1 Flat Memory Model



ROM usually is put at the top of the physical address space because the microprocessor begins execution at 0FFFFFFF0h. RAM is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0. For a flat model, each descriptor has a base address of 0 and a segment limit of 4 Gbytes. By setting the segment limit to 4 Gbytes, the segmentation mechanism is kept from generating exceptions for memory references that fall outside of a segment. Exceptions could still be generated by the paging or segmentation protection mechanisms, but these also can be removed from the memory model.

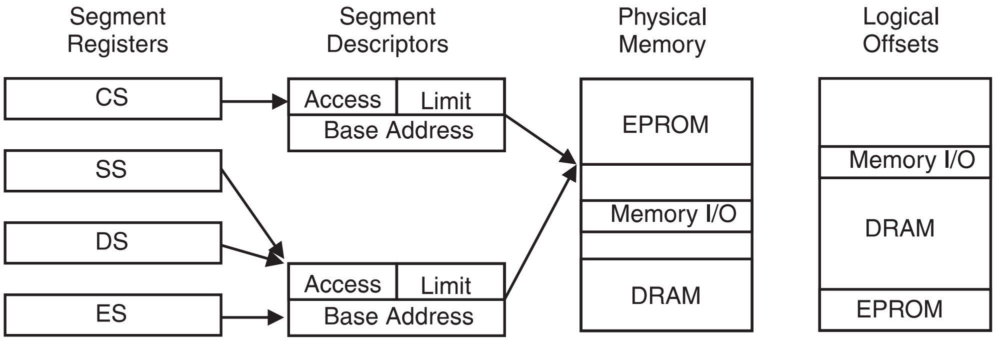
A.2.2.3.2 Protected Flat Model

The protected flat model is similar to the flat model, except the segment limits are set to include only the range of addresses for which memory actually exists. A general protection exception is generated by any attempt to access unimplemented memory. This provides a minimum level of hardware protection against unexpected programming results when the paging mechanism is disabled in a system.

In this model, the segmentation hardware prevents programs from addressing nonexistent memory locations. The consequences of being allowed access to these memory locations are hardware-dependent. For example, if the microprocessor does not receive a $\overline{\text{READY}}$ signal (the signal used to acknowledge and terminate a bus cycle), the bus cycle does not terminate and program execution stops. Although no program should make an attempt to access these memory locations, an attempt may occur as a result of programming errors. Without hardware checking of addresses, it is possible that an expected programming result could suddenly stop program execution. With hardware checking, programs fail in a controlled way. A diagnostic message can appear and recovery procedures can be attempted.

An example of a protected flat model is shown in Figure A-2. Here, segment descriptors have been set up to cover only those ranges of memory that exist. A code and a data segment cover the EPROM and DRAM of physical memory. The code segment limit can be optionally set to allow access to DRAM area. The data segment limit must be set to the sum of EPROM and DRAM sizes. If memory-mapped I/O is used, it can be addressed just beyond the end of the DRAM area.

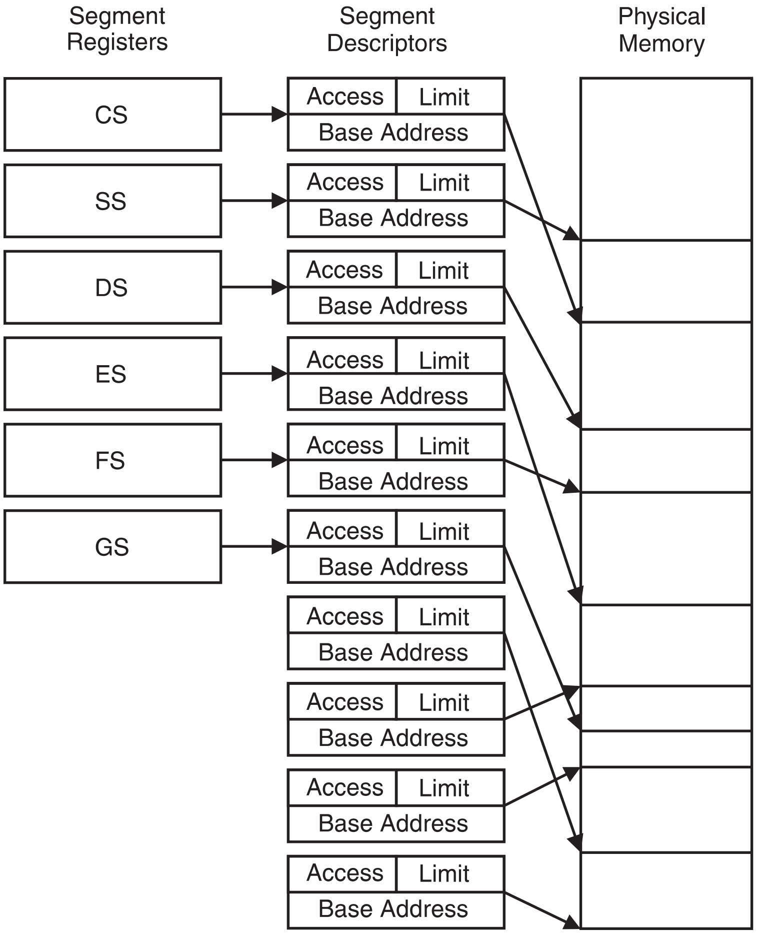
Figure A-2 Protected Flat Memory Model



A.2.2.3.3 *Multisegment Model*

The most sophisticated model is the multisegment model. Here the full capabilities of the segmentation mechanism are used. Each program is given its own table of segment descriptors, and its own segments. The segments can be completely private to the program, or they can be shared with specific other programs. Access between programs and particular segments can be individually controlled. Up to six segments can be ready for immediate use. These are the segments that have segment selectors loaded in the segment registers. Other segments are accessed by loading their segment selectors into the segment registers (see Figure A-3).

Figure A-3 Multisegment Memory Model



Each segment is a separate address space. Even though they may be placed in adjacent blocks of physical memory, the segmentation mechanism prevents access to the contents of one segment by reading beyond the end of another. Every memory operation is checked against the limit specified for the segment it uses. An attempt to address memory beyond the end of the segment generates a general-protection exception.

The segmentation mechanism only enforces the address range specified in the segment descriptor. It is the responsibility of the operating system to allocate separate address ranges to each segment. There may be situations in which it is desirable to have segments that share the same range of addresses. For example, a system may have both code and data stored in a ROM. A code segment descriptor is used when the ROM is accessed for instruction fetches. A data segment descriptor is used when the ROM is accessed as data.

A.2.2.4 Segment Translation

A logical address consists of the 16-bit segment selector for its segment and a 32-bit offset into the segment. A logical address is translated into a linear address by adding the offset to the base address of the segment. The base address comes from the segment descriptor, a data structure in memory that provides the size and location of a segment, as well as access control information. The segment descriptor comes from one of two tables, the global descriptor table (GDT) or the local descriptor table (LDT). There is one GDT for all programs in the system, and one LDT for each separate program being run. If the operating system allows, different programs can share the same IDT. The system also may be set up with no LDTs; all programs will then use the GDT.

Every logical address is associated with a segment (even if the system maps all segments into the same linear address space). Although a program may have thousands of segments, only six may be available for immediate use. These are the six segments whose segment selectors are loaded in the microprocessor. The segment selector holds information used to translate the logical address into the corresponding linear address.

Separate segment registers exist in the microprocessor for each kind of memory reference (code space, stack space, and data spaces). They hold the segment selectors for the segments currently in use. Access to other segments requires loading a segment register using a form of the MOV instruction. Up to four data spaces may be available at the same time, thus providing a total of six segment registers.

When a segment selector is loaded, the base address, segment limit, and access control information also are loaded into the segment register. The microprocessor does not reference the descriptor tables again until another segment selector is loaded. The information saved in the microprocessor allows it to translate addresses without making extra bus cycles. In systems in which multiple microprocessors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified.

The segment selector contains a 13-bit index into one of the descriptor tables. The index is scaled by 8 (the number of bytes in a segment descriptor) and added to the 32-bit base address of the descriptor table. The base address comes from either the global descriptor table register (GDTR) or the local descriptor table register (LDTR). These registers hold the linear address of the beginning of the descriptor tables. A bit in the segment selector specifies which table to use (see Figure A-4).

Figure A-4   TI Bit Selects Descriptor Table

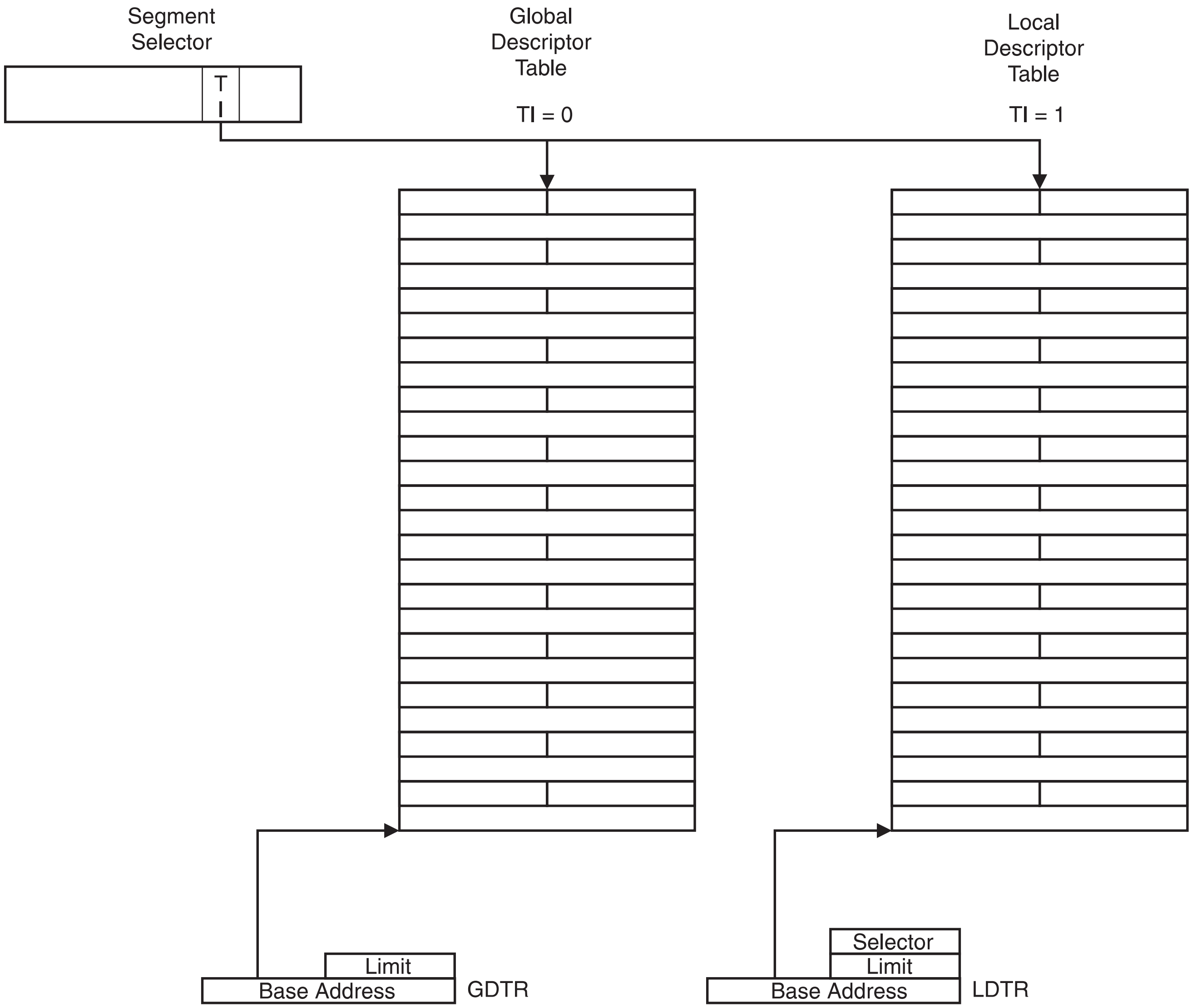
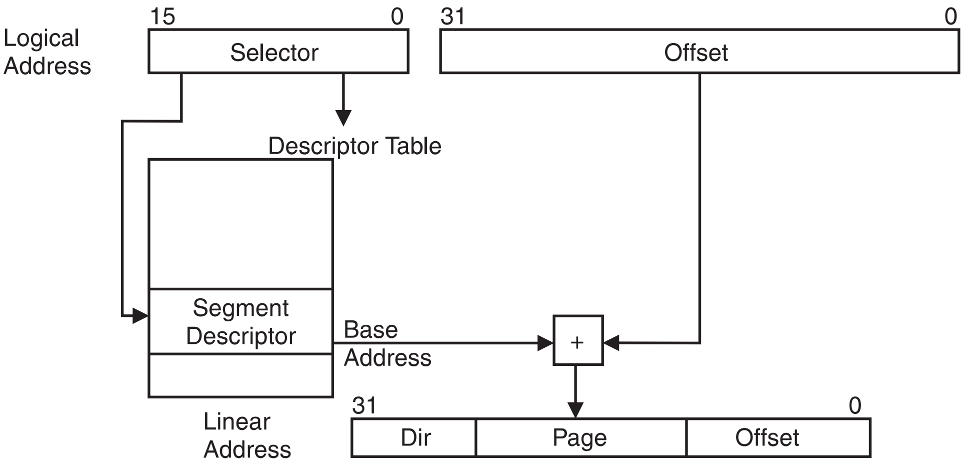


Figure A-5 Segment Translation



The translated address is the linear address (see Figure A-5). If paging is not used, the translated address is also the physical address. If paging is used, a second level of address translation produces the physical address. This translation is described in Section A.2.2.5.

A.2.2.4.1 Segment Registers

Each kind of memory reference is associated with a segment register. Code, data, and stack references each access the segment specified by their segment register contents. More segments can be made available by loading their segment selectors into these registers during program execution. Every segment register has a “visible” part and an “invisible” part (see Figure A-6). There are forms of the MOV instruction to load the visible part of these segment registers. The invisible part is loaded by the microprocessor.

The operations that load these registers are instructions for application programs (described in Chapter 2). There are two kinds of these instructions:

- Direct load instructions such as the MOV, POP, LDS, LES, LFS, LGS, and LSS instructions. These instructions explicitly reference the segment registers.
- Implied load instructions such as the far pointer versions of the CALL and JMP instructions. These instructions change the contents of the CS register as an incidental part of their function.

Figure A-6 Segment Registers

Visible Part	Invisible Part	
Selector	Base Address, Limit, etc.	CS
		SS
		DS
		ES
		FS
		GS

When these instructions are used, the visible part of the segment register is loaded with a segment selector. The microprocessor automatically fetches the base address, limit, type, and other information from the descriptor table and loads the invisible part of the segment register.

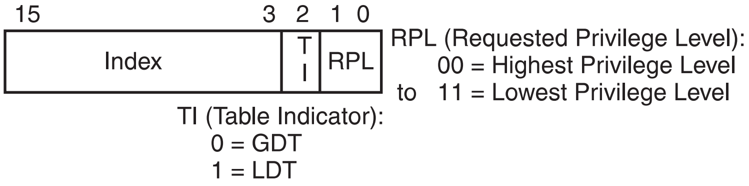
Because most instructions refer to segments whose selectors already have been loaded into segment registers, the microprocessor can add the logical-address offset to the segment base address with no performance penalty.

A.2.2.4.2 *Segment Selectors*

A segment selector points to the information that defines a segment, called a segment descriptor. A program may have more segments than the six whose segment selectors occupy segment registers. When this is true, the program uses forms of the MOV instruction to change the contents of these registers when it needs to access a new segment.

A segment selector identifies a segment descriptor by specifying a descriptor table and a descriptor within that table. Segment selectors are visible to application programs as a part of a pointer variable, but the values of selectors are usually assigned or modified by link editors or linking loaders, not application programs. Figure A-7 shows the format of a segment selector.

Figure A-7 Segment Selector


15
3
2
1
0
Index
T
I
RPL
RPL (Requested Privilege Level):
00 = Highest Privilege Level
to 11 = Lowest Privilege Level
TI (Table Indicator):
0 = GDT
1 = LDT

- *Index:* Selects one of 8192 descriptors in a descriptor table. The microprocessor multiplies the index value by 8 (the number of bytes in a segment descriptor) and adds the result to the base address of the descriptor table (from the GDTR or LDTR register).
- *Table Indicator bit:* Specifies the descriptor table to use. A clear bit selects the GDT; a set bit selects the current LDT.
- *Requester Privilege Level:* When this field contains a privilege level having a greater value (i.e., less privileged) than the program, it overrides the program's privilege level. When a program uses a less privileged segment selector, memory accesses take place at the lesser privilege level. This is used to guard against a security violation in which a less privileged program uses a more privileged program to access protected data.

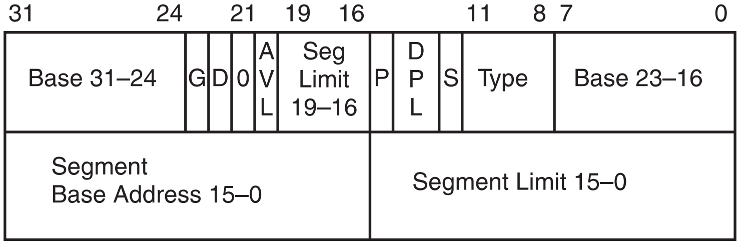
For example, system utilities or device drivers must run with a high level of privilege in order to access protected facilities such as the control registers of peripheral interfaces. But they must not interfere with other protected facilities, even if a request to do so is received from a less privileged program. If a program requested reading a sector of disk into memory occupied by a more privileged program, such as the operating system, the RPL can be used to generate a general-protection exception when the less privileged segment selector is used. This exception occurs even though the program using the segment selector would have a sufficient privilege level to perform the operation on its own.

Because the first entry of the GDT is not used by the microprocessor, a selector that has an index of 0 and a table indicator of 0 (i.e., a selector that points to the first entry of the GDT) is used as a "null selector." The microprocessor does not generate an exception when a segment register (other than the CS or SS registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. This feature can be used to initialize unused segment registers.

A.2.2.4.3 *Segment Descriptors*

A segment descriptor is a data structure in memory that provides the microprocessor with the size and location of a segment, as well as control and status information. Descriptors are typically created by compilers, linkers, loaders, or the operating system, but not application programs. Figure A-8 illustrates the general descriptor format.

Figure A-8 Segment Descriptor



Base	Segment Base Address
G	Granularity (0 = range is 1 byte–1 Mbyte; 1 = range is 4 Kbyte–4 Gbyte)
D	Default Operation Size (Code Segment only: 0 = 16-bit; 1 = 32-bit)
0	Reserved—always 0
AVL	Available for use by system software
Seg Limit	Segment Limit
P	Segment Present
DPL	Descriptor Privilege Level (00 = Highest — 11 = Lowest)
Type	Segment Type

All types of segment descriptors take one of these formats:

- *Base:* Defines the location of the segment within the 4-Gbyte physical address space. The microprocessor puts together the three base address fields to form a single 32-bit value. Segment base values should be aligned to 16-byte boundaries to allow programs to maximize performance by aligning code/data on 16-byte boundaries.
- *Granularity bit:* Turns on scaling of the limit field by a factor of 4096 (2^{12}). When the bit is clear, the segment limit is interpreted in units of 1 byte; when set, the segment limit is interpreted in units of 4 Kbytes (one page). Note that the twelve least-significant bits of the address are not tested when scaling is used. For example, a limit of 0 with the Granularity bit set results in valid offsets from 0 to 4095. Also note that only the Limit field is affected. The base address remains byte-granular.
- *Limit:* Defines the size of the segment. The microprocessor puts together the two limit fields to form a 20-bit value. The microprocessor interprets the limit in one of two ways, depending on the setting of the Granularity bit:

  — If the Granularity bit is clear, the limit has a value from 1 byte to 1 Mbyte, in increments of 1 byte.

  — If the Granularity bit is set, the Limit has a value from 4 Kbytes to 4 Gbytes, in increments of 4 Kbytes.

- *Offset:* For most segments, a logical address may have an offset ranging from 0 to the limit. Other offsets generate exceptions. Expand-down segments reverse the sense of the Limit field; they may be addressed with any offset except those from 0 to the limit (see the Type field, below). This is done to allow segments to be created in which increasing the value held in the Limit field allocates new memory at the bottom of the segment's address space, rather than at the top. Expand-down segments are intended to hold stacks, but it is not necessary to use them. If a stack is going to be put in a segment that does not need to change size, it can be a normal data segment.
- *S bit:* Determines whether a given segment is a system segment or a code or data segment. If the S bit is set, then the segment is either a code or a data segment. If it is clear, then the segment is a system segment.
- *D bit:* The code segment D bit indicates the default length for operands and effective addresses. If the D bit is set, then 32-bit operands and 32-bit effective addressing modes are assumed. If it is clear, then 16-bit operands and addressing modes are assumed.
- *Type:* The interpretation of this field depends on whether the segment descriptor is for an application segment or a system segment. System segments have a slightly different descriptor format. The Type field of a memory descriptor specifies the kind of access that may be made to a segment, and its direction of growth (see Table A-1).

Table A-1 Application Segment Types

Number	E	W	A	Descriptor Type	Description
0	0	0	0	Data	Read-Only
1	0	0	1	Data	Read-Only, accessed
2	0	1	0	Data	Read/Write
3	0	1	1	Data	Read/Write, accessed
4	1	0	0	Data	Read-Only, expand-down
5	1	0	1	Data	Read-Only, expand-down, accessed
6	1	1	0	Data	Read/Write, expand-down
7	1	1	1	Data	Read/Write, expand-down, accessed
Number	**C**	**R**	**A**	**Descriptor Type**	**Description**
8	0	0	0	Code	Execute-Only
9	0	0	1	Code	Execute-Only, accessed
10	0	1	0	Code	Execute/Read
11	0	1	1	Code	Execute/Read, accessed
12	1	0	0	Code	Execute-Only, conforming
13	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	0	Code	Execute/Read, conforming
15	1	1	1	Code	Execute/Read, conforming, accessed

For data segments, the three lowest bits of the type field can be interpreted as expand-down (E), write-enable (W), and accessed (A). For code segments, the three lowest bits of the type field can be interpreted as conforming (C), read-enable (R), and accessed (A).

Data segments can be read-only or read/write. Stack segments are data segments that must be read/write. Loading the SS register with a segment selector for any other type of segment generates a general-protection exception. If the stack segment needs to be able to change size, it can be an expand-down data segment. The meaning of the segment limit is reversed for an expand-down segment. While an offset in the range from 0 to the segment limit is valid for other kinds of segments (outside this range a general protection exception is generated), in an expand-down segment these offsets are the ones that generate exceptions. The valid offsets in an expand-down segment are those that generate exceptions in the other kinds of segments. Expand-up segments must be addressed by offsets that are equal to or less than the segment limit. Offsets into expand down segments always must be greater than the segment limit. This interpretation of the segment limit causes memory space to be allocated at the bottom of the segment when the segment limit is decreased, which is correct for stack segments because they grow toward lower addresses. If the stack is given a segment that does not change size, it does not need to be an expand-down segment.

Code segments can be execute-only or execute/read. An execute/read segment might be used, for example, when constants have been placed with instruction code in a ROM. In this case, the constants can be read either by using an instruction with a CS override prefix or by placing a segment selector for the code segment in a segment register for a data segment.

Code segments can be either conforming or non-conforming. A transfer of execution into a more privileged conforming segment keeps the current privilege level. A transfer into a non-conforming segment at a different privilege level results in a general protection exception, unless a task gate is used. System utilities that do not access protected facilities, such as data-conversion functions (e.g., EBCDIC/ASCII translation, Huffman encoding/decoding, math library) and some types of exceptions (e.g., Divide Error, INTO-detected overflow, and BOUND range exceeded) may be loaded in conforming code segments.

The Type field also reports whether the segment has been accessed. Segment descriptors initially report a segment as having been accessed. If the Type field then is set to a value for a segment that has not been accessed, the microprocessor restores the value if the segment is accessed. By clearing and testing the Low bit of the Type field, software can monitor segment usage (the Low bit of the Type field also is called the Accessed bit).

For example, a program development system might clear all of the Accessed bits for the segments of an application. If the application crashes, the states of these bits can be used to generate a map of all the segments accessed by the application. Unlike the breakpoints provided by the debugging mechanism, the usage information applies to segments rather than physical addresses.

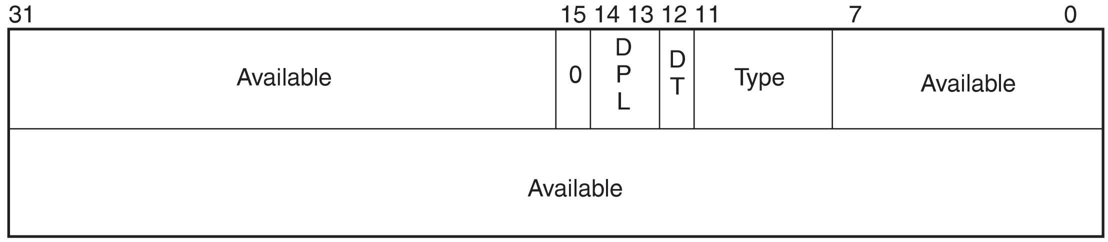
The microprocessor may update the Type field when a segment is accessed, even if the access is a read cycle. If the descriptor tables have been put in ROM, it may be necessary for hardware to prevent the ROM from being enabled onto the data bus during a write cycle. It also may be necessary to return the $\overline{\text{READY}}$ signal to the microprocessor when a write cycle to ROM occurs, otherwise the cycle does not terminate. These features of the hardware design are necessary for using ROM-based descriptor tables with the Am386DX microprocessor, which always sets the Accessed bit when a segment descriptor is loaded. The Am486 microprocessor, however, only sets the Accessed bit if it is not already set.

Writes to descriptor tables in ROM can be avoided by setting the Accessed bits in every descriptor.

- *DPL (Descriptor Privilege level):* Defines the privilege level of the segment. This is used to control access to the segment, using the protection mechanism described in Section A.2.3.
- *Segment-Present bit:* If this bit is clear, the microprocessor generates a segment-not-present exception when a selector for the descriptor is loaded into a segment register. This is used to detect access to segments that have become unavailable. A segment can become unavailable when the system needs to create free memory. Items in memory, such as character fonts or device drivers, which currently are not being used are deallocated. An item is deallocated by marking the segment “not present” (this is done by clearing the Segment-Present bit). The memory occupied by the segment then can be put to another use. The next time the deallocated item is needed, the segment-not-present exception will indicate the segment needs to be loaded into memory. When this kind of memory management is provided in a manner invisible to application programs, it is called virtual memory. A system may maintain a total amount of virtual memory far larger than physical memory by keeping only a few segments present in physical memory at any one time.

Figure A-9 shows the format of a descriptor when the Segment-Present bit is clear. When this bit is clear, the operating system is free to use the locations marked Available to store its own data, such as information regarding the whereabouts of the missing segment.

Figure A-9 Segment Descriptor (Segment Not Present)



The P bit (bit 15 of the high doubleword) is 0, indicating that the segment is not present in memory.

A.2.2.4.4 *Segment Descriptor Tables*

A segment descriptor table is an array of segment descriptors. There are two kinds of descriptor tables:

- The global descriptor table (GDT)
- The local descriptor tables (LDT)

There is one GDT for all tasks, and an LDT for each task being run. A descriptor table is an array of segment descriptors (see Figure A-10).

A descriptor table is variable in length and may contain up to 8192 (2^{13}) descriptors. The first descriptor in the GDT is not used by the microprocessor. A segment selector to this “null descriptor” does not generate an exception when loaded into a segment register, but it always generates an exception when an attempt is made to access memory using the descriptor. By initializing the segment registers with this segment selector, accidental reference to unused segment registers can be guaranteed to generate an exception.

Figure A-10 Descriptor Tables

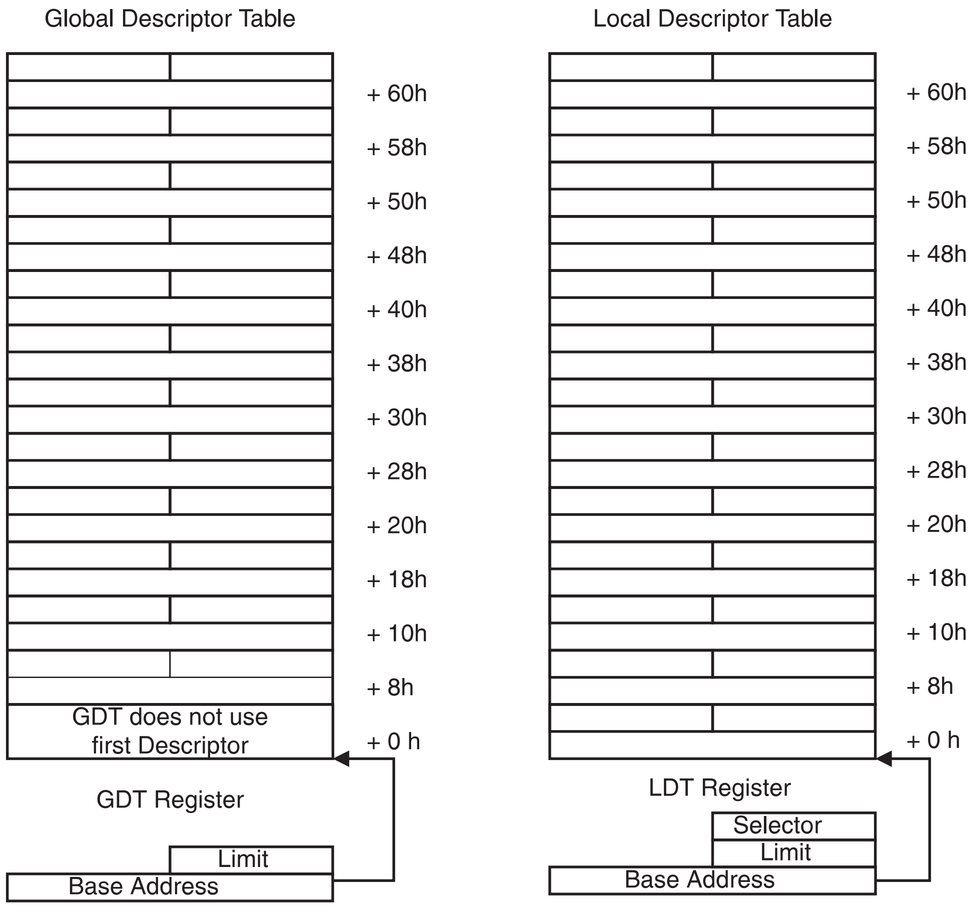


Figure A-11 Pseudo-Descriptor Format

47	39	31	23	15	7 0
Base Address				Limit	
Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0

A.2.2.4.5 Descriptor Table Base Registers

The microprocessor finds the global descriptor table (GDT) and interrupt descriptor table (IDT) using the GDTR and IDTR registers. These registers hold 32-bit base addresses for tables in the linear address space. They also hold 16-bit limit values for the size of these tables. When the registers are loaded or stored, a 48-bit “pseudo-descriptor” is accessed in memory (see Figure A-11). The GDT and IDT should be aligned on a 16-byte boundary to maximize performance due to cache line fills. The limit value is expressed in bytes. As with segments, the limit value is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly one valid byte. Because segment descriptors are always 8 bytes, the limit should always be one less than an integral multiple of eight (that is, 8N – 1). The LGDT and SGDT instructions read and write the GDTR register; the LIDT and SIDT instructions read and write the IDTR register.

A third descriptor table is the local descriptor table (LDT). It is identified by a 16-bit segment selector held in the LDTR register. The LLDT and SLDT instructions read and write the segment selector in the LDTR register. The LDTR register also holds the base address and limit for the LDT, but these are loaded automatically by the microprocessor from the segment descriptor for the LDT. The LDT should be aligned on a 16-byte boundary to maximize performance due to cache line fills.

Alignment check faults may be generated by storing a pseudo-descriptor in user mode (privilege level 3). User-mode programs normally do not store pseudo-descriptors, but the possibility of generating an alignment check fault in this way can be avoided by placing the pseudo-descriptor at an odd word address (i.e., an address which is 2 MOD 4). This causes the microprocessor to store an aligned word, followed by an aligned doubleword.

A.2.2.5 Page Translation

A linear address is a 32-bit address into a uniform, unsegmented address space. This address space may be a large physical address space (i.e., an address space composed of 4 Gbytes of RAM), or paging can be used to simulate this address space using a small amount of RAM and some disk storage. When paging is used, a linear address is translated into its corresponding physical address or an exception is generated. The exception gives the operating system a chance to read the page from disk (perhaps sending a different page out to disk in the process), then restart the instruction that generated the exception.

Paging differs from segmentation by its use of small, fixed-size pages. Unlike segments, which vary in size depending on the data structures they hold, Am486 microprocessor pages are always 4 Kbytes. If segmentation is the only form of address translation that is used, a data structure present in physical memory has all of its parts in memory. If paging is used, a data structure may be partly in memory and partly in disk storage.

Information that maps linear addresses into physical addresses and exceptions is held in data structures in memory called page tables. As with segmentation, this information is cached in microprocessor registers to minimize the number of bus cycles required for address translation. Unlike segmentation, these microprocessor registers are completely invisible to application programs. For testing purposes, however, these registers are visible to programs running with maximum privileges.

The paging mechanism treats the 32-bit linear address as having three parts, two 10-bit indexes into the page tables and a 12-bit offset into the page addressed by the page tables. Because both the virtual pages in the linear address space and the physical pages of memory are aligned to 4-Kbyte page boundaries, there is no need to modify the Low 12 bits of the address. These 12 bits pass straight through the paging hardware, whether paging is enabled or not. Note that this is different from segmentation, because segments can start at any byte address.

The upper 20 bits of the address are used to index into the page tables. If every page in the linear address space were mapped by a single page table in RAM, 4 Mbytes would be needed. This is not done. Instead, two levels of page tables are used. The top level page table is called the page directory. It maps the upper 10 bits of the linear address to the second level of page tables. The second level of page tables maps the middle 10 bits of the linear address to the base address of a page in physical memory (called a page frame address).

An exception may be generated based on the contents of the page table or the page directory. An exception gives the operating system a chance to bring in a page table from disk storage. By allowing the second-level page tables to be sent to disk, the paging mechanism can support mapping of the entire linear address space using only a few pages in memory.

The CR3 register holds the page frame address of the page directory. For this reason, it also is called the Page Directory Base Register or PDBR. The upper 10 bits of the linear address are scaled by four (the number of bytes in a page table entry) and added to the value in the PDBR register to get the physical address of an entry in the page directory. Because the page frame address is always clear in its lowest 12 bits, this addition is performed by concatenation (replacement of the Low 12 bits with the scaled index).

When the entry in the page directory is accessed, several checks are performed. Exceptions may be generated if the page is protected or is not present in memory. If no exception is generated, the upper 20 bits of the page table entry are used as the page frame address of a second-level page table. The middle 10 bits of the linear address are scaled by four (again, the size of a page table entry) and concatenated with the page frame address to get the physical address of an entry in the second-level page table.

Again, access checks are performed and exceptions may be generated. If no exception occurs, the upper 20 bits of the second-level page table entry are concatenated with the lowest 12 bits of the linear address to form the physical address of the operand (data) in memory.

Although this process may seem complex, it requires very little overhead. The microprocessor has a cache for page table entries called the Translation Lookaside Buffer (TLB). The TLB satisfies most requests for reading the page tables. Extra bus cycles occur only when a new page is accessed. The page size (4 Kbytes) is large enough so that very few bus cycles are made to the page tables, compared to the number of bus cycles made to instructions and data. At the same time, the page size is small enough to make efficient use of memory. (No matter how small a data structure is, it occupies at least one page of memory.)

A.2.2.5.1 PG Bit Enables Paging

If paging is enabled, a second stage of address translation is used to generate the physical address from the linear address. If paging is not enabled, the linear address is used as the physical address. Paging is enabled when bit 31 (the PG bit) of the CR0 register is set. This bit usually is set by the operating system during software initialization. The PG bit must be set if the operating system is running more than one program in Virtual 8086 Mode or if demand-paged virtual memory is used.

A.2.2.5.2 Linear Address

Figure A-12 shows the format of a linear address.

Figure A-12 Linear Address Format

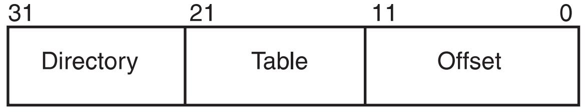
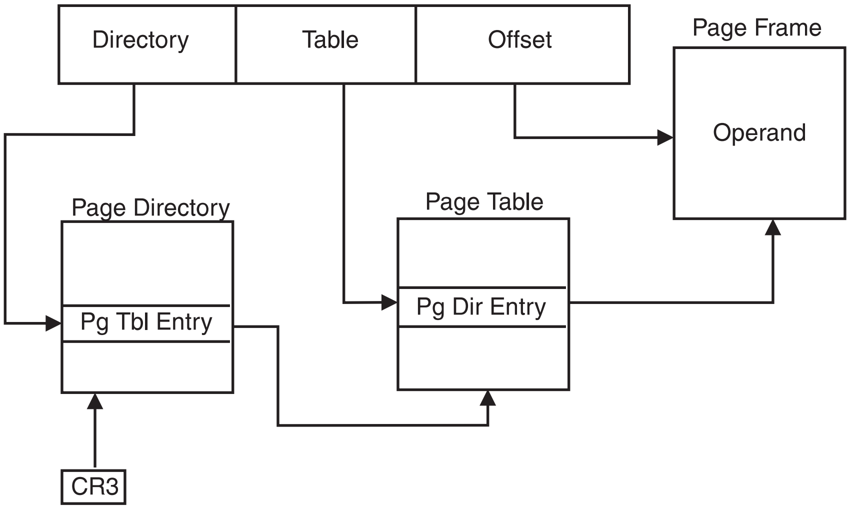


Figure A-13 shows how the microprocessor translates the DIRECTORY, TABLE, and OFFSET fields of a linear address into the physical address using two levels of page tables. The paging mechanism uses the DIRECTORY field as an index into a page directory, the TABLE field as an index into the page table determined by the page directory, and the OFFSET field to address an operand within the page specified by the page table.

Figure A-13 Page Translation



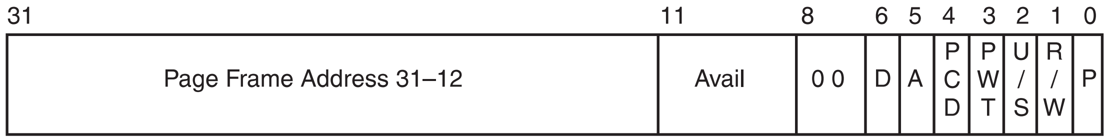
A.2.2.5.3 Page Tables

A page table is an array of 32-bit entries. A page table is itself a page, and contains 4096 bytes of memory or, at most, 1K 32-bit entries. All pages, including page directories and page tables, are aligned to 4-Kbyte boundaries.

A page of memory uses a two-tier reference system. The top tier is the *page directory*. The page directory addresses up to 1K or 2^{10} *page tables*, the second tier. A page table addresses up to 1K or 2^{10} pages in physical memory. Therefore, one page directory can address 1M or 2^{20} pages. Because each page contains 4K or 2^{12} bytes, one page directory can span the entire linear address space of the Am486 microprocessor (4G or 2^{32} bytes).

The physical address of the current page directory is stored in the CR3 register, also called the Page Directory Base Register (PDBR). Memory management software has the option of using one page directory for all tasks, one page directory for each task, or some combination of the two.

Figure A-14 Page Table Entry Format



Avail	Available for system user programs
0 0	Reserved, always 00
D	Dirty
A	Accessed
PCD	Page Cache Disable
PWT	Page Write Transparent
U/S	User/Supervisor
R/W	Read/Write
P	Present

A.2.2.5.4 Page Table Entries

Entries in either level of page tables have the same format, except that the page directory has no Dirty bit. Figure A-14 illustrates this format. The bit position of the D bit is reserved for future AMD use.

A.2.2.5.5 Page Frame Address

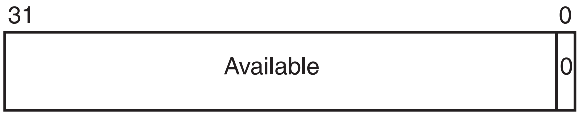
The page frame address is the base address of a page. In a page table entry, the upper 20 bits are used to specify a page frame address, and the lowest 12 bits specify control and status bits for the page. In a page directory, the page frame address is the address of a page table. In a second-level page table, the page frame address is the address of a page containing instructions or data.

A.2.2.5.6 Present Bit

The Present bit indicates whether the page frame address in a page table entry maps to a page in physical memory. When set, the page is in memory.

When the Present bit is clear, the page is not in memory, and the rest of the page table entry is available for the operating system, for example, to store information regarding the whereabouts of the missing page. Figure A-15 illustrates the format of a page table entry when the Present bit is clear.

Figure A-15 Page Table Entry Format for a Not-Present Page



If the Present bit is clear in either level of page tables when an attempt is made to use a page table entry for address translation, a page-fault exception is generated. In systems that support demand-paged virtual memory, the following sequence of events then occurs:

1. The operating system copies the page from disk storage into physical memory.
2. The operating system loads the page frame address into the page table entry and sets its Present bit. Other bits, such as the R/W bit, may be set as well.

3. Because a copy of the old page table entry may still exist in the translation lookaside buffer (TLB), the operating system empties it.
4. The program that caused the exception is then restarted.

Since there is no Present bit in CR3 to indicate when the page directory is not resident in memory, the page directory pointed to by CR3 should always be present in physical memory.

A.2.2.5.7 *Accessed and Dirty Bits*

These bits provide data about page usage in both levels of page tables. The Accessed bit is used to report read or write access to a page or second-level page table. The Dirty bit is used to report write access to a page.

With the exception of the Dirty bit in a page directory entry, these bits are set by the hardware; however, the microprocessor does not clear either of these bits. The microprocessor sets the Accessed bits in both levels of page tables before a read or write operation to a page. The microprocessor sets the Dirty bit in the second-level page table before a write operation to an address mapped by that page table entry. The Dirty bit in directory entries is undefined.

The operating system may use the Accessed bit when it needs to create some free memory by sending a page or second-level page table to disk storage. By periodically clearing the Accessed bits in the page tables, it can see which pages have been used recently. Pages that have not been used are candidates for sending out to disk.

The operating system may use the Dirty bit when a page is sent back to disk. By clearing the Dirty bit when the page is brought into memory, the operating system can see if it has received any write access. If there is a copy of the page on disk and the copy in memory has not received any writes, there is no need to update disk from memory.

A.2.2.5.8 *Read/Write and User/Supervisor Bits*

The Read/Write and User/Supervisor bits are used for protection checks applied to pages, which the microprocessor performs at the same time as address translation. See Section A.2.3.1 for more information on protection.

A.2.2.5.9 *Page-Level Cache Control Bits*

The PCD and PWT bits are used for page-level cache management. Software can control the caching of individual pages or second-level page tables using these bits.

A.2.2.5.10 *Translation Lookaside Buffer (TLB)*

The microprocessor stores the most recently used page table entries in an on-chip cache called the translation lookaside buffer or TLB. Most paging is performed using the contents of the TLB. Bus cycles to the page tables are performed only when a new page is used.

The TLB is invisible to application programs, but not to operating systems. Operating system programmers must flush the TLB (dispose of its page table entries) when entries in the page tables are changed. If this is not done, old data that has not received the changes might get used for address translation. A change to an entry for a page that is not present in memory does not require flushing the TLB, because entries for not-present pages are not cached.

The TLB is flushed when the CR3 register is loaded. The CR3 register can be loaded in either of two ways:

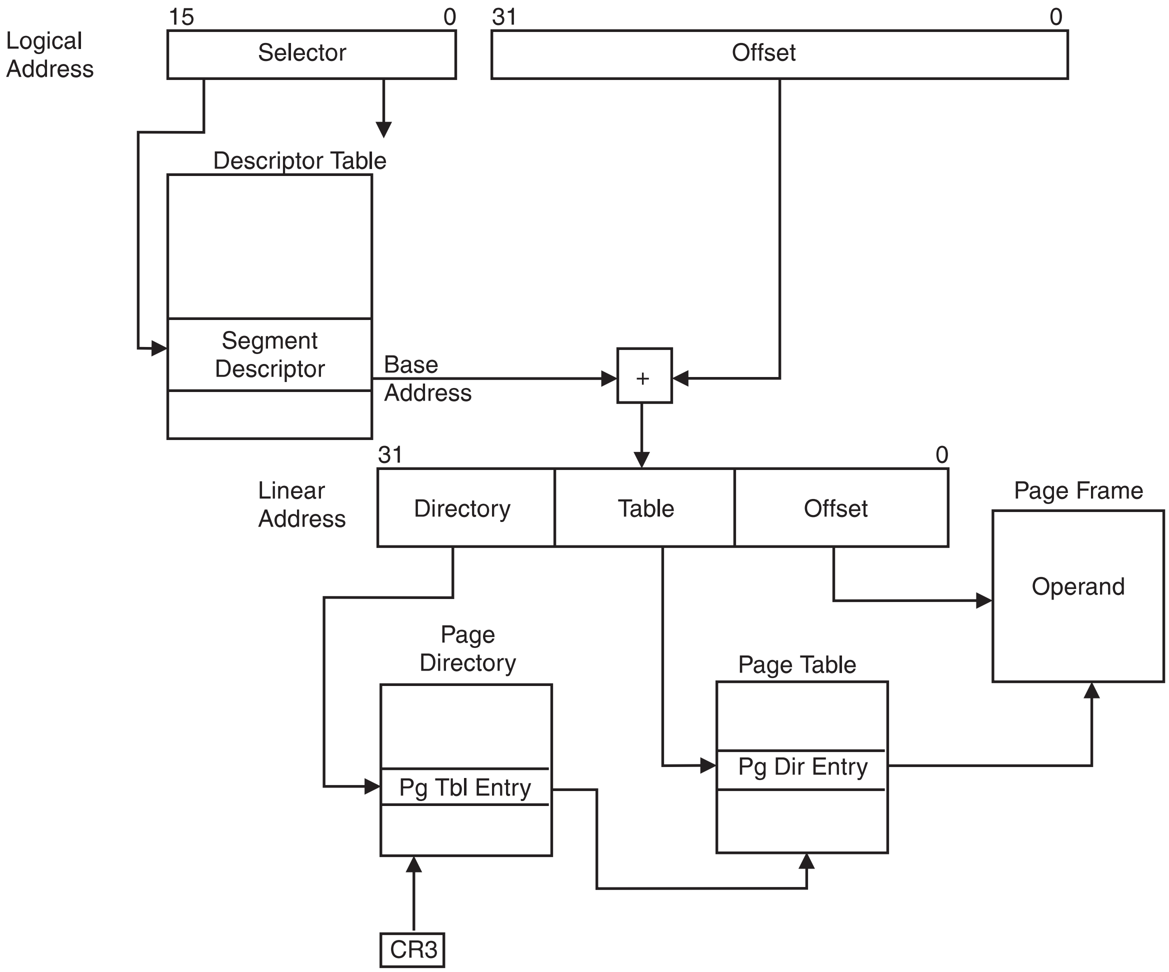
- Explicit loading using MOV instructions, such as: MOV CR3, EAX
- Implicit loading by a task switch that changes the contents of the CR3 register

An individual entry in the TLB can be flushed using an INVLPG instruction. This is useful when the mapping of an individual page is changed.

A.2.2.6 Combining Segment and Page Translation

Figure A-16 summarizes both stages of translation from a logical address to a physical address when paging is enabled. Options available in both stages of address translation can be used to support several different styles of memory management.

Figure A-16 Combining Segment and Page Address Translation



A.2.2.6.1 Flat Model

When the Am486 microprocessor is used to run software written without segments, it may be desirable to remove the segmentation features of the Am486 microprocessor. The Am486 microprocessor does not have a mode bit for disabling segmentation, but the same effect can be achieved by mapping the stack, code, and data spaces to the same range of linear addresses. The 32-bit offsets used by Am486 microprocessor instructions can cover the entire linear address space.

When paging is used, the segments can be mapped to the entire linear address space. If more than one program is being run at the same time, the paging mechanism can be used to give each program a separate address space.

A.2.2.6.2 Segments Spanning Several Pages

The architecture allows segments that are larger than the size of a page (4 Kbytes). For example, a large data structure may span thousands of pages. If paging were not used, access to any part of the data structure would require the entire data structure to be present in physical memory. With paging, only the page containing the part being accessed needs to be in memory.

A.2.2.6.3 Pages Spanning Several Segments

Segments also may be smaller than the size of a page. If one of these segments is placed in a page that is not shared with another segment, the extra memory is wasted. For example, a small data structure, such as a 1-byte semaphore, occupies 4 Kbytes if it is placed in a page by itself. If many semaphores are used, it is more efficient to pack them into a single page.

A.2.2.6.4 Non-Aligned Page and Segment Boundaries

The architecture does not enforce any correspondence between the boundaries of pages and segments. A page may contain the end of one segment and the beginning of another. Likewise, a segment may contain the end of one page and the beginning of another.

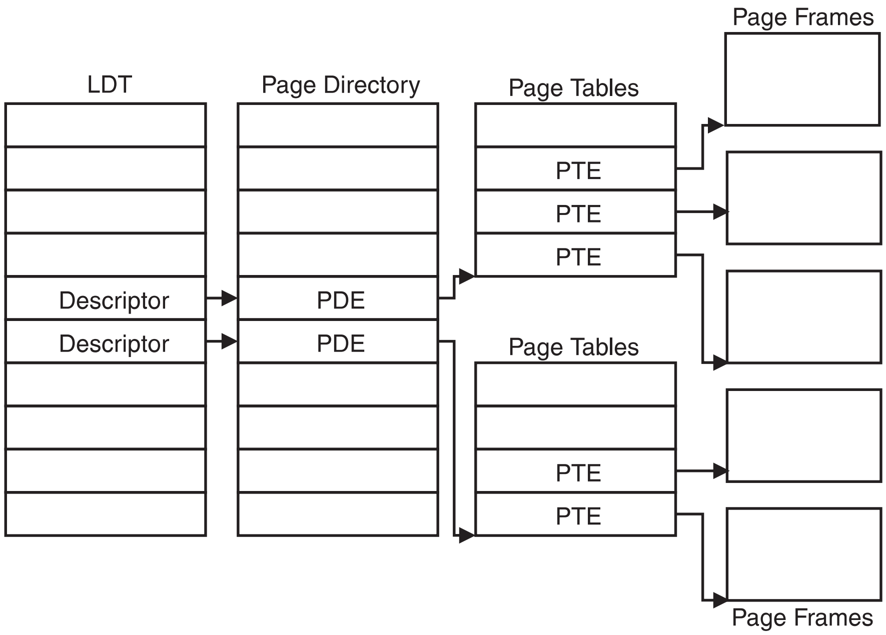
A.2.2.6.5 Aligned Page and Segment Boundaries

Memory-management software may be simpler and more efficient if it enforces some alignment between page and segment boundaries. For example, if a segment that may fit in one page is placed in two pages, there may be twice as much paging overhead to support access to that segment.

A.2.2.6.6 Page-Table Per Segment

An approach to combining paging and segmentation that simplifies memory management software is to give each segment its own page table (see Figure A-17). This gives the segment a single entry in the page directory that provides the access control information for paging the segment.

Figure A-17 Separate Page Tables for Each Segment



A.2.3 Internal System Protection

The internal system protection mechanism allows the programmer to prevent interference between tasks. Protection can keep one task from overwriting the instructions or data of another task. During program development, the protection mechanism can also give a clearer picture of program bugs. When a program makes an unexpected reference to the wrong memory space, the protection mechanism can block the event and report its occurrence.

In end-user systems, the protection mechanism can guard against the possibility of software failures caused by undetected program bugs. If a program fails, its effects can be confined to a limited domain, protecting the operating system against damage. With the proper exception routines, the system can record diagnostic information and attempt automatic recovery.

Programmers can also apply protection to segments and pages. Two bits in a microprocessor register define the privilege level of the program currently running (called the current privilege level or CPL). The CPL is checked during address translation for segmentation and paging.

Although there is no control register or mode bit for turning off the protection mechanism, the same effect can be achieved by assigning privilege level 0 (the highest level of privilege) to all segment selectors, segment descriptors, and page table entries.

A.2.3.1 Segment-Level Protection

Protection provides the ability to limit the amount of interference that a malfunctioning program can inflict on other programs and their data. Protection is a valuable aid in software development because it allows software tools (operating system, debugger, etc.) to survive in memory, undamaged. When an application program fails, the software is available to report diagnostic messages and the debugger is available for post-mortem analysis of memory and registers. In production, protection can make software more reliable by giving the system an opportunity to initiate recovery procedures.

Each memory reference is checked to verify that it satisfies the protection checks. All checks are made before the memory cycle is started; any violation prevents the cycle from starting and results in an exception. Because checks are performed in parallel with address translation, there is no performance penalty. There are five protection checks:

- Type check
- Limit check
- Restriction of addressable domain
- Restriction of procedure entry points
- Restriction of instruction set

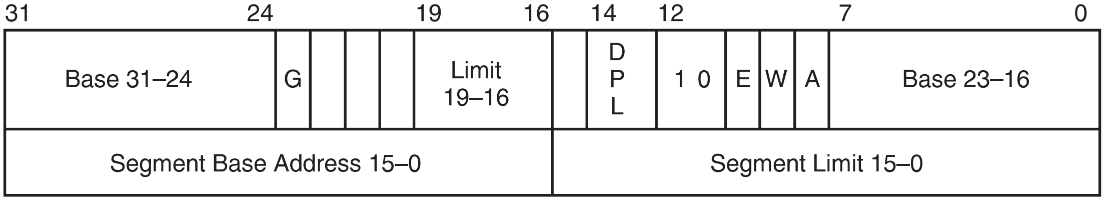
A protection violation results in an exception. This chapter describes the protection violations that lead to exceptions.

A.2.3.2 Segment Descriptors and Protection

Figure A-18 shows the fields of a segment descriptor which are used by the protection mechanism. Individual bits in the Type field also are referred to by the names of their functions.

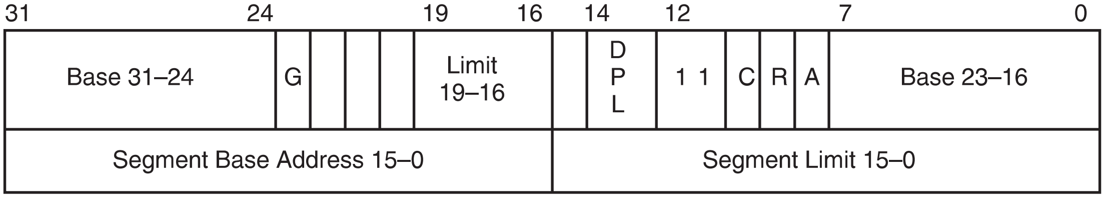
Figure A-18 Description Fields Used for Protection

Data Segment Descriptor



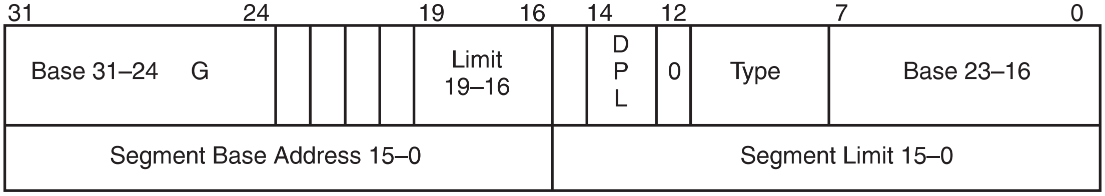
Base	Segment Base Address
G	Granularity
Limit	Segment Limit
DPL	Descriptor Privilege Level (00 = Highest – 11 = Lowest)
1 0	Reserved — always 10
E	Expand down
W	Writable
A	Accessed

Code Segment Descriptor



Base	Segment Base Address
G	Granularity
Limit	Segment Limit
DPL	Descriptor Privilege Level (00 = Highest – 11 = Lowest)
1 1	Reserved — always 11
C	Conforming
R	Readable
A	Accessed

System Segment Descriptor



Base	Segment Base Address
G	Granularity
Limit	Segment Limit
DPL	Descriptor Privilege Level (00 = Highest – 11 = Lowest)
0	Reserved, always 0
Type	Segment Type

Protection parameters are placed in the descriptor when it is created. In general, application programmers do not need to be concerned about protection parameters.When a program loads a segment selector into a segment register, the microprocessor loads both the base address of the segment and the protection information. The invisible part of each segment register stores the base, limit, type, and privilege level. While this information is resident in the segment register, subsequent protection checks on the same segment can be performed with no performance penalty.

A.2.3.2.1 *Type Checking*

In addition to the descriptors for application code and data segments, the Am486 microprocessor has descriptors for system segments and gates. These are data structures used for managing tasks and exceptions/interrupts. Table A-2 lists all the types defined for system segments and gates.

Note: *Not all descriptors define segments; gate descriptors hold pointers to procedure entry points.*

Table A-2 System Segment and Gate Types

Type	Description
0	Reserved
1	Available 80286 TSS
2	LDT
3	Busy 80286 TSS
4	Call Gate
5	Task Gate
6	80286 Interrupt Gate
7	80286 Trap Gate
8	Reserved
9	Available Am486 processor TSS
10	Reserved
11	Busy Am486 processor TSS
12	Am486 processor Call Gate
13	Reserved
14	Am486 processor Interrupt Gate
15	Am486 processor Task Gate

The Type fields of code and data segment descriptors include bits that further define the purpose of the segment (see Figure A-18):

- The Writable bit in a data-segment descriptor controls whether programs can write to the segment.
- The Readable bit in an executable-segment descriptor specifies whether programs can read from the segment (e.g., to access constants stored in the code space). A readable, executable segment may be read in two ways:
  - — With the CS register, by using a CS override prefix
  - — By loading a selector for the descriptor into a data-segment register (the DS, ES, FS, or GS registers)

Type checking can detect programming errors due to attempts to use segments in ways not intended by the programmer. The microprocessor examines type information under two circumstances:

- When a selector for a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; for example:
  - — The CS register only can be loaded with a selector for an executable segment.

— Selectors of executable segments that are not readable cannot be loaded into data-segment registers.

— Only selectors of writable data segments can be loaded into the SS register.

- Certain segments can be used by instructions only in certain predefined ways; for example:

— No instruction may write into an executable segment.

— No instruction may write into a data segment if the writable bit is not set.

— No instruction may read an executable segment unless the readable bit is set.

A.2.3.2.2 Limit Checking

The Limit field of a segment descriptor prevents programs from addressing outside the segment. The effective value of the limit depends on the setting of the G bit (Granularity bit). For data segments, the limit also depends on the E bit (Expansion Direction bit). The E bit is a designation for one bit of the Type field, when referring to data segment descriptors.

When the G bit is clear, the limit is the value of the 20-bit Limit field in the descriptor. In this case, the limit ranges from 0 to 0FFFFFh (2^{20} –1 or 1 Mbyte). When the G bit is set, the microprocessor scales the value in the Limit field by a factor of 2^{12}. In this case, the limit ranges from 0FFFh (2^{12} –1 or 4 Kbytes) to 0FFFFFFFFh (2^{32} – 1 or 4 Gbytes).

Note: *When scaling is used, the lower twelve bits of the address are not checked against the limit; when the G bit is set and the segment limit is 0, valid offsets within the segment are 0 through 4095.*

For all types of segments except expand-down data segments (stack segments), the value of the limit is one less than the size of the segment in bytes. The microprocessor causes a general-protection exception in any of these cases:

- Attempt to access a memory byte at an address > limit
- Attempt to access a memory word at an address > (limit – 1)
- Attempt to access a memory doubleword at an address > (limit – 3)

For expand-down data segments, the limit has the same function but is interpreted differently. In these cases, the range of valid offsets is from (limit + 1) to 2^{32} – 1. An expand-down segment has maximum size when the segment limit is 0.

Limit checking catches programming errors such as runaway subscripts and invalid pointer calculations. These errors are detected when they occur, so identification of the cause is easier. Without limit checking, these errors could overwrite critical memory in another module, and the existence of these errors would not be discovered until the damaged module crashed, an event that may occur long after the actual error. Protection can block these errors and report their source.

In addition to limit checking on segments, there is limit checking on the descriptor tables. The GDTR and IDTR registers contain a 16-bit limit value. It is used by the microprocessor to prevent programs from selecting a segment descriptor outside the descriptor table. The limit of a descriptor table identifies the last valid byte of the table. Because each descriptor is 8 bytes long, a table that contains up to N descriptors should have a limit of 8N – 1.

A descriptor may be given a zero value. This refers to the first descriptor in the GDT, which is not used. Although this descriptor may be loaded into a segment register, any attempt to reference memory using this descriptor generates a general-protection exception.

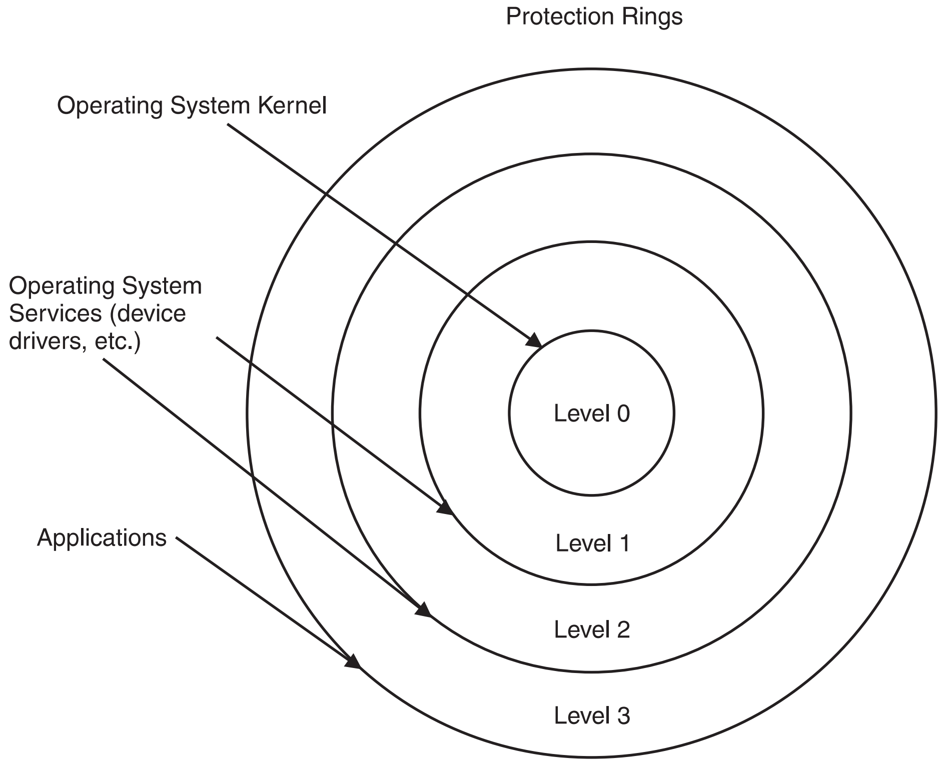
A.2.3.2.3 Privilege Levels

The protection mechanism recognizes four privilege levels: from 0 to 3. The greater numbers have lower privilege. If all other protection checks are satisfied, a general-protection exception occurs if a program with a higher privilege number attempts to access a segment with a lower privilege number. Although no control register or mode bit exists to disable the protection mechanism, you can achieve the same effect by assigning 0 to all operations. (The PE bit in the CR0 register does not enable the protection mechanism alone; it enables Protected Mode, the full 32-bit architecture execution mode. When Protected Mode is disabled, the microprocessor operates in Real Address Mode.)

You can use privilege levels to improve operating system reliability. By giving the operating system the highest privilege level, it is protected from damage by bugs in other programs. If a program crashes, the operating system can generate a diagnostic message and attempt recovery procedures. Another level of privilege can be established for other parts of the system software, such as the programs that handle peripheral devices, both in BIOS and specific device drivers. Device drivers should be given an intermediate privilege level between the operating system and the application programs. This protects both the operating system from errors in the drivers or BIOS, and it protects the drivers from bugs in application programs. Application programs are given the lowest privilege level.

Figure A-19 shows how these levels of privilege can be interpreted as rings of protection. The center is for the segments containing the most critical software, usually the kernel of an operating system. Outer rings are for less critical software.

Figure A-19 Protection Rings



The following data structures contain privilege levels:

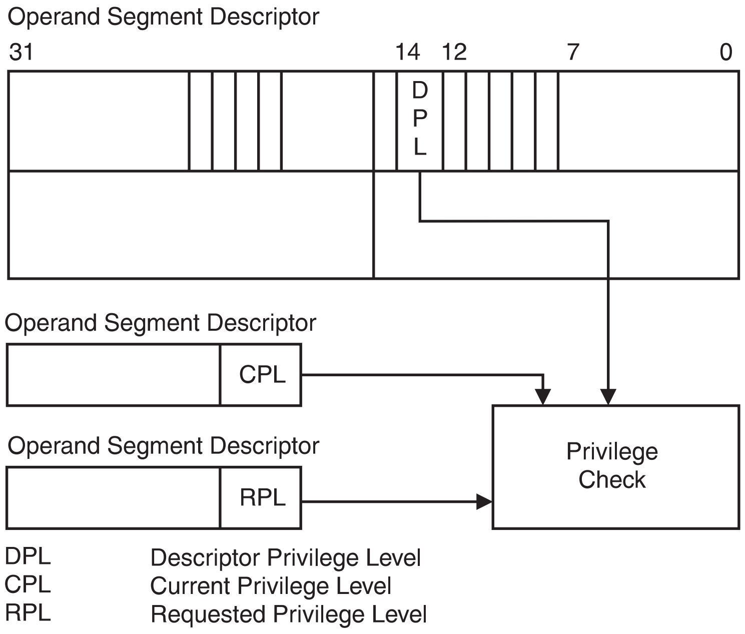
- The lowest two bits of the CS segment register hold the current privilege level (CPL). This is the privilege level of the program being run. The lowest two bits of the SS register also hold a copy of the CPL. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The CPL changes when control is transferred to a code segment with a different privilege level.
- Segment descriptors contain a field called the descriptor privilege level (DPL). The DPL is the privilege level applied to a segment.
- Segment selectors contain a field called the requester privilege level (RPL). The RPL is intended to represent the privilege level of the procedure that created the selector. If the RPL is a less privileged level than the CPL, it overrides the CPL. When a more privileged program receives a segment selector from a less privileged program, the RPL causes the memory access to take place at the less privileged level.

Privilege levels are checked when the selector of a descriptor is loaded into a segment register. The checks used for data access differ from those used for transfers of execution among executable segments; therefore, the two types of access are considered separately in the following sections.

A.2.3.3 Restricting Access to Data

To address operands in memory, a segment selector for a data segment must be loaded into a data-segment register (the DS, ES, FS, GS, or SS registers). The microprocessor checks the segment's privilege levels. The check is performed when the segment selector is loaded. As Figure A-20 shows, three different privilege levels enter into this type of privilege check.

Figure A-20 Privilege Check for Data Access


Operand Segment Descriptor
31
14
12
7
0
D
P
L
Operand Segment Descriptor
CPL
Operand Segment Descriptor
RPL
Privilege
Check
DPL Descriptor Privilege Level
CPL Current Privilege Level
RPL Requested Privilege Level

The three privilege levels that are checked are:

1. The CPL (current privilege level) of the program—this is held in the two least-significant bit positions of the CS register.
2. The DPL (descriptor privilege level) of the segment descriptor of the segment containing the operand
3. The RPL (requester's privilege level) of the selector used to specify the segment containing the operand—this is held in the two lowest bit positions of the segment register used to access the operand (the SS, DS, ES, FS, or GS registers). If the operand is in the stack segment, the RPL is the same as the CPL.

Instructions may load a segment register only if the DPL of the segment is the same or a less privileged level (greater privilege number) than the less privileged of the CPL and the selector's RPL.

The addressable domain of a task varies as its CPL changes. When the CPL is 0, data segments at all privilege levels are accessible; when the CPL is 1, only data segments at privilege levels 1 through 3 are accessible; when the CPL is 3, only data segments at privilege level 3 are accessible.

It may be desirable to store data in a code segment, for example, when both code and data are provided in ROM. Code segments may legitimately hold constants; it is not possible to write to a segment defined as a code segment, unless a data segment is mapped to the same address space. The following methods of accessing data in code segments are possible:

- Load a data-segment register with a segment selector for a non-conforming, readable, executable segment.
- Load a data-segment register with a segment selector for a conforming, readable, executable segment.
- Use a code-segment override prefix to read a readable, executable segment whose selector already is loaded in the CS register.

The same rules for access to data segments apply to case 1. Case 2 is always valid because the privilege level of a code segment with a set Conforming bit is effectively the same as the CPL, regardless of its DPL. Case 3 is always valid because the DPL of the code segment selected by the CS register is the CPL.

A.2.3.4 Restricting Control Transfers

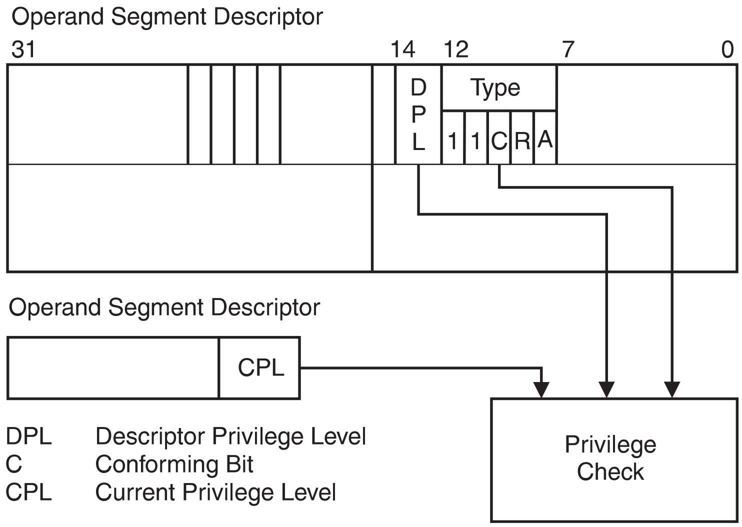
Control transfers are provided by the JMP, CALL, RET, INT, and IRET instructions, as well as by the exception and interrupt mechanisms. This section discusses only the JMP, CALL, and RET instructions.

The “near” forms of the JMP, CALL, and RET instructions transfer program control within the current code segment, and therefore are subject only to limit checking. The microprocessor checks that the destination of the JMP, CALL, or RET instruction does not exceed the limit of the current code segment. This limit is cached in the CS register, so protection checks for near transfers require no performance penalty.

The operands of the “far” forms of the JMP and CALL instruction refer to other segments, so the microprocessor performs privilege checking. There are two ways a JMP or CALL instruction can refer to another segment:

- The operand selects the descriptor of another executable segment.
- The operand selects a call gate descriptor.

Figure A-21 Privilege Check for Control Transfer Without Gate


Operand Segment Descriptor
31
14
12
7
0
D
P
L
Type
1
1
C
R
A
Operand Segment Descriptor
CPL
Privilege Check
DPL Descriptor Privilege Level
C Conforming Bit
CPL Current Privilege Level

As Figure A-21 shows, two different privilege levels enter into a privilege check for a control transfer that does not use a call gate:

- The CPL (current privilege level)
- The DPL of the descriptor of the destination code segment

Normally the CPL is equal to the DPL of the segment that the microprocessor is currently executing. The CPL may, however, be greater (less privileged) than the DPL if the current code segment is a conforming segment (as indicated by the Type field of its segment descriptor). A conforming segment runs at the privilege level of the calling procedure. The microprocessor keeps a record of the CPL cached in the CS register; this value can be different from the DPL in the segment descriptor of the current code segment.

The microprocessor only permits a JMP or CALL instruction directly into another segment if one of the following privilege rules is satisfied:

- The DPL of the segment is equal to the current CPL.
- The segment is a conforming code segment, and its DPL is less (higher privilege) than the current CPL.

Conforming segments are used for programs, such as math libraries and some kinds of exception handlers, that support applications but do not require access to protected system facilities. When control is transferred to a conforming segment, the CPL does not change, even if the selector used to address the segment has a different RPL. This is the only condition in which the CPL may be different from the DPL of the current code segment.

Most code segments are non-conforming. For these segments, control can be transferred without a gate only to other code segments at the same level of privilege. It is sometimes necessary, however, to transfer control to higher privilege levels. This is accomplished with the CALL instruction using call-gate descriptors. The JMP instruction may never transfer control to a non-conforming segment whose DPL does not equal the CPL.

A.2.3.5 Gate Descriptors

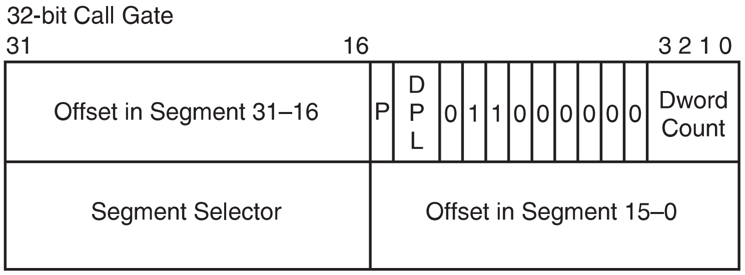
To provide protection for control transfers among executable segments at different privilege levels, the Am486 microprocessor uses gate descriptors. There are four kinds of gate descriptors:

- Task gates
- Trap gates
- Interrupt gates
- Call gates

Task gates are used for task switching. Trap gates and interrupt gates are used by exceptions and interrupts. Call gates are a form of protected control transfer. They are used for control transfers between different privilege levels. They only need to be used in systems in which more than one privilege level is used.

Figure A-22 illustrates the format of a call gate.

Figure A-22 Call Gate Format



P Segment Present
DPL Descriptor Privilege Level

A call gate has two main functions:

- To define an entry point of a procedure
- To specify the privilege level required to enter a procedure

Call gate descriptors are used by CALL and JUMP instructions in the same manner as code segment descriptors. When the hardware recognizes that the destination segment selector refers to a gate descriptor, the call gate contents determine the operation of the instruction. A call gate descriptor may reside in the GDT or in an LDT, but not in the IDT. The selector and offset fields of a gate form a pointer to the entry point of a procedure. A call gate guarantees that all control transfers to other segments go to a valid entry point, rather than to the middle of a procedure (or worse, to the middle of an instruction). The operand of the control transfer instruction is not the segment selector and is not offset within the segment to the procedure's entry point. Instead, the segment selector points to a gate descriptor, and the offset is not used. Figure A-23 shows this form of addressing. As shown in Figure A-24, four different privilege levels are used to check the validity of a control transfer through a call gate.

Figure A-23 Call Gate Mechanism

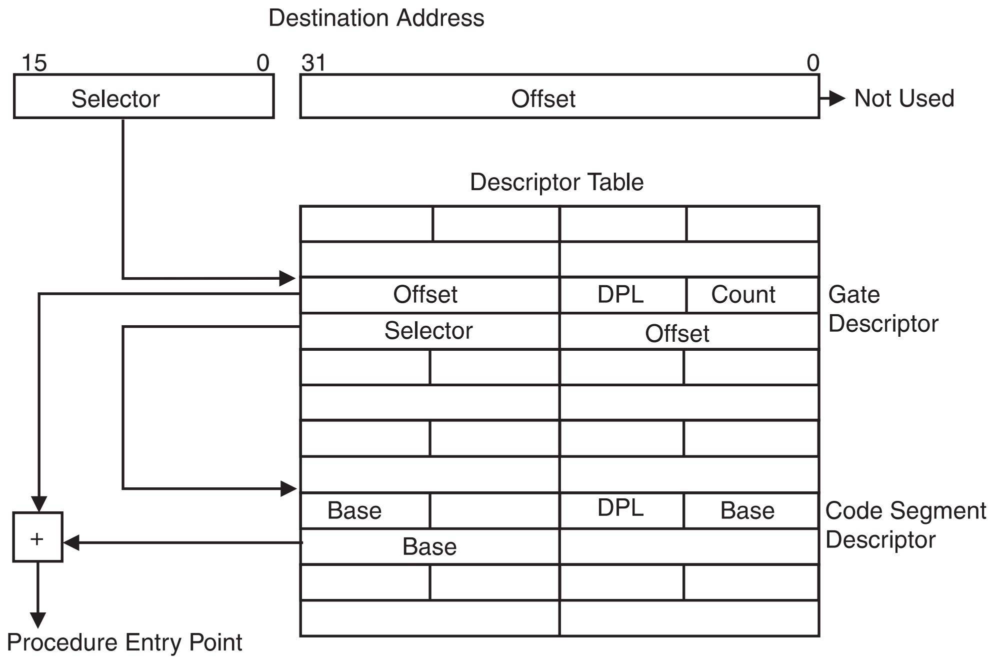
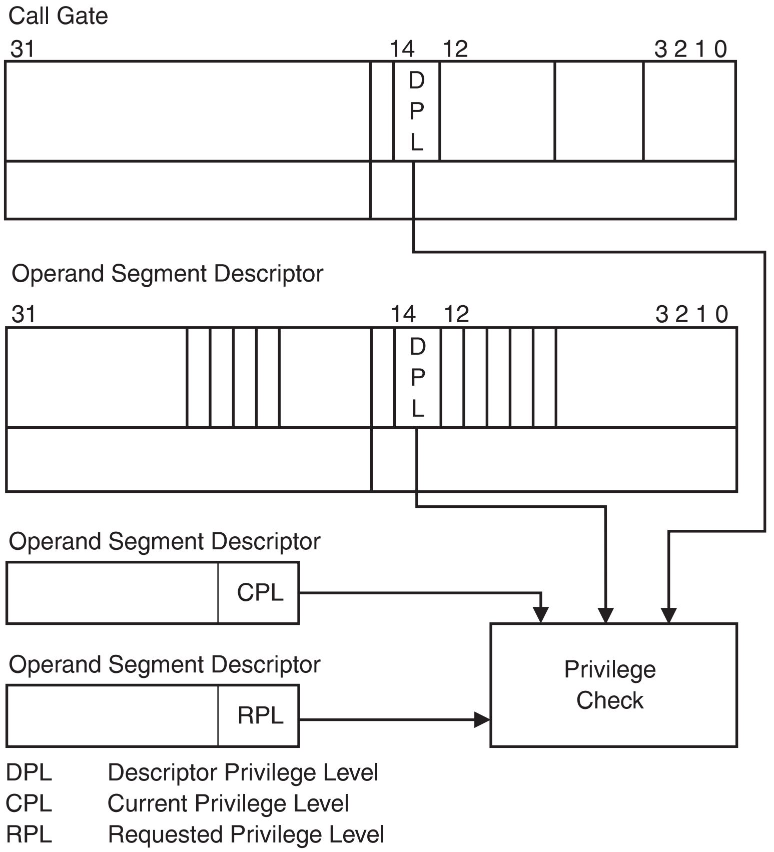


Figure A-24 Privilege Check for Control Transfer with Call Gate


Call Gate
31
14
12
3 2 1 0
D
P
L
Operand Segment Descriptor
31
14
12
3 2 1 0
D
P
L
Operand Segment Descriptor
CPL
Operand Segment Descriptor
RPL
Privilege
Check
DPL Descriptor Privilege Level
CPL Current Privilege Level
RPL Requested Privilege Level

The privilege levels checked during a transfer of execution through a call gate are:

- The CPL (current privilege level)
- The RPL (requester's privilege level) of the segment selector used to specify the call gate
- The DPL (descriptor privilege level) of the gate descriptor
- The DPL of the segment descriptor of the destination code segment

The DPL field of the gate descriptor determines the privilege levels that can access the gate. One code segment can have procedures used by different privilege levels. For example, an operating system may have some services used by both the operating system and application software, such as routines to handle character I/O, while other services may be for use only by the operating system itself, such as routines to initialize device drivers.

Gates can be used for control transfers to higher privilege levels or to the same privilege level (though they are not necessary for same-level transfers). Only CALL instructions can use gates to transfer to higher privilege levels. A JMP instruction may use a gate only to transfer control to a code segment with the same privilege level, or to a conforming code segment with the same or a higher privilege level.

To use a JMP instruction to transfer to a non-conforming segment, both of the following privilege rules must be satisfied; otherwise, a general-protection exception occurs:

- MAX (CPL,RPL) ≤ gate DPL
- Destination code segment DPL = CPL

For a CALL instruction (or for a JMP instruction to a conforming segment), both of the following privilege rules must be satisfied; otherwise, a general-protection exception occurs.

- MAX (CPL,RPL) ≤ gate DPL
- Destination code segment DPL ≤ CPL

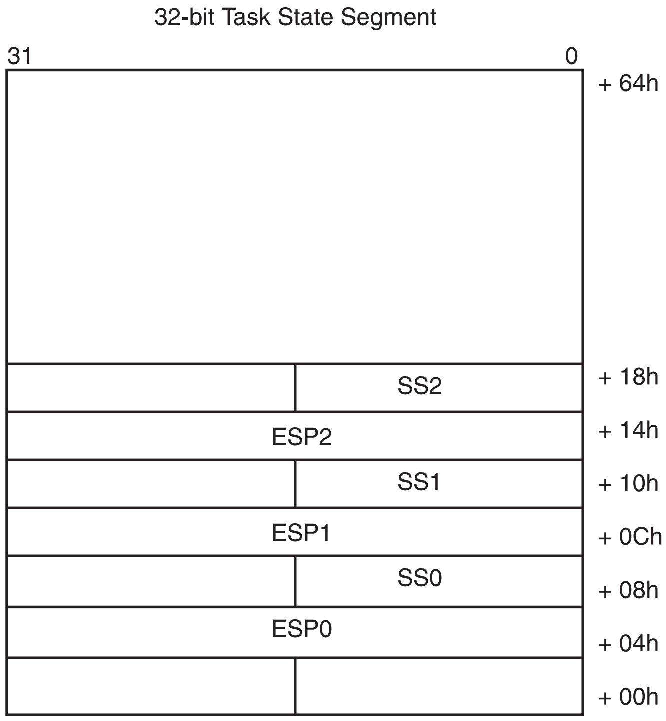
A.2.3.5.1 Stack Switching

A procedure call to a more privileged level does the following:

- Changes the CPL
- Transfers control (execution)
- Switches stacks

All inner protection rings (privilege levels 0, 1, and 2) have their own stacks for receiving calls from less privileged levels. If the caller were to provide the stack and the stack were too small, the called procedure might fail due to insufficient stack space. The system design avoids this problem by creating a new stack when a call is made to a more privileged level. The mechanism creates a new stack, copies the parameters from the old stack, and saves the register contents; then execution proceeds normally. When the procedure returns, the contents of the saved registers restore the original stack.

Figure A-25 Initial Stack Pointers in a TSS



The microprocessor finds the space to create new stacks using the task state segment (TSS) (see Figure A-25). Each task has its own TSS. The TSS contains initial stack pointers for the inner protection rings. The operating system is responsible for creating each TSS and initializing its stack pointers. An initial stack pointer consists of a segment selector and an initial value for the ESP register (an initial offset into the segment). The initial stack pointers are strictly read-only values. The microprocessor does not change them while the task runs. These stack pointers are used only to create new stacks when calls are made to more privileged levels. These stacks disappear when the called procedure returns. The next time the procedure is called, a new stack is created using the initial stack pointer.

When a call gate is used to change privilege levels, a new stack is created by loading an address from the TSS. The microprocessor uses the DPL of the destination code segment (the new CPL) to select the initial stack pointer for privilege level 0, 1, or 2.

The DPL of the new stack segment must equal the new CPL; if not, a stack-fault exception is generated. It is the responsibility of the operating system to create stacks and stack-segment descriptors for all privilege levels that are used. The stacks must be read/write as specified in the Type field of their segment descriptors. They must contain enough space, as specified in the Limit field, to hold the contents of the SS and ESP registers, the return address, and the parameters and temporary variables required by the called procedure.

As with calls within a privilege level, parameters for the procedure are placed on the stack. The parameters are copied to the new stack. The parameters can be accessed within the called procedure using the same relative addresses that would have been used if no stack switching had occurred. The count field of a call gate tells the microprocessor how many doublewords (up to 31) to copy from the caller's stack to the stack of the called procedure. If the count is 0, no parameters are copied.

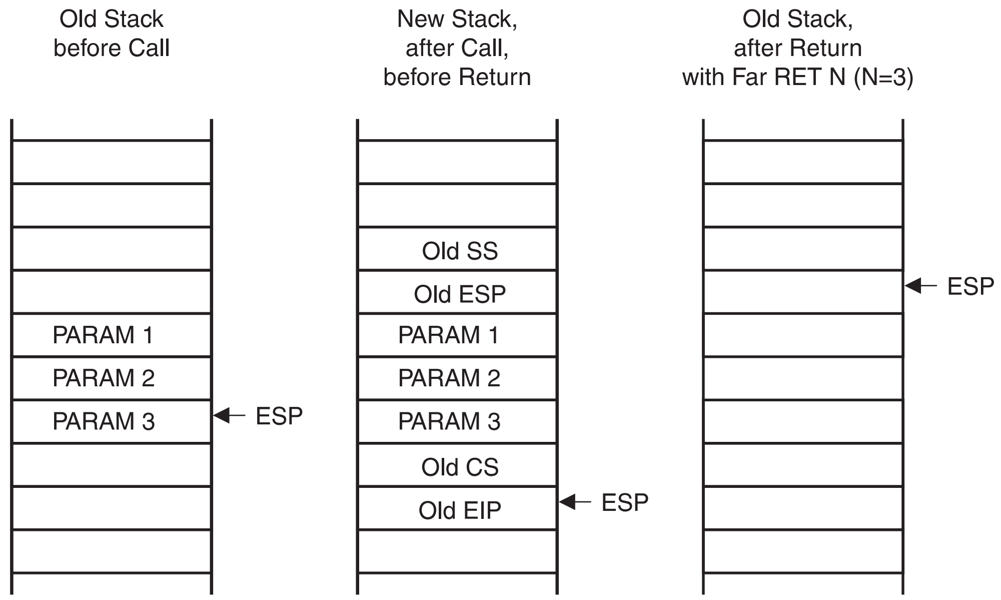
If more than 31 doublewords of data need to be passed to the called procedure, one of the parameters can be a pointer to a data structure, or the saved contents of the SS and ESP registers may be used to access parameters in the old stack space.

The microprocessor performs the following stack-related steps in executing a procedure call between privilege levels:

- The stack of the called procedure is checked to make certain it is large enough to hold the parameters and the saved contents of registers; if not, a stack exception is generated.
- The old contents of the SS and ESP registers are pushed onto the stack of the called procedure as two doublewords (the 16-bit SS register is zero-extended to 32 bits; the zero-extended upper word is AMD reserved; do not use).
- The parameters are copied from the stack of the caller to the stack of the called procedure.
- A pointer to the instruction after the CALL instruction (the old contents of the CS and EIP registers) is pushed onto the new stack. The contents of the SS and ESP registers after the call point to this return pointer on the stack.

Figure A-26 illustrates the stack frame before, during, and after a successful interlevel procedure call and return.

Figure A-26 Stack Frame During Interlevel CALL



The TSS does not have a stack pointer for a privilege level 3 stack, because a procedure at privilege level 3 cannot be called by a less privileged procedure. The stack for privilege level 3 is preserved by the contents of the SS and EIP registers that have been saved on the stack of the privilege level called from level 3.

A call using a call gate does not check the values of the words copied onto the new stack. The called procedure should check each parameter for validity. A later section discusses how the ARPL, VERR, VERW, LSL, and LAR instructions can be used to check pointer values.

A.2.3.5.2 Returning from a Procedure

The "near" forms of the RET instruction only transfer control within the current code segment and therefore, are subject only to limit checking. The microprocessor checks the offset to ensure that it does not exceed the current code segment limit.

The "far" form of the RET instruction pops the return address that was pushed onto the stack by an earlier far CALL instruction. Under normal conditions, the return pointer is valid. Nevertheless, the microprocessor performs privilege checking because the current procedure can alter the pointer or fail to maintain the stack properly. The RPL of the code-segment selector popped off the stack should have the privilege level of the calling procedure.

A return to another segment can change privilege levels, but only to a lower privilege level. When RET encounters a saved CS value whose RPL is numerically greater than the CPL (less privileged level), a return across privilege levels occurs. A return of this kind performs these steps:

- The checks shown in Table A-3 are made, and the CS, EIP, SS, and ESP registers are loaded with their former values, which were saved on the stack.
- The old contents of the SS and ESP registers (from the top of the current stack) are adjusted by the number of bytes indicated in the RET instruction. The resulting ESP value is not checked against the limit of the stack segment. An ESP value beyond the limit is not recognized until the next stack operation. (The returning procedure SS and ESP register contents are not preserved; normally, their values equal those in the TSS.)
- The DS, ES, FS, and GS segment register contents are checked. If any of these registers refer to segments whose DPL is less than the new CPL (excluding conforming code segments), the segment register is loaded with the null selector (Index = 0, TI = 0). The RET instruction itself does not signal exceptions in these cases, but any subsequent memory reference using a segment register with the null selector causes a general-protection exception. This prevents less privileged code from accessing more privileged segments using selectors left in the segment registers by a more privileged procedure.

Table A-3 Interlevel Return Checks

Type of Check	Exception Type	Error Code
Top-of-stack must be within stack segment limit	stack	0
Top-of-stack + 7 must be within stack segment limit	stack	0
RPL of return code segment must be greater than the CPL	protection	return CS
Return code segment must be non-null	protection	return CS
Return code segment descriptor must be within descriptor table limit	protection	return CS
Return segment descriptor must be a code segment	protection	return CS
Return code segment is present	protection	return CS
Return non-conforming code segment DPL must equal return code segment selector RPL; or return conforming code segment DPL must be less than or equal the return code segment selector RPL	protection	return CS
ESP + RET operand + 15 must be within the stack segment limit	protection	return CS
Segment descriptor at ESP+ RET operand +12 must be non-null	protection	return CS
Segment descriptor at ESP+ RET operand +12 must be within descriptor table limit	protection	return CS
Stack segment must be read/write	protection	return CS
Stack segment must be present	protection	return CS
Old stack segment DPL must equal old code segment RPL	protection	return CS
Old stack selector RPL must equal old stack segment CPL	protection	return CS

A.2.3.6 Instructions Reserved for the Operating System

Instructions that can affect the protection mechanism or influence general system performance can only be executed by trusted procedures. The Am486 microprocessor has two classes of such instructions:

- Privileged instructions—those used for system control
- Sensitive instructions—those used for I/O and I/O-related activities

A.2.3.6.1 Privileged Instructions

The instructions that affect protected facilities can be executed only when the CPL is 0 (most privileged). If one of these instructions is executed when the CPL is not 0, a general-protection exception is generated. These instructions include:

CLTS	– Clear Task-Switched Flag
HLT	– Halt Microprocessor
INVD	– Invalidate Cache
INVLPG	– Invalidate TLB Entry
LGDT	– Load GDT Register
LIDT	– Load IDT Register
LIDT	– Load LDT Register
LMSW	– Load Machine Status Word
LTR	– Load Task Register
MOV CR0	– Move to/from Control Register 0
MOV DRn	– Move to/from Debug Register n
MOV TRn	– Move to/from Test Register n
WBINVD	– Write Back and Invalidate Cache

A.2.3.6.2 Sensitive Instructions

Instructions that deal with I/O need to be protected, but they also need to be used by procedures executing at privilege levels other than 0 (the most privileged level). The mechanisms for protection of I/O operations are covered in detail in Section A.2.9.

A.2.3.7 Instructions for Pointer Validation

Pointer validation is necessary for maintaining isolation between privilege levels. It consists of the following steps:

1. Checks if the supplier of the pointer is allowed to access the segment.
2. Checks if the segment type is compatible with its use.
3. Checks if the pointer offset exceeds the segment limit.

Although the Am486 microprocessor automatically performs checks 2 and 3 during instruction execution, software must assist in performing the first check. The ARPL instruction is provided for this purpose. Software also can use steps 2 and 3 to check for potential violations, rather than waiting for an exception to be generated. The LAR, LSL, VERR, and VERW instructions are provided for this purpose.

An additional check, the alignment check, can be applied in user mode. When both the AM bit in CR0 and the AC flag are set, unaligned memory references generate exceptions. This is useful for programs that use the Low two bits of pointers to identify the type of data structure they address. For example, a subroutine in a math library may accept pointers to numeric data structures. If the type of this structure is assigned a code of 10 (binary) in the lowest two bits of pointers to this type, math subroutines can correct for the type code by adding a displacement of –10 (binary). If the subroutine should ever receive the wrong pointer type, an unaligned reference would be produced, which would generate an exception. Alignment checking accelerates the processing of programs written in symbolic-processing (i.e., Artificial Intelligence) languages such as Lisp, Prolog, Smalltalk, and C++. It can be used to speed up pointer tag type checking.

LAR (Load Access Rights) is used to verify that a pointer refers to a segment of a compatible privilege level and type. The LAR instruction has one operand, a segment selector for the descriptor whose access rights are to be checked. The segment descriptor must be readable at a privilege level that is numerically greater (less privileged) than the CPL and the selector's RPL. If the descriptor is readable, the LAR instruction gets the second doubleword of the descriptor, masks this value with 00FxFF00h, stores the result into the specified 32-bit destination register, and sets the Zero Flag (ZF). (The x indicates that the corresponding four bits of the stored value are undefined.) Once loaded, the access rights can be tested. All valid descriptor types can be tested by the LAR instruction. If the RPL or CPL is greater than the DPL, or if the segment selector would exceed the limit for the descriptor table, no access rights are returned and ZF is cleared. Conforming code segments may be accessed from any privilege level.

LSL (Load Segment Limit) allows software to test the limit of a segment descriptor. If the descriptor referenced by the segment selector (in memory or a register) is readable at the CPL, the LSL instruction loads the specified 32-bit register with a 32-bit, byte granular limit calculated from the concatenated limit fields and the G bit of the descriptor. This only can be done for descriptors that describe segments (data, code, task state, and local descriptor tables); gate descriptors are inaccessible. (Table A-4 lists in detail which types are valid and which are not.) Interpreting the limit is a function of the segment type. For example, downward-expandable data segments (stack segments) treat the limit differently than other kinds of segments. For both the LAR and LSL instructions, ZF is set if the load was successful; otherwise, ZF is cleared.

Table A-4 Valid Descriptor Types for LSL Instruction

Type Code	Descriptor Type	Valid?
0	Reserved	no
1	Reserved	no
2	LDT	yes
3	Reserved	no
4	Reserved	no
5	Task Gate	no
6	Reserved	no
7	Reserved	no
8	Reserved	no
9	Available Am486 processor TSS	yes
A	Reserved	no
B	Busy Am486 processor TSS	yes
C	Am486 processor Call Gate	no
D	Reserved	no
E	Am486 processor Interrupt Gate	no
F	Am486 processor Task Gate	no

Note: *Conforming segments are not checked for privilege level.*

A.2.3.7.1 *Descriptor Validation*

The Am486 microprocessor has two instructions, VERR and VERW, which determine whether a segment selector points to a segment that can be read or written using the CPL. Neither instruction causes a protection fault if the segment cannot be accessed.

VERR (Verify for Reading) verifies a segment for reading and sets ZF if that segment is readable using the CPL. The VERR instruction checks the following:

- The segment selector points to a segment descriptor within the bounds of the GDT or an LDT.
- The segment selector indexes to a code or data segment descriptor.
- The segment is readable and has a compatible privilege level.
- The privilege check for data segments and non-conforming code segments verifies that the DPL must be a less privileged level than either the CPL or the selector's RPL.

VERW (Verify for Writing) provides the same capability as the VERR instruction for verifying writability. Like the VERR instruction, the VERW instruction sets ZF if the segment can be written. The instruction verifies the descriptor is within bounds, is a segment descriptor, is writable, and has a DPL that is a less privileged level than either the CPL or the selector's RPL. Code segments are never writable, whether conforming or not.

A.2.3.7.2 *Pointer Integrity and RPL*

The requester's privilege level (RPL) can prevent accidental use of pointers that can cause system lockup when moving to a higher privilege code from a lower privilege level.

A common example is a file system procedure, FREAD (file_id, n_bytes, buffer_ptr). This hypothetical procedure reads data from a disk file into a buffer, overwriting whatever is already there. It services requests from programs operating at the application level, but it must run in a privileged mode (not level 3) in order to read from the system I/O buffer. If the application program passes a bad buffer pointer to the procedure that points to critical code or data in a privileged address space, the procedure can lockup the system.

Use of the RPL can avoid this problem. The RPL allows a privilege override to be assigned to a selector. This privilege override is the privilege level of the code segment that generates the segment selector. In the above example, the RPL is the CPL of the application program that called the system level procedure. The Am486 microprocessor automatically checks any segment selector loaded into a segment register to determine whether its RPL allows access.

To take advantage of the microprocessor's checking of the RPL, the called procedure need only check that all segment selectors passed to it have an RPL for the same or a less privileged level as the original caller's CPL. This guarantees that the segment selectors are not more privileged than their source. If a selector is used to access a segment that the source would not be able to access directly (i.e., the RPL is less privileged than the segment's DPL), a general-protection exception is generated when the selector is loaded into a segment register.

ARPL (Adjust Requested Privilege Level) adjusts the RPL field of a segment selector to be the larger (less privileged) of its original value and the value of the RPL field for a segment selector stored in a general register. The RPL fields are the two least-significant bits of the segment selector and the register. The latter normally is a copy of the caller's CS register on the stack. If the adjustment changes the selector's RPL, ZF is set; otherwise, ZF is cleared.

A.2.3.8 Page-Level Protection

Protection applies to both segments and pages. When the flat model for memory segmentation has been used, page-level protection prevents programs from interfering with each other.

Each memory reference is checked to verify that it satisfies the protection checks. All checks are made before the memory cycle is started; any violation prevents the cycle from starting and results in an exception. Because checks are performed in parallel with address translation, there is no performance penalty. There are two page-level protection checks:

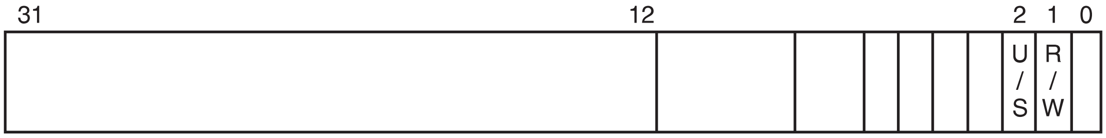
- Restriction of addressable domain
- Type checking

A protection violation results in an exception. See Section A.2.8 for an explanation of the exception mechanism. This section describes the protection violations that lead to exceptions.

A.2.3.8.1 Page-Table Entries Hold Protection Parameters

Figure A-27 highlights the fields of a page table entry that control access to pages. The protection checks are applied for both first and second-level page tables.

Figure A-27 Protection Holds



U/S User/Supervisor
R/W Read/Write

Privilege is interpreted differently for pages and segments. With segments, there are four privilege levels, ranging from 0 (most privileged) to 3 (least privileged). With pages, there are two levels of privilege:

- Supervisor level (U/S = 0): for the operating system, other system software (such as device drivers), and protected system data (such as page tables).
- User level (U/S = 1): for application code and data. The privilege levels used for segmentation are mapped into the privilege levels used for paging. If the CPL is 0, 1, or 2, the microprocessor is running at supervisor level. If the CPL is 3, the microprocessor is running at user level. When the microprocessor is running at supervisor level, all pages are accessible. When the microprocessor is running at user level, only pages from the user level are accessible.

Only two types of pages are recognized by the protection mechanism:

- Read-only access (R/W = 0)
- Read/write access (R/W = 1)

When the microprocessor is running at supervisor level with the WP bit in the CR0 register clear (its state following reset initialization), all pages are both readable and writable (write-protection is ignored). When the microprocessor is running at user level, only pages that belong to user level and are marked for read/write access are writable. User-level pages that are read/write or read-only are readable. Pages from the supervisor level are neither readable nor writable from user level. A general-protection exception is generated on any attempt to violate the protection rules.

Unlike the Am386DX microprocessor, the Am486 microprocessor allows user-mode pages to be write-protected against supervisor mode access. Setting the WP bit in the CR0 register enables supervisor-mode sensitivity to user-mode, write-protected pages. This feature is useful for implementing the copy-on-write strategy used by some operating systems, such as UNIX, for task creation (also called forking or spawning).

When a new task is created, it is possible to copy the entire address space of the parent task. This gives the child task a complete, duplicate set of the parent's segments and pages. The copy-on-write strategy saves memory space and time by mapping the child's segments and pages to the same segments and pages used by the parent task. A private copy of a page gets created only when one of the tasks writes to the page.

A.2.3.8.2 Combining Protection of Both Levels of Page Tables

For any one page, the protection attributes of its page directory entry (first-level page table) may differ from those of its second-level page table entry. The Am486 microprocessor checks the protection for a page by examining the protection specified in both the page directory (first-level page table) and the second-level page table. Table A-5 shows the protection provided by the possible combinations of protection attributes when the WP bit is clear.

Table A-5 Combined Page Directory and Page Table Protection

Page Directory Entry		Page Table Entry		Combined Effect	
Privilege	**Access Type**	**Privilege**	**Access Type**	**Privilege**	**Access Type**
User	Read-Only	User	Read-Only	User	Read-Only
User	Read-Only	User	Read/Write	User	Read-Only
User	Read/Write	User	Read-Only	User	Read-Only
User	Read/Write	User	Read/Write	User	Read/Write
User	Read-Only	Supervisor	Read-Only	User	Read-Only
User	Read-Only	Supervisor	Read/Write	User	Read-Only
User	Read/Write	Supervisor	Read-Only	User	Read-Only
User	Read/Write	Supervisor	Read/Write	User	Read/Write
Supervisor	Read-Only	User	Read-Only	User	Read-Only
Supervisor	Read-Only	User	Read/Write	User	Read-Only
Supervisor	Read/Write	User	Read-Only	User	Read-Only
Supervisor	Read/Write	User	Read/Write	User	Read/Write
Supervisor	Read-Only	Supervisor	Read-Only	Supervisor	Read-Only
Supervisor	Read-Only	Supervisor	Read/Write	Supervisor	Read/Write
Supervisor	Read/Write	Supervisor	Read-Only	Supervisor	Read/Write
Supervisor	Read/Write	Supervisor	Read/Write	Supervisor	Read/Write

A.2.3.8.3 Overrides to Page Protection

Certain accesses are checked as if they are privilege level 0 accesses, for any value of CPL:

- Access to segment descriptors (LDT, GDT, TSS and IDT)
- Access to inner stack during a CALL instruction, or exceptions and interrupts, when a change of privilege level occurs

A.2.3.9 Combining Page and Segment Protection

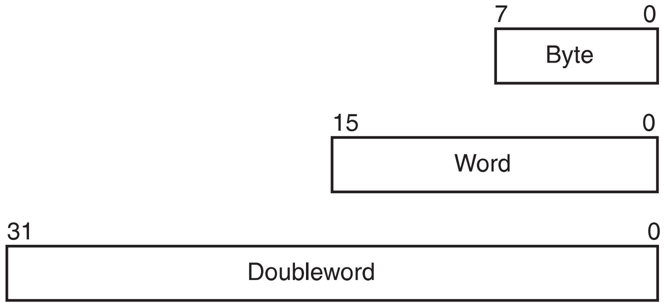
When paging is enabled, the Am486 microprocessor first evaluates segment protection, then evaluates page protection. If the microprocessor detects a protection violation at either the segment level or the page level, the operation does not go through; an exception occurs instead. If an exception is generated by segmentation, no paging exception is generated for the operation.

For example, it is possible to define a large data segment which has some parts that are read-only and other parts that are read/write. In this case, the page directory (or page table) entries for the read-only parts would have the U/S and R/W bits specifying no write access for all the pages described by that directory entry (or for individual pages specified in the second-level page tables). This technique might be used, for example, to define a large data segment, part of which is read-only (for shared data or constants in ROM). This approach defines a single “flat” data space in one large segment that uses “flat” pointers, but protects shared data that is mapped into the same virtual space using page-defined supervisor areas.

A.2.4 Data Types

There are two ways in which data is stored and used by the Am486 processor family. When stored in memory, the microprocessor accesses data using bytes, words, and doublewords (see Figure A-28). Instructions use the accessed data in multiple ways, both by accessing multiple sets of bytes, word, or doublewords and by reinterpreting the data stored in them (such as strings, signed and unsigned integers, BCD values, and real/floating-point numbers).

Figure A-28 Data Types in Memory



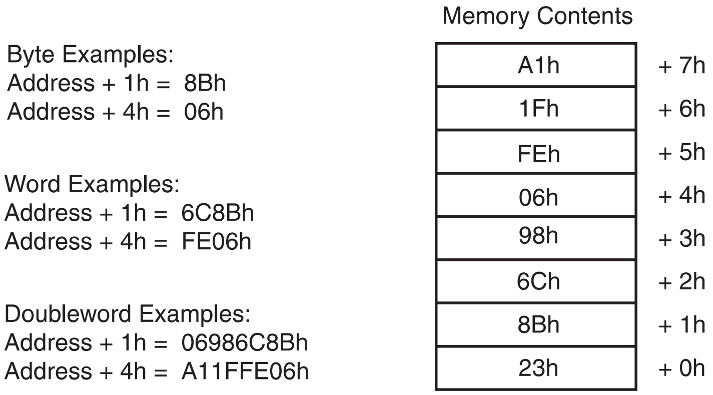
A.2.4.1 Data Types in Memory

Byte—8 bits. The bits are numbered 0 through 7, bit 0 being the least-significant bit (LSB).

Word—two bytes occupying any two consecutive addresses. A word contains 16 bits. The bits of a word are numbered from 0 through 15, bit 0 again being the least-significant bit. The byte containing bit 0 of the word is called the Low byte; the byte containing bit 15 is called the High byte. On the Am486 microprocessor, the Low byte is stored in the byte with the lower address. The address of the Low byte also is the address of the word. The address of the High byte is used only when the upper half of the word is being accessed separately from the lower half.

Doubleword—four bytes occupying any four consecutive addresses. A doubleword contains 32 bits. The bits of a doubleword are numbered from 0 through 31, bit 0 again being the least-significant bit. The word containing bit 0 of the doubleword is called the Low word; the word containing bit 31 is called the High word. The Low word is stored in the two bytes with the lower addresses. The address of the lowest byte is the address of the doubleword. The higher addresses are used only when the upper word is being accessed separately from the lower word, or when individual bytes are being accessed. Figure A-29 illustrates the arrangement of bytes within words and doublewords.

Figure A-29 Bytes, Words, and Doublewords in Memory



Byte Examples:
Address + 1h = 8Bh
Address + 4h = 06h

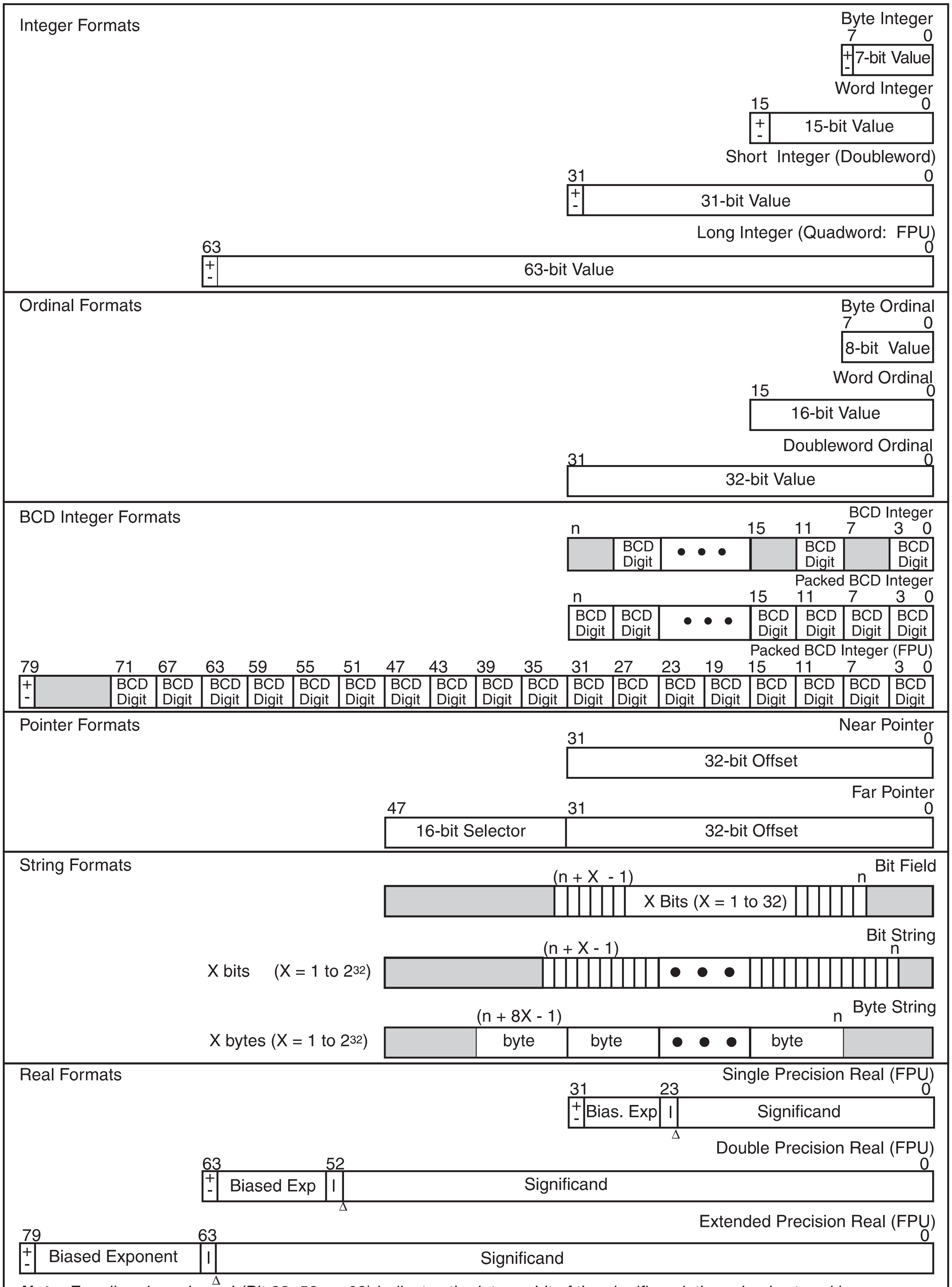
Word Examples:
Address + 1h = 6C8Bh
Address + 4h = FE06h

Doubleword Examples:
Address + 1h = 06986C8Bh
Address + 4h = A11FFE06h

Memory Contents	
A1h	+ 7h
1Fh	+ 6h
FEh	+ 5h
06h	+ 4h
98h	+ 3h
6Ch	+ 2h
8Bh	+ 1h
23h	+ 0h

***Note:** Words do not need to be aligned at even-numbered addresses and doublewords do not need to be aligned at addresses evenly divisible by four. This allows maximum flexibility in data structures (e.g., records containing mixed byte, word, and doubleword items) and efficiency in memory utilization. Because the Am486 microprocessor has a 32-bit data bus, communication between microprocessor and memory takes place as doubleword transfers aligned to addresses evenly divisible by four; the microprocessor converts doubleword transfers aligned to other addresses into multiple transfers. These unaligned operations reduce speed by requiring extra bus cycles. For maximum speed, data structures (especially stacks) should be designed so that, whenever possible, word operands are aligned to even addresses and doubleword operands are aligned to addresses evenly divisible by four.*

Figure A-30 Data Types



Note: *For all real numbers, I (Bit 23, 52, or 63) indicates the integer bit of the significand; the value is stored in temporary Real Numbers and implicit in Single and Double Precision Real Numbers.*

Δ indicates the location of the implicit binary point.

A.2.4.2 Operand Formats

Although bytes, words, and doublewords represent the way the microprocessor accesses data in memory, specialized instructions can interpret and manipulate this digital information in different forms. These operand forms include the following types (shown in Figure A-30):

- Integer—a signed binary number held in a 32-bit doubleword, 16-bit word, or 8-bit byte. All operations assume a two's complement representation. The sign bit is located in bit 7 in a byte, bit 15 in a word, and bit 31 in a doubleword. The sign bit is set for negative integers, clear for positive integers and zero. The value of an 8-bit integer is from –128 to + 127; a 16-bit integer from 32,768 to + 32,767; a 32-bit integer from $–2^{31}$ to + 2^{31}–1. When used by the FPU, they are automatically converted to the 80-bit extended real format, shown as a signed 79-bit integer in Figure A-30. All binary integers are exactly representable in the extended real format.
- Ordinal—an unsigned binary number contained in a 32-bit doubleword, 16-bit word, or 8-bit byte. The value of an 8-bit ordinal is from 0 to 255; a 16-bit ordinal from 0 to 65,535; a 32-bit ordinal from 0 to 2^{32} –1.
- Pointer—an offset address, or segment plus offset, used by a JUMP, conditional JUMP, LOOP, or conditional LOOP instruction.
  - — *Near Pointer:* A 32-bit logical address. A near pointer is an offset within a segment. Near pointers are used for all pointers in a flat memory model, or for references within a segment in a segmented model.
  - — *Far Pointer:* A 48-bit logical address consisting of a 16-bit segment selector and a 32-bit offset. Far pointers are used in a segmented memory model to access other segments.
- String—a contiguous sequence of bits, bytes, words, or doublewords. A string may contain from zero to 2^{32} – 1 bytes (4 Gbytes). The bit sequences can be one of two types:
  - — *Bit field:* A contiguous sequence of bits. A bit field may begin at any bit position of any byte and may contain up to 32 bits.
  - — *Bit string:* A contiguous sequence of bits. A bit string may begin at any bit position of any byte and may contain up to 2^{32} – 1 bits.
- BCD—a representation of a binary-coded decimal (BCD) digit in the range 0–9. Unpacked decimal numbers are stored as unsigned byte quantities. One digit is stored in each byte. The magnitude of the number is the binary value of the Low-order half-byte; values 0–9 are valid and are interpreted as the value of a digit. The High-order half-byte must be zero during multiplication and division; it may contain any value during addition and subtraction. Packed BCD formats use a representation of binary-coded decimal digits, each in the range 0–9. One digit is stored in each half-byte, two digits in each byte. The digit in bits 4–7 is more significant than the digit in bits 0–3. Values 0–9 are valid for a digit.
- Real—the Am486 microprocessor represents real numbers of the form:

$$(-1)^{s} \cdot 2E(b_{0\Delta}b_{1}b_{2}b_{3}...b_{p-1})$$

where:

^{s} = 0 or 1

E = any integer between E_{min} and E_{max}, inclusive

b_i = 0 or 1

$_{\Delta}$ = implicit binary point

p = number of bits of precision

The Am486 microprocessor stores real numbers in a three-field binary format that resembles scientific, or exponential, notation. The format consists of the following fields:

- The number's significant digits are held in the significand field, $b_{0\Delta}b_1b_2b_3...b_{p-1}$ (the term "significand" is analogous to the term "mantissa" used to describe floating-point numbers on some computers; $_\Delta$ indicates the implicit binary point in the bit field).
- The exponent field, e = E + bias, locates the binary point within the significant digits (and therefore determines the number's magnitude). The term "exponent" is analogous to the term "characteristic" used to describe floating-point numbers on some computers.
- The 1-bit sign field indicates whether the number is positive or negative. Negative numbers differ from positive numbers only in the sign bits of their significands.

Table A-6 shows how the real number 178.125 (decimal) is stored in the single real format. The table lists a progression of equivalent notations that express the same value to show how a number can be converted from one form to another. (The ASM386/486 and PL/M386/486 language translators perform a similar process when they encounter programmer-defined real number constants.)

Table A-6 Real Number Notation

Notation	Value		
Ordinary Decimal	178.125		
Scientific Decimal	1_Δ78125E2		
Scientific Binary	1_Δ0110010001E111		
Scientific Binary (Biased Exponent)	1_Δ0110010001E10000110		
Single Format (Normalized)	Sign	Biased Exponent	Significand
	0	10000110	01100100010000000000000 1_Δ(implicit)

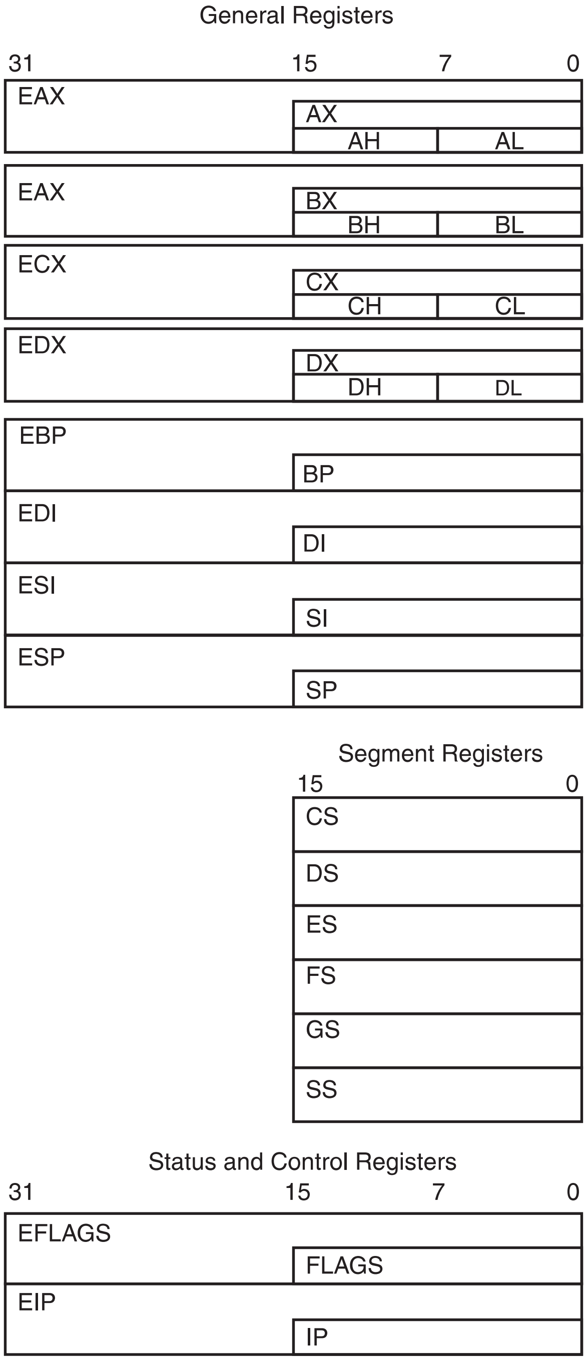
Note: *Not every decimal fraction has an exact binary equivalent. The decimal number 1/10, for example, cannot be expressed exactly in binary (just as the number 1/3 cannot be expressed exactly in decimal). When a translator encounters such a value, it produces a rounded binary approximation of the decimal value.*

A.2.5 Application Registers

The Am486 microprocessor contains sixteen registers that may be used by an application programmer. As Figure A-31 shows, these registers may be grouped as:

- *General Registers:* These eight 32-bit registers are free for use by the programmer.
- *Segment Registers:* These registers hold segment selectors associated with different forms of memory access. For example, there are separate segment registers for access to code and stack space. These six registers determine, at any given time, which segments of memory are currently available.
- *Status and Control Registers:* These registers report and allow modification of the state of the Am486 microprocessor.

Figure A-31 Application Register Set



In Am486DX and DX2 processors, there are also the following registers that are part of the floating-point unit (FPU) in the microprocessor:

- FPU Register Stack—eight 80-bit numeric registers that are organized as a register stack.
- Status and Control Registers—16-bit registers that contain the FPU status, control, and tag words.

- Error Pointers—five registers including two 16-bit registers that hold selectors for the last 16-bit operation, two 32-bit registers that hold selectors for the last 32-bit operation, and one 11-bit register that contains the opcode of the last non-control FPU instruction.

A.2.5.1 General Registers

The general registers are the 32-bit registers EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI. These registers are used to hold operands for logical and arithmetic operations. They also may be used to hold operands for address calculations (except the ESP register cannot be used as an index operand). The names of these registers are derived from the names of the general registers on the 8086 microprocessor, the AX, BX, CX, DX, BP, SP, SI, and DI registers. As Table A-7 shows, the Low 16 bits of the general registers can be referenced using these names.

Table A-7 Register Names

8 Bit	16 Bit	32 Bit
AL	AX	EAX
AH		
BL	BX	EBX
BH		
CL	CX	ECX
CH		
DL	DX	EDX
DH		
	SI	ESI
	DI	EDI
	BP	EBP
	SP	ESP

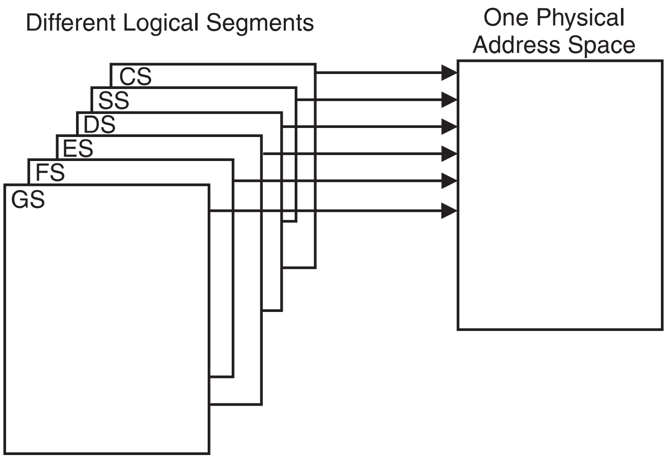
***Note:** The 8-bit registers are the upper and lower bytes of the first four 16-bit registers. The first four 16-bit registers are the lower words in the first four 32-bit registers. The position in this table is designed to suggest the register interrelationships.*

Each byte of the 16-bit registers AX, BX, CX, and DX also have other names. The byte registers are named AH, BH, CH, and DH (High bytes) and AL, BL, CL, and DL (Low bytes). All of the general-purpose registers are available for address calculations and for the results of most arithmetic and logical operations; however, a few instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. By assigning specific registers for these functions, the instruction set can be encoded more compactly. The instructions using specific registers include: double-precision multiply and divide, I/O, strings, translate, loop, variable shift and rotate, and stack operations.

A.2.5.2 Segment Registers

Segmentation gives system designers the flexibility to choose among various models of memory organization. Implementation of memory models is the subject of Section A.2.2. The segment registers contain 16-bit segment selectors, which index into tables in memory. The tables hold the base address for each segment, as well as other information regarding memory access. An unsegmented model is created by mapping each segment to the same place in physical memory (see Figure A-32).

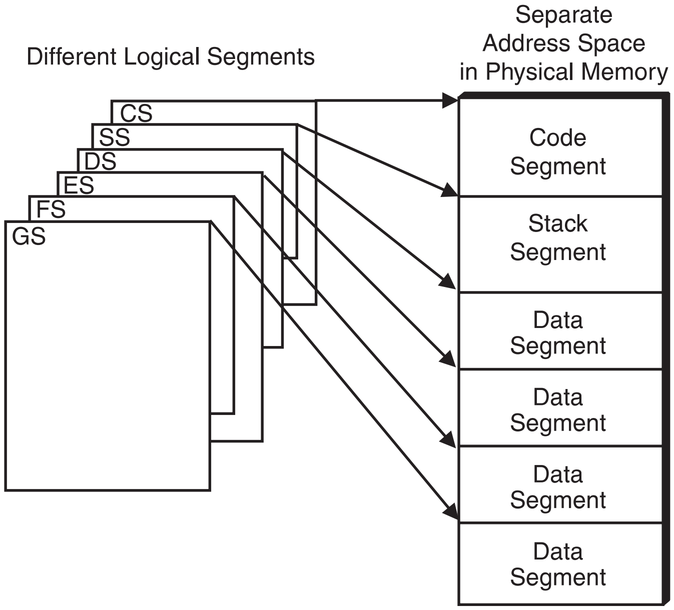
Figure A-32 Unsegmented Memory



At any instant, up to six segments of memory are immediately available. The segment registers CS, DS, SS, ES, FS, and GS hold the segment selectors for these six segments. Each register is associated with a particular kind of memory access (code, data, or stack). Each register specifies a segment (from among the six possible segments available to each program) used for its particular type of access (see Figure A-33). Other segments can be used by loading their segment selectors into the segment registers.

The segment containing the instructions being executed is called the code segment. Its segment selector is held in the CS register. The Am486 microprocessor fetches instructions from the code segment, using the contents of the EIP register as an offset into the segment. The CS register is loaded by interrupts, exceptions, and instructions that transfer control between segments (e.g., the CALL, IRET, and JMP instructions).

Figure A-33 Segmented Memory



All stack operations use the SS register to find the stack segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up stacks. Before a procedure is called, the SS allocates a stack to hold the return address, parameters passed by the calling routine, and temporary variables allocated by the procedure.

The DS, ES, FS, and GS registers allow as many as four data segments to be available simultaneously. Four data segments give efficient and secure access to different types of data structures. For example, separate data segments can be created for the data structures of the current module, data exported from a higher-level module, a dynamically created data structure, and data shared with another program. If a bug causes a program to run wild, the segmentation mechanism can limit the damage to only those segments allocated to the program. An operand within a data segment is addressed by specifying its offset either in an instruction or a general register.

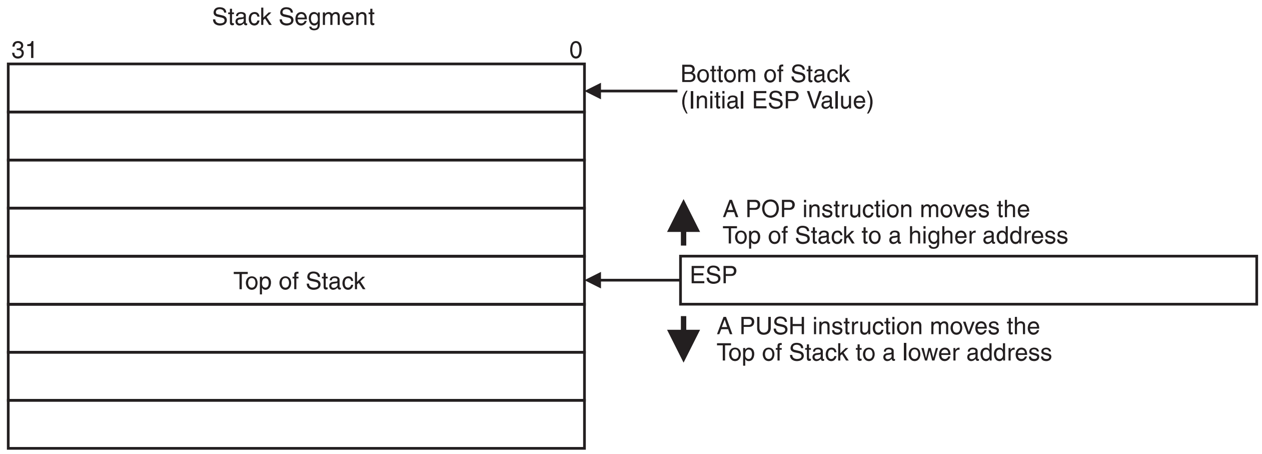
Depending on the structure of data (i.e., the way data is partitioned into segments), a program may require access to more than four data segments. To access additional segments, the DS, ES, FS, and GS registers can be loaded by an application program during execution. The only requirement is to load the appropriate segment register before accessing data in its segment.

A base address is kept for each segment. To address data within a segment, a 32-bit offset is added to the segment's base address. Once a segment is selected (by loading the segment selector into a segment register), an instruction only needs to specify the offset. Simple rules define which segment register is used to form an address when only an offset is specified.

Stack operations are supported by three registers:

- *Stack Segment (SS) Register:* Stacks reside in memory. The number of stacks in a system is limited only by the maximum number of segments. A stack may be up to 4 Gbytes long, the maximum size of a segment on the Am486 microprocessor. One stack is available at a time—the stack whose segment selector is held in the SS register. This is the current stack, often referred to simply as "the stack." The SS register is used automatically by the microprocessor for all stack operations.
- *Stack Pointer (ESP) Register:* The ESP register holds an offset to the top-of-stack (TOS) in the current stack segment. It is used by PUSH and POP operations, subroutine calls and returns, exceptions, and interrupts. When an item is pushed onto the stack (see Figure A-34), the microprocessor decrements the ESP register, then writes the item at the new TOS. When an item is popped off the stack, the microprocessor copies it from the TOS, then increments the ESP register. In other words, the stack grows down in memory toward lesser addresses.
- *Stack-Frame Base Pointer (EBP) Register:* The EBP register typically is used to access data structures passed on the stack. For example, on entering a subroutine, the stack contains the return address and some number of data structures passed to the subroutine. The subroutine adds to the stack whenever it needs to create space for temporary local variables. As a result, the stack pointer moves around as temporary variables are pushed and popped. If the stack pointer is copied into the base pointer before anything is pushed on the stack, the base pointer can be used to reference data structures with fixed offsets. If this is not done, the offset to access a particular data structure would change whenever a temporary variable is allocated or deallocated.

Figure A-34 Stacks


Stack Segment
31
0
Bottom of Stack
(Initial ESP Value)
A POP instruction moves the
Top of Stack to a higher address
Top of Stack
ESP
A PUSH instruction moves the
Top of Stack to a lower address

When the EBP register is used to address memory, the current stack segment is selected (i.e., the SS segment). Because the stack segment does not have to be specified, instruction encoding is more compact. The EBP register also can be used to address other segments.

Instructions, such as the ENTER and LEAVE instructions, are provided. These automatically set up the EBP register for convenient access to variables.

Instructions that use the stack implicitly (for example: POP EAX) also have a stack address-size attribute of either 16 or 32 bits. Instructions with a stack address-size attribute of 16 use the 16-bit SP stack pointer register; instructions with a stack address-size attribute of 32 bits use the 32-bit ESP register to form the address of the top of the stack. The stack address-size attribute is controlled by the B bit of the data-segment descriptor in the SS register. A value of zero in the B bit selects a stack address-size attribute of 16; a value of one selects a stack address-size attribute of 32.

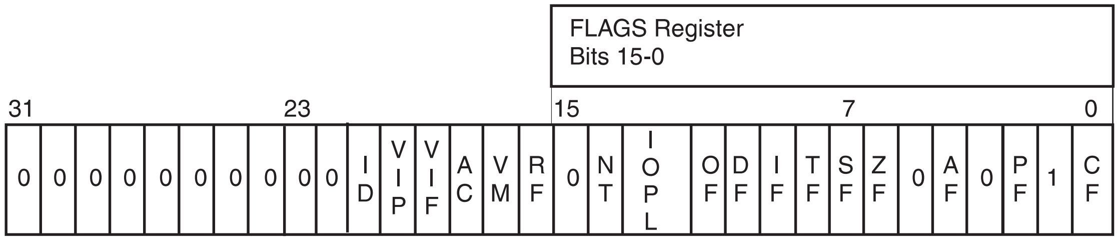
A.2.5.3 Status and Control Registers

The status and control registers include the 16-bit FLAGS and the 32-bit EFLAGS registers and the 16-bit IP and the 32-bit EIP registers. The 16-bit registers provide compatibility with systems using 16-bit memory access.

A.2.5.3.1 Flags Register

Condition codes (e.g., carry, sign, overflow) and mode bits are kept in a 32-bit register named EFLAGS. Figure A-35 defines the bits within this register. The flags control certain operations and indicate the status of the Am486 microprocessor. The flags may be considered in three groups: status flags, control flags, and system flags.

Figure A-35 EFLAGS Register



0 or 1	Reserved, always indicated value	
ID	ID Flag	System Flag
VIP	Virtual Interrupt Pending	System Flag
VIF	Virtual Interrupt Flag	System Flag
AC	Alignment Check	System Flag
VM	Virtual Mode	System Flag
RF	Resume Flag	System Flag
NT	Nested Task	System Flag
IOPL	I/O Privilege Level	System Flag
OF	Overflow Flag	Status Flag
DF	Direction Flag	Control Flag
IF	Interrupt-Enable Flag	System Flag
TF	Trap Flag	System Flag
SF	Sign Flag	Status Flag
ZF	Zero Flag	Status Flag
AF	Auxiliary-Carry Flag	Status Flag
PF	Parity Flag	Status Flag
CF	Carry Flag	Status Flag

The status flags of the EFLAGS register report the kind of result produced from the execution of arithmetic instructions. The MOV instruction does not affect these flags. Conditional jumps and subroutine calls allow a program to sense the state of the status flags and respond to them. For example, when the counter controlling a loop is decremented to zero, the state of ZF changes, and this change can be used to suppress the conditional jump to the start of the loop. The status flags are shown in Table A-8.

Table A-8 Status Flags

Name	Purpose	Condition Reported
OF	overflow	Result exceeds positive or negative limit of number range
SF	sign	Result is negative (less than zero)
ZF	zero	Result is zero
AF	auxiliary carry	Carry out of bit position 3 (used for BCD)
PF	parity	Low byte of result has even parity (even number of set bits)
CF	carry	Carry out of most-significant bit of result

The control flag DF of the EFLAGS register causes string instructions to auto-decrement (i.e., to process strings from High addresses to Low addresses). Clearing DF causes string instructions to auto-increment (i.e., to process strings from Low addresses to High addresses).

A.2.5.3.2 Instruction Pointer

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to execute. The instruction pointer is not directly available to the programmer; it is controlled implicitly by control transfer instructions (jumps, returns, etc.), interrupts, and exceptions. The EIP register advances from one instruction boundary to the next.

Because of instruction prefetching, the instruction boundary is only an approximate indication of the bus activity that loads instructions into the microprocessor. The Am486 microprocessor does not fetch single instructions. The microprocessor prefetches aligned 128-bit blocks of instruction code in advance of instruction execution (an aligned 128-bit block begins at an address that is clear in its Low four bits). These blocks are fetched without regard to the boundaries between instructions. By the time an instruction starts to execute, it already has been loaded into the microprocessor and decoded. This is a performance feature, because it allows instruction execution to be overlapped with instruction prefetch and decode.

When a jump or call executes, the microprocessor prefetches the entire aligned block containing the destination address and discards instructions that are already prefetched or decoded. This can be a benefit because the microprocessor does not generate an exception until the causative code actually executes. So, if the original prefetched range sequence includes some action that could generate an exception, such as code that is beyond the end of the code segment, a jump or call that replaces that range actually prevents the exception occurrence.

In Real Address Mode, prefetching may cause the microprocessor to access addresses not anticipated by programmers. In Protected Mode, exceptions are correctly reported when these addresses are executed. There may not be hardware mechanisms that account for Real Address Mode behavior of the microprocessor. For example, if a system does not return the $\overline{\text{READY}}$ signal (the signal that terminates a bus cycle) for bus cycles to unimplemented addresses, prefetching must not reference these addresses. If a system implements parity checking, prefetching must not access addresses beyond the end of parity-protected memory. (Alternatively, the hardware design can cause $\overline{\text{READY}}$ to be returned even for unimplemented address bus cycles, and parity errors can be ignored for prefetches beyond the end of parity-protected memory.)

Prefetching can be kept from referencing a particular address by placing enough distance between the address and the last executable byte. For example, to keep prefetching away from addresses in the block from 10000h to 1000Fh, the last executable byte should be no closer than 0FFEEh. This places one free byte followed by one free, aligned, 128-bit block between the last byte of the last instruction and the address that must not be referenced. The prefetching behavior of the Am486 microprocessor is implementation-dependent; future AMD products may have different prefetching behavior.

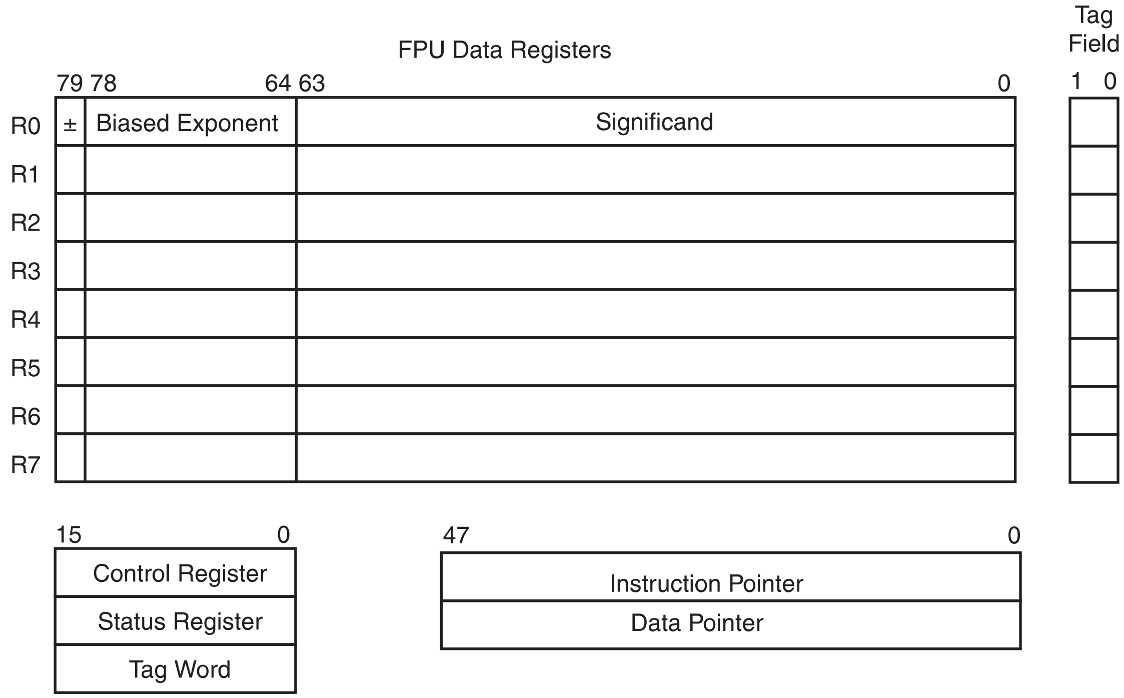
A.2.5.4 FPU Registers

The FPU uses the following Registers shown in Figure A-36:

- Eight individually-addressable 80-bit numeric registers, organized as a register stack
- Three 16-bit registers:
  - — FPU Status Word
  - — FPU Control Word
  - — Tag Word
- Error pointers:
  - — Two 16-bit registers containing selectors for the last instruction and operand
  - — Two 32-bit registers containing offsets for the last instructions and operand
  - — One 11-bit register containing the opcode of the last non-control FPU instruction

The FPU instructions use the contents of these registers for their operations.

Figure A-36 Am486 Microprocessor FPU Register Set



A.2.5.4.1 FPU Register Stack

The FPU register stack is shown in Figure A-36. Each of the eight numeric registers in the stack is 80-bits wide and is divided into fields corresponding to the Am486 microprocessor's extended real data type.

Numeric instructions address the data registers relative to the register on the top of the stack. At any point in time, this top-of-stack register is indicated by the TOP (stack TOP) field in the FPU status word. Load or push operations decrement TOP by one and load a value into the new top register. A store-and-pop operation stores the value from the current TOP register and then increments TOP by one. Like stacks in memory, the FPU register stack grows down toward lower-addressed registers.

Many numeric instructions have several addressing modes that permit the programmer to implicitly operate on the top of the stack, or to explicitly operate on specific registers relative to the TOP. The ASM386/486 assembler supports these register addressing modes, using the expression ST(0), or simply ST, to represent the current Stack Top and ST(*i*) to specify the *i*th register from TOP in the stack (0 ≤ i ≤ 7). For example, if TOP contains 011B (register 3 is the top of the stack), the following statement would add the contents of two registers in the stack (registers 3 and 5):

```
   FADDST, ST(2)
```

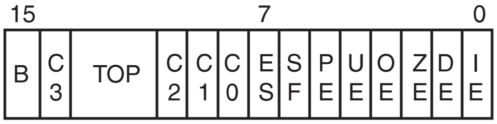
The stack organization and top-relative addressing of the numeric registers simplify subroutine programming by allowing routines to pass parameters on the register stack. By using the stack to pass parameters rather than using “dedicated” registers, calling routines gain more flexibility in how they use the stack. As long as the stack is not full, each routine simply loads the parameters onto the stack before calling a particular subroutine to perform a numeric calculation. The subroutine then addresses its parameters as ST, ST(1), etc., even though TOP may, for example, refer to physical register 3 in one invocation and physical register 5 in another.

A.2.5.4.2 FPU Status and Control Registers

The three 16-bit status and control registers perform control and monitoring functions for the FPU. They include:

FPU Status Word—this 16-bit status word reflects the overall state of the FPU (see Figure A-37). This status word may be stored into memory using the FSTSW/FNSTSW, FSTENV/FNSTENV, and FSAVE/FNSAVE instructions, and can be transferred into the AX register with the FSTSW AX/FNSTSW AX instructions, allowing the FPU status to be inspected by the Integer Unit.

Figure A-37 FPU Status Word



B	FPU Busy
C3	Condition Code 3

TOP	Top of Stack: 000 = R0; 001 = R1;
	010 = R2; 011 = R3; 100 = R4;
	101 = R5; 110 = R6; 111 = R7

C2	Condition Code 2
C1	Condition Code 1
C0	Condition Code 0
ES	Error Summary Status
SF	Stack Fault
PE	Precision Exception
UE	Underflow Exception
OE	Overflow Exception
ZE	Zero Divide Exception
DE	Denormalized Operand Exception
IE	Invalid Operation Exception

Note: *The B-bit (bit 15) is included for 8087 compatibility only. It reflects the contents of the ES bit (bit 7 of the status word).*

The four FPU condition code bits (C3–CO) are similar to the other status flags. The Am486 microprocessor updates these bits to reflect the outcome of arithmetic operations. Table A-9 summarizes the effect of these instructions on the condition code bits. The condition code bits are used principally for conditional branching. The FSTSW AX instruction stores the FPU status word directly into the AX register, allowing easy access for other code inspection. The SAHF instruction can copy C3–C0 directly to Am486 microprocessor flag bits to simplify conditional branching. Table A-10 shows the mapping of these bits to the flag bits.

Table A-9 Condition Code Interpretation

Instruction	C0	C3	C2	C1
FCOM, FCOMP, FCOMPP, FTST, FUCOM, FUCOMP, FUCOMPP, FICOM, FICOMP	Result of Comparison		Operand is not comparable	Zero or O/$\overline{U}$
FXAM	Operand class			Sign or O/$\overline{U}$
FPREM, FPREM1	Q2	Q0	0 = reduction complete 1 = reduction incomplete	Q1 or O/$\overline{U}$
FIST, FBSTP, FRNDINT, FST, FSTP, FADD, FMUL, FDIV, FDIVR, FSUB, FSUBR, FSCALE, FSQRT, FPATAN, F2XM1, FYL2X, FYL2XP1	Undefined			Roundup or O/$\overline{U}$
FPTAN, FSIN, FCOS, FSINCOS	Undefined		0 = reduction complete 1 = reduction incomplete	Roundup or O/$\overline{U}$ (Undefined if C2 = 1)
FCHS, FABS, FXCH, FINCSTP, FLD, FILD, Constant Loads (FLDxx), FXTRACT, FBLD, FSTP (ext. real)	Undefined			Zero or O/$\overline{U}$
FLDENV, FRSTOR	Each bit loaded from memory			
FLDCW, FSTENV, FSTCW, FSTSW, FCLEX	Undefined			
FINIT, FSAVE	Zero	Zero	Zero	Zero

Notes:

O/$\overline{U}$: When both IE and SF bits of the status word are set, indicating a stack exception, this bit distinguishes between a stack overflow (C1 = 1) and underflow (C1 = 0).

Reduction: If FPREM or FPREM1 produces a remainder less than the modulus, reduction is complete. Incomplete reduction leaves a partial remainder value at the top of the stack. This remainder can be used for further reduction. For FPTAN, FSIN, FCOS, and FSINCOS, the bit is set if the operand at the top of stack is too large; for this case, the original operand remains at the top of the stack.

Undefined: No specific value is defined for these bits.

Table A-10 Correspondence between FPU Flags and Processor Flag Bits

FPU Flag	Processor Flag
C0	CF
C1	(none)
C2	PF
C3	ZF

Bits 11–13 of the status word point to the FPU register that is the current Top of Stack (TOP). The significance of the stack top has been described in the prior section on the register stack.

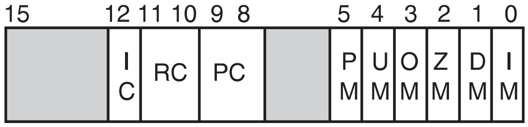
Figure A-37 shows the six exception flags in bits 0–5 of the status word. Bit 7 is the exception summary status (ES) bit. ES is set if any unmasked exception bits are set, and is cleared otherwise. Bits 0–5 indicate whether the FPU has detected one of six possible exception conditions since these status bits were last cleared or reset. They are “sticky” bits, and can only be cleared by the instructions FINIT, FCLEX, FLDENV, FSAVE, and FRSTOR.

Bit 6 is the stack fault (SF) bit. This bit distinguishes invalid operations due to stack overflow or underflow from other kinds of invalid operations. When SF is set, bit 9 (C1) distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0).

A.2.5.4.3 Control Word

The FPU provides the programmer with several processing options, which are selected by loading a word from memory into the control word. Figure A-38 shows the format and encoding of the fields in the control word.

Figure A-38 FPU Control Word Format



IC Infinity Control for
10287 Compatibility
RC Rounding Control:
00 = Round to Nearest or Even
01 = Round Down (to – ∞)
10 = Round Up (to + ∞)
11 = Chop (Truncate Towards 0)
PC Precision Control:
00 = 24 bits (Single)
01 = Reserved (Not Used)
10 = 53 bits (Double)
11 = 64 bits (Extended)
Exception Masks:
PM Precision Mask
UM Underflow Mask
OM Overflow Mask
ZM Zero Divide Mask
DM Denormalized Operand Mask
IM Invalid Operation Mask

The Low-order byte of this control word configures the numerical exception masking. Bits 0–5 of the control word contain individual masks for each of the six floating-point exception conditions recognized by the Am486 microprocessor. The High-order byte of the control word configures the FPU processing options, including:

- Precision control
- Rounding control

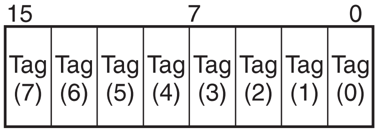
The precision-control bits (bits 8–9) can be used to set the FPU internal operating precision at less than the default precision (64-bit significand). These control bits can be used to provide compatibility with the earlier-generation arithmetic processors having less precision than the 486 microprocessor or 387 math coprocessor. The precision control bits affect the results of only the following five arithmetic instructions: ADD, SUB(R), MUL, DIV(R), and SQRT. No other operations are affected by precision control.

The rounding-control bits (bits 10–11) provide for the common round-to-nearest mode, as well as directed rounding and true chop. Rounding control affects the arithmetic instructions (refer to Chapter 2 for lists of arithmetic and non-arithmetic instructions) and certain non-arithmetic instructions, namely FLD constant and FST(P)mem instructions.

A.2.5.4.4 *FPU Tag Word*

The tag word indicates the contents of each register in the stack (see Figure A-39). The tag word is used by the FPU itself to distinguish between empty and nonempty register locations. Programmers of exception handlers may use this tag information to check the contents of a numeric register without performing complex decoding of the actual data in the register. The tag values from the tag word correspond to physical registers 0–7. Programmers must use the current top-of-stack (TOP) pointer stored in the FPU status word to associate these tag values with the relative stack registers ST(0)–ST(7).

Figure A-39 Tag Word Format



Tag Values:
00 = Valid
01 = Zero
10 = Special/Invalid
(NaN, Unsupported),
Infinity, or Denormal
11 = Empty

The exact values of the tags are generated during execution of the FSTENV and FSAVE instructions according to the actual contents of the non-empty stack locations. During execution of other instructions, the Am486 microprocessor updates the tag values only to indicate whether a stack location is empty or non-empty.

A.2.5.4.5 *Numeric Instruction and Data Pointers*

The instruction and data pointers provide support for programmed exception-handlers. These registers are accessed by the ESC instructions FLDENV, FSTENV, FSAVE, and FRSTOR. Whenever the Am486 microprocessor decodes an ESC instruction, it saves the instruction address, the operand address (if present), and the instruction opcode.

When stored in memory, the instruction and data pointers appear in one of four formats, depending on the operating mode of the microprocessor (Protected Mode or Real Address Mode) and depending on the operand-size attribute in effect (32-bit operand or 16-bit operand). In Virtual 8086 Mode, the Real Address Mode formats are used. Figures A-40 through A-43 show these pointers as they are stored following an FSTENV instruction.

Figure A-40 Protected Mode Numeric Instruction and Data Pointer Image in Memory, 32-Bit Format

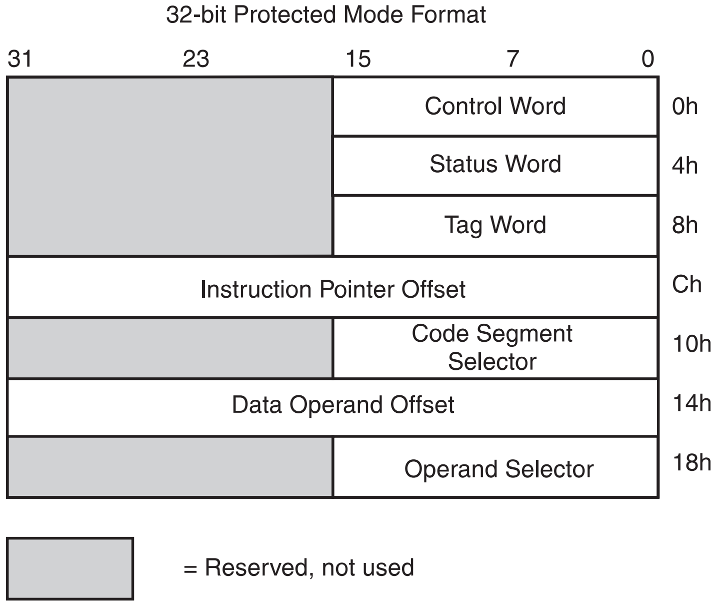


Figure A-41 Real Mode Numeric Instruction and Data Pointer Image in Memory, 32-Bit Format

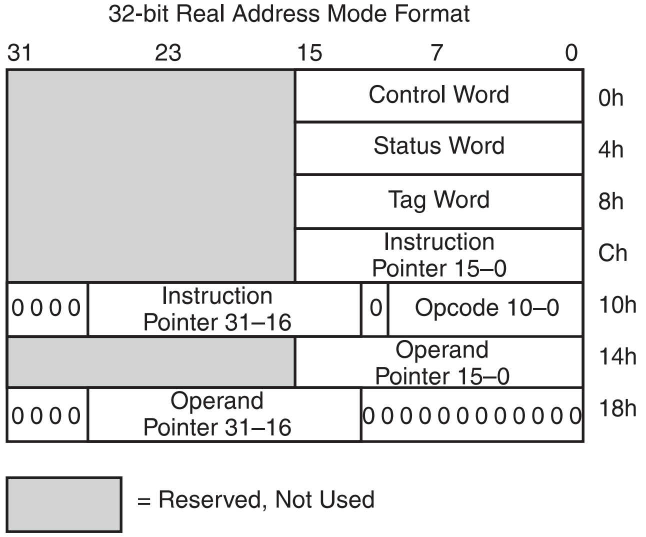
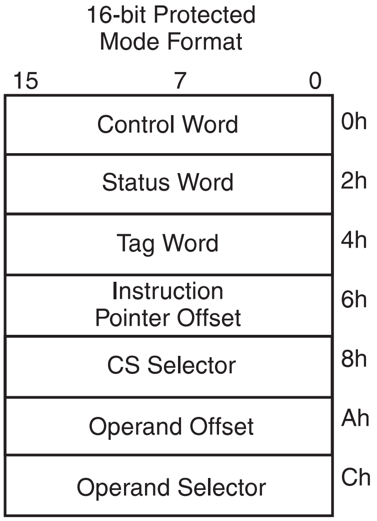


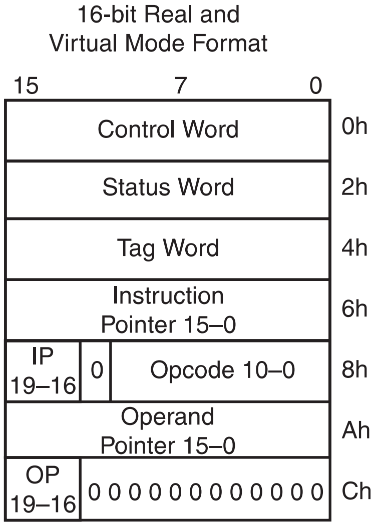
Figure A-42 Protected Mode Numeric Instruction and Data Pointer Image in Memory, 16-Bit Format



16-bit Protected Mode Format

15 7 0	
Control Word	0h
Status Word	2h
Tag Word	4h
Instruction Pointer Offset	6h
CS Selector	8h
Operand Offset	Ah
Operand Selector	Ch

Figure A-43 Real Mode Numeric Instruction and Data Pointer Image in Memory, 16-Bit Format



16-bit Real and Virtual Mode Format

15		7 0	
Control Word			0h
Status Word			2h
Tag Word			4h
Instruction Pointer 15–0			6h
IP 19–16	0	Opcode 10–0	8h
Operand Pointer 15–0			Ah
OP 19–16	0 0 0 0 0 0 0 0 0 0 0 0		Ch

The FSTENV and FSAVE instructions store this data into memory, allowing exception handlers to determine the precise nature of any numeric exceptions that may be encountered.

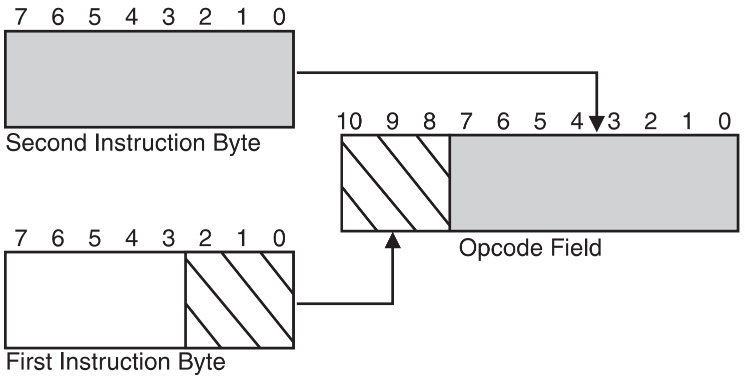
The saved instruction address points to any prefixes that preceded the instruction, as in the 387 and 287 math coprocessors. This is different from the 8087 coprocessor, for which the instruction address points only to the ESC instruction opcode.

***Note:** The microprocessor control instructions FINIT, FLDCW, FSTCW FSTSW, FCLEX, FSTENV, FLDENV, FSAE, and FRSTOR do not affect the data pointer. Also, except for the instructions just mentioned, the value of the data pointer is undefined if the prior ESC instruction did not have a memory operand.*

A.2.5.4.6 Opcode Field of Last Instruction

The opcode field in Figure A-44 describes the 11-bit format of the last non-control FPU instruction executed. The first and second instruction bytes (after all prefixes) are combined to form the opcode field. Since all floating-point instructions share the same 5 upper bits in the first instruction byte (following prefixes), they are not stored in the opcode field. Note that the second instruction byte is actually located in the Low-order byte of the stored opcode field.

Figure A-44 Opcode Field



A.2.6 Instruction Format

The instruction format uses a combination of explicit and implicit conditions to set the environment for executing a specific command. For example, when executing an instruction, the Am486 microprocessor can address memory using either 16- or 32-bit addresses. Accordingly, each instruction that uses memory addresses has an associated address-size attribute of either 16 or 32 bits. Using a 16-bit address implies both the use of 16-bit displacements in instructions and the generation of 16-bit address offsets (segment relative addresses) as the result of the effective address calculations. Using 32-bit addresses implies the use of 32-bit displacements and the generation of 32-bit address offsets. Similarly, an instruction that accesses words (16 bits) or doublewords (32 bits) has an operand-size attribute of either 16 or 32 bits.

The attributes are determined by a combination of defaults, instruction prefixes, and (for programs executing in Protected Mode) size-specification bits in segment descriptors.The information encoded in an instruction includes a specification of the operation to be performed, the type of the operands to be manipulated, and the location of these operands. If an operand is located in memory, the instruction also must select, explicitly or implicitly, the segment that contains the operand.

Chapter 2 provides a complete listing and description of Am486 microprocessor instructions. All non-floating-point instruction encodings are subsets of the general instruction format shown in Figure A-45.

Figure A-45 General Instruction Format

Instruction Prefix	AddressSize Prefix	OperandSize Prefix	Segment Override
0 or 1	0 or 1	0 or 1	0 or 1
Number of Bytes			

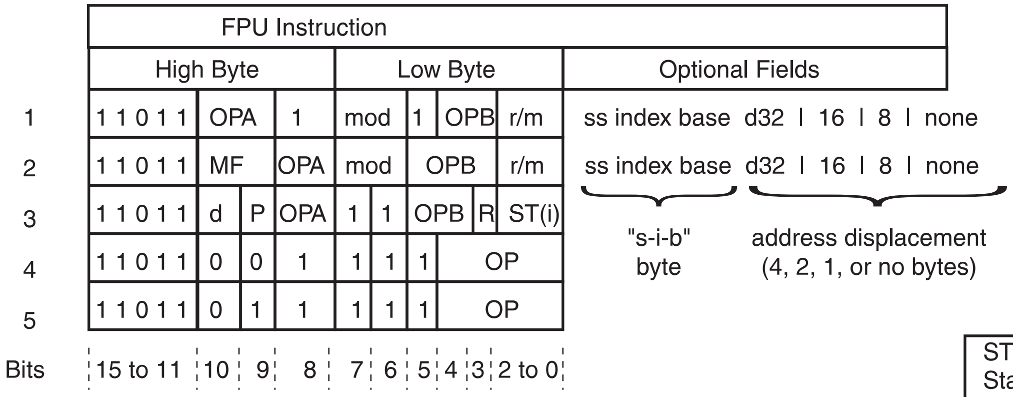
Opcode	mod R/M	s-i-b	Displacement	Immediate
1 or 2	0 or 1	0 or 1	0, 1, 2 or 4	0, 1, 2, or 4
Number of Bytes				

Instructions consist of:

- Instruction prefixes (optional)
- Primary opcode bytes (one or two)
- Address specifier with *mod r/m* byte and Scale Index Base (*s-i-b*) byte, if required
- Displacement, if required
- Immediate data field, if required

Floating-point instructions all begin with the letter “F” and have a basic 2-byte format that may have a 1- or 2-byte optional address specifier field. The basic FPU instruction layout is included in Figure A-46.

Figure A-46 Floating-Point Instruction Formats



ST(i) = Register Stack Element

000 = Stack Top
001 = Top + 1
010 = Top + 2
011 = Top + 3
100 = Top + 4
101 = Top + 5
110 = Top + 6
111 = Top + 7

OP = Instruction Opcode OPA/OPB = Opcode split with A and B parts
MF = Memory Format (00 = 32-bit real; 01 = 32-bit int.; 10 = 64-bit real; 11 = 64-bit int.)
d = Destination (0 = Destination is ST(0); 1 = Destination is ST(i))
R XOR d = 0: Destination OP Source; R XOR d = 1: Source OP Destination
P = Pop (0 = Do not pop stack after operation; 1 = Pop stack after operation)
mod, r/m, s-i-b, and displacement fields are address mode encoding parameters
(see Tables A-11 through A-14)

Table A-11 Address Mode Field (mod/rm) Definitions (no s-i-b present)

Value (mod r/m =)	Effective Address	
	16-Bit Address Mode	**32-Bit Address Mode**
00 000	DS:[BX + SI]	DS:[EAX]
00 001	DS:[BX + DI]	DS:[ECX]
00 010	SS:[BP + SI]	DS:[EDX]
00 011	SS:[BP + DI]	DS:[EBX]
00 100	DS:[SI]	s-i-b present (see Tables A-12 through A-14)
00 101	DS:[DI]	DS:immediate doubleword
00 110	DS:immediate word	DS:[ESI]
00 111	DS:[BX]	DS:[EDI]
01 000	DS:[BX + SI + immediate byte]	DS:[EAX + immediate byte]
01 001	DS:[BX + DI + immediate byte]	DS:[ECX + immediate byte]
01 010	SS:[BP + SI + immediate byte]	DS:[EDX + immediate byte]
01 011	SS:[BP + DI + immediate byte]	DS:[EBX + immediate byte]
01 100	DS:[SI + immediate byte]	s-i-b present (see Tables A-12 through A-14)
01 101	DS:[DI + immediate byte]	SS:[EBP + immediate byte]
01 110	SS:[BP + immediate byte]	DS:[ESI + immediate byte]
01 111	DS:[BX + immediate byte]	DS:[EDI + immediate byte]
10 000	DS:[BX + SI + immediate word]	DS:[EAX + immediate doubleword]
10 001	DS:[BX + DI + immediate word]	DS:[ECX + immediate doubleword]

Table A-11 Address Mode Field (mod/rm) Definitions (no s-i-b present) (continued)

Value (mod r/m =)	Effective Address			
	16-Bit Address Mode		32-Bit Address Mode	
10 010	SS:[BP + SI + immediate word]		DS:[EDX + immediate doubleword]	
10 011	SS:[BP + DI + immediate word]		DS:[EBX + immediate doubleword]	
10 100	DS:[SI + immediate word]		s-i-b present (see Tables A-12 through A-14)	
10 101	DS:[DI + immediate word]		SS:[EBP + immediate doubleword]	
10 110	SS:[BP + immediate word]		DS:[ESI + immediate doubleword]	
10 111	DS:[BX + immediate word]		DS:[EDI + immediate doubleword]	
The following values specify General Registers	**16-Bit Data Operations**		**32-Bit Data Operations**	
	w = 0	**w =1**	**w =0**	**w =1**
11 000	AL	AX	AL	EAX
11 001	CL	CX	CL	ECX
11 010	DL	DX	DL	EDX
11 011	BL	BX	BL	EBX
11 100	AH	SP	AH	ESP
11 101	CH	BP	CH	EBP
11 110	DH	SI	DH	ESI
11 111	BH	DI	BH	EDI

Table A-12 Scale Field (ss) Definitions

Value (ss=)	Scale Factor
00	x1
01	x2
10	x4
11	x8

Table A-13 Index Field (index) Definitions

Value (index=)	Indexed Register
000	EAX
001	ECX
010	EDX
011	EBX
100	no index register
101	EBP
110	ESI
111	EDI

***Note:** When index = 100, the **ss** field must equal 00. If not, the effective address is undefined.*

Table A-14 Base Field (base) Definitions

mod r/m =	Value (base=)	Effective Address
00 100	000	DS:[EAX + (scaled index)]
00 100	001	DS:[ECX + (scaled index)]
00 100	010	DS:[EDX + (scaled index)]
00 100	011	DS:[EBX + (scaled index)]
00 100	100	SS:[ESP + (scaled index)]
00 100	101	DS:[immediate doubleword + (scaled index)]
00 100	110	DS:[ESI + (scaled index)]
00 100	111	DS:[EDI + (scaled index)]
01 100	000	DS:[EAX + (scaled index) + immediate byte]
01 100	001	DS:[ECX + (scaled index) + immediate byte]
01 100	010	DS:[EDX + (scaled index) + immediate byte]
01 100	011	DS:[EBX + (scaled index) + immediate byte]
01 100	100	SS:[ESP + (scaled index) + immediate byte]
01 100	101	SS:[EBP + (scaled index) + immediate byte]
01 100	110	DS:[ESI + (scaled index) + immediate byte]
01 100	111	DS:[EDI + (scaled index) + immediate byte]
10 100	000	DS:[EAX + (scaled index) + immediate doubleword]
10 100	001	DS:[ECX + (scaled index) + immediate doubleword]
10 100	010	DS:[EDX + (scaled index) + immediate doubleword]
10 100	011	DS:[EBX + (scaled index) + immediate doubleword]
10 100	100	SS:[ESP + (scaled index) + immediate doubleword]
10 100	101	SS:[EBP + (scaled index) + immediate doubleword]
10 100	110	DS:[ESI + (scaled index) + immediate doubleword]
10 100	111	DS:[EDI + (scaled index) + immediate doubleword]

A.2.6.1 Instruction Prefixes

Allowable instruction prefix codes include:

- REP/REPE/REPNE/REPNZ/REPZ: Repeat instruction codes used with string instructions
- LOCK: Forces the system to invoke the $\overline{\text{LOCK}}$ signal
- Segment Override: Requires the instruction to use the specified segment register (CS, DS, ES, FS, GS, or SS)
- Operand size override: Requires the instruction to use the specified operand size instead of the default value
- Address size override: Requires the instruction to use the specified address size instead of the default value

Note: *For programs running in Protected Mode, the D bit in executable-segment descriptors specifies the default attribute for both address size and operand size. These default attributes apply to the execution of all instructions in the segment. A clear D bit sets the default address size and operand size to 16 bits; a set D bit, to 32 bits. Programs that execute in Real Mode or Virtual 8086 Mode have 16-bit addresses and operands by default.*

A.2.6.2 Opcode Fields

The opcode fields define the operation, but all can have smaller encoding fields within them that define the operation direction, displacement sizes, the register encoding, or sign extension; encoding fields vary depending on the class of operation.

A.2.6.3 Address Specifier

Most instructions that can refer to an operand in memory have an addressing form byte after the primary opcode byte(s). This byte, called the mod r/m byte, specifies the address form to be used. Certain encodings of the mod r/m byte indicate a second addressing byte, the s-i-b byte, which follows the mod r/m byte and is required to fully specify the addressing form. Addressing forms can include a displacement immediately following either the mod r/m or s-i-b byte. If a displacement is present, it can be 8, 16, or 32 bits. The 8-bit form is used in the common case when the displacement is sufficiently small. The microprocessor extends an 8-bit displacement to 16 or 32 bits, taking into account the sign.

The mod r/m and s-i-b bytes contain the following information:

- The indexing type or register number to be used in the instruction
- The register to be used, or more information to select the instruction
- The base, index, and scale information

The mod r/m byte contains three fields of information:

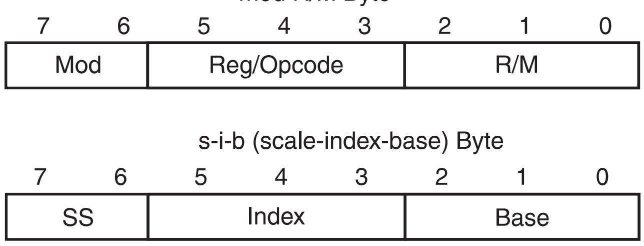
- The mod field, which occupies the two most-significant bits of the byte, combines with the r/m field to form 32 possible values: eight registers and 24 indexing modes.
- The reg field, which occupies the next three bits following the mod field, specifies either a register number or three more bits of opcode information. The meaning of the reg field is determined by the first (opcode) byte of the instruction.
- The r/m field, which occupies the three least-significant bits of the byte, can specify a register as the location of an operand, or can form part of the addressing-mode encoding in combination with the mod field as described above.

The based indexed and scaled indexed forms of 32-bit addressing require the s-i-b byte. The presence of the s-i-b byte is indicated by certain encodings of the mod r/m byte. The s-i-b byte then includes the following fields:

- The ss field (the two most-significant bits of the byte) specifies the scale factor
- The index field (the next three bits after the ss field) specifies the index register number
- The base field (the three least-significant bits of the byte) specifies the base register number

Figure A-47 shows the formats of the mod r/m and s-i-b bytes. (See also Tables A-11–A-14.)

Figure A-47 mod R/M and s-i-b Byte Formats



A.2.6.4 Immediate Operand

If the instruction specifies an immediate operand, the immediate operand always follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction. Immediate operands may be bytes, words, or doublewords. In cases where an 8-bit immediate operand is used with a 16- or 32-bit operand, the microprocessor extends the 8-bit operand to an integer of the same sign and magnitude in the larger size. In the same way, a 16-bit operand is extended to 32-bits.

A.2.7 Operand Selection

An instruction acts on zero or more operands. An example of a zero-operand instruction is the NOP instruction (no operation). An operand can be held in any of these places:

- In the instruction itself (an immediate operand)
- In a register (in the case of 32-bit operands, EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP; in the case of 16-bit operands AX, BX, CX, DX, SI, DI, SP, or BP; in the case of 8-bit operands AH, AL, BH, BL, CH, CL, DH, or DL; the segment registers; or the EFLAGS register for flag operations). Use of 16-bit register operands requires use of the 16-bit operand size prefix (a byte with the value 67h preceding the instruction).
- In memory
- At an I/O port

Access to operands is very fast. Register and immediate operands are available on-chip (the latter because they are prefetched as part of interpreting the instruction). Memory operands residing in the on-chip cache can be accessed just as fast.

Of the instructions that have operands, some specify operands implicitly; others specify operands explicitly; still others use a combination of both. For example:

- Implicit operand: AAM
  - — By definition, AAM (ASCII adjust for multiplication) operates on the contents of the AX register
- Explicit operand: XCHG EAX, EBX
  - — The operands to be exchanged are encoded in the instruction with the opcode
- Implicit and explicit operands: PUSH COUNTER
  - — The memory variable COUNTER (the explicit operand) is copied to the top of the stack (the implicit operand)

Note: *Most instructions have implicit operands. All arithmetic instructions, for example, update the EFLAGS register.*

An instruction can explicitly reference one or two operands. Two-operand instructions, such as MOV, ADD, and XOR, generally overwrite one of the two participating operands with the result. This is the difference between the source operand (the one unaffected by the operation) and the destination operand (the one overwritten by the result).

For most instructions, one of the two explicitly specified operands—either the source or the destination—can be either in a register or in memory. The other operand must be in a register or it must be an immediate source operand. This puts the explicit two-operand instructions into the following groups:

- Register to register
- Register to memory
- Memory to register
- Immediate to register
- Immediate to memory

Certain string instructions and stack manipulation instructions, however, transfer data from memory to memory. Both operands of some string instructions are in memory and are specified implicitly. Push and pop stack operations allow transfer between memory operands and the memory-based stack. Several three-operand instructions are provided, such as the IMUL, SHRD, and SHLD instructions. Two of the three operands are specified explicitly, as for the two-operand instructions, while a third is taken from the ECX register or supplied as an immediate value. Other three-operand instructions, such as the string instructions when used with a repeat prefix, take all their operands from registers.

For programs running in Protected Mode, the D bit in executable-segment descriptors specifies the default attribute for both address size and operand size. These default attributes apply to the execution of all instructions in the segment. A clear D bit sets the default address size and operand size to 16 bits; a set D bit, to 32 bits. Programs that execute in Real Mode or Virtual 8086 Mode have 16-bit addresses and operands by default.

A.2.7.1 Immediate Operands

Certain instructions use data from the instruction itself as one (and sometimes two) of the operands. Such an operand is called an immediate operand. It may be a byte, word, or doubleword. For example:

```
SHR PATTERN, 2
```

One byte of the instruction holds the value 2, the number of bits by which to shift the variable PATTERN.

```
TEST PATTERN, 0FFFF00FFh
```

A doubleword of the instruction holds the mask that is used to test the variable PATTERN.

```
IMUL CX, MEMWORD, 3
```

A word in memory is multiplied by an immediate 3 and stored into the CX register.

All arithmetic instructions (except divide) allow the source operand to be an immediate value. When the destination is the EAX or AL register, the instruction encoding is one byte shorter than with the other general registers.

A.2.7.2 Register Operands

Operands may be located in one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP), in one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, or BP), or in one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, or DL). The Am486 microprocessor has instructions for referencing the segment registers (CS, DS, ES, SS, FS, and GS). These instructions are used by application programs only if system designers have chosen a segmented memory model. The Am486 microprocessor also has instructions for changing the state of individual flags in the EFLAGS register. Instructions have been provided for setting and clearing flags that often need to be accessed. The other flags, which are not accessed so often, can be changed by pushing the contents of the EFLAGS register on the stack, making changes to it while it’s on the stack, and popping it back into the register.

A.2.7.3 Memory Operands

Instructions with explicit operands in memory must reference the segment containing the operand and the offset from the beginning of the segment to the operand. Segments are specified using a segment-override prefix, which is a byte placed at the beginning of an instruction. If no segment is specified, simple rules assign the segment by default. The offset is specified in one of the following ways:

- Most instructions that access memory contain a byte for specifying the addressing method of the operand. The byte, called the *mod r/m* byte, comes after the opcode and specifies whether the operand is in a register or in memory. If the operand is in memory, the address is calculated from a segment register and any of the following values: a base register, an index register, a scaling factor, and a displacement. When an index register is used, the *mod r/m* byte also is followed by another byte to specify the index register and scaling factor. This form of addressing is the most flexible.
- A few instructions use implied address modes: A MOV instruction with the AL or EAX register as either source or destination can address memory with a doubleword encoded in the instruction. This special form of the MOV instruction allows no base register, index register, or scaling factor to be used. This form is one byte shorter than the general-purpose form.

String operations address memory in the DS segment using the ESI register, (the MOVS, CMPS, OUTS, and LODS instructions) or using the ES segment and EDI register (the MOVS, CMPS, INS, SCAS, and STOS instructions).

Stack operations address memory in the SS segment using the ESP register (the PUSH, POP, PUSHA, PUSHAD, POPA, POPAD, PUSHF, PUSHFD, POPF, POPFD, CALL, LEAVE, RET, IRET, and IRETD instructions, exceptions, and interrupts).

A.2.7.3.1 Segment Selection

Explicit specification of a segment is optional. If a segment is not specified by a segment-override prefix, the microprocessor automatically chooses a segment according to the rules of Table A-15. (If a flat model of memory organization is used, the rules for selecting segments are not apparent to application programs.) Different kinds of memory access have different default segments. Data operands usually use the main data segment (the DS segment). However, the ESP and EBP registers are used for addressing the stack, so when either register is used, the stack segment (the SS segment) is selected.

Table A-15 Default Segment Selection Rules

Type of Reference	Segment Used Register Used	Default Selection Rule
Instructions	Code Segment CS Register	Automatic with instruction fetch
Stack	Stack Segment SS Register	All stack PUSHes and POPs. Any memory reference that uses ESP or EBP as a base register.
Local Data	Data Segment DS Register	All data references except when relative to stack or string destination
Destination Strings	E-Space Segment ES Register	Destination of string instructions

Segment-override prefixes are provided for each of the segment registers. Only the following special cases have a default segment selection that is not affected by a segment-override prefix:

- Destination strings in string instructions use the ES segment
- Destination of a push or source of a pop uses the SS segment
- Instruction fetches use the CS segment

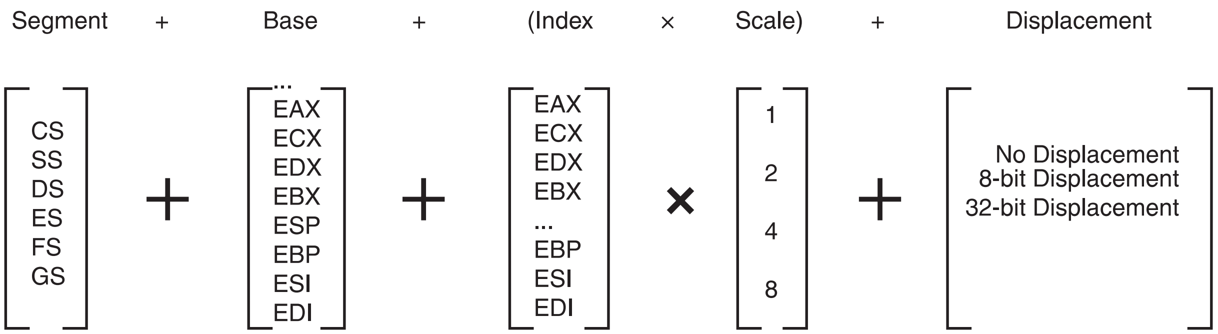
A.2.7.3.2 Effective-Address Computation

The *mod r/m* byte provides the most flexible form of addressing. Instructions that have a *mod r/m* byte after the opcode are the most common in the instruction set. For memory operands specified by a *mod r/m* byte, the offset within the selected segment is the sum of three components:

- Displacement
- Base register
- Index register (the index register may be multiplied by a factor of 2, 4, or 8)

The offset that results from adding these components is called an effective address. Each of these components may have either a positive or negative value. Figure A-48 illustrates the full set of possibilities for mod r/m addressing.

Figure A-48 Effective Address Computation



The displacement component, because it is encoded in the instruction, is useful for relative addressing by fixed amounts, such as:

- Location of simple scalar operands
- Beginning of a statically allocated array
- Offset to a field within a record

The base and index components have similar functions. Both use the same set of general registers. Both can be used for addressing that changes during program execution, such as:

- Location of procedure parameters and local variables on the stack.
- The beginning of one record among several occurrences of the same record type or in an array of records
- The beginning of one dimension of multiple dimension array
- The beginning of a dynamically allocated array

The uses of general registers as base or index components differ in the following respects:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default selection. In all other cases, the DS segment is the default selection.
- The scaling factor permits efficient indexing into an array when the array elements are 2, 4, or 8 bytes. The scaling of the index register is done in hardware at the time the address is evaluated. This eliminates an extra shift or multiply instruction.

The base, index, and displacement components may be used in any combination; any of these components may be null. A scale factor can be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language. Suggested uses for some combinations of address components are described below:

- Displacement—indicates the offset of the operand. This form of addressing is used to access a statically allocated scalar operand. A byte, word, or doubleword displacement can be used.
- Base—the offset to the operand is specified indirectly in one of the general registers, as for “based” variables.
- Base + Displacement—a register and a displacement can be used together for two distinct purposes:
  - — Index into static array when the element size is not 2, 4, or 8 bytes. The displacement component encodes the offset of the beginning of the array. The register holds the results of a calculation to determine the offset to a specific element within the array.
  - — Access a field of a record. The base register holds the address of the beginning of the record, while the displacement is an offset to the field.

Note: *An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a subroutine is entered. In this case, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.*

- (Index · Scale) + Displacement—this combination is an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement addresses the beginning of the array, the index register holds the subscript of the desired array element, and the microprocessor automatically converts the subscript into an index by applying the scaling factor.
- Base + Index + Displacement—two registers used together that support either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).
- Base + (Index · Scale) + Displacement—provides efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

A.2.8 Interrupts and Exceptions

Interrupts and exceptions are forced transfers of execution to a task or a procedure. The task or procedure is called a handler. Interrupts occur at random times during the execution of a program in response to signals from hardware. Exceptions occur when instructions that provoke exceptions are executed. Usually, the servicing of interrupts and exceptions is performed in a manner transparent to application programs. Interrupts are used to handle events external to the microprocessor, such as requests to service peripheral devices. Exceptions handle conditions detected by the microprocessor in the course of executing instructions, such as division by 0.

There are two sources for interrupts and two sources for exceptions:

- Interrupts
  - — *Maskable interrupts:* invoked by a signal to the INTR input if not masked by IF
  - — *Non-maskable interrupts:* invoked by a signal to the NMI input
- Exceptions
  - — *Microprocessor-detected exceptions*: faults, traps, and aborts
  - — *Programmed exceptions*: triggered by INTO, INT 3h, INT nh, and BOUND instructions

Application programmers normally are not concerned with handling exceptions or interrupts. The operating system, monitor, or device driver handles them. Certain kinds of exceptions, however, are relevant to application programming, and many operating systems give application programs the opportunity to service these exceptions. However, the operating system defines the interface between the application program and the exception mechanism of the Am486 microprocessor.

Table A-16 lists the exceptions and interrupts.

Table A-16 Exceptions and Interrupts

Vector Number	Description
0	Divide Error
1	Debugger Call
2	NMI
3	Breakpoint
4	INTO-detected Overflow
5	BOUND Range Exceeded
6	Invalid Opcode
7	Device Not Available
8	Double Fault
9	Reserved
10	Invalid Task State Segment
11	Segment Not Present
12	Stack Exception
13	General Protection
14	Page Fault
15	Reserved
16	Floating-Point Error
17	Alignment Check
18–31	Reserved
32–255	Maskable Interrupts

- A divide-error exception results when the DIV or IDIV instruction is executed with a zero denominator or when the quotient is too large for the destination operand.
- A debug exception may be sent back to an application program if it results from the Trap Flag (TF).
- A breakpoint exception results when an INT3 instruction is executed. This instruction is used by some debuggers to stop program execution at specific points.
- An overflow exception results when the INTO instruction is executed and the Overflow Flag (OF) is set.
- A bounds-check exception results when the BOUND instruction is executed with an array index that falls outside the bounds of the array.
- The device-not-available exception occurs whenever the microprocessor encounters an escape instruction and either the TS (task switched) or the EM (emulate coprocessor) bit of the CR0 control register is set.
- An alignment-check exception is generated for unaligned memory operations in user mode (privilege level 3), provided both AM and AC are set. Memory operations at supervisor mode (privilege levels 0, 1, and 2), or memory operations that default to supervisor mode, do not generate this exception.

The INT instruction generates an interrupt whenever it is executed; the microprocessor treats this interrupt as an exception. Its effects (and the effects of all other exceptions) are determined by exception handler routines in the application program or the operating system.

Exceptions caused by segmentation and paging are handled differently than interrupts. Normally, the contents of the program counter (EIP register) are saved on the stack when an exception or interrupt is generated. But exceptions resulting from segmentation and paging restore the contents of some microprocessor registers to the state they held prior to instruction interpretation. The saved contents of the program counter address the instruction that caused the exception, rather than the instruction after it. This lets the operating system fix the exception-generating condition and restart the program at the instruction that generated the exception. This mechanism is completely transparent to the program.

A.2.9 Input/Output

This chapter explains the input/output architecture of the Am486 microprocessor. Input/output is accomplished through I/O ports, which are registers connected to peripheral devices. An I/O port can be an input port, an output port, or a bidirectional port. Some I/O ports are used for carrying data, such as the transmit and receive registers of a serial interface. Other I/O ports are used to control peripheral devices, such as the control registers of a disk controller.

The Am486 microprocessor always synchronizes I/O instruction execution with external bus activity. All previous instructions are completed before an I/O operation begins. In particular, all writes held pending in the Am486 CPU write buffers are completed before an I/O read or write is performed.

The input/output architecture is the programmer’s model of how these ports are accessed. The discussion of this model includes:

- Methods of addressing I/O ports
- Instructions that perform I/O operations
- The I/O protection mechanism

A.2.9.1 I/O Addressing

The Am486 microprocessor allows I/O ports to be addressed in either of two ways:

- Through a separate I/O address space accessed using I/O instructions
- Through memory-mapped I/O, where I/O ports appear in the address space of physical memory

The use of a separate I/O address space is supported by special instructions and a hardware protection mechanism. When memory-mapped I/O is used, the general purpose instruction set can be used to access I/O ports, and protection is provided using segmentation or paging. Some system designers may prefer to use the I/O facilities built into the microprocessor, while others may prefer the simplicity of a single physical address space.

If segmentation or paging is used for protection of the I/O address space, the AVL fields in segment descriptors or page-table entries may be used to mark pages containing I/O as unrelocatable and unswappable. The AVL fields are provided for this kind of use, where a system programmer needs to make an extension to the address translation and protection mechanisms.

Hardware designers use these ways of mapping I/O ports into the address space when they design the address decoding circuits of a system. I/O ports can be mapped so that they appear in the I/O address space or the address space of physical memory (or both). System programmers may need to discuss with hardware designers the kind of I/O addressing they would like to have.

A.2.9.1.1 I/O Address Space

The Am486 microprocessor provides a separate I/O address space, distinct from the address space for physical memory, where I/O ports can be placed. The I/O address space consists of 2^{16} (64K) individually addressable 8-bit ports; any two consecutive 8-bit ports can be treated as a 16-bit port, and any four consecutive ports can be a 32-bit port. Extra bus cycles are required if a port crosses the boundary between two doublewords in physical memory.

The M/$\overline{\text{IO}}$ pin on the Am486 microprocessor indicates when a bus cycle to the I/O address space occurs. When a separate I/O address space is used, it is the responsibility of the hardware designer to make use of this signal to select I/O ports rather than memory. In fact, the use of the separate I/O address space simplifies the hardware design because these ports can be selected by a single signal; unlike other microprocessors, it is not necessary to decode a number of upper address lines in order to set up a separate I/O address space.

A program can specify the address of a port in two ways. With an immediate byte constant, the program can specify:

- 256 8-bit ports numbered 0–255
- 128 16-bit ports numbered 0, 2, 4, . . . , 252, 254
- 64 32-bit ports numbered 0, 4, 8, . . . , 248, 252

Using a value in the DX register, the program can specify:

- 8-bit ports numbered 0–65535
- 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534
- 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532

The Am486 microprocessor can transfer 8, 16, or 32 bits to a device in the I/O space. Like words in memory, 16-bit ports should be aligned to even addresses so that all 16 bits can be transferred in a single bus cycle. Like doublewords in memory, 32-bit ports should be aligned to addresses that are multiples of 4. The microprocessor supports data transfers to unaligned ports, but there is a performance penalty because an extra bus cycle must be used.

- The IN and OUT instructions move data between a register and a port in the I/O address space. The instructions INS and OUTS move strings of data between the memory address space and ports in the I/O address space.
- I/O port addresses 0F8h through 0FFh are reserved for use by AMD. Do not assign I/O ports to these addresses.
- The exact order of bus cycles used to access ports that require more than one bus cycle is undefined. For example, an OUT instruction that loads an unaligned doubleword port at location 2h accesses the word at 4h before accessing the word at 2h. This behavior is neither defined, nor guaranteed to remain the same in future AMD products.
- If software needs to produce a particular order of bus cycles, this order must be specified explicitly. For example, to load a word-length port at 4h followed by loading a word port at 2h, two word-length instructions must be used, rather than a single doubleword instruction.

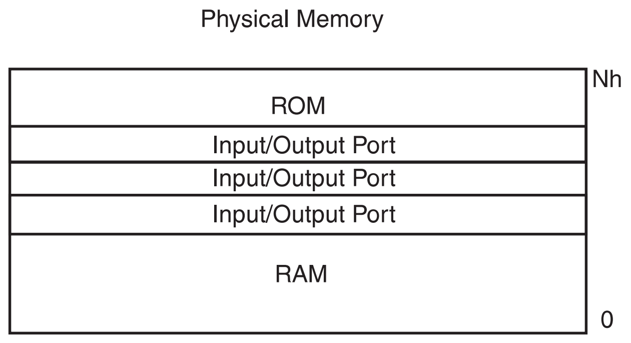
***Note:** Although the Am486 microprocessor automatically masks parity errors for certain types of bus cycles, such as interrupt acknowledge cycles, it does not mask parity for bus cycles to the I/O address space. Programmers may need to be aware of this behavior as a possible source of spurious parity efforts.*

A.2.9.1.2 Memory-Mapped I/O

I/O devices may be placed in the address space for physical memory. This is called memory-mapped I/O. As long as the devices respond like memory components, they can be used with memory-mapped I/O.

Memory-mapped I/O provides additional programming flexibility. Any instruction that references memory may be used to access an I/O port located in the memory space. For example, the MOV instruction can transfer data between any register and a port. The AND, OR, and TEST instructions may be used to manipulate bits in the control and status registers of peripheral devices (see Figure A-49). Memory-mapped I/O can use the full instruction set and the full complement of addressing modes to address I/O ports.

Figure A-49 Memory Mapped I/O



To optimize performance, the Am486 CPU allows reads to be re-ordered ahead of buffered writes in certain precisely-defined circumstances. Using memory-mapped I/O on the Am486 CPU therefore creates the possibility that an I/O read will be performed before the memory write of a previous instruction. To eliminate this possibility, use an I/O instruction for the read. Using an I/O instruction for an I/O write can also be advantageous because it guarantees that the write will be completed before the next instruction begins execution. If I/O writes are used to control system hardware, then this sequence of events is desirable, since it guarantees that the next instruction will be executed in the new state.

- If caching is enabled, either external hardware or the paging mechanism (the PCD bit in the page table entry) must be used to prevent caching of I/O data.
- Memory-mapped I/O, like any other memory reference, is subject to access protection and control. See Section A.2.3 for a discussion of memory protection.

A.2.9.2 I/O Instructions

The I/O instructions of the Am486 microprocessor provide access to the microprocessor’s I/O ports for the transfer of data. These instructions have the address of a port in the I/O address space as an operand. There are two kinds of I/O instructions:

- Those that transfer a single item (byte, word, or doubleword) to or from a register.
- Those that transfer strings of items (strings of bytes, words, or doublewords) located in memory. These are known as “string I/O instruction” or “block I/O instructions.” These instructions cause the M/$\overline{\text{IO}}$ signal to be driven Low (logic 0) during a bus cycle, which indicates to external hardware that access to the I/O address space is taking place.

If memory-mapped I/O is used, there is no reason to use I/O instructions.

A.2.9.3 Register I/O Instructions

The I/O instructions IN and OUT move data between I/O ports and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The IN and OUT instructions address I/O ports either directly, with the address of one of 256 port addresses coded in the instruction, or indirectly using an address in the DX register to select one of 64K port addresses. These instructions synchronize program execution to external hardware. The Am486 microprocessor write buffers are cleared and program execution delayed until the last ready of the last bus cycle has been returned.

- IN (Input from Port)—transfers a byte, word, or doubleword from an input port to the AL, AX, or EAX registers. A byte IN instruction transfers 8 bits from the selected port to the AL register. A word IN instruction transfers 16 bits from the port to the AX register. A doubleword IN instruction transfers 32 bits from the port to the EAX register.
- OUT (Output from Port)—transfers a byte, word, or doubleword from the AL, AX, or EAX registers to an output port. A byte OUT instruction transfers 8 bits from the AL register to the selected port. A word OUT instruction transfers 16 bits from the AX register to the port. A doubleword OUT instruction transfers 32 bits from the EAX register to the port.

A.2.9.4 Block I/O Instructions

The INS and OUTS instructions move blocks of data between I/O ports and memory. Block I/O instructions use an address in the DX register to address a port in the I/O address space. These instructions use the DX register to specify:

- 8-bit ports numbered 0–65535
- 16-bit ports numbered 0, 2, 4, . . . , 65532, 65534
- 32-bit ports numbered 0, 4, 8, . . . , 65528, 65532

Block I/O instructions use either the SI or DI register to address memory. For each transfer, the SI or DI register is incremented or decremented, as specified by DF.

The INS and OUTS instructions, when used with repeat prefixes, perform block input or output operations. The repeat prefix REP modifies the INS and OUTS instructions to transfer blocks of data between an I/O port and memory. These block I/O instructions are string instructions. They simplify programming and increase the speed of data transfer by eliminating the need to use a separate LOOP instruction or an intermediate register to hold the data. The string I/O instructions operate on byte strings, word strings, or doubleword strings. After each transfer, the memory address in the ESI or EDI registers is incremented or decremented by 1 for byte operands, by 2 for word operands, or by 4 for doubleword operands. DF controls whether the register is incremented (DF is clear) or decremented (DF is set).

- INS (Input String from Port)—transfers a byte, word, or doubleword string element from an input port to memory. The INSB instruction transfers a byte from the selected port to the memory location addressed by the ES and EDI registers. The INSW instruction transfers a word. The INSD instruction transfers a doubleword. A segment override prefix cannot be used to specify an alternate destination segment. Combined with a REP prefix, an INS instruction makes repeated read cycles to the port, and puts the data into consecutive locations in memory.
- OUTS (Output String from Port)—transfers a byte, word, or doubleword string element from memory to an output port. The OUTSB instruction transfers a byte from the memory location addressed by the DS and ESI registers to the selected port. The OUTSW instruction transfers a word. The OUTSD instruction transfers a doubleword. A segment override prefix cannot be used to specify an alternate source segment. Combined with a REP prefix, an OUTS instruction reads consecutive locations in memory and writes the data to an output port.

A.2.9.5 Protection and I/O

The I/O architecture has two protection mechanisms:

- The IOPL field in the EFLAGS register controls access to the I/O instructions.
- The I/O permission bit map of a TSS segment controls access to individual ports in the I/O address space.

These protection mechanisms are available only when a separate I/O address space is used. When memory-mapped I/O is used, protection is provided using segmentation or paging.

A.2.9.5.1 I/O Privilege Level

In systems that use I/O protection, the IOPL field in the EFLAGS register controls access to I/O instructions. This permits the operating system to adjust the privilege level needed to perform I/O operations. In a typical protection ring model, privilege levels 0 and 1 have access to the I/O instructions. This lets the operating system and the device drivers perform I/O, but keeps applications and less privileged device drivers from accessing the I/O address space. Applications access I/O through the operating system. The following instructions can be executed only if CPL ≤ IOPL:

```
IN    –Input
INS   –Input String
OUT   –Output
OUTS  –Output String
CLI   –Clear Interrupt-Enable Flag
STI   –Set Interrupt-Enable Flag
```

These instructions are called “sensitive” instructions, because they are sensitive to the IOPL field. In Virtual-8086 Mode, IOPL is not used; only the I/O permission bit map limits access to I/O ports.

To use sensitive instructions, a procedure must run at a privilege level at least as privileged as that specified by the IOPL field. Any attempt by a less privileged procedure to use a sensitive instruction results in a general-protection exception. Because each task has its own copy of the EFLAGS register, each task can have a different IOPL.

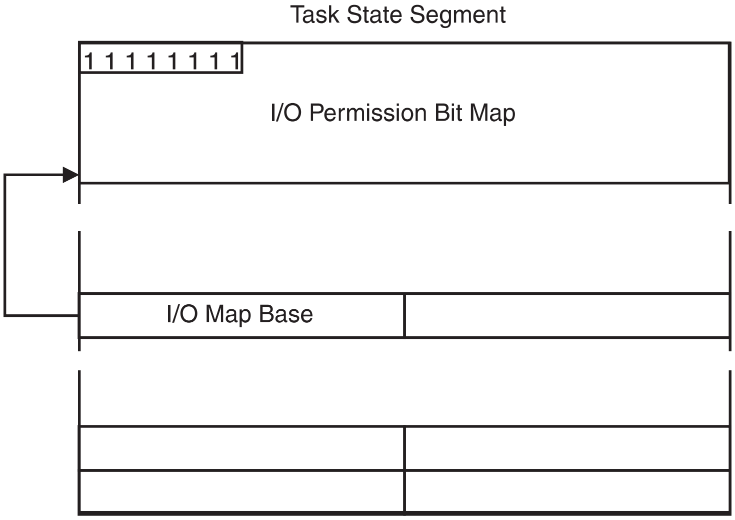
A task can change IOPL only with the POPF instruction; however, such changes are privileged. No procedure may change its IOPL unless it is running at privilege level 0. An attempt by a less privileged procedure to change the IOPL does not result in an exception; the IOPL simply remains unchanged.

The POPF instruction also may be used to change the state of IF (as can the CLI and STI instructions); however, changes to IF using the POPF instruction are IOPL-sensitive. A procedure may change the setting of IF with a POPF instruction only if it runs with a CPL at least as privileged as the IOPL. An attempt by a less privileged procedure to change IF does not result in an exception; IF simply remains unchanged.

A.2.9.5.2 I/O Permission Bit Map

The Am486 microprocessor can generate exceptions for references to specific I/O addresses. These addresses are specified in the I/O permission bit map in the TSS (see Figure A-50). The size of the map and its location in the TSS are variable. The microprocessor finds the I/O permission bit map with the I/O map base address in the TSS. The base address is a 16-bit offset into the TSS. This is an offset to the beginning of the bit map. The limit of the TSS is the limit on the size of the I/O permission bit map.

Figure A-50 I/O Permission Bit Map



Note: *The Base Address for the I/O Bit Map must not exceed DFFFh. The last byte of the Bit Map must be followed by a byte with all ones.*

Because each task has its own TSS, each task has its own I/O permission bit map. Access to individual I/O ports can be granted to individual tasks.

If CPL is less than or equal to IOPL in Protected Mode, then the microprocessor allows I/O operations to proceed. If CPL is greater than IOPL, or if the microprocessor is operating in Virtual 8086 Mode, then the microprocessor checks the I/O permission map. Each bit in the map corresponds to an I/O port byte address; for example, the control bit for address 41 (decimal) in the I/O address space is found at bit position 1 of the sixth byte in the bit map. The microprocessor tests all the bits corresponding to the I/O port being addressed; for example, a doubleword operation tests four bits corresponding to four adjacent byte addresses. If any tested bit is set, a general-protection exception is generated. If all tested bits are clear, the I/O operation proceeds.

Because I/O ports that are not aligned to word and doubleword boundaries are permitted, it is possible that the microprocessor may need to access two bytes in the bit map when I/O permission is checked. For maximum speed, the microprocessor has been designed to read two bytes for every access to an I/O port. To prevent exceptions from being generated when the ports with the highest addresses are accessed, an extra byte needs to come after the table. This byte must have all of its bits set, and it must be within the segment limit.

It is not necessary for the I/O permission bit map to represent all the I/O addresses. I/O addresses not spanned by the map are treated as if they had set bits in the map. For example, if the TSS segment limit is 10 bytes past the bit map base address, the map has 11 bytes and the first 80 I/O ports are mapped. Higher addresses in the I/O address space generate exceptions.

If the I/O bit map base address is greater than or equal to the TSS segment limit, there is no I/O permission map, and all I/O instructions generate exceptions. The base address must be less than or equal to 0DFFFh.

A.3 DEBUGGING

The Am486 microprocessor has advanced debugging facilities that are particularly important for sophisticated software systems, such as multitasking operating systems. The failure conditions for these software systems can be very complex and time-dependent. The debugging features of the Am486 microprocessor give the system programmer valuable tools for looking at the dynamic state of the microprocessor.

The debugging support is accessed through the debug registers. They hold the addresses of memory locations, called breakpoints, that invoke debugging software. An exception is generated when a memory operation is made to one of these addresses. A breakpoint is specified for a particular form of memory access, such as an instruction fetch or a doubleword write operation. The debug registers support both instruction breakpoints and data breakpoints.

With other microprocessors, instruction breakpoints are set by replacing normal instructions with breakpoint instructions. When the breakpoint instruction is executed, the debugger is called. But with the debug registers of the Am486 microprocessor, this is not necessary. By eliminating the need to write into the code space, the debugging process is simplified (there is no need to set up a data segment mapped to the same memory as the code segment) and breakpoints can be set in ROM-based software. In addition, breakpoints can be set on reads and writes to data that allows real-time monitoring of variables.

A.3.1 Debugging Support

The features of the architecture that support debugging are:

- Reserved Debug Interrupt Vector—specifies a procedure or task to call when an event for the debugger occurs
- Debug Address Registers—specifies the addresses of up to four breakpoints
- Debug Control Register—specifies the forms of memory access for the breakpoints
- Debug Status Register—reports conditions in effect at the time of the exception
- Trap Bit of TSS (T-bit)—generates a debug exception when an attempt is made to perform a task switch to a task with this bit set in its TSS
- Resume Flag (RF)—suppresses multiple exceptions to the same instruction
- Trap Flag (TF)—generates a debug exception after every execution of an instruction
- Breakpoint Instruction—calls the debugger (generates a debug exception). This instruction is an alternative way to set code breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are placed in the source code.
- Reserved Interrupt Vector for Breakpoint Exception—calls a procedure or task when a breakpoint instruction is executed

These features allow a debugger to be called either as a separate task or as a procedure in the context of the current task. The following conditions are used to call the debugger:

- Task switch to a specific task
- Execution of the breakpoint instruction
- Execution of any instruction
- Execution of an instruction at a specified address
- Read or write of a byte, word, or doubleword at a specified address
- Write to a byte, word, or doubleword at a specified address
- Attempt to change the contents of a debug register

A.3.2 Debug Registers

Six registers control debugging. The registers are accessed by a MOV instruction. A debug register can be the source or destination operand for the instruction. Debug registers are privileged resources; MOV instructions that access them can execute only at privilege level 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception. Figure A-51 shows the debug register format.

A.3.2.1 Debug Address Registers (DR3–DR0)

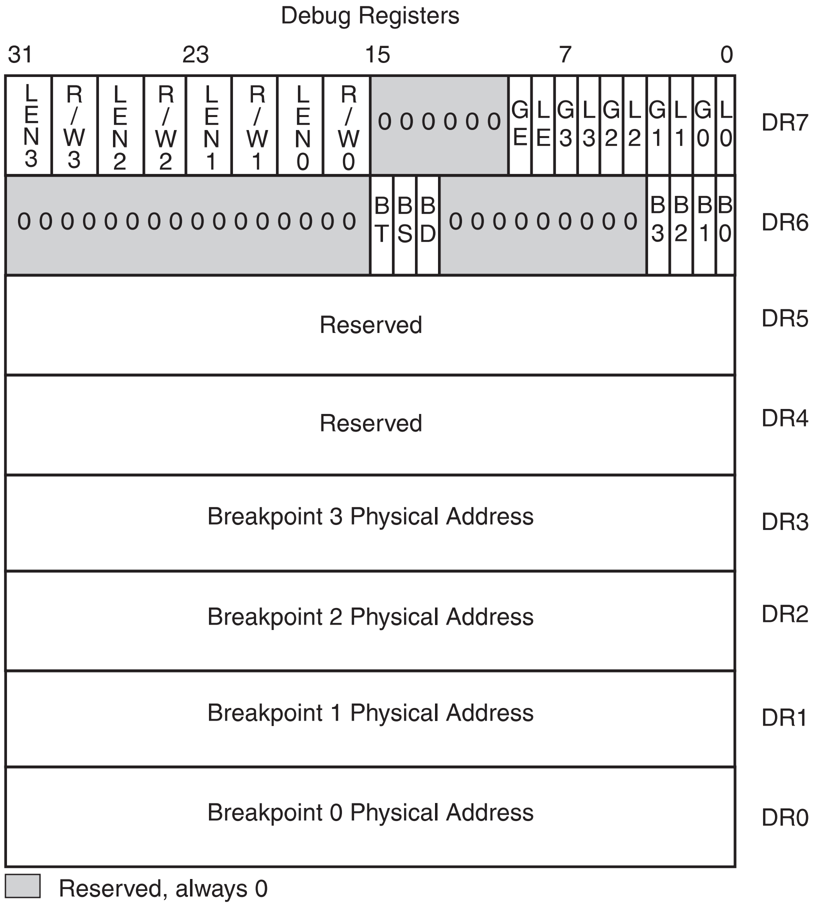
Each of these registers holds the linear address for one of the four breakpoints. That is, breakpoint comparisons are made before physical address translation occurs. Each breakpoint condition is specified further by the contents of the DR7 register.

A.3.2.2 Debug Control Register (DR7)

The debug control register shown in Figure A-51 specifies the sort of memory access associated with each breakpoint. Each address in registers DR3–DR0 corresponds to a field R/W3–R/W0 in the DR7 register. The microprocessor interprets these bits as follows:

- 00—Break on instruction execution only
- 01—Break on data writes only
- 10—Undefined
- 11—Break on data reads or writes but not instruction fetches

Figure A-51 Debug Registers



The LEN3–LEN0 fields in the DR7 register specify the size of the breakpoint. The length fields are interpreted as follows:

- 00—One-byte length
- 01—Two-byte length
- 10—Undefined
- 11—Four-byte length

Note: *If RWn is 00 (instruction execution), then LENn should also be 00. The effect of using any other length is undefined.*

The GD bit enables the debug register protection condition that is flagged by BD of DR6. Note that GD is cleared at entry to the debug exception handler by the microprocessor. This allows the handler free access to the debug registers.

The Low 8 bits of the DR7 register (fields L3–L0 and G3–G0) individually enable the four address breakpoint conditions. There are two levels of enabling: the local (L3–L0) and global (G3–G0) levels. The local enable bits are automatically cleared by the microprocessor on every task switch to avoid unwanted breakpoint conditions in the new task. They are used to breakpoint conditions in a single task. The global enable bits are not cleared by a task switch. They are used to enable breakpoint conditions that apply to all tasks.

The Am486 microprocessor always uses exact data breakpoint matching in debugging. That is, if any of the Ln/Gn bits are set, the microprocessor slows execution so that data breakpoints are reported for the instruction that triggers the breakpoint, rather than the next instruction. In such a case, one-clock instructions that access memory take two clocks to execute.

In the Am386 microprocessor, exact data breakpoint matching does not occur unless it is enabled by setting either the LE or the GE bit. The Am486 microprocessor ignores these bits.

A.3.2.3 Debug Status Register (DR6)

The debug status register shown in Figure A-51 reports conditions sampled at the time the debug exception was generated. Among other information, it reports which breakpoint triggered the exception. Update only occurs if the exception is taken, then all bits will be updated.

When an enabled breakpoint generates a debug exception, it loads the Low four bits of this register (B0–B3) before entering the debug exception handler. The B bit is set if the condition described by the DR, LEN, and R/W bits is true, even if the breakpoint is not enabled by the L and G bits. The microprocessor sets the B bits for all breakpoints that match the conditions present at the time the debug exception is generated, whether or not they are enabled.

The BT bit is associated with the T bit (debug trap bit) of the TSS. The microprocessor sets the BT bit before entering the debug handler if a task switch has occurred to a task with a set T bit in its TSS. There is no bit in the DR7 register to enable or disable this exception; the T bit of the TSS is the only enabling bit.

The BS bit is associated with TF. The BS bit is set if the debug exception is triggered by the single-step execution mode (TF set). The single-step mode is the highest-priority debug exception; when the BS bit is set, any of the other debug status bits may also be set.

The BD bit is set if the next instruction reads or writes one of the eight debug registers while it is being used by in-circuit emulation.

Note: *The contents of the DR6 register are never cleared by the microprocessor. To avoid any confusion in identifying debug exceptions, the debug handler should clear the register before returning.*

A.3.2.4 Breakpoint Field Recognition

The address and LEN bits for each of the four breakpoint conditions define a range of sequential byte addresses for a data breakpoint. The LEN bits permit specification of a 1-, 2-, or 4-byte range. Align 2-byte ranges on word boundaries (addresses that are multiples of 2) and 4-byte ranges on doubleword boundaries (addresses that are multiples of 4). These requirements are enforced by the microprocessor; it uses the LEN bits to mask the lower address bits in the debug registers. Unaligned code or data breakpoint addresses do not yield the expected results.

A data breakpoint for reading or writing is triggered if any of the bytes participating in a memory access is within the range defined by a breakpoint address register and its LEN bits. A data breakpoint for an unaligned operand can be made from two entry sets in the breakpoint registers where each entry is byte-aligned, and the two entries cover the operand. This breakpoint generates exceptions for the operand, not for any neighboring bytes. Instruction breakpoint addresses must have a 1-byte length specification (LEN = 00); the behavior of code breakpoints for other operand sizes is undefined.

Table A-17 Breakpoint Examples

Comment	Address	Length in bytes
DR0 Contents	A0001h	1 (LEN0 = 00)
DR1 Contents	A0002h	1 (LEN0 = 00
DR2 Contents	B0002h	2 (LEN0 = 01)
DR3 Contents	C0000h	4 (LEN0 =11)
Memory Operations That Trap	A0001h	1
	A0002h	1
	A0001h	2
	A0002h	2
	B0002h	2
	B0001h	4
	C0000h	4
	C0001h	2
	C0003h	1
Memory Operations That Do Not Trap	A0000h	1
	A0003h	4
	B0000h	2
	C0004h	4

Table A-17 gives some examples of combinations of addresses and fields with memory references that do and do not cause traps. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has any prefixes, the breakpoint address must point to the first prefix.

A.3.3 Debug Exceptions

Two of the interrupt vectors of the Am486 microprocessor are reserved for debug exceptions. The debug exception is the usual way to invoke debuggers designed for the Am486 microprocessor. The breakpoint exception is intended to put breakpoints in debuggers.

A.3.3.1 Interrupt 1—Debug Exceptions

The handler for this exception usually is a debugger or part of a debugging system. The microprocessor generates a debug exception for any of several conditions. The debugger can check flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions also might apply. Table A-18 shows the states of these bits for each kind of breakpoint condition.

Table A-18 Debug Exception Conditions

Flags Tested	Description
BS = 1	Single-step trap
B0 = 1 and (GE0 = 1 or LE0 = 1)	Breakpoint defined by DR0, LEN0, and R/W0
B1 = 1 and (GE1 = 1 or LE1 = 1)	Breakpoint defined by DR1, LEN1, and R/W1
B2 = 1 and (GE2 = 1 or LE2 = 1)	Breakpoint defined by DR2, LEN2, and R/W2
B3 = 1 and (GE3 = 1 or LE3 = 1)	Breakpoint defined by DR3, LEN3, and R/W3
BD = 1	Debug registers in use for in-circuit emulation
BT = 1	Task switch

Instruction breakpoints are faults; other debug exceptions are traps. The debug exception may report either or both at one time. The following sections present details for each class of debug exception.

A.3.3.1.1 Instruction-Breakpoint Fault

The microprocessor reports an instruction breakpoint before it executes the breakpointed instruction (i.e., a debug exception caused by an instruction breakpoint is a fault).

The Resume Flag (RF) permits the debug exception handler to restart instructions that cause faults other than debug faults. When a debug fault occurs, the system software writer must set the RF bit in the copy of the EFLAGS register that is pushed on the stack in the debug exception handler routine. This bit is set in preparation of resuming the program's execution at the breakpoint address without generating another breakpoint fault on the same instruction.

Note: *RF does not cause breakpoint traps nor other kinds of faults to be ignored.*

The microprocessor clears RF at the successful completion of every instruction except after the IRET instruction, the POPF instruction, POPFD instruction, and JMP, CALL or INT instructions that cause a task switch. These instructions set RF to the value specified by the saved copy of the EFLAGS register.

The microprocessor sets RF in the copy of the EFLAGS register pushed on the stack before entry into any fault handler. When the fault handler is entered for instruction breakpoints, for example, RF is set in the copy of the EFLAGS register pushed on the stack; therefore, the IRET instruction that returns control from the exception handler sets RF in the EFLAGS register and execution resumes at the breakpointed instruction without generating another breakpoint for the same instruction.

If, after a debugger RF is set and the debug handler retries the faulting instruction, it is possible that retrying the instruction will generate other faults. The restart of the instruction after these faults also occurs with RF set, so repeated debug faults continue to be suppressed. The microprocessor clears RF only after successful completion of the instruction.

A.3.3.1.2 Data-Breakpoint Trap

A data-breakpoint exception is a trap (i.e., the processor generates an exception for a data breakpoint after executing the instruction that accesses the breakpointed memory location). The Am486 microprocessor always does exact data breakpoint matching, regardless of GE/LE bit settings. Exact reporting is provided by forcing the Am486 microprocessor execution unit to wait for completion of data operand transfers before beginning execution of the next instruction.

If a debugger needs to save the contents of a write-breakpoint location, it should save the original contents before saving the breakpoint. Because data breakpoints are traps, the original data is overwritten before the trap exception is generated. The handler can report the saved value after the breakpoint is triggered. The data in the debug registers can be used to address the new value stored by the instruction that triggered the breakpoint.

A.3.3.1.3 General-Detect Fault

The general-detect fault occurs when an attempt is made to use the debug registers at the same time they are being used by in-circuit emulation. This additional protection feature is provided to guarantee emulators can have full control over the debug registers when required. The exception handler can detect this condition by checking the state of the BD bit of the DR6 register.

A.3.3.1.4 *Single-Step Trap*

This trap occurs after an instruction is executed if TF was set before the instruction was executed. Note the exception does not occur after an instruction that sets TF. For example, if the POPF instruction is used to set TF, a single-step trap does not occur until after the instruction following the POPF instruction.

The microprocessor clears TF before calling the exception handler. If TF was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The single-step flag normally is not cleared by changing privilege levels inside a task. INT instructions do, however, clear TF. Therefore, software debuggers that single-step code must recognize and emulate INT*n* or INTO instructions rather than executing them directly.

To maintain protection, the operating system should check the current execution privilege level after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This clears TF. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT*n* instruction that calls the interrupt handler.

A.3.3.1.5 *Task-Switch Trap*

The debug exception also occurs after a task switch if the T bit of the new task's TSS is set. The exception occurs after control has passed to the new task, but before the first instruction of that task is executed. The exception handler can detect this condition by examining the BT bit of the DR6 register.

Note: *If the debug exception handler is a task, the T bit of its TSS should not be set. Failure to observe this rule will put the microprocessor in a loop.*

A.3.3.2 Interrupt 3—Breakpoint Instruction

The breakpoint trap is caused by execution of the INT 3h instruction. Typically, a debugger prepares a breakpoint by replacing the first opcode byte of an instruction with the opcode for the breakpoint instruction. When execution of the INT 3h instruction calls the exception handler, the return address points to the first byte of the instruction following the INT 3h instruction.

With older microprocessors, this feature is used extensively for setting instruction breakpoints. With the Am486 microprocessor, this use is more easily handled using the debug registers. However, the breakpoint exception still is useful for breakpointing debuggers, because the breakpoint exception can call an exception handler other than itself. The breakpoint exception also can be useful when it is necessary to set a greater number of breakpoints than permitted by the debug registers, or when breakpoints are being placed in the source code of a program under development.

A.4 CACHING

The Am486 microprocessor has an on-chip internal cache for storing 8 Kbytes of instructions and data. The cache raises system performance by satisfying an internal read request more quickly than a bus cycle to memory. This also reduces the microprocessor's use of the external bus. The internal cache is transparent to program operation.

The Am486 microprocessor can use an external second-level cache outside of the processor chip. An external cache normally improves performance and reduces bus bandwidth required by the Am486 microprocessor.

Caches require special consideration in multiprocessor systems. When one microprocessor accesses data cached in another microprocessor, it must not receive incorrect data. If it modifies data, all other microprocessors that access that data must receive the modified data. This property is called cache consistency. The Am486 microprocessor provides mechanisms that maintain cache consistency in the presence of multiple microprocessors and external caches.

The operation of internal and external caches is transparent to application software, but knowledge of the behavior of these caches may be useful in optimizing software performance. In multiprocessor systems, maintenance of cache consistency may require intervention by system software.

The cache is available in all execution modes: Real Mode, Protected Mode, and Virtual 8086 Mode. For properly designed single-processor systems, the cache can be initially enabled and not require further control.

A.4.1 Introduction to Caching

Caches are often implemented as associative memories. An associative memory has extra storage for each unit of memory, called a tag. When an address is applied to an associative memory, each tag simultaneously compares itself against the address. If a tag matches the address, access is provided to the unit of memory associated with the tag.

This is called a cache hit. If no match occurs, the cache signals a cache miss. A cache miss requires a bus cycle to access main memory. To gain efficiency in the implementation of the internal cache, storage is allocated in chunks of 128 bits, called cache lines. External caches are not likely to use cache lines smaller than those of the internal cache.

The cache of the Am486 microprocessor does not support partially-filled cache lines, so caching a single doubleword requires caching four doublewords. This would be an inefficient use of the cache if it were not for the fact that the microprocessor rarely accesses random locations in memory. Over any small span of time, the microprocessor usually accesses a small number of areas in memory, such as the code segment or the stack, and it usually accesses many neighboring addresses in these areas.

To simplify the hardware implementation, cache lines can only be mapped to aligned 128-bit blocks of main memory. (An aligned 128-bit block begins at an address that is clear in its Low four bits.) When a new cache line is allocated, the microprocessor loads a block from main memory into the cache line. This operation is called a cache line fill. Allocated cache lines are said to be valid. Unallocated cache lines are invalid.

Caching can be write-through or write-back. On reads, both forms of caching operate as described above. On writes, write-through caching updates both cache memory and main memory; write-back caching updates only the cache memory. Write-back caching updates main memory when a write-back operation is performed. Write-back operations are triggered when cache lines need to be deallocated, such as when new cache lines are being allocated in a cache that is already full. Write-back operations also are triggered by the mechanisms used to maintain cache consistency.

The internal cache of the Am486 microprocessor is a write-through cache. It can be used with external caches that are write-through, write-back, or a mixture of both.

A.4.2 Operation of the Internal Cache

Software controls the operating mode of the cache. Caching can be enabled (its state following reset initialization), caching can be disabled while valid cache lines exist (a mode in which the cache acts like a fast, internal RAM), or caching can be fully disabled.

Precautions must be followed when disabling the cache. Whenever CD is set to 1, the Am486 microprocessor does not read external memory if a copy is still in the cache. Whenever NW is set to 1, the Am486 microprocessor does not write to external memory if the data is in the cache. This means stale data can develop in the Am486 CPU cache. This stale data is not written to external memory if NW is later set to 0 or that cache line is later overwritten as a result of a cache miss. In general, the cache should be flushed when disabled. It is possible to freeze data in the cache by loading it using test registers while CD and NW are set. This is useful to provide guaranteed cache hits for time critical interrupt code and data.

***Note:** All segments should start on 16-byte boundaries to allow programs to align code/ data in cache lines.*

A.4.2.1 Cache Disabling Bits

Table A-19 summarizes the modes enabled by the CD and NW bits.

Table A-19 Cache Operating Modes

CD	NW	Description
1	1	Caching is disabled, but valid cache lines continue to respond. To disable the cache completely, enter this mode and perform a cache flush. To use the cache as fast internal RAM, preload the cache with valid cache lines by carefully choosing memory operations or by using the test registers. In this mode, writes to valid cache lines update the cache, but do not update main memory.
1	0	No new cache lines are allocated, but valid cache lines continue to respond.
0	1	Invalid setting. A general-protection exception with an error code 0 occurs.
0	0	Caching is enabled.

A.4.2.2 Cache Management Instructions

The INVD and WBINVD instructions are used to invalidate the contents of the internal and external caches. The INVD instruction flushes the internal cache and generates a special bus cycle that indicates that external caches also should be flushed. (The response of hardware to receiving a cache flush bus cycle is implementation dependent; hardware might use some other mechanism for maintaining cache consistency.)

There is only one difference between the WBINVD and INVD instructions. The WBINVD instruction generates a special bus cycle that indicates external, write-back caches should write-back modified data to main memory. This cycle is produced immediately before the cycle to flush the cache.

A.4.2.3 Self-Modifying Code

A write to an instruction in the cache modifies it in both cache and memory, but if the instruction is prefetched before the write, the old version of the instruction can be the one executed. To prevent this, flush the instruction prefetch unit by coding a jump instruction immediately after any write that modifies an instruction.

A.4.3 Page-Level Cache Management

The Am486 microprocessor defines two bits in entries in the page directory and second-level page tables that are reserved on Am386 microprocessors. These bits are used to drive microprocessor output pins. These bits are used to manage the caching of pages.

The PCD and PWT bits control caching on a page-by-page basis. The PCD bit (page-level cache disable) affects the operation of the internal cache. Both the PCD bit and the PWT bit (page-level write-through) drive microprocessor output pins for controlling external caches. The treatment of these signals by external hardware is implementation dependent; for example, some hardware systems may control the caching of pages by decoding some of the High address bits.

There are three potential sources of the bits used to drive the PCD and PWT outputs of the microprocessor: the CR3 register, the page directory, and the second-level page tables. The microprocessor outputs are driven by the CR3 register for bus cycles where paging is not used to generate the address, such as the loading of an entry in the page directory. The outputs are driven by a page directory entry when an entry from a second-level page table is accessed. The outputs are driven by a second-level page table entry when instructions or data in memory are accessed. When paging is disabled, these bits are ignored (CPU assumes PCD = 0 and PWT = 0).

A.4.3.1 PCD Bit

When a page table entry has a set PCD bit (bit position 4), caching of the page is disabled, even if hardware is requesting caching by asserting the $\overline{\text{KEN}}$ input. When the PCD bit is clear, caching may be requested by hardware on a cycle-by-cycle basis. Disabling caching is necessary for pages that contain memory-mapped I/O ports. It also is useful for pages that do not provide a performance benefit when cached, such as initialization software.

Regardless of the page-table entries, the Am486 microprocessor ignores the PCD output (assume PCD =O) whenever the CD (Cache Disable) bit in CR0 is set.

A.4.3.2 PWT Bit

When a page table entry has a set PWT bit (bit position 3), a write-through caching policy is specified for data in the corresponding page. Clearing the PWT bit allows the possibility of using a write-back policy for the page. Since the internal cache of the Am486 microprocessor is a write-through cache, it is not affected by the state of the PWT bit. External caches however may use write-back caching, and so they can use the output signal driven by the PWT bit to control caching policy on a page-by-page basis.

In multiprocessor systems, enabling write-through may be advantageous for shared memory, particularly for memory locations written infrequently by one microprocessor, but read often by many microprocessors.

APPENDIX

B OPCODE MAP

B.1 GENERAL

The opcode tables aid in interpreting the 486 processor object code. Use the high-order four bits of the opcode as an index to a row of the opcode table; use the low order four bits as an index to a column of the table. If the opcode is 0Fh, refer to the two-byte opcode table and use the second byte of the opcode to index the rows and columns of that table.

B.2 KEY TO ABBREVIATIONS

Operands are identified by a two-character code of the form Zz. The uppercase letter specifies the addressing method; the lowercase letter specifies the type of operand.

B.3 CODES FOR ADDRESSING METHOD

A Direct address; the instruction has no mod R/M byte; the operand address is encoded in the instruction; no base register, index register, or scaling factor can be applied; for example, JMP (EA)

C The reg field of the mod R/M byte selects a control register; for example, MOV (0F20, 0F22)

D The reg field of the mod R/M byte selects a debug register; for example, MOV (0F21,0F23)

E A mod R/M byte follows the opcode and specifies the operand, either a general register or a memory address. If a memory address, it is computed from a segment register and any of the following values: a base register, an index register, a scaling factor, a displacement.

F Flags Register

G The reg field of the mod R/M byte selects a general register; for example, ADD (00)

I Immediate data. The value of the operand is encoded in subsequent bytes of the instruction.

J The instruction contains a relative offset to be added to the instruction pointer register; for example, JMP short, LOOP.

M The mod R/M byte only refers to memory; for example, BOUND, LES, LDS, LSS, LFS, LGS.

O The instruction has no mod R/M byte; the offset of the operand is coded as a word or doubleword (depending on address size attribute) in the instruction. No base register, index register, or scaling factor can be applied; for example, MOV (A3–A0).

R The mod field of the mod R/M byte may refer only to a general register; for example, MOV (0F20–0F24, 0F26).

S The reg field of the mod R/M byte selects a segment register; for example, MOV (8C,8E).

T The reg field of the mod R/M byte selects a test register; for example, MOV (0F24,F26)

X Memory addressed by the DS:SI register pair; for example, MOVS, COMPS, OUTS, LODS, SCAS.

Y Memory addressed by the ES:DI register pair; for example, MOVS, CMPS, INS, STOS.

B.4 CODES FOR OPERAND TYPE

a Two one-word operands in memory or two doubleword operands in memory, depending on operand size attribute (used only by BOUND)

b Byte (regardless of operand size attribute)

c Byte or word, depending on operand size attribute

d Doubleword (regardless of operand size attribute)

p 32-bit or 48-bit pointer, depending on operand size attribute

s 6-byte pseudo-descriptor

v Word or doubleword, depending on operand size attribute

w Word (regardless of operand size attribute)

B.5 REGISTER CODES

The register name in the opcode indicates whether the register is 32-, 16-, or 8-bits wide. A register identifier in the form eXX indicates the width of the register depends on the operand size; for example eAX indicates the AX register (16 bit) or the EAX register (32 bit).

One-Byte Opcode Map

	0	1	2	3	4	5	6	7
0	ADD						PUSH ES	POP ES
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
1	ADC						PUSH SS	POP SS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
2	AND						SEG=ES	DAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
3	XOR						SEG=SS	AAA
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
4	INC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eSI
5	PUSH general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eSI
6	PUSHA	POPA	BOUND Gv,Ma	ARPL Ew,Rw	SEG=FS	SEG=GS	Operand Size	Address Size
7	Short-displacement jump on condition (Jxx)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
8	Immediate Grpl		MOVB	Grpl Ev,Ib	TEST		XCHG	
	Eb,Ib	Ev,Iv	AL,imm8		Eb,Gb	Ev,Gv	Eb,Gb	Ev,Gv
9	NOP	XCHG word or doubleword register with eAX						
		eCX	eDX	eBX	eSP	eBP	eSI	eDI
A	MOV				MOVSB Xb,Yb	MOVSW/D Xv,Yv	CMPSB Xb,Yb	CMPSW/D Xv,Yv
	AL,Ob	eAX,Ov	Ob,AL	Ov,eAX				
B	MOV immediate byte into byte register							
	AL	CL	DL	BL	AH	CH	DH	BH
C	Shift Grp2		RET near		LES Gv,Mp	LDS GV,Mp	MOV	
	Eb,Ib	Ev,Ib	Iw				Eb,Ib	Ev,Iv
D	Shift Grp2				AAM	AAD		XLAT
	Eb,1	Ev,1	Eb,CL	Ev,CL				
E	LOOPNE Jb	LOOPE Jb	LOOP Jb	JCXZ Jb	IN		OUT	
					AL,Ib	eAX	Ib,AL	Ib,eAX
F	LOCK		REPNE	REP REPE	HLT	CMC	Unary Grp3	
							Eb	Ev

One-Byte Opcode Map

	8	9	A	B	C	D	E	F
0	OR						PUSH CS	POP CS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
1	SBB						PUSH DS	POP DS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
2	SUB						SEG=CS	DAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
3	CMP						SEG=DS	AAS
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev	AL,Ib	eAX,Iv		
4	DEC general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eSI
5	POP into general register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eSI
6	PUSH Iv	IMUL GvEvIv	PUSH Ib	IMUL GvEvIb	INSB Yb,DX	INSW/D Yv,DX	OUTSB DX,Xb	OUTSW/D DX,Xv
7	Short-displacement jump on condition (Jxx)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
8	MOV				MOV Ew,Sw	LEA Gv,M	MOV Sw,Ew	POP Ev
	Eb,Gb	Ev,Gv	Gb,Eb	Gv,Ev				
9	CBW	CWD	CALL Ap	WAIT	PUSHF Fv	POPF Fv	SAHF	LAHF
A	TEST		STOSB Yb,AL	STOSW/D Yv,eAX	LODSB AL,Xb	LODSW/D eAX,Xv	SCASB AL,Xb	SCASW/D eAX,Xv
	AL,Ib	eAX,Iv						
B	MOV immediate word or doubleword into word or doubleword register							
	eAX	eCX	eDX	eBX	eSP	eBP	eSI	eDI
C	ENTER Iw,iB	LEAVE	RET far		INT 3	INT Ib	INTO	IRET
			Iw					
D	ESC (Escape to coprocessor instruction set)							
E	CALL Jv	JMP			IN		OUT	
		JV	AP	Jb	AL,DX	eAX,DX	DX,AL	DX,eAX
F	CLC	STC	CLI	STI	CLD	STD	INC/DEC Grp4	INC/DEC Grp5

Two-Byte Opcode Map (first byte is 0Fh)

	0	1	2	3	4	5	6	7
0	Grp6	Grp7	LAR Gv,Ew	LSL Gv,Ew			CLTS	
1								
2	MOV Cd,Rd	MOV Dd,Rd	MOV Rd,Cd	MOV Td,Rd		MOV Rd,Td		
3								
4								
5								
6								
7								
8	Long-displacement jump on condition (Jxx)							
	JO	JNO	JB	JNB	JZ	JNZ	JBE	JNBE
9	Byte Set on condition (Eb)							
	SETO	SETNO	SETB	SETNB	SETZ	SETNZ	SETBE	SETNBE
A	PUSH FS	POP FS		BT Ev,Gv	SHLD EvGvIb	SHLD EvGvCL	CMPXCHG Eb,Gb	CMPXCHG Ev,Gv
B			LSS Mp	BTR Ev,Gv	LFS Mp	LGS Mp	MOVZX	
							Gv,Eb	Gv,Ew
C	XADD Eb,Gb	XADD Ev,Gv						
D								
E								
F								

Two-Byte Opcode Map (first byte is 0Fh)

	8	9	A	B	C	D	E	F
0	INVD	WBINVD						
1								
2								
3								
4								
5								
6								
7								
8	Long-displacement jump on condition (Jxx)							
	JS	JNS	JP	JNP	JL	JNL	JLE	JNLE
9	Byte Set on condition (Eb)							
	SETS	SETNS	SETP	SETNP	SETL	SETNL	SETLE	SETNLE
A	PUSH GS	POP GS		BTS Ev,Gv	SHRD EvGvIb	SHRD EvGvCL		IMUL Gv,Ev
B			Grp8 Ev,Ib	BTC Ev,Gv	BSF Gv,Ev	BSR Gv,Ev	MOVSX	
							Gv,Eb	Gv,Ew
C	BSWAP EAX	BSWAP ECX	BSWAP EDX	BSWAP EBX	BSWAP ESP	BSWAP EBP	BSWAP ESI	BSWAP EDI
D								
E								
F								

Opcodes determined by bits 5, 4, 3 or mod R/M byte:

mod	nnn	R/M

	000	001	010	011	100	101	110	111
1	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
2	ROL	ROR	RCL	RCR	SHL	SHR	SHL	SAR
3	TEST Ib/Iv	TEST Ib/Iv	NOT	NEG	MUL AL/eAX	IMUL AL/eAX	DIV AL/eAX	IDIV AL/eAX
4	INC Eb	DEC Eb						
5	INC Ev	IDEC Ev	CALL Ev	CALL eP	JMP Ev	JMP Ep	PUSH Ev	
6	SLDT Ew	STR Ew	LLDT Ew	LTR Ew	VERR Ew	VERW Ew		
7	SGDT Ms	SIDT Ms	LGDT Ms	LIDT Ms	SMSW Ew		LMSW Ew	
8					BT	BTS	BTR	BTC

APPENDIX

C FLAG CROSS-REFERENCE

C.1 KEY TO CODES

T	=	Instruction test flags
M	=	Instruction modifies flag (either sets or resets depending on operands)
0	=	Instruction resets flag
1	=	Instruction sets flag
—	=	Instruction’s effect on flag is undefined
R	=	Instruction restores prior value of flag
blank	=	Instruction does not affect flag

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
AAA	—	—	—	TM	—	M					
AAD	—	M	M	—	M	—					
AAM	—	M	M	—	M	—					
AAS	—	—	—	TM	—	M					
ADC	M	M	M	M	M	TM					
ADD	M	M	M	M	M	M					
AND	0	M	M	—	M	0					
ARPL			M								
BOUND											
BSF/BSR	—	—	M	—	—	—					
BSWAP											
BT/BTS/BTR/BTC	—	—	—	—	—	M					
CALL											
CBW											
CLC						0					
CLD									0		
CLI								0			
CLTS											
CMC						M					
CMP	M	M	M	M	M	M					
CMPS	M	M	M	M	M	M				T	
CMPSCHG	M	M	M	M	M	M					
CWD											
DAA	—	M	M	TM	M	TM					
DAS	—	M	M	TM	M	TM					
DEC	M	M	M	M	M						
DIV	—	—	—	—	—	—					
ENTER											
ESC											
HLT											
IDIV	—	—	—	—	—	—					
IMUL	M	—	—	—	—	M					
IN											
INC	M	M	M	M	M						
INS									T		
INT							0			0	
INTO	T						0			0	
INVD											
INVLPG											
IRET	R	R	R	R	R	R	R	R	R	T	
Jcond	T	T	T		T	T					
JCXZ											
JMP											

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
LAHF											
LAR			M								
LDS/LES/LSS/LFS/LGS											
LEA											
LEAVE											
LGDT/LIDT/LLDT/LMSW											
LOCK											
LODS									T		
LOOP											
LOOPE/LOOPNE			T								
LSL			M								
LTR											
MOV											
MOV control, debug	—	—	—	—	—	—					
MOVS									T		
MOVSX/MOVZX											
MUL	M	—	—	—	—	M					
NEG	M	M	M	M	M	M					
NOP											
NOT											
OR	0	M	M	—	M	0					
OUT											
OUTS											
POP/POPA									T		
POPF	R	R	R	R	R	R	R	R	R	R	
PUSH/PUSHA/PUSHF											
RCL/RCR 1	M					TM					
RCL/RCR count	—					TM					
REP/REPE/REPNE											
RET											
ROL/ROR 1	M					M					
ROL/ROR count	—					M					
SAHF		R	R	R	R	R					
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M					
SAL/SAR/SHL/SHR count	—	M	M	—	M	M					
SBB	M	M	M	M	M	TM					
SCAS	M	M	M	M	M	M			T		
SET cond	T	T	T		T	T					
SGDT/SIDT/SLDT/SMSW											
SHLD/SHRD	—	M	M	—	M	M					
STC						1					
STD									1		
STI								1			
STOS									T		
STR											
SUB	M	M	M	M	M	M					
TEST	0	M	M	—	M	0					
VERR/VERW			M								
WAIT											
WBINVD											
XADD	M	M	M	M	M	M					
XCHG											
XLAT											
XOR	0	M	M	—	M	0					

Flag Definitions:
OF = Overflow Flag: When set, the number of digits in the result exceeds the destination operand size.
SF = Sign Flag: When set, the result is negative.
ZF = Zero Flag: When set, the result is zero.
AF = Adjust Flag: When set, there is a carry from or borrow to the low order 4 bits of AL in decimal.
PF = Parity Flag: When set, the low order byte of the result has an even number of 1 bits.
CF = Carry Flag: When set, there is a high order bit carry to or borrow.
TF = Trap Flag: When set, the processor goes into single-step mode for debugging.
IF = Interrupt Enable Flag: When set, the processor can respond to maskable interrupt requests.
DF = Directory Flag: When set, the processor decrements the index registers ESI and EDI.
NT = Nested Flag: Used to control chaining of interrupted and called tasks.
RF = Resume Flag: When set, temporarily disables debug exceptions to allow normal running.

APPENDIX

D CONDITION CODES

D.1 CONDITION CODES FOR CONDITIONAL JUMP AND SET INSTRUCTIONS

Mnemonic	Meaning	Instruction Subcode	Condition Tested
A	Above	0111	(CF or ZF) = 0
AE	Above or equal	0011	CF = 0
B	Below	0010	CF=1
BE	Below or equal	0110	(CF or ZF) = 1
E	Equal	0100	ZF = 1
GE	Great or equal	1101	(SF xor OF) = 0
L	Less	1100	(SF xor OF) = 1
LE	Greater	1111	((SF xor OF) or ZF) = 0
LE	Less or equal	1110	((SF xor OF) or ZF) = 1
LE	Neither less nor equal	1111	((SF xor OF) or ZF) = 0
NA	Not above	0110	(CF or ZF) = 1
NAE	Neither above nor equal	0010	CF = 1
NB	Not below	0011	CF = 0
NBE	Not below or equal	0111	(CF or ZF) = 0
NE	Not equal	0101	ZF = 0
NG	Not greater	1110	((SF xor OF) or ZF) = 1
NGE	Not greater nor equal	1100	(SF xor OF) = 1
NL	Not less	1101	(SF xor OF) = 0
NO	No overflow	0001	OF = 0
NP	No parity	1011	PF = 0
NS	No sign	1001	SF = 0
NZ	Not zero	0101	ZF = 0
O	Overflow	0000	OF = 1
P	Parity	1010	PF = 1
PE	Parity even	1010	PF = 1
PO	Parity odd	1011	PF = 0
S	Sign	1000	SF = 1
Z	Zero	0100	ZF = 1

Note: *The terms “above” and “below” refer to the relation between two unsigned values (neither the SF flag nor the OF flag is tested). The terms “greater” and “less” refer to the relation between two signed values (the SF and OF flags are tested).*

E INSTRUCTION FORMAT AND TIMING

E.1 INSTRUCTION ENCODING AND CLOCK COUNT SUMMARY

To calculate elapsed time for an instruction, multiply the instruction clock count, as listed in Table E-1, by the processor clock period. For more detailed information on the encodings of instructions, refer to Section E.3, Instruction Encodings. Section E.3 explains the general structure of instruction encodings and defines the exact encodings of all fields contained within the instruction.

The Am486 microprocessor instruction clock count tables give clock counts, assuming data and instruction accesses hit in the cache. A separate penalty column defines clocks to add if a data access misses in the cache. The combined instruction and data cache hit rate is over 90%.

A cache miss forces the Am486 microprocessor to run an external bus cycle. The Am486 microprocessor 32-bit burst bus is defined as r-b-w, where:

- r = The number of clocks in the first cycle of a burst read or the number of clocks per data cycle in a non-burst read.
- b = The number of clocks for the second and subsequent cycles in a burst read.
- w = The number of clocks for a write.

The fastest bus the Am486 microprocessor can support is 2-1-2, assuming 0 wait states. The clock counts in the cache miss penalty column assume a 2-1-2 bus. For slower buses, add r-2 clocks to the cache miss penalty for the first dword accessed.

E.2 FACTORS THAT AFFECT INSTRUCTION CLOCK COUNTS

1. The external bus is available for reads or writes at all times. Else, add clocks to reads until the bus is available.
2. Accesses are aligned. Add three clocks to each misaligned access.
3. Cache fills complete before subsequent accesses to the same line. If a read misses the cache during a cache fill due to a previous read or prefetch, the read must wait for the cache fill to complete. If a read or write accesses a cache line still being filled, it must wait for the fill to complete.
4. If an effective address is calculated, the base register is not the destination register of the preceding instruction. If the base register is the destination register of the preceding instruction, add 1 to the clock counts shown. Back-to-back PUSH and POP instructions are not affected by this rule.
5. An effective address calculation uses one base register and does not use an index register. However, if the effective address calculation uses an index register, one clock may be added to the clock count shown.
6. The target of a jump is in the cache. If not, add r clocks for accessing the destination instruction of a jump. If the destination instruction is not completely contained in the first dword read, add a maximum of 3b clocks. If the destination instruction is not completely contained in the first 16-byte burst, add a maximum of another r+3b clocks.

7. If no write buffer delay, w clocks are added only in the case in which all write buffers are full. This case rarely occurs.

8. Displacement and immediate are not used together. If displacement and immediate are used together, one clock can be added to the clock count shown.

9. No invalidate cycles. Add a delay of one clock for each invalidate cycle if the invalidate cycle contends for the internal cache/external bus when the Am486 CPU needs to use it.

10. Page translation hits in TLB. A TLB miss adds 13, 21, or 28 clocks to the instruction, depending on whether the accessed and/or dirty bit in neither, one, or both of the page entries needs to be set in memory. This assumes that neither page entry is in the data cache and a page fault does not occur on the address translation.

11. No exceptions are detected during instruction execution.

12. Instructions that read multiple consecutive data items (i.e., task switch, POPA, etc.) and miss the cache are assumed to start the first access on a 16-byte boundary. If not, an extra cache line fill might be necessary and might add up to (r+3b) clocks to the cache miss penalty.

Table E-1 Instruction Clock Count Summary

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
AAA = ASCII adjust AL after add	00110111			3		
AAD = ASCII adjust AX before divide	11010101	00001010		14		
AAM = ASCII adjust AX after multiply	11010100	00001010		15		
AAS = ASCII Adjust AL after subtract	00111111			3		
ADC = Add with carry						
reg1 to reg2	0001000w	11 reg1 reg2		1		
reg2 to reg1	0001001w	11 reg1 reg2		1		
memory to register	0001001w	mod reg r/m		2	2	
register to memory	0001000w	mod reg r/m		3	6/2	No LOCK/LOCK
immediate to register	100000sw	11 010 reg	immediate register	1		
immediate to accumulator	0001010w	immediate data		1		
immediate to memory	100000sw	mod 010 r/m	immediate data	3	6/2	No LOCK/LOCK
ADD = Add						
reg1 to reg2	0000000w	11 reg1 reg2		1		
reg2 to reg1	0000001w	11 reg1 reg2		1		
memory to register	0000001w	mod reg r/m		2	2	
register to memory	0000000w	mod reg r/m		3	6/2	No LOCK/LOCK
immediate to register	100000sw	11 000 reg	immediate register	1		
immediate to accumulator	0000010w	immediate data		1		
immediate to memory	100000sw	mod 000 r/m	immediate data	3	6/2	No LOCK/LOCK
Address Size	01100111			1		Prefix

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
AND = Logical AND						
reg1 to reg2	0010000w	11 reg1 reg2		1		
reg2 to reg1	0010001w	11 reg1 reg2		1		
memory to register	0010001w	mod reg r/m		2	2	
register to memory	0010000w	mod reg r/m		3	6/2	No LOCK/LOCK
immediate to register	100000sw	11 100 reg	immediate register	1		
immediate to accumulator	0010010w	immediate data		1		
immediate to memory	100000sw	mod 100 r/m	immediate data	3	6/2	No LOCK/LOCK
ARPL = Adjust RPL field of selector						
From Register	01100011	11 reg1 reg2		9		
From Memory	01100011	mod reg r/m		9		
BOUND = Check array index bounds (generates INT 5 if out of bounds)						
If in range	01100010	mod reg r/m		7	7	
If out of range	01100010	mod reg r/m				
Real Mode				50	7	
Protected Mode:						
Int/Trap Gate, same level				68	7	Add 11 clocks for each unaccessed descriptor load.
Int/Trap Gate, diff. level				95	7	
Task Gate:						
VM/486/286 to 486 TSS				223	7	
VM/486/286 to 286 TSS				204	7	
VM/486/286 to VM TSS				201	7	
Virtual Mode:						
Int/Trap Gate, diff. level				106	7	
Task Gate:						
VM/486/286 to 486 TSS				223	7	
VM/486/286 to 286 TSS				204	7	
VM/486/286 to VM TSS				201	7	
BSF = Bit can Forward				Variables: b = number of bytes not 0 (0–3) i = number of nibbles/byte not 0 (0–1) n = number of bits/nibble not 0 (0–3)		
reg1, reg2	00001111	10111100	11 reg2 reg1	6 to 42		If operand2 is 0, clocks = 6. Else, clocks = 8 + 4(b+1) + 3(i+1) + 3(n+1)
memory, reg	00001111	10111100	mod reg r/m	7 to 43	2	If operand2 is 0, clocks = 7. Else, clocks = 9 + 4(b+1) + 3(i+1) + 3(n+1)
BSR = Bit Scan Reverse				Variable: n = bit position number (0–31)		
reg1, reg2	00001111	10111101	11 reg2 reg1	6 to 103		If operand2 is 0, clocks = 6. Else, clocks = 7+ 3(32 – n)
memory, reg	00001111	10111101	mod reg r/m	7 to 104	1	If operand2 is 0, clocks = 7. Else, clocks = 8 + 3(32 – n)
BSWAP = Byte Swap	00001111	11001 reg		1		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
BT = Bit Test						
register, immediate	00001111	10111010	11 100 reg imm. byte	3		
memory, immediate	00001111	10111010	mod 100 r/m imm. byte	3	1	
reg1, reg2	00001111	10100011	11 reg2 reg1	3		
memory, reg	00001111	10100011	mod reg r/m	8	2	
BTC = Bit Test and Complement						
register, immediate	00001111	10111010	11 111 reg imm. byte	6		
memory, immediate	00001111	10111010	mod 111 r/m imm. byte	8	2/0	No LOCK/LOCK
reg1, reg2	00001111	10111011	11 reg2 reg1	6		
memory, reg	00001111	10111011	mod reg r/m	13	3/1	No LOCK/LOCK
BTR = Bit Test and Reset						
register, immediate	00001111	10111010	11 110 reg imm. byte	6		
memory, immediate	00001111	10111010	mod 110 r/m imm. byte	8	2/0	No LOCK/LOCK
reg1, reg2	00001111	10110011	11 reg2 reg1	6		
memory, reg	00001111	10110011	mod reg r/m	13	3/1	No LOCK/LOCK
BTS = Bit Test and Set						
register, immediate	00001111	10111010	11 101 reg imm. byte	6		
memory, immediate	00001111	10111010	mod 101 r/m imm. byte	8	2/0	No LOCK/LOCK
reg1, reg2	00001111	10101011	11 reg2 reg1	6		
memory, reg	00001111	10101011	mod reg r/m	13	3/1	No LOCK/LOCK
CALL = Call Procedure						
Within Segment						Real Mode; assumes memory read, stack push/pop, and branch in different cache sets; clocks include 1 for displacement + immediate
Direct	11101000	full displacement		3	See factor 6, p. E-1	
Register indirect	11111111	11 010 reg		5		
Memory indirect	11111111	mod 010 r/m		5	5	
Direct Intersegment	10011010	unsigned full offset, selector		18	2	
same level				20	3	Add 11 clocks for each unaccessed descriptor load.
thru Gate — same level				35	6	
inner level, no parameters				69	17	
inner level, x parameters (d) words				77 + 4(x)	17 + 4(x)	
to TSS:						
VM/486/286 to 486 TSS				199	3	
VM/486/286 to 286 TSS				180	3	
VM/486/286 to VM TSS				177	3	
thru Task Gate:						
VM/486/286 to 486 TSS				200	3	
VM/486/286 to 286 TSS				181	3	
VM/486/286 to VM TSS				178	3	

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
CALL (continued)						
Indirect Intersegment	11111111	mod 011 r/m		17	8	Real Mode; assumes mem. read, stack push/pop, and branch in different cache sets; clocks include 1 for displacement + immediate
same level				20	10	Add 11 clocks for each unaccessed descriptor load.
thru Gate — same level				35	13	
inner level, no parameters				69	24	
inner level, x = number of parameter words				77 + 4(x)	24 + 4(x)	
to TSS:						
VM/486/286 to 486 TSS				199	10	
VM/486/286 to 286 TSS				180	10	
VM/486/286 to VM TSS				177	10	
thru Task Gate:						
VM/486/286 to 486 TSS				200	10	
VM/486/286 to 286 TSS				181	10	
VM/486/286 to VM TSS				178	10	
CBW = Convert Byte to Word	10011000			3		
CDQ = Convert Dword to Qword	10011001			3		
CLC = Clear Carry Flag	11111000			2		
CLD = Clear Direction Flag	11111100			2		
CLI = Clear Interrupt-Enable Flag	11111010			2		
CLTS = Clear Task Switched Flag	00001111	00000110		7	2	
CMC = Complement Carry Flag	11110101			2		
CMP = Compare						
reg1 with reg2	0011100w	11 reg1 reg2		1		
reg2 with reg1	0011101w	11 reg1 reg2		1		
memory with register	0011100w	mod reg r/m		2	2	
register with memory	0011101w	mod reg r/m		2	2	
immediate with register	100000sw	11 111 reg	immediate data	1		
immediate with accumulator	0011110w	immediate data		1		
immediate with memory	100000sw	mod 111 r/m	immediate data	2	2	
CMPS = Compare 2 Strings **CMPSB = Compare 2 Bytes** **CMPSD = Compare 2 Dwords** **CMPSW = Compare 2 Words**	1010011w			8	6	16
CMPXCHG = Compare/Exchange						
reg1, reg2	00001111	1011000w	11 reg2 reg1	6		
memory, reg:	00001111	1011000w	mod reg r/m			
equal				7	2	
not equal				10	2	
CWD = Convert Word to Dword	10011001			3		
CWDE = Convert Word to Dword	10011000			3		
DAA = Decimal Adjust after Add	00100111			2		
DAS = Decimal Adjust after Subtract	00101111			2		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
DEC = Decrement						
reg	1111111w	11 001 reg		1		
or	01001 reg			1		
memory	1111111w	mod 001 r/m		3	6/2	No LOCK/LOCK
DIV = Divide (unsigned)						
accumulator by reg.	1111011w	11 110 reg				
divisor-byte				16		
divisor-word				24		
divisor-dword				40		
accumulator by mem.	1111011w	mod 110 r/m				
divisor-byte				16		
divisor-word				24		
divisor-dword				40		
ENTER = Enter Procedure	11001000	16-bit displacement, 8-bit level				
Level = 0				14		
Level = 1				17		
Level (L) > 1				17 + 3(L)	6(n)	n = number of words copied to new stack frame
				Avg. (range)		Concurr. Exec.
F2XM1 = Compute $2^{ST(0)}$ –1	11011001	11110000		242 (140–279)		2
						Continuous INT polling to ensure short interrupt latency.
FABS = Absolute Value of ST(0)	11011001	11100001		3		
FADD = Add Real to ST(0)						Concurr. Exec.
				Avg. (range)		Avg. (range)
ST(0) ⟵ ST(0) + 32-bit memory	11011 000	mod 000 r/m	s-i-b/displacement	10 (8–20)	2	7 (5–17)
ST(0) ⟵ ST(0) + 64-bit memory	11011 100	mod 000 r/m	s-i-b/displacement	10 (8–20)	3	7 (5–17)
ST(d) ⟵ ST(0) + ST(i)	11011 d00	11100 ST(i)		10 (8–20)		7 (5–17)
FADDP = Add Floating-Point and	11011 110	11000 ST(i)		Avg. (range)		Concurr. Exec.
Pop Stack				10 (8–10)		Avg. (range)
						7 (5–17)
FBLD = Load BCD to ST(0)	11011 111	mod 100 r/m	s-i-b/displacement	Avg. (range)	4	Concurr. Exec.
				75 (70–103)		Avg. (range)
						7.7 (2–8)
FBSTP = Store BCD & Pop Stack	11011 111	mod 110 r/m	s-i-b/displacement	Avg. (range)		
				175 (172–176)		
FCHS = Change Sign	11011 001	1110 0000		6		
FCLEX = Clear Exceptions after	1001 1011	11011 011	1110 0010			
Checking for FPU Error						
No error pending				7		
Error pending				24		
FCOM = Compare ST and Real						Concurr. Exec.
32-bit memory	11011 000	mod 010 r/m	s-i-b/displacement	4	2	1
64-bit memory	11011 100	mod 010 r/m	s-i-b/displacement	4	3	1
ST(i)	11011 000	11010 ST(i)		4		1
FCOMP = Compare Real and Pop						Concurr. Exec.
32-bit memory	11011 000	mod 011 r/m	s-i-b/displacement	4	2	1
64-bit memory	11011 100	mod 011 r/m	s-i-b/displacement	4	3	1
ST(i)	11011 000	11011 ST(i)		4		1

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
FCOMPP = Compare Real and Pop Stack Twice	11011 110	1101 1001		5		Concurr. Exec. 1
FCOS = Cosine ST(0)	11011 001	1111 1111		Avg. (range) 241 (193–279) If lST(0)l > π/4, add *n*, where *n* = [ST(0)/(π/4)]		Concurr. Exec. 2 Continuous INT polling to ensure short interrupt latency.
FDECSTP = Decrement Stack Pointer	11011 001	1111 0110		3		
FDIV = Divide Real						Concurr. Exec.
ST(0) ⟵ ST(0) / 32-bit mem	11011 000	mod 110 r/m	s-i-b/displacement	73	2	70
24-bit precision				35		32
53-bit precision				62		59
ST(0) ⟵ ST(0) / 64-bit mem	11011 100	mod 110 r/m	s-i-b/displacement	73	3	70
24-bit precision				35		32
53-bit precision				62		59
ST(d) ⟵ ST(0) / ST(i)	11011 d00	11111 ST(i)		73		70
24-bit precision				35		32
53-bit precision				62		59
						Concurr. Exec.
FDIVP = Divide Real and Pop	11011 110	111111 ST(i)		73		70
24-bit precision				35		32
53-bit precision				62		59
FDIVR = Reverse Divide Real						Concurr. Exec.
ST(0) ⟵ 32-bit mem / ST(0)	11011 000	mod 111 r/m	s-i-b/displacement	73	2	70
24-bit precision				35		32
53-bit precision				62		59
ST(0) ⟵ 64-bit mem / ST(0)	11011 100	mod 111 r/m	s-i-b/displacement	73	3	70
24-bit precision				35		32
53-bit precision				62		59
ST(d) ⟵ ST(i) / ST(0)	11011 d00	11110 ST(i)		73		70
24-bit precision				35		32
53-bit precision				62		59
FDIVRP = Reverse Divide Real						Concurr. Exec.
and Pop Stack	11011 110	111110 ST(i)		73		70
24-bit precision				35		32
53-bit precision				62		59
FFREE = Free Floating-Point Register	11011 101	11000 ST(i)		3		
FIADD = Add Integer						Concurr. Exec.
				Avg. (range)		Avg. (range)
ST(0) ⟵ ST(0) + 16-bit memory	11011 110	mod 000 r/m	s-i-b/displacement	24 (20–35)	2	7 (5–17)
ST(0) ⟵ ST(0) + 32-bit memory	11011 010	mod 000 r/m	s-i-b/displacement	22.5 (19–32)	2	7 (5–17)
FICOM = Compare Integer				Avg. (Range)		Concurr. Exec.
16-bit memory	11011 110	mod 010 r/m	s-i-b/displacement	18 (16–20)	2	1
32-bit memory	11011 010	mod 010 r/m	s-i-b/displacement	16.5 (15–17)	2	1
FICOMP = Compare Integer and						
Pop Stack				Avg. (Range)		Concurr. Exec.
32-bit memory	11011 110	mod 011 r/m	s-i-b/displacement	18 (16–20)	2	1
64-bit memory	11011 010	mod 011 r/m	s-i-b/displacement	16.5 (15–17)	2	1

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
FIDIV = Divide Integer				Avg. (range)		Concurr. Exec.
ST(0) ⟵ ST(0) / 16-bit memory	11011 110	mod 110 r/m	s-i-b/displacement	87 (85–89)	2	70
24-bit precision				49 (47–51)	2	32
53-bit precision				76 (74–78)	2	59
ST(0) ⟵ ST(0) / 32-bit memory	11011 010	mod 110 r/m	s-i-b/displacement	85.5 (84–86)	2	70
24-bit precision				47.5 (46–48)	2	32
53-bit precision				74.5 (73–75)	2	59
FIDIVR = Reverse Divide Integer				Avg. (range)		Concurr. Exec.
ST(0) ⟵ 16-bit memory / ST(0)	11011 110	mod 111 r/m	s-i-b/displacement	87 (85–89)	2	70
24-bit precision				49 (47–51)	2	32
53-bit precision				76 (74–78)	2	59
ST(0) ⟵ 32-bit memory / ST(0)	11011 010	mod 111 r/m	s-i-b/displacement	85.5 (84–86)	2	70
24-bit precision				47.5 (46–48)	2	32
53-bit precision				74.5 (73–75)	2	59
FILD = Load Integer ST(0)						Concurr. Exec.
				Avg. (range)		Avg. (range)
32-bit memory	11011111	mod 000 r/m	s-i-b/displacement	14.5 (13–16)	2	4
64-bit memory	11011011	mod 000 r/m	s-i-b/displacement	11.5 (9–12)	2	4 (2–4)
80-bit memory	11011111	mod 101 r/m	s-i-b/displacement	16.8 (10–18)	3	7.8 (2–8)
FIMUL = Multiply Integer				Avg. (range)		Concurr. Exec.
ST(0) ⟵ ST(0) · 16-bit mem	11011 110	mod 001 r/m	s-i-b/displacement	25 (23–27)	2	8
ST(0) ⟵ ST(0) · 32-bit mem	11011 010	mod 001 r/m	s-i-b/displacement	23.5 (22–24)	2	8
FINCSTP = Increment Stack Pointer	11011 001	1111 0111		3		
FINIT = Initialize FPU after Checking for Unmasked Error	1001 1011	11011 011	1110 0011			
No error pending				17		
Error pending				34		
FIST = Store Integer from ST(0)				Avg. (range)		
16-bit memory	11011 111	mod 010 r/m	s-i-b/displacement	33.4 (29–34)		
32-bit memory	11011 011	mod 010 r/m	s-i-b/displacement	32.4 (28–34)		
FISTP = Store Integer and Pop Stack				Avg. (range)		
16-bit memory	11011 111	mod 011 r/m	s-i-b/displacement	33.4 (29–34)		
32-bit memory	11011 011	mod 011 r/m	s-i-b/displacement	33.4 (29–34)		
64-bit memory	11011 111	mod 111 r/m	s-i-b/displacement	33.4 (29–34)		
FISUB = Subtract Integer						Concurr. Exec.
				Avg. (range)		Avg. (range)
ST(0) ⟵ ST(0) – 16-bit memory	11011 110	mod 100 r/m	s-i-b/displacement	24 (20–35)	2	7 (5–17)
ST(0) ⟵ ST(0) – 32-bit memory	11011 010	mod 100 r/m	s-i-b/displacement	22.5 (19–32)	2	7 (5–17)
FISUBR = Reverse Subtr. Integer						Concurr. Exec.
				Avg. (range)		Avg. (range)
ST(0) ⟵ 16-bit memory – ST(0)	11011 110	mod 101 r/m	s-i-b/displacement	24 (20–35)	2	7 (5–17)
ST(0) ⟵ 32-bit memory – ST(0)	11011 010	mod 101 r/m	s-i-b/displacement	22.5 (19–32)	2	7 (5–17)
FLD = Load Real to ST(0)				Avg. (lo–hi)		
32-bit memory	11011001	mod 000 r/m	s-i-b/displacement	3	2	
64-bit memory	11011101	mod 000 r/m	s-i-b/displacement	3	3	
80-bit memory	11011011	mod 101 r/m	s-i-b/displacement	6	4	
ST(i)	11011001	11000 ST(i)		4		
FLD1 = Load Constant +1.0	11011 001	1110 1000		4		
FLDCW = Load Control Word	11011 001	mod 101 r/m	s-i-b/displacement	4	2	

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
FLDENV = Load FPU Environment	11011 001	mod 100 r/m	s-i-b/displacement			
Real/Virtual Mode 16-bit addr.				44	2	
Real/Virtual Mode 32-bit addr.				44	2	
Protected Mode 16-bit address				34	2	
Protected Mode 32-bit address				34	2	
FLDL2E = Load Constant log$_2$e	11011 001	1110 1010		8		Concurr. Exec. 2
FLDL2T = Load Constant log$_2$10	11011 001	1110 1001		8		Concurr. Exec. 2
FLDLG2 = Load Constant log$_{10}$2	11011 001	1110 1100		8		Concurr. Exec. 2
FLDLN2 = Load Constant log$_e$2	11011 001	1110 1101		8		Concurr. Exec. 2
FLDPI = Load Constant π	11011 001	1110 1011		8		Concurr. Exec. 2
FLDZ = Load Constant +0.0	11011 001	1110 1110		4		
FMUL = Multiply Real						Concurr. Exec.
ST(0) $\longleftarrow$ ST(0) $\cdot$ 32-bit mem	11011 000	mod 001 r/m	s-i-b/displacement	11	2	8
ST(0) $\longleftarrow$ ST(0) $\cdot$ 64-bit mem	11011 100	mod 001 r/m	s-i-b/displacement	14	3	11
ST(d) $\longleftarrow$ ST(0) $\cdot$ ST(i)	11011 d00	11001 ST(i)		16		13
FMULP = Multiply Real and Pop Stack	11011 110	11001 ST(i)		16		Concurr. Exec. 13
FNCLEX = Clear Exceptions without Checking for Error	11011 011	1110 0010		7		
FNINIT = Initialize FPU without Checking for Error	11011 011	1110 0011		17		
FNOP = No operation	11011 001	1101 0000		3		
FNSAVE = Store FPU State without Checking for Error	11011 101	mod 110 r/m	s-i-b/displacement			
Real/Virtual Mode 16-bit addr.				154		
Real/Virtual Mode 32-bit addr.				154		
Protected Mode 16-bit address				143		
Protected Mode 32-bit address				143		
FNSTCW = Store Control Word without Checking for Error	11011 001	mod 111 r/m	s-i-b/displacement	3		
FNSTENV = Store FPU Environment without Checking for Error	11011 001	mod 110 r/m	s-i-b/displacement			
Real/Virtual Mode 16-bit addr.				67		
Real/Virtual Mode 32-bit addr.				67		
Protected Mode 16-bit address				56		
Protected Mode 32-bit address				56		
FNSTSW = Store Status Word without Checking for Error						
Into AX	11011 111	1110 0000		3		
Into memory	11011 101	mod 111 r/m	s-i-b/displacement	3		
FPATAN = Partial Arctangent	11011 001	1111 0011		Avg. (range) 289 (218–303)		Concurr. Exec. Avg. (range) 5 (2–17) Continuous INT polling to ensure short interrupt latency.
FPREM = Partial Remainder	11011 001	1111 1000		Avg. (range) 84 (70–138)		Concurr. Exec. Avg. (range) 2 (2–8)

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT				Clocks if Cache Hit	Penalty if Cache Miss	Notes
FPREM1 = Partial Remainder (IEEE 754 compliant)	11011 001	1111 0101			Avg. (range) 94.5 (72–167)		Concurr. Exec. Avg. (range) 5.5 (2–18)
FPTAN = Partial Tangent	11011 001	1111 0010			Avg. (range) 244 (200–273) If |ST(0)| > π/4, add *n*, where *n* = [ST(0)/(π/4)]		Concurr. Exec. 70 Continuous INT polling to ensure short interrupt latency.
FRNDINT = Round to Integer	11011 001	1111 1100			Avg. (range) 29.1 (21–30)		Concurr. Exec. Avg. (range) 5.5 (2–18)
FRSTOR = Restore FPU State	11011 101	mod 100 r/m	s-i-b/displacement				
Real/Virtual Mode 16-bit addr.					131	23	
Real/Virtual Mode 32-bit addr.					131	27	
Protected Mode 16-bit address					120	23	
Protected Mode 32-bit address					120	27	
FSAVE = Store FPU State after checking for error	1001 1011	11011 101	mod 110 r/m	s-i-b/displacement			
Real/Virtual Mode 16-bit addr.							
No error pending					154		
Error pending					171		
Real/Virtual Mode 32-bit addr.							
No error pending					154		
Error pending					171		
Protected Mode 16-bit address							
No error pending					143		
Error pending					160		
Protected Mode 32-bit address							
No error pending					143		
Error pending					160		
FSCALE = Scale	11011 001	1111 1101			Avg. (range) 31 (30–32)		Concurr. Exec. 2
FSIN = Sine	11011 001	1111 1110			Avg. (range) 241 (193–279) If |ST(0)| > π/4, add *n*, where *n* = [ST(0)/(π/4)]		Concurr. Exec. 2 Continuous INT polling to ensure short interrupt latency.
FSINCOS = Sine and Cosine	11011 001	1111 1011			Avg. (range) 291(243–329) If |ST(0)| > π/4, add *n*, where *n* = [ST(0)/(π/4)]		Concurr. Exec. 2 Continuous INT polling to ensure short interrupt latency.
FSQRT = Square Root	11011 001	1111 1010			Avg. (range) 85.5 (83–87)		Concurr. Exec. 70
FST = Store Real							
32-bit memory	11011 001	mod 010 r/m	s-i-b/displacement		7		If op.=0, clks=27
64-bit memory	11011 101	mod 010 r/m	s-i-b/displacement		8		If op.=0, clks=28
ST(i)	11011 101	11010 ST(i)			3		
FSTCW = Store Control Word after checking for error	1001 1011	11011 001	mod 111 r/m	s-i-b/displacement			
No error pending					3		
Error pending					21		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT				Clocks if Cache Hit	Penalty if Cache Miss	Notes
FSTENV = Store Environment after checking for error	1001 1011	11011 001	mod 110 r/m	s-i-b/displacement			
Real/VirtualMode/16-bit Addr.							
No error pending					67		
Error pending					84		
Real/Virtual Mode/32-bit Addr.							
No error pending					67		
Error pending					84		
Protected Mode/16-bit Addr.							
No error pending					56		
Error pending					70		
Protected Mode/32-bit Addr.							
No error pending					56		
Error pending					70		
FSTP = Store Real and Pop Stack							
32-bit memory	11011 001	mod 011 r/m	s-i-b/displacement		7		If op.=0, clks=27
64-bit memory	11011 101	mod 011 r/m	s-i-b/displacement		8		If op.=0, clks=28
80-bit memory	11011 011	mod 111 r/m	s-i-b/displacement		6		
ST(i)	11011 101	11001 ST(i)			3		
FSTSW = Store Status Word after checking for error							
Into AX	1001 1011	11011 111	1110 0000				
No error pending					3		
Error pending					21		
In memory	1001 1011	11011 101	mod 111 r/m	s-i-b/displacement			
No error pending					3		
Error pending					21		
FSUB = Subtract Real							Concurr. Exec.
					Avg. (range)		Avg. (range)
ST(0) ⟵ ST(0) – 32-bit memory	11011 000	mod 100 r/m	s-i-b/displacement		10 (8–20)	2	7 (5–17)
ST(0) ⟵ ST(0) – 64-bit memory	11011 100	mod 100 r/m	s-i-b/displacement		10 (8–20)	3	7 (5–17)
ST(d) ⟵ ST(0) – ST(i)	11011 d00	11101 ST(i)			10 (8–20)		7 (5–17)
FSUBP = Subtract Real and Pop Stack ST(i) ⟵ ST(0) – ST(i)	11011 110	11001 ST(i)			Avg. (range) 10 (8–10)		Concurr. Exec. Avg. (range) 7 (5–17)
FSUBR = Reverse Subtract Real							Concurr. Exec.
					Avg. (range)		Avg. (range)
ST(0) ⟵ 32-bit memory – ST(0)	11011 000	mod 101 r/m	s-i-b/displacement		10 (8–20)	2	7 (5–17)
ST(0) ⟵ 64-bit memory – ST(0)	11011 100	mod 101 r/m	s-i-b/displacement		10 (8–20)	3	7 (5–17)
ST(d) ⟵ ST(i) – ST(0)	11011 d00	11100 ST(i)			10 (8–20)		7 (5–17)
FSUBRP = Reverse Subtract Real and Pop Stack ST(i) ⟵ ST(i) – ST(0)	11011 110	11100 ST(i)			Avg. (range) 10 (8–10)		Concurr. Exec. Avg. (range) 7 (5–17)
FTST = Compare ST(0) to 0.0	11011 001	1110 0100			4		Concurr. Exec. 1
FUCOM = Unordered Compare Real – ST(0) to ST(i)	11011 101	11100 ST(i)			4		Concurr. Exec. 1
FUCOMP = Unordered Compare Real and Pop Stack	11011 101	11101 ST(i)			4		Concurr. Exec. 1
FUCOMPP = Unordered Compare Real and Pop Stack Twice	11011 101	1110 1001			5		Concurr. Exec. 1
FWAIT = Wait	10011011				1 to 3		
FXAM = Examine	11011 001	1110 0101			8		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT		Clocks if Cache Hit	Penalty if Cache Miss	Notes
FXCH = Exchng. ST(0) and ST(i)	11011 001	11001 ST(i)	4		
FXTRACT = Extract Exponent and Significand	11011 001	1111 0100	Avg. (range) 19 (16–20)		Concurr. Exec. Avg. (range) 4 (2–4)
FYL2X = Compute ST(1) · $\log_2$ST(0)	11011 001	1111 0001	Avg. (range) 311 (196–329)		Concurr. Exec. 13 Continuous INT polling to ensure short interrupt latency.
FYL2XP1 = Compute ST(1) · $\log_2$[ST(0) +1]	11011 001	1111 1001	Avg. (range) 313(171–326)		Concurr. Exec. 13 Continuous INT polling to ensure short interrupt latency.
HLT = Halt	11110100		4		
IDIV = Integer Divide (signed)					
accumulator by register	1111011w	11 111 reg			
Divisor:					
Byte			19		
Word			27		
Dword			43		
accumulator by memory	1111011w	mod 111 r/m			
Divisor					
Byte			20		
Word			28		
Dword			44		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
IMUL = Integer Multiply (signed)						
accumulator with register	1111011w	11 101 reg				
Multiplier:						
Byte				13 to 18		For all cases, clocks = 10 + max(log$_2$($\lvert m \rvert$),n) where m = multiplier n = 3/5 for ± m if m = 0, clocks = 13
Word				13 to 26		
Dword				13 to 42		
accumulator with memory	1111011w	mod 101 r/m				
Multiplier:						
Byte				13 to 18		
Word				13 to 26		
Dword				13 to 42		
reg1 with reg2	00001111	10101111	11 reg1 reg2			
Multiplier:						
Byte				13 to 18		
Word				13 to 26		
Dword				13 to 42		
register with memory	00001111	10101111	mod reg r/m			
Multiplier:						
Byte				13 to 18	1	
Word				13 to 26	1	
Dword				13 to 42	1	
reg1 with imm. to reg2	011010s1	11 reg1 reg2	immediate data			
Multiplier:						
Byte				13 to 18		
Word				13 to 26		
Dword				13 to 42		
mem. with imm. to reg.	011010s1	mod reg r/m	immediate data			
Multiplier:						
Byte				13 to 18	2	
Word				13 to 26	2	
Dword				13 to 42	2	
IN = Input from Port						
Fixed Port	1110010w	port number				
Real Mode				14		
Protected Mode:						
CPL ≤ IOPL				9		
CPL > IOPL				29		
Virtual Mode				27		
Variable Port	1110110w					
Real Mode				14		
Protected Mode:						
CPL ≤ IOPL				8		
CPL > IOPL				28		
Virtual Mode				27		
INC = Increment						
reg	1111111w	11 000 reg		1		
or	01000 reg			1		
memory	1111111w	mod 000 r/m		3	6/2	No LOCK/LOCK
INS = Input String from Port	0110110w					
INSB = Input Byte from Port						
INSD = Input Dword from Port						
INSW = Input Word from Port						
Real Mode				17		
Protected Mode:						
CPL ≤ IOPL				10		
CPL > IOPL				32		
Virtual Mode				30		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT	Clocks if Cache Hit	Penalty if Cache Miss	Notes
INT = Call to Interrupt Procedure				
INT n = Interrupt Type n	11001101 type			
Real Mode		26		
Protected Mode:				
Int/Trap Gate, same level		44		Add 11 clocks for each unaccessed descriptor load.
Int/Trap Gate, diff. level		71		
Task Gate:				
VM/486/286 to 486 TSS		199		
VM/486/286 to 286 TSS		180		
VM/486/286 to VM TSS		177		
Virtual Mode:				
Int/Trap Gate, diff. level		82		
Task Gate:				
VM/486/286 to 486 TSS		199		
VM/486/286 to 286 TSS		180		
VM/486/286 to VM TSS		177		
INT 3 = Interrupt Type 3	11001100			
Real Mode		26		
Protected Mode:				
Int/Trap Gate, same level		44		Add 11 clocks for each unaccessed descriptor load.
Int/Trap Gate, diff. level		71		
Task Gate:				
VM/486/286 to 486 TSS		199		
VM/486/286 to 286 TSS		180		
VM/486/286 to VM TSS		177		
Virtual Mode:				
Int/Trap Gate, diff. level		82		
Task Gate:				
VM/486/286 to 486 TSS		199		
VM/486/286 to 286 TSS		180		
VM/486/286 to VM TSS		177		
Hardware Interrupts:				
External Interrupt				
Real Mode		37		
Protected Mode:				
Int/Trap Gate, same level		55		Add 11 clocks for each unaccessed descriptor load.
Int/Trap Gate, diff. level		82		
Task Gate:				
VM/486/286 to 486 TSS		210		
VM/486/286 to 286 TSS		191		
VM/486/286 to VM TSS		188		
Virtual Mode:				
Int/Trap Gate, diff. level		93		
Task Gate:				
VM/486/286 to 486 TSS		210		
VM/486/286 to 286 TSS		191		
VM/486/286 to VM TSS		188		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT	Clocks if Cache Hit	Penalty if Cache Miss	Notes
Hardware Interrupts (continued):				
NMI				
Real Mode		29		
Protected Mode:				
Int/Trap Gate, same level		47		Add 11 clocks for each unaccessed descriptor load.
Int/Trap Gate, diff. level		74		
Task Gate:				
VM/486/286 to 486 TSS		202		
VM/486/286 to 286 TSS		183		
VM/486/286 to VM TSS		180		
Virtual Mode:				
Int/Trap Gate, diff. level		85		
Task Gate:				
VM/486/286 to 486 TSS		202		
VM/486/286 to 286 TSS		183		
VM/486/286 to VM TSS		180		
Page Fault				
Real Mode		50		
Protected Mode:				
Int/Trap Gate, same level		68		Add 11 clocks for each unaccessed descriptor load.
Int/Trap Gate, diff. level		95		
Task Gate:				
VM/486/286 to 486 TSS		223		
VM/486/286 to 286 TSS		204		
VM/486/286 to VM TSS		201		
Virtual Mode:				
Int/Trap Gate, diff. level		106		
Task Gate:				
VM/486/286 to 486 TSS		223		
VM/486/286 to 286 TSS		204		
VM/486/286 to VM TSS		201		
INTO = Interrupt 4 if OF=1	11001110			
Taken:				
Real Mode		28		
Protected Mode:				
Int/Trap Gate, same level		46		Add 11 clocks for each unaccessed descriptor load.
Int/Trap Gate, diff. level		73		
Task Gate:				
VM/486/286 to 486 TSS		201		
VM/486/286 to 286 TSS		182		
VM/486/286 to VM TSS		179		
Virtual Mode:				
Int/Trap Gate, diff. level		84		
Task Gate:				
VM/486/286 to 486 TSS		201		
VM/486/286 to 286 TSS		182		
VM/486/286 to VM TSS		179		
Not taken		3		
INVD = Invalidate Cache	00001111 00001000	4		
INVLPG = Invalidate TLB Entry	00001111 00000001 mod 111 r/m			
Hit		12		
No hit		11		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
IRET/IRETD = Interrupt Return	11001111					
Real Mode/Virtual Mode				15	8	
Protected Mode:						Add 11 clocks for each unaccessed descriptor load.
To same level				20	11	
To outer level				36	19	
To nested task (NT=1):						
VM/486/286 TSS to 486 TSS				194	59	
VM/486/286 TSS to 286 TSS				175	35	
VM/486/286 TSS to VM TSS				172	41	
JA = Jump if Above						
8-bit displacement	01110111	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000111	full displacement			
Jump taken				3		
Jump not taken				1		
JAE = Jump if Above/Equal						
8-bit displacement	01110011	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000011	full displacement			
Jump taken				3		
Jump not taken				1		
JB = Jump if Below						
8-bit displacement	01110010	8-bit displacement				Add 11 clocks for each unaccessed descriptor load.
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000010	full displacement			
Jump taken				3		
Jump not taken				1		
JBE = Jump if Below/Equal						
8-bit displacement	01110110	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000110	full displacement			
Jump taken				3		
Jump not taken				1		
JC = Jump if Carry						
8-bit displacement	01110010	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000010	full displacement			
Jump taken				3		
Jump not taken				1		
JCXZ = Jump Short if CX=0						
8-bit displacement	11100011	8-bit displacement				
Jump taken				8	See factor 6, p. E-1	
Jump not taken				5		
JE = Jump Short if Equal						
8-bit displacement	01110100	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000100	full displacement			
Jump taken				3		
Jump not taken				1		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
JECXZ = Jump Short if ECX=0						
8-bit displacement	11100011	8-bit displacement				
Jump taken				8	See factor 6, p. E-1	
Jump not taken				5		
JG = Jump if Greater						
8-bit displacement	01111111	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001111	full displacement			
Jump taken				3		
Jump not taken				1		
JGE = Jump if Greater/Equal						
8-bit displacement	01111101	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001101	full displacement			
Jump taken				3		
Jump not taken				1		
JL = Jump if Less						
8-bit displacement	01111100	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001100	full displacement			
Jump taken				3		
Jump not taken				1		
JLE = Jump if Less/Equal						
8-bit displacement	01111110	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001110	full displacement			
Jump taken				3		
Jump not taken				1		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
JMP = Jump						
within segment						Assumes mem. rd, stack push/ pop, and branch in diff. cache sets.
Short	11101011	8-bit displacement		3	See factor 6, p. E-1	
Direct	11101001	full displacement		3		
Register indirect	11111111	11 100 reg		5		
Memory indirect	11111111	mod 100 r/m		5	5	
direct intersegment	11101010	unsigned full offset, selector		17	2	Real Mode; assumes mem. rd, stack push/ pop, and branch in diff. cache sets; clocks include 1 for displacement+ immediate
to same level				19	3	Add 11 clocks for each unaccessed descriptor load.
thru Call Gate to same level				32	6	
thru TSS						
VM/486/286 to 486 TSS				204	3	
VM/486/286 to 286 TSS				185	3	
VM/486/286 to VM TSS				182	3	
thru Task Gate						
VM/486/286 to 486 TSS				205	3	
VM/486/286 to 286 TSS				186	3	
VM/486/286 to VM TSS				183	3	
indirect intersegment	11111111	mod 101 r/m		13	9	Real Mode; assumes mem. rd, stack push/ pop, and branch in diff. cache sets; add 11 clocks for each unaccessed descriptor load.
to same level				18	10	Add 11 clocks for each unaccessed descriptor load.
thru Call Gate to same level				31	13	
thru TSS						
VM/486/286 to 486 TSS				203	10	
VM/486/286 to 286 TSS				184	10	
VM/486/286 to VM TSS				181	10	
thru Task Gate						
VM/486/286 to 486 TSS				204	10	
VM/486/286 to 286 TSS				185	10	
VM/486/286 to VM TSS				182	10	
JNA = Jump if Not Above						
8-bit displacement	01110110	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000110	full displacement			
Jump taken				3		
Jump not taken				1		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
JNAE = Jump if Not Above/ Equal						
8-bit displacement	01110010	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10000010	full displacement			
Jump taken				3		
Jump not taken				1		
JNB = Jump if Not Below						
8-bit displacement	01110011	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10000011	full displacement			
Jump taken				3		
Jump not taken				1		
JNBE = Jump if Not Below/ Equal						
8-bit displacement	01110111	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10000111	full displacement			
Jump taken				3		
Jump not taken				1		
JNC = Jump if Not Carry						
8-bit displacement	01110011	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10000011	full displacement			
Jump taken				3		
Jump not taken				1		
JNE = Jump if Not Equal						
8-bit displacement	01110101	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10000101	full displacement			
Jump taken				3		
Jump not taken				1		
JNG = Jump if Not Greater						
8-bit displacement	01111110	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10001110	full displacement			
Jump taken				3		
Jump not taken				1		
JNGE = Jump if Not Greater/						
Equal	01111100	8-bit displacement				
8-bit displacement				3	See factor 6,	
Jump taken				1	p. E-1	
Jump not taken	00001111	10001100	full displacement			
Full displacement				3		
Jump taken				1		
Jump not taken						

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
JNL = Jump if Not Less						
8-bit displacement	01111101	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001101	full displacement			
Jump taken				3		
Jump not taken				1		
JNLE = Jump if Not Less/Equal						
8-bit displacement	01111111	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001111	full displacement			
Jump taken				3		
Jump not taken				1		
JNO = Jump if Not Overflow						
8-bit displacement	01110001	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000001	full displacement			
Jump taken				3		
Jump not taken				1		
JNP = Jump if Not Parity						
8-bit displacement	01111011	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001011	full displacement			
Jump taken				3		
Jump not taken				1		
JNS = Jump if Not Sign						
8-bit displacement	01111001	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10001001	full displacement			
Jump taken				3		
Jump not taken				1		
JNZ = Jump if Not Zero						
8-bit displacement	01110111	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000111	full displacement			
Jump taken				3		
Jump not taken				1		
JO = Jump if Overflow						
8-bit displacement	01110101	8-bit displacement				
Jump taken				3	See factor 6, p. E-1	
Jump not taken				1		
Full displacement	00001111	10000101	full displacement			
Jump taken				3		
Jump not taken				1		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
JP = Jump if Parity						
8-bit displacement	01111010	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10001010	full displacement			
Jump taken				3		
Jump not taken				1		
JPE = Jump if Parity Even						
8-bit displacement	01111010	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10001010	full displacement			
Jump taken				3		
Jump not taken				1		
JPO = Jump if Parity Odd						
8-bit displacement	01111011	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10001011	full displacement			
Jump taken				3		
Jump not taken				1		
JS = Jump if Sign						
8-bit displacement	01111000	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10001000	full displacement			
Jump taken				3		
Jump not taken				1		
JZ = Jump if Zero						
8-bit displacement	01110100	8-bit displacement				
Jump taken				3	See factor 6,	
Jump not taken				1	p. E-1	
Full displacement	00001111	10000100	full displacement			
Jump taken				3		
Jump not taken				1		
LAHF = Load Flags into AH	1001 1111			3		
LAR = Load Access Rights Byte						
From Register	00001111	00000010	11 reg1 reg2	11	3	
From Memory	00001111	00000010	mod reg r/m	11	5	
LDS = Load Pointer Using DS	11000101	mod reg r/m				Add 11 clocks
Real and Virtual Mode				6	7	for each
Protected Mode				12	10	unaccessed
						descriptor load.
LEA = Load EA to Register	10001101	mod reg r/m				
no index register				1		
with index register				2		
LEAVE = Leave Procedure	11001001			5	1	
LES = Load Pointer Using ES	11000100	mod reg r/m				Add 11 clocks
				6	7	for each
				12	10	unaccessed
						descriptor load.

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
LFS = Load Pointer Using FS	00001111	10110100	mod reg r/m			Add 11 clocks
				6	7	for each
				12	10	unaccessed
						descriptor load.
LGDT = Load Global Descriptor						
Table Register	00001111	00000001	mod 010 r/m	12	5	
LGS = Load Pointer Using GS	00001111	10110101	mod reg r/m			Add 11 clocks
				6	7	for each
				12	10	unaccessed
						descriptor load.
LIDT = Load Interrupt Descriptor						
Table Register	00001111	00000001	mod 011 r/m	12	5	
LLDT = Load Local Descriptor						
Table Register from Register	00001111	00000000	11 010 reg	11	3	
Table Register from Memory	00001111	00000000	mod 010 r/m	11	6	
LMSW = Load Machine Status Word						
From Register	00001111	00000001	11 110 reg	13		
From Memory	00001111	00000001	mod 110 r/m	13	1	
LOCK = Assert $\overline{\text{LOCK}}$ Signal	11110000			1		Prefix
LODS = Load String	1010110w			5	2	
LODSB = Load String Byte						
LODSD = Load String Dword						
LODSW = Load String Word						
LOOP = Loop CX times	11100010	8-bit displacement				
Loop				7	See factor 6,	
No loop				6	p. E-1.	
LOOPE = Loop if Equal	11100001	8-bit displacement				
Loop				9	See factor 6,	
No loop				6	p. E-1.	
LOOPNE = Loop if Not Equal	11100000	8-bit displacement				
Loop				9	See factor 6,	
No loop				6	p. E-1.	
LOOPNZ = Loop if Not Zero	11100000	8-bit displacement				
Loop				9	See factor 6,	
No loop				6	p. E-1.	
LOOPZ = Loop if Zero	11100001	8-bit displacement				
Loop				9	See factor 6,	
No loop				6	p. E-1.	
LSL = Load Segment Limit						
From Register	00001111	00000011	11 reg1 reg2	10	3	
From Memory	00001111	00000011	mod reg r/m	10	6	
LSS = Load Pointer using SS	00001111	10110010	mod reg r/m			Add 11 clocks
				6	7	for each
				12	10	unaccessed
						descriptor load.
LTR = Load Task Register						
From Register	00001111	00000000	11 001 reg	20		
From Memory	00001111	00000000	mod 001 r/m	20		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
MOV = Move						
reg1 to reg2	1000100W	11 reg1 reg2		1		
reg2 to reg1	1000101W	11 reg1 reg2		1		
memory to reg	1000101w	mod reg r/m		1	2	
reg to memory	1000100w	mod reg r/m		1		
immediate to reg	1100011w	11000 reg	immediate data	1		
or	1011w reg	immediate data		1		
Immediate to Memory	1100011w	mod 000 r/m	displacement immediate	1		
Memory to Accumulator	1010000w	full displacement		1	2	
Accumulator to Memory	1010001w	full displacement		1		
reg to segment reg	10001110	11 sreg3 reg				
Real and Virtual Mode				3	0	Add 11 clocks for each unaccessed descriptor load.
Protected Mode				9	3	
memory to segment reg	10001110	mod sreg3 r/m				
Real and Virtual Mode				3	2	
Protected Mode				9	5	
segment reg to reg	10001100	11 sreg3 reg		3		For sreg3:
segment reg to memory	10001100	mod sreg3 r/m		3		CS = 001
CR0 from Register	00001111	00100010	11 000 reg	17	2	DS = 011
CR2 from Register	00001111	00100010	11 010 reg	4		ES = 000
CR3 from Register	00001111	00100010	11 011 reg	4		FS = 100
Register from CR0	00001111	00100000	11 000 reg	4		GS = 101
Register from CR2	00001111	00100000	11 010 reg	4		SS = 010
Register from CR3	00001111	00100000	11 011 reg	4		
DR0 from Register	00001111	00100011	11 000 reg	10		
DR1 from Register	00001111	00100011	11 001 reg	10		
DR2 from Register	00001111	00100011	11 010 reg	10		
DR3 from Register	00001111	00100011	11 011 reg	10		
DR6 from Register	00001111	00100011	11 110 reg	10		
DR7 from Register	00001111	00100011	11 111 reg	10		
Register from DR0	00001111	00100001	11 000 reg	9		
Register from DR1	00001111	00100001	11 001 reg	9		
Register from DR2	00001111	00100001	11 010 reg	9		
Register from DR3	00001111	00100001	11 011 reg	9		
Register from DR6	00001111	00100001	11 110 reg	9		
Register from DR7	00001111	00100001	11 111 reg	9		
TR3 from Register	00001111	00100110	11 011 reg	4		
TR4 from Register	00001111	00100110	11 100 reg	4		
TR5 from Register	00001111	00100110	11 101 reg	4		
TR6 from Register	00001111	00100110	11 110 reg	4		
TR7 from Register	00001111	00100110	11 111 reg	4		
Register from TR3	00001111	00100100	11 011 reg	3		
Register from TR4	00001111	00100100	11 100 reg	4		
Register from TR5	00001111	00100100	11 101 reg	4		
Register from TR6	00001111	00100100	11 110 reg	4		
Register from TR7	00001111	00100100	11 111 reg	4		
MOVS = Move String to String	1010010w			7	2	Assumes the two string addresses fall in different cache sets.
MOVSB = Move Byte to Byte						
MOVSD = Move Dword to Dword						
MOVSW = Move Word to Word						
MOVSX = Move with Sign Extension						
reg2 to reg1	00001111	1011111w	11 reg1 reg2	3		
memory to reg	00001111	1011111w	mod reg r/m	3	2	
MOVZX = Move with Zero Extension						
reg2 to reg1	00001111	1011011w	11 reg1 reg2	3		
memory to reg	00001111	1011011w	mod reg r/m	3	2	

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
MUL = Multiply (unsigned)						
accumulator with register	1111011w	11 100 reg				
Multiplier:						
Byte				13 to 18		For all cases,
Word				13 to 26		clocks = 10 +
Dword				13 to 42		max(log$_2$(|m|),n)
accumulator with memory	11 100 reg	mod 100 r/m				where
Multiplier:						m = multiplier
Byte				13 to 18	1	n = 3/5 for ± m
Word				13 to 26	1	if m = 0,
Dword				13 to 42	1	clocks = 13
NEG = Negate						
reg	1111011w	11 011 reg		1		
memory	1111011w	mod 011 r/m		3	6/2	No LOCK/LOCK
NOP = No Operation	10010000			1		
NOT = Logical Complement						
reg	1111011w	11 010 reg		1		
memory	1111011w	mod 010 r/m		3	6/2	No LOCK/LOCK
Operand Size	01100110			1		Prefix
OR = Logical OR						
reg1 to reg2	0000100w	11 reg1 reg2		1		
reg2 to reg1	0000101w	11 reg1 reg2		1		
memory to register	0000101w	mod reg r/m		2	2	
register to memory	0000100w	mod reg r/m		3	6/2	No LOCK/LOCK
immediate to register	100000sw	11 001 reg	immediate register	1		
immediate to accumulator	0000110w	immediate data		1		
immediate to memory	100000sw	mod 001 r/m	immediate data	3	6/2	No LOCK/LOCK
OUT = Output to Port						
Fixed Port	1110011w	port number				
Real Mode				16		
Protected Mode:						
CPL ≤ IOPL				11		
CPL > IOPL				31		
Virtual Mode				29		
Variable Port	1110111w					
Real Mode				16		
Protected Mode:						
CPL ≤ IOPL				10		
CPL > IOPL				30		
Virtual Mode				29		
OUTS = Output String to Port	0110111w					
OUTSB = Output Byte to Port						
OUTSD = Output Dword to Port						
OUTSW = Output Word to Port						
Real Mode				17	2	
Protected Mode:						
CPL ≤ IOPL				10	2	
CPL > IOPL				32	2	
Virtual Mode				30	2	

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT		Clocks if Cache Hit	Penalty if Cache Miss	Notes
POP = Pop					
reg	10001111	11 000 reg	4	1	
or	01011 reg		1	2	
memory	10001111	mod 000 r/m	5	2	
segment registers:					
CS	000 01 111				
Real or Virtual Mode			3	2	Add 11 clocks for each unaccessed descriptor load.
Protected Mode			9	5	
DS	000 11 111				
Real or Virtual Mode			3	2	
Protected Mode			9	5	
ES	000 00 111				
Real or Virtual Mode			3	2	
Protected Mode			9	5	
FS	00001111	10 100 001			
Real or Virtual Mode			3	2	
Protected Mode			9	5	
GS	00001111	10 101 001			
Real or Virtual Mode			3	2	
Protected Mode			9	5	
SS	000 10 111				
Real or Virtual Mode			3	2	
Protected Mode			9	5	
POPA = Pop All (16-bit)	01100001		9	7/15	16/32
POPAD = Pop All (32-bit)					
POPF = Pop into FLAGS	10011101				
Virtual and Real Mode			9		
Protected Mode			6		
POPFD = Pop into EFLAGS					
PUSH = Push					
reg	11111111	11 110 reg	4		
or	01010 reg		1		
memory	11111111	mod 110 r/m	4	1	Assumes operand and stack addresses are in different cache sets.
immediate	011010s0	immediate data	1		
segment registers:					
CS	000 01 110		3		
DS	000 11 110		3		
ES	000 00 110		3		
FS	00001111	10 100 000	3		
GS	00001111	10 101 000	3		
SS	000 10 110		3		
PUSHA = Push All (16-bit)	01100000		11		
PUSHAD = Push All (32-bit)					
PUSHF = Push FLAGS	10011100				
Real and Virtual Mode			4		
Protected Mode			3		
PUSHFD = Push EFLAGS					

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
RCL = Rotate thru Carry Left						
reg by 1	1101000w	11 010 reg		3		
memory by 1	1101000w	mod 010 r/m		4	6	
reg by CL	1101001w	11 010 reg		8 to 30		if CL≤op. length, clocks = 8(mem) or 9 (reg) else clocks = 9 + (CL/op. lngth)*7
memory by CL	1101001w	mod 010 r/m		9 to 31		
reg by immediate count	1100000w	11 010 reg	immediate 8-bit data	8 to 30		
mem by immediate count	1100000w	mod 010 r/m	immediate 8-bit data	9 to 31		
RCR = Rotate thru Carry Right						
reg by 1	1101000w	11 011 reg		3		
memory by 1	1101000w	mod 011 r/m		4	6	
reg by CL	1101001w	11 011 reg		8 to 30		if CL≤op. length, clocks = 8(mem) or 9 (reg) else clocks = 9 + (CL/op. lngth)*7
memory by CL	1101001w	mod 011 r/m		9 to 31		
reg by immediate count	1100000w	11 011 reg	immediate 8-bit data	8 to 30		
mem by immediate count	1100000w	mod 011 r/m	immediate 8-bit data	9 to 31		
REP = Repeat String Instruction				c = (E)CX count		
REP LODS = Load String	11110010	1010110w				
c = 0				5		
c > 0				7 + 4c	6 per 16 bytes on first load	Assumes string addresses in diff. cache sets.
REP INS = Input String	11110010	0110110w				
Real Mode				16 + 8c		
Protected Mode:						
CPL ≤ IOPL				10 + 8c		
CPL > IOPL				30 + 8c		
Virtual Mode				29 + 8 c		
REP MOVS = Load String	11110010	1010010w				
c = 0				5		
c = 1				13	1	Assumes string addresses in diff. cache sets.
c > 1				12 + 3c	4 per 16 bytes; 1 on first move and 3 on second	Assumes string addresses in diff. cache sets.
REP OUTS = Output String	11110010	0110111w				For all REP OUTS, the entire penalty is on the second operation.
Real Mode				17 + 5c	2 per 16 bytes	
Protected Mode:						
CPL ≤ IOPL				11 + 5c	2 per 16 bytes	
CPL > IOPL				31 + 5c	2 per 16 bytes	
Virtual Mode				30 + 5c	2 per 16 bytes	
REP STOS = Load String	11110010	1010101w				
c = 0				5		
c > 0				7 + 4c		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT		Clocks if Cache Hit	Penalty if Cache Miss	Notes
REPE = Repeat if Equal			c = (E)CX count		
REPE CMPS = Compare String	11110011	1010011w			
c = 0			5		
c > 0			7 + 7c	6 per 16 bytes; all on first compare	Assumes string addresses fall in different cache sets
REPE SCAS = Scan String	11110011	1010111w			
c = 0			5		
c > 0			7 + 5c	4 per 16 bytes; 2 on first and 2 on second compare	
REPNE = Repeat if Not Equal			c = (E)CX count		
REPNE CMPS = Comp. String	11110010	1010011w			
c = 0			5		
c > 0			7 + 7c	6 per 16 bytes; all on first compare	Assumes string addresses fall in different cache sets
REPNE SCAS = Scan String	11110010	1010111w			
c = 0			5		
c > 0			7 + 5c	4 per 16 bytes; 2 on first and 2 on second compare	
REPNZ = Repeat if Not Zero			c = (E)CX count		
REPNZ CMPS = Comp. String	11110010	1010011w			
c = 0			5		
c > 0			7 + 7c	6 per 16 bytes; all on first compare	Assumes string addresses fall in different cache sets
REPNZ SCAS = Scan String	11110010	1010111w			
c = 0			5		
c > 0			7 + 5c	4 per 16 bytes; 2 on first and 2 on second compare	
REPZ = Repeat if Zero			c = (E)CX count		
REPZ CMPS = Compare String	11110011	1010011w			
c = 0			5		
c > 0			7 + 7c	6 per 16 bytes; all on first compare	Assumes string addresses fall in different cache sets
REPZ SCAS = Scan String	11110011	1010111w			
c = 0			5		
c > 0			7 + 5c	4 per 16 bytes; 2 on first and 2 on second compare	

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
RET = Return						
within segment	11000011			5	5	
Adding imm. to SP	11000010	16-bit displacement		5	5	
intersegment	11001011			13	8	Real Mode; assumes mem. rd, stack push/ pop, and branch in diff. cache sets.
to same level				17	9	Protected Mode; add 11 clocks per unaccessed descripter load.
to outer level				35	12	
intersegment add imm. to SP	11001010	16-bit displacement		14	8	Real Mode; assumes mem. rd, stack push/ pop, and branch in diff. cache sets.
to same level				18	9	Protected Mode; add 11 clocks per unaccessed descripter load.
to outer level				36	12	
ROL = Rotate Left						
reg by 1	1101000w	11 000 reg		3		
memory by 1	1101000w	mod 000 r/m		4	6	
reg by CL	1101001w	11 000 reg		3		
memory by CL	1101001w	mod 000 r/m		4	6	
reg by immediate count	1100000w	11 000 reg	immediate 8-bit data	2		
mem by immediate count	1100000w	mod 000 r/m	immediate 8-bit data	4	6	
ROR = Rotate Right						
reg by 1	1101000w	11 001 reg		3		
memory by 1	1101000w	mod 001 r/m		4	6	
reg by CL	1101001w	11 001 reg		3		
memory by CL	1101001w	mod 001 r/m		4	6	
reg by immediate count	1100000w	11 001 reg	immediate 8-bit data	2		
mem by immediate count	1100000w	mod 001 r/m	immediate 8-bit data	4	6	
SAHF = Store AH into Flags	10011110			2		
SAL = Shift Arithmetic Left						
reg by 1	1101000w	11 100 reg		3		
memory by 1	1101000w	mod 100 r/m		4	6	
reg by CL	1101001w	11 100 reg		3		
memory by CL	1101001w	mod 100 r/m		4	6	
reg by immediate count	1100000w	11 100 reg	immediate 8-bit data	2		
mem by immediate count	1100000w	mod 100 r/m	immediate 8-bit data	4	6	
SAR = Shift Arithmetic Right						
reg by 1	1101000w	11 111 reg		3		
memory by 1	1101000w	mod 111 r/m		4	6	
reg by CL	1101001w	11 111 reg		3		
memory by CL	1101001w	mod 111 r/m		4	6	
reg by immediate count	1100000w	11 111 reg	immediate 8-bit data	2		
mem by immediate count	1100000w	mod 111 r/m	immediate 8-bit data	4	6	

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
SBB = Subtract with Borrow						
reg1 to reg2	0001100w	11 reg1 reg2		1		
reg2 to reg1	0001101w	11 reg1 reg2		1		
memory to register	0001101w	mod reg r/m		2	2	
register to memory	0001100w	mod reg r/m		3	6/2	No LOCK/LOCK
immediate to register	100000sw	11 011 reg	immediate register	1		
immediate to accumulator	0001110w	immediate data		1		
immediate to memory	100000sw	mod 011 r/m	immediate data	3	6/2	No LOCK/LOCK
SCAS = Scan String	1010111w			6	2	
SCASB = Scan Byte						
SCASD = Scan Dword						
SCASW = Scan Word						
Segment Override						Prefix
CS	00101110			1		
DS	00111110			1		
ES	00100110			1		
FS	01100100			1		
GS	01100101			1		
SS	00110110			1		
SETA = Set Byte if Above						
Register	00001111	10010111	11 000 reg			
True				4		
False				3		
Memory	00001111	10010111	mod 000 r/m			
True				3		
False				4		
SETAE = Set Byte if Above or Equal						
Register	00001111	10010011	11 000 reg			
True				4		
False				3		
Memory	00001111	10010011	mod 000 r/m			
True				3		
False				4		
SETB = Set Byte if Below						
Register	00001111	10010010	11 000 reg			
True				4		
False				3		
Memory	00001111	10010010	mod 000 r/m			
True				3		
False				4		
SETBE = Set Byte if Below or Equal						
Register	00001111	10010110	11 000 reg			
True				4		
False				3		
Memory	00001111	10010110	mod 000 r/m			
True				3		
False				4		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
SETC = Set Byte if Carry						
Register	00001111	10010010	11 000 reg			
True				4		
False				3		
Memory	00001111	10010010	mod 000 r/m			
True				3		
False				4		
SETE = Set Byte Short if Equal						
Register	00001111	10010100	11 000 reg			
True				4		
False				3		
Memory	00001111	10010100	mod 000 r/m			
True				3		
False				4		
SETG = Set Byte if Greater						
Register	00001111	10011111	11 000 reg			
True				4		
False				3		
Memory	00001111	10011111	mod 000 r/m			
True				3		
False				4		
SETGE = Set Byte if Greater or Equal						
Register	00001111	10011101	11 000 reg			
True				4		
False				3		
Memory	00001111	10011101	mod 000 r/m			
True				3		
False				4		
SETL = Set Byte if Less						
Register	00001111	10011100	11 000 reg			
True				4		
False				3		
Memory	00001111	10011100	mod 000 r/m			
True				3		
False				4		
SETLE = Set Byte if Less or Equal						
Register	00001111	10011110	11 000 reg			
True				4		
False				3		
Memory	00001111	10011110	mod 000 r/m			
True				3		
False				4		
SETNA = Set Byte if Not Above						
Register	00001111	10010110	11 000 reg			
True				4		
False				3		
Memory	00001111	10010110	mod 000 r/m			
True				3		
False				4		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
SETNAE = Set Byte if Not Above or Equal						
Register	00001111	10010010	11 000 reg			
True				4		
False				3		
Memory	00001111	10010010	mod 000 r/m			
True				3		
False				4		
SETNB = Set Byte if Not Below						
Register	00001111	10010011	11 000 reg			
True				4		
False				3		
Memory	00001111	10010011	mod 000 r/m			
True				3		
False				4		
SETNBE = Set Byte if Not Below or Equal						
Register	00001111	10010111	11 000 reg			
True				4		
False				3		
Memory	00001111	10010111	mod 000 r/m			
True				3		
False				4		
SETNC = Set Byte if Not Carry						
Register	00001111	10010011	11 000 reg			
True				4		
False				3		
Memory	00001111	10010011	mod 000 r/m			
True				3		
False				4		
SETNE = Set Byte if Not Equal						
Register	00001111	10010101	11 000 reg			
True				4		
False				3		
Memory	00001111	10010101	mod 000 r/m			
True				3		
False				4		
SETNG = Set Byte if Not Greater						
Register	00001111	10011110	11 000 reg			
True				4		
False				3		
Memory	00001111	10011110	mod 000 r/m			
True				3		
False				4		
SETNGE = Set Byte if Not Greater or Equal						
Register	00001111	10011100	11 000 reg			
True				4		
False				3		
Memory	00001111	10011100	mod 000 r/m			
True				3		
False				4		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
SETNL = Set Byte if Not Less						
Register	00001111	10011101	11 000 reg			
True				4		
False				3		
Memory	00001111	10011101	mod 000 r/m			
True				3		
False				4		
SETNLE = Set Byte if Not Less/ Equal						
Register	00001111	10011111	11 000 reg			
True				4		
False				3		
Memory	00001111	10011111	mod 000 r/m			
True				3		
False				4		
SETNO = Set Byte if Not Overflow						
Register	00001111	10010001	11 000 reg			
True				4		
False				3		
Memory	00001111	10010001	mod 000 r/m			
True				3		
False				4		
SETNP = Set Byte if Not Parity						
Register	00001111	10011011	11 000 reg			
True				4		
False				3		
Memory	00001111	10011011	mod 000 r/m			
True				3		
False				4		
SETNS = Set Byte if Not Sign						
Register	00001111	10011001	11 000 reg			
True				4		
False				3		
Memory	00001111	10011001	mod 000 r/m			
True				3		
False				4		
SETNZ = Set Byte if Not Zero						
Register	00001111	10010101	11 000 reg			
True				4		
False				3		
Memory	00001111	10010101	mod 000 r/m			
True				3		
False				4		
SETO = Set Byte if Overflow						
Register	00001111	10010000	11 000 reg			
True				4		
False				3		
Memory	00001111	10010000	mod 000 r/m			
True				3		
False				4		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
SETP = Set Byte if Parity						
Register	00001111	10011010	11 000 reg			
True				4		
False				3		
Memory	00001111	10011010	mod 000 r/m			
True				3		
False				4		
SETPE = Set Byte if Parity Even						
Register	00001111	10011010	11 000 reg			
True				4		
False				3		
Memory	00001111	10011010	mod 000 r/m			
True				3		
False				4		
SETPO = Set Byte if Parity Odd						
Register	00001111	10011011	11 000 reg			
True				4		
False				3		
Memory	00001111	10011011	mod 000 r/m			
True				3		
False				4		
SETS = Set Byte if Sign						
Register	00001111	10011000	11 000 reg			
True				4		
False				3		
Memory	00001111	10011000	mod 000 r/m			
True				3		
False				4		
SETZ = Set Byte if Zero						
Register	00001111	10010100	11 000 reg			
True				4		
False				3		
Memory	00001111	10010100	mod 000 r/m			
True				3		
False				4		
SGDT = Store Global Descriptor						
Table Register	00001111	00000001	mod 000 r/m	10		
SHL = Shift Logical Left						
reg by 1	1101000w	11 100 reg		3		
memory by 1	1101000w	mod 100 r/m		4	6	
reg by CL	1101001w	11 100 reg		3		
memory by CL	1101001w	mod 100 r/m		4	6	
reg by immediate count	1100000w	11 100 reg	immediate 8-bit data	2		
mem by immediate count	1100000w	mod 100 r/m	immediate 8-bit data	4	6	
SHLD = Shift Left Double Precision						
reg by immediate count	00001111	10100100	11 reg2 reg1 imm. 8-bit	2		
mem by immediate count	00001111	10100100	mod reg r/m imm. 8-bit	3	6	
reg by CL	00001111	10100101	11 reg2 reg1	3		
memory by CL	00001111	10100101	mod reg r/m	4	5	

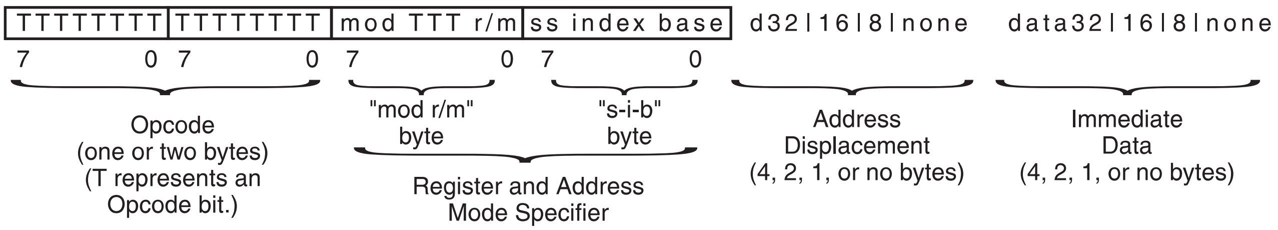
Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
SHR = Shift Logical Right						
reg by 1	1101000w	11 101 reg		3		
memory by 1	1101000w	mod 101 r/m		4	6	
reg by CL	1101001w	11 101 reg		3		
memory by CL	1101001w	mod 101 r/m		4	6	
reg by immediate count	1100000w	11 101 reg	immediate 8-bit data	2		
mem by immediate count	1100000w	mod 101 r/m	immediate 8-bit data	4	6	
SHRD = Shift Right Double Precision						
reg by immediate count	00001111	10101100	11 reg2 reg1 imm. 8-bit	2		
mem by immediate count	00001111	10101100	mod reg r/m imm. 8-bit	3	6	
reg by CL	00001111	10101101	11 reg2 reg1	3		
memory by CL	00001111	10101101	mod reg r/m	4	5	
SIDT = Store Interrupt Descriptor						
Table Register	00001111	00000001	mod 001 r/m	10		
SLDT = Store Local Descriptor						
Table Register to register	00001111	00000000	11 000 reg	2		
Table Register to memory	00001111	00000000	mod 000 r/m	3		
SMSW = Store Machine Status Word						
To register	00001111	00000001	11 100 reg	2		
To memory	00001111	00000001	mod 000 r/m	3		
STC = Set Carry Flag	11111001			2		
STD = Set Direction Flag	11111101			2		
STI = Set Interrupt-Enable Flag	11111011			2		
STOS = Store String	1010101w			5		
STOSB = Store String Byte						
STOSD = Store String Dword						
STOSW = Store String Word						
STR = Store Task Register						
To register	00001111	00000000	11 001 reg	2		
To memory	00001111	00000000	mod 001 r/m	3		
SUB = Subtract						
reg1 to reg2	0010100w	11 reg1 reg2		1		
reg2 to reg1	0010101w	11 reg1 reg2		1		
memory to register	0010101w	mod reg r/m		2	2	
register to memory	0010100w	mod reg r/m		3	6/2	No LOCK/LOCK
immediate to register	100000sw	11 101 reg	immediate register	1		
immediate to accumulator	0010110w	immediate data		1		
immediate to memory	100000sw	mod 101 r/m	immediate data	3	6/2	No LOCK/LOCK
TEST = Logical Compare						
reg1 and reg2	1000010w	11 reg1 reg2		1		
memory and register	1000010w	mod reg r/m		2	2	
immediate and register	1111011w	11 000 reg	immediate data	1		
immediate and accumulator	1010100w	immediate data		1		
immediate and memory	1111011w	mod 000 r/m	immediate data	2	2	
VERR = Verify Read						
Register	00001111	00000000	11 100 r/m	11	3	
Memory	00001111	00000000	mod 100 r/m	11	7	
VERW = Verify Write						
Register	00001111	00000000	11 101 reg	11	3	
Memory	00001111	00000000	mod 101 r/m	11	7	
WAIT = Wait	10011011			1 to 3		

Table E-1 Instruction Clock Count Summary (continued)

INSTRUCTION	FORMAT			Clocks if Cache Hit	Penalty if Cache Miss	Notes
WBINVD = Writeback and Invalidate Cache	00001111	00001001		5		
XADD = Exchange and Add						
reg1, reg2	00001111	1100000w	11 reg2 reg1	3		
memory, reg	00001111	1100000w	mod reg r/m	4	6/2	No LOCK/LOCK
XCHG = Exchange						
reg1 with reg1	1000011w	11 reg1 reg2		3		
Accumulator with reg	10010 reg			3		
Memory with reg	1000011w	mod reg r/m		5		
XLAT/XLATB = Table Look-Up Translation	11010111			4	2	
XOR = Logical Exclusive OR						
reg1 to reg2	0011000w	11 reg1 reg2		1		
reg2 to reg1	0011001w	11 reg1 reg2		1		
memory to register	0011001w	mod reg r/m		2	2	
register to memory	0011000w	mod reg r/m		3	6/2	No LOCK/LOCK
immediate to register	100000sw	11 110 reg	immediate register	1		
immediate to accumulator	0011010w	immediate data		1		
immediate to memory	100000sw	mod 110 r/m	immediate data	3	6/2	No LOCK/LOCK

Figure E-1 General Instruction Format


TTTTTTTT
TTTTTTTT
mod TTT r/m
ss index base
d32 | 16 | 8 | none
data32 | 16 | 8 | none
7 0 7 0 7 0 7 0
Opcode (one or two bytes) (T represents an Opcode bit.)
"mod r/m" byte
"s-i-b" byte
Register and Address Mode Specifier
Address Displacement (4, 2, 1, or no bytes)
Immediate Data (4, 2, 1, or no bytes)

E.3 General Instruction Encoding

Figure E-1 shows the general instruction format. All instruction encodings are subsets of this format. Instructions include one or two primary opcode bytes, possibly an address specifier consisting of the “mod r/m” byte and “scale-index-base” byte, a displacement if required, and an immediate data field if required. Within the primary opcode or opcodes, smaller encoding fields can be defined. These fields vary according to the class of operation. The fields define such information as direction of the operation, size of the displacements, register encoding, or sign extension.

Almost all instructions referring to an operand in memory have an addressing mode byte following the primary opcode byte(s). This byte, the mod r/m byte, specifies the address mode to be used. Certain encodings of the mod r/m byte indicate a second addressing byte, the scale-index-base byte, follows the mod r/m byte to fully specify the addressing mode. Addressing modes can include a displacement immediately following the mod r/m byte, or scaled index byte. If a displacement is present, the possible sizes are 8, 16, or 32 bits. If the instruction specifies an immediate operand, the immediate operand follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

Several smaller fields also appear in certain instructions, sometimes within the opcode bytes themselves. Table E-2 is a complete list of all fields appearing in the Am486 microprocessor instruction set. Detailed tables for each field follow this table.

Table E-2 Instruction Fields

Field Name	Description	No. of bits	Ref. Table
w	Specifies whether data is byte or full size word/dword	1	E-3
d	Specifies direction of data operation	1	E-4
s	Specifies whether the immediate field must be sign-extended	1	E-5
reg	Specifies general register	3	E-6
mod r/m	Specifies address mode (effective address can be general register)	2 (mod) or 3 (r/m)	E-7
ss	Specifies scale factor for scaled indexed address mode	2	E-8
index	Specifies General Register to use as Index Register	3	E-9
base	Specifies General Register to use as Base Register	3	E-10

Table E-3 Operand Length Field (w) Definitions

Value (w=)	16-Bit Operations	32-Bit Operations
0	8 bits	8 bits
1	16 bits	32 bits

Table E-4 Direction Field (d) Definitions

Value (d=)	Operation Direction
0	Register/Memory ⟵ Register “reg” = Source operand “mod r/m” or “mod s-i-b” = Destination operand
1	Register ⟵ Register/Memory “reg” = Destination operand “mod r/m” or “mod s-i-b” = Source operand

Table E-5 Sign-Extend Field (s) Definitions

Value (s=)	Effect on Immediate Byte	Effect on Immediate Word/Dword
0	None	None
1	Sign-Extend immediate byte to fill word or dword destination.	None

Table E-6 General Register Field (reg) Definitions

Value (reg=)	General Register Selected					
	16-Bit Data Operations			32-Bit Data Operations		
	No w field	w = 0	w =1	No w field	w =0	w =1
000	AX	AL	AX	EAX	AL	EAX
001	CX	CL	CX	ECX	CL	ECX
010	DX	DL	DX	EDX	DL	EDX
011	BX	BL	BX	EBX	BL	EBX
100	SP	AH	SP	ESP	AH	ESP
101	BP	CH	BP	EBP	CH	EBP
110	SI	DH	SI	ESI	DH	ESI
111	DI	BH	DI	EDI	BH	EDI

Table E-7 Address Mode Field (mod/rm) Definitions (no s-i-b present)

<table>
<tr><th rowspan="2">Value
(mod r/m =)</th><th colspan="4">Effective Address</th></tr>
<tr><th colspan="2">16-Bit Address Mode</th><th colspan="2">32-Bit Address Mode</th></tr>
<tr><td>00 000</td><td colspan="2">DS:[BX + SI]</td><td colspan="2">DS:[EAX]</td></tr>
<tr><td>00 001</td><td colspan="2">DS:[BX + DI]</td><td colspan="2">DS:[ECX]</td></tr>
<tr><td>00 010</td><td colspan="2">SS:[BP + SI]</td><td colspan="2">DS:[EDX]</td></tr>
<tr><td>00 011</td><td colspan="2">SS:[BP + DI]</td><td colspan="2">DS:[EBX]</td></tr>
<tr><td>00 100</td><td colspan="2">DS:[SI]</td><td colspan="2">s-i-b present (see Tables E-8 through E-10)</td></tr>
<tr><td>00 101</td><td colspan="2">DS:[DI]</td><td colspan="2">DS:immediate dword</td></tr>
<tr><td>00 110</td><td colspan="2">DS:immediate word</td><td colspan="2">DS:[ESI]</td></tr>
<tr><td>00 111</td><td colspan="2">DS:[BX]</td><td colspan="2">DS:[EDI]</td></tr>
<tr><td>01 000</td><td colspan="2">DS:[BX + SI + immediate byte]</td><td colspan="2">DS:[EAX + immediate byte]</td></tr>
<tr><td>01 001</td><td colspan="2">DS:[BX + DI + immediate byte]</td><td colspan="2">DS:[ECX + immediate byte]</td></tr>
<tr><td>01 010</td><td colspan="2">SS:[BP + SI + immediate byte]</td><td colspan="2">DS:[EDX + immediate byte]</td></tr>
<tr><td>01 011</td><td colspan="2">SS:[BP + DI + immediate byte]</td><td colspan="2">DS:[EBX + immediate byte]</td></tr>
<tr><td>01 100</td><td colspan="2">DS:[SI + immediate byte]</td><td colspan="2">s-i-b present (see Tables E-8 through E-10)</td></tr>
<tr><td>01 101</td><td colspan="2">DS:[DI + immediate byte]</td><td colspan="2">SS:[EBP + immediate byte]</td></tr>
<tr><td>01 110</td><td colspan="2">SS:[BP + immediate byte]</td><td colspan="2">DS:[ESI + immediate byte]</td></tr>
<tr><td>01 111</td><td colspan="2">DS:[BX + immediate byte]</td><td colspan="2">DS:[EDI + immediate byte]</td></tr>
<tr><td>10 000</td><td colspan="2">DS:[BX + SI + immediate word]</td><td colspan="2">DS:[EAX + immediate dword]</td></tr>
<tr><td>10 001</td><td colspan="2">DS:[BX + DI + immediate word]</td><td colspan="2">DS:[ECX + immediate dword]</td></tr>
<tr><td>10 010</td><td colspan="2">SS:[BP + SI + immediate word]</td><td colspan="2">DS:[EDX + immediate dword]</td></tr>
<tr><td>10 011</td><td colspan="2">SS:[BP + DI + immediate word]</td><td colspan="2">DS:[EBX + immediate dword]</td></tr>
<tr><td>10 100</td><td colspan="2">DS:[SI + immediate word]</td><td colspan="2">s-i-b present (see Tables E-8 through E-10)</td></tr>
<tr><td>10 101</td><td colspan="2">DS:[DI + immediate word]</td><td colspan="2">SS:[EBP + immediate dword]</td></tr>
<tr><td>10 110</td><td colspan="2">SS:[BP + immediate word]</td><td colspan="2">DS:[ESI + immediate dword]</td></tr>
<tr><td>10 111</td><td colspan="2">DS:[BX + immediate word]</td><td colspan="2">DS:[EDI + immediate dword]</td></tr>
<tr><td rowspan="2"><i>The following values specify General Registers</i></td><th colspan="2">16-Bit Data Operations</th><th colspan="2">32-Bit Data Operations</th></tr>
<tr><th>w = 0</th><th>w =1</th><th>w =0</th><th>w =1</th></tr>
<tr><td>11 000</td><td>AL</td><td>AX</td><td>AL</td><td>EAX</td></tr>
<tr><td>11 001</td><td>CL</td><td>CX</td><td>CL</td><td>ECX</td></tr>
<tr><td>11 010</td><td>DL</td><td>DX</td><td>DL</td><td>EDX</td></tr>
<tr><td>11 011</td><td>BL</td><td>BX</td><td>BL</td><td>EBX</td></tr>
<tr><td>11 100</td><td>AH</td><td>SP</td><td>AH</td><td>ESP</td></tr>
<tr><td>11 101</td><td>CH</td><td>BP</td><td>CH</td><td>EBP</td></tr>
<tr><td>11 110</td><td>DH</td><td>SI</td><td>DH</td><td>ESI</td></tr>
<tr><td>11 111</td><td>BH</td><td>DI</td><td>BH</td><td>EDI</td></tr>
</table>

Table E-8 Scale Field (ss) Definitions

Value (ss=)	Scale Factor
00	x1
01	x2
10	x4
11	x8

Table E-9 Index Field (index) Definitions

Value (index=)	Indexed Register
000	EAX
001	ECX
010	EDX
011	EBX
100	no index register
101	EBP
110	ESI
111	EDI

Note: *When index = 100, the ss field must equal 00. If not, the effective address is undefined.*

Table E-10 Base Field (base) Definitions

mod r/m =	Value (base=)	Effective Address
00 100	000	DS:[EAX + (scaled index)]
00 100	001	DS:[ECX + (scaled index)]
00 100	010	DS:[EDX + (scaled index)]
00 100	011	DS:[EBX + (scaled index)]
00 100	100	SS:[ESP + (scaled index)]
00 100	101	DS:[immediate dword + (scaled index)]
00 100	110	DS:[ESI + (scaled index)]
00 100	111	DS:[EDI + (scaled index)]
01 100	000	DS:[EAX + (scaled index) + immediate byte]
01 100	001	DS:[ECX + (scaled index) + immediate byte]
01 100	010	DS:[EDX + (scaled index) + immediate byte]
01 100	011	DS:[EBX + (scaled index) + immediate byte]
01 100	100	SS:[ESP + (scaled index) + immediate byte]
01 100	101	SS:[EBP + (scaled index) + immediate byte]
01 100	110	DS:[ESI + (scaled index) + immediate byte]
01 100	111	DS:[EDI + (scaled index) + immediate byte]
10 100	000	DS:[EAX + (scaled index) + immediate dword]
10 100	001	DS:[ECX + (scaled index) + immediate dword]
10 100	010	DS:[EDX + (scaled index) + immediate dword]

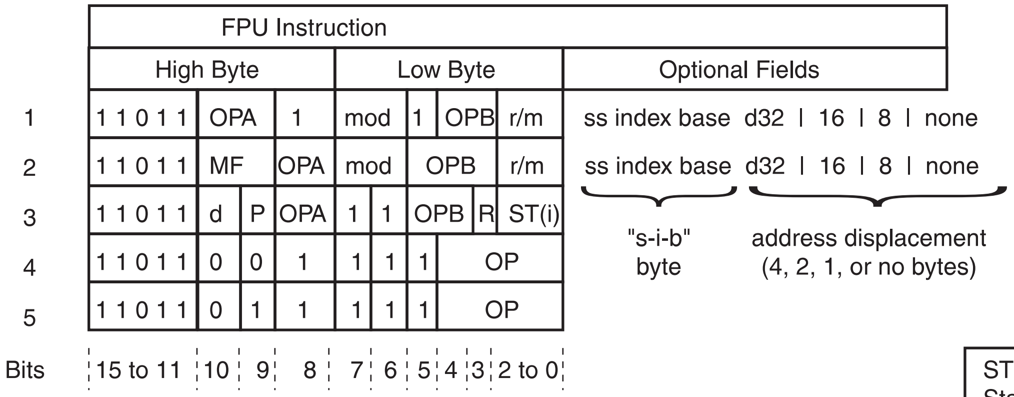
Table E-10 Base Field (base) Definitions (continued)

mod r/m =	Value (base=)	Effective Address
10 100	011	DS:[EBX + (scaled index) + immediate dword]
10 100	100	SS:[ESP + (scaled index) + immediate dword]
10 100	101	SS:[EBP + (scaled index) + immediate dword]
10 100	110	DS:[ESI + (scaled index) + immediate dword]
10 100	111	DS:[EDI + (scaled index) + immediate dword]

E.4 ENCODING OF FLOATING-POINT INSTRUCTION FIELDS

Instructions for the FPU assume one of the five forms shown in Figure E-2. The s-i-b (scale index base) byte and displacement are optionally present in instructions that have mod and r/m fields. Their presence depends on the values of mod and r/m.

Figure E-2 Floating-Point Instruction Format



	FPU Instruction										
	High Byte				Low Byte						Optional Fields
1	1 1 0 1 1	OPA		1	mod		1	OPB		r/m	ss index base d32 | 16 | 8 | none
2	1 1 0 1 1	MF		OPA	mod		OPB			r/m	ss index base d32 | 16 | 8 | none
3	1 1 0 1 1	d	P	OPA	1	1	OPB		R	ST(i)	"s-i-b" byte / address displacement (4, 2, 1, or no bytes)
4	1 1 0 1 1	0	0	1	1	1	1	OP			
5	1 1 0 1 1	0	1	1	1	1	1	OP			
Bits	15 to 11	10	9	8	7	6	5	4	3	2 to 0	

OP = Instruction Opcode OPA/OPB = Opcode split with A and B parts
MF = Memory Format (00 = 32-bit real; 01 = 32-bit int.; 10 = 64-bit real; 11 = 64-bit int.)
d = Destination (0 = Destination is ST(0); 1 = Destination is ST(i))
R XOR d = 0: Destination OP Source; R XOR d = 1: Source OP Destination
P = Pop (0 = Do not pop stack after operation; 1 = Pop stack after operation)
mod, r/m, s-i-b, and displacement fields are address mode encoding parameters
(see Tables E-7 through E-10)

ST(i) = Register Stack Element
000 = Stack Top
001 = Top + 1
010 = Top + 2
011 = Top + 3
100 = Top + 4
101 = Top + 5
110 = Top + 6
111 = Top + 7

APPENDIX

F NUMERIC EXCEPTION SUMMARY

The following table lists the numeric (floating point; real) instruction mnemonics in alphabetical order. For each mnemonic, it summarizes the exceptions that the instruction can generate. When writing numeric programs that may be used in an environment that employs numeric exception handlers, assembly-language programmers should be aware of the possible exceptions generated by each instruction in order to determine the need for exception synchronization.

Table F-1 Exception Summary for Floating-Point Instructions

Mnemonic	Instruction	IS	I	D	Z	O	U	P
F2XM1	2^x–1	Y	Y	Y			Y	Y
FABS	Absolute value	Y						
FADD(P)	Add real (and pop)	Y	Y	Y		Y	Y	Y
FBLD	Load BCD	Y						Y
FBSTP	Store BCD and pop	Y	Y					Y
FCHS	Change sign	Y						
FCLEX	Clear exceptions							
FCOM(P)(P)	Compare real (and pop) (and pop)	Y	Y	Y				
FCOS	Cosine	Y	Y	Y			Y	Y
FDECSTP	Decrement stack top pointer							
FDIV(R)(P)	Divide real (or reverse divide) (and pop)							
FFREE	Free register	Y	Y	Y	Y	Y	Y	Y
FIADD	Add integer	Y	Y	Y	Y	Y	Y	Y
FICOM(P)	Compare integer (and pop)	Y	Y	Y	Y	Y	Y	Y
FIDIV	Divide integer	Y	Y	Y	Y	Y	Y	Y
FIDIVR	Reverse divide integer	Y	Y	Y	Y	Y	Y	Y
FILD	Load integer	Y	Y	Y	Y	Y	Y	Y
FIMUL	Multiply integer	Y	Y	Y	Y	Y	Y	Y
FINCSTP	Increment stack pointer	Y	Y	Y	Y	Y	Y	Y
FINIT	Initialize FPU	Y	Y	Y	Y	Y	Y	Y
FIST(P)	Store integer (and pop)	Y	Y	Y	Y	Y	Y	Y
FISUB(R)	Subtract integer (or reverse subtract)	Y	Y	Y	Y	Y	Y	Y
FLD1	Load constant +1.0	Y	Y	Y	Y	Y	Y	Y
FLD	Load real	Y	Y	Y	Y	Y	Y	Y
FLDCW	Load control word	Y	Y	Y	Y	Y	Y	Y
FLDENV	Load FPU environment	Y	Y	Y	Y	Y	Y	Y
FLDL2E	Load constant $\log_2 e$	Y	Y	Y	Y	Y	Y	Y
FLDL2T	Load constant $\log_2 10$	Y	Y	Y	Y	Y	Y	Y

Table F-1 Exception Summary for Floating-Point Instructions (continued)

Mnemonic	Instruction	IS	I	D	Z	O	U	P
FLDLG2	Load constant $\log_{10}2$	Y	Y	Y	Y	Y	Y	Y
FLDLN2	Load constant $\log_e 2$	Y	Y	Y	Y	Y	Y	Y
FLDPI	Load constant π	Y	Y	Y	Y	Y	Y	Y
FLDZ	Load constant + 0.0	Y	Y	Y	Y	Y	Y	Y
FMUL(P)	Multiply real (and pop)	Y	Y	Y	Y	Y	Y	Y
FNOP	No operation	Y	Y	Y	Y	Y	Y	Y
FPATAN	Partial arctangents	Y	Y	Y	Y	Y	Y	Y
FPREM1	Partial remainder (IEEE 754 compliant)	Y	Y	Y	Y	Y	Y	Y
FPREM	Partial remainder	Y	Y	Y	Y	Y	Y	Y
FPTAN	Partial tangent	Y	Y	Y	Y	Y	Y	Y
FRNDINT	Round to integer	Y	Y	Y	Y	Y	Y	Y
FRSTOR	Restore state	Y	Y	Y	Y	Y	Y	Y
FSAVE	Store state	Y	Y	Y	Y	Y	Y	Y
FSCALE	Scale	Y	Y	Y	Y	Y	Y	Y
FSIN	Sine	Y	Y	Y	Y	Y	Y	Y
FSINCOS	Sine and cosine	Y	Y	Y	Y	Y	Y	Y
FSQRT	Square root	Y	Y	Y	Y	Y	Y	Y
FST(P) stack or extended	Store real (and pop)	Y	Y	Y	Y	Y	Y	Y
FST(P) single or double	Store real (and pop)	Y	Y	Y	Y	Y	Y	Y
FSTCW	Store control word	Y	Y	Y	Y	Y	Y	Y
FSTENV	Store environment	Y	Y	Y	Y	Y	Y	Y
FSTSW	Store status word	Y	Y	Y	Y	Y	Y	Y
FSUB(R)(P)	Subtract real (or reverse subtract) (and pop)	Y	Y	Y	Y	Y	Y	Y
FTST	Test	Y	Y	Y	Y	Y	Y	Y
FUCOM	Unordered compare real	Y	Y	Y	Y	Y	Y	Y
FUCOMP	Unordered compare real and pop	Y	Y	Y	Y	Y	Y	Y
FUCOMP	Unordered compare real and pop twice	Y	Y	Y	Y	Y	Y	Y
FWAIT	Wait	Y	Y	Y	Y	Y	Y	Y
FXAM	Examine stack top	Y	Y	Y	Y	Y	Y	Y
FXCH	Exchange register	Y	Y	Y	Y	Y	Y	Y
FXTRACT	Extract exponent and significand	Y	Y	Y	Y	Y	Y	Y
FYL2X	$Y \cdot \log_2 x$	Y	Y	Y	Y	Y	Y	Y
FYL2XP1	$Y \cdot \log_2(x+1)$	Y	Y	Y	Y	Y	Y	Y

Exception Description:

IS — Invalid operand due to stack overflow/underflow
I — Invalid operand due to other cause
D — Denormal operand
Z — Zero-divide
O — Overflow
U — Underflow
P — Inexact result (precision)

APPENDIX

G CODE OPTIMIZATION

The Am486 processor is binary-compatible with 386 processors. Only three new application-level instructions, useful in special situations, are added. Any existing 8086/8088, 80286, and 386 processor applications can execute on the 486 processor immediately without any modification or recompilation. Any compiler that currently generates code for the 386 processor family can generate code that runs on the 486 processor without modification. There are, however, certain code-optimization techniques that make applications execute faster on the 486 processor. The techniques rely upon instruction sequence selection and instruction scheduling to take advantage of the 486 processor internal pipelined execution units and the on-chip cache.

G.1 ADDRESSING MODES

Like 386 processors, the 486 processor needs an additional clock cycle to generate an effective address when using an index register. Therefore, if you use only one indexing component and no scaling is necessary, it is faster to use a register as the base. For example:

```
MOV EAX, [ESI]; use ESI as base
MOV EAX, [ESI*]; use ESI as index, 1 clock penalty
```

If you use a base and an index, or if you require scale indexing, it is faster to use the combined addressing mode, in spite of the one clock penalty.

When you use a register as the base component, you use an additional clock cycle if the register is the destination of the immediately preceding instruction (assuming all instructions are in the prefetch queue). For best performance, separate the two instructions by at least one other instruction. For example:

```
ADD ESI, EAX; ESI is destination register
MOV EAX, [ESI]; ESI is base, 1 clock penalty
```

There are other hidden or implicit usages of destination and base registers, primarily the stack pointer register ESP. The ESP register is the implicit base of all PUSH/POP/RET instructions and it is the implicit destination for the CALL/ENTER/LEAVE/RET/PUSH/POP instruction. Therefore, a LEAVE instruction followed immediately by a RET instruction will use one additional clock. But if the LEAVE and RET are rearranged so that they are separated by another instruction, then no such penalty is entailed. (See other recommendations regarding the LEAVE instruction.) It is not necessary to separate back-to-back PUSH/POP instructions. The 486 processor allows this sequence without incurring an additional clock. All such instruction rearrangements of the instructions will not affect the performance of 386 processors.

The 486 processor also takes an additional clock to execute an instruction that has both an immediate data field and a memory offset field. For example:

```
MOV dword ptr FOO, 1234h; both immediate and memory offset
MOV dword ptr BAX, 1234h
MOV [EBP–200], 1234h
```

When it is necessary to use constants, it would still be more efficient to use immediate data instead of loading the constant into a register first. But if the same immediate data is used more than once, then it would be faster to load the constant in a register and then use the register multiple times. This optimization will not affect the performance of 386 processors. The following sequence is faster than the one above, if all instructions are in the prefetch queue, and because the instructions are shorter, it will actually make it easier to prefetch:

```
MOV EAX, 1234h
MOV dword ptr FOO, EAX; FOO IS VARIABLE
MOV dword ptr BAZ, EAX; BAZ IS VARIABLE
MOV [EBP–2300], EAX
```

G.2 PREFETCH UNIT

The 486 processor prefetch unit accesses the on-chip cache to fill the prefetch queue whenever the cache is idle, and there is enough room in the queue for another cache line (16 bytes). If the prefetch queue becomes empty, it can take up to three additional clocks to start the next instruction. The prefetch queue is 32 bytes in size (2 cache lines).

Because data accesses always have priority over prefetch requests, keeping the cache busy with data accesses can lock out the prefetch unit.

Therefore, arrange instructions so that the memory bus is not used continuously by a series of memory reference instructions. Arrange the instructions so that there is a non-memory-referencing instruction (such as a register/register instruction) at least two clocks before the prefetch queue becomes full. This allows the prefetch unit to transfer a cache line into the queue. For example:

Instruction	Length
MOV mem, 1234567h	10 bytes
MOV mem, 1234567h	10 bytes
MOV mem, 1234567h	10 bytes
MOV mem, 1234567h	10 bytes
MOV mem, 1234567h	10 bytes
ADD reg, reg	2 bytes

If the prefetch queue started out full, then by the third MOV instruction, there is enough room for another cache line in the queue, but because the memory bus is continuously used, there is no time for the transfer from the cache to the prefetch queue. If you do not insert a non-memory instruction before or after the third MOV instruction, the queue is exhausted by the fourth MOV instruction. In this case, rearrange the instructions so that the ADD instruction is before or after the third MOV instruction. This allows the cache to transfer another instruction line to the prefetch unit.

Note: *Rearranging the instructions has no effect on 386 processor performance.*

G.3 CACHE AND CODE ALIGNMENT

The prefetch unit in a 386 processor fetches four bytes at a time on aligned boundaries; therefore, align the destination of any JUMP/CALL/RET instruction on a 0-mod-4 address to help the prefetch unit fill the prefetch queue as quickly as possible. The 486 processor fetches 16 bytes by using the on-chip cache; therefore, align JUMP/CALL/RET destinations at 0-mod-16 addresses for better performance.

The drawback of the 0-mod-16 alignment is that it causes code to grow bigger, requiring you to balance execution speed and code size. The recommended compromise is to align function entry addresses (that is, CALL destinations) on a 0-mod-16 address, but to align labels (that is, JUMP destinations) on a 0-mod-4 address.

On the 486 processor, it takes up to five additional clocks to start executing an instruction if it splits across two 16-byte cache lines. For example, if a CALL instruction ends at address 0x0000000E and the next instruction is a multiple-byte instruction, then when processing returns from the CALL, the processor takes five additional clocks to fill the prefetch queue if the target instruction is not already in the cache. Even if the instruction is in the cache, the processor requires two clocks to transfer it into the prefetch unit. In this situation, it is faster to insert a filler instruction (either by rearranging the instructions or adding a NOP instruction) so that the multiple-byte instruction starts on an aligned address. This instruction alignment also improves 386 processor performance.

G.4 NOP INSTRUCTIONS

Sometimes programs need fillers between instructions to align them. On 386 and 486 processors, the one-byte NOP instruction (an exchange EAX with EAX) performs this function. You can also use other instructions to provide different length instructions with a single clock, as shown below:

```
INC reg           ; 1 byte — modifies register and flags
MOV reg,reg       ; 2 bytes — true NOP
LEA reg,0[reg]    ; 3 bytes — true NOP, uses 8-bit displacement
MOV EAX,0         ; 5 bytes — modifies eax register
ADD EAX,0         ; 5 bytes — modifies flags
LEA reg,0[EAX]    ; 6 bytes — true NOP, uses 32-bit displacement
```

Many of the 386/486 instructions can perform this function, using several forms and lengths, different-sized immediate data, or different-sized memory offsets. Some of the instructions have shorter forms if the destination register is EAX/AX/AL. The different forms may use different clocks. For example, PUSH/POP instructions use one clock in the 1-byte form, but use four clocks when coded in the 2-byte form.

NOP replacement instructions execute faster than the XCHG instruction on 386 processors. Using different forms of the same instruction does not affect 386 processor performance.

G.5 INTEGER INSTRUCTIONS

Most frequently used 486 processor instructions execute in one clock. However, unlike 386 processors, some memory operations take more clocks than corresponding register instructions. For example, for the PUSH MEM instruction:

Instruction	386 Processor Clocks	486 Processor Clocks
MOV reg,mem	4	1
PUSH reg	2	1
PUSH mem	5	4

For the 486 processor, loading a value from memory into a register and then pushing the register results in a net saving of two clocks. The same sequence in a 386 processor imposes a one-clock penalty. If however, available registers are limited, you may choose to sacrifice efficiency to save reusable data stored in the registers.

Another example of 386 versus 486 differences is shown by the LEAVE instruction:

Instruction	386 Processor Clocks	486 Processor Clocks
MOV ESP,EBP	2	1
POP EBP	4	1 + 1 (esp. penalty)
LEAVE	4	5

For the 486 processor, executing the MOV/POP sequence results in a net saving of two clocks over the LEAVE instruction. On the 386 processor, LEAVE is both faster and shorter. You can increase the efficiency on the 486 processor by separating the MOV and POP instruction by one instruction. Because the MOV instruction uses ESP as the destination register and the POP instruction implicitly uses ESP register as a base, there is an inherent one-clock penalty. Separating the instructions with a useful instruction results in a net savings of three clocks over the LEAVE instruction.

Because the 486 processors access operands from registers faster than from memory, it is important for any compiler to have good register allocation and value tracking optimization capability. However, unlike RISC architecture, there is no advantage to loading every possible value before using it. The processor performs reg,mem type ALU operations just as fast as load/op/store sequences. For example, for the assignment:

```
mem1 = mem1 + mem2
```

you can use the following instruction sequences, which yield varying total clock counts (11 or 12) on 386 processors, but identical total clock counts (4) on a 486 processor:

Instruction	386 Processor Clocks	486 Processor Clocks
MOV EAX,mem1	4	1
MOV EBX,mem2	4	1
ADD EAX,EBX	2	1
MOV mem1, EAX	2	1
MOV EAX,mem1	4	1
ADD EAX,mem2	6	2
MOV mem1,EAX	2	1
MOV EAX,mem1	4	1
ADD mem2,EAX	7	3

The MOVZX instruction is another example in which the 486 processor executes faster using simple instructions if the destination is a byte addressable register. For example:

Instruction	386 Processor Clocks	486 Processor Clocks
MOVZX EAX,mem1	6	3 + 1 (0Fh prefix)
XOR EAX,EAX	2	1
MOVB AL,mem1	4	1

For the 486 processor, clearing the register first and then loading the byte value may result in a net savings of two clocks (depending on whether the prefix decode clock can overlap the previous instruction), though there is no comparable difference on the 386 processor.

G.6 CONDITION CODES

In some high level languages, it is sometimes necessary to convert the result of a boolean condition (e.g., equal, greater than, or less than) to a true-false (0/1) value. The flags registers in 386 and 486 processors normally maintain comparison results. In order to convert a comparison result to a true/false value, you must convert the flag settings to an integer value.

The conditional SET instructions can perform the conversions, but require, on a 486 processor, three to four clocks to execute depending on whether the condition tested is true or false. When comparing unsigned values for greater-than or less-than, there is an optional sequence to use. For example, if “x” and “y” are both unsigned values loaded into registers EAX and ECX, respectively, then you can generate the code for “(x < y)” in several ways:

Instruction	386 Processor Clocks	486 Processor Clocks
CMP EAX,ECX	2	1
MOV EAX,0	2	1
JNB L1	7 + m/3	3/1
MOV EAX,1	2	1
L1:		
CMP EAX,ECX	2	1
SETB AL	4/5	4/3
MOVSX EAX,AL	3	3
CMP EAX,ECX	2	1
SBB EAX,EAX	2	1
NEG EAX	2	1

Using the SBB instruction to capture the flag settings of an unsigned compare gives the fastest performance. Because there are no jumps, it does not break the prefetch pipeline. Although this is specific for the “(x < y)” condition, it is possible to transform other tests to this form by either negating the condition or by exchanging the operands.

These condition code instruction replacements also improve 386 processor performance.

G.7 STRING INSTRUCTIONS

Like a 386 processor, a 486 processor executes string instructions slower than the load/store instructions. For example, the LODS instructions:

Instruction	386 Processor Clocks	486 Processor Clocks
MOV EAX,[ESI]	4	1
ADD ESI,4	2	1
LODS	5	4

The LODS instruction loads the string and updates the ESI register. If the register update is unnecessary, the MOV instructions saves three clocks on 386 and 486 processors. If code length is more important, however, the LODS instruction is shorter than MOV.

In a non-REPeated instruction, individual MOV instructions are always faster than MOVS. Even in a REPeated loop, if the loop is small enough, it is faster to use individual load/store instructions than to set up REPeated MOVS instructions. The tradeoff is speed versus code space. The REP MOVS loop is shorter, but slower.

Another consideration is that a long sequence of load/store instructions prevents the prefetch unit from filling the prefetch queue, which slows the processor. To prevent this, do not move more than 16 bytes using load/store instructions within any sequence. Insert a non-memory instruction to allow the prefetch unit to access the cache.

Similar optimizations can be made for STOS and other string instructions. Such optimizations also improve 386 processor performance.

G.8 FLOATING-POINT INSTRUCTIONS

Like the 386 processor/387 coprocessor combination, the floating-point unit in the 486 processor is a separate independent execution unit that operates in parallel with the integer unit. Any instruction sequence that allows the two independent units to execute in parallel is faster than one that uses sequential processing.

Do not place floating-point instructions in direct sequence. Rearrange instructions so that non-floating-point instructions separate the floating-point instructions to allow both execution units to operate in parallel. Schedule the integer instructions (by clock counts) so that they can execute without causing the floating-point unit to wait for its next instruction. These rearrangements also improve 386/387 processor/coprocessor performance, but the clock counts for 387 operations are much higher than the 486 floating-point unit.

Note: *Use the integer unit and integer instructions for simple floating-point value arrangement or movement. FWAITs are never required around simple floating-point instructions.*

G.9 PREFIX OPCODES

On 386 and 486 processors, all prefix opcodes require an additional clock to decode. You can overlap this clock with the execution of the previous instruction if that instruction takes more than one clock to execute. Because of the decode clock requirement, it is faster to expand 16-bit operands to 32-bit operands instead of using the 66h prefix to operate on 16-bit operands, for example. Another reason for the conversion is that if an instruction with a 16-bit destination is followed by an instruction with a 32-bit operand register, there is another one-clock penalty. If you must use this combination, separate the instructions with another instruction.

If you must use prefix opcodes, try to rearrange the instructions so that the prefixed instruction executes after a multiple-clock instruction.

G.10 OVERLAPPED CLOCKS

As mentioned before, an instruction may require an extra clock to execute. However, some of the clock penalties can overlap. In particular, the following combinations overlap:

- Having an index register and an immediate field with a memory offset field only incurs a one-clock penalty.
- Having a prefix opcode and using the result register of the previous instruction as a base only incurs a one-clock penalty.
- Having a prefix opcode after a multiclock instruction does not incur any clock penalty.

G.11 MISCELLANEOUS GUIDELINES

The 386 processor instruction design considered certain programming practices. Many of these considerations apply to 486 processor programming and are applicable to compiler design as well.

- Use the EAX register when possible. Many instructions are 1 byte shorter when using this register, such as loads and stores to memory with absolute addressing, transfers between registers with XCHG, and operations using immediate operands.
- Use the DS register when possible. Instructions that use the DS register are 1 byte shorter than instructions using the other data segment registers; no data segment prefix is required.
- Use short 1-, 2-, and 3-byte instructions when possible. Because 486 processor instructions begin and end on byte boundaries, many instruction encodings are more compact than those in word-aligned instruction sets. Byte alignment reduces code size and increases execution speed.
- Use MOVSX and MOVZX to access 16-bit data. These instructions sign-extend and zero-extend word operands to doubleword length, eliminating the need for an extra instruction to initialize the high word.

- Use the NMI interrupt when possible for faster interrupt response.
- Instead of ENTER at lexical level 0, use a code sequence like:

```
PUSH EBP
MOV EBP,ESP
SUB ESP,byte_count
```

This executes in seven clock cycles instead of the ten required to execute ENTER.

Optimize systems using the following techniques to enhance system speed after the basic functions are implemented:

- If supported by your assembler and acceptable for your application, use the short form of the JUMP instruction. The short form uses an immediate byte for relative jumps in the range from 128 bytes back to 127 bytes forward. The assembly generates an error if it does not support the function. Some assemblers perform this optimization automatically.
- Use the ESP register to reference the stack in the deepest level of subroutines. Do not set up the EBP register and stack frame.
- For fastest task switching, switch tasks in software; this saves and restores a smaller processor state.
- Use the LEA instruction to add registers. If you use a base register and index register, LEA loads the destination with the sum. You can scale the register contents by 2, 4, or 8.
- Use the LEA instruction to add a constant to a register. If you use a base register and a displacement, LEA loads the destination with their sum. You can use LEA with a base register, index register, scale factor, and displacement.
- Use integer move instructions to transfer floating-point data.
- Use RET in the form that takes an immediate value for byte count, rather than an ADD ESP instruction. It saves one clock cycle and 3 bytes on every subroutine call.
- If you need to make several references to a variable addressed with a displacement, load the displacement into a register.
- For PUSH/POP instructions using an operand in memory, use an equivalent two-instruction sequence to move the operand through a general register before pushing it on the stack. This saves two clock cycles.
- For LOOP instructions, use an equivalent decrement and conditional jump instruction combination. This saves two clock cycles.
- For JECXZ instructions, use an equivalent compare and conditional jump instruction combination. This saves one clock cycle.

APPENDIX

H BIOS DATA AREA MAP

When an IBM-compatible personal computer system initializes, the microprocessor, under the direction of the POST in the BIOS software, creates a BIOS data map at location 000400h. This map is 256 bytes in length (address range from 000400h to 0004FFh). The BIOS software uses this memory space to store data and environmental control variables. Programs can access and change the values stored in this area to change the conditions under which the system operates. The following table identifies the standard contents of the BIOS data area locations:

Table H-1 BIOS Map Contents

Address	BIOS Service	Description
000400h	INT 14h	Serial Port (COM) 1 — least-significant byte
000401h	INT 14h	Serial Port (COM) 1 — most-significant byte
000402h	INT 14h	Serial Port (COM) 2 — least-significant byte
000403h	INT 14h	Serial Port (COM) 2 — most-significant byte
000404h	INT 14h	Serial Port (COM) 3 — least-significant byte
000405h	INT 14h	Serial Port (COM) 3 — most-significant byte
000406h	INT 14h	Serial Port (COM) 4 — least-significant byte
000407h	INT 14h	Serial Port (COM) 4 — most-significant byte
000408h	INT 17h	Parallel Port (LPT) 1 — least-significant byte
000409h	INT 17h	Parallel Port (LPT) 1 — most-significant byte
00040Ah	INT 17h	Parallel Port (LPT) 2 — least-significant byte
00040Bh	INT 17h	Parallel Port (LPT) 2 — most-significant byte
00040Ch	INT 17h	Parallel Port (LPT) 3 — least-significant byte
00040Dh	INT 17h	Parallel Port (LPT) 3 — most-significant byte
00040Eh	POST	Extended BIOS Data Area Segment — least-significant byte
00040Fh	POST	Extended BIOS Data Area Segment — most-significant byte

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description
000410h – 000411h	INT 11h	Equipment List: Bits Definition 15 – 14 Number of installed parallel adapters 00 = None 01 = One 10 = Two 11 = Three 13 – 12 Reserved 11 – 9 Number of installed serial adapters 000 = None 001 = One 010 = Two 011 = Three 100 = Four 101 to 111 = Reserved, not used 8 Reserved 7 – 6 Number of diskette drives 00 = One drive 01 = Two drives 10 to 11 = Reserved 5 – 4 Initial video mode 00 = EGA or PGA 01 = 40 x 25 color 10 = 80 x 25 color 11 = 80 x 25 monochrome 3 Reserved 2 PS/2-type point device 0 = Not present 1 = Present 1 Math coprocessor 0 = Not present 1 = Present 0 Diskette drive A 0 = Not present 1 = Present
000412h	POST	Interrupt Flag used in POST
000413h	INT 12h	Memory size in KB — least-significant byte
000414h	INT 12h	Memory size in KB — most-significant byte
000415h – 000416h		Reserved
000417h	INT 16h	Keyboard Status Byte: Bits Definition 7 Insert mode: 0 = Off 1 = On 6 Caps Lock mode: 0 = Off 1 = On 5 Num Lock mode: 0 = Off 1 = On 4 Scroll Lock mode: 0 = Off 1 = On 3 *Alt* key pressed: 0 = No 1 = Yes 2 *Ctrl* key pressed: 0 = No 1 = Yes 1 Left *Shift* key pressed:0 = No 1 = Yes 0 Rt *Shift* key pressed: 0 = No 1 = Yes

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description
000418h	INT 16h	Extended Keyboard Status Byte: Bits Definition 7 *Ins* key pressed: 0 = No 1 = Yes 6 *Caps Lock* pressed: 0 = No 1 = Yes 5 *Num Lock* pressed: 0 = No 1 = Yes 4 *Scroll Lock* pressed: 0 = No 1 = Yes 3 *Ctrl / NumLock* active: 0 = No 1 = Yes 2 *SysRq* key pressed: 0 = No 1 = Yes 1 Left *Alt* key pressed: 0 = No 1 = Yes 0 Left *Ctrl* key pressed: 0 = No 1 = Yes
000419h		Reserved
00041Ah – 00041Bh	INT 16h	Pointer to the address of the next character in the keyboard buffer
00041Ch – 00041Dh	INT 16h	Pointer to the address of the last character in the keyboard buffer
00041Eh – 00043Dh	INT 16h	Keyboard buffer (32 bytes) — if the address in 00041Ah is the same as the address in 00041Ch, the buffer is empty. If the address in 00041Ch is two bytes from the address in 00041Ah, the buffer is full.
00043Eh	INT 13h	Diskette Drive Calibration Status: Bits Definition 7 – 4 Reserved, should be 0000 3 – 2 Reserved 1 Drive B recalib. reqd.? 0 = Yes 1 = No 0 Drive A recalib. reqd.? 0 = Yes 1 = No
00043Fh	INT 13h	Diskette Drive Motor Status: Bits Definition 7 Current operation: 0 = Write or Format 1 = Read or Verify 6 Reserved 5 – 4 Drive select: 00 = Drive A selected 01 = Drive B selected 10 to 11 = Reserved 3 – 2 Reserved 1 Drive A: 0 = Motor is off 1 = Motor is on 0 Drive B: 0 = Motor is off 1 = Motor is on
000440h	INT 13h	Diskette Drive Motor Timeout: The system uses the INT 08h timer interrupt (occurs at a rate of 18.2 times per second) to decrement this value. When the value goes to zero, the system turns off the drive motor power. The signal applies to the last drive accessed.

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description		
000441h	INT 13h	Status for	Bits	Definition
		last accessed	7	Ready Status:
		Diskette Drive:		0 = Ready
				1 = Not ready
			6	Seek Error:
				0 = None detected
				1 = Error detected
			5	Drive Failure:
				0 = None detected
				1 = Failure detected
			4 – 0	Error Codes:
				00000 = No error
				00001 = Illegal function
				00010 = Address mark not found
				00011 = Write protect error
				00100 = Sector not found
				00101 = Reserved
				00110 = Drive door open
				00111 = Reserved
				01000 = DMA overrun error
				01001 = DMA boundary error
				01010 to 01011 = Reserved
				01100 = Unknown media
				01101 to 01111 = Reserved
				10000 = CRC failed on read
				10001 to 11111 = Reserved
000442h – 000448h	INT 13h	Diskette drive command and status bytes		
000449h	INT 10h	Current video display mode		
00044Ah – 00044Bh	INT 10h	Number of text columns per line of current video mode		
00044Ch – 00044Dh	INT 10h	Current page size in bytes		
00044Eh – 00044Fh	INT 10h	Offset address of current display page, relative to the start of video RAM — video RAM starts at B800h in CGA and B000h in MDA.		

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description
000450h – 00045Fh	INT 10h	Current cursor position for each of the eight possible video display pages — two bytes store the current cursor position for each page: the MSB specifies the row (line) value; the LSB specifies the column value. ***Note:*** *DO NOT CHANGE THE VALUES AT THIS LOCATION! Use INT 10h functions to change the video page values.*
000460h	INT 10h	Starting line of the cursor
000461h	INT 10h	Ending line of the cursor
000462h	INT 10h	Current video display page number
000463h – 000464h	INT 10h	I/O port address of the video display adapter: 3B4h = monochrome adapter 3D4h = color adapter
000465h	INT 10h	Video display adapter mode register: 3B8h = monochrome adapter 3D8h = CGA adapter 3D9h = EGA or VGA adapter
000466h	INT 10h	Current palette color
000467h – 00046Bh		Adapter ROM address
00046Ch – 00046Fh	INT 1Ah	Counter for INT 1Ah — the system increments this counter using the INT 08h timer interrupt (occurs 18.2 times per second). After 24 hours, the system resets the timer to 0.
000470h	INT 1Ah	Timer 24-hour flag Bits / Definition 7 – 1 Reserved 0 Flag value: 0 = Timer value is 0 – 24 hours 1 = Timer value > 24 hours (requires manual reset)
000471h	INT 16h	Break Status Bits / Definition 7 0 = No break signaled 1 = *Ctrl* & *Break* or *Ctrl* & *C* keys pressed 6 – 0 Reserved, not used
000472h – 000473h	POST	Soft reset flag — if value = 1234h, reboot skips the memory test.

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description
000474h	INT 13h	Status of last hard drive operation: Value Definition 00h No error 01h Invalid function request 02h Address mark not found 03h Reserved 04h Sector not found 05h Reset failed 06h Reserved 07h Drive parameter activity failed 08h DMA overrun on operation 09h Data boundary error 0Ah Bad sector flag selected 0Bh Bad track detected 0Ch Reserved 0Dh Invalid number of sectors on format 0Eh Control data address mark detected 0Fh DMA arbitration level out of range 10h Uncorrectable ECC or CRC error 11h ECC corrected data error 12h – 1Fh Reserved 20h General controller failure 21h – 3Fh Reserved 40h Seek operation failure 41h – 7Fh Reserved 80h Timeout 81h – A9h Reserved AAh Drive not ready ABh – BAh Reserved BBh Undefined error occurred BCh – CBh Reserved CCh Write fault on selected drive CDh – DFh Reserved E0h Status error, or error register is 0 E1h – FEh Reserved FFh Sense operation failed
000475h	INT 13h	Number of hard drives
000476h – 000477h	INT 13h	Hard drive work area
000478h	INT 17h	Parallel Port (LPT) 1 timeout counter
000479h	INT 17h	Parallel Port (LPT) 2 timeout counter
00047Ah	INT 17h	Parallel Port (LPT) 3 timeout counter
00047Bh		Reserved

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description
00047Ch	INT 14h	Serial Port (COM) 1 timeout counter
00047Dh	INT 14h	Serial Port (COM) 2 timeout counter
00047Eh	INT 14h	Serial Port (COM) 3 timeout counter
00047Fh	INT 14h	Serial Port (COM) 4 timeout counter
000480h – 000481h	INT 16h	Starting address of the keyboard buffer (usually 01Eh)
000482h – 000483h	INT 16h	Ending address of the keyboard buffer (usually 03Eh)
000484h	INT 10h	Number of displayed character rows minus one
000485h – 000486h	INT 10h	Height of character matrix
000487h	INT 10h	Video Status: Bits Definition 7 Equals bit 7 of the video mode number passed through INT 10h by the programmer 6 – 4 Video RAM size: 000 = 64K 001 = 128K 010 = 192K 011 = 256K 100 = 512K 101 = Reserved 110 = 1024K 111 = Reserved 3 Video subsystem status: 0 = Active 1 = Inactive 2 Reserved 1 Monitor type: 0 = Color 1 = Monochrome 0 Alphanumeric cursor emulation 0 = Disabled 1 = Enabled
000488h	INT 13h	Hard disk drive data transmission speed

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description				
000489h	INT 10h	VGA Video Flags:	Bits	Definition		
			7 & 4	Mode:		
				0XX0 = 350 lines		
				0XX1 = 400 lines		
				1XX0 = 200 lines		
				1XX1 = Reserved		
			6	Display switch:		
				0 = Disabled		
				1 = Enabled		
			5	Reserved		
			4	See 7 & 4 above		
			3	Default palette loading:		
				0 = Disabled		
				1 = Enabled		
			2	Monitor type:		
				0 = Color		
				1 = Monochrome		
			1	Grayscale summing		
				0 = Disabled		
				1 = Enabled		
			0	VGA		
				0 = Inactive		
				1 = Active		
00048Ah – 00048Bh		Reserved				
00048Ch – 000495h	INT 13h	Hard drive and diskette drive variables				
000496h	INT 16h	Extended Keyboard	Bits	Definition		
		Status:	7	Read ID in progress:	0 = No	1 = Yes
			6	Last code was 1st ID:	0 = No	1 = Yes
			5	Forced Num Lock:	0 = No	1 = Yes
			4	101/102 keyboard:	0 = No	1 = Yes
			3	Right *Alt* key active:	0 = No	1 = Yes
			2	Right *Ctrl* key active:	0 = No	1 = Yes
			1	Last code was E0h:	0 = No	1 = Yes
			0	Last code was E1h:	0 = No	1 = Yes
000497h	INT 16h	Extended Keyboard	Bits	Definition		
		Status:	7	Keyboard error:	0 = No	1 = Yes
			6	LED updating:	0 = No	1 = Yes
			5	Resend code recd.:	0 = No	1 = Yes
			4	Ack. code recd.:	0 = No	1 = Yes
			3	Reserved		
			2	Caps Lock LED on:	0 = No	1 = Yes
			1	Num Lock LED on:	0 = No	1 = Yes
			0	Scroll Lock LED on:	0 = No	1 = Yes
000498h – 000499h		Segment part of user wait flag address				
00049Ah – 00049Bh		Offset part of user wait flag address				

Table H-1 BIOS Map Contents (continued)

Address	BIOS Service	Description
00049Ch – 00049Fh		Wait count
0004A0h	INT 1Ah	Wait active flag: Bits / Definition 7 Wait time elapsed: 0 = No 1 = Yes 6 – 1 Reserved 0 INT 15h AH = 86h occurred: 0 = No 1 = Yes
0004A1h – 0004A7h		Reserved
0004A8h – 0004ABh	INT 10h	Pointer to EGA and VGA parameter control block
0004ACh – 0004EFh		Reserved
0004F0h – 0004FFh		Intra-applications communication area — stores data available to application programs.

APPENDIX

I TYPICAL CMOS RAM MAP

IBM-compatible personal computer systems that conform to the ISA standard have at least 64 bytes of CMOS RAM to store system initialization and configuration parameter values. Typically, the values are set using a BIOS setup utility. The setup utility is usually ROM- or Flash RAM-based. Some utilities can only be accessed at system startup; others can be invoked at any time from the DOS prompt using a “hot key” combination, such as Alt + Ctrl + Esc or Alt + Ctrl + S.

The following table identifies the elements in a typical CMOS RAM map:

Table I-1 Example CMOS RAM Map

Offset	Description		
00h	Real-Time Clock — Seconds. Contains the seconds value for the current time.		
01h	Real-Time Clock — Seconds alarm. Contains the seconds value for the RTC alarm.		
02h	Real-Time Clock — Minutes. Contains the minutes value for the current time.		
03h	Real-Time Clock — Minutes alarm. Contains the minutes value for the RTC alarm.		
04h	Real-Time Clock — Hours. Contains the hours value for the current time.		
05h	Real-Time Clock — Hours alarm. Contains the hours value for the RTC alarm.		
06h	Real-Time Clock — Day of the Week. Contains the current day of the week.		
07h	Real-Time Clock — Date. Contains the day (1 – 31) of the current month.		
08h	Real-Time Clock — Month. Contains the current month (1 – 12).		
09h	Real-Time Clock — Year. Contains the current year (00 – 99).		
0Ah	Status Register A	Bits: 7 6 – 4 3 – 0	Description: Update in progress (cannot read date/time): 0 = No 1 = Yes Selects the clock divider frequency Default = 010 (32.768 KHz) Selects the output frequency and periodic interrupt rate Default = 0110 (1.024 KHz and 976.562 seconds)

Table I-1 Example CMOS RAM Map (continued)

Offset	Description		
0Bh	Status Register B	Bits:	Description:
		7	Halt cycle to set clock: 0 = Updates counter once per second 1 = Halts the counter to set the clock
		6	Periodic interrupt: 0 = Disable 1 = Enable
		5	Alarm Interrupt: 0 = Disable 1 = Enable
		4	Update-Ended Interrupt: 0 = Disable 1 = Enable
		3	Square Wave: 0 = Disable 1 = Use square wave rate set by Status Register A
		2	Date and Time Mode: 0 = Use BCD format 1 = Use binary format
		1	24/12-Hour Mode: 0 = Use 12-hour mode 1 = Use 24-hour mode
		0	Daylight Savings Time 0 = Disable 1 = Enable
0Ch	Status Register C	Bits:	Description:
		7	IRQ Flag (read only)
		6	Periodic Interrupt Flag (read only)
		5	Alarm Interrupt Flag (read only)
		4	Update Interrupt Flag (read only)
		3 – 0	Reserved, should always be 0000
0Dh	Status Register D	Bits:	Description:
		7	CMOS RAM valid: 0 = Battery low, CMOS RAM not valid 1 = Battery good, CMOS RAM valid
		6 – 0	Reserved, should always be 000 0000

Table I-1 Example CMOS RAM Map (continued)

Offset	Description
0Eh	Diagnostic Status Bits: Description: 7 RTC Chip Power: 0 = Power valid 1 = Power invalid 6 CMOS RAM Checksum error: 0 = Checksum valid 1 = Checksum invalid 5 CMOS RAM Configuration Mismatch: 0 = Configuration match 1 = CMOS RAM configuration does not match system configuration 4 CMOS RAM Memory Size Mismatch: 0 = Memory matches configuration 1 = CMOS RAM memory size does not match detected size 3 Hard drive C: initialization: 0 = Drive initialized, attempting to boot 1 = Drive failed to initialize; no boot attempted 2 Time Status indicator: 0 = Time is valid 1 = Time is invalid 1 – 0 Reserved, should always be 00
0Fh	Shutdown Status. When the processor switches from protected mode to real mode, it saves the contents of its registers to memory and performs a reset. If a program requests a shutdown (by requesting a DWORD JMP instruction), the processor stores the segment address at 40:67h and the offset address at 40:69h. Before performing the reset, the processor writes a shutdown code to the CMOS RAM offset 0Fh. This allows the programmer to determine the cause of the shutdown after the system resets. Code Value Description 00h Normal POST execution 01h Chipset initialization for Real Mode reentry 02h – 03h Used internally by BIOS 04h Jump to bootstrap code 05h User-defined shutdown. The routine issues an EOI, flushes the keyboard buffer, initializes the interrupt controller and math coprocessor, and jumps to the doubleword pointer at 40:67h. 06h Jump to the doubleword pointer at 40:67h without issuing an EOI 07h Return to INT 15h Function 87h 08h Return POST memory test 09h INT 15h Function 87h Block Move shutdown request 0Ah User-defined shutdown requested. The BIOS jumps to the doubleword pointer at 40:67h without issuing an EOI or initializing the interrupt controller or math coprocessor. 0Bh Return through the doubleword pointer at 40:67h The remainder of the possible codes are not defined.

Table I-1 Example CMOS RAM Map (continued)

Offset	Description		
10h	Diskette Drive Type:	Bits:	Description:
		7 – 4	Drive A type:
			0000 = No drive
			0001 = 360 Kbyte drive
			0010 = 1.2 Mbyte drive
			0011 = 720 Kbyte drive
			0100 = 1.44 Mbyte drive
			0101 – 1111 = Undefined
		3 – 0	Drive B type:
			0000 = No drive
			0001 = 360 Kbyte drive
			0010 = 1.2 Mbyte drive
			0011 = 720 Kbyte drive
			0100 = 1.44 Mbyte drive
			0101–1111 = Undefined
11h	Advance Setup	Bits:	Description:
	Options:	7	PS/2 mouse:
			0 = Disable
			1 = Enable
		6	Test memory above 1 Mbyte:
			0 = Disable
			1 = Enable
		5	Memory test tick sound:
			0 = Disable
			1 = Enable
		4	Memory parity error check:
			0 = Disable
			1 = Enable
		3	Message display during boot:
			0 = Disable
			1 = Enable
		2	User-defined hard disk type:
			0 = Store at 0:300h
			1 = Store in upper 1 Kbyte of DOS area
		1	Wait for F1 key message if error occurs:
			0 = Disable
			1 = Enable
		0	Num Lock at boot
			0 = Off
			1 = On

Table I-1 Example CMOS RAM Map (continued)

Offset	Description		
12h	Hard Drive Type:	Bits:	Description:
		7 – 4	Drive C type:
		3 – 0	Drive D type:
			Values for both:
			0000 = No drive
			0001 = Type 1
			0010 = Type 2
			0011 = Type 3
			0100 = Type 4
			0101 = Type 5
			0110 = Type 6
			0111 = Type 7
			1000 = Type 8
			1001 = Type 9
			1010 = Type 10
			1011 = Type 11
			1100 = Type 12
			1101 = Type 13
			1110 = Type 14
			1111 = Types 16 – 46
			(actual value stored in 19h for C or 1Ah for D)
13h	Keyboard	Bits:	Description:
	Typematic Data:	7	Typematic Function:
			0 = Disable
			1 = Enable
		6 – 5	Typematic rate delay:
			00 = 250 ms
			01 = 500 ms
			10 = 750 ms
			11 = 1000 ms
		4 – 2	Typematic Rate:
			000 = 6 cps
			001 = 8 cps
			010 = 10 cps
			011 = 12 cps
			100 = 15 cps
			101 = 20 cps
			110 = 24 cps
			111 = 30 cps

Table I-1 Example CMOS RAM Map (continued)

Offset	Description
14h	Equipment Byte Bits: Description: 7 – 6 Number of Diskette Drives: 00 = None 01 = One 10 = Two 11 = Reserved, not used 5 – 4 Monitor Type: 00 = Not CGA or MDA 01 = 40x25 CGA 10 = 80x25 CGA 11 = MDA (monochrome) 3 Display: 0 = Not installed 1 = Installed 2 Keyboard: 0 = Not installed 1 = Installed 1 Math coprocessor: 0 = Not installed 1 = Installed 0 Diskette drive installed, always 1
15h	Base memory in 1K increments, least-significant byte
16h	Base memory in 1K increments, most-significant byte
17h	Extended memory in 1K increments, least-significant byte
18h	Extended memory in 1K increments, most-significant byte
19h	Hard drive C: drive type if 12h, bits 7 – 4 = 1111 Values 00h to 0Fh are reserved; 10h to 2Eh equal drive types 16 – 46, respectively
1Ah	Hard drive D: drive type if 12h, bits 3 – 0= 1111 Values 00h to 0Fh are reserved; 10h to 2Eh equal drive types 16 – 46, respectively
1Bh	Hard drive C: Least-significant byte of the cylinder number for user-defined hard drive type
1Ch	Hard drive C: Most-significant byte of the cylinder number for user-defined hard drive type
1Dh	Hard drive C: Head number for user-defined hard drive type
1Eh	Hard drive C: Least-significant byte of the write-precompensation cylinder number for user-defined hard drive type
1Fh	Hard drive C: Most-significant byte of the write-precompensation cylinder number for user-defined hard drive type
20h	Hard drive C: Control byte (= 80h if head number ≥ 8) for user-defined hard drive type
21h	Hard drive C: Least-significant byte of the landing zone number for user-defined hard drive type
22h	Hard drive C: Most-significant byte of the landing zone number for user-defined hard drive type
23h	Hard drive C: Number of sectors for user-defined hard drive type
24h	Hard drive D: Least-significant byte of the cylinder number for user-defined hard drive type
25h	Hard drive D: Most-significant byte of the cylinder number for user-defined hard drive type
26h	Hard drive D: Head number for user-defined hard drive type
27h	Hard drive D: Least-significant byte of the write-precompensation cylinder number for user-defined hard drive type

Table I-1 Example CMOS RAM Map (continued)

Offset	Description		
28h	Hard drive D: Most-significant byte of the write-precompensation cylinder number for user-defined hard drive type		
29h	Hard drive D: Control byte (= 80h if head number ≥ 8) for user-defined hard drive type		
2Ah	Hard drive D: Least-significant byte of the landing zone number for user-defined hard drive type		
2Bh	Hard drive D: Most-significant byte of the landing zone number for user-defined hard drive type		
2Ch	Hard drive D: Number of sectors for user-defined hard drive type		
2Dh	Miscellaneous	Bits:	Description:
	BIOS options:	7	Weitek coprocessor: 0 = Not installed 1 = Present
		6	Diskette Drive Seek 0 = Disabled for fast boot 1 = Enabled
		5	System Boot Sequence: 0 = C:, then A: 1 = A:, then C:
		4	System Speed at Bootup 0 = Fast 1 = Slow
		3	External Cache Memory Test: 0 = Disable (use if no external cache installed) 1 = Enable
		2	Internal Cache Memory Test: 0 = Disable 1 = Enable
		1	Fast Gate A20: 0 = Disable (use if system does not use Fast Gate A20) 1 = Enable
		0	Turbo Switch: 0 = Disable 1 = Enable
2Eh	Standard CMOS checksum, most-significant byte		
2Fh	Standard CMOS checksum, least-significant byte		
30h	Extended memory found by BIOS, least-significant byte		
31h	Extended memory found by BIOS, most-significant byte		
32h	Century byte — the BCD value for the current century		
33h	Information Flag	Bits:	Description:
		7	BIOS Length: 0 = 64K 1 = 128K
		6 – 1	Reserved, should be 000 000. Used as scratchpad during POST by chipsets.
		0	POST Cache Test results: 0 = Cache bad 1 = Cache good

Table I-1 Example CMOS RAM Map (continued)

Offset	Description		
34h	Shadowing and Password:	Bits:	Description:
		7	Boot sector virus protection: 0 = Disabled 1 = Enabled
		6	Password 0 = Disabled 1 = Enabled
		5	C8000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		4	CC000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		3	D0000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		2	D4000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		1	D8000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		0	DC000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
35h	Shadowing:	Bits:	Description:
		7	E0000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		6	E4000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		5	E8000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		4	EC000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		3	F0000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		2	C0000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		1	C4000h Shadow 16K Adaptor ROM: 0 = Disabled 1 = Enabled
		0	Math Coprocessor Test: 0 = Disabled 1 = Enabled
36h	Chipset specific information		

Table I-1 Example CMOS RAM Map (continued)

Offset	Description		
37h	Password Seed and Color Option:	Bits: 7 – 4 3 – 0	Description: Password seed used in the password encryption algorithm DO NOT CHANGE! Setup Screen Color — if used, colors are BIOS dependent
38h – 3Dh	Encrypted Password		
3Eh	MSB of Extended CMOS Checksum		
3Fh	LSB of Extended CMOS Checksum		

APPENDIX

J STANDARD I/O PORT ADDRESSING

IBM-compatible personal computer systems communicate with internal and external peripheral devices using a system of industry standard port addresses. System and peripheral device designers use a variety of decoding techniques to convert these address signals into a *chip select* signal that enables communications with a specific peripheral device. Because the addresses are standardized within the personal computer industry, peripheral devices from a variety of manufacturers can operate with a variety of personal computers. New addresses continue to be defined as new peripheral devices become available. This appendix provides a cross-reference for addressing only. Refer to the peripheral device data sheet to determine how individual bits are used in a specific application or design.

Table J-1 is an I/O address map that includes the most typical peripheral devices.

Table J-1 Standard I/O Port Addresses

I/O Port	Read/Write	Description
000h	R/W	DMA channel 0 address bytes 0 and 1
001h	R/W	DMA channel 0 word count bytes 0 and 1
002h	R/W	DMA channel 1 address bytes 0 and 1
003h	R/W	DMA channel 1 word count bytes 0 and 1
004h	R/W	DMA channel 2 address bytes 0 and 1
005h	R/W	DMA channel 2 word count bytes 0 and 1
006h	R/W	DMA channel 3 address bytes 0 and 1
007h	R/W	DMA channel 3 word count bytes 0 and 1
008h	R	DMA channel 0 – 3 Status Register
	W	DMA channel 0 – 3 Command Register
009h	W	DMA channel 0 – 3 Request Register
00Ah	R/W	DMA channel 0 – 3 Mask Register
00Bh	W	DMA channel 0 – 3 Mode Register
00Ch	W	DMA channel 0 – 3 Clear Byte Pointer Flip/Flop
00Dh	R	DMA channel 0 – 3 Temporary Register
00Eh	W	DMA channel 0 – 3 Clear Mask Register
00Fh	W	DMA channel 0 – 3 Write Mask Register
010h – 01Fh		Reserved or not assigned

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
020h	R	*Interrupt Controller 1* Interrupt Request Register (IRR) or In-Service Register (ISR) (as selected by OCW3)
	W	*Interrupt Controller 1* Initialization Command Word 1 (ICW1) Register (if bit 4 = 1) or Operational Command Word 3 (OCW3) Register (if bit 4 = 0 and bit 2 = 1)
021h	R/W	*Interrupt Controller 1* Operation Control Word 1 (OCW1) Register (Mask Register)
	W	*Interrupt Controller 1* Initialization Control Word 2 (ICW2) Register Initialization Control Word 3 (ICW3) Register Initialization Control Word 4 (ICW4) Register (if enabled by ICW1) Operation Control Word 2 (OCW2) Register (if bit 4 = 0 and bit 3 = 0)
022h – 03Fh		Reserved or not assigned
040h	R/W	Programmable Counter/Timer 0
041h	R/W	Programmable Counter/Timer 1
042h	R/W	Programmable Counter/Timer 2
043h	W	Programmable Counter/Timer Control Word Register
044h – 05Fh		Reserved or not assigned
060h	R	Keyboard Controller Data Port or Keyboard Input Buffer
	W	Keyboard Output Port
061h	R/W	Port B Control Register
062h – 063h		Reserved or not assigned
064h	R	Keyboard Controller Status Register or Keyboard Input Buffer
	W	Keyboard Output Port (alternate)
065h – 06Fh		Reserved or not assigned
070h	R	RTC Register (bits 6 – 0) and NMI Mask (bit 7)
071h	R/W	CMOS RAM Data Register Port
072h – 07Fh		Reserved or not assigned
080h	R	Manufacturing test port (for POST checkpoints)
	R/W	DMA Page Register temporary storage
081h	R/W	DMA channel 2 address byte 2
082h	R/W	DMA channel 3 address byte 2
083h	R/W	DMA channel 1 address byte 2
084h	R/W	Additional DMA page register

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
085h	R/W	Additional DMA page register
086h	R/W	Additional DMA page register
087h	R/W	DMA channel 0 address byte 2
088h	R/W	Additional DMA page register
089h	R/W	DMA channel 6 address byte 2
08Ah	R/W	DMA channel 7 address byte 2
08Bh	R/W	DMA channel 5 address byte 2
08Ch	R/W	Additional DMA page register
08Dh	R/W	Additional DMA page register
08Eh	R/W	Additional DMA page register
08Fh	R/W	DMA refresh page register
090h – 09Fh		Reserved or not assigned
0A0h	R	*Interrupt Controller 2* Interrupt Request Register (IRR) or In-Service Register (ISR) (as selected by OCW3)
	W	*Interrupt Controller 2* Initialization Command Word 1 (ICW1) Register (if bit 4 = 1) or Operational Command Word 3 (OCW3) Register (if bit 4 = 0 and bit 2 = 1)
0A1h	R/W	*Interrupt Controller 2* Operation Control Word 1 (OCW1) Register (Mask Register)
	W	*Interrupt Controller 2* Initialization Control Word 2 (ICW2) Register Initialization Control Word 3 (ICW3) Register Initialization Control Word 4 (ICW4) Register (if enabled by ICW1) Operation Control Word 2 (OCW2) Register (if bit 4 = 0 and bit 3 = 0)
0A2h – 0BFh		Reserved or not assigned
0C0h – 0C1h	R/W	DMA channel 4 address bytes 0 and 1
0C2h – 0C3h	R/W	DMA channel 4 word count bytes 0 and 1
0C4h – 0C5h	R/W	DMA channel 5 address bytes 0 and 1
0C6h – 0C7h	R/W	DMA channel 5 word count bytes 0 and 1
0C8h – 0C9h	R/W	DMA channel 6 address bytes 0 and 1
0CAh – 0CBh	R/W	DMA channel 6 word count bytes 0 and 1
0CCh – 0CDh	R/W	DMA channel 7 address bytes 0 and 1
0CEh – 0CFh	R/W	DMA channel 7 word count bytes 0 and 1

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
0D0h – 0D1h	R	DMA channel 4 – 7 Status Register
	W	DMA channel 4 – 7 Command Register
0D2h – 0D3h	W	DMA channel 4 – 7 Request Register
0D4h – 0D5h	R/W	DMA channel 4 – 7 Mask Register
0D6h – 0D7h	W	DMA channel 4 – 7 Mode Register
0D8h – 0D9h	W	DMA channel 4 – 7 Clear Byte Pointer Flip/Flop
0DAh – 0DBh	R	DMA channel 4 – 7 Temporary Register
0DCh – 0DDh	W	DMA channel 4 – 7 Clear Mask Register
0DEh – 0DFh	W	DMA channel 4 – 7 Write Mask Register
0E0h – 0EFh		Reserved or not assigned
0F0h		Math coprocessor clear busy latch
0F1h		Math coprocessor reset
0F2h – 0FFh	R/W	Math coprocessor
100h – 16Fh		Reserved or not assigned
170h	R/W	Hard drive 1 Data Register
171h	R	Hard drive 1 Error Register
	W	Hard drive 1 Write Precompensation Register
172h	R/W	Hard drive 1 Sector Count
173h	R/W	Hard drive 1 Sector Number
174h	R/W	Hard drive 1 Cylinder Number (low byte)
175h	R/W	Hard drive 1 Cylinder Number (high byte)
176h	R/W	Hard drive 1 Drive/Head Number
177h	R	Hard drive 1 Status Register
	W	Hard drive 1 Command Register
178h – 1EFh		Reserved or not assigned
1F0h	R/W	Hard drive 0 Data Register
1F1h	R	Hard drive 0 Error Register
	W	Hard drive 0 Write Precompensation Register
1F2h	R/W	Hard drive 0 Sector Count
1F3h	R/W	Hard drive 0 Sector Number
1F4h	R/W	Hard drive 0 Cylinder Number (low byte)
1F5h	R/W	Hard drive 0 Cylinder Number (high byte)
1F6h	R/W	Hard drive 0 Drive/Head Number

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
1F7h	R	Hard drive 0 Status Register
	W	Hard drive 0 Command Register
1F8h – 1FFh		Reserved or not assigned
200h – 20Bh	R/W	Game controller ports
20Ch – 277h		Reserved or not assigned
278h	R/W	Parallel Port 2 Data Port
279h	R/W	Parallel Port 2 Status Port
27Ah	R/W	Parallel Port 2 Control Port
27Bh	R/W	Reserved or not assigned
27Ch	R/W	Parallel Port 2 Data Port (DUP)
27Dh	R/W	Parallel Port 2 Status Port (DUP)
27Eh	R/W	Parallel Port 2 Control Port (DUP)
27Fh – 2E7h		Reserved or not assigned
2E8h	R	Serial Port 4 Receiver Buffer Register
	R/W	Serial Port 4 Divisor Latch Low Byte
2E9h	R/W	Serial Port 4 Interrupt Enable Register
2EAh	R	Serial Port 4 Interrupt ID Register
2EBh	R/W	Serial Port 4 Line Control Register
2ECh	R/W	Serial Port 4 Modem Control Register
2EDh	R	Serial Port 4 Line Status Register
2EEh	R	Serial Port 4 Modem Status Register
2EFh	R/W	Serial Port 4 Scratch Register
2F0h – 2F7h		Reserved or not assigned
2F8h	R	Serial Port 2 Receiver Buffer Register
	R/W	Serial Port 2 Divisor Latch Low Byte
2F9h	R/W	Serial Port 2 Interrupt Enable Register
2FAh	R	Serial Port 2 Interrupt ID Register
2FBh	R/W	Serial Port 2 Line Control Register
2FCh	R/W	Serial Port 2 Modem Control Register
2FDh	R	Serial Port 2 Line Status Register
2FEh	R	Serial Port 2 Modem Status Register
2FFh	R/W	Serial Port 2 Scratch Register
300h – 31Fh		Reserved for Prototype Card
320h – 371h		Reserved or not assigned

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
372h	W	Diskette Drive Controller 2 Digital Output Register
373h		Reserved or not assigned
374h	R	Diskette Drive Controller 2 Status Register
375h	R/W	Diskette Drive Controller 2 Data Register
376h	R	Diskette Drive Controller 2 Control Port
377h	R	Diskette Drive Controller 2 Digital Input Register
	W	Diskette Drive Controller 2 Select Register for Data Transfer Rate
378h	R/W	Parallel Port 1 Data Port
379h	R/W	Parallel Port 1 Status Port
37Ah	R/W	Parallel Port 1 Control Port
37Bh	R/W	Hercules-compatibility Configuration Switch Registers
37Ch	R/W	Parallel Port 1 Data Port (DUP)
37Dh	R/W	Parallel Port 1 Status Port (DUP)
37Eh	R/W	Parallel Port 1 Control Port (DUP)
37Fh		Reserved or not assigned
380h	R/W	Bisynchronous Device 2 Port A (8255A-5)
381h	R/W	Bisynchronous Device 2 Port B (8255A-5)
382h	R/W	Bisynchronous Device 2 Port C (8255A-5)
383h	W	Bisynchronous Device 2 Mode Set Register (8255)
384h	R/W	Bisynchronous Device 2 Counter 0 (8253)
385h	R/W	Bisynchronous Device 2 Counter 1 (8253)
386h	R/W	Bisynchronous Device 2 Counter 2 (8253)
387h	R/W	Bisynchronous Device 2 Control Word/Mode Register (8253/5)
388h	R	Bisynchronous Device 2 Status Register (8253)
	W	Bisynchronous Device 2 Command Register (8273)
389h	R	Bisynchronous Device 2 Parameter Result (8273)
38Ah	R/W	Bisynchronous Device 2 Transmit INT Status (8273)
38Bh	R/W	Bisynchronous Device 2 Receive INT Status (8273)
38Ch	R/W	Bisynchronous Device 2 Data (8273)
38Dh – 39Fh		Reserved or not assigned
3A0h	R/W	Bisynchronous Device 1 Port A (8255)
3A1h	R/W	Bisynchronous Device 1 Port B (8255)
3A2h	R/W	Bisynchronous Device 1 Port C (8255)
3A3h	W	Bisynchronous Device 1 Mode Set Register(8255)

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
3A4h	R/W	Bisynchronous Device 1 Counter 0 (8253)
3A5h	R/W	Bisynchronous Device 1 Counter 1 (8253)
3A6h	R/W	Bisynchronous Device 1 Counter 2 (8253)
3A7h	R/W	Bisynchronous Device 1 Control Word/Mode Register (8253/5)
3A8h	W	Bisynchronous Device 1 Data Select (8253/5)
3A9h	R/W	Bisynchronous Device 1 Mode Instruction and Command Instruction (8253/5)
3AAh – 3B3h		Reserved or not assigned
3B4h	R/W	MDA CRTC Index Register
3B5h	R/W	MDA Video CRTC data registers: Index Function 00h Horizontal total 01h Horizontal displayed 02h Horizontal sync position 03h Horizontal sync pulse width 04h Vertical total 05h Vertical displayed 06h Vertical sync position 07h Vertical sync pulse width 08h Interleaved mode 09h Maximum scan lines 0Ah Cursor start 0Bh Cursor end 0Ch Start address (high byte) 0Dh Start address (low byte) 0Eh Cursor location (high byte) 0Fh Cursor location (low byte) 10h Light pen (high byte) 11h Light pen (low byte) 12h – FFh Undefined
3B6h – 3B7h		Reserved or not assigned
3B8h	W	MDA Mode Control Register
3B9h		Reserved or not assigned
3BAh	R	CRT Status Port
3BBh		Reserved or not assigned
3BCh	R/W	Parallel Port 3 Data Port
3BDh	R/W	Parallel Port 3 Status Port
3BEh	R/W	Parallel Port 3 Control Port
3BFh – 3C1h		Reserved or not assigned
3C2h	R	CGA Input Status Register
3C3h	R/W	Video Subsystem Enable
3C4h	R/W	CGA Sequencer Index Register

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
3C5h	R/W	CGA Sequencer Data Registers
3C6h – 3C9h		Reserved or not assigned
3CAh	R	CGA Feature Control Register
3CBh – 3CFh		Reserved or not assigned
3D0h – 3D1h	R/W	6845 Registers
3D2h – 3D3h		Reserved or not assigned
3D4h	W	CGA Video CRTC index register
3D5h	W	CGA Video CRTC data registers: Index Function 00h Horizontal total 01h Horizontal displayed 02h Horizontal sync position 03h Horizontal sync pulse width 04h Vertical total 05h Vertical displayed 06h Vertical sync position 07h Vertical sync pulse width 08h Interleaved mode 09h Maximum scan lines 0Ah Cursor start 0Bh Cursor end 0Ch Start address (high byte) 0Dh Start address (low byte) 0Eh Cursor location (high byte) 0Fh Cursor location (low byte) 10h Light pen (high byte) 11h Light pen (low byte) 12h – FFh Undefined
3D6h – 3D7h		Reserved or not assigned
3D8h	R/W	CGA Mode Control Register
3D9h	R/W	CGA Palette Register
3DAh	R/W	CRT Status Register
3DBh	W	Clear Light Pen Latch
3DCh	W	Preset Light Pen Latch
3DDh – 3E7h		Reserved or not assigned
3E8h	R	Serial Port 3 Receiver Buffer Register
	R/W	Serial Port 3 Divisor Latch Low Byte
3E9h	R/W	Serial Port 3 Interrupt Enable Register
3EAh	R	Serial Port 3 Interrupt ID Register
3EBh	R/W	Serial Port 3 Line Control Register
3ECh	R/W	Serial Port 3 Modem Control Register
3EDh	R	Serial Port 3 Line Status Register

Table J-1 Standard I/O Port Addresses (continued)

I/O Port	Read/Write	Description
3EEh	R	Serial Port 3 Modem Status Register
3EFh	R/W	Serial Port 3 Scratch Register
3F0h – 3F1h		Reserved or not assigned
3F2h	W	Diskette Drive Controller 1 Digital Output Register
3F3h		Reserved or not assigned
3F4h	R	Diskette Drive Controller 1 Status Register
3F5h	R/W	Diskette Drive Controller 1 Data Register
3F6h	R	Diskette Drive Controller 1 Control Port
3F7h	R	Diskette Drive Controller 1 Digital Input Register
	W	Diskette Drive Controller 1 Select Register for Data Transfer Rate
3F8h	R	Serial Port 1 Receiver Buffer Register
	R/W	Serial Port 1 Divisor Latch Low Byte
3F9h	R/W	Serial Port 1 Interrupt Enable Register
3FAh	R	Serial Port 1 Interrupt ID Register
3FBh	R/W	Serial Port 1 Line Control Register
3FCh	R/W	Serial Port 1 Modem Control Register
3FDh	R	Serial Port 1 Line Status Register
3FEh	R	Serial Port 1 Modem Status Register
3FFh	R/W	Serial Port 1 Scratch Register

GLOSSARY

Abort An unrecoverable exception.

Address See I/O Address, Logical Address, Linear Address, and Physical Address.

Address Line A signal line that is part of an address bus. For Am486 CPU-based systems, the bus uses 32 address lines to connect to memory or devices on the I/O bus. The processor uses the M/$\overline{\text{IO}}$ signal to specify whether the microprocessor is addressing memory or an I/O device.

Address Space The range of addressable memory locations.

AddressSize Prefix Optional programming code used before an instruction that defines the size of address offsets, which can be 16 or 32 bits in length. The D bit in the instruction code segment defines a default AddressSize, but the prefix overrides that default.

Address Translation Remapping of memory locations that allows the same physical memory address space to be used by multiple applications. Segmentation and paging use address translation to protect memory locations from being overwritten. Paging uses the Present bit to swap data between disk storage and memory, expanding the translation capability.

Alignment The placement of code or data on a 2-, 4-, 8-, 16-, or 32-byte boundary depending on the operand or cache-line size.

Application Program A higher level user program generally assigned the highest privilege number and lowest privilege level. Application programs require an operating system and, for some applications, an interface program, such as Microsoft Windows, to run correctly.

ASCII *American Standard Code for Information Interchange.* An international standard for coding text characters that uses 7 or 8 bits per character. The standard set of characters uses the first 128 value combinations (0 to 127 decimal); some older serial communication protocols only used 7 bits of data (bits 6–0) per byte, reserving the top bit (7) for control purposes. The extended ASCII character set uses all eight bits per byte and assigns 128 additional characters for the values 128 to 255 decimal.

Base Address A defined address that indicates the beginning of a data structure or table in memory. Using a base address allows greater flexibility to locate and access segments, descriptor tables, pages, page tables, and for input/output devices, configuration tables.

Base Register A register that stores a base address for a set of data. Data within the data set is addressed via offsets from the address in the base register.

Baud A variable unit of measure used for serial transmission of binary data across data lines; usually equal to one bit per second.

Biased Exponent

The form of the exponent used by the floating-point unit. The biased exponent is interpreted as an unsigned, positive number. The value is computed by adding a constant (the bias) to the true exponent of the real number. To get the true exponent for a non-zero number, subtract the bias for the precision level (127 for single, 1023 for double, and 32767 for extended) from the value in the exponent field.

Binary

A number system based on the value of two. It is the system used by computers at a circuit level because the basic computer circuit has two states, On and Off, that are interpreted as numerical 1’s and 0’s.

Binary Coded Decimal (BCD)

A method of representing base 10 (decimal) numbers using binary encoding. Each decimal digit uses four bits; the values 1010 through 1111 are not used. Standard BCD format encodes the four bits as part of a byte, ignoring the upper four bits. The Am486 microprocessor floating-point unit supports a fixed-length (18 digits) packed BCD format that stores two decimal numbers per byte, using both the lower and upper four bits of the byte.

Binary Integer

A whole number represented in the binary (base 2) form using only the symbols 0 and 1.

Binary Point

The binary equivalent of the decimal point in real number format.

BIOS

Basic Input/Output System. The system drivers that define the default system handlers for system interrupts and exceptions. The BIOS software is stored on a static memory device, such as ROM or Flash memory, that retains data with no power supplied. Whenever the microprocessor is reset to its initial state, it begins operation by reading the BIOS into memory and executing power-on self-tests, loading the vector addresses into low memory, and loading the handlers into memory at the vector-referenced locations. When the tests are complete and the vector addresses and handlers are loaded into memory, the BIOS relinquishes control of the computer to the operating system software.

Bit

The smallest unit of information storage in a computer system. The basic computer circuit used to represent logical values has two states: On and Off. The output from this circuit is typically interpreted as a logical 1 for On and a logical 0 for Off. Multiple bits are read in parallel in groups of 8 (called a byte), 16 (called a word), 32 (called a doubleword or dword), and 64 (called a quadword or qword). Although the actual number representations of the bits are in binary form (base 2 number system), typically users and programmers read the bits in sets of four using the hexadecimal number system (base 16), which is closer to the more common base ten and easier to evaluate mathematically than long binary strings.

Bit Field

A sequence of 1 to 32 bits starting at any position in a byte address.

Bit String

A sequence of 1 to 2^{32}–1 bits starting at any position in a byte address.

Boot

The common term for restarting a computer, shortened from the “bootstrap” routines required to start older mainframe computers. Personal computers typically use a change-of-state of the POWER GOOD signal from the computer power supply (COLD BOOT) or the keyboard controller (WARM BOOT) to initiate a microprocessor reset. Applying a signal to the RESET input of the microprocessor causes it to return to a known state and initialize by reloading the BIOS and restarting system operations.

Breakpoint A defined address range used by debugging handlers to trap information for evaluation by designers and programmers. Four breakpoints can be defined in the Debug Registers. The user can specify the breakpoint for a particular form of memory access.

Bus A set of signal lines that transmit electronic signal sets between devices in a computer. This can be a data bus that transmits data and code between components or an address bus that selects memory locations or system devices. Some devices use a single address/data bus that alternate transmissions between addressing and data/code. Other control signals determine how the system interprets the bus information.

Bus Speed The clock speed used to transmit data across a system bus.

Byte 8 bits. Memory and disk storage capacities are normally defined in bytes.

C3 – C0 The condition code bits in the floating-point unit (FPU) Status Word. These bits define the status of the outcome of some of the FPU instructions.

Cache Special fast memory that can be both internal and external to the microprocessor. The cache memory retains copies of the most recently read memory contents for quick reaccess by the microprocessor.

Cache Flush Clearing the cache memory by forcing the microprocessor to read from system memory and overwrite the contents before accessing the cache.

Cache Hit A request for data that is available in the cache memory.

Cache Line The smallest unit of cache storage. The internal cache of the Am486 microprocessor has a 128-bit cache line size.

Cache Miss A request for data that is not available in the cache memory, requiring a read from the system memory.

Call Gate A gate descriptor used by a CALL, JMP, Jcc, or LOOPcc instruction.

Cascade A method of linking controller circuits that can only input one value into the microprocessor. For example, an interrupt controller evaluates up to eight input signals, prioritizes them, and presents one interrupt signal at a time to the microprocessor. By taking a second similar controller and tying its output to one of the eight inputs to the first controller, you can process 15 interrupt signals. DMA controllers use a similar scheme to expand the DMA channel processing capability.

CD-ROM A data storage method that uses laser technology and an encoded disk to store digital data. The method uses the same technology as music recording.

CGA *Color Graphics Adapter.* The earliest color display controller that supported four-color graphic displays.

Clocking The method by which data is transferred and sampled in a digital circuit. Data is detected when a voltage level changes state (from 1 to 0, or 0 to 1); the clock signal causes this change. The clock pulse rate determines how quickly a digital circuit can move between sets of information. Typically, as newer microprocessor clock rates increased, constraints (signal loss) on the expansion buses required that a separate, slower clocking rate be used for I/O data transfers. Newer bus designs (VESA LB and PCI) allow the buses to transfer data using the same input clock and, therefore, clocking rate as the microprocessor.

CMOS Memory Although the term refers to a specific memory manufacturing type, *Complementary Metal-Oxide Semiconductor*, this phrase commonly refers to the battery-backed up memory associated with the Real-Time Clock Circuit that stores the basic computer configuration data, such as memory size and type, drive sizes and types, video interface, etc. used by the computer when it starts to select the correct BIOS handlers and parameter values.

Code Also known as instructions or instruction code. A type of binary information transmitted to a processing device that activates specific functionality within the processing device.

Code Descriptor The data table in memory that defines the type of information in the specified code segment.

Code Segment An address space containing instructions; also called an executable segment. An instruction fetch cycle must address a code segment.

COM The DOS-assigned name for a serial port. In later versions of DOS, a port can be designated as COM1, COM2, COM3, or COM4. The name implies a specific I/O address for the set of operational registers associated with the serial port.

Command A user-entered instruction name, typically associated with an operating system or command line driven application program. Typically used DOS commands include COPY, MAKEDIR, DEL, and so forth.

Condition Code See C3 – C0.

Configuration The specific details used by a computer to interact correctly with its built-in and installed devices. The configuration information may be read from hardware (such as ROM-based information or specific jumper or switch settings) or may be stored configuration files, such as those maintained in the CMOS battery-backed up memory.

Conforming Segment A code segment that executes with the Requested Privilege Level (RPL) of the segment selector or the Current Privilege Level (CPL) of the calling program, whichever has a lower privilege level (higher value).

Control Word A 16-bit register used by the floating-point unit (FPU). The user can define which modes the FPU uses and the interrupts that are enabled.

Controller An electronic device or circuit used to provide a hardware interface between the microprocessor and other system devices. Examples include drive controllers, keyboard controllers, and video controllers.

CPU *Central Processor Unit.* See microprocessor.

Current Privilege Level (CPL) The privilege level assigned to the currently executing program. Typically, the level is the Descriptor Privilege Level of the code segment descriptor assigned to the program. If, however, execution has been transferred to a conforming code segment (in which case the CPL is carried from the previous execution), the CPL may be different from the current DPL assigned to the executing code.

Data Line One of the individual signal connections in the Data Bus (see Bus).

Data Segment An address space containing data. The microprocessor provides four segment registers (DS, ES, FS, and GS) to access data segments. The respective segment descriptors describe the type of information stored in each segment.

Data Structure A memory area defined for particular use by hardware or software, such as a page table or task state segment (TSS).

Debug Registers A set of registers used to define hardware breakpoints for debugging.

Decimal Integer A whole number represented in BCD form.

Descriptor Privilege Level (DPL) The privilege level assigned to a segment through the DPL field in the segment descriptor.

Descriptor Table An array of segment descriptors. The Global Descriptor Table (GDT) defines the overall memory layout. The Local Descriptor Tables (LDT) define individual memory segments.

Device Driver A special program designed to manage the interface between the microprocessor and a peripheral device (such as a video adapter).

Dirty Bit A bit used when the microprocessor is set for Write-Back cache mode to indicate that the microprocessor wrote to the cache, but that the new value has not yet been written to memory. When the new data is transferred to memory, the microprocessor resets the dirty bit (to 0).

Disk A data storage medium with embedded data recording tracks. Hard drives use one or more of these disks.

Diskette A portable magnetic data medium that fits into a diskette drive. Like the disk, the diskette uses a single metal disk embedded with data recording tracks, but it is stored in a plastic sleeve or housing that protects the diskette contents. After recording data on the diskette, you can lock its contents mechanically to make the diskette read-only.

Diskette Drive A data storage device that has a drive motor, a set of read and write heads, and a mechanism (stepping motor/actuator) to move the heads across a diskette surfaces that uses removable diskettes to store and read data. Typically, diskette drives support 3-1/2" (720 Kbytes, 1.44 Mbytes, or 2.88 Mbytes of storage) and/or 5-1/4" (360 Kbytes or 1.2 Mbytes of storage) diskettes.

Displacement A constant used to calculate an effective address. A displacement modifies the address independently of any scaled indexing. Displacement is often used to indicate the address of operands that have a fixed relation to some other address, such as a base address or a record field in an array.

DMA *Direct Memory Access.* A method of buffering information between the I/O bus, which typically uses slower clocking, and the memory bus connected directly to the microprocessor.

DMA Channels The circuits in a DMA controller that allow it to handle multiple devices. A typical DMA controller provides four channels for data transfer. By using a cascade approach, two controllers can support seven DMA channels. Typically, the first controller supports 8-bit transfers and the second controller supports 16-bit transfers.

DMA Controller A device that provides the interface between I/O devices that require DMA support to transfer data between the I/O bus and the memory bus.

Doubleword (dword) 32 bits.

DRAM *Dynamic Random Access Memory.* Memory that can be accessed and programmed by a computer system, but loses the last value written to the memory if it loses power. This form of memory is faster and less expensive than SRAM, but uses additional power through the refresh cycles required to maintain its contents.

Driver A software program that allows the operating system software to communicate with a specific device, such as a video circuit or printer.

Effective Address The results of a calculated address. The calculation method depends on the addressing method used.

EFLAGS The Extended FLAGS register added by 32-bit processors. The FLAGS register is embedded in the lower 16 bits of EFLAGS. The flag bits in the upper word of the register add functions required by the 32-bit (386- and 486-type) and higher level processors.

EGA *Enhanced Graphics Adapter.* A color video controller that supports 16-color graphic displays.

EISA *Extended Instrumentation Society of America.* Standard for personal computer expansion slots. The EISA slots use the same basic footprint as ISA slots on a motherboard, but provide two levels of contacts that allow expansion of the data bus from 16-bits to 32-bits.

Enable Make available for use by the computer. Typically, this term is used in relation to turning on a specific functionality, such as a serial or parallel port or interrupt capability.

Exception A fault, trap, abort, or software-initiated interrupt that causes the microprocessor to execute a recovery subroutine. Exception causes can include dividing by zero, stack overflow, undefined opcode, and memory protection violation.

Far Pointer A memory reference that includes both a segment selector and an offset value.

Fault An exception reported at the instruction boundary before the instruction that generates the exception. After the exception handler fixes the source of the exception, such as a segment or page not present in memory, execution restarts.

Fax/Modem A serial device that supports the transmission and receipt of document facsimiles and other serial communications via telephone transmission lines.

FDC *Floppy Drive Controller.* A device that controls the data interface between the computer system and a diskette drive.

Firmware A program or set of programs recorded on a static memory device (such as ROM or Flash RAM) that is required for system or device functionality. The system level firmware is the BIOS. Essentially, because it is physically installed, firmware is a piece of hardware that performs software functions.

FLAGS A status register developed by the x86 microprocessor family. The FLAGS register stores a set of 1-bit system, status, and control flags.

Flag A bit whose value reflects the status of the computer system or the result of a particular operation, such as the Zero Flag (ZF) or Carry Flag (CF) in the EFLAGS or FLAGS register.

Flash Memory A reprogrammable type of long-term memory storage device used to store programs required by systems at startup or reset, such as BIOS, that does not require continuous power or refresh to maintain its contents.

Flat Model A memory management scheme in which all six segments are mapped to the same linear address range. Essentially, this scheme eliminates segmentation.

Floating-Point Unit (FPU) The part of the Am486 processor that contains the FPU registers and performs the operations requested by the floating-point instructions.

Gate Descriptor The segment descriptor that can be the destination of a CALL, JMP, Jcc, or LOOPcc instruction. You can also use a gate descriptor to invoke a procedure or task at another privilege level. The four types of gate descriptors are: call gates, trap gates, interrupt gates, and task gates.

General Register The Am486 microprocessor supports eight 32-bit general registers: EAX, EBX, ECX, EDX, EBP, EDI, ESI, and ESP. You can access the lower word in each of these registers as eight 16-bit registers: AX, BX, CX, DX, BP, DI, SI, and SP. In addition, you can access the high (H) and low (L) bytes of the first four 16-bit registers as eight 8-bit registers: AH, AL, BH, BL, CH, CL, DH, and DL.

Global Descriptor Table (GDT) An array of segment descriptors for all programs in a system. There is only one GDT in a system.

Graphical Interface A user program, such as Microsoft Windows, that provides icons that when accessed by a mouse or other pointing device initiate execution of programs referenced by the icons.

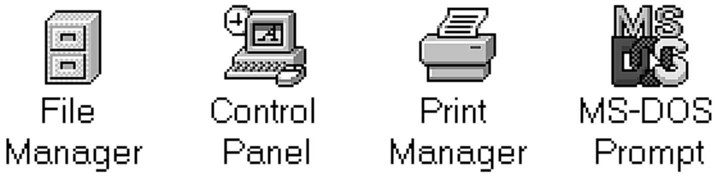
Handler A program called as a result of an exception or interrupt.

Hard Drive A data storage device that uses one or more disks, a drive motor to spin the disks, a set of read and write heads, and a mechanism (stepping motor/actuator) to move the heads across the disk surfaces.

Hertz (Hz) The unit of frequency used to describe clock speeds. It is equal to 1 cycle per second.

Hexadecimal A number system based on the number 16. This system uses the standard decimal digits 0 through 9 and adds the alphabetic characters A to F to provide 16 symbols. This document indicates numbers in the hexadecimal form by adding “h” after the number (e.g., 007Fh).

Icon A simple picture representation of a function or program used by a Graphical Interface package, such as Microsoft Windows, to initiate program execution.



IDE *Integrated Drive Electronics.* A type of hard drive in which the controller electronics are built into the drive.

Immediate Operand Data encoded into the instruction.

Index A number used to access a table. An index is scaled (multiplied by shifting left) to account for the size of the operand. The scaled index is added to the base address of the table to get the address of the table entry.

Input Device A device used by the operator to input data into a personal computer. The keyboard was the first basic input device used with the personal computer, but with the development of graphical interface software, the user can now input data using a mouse or other pointing device (trackball, for example). The newest developments in input devices include handwriting recognition devices and voice recognition devices.

Instruction A set of encoded symbol sequences that cause a microprocessor to perform a requested function. At the lowest level, an instruction can be a variable length binary code fed into the microprocessor on the data bus. Typically programmers enter the code as hexadecimal or alphanumeric values, which are recoded (compiled) to the binary level required by the microprocessor. For example, to add two numbers, 4 and 3, a programmer might have to load the first number into a register (MOV AL 4) and then add the second number (ADD AL 3). A compiler recodes these instructions as:

```
1100 0110 1100 0000 0000 0100 (MOV AL 4), and
1000 0000 1100 0000 0000 0011 (ADD AL 3).
```

Integer A number (positive, negative, or zero) that is finite and has no fractional part.

Interrupt A forced transfer of program control to a handler. In a personal computer system, the signal can come from the interrupt controller (hardware generated) or be induced by the INT instruction.

Interrupt Controller A device that provides an interface between peripheral devices requiring service and the microprocessor. An interrupt is a hardware signal sent by the device to communicate with the microprocessor. Because the microprocessor only has one device interrupt line, the controller must handle and prioritize the multiple inputs and generate only a single signal at a time. Typically, an interrupt controller can handle up to eight interrupt lines, but by using a cascade scheme, you can combine two controllers to support up to 15 interrupt lines (although several are reserved for system use). Typically, users refer to an interrupt as an IRQ, such as IRQ4 which is typically used as the interrupt line for the COM1 serial port. The interrupt controller tells the microprocessor the interrupt type by transmitting a vector number associated with the correct handler.

Interrupt Descriptor Table (IDT) An array of gate descriptors that invoke exception and interrupt handlers.

Interrupt Gate A gate descriptor used to invoke a specific interrupt handler. An interrupt gate is different from a trap gate only in its effect on the IF flag. An interrupt gate clears IF for the duration of the handler.

Invalid Operations The general exception condition for the FPU that includes stack overflow, stack underflow, NaN inputs, illegal infinite inputs, out-of-range inputs, and inputs in unsupported formats.

I/O Address A combination of signal values fed across the Address Bus to initiate contact with an input/output device on the bus. Typically, the device uses an input decode circuit to evaluate the address lines and generate a single chip select signal to activate the device connection.

I/O Device A device that performs input and/or output functions in the personal computer system. The device may be part of a supporting chipset on the motherboard, on an expansion card installed in a slot on the motherboard, or, in some newer designs, integrated into the microprocessor chip.

ISA Bus *Instrumentation Society of America.* Standard for personal computer expansion slot connections. The original specification was designed by IBM to support the 8-bit external bus on the early x86-based machines. When later processors expanded the bus to a 16-bit interface, the standard added contacts in a separate interface connector to provide the additional data lines.

Jcc A conditional JUMP instruction. A JUMP instruction that occurs only if the condition specified by the Jcc instruction is true.

JUMP A programming instruction that causes the processor to stop executing instructions consecutively within a set of sequential address locations and transfer operation to an instruction at the address specified by the JUMP instruction.

Kbyte 1024 bytes.

Keyboard An input device that is based on a typewriter keyboard. The signals from the keyboard report the x-y location of the key within the keyboard matrix, along with the current status of other control keys (such as Num Lock, Shift, Ctrl, Alt, etc.). The keyboard controller uses translation tables to convert the input into a character set recognized by the computer system. Early keyboards used an 85-key layout, but newer keyboards implement some variation of the 101-key (102-key in Europe) layout. Some keyboards used with portable computers implement the 101-key layout using fewer physical keys by implementing an embedded keyboard concept that overlays the functionality of the numeric keypad on the standard QWERTY layout.

Keyboard Controller The device that provides an interface between a keyboard and the computer system. Typically, the keyboard controller is a general purpose 8-bit processor. Newer controllers add additional functionality including power management support and PS/2 mouse and/or trackball support.

Limit Checking One of the five protection checks provided by segmentation. All segment descriptors include a 16-bit limit value that sets the lower or upper segment limit (depending on whether the Direction Flag is set to forward or reverse accessing.

Linear Address A 32-bit address in a large unsegmented address space. If paging is enabled, the linear address is translated into a physical address. If paging is disabled, the linear address is the physical address.

Local Bus A 32-bit expansion bus that uses the microprocessor input clock for data clocking. This provides higher transfer rates than allowed by the standard ISA bus.

Local Descriptor Table (LDT) An array of segment descriptors used by a particular program. A program may have a unique LDT, share an LDT with other programs, or no LDT (it uses GDT only).

LOCK An optional instruction prefix used with selected string operations that invokes the $\overline{\text{LOCK}}$ signal. This prefix can reduce required clock counts in some situations.

Logical Address Computed from a 16-bit segment selector and a 32-bit offset. The segment selector specifies an independent, protected address space. The offset defines an address within that segment. The segmentation handling in the Am486 processor converts the logical address to a linear address.

LOOPcc *Conditional Loop Instruction.* A loop that repeats until the specified condition is satisfied.

LPT The DOS-assigned name for a parallel port. Depending on system design and BIOS support, a port can be designated as LPT1, LPT2, or LPT3. The name implies a specific I/O address for the set of operational registers associated with the serial port.

Mask/Masking By setting a bit in a control register, you can mask (disable) a particular function. For example, you can mask the six FPU exceptions through the FPU control word.

Mbyte 1,048,576 bytes.

Memory Electronic circuits used to store binary data for use by the microprocessor and other devices in a computer system.

Memory Management A method of controlling access to memory. The Am486 microprocessor allows you to address memory directly or indirectly and provides two basic protection methods: segmentation and paging. Segmentation allows you to divide memory into independent and protected address spaces. Paging allows you to increase the virtual memory size by swapping data between memory and disk storage.

MGA *Monochrome Graphics Adapter.* An early video display type that supported one-color graphic displays.

Microprocessor The main execution device in a personal computer that executes instructions.

Modem *Mo*dulator-*Dem*odulator. A circuit that encodes and decodes serially transmitted digital data, typically across telephone communication lines.

modR/M byte A byte following an instruction opcode that specifies instruction operands.

Monographics One-color video displays.

Motherboard A printed circuit board that includes a microprocessor (or at least a socket for a microprocessor), memory (or memory sockets), and circuits to link other devices to the microprocessor and memory. Typically, the motherboard has expansion slots or other interface devices or connectors to allow you to expand its basic functionality. With the continuing growth of integrated circuits in microchips, the motherboard contains a greater array of components including support circuitry (chipsets) that handles the basic I/O interface requirements, including: interrupt and DMA control as well as serial and parallel ports, keyboard control, drive control (diskette, hard drive, CD-ROM drive, etc.), video control, and so forth.

Mouse A hand-held pointing device that connects to the computer through a special bus device, a serial port, or a special PS/2 type interface. The mouse movements (translated by the movement of a ball on the underside of the mouse against electronic switches into an electronic input into the computer) control the position of the cursor on the video display. The buttons on the mouse allow the user to execute programs through icons on the display.

Multimedia Developments in computer technology that allow the computer system to incorporate graphics, audio, video, and animation. The expanded storage space provided by CD-ROM technology has made the large amounts of data required for multimedia programming available.

Multisegmented Model A memory organization in which different segments are mapped to different ranges of linear addresses. This protects data structures from damage caused by program execution errors.

NaN Abbreviation for “Not a Number”. This floating-point quantity does not represent any numeric or infinite quantity.

Near Pointer A memory reference that includes an offset only without a segment selector.

Network A linkage system that allows individual workstations to communicate and share storage space.

Nibble 4 bits. A half byte.

NMI *Non-Maskable Interrupt.* The single NMI input line on the microprocessor that cannot be masked by the microprocessor.

Offset A 16-bit or 32-bit number that specifies a memory location relative to the base address in the segment. The code segment descriptor specifies a default value, but the programmer can override the default by adding an AddressSize prefix before the instruction opcode.

Opcode The numeric representation of an instruction.

Operand Data in a register or memory that the instructions reads/writes.

OperandSize Optional programming code used before an instruction that defines the size of integer operands, which can be 8 bits and 16 bits or 8 bits and 32 bits in length. The D bit in the instruction code segment defines a default OperandSize, but the prefix overrides that default.

Operating System A computer program that provides the principal user interface to the microprocessor. Typically, the operating system converts its commands to the instruction format used by the microprocessor. Application programs also can also use the operating system command set to interact with the microprocessor.

Overflow Numeric: A floating-point exception that occurs when a result is finite, but is too large to be represented in the destination format.

Stack: An exception caused by attempting to push down an non-empty stack location.

Page A 4-Kbyte block of consecutive memory locations used as the base size by the system paging mechanism.

Paging A form of memory management used to simulate a large, unsegmented address space by swapping data between memory and disk storage.

Parallel Interface An interface that uses a data bus (multiple data lines) to transfer information instead of a single data line. Typically a parallel interface transfers information in bytes (8 bits), words (16 bits), or doublewords (32 bits). Newer technology can implement quadword (64 bits) or larger parallel transfers. Parallel transfers use one clock to transfer a set of data instead of one bit at a time as in serial transfers, and therefore provides faster data transfer.

Parallel Port A hardware connector used to transfer data via a parallel interface. Typically, computer systems use a 25-pin D-shell connector. Printers that use this interface typically have a Centronics-type connector.

Parity A method used to verify the accuracy of stored or transferred data. Data is typically stored and transferred between devices as bytes. Typical parity schemes use a ninth bit called the parity bit. In an even parity scheme, it you add the eight bits and the parity bit, the result is always even if the data is correct. For odd parity, the result is always odd.

PCD Bit *Page-Level Cache Disable,* Bit 4 in CR3. This bit drives the value of the PCD output pin on the microprocessor during unpaged bus cycles, such as interrupt acknowledge, when paging is enabled, and all bus cycles when paging is disabled. The value and PCD pin output controls caching in an external cache on a cycle-by-cycle basis.

PCI Bus A newer expansion bus design that uses a local bus transfer rate, typically using the microprocessor input clock. It is faster than the conventional ISA bus, and also requires defined storage space for device configuration information.

PCI Device A device that conforms to the PCI specification both in terms of the hardware interface used on the PCI bus, but also by having a set of configuration information stored in the device and accessible by the microprocessor through the PCI space defined by the specification.

Peripheral Device Any input, output, or input/output device connected to a personal computer. Typically, this includes a hard drive, diskette drive, display unit, mouse, trackball, or similar devices.

Physical Address The address on the local bus.

Physical Memory The address space on the local bus; hardware implementation of memory.

Pointers A value that references an address location. See also *Far Pointer* and *Near Pointer*.

Power Good Signal An output signal provided by most computer power supplies to indicate the status of the power provided by the supply. Computer designs typically use the change of state of the Power Good signal that occurs when power is turned on as one of the inputs to the Reset pin on the microprocessor that initiates processor initialization.

Power Management A method of controlling overall power usage in a computer. Interest in power management techniques grew out of the increased market interest in battery operated portable computers (laptops, notebooks, subnotebooks, palmtops, etc.). The need for longer battery life and smaller systems pushed technology to develop ways to shut down power automatically to devices not being used. As the possibilities for power management were becoming realized, the environmental movement began to realize that the same principles could be applied to desktop systems to conserve power on a larger scale. This led to the U. S. Environmental Protection Agency's initiative calling for an "Energy Star" program and the vision of the "green" PC.

Precision The number of bits used by the FPU as the significand of a real number in the floating-point format. The FPU can represent a real number using one of three precision levels: single precision (24 bits), double precision (53 bits), or extended precision (64 bits).

Prefix An optional instruction byte that a programmer can add to the instruction format. The Am486 processor supports four prefix types: OperandSize, AddressSize, Segment Override, Instruction (REP, REPcc, LOCK). The prefixes override the default settings of a specific instruction.

Present Bit A bit in the Task Segment Descriptor that indicates whether the segment is present in memory. The Present Bit allows the system to generate an exception, store the data currently in the segment location, and restore data from a hard drive or other storage device, and then execute the requested task.

Privilege Level A protection value assigned to segments and segment selectors. There are four privilege levels, ranging from 0 (most privileged) to 3 (least privileged).

Programmable Device A device for which you can change the interface characteristics through programming. Typically, the driver software allows you to select a specific interrupt (IRQ) line, DMA channel, and a memory location to shadow the device BIOS. Some bus designs (such as EISA or PCI) include software configuration tables that cross-check the device configurations in a system to reduce the possibility of device conflict and system lockup.

Programmable Register A register whose bit values control the operation or configuration of a device or function, which is accessible and programmable by the user, that is, the user can select specific bit values in the register.

Programming A designed sequence of instruction code that performs a desired function.

Protected Mode An execution mode in which the full 32-bit architecture is available.

Protection A mechanism used to protect the operating system and application programs from execution errors. Protection includes defining the types of address available to a program, the kind of memory references that can be made, and the privilege level required for access. A violation of these limits causes a general protection exception.

PWT Bit *Page-Level Writes Transparent,* bit 3 of CR3. This bit drives the value of the PWT output pin on the microprocessor during unpaged bus cycles, such as interrupt acknowledge, when paging is enabled, and all bus cycles when paging is disabled. The value and PWT pin output controls write-through in an external cache on a cycle-by-cycle basis.

Quadword (qword) 64 bits.

RAM *Random Access Memory.* A type of memory that can be organized within an addressable array so that any memory location can be accessed electronically by address, rather than by any mechanical access method (such as that provided by drives). There are a variety of types of this memory including DRAM (Dynamic RAM), SRAM (Static RAM), and Flash RAM.

Read Only Typically used to describe a register or a protected memory field that can only be read. Some read-only registers are not writable because they control critical operations within a system. Others simply reflect status and do not affect operation. Some read-only registers share an I/O address with a write-only register.

Read/Write A register or memory field that is both readable and writable.

Read/Write (R/W) Bit A bit in the page directory entry or the page table entry that indicates the type of access accorded to the program accessing the pages. This bit is used with the User/Supervisor bit. If the operation is in User Mode (U/S = 1), only pages belonging to the current user have read/write access (R/W = 1). For all other pages, the user has read-only access (R/W = 0).

Real Address Mode A execution mode in which the microprocessor emulates the architecture of an 8086 processor; also called Real or Virtual Mode.

Real Number Any finite value (negative, positive, or zero) that can be represented by a possibly infinite numeric expansion.

Reboot Initialize the system. You can initialize the microprocessor by applying a signal (such as POWER GOOD) to the microprocessor RESET pin.

Register A defined set of bits in a microprocessor or other device, or a defined space within memory. Typically, a register is a set of 8 bits (byte register), 16 bits (word register), or 32 bits (doubleword register), but it can be any length from 1 bit up. Microprocessor registers are addressed by name (or register code value). Other registers require an I/O address for accessing. Typically, registers are assigned names for reference convenience. Individual bit positions and bit sets or fields within the register may also have names.

Requested Privilege Level (RPL) The privilege assigned to a segment selector. If the RPL is less than the CPL, access to a segment occurs at the RPL level. This prevents access to more privileged software by lower privilege applications, protecting operating systems and BIOS software.

Reset System Level: The signal input that causes the microprocessor to reinitialize and go to a known state.

Bit: To force a bit to the 0 level.

Rounding A numerical operation that converts an extended fraction to a fixed length. Rounding control in the microprocessors numeric functions allow the user to select a particular type of rounding control: *Up* toward +∞, *Down* toward -∞, or *Truncate*.

RTC *Real Time Clock.* This circuit supports current day, month, year, day of the week, hour, minute, and second for display and use by the computer system. Because the circuit requires battery support to maintain the current time when the system is turned off, the clock circuits have traditionally been used to store system configuration information required at startup.

SCSI Bus *Small Computer Systems Interface Bus.* An external communications system that allows the interconnection of serval types of external devices to a computer through sets of daisy-chain cables. As many as 256 devices can be interconnected to a single system through daisy-chained SCSI controllers.

SCSI Device A device that has the built-in circuitry and connectors to attach to a SCSI bus. Like IDE drives, SCSI hard drives have built-in controller circuitry as well as the SCSI interface.

Segment An independent, protected address space. A program can access as many as 16,383 segments, each of which can be as large as 4 Gbytes.

Segment Descriptor A 64-bit data structure used for segmentation. It includes the segment base address, its size (limit), its type, and protection information. It is setup by operating system software and accessed by segmentation hardware.

Segment Override Prefix Optional programming code used before an instruction to override the default segment selection. There are six segment override prefixes, one for each segment register.

Segment Register One of six special purpose registers: CS, SS, DS, ES, FS, and GS. The registers store the segment selectors that identify the independent address spaces addressable by a particular program. The CS register defines the address space in memory for code storage. The SS register defines the stack space, that is the area used as temporary storage registers and accessed by the PUSH and POP instructions. The remaining four registers (DS, ES, FS, and GS) define four independent address spaces to store data for program use.

Segment Selector A 16-bit number used to specify an address space (segment). Bits 15–3 are the index into the descriptor table. Bit 2 specifies whether to use the GDT or an LDT. Bits 1–0 define the RPL as an additional protection check.

Segmentation A form of memory management that provides multiple, independent, protected address spaces. Segmentation allows you to define as many as 16383 segments, each of which can be as large as 4 Gbytes.

Serial Interface An interface that uses a single data line. A serial interface transfers one bit at a time.

Serial Port A hardware connector used to transfer data via a serial interface. Typically, computer systems use a 25-pin or 9-pin D-shell connector, but can also use an RJ-11 telephone connector.

SETcc A conditional set byte command. If the condition is met, the specified byte is set to a value of 1. If the condition is not met, the specified byte is 0.

Setup A program that is typically part of BIOS that allows the user to program the system configuration information stored in the CMOS in the RTC.

Shadow Register A device register that contains the same information as another, typically write-only, register. For some power management solutions, designs may include shadow registers to save and restore the contents of write-only registers when normal operation resumes after a system has been in one of the reduced-power modes.

Shadowing Copying information from one source to another. In older personal computer systems, accessing BIOS or other firmware programs from the static memory devices was slower than accessing data from RAM. One solution was to “shadow” the ROM contents into RAM for more efficient computer operation. Shadowing is now also used to preserve the contents of write-only registers; see *Shadow Register*.

s-i-b Byte A byte following an instruction opcode and modR/M bytes that specifies a scale factor, index, and base register.

SIMM *Single In-Line Memory Module.* Sets of memory chips mounted on an easily installed circuit board for quick access and servicing.

SMM *System Management Mode.* A special operational mode accessible only by BIOS and other firmware that is used to develop power management or security support systems for the computer.

SRAM *Static Random Access Memory.* Memory that can be accessed and programmed in a computer system, but does not require power to maintain the last value written to the memory.

Stack A set of consecutive memory locations used as scratch space by application and other programs. The FPU has an internal stack consisting of eight 80-bit registers.

Stack Fault A special case of the invalid-operand exception indicated by the SF bit in the FPU Status Word. It is usually caused by a stack underflow or overflow.

Stack Segment A memory segment used to hold a stack. Only one stack segment is available to the microprocessor at a time, the segment whose descriptor is currently in the SS register. The segment descriptor defines the segment.

Status Word A 16-bit FPU register indicating the current FPU status. It contains the condition codes, the FPU stack pointer, busy and interrupt bits, and exception flags.

String A sequence of bytes, words, or doublewords that may start at any byte address in memory.

Tag Word A 16-bit FPU register that for each stack space in the FPU, tells if that space stores a number and what type number it is.

Task Register (TR) A register that holds the segment selector for the current task. The selector references a task state segment (TSS). Like the segment registers, the TR has a visible part and an invisible part. The visible part holds the segment selector; the invisible part holds information cached from the segment descriptor for the TSS.

Task State Segment (TSS) A segment that stores the processor state during a task switch. If a separate I/O address space is used, the TSS holds permission bits that control access to the I/O space.

Task Switch A transfer of execution between tasks. The TSS saves most of the processor state.

Task A program running (executing).

Test Register One of five registers that provide test and status information of the internal cache functionality of the Am486 microprocessor.

TOP The 3-bit field in the FPU Status Word that indicates which FPU register is at the top of the stack.

Translation Lookaside Buffer (TLB) The on-chip cache for page table entries.

Trap An exception that is reported at the instruction boundary immediately following the instruction that generated the exception.

Trap Gate A gate descriptor used to invoke an exception handler. A trap gate is different from an interrupt gate only in its effect on the IF flag. Unlike an interrupt gate, which clears the flag for the duration of the handler, a trap gate leaves the flag unchanged.

Type Checking One of the five types of protection provided by segmentation. The TYPE field in the system segment descriptor defines the capabilities allowed to a particular program and generates exceptions if the program attempts to perform executions not allowed by the specified TYPE information.

Underflow Numeric: An exception condition in which the correct answer is not zero, but is so small that it cannot be represented as a real number by the floating-point unit.

Stack: An exception caused by trying to read an operand from an empty stack location.

User/Supervisor (U/S) bit A bit in the page directory entry or the page table entry that indicates the level of privilege accorded to the program accessing the pages. An application with a CPL = 3 is assigned a user status (U/S = 1). Any other CPL value is considered supervisory (U/S = 0).

Vcc The typical notation used to indicate the input dc power for computer logical operations. Typically, the voltage is in the range from +3 to +5 volts.

Vector A number used by the microprocessor to access interrupt and exception handlers.

VESA *Video Electronics Standard Association.* Video and local bus standards codified by the association to support development of compatible video components industry-wide.

VGA *Video Graphics Array.* A video controller that supports high resolution color graphic displays. Actual support varies depending on the type and size of the RAMDAC used in the controller and the amount of video memory available.

Video Controller The electronic circuits that provide the interface between a personal computer system and a display unit (CRT, LCD or other display type).

Virtual Mode Similar to Real Mode, the processor emulates the 8086 processor architecture.

VL Bus One of several types of Local Bus design protocols that allow a device to transfer data and code to and from the microprocessor at the same speed as the processor input clock. The clocking rate (typically 25 MHz and higher) is faster than transfer clocking used by the older ISA or EISA bus standard (4.77 MHz to 12 MHz; typically 8 MHz). See also VESA.

Vss The typical notation used to indicate a ground bus pinout in a chip.

Word 16 bits.

Wait States Programmed or hardware-determined delays that allow slow transfer processes to finish their operation. This is typically used with slow hard drives and DMA controllers. Most new microprocessors support Zero Wait State operation, that is without induced processor delay states.

Write Back A form of caching in which memory writes load only the cache memory. Data is transferred to the system memory before a memory read accesses the location.

Write Only A register in a microprocessor or controller that can be loaded with a write command, but which cannot be read by a read command. Typically, this register shares an I/O address location with a read-only register.

Write Through A form of caching in which memory writes load both the cache and system memory.

ASCII Codes (based on ANSI x3.4 1968)

Hex	**Character** (Control Kybd Equiv.)	Hex	Character	Hex	Character	Hex	Character
00	NUL (@)	20	SP	40	@	60	‘
01	SOH (A)	21	!	41	A	61	a
02	STX (B)	22	“	42	B	62	b
03	ETX (C)	23	#	43	C	63	c
04	EOT (D)	24	$	44	D	64	d
05	ENQ (E)	25	%	45	E	65	e
06	ACK (F)	26	&	46	F	66	f
07	BEL (G)	27	‘	47	G	67	g
08	BS (H)	28	(	48	H	68	h
09	HT (I)	29	)	49	I	69	i
0A	LF (J)	2A	*	4A	J	6A	j
0B	VT (K)	2B	+	4B	K	6B	k
0C	FF (L)	2C	,	4C	L	6C	l
0D	CR (M)	2D	-	4D	M	6D	m
0E	SO (N)	2E	.	4E	N	6E	n
0F	SI (O)	2F	/	4F	O	6F	o
10	DLE (P)	30	0	50	P	70	p
11	DC1 (Q)	31	1	51	Q	71	q
12	DC2 (R)	32	2	52	R	72	r
13	DC3 (S)	33	3	53	S	73	s
14	DC4 (T)	34	4	54	T	74	t
15	NAK (U)	35	5	55	U	75	u
16	SYN (V)	36	6	56	V	76	v
17	ETB (W)	37	7	57	W	77	w
18	CAN (X)	38	8	58	X	78	x
19	EM (Y)	39	9	59	Y	79	y
1A	SUB (Z)	3A	:	5A	Z	7A	z
1B	ESC ([)	3B	;	5B	[	7B	{
1C	FS (\)	3C	<	5C	\	7C	|
1D	GS (])	3D	=	5D	]	7D	}
1E	RS (^)	3E	>	5E	^	7E	~
1F	US (_)	3F	?	5F	_	7F	DEL