VLSI TECHNOLOGY, INC.

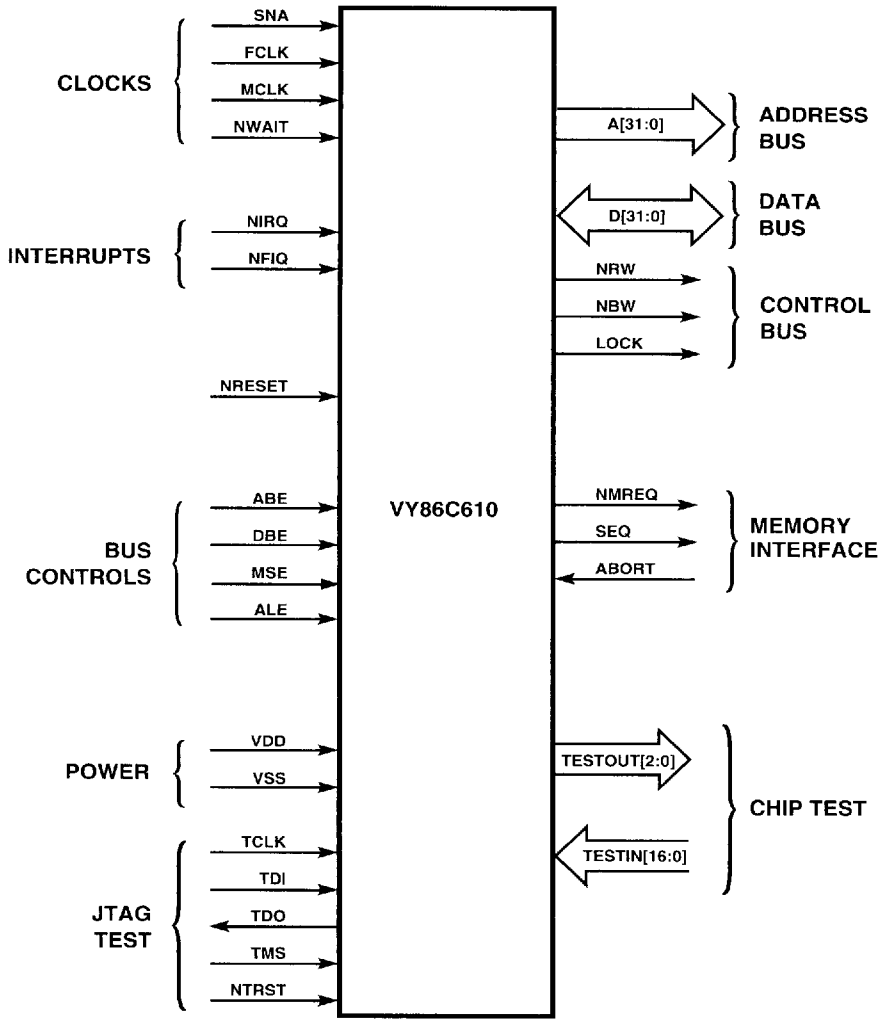
# VY86C610

## 32-BIT ENHANCED MICROPROCESSOR

## FEATURES

- 32-bit data bus
- 32-bit data address bus
- Simple but powerful instruction set
- On-chip Memory Management Unit (MMU)
- On-chip 32-entry Translation Look-aside Buffer (TLB)
- On-chip 4 Kbyte Cache
- On-chip Write Buffer
- Fully static
- Low power
- Cache access speed independent of external memory timing
- IEEE 1149.1-1990 Boundary Scan Standard Test Port
- C compiler support
- 25-MHz cache clock
- 12-MHz bus clock
- 144-lead Quad Flat Pack (QFP) or Thin Quad Flat Pack (TQFP)

## FUNCTIONAL DIAGRAM




## DESCRIPTION

VY86C610 is a general-purpose 32-bit microprocessor with 4 Kbyte cache, Write Buffer, and Memory Management Unit (MMU) combined in a single chip. It is software compatible with the ARM™ family and can be used with the existing ARM support chips.

The VY86C610 architecture is based on "Reduced Instruction Set Computer" (RISC) principles, and the instruction set and related decode mechanism are greatly simplified compared with micro-programmed "Complex Instruction Set Computer" (CISC).

The on-chip mixed data and instruction cache substantially raises the average execution speed, and reduces the average amount of memory bandwidth required by the processor. This allows the external memory to support additional processors or Direct Memory Access (DMA) channels with minimal performance loss.

The instruction set comprises ten basic instruction types. Two of these make use of the on-chip arithmetic logic unit, barrel shifter, and multiplier to perform high-speed operations on the data in a bank of 31 registers, each 32 bits wide. Three classes of instruction control the transfer of data between main memory and the register bank, one optimized for flexibility of addressing, another for rapid context switching and the third for swapping data. Two instructions control the flow and privilege level of execution, and another three types are dedicated to the control of external coprocessors that allow the functionality of the instruction set to be extended off-chip in an open and uniform way.

The ARM instruction set has proved to be a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors that depend on sophisticated compiler technology to manage complicated instruction interdependencies.


VLSI Technology, Inc. • 1109 McKay Drive San Jose, • CA 95131 408-434-3100

Pipelining is employed so that all parts of the processor and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched.

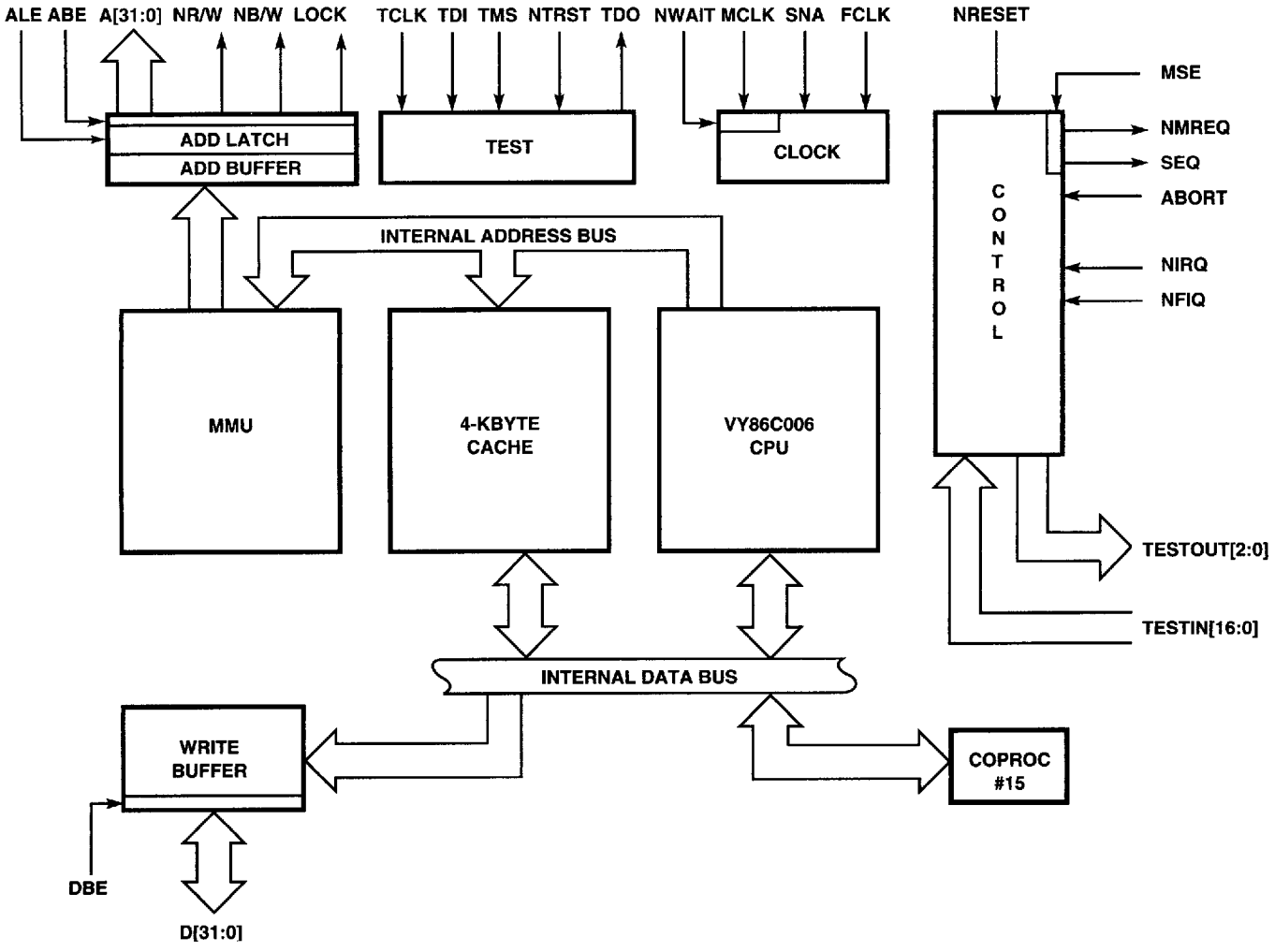
The memory interface has been designed to allow the performance potential to be realized without incurring high costs in the memory system. Speed-critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic. These control signals facilitate the exploitation of the fast local access modes offered by industry-standard dynamic random access memories (DRAMs).

The MMU supports a conventional two-level page-table structure and a number of extensions which make it ideal for embedded control, UNIX™, and object-oriented systems.

The VY86C610 is a fully static and has been designed to minimize its power requirements. This makes it ideal for portable applications where these features are essential.

The VY86C610 is a variant of the VY86C600. The external coprocessor interface has been removed, but one internal coprocessor (#15) is retained for device control. The ALE (Addess Latch Enable) signal has been added to allow operation in systems with static RAM and ROM without external latches.

## BLOCK DIAGRAM



## SIGNAL DESCRIPTIONS

| Name | Pin | Type | Description |
|---|---|---|---|
| A[31:0] | 116-114, 111-104, 101-94, 89-80, 77-75 | OCZ | Address bus. This bus signals the address requested for memory access It changes during MCLK HIGH. |
| ABE | 74 | IT | Address bus enable. When this input is LOW, the address bus A[31:0] is put into a high impedance state (Note 1). |
| ABORT | 138 | IT | External abort. This input allows the memory system to tell the processor that a requested access has failed. This input is only monitored when the the VY86C610 is accessing external memory. |
| ALE | 117 | IT | Address latch enable. This input is used to control transparent latches on the address bus A[31:0], NBW, NRW, and LOCK. Normally these signals change during MCLK HIGH, but they may be held by driving ALE LOW. |
| D[31:0} | 47–43, 39-33 | ITOTZ | Data bus. These are bidirectional signal paths which are used for data transfers between the processor and external memory. |
| | 30-20 | | For read operations (when NRW is LOW), the input data must be valid before the falling edge of MCLK. |
| | 15-7 | | For write operations (when NRW is HIGH), the output data will become valid while MCLK is LOW. |
| DBE | 4 | IT | Data bus enable. When this input is LOW, the data bus, D[31:0] is put into a high impedance state (Note 1). The drivers will always be high impedance except during write operations, and DBE must be driven HIGH in systems that do not require the data bus for DMA or similar activities. |
| FCLK | 139 | IT | Fast clock input. When the VY86C610 CPU is accessing the cache, performing an internal cycle, or communicating directly with the coprocessor, it is clocked with the Fast clock, FCLK. |
| LOCK | 73 | OCZ | Locked operation. LOCK is driven HIGH, to signal a "locked" memory access sequence, and the memory manager should wait until LOCK goes LOW before allowing another device to access the memory. LOCK changes while MCLK is HIGH, and remains HIGH for the duration of the locked memory sequence. |
| MCLK | 140 | IT | Memory clock input. This clock times all VY86C610 memory accesses. The LOW or HIGH period of MCLK may be stretched when accessing slow peripherals; alternatively, the NWAIT input may be used with a free-running MCLK to achieve similar effects. |
| MSE | 1 | IT | Memory request/sequential enable. When this input is LOW, the following control outputs are put into a high impedance state[1]. NMREQ, SEQ. |
| NB/W | 72 | OCZ | Not byte/word. An output signal used by the processor to indicate to the external memory system when a data transfer of a byte length is required. NBW is HIGH for word transfers and LOW for byte transfers, and is valid for both read and write operations. |
| NFIQ | 131 | IT | Not fast interrupt request. If FIQs are enabled, the processor will respond to a LOW level on this input by taking the FIQ interrupt exception. This is an asynchronous, level-sensitive input, and must be held LOW until a suitable response is received from the processor. |
| NIRQ | 132 | IT | Not interrupt request. As NFIQ, but with lower priority. May be taken LOW asynchronously to interrupt the processor when the IRQ enable is active. |
| NMREQ | 2 | OCZ | Not memory request. This is a pipelined signal that changes while MCLK is LOW to indicate whether the following cycle will be Active (processor accessing external memory) or Latent (processor not accessing external memory). An Active cycle is flagged when NMREQ is LOW. |

**Note:**

1. When output pads are placed in the high impedance state for long periods, care must be taken to ensure that they do not float to an undefined logic level, as this can dissipate power, especially in the pads.

## SIGNAL DESCRIPTIONS (Cont.)

| Name | Pin | Type | Description |
|---|---|---|---|
| NRESET | 137 | IT | Not reset. This is a level-sensitive input signal that is used to start the processor from a known address. A LOW level will cause the instruction being executed to terminate abnormally, and the on-chip caches, MMU, and write buffer to be disabled. When NRESET is driven HIGH, the processor will restart from address 0. NRESET must remain LOW (and NWAZT must remain HIGH) for at least two FLCK clock cycles, and five MCLK clock cycles. During the LOW period, the processor will perform idle cycles but with incrementing addresses. |
| NR/W | 71 | OCZ | Not read/write. When HIGH, this signal indicates a processor write operation; when LOW, a read operation. The signal changes while MCLK is HIGH. |
| NTRST | 50 | IT | Test interface reset. Note this pin does NOT have an internal pull-up resistor. |
| NWAIT | 143 | IT | Not wait. When LOW, this signal allows extra MCLK cycles to be inserted into memory accesses. It must change during the LOW phase of the MCLK cycle that is to be extended. |
| SEQ | 3 | OCZ | Sequential address. This signal is the inverse of NMREQ, and is provided for compatibility with existing ARM memory systems. |
| SNA | 144 | IT | Synchronous / not Asynchronous. This pin determines the bus interface mode, and should be hardwired HIGH or LOW, depending on the relationship between FCLK and MCLK in the application. |
| TCK | 53 | IT | Test interface reference Clock. Provides timing for all transfers on the test interface. |
| TDI | 49 | IT | Test interface data input |
| TDO | 48 | OCZ | Test interface data output |
| TESTIN[16:0] | 136,<br>123-130,<br>68-63,<br>60-59 | IT | Test bus input. This bus is used for off-board testing of the device. When the device is fitted to a circuit all these pins must be tied LOW. The signals are further defined within the chip as: TESTIN[16] tsidc<br>TESTIN[7:0]<br>TESTIN[15:10]<br>TESTIN[9:8] |
| TESTOUT[2:0] | 135-133 | OCZ | Test bus ouput. This bus is used for off-board testing of the device. When the device is fitted to a circuit, and all the TESTIN[16:0] pins are driven LOW, these three outputs will be three-stated, except via the JTAG test port. |
| TMS | 52 | IT | Test interface mode select |
| VDD1 | 19, 61, 92,<br>122, 51 | | Positive Supply |
| VDD2 | 6, 17, 31, 70,<br>79, 90,102,<br>112, 141 | | Positive Supply |
| VSS1 | 18, 40, 62,<br>91,121 | | Ground Supply |
| VSS2 | 5, 16, 32, 41,<br>69, 78, 93,<br>103, 142, 113 | | Ground Supply |

## PROGRAMMER'S MODEL

### INTRODUCTION

The VY86C610 has a 32-bit data bus and a 32-bit address bus. The data types the processor supports are: Bytes (8 bits) and Words (32 bits), where words must be aligned to four-byte boundaries. Instructions are exactly one word, and data operations (e.g., ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words.

The VY86C610 supports six modes of operation:

(1) User mode: the normal program execution state

(2) FIQ mode (fiq): designed to support a data transfer or channel process

(3) IRQ mode (irq): used for general-purpose interrupt handling

(4) Supervisor mode (scv): a protected mode for the operating system

(5) Abort mode (abt): Entered after a data or instruction prefetch abort

(6) Undefined mode (und): entered when an undefined instruction is executed.

Mode changes may be made under software control or may be brought about by external interrupts or exception processing. Most application programs will execute in User mode. The other modes, known as privileged modes, will be entered to service interrupts, exceptions, or to access protected resources.

### REGISTERS

The processor has a total of 37 registers made up of 31 general-purpose 32-bit registers and six status registers. At any one time, 16 general-purpose registers (R0 to R15) and one or two status registers are visible to the programmer. The visible registers depend on the processor's mode of operation. The other registers (the banked registers) are switched in to support IRQ, FIQ, Supervisor, Abort, and Undefined mode processing. The register bank organization is shown below. The banked registers are shaded in the diagram.

In all modes, 16 registers, R0 to R15, are directly accessible. All registers, except R15, are general-purpose and may be used to hold data or address values. Register R15 holds the Program Counter (PC). When R15 is read, bits [1:0] are zero and bits [31:2] contain the PC. A seventeenth register (the CPSR – Current Program Status Register) is also accessible. It contains condition code flags and the current mode bits and may be thought of as an extension to the PC.

R14 is used as the subroutine link register and receives a copy of R15 when a Branch and Link instruction is executed. It may be treated as a general-purpose register at all other times. R14_svc, R14_irq, R14_fiq, R14_abt, and R14_und are used similarly to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within interrupt or exception routines.

## REGISTER ORGANIZATION

### GENERAL REGISTERS AND PROGRAM COUNTER

| User32 Mode | FIQ32 Mode | Supervisor32 Mode | Abort32 Mode | IRQ32 Mode | Undefined32 Mode |
|---|---|---|---|---|---|
| R0 | R0 | R0 | R0 | R0 | R0 |
| R1 | R1 | R1 | R1 | R1 | R1 |
| R2 | R2 | R2 | R2 | R2 | R2 |
| R3 | R3 | R3 | R3 | R3 | R3 |
| R4 | R4 | R4 | R4 | R4 | R4 |
| R5 | R5 | R5 | R5 | R5 | R5 |
| R6 | R6 | R6 | R6 | R6 | R6 |
| R7 | R7 | R7 | R7 | R7 | R7 |
| R8 | R8_fiq | R8 | R8 | R8 | R8 |
| R9 | R9_fiq | R9 | R9 | R9 | R9 |
| R10 | R10_fiq | R10 | R10 | R10 | R10 |
| R11 | R11_fiq | R11 | R11 | R11 | R11 |
| R12 | R12_fiq | R12 | R12 | R12 | R12 |
| R13 | R13_fiq | R13_svc | R13_abt | R13_irq | R13_undef |
| R14 | R14_fiq | R14_svc | R14_abt | R14_irq | R14_undef |
| R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) | R15 (PC) |

### PROGRAM STATUS REGISTERS

| | | | | | |
|---|---|---|---|---|---|
| CPRS | CPRS | CPRS | CPRS | CPRS | CPRS |
| | SPSR_fiq | SPSR_svc | SPSR_abt | SPSR_irq | SPSR_undef |

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). Many FIQ programs will not need to save any registers.

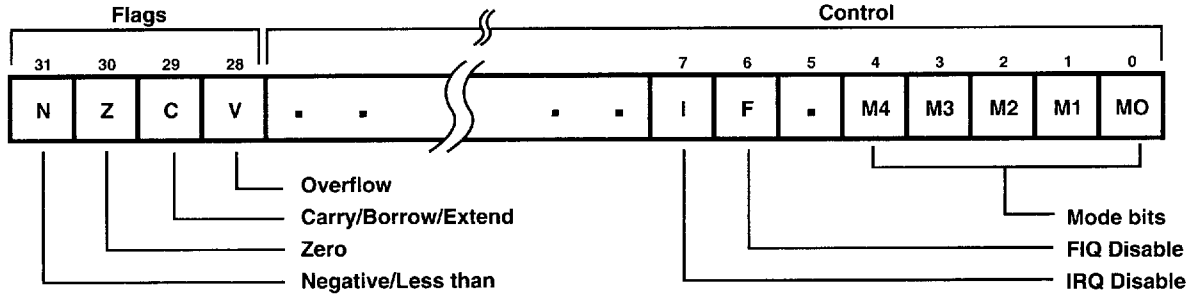
User mode, IRQ mode, Supervisor mode, Abort mode, and Undefined mode each have two banked registers mapped to R13 and R14. The two banked registers allow these modes to each have a private stack pointer and link register. Supervisor, IRQ, Abort, and Undefined mode programs which require more than these two banked registers are expected to save some or all of the caller's registers (R0 to R12) on their respective stacks. They are then free to use these registers which they will restore before returning to the caller. In addition, there are also five SPSRs (Saved Program Status Registers) that are loaded with the CPSR when an exception occurs. There is one SPSR for each privileged mode.

The format of the Program Status Registers is shown below. The N, Z, C and V bits are the *condition code flags*. The condition code flags in the CPSR may be changed as a result of arithmetic and logical operations in the processor and may be tested by all instructions to determine if the instruction is to be executed.

The I and F bits are the *interrupt disable bits*. The I bit disables IRQ interrupts when it is set and the F bit disables FIQ interrupts when it is set. The M0, M1, M2, M3 and M4 bits (M[4:0]) are the *mode bits*, and these determine the mode in which the processor operates. The interpretation of the mode bits is shown below. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described shall be used.

The bottom 28 bits of a PSR (incorporating I, F and M[4:0]) are known collectively as the *control bits*. The control bits will change when an exception arises, and can be manipulated by software when the processor is in a privileged mode. Unused bits in the PSRs are reserved and their state shall be preserved when changing the flag or control bits. Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

## FORMAT OF THE PROGRAM STATUS REGISTERS (PSRS)




## MODE BITS

| M[4:0] | Mode | Accessible Register Set | |
|---|---|---|---|
| 10000 | usr | PC, R14.. R0 | CPSR |
| 10001 | fiq | PC, R14_fiq..R8_fiq, R7..R0 | CPSR, SPSR_fiq |
| 10010 | irq | PC, R14_irq..R13_irq, | R12..R0 CPSR, SPSR_irq |
| 10011 | svc | PC, R14_svc..R13_svc, | R12..R0 CPSR, SPSR_svc |
| 10111 | abt | PC, R14_abt..R13_abt, | R12..R0 CPSR, SPSR_abt |
| 11011 | und | PC, R14_und..R13_und, | R12..R0 CPSR, SPSR_und |

## EXCEPTIONS

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that (for example) the processor can be diverted to handle an interrupt from a peripheral. The processor state just prior to handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. Many exceptions may arise at the same time.

The VY86C610 handles exceptions by making use of the banked registers to save state. The old PC and CPSR contents are copied into the appropriate R14 and SPSR and the PC and mode bits in the CPSR bits are forced to a value that depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before re-enabling the interrupt; when transferring the SPSR register to and from a stack, it is important to transfer the whole 32-bit value, and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled (see Exception Priorities, page 9).

### FIQ

The FIQ (Fast Interrupt request) exception is externally generated by taking the NFIQ input LOW. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronization before it can affect the processor execution flow. It is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications (thus minimizing the overhead of context switching). The FIQ exception may be disabled by setting the F flag in the CPSR (but note that this is not possible from User mode). If the F flag is clear, the VY86C610 checks for a LOW level on the output of the FIQ synchronizer at the end of each instruction.

When a FIQ is detected, the VY86C610 performs the following:

(1) Saves the address of the next instruction to be executed plus 4 in R14_fiq; saves CPSR in SPSR_fiq

(2) Forces M[4:0]=%10001 (FIQ mode) and sets the F and I bits in the CPSR

(3) Forces the PC to fetch the next instruction from address &1C

To return normally from FIQ, use

SUBS PC, R14_fiq,#4

which will restore both the PC (from R14) and the CPSR (from SPSR_fiq) and resume execution of the interrupted code.

### IRQ

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the NIRQ input. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the PC (but note that this is not possible from User mode). If the I flag is clear, the VY86C610 checks for a LOW level on the output of the IRQ synchronizer at the end of each instruction.

When an IRQ is detected, the VY86C610 performs the following:

(1) Saves the address of the next instruction to be executed plus 8 in R14_irq; saves CPSR in SPSR_irq

(2) Forces M[4:0]=%10010 (IRQ mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address &18

To return normally from IRQ, use

SUBS PC,R14_irq, #4

which will restore both the PC and the CPSR and resume execution of the interrupted code.

### Abort

The Abort signal comes from an external memory management system, and indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. The VY86C610 checks for an Abort during memory access (N and S) cycles and distinguishes between two types of aborts.

(1) If the abort occurred during an instruction prefetch (a Prefetch Abort), the prefetched instruction is marked as invalid but the abort exception does not occur immediately. If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, no abort will occur. An abort will take place if the instruction reaches the head of the pipeline and is about to be executed.

(2) If the abort occurred during a data access (a Data Abort), the action depends on the instruction type.

  (a) Data transfer instructions (LDR, STR) are aborted as though the instruction had not executed if the processor is configured for Early Abort. When configured for Late Abort, these instructions are able to write-back modified base registers and the Abort handler must be aware of this.

  (b) The swap (SWP) instruction is aborted as though it had not executed.

  (c) LDM and STM instructions complete, and if write-back is set, the base is updated. If the instruction would normally have overwritten the base with data (i.e. LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

When either a prefetch or data abort occurs, the VY86C610 performs the following:

(1) Saves the address of the aborted instruction plus 4 (for Prefetch Aborts) or 8 (for Data Aborts) in R14_abt; saves CPSR in SPSR_abt

(2) Forces M[4:0]=%10111 (Abort Mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from either address &0C (Prefetch Abort) or address &10 (Data Abort)

To return after fixing the reason for the abort, use SUBS PC,R14_abt,#4 (for a Prefetch Abort) or SUBS PC,R14_abt, #8 (for a Data Abort). This will restore both the PC and the CPSR and retry the aborted instruction.

The abort mechanism allows a demand paged virtual memory system to be implemented when a suitable memory manager is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable the memory manager signals an abort. The processor traps into system software that must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

**Software Interrupt**

The software interrupt instruction (SWI) is used for getting into Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, the VY86C610 performs the following:

(1) Saves the addresses of the SWI instruction plus 4 in R14_svc; saves CPSR in SPSR_svc

(2) Forces M[4:0]=%10011 (Supervisor mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address &08

To return from a SWI, use MOVS PC,R14_svc. This will restore the PC and CPSR and return to the instruction following the SWI.

**Undefined Instruction Trap**

When the VY86C610 executes a coprocessor instruction or an undefined instruction, it offers it to any coprocessors which may be present. If a coprocessor signals that it can perform this instruction but is busy at that moment, the VY86C610 will wait until the coprocessor is ready. If no coprocessor can handle the instruction, the VY86C610 will take the undefined instruction trap.

The trap may be used for software emulation of a coprocessor in a system which does not have the coprocessor hardware, or for general-purpose instruction set extension by software emulation.

When the VY86C610 takes the undefined instruction trap it performs the following:

(1) Saves the address of the Undefined or coprocessor instruction plus four in R14_und; saves CPSR in SPSR_und

(2) Forces M[4:0]=%11011 (Undefined mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address &04

To return from this trap after emulating the failed instruction, use MOVS PC,R14_und. This will restore the CPSR and return to the instruction following the undefined instruction.

**Reset**

When the NRESET signal goes LOW, the VY86C610 abandons the currently executing instruction and then continues to fetch instructions from memory that it interprets as NOPs.

When NRESET goes HIGH again, the VY86C610 does the following:

(1) Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and CPSR is not defined

(2) Forces M[4:0]=%10011 (Supervisor mode) and sets the I and F bits in the CPSR

(3) Forces the PC to fetch the next instruction from address &00

## VECTOR SUMMARY

| Address | Exception | Mode on Entry |
|---|---|---|
| &00000000 | Reset | Supervisor |
| &00000004 | Undefined instruction | Undefined |
| &00000008 | Software interrupt | Supervisor |
| &0000000C | Abort (prefetch) | Abort |
| &00000010 | Abort (data) | Abort |
| &00000014 | —Reserved— | — |
| &00000018 | IRQ | IRQ |
| &0000001C | FIQ | FIQ |

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine. The FIQ routine might reside at &1C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

The reserved entry is for an Address Exception vector that is only operative when the processor is configured for a 26-bit program space.

**Exception Priorities**
When multiple exceptions arise at the same time, a fixed priority system determines the order in which they will be handled:

(1) Reset (highest priority)

(2) Data abort

(3) FIQ

(4) IRQ

(5) Prefetch abort

(6) Undefined Instruction, Software interrupt (lowest priority)

Note that not all exceptions can occur at once. Undefined instruction and software interrupt are mutually exclusive since they each correspond to particular (non-overlapping) decodings of the current instruction.

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (i.e., the F flag in the CPSR is clear), the VY86C610 will enter the data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing the data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection; the time for this exception entry should be added to worst-case FIQ latency calculations.

**Interrupt Latencies**
The worst-case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchronizer *(Tsyncmax)*, plus the time for the longest instruction to complete *(Tldm*, the longest instruction is load multiple registers), plus the time for address exception or data abort entry *(Texc)*, plus the time for FIQ entry *(Tfiq)*. At the end of this time the VY86C610 will be executing the instruction at &1C.

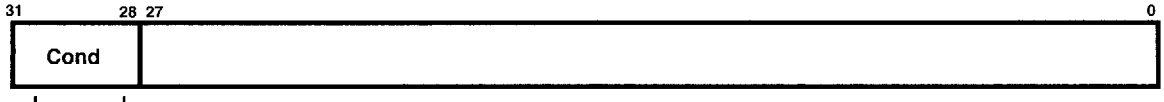
*Tsyncmax* is three processor cycles, *Tldm* is 20 cycles, *Texc* is three cycles, and *Tfiq* is two cycles. The total time is therefore 28 processor cycles, which is just over one microsecond in a system that uses a continuous 25-MHz processor clock. In a DRAM-based system running at 4 and 8 MHz, this time becomes 4.5 microseconds. If bus bandwidth is being used to support video or other DMA activity, the time will increase accordingly.

The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time.

The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchronizer *(Tsyncmin)* plus *Tfiq*. This is four processor cycles.

## INSTRUCTION SET

### CONDITION FIELD



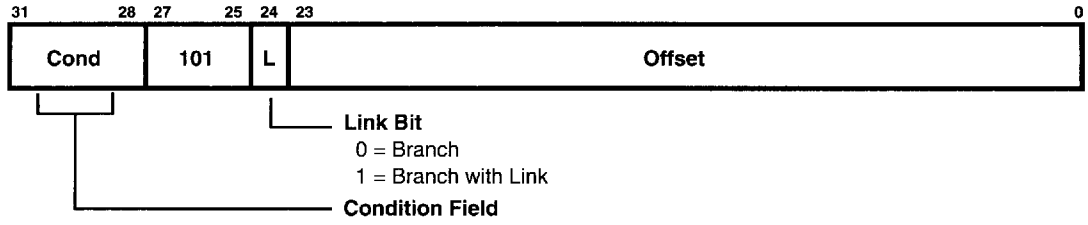

**Condition Field**
0000 = EQ – Z set (equal)
0001 = NE – Z clear (not equal)
0010 = CS – C set (unsigned higher or same)
0011 = CC – C clear (unsigned lower)
0100 = MI – N set (negative)
0101 = PL – N clear (positive or zero)
0110 = VS – V set (overflow)
0111 = VC – V clear (no overflow)
1000 = HI – C set and Z clear (unsigned higher)
1001 = LS – C clear or Z set (unsigned lower or same)
1010 = GE – N set and V set, or N clear and V clear (greater or equal)
1011 = LT – N set and V clear, or N clear and V set (less than)
1100 = GT – Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE – Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL – always
1111 = NV – never

All VY86C610 instructions are conditionally executed (their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR).

If the ALways condition is specified, the instruction will be executed irrespective of the flags. The NeVer class of condition codes should not be used as they will be redefined in future variants of the ARM architecture. If a NOP is required it is suggested that MOV R0, R0 be used.

The other condition codes have meanings as detailed above. For instance, code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction will not be executed.

## BRANCH AND BRANCH WITH LINK (B, BL)




The instruction is only executed if the condition specified in the condition field is true.

Branch instructions contain a signed two's-complement 24-bit offset. This is shifted left two bits, sign extended to 32-bits, and added to the PC. The instruction can therefore specify a branch of +/– 32 Mbytes. The branch offset must take into account the prefetch operation, which causes the PC to be two words (8 bytes) ahead of the current instruction.

Branches beyond +/– 32 Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case, the PC should be manually saved in R14 if a Branch with Link type operation is required.

**Link Bit**

Branch with Link writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC.

To return from a routine called by Branch with Link, use MOV PC, R14 if the link register is still valid or LDM Rn!, {..PC} if the link register has been saved onto a stack pointed to by Rn.

**Assembler Syntax**

B{L}{cond} <expression>

{L} is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.

{cond} is a two-character mnemonic as shown in the Condition Field (EQ, NE, VS, etc.). If absent then AL (ALways) will be used.

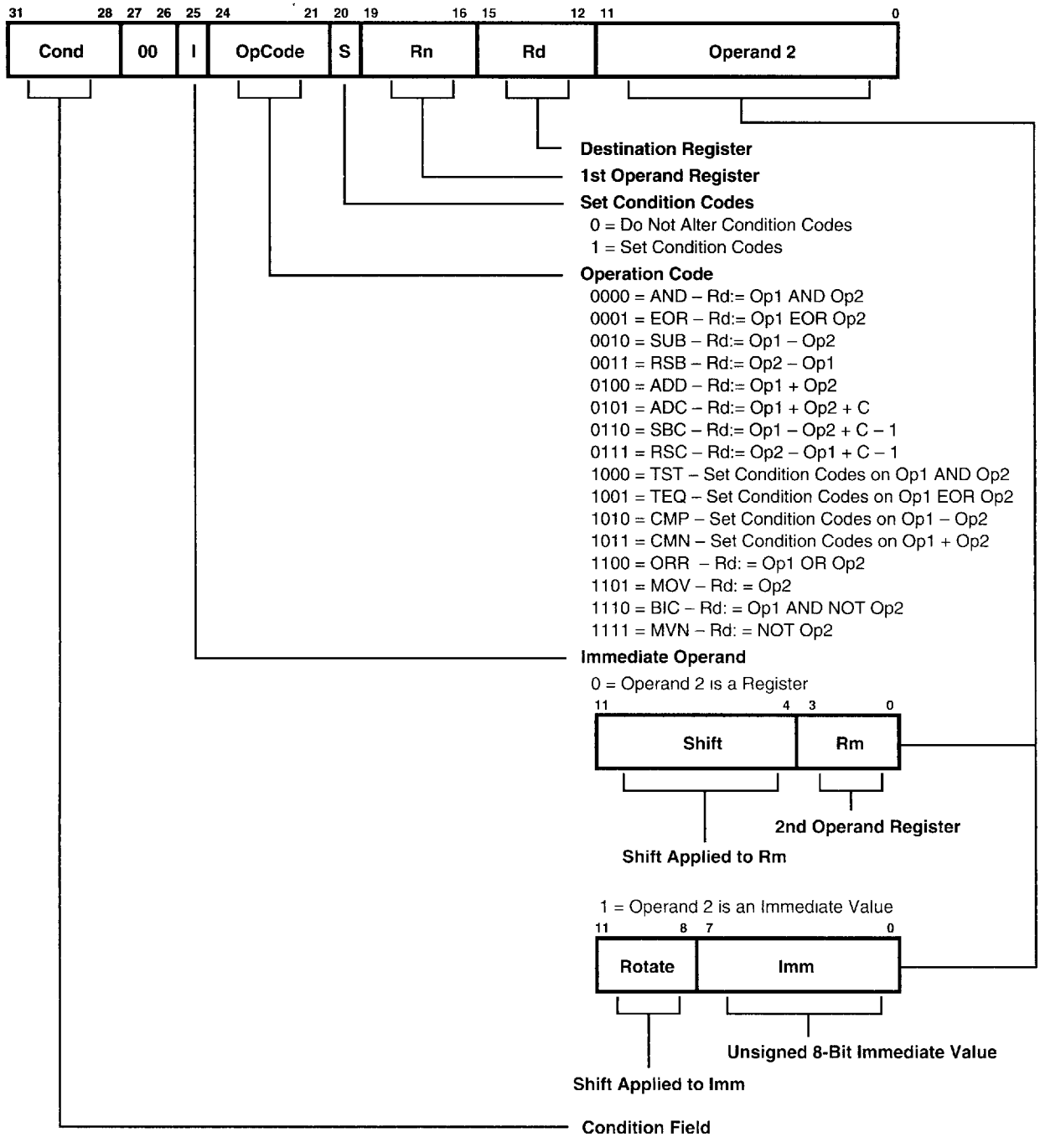
<expression> is the destination. The assembler calculates the offset.

Items in {} are optional. Items in <> must be present.

**Examples**

```
here BAL here       ; assembles to &EAFFFFFE (note effect of PC offset)
B there             ; ALways condition used as default
CMP R1,#0           ; compare R1 with zero and branch to fred if R1
BEQ fred            ; was zero otherwise continue to next instruction
BL sub + ROM        ; call subroutine at computed address
ADDS R1,#1          ; add 1 to register 1, setting CPSR flags on the
BLCC sub            ; result then call subroutine if the C flag is clear,
                    ; which will be the case unless R1 held &FFFFFFFF
```

## DATA PROCESSING




The instruction is only executed if the condition specified in the condition field is true.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn).

The second operand may be a shifted register (Rm) or a rotated 8-bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set.

### Operations

The operations supported are:

**ASSEMBLER**

| Mnemonic | OpCode | Action |
|---|---|---|
| AND | 0000 | operand1 AND operand2 |
| EOR | 0001 | operand1 EOR operand2 |
| SUB | 0010 | operand1 – operand2 |
| RSB | 0011 | operand2 – operand1 |
| ADD | 0100 | operand1 + operand2 |
| ADC | 0101 | operand1 + operand2 + carry (CPSR C flag) |
| SBC | 0110 | operand1 – operand2 + carry – 1 |
| RSC | 0111 | operand2 – operand1 + carry – 1 |
| TST | 1000 | as AND, but result is not written |
| TEQ | 1001 | as EOR, but result is not written |
| CMP | 1010 | as SUB, but result is not written |
| CMN | 1011 | as ADD, but result is not written |
| ORR | 1100 | operand1 OR operand2 |
| MOV | 1101 | operand2 (operand1 is ignored) |
| BIC | 1110 | operand1 AND NOT operand2 (bit clear) |
| MVN | 1111 | NOT operand2 (operand1 is ignored) |

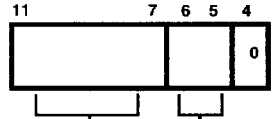
### CPSR Flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15), the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of Bit 31 of the result.

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or two's-complement signed, the two are equivalent). If the S bit is set (and Rd is not R15), the V flag in the CPSR will be set if an overflow occurs into Bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were two's-complement signed. The C flag will be set to the carry out of Bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of Bit 31 of the result (indicating a negative result if the operands are considered to be two's-complement signed).

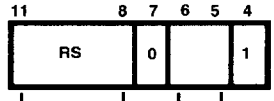
### Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15):




**Shift Type**
00 = Logical Left
01 = Logical Right
10 = Arithmetic Right
11 = Rotate Right

**Shift Amount**
5-bit Unsigned Integer

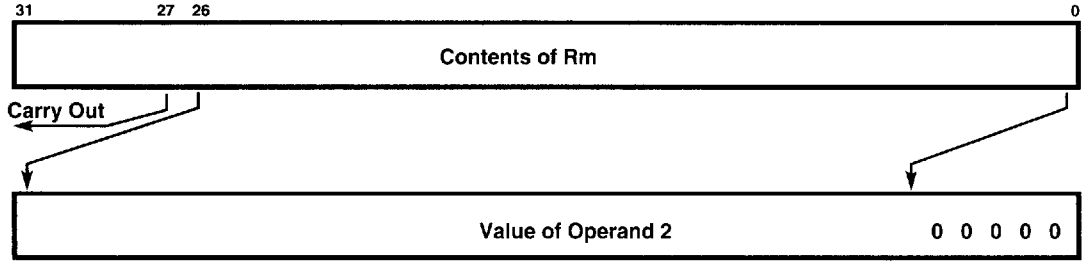



**Shift Type**
00 = Logical Left
01 = Logical Right
10 = Arithmetic Right
11 = Rotate Right

**Shift Register**
Shift amount specified in bottom byte of Rs

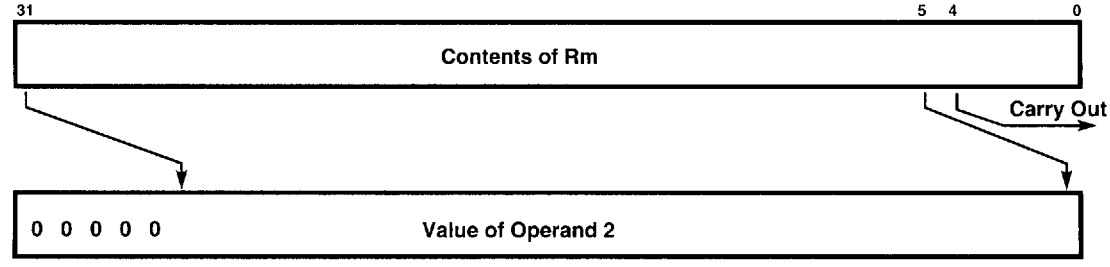
**INSTRUCTION SPECIFIED SHIFT AMOUNT**

When the shift amount is specified in the instruction, it is contained in a 5-bit field that may take any value from 0 to 31. A Logical Shift Left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm that do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class. For example, the effect of LSL #5 is:



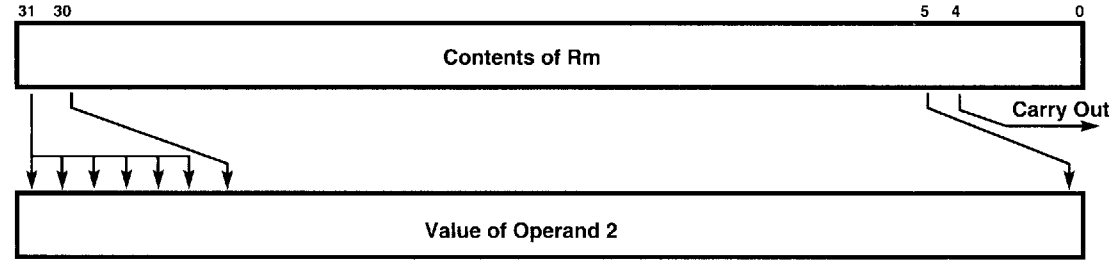

Note that LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has this effect:



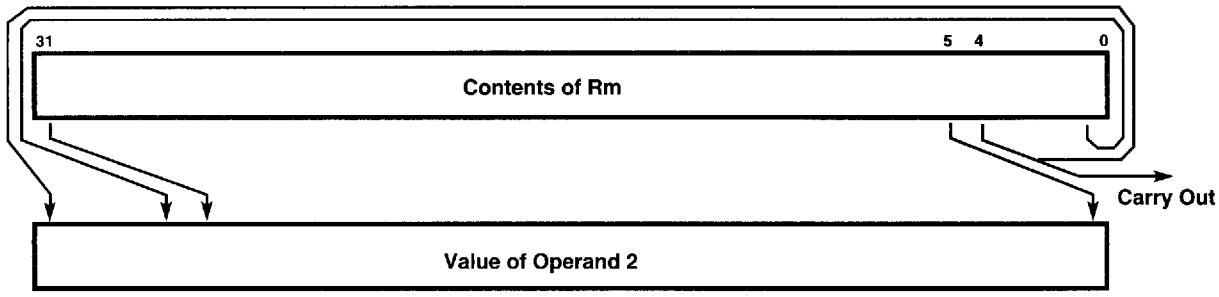

The form of the shift field that might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with Bit 31 of Rm as the carry output. Logical Shift Right zero is redundant as it is the same as Logical Shift Left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An Arithmetic Shift Right (ASR) is similar to Logical Shift Right, except that the high bits are filled with Bit 31 of Rm instead of zeros. This preserves the sign in two's-complement notation. For example, ASR #5:



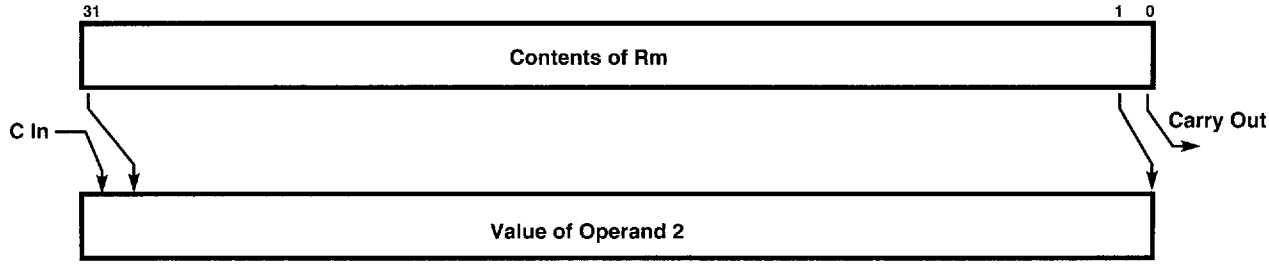
The form of the shift field that might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of Operand 2 is also equal to Bit 31 of Rm. The result is therefore all-ones or all zeros, according to the value of Bit 31 of Rm.

Rotate Right (ROR) operations reuse the bits which 'overshoot' in a Logical Shift Right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5:




The form of the shift field that might be expected to give ROR #0 is used to encode a special function of the barrel shifter, Rotate Right Extended (RRX).

This is a rotate right by one bit position of the 33-bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm:




**REGISTER SPECIFIED SHIFT AMOUNT**

Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output.

If the byte has a value between one and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shifting processes described above:

(i) LSL by 32 has result zero, carry out equal to Bit 0 of Rm.

(ii) LSL by more than 32 has result zero, carry out zero.

(iii) LSR by 32 has result zero, carry out equal to Bit 31 of Rm.

(iv) LSR by more than 32 has result zero, carry out zero.

(v) ASR by 32 or more has result filled with and carry out equal to Bit 31 of Rm.

(vi) ROR by 32 has result equal to Rm, carry out equal to Bit 31 of Rm.

(vii) ROR by n, where n is greater than 32, will give the same result and carry out as ROR by n-32; there fore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note that the zero in Bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or a data swap instruction.

**Immediate Operand Rotates**
The immediate operand rotate field is a four-bit unsigned integer which specifies a shift operation on the eight-bit immediate value. The immediate value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example: all powers of two.

**Writing to R15**
When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags.

When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which automatically restore both PC and CPSR. This form of instruction shall not be used in User mode.

**Using R15 as an Operand**
If R15 (the PC) is used as an operand in a data processing instruction the register is used directly.

The PC value will be the address of the instruction, plus eight or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be eight bytes ahead. If a register is used to specify the shift amount, the PC will be 12 bytes ahead.

**TEQ, TST, CMP and CMN Opcodes**
These instructions do not write the result of their operation but do set flags in the CPSR. An assembler shall always set the S flag for these instructions even if it is not specified in the mnemonic.

The TEQP form of the instruction used in earlier processors shall not be used in the 32-bit modes, the PSR transfer operations should be used instead. If used in these modes, its effect is to move SPSR_<mode> to CPSR if the processor is in a privileged mode and to do nothing if in User mode.

**Assembler Syntax**
(i) MOV, MVN – single operand instructions
<opcode>{cond}{S} Rd,<Op2>

(ii) CMP,CMN,TEQ,TST – instructions which do not produce a result
<opcode>{cond} Rn,<Op2>

(iii) AND, EOR, SUB, RSB, ADD, ADC, SBC, RSC, ORR, BIC
<opcode>{cond}{S} Rd,Rn,<Op2>

where <Op2> is Rm{,<shift>} or, <#expression>

{cond} – two-character condition mnemonic

{S} – set condition codes if S present (implied for CMP, CMN, TEQ, TST)

Rd, Rn and Rm are expressions evaluating to a register number.

If <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

<shift> is <shiftname> <register> or <shiftname> #expression, or RRX (rotate right one bit with extend).

<shiftname>s are: ASL, LSL, LSR, ASR, ROR

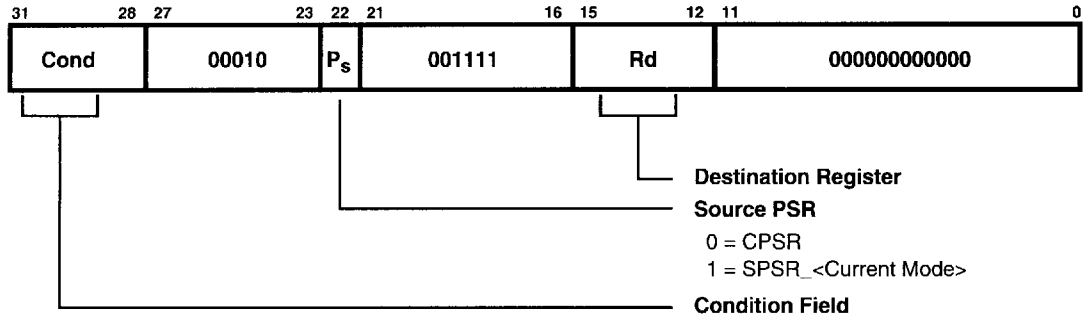
(ASL is a synonym for LSL, the two assemble to the same code.)

**Examples**

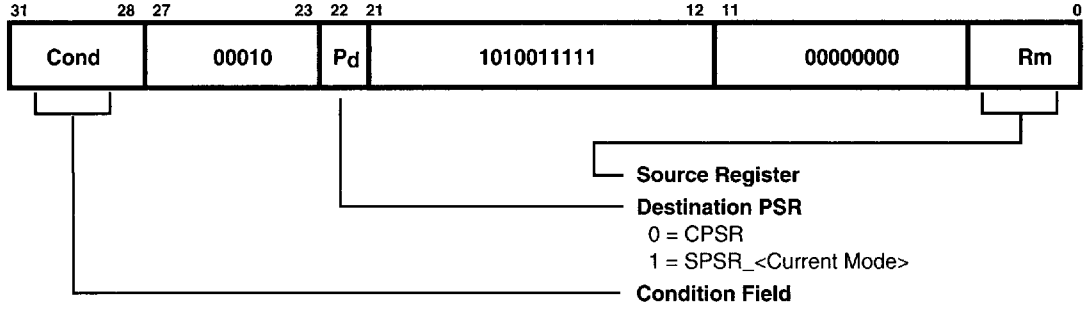
```
ADDEQ R2,R4,R5          ; if the Z flag is set make R2:=R4+R5
TEQS R4,#3              ; test R4 for equality with 3
                        ; (the S is in fact redundant as the
                        ; assembler inserts it automatically)
SUB R4, R5, R7, LSR R2  ; logical right shift R7 by the number in
                        ; the bottom byte of R2, subtract the result
                        ; from R5, and put the answer into R4
MOV PC, R14             ; return from subroutine
MOVS PC, R14            ; return from exception & restore CPSR_<mode>
```

## PSR TRANSFER (MRS, MSR)

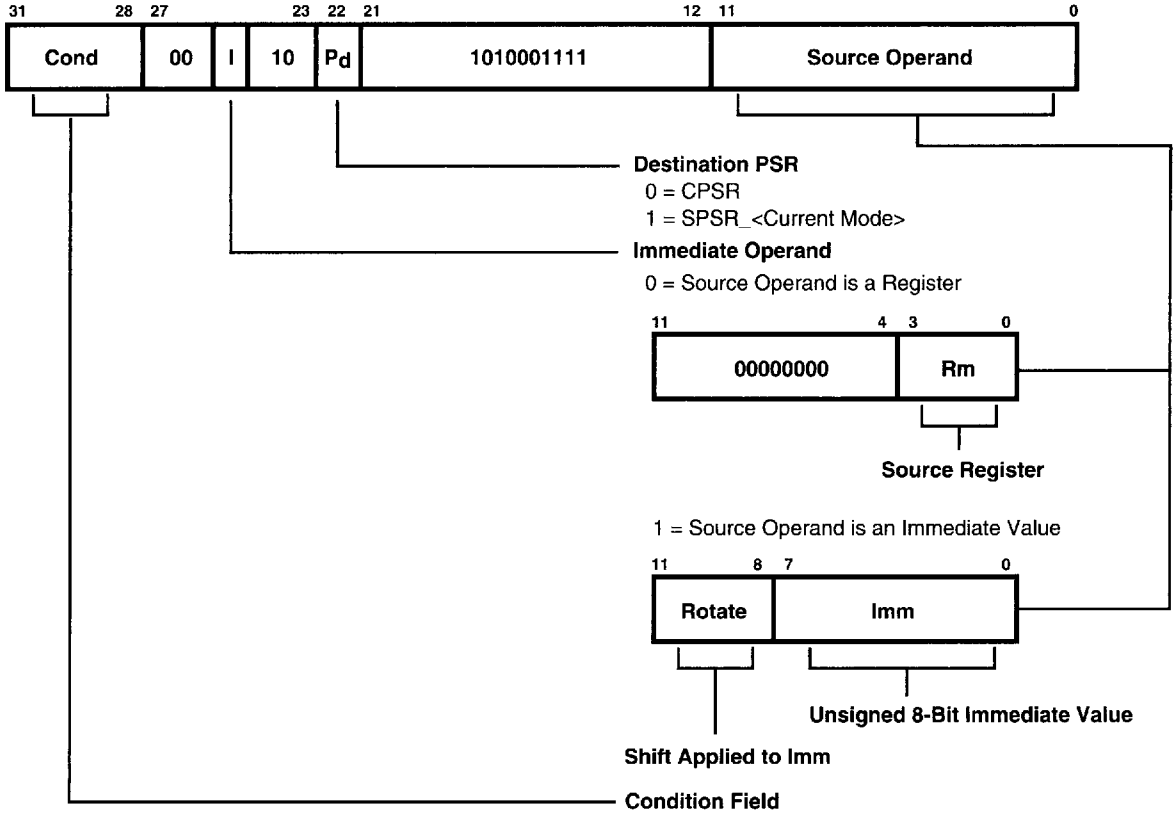
### MRS (Transfer PSR Contents to a Register)




### MSR (Transfer Register Contents to PSR)




### MSR (Transfer Register Contents or Immediate Value to PSR Flag Bits Only)



These instructions allow access to the CPSR and SPSR registers and are only executed if the condition is true. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register. The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register. R15 shall not be specified as the source or destination register.

The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V3) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32-bit immediate value are written to the top four bits of the relevant PSR.

**Operand Restrictions**

In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

Acess to the SPSR register depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.

A further restriction is that no attempt shall be made to access an SPSR in User mode, since no such register exists.

**Reserved Bits**

Only 11 bits of the PSR are defined in the VY86C610 (N,Z,C,V,I,F & M[4:0]); the remaining bits (= PSR[27:8,5]) are reserved for use in future versions of the processor. To ensure the maximum compatibility between the VY86C610 programs and future processors, the following rules should be observed:

(1) The reserved bits shall be preserved when changing the value in a PSR.

(2) Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

A read-modify-write strategy should therefore be used when altering the control bits of any PSR register. This involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits and then transferring the modified value pack to the PSR register using the MSR instruction.

e.g. The following sequence performs a mode change:

```
MRS   Rtmp,CPSR          ; take a copy of the CPSR
BIC   Rtmp,Rtmp,#&1F      ; clear the mode bits
ORR   Rtmp,Rtmp,#new_mode ; select new mode
MSR   CPSR,Rtmp           ; write-back the modified CPSR
```

When the aim is simply to change the condition code flags in a PSR, an immediate value can be written directly to the flag bits without disturbing the control bits.

For example, the following instruction sets the N,Z,C & V flags:

```
MSR   CPSR_flg,#&F0000000  ; set all the flags regardless of
                           ; their previous state (does not
                           ; affect any control bits)
```

No attempt shall be made to write an 8-bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.

**Assembler Syntax**

(i) MRS - transfer PSR contents to a register

MSR{cond} Rd,<cpsr>

(ii) MSR - transfer register contents to PSR

MSR{cond} <psr>,Rm

(iii) MSR - transfer register contents to PSR flag bits only

MSR{cond} <psrf>,Rm

The most significant four bits of the register contents are written to the N,Z,C, and V flags respectively.

(iiii) MSR - transfer immediate value to PSR flag bits only

MSR{cond} <psrf>,<#expression>

The expression should symbolize a 32-bit value of which the most significant four bits are written to the N,Z,C & V flags respectively.

{cond} - two-character condition mnemonic

Rd and Rm are expressions evaluating to a register number other than R15.

<psr> is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)

<psrf> is CPSR_flg or SPSR_flg

Where <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

**Examples**

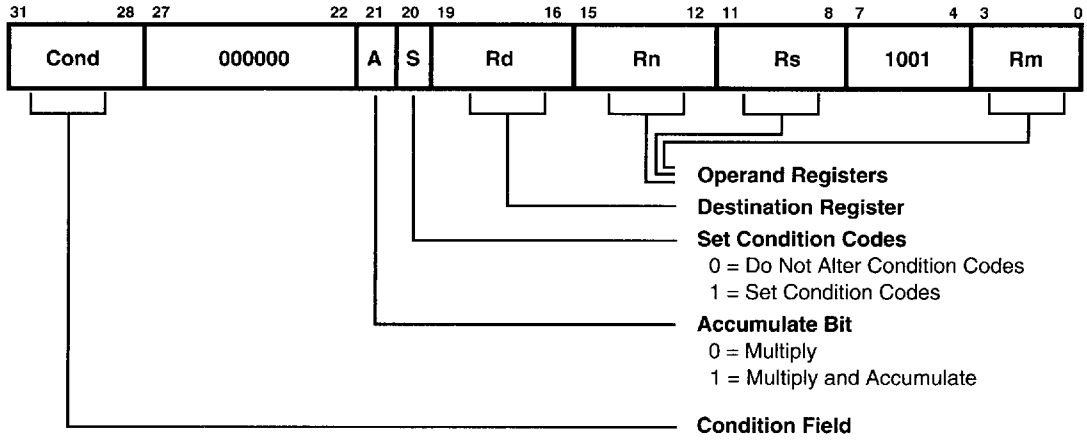
In User mode, the instructions behave as follows:

```
MSR   CPSR_all,Rm          ; CPSR[31:28] <- Rm [31:28]
MSR   CPSR_flg,Rm          ; CPSR[31:28] <- Rm [31:28]
MSR   CPSR_flg,#&A0000000  ; CPSR[31:28] <- &A
                           ; (i.e. set N,C; clear Z,V)
MRS   Rd,CPSR              ; Rd[31:0] <- CPSR[31:0]
```

In privileged modes, the instructions behave as follows:

```
MSR   CPSR_all,Rm          ; CPSR[31:0] <- Rm[31:0]
MSR   CPSR_flg,Rm          ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg,#&50000000  ; CPSR[31:28] <- &5
                           ; (i.e. set Z,V; clear N,C)
MRS   Rd,CPSR              ; Rd[31:0] <- CPSR[3l:0]
MSR   SPSR_all,Rm          ; SPSR_<mode>[31:0] <- Rm[31:0]
MSR   SPSR_flg,Rm          ; SPSR_<mode>[31:28] <- Rm[31:28]
MSR   SPSR_flg,#&C0000000  ; SPSR_<mode>[31:28] <- &C
                           ; (i.e. set N,Z; clear C,V)
MRS   Rd,SPSR              ; Rd[31:0] <- SPSR_<mode>[31:0]
```

**MULTIPLY AND MULTIPLY-ACCUMULATE (MUL, MLA)**



The instruction is only executed if the condition specified in the condition field is true.

The multiply and multiply-accumulate instructions use a two-bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32-bit operands, and may be used to synthesize higher precision multiplications.

The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

Both forms of the instruction work on operands which may be considered as signed (two's complement) or unsigned integers.

**Operand Restrictions**

Due to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the Rm operand register, as Rd is used to hold intermediate values and Rm is used repeatedly during the multiply. A MUL will give a zero result if Rm=Rd, and a MLA will give a meaningless result. R15 shall not be used as an operand or as the destination register.

All other register combinations will give correct results, and Rd, Rn, and Rs may use the same register when required.

**CPSR Flags**

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N and Z flags are set correctly on the result (N is equal to Bit 31 of the result, Z is set if and only if the result is zero), the V flag is unaffected by the instruction (as for logical data processing instructions), and the C flag is set to a meaningless value.

**Assembler Syntax**

MUL{cond}{S} Rd, Rm, Rs

MLA{cond}{S} Rd, Rm, Rs, Rn

{cond} – two-character condition mnemonic

{S} – set condition codes if S present

Rd, Rm, Rs and Rn are expressions evaluating to a register number other than R15.

**Examples**

```
MUL R1, R2, R3          ; R1:=R2*R3
MLAEQS R1, R2, R3, R4   ; conditionally R1:=R2*R3+R4,
                        ; setting condition codes
```

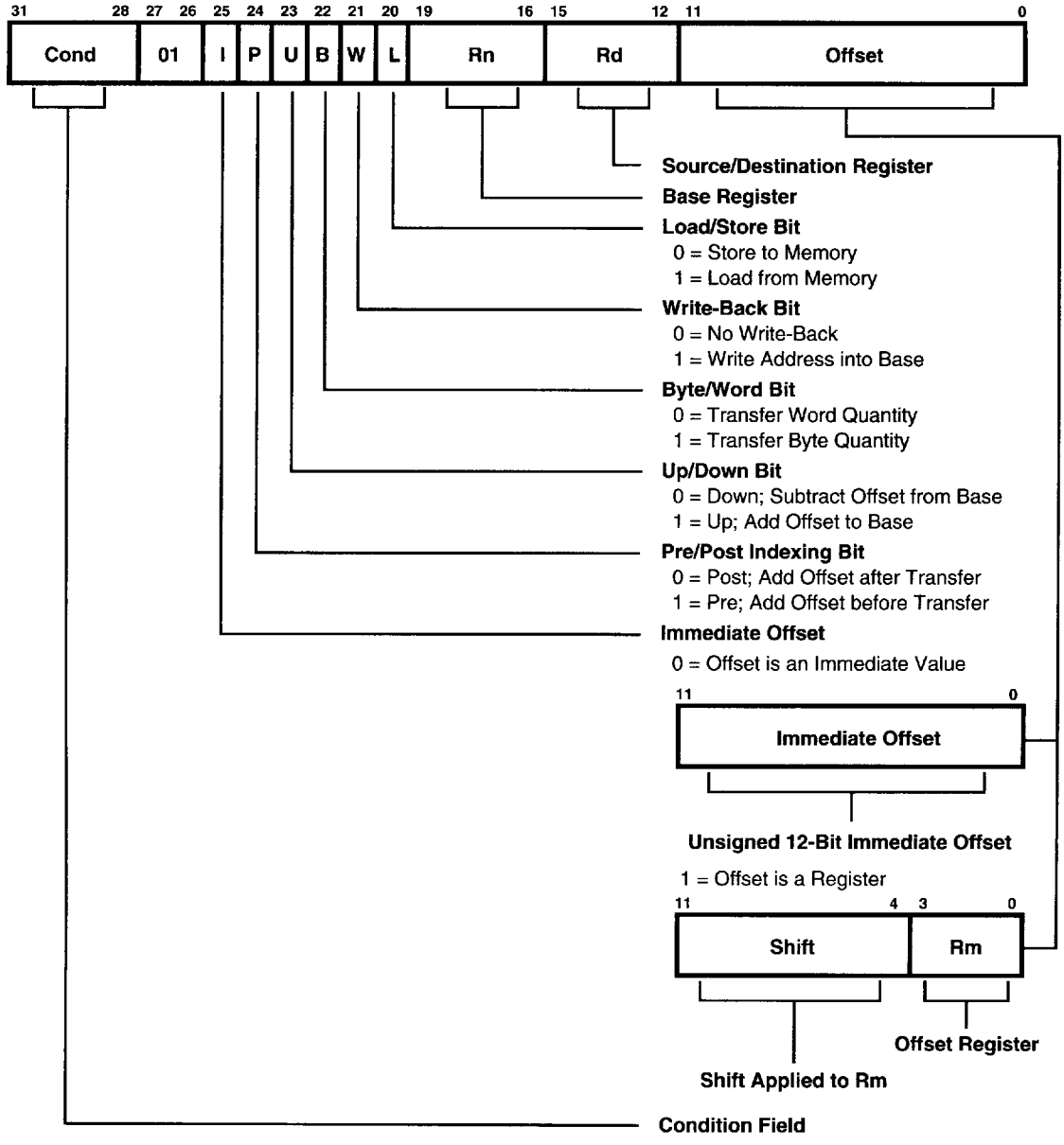
The multiply instruction may be used to synthesize higher precision multiplications for instance, to multiply two 32-bit integers and generate a 64-bit result:

MUL64

```
MOV     a1, A, LSR #16      ; a1:=top half of A
MOV     D, B, LSR #16       ; D:=top half of B
BIC     A, A, a1, LSL #16   ; A:=bottom half of A
BIC     B, B, D, LSL #16    ; B:=bottom half of B
MUL     C, A, B             ; low section of result
MUL     B, a1, B            ; ) middle sections
MUL     A, D, A             ; ) of result
MUL     D, a1, D            ; high section of result
ADDS    A, B, A             ; add middle sections
                            ; (couldn't use MLA as we need C correct)
ADDCS   D, D, #&10000       ; carry from above add
ADDS    C, C, A, LSL #16    ; C is now bottom 32 bits of product
ADC     D, D, A, LSR #16    ; D is top 32 bits
```

(A, B are registers containing the 32-bit integers; C, D are registers for the 64-bit result; a 1 is a temporary register. A and B are overwritten during the multiply.)

## SINGLE DATA TRANSFER (LDR, STR)




The instruction is only executed if the condition specified in the condition field is true.

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if 'auto-indexing' is required.

**Offsets and Auto-Indexing**

The offset from the base may be either a 12-bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write-back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write-back the modified base.

### Shifted Register Offset

The eight shift control bits are described in the data processing instructions but the register specified shift amounts are not available in this instruction class.

### Bytes and Words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an VY86C610 register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the BIGEND configuration signal to the processor. The two possible configurations are described below.

### Little Endian Configuration

A byte load (LDRB) expects the data on D[7:0] if the supplied address is on a word boundary, on D[15:8] if it is a word address plus one byte, and so on. The selected byte is placed in the bottom eight bits of the destination register, and the remaining bits of the register are filled with zeros.

A byte store (STRB) repeats the bottom eight bits of the source register four times across D[31:0]. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies Bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into Bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, Bit 31 of the register being stored always appears on D[31].

### Big Endian Configuration

A byte load (LDRB) expects the data on D[31:24] if the supplied address is on a word boundary, on D[23:16] if it is a word address plus one byte, and so on. The selected byte is placed in the bottom eight bits of the destination register and the remaining bits of the register are filled with zeros.

A byte store (STRB) repeats the bottom eight bits of the source register four times across D[31:0]. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies Bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into Bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies Bits 15 through 8.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, Bit 31 of the register being stored always appears on D[31].

### Use of R15

Write-back shall not be specified if R15 is specified as the base register (Rn). When using R15 as the base register one must remember that it contains an address eight bytes on from the address of the current instruction.

R15 shall not be specified as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 12.

### Restriction on the Use of Base Register

When configured for late aborts, the following code is very difficult to unwind as Rm gets updated before the abort handler is entered. In certain circumstances it may be impossible to calculate the initial value.

<LDR|STR> Rd, [Rn], {+/–}Rn{,<shift>}

A post-indexed LDR|STR where Rm=Rn shall not be used.

### Data Aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor abort signal HIGH, whereupon the data transfer instruction will be prevented from changing the processor state and the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

The VY86C610 supports two types of Data Abort processing depending on the lateabt configuration input. When configured for Early Aborts, any base register write-back which would have occurred is prevented from happening in the event of an Abort. When configured for Late Aborts, this write-back is allowed to take place and the Abort handler must correct this before allowing the instruction to be re-executed.

**Assembler Syntax**

<LDR|STR>{cond}{B} Rd,<Address>

LDR – load from memory into a register

STR – store from a register into memory

{cond} – two-character condition mnemonic

{B} – if B is present then byte transfer, otherwise word transfer

Rd is an expression evaluating to a valid register number.

<Address> can be:

(i) An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

[Rn,<#expression>]{!} offset of <expression> bytes

[Rn,{+/–}Rm{,<shift>}]{!} offset of +/– contents of index register, shifted by <shift>

(iii) A post-indexed addressing specification:

[Rn],<#expression> offset of <expression> bytes

[Rn],{+/–}Rm{,<shift>} offset of +/– contents of index register, shifted as by <shift>

Rn and Rm are expressions evaluating to a valid register number. NOTE: If Rn is R15 then the assembler will subtract eight from the offset value to allow for VY86C610 pipelining. In this case, Base write-back shall not be specified.

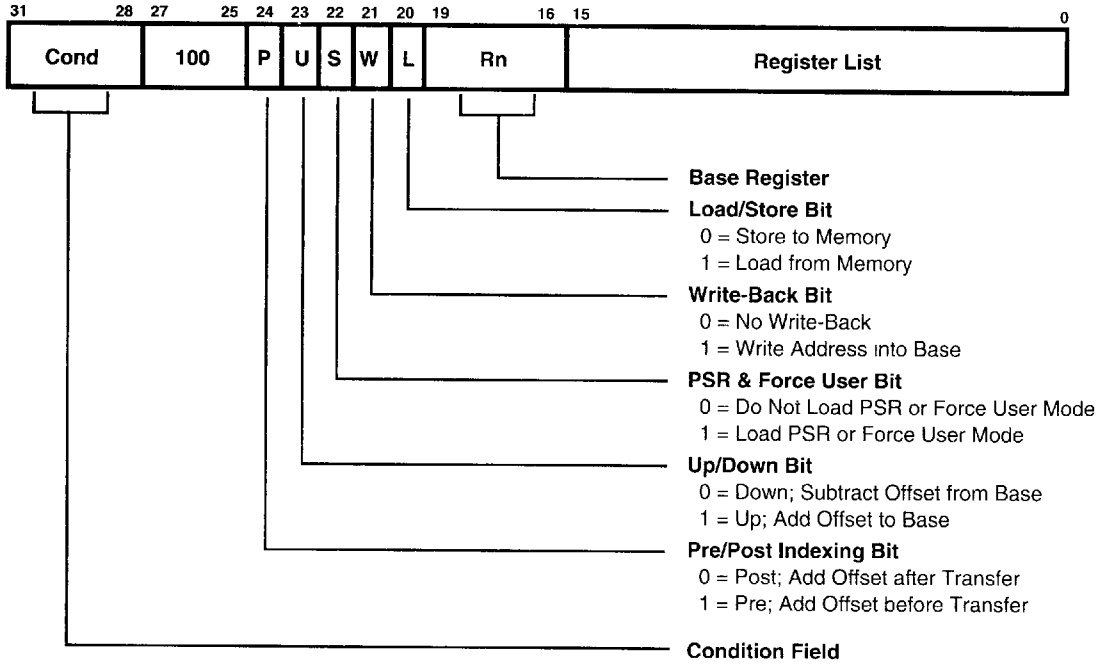
<shift> is a general shift operation (see section on Data Processing Instructions), but note that the shift amount may not be specified by a register.

{!} write-back the Base register (set the W bit) if ! is present

**Examples**

```
STR R1, [BASE,INDEX]!          ; store R1 at BASE+INDEX (both of which are
                               ; registers) and write-back address to BASE

STR R1, [BASE],INDEX           ; store R1 at BASE and write-back
                               ; BASE+INDEX to BASE

LDR R1, [BASE, #16]            ; load R1 from contents of BASE+16
                               ; Don't write-back

LDR R1, [BASE,INDEX,LSL #2]    ; load R1 from contents of BASE+INDEX*4

LDREQB R1, [BASE,#5]           ; conditionally load byte at BASE+5 into
                               ; R1 bits 0 to 7, filling Bits 8 to 31
                               ; with zeros

STR R1, PLACE                  ; generate PC relative offset to address
        .                      ; PLACE
        .

PLACE
```

## BLOCK DATA TRANSFER (LDM, STM)




The instruction is only executed if the condition specified in the condition field is true.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around the main memory.

### Register List

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the User bank). The register list is a 16-bit field in the instruction, with each bit corresponding to a register. A 1 in Bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly Bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 12.

### Addressing Modes

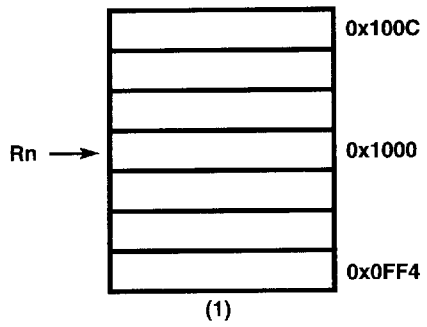
The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5, and R7 in the case where Rn=0x1000 and write-back of the modified base is required (W=1). The following figures show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

(In all cases, had write-back of the modified base not been required (W=0), Rn would have retained its initial value of 0x1000 unless it was also in the transfer list of a load-multiple register instruction, when it would have been overwritten with the loaded value.)

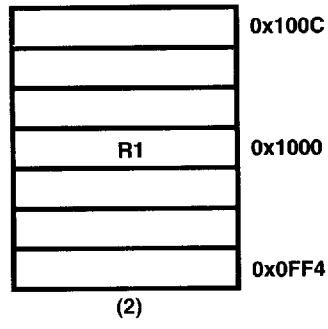
### Address Alignment

The address should normally be a word aligned quantity and non-word aligned addresses do not affect the instruction. However, the bottom two bits of the address will appear on A[1:0] and might be interpreted by the memory system.

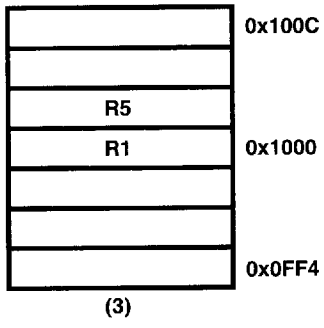
## POST-INCREMENT ADDRESSING



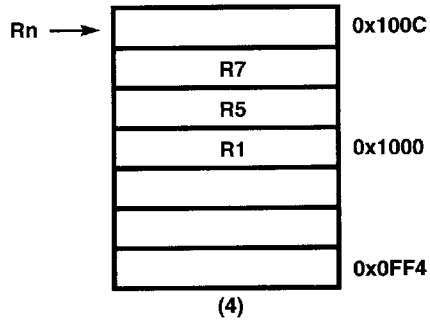

(1)




(2)

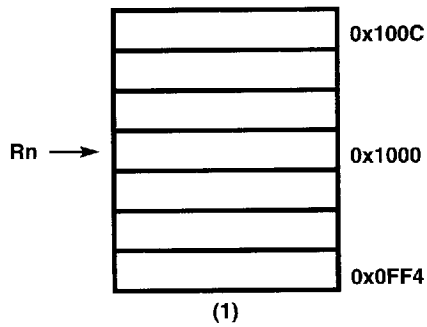



(3)

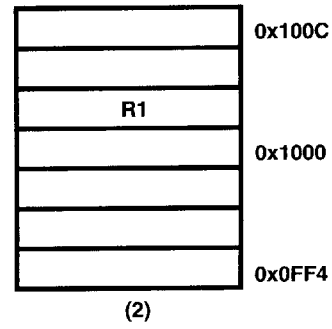



(4)

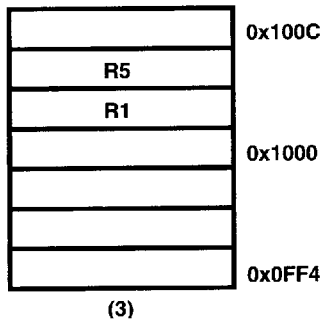
## PRE-INCREMENT ADDRESSING



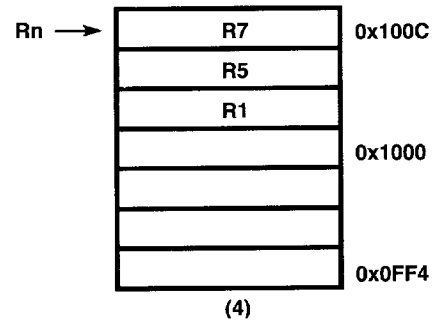

(1)




(2)

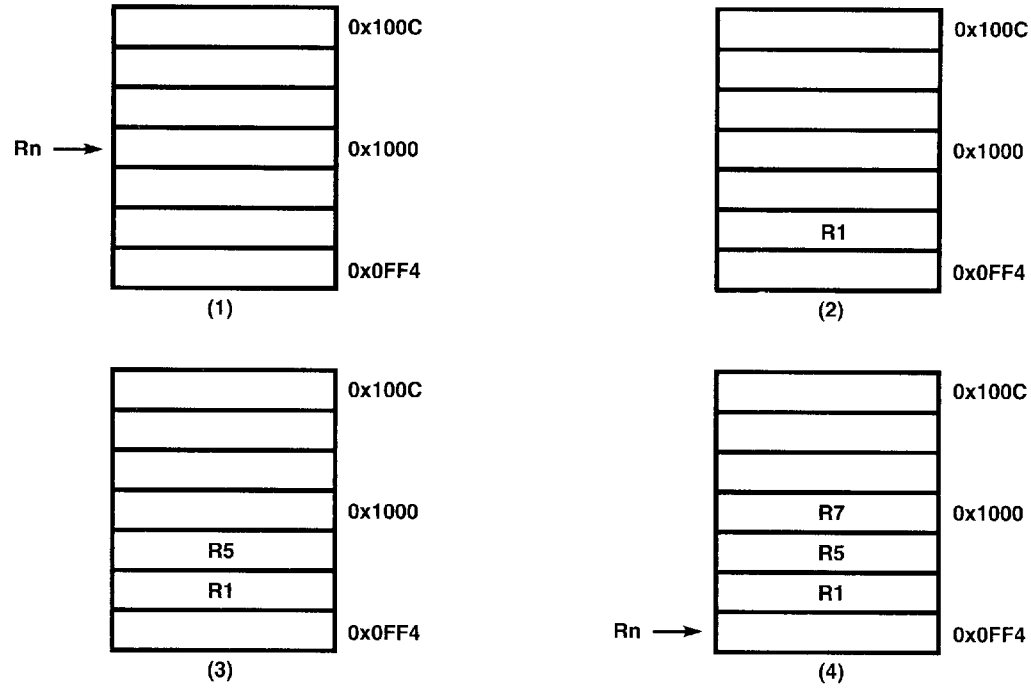



(3)

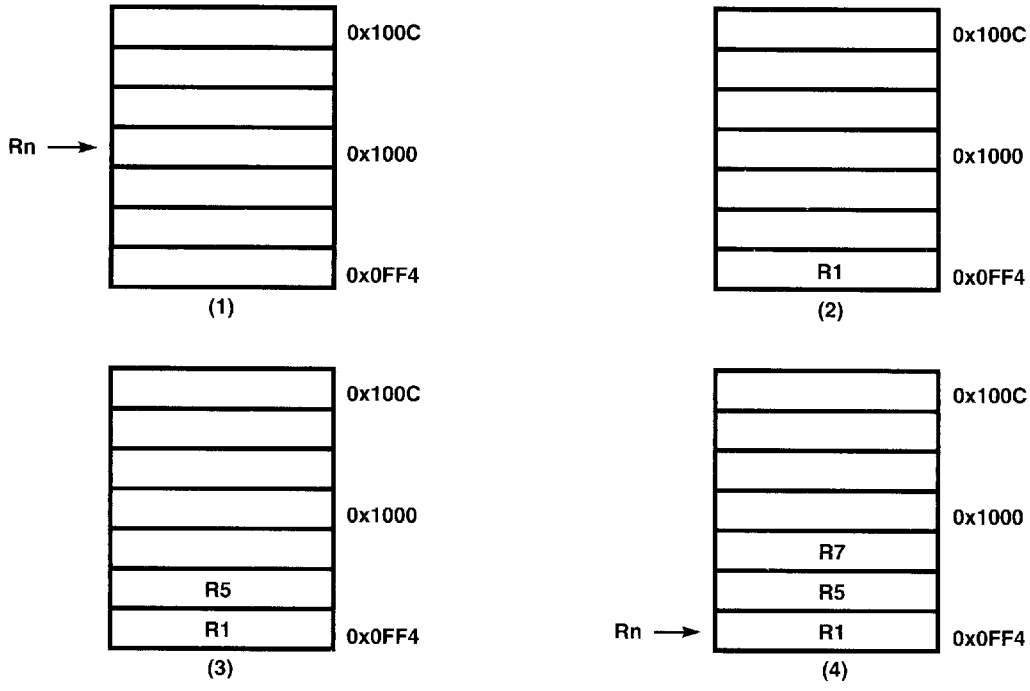



(4)

## POST-DECREMENT ADDRESSING




## PRE-DECREMENT ADDRESSING



**Use of S Bit**
When the S bit is set in a LDM/STM instruction, its meaning depends on whether or not R15 is in the transfer list, and on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode.

**LDM With R15 in Transfer List and S Bit Set (Mode Changes)**
If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

**STM With R15 in Transfer List and S Bit Set (User Bank Transfer)**
The registers transferred are taken from the User bank rather than the bank corresponding to the current mode. This is useful for saving the User state on process switches. Base write-back shall not be used when this mechanism is employed.

**R15 Not in List and S Bit Set (User Bank Transfer)**
For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode. This is useful for saving the User state on process switches. Base write-back shall not be used when this mechanism is employed.

When the instruction is LDM, care must be taken not to read from a banked register during the following cycle (inserting a NOP after the LDM will ensure safety).

**Use of R15 as Base**
R15 shall not be used as the Base register in any LDM or STM instruction.

**Inclusion of Base in Register List**
When write-back is specified, the Base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the Base, with the Base as the first register to be stored, will therefore store the unchanged value, whereas with the Base second or later in the transfer order, will store the modified value. A LDM will always overwrite the updated Base if the Base is in the list.

**Data Aborts**
Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the abort signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if the VY86C610 is to be used in a virtual memory system.

The state of the LATEABT input does not affect the behavior of LDM and STM instructions in the event of an Abort exception.

**Aborts During STM Instructions**
If the abort occurs during a store multiple instruction, the VY86C610 takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

**Aborts During LDM Instructions**
When the VY86C610 detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

(1) Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have overwritten registers. The PC is always the last register to be written and so will always be preserved.

(2) The base register is restored to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

**Assembler Syntax**
<LDM|STM>{cond}<FD|ED|FA|EA|IA|IB|DA|DB> RN{!},<Rlist>{^}

{cond} – two character condition mnemonic

Rn is an expression evaluating to a valid register number.

<Rlist> can be either a list of registers and register ranges enclosed in {} (e.g. {R0, R2-R7, R10}), or an expression evaluating to the 16-bit operand.

{!} if present, requests write-back (W=1), otherwise W=0

{^} if present, set S bit to load the CPSR along with the PC, or force transfer of the User bank when in privileged mode

### Addressing Mode Names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes as shown in the adjacent table.

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a "full" or "empty" stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine.

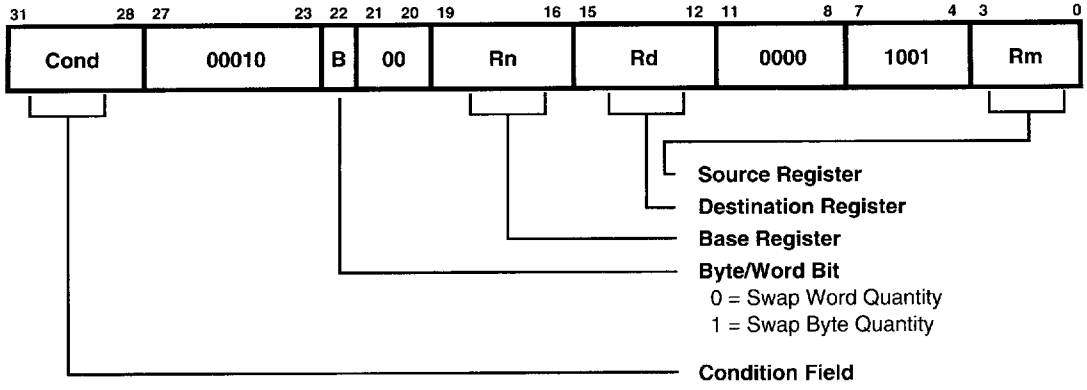
**NAMES AND BIT PATTERNS**

| Name | Stack | Other | L Bit | P Bit | U Bit |
|---|---|---|---|---|---|
| Pre-increment load | LDMED | LDMIB | 1 | 1 | 1 |
| Post-increment load | LDMFD | LDMIA | 1 | 0 | 1 |
| Pre-decrement load | LDMEA | LDMDB | 1 | 1 | 0 |
| Post-decrement load | LDMFA | LDMDA | 1 | 0 | 0 |
| Pre-increment store | STMFA | STMIB | 0 | 1 | 1 |
| Post-increment store | STMEA | STMIA | 0 | 0 | 1 |
| Pre-decrement store | STMFD | STMDB | 0 | 1 | 0 |
| Post-decrement store | STMED | STMDA | 0 | 0 | 0 |

### Examples

```
LDMFD SP!, {R0, R1, R2}      ; unstack 3 registers
STMIA BASE, {R0-R15}         ; save all registers
LDMFD SP!, {R15}             ; R15 <- (SP), CPSR unchanged
LDMFD SP!, {R15}^            ; R15 <- (SP), CPSR <- SPSR_mode (allowed only
                             ; in privileged modes)
STMFD R13, {R0-R14}^         ; Save user mode regs on stack (allowed only
                             ; in privileged modes)
STMED SP!, {R0-R3, R14}      ; save R0 to R3 to use as workspace
                             ; and R14 for returning
BL somewhere                 ; this nested call will overwrite R14
LDMED SP!, {R0-R3, R15}      ; restore workspace and return
```

## SINGLE DATA SWAP (SWP)




The instruction is only executed if the condition specified in the condition field is true.

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are locked together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. It then writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

The lock output goes HIGH for the duration of the read and write operations to signal to the external memory manager that they are locked together, and should be allowed to complete without interruption. This is important in multiprocessor systems where the swap instruction is the only indivisible instruction which may be used to implement semaphores; control of the memory must not be removed from a processor while it is performing a locked operation.

### Bytes and Words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between a VY86C610 register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

### Use of R15

R15 shall not be used as an operand (Rd, Rn or Rs) in a SWP instruction.

### Data Aborts

If the address used for the swap is unacceptable to a memory management system, the memory manager can flag the problem by driving abort HIGH. This can happen on either the read or the write cycle (or both), and in either case, the data swap instruction will be prevented from changing the processor state and the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Because no base register write-back is allowed, the behavior of an aborted SWP instruction is the same regardless of the state of the LATEABT configuration input.

### Assembler Syntax

<SWP>{cond}{B} Rd, Rm, [Rn]

{cond} – two-character condition mnemonic.

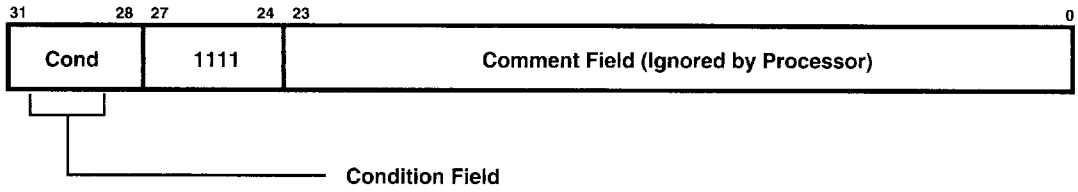
{B} – if B is present then byte transfer, otherwise word transfer.

Rd, Rm, Rn are expressions evaluating to valid register numbers.

### Examples

```
SWP R0, R1, [BASE]      ; load R0 with the contents of BASE, and
                        ; store R1 at BASE
SWPB R2, R3, [BASE]     ; load R2 with the byte at BASE, and
                        ; store Bits 0 to 7 of R3 at BASE
SWPEQ R0, R0, [BASE]    ; conditionally swap the contents of BASE
                        ; with R0
```

## SOFTWARE INTERRUPT (SWI)




The instruction is only executed if the condition specified in the condition field is true.

The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is forced to a fixed value (&08) and the CPSR is saved in SPSR_svc. If this address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

### Return From the Supervisor

The PC is saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS PC,R14_svc will return to the calling program and restore the CPSR.

Note that the link mechanism is not reentrant. If the supervisor code wishes to use software interrupts within itself, it must first save a copy of the return address and SPSR.

### Comment Field

The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

### Assembler Syntax

SWI{cond} <expression>

{cond} – two character condition mnemonic

<expression> is evaluated and placed in the comment field (which is ignored by the VY86C610).

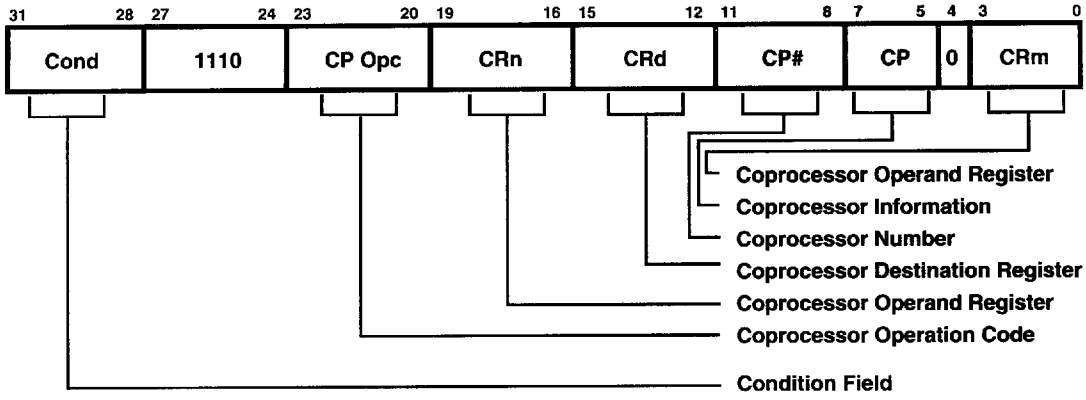
### Examples

| | | |
|---|---|---|
| SWI | ReadC | get next character from read stream |
| SWI | WriteI+"k" | output a "k" to the write stream |
| SWINE | 0 | conditionally call supervisor with 0 in comment field |

The above examples assume that suitable supervisor code exists, for instance:

```
&08 B Supervisor            ; SWI entry point

EntryTable                  ; addresses of supervisor routines
    & ZeroRtn
    & ReadCRtn
    & WriteIRtn
    . . .

Zero    * 0

ReadC   * 256

WriteI  * 512

Supervisor

; SWI has routine required in Bits 8–23 and data (if any) in Bits 0–7

; Assumes R13_svc points to a suitable stack
    STM R13, {R0-R2, R14}       ; save work registers and return address
    LDR R0, [R14, #-4]          ; get SWI instruction
    BIC R0, R0, #&FF000000      ; clear top 8 bits
    MOV R1, R0, LSR #8          ; get routine offset
    ADR R2, EntryTable          ; get start address of entry table
    LDR R15, [R2, R1, LSL #2]   ; branch to appropriate routine

WriteIRtn                       ; enter with character in R0 Bits 0–7
    . . . . . .
LDM R13, {R0–R2, R15}^          ; restore workspace and return
```

## COPROCESSOR DATA OPERATIONS (CDP)




The instruction is only executed if the condition specified in the condition field is true.

This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to the VY86C610, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other VY86C610 activity allowing the coprocessor and the VY86C610 to perform independent tasks in parallel.

**Coprocessor Fields**

Only Bit 4 and Bits 24 to 31 are significant to the VY86C610; the remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

**Assembler Syntax**

CDP{cond} CP#,<expression1>,CRd, CRn, CRm{,<expression2>}

{cond} – two character condition mnemonic

CP# – the unique number of the required coprocessor

<expression1> – evaluated to a constant and placed in the CP Opc field

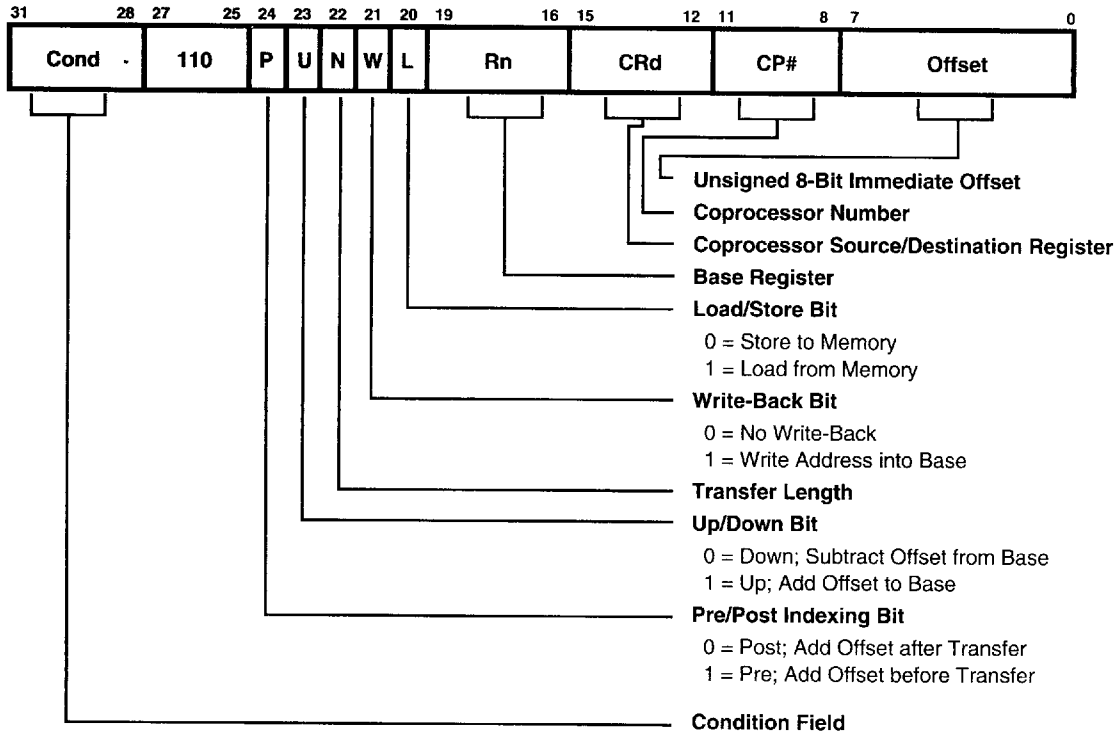
CRd, CRn and CRm are expressions evaluating to a valid coprocessor register number.

<expression2> – where present is evaluated to a constant and placed in the CP field

**Examples**

```
CDP 1, 10, CR1, CR2, CR3        ; request coproc 1 to do operation 10
                                ; on CR2 and CR3, and put the result in CR1

CDPEQ 2, 5, CR1, CR2, CR3, 2    ; if Z flag is set request coproc 2 to do
                                ; operation 5 (type 2) on CR2 and CR3,
                                ; and put the result in CR1
```

## COPROCESSOR DATA TRANSFERS (LDC, STC)




The instruction is only executed if the condition specified in the condition field is true.

This class of instruction is used to transfer one or more words of data between a coprocessor and main memory. The VY86C610 is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred. This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessor's registers directly to memory.

### Coprocessor Fields

The CP# field is used to identify the coprocessor which is required to supply or accept the data, and a coprocessor will only respond if its number matches the contents of this field.

The CRd field and the N bit contain information for the coprocessor which may be interpreted in different ways by different coprocessors, but by convention CRd is the register to be transferred (or the first register when more than one is to be transferred), and the N bit is used to choose one of two transfer length options. For instance N=0 could select the transfer of a single register, and N=1 could select the transfer of all the registers for context switching.

### Addressing Modes

The VY86C610 is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note that the immediate offsets are eight bits wide and specify word offsets for coprocessor data transfers, whereas for single data transfer they are 12 bits wide and specify byte offsets.

The eight-bit unsigned immediate offset is shifted left 2 bits, and either added to (U=1) or subtracted from (U=0) the base register (Rn); this calculation may be performed either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0). Note that post-indexed addressing modes require explicit setting of the W bit, unlike LDR and STR which always write-back when post-indexed.

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to, or come from, an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

**Address Alignment**
The base address should normally be a word aligned quantity. The bottom two bits of the address will appear on A[1:0] and might be interpreted by the memory system.

**Use of R15**
If Rn is R15, the value used will be the address of the instruction plus eight bytes. Base write-back shall not be specified.

**Data Aborts**
If the address is legal but the memory manager generates an abort, the data abort trap will be taken. The write-back of the modified base will take place, but all other processor state will be preserved. The coprocessor is partly responsible for ensuring that the data transfer can be restarted after the cause of the abort has been resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

The state of the LATEABT input does not affect the behavior of LDC and STC instructions in the event of an Abort exception.

**Assembler Syntax**
<LDC|STC>{cond}{L} CP#,CRd,<Address>

LDC – load from memory to coprocessor

STC – store from coprocessor to memory

{L} – when present, perform long transfer (N=1), otherwise perform short transfer (N=0).

{cond} – two-character condition mnemonic

CP# – the unique number of the required coprocessor

CRd is an expression evaluating to a valid coprocessor register number.

<Address> can be:

(i) An expression which generates an address:
<expression>
The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:
[Rn] offset of zero
[Rn,<#expression>]{!} offset of <expression> bytes

(iii) A post-indexed addressing specification:
[Rn],<#expression> offset of <expression> bytes

Rn is an expression evaluating to a valid VY86C610 register number. NOTE if Rn is R15 then the assembler will subtract eight from the offset value to allow for the VY86C610 pipelining.

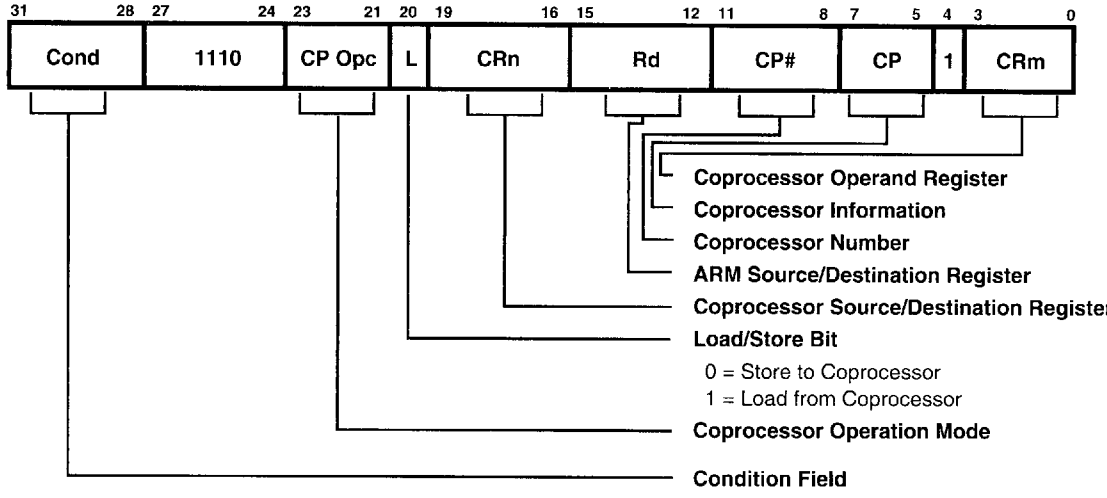
{!} write-back the Base register (set the W bit) if ! is present

**Examples**

```
LDC 1, CR2, table          ; load CR2 of coproc 1 from address table,
                           ; using a PC relative address

STCEQL 2, CR3, [R5,#24]!   ; conditionally store CR3 of coproc 2 into
                           ; an address 24 bytes up from R5, write this
                           ; address back into R5, and use long transfer
                           ; option (probably to store multiple words)
```

Note that though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

## COPROCESSOR REGISTER TRANSFERS (MRC, MCR)




The instruction is only executed if the condition specified in the condition field is true.

This class of instruction is used to communicate information directly between the VY86C610 and a coprocessor. An example of a coprocessor to the VY86C610 register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32-bit integer within the coprocessor, and the result is then transferred to a VY86C610 register. A FLOAT of a 32-bit value in a VY86C610 register into a floating point value within the coprocessor illustrates the use of a VY86C610 register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the VY86C610 CPSR flags. As an example, the result of a comparison of two floating point values within a coprocessor can be moved to the CPSR to control the subsequent flow of execution.

### Coprocessor Fields

The CP# field is used, as for all coprocessor instructions, to specify which coprocessor is being called upon to respond.

The CP Opc, CRn, CP and CRm fields are used only by the coprocessor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the coprocessor functionality is incompatible with this one. The conventional interpretation is that the CP Opc and CP fields specify the operation the coprocessor is required to perform, CRn is the coprocessor register which is the source or destination of the transferred information, and CRm is a second coprocessor register which may be involved in some way which depends on the particular operation specified.

### Transfers to R15

When a coprocessor register transfer to the VY86C610 has R15 as the destination, Bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other CPSR bits are unaffected by the transfer.

### Transfers From R15

A coprocessor register transfer from VY86C610 with R15 as the source register will store the PC+12.

### Assembler Syntax

<MCR|MRC>{cond} CP#,<expression1>,Rd,CRn,CRm{,expression2>}

MRC – move from coprocessor to VY86C610 register (L=1)

MCR – move from the VY86C610 register to coprocessor (L=0)

{cond} – two-character condition mnemonic

CP# – the unique number of the required coprocessor

<expression1> – evaluated to a constant and placed in the CP Opc field

Rd is an expression evaluating to a valid VY86C610 register number.

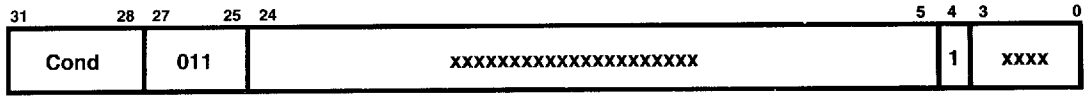
CRn and CRm are expressions evaluating to a valid coprocessor register number.

<expression2> – where present, is evaluated to a constant and placed in the CP field

### Examples

```
MRC 2, 5, R3, CR5, CR6          ; request coproc 2 to perform operation 5
                                ; on CR5 and CR6, and transfer the (single
                                ; 32-bit word) result back to R3

MCR 6, 0, R4, CR6               ; request coproc 6 to perform operation 0
                                ; on R4 and place the result in CR6

MRCEQ 3, 9, R3, CR5, CR6, 2     ; conditionally request coproc 2 to perform
                                ; operation 9 (type 2) on CR5 and CR6, and
                                ; transfer the result back to R3
```

## UNDEFINED INSTRUCTION[1]




If the condition specified in the condition field is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering this instruction to any coprocessors which may be present, and all coprocessors must refuse to accept it by driving CPA and CPB HIGH.

**Assembler Syntax**
Currently, the assembler has no mnemonics for generating this instruction. If it is adopted in the future for a specified use, appropriate mnemonics will be added to the assembler. Until such time, this instruction shall not be used.

## INSTRUCTION SET SUMMARY[1]

| 31–28 | 27–26 | 25 | 24–23 | 22 | 21 | 20 | 19–16 | 15–12 | 11–8 | 7–5 | 4 | 3–0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Cond | 00 | I | OpCode | | S | | Rn | Rd | Operand 2 | | | | Data Processing PSR Transfer |
| Cond | 000000 | | | | A | S | Rn | Rd | Rs | 1001 | | Rm | Multiply |
| Cond | 00010 | | | B | 00 | | Rn | Rd | 0000 | 1001 | | Rm | Single Data Swap |
| Cond | 01 | I | P | U | B | W | L | Rn | Rd | Offset | | | Single Data Transfer |
| Cond | 011 | xxxxxxxxxxxxxxxxxxxx | | | | | | | | | 1 | xxxx | Undefined |
| Cond | 100 | P | U | S | W | L | Rn | Register List | | | | | Block Data Transfer |
| Cond | 101 | L | Offset | | | | | | | | | | Branch |
| Cond | 110 | P | U | N | W | L | Rn | CRd | CP# | Offset | | | Coproc Data Transfer |
| Cond | 1110 | Cp Opc | | | | | CRn | CRd | CP# | CP | 0 | CRm | Coproc Data Operation |
| Cond | 1110 | Cp Opc | | | | L | CRn | Rd | CP# | CP | 1 | CRm | Coproc Register Transfer |
| Cond | 1111 | Ignored by Processor | | | | | | | | | | | Software Interrupt |

**Note:**
1. Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken (i.e., a Multiply instruction with Bit 5 or Bit 6 changed to a 1). These instructions shall not be used, as their action may change in future ARM implementations.

## INSTRUCTION SET EXAMPLES

The following examples show ways in which the basic VY86C610 instructions can be combined to provide more efficient code.

### Using the Conditional Instructions

(1) using conditionals for logical OR

```
        CMP     RN, #p          ; if Rn=p OR Rm=q THEN GOTO
                                  Label
        BEQ     Label
        CMP     Rm, #q
        BEQ     Label
```

can be replaced by

```
        CMP     Rn, #p
        CMPNE   RM, #q          ; if condition not satisfied try other
                                  test
        BEQ     Label
```

(2) absolute value

```
        TEQ     Rn, #0          ; test sign
        RSBMI   Rn, Rn, #0      ; and two's-complement if necessary
```

(3) multiplication by 4, 5 or 6 (run time)

```
        MOV     Rc, Ra, LSL #2  ; multiply by 4
        CMP     Rb, #5          ; test value
        ADDCS   Rc, Rc, Ra      ; complete multiply by 5
        ADDHI   Rc, Rc, Ra      ; complete multiply by 6
```

(4) combining discrete and range tests

```
        TEQ     Rc, #127        ; discrete test
        CMPNE   Rc, #" "-1      ; range test
        MOVLS   Rc, #"."        ; IF Rc<=" " OR Rc=CHR$127
                                ; THEN RC:="."
```

(5) division and remainder

```
    ; enter with numbers in Ra and Rb
    ;
        MOV     Rcnt, #1        ; bit to control the division
Div1    CMP     Rb, #&80000000  ; move Rb until greater than Ra
        CMPCC   Rb, Ra
        MOVCC   Rb, Rb, ASL #1
        MOVCC   Rcnt, Rcnt, ASL #1
        BCC     Div1
        MOV     Rc,#0
Div2    CMP     Ra, Rb          ; test for possible subtraction
        SUBCS   Ra, Ra, Rb      ; subtract if ok
        ADDCS   Rc, Rc, Rcnt    ; put relevant bit into result
        MOVS    Rcnt, Rcnt, LSR #1 ; shift control bit
        MOVNE   Rb, Rb, LSR #1  ; halve unless finished
        BNE     Div2
;
; divide result in Rc
; remainder in Ra
```

**Pseudo-Random Binary Sequence Generator**

It is often necessary to generate pseudo-random numbers. The most efficient algorithms are based on shift generators with Exclusive_OR feedback, like a cyclic redundancy check generator. Unfortunately, the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (i.e. $2^{32}$-1 cycles before repetition). This example uses a 33-bit register with taps at Bits 33 and 20. The basic algorithm is newbit:=bit33 Xor bit20, shift left the 33-bit number and put in newbit at the bottom. This operation is performed for all the newbits needed (i.e. 32-bits). The entire operation can be done in 5 cycles:

```
; enter with seed in Ra (32 bits), Rb (1 bit in Rb 1sb), uses Rc
;
        TST     Rb, Rb, LSR #1      ; top bit into carry
        MOVS    Rc, Ra, RRX         ; 33-bit rotate right
        ADC     Rb, Rb, Rb          ; carry into 1sb of Rb
        EOR     Rc, Rc, Ra, LSL#12  ; (involved!)
        EOR     Ra, Rc, Rc, LSR#20  ; (similarly involved!)
;
; new seed in Ra, Rb as before
```

**Multiplication by Constant Using the Barrel Shifter**

(1) Multiplication by 2^n (1, 2, 4, 8, 16, 32..)

```
        MOV     Ra, Ra, LSL #n
```

(2) Multiplication by 2^n+1 (3, 5, 9, 17..)

```
        ADD     Ra, Ra, Ra, LSL #n
```

(3) Multiplication by 2^n–1 (3, 7, 15..)

```
        RSB     Ra, Ra, Ra, LSL #n
```

(4) Multiplication by 6

```
        ADD     Ra, Ra, Ra, LSL #1  ; multiply by 3
        MOV     Ra, Ra, LSL #1      ; and then by 2
```

(5) Multiply by 10 and add in extra number

```
        ADD     Ra, Ra, Ra, LSL #2  ; multiply by 5
        ADD     Ra, Rc, Ra, LSL #1  ; multiply by 2 and add in next digit
```

(6) General recursive method for Rb := Ra*C, where C is a constant:

(a) If C even, say C = 2^n*D, D odd:

```
D=1:    MOV     Rb, Ra, LSL #n
D<>1:   {Rb := Ra*D}
        MOV     Rb, Rb, LSL #n
```

(b) If C MOD 4 = 1, say C = 2^n*D+1, D odd, n>1:

```
D=1:    ADD     Rb, Ra, Ra, LSL #n
D<>1:   {RB := Ra*D}
        ADD     Rb, Ra, Rb, LSL #n
```

(c) If C MOD 4 = 3, say C = 2^n*D–1, D odd, n>1:

```
D=1:    RSB     Rb, Ra, Ra, LSL #n
D<>1:   {Rb := Ra*D}
        RSB     Rb, Ra, Rb, LSL #n
```

This method is not quite optimal, but close. An example of where it is less than optimal is multiply by 45 which is done by:

```
        RSB     Rb, Ra, Ra, LSL #2  ; multiply by 3
        RSB     Rb, Ra, Rb, LSL #2  ; multiply by 4*3–1 = 11
        ADD     Rb, Ra, Rb, LSL #2  ; multiply by 4*11+1 = 45
```

rather than by:

```
        ADD     Rb, Ra, Ra, LSL #3  ; multiply by 9
        ADD     Rb, Rb, Rb, LSL #2  ; multiply by 5*9 = 45
```

**Loading a Word From an Unknown Alignment**

```
; enter with address in Ra (32 bits)
; uses Rb, Rc; result in Rd
; Note d must be less than c (e.g. 0,1)
;
   BIC      Rb, Ra, #3          ; get word aligned address
   LDMIA    Rb, {Rd, Rc}        ; get 64 bits containing answer
   AND      Rb, Ra, #3          ; correction factor in bytes
   MOVS     Rb, Rb, LSL #3      ; ...now in bits and test if aligned
   MOVNE    Rd, Rd, LSR Rb      ; produce bottom of result word
                                ; (if not aligned)
   RSBNE    Rb, Rb, #32         ; get other shift amount
   ORRNE    Rd, Rd, Rc, LSL Rb  ; combine two halves to get result
```

**Loading a Halfword (Little Endian)**

```
   LDR      Ra, [Rb, 2]         ; Get halfword to Bits 15:0
   MOV      Ra, Ra, LSL #16     ; move to top
   MOV      Ra, Ra, LSR #16     ; and back to bottom
                                ; use ASR to get sign extended version
```

**Loading a Halfword (Big Endian)**

```
   LDR      Ra, [Rb, 2]         ; Get halfword to Bits 31:16
   MOV      Ra, Ra, LSR #16     ; and back to bottom
                                ; use ASR to get sign extended version
```

## ADDITIONAL INSTRUCTIONS FOR VY86C610

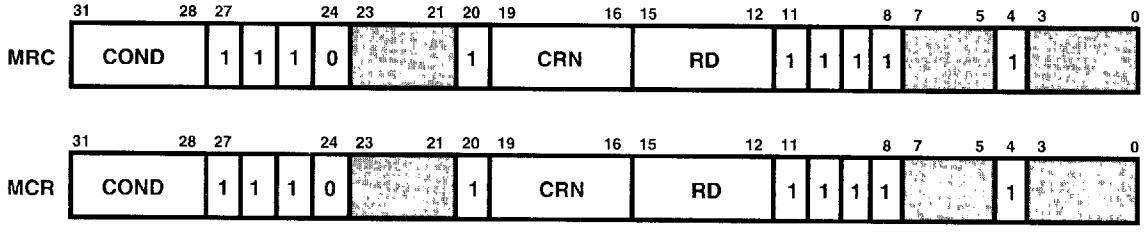
The operation and configuration of the VY86C610 is controlled both directly via coprocessor instructions and indirectly via the Memory Management Page tables. The coprocessor instructions manipulate a number of on-chip registers that control the configuration of the Cache, Write Buffer, MMU and a number of other configuration options.

To ensure backward compatibility of future CPUs, all reserved or unused bits in registers and coprocessor instructions shall be programmed to '0'. Invalid registers must not be read/written. The following bits shall be programmed to '0'.

Register 1 bits[30:9]
Register 2 bits[13:0]
Register 5 bits[31:0]
Register 6 bits[11:0]
Register 7 bits[31:0]

### INTERNAL COPROCESSOR INSTRUCTIONS

The on-chip registers may be read using MRC instructions and written using MCR instructions. These operations are only allowed in non-user modes, and the undefined instruction trap will be taken if accesses are attempted in user mode.

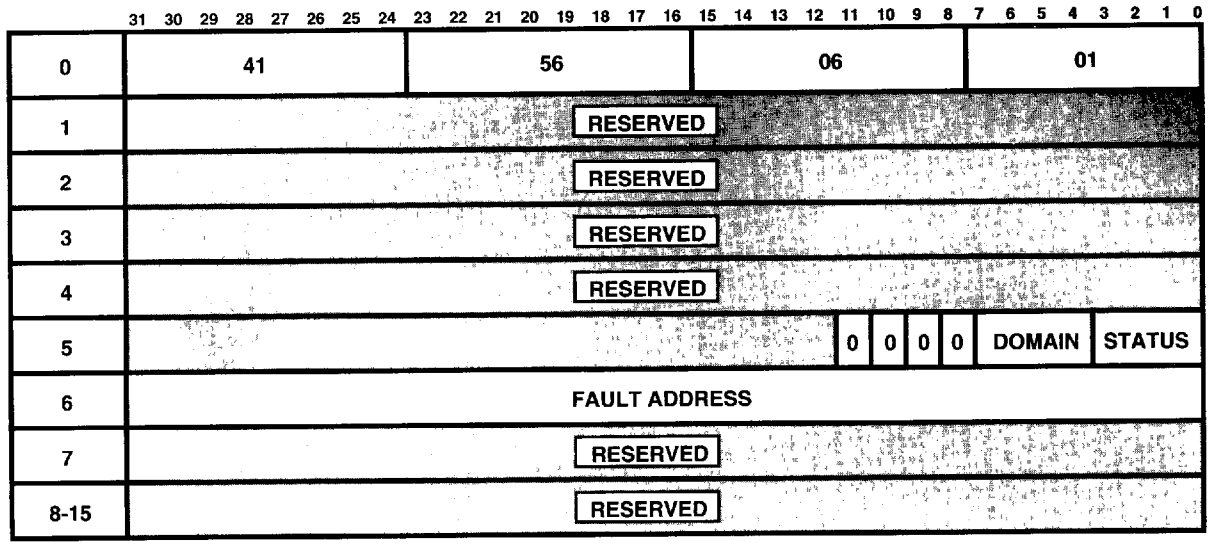



Cond = ARM condition codes
CRn = VY86C610 Register
Rd = ARM Register

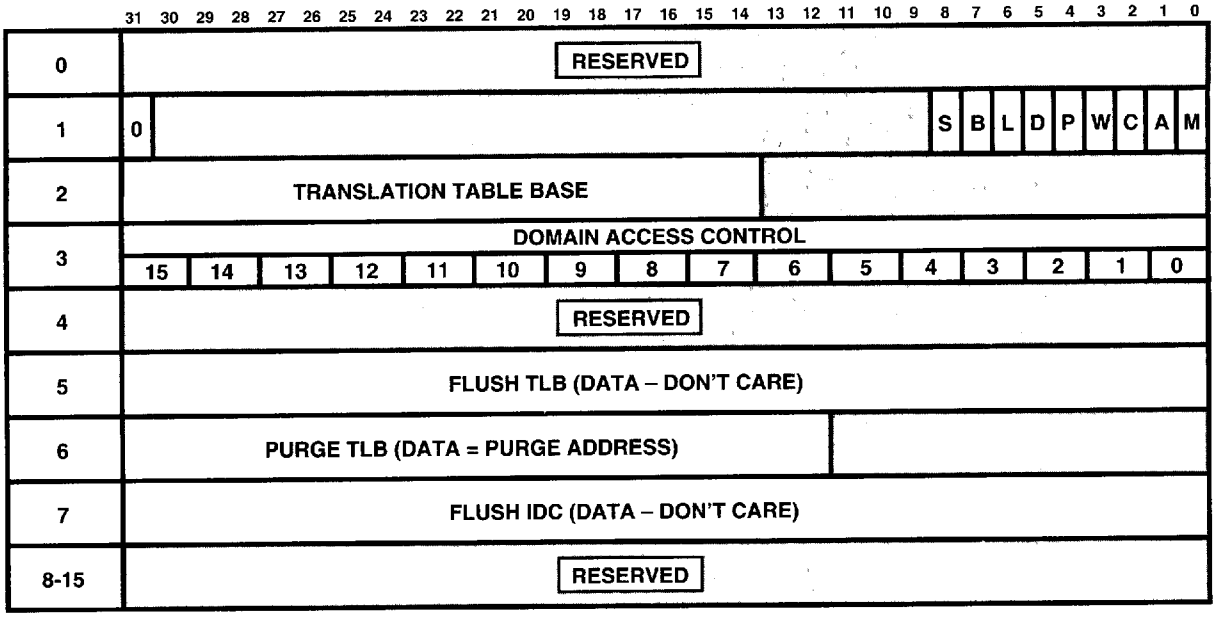
## VY86C610 REGISTERS

The VY86C610 contains registers which control the cache and MMU operation. These registers are accessed using CPRT instructions to Coprocessor #15 with the processor in a privileged mode. Only some of Registers 0-7 are valid: an access to an invalid register will not be denied, nor will an undefined instruction trap be generated, and therefore should never be carried out. An access to any of the Registers 8-15 will cause the undefined instruction trap to be taken.

### REGISTER READS

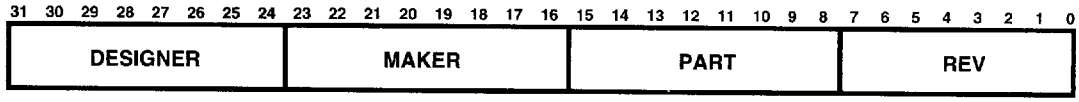



### REGISTER WRITES



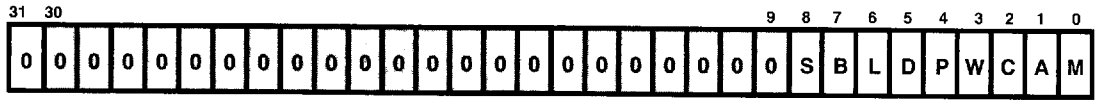
**REGISTER 0. ID**

Register 0 is a read-only identity register that returns the ARM Ltd. code for this chip.




**REGISTER 1. CONTROL**

Register 1 is a write-only register containing a number of control bits. All bits in this register are forced LOW by reset.




M Bit 0 MMU Enable/disable
- 0 – On-chip Memory Management Unit turned off
- 1 – On-chip Memory Management Unit turned on

A Bit 1 Address Fault Enable/ Disable
- 0 – Alignment fault disabled
- 1 – Alignment fault enabled

C Bit 2 Cache Enable/Disable
- 0 – Instruction/data cache turned off
- 1 – Instruction/data cache turned on

W Bit 3 Write buffer Enable/Disable
- 0 – Write buffer turned off
- 1 – Write buffer turned on

P Bit 4 ARM 32/26-Bit Program Space
- 0 – 26-bit Program Space selected
- 1 – 32-bit Program Space selected

D Bit 5 ARM 32/26-Bit Data Space
- 0 – 26-bit Data Space selected
- 1 – 32-bit Data Space selected

L Bit 6 Late Abort Timing
- 0 – Early abort mode selected
- 1 – Late abort mode selected

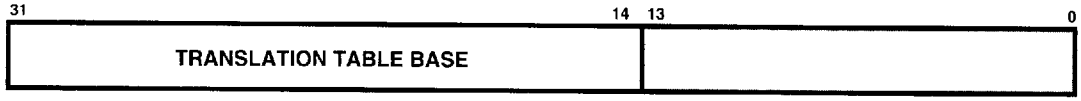
B Bit 7 Big/Little Endian
- 0 – Little-endian operation
- 1 – Big-endian operation

S Bit 8 System

This bit controls the VY86C610 permission system selection.

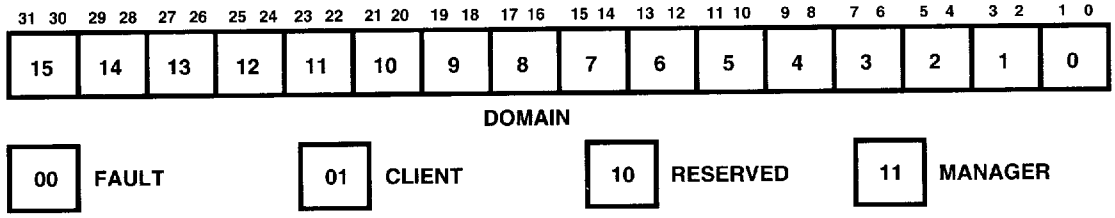
**REGISTER 2. TRANSLATION TABLE BASE**

Register 2 is a write-only register which holds the base of the currently active Level One page table.



**REGISTER 3. DOMAIN ACCESS CONTROL**

Register 3 is a write-only register that holds the current access control for domains 0 to 15. (For additional information on the MMU see page 43.)




**REGISTER 4. RESERVED**

Register 4 is Reserved. Accessing this register has no effect, but should never be attempted for normal operation.

**REGISTER 5.**
**Read: Fault Status**

Reading Register 5 returns the status of the last data fault. It is not updated for a prefetch fault. (For additional information on the MMU see page 43.) Note that only the bottom 12 bits are returned. The upper 20 bits will be the last value on the internal data bus, and therefore will have no meaning. Bits 11:8 are always returned as zero.

**Write: Translation Lookaside Buffer Flush**

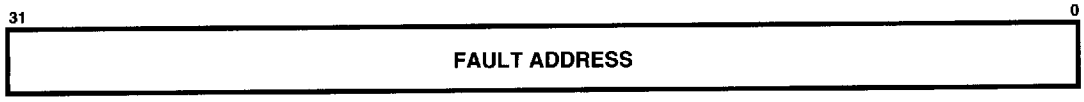
Writing Register 5 flushes the Translation Lookaside Buffer (TLB).

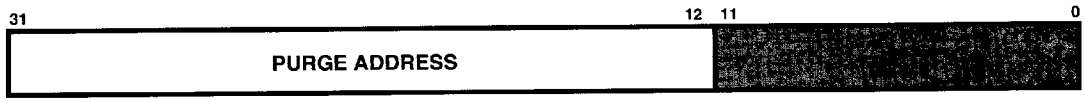

**REGISTER 6.**
**Read: Fault Address**

Reading Register 6 returns the virtual address of the last data fault.

**Write: TLB Purge**

Writing Register 6 purges the TLB. The data is treated as an address, and the TLB is searched for a corresponding page table descriptor. If a match is found, the corresponding entry is marked as invalid. This allows the page table descriptors in main memory to be updated and invalid entries in the on-chip TLB to be purged without requiring the entire TLB to be flushed.




**REGISTER 7. IDC FLUSH**

Register 7 is a write-only register. The data written to this register is discarded and the Instruction Data Cache is flushed.

**REGISTER 8-15. RESERVED**

Accessing any of these registers will cause the undefined instruction trap to be generated.

## INSTRUCTION AND DATA CACHE (IDC)

The VY86C610 contains a 4 Kbyte mixed instruction and data cache; the IDC has 256 lines of 16 bytes (four words), organized as four blocks of 64 lines (making it 64-way set associative), and uses the virtual addresses generated by the processor core. The IDC is always reloaded a line at a time (four words). It may be enabled or disabled via the VY86C610 Control Register and is disabled on RESET. The operation of the cache is further controlled by two bits: Cacheable and Updateable, which are stored in the Memory Management Page Tables. In order to use the IDC, the MMU must be enabled. The two functions may be enabled simultaneously, with a single write to the Control Register.

### CACHEABLE BIT

The Cacheable bit determines whether data being read may be placed in the IDC and used for subsequent read operations. Typically, main memory will be marked as Cacheable to improve system performance, and I/O space as non-cacheable to stop the data being stored in the VY86C610's cache. (For example, if the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of initial data held in the cache). The Cacheable bit can be configured for both pages and sections.

### UPDATEABLE BIT

The Updateable bit determines whether the data in the cache should be updated during a write operation to maintain consistency with the external memory. (In certain cases, automatic updating of cached data is not required.) For instance, when using the MEMC1a memory manager, a read operation in the address space between 0x3400000-0x3FFFFFF would access the ROMs. A write operation in the same address space would change a MEMC register, and should not affect the cached ROM data). The Updateable bit can only be configured by the Level One descriptor (i.e., an entire section or all the pages for a single Level One descriptor share the same configuration).

### IDC OPERATION

When the processor performs a read or write operation, the translation entry of that address is inspected and the state of the Cacheable and Updateable bits determines the subsequent action.

#### Cacheable Reads C=1

The cache is searched for the relevant data; if found in the cache, the data is fed to the processor using fast clock cycle (from FCLK). If the data is not found in the cache, an external memory access is initiated to read the appropriate line of data (four words) from external memory and it is stored in a pseudo-randomly chosen entry in the cache (a linefetch operation).

#### Non-Cacheable Reads C=0

The cache is not searched for the relevant data; instead an external memory access is initiated. No linefetch operation is performed, and the cache is not updated.

#### Updateable Writes U=1

An external memory access is initiated, and the cache is searched; if the cache holds a copy of the data from the address being written to, then the cache data is simultaneously updated.

#### Non-Updateable Writes U=0

An external memory access is initiated, but the cache is not searched and the contents of the cache are not affected.

### IDC VALIDITY

The IDC operates with virtual addresses, so care must be taken to ensure that its contents remain consistent with the virtual to physical mappings performed by the Memory Management Unit. If the memory mappings are changed, the IDC validity must be ensured.

#### Software IDC Flush

The entire IDC may be marked as invalid by writing to the VY86C610 IDC Flush Register (Register 7). The cache will be flushed immediately after the register is written, but note two subsequent instruction fetches may come from the cache before the register is written because the pipeline may have been filled already.

#### Doubly Mapped Space

Since the cache works with virtual addresses, it is assumed that every virtual address maps to a different physical address. If the same physical location is accessed by more than one virtual address, the cache cannot maintain consistency, since each virtual address will have a separate entry in the cache, and only one entry will be updated on a processor write operation. To avoid any cache inconsistencies, both doubly-mapped virtual addresses should be marked as uncacheable.

### READ-LOCK-WRITE

The IDC treats the Read-Lock-Write instruction as a special case. The read phase always forces a read of external memory, regardless of whether the data is contained in the cache. The write phase is treated as a normal write operation. If marked as Updateable, and the data is already in the cache, the cache will be updated. Externally, the two phases are flagged as indivisible by asserting the LOCK signal.

### IDC ENABLE/DISABLE AND RESET

The IDC is automatically disabled and flushed on RESET. Once enabled, cacheable read accesses will cause lines to be placed in the cache. If subsequently disabled, no new lines will be placed in the cache, and the cache is not searched. However, updateable write operations will continue to operate, maintaining consistency with the external memory. If the cache is subsequently re-enabled, it must be flushed if data already in the cache no longer matches that in external memory.

#### To Enable the IDC

Ensure the MMU is enabled (set Bit 0 in the Control Register).

Enable the IDC (set Bit 2 in the Control Register).[1]

#### To Disable the IDC

Disable the IDC (clear Bit 2 in the Control Register).[2]

**Notes:**

1. The MMU and IDC may be enabled simultaneously.
2. Updateable writes continue but no line fetches are performed. To fully inhibit the cache's operation, it should be disabled and then flushed to ensure it contains no valid entries.

## WRITE BUFFER (WB)

The VY86C610 Write Buffer is provided to improve system performance. It can buffer up to eight words of data, and two independent addresses. It may be enabled or disabled via the W bit (Bit 3) in the VY86C610 Control Register. The buffer is disabled and flushed on RESET. The operation of the Write Buffer is further controlled by one bit, B, or Bufferable, which is stored in the Memory Management Page Tables. In order to use the Write Buffer, the MMU must be enabled. The two functions may be enabled simultaneously with a single write to the Control Register. For a write to use the Write Buffer, both the W bit in the Control Register, and the B bit in the corresponding page table must be set.

### BUFFERABLE BIT

This bit controls whether a write operation may or may not use the Write Buffer. (Typically main memory will be bufferable and I/O space unbufferable.) The Bufferable bit can be configured for both pages and sections.

### WRITE BUFFER OPERATION

When the CPU performs a write operation, the translation entry for that address is inspected and the state of the B bit determines the subsequent action. If the Write Buffer is disabled via the VY86C610 Control Register, bufferable writes are treated in the same way as unbuffered writes.

### Bufferable Write

If the Write Buffer is enabled and the processor performs a write to a bufferable area, the data is placed in the Write Buffer at FCLK speeds and the CPU continues execution. The Write Buffer then performs the external write in parallel. If the Write Buffer is full (because there are eight words of data in the buffer, or there is no slot for the new address), then the processor is stalled until there is sufficient space in the buffer.

### Unbufferable Writes

If the Write Buffer is disabled or the CPU performs a write to an unbufferable area, the processor is stalled until the write completes externally, which may require synchronization and several external clock cycles.

### Read-Lock-Write

The write phase of a read-lock-write sequence is treated as an unbuffered write, even if it is marked as buffered. Note that a single write requires one address slot and one data slot in the write buffer; a sequential write of n words requires one address slot and n data slots. The total of eight data slots in the buffer may be used as required. For example, there could be one non-sequential write and one sequential write of seven words in the buffer, and the processor could continue as normal. A third write or an eighth word in the second write would stall the processor until the first write had completed.

### To Enable Write Buffer

Ensure the MMU is enabled (set Bit 0 in Control Register).

Enable the Write Buffer (set Bit 3 in Control Register).[1]

### To Disable Write Buffer

Disable the Write Buffer (clear Bit 3 in Control Register). [2]

**Notes:**

1. The MMU and Write Buffer may be enabled simultaneously.
2. Any writes already in the Write Buffer will complete normally.

## MEMORY MANAGEMENT UNIT

The MMU performs two primary functions: it translates virtual addresses into physical addresses, and it controls memory access permissions. The MMU hardware required to perform these functions consists of a Translation Look-aside Buffer (TLB), Access Control Logic, and Translation Table Walking Logic.

The MMU supports memory accesses based on Sections or Pages. Sections are comprised of 1 MB blocks of memory. Two different page sizes are supported: Small Pages consist of 4 Kb blocks of memory and Large Pages consist of 64 Kb blocks of memory. (Large Pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB). Additional access control mechanisms are extended within Small Pages to 1 Kb Sub-Pages and within Large Pages to 16 Kb Sub-Pages.

The MMU also supports the concept of domains – areas of memory that can be defined to possess individual access rights. The Domain Access Control Register is used to specify access rights for up to 16 separate domains.

The TLB caches 32 translated entries. During most memory accesses, the TLB provides the translation information to the access control logic.

If the TLB contains a translated entry for the virtual address, the access control logic determines whether access is permitted. If access is permitted, the MMU outputs the appropriate physical address corresponding to the virtual address. If access is not permitted, the MMU signals the CPU to abort.

If the TLB misses (it does not contain a translated entry for the virtual address), the translation table walk hardware is invoked to retrieve the translation information from a translation table in physical memory. Once retrieved, the translation information is placed into the TLB, possibly overwriting an existing value. The entry to be overwritten is chosen cyclically.

When the MMU is turned off (as occurs on RESET), the virtual address is output directly onto the physical address bus.

### MMU PROGRAM ACCESSIBLE REGISTERS

The VY86C610 Processor provides several 32-bit registers which determine the operation of the MMU. The format for these registers is shown on page 44. A brief description of the registers is provided below. Each register will be discussed in more detail within the section that describes its use.

Data is written to and read from the MMU's registers using the ARM CPU's MRC and MCR coprocessor instructions.

**Translation Table Base Register** holds the physical address of the base of the translation table maintained in main memory. Note that this base must reside on 16 Kb boundaries.

**Domain Access Control Register** consists of 16 two-bit fields, each of which defines the access permissions for one of the 16 domains (D15-D0).

**Fault Status Register** indicates the domain and type of access being attempted when an abort occurred. Bits 7:4 specify which of the 16 domains (D15-D0) was being accessed when a fault occurred. Bits 3:1 indicate the type of access being attempted. The encoding of these bits is different for internal and external faults (as indicated by Bit 0 in the register) and is shown in the table on page 56. A write to this register flushes the TLB.

**Fault Address Register** holds the virtual address of the access which was attempted when a fault occurred. A write to this register causes the data written to be treated as an address and if it is found in the TLB, the entry is marked as invalid. (This operation is known as a TLB purge). The Fault Status Register and Fault Address Register are only updated for data faults, not for prefetch faults.

### ADDRESS TRANSLATION

The MMU translates virtual addresses generated by the CPU into physical addresses to access external memory, and also derives and checks the access permission. Translation information consisting of both the address translation data and the access permission data, resides in a translation table located in physical memory. The MMU provides the logic needed to traverse this translation table, obtain the translated address, and check the access permission.

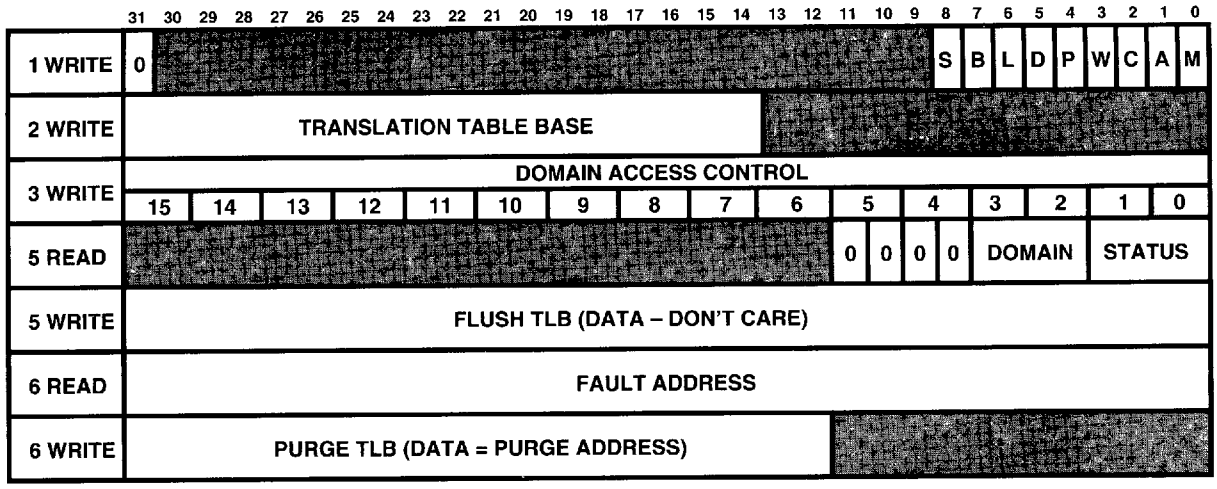
There are three routes by which the address translation (and permission check) takes place. The route taken depends on whether the address in question has been marked as a section-mapped access or a page-mapped access, and the size of page-mapped access (large pages and small pages). However, the trans-lation process always starts out in the same way (as described below), with a Level One fetch. A section-mapped access only requires a Level One fetch, but a page-mapped access also re-quires a Level Two fetch.

### TRANSLATION PROCESS

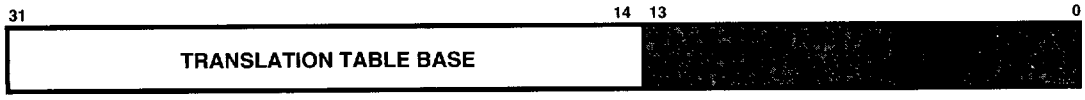
#### Translation Table Base

The translation process is initiated when the on-chip TLB does not contain an entry for the requested virtual address. The Translation Table Base (TTB) Register points to the base of a table in physical memory which contains Section and/or Page descriptors. The 14 low-order bits of the TTB Register are set to zero as illustrated on the page 44. The table must reside on a 16 Kb boundary.

## MMU REGISTER SUMMARY



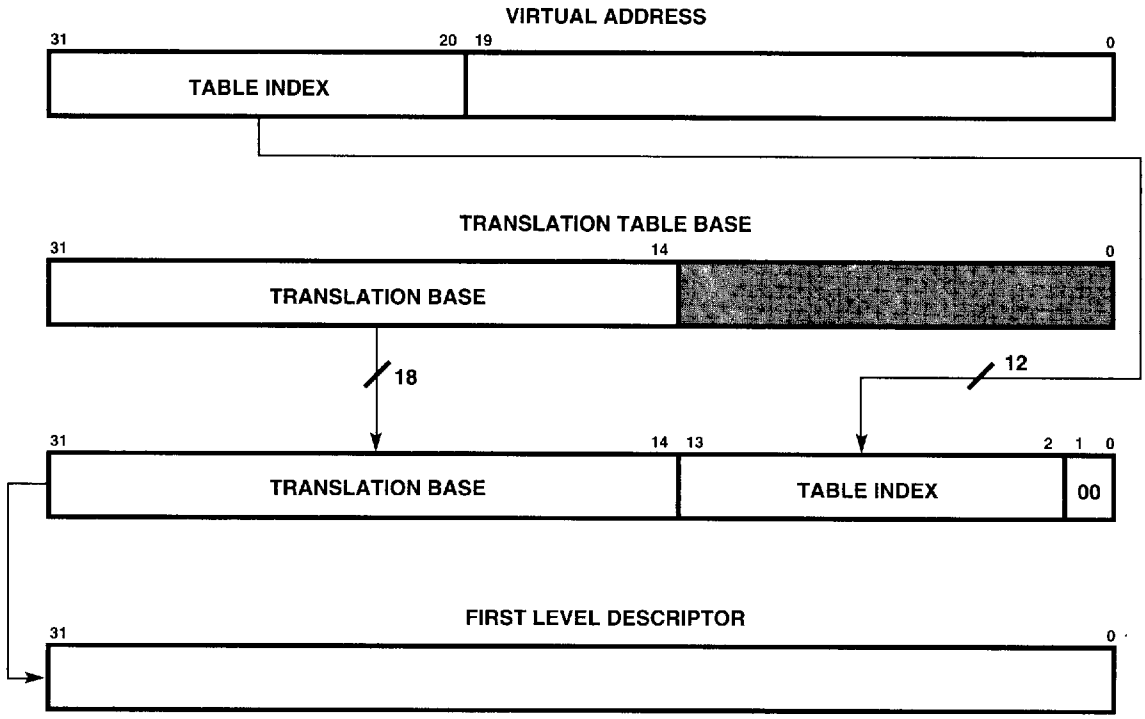

## TRANSLATION TABLE BASE REGISTER



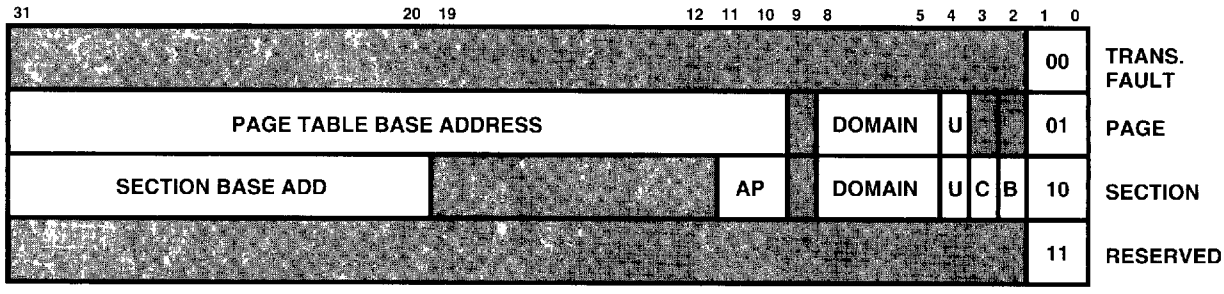
**Level One Fetch**
Bits 31:14 of the Translation Table Base register are concatenated with Bits 31:20 of the virtual address to produce a 30-bit address as illustrated below. This address selects a four-byte translation table entry which is a First Level Descriptor for either a Section or a Page (Bit 1 of the descriptor returned specifies whether it is for a Section or Page).

**Level One Descriptor**
The Level One Descriptor returned is either a Page Table Descriptor or a Section Descriptor, and its format varies accordingly. (The format of Level One Descriptors is illustrated in the figure below.)

## ACCESSING TRANSLATION TABLE FIRST LEVEL DESCRIPTORS




## LEVEL ONE DESCRIPTORS



The two least significant bits indicate the descriptor type and validity, and are interpreted as shown in the table.

**Page Table Descriptor**

- Bits 3:2 are always written as 0.
- Bit 4 Updateable: indicates that the data in the cache should be updated during a write operation to maintain consistency with external memory (if the cache is enabled).
- Bits 8:5 specify one of the 16 possible domains (held in the Domain Access Control Register) that contain the primary access controls.
- Bits 31:10 form the base for referencing the Page Table Entry. The page table index for the entry is derived from the virtual address (as illustrated on page 49).

If a Page Table Descriptor is returned from the Level One fetch, a Level Two fetch is initiated as described below.

**Section Descriptor**

Bits 4:2 (U,C, & B) control the cache- and write-buffer-related functions as follows:

U – Updateable: indicates that the data in the cache should be updated during a write operation to maintain consistency with external memory (if the cache is enabled).

C – Cacheable: indicates that data at this address will be placed in the cache (if the cache is enabled).

B – Bufferable: indicates that data at this address will be written through the write buffer (if the write buffer is enabled).

- Bits 8:5 specify one of the 16 possible domains (held in the Domain Access Control Register) that contain the primary access controls.
- Bits 11:10 (AP) specify the access permissions for this section and are interpreted as shown in table directly above. Their interpretation is dependent upon the setting of the S bit (Control Register Bit 8). Note that the Domain Access Control specifies the primary access control; the AP bits only have an effect in client mode. Refer to section on access permissions.
- Bits 19:12 are always written as 0.
- Bits 31:20 form the corresponding bits of the physical address for the 1 Mbyte section.

**INTERPRETING LEVEL ONE DESCRIPTOR BITS [1:0]**

| Value | Meaning | Notes |
|---|---|---|
| 00 | Invalid | Generates a Section Translation Fault |
| 01 | Page | Indicates that this is a Page Descriptor |
| 10 | Section | Indicates that this is a Section Descriptor |
| 11 | Reserved | Reserved for future use (currently as for invalid) |

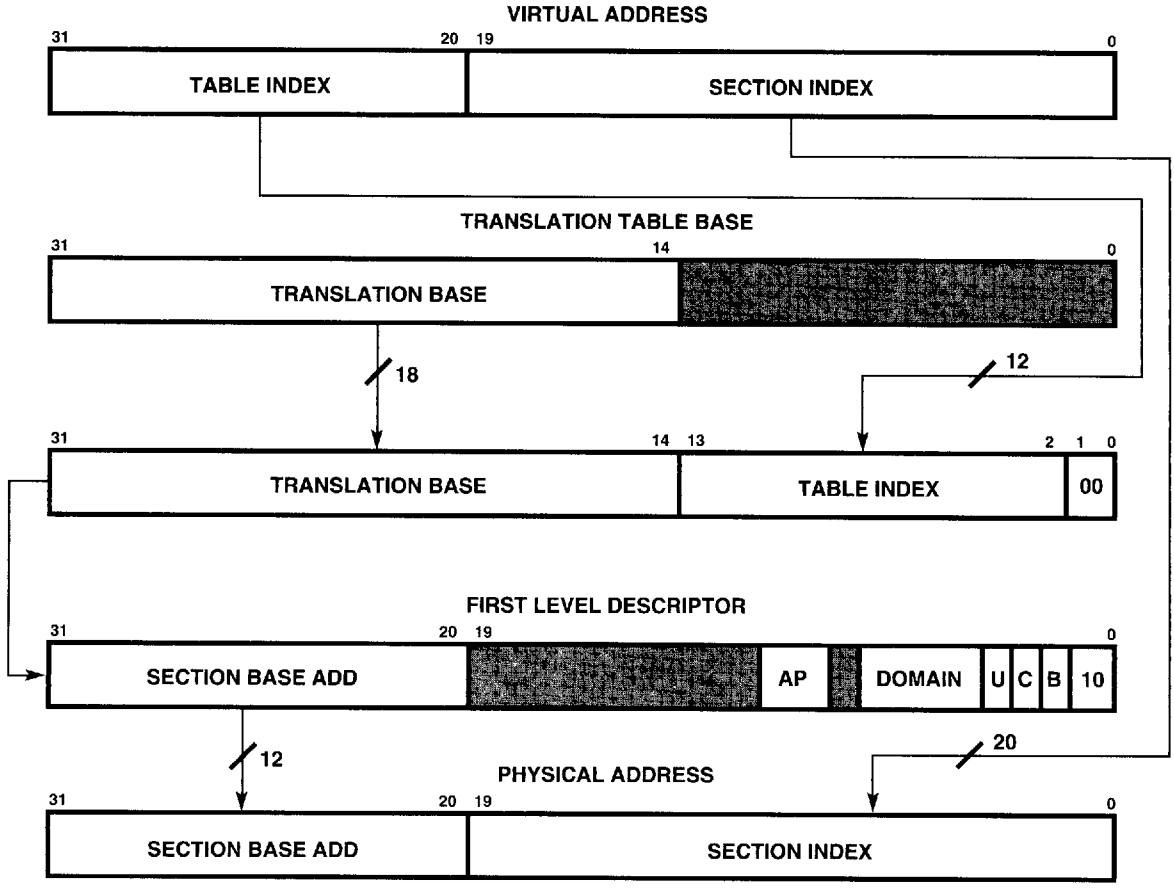
**INTERPRETING ACCESS PERMISSION (AP) BITS**

| | | Permissions | | |
|---|---|---|---|---|
| AP | S | Supervisor | User | Notes |
| 00 | 0 | No Access | No Access | Any access generates a permission fault |
| 00 | 1 | Read Only | No Access | |
| 01 | x | Read/Write | No Access | Access allowed only in Supervisor mode |
| 10 | x | Read/Write | Read Only | Writes in User mode cause permission fault |
| 11 | x | Read/Write | Read/Write | All access types permitted in both modes |

### Translating Section References

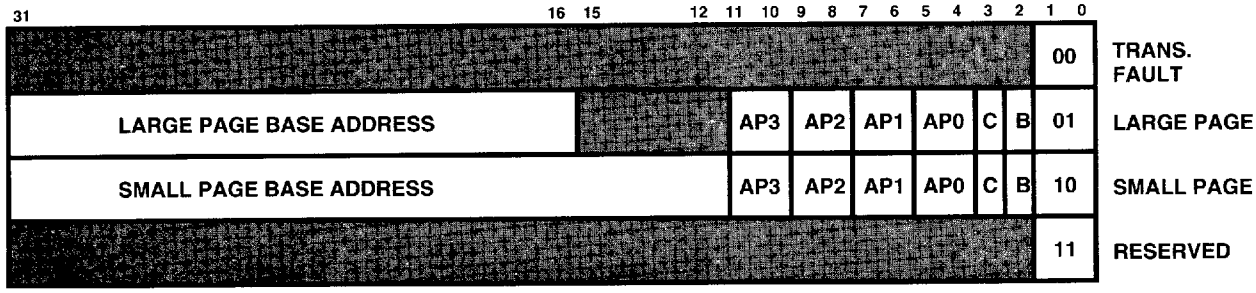
The figure below illustrates the complete Section translation sequence. Note that the access permissions contained in the Level One descriptor must be checked before the physical address is generated. The sequence for checking access permissions is described below.

**SECTION TRANSLATION**



**Level Two Descriptor**
If the Level One fetch returns a Page Table Descriptor, this provides the base address of the page table to be used. The page table is then accessed (as shown on page 49), and a Page Table Entry, or Level Two Descriptor, is returned. This, in turn, may define either a Small Page or a Large Page access. The illustration below shows the format of Level Two Descriptors.

**PAGE TABLE ENTRY (LEVEL TWO DESCRIPTOR)**




The two least-significant bits indicate the page size and validity, and are interpreted as shown in the table.

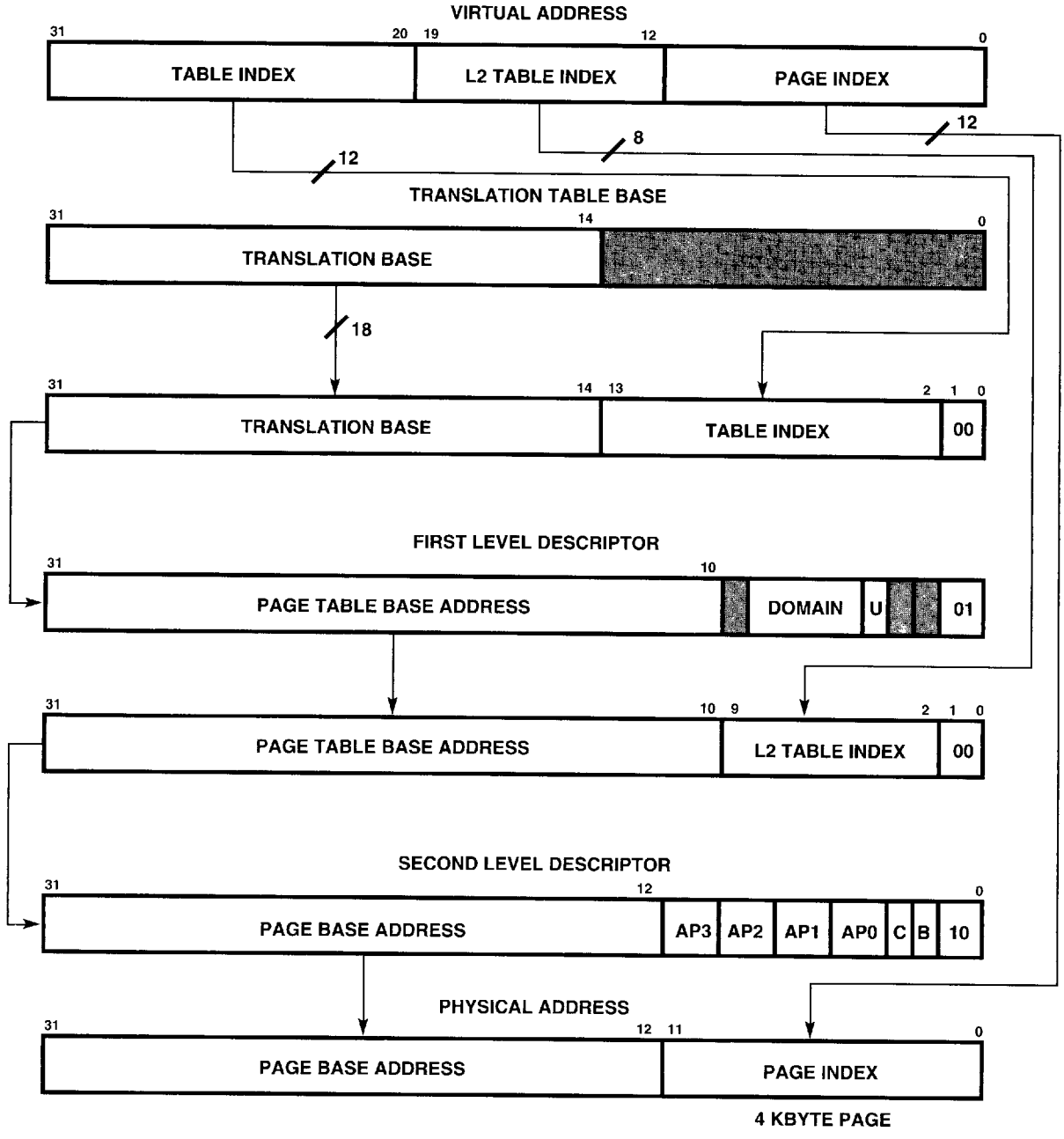
- Bit 2 C – Cacheable: indicates that data at this address will be placed in the IDC (if the cache is enabled).
- Bit 3 B – Bufferable: indicates that data at this address will be written through the Write Buffer (if the Write Buffer is enabled).
- Bits 11:4 specify the access permissions (AP3 – AP0) for the four sub-pages and interpretation of these bits is described in the table located on page 51.
- For large pages, Bits 15:12 are programmed as 0.
- Bits 31:12 (small pages) or Bits 31:16 (large pages) are used to form the corresponding bits of the physical address – the physical page number. (The page index is derived from the virtual address as illustrated on pages 49 and 50.)

**INTERPRETING PAGE TABLE ENTRY BITS 1:0**

| Value | Meaning | Notes |
|---|---|---|
| 00 | Invalid | Generates a Page Translation Fault |
| 01 | Large Page | Indicates that this is a 64 Kbyte Page |
| 10 | Small Page | Indicates that this is a 4 Kbyte Page |
| 11 | Reserved | Reserved for future use (currently as for invalid) |

**Translating Small Page References**
The illustration below shows the complete translation sequence for a 4 Kb Small Page. Page translation involves one additional step beyond that of a section translation: the Level One descriptor is the Page Table descriptor, and this is used to point to the Level Two descriptor, of a Page Table Entry. (Note that the access permissions are now contained in the Level Two descriptor and must be checked before the physical address is generated. The sequence for checking access permissions is described on page 53.)

**SMALL PAGE TRANSLATION**



**Translating Large Page References**
The figure below illustrates the complete translation sequence for a 64-Kbyte Large Page. Note that since the upper four bits of the Page Index and low-order four bits of the Page Table index of overlap, each Page Table Entry for a Large Page must be duplicated 16 times (in consecutive memory locations) in the Page Table.

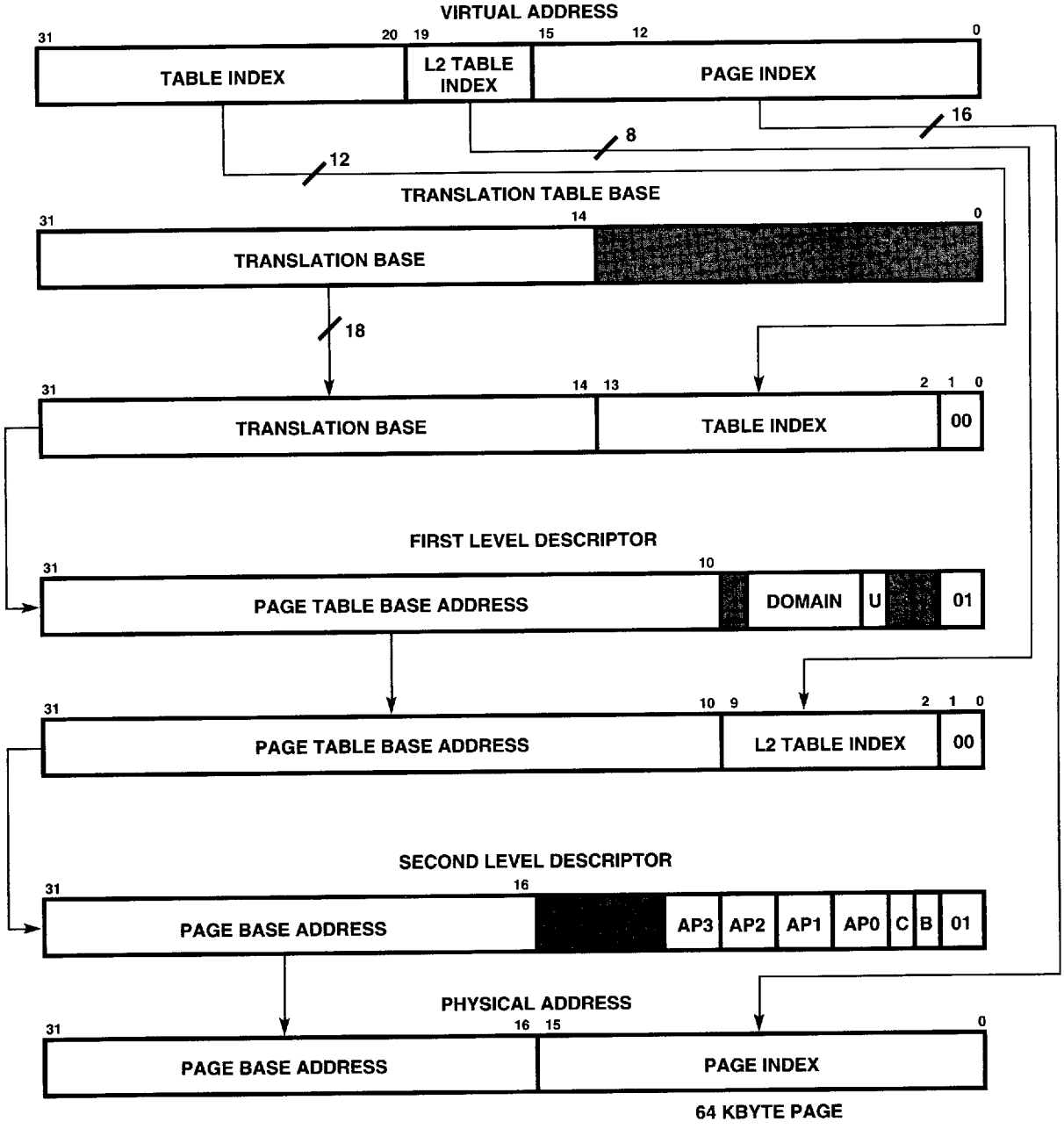
**MMU FAULTS AND CPU ABORTS**
The MMU generates four types of faults:

- Alignment Fault
- Translation Fault
- Domain Fault
- Permission Fault

In addition, an external abort may be raised on external data access.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as the result of a memory access, the MMU will abort the access and signal the fault condition to the CPU. The MMU is also capable of retaining status and address information about the abort. The CPU recognizes two types of abort: data aborts and prefetch aborts, and these are treated differently by the MMU.

**LARGE PAGE TRANSLATION**



If the MMU detects an access violation, it will do so before the external memory access takes place, and will inhibit the access. External aborts will not necessarily inhibit the external access, as described in the section on external aborts.

**Fault Address & Fault Status Registers (FAR & FSR)**

Aborts resulting from data accesses (data aborts) are acted upon by the CPU immediately, and the MMU places an encoded 4-bit value FS[3:0], along with the 4-bit encoded domain number in the Fault Status Register (FSR). In addition, the virtual processor address which caused the data abort is latched into the Fault Address Register (FAR). If an access violation simultaneously generates more than one source of abort, they are encoded in the priority given in the table below.

CPU instructions are prefetched, so prefetch aborts flag the instruction as it enters the instruction pipeline. Only when the instruction is executed does it cause an abort. An abort is not acted upon if the instruction is not used (i.e., it is branched around). As a result, instruction prefetch aborts may or may not be acted upon, and the MMU status information is not preserved for the resulting CPU abort. For a prefetch abort, the MMU does not update the FSR or FAR.

The sections that follow describe the various access permissions and controls supported by the MMU and detail how these are interpreted to generate faults.

## PRIORITY ENCODING OF FAULT STATUS[1]

| | Source | | FS [3210] | Domain [3:0] | FAR | Note |
|---|---|---|---|---|---|---|
| Highest | Write Buffer | | 00x0 | x | | 2 |
| | Bus Error | LF Section | 0100 | Valid | Valid | 3 |
| | | Page | 0110 | Valid | Valid | |
| | Bus Error | Section | 1000 | Valid | Valid | |
| | | Page | 1010 | Valid | Valid | |
| | Alignment | | 00x1 | x | Valid | |
| | Bus Error Trans | L1 | 1100 | x | Valid | |
| | | L2 | 1110 | Valid | Valid | |
| | Translation | Section | 0101 | See note | Valid | 4 |
| | | Page | 0111 | Valid | Valid | |
| | Domain | Section | 1001 | Valid | Valid | |
| | | Page | 1011 | Valid | Valid | |
| | Permission | Section | 1101 | Valid | Valid | |
| Lowest | | Page | 1111 | Valid | Valid | |

x is undefined: may read as 0 or 1

**Notes:**

1. Any abort masked by the priority encoding may be regenerated by fixing the primary abort and restarting the instruction.
2. The Write Buffer Bus Error is asynchronous and the Fault Address Register reflects the first data operation that could be aborted. This instruction must be restarted, and could generate an abort of its own.
3. This assumes that the error was flagged on Word 0 of the linefetch.
4. In fact this register will contain bits [8:5] of the Level One entry which are undefined, but would encode the domain in a valid entry.

### Access Control

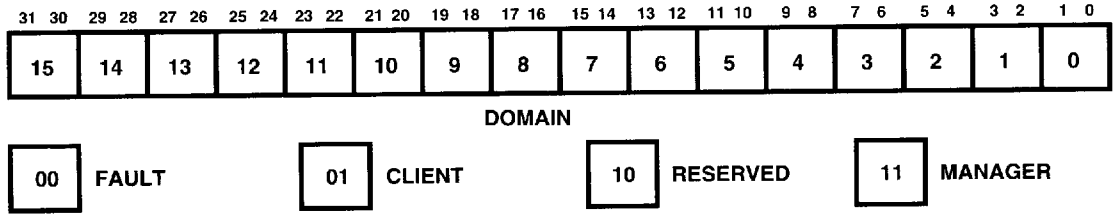
MMU accesses are primarily controlled via domains. There are 16 domains, each defined by a two-bit field. Two basic kinds of users are supported: Clients and Managers. Clients use a domain; Managers control the behavior of the domain. The domains are defined in the Domain Access Control Register. The figure below illustrates how the 32 bits of the register are allocated to define the 16 two-bit domains.

The table defines how the bits within each domain are interpreted to specify the access permissions.

**INTERPRETING ACCESS BITS IN DOMAIN ACCESS CONTROL REGISTER**

| Value | Meaning | Notes |
|---|---|---|
| 00 | No Access | Any access will generate a Domain fault |
| 01 | Client | Accesses are checked against the Access permission |
| 10 | Reserved | Reserved. Currently behaves like the no access mode |
| 11 | Manager | Accesses are NOT checked against the Access permission bits so a Permission fault cannot be generated |

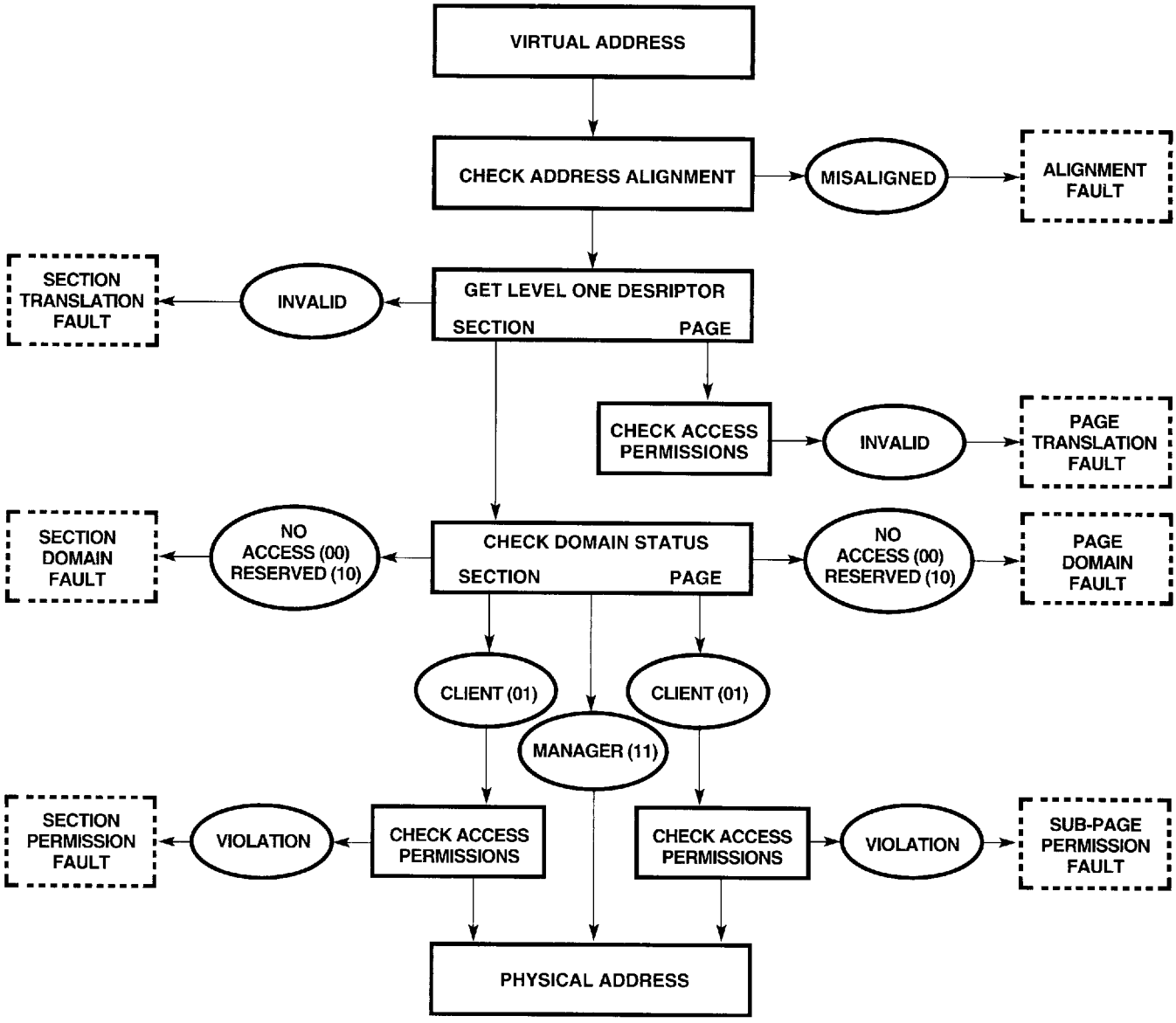
**DOMAIN ACCESS CONTROL REGISTER FORMAT**



## FAULT CHECKING SEQUENCE

The sequence by which the MMU checks for access faults is slightly different for Sections and Pages. The figure below illustrates the sequence for both types of accesses. The sections and figures that follow describe the conditions that generate each of the faults.

## SEQUENCE FOR CHECKING FAULTS



**Alignment Fault**
If Alignment Fault is enabled (Bit 1 in Control Register set), the MMU will generate an alignment fault for any data word access whose address is not word-aligned, regardless of whether the MMU is enabled or not. Alignment Fault will not be generated on any instruction fetch, nor on any byte access. Note that if the access generates an alignment fault, the access sequence will abort without reference to further permission checks.

**Translation Fault**
There are two types of Translation Fault: Section and Page.

- A Section Translation Fault is generated if the Level One descriptor is marked as invalid. This occurs if bits[1:0] of the descriptor are both 0 or both 1.
- A Page Translation Fault is generated if the Page Table Entry is marked as invalid. This occurs if bits[1:0] of the entry are both 0 or both 1.

**Domain Fault**
There are two types of Domain Fault: section and page.

In both cases, the Level One descriptor holds the four-bit Domain field that selects one of the 16 two-bit domains in the Domain Access Control Register. The two bits of the specified domain are then checked for access permissions (as detailed in the table on page 57). In the case of a section, the domain is checked once the Level One descriptor is returned. In the case of a page, the domain is checked once the Page Table Entry is returned.

If the specified access is either No Access (00), or Reserved (10), then either a Section Domain Fault or Page Domain Fault occurs.

**Permission Fault**
There are two types of Permission Fault: Section and Sub-page. The Permission Fault is checked at the same time as the Domain Fault. If the two-bit domain field returns Client (01), then the Permission access check is invoked as follows:

**Section**
If the Level One descriptor defines a section-mapped access, then the AP bits of the descriptor define whether or not the access is allowed according to the table below. Their interpretation is dependent upon the settling of the S bit (Control Register Bit 8). If the access is not allowed, then a Section Permission fault is generated.

**Sub-page**
If the Level One descriptor defines a page-mapped access, then the Level Two descriptor specifies four access permission fields (AP3...AP0), each corresponding to one quarter of the page. Hence for small pages, AP3 is selected by the top 1 Kbyte of the page, and AP0 is selected by the bottom 1 Kbyte of the page. For large pages, AP3 is selected by the top 4 Kbyte of the page, and AP0 is selected by the bottom 4 Kbyte of the page. The selected AP bits are then interpreted in exactly the same way as for a section (see table below), the only difference being that the fault generated is a Sub-page Permission Fault.

## INTERPRETING ACCESS PERMISSION (AP) BITS

| | | Permissions | | |
|---|---|---|---|---|
| AP | S | Supervisor | User | Notes |
| 00 | 0 | No Access | No Access | Any access generates a Permission fault |
| 00 | 1 | Read-Only | No Access | Supervisor read-only permitted |
| 01 | x | Read/Write | No Access | Supervisor read- or write-only permitted |
| 10 | x | Read/Write | Read-Only | Writes in User mode cause Permission faults |
| 11 | x | Read/Write | Read/Write | All access types permitted in both modes |

## EXTERNAL ABORTS

In addition to the MMU-generated aborts, the VY86C610 has an external abort pin which may be used to flag an error on an external memory access. However, some accesses aborted in this way are not restartable, so this pin must be used with great care. The following section describes the restrictions.

- Uncacheable reads
- Unbuffered writes
- Level One descriptor fetch
- Level Two descriptor fetch
- Read-Lock-Write sequence

These accesses may be aborted and restarted safely. If any of the above are aborted, the external access will cease on the following cycle. In the case of a Read-Lock-Write sequence in which the read aborts, the write will not occur.

### Cacheable Reads (Linefetches)

A Linefetch may be aborted safely provided that the abort is flagged on word 0. In this case, the IDC will not be updated or corrupted, and the access will be restartable. It is not advisable to flag an abort on any word other than word 0 of a Linefetch, as the IDC will contain a corrupt line and the instruction may not be restartable. Externally, a Linefetch that is externally aborted will continue to the end as though it had not aborted.

### Buffered Writes

Buffered writes cannot be externally aborted safely, because the processor will have moved on before the external abort is received. This class of abort is not restartable. If the system does flag this type of abort, the Fault Status Register will record the fact, but this is a non-recoverable error, and the machine must be reset. The system should be configured such that it does not do buffered writes to areas of memory which are capable of flagging an external abort. If a buffered write burst is externally aborted, the external write will continue to the end.

## INTERACTION OF THE MMU, IDC AND WRITE BUFFER

The MMU, IDC and WB may be enabled/disabled independently. However, there are only five valid combinations. There are no hardware interlocks on these restrictions, so invalid combinations will cause undefined results.

| MMU | IDC | WB |
|---|---|---|
| Off | Off | Off |
| On | Off | Off |
| On | On | Off |
| On | Off | On |
| On | On | On |

The following procedures must be observed:

### To Enable the MMU

Program the Translation Table Base and Domain Access Control Registers.

Program Level One and Level Two page tables, as required.

Enable the MMU by setting Bit 0 in the Control Register.[1]

### To Disable the MMU

Disable the WB by clearing Bit 3 in the Control Register.

Disable the IDC by clearing Bit 2 in the Control Register.

Disable the MMU by clearing Bit 0 in the Control Register.[2]

**Notes:**

1. Care must be taken if the translated address differs from the untranslated address as the two instructions following the enabling of the MMU will have been fetched using "flat translation" and enabling the MMU may be considered as a branch with delayed execution. A similar situation occurs when the MMU is disabled as illustrated in the following code sequence:

```
MOV R1, #&1
MCR 15,0,R1,0,0; Enable MMU
Fetch Flat
Fetch Flat
Fetch Translated
```

2. If the MMU is enabled, then disabled and subsequently re-enabled, the contents of the TLB will have been reserved. If these are now invalid, the TLB should be flushed before re-enabling the MMU. Disabling of all three functions may be done simultaneously.

## VY86C610 BUS INTERFACE

### OVERVIEW

The VY86C610 bus interface has two distinct modes of operation: synchronous and asynchronous. The mode is configured by tying the SNA pin either HIGH or LOW. The memory interface is always clocked by MCLK, and the core is clocked by a combination of FCLK and MCLK. The two modes differ in the relationship between FCLK and MCLK: in asynchronous mode, (SNA tied LOW), the two clocks may be completely asynchronous of unrelated frequency; in synchronous mode (SNA tied HIGH), MCLK may only make transitions on the falling edge of FCLK. In systems where such a relationship between the two clocks exist, the synchronization penalty can be removed by selecting synchronous operation.

### ASYNCHRONOUS MODE

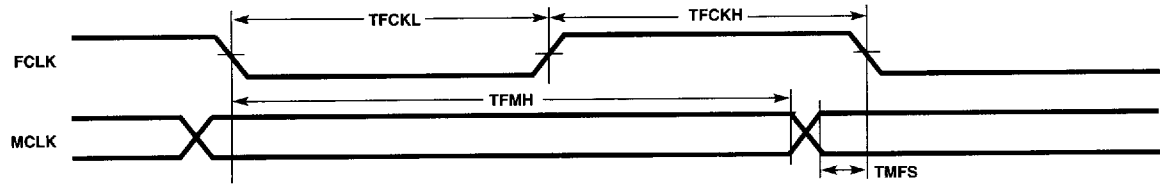
***Not supported at this time.***
***Please consult factory.***

### SYNCHRONOUS MODE

In this mode, there is a tightly defined relationship between FCLK and MCLK. MCLK may only make transitions on the falling edge of FCLK. An amount of jitter between the two clocks is permitted, but MCLK must not be later than FCLK. MCLK may lead FCLK by up to the difference in MCLK and FCLK periods without affecting the critical paths.

### RELATIONSHIP BETWEEN FCLK & MCLK

| Symbol | Parameter | Min | Typ | Max | Units | Note |
|---|---|---|---|---|---|---|
| TFCKL | FCLK LOW Time | 20 | 10 | | ns | 1 |
| TFCKH | FCLK HIGH Time | 20 | 14 | | ns | 1 |
| TFMH | FCLK – MCLK Hold Time | 25 | - | | ns | 2, 3 |
| TMFS | MCLK – FCLK Set-up | 0 | - | | ns | 2, 3 |




**Notes:**

1. FCLK timings measured at 50% of Vdd.
2. This parameter is only valid in synchronous mode (when the pin SNA is tied to Vdd). In asynchronous mode, there need be no relationship between FCLK and MCLK.Current VY86C610 devices are synchronous only
3. MCLKf may lead FCLKf by 1/2 FCLK period

## CYCLE SPEED

The speed of the memory bus interface may be controlled in two ways:

- The LOW and HIGH phases of MCLK may be stretched.
- The NWAIT pin can be used to insert entire MCLK cycles into any access. When LOW, this signal disables MCLK in the processor.

## CYCLE TYPES

The VY86C610 can perform many different bus cycles, and they may be combined in any order.

- Idle
- Line fetch
- Unbuffered Write
- Uncacheable Read
- Buffered Write
- Translation Level One Fetch
- Translation Level Two Fetch
- Read-Lock-Write

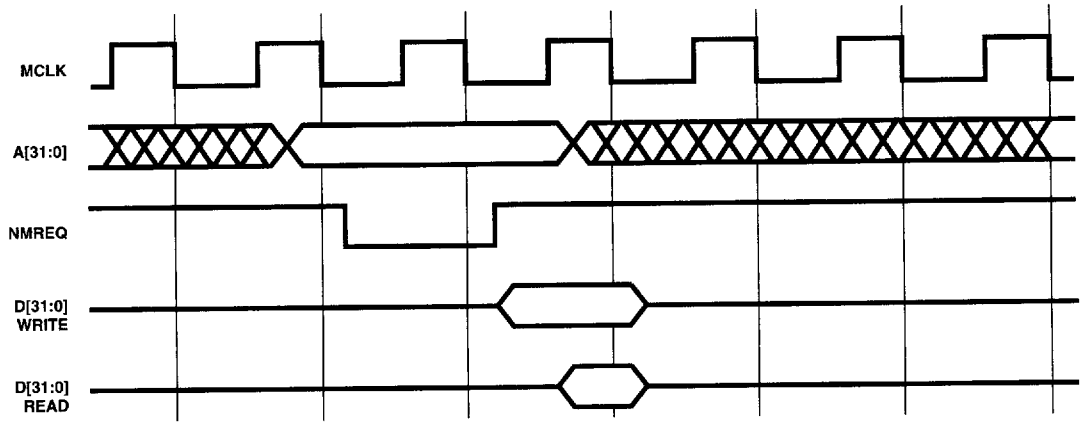
Linefetch, Unbuffered Writes, Uncacheable Reads and Buffered Writes all have very similar bus cycles. They differ only in the number of idle cycles at the end, that in turn depends on what cycle follows. The following table defines this for each bus cycle. The diagrams give examples of typical sequences.

## CYCLE TYPE DETAILS

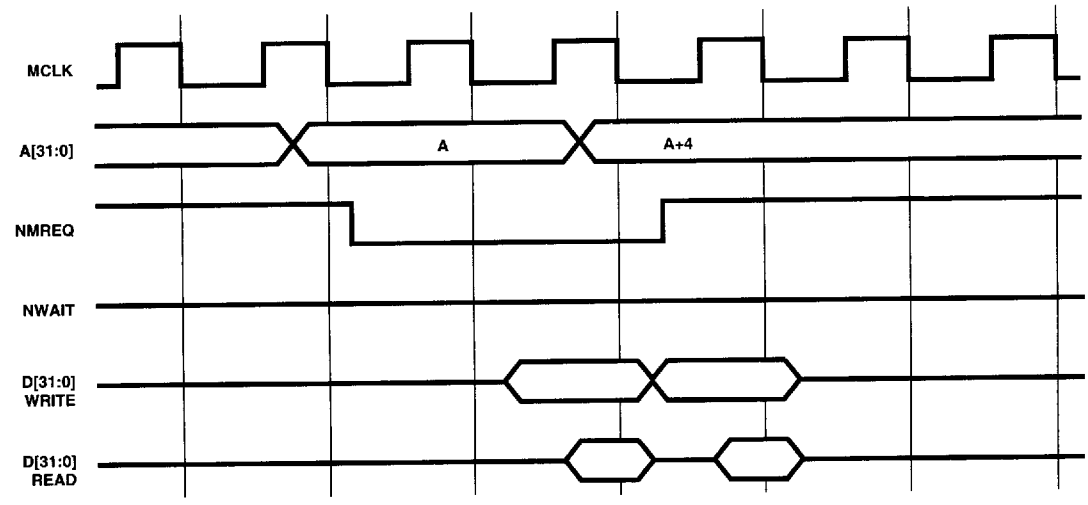
| | A[31:0] | NRW | NMREQ | D[31:0] | |
|---|---|---|---|---|---|
| IDLE | OLD | OLD | 1 | | |
| | | | | | |
| LINEFETCh | R | A | I | | |
| | R | A | M | | |
| | R | A+4 | M | D | |
| | R | A+8 | M | D | |
| | R | A+C | M | D | |
| | R | A+C | I | D | |
| | | | | | |
| ONE WORD | R/W | A | I | | UNCACHEABLE READ/ UNBUFFERED WRITE |
| | R/W | A | M | D | |
| | R/W | A | I | | |
| | | | | | |
| MANY WORD START | R/W | A | I | | |
| | R/W | A | M | | |
| | | | | D | |
| REPEAT | R/W | A+4 | M | | |
| | | | | D | |
| END | R/W | A+4 | M | | |
| | R/W | A+4 | I | D | |
| | | | | | |
| BUFFERED WRITE | W | A | I | | |
| | R | A | M | | |
| | | | | D | |
| MORE | W | A | M | | |
| | | | | D | |
| LEVEL ONE OR TWO | R | A | I | | |
| | R | A | M | | |
| | R | A | I | D | |
| | | | | | |
| READ PHASE | R | aL | I | | |
| | R | aL | M | | |
| | R | aL | I | D | |
| | | | | | |
| WRITE PHASE UNBUFFERED | W | aL | I | | READ-LOCKED-WRITE |
| | W | aL | M | | |
| | W | aL | I | D | |
| | | | | | |
| WRITE PHASE BUFFERED | W | aL | I | | |
| | W | aL | M | | |
| | | | | D | |
| WRITE PHASE ABORTED | W | aL | I | | |
| | W | aL | I | | |
| | | | | | |

## EXAMPLE CYCLES

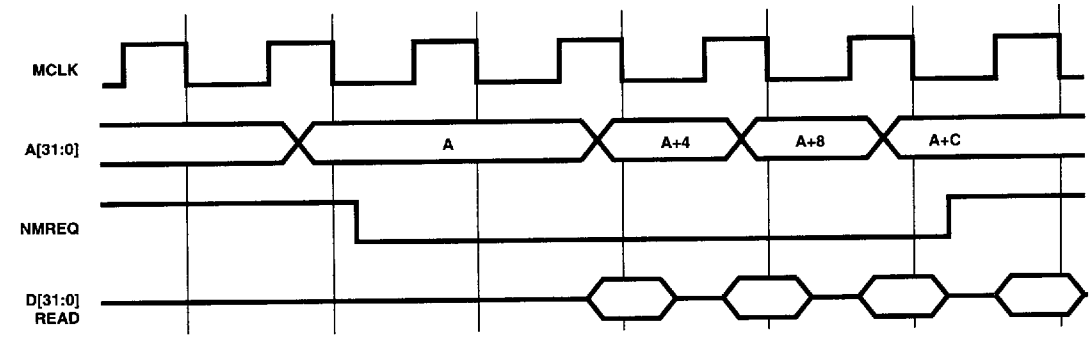
### ONE WORD READ OR WRITE



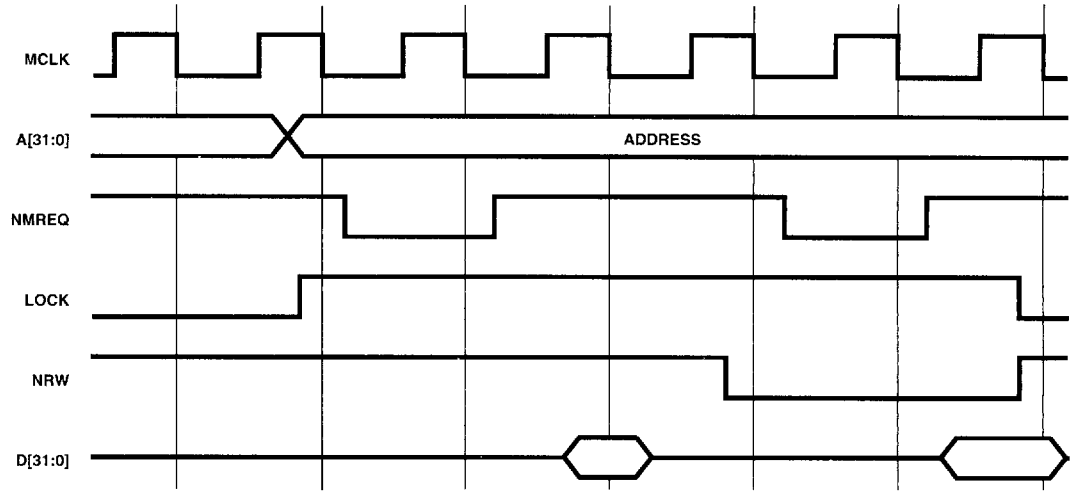

### TWO WORD WRITE OR READ



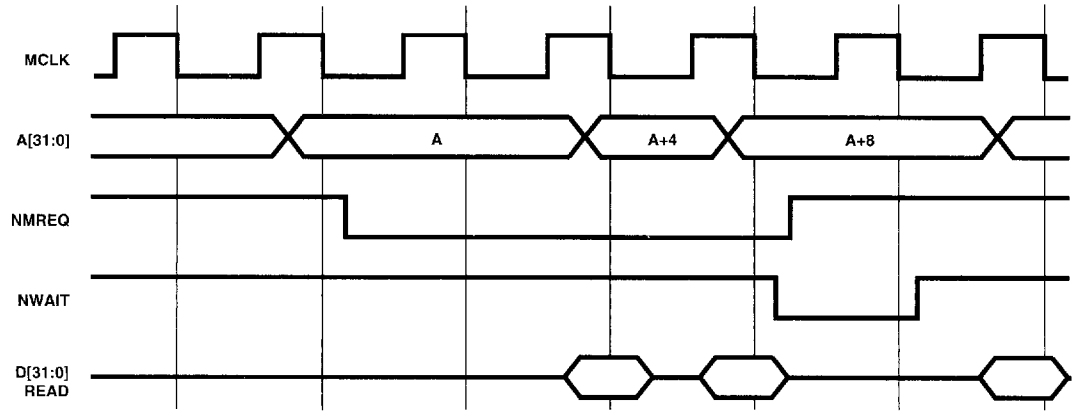

### LINEFETCH



### READ-LOCKED-WRITE

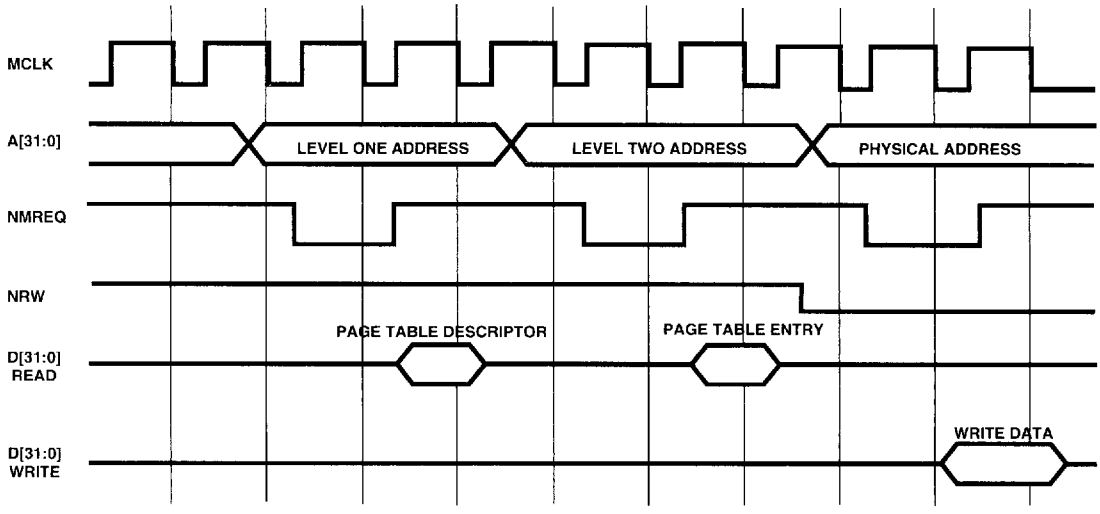



### USE OF NWAIT PIN TO STRETCH A CYCLE




In the example above, the NWAIT pin has been used to introduce an extra MCLK cycle to ease the system timing. This allows all control signals to be sampled on the rising edges of MCLK. More sophisticated systems may sample control signals on both rising and falling edges of MCLK to improve performance.

### TRANSLATION TABLE-WALKING SEQUENCE (WRITE)



## BOUNDARY SCAN TEST INTERFACE

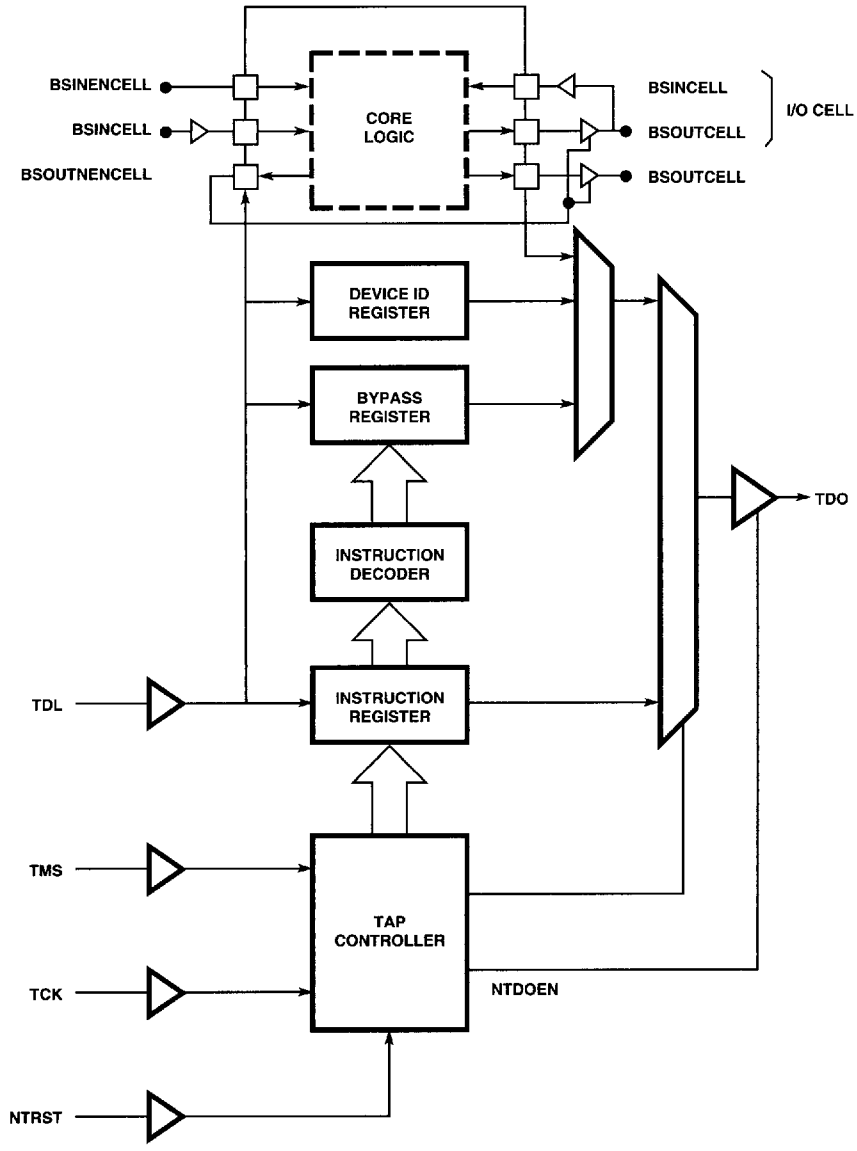
The Boundary-Scan Interface conforms to the IEEE Std. 1149.1 – 1990, Standard Test Access Port and Boundary-Scan Architecture (please refer to this document for an explanation of the terms used in this section and for a description of the TAP controller states).

### INSTRUCTION REGISTER

The Instruction Register is four bits in length. There is no parity bit.

The fixed value loaded into the instruction register during the *CAPTURE-IR* controller state is: 0001

## BOUNDARY SCAN FUNCTIONAL DIAGRAM




### PUBLIC INSTRUCTIONS

The following public instructions are supported:

| Instruction | Binary Code |
|---|---|
| BYPASS | 1111 |
| SAMPLE/PRELOAD | 0011 |
| EXTEST | 0000 |
| INTEST | 1100 |
| IDCODE | 1110 |
| HI-Z | 0111 |
| CLAMP | 0101 |
| CLAMPZ | 1001 |

In the descriptions that follow, TDI and TMS are sampled on the rising edge of TCK and all output transitions on TDO occur as a result of the falling edge of TCK.

### BYPASS (1111)

The Bypass instruction connects a 1- bit shift register (the Bypass register) between TDI and TDO.

When the Bypass instruction is loaded into the instruction register, all the boundary-scan cells are placed in their normal (system) mode of operation. This instruction has no effect on the system pins.

In the CAPTURE-DR state, a Logic 0 is captured by the Bypass register. In the SHIFT-DR state, test data is shifted into the Bypass register via TDI and out via TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The Bypass register is not affected in the UPDATE-DR state.

### SAMPLE/PRELOAD (0011)

The BS (Boundary-Scan) register is placed in test mode by the SAMPLE/PRELOAD instruction. The SAMPLE/PRELOAD instruction connects the BS register between TDI and TDO.

When the instruction register is loaded with the SAMPLE/PRELOAD instruction, all the boundary-scan cells are placed in their normal (system) mode of operation.

In the CAPTURE-DR state, a snapshot of the signals at the boundary-scan cells is taken on the rising edge of TCK. Normal system operation is unaffected. In the SHIFT-DR state, the sampled test data is shifted out of the BS register via the TDO pin, while new data is shifted in via the TDI pin to preload the BS register parallel input latch. In the UPDATE-DR state, the preloaded data is transferred into the BS register parallel output latch. Note that this data is not applied to the system logic or system pins while the SAMPLE/PRELOAD instruction is active. This instruction should be used to preload the boundary-scan register with known data prior to selecting the INTEST or EXTEST instructions (see table on page 62 for appropriate guard values to be used for each boundary-scan cell).

**EXTEST (0000)**
The BS register is placed in test mode by the EXTEST instruction. The EXTEST instruction connects the BS register between TDI and TDO.

When the instruction register is loaded with the EXTEST instruction, all the boundary-scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, inputs from the system pins and outputs from the boundary-scan output cells to the system pins are captured by the boundary-scan cells. In the SHIFT-DR state, the previously captured test data is shifted out of the BS register via the TDO pin, while new test data is shifted in via the TDI pin to the BS register parallel input latch. In the UPDATE-DR state, the new test data is transferred into the BS register parallel output latch. Note that this data is applied immediately to the system logic and system pins. The first EXTEST vector should be clocked into the BS register, using the SAMPLE/PRELOAD instruction, prior to selecting INTEST to ensure that known data is applied to the system logic.

**INTEST (1100)**
The BS register is placed in test mode by the INTEST instruction. The INTEST instruction connects the BS register between TDI and TDO.

When the instruction register is loaded with the INTEST instruction, all the boundary-scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, the inverse of the data supplied to the core logic from input boundary-scan cells is captured, while the true value of the data that is output from the core logic to output boundary-scan cells is captured. In the SHIFT-DR state, the previously captured test data is shifted out of the BS register via the TDO pin, while new test data is shifted in via the TDI pin to the BS register parallel input latch. In the UPDATE-DR state, the new test data is transferred into the BS register parallel output latch. Note that this data is applied immediately to the system logic and system pins. The first INTEST vector should be clocked into the BS register, using the SAMPLE/PRELOAD instruction, prior to selecting INTEST to ensure that known data is applied to the system logic.

Single-step operation is possible using the INTEST instruction.

**IDCODE (1110)**
The IDCODE instruction connects the device identification register (or ID register) between TDI and TDO. The ID register is a 32-bit register that allows the manufacturer, part number and version of a component to be determined through the TAP.

When the instruction register is loaded with the IDCODE instruction, all the boundary-scan cells are placed in their normal (system) mode of operation.

In the CAPTURE-DR state, the device identification code (specified at the end of this section) is captured by the ID register. In the SHIFT-DR state, the previously captured device identification code is shifted out of the ID register via the TDO pin, while data is shifted in via the TDI pin into the ID register. In the UPDATE-DR state, the ID register is unaffected.

**HI-Z (0111)**
The HI-Z instruction connects a 1-bit shift register (the Bypass register) between TDI and TDO.

When the HI-Z instruction is loaded into the instruction register, all outputs are placed in an inactive drive state.

In the CAPTURE-DR state, a logic 0 is captured by the bypass register. In the SHIFT-DR state, test data is shifted into the bypass register via TDI and out via TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The Bypass register is not affected in the UPDATE-DR state.

**CLAMP (0101)**
The CLAMP instruction connects a 1-bit shift register (the Bypass register) between TDI and TDO.

When the CLAMP instruction is loaded into the instruction register, the state of all output signals are defined by the values previously loaded into the BS register. A guarding pattern (specified for this device at the end of this section) should be pre-loaded into the BS register using the SAMPLE/PRELOAD instruction, prior to selecting the CLAMP instruction.

In the CAPTURE-DR state, a logic 0 is captured by the Bypass register. In the SHIFT-DR state, test data is shifted into the Bypass register via TDI and out via TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The Bypass register is not affected in the UPDATE-DR state.

**CLAMPZ (1001)**
The CLAMPZ instruction connects a 1- bit shift register (the Bypass register) between TDI and TDO.

When the CLAMPZ instruction is loaded into the instruction register, all outputs are placed in an inactive drive state, but the data supplied to the disabled output drivers is derived from the boundary scan cells. The purpose of this instruction is to ensure that during production testing, each output driver can be disabled when its data input is either a 0 or a 1.

In the CAPTURE-DR state, a logic 0 is captured by the Bypass register. In the SHIFT-DR state, test data is shifted into the Bypass register via TDI and out via TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The Bypass register is not affected in the UPDATE-DR state.

## TEST DATA REGISTERS

### BYPASS REGISTER

This is a single-bit register which can be selected as the path between TDI and TDO to allow the device to be bypassed during boundary-scan testing.

When the Bypass instruction is the current instruction in the instruction register, serial data is transferred from TDI to TDO in the SHIFT-DR state with a delay of one TCK cycle.

There is no parallel output from the Bypass register.

A logic 0 is loaded from the parallel input of the Bypass register in the CAPTURE-DR state.

### VY86C610 DEVICE IDENTIFICATION REGISTER

This register is used to read the 32-bit device identification code.

When the IDCODE instruction is current, the ID register is selected as the serial path between TDI and TDO.

There is no parallel output from the ID register.

The 32-bit device identification code is loaded into the ID register from its parallel inputs during the CAPTURE-DR state.

**MANUFACTURER'S IDENTIFICATION CODE:**
000 0101 0111

**PART NUMBER CODE:**
0000 0000 0100 0110

**VERSION CODE:**
0000

The complete 32-bit device identification code for the VY86C610 is shown in the diagram below.

### BOUNDARY SCAN (BS) REGISTER

The BS register consists of a serially-connected set of cells around the periphery of the device, at the interface between the system (or core) logic, and the system input/output pads. This register can be used to isolate the system logic from the pins and then apply tests to the system logic, or conversely to isolate the pins from the system logic and then drive or monitor the system pins.

The BS register is selected as the register to be connected between TDI and TDO only during the SAMPLE/PRELOAD, EXTEST and INTEST instructions. Values in the BS register are used, but are not changed, during the CLAMP and CLAMPZ instructions.

In the normal (system) mode of operation, straight-through connections between the system logic and pins are maintained and normal system operation is unaffected.

In TEST mode (i.e., when either EXTEST or INTEST is the currently selected instruction), values can be applied to the system logic or output pins independently of the actual values on the input pins and system logic outputs respectively. Additional boundary-scan cells are interposed in the scan chain in order to control the enabling of three-stateable buses.

### VY86C610 DEVICE IDENTIFICATION REGISTER

| 31 30 29 28 | 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 | 11 10 9 8 7 6 5 4 3 2 1 | 0 |
|---|---|---|---|
| VERSION | PART NUMBER | MANUFACTURER IDENTITY | |
| 0000 | 0000 0000 0100 0011 | 0000 0100 001 | 1 |

**IN HEXADECIMAL: 00043057**

The correspondence between system pins and boundary-scan cells, system direction controls and system output enables, is shown below. The cells are listed in the order in which they are connected in the BS register, starting with the cell closest to TDI. All outputs are three-state outputs. All BS register cells at input pins can apply tests to the on-chip system logic. The EXTEST guard values specified in the table below should be clocked into the BS register (using the SAMPLE/PRELOAD instruction) before the EXTEST instruction is selected, to ensure that known data is applied to the system logic during the test. The INTEST guard values shown in the table below should be clocked into the BS register (using the SAMPLE/PRELOAD instruction) before the INTEST instruction is selected to ensure that all outputs are disabled. These guard values should also be used when new EXTEST or INTEST ventors are clocked into the BS register.

**OUTPUT ENABLE BOUNDARY-SCAN CELLS**

The BS register cells NENDOUT, NENCPDH, NENCPDLO, NABE, NCBE, NCPE, and NMSE control the output drivers of three-state outputs as shown in the table on page 62. In the case of type OUTEN0 enable cells (e.g., NENDOUT), loading a 1 into the cell will place the associated drivers into the three-state condition, while in the case of type INEN1 enable cells (e.g., NABE), loading a 0 into the cell will three-state the associated drivers.

If the on-chip system logic causes the drivers controlled by NENDOUT, for example, to be three-state ( i.e., by driving the signal NENDOUT HIGH), then a 1 will be observed on this cell if the SAMPLE/PRELOAD or INTEST instructions are active.

To put all the VY86C610 three-state outputs into their high impedance states, a Logic 1 should be clocked into the output enable boundary-scan cells NENDOUT, NENCPDHI and NENCPDLO and a Logic 0 should be clocked into NABE, NCBE, NCPE, and NMSE. Alternatively, the HI-Z instruction can be used.

**SINGLE-STEP OPERATION**

The VY86C610 is a static design and there is no minimum clock speed. It can therefore be single-stepped while the INTEST instruction is selected. This can be achieved by serializing a parallel stimulus and clocking the resulting serial vectors into the BS register. When the BS register is updated, new test stimuli are applied to the core logic inputs; the effect of these stimuli can then be observed on the core logic outputs by capturing them in the BS register.

## BOUNDARY SCAN TABLE

| No. | Cell Name | Pin | Type | Output Enable BS Cell | INTEST Guard Value | EXTEST/CLAMP Guard Value |
|---|---|---|---|---|---|---|
| 1 | DIN23 | D[23] | IN | – | | |
| 2 | DOUT23 | D[23] | OUT | NENDOUT | | |
| 3 | DIN22 | D[22] | IN | – | | |
| 4 | DOUT22 | D[22] | OUT | NENDOUT | | |
| 5 | DIN21 | D[21] | IN | – | | |
| 6 | DOUT21 | D[21] | OUT | NENDOUT | | |
| 7 | DIN20 | D[20] | IN | – | | |
| 8 | DOUT20 | D[20] | OUT | NENDOUT | | |
| 9 | DIN94 | D[19] | IN | – | | |
| 10 | DOUT19 | D[19] | OUT | NENDOUT | | |
| 11 | DIN18 | D[18] | IN | – | | |
| 12 | DOUT18 | D[18] | OUT | NENDOUT | | |
| 13 | DIN17 | D[17] | IN | – | | |
| 14 | DOUT17 | D[17] | OUT | NENDOUT | | |
| 15 | DIN16 | D[16] | IN | – | | |
| 16 | DOUT16 | D[16] | OUT | NENDOUT | | |
| 17 | DIN15 | D[15] | IN | – | | |
| 18 | DOUT15 | D[15] | OUT | NENDOUT | | |
| 19 | DIN14 | D[14] | IN | – | | |
| 20 | DOUT14 | D[14] | OUT | NENDOUT | | |

## BOUNDARY SCAN TABLE (Cont.)

| No. | Cell Name | Pin | Type | Output Enable BS Cell | INTEST Guard Value | EXTEST/CLAMP Guard Value |
|---|---|---|---|---|---|---|
| 21 | DIN13 | D[13] | IN | – | | |
| 22 | DOUT13 | D[13] | OUT | NENDOUT | | |
| 23 | DIN12 | D[21] | IN | – | | |
| 24 | DOUT12 | D[12] | OUT | NENDOUT | | |
| 25 | DIN11 | D[11] | IN | – | | |
| 26 | DOUT11 | D[11] | OUT | NENDOUT | | |
| 27 | DIN10 | D[10] | IN | – | | |
| 28 | DOUT10 | D[10] | OUT | NENDOUT | | |
| 29 | DIN9 | D[9] | IN | – | | |
| 30 | DOUT9 | D[9] | OUT | NENDOUT | | |
| 31 | NENDOUT | – | OUTEN0 | – | | |
| 32 | DIN8 | D[8] | IN | – | | |
| 33 | DOUT8 | D[8] | OUT | NENDOUT | | |
| 34 | DIN7 | D[7] | IN | – | | |
| 35 | DOUT7 | D[7] | OUT | NENDOUT | | |
| 36 | DIN6 | D[6] | IN | – | | |
| 37 | DOUT6 | D[6] | OUT | NENDOUT | | |
| 38 | DIN5 | D[5] | IN | – | | |
| 39 | DOUT5 | D[5] | OUT | NENDOUT | | |
| 40 | DIN4 | D[4] | IN | – | | |
| 41 | DOUT4 | D[4] | OUT | NENDOUT | | |
| 42 | DIN3 | D[3] | IN | – | | |
| 43 | DOUT3 | D[3] | OUT | NENDOUT | | |
| 44 | DIN2 | D[2] | IN | – | | |
| 45 | DOUT2 | D[2] | OUT | NENDOUT | | |
| 46 | DIN1 | D[1] | IN | – | | |
| 47 | DOUT1 | D[1] | OUT | NENDOUT | | |
| 48 | DIN0 | D[0] | IN | – | | |
| 49 | DOUT0 | D[0] | OUT | NENDOUT | | |
| 50 | DBE | DBE | IN | – | | |
| 51 | SEQ | SEQ | OUT | NMSE | | |
| 52 | NMREQ | NMREQ | OUT | NMSE | | |
| 53 | NMSE | MSE | INEN1 | – | 0 | |
| 54 | ANA | SNA | IN | – | | |
| 55 | NWAIT | NWAIT | IN | – | | |
| 56 | MCLK | MCLK | IN | – | | 0 |
| 57 | FCLK | FCLK | OUT | – | | 0 |
| 58 | ABORT | ABORT | IN | – | | |
| 59 | NRESET | NRESET | IN | – | | |
| 60 | TESTIN[16] | TESTIN[16] | IN | – | | 0 |

## BOUNDARY SCAN TABLE (Cont.)

| No. | Cell Name | Pin | Type | Output Enable BS Cell | INTEST Guard Value | EXTEST/CLAMP Guard Value |
|---|---|---|---|---|---|---|
| 61 | TESTOUT[2] | TESTOUT[2] | OUT | NTBE | | |
| 62 | TESTOUT[1] | TESTOUT[1] | OUT | NTBE | | |
| 63 | TESTOUT[0] | TESTOUT[0] | OUT | NTBE | | |
| 64 | NIRQ | NIRQ | IN | – | | |
| 65 | NFIG | NFIG | IN | – | | |
| 66 | TESTIN[7] | TESTIN[7] | IN | – | | 0 |
| 67 | TESTIN[6] | TESTIN[6] | IN | – | | 0 |
| 68 | TESTIN[5] | TESTIN[5] | IN | – | | 0 |
| 69 | TESTIN[4] | TESTIN[4] | IN | – | | 0 |
| 70 | TESTIN[3] | TESTIN[3] | IN | – | | 0 |
| 71 | TESTIN[2] | TESTIN[2] | IN | – | | 0 |
| 72 | TESTIN[1] | TESTIN[1] | IN | – | | 0 |
| 73 | TESTIN[0] | TESTIN[0] | IN | – | | 0 |
| 74 | NTBE | – | OUTENO | – | 1 | |
| 75 | ALE | ALE | IN | – | | |
| 76 | A31 | A[31] | OUT | NABE | | |
| 77 | A30 | A[30] | OUT | NABE | | |
| 78 | A29 | A[29] | OUT | NABE | | |
| 79 | A28 | A[28] | OUT | NABE | | |
| 80 | A27 | A[27] | OUT | NABE | | |
| 81 | A26 | A[26] | OUT | NABE | | |
| 82 | A25 | A[25] | OUT | NABE | | |
| 83 | A24 | A[24] | OUT | NABE | | |
| 84 | A23 | A[23] | OUT | NABE | | |
| 85 | A22 | A[22] | OUT | NABE | | |
| 86 | A21 | A[21] | OUT | NCBE | | |
| 87 | A20 | A[20] | OUT | NCBE | | |
| 88 | A19 | A[19] | OUT | NCBE | | |
| 89 | A18 | A[18] | OUT | NABE | | |
| 90 | A17 | A[17] | OUT | NABE | | |
| 91 | A16 | A[16] | OUT | NABE | | |
| 92 | A15 | A[15] | OUT | NABE | | |
| 93 | A14 | A[14] | OUT | NABE | | |
| 94 | A13 | A[13] | OUT | NABE | | |
| 95 | A12 | A[12] | OUT | NABE | | |
| 96 | A11 | A[11] | OUT | NABE | | |
| 97 | A10 | A[10] | OUT | NABE | | |
| 98 | A09 | A[09] | OUT | NABE | | |
| 99 | A08 | A[08] | OUT | NABE | | |

## BOUNDARY SCAN TABLE (Cont.)

| No. | Cell Name | Pin | Type | Output Enable BS Cell | INTEST Guard Value | EXTEST/CLAMP Guard Value |
|---|---|---|---|---|---|---|
| 100 | A07 | A[07] | OUT | NABE | | |
| 101 | A06 | A[06] | OUT | NABE | | |
| 102 | A05 | A[05] | OUT | NABE | | |
| 103 | A04 | A[04] | OUT | NABE | | |
| 104 | A03 | A[03] | OUT | NABE | | |
| 105 | A02 | A[02] | OUT | NABE | | |
| 106 | A01 | A[01] | OUT | NABE | | |
| 107 | A00 | A[00] | OUT | NABE | | |
| 108 | NABE | ABE | INEN1 | – | 0 | |
| 109 | RLW | LOCK | OUT | NABE | | |
| 110 | NBW | NBW | OUT | NABE | | |
| 111 | NRW | NRW | OUT | NABE | | |
| 112 | TESTIN[15] | TESTIN[15 | ]IN | – | | 0 |
| 113 | TESTIN[14] | TESTIN[14] | IN | – | | 0 |
| 114 | TESTIN[13] | TESTIN[13] | IN | – | | 0 |
| 115 | TESTIN[12] | TESTIN[12] | IN | – | | 0 |
| 116 | TESTIN[11] | TESTIN[11 | IN | – | | 0 |
| 117 | TESTIN[10] | TESTIN[10] | IN | – | | 0 |
| 118 | TESTIN[09] | TESTIN[09] | IN | – | | 0 |
| 119 | TESTIN[08] | TESTIN[08] | IN | – | | 0 |
| 120 | DIN31 | D[31] | IN | – | | |
| 121 | DOUT31 | OUT | NENDOUT | | | |
| 122 | DIN30 | D[30] | IN | – | | |
| 123 | DOUT30 | D[30] | OUT | NENDOUT | | |
| 124 | DIN29 | D[29] | IN | – | | |
| 125 | DOUT29 | D[29] | OUT | NENDOUT | | |
| 126 | DIN28 | D[28] | IN | – | | |
| 127 | DOUT28 | D[28] | OUT | NENDOUT | | |
| 128 | DIN27 | D[27] | IN | – | | |
| 129 | DOUT27 | D[27] | OUT | NENDOUT | | |
| 130 | DIN26 | D[26] | IN | – | | |
| 131 | DOUT26 | D[26] | OUT | NENDOUT | | |
| 132 | DIN25 | D[25] | IN | – | | |
| 133 | DOUT25 | D[25] | OUT | NENDOUT | | |
| 134 | DIN24 | D[24] | IN | – | | |
| 135 | DOUT24 | D[24] | OUT | NENDOUT | | |

**Type Key**

| | |
|---|---|
| IN | Input Pad |
| OUT | Output Pad |
| INEN1 | Input Enable Active HIGH |
| OUTEN0 | Output Enable Active LOW |

# DC PARAMETERS

## ABSOLUTE MAXIMUM RATINGS

| Symbol | Parameter | Min | Max | Units | Notes |
|---|---|---|---|---|---|
| VDD | Supply Voltage | VSS–0.3 | VSS+7.0 | V | 1 |
| VIP | Voltage Applied to Any Pin | VSS–0.3 | VDD+0.3 | V | 1 |
| TS | Storage Temperature | –40 | +125 | °C | 1 |

## DC OPERATING CONDITIONS

| Symbol | Parameter | Min | Typ | Max | Units | Notes |
|---|---|---|---|---|---|---|
| VDD | Supply Voltage | 4.75 | 5.0 | 5.25 | V | |
| VIBC | IC Input HIGH Voltage | 3.5 | - | VDD | V | 2,3 |
| VILC | IC Input LOW Voltage | 0.0 | - | 1.5 | V | 2,3 |
| VIHT | IT/ITP Input HIGH Voltage | 2.4 | 3.0 | VDD | V | 2,4,5 |
| VILT | IT/ITP Input LOW Voltage | 0.0 | 0 | 0.8 | V | 2,4,5 |
| TA | Ambient Operating Temperature | 0 | 25 | +70 | °C | |

## DC CHARACTERISTICS FCLK = 25 MHz, Ta = 25° C

| Symbol | Parameter | Typ | Units | Notes |
|---|---|---|---|---|
| IDD | Supply Current | 1.0 | nA | |
| ISC | Output Short Circuit Current | 100 | nA | |
| ILU | D.C. Latch-up Current | 100 | nA | |
| LIN | IT Input Leakage Current | 0 | µA | |
| LINP | ITP Input Leakage Current | 0 | µA | |
| LOH | Output HIGH Current (Vout=VSS+0.4V) | 100 | nA | |
| IOL | Output LOW Current (Vout=VSS+0.4V) | 100 | nA | |
| VIHTK | IC Input HIGH Voltage Threshold | - | V | |
| VILTK | IC Input LOW Voltage Threshold | - | V | |
| VIHTT | IT/ITP Input HIGH Voltage Threshold | 3 | V | |
| VILTT | IT/ITP Input LOW Voltage Threshold | 0 | V | |
| CIN | Input Capacitance | 15 | pF | |

**Notes:**

1. These are stress ratings only. Exceeding the absolute maximum ratings may permanently damage the device. Operating the device at absolute maximum ratings for extended periods may affect device reliability, and void warantee.
2. Voltages measured with respect to VSS.
3. IC – CMOS-level inputs.
4. IT – TTL-level inputs (includes IT and ITOTZ pin types).
5. ITP – TTL-level inputs with pullups.
6. Nominal values shown are derived from transient analysis simulations.

## MAIN BUS SIGNALS[4] TA = +25° C, VDD = 5 V ± 5%

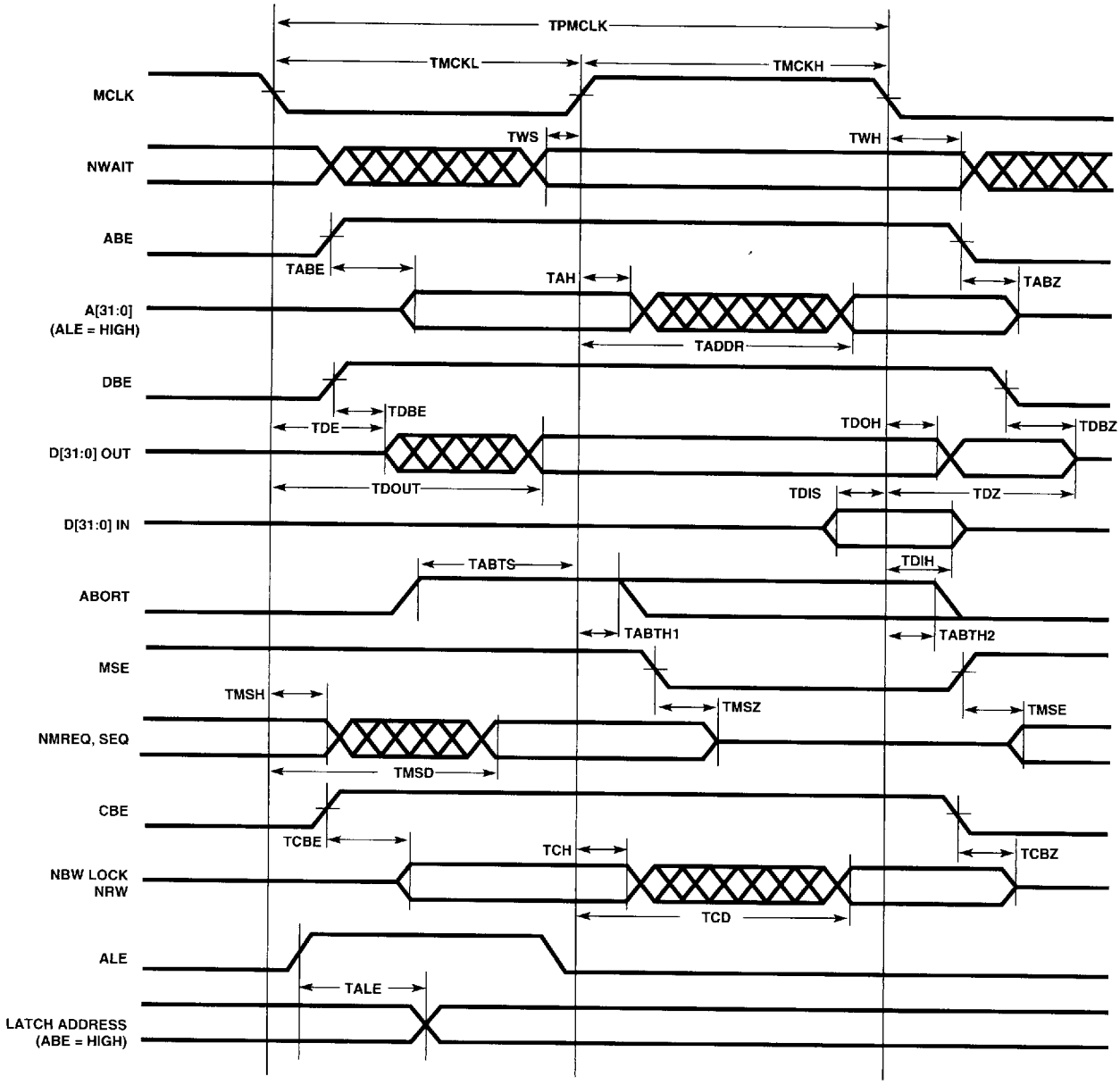
| Symbol | Parameter | Min | Typ | Max | Units | Notes |
|---|---|---|---|---|---|---|
| TMCKL | MCLK LOW Time | 25 | 30 | | ns | 1 |
| TMCKH | MCLK HIGH Time | 25 | 30 | | ns | |
| TWS | NWAIT Set-up to MCLK | 0 | 2 | | ns | |
| TWH | NWAIT Hold From MCLK | 5 | 2 | | ns | |
| TALE | Address Latch Enable | | 15 | 17 | | |
| TABE | Address Bus Enable | | 12 | 15 | ns | 2 |
| TABZ | Address Bus Disable | | 12 | 15 | ns | |
| TADDR | MCLK to Address Delay | | 20 | 25 | ns | 2 |
| TAH | Address Hold Time | 5 | 10 | | ns | 2 |
| TDBE | DBE to Data Enable | | 12 | 15 | ns | 2 |
| TDE | MCLK to Data Enable | | 16 | 17 | ns | 2 |
| TDBZ | DBE to Data Disable | | 16 | 17 | ns | |
| TDZ | MCLK to Data Disable | | 18 | 20 | ns | |
| TDOUT | Data Out Delay | | 30 | 32 | ns | 2 |
| TDOH | Data Out Hold | 5 | 8 | | ns | 2 |
| TDIS | Data in Set-up | 2 | 0 | | ns | |
| TDIH | Data in Hold | 10 | 5 | | ns | |
| TABTS | ABORT Set-up Time | 10 | 5 | | ns | |
| TABTH1 | ABORT Hold Time | 5 | 2 | | ns | 3 |
| TABTH2 | ABORT Hold Time | 5 | 2 | | ns | 3 |
| TMSE | NMREQ & SEQ Enable | | 8 | 10 | ns | |
| TMSZ | NMREQ & SEQ Disable | | 12 | 15 | ns | |
| TMSD | NMREQ & SEQ Delay | | 30 | 35 | ns | |
| TMSH | NMREQ & SEQ Hold | 5 | 7 | | ns | |
| TBSCL | TCK LOW Period | 50 | 20 | | ns | |
| TBSCH | TCK HIGH Period | 50 | 20 | | ns | |
| TBSIS | TDI, TMS Set-up to (TCr) | 10 | 0 | | ns | |
| TBSIH | TDI, TMS Hold from (TCr) | 10 | 0 | | ns | |
| TBSOD | TCF to TDO Valid | | 30 | 40 | ns | |
| TBSOH | TDO Hold Time | 5 | 15 | | ns | |
| TBSOE | TDO Enable Time | 5 | 15 | | ns | |
| TBSOZ | TDO Disable Time | | 15 | 40 | ns | |
| TBSSS | I/O Signal Set-up to (TCr) | 5 | 0 | | ns | |
| TBSSH | I/O Signal Set-up to (TCr) | 20 | 10 | | ns | |
| TBSDD | TCF to Data Output Valid | | 30 | 40 | ns | |

**Notes:**
1. MCLK timings measured between clock edges at 50% of Vdd.
2. The timings of these buses are measured to TTL levels.
3. TABTH1 is a requirement for VY86C610. To ensure compatibility with future processors, designs should meet TABTH2. TABTH2 is not tested on the VY86C610.
4. All AC test loads are 50 pF.

## MAIN BUS SIGNALS (Cont.)

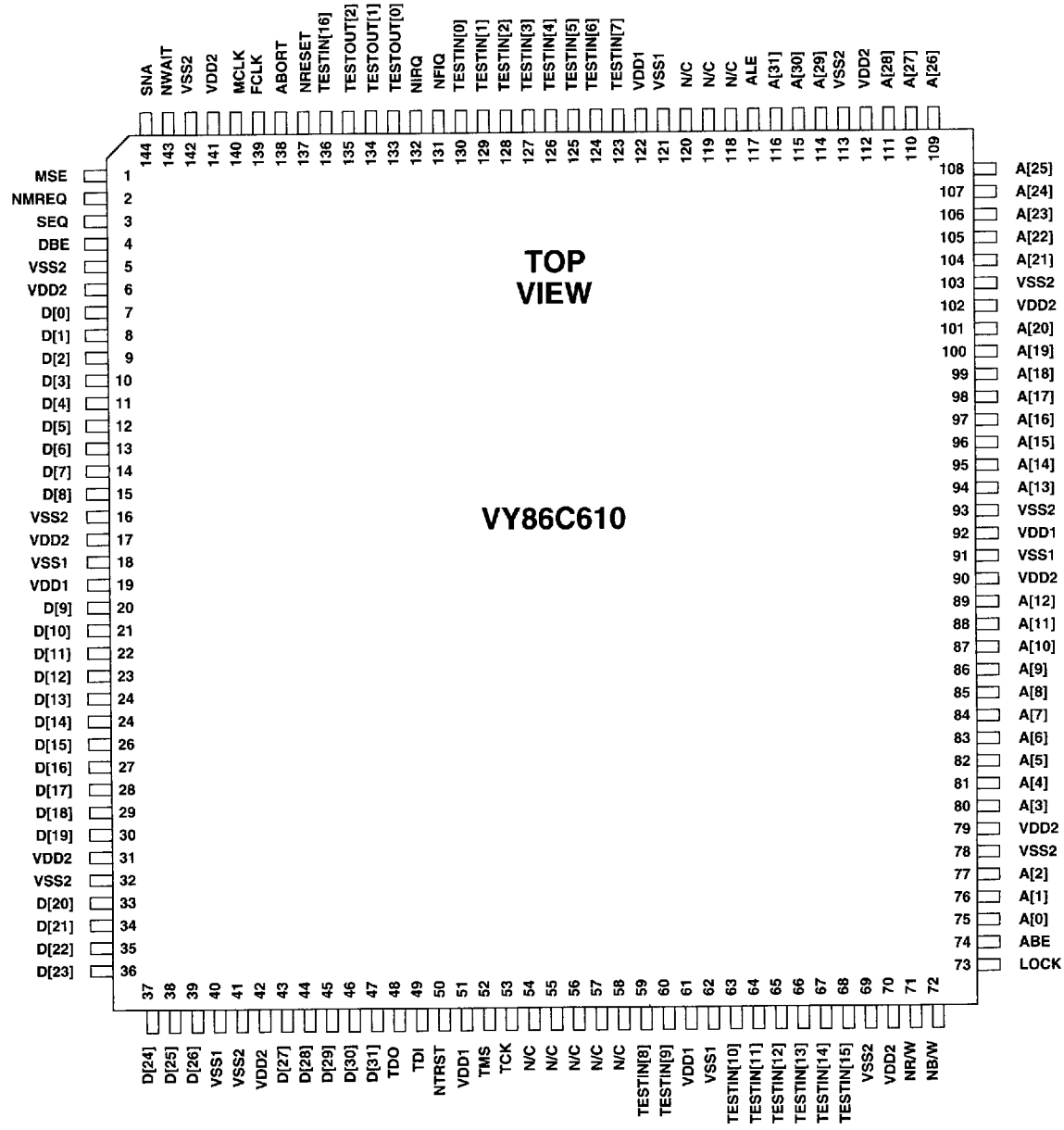
| Symbol | Parameter | Min | Typical | Max | Units | Notes |
|---|---|---|---|---|---|---|
| TBSDH | Data Output Hold Time | 5 | | | ns | |
| TBSDE | Data Output Enable Time | 5 | | | ns | |
| TBSDZ | Data Output Disable Time | | | 40 | ns | |
| TBSR | Reset Period | 30 | | | ns | |
| TBSRS | TMS Set-up to (TRr) | 10 | | | ns | |
| TBSRH | TMS Hold from (TRr) | 10 | | | ns | |

### MAIN BUS SIGNALS

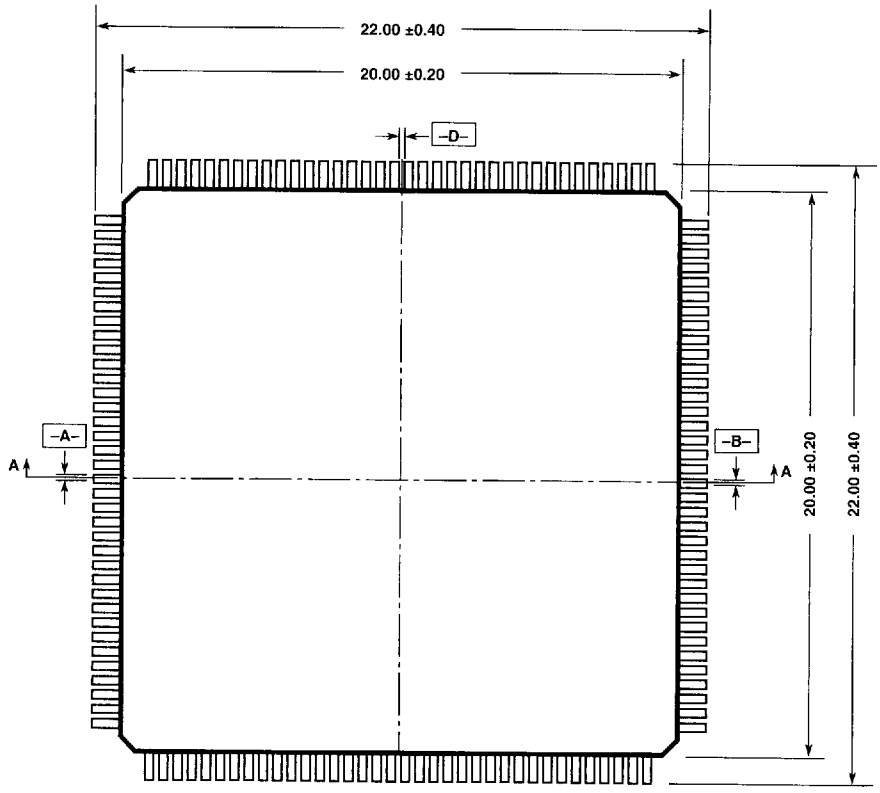


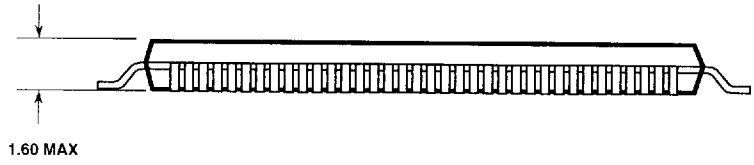
## PIN DIAGRAM

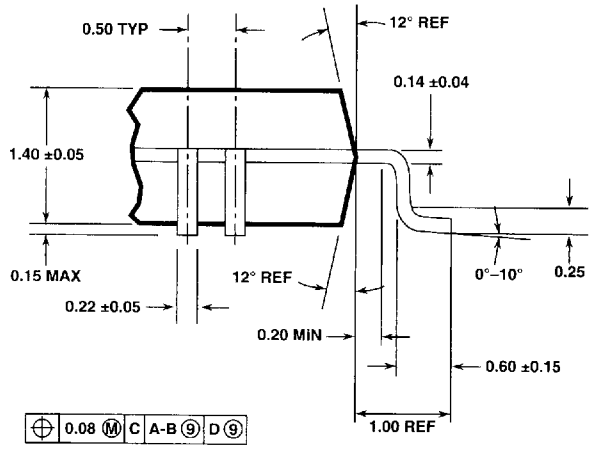
The VY86C610 is packaged in a 144-pin Thin Quad Flat Pack.



## PACKAGE MECHANICAL, NOTES 1-4




**Notes:**
1. Dimensions are in milimeters
2. Leadframe material: Copper
3. Mark of SG indicated same side gate at mold
4. To be molded cavity down (see section A-A in the top drawing)

## ORDER INFORMATION

| Part Number | Description | Package |
|---|---|---|
| VY86C61020BC | 20-MHz RISC MPU MMU, 4K Cache | 144-Pin Thin Quad Flat Pack |