VLSI TECHNOLOGY, INC.

VL86C010

32-BIT RISC MICROPROCESSOR

FEATURES

- 32-bit internal architecture
- 32-bit external data bus
- 64M-byte linear address space
- Bus timing optimized for standard DRAM usage with page mode operation
- 48M-byte/second bus bandwidth
- Simple/powerful instruction set providing an excellent high level language compiler target
- Hardware support for virtual memory systems
- Low interrupt latency for real-time application requirements
- Full CMOS implementation results in low power consumption
- Single 5 V ± 5% operation
- 84-pin plastic leaded chip carrier (PLCC)

DESCRIPTION

The VL86C010 Acorn RISC Machine (ARM) is a full 32-bit general-purpose microprocessor designed using reduced instruction set computer (RISC) methodologies. The processor is targeted for the microcomputer, graphics, industrial and controller markets for use in stand-alone or embedded systems. Applications in which the processor is useful include laser printers, formatters, graphics engines, Numerical Control machines, or any other systems requiring fast real-time response to external interrupt sources and high processing throughput.

The VL86C010 features a 32-bit data bus, 27 registers of 32 bits each, a load-store architecture, a partially overlapping register set, 22.5 clocks worst-case interrupt latency, conditional instruction execution, a 26-bit linear address space and an average instruction execution rate of from five-to-six million instructions per second (MIPS). Additionally, the processor supports two addressing modes: program counter (PC) and base register relative modes.

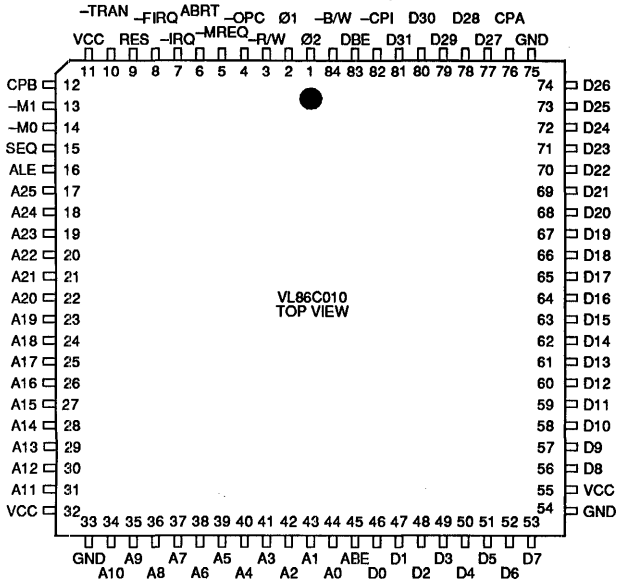
The ability to do pre- and post-indexing allows stacks and queues to be easily implemented in software. All instructions are 32 bits long (aligned on word boundaries), with register-to-register operations executing in one cycle. The two data types supported are 8-bit bytes and 32-bit words.

Using a load-store architecture simplifies the execution unit of the processor, because only a few instructions deal directly with memory and the rest operate register-to-register. Load and store multiple register instructions provide enhanced performance, making context switches faster and exploiting sequential memory access modes.

The processor supports two types of interrupts that differ in priority and register usage. The lowest latency is provided by the fast interrupt request (FIRQ) which is used primarily for I/O to peripheral devices. The other interrupt type (IRQ) is used for interrupt routines that do not demand low-latency service or where the overhead of a full context switch is small compared with the interrupt process execution time.

PIN DIAGRAM

PLASTIC LEADED CHIP CARRIER (PLCC)

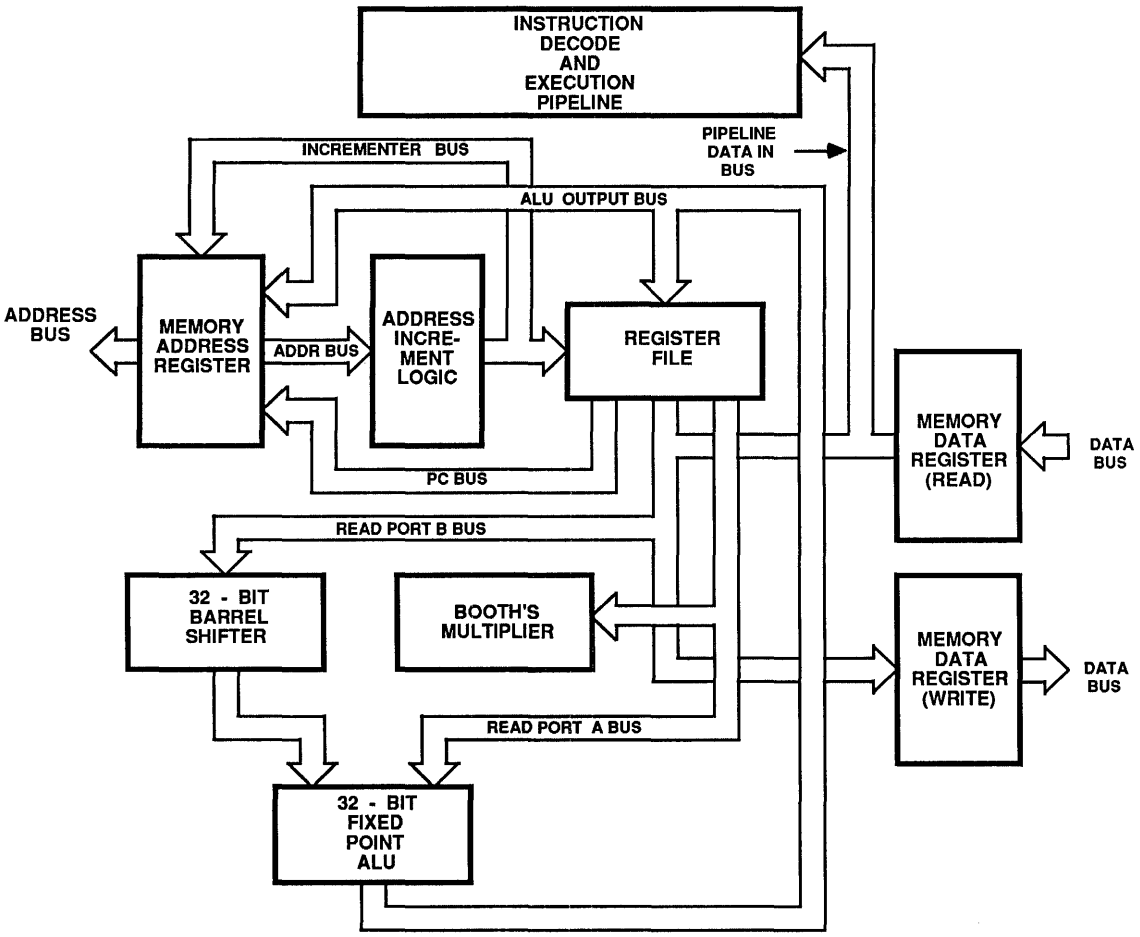


ORDER INFORMATION

Part Number	Clock Frequency	Package
VL86C010-10QC	10 MHz	Plastic Leaded Chip Carrier (PLCC)
VL86C010-12QC	12 MHz	Plastic Leaded Chip Carrier (PLCC)

Note: Operating temperature range is 0°C to +70°C.

BLOCK DIAGRAM



SIGNAL DESCRIPTIONS

Signal Name	Pin Number	Signal Description
Ø1, Ø2	2,1	Processor Clock Ø1 and Ø2 Inputs - These two inputs provide the clock to the processor. In order to minimize clock skew, these inputs are not buffered internally and therefore must swing monotonically between GND and VCC without overshoot. The clocks must be non-overlapping and should be driven directly by 74HCXX outputs. A typical circuit is shown on the following page. The VL86C110 (MEMC) will normally drive these inputs directly.
–IRQ	7	Interrupt Request Input - This is the normal interrupt request pin. It may be asserted asynchronously to cause the processor to be interrupted. It is active low.
–FIRQ	8	Fast Interrupt Request Input - This interrupt request line has a higher priority than IRQ, but otherwise is the same. It too is active low.
RES	9	Reset Input - This is the reset signal for the processor. While active, the processor executes no-ops (with –MREQ and SEQ both held active) until the RES signal goes inactive, from which point execution starts at the reset exception vector location. This signal is active high.
ABRT	6	Abort Input - This signal can be used to abort the current bus cycle being executed by the processor. Typically, it is connected to a memory management unit, such as the VL86C110, to control accesses for protection purposes. The abort signal is active high and requires a two clock minimum pulse to insure the reset operation will occur.
D31 - D0	81-77, 74-56 46-53	Data31- Data0 - This is the 32-bit bidirectional data bus used to transfer data to and from the memory. These lines are three-state and active high.
DBE	83	Data Bus Enable Input - This is the asynchronous three-state control signal for controlling the drivers of the data bus. When asserted the data bus is enabled and when low the data bus drivers are forced into the high-impedance state. During read operations the bus drivers are in the high-impedance state as well. This signal is active high. Systems that do not require the data bus for DMA or similar activities may tie this signal high.
–B/W	84	Not Byte/Word Output - This pipeline (note 1) signal indicates to the memory system that the current memory cycle is a byte rather than a word operation. It is asserted during the last portion of the cycle preceding the byte operation. When asserted (low) the memory system should deal with bytes by decoding the A1, A0 address lines. It is active low.
–M1, –M0	13, 14	Mode 1,0 Outputs - These two signals are used to indicate the current operating mode of the processor. They can be used as address space modifiers to increase the address space, or to assist a memory management unit in offering various protection modes. The lines are active low and the inverse of bits 1, 0 of the processor status register. –M1 –M0 MODE 0 0 Supervisor 0 1 IRQ 1 0 FIRQ 1 1 USER
A25 - A0	17 - 31, 34 - 44	Address 25 - Address 0 Outputs - These are the 26 address lines. A1 and A0 are byte addresses. During jumps and opcode fetches, the current mode value appears on these signals. The address lines are three-state and active high. A0, A1 are valid bits for all indexed transfers but are mode bits.
ABE	45	Address Bus Enable Input - This is the asynchronous three-state control signal for controlling the drivers of the address bus. When asserted, the address bus is enabled. The signal is active high.
ALE	16	Address Latch Enable Input - This signal is used to control internal transparent latches on the address outputs. When ALE is high the address outputs change during Ø2 to the value required for the next cycle. Direct interfacing to ROMs requires address lines to be stable until the end of Ø2. Holding ALE low until the end of Ø2 will latch the address outputs for ROM cycles. Systems that do not directly interface to ROMs may tie ALE high.

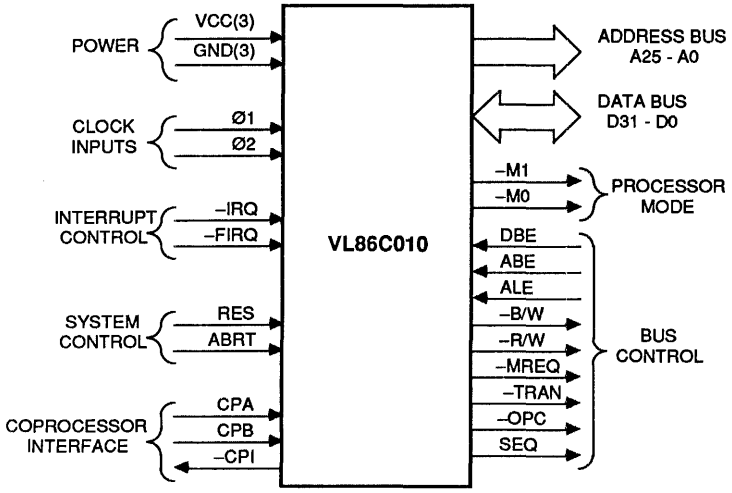
Note:

1. Pipeline signals are asserted during the last portion of the cycle preceding the cycle for which they will be used.

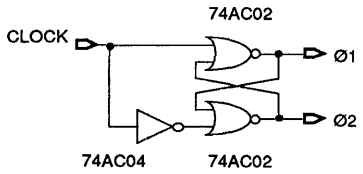
SIGNAL DESCRIPTIONS (Cont.)

Signal Name	Pin Number	Signal Description
–R/W	3	Not Read/Write Output - This is the read/write signal from the processor. When asserted (low), it indicates that the processor is performing a read operation. When negated (high), the processor is performing a write operation. This signal is a pipeline (note 1) signal and is active low.
–MREQ	5	Next Memory Cycle Start Output - This is an pipeline (note 1) indicator that is asserted before the processor will start a memory cycle during the next clock phase. This signal is active low. During the reset condition this signal is held active as the processor executes no-ops.
–TRAN	10	Translate Enable Output - This signal, when asserted by the processor tells a memory management unit that translation should be done on the current address. When negated, it indicates that the address should pass through untranslated. This signal is active low.
–OPC	4	Instruction Fetch Output - This pipeline (note 1) signal when asserted indicates that the current bus cycle is an instruction fetch. This signal is active low.
SEQ	15	Next Address Sequential Output - This pipeline (note 1) signal is asserted when the processor will generate a sequential address during the next memory cycle. It may be used to control fast memory access modes. This signal is active high. During the reset condition this signal is held active as the processor executes no-ops.
–CPI	82	Coprocessor Instruction (CMOS level output) - When the VL86C010 executes a coprocessor instruction, this output is driven low and the processor will wait for a response from an attached coprocessor device. The action taken is dependent upon the coprocessor response signalled on the CPA and CPB inputs.
CPB	12	Coprocessor Busy (TTL level input) - An attached coprocessor that is capable of performing the operation which the VL86C010 is requesting (by asserting the –CPI), but cannot begin immediately, should indicate the busy condition by driving this signal high. When the coprocessor is ready to start it should bring the CPB signal low. The VL86C010 samples this signal on the falling edge of the Ø1 clock while the –CPI is active (low).
CPA	76	Coprocessor Absent (TTL level input) - A coprocessor capable of executing the operation currently requested by the VL86C010 (–CPI active) should bring the CPA low immediately. If the CPA is high on the falling edge of the Ø1 clock, the processor will abort the coprocessor handshake and take the undefined instruction trap. If the CPA is low and remains low during the –CPI active time, then the VL86C010 will busy-wait until the CPB signal becomes low and complete the coprocessor instruction.

FUNCTIONAL PIN DIAGRAM



TYPICAL CLOCK GENERATOR



FUNCTIONAL DESCRIPTION

The philosophy of RISC processor design is based on the idea that some processing functions can be moved from hardware to software with the result that the simplified hardware can actually execute functions in software faster than with complicated hardware. Analysis done several years ago at major research centers has shown that a processor and compiler combination can replace the traditional processor-alone architectures. A historical fact of the 16-bit processor world is that after chip designers spent many man-months figuring out how to implement universally acceptable complicated instructions to do things, few compiler writers actually took advantage of these complex instructions. Most compilers only use a fraction of the instructions and addressing modes of traditional computer architectures.

The customers pay for the unused silicon required to implement these instructions. They pay for the inefficient utilization in both cost of the processor and in lower performance. The silicon spent for complex instruction decoding and micro-sequencing could have been used for additional pipelining, larger register sets, or other special-purpose hardware that can be used efficiently. If the addition of a new instruction causes all instructions to execute 10% slower due to internal processor delays, then the new instruction had better be used more than 10% of the time, otherwise, overall performance has been sacrificed. This makes an argument for simple performance oriented architectures that are more dependent on compiler technology to implement less frequently used instructions.

COMPARISON OF PROCESSORS

Inherent in the concept of RISC processors is the notion that more instructions are required to implement the same functions that could be done by fewer instructions with a complex instruction set computer (CISC) processor. In most cases even when more instructions are needed by RISC processors, the function can still be performed quicker on RISC processors than CISC processors. This is causing the industry to doubt the Million Instruction Per Second (MIPS) ratings of RISC processors, for good reason. MIPS are often used exclusively as a means of benchmarking performance. A better measure of performance is to time actual execution of real-world problems, independent of the number of instructions required to implement the function.

Benchmarks such as Dhrystone 1.1 attempt to approximate real conditions. Measurement is based in Dhrystone loops per second. The VL86C010 delivers about 740 loops per second using DRAM, and about 1000 per second using SRAM, per clock megahertz.

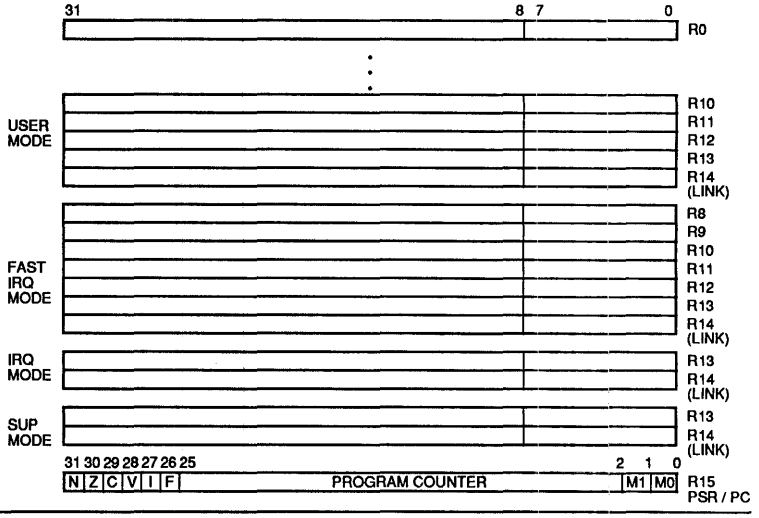
An important parameter to keep constant when benchmarking processors is the memory access times, since not all processors will meet performance claims when working with commodity memories.

Another traditional measure of performance in the microprocessor world is the clock frequency of the processor. Faster is better has been the rule of thumb, but what is actually the most important consideration is the average number of bus cycles per instruction. A processor with a low clock frequency and a low number of bus cycles per instruction can actually outperform a processor with a high clock frequency and a higher number of bus clock cycles per instruction. The best choice of processors is one that benchmarks high while using a relatively low clock frequency and a small number of clocks per instruction executed. The VL86C010 possesses these characteristics, giving it the best future evolution path to exploit advances in process technology.

PROGRAMMING MODEL

The VL86C010 contains a large, partially overlapping set of 27 32-bit registers, although the programmer can access only 16 registers in any mode of operation. Fifteen of the registers are general purpose; with the remaining 12 dedicated to functions such as User Mode, FIRQ Mode, IRQ Mode, Supervisor Mode and the Program Counter (PC)/Processor Status Register (PSR). Figure 1 shows the register model of the VL86C010. Registers R0-to-R13 are accessible from the user mode for any purpose. The fifteenth register, user-mode return-link register, is specific to the user mode. Its contents are mapped with those of other return-link registers as the mode is changed. The return-link register is used by the Branch-and-Link instruction in a procedure call sequence but may be used as a general-purpose register at other times. The least significant two bits of the processor status word (PSW) define the current mode of operation.

FIGURE 1. VL86C010 REGISTER MODEL



Seven registers are dedicated to the FIRQ mode and overlie user-mode registers R8-to-R14 when the fast interrupt request is serviced. The registers R8 FIRQ-to-R13 FIRQ are local to the fast interrupt service routine and are used instead of the user-mode registers R8-R13. Register R14 FIRQ holds the address used to restart the interrupted program instead of pushing it onto a stack at the expense of another memory cycle. Using a link-register helps provide very fast servicing of I/O related interrupts without disturbing the contents of the general-purpose register set although the FIRQ routine can access the R0-to-R7 user-mode registers if desired. The FIRQ mode is used typically for very short interrupt service routines that might fetch and store characters in a disk-or-tape-controller application.

The next two registers are dedicated to the IRQ mode and overlie user mode registers R13 and R14 when the IRQ is serviced. Once again, R14 IRQ is the return link register that holds the restart address and R13 IRQ is general-purpose and dedicated to the IRQ mode. This mode is used when the interrupt service routine will be lengthy and the overhead of saving and reloading the register set will not be a significant portion of the overall execution time.

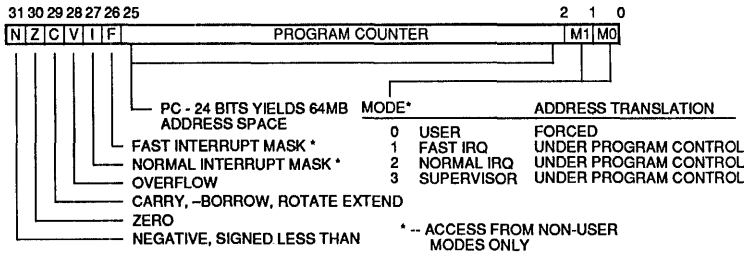
Two registers are dedicated to the supervisor mode and overlie user mode registers R13 and R14 when a supervisor mode switch is made using a software interrupt (SWI) instruction. Operation of these two registers is the same as previously discussed.

The last register (R15) contains the processor status word and program counter and is shared by all modes of operation. The upper six bits are processor status, the next 24 bits are the program counter (word address), and the last two indicate the mode.

PROCESSOR STATUS REGISTER

Like most 32-bit processors, the VL86C010 makes a distinction between user and supervisor modes: the user executes at the lowest privilege level, and the supervisor and interrupts execute at higher levels of privilege. Figure 2 shows the processor status word containing the control line states associated with each mode.

FIGURE 2. PROCESSOR STATUS REGISTER



Translate is a control signal provided by the processor for control of an external memory management unit. The translate line is enabled in the user mode and disabled in the supervisor, fast interrupt and normal interrupt modes, since all modes except for the user mode are expected to be running secure code. Translated fetches can be made from the non-user modes by setting an optional bit in the load/store instructions.

The processor status register (PSR) contains the program counter, mode control bits, and condition codes as shown in Figure 2. The bits marked with an asterisk are alterable only from non-user modes. If the user tries to write to these bits, they remain unchanged and the processor continues operation in the user mode. In other words, this is not a trap condition. The flags in the processor status register are the standard Negative, Zero, Carry, and Overflow. The 16 allowable combinations of the condition code bits are shown in Table 1. These combinations are used in all instruction executions since a conditional branch is nothing more than a jump instruction with conditional execution.

EXCEPTIONS

The VL86C010 supports a partially overlapping register set so that when interrupts are taken, the contents of the register array do not have to be saved before new operations can begin. Improved response time is accomplished, in the case of the fast interrupt, by dedicating six general-purpose registers, in addition to a return-link register, that are only accessible in the FIRQ mode. These dedicated registers can contain all the pointers and byte-counts for simple I/O service routines thus incurring no overhead when context switching between processing and servicing interrupts at high rates. The other modes (IRQ and SUP) each have one general-purpose and one return address (link) register dedicated to them. The general-purpose register is ideally suited for implementing a local stack for each mode. The need for dedicated registers in these modes is not as great since the time spent in an interrupt or supervisor routine is on the average much greater than the time spent in transition between the routines. The working registers can be saved and restored from stacks without significant overhead.

The interrupt latency of the VL86C010 is very short because the instruction execution time is typically two clocks, with a maximum of 18 clocks (for a load-multiple instruction, loading 16 registers). Once the processor recognizes an interrupt is pending, the time to begin processing is 4.5 making a total worst-case interrupt latency of 22.5 clocks. Systems supporting virtual memory should add three clocks as the address exception and data abort exceptions are higher priority and must be entered first to prevent losing status. In addition to interrupts, six other types of exceptions are supported by the processor. These are address exceptions, data-fetch cycle aborts, instruction-fetch cycle aborts, software interrupts, undefined instruction traps and reset.

The VL86C010 supports a 26-bit linear address space allowing a total of 64 Mbytes of physical memory. Data references outside the range 0-to-3FF,FFFFH cause an address exception trap which can be used to detect a run-away program. The program counter will wrap around to 0000H without causing an address exception condition.

TABLE 1. INSTRUCTION CONDITION CODES

Condition	Encoded Value	Operation
AL	E	Always
CC	3	Carry Clear/Unsigned Lower Than
CS	2	Carry Set/Unsigned Higher Or Same
EQ	0	Equal (Z Set)
GE	A	Greater Than Or Equal (N • V) + (–N • –V)
GT	C	Greater (((N • V) + (– N • –V)) • –Z)
HI	8	Higher Unsigned (C • –Z)
LE	D	Less Than Or Equal (((N • –V) + (–N • V)) + Z)
LS	9	Lower Or Same Unsigned (–C + Z)
LT	B	Less Than (N • –V) + (–N • V))
MI	4	Negative (N)
NE	1	Not Equal (–Z)
NV	F	Never
PL	5	Positive (–N)
VC	7	Overflow Clear
VS	6	Overflow Set

If the abort signal is asserted by the memory management unit during a data fetch the processor will abort data transfer instructions (LDR, STR) as if they had never been executed. If the instruction was a block data transfer (LDM, STM) the processor will allow the instruction to complete. If the write back control bit in these instructions is set, the base address will be updated even if it would have been overwritten during the instruction execution. An example of this would be execution of a block data transfer instruction with the base register in the list of registers to be overwritten.

Software interrupt instructions are used to change from user mode to supervisor mode. When an SWI is encountered the processor will save the current program counter (R15) into R14 SUP, set the mode bits to the supervisor mode, and start execution at the software interrupt vector address. An undefined instruction will cause a trap similar to the execution of a software interrupt except that the Undefined Instruction Vector will be used as the next address. Reset is treated similarly to the other traps and will start the processor from a known address. When the reset condition is recognized the currently executing instruction will terminate abnormally, the processor will enter the supervisor mode, disable both the FIRQ and IRQ interrupts, and begin execution at address 0000H. While the reset condition remains, the processor will execute dummy instruction fetches with –MREQ and SEQ held active.

The processor exception vector map is illustrated in Table 2. The exceptions are prioritized reset (highest), address exception, data abort, FIRQ, IRQ, prefetch abort, undefined instruction, and software interrupt (lowest). These vector addresses normally will contain a branch instruction to the associated service routine except for the FIRQ entry. In order to further reduce latency, the FIRQ service routine may begin at address 001CH if the software designer so chooses.

Whenever the processor enters the supervisor mode, whether from an SWI, address exception, undefined instruction trap, prefetch or data abort, the IRQ is disabled and the FIRQ unchanged.

INSTRUCTION SET

The VL86C010 supports five basic types of instructions, with several options available to the programmer. These instruction types are: data processing, data transfer, block data transfer, branch, and software interrupt. All instructions contain a 4-bit conditional execution field (shown in Table 1) that can cause an instruction to be skipped if the condition specified is not true. The execution time for a skipped instruction is one sequential cycle (100 ns for a 10 MHz processor).

Data processing instructions operate only on the internal register file, and each has three operand references: a destination and two source fields. The destination (Rd) can be any of the registers including the processor status

TABLE 2. EXCEPTION VECTOR MAP

Address (Hex)	Function	Priority Level
000 0000	Reset	0
000 0004	Undefined Instruction Trap	6
000 0008	Software Interrupt	7
000 000C	Abort (Prefetch)	5
000 0010	Abort (Data)	2
000 0014	Address Exception	1
000 0018	Normal Interrupt (IRQ)	4
000 001C	Fast Interrupt (FIRQ)	3

TABLE 3. DATA PROCESSING INSTRUCTIONS

Instruction	Function	Operation	Flags Affected
ADC	Add With Carry	Rd:=Rn+Shift(S2)+C	N, Z, C, V
ADD	Add	Rd:=Rn+Shift(S2)	N, Z, C, V
AND	And	Rd:= Rn • Shift(S2)	N, Z, C
BIC	Bit Clear	Rd:= Rn • –Shift(S2)	N, Z, C
CMN	Compare Negative	Shift(S2)+Rn	N, Z, C, V
CMP	Compare	Rn-Shift(S2)	N, Z, C, V
EOR	Exclusive OR	Rd:=Rn⊕ Shift(S2)	N, Z, C
MLA	Multiply with Accumulate	Rd:=Rm * Rs + Rd	N, Z, C, V
MOV	Move	Rd:=Shift(S2)	N, Z, C
MUL	Multiply	Rd:=Rm * Rs	N, Z, C, V
MVN	Move Negative	Rd:= –Shift(S2)	N, Z, C
ORR	Inclusive OR	Rd:=Rn+Shift(S2)	N, Z, C
RSB	Reverse Subtract	Rd:=Shift(S2)-Rn	N, Z, C, V
RSC	Reverse Subtract With Carry	Rd:=Shift(S2)-Rn-1+C	N, Z, C, V
SBC	Subtract With Carry	Rd:=Rn-Shift(S2)-1+C	N, Z, C, V
SUB	Subtract	Rd:=Rn-Shift(S2)	N, Z, C, V
TEQ	Test For Equality	Rn⊕ Shift(S2)	N, Z, C
TST	Test Masked	Rn • Shift(S2)	N, Z, C

TABLE 4. MEMORY ADDRESSING MODES

Addressing Mode	Operation	Syntax
PC Relative	EA* = PC +/– Offset (12 Bits)	LABEL
Base Register Offset With Post-Increment	EA* = Rn Rn +/– Offset —▶ Rn	[Rn],Off
Base Register Offset With Pre-Increment**	EA* = Rn +/– Offset (12 Bits) Rn +/– Offset —▶ Rn	[Rn,Off]
Base Register Index With Post-Increment	EA* = Rn Rn +/– Rm —▶ Rn	[Rn],Rm
Base Register Index With Pre-Increment**	EA* = Rn +/– Rm Rn +/– Rm —▶ Rn	[Rn,Rm]

* Effective Address

** Program control of index register update; i.e., Rn may be left unchanged.

register, although some bits in R15 can only be changed in particular modes. The source operands can have two forms: both can be registers (Rm and Rn) or a register (Rn) and an 8-bit immediate value. Both forms of operand specification provide for the optional shifting of one of the source values using the on-board barrel shifter. If both operands are registers, the Rm can be shifted. For the other case, it is the immediate value that can pass through the shifter. Another field in these instructions allows for the optional updating of the condition codes as a result of execution of the operation. Table 3 shows the possible data processing operations and the status flags affected.

Data transfer instructions are used to move data between memory and the register file (load), or vice-versa (store). The effective address is calculated using the contents of the source register (Rn) plus an offset of either a 12-bit immediate value or the contents of another register (Rm). When the offset is a register it can optionally be shifted before the address calculation is made. Table 4 shows the addressing modes supported and their corresponding assembler syntax. The offset may be added to, or subtracted from the index register Rn. Indexing can be either pre- or post- depending on the desired addressing mode. In the post-indexed mode the transfer is performed using the contents of the index register as the effective address and the index register is modified by the offset and re-written. In the pre-indexed mode the effective address is the index register modified in the appropriate manner by the offset. The modified index register can be written back to Rn if the write back bit is set or left unchanged if desired. When a register is used as the offset, it can be pre-scaled by the barrel shifter in a similar manner as with data processing instructions.

Data transfer instructions can manipulate bytes or words in memory. When a byte is read from the memory, it is placed in the low-order eight bits of the register and zero-extended to a full word. For byte writes the lower eight bits of the register are written to the byte address referenced and the other bytes within the word are unaffected.

Words are written into the address space as least-significant byte first. That is, the byte at the lowest address will be found right justified in a register.

The VL86C010 supports both logical and physical address spaces at a lower level in hardware than other processors. Data transfer instructions contain a translate enable bit that allows non-user mode programs to select the logical or physical address space as desired. The bit from the instruction is placed on the –TRAN pin of the processor to signal an external memory management unit (MMU) whether to translate or pass the address from the processor bus to the memory. This allows programs executing in the supervisor or interrupt modes to have easy access to user memory areas for page fault correction or to have bounds checking performed on dynamic data structures in the system space by the MMU. In the user mode, addresses are always translated by the MMU if it is implemented in the system.

The block data transfer instructions allow multiple registers to be moved in a single instruction. The instruction has a field containing a bit for each of the sixteen registers visible in the current mode. Bit 0 corresponds to R0, and bit 15 corresponds to R15, the program counter. A bit set in a particular position means that the corresponding register will be affected by the transfer. The registers are always saved from lowest to highest, and R0 will always appear at a lower address than R1. The ability to pre- or post-increment or decrement allows both stacks and queues to be implemented efficiently with any convention chosen by the programmer.

The branch instruction has two forms, branch and branch-with-link. The branch instruction causes execution to start at the current program counter plus a 24-bit offset contained in the instruction. The offset is left-shifted by two bits (forming a 26-bit address) before it is added to the program counter. Since all instructions are word-aligned, a branch can reach any location in the address space. The branch-with-link instruction copies the program counter and processor status register into R14 prior to branching to the new address. Returning from the branch-with-link simply involves reloading the program counter from R14 (MOV PC,R14). The PSR can optionally be restored from R14 (MOVS PC,R14).

The software interrupt instruction format is used primarily for supervisor service calls. When this instruction is executed, the PC and PSR are saved in R14 SUP. The PC is then set to the SWI vector location and the processor placed in the supervisor mode.

Instructions operate at speeds dependent upon the options selected. Table 5 shows the instruction types, execution rates and adjustments for operand shifting or affecting the program counter. The table is expressed in terms of N and S cycles representing Non-sequential and Sequential cycles respectively. The processor is able to take advantage of memories that have faster access times when accessed sequentially in the nibble or column mode. These faster cycles are designated as S-cycles, while the N-cycles typically take twice as long. If faster static memory is used, the N and S cycles would be equal.

The VL86C010 is offered in an 84-pin Plastic Leaded Chip Carrier (PLCC) package for lower cost applications. The PLCC package can be either surface mounted directly onto the board or socketed with currently available standard sockets depending on manufacturing requirements and/or capabilities.

TABLE 5. INSTRUCTION EXECUTION TIMES

Operation	Base Execution Time	Adjustment for Source Shift	Adjustment for PC Modification
RS • # → RD	1S	1S for Shift(RS)	1S + 1N if PC Modified
RS • RS → RD	1S	1S for Shift(RS)	1S + 1N if PC Modified
LDR	2S + 1N		1S + 1N if PC Modified
STR	2N		
LDM	(n* + 1)S + 1N		1S + 1N if PC Modified
STM	(n* - 1)S + 2N		
BR	2S + 1N		
BR & LINK	2S + 1N		
SWI	2S + 1N		
MUL, MLA	16S**		

* - The number of registers transfered in a Load/Store Multiple instruction. If the condition field in an instruction is not true, the instruction is skipped and the execution time is 1S cycle.
** - This is the worst case time. The actual time is a function of the value in the Rs register.

S implies a sequential cycle.

N implies a non sequential cycle.

EXAMPLES OF THE INSTRUCTION SET

The following examples illustrate methods by which basic processor instructions can be combined to yield efficient code. None of the methods saves a large amount of execution time, although they all save some, mostly they result in more compact code.

EXAMPLE 1 - USING THE CONDITIONAL EXECUTION FOR THE LOGICAL-OR FUNCTION

```
        CMP     Rn, p           ; IF Rn = p OR Rm = q THEN
        BEQ     Label           ;    GOTO Label
        CMP     Rm, q
        BEQ     Label
```

By using conditional execution, the routine compresses to:

```
        CMP     Rn, p
        CMPNE   Rm, q           ; if Rn not equal p, try other test
        BEQ     Label
```

EXAMPLE 2 - ABSOLUTE VALUE

```
        TEQ     Rn, 0           ; check sign
        RSBMI   Rn, Rn, 0       ; and 2's complement if required
```

EXAMPLE 3 - UNSIGNED 32-BIT MULTIPLY

```
                                ; Enter with numbers in Ra, Rb - product contained in Rm
        MOV     Rm, 0           ; init result register
LOOP    MOVS    Ra, Ra LSR 1    ; stops when Ra becomes zero
        ADDCS   Rm, Rm, Rb      ; Rm = Ra * Rb
        ADD     Rb, Rb, Rb
        BNE     LOOP            ; ( Ra = 0, Rb is altered )
```

EXAMPLE 4 - MULTIPLICATION BY 4, 5, OR 6 AT RUN TIME

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

EXAMPLE 5 - MULTIPLICATION BY CONSTANT (2^N)+1 USING THE BARREL SHIFTER (3,5,9,17, ...)

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

EXAMPLE 6 - MULTIPLICATION BY CONSTANT (2^N) - 1 (3, 7, 15, ...)

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

EXAMPLE 7 - MULTIPLICATION BY 6

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

EXAMPLE 8 - MULTIPLY BY 10 AND ADD EXTRA NUMBER (DECIMAL TO BINARY CONVERSION)

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

EXAMPLE 9 - DIVISION AND REMAINDER

```
; enter with numbers in Ra and Rb
        MOV     Rcnt, 1         ; bit to control the division
DIV1    CMP     Rb, Ra          ; move Rb until greater than Ra
        MOVCC   Rb, Rb LSL 1    ; result in Rc
        MOVCC   Rcnt, Rcnt LSL 1    ; remainder in Ra
        BCC     DIV1
        MOV     Rc, 0
DIV2    CMP     Ra, Rb          ; test for possible subtraction
        SUBCS   Ra, Ra, Rb      ; subtract if valid
        ADDCS   Rc, Rc, Rcnt    ; put relevant bits in result
        MOVS    Rcnt, Rcnt LSR 1    ; shift control bit
        MOVNE   Rb, Rb LSR 1    ; halve unless finished
        BNE     DIV2
```

INSTRUCTION CYCLE OPERATIONS

In the following tables –MREQ and SEQ (which are pipelined up to one cycle ahead of the cycle to which they apply) are shown in the cycle in which they appear, so they predict the address of the next cycle. The address bus value, –B/W, –R/W, and –OPC (which appear up to half a cycle ahead) are shown in the cycle to which they apply.

BRANCH AND BRANCH WITH LINK

A branch instruction calculates the branch destination in the first cycle, while performing a prefetch from the current PC. This prefetch is done in all cases, since by the time the decision to take the branch has been reached it is already too late to prevent the prefetch.

During the second cycle a fetch is performed from the branch destination, and the return address is stored in register 14 if the link bit is set.

The third cycle performs a fetch from the destination + 4, refilling the instruction pipeline, and if the branch is with link, R14 is modified (4 is subtracted from it) to simplify return from SUB PC, R14, #4 to MOV PC,R14. This makes the STM . .(R14) LDM . . (PC) type of subroutine work correctly.

2

TABLE 6. BRANCH AND BRANCH WITH LINK

Cycle	Address	–B/W	–R/W	Data	SEQ	–MREQ	–OPC
1	PC+8	1	0	(PC+8)	0	0	0
2	ALU	1	0	(ALU)	1	0	0
3	ALU+4	1	0	(ALU+4)	1	0	0

(PC is the address of the branch instruction, ALU is an address calculated by the processor, (ALU) is the contents of that address, etc.)

DATA OPERATIONS

A data operation executes in a single datapath cycle except where the shift is determined by the contents of a register. A register is read onto the A Bus, and a second register or the immediate field onto the B Bus. The ALU combines the A Bus source and the shifted B Bus source according to the operation specified in the instruction, and the result (when required) is written to the destination register. (Compares and tests do not produce results, only the ALU status flags are changed.)

An instruction prefetch occurs at the same time as the above operation, and the program counter is incremented.

When the shift length is specified by a register, an additional datapath cycle occurs before the above operation to copy the bottom eight bits of that register into a holding latch in the barrel shifter. The instruction prefetch will occur during this first cycle, and the operation cycle will be internal (i.e., will not request memory). This internal cycle is configured to merge with the next cycle into a single memory N-cycle when the VL86C110 is used as the memory interface.

The PC may be any (or all) of the register operands. When read onto the A Bus it appears without the PSR bits, on the B Bus it appears with them. Neither will affect external bus activity. When it is the destination, however, external bus activity may be affected. If the result is written to the PC, the contents of the instruction pipeline are invalidated, and the address for the next instruction prefetch is taken from the ALU rather than the address incrementer. The instruction pipeline is refilled before any further execution takes place, and during this time exceptions are locked out.

TABLE 7. DATA OPERATIONS

Type	Cycle	Address	–B/W	–R/W	Data	SEQ	–MREQ	–OPC
Normal	1	PC+8	1	0	(PC+8)	1	0	0
		PC+12						
Dest=PC	1	PC+8	1	0	(PC+8)	0	0	0
	2	ALU	1	0	(ALU)	1	0	0
	3	ALU+4	1	0	(ALU+4)	1	0	0
		ALU+8						
Shift (RS)	1	PC+8	1	0	(PC+8)	0	1	0
	2	PC+12	1	0	–	1	0	1
		PC+12						
Shift (RS), Dest=PC	1	PC+8	1	0	(PC+8)	0	1	0
	2	PC+12	1	0	–	0	0	1
	3	ALU	1	0	(ALU)	1	0	0
	4	ALU+4	1	0	(ALU+4)	1	0	0
		ALU+8						

INSTRUCTION CYCLE OPERATIONS (Cont.)

MULTIPLY AND MULTIPLY ACCUMULATE

The multiply instructions make use of special hardware which implements a two-bit Booth's algorithm with early termination. During the first cycle the accumulate register is brought to the ALU, which either transmits it or produces zero (according to whether the instruction is MLA or MUL) to initialize the destination register. During the same cycle one of the operands is loaded into the Booth's shifter via the A Bus.

The datapath then cycles, adding the second operand to, subtracting it from, or just transmitting, the result register. The second operand is shifted in the Nth cycle by 2N or 2N + 1 bit, under control of the Booth's logic. The first operand is shifted right two bits per cycle, and when it is zero the instruction terminates (possibly after an additional cycle to clear a pending borrow). All cycles except the first are internal.

If the destination is the PC, all writing to it is prevented. The instruction will proceed as normal except that the PC will be unaffected. (If the S bit is set the PSR flags will be meaningless).

TABLE 8. MULTIPLY AND MULTIPLY ACCUMULATE

Type	Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC
(Rs) = 0,1	1	PC+8	1	0	(PC+8)	0	1	0
	2	PC+12	1	0	–	1	0	1
		PC+12						
(Rs) > 1	1	PC+8	1	0	(PC+8)	0	1	0
	2	PC+12	1	0	–	0	1	1
	•	PC+12	1	0	–	0	1	1
	M	PC+12	1	0	–	0	1	1
	M+1	PC+12	1	0	–	1	0	1
		PC+12						

(M is the number cycles required by the Booth's algorithm; see the section on instruction speeds.)

LOAD REGISTER

The first cycle of a load register instruction performs the address calculation. The data is fetched from memory during the second cycle, and the base register modification is performed during this cycle (if required). During the third cycle, the data is transferred to the destination register and external memory is unused. This third cycle may normally be merged with the following prefetch to form one memory N-cycle.

Either the base or the destination (or both) may be the PC, and the prefetch sequence will be changed if the PC is affected by the instruction.

The data fetch may abort, and in this case, the base and destination modifications are prevented.

TABLE 9. LOAD REGISTER

Type	Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC	-TRAN
Normal	1	PC+8	1	0	(PC+8)	0	0	0	
	2	ALU	B/W	0	(ALU)	0	1	1	t
	3	PC+12	1	0	–	1	0	1	
		PC+12							
Dest = PC	1	PC+8	1	0	(PC+8)	0	0	0	
	2	ALU	B/W	0	(ALU)	0	1	1	t
	3	PC+12	1	0	–	0	0	1	
	4	(ALU)	1	0	((ALU))	1	0	0	
	5	(ALU)+4	1	0	((ALU)+4)	1	0	0	
		(ALU)+8							
Base = PC, Write back, Dest #PC	1	PC+8	1	0	(PC+8)	0	0	0	
	2	ALU	B/W	0	(ALU)	0	1	1	t
	3	PC'	1	0	–	0	0	1	
	4	PC'	1	0	(PC')	1	0	0	
	5	PC'+4	1	0	(PC'+4)	1	0	0	
		PC'+8							
Base=PC, Write back Dest=PC	1	PC+8	1	0	(PC+8)	0	0	0	
	2	ALU	B/W	0	(ALU)	0	1	1	t
	3	PC'	1	0	–	0	0	1	
	4	(ALU)	1	0	((ALU))	1	0	0	
	5	(ALU)+4	1	0	((ALU)+4)	1	0	0	
		(ALU)+8							

(PC' is the PC value modified by write back; t shows the cycle where the force translation option in the instruction may be used.)

INSTRUCTION CYCLE OPERATIONS (Cont.)

STORE REGISTER

The first cycle of a store register is similar to the first cycle of load register. During the second cycle, the base modification is performed and at the same time, the data is written to memory. There is no third cycle.

The PC will only be modified if it is the base and write back occurs. A data abort prevents the base write back.

TABLE 10. STORE REGISTER

Type	Cycle	Address	–B/W	–R/W	Data	SEQ	–MREQ	–OPC	–TRAN
Normal	1	PC+8	1	0	(PC+8)	0	0	0	
	2	ALU	B/W	1	RD	0	0	1	t
		PC+12							
Base=PC, Write back, Dest = PC	1	PC+8	1	0	(PC+8)	0	0	0	
	2	ALU	B/W	1	RD	0	0	1	t
	3	PC'	1	0	(PC')	1	0	0	
	4	PC'+4	1	0	(PC'+4)	1	0	0	
		PC'+8							

2

LOAD MULTIPLE REGISTERS

The first cycle of LDM is used to calculate the address of the first word to be transferred while performing a prefetch from memory. The second cycle fetches the first word and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is moved to the ALU A Bus input latch for holding in case it is needed to patch up after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The last cycle may be merged with the next instruction prefetch to form a single memory N-cycle.

If an abort occurs, the instruction continues to completion, but all register writing after the abort is disabled. The final cycle is altered to restore the modified base register (which may have been overwritten by the load activity before the abort occurred). If the PC is the base, write back is prevented.

When the PC is in the list of registers to be loaded, and assuming that no abort takes place, the current instruction pipeline must be invalidated. Note that the PC is always the last register to be loaded, so an abort at any point will prevent the PC from being overwritten.

TABLE 11. LOAD MULTIPLE REGISTERS

Type	Cycle	Address	–B/W	–R/W	Data	SEQ	–MREQ	–OPC
One Register	1	PC+8	1	0	(PC+8)	0	0	0
	2	ALU	1	0	ALU	0	1	1
	3	PC+12	1	0	–	1	0	1
		PC+12						
One Register, Dest = PC	1	PC+8	1	0	(PC+8)	0	0	0
	2	ALU	1	0	PC'	0	1	1
	3	PC+12	1	0	–	0	0	1
	4	PC'	1	0	(PC')	1	0	0
	5	PC'+4	1	0	(PC'+4)	1	0	0
		PC'+8						
N Registers, (N>1)	1	PC+8	1	0	(PC+8)	0	0	0
	2	ALU	1	0	(ALU)	1	0	1
	•	ALU+.	1	0	(ALU+.)	1	0	1
	N	ALU+.	1	0	(ALU+.)	1	0	1
	N+1	ALU+.	1	0	(ALU+.)	0	1	1
	N+2	PC+12	1	0	–	1	0	1
		PC+12						
N Registers, (N>1, incl. PC)	1	PC+8	1	0	(PC+8)	0	0	0
	2	ALU	1	0	(ALU)	1	0	1
	•	ALU+.	1	0	(ALU+.)	1	0	1
	N	ALU+.	1	0	(ALU+.)	1	0	1
	N+1	ALU+.	1	0	PC'	0	1	1
	N+2	PC+12	1	0	–	0	0	1
	N+3	PC'	1	0	(PC')	1	0	0
	N+4	PC'+4	1	0	(PC'+4)	1	0	0
		PC'+8						

INSTRUCTION CYCLE OPERATIONS (Cont.)

STORE MULTIPLE REGISTERS

Store multiple proceeds very much as load multiple, but without the final cycle. The restart problem is much more straightforward here, as there is no wholesale overwriting of registers with which to contend.

TABLE 12. STORE MULTIPLE REGISTERS

Type	Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC
One register	1	PC+8	1	0	(PC+8)	0	0	0
	2	ALU	1	1	RA	0	0	1
N Registers, (N>1)	1	PC+8	1	0	(PC+8)	0	0	0
	2	ALU	1	1	RA	1	0	1
	•	ALU+.	1	1	R.	1	0	1
	N	ALU+.	1	1	R.	1	0	1
	N+1	ALU+.	1	1	R.	0	0	1

SOFTWARE INTERRUPT AND EXCEPTION ENTRY

Exceptions (and software interrupts) force the PC to a particular value and refill the instruction pipeline from there. During the first cycle, the forced address is constructed and a mode change may take place. The return address is moved to register 14.

During the second cycle, the return address is modified to facilitate return. This modification is less useful than in the case of branch with link.

The third cycle is required only to complete the refilling of the instruction pipeline.

TABLE 13. SOFTWARE INTERRUPT & EXCEPTION ENTRY

Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC	-TRAN
1	PC + 8	1	0	(PC+8)	0	0	0	1
2	Xn	1	0	(Xn)	1	0	0	1
3	Xn+4	1	0	(Xn+4)	1	0	0	1

(For software interrupt, PC is the address of the SWI instruction; for interrupts and reset, PC is the address of the instruction following the last one to be executed before entering the exception; for prefetch abort, PC is the address of the aborting instruction; for data abort, PC is the address of the instruction following the one which attempted the aborted data transfer. Xn is the appropriate trap address.)

COPROCESSOR DATA OPERATION

A coprocessor data operation is a request from VL86C010 for the coprocessor to initiate some action. The action need not be completed for some time, but the coprocessor must commit to doing it before pulling CPB low.

If the coprocessor can never do the requested task, it should leave CPA and CPB to float high. If it can do the task, but can't commit right now, it should pull CPA low but leave CPB high until it can commit. VL86C010 will busy-wait until CPB goes low.

TABLE 14. COPROCESSOR DATA OPERATION

Type	Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC	-CPI	CPA	CPB
Ready	1	PC+8	1	0	(PC+8)	1	0	0	0	0	0
		PC+12									
Not Ready	1	PC+8	1	0	(PC+8)	0	1	0	0	0	1
	2	PC+8	1	0	–	0	1	1	0	0	1
	•	PC+8	1	0	–	0	1	1	0	0	1
	N	PC+8	1	0	–	0	0	1	0	0	0

INSTRUCTION CYCLE OPERATIONS (Cont.)

CO-PROCESSOR DATA TRANSFER (FROM MEMORY TO COPROCESSOR)

Here the coprocessor should commit to the transfer only when it is ready to accept the data. When CPB goes low, the CPU will produce addresses and expect the coprocessor to take the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred, and indicates the last transfer cycle by allowing CPA and CPB to float high.

The VL86C010 spends the first cycle (and any busy-wait cycles) generating the transfer address, and performs the write-back of the address during the transfer cycles.

2

TABLE 15. COPROCESSOR DATA TRANSFER (FROM MEMORY TO COPROCESSOR)

Type	Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC	-CPI	CPA	CPB
One Register Ready	1	PC+8	1	0	(PC+8)	0	0	0	0	0	0
	2	ALU	1	0	(ALU)	0	0	1	1	1	1
		PC+12									
One Register Not Ready	1	PC+8	1	0	(PC+8)	0	1	0	0	0	1
	2	PC+8	1	0	–	0	1	1	0	0	1
	•	PC+8	1	0	–	0	1	1	0	0	1
	N	PC+8	1	0	–	0	0	1	0	0	0
	N +1	ALU	1	0	(ALU)	0	0	1	1	1	1
		PC+12									
N Registers (N>1) Ready	1	PC+8	1	0	(PC+8)	0	0	0	0	0	0
	2	ALU	1	0	(ALU)	1	0	1	1	0	0
	•	ALU+.	1	0	(ALU+.)	1	0	1	1	0	0
	N	ALU+.	1	0	(ALU+.)	1	0	1	1	0	0
	N +1	ALU+.	1	0	(ALU+.)	0	0	1	1	1	1
		PC+12									
M Registers (M>1) Not Ready	1	PC+8	1	0	(PC+8)	0	1	0	0	0	1
	2	PC+8	1	0	–	0	1	1	0	0	1
	•	PC+8	1	0	–	0	1	1	0	0	1
	N	PC+8	1	0	–	0	0	1	0	0	0
	N+1	ALU	1	0	(ALU)	1	0	1	1	0	0
Ready	•	ALU+.	1	0	(ALU+.)	1	0	1	1	0	0
	N+M	ALU+.	1	0	(ALU+.)	1	0	1	1	0	0
	N+M+1	ALU+.	1	0	(ALU+.)	0	0	1	1	1	1
		PC+12									

INSTRUCTION CYCLE OPERATIONS (Cont.)

COPROCESSOR DATA TRANSFER (FROM CO-PROCESSOR TO MEMORY)

The VL86C010 controls these instructions exactly as for memory to coprocessor transfers, with the one exception that the –R/W line is inverted during the transfer cycle.

TABLE 16. COPROCESSOR DATA TRANSFER (FROM COPROCESSOR TO MEMORY)

Type	Cycle	Address	–B/W	–R/W	Data	SEQ	–MREQ	–OPC	–CPI	CPA	CPB
One Register Ready	1	PC+8	1	0	(PC+8)	0	0	0	0	0	0
	2	ALU	1	1	CPdata	0	0	1	1	1	1
		PC+12									
One Register Not Ready	1	PC+8	1	0	(PC+8)	0	1	0	0	0	1
	2	PC+8	1	0	–	0	1	1	0	0	1
	•	PC+8	1	0	–	0	1	1	0	0	1
	N	PC+8	1	0	–	0	0	1	0	0	0
	N+1	ALU	1	1	CPdata	0	0	1	1	1	1
		PC+12									
N Registers (N>1) Ready	1	PC+8	1	0	(PC+8)	0	0	0	0	0	0
	2	ALU	1	1	CPdata	1	0	1	1	0	0
	•	ALU+.	1	1	CPdata	1	0	1	1	0	0
	N	ALU+.	1	1	CPdata	1	0	1	1	0	0
	N +1	ALU+.	1	1	CPdata	0	0	1	1	1	1
		PC+12									
M Registers (M>1) Not Ready	1	PC+8	1	0	(PC+8)	0	1	0	0	0	1
	2	PC+8	1	0	–	0	1	1	0	0	1
	•	PC+8	1	0	–	0	1	1	0	0	1
	N	PC+8	1	0	–	0	0	1	0	0	0
	N+1	ALU	1	1	CPdata	1	0	1	1	0	0
Ready	•	ALU+.	1	1	CPdata	1	0	1	1	0	0
	N+M	ALU+.	1	1	CPdata	1	0	1	1	0	0
	N+M+1	ALU+.	1	1	CPdata	0	0	1	1	1	1
		PC+12									

INSTRUCTION CYCLE OPERATIONS (Cont.)

COPROCESSOR REGISTER TRANSFER (LOAD FROM COPROCESSOR)
Here the busy-wait cycles are much as the previous cycles, but the transfer is limited to one data word, and the CPU puts the word into the destination register in the third cycle. The third cycle may be merged with the following prefetch cycle into one memory N-cycle as with all VL86C010 register load instructions.

TABLE 17. COPROCESSOR REGISTER TRANSFER (LOAD FROM COPROCESSOR)

Type	Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC	-CPI	CPA	CPB
Ready	1	PC+8	1	0	(PC+8)	1	1	0	0	0	0
	2	PC+12	1	0	CPdata	0	1	1	1	1	1
	3	PC+12	1	0	–	1	0	1	1	–	–
		PC+12									
Not Ready	1	PC+8	1	0	(PC+8)	0	1	0	0	0	1
	2	PC+8	1	0	–	0	1	1	0	0	1
	•	PC+8	1	0	–	0	1	1	0	0	1
	N	PC+8	1	0	–	1	1	1	0	0	0
	N+1	PC+12	1	0	CPdata	0	1	1	1	1	1
	N+2	PC+12	1	0	–	1	0	1	1	–	–
		PC+12									

COPROCESSOR REGISTER TRANSFER (STORE TO COPROCESSOR)
This is the same as for the load from coprocessor, except that the last cycle is omitted.

TABLE 18. COPROCESSOR REGISTER TRANSFER (STORE TO COPROCESSOR)

Type	Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC	-CPI	CPA	CPB
Ready	1	PC+8	1	0	(PC+8)	1	1	0	0	0	0
	2	PC+12	1	1	Rd	1	0	1	1	1	1
		PC+12			–						
Not Ready	1	PC+8	1	0	(PC+8)	0	1	0	0	0	1
	2	PC+8	1	0	–	0	1	1	0	0	1
	•	PC+8	1	0	–	0	1	1	0	0	1
	N	PC+8	1	0	–	1	1	1	0	0	0
	N+1	PC+12	1	1	Rd	1	0	1	1	1	1
		PC+12									

INSTRUCTION CYCLE OPERATIONS (Cont.)

UNDEFINED INSTRUCTIONS AND COPROCESSOR ABSENT

When a coprocessor detects a coprocessor instruction which it cannot perform, and this must include all undefined instructions, it must not drive CPA or CPB. These will float high, causing the undefined instruction trap to be taken.

TABLE 19. UNDEFINED INSTRUCTIONS AND COPROCESSOR ABSENT

Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC	-CPI	CPA	CPB
1	PC+8	1	0	(PC+8)	0	1	0	0	1	1
2	PC+8	1	0	–	0	0	0	1	1	1
3	Xn	1	0	(Xn)	1	0	0	1	1	1
4	Xn+4	1	0	(Xn+4)	1	0	0	1	1	1
	Xn+8									

UNEXECUTED INSTRUCTIONS

Any instruction whose condition code is not met will fail to execute. It will add one cycle to the execution time of the code segment in which it is embedded.

TABLE 20. UNEXECUTED INSTRUCTIONS

Cycle	Address	-B/W	-R/W	Data	SEQ	-MREQ	-OPC
1	PC+8	1	0	(PC+8)	1	0	0
	PC+8						

INSTRUCTION SPEEDS

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle, one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for the routine may be calculated from these figures.

If the condition is met the instruction execution time is shown in Table 16 below.

TABLE 21. INSTRUCTION SPEEDS

Instruction Type	Instruction Timing Equation
Data Processing	1 S
Data Processing With Register Controlled Shift	1 S + 1 S
Data Processing With PC Modified	2 S + 1 N
Load Register	1 S + 1 N + 1 I
Load Register With PC Loaded	2 S + 2 N + 1 I
Store Register	2 N
Load Multiple	n S + 1 N + 1 I
Load Multiple With PC Loaded	(n + 1) S + 2 N + 1 I
Store Multiple	(n-1) S + 2 N
Branch and Branch With Link	2 S + 1 N
Software Interrupt, Trap	2 S + 1 N
Multiply and Multiple With Accumulate	1 S + m I
Coprocessor Data Operation	1 S + b I
Load or Store Coprocessor Data To Memory	1 S + 2 N + b I
Move From VL86C010 To Coprocessor Register	1 S + b I + 1 C
Move From Coprocessor To VL86C010 Register	1 S + (b + 1) I + 1 C

n is the number of words transferred.

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between 2^(2m-3) and 2^(2m-1)-1 inclusive takes m cycles for m>1. Multiplication by 0 or 1 takes 1 cycle. The maximum value m can take is 16.

I is an internal cycle. For systems using the VL86C110 Memory Controller, internal cycles are one clock, the same as S cycles.

b is the number of cycles spent in the coprocessor busy-wait loop.

If the condition is not met, all instructions take one S cycle.

TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter	VL86C010 - 10			VL86C010 - 12			Units	Conditions
		Min.	Typ.	Max.	Min.	Typ.	Max.		
tV	Clock Non-overlap Time	0	–	–	0	–	–	ns	
tCK	Clock Period	100	–	10000	80	–	10000	ns	
tCKL	Clock Period Low	40	–	10000	38	–	10000	ns	
tCKH	Clock Period High	40	–	10000	38	–	10000	ns	
tABE	Address Bus Enable	–	–	30	–	–	30	ns	
tABZ	Address Bus Disable	–	–	30	–	–	30	ns	
tALE	Address Latch Fall-Through	–	–	25	–	–	22	ns	
tALEL	ALE Low Time	–	–	10000	–	–	10000	ns	See Note 1
tADDRS	Ø2 To Address Valid Delay	–	–	35	–	–	35	ns	See Note 2
tADDRN	Ø1 To Address Valid Delay	–	–	95	–	–	90	ns	
tAH	Address Bus Hold Time	5	–	–	5	–	–	ns	
tDBE	Data Bus Enable Time	–	–	45	–	–	40	ns	
tDBZ	Data Bus Disable Time	–	–	45	–	–	40	ns	
tDOUT	Data Bus Output Delay	–	–	55	–	–	50	ns	
tDOH	Data Bus Hold Time	10	–	–	10	–	–	ns	
tDIS	Data In Setup Time	10	–	–	10	–	–	ns	
tDIH	Data In Hold Time	5	–	–	5	–	–	ns	
tABTS	ABRT Setup Time	25	–	–	20	–	–	ns	
tABTH	ABRT Hold Time	5	–	–	5	–	–	ns	
tIRS	Interrupt Setup Time	10	–	–	10	–	–	ns	See Note 3
tRWD	Ø2 To –R/W Valid	–	–	40	–	–	35	ns	See Note 4
tRWH	–R/W Hold Time	5	–	–	5	–	–	ns	
tMSD	Ø1 To –MREQ And SEQ Delay	–	–	55	–	–	45	ns	
tMSH	–MREQ And SEQ Hold Time	5	–	–	5	–	–	ns	
tBWD	Ø2 To –B/W Valid	–	–	40	–	–	35	ns	
tBWH	–B/W Hold Time	5	–	–	5	–	–	ns	
tMDD	Ø1 To M1 - M0 Valid	–	–	35	–	–	35	ns	
tMDH	M1 - M0 Hold Time	5	–	–	5	–	–	ns	

Notes:

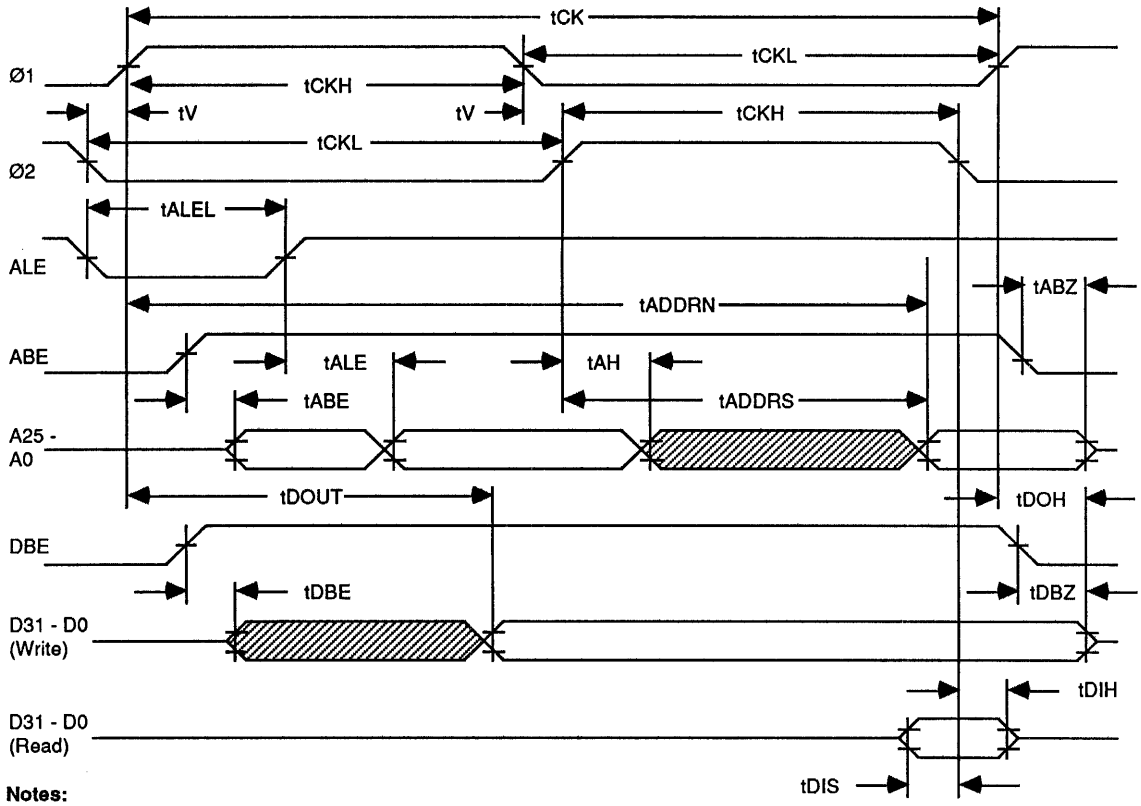
1. ALE controls a dynamic storage latch; this parameter is specified to ensure that the stored charge cannot leak sufficiently to generate intermediate logic levels in the associated logic.
2. The Ø1 to address delay only applies to non-sequential cycles, when the address is being calculated in the ALU. For sequential cycles the address will be valid earlier, at the time given from Ø2. TADDRS applies to sequential and non-sequential cycles.
3. The interrupt and reset inputs may be asynchronous. This time will guarantee that the interrupt request is latched during this cycle.
4. The worst case for –R/W occurs only when an address exception happens during a data store operation. The address exception causes –R/W to switch to read to prevent erroneous writing of memory.

TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter	VL86C010 - 10			VL86C010 - 12			Units	Conditions
		Min.	Typ.	Max.	Min.	Typ.	Max.		
tOPCD	Ø2 To –OPC Valid	–	–	40	–	–	40	ns	
tOPCH	–OPC Hold Time	5	–	–	5	–	–	ns	
tTRMD	Ø1 To –TRAN Valid	–	–	35	–	–	30	ns	
tTRMH	–TRAN Hold Time	5	–	–	5	–	–	ns	
tTRDD	Ø2 To –TRAN Valid	–	–	45	–	–	40	ns	See Note 1
tTRDH	–TRAN Hold Time	5	–	–	5	–	–	ns	
tCPS	CPA, –CPB Setup Time	35	–	–	30	–	–	ns	
tCPH	CPA, –CPB Hold Time	5	–	–	5	–	–	ns	
tCPI	Ø1 To –CPI Delay	–	–	35	–	–	30	ns	
tCPIH	–CPI Hold Time	5	–	–	5	–	–	ns	

TIMING DIAGRAMS

PROCESSOR DATA BUS

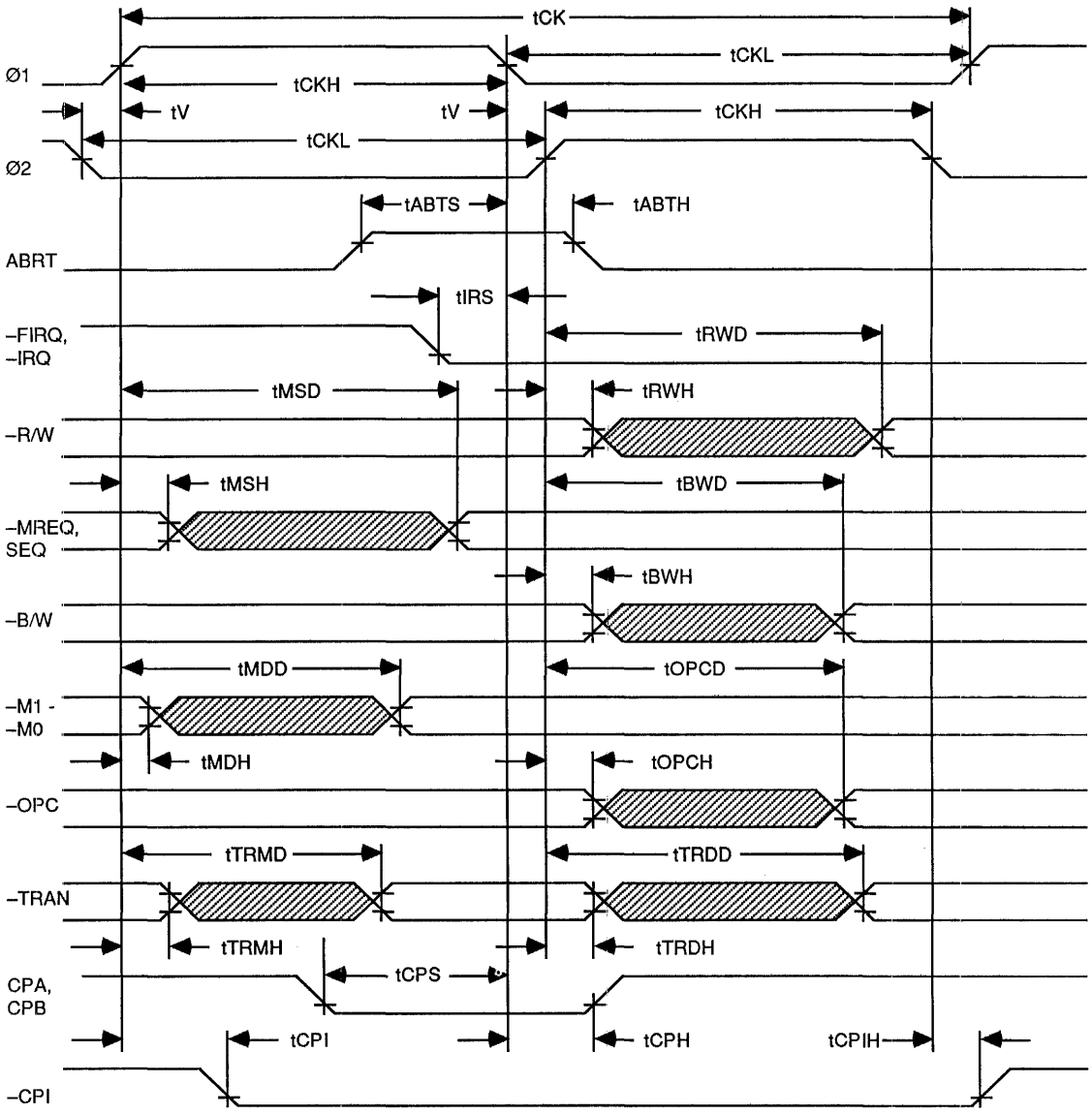


Notes:

1. –TRAN will only change during Ø2 as the result of a forced translation single data transfer operation while in the user mode. Otherwise, it will change during Ø1 when the mode change to/from user mode occurs.

TIMING DIAGRAMS

PROCESSOR CONTROL SIGNALS



ABSOLUTE MAXIMUM RATINGS

Ambient Operating Temperature	−10°C to +80°C
Storage Temperature	−65°C to +150°C
Supply Voltage to Ground Potential	−0.5 V to VCC +0.3 V
Applied Output Voltage	−0.5 V to VCC +0.3 V
Applied Input Voltage	−0.5 V to +7.0 V
Power Dissipation	2.0 W

Stresses above those listed may cause permanent damage to the device. These are stress ratings only. Functional operation of this device at these or any other conditions above those indicated in this data sheet is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

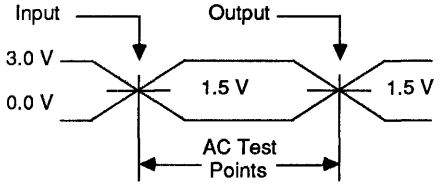
DC CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ± 5%

Symbol	Parameter		Min	Typ	Max	Unit	Conditions
VOHT	Output High Voltage, TTL-DATABUS		VCC−0.75	–	VCC	V	IOH = −5.0 mA
VOLT	Output Low Voltage, TTL-DATABUS		–	–	0.8	V	IOL = 5.0 mA
VOHC	Output High Voltage, CMOS		VCC−0.75	–	VCC	V	IOH = −2.5 mA
VOLC	Output Low Voltage, CMOS		–	–	0.4	V	IOL = 2.5 mA
VIH	Input High Voltage	Ø1, Ø2	VCC−0.3	–	VCC+0.3	V	
		All Others	2.4	–	VCC+0.3	V	
VIL	Input Low Voltage	Ø1, Ø2	−0.3	–	0.3	V	
		All Others	−0.3	–	0.8	V	
ILI	Input Leakage Current		–	–	10	µA	VIN = 0 V to VCC
ILO	Output Leakage Current		–	–	10	µA	VOUT = 0 V to VCC
ICC	Operating Supply Current		–	20	40	mA	(Note 1)
IOS	Output Short Circuit Current		–	–	40	mA	

CAPACITANCE: TA = 25°C, f = 1.0 MHz

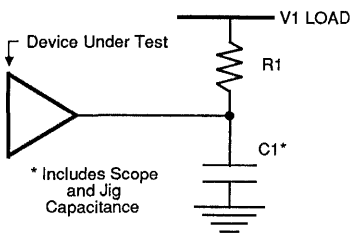
Symbol	Parameter	Min	Max	Unit	Conditions
CI	Clock Input Capacitance (Ø1, Ø2)	–	15	pF	VIN = 0 V (Note2)
	Other Input Capacitance	–	5	pF	VIN = 0 V (Note2)
CO	Output Capacitance	–	8	pF	VOUT = 0 V (Note 2)

FIGURE 3. TEST WAVEFORMS



V1 LOAD = 2.4 V, DATABUS
V1 LOAD = 2.3 V, OTHERS
R1 = 160Ω, DATABUS
R1 = 750Ω, OTHER OUTPUTS
C1 = 100 pF, DATABUS
C1 = 50 pF, CPI, ADDR.BUS
C1 = 15 pF, OTHER OUTPUTS

FIGURE 4. TEST LOAD CIRCUIT



Notes:
1. Measured with outputs unloaded, at 10 MHz. Add 4 mA per MHz.
2. Periodically sampled, rather than 100% tested.

PROGRAMMERS' MODEL

The VL86C010 processor has a 32-bit data bus and a 26-bit address bus. The processor supports two data types, eight-bit bytes and 32-bit words, where words must be aligned on 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 VL86C010 supports four modes of operation, including protected supervisor and interrupt handling modes.

BYTE SIGNIFICANCE

Some programming techniques may write a 32-bit (word) quantity to memory, but will later retrieve the data as a sequence of byte (8-bit) items. For these purposes, the processor stores word data in least-significant-first (LSB first) order. This means that the least significant bytes of a 32-bit word occupies the lowest byte address. (The VLSI Technology, Inc. assemblers, none the less, display compiled data in MSBs-first order, but for the sake of clarity only. The internal machine representation is preserved as LSBs-first.)

REGISTERS

The processor has 27 registers (32-bits each), 16 of which are visible to the programmer at any time. The visible subset depends on the current processor mode; special registers are switched in to support interrupt and supervisor processing. The register bank organization is shown in Table 17.

User mode is the normal program execution state; registers R15 - R0 are directly accessible.

All registers are general purpose and may be used to hold data or address values, except that register R15 contains the Program Counter (PC) and the Processor Status Register (PSR). Special bits in some instructions allow the PC and PSR to be treated together or separately as required. Figure 5 shows the allocation of bits within R15.

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 and R14_firq are used similarly to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within supervisor or interrupt routines.

TABLE 22. REGISTER ORGANIZATION

					Typical Use
R0	General				
R1	General				
R2	General				
R3	General				
R4	General				
R5	General				
R6	General				
R7	General				General Usage
R8	General			FIRQ	
R9	General			FIRQ	
R10	General			FIRQ	
R11	General			FIRQ	
R12 (FP)	General			FIRQ	Data Frame (by convention)
R13 (SP)	General	Supervisor	IRQ	FIRQ	Stack Pointer (by convention)
R14 (LK)	General	Supervisor	IRQ	FIRQ	R15 Save Area for BL or Interrupts
R15 (PC)	(Shared by all Modes)				System Program Counter

TABLE 23. BYTE ADDRESSING

31			0	Word Address Value
Byte Addr. 0003	Byte Addr. 0002	Byte Addr. 0001	Byte Addr. 0000	0000
Byte Addr. 0007	Byte Addr. 0006	Byte Addr. 0005	Byte Addr. 0004	0001

FIRQ Processing - The FIRQ mode (described in the Exceptions section) has seven private registers mapped to R14 - R8 (R14_fiq-R8_fiq). Many FIRQ programs will not need to save any registers.

IRQ Processing - The IRQ state has two private registers mapped to R14 and R13 (R14_irq and R13_irq).

Supervisor Mode - The SVC mode (entered on SWI instructions and other traps) has two private registers mapped to R14 and R13 (R14_svc and R13_svc).

The two private registers allow the IRQ and supervisor modes each to have a private stack pointer and link register. Supervisor and IRQ mode programs are expected to save the user state on their respective stacks and then use the user registers, remembering to restore the user state before returning.

User mode registers are accessible in the other modes by using LDM or STM and setting the S bit.

In user mode only the N, Z, C, and V bits of the PSR may be changed. The I, F, and Mode flags will change only when an exception arises. In supervisor and interrupt modes all flags may be manipulated directly.

EXCEPTIONS

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that (for instance) 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 processor handles exceptions by using the banked registers to save state. The old PC and PSR are copied into the appropriate R14, and the PC and processor mode bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 should be saved onto a stack in main memory before re-enabling the interrupt. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled.

FIRQ - The FIRQ (Fast Interrupt Request) exception is externally generated by taking the –FIRQ pin 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; therefore, the overhead of context switching is minimized. The FIRQ exception may be disabled by setting the F flag in the PSR (but note that this is not possible from user mode). If the F flag is clear the processor checks for a low level on the output of the FIRQ synchronizer at the end of each instruction.

The impact upon execution of an FIRQ interrupt is defined in Table 19. The return-from-interrupt sequence is also defined there. This will restore the original processor state and cause execution to resume at the instruction following the interrupted one.

IRQ - The IRQ (Interrupt Request) exception is a normal interrupt caused by a low level on the –IRQ pin. It has a lower priority than FIRQ, and is masked out when a FIRQ 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 processor checks for a low level on the output of the IRQ synchronizer at the end of each instruction.

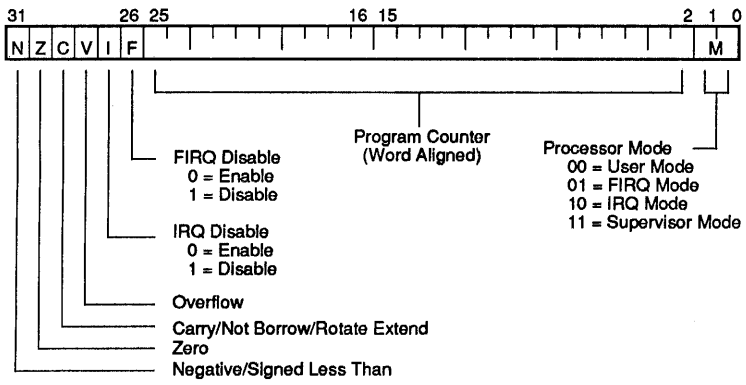
The impact upon execution of an IRQ interrupt is defined in Table 19. The return-from-interrupt sequence is also defined there. This will restore the original processor state and re-enable the IRQ interrupt and will cause execution to resume at the instruction following the interrupted one.

Address Exception Trap - An address exception arises whenever a data transfer is attempted with a calculated address above 3FFFFFFH. The VL86C010 address bus is 26 bits wide, and an address calculation will have a 32-bit result. If this result has a logic one in any of the top six bits it is assumed that the address is an error and the address exception trap is taken.

Note that a branch cannot cause an address exception and a block data transfer instruction, which starts in the legal area but increments into the illegal area, will not trap. The check is performed only on the address of the first word to be transferred.

When an address exception is seen, the processor will respond as defined in Table 19. The return-from-interrupt sequence is also defined there. This will resume execution of the interrupted code sequence and restore the original processor state.

FIGURE 5. PROGRAM COUNTER AND PROCESSOR STATUS REGISTER



Normally, an address exception is caused by erroneous code and it is inappropriate to resume execution. If a return is required from this trap, use SUBS PC, R14_svc, 4, as defined in Table 19. This will return to the instruction after the one causing the trap.

Abort - The ABRT 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 disk, and considerable processor activity may be required to recover the data before the access can be performed successfully. The processor checks for an abort at the end of the first phase of each bus cycle. When successfully aborted, the VL86C010 will respond in one of three ways:

(i) If the abort occurred during an instruction prefetch (a prefetch abort), the prefetched instruction is marked as invalid; when it comes to execution, it is reinterpreted as below. (If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, the abort will have no effect.)

(ii) If the abort occurred during a data access (a data abort), the action depends on the instruction type. Data transfer instructions (LDR, STR) are aborted as though they had not executed. The 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.

(iii) If the abort occurred during an internal cycle it is ignored.

Then, in cases (i) and (ii), the processor will respond as defined in Table 19.

The return from Prefetch Abort defined in Table 19 will attempt to execute the aborting instruction (which will only be effective if action has been taken to remove the cause of the original abort). A Data Abort requires any auto-indexing to be reversed before returning to re-execute the offending instruction. The return is performed as defined in the Table 19.

The abort mechanism allows a demand paged virtual memory system to be implemented when a suitable memory management unit (such as the VL86C110) 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 which 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 is used for getting into supervisor mode, usually to request a particular supervisor function. The processor response to the SWI instruction is defined in Table 19, as is the method of returning. The indicated return method will return to the instruction following the SWI.

2

TABLE 24. EXCEPTION TRAP CONSIDERATIONS

Trap Type	CPU Trap Activity	Program Return Sequence
Reset	1. Save R15 in R14 (SVC). 2. Force M1, M0 to SVC mode, and set F & I status bits in PC. 3. Force PC to 0x000000.	(n/a)
Undefined Instruction	1. Save R15 in R14 (SVC). 2. Force M1, M0 to SVC mode, and set I status bit in the PC. 3. Force PC to 0x000004.	MOVS PC, R14 ; SVC's R14.
Software Interrupt	1. Save R15 in R14 (SVC). 2. Force M1, M0 to SVC mode, and set I status bit in the PC. 3. Force PC to 0x000008.	MOVS PC, R14 ; SVC's R14.

Prefetch and Data Aborts	1. Save R15 in R14 (SVC).
	2. Force M1, M0 to SVC mode, and set I status bit in the PC.
	3. Force PC to 0x000010-data.
	Force PC to 0x0000C-Pre-.
		Prefetch Abort:
		SUBS PC, R14,4 ; SVC's R14.
		Data Abort:
		SUBS PC, R14,8 ; SVC's R14.

Address Exception	1. Convert Stores to Loads. 2. Complete the instruction (see text for details). 3. Save R15 in R14 (SVC). 4. Force M1, M0 to SVC mode, and set I status bit in the PC. 5. Force PC to 0x000014.	SUBS PC, R14,4 ; SVC's R14. (Returns CPU to address following the one causing the trap.)
IRQ	1. Save R15 in R14 (IRQ). 2. Force M1, M0 to IRQ mode, and set I status bit in the PC. 3. Force PC to 0x000018.	SUBS PC, R14,4 ; IRQ's R14.
FIRQ	1. Save R15 in R14 (FIRQ). 2. Force M1, M0 to FIRQ mode, and set the F and I status bits in the PC. 3. Force PC to 0x00001C.	SUBS PC, R14,4 ; FIRQ's R14.

Undefined Instruction Trap - When the VL86C010 executes a coprocessor instruction or an undefined instruction, it offers it to any coprocessors which may be present. If a coprocessor can perform this instruction but is busy at that moment, the processor will wait until the coprocessor is ready. If no coprocessor can handle the instruction, the VL86C010 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 undefined instruction trap is taken, the VL86C010 will respond as defined in Table 19. The return from this trap (after performing a suitable emulation of the required function) defined in Table 19 will return to the instruction following the undefined instruction.

Reset - When RES goes high, the processor will stop the currently executing instruction and start executing no-ops. When Reset goes low again, it will respond as defined in Table 19. There is no meaningful return from this condition.

Vector Table
The conventional means of implementing an interrupt dispatch function is to provide a table of jumps to the appropriate processing table as shown below:

Address	Function
0000000	Reset
0000004	Undefined instruction
0000008	Software interrupt
000000C	Abort (prefetch)
0000010	Abort (data)
0000014	Address exception
0000018	IRQ
000001C	FIRQ

These are byte addresses, and each contains a branch instruction pointing to the relevant routine. The FIRQ routine might reside at 000001CH onwards, and thereby avoid the need for (and execution time of) a branch instruction.

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) Address exception and Data aborts
3) FIRQ
4) IRQ
5) Prefetch abort
6) Undefined Instruction and SWIs (lowest priority)

Note that not all exceptions can occur at once. Address exception and data abort are mutually exclusive, since if an address is illegal the processor ignores the ABRT input. Undefined instruction and software interrupt are also mutually exclusive since they each correspond to particular (non-overlapping) decodings of the current instruction.

If an address exception or data abort occurs at the same time as a FIRQ and FIRQs are enabled (i.e., the F flag in the PSR is clear), the processor will enter the address exception or data abort handler and then immediately proceed to the FIRQ vector. A normal return from FIRQ will cause the address exception or data abort handler to resume execution. Placing address exception and data abort at a higher priority than FIRQ is necessary to ensure that the transfer error does not escape detection, but the time for this exception entry should be reflected in worst case FIRQ latency calculations.

Interrupt Latencies - The worst case latency for FIRQ, 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 FIRQ entry (Tfiq). At the end of this time, the processor will be executing the instruction at 1CH.

Tsyncmax is 2.5 processor cycles, Tldm is 18 cycles, Texc is three cycles, and Tfiq is two cycles. The total time is 25.5 processor cycles, which is just over 2.5 microseconds in a system using a continuous 10 MHz processor clock. In a DRAM based system running at 4 and 8 MHz (for example, using the VL86C110) this time becomes 4.5 microseconds, and 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 FIRQ has higher priority and can delay entry into the IRQ handling routine for an arbitrary length of time.

The minimum lag for interrupt recognition for FIRQ or IRQ consists of the shortest time the request can take through the synchronizer (Tsyncmin) plus Tfiq. This is 3.5 processor cycles. The FIRQ should be held until the mode bits indicate FIRQ mode. It may be safely held until cleared by an I/O instruction in the FIRQ service routine.

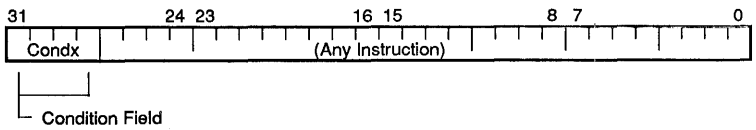
INSTRUCTION SET

All VL86C010 instructions are conditionally executed which means that their execution may or may not take place depending on the values of the N, Z, C, and V flags in the PSR at the end of the preceding instruction.

If the ALways condition is specified, the instruction will be executed irrespective of the flags, and likewise the Never condition will cause it not to be executed (it will be a no-op, taking one cycle and having no effect on the processor state).

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 were different, the compare instruction would have cleared the Z flag, and the instruction will not be executed.

FIGURE 6. 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

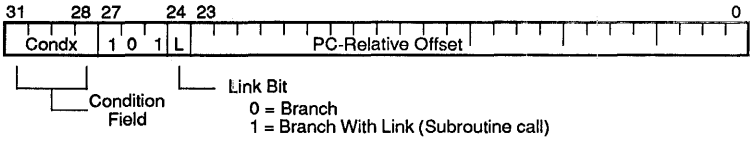
BRANCH, BRANCH AND LINK (B, BL)

The B and BL instructions are only executed if the condition code field is true.

All branches support a 24-bit offset. The offset is shifted left two bits and added to the PC, with overflows being ignored. The branch can, therefore, reach any word aligned address within the address space. The branch offset must take account of the prefetch operation, which causes the PC to be two words ahead of the current instruction.

Link bit - Branch with Link writes the old PC and PSR into R14 of the current bank. The PC value written into the link register (R14) is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction.

FIGURE 7. BRANCH, AND BRANCH WITH LINK (B, BL)



Return from Subroutine - When returning to the caller, there is an option to restore or to not restore the PSR. The following table illustrates the available combinations.

	Link Register Valid	Link Saved to a Stack
Restoring PSR:	MOVS PC,R14	LDM Rn!, (PC)^
Not Restoring PSR:	MOV PC,R14	LDM Rn!, (PC)

Syntax:

B(L){cond} <expression>

where *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 Condition Code section (EQ, NE, VS, etc.). If absent then AL (Always) will be used.

expression is the destination. The assembler calculates the relative (word) offset.

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

Examples:

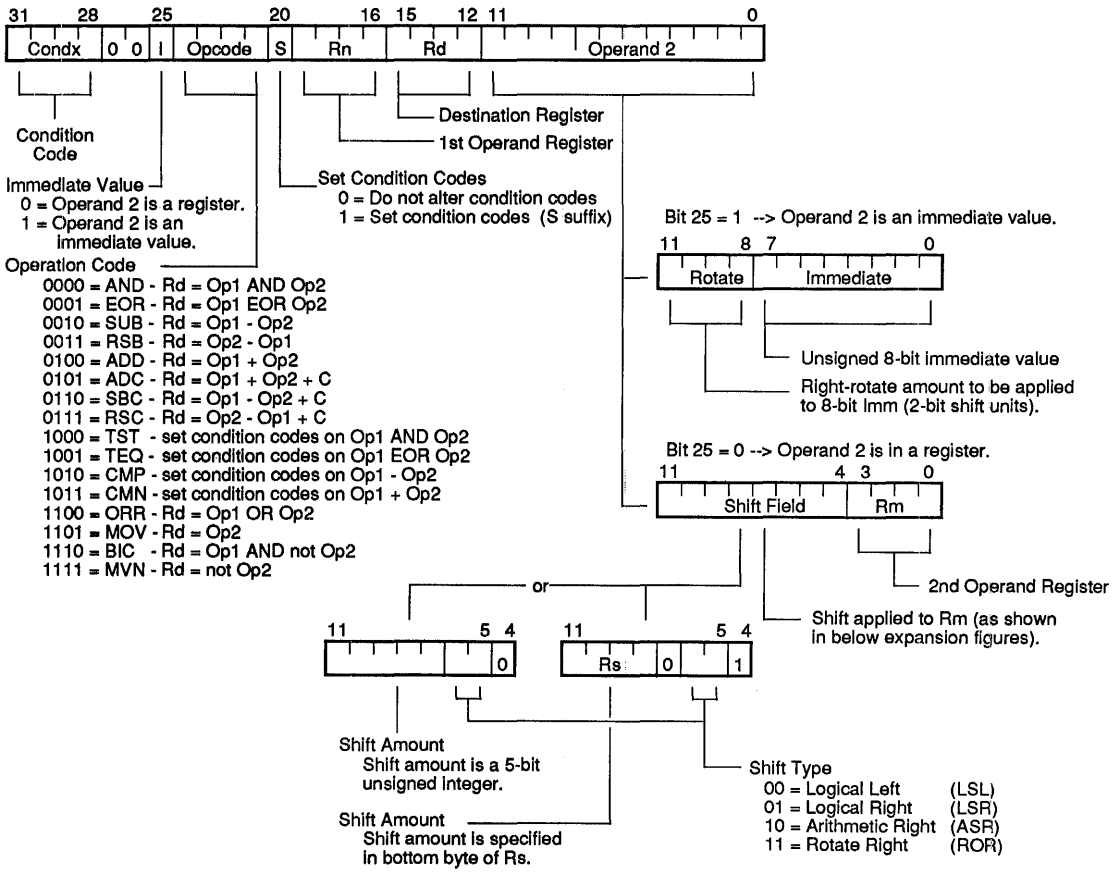
```
Here    BAL     Here            ; Assembles to EAFFFFFE. (Note effect of PC offset)
        B       There           ; Always condition used as default
        CMP     R1,0            ; Compare register one with zero, and branch to Fred if
        BEQ     Fred            ; register one was zero. Else continue next instruction.
        BL      ROM + Sub       ; Unconditionally call subroutine at computed address.
        ADDS    R1, 1           ; Add one to register one, setting PSR flags on the result.
        BLCC    Sub             ; Call Sub if the C flag is clear, which will be the case unless
                                ; R1 contained FFFFFFFFH. Else continue next instruction.
        BLNV    Sub             ; Never call subroutine (this is a NO-OP).
```

ALU INSTRUCTIONS

The ALU-type instruction is only executed if the condition is true. The various conditions are defined in the Condition Code section.

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 PSR 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 therefore should always have the S bit set. (The assembler treats TST, TEQ, CMP and CMN as TSTS, TEQS, CMPS and CMNS by default).

FIGURE 8. ALU INSTRUCTION TYPES



DATA PROCESSING OPERATIONS

Assembler Mnemonic	Opcode	Action
AND	0000	Bit-wise logical AND of operands
EOR	0001	Bit-wise logical Exclusive Or of operands
SUB	0010	Subtract operand 2 from operand 1
RSB	0011	Subtract operand 1 from operand 2
ADD	0100	Add operands
ADC	0101	Add operands plus carry (PSR C flag)
SBC	0110	Subtract operand 2 from operand 1 plus carry
RSC	0111	Subtract operand 1 from operand 2 plus carry
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	Bit-wise logical OR of operands
MOV	1101	Move operand 2 (operand 1 is ignored)
BIC	1110	Bit clear (bit-wise AND of operand 1 and NOT operand 2)
MVN	1111	Move NOT operand 2 (operand 1 is ignored)

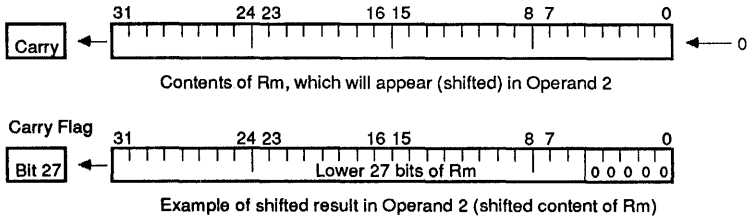
PSR Flags - The operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15) the V flag in the PSR will 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 zeroes, 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 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the PSR 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 2'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 2'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 as shown in Figure 8.

When the shift amount is specified in the instruction, it is contained in a 5-bit field which may take any value from zero 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 zeroes, and the high bits of Rm which 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 PSR when the ALU operation is in the logical class (see above). For example, the effect of LSL 5 is:

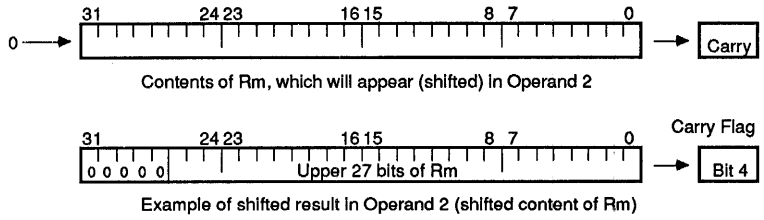
FIGURE 9. LOGICAL SHIFT LEFT (LSL)



Note that LSL 0 is a special case where the shifter carry out is the old value of the PSR 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 the following effect:

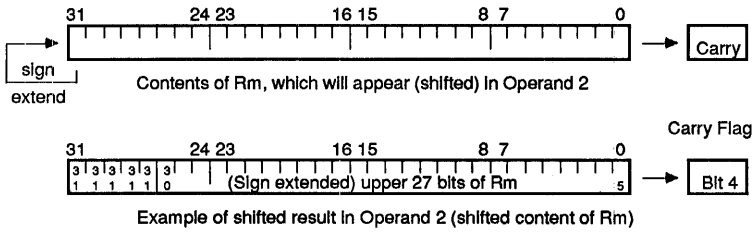
FIGURE 10. LOGICAL SHIFT RIGHT (LSR)



The form of the shift field which might be expected to correspond to LSR 0 is used to encode LSR 32, which has the 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. Therefore, the assembler converts LSR 0, ASR 0, and ROR 0 into LSL 0, and allows LSR 32 to be specified.

The Arithmetic Shift Right (ASR) is similar to the logical shift right, except that the high bits are filled with replicates of the sign bit (bit 31) of the Rm register, instead of zeros. This signed shift preserves the correct representation of a (signed) negative integer to be divided by powers of two via a right shift. For example, ASR 5 has the following effect:

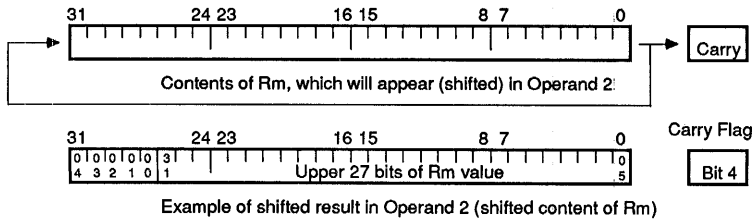
FIGURE 11. ARITHMETIC SHIFT RIGHT (ASR)



The form of the shift field which 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 the sign bit (bit 31) of Rm. The result is, therefore, all ones or all zeros according to the value of bit 31 of Rm.

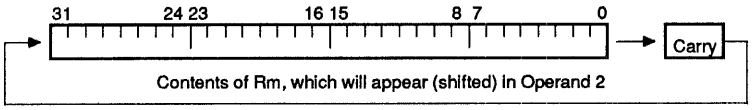
Rotate Right (ROR) operations reuse the bits which "overshoot" in a logical shift right operation by wrapping them around at the high end of the result. For example, the effect of a ROR 5 is:

FIGURE 12. ROTATE RIGHT (ROR)



The form of the shift field which 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 PSR C flag to the most significant end of the contents of Rm:

FIGURE 13. ROTATE RIGHT EXTENDED (RRX)



Register-Based Shift Counts - Only the least significant byte of the contents of Rs is used to determine the shift amount. If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the PSR 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.

Shifts of 32 or More - The result will be a logical extension of the shifting processes described above:

Shift	Action
LSL by 32	Result zero, carry out equal to bit zero of Rm.
LSL by more than 32	Result zero, carry out zero.
LSR by 32	Result zero, carry out equal to bit 31 of Rm.
LSR by more than 32	Result zero, carry out zero.
ASR by 32 or more	Result filled with, and carry out equal to, bit 31 of Rm.
ROR by 32	Result equal to Rm, and carry out equal to, bit 31 of Rm.
ROR by more than 32	Same result and carry out as ROR by n-32. Therefore, repeatedly subtract 32 from count until within the range one to 32.

Note: The zero in bit seven of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or an undefined instruction.

Immediate Operand Rotation - The immediate operand rotate field is a 4-bit unsigned integer which specifies a shift operation on the 8-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 command constants to be generated, for example all powers of two. Another example is that the 8-bit constant may be aligned with the PSR flags (bits zero, one, and 26 to 31). All the flags can thereby be initialized in one TEQP instruction.

Writing to R15 - When Rd is a register other than R15, the condition code flags in the PSR may be updated from the ALU flags as described above. When Rd is R15 and the S flag in the instruction is set, the PSR is overwritten by the corresponding bits in the ALU result, so bit 31 of the result goes to the N flag, bit 30 to the Z flag, bit 29 to the C flag and bit 28 to the V flag. In user mode the other flags (I, F, MI, MO) are protected from direct change, but in non-user modes these will also be affected, accepting copies of bits 27, 26, one and zero of the result respectively.

When one of these instructions is used to change the processor mode (which is only possible in a non-user mode), the following instruction should not access a banked register (R14-R8) during its first cycle. A no-op should be inserted if the next instruction must access a banked register. Accesses to the unbanked registers (R7-R0 and R15) are safe.

If the S flag is clear when Rd is R15, only the 24 PC bits of R15 will be written. Conversely, if the instruction is of a type which does not normally produce a result (CMP, CMN, TST, TEQ) but Rd is R15 and the S bit is set, the result will be used in this case to update those PSR flags which are not protected by virtue of the processor mode.

Setting PSR Bits - It is suggested that TEQP be used to set PSR bits in SVC mode. Because these bits are not presented to the ALU input (even when R15 is the operand), the TEQP's operands replace all current PSR bits.

For example, to remain in SVC mode but set the interrupt-disable bits, use a 'TEQP PC, ØxC000003' instruction.

R15 as an Operand - If R15 is used as an operand in a data processing instruction it can present different values depending on which operand position it occupies. It will always contain the value of the PC. It may or may not contain the values of the PSR flags as they were at the completion of the previous instruction.

When R15 appears in the Rm position it will give the value of the PC together with the PSR flags to the barrel shifter.

When R15 appears in either of the Rn or Rs positions, it will give the value of the PC alone with the PSR bits replaced by zeroes.

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 eight bytes ahead when used as Rs, and 12 bytes ahead when used as Rn or Rm.

Syntax:

MOV, MVN single operand instructions:
<opcode>{cond}{S} Rd,<Op2>

CMP, CMN, TEQ, TST - instructions not producing a result:
<opcode>{cond}{P} Rn,<Op2>

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

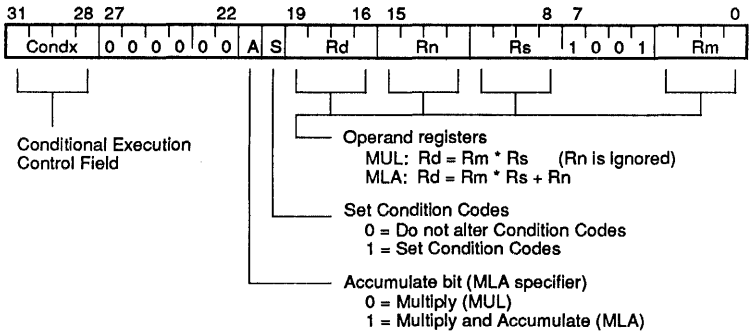
where *Op2*	Is *Rm{<shift>}* or, *<expression>*
cond	Two-character condition mnemonic, see Condition Code section.
S	Set condition codes if S present (implied for CMP, CMN, TEQ, TST).
P	Make Rd = R15 in instructions where Rd is not specified, otherwise Rd will default to R0. (Used for changing the PSR directly from the ALU result.)
Rd, Rn and *Rm*	Are any valid register name, such as R0-R15, PC, SP, or LK.
<shift>	Is *<shiftname> <register>* or *<shiftname> expression*, or *RRX* (rotate right one bit with extend).
<shiftname>s	Are any of: *ASL, LSL, LSR, ASR*, or *ROR*.

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

Examples:

```
ADDEQ     R2, R4, R5          ; Equivalent to: if (ZFLAG) R2 = R4+R5.

TEQS      R4, 3               ; Test R4 for equality with 3 (The S is redundant, as the assembler
                              ; assumes it. Equivalent to: ZFLAG = R4==3.

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.
                              ; Equivalent to: R4 = R5 - (R7>>R2).

TEQP      R15, 0;             ; (Assume non-user mode here). Change to
                              ; user mode and clear the N,Z,C,V,I, and F
                              ; flags. Note that R15 is in the Rn position, so
                              ; it comes without the PSR flags.
                              ; Equivalent to: R15 = FLAGS = 0.

MOVNV R0, R0                  ; Is a no-op, avoiding mode-change hazard.
                              ; Equivalent to: R0 = R0.

MOV       PC, LK              ; Equivalent to: PC = LK, or PC = R14.
                              ; Return from subroutine (R14 is an active one).

MOVS      PC, R14             ; Equivalent to: PC, PSR = R14.
                              ; Return from subroutine, restoring the status.
```

FIGURE 14. MULTIPLY, AND MULTIPLY-ACCUMULATE (MUL, MLA)



The Multiply and Multiply-Accumulate instructions use a 2-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 violated.)

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.

The destination register Rd should not be R15 since it is protected from modification by these instructions. The instruction will have no effect, except that meaningless values will be placed in the PSR flags if the S bit is set. All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

PSR Flags - Setting the PSR 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.

Writing to R15 - As mentioned above, R15 must not be used as the destination register (Rd). If it is so used, the instruction will have no effect except possibly to scramble the PSR flags.

R15 As An Operand - R15 may be used as one or more of the operands, though the result will rarely be useful. When used as Rs, the PC bits will be used without the PSR flags and the PC value will be eight bytes on from the address of the multiply instruction. When used as Rn, the PC bits will be used along with the PSR flags, and the PC will again be eight bytes on from the address of the instruction. When used as Rm, the PC bits will be used together with the PSR flags, but the PC will be the address of the instruction plus 12 bytes in this case.

Syntax

	MUL{cond}{S}	Rd, Rm, Rs
	MLA {cond}{S}	Rd, Rm, Rs, Rn
where	*cond*	Is a two-character condition code mnemonic
	S	Set condition codes if present.
	Rd, Rm, Rs and *Rn*	Are valid register mnemonics, such as R0-R15, SP, LK, or PC.

Notes:

Rd must not be R15 (PC), and must not be the same as Rm.

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

Examples:

```
MUL       R1, R2, R3        ; R1 = R2 * R3. (R1,R2,R3 = Rd,Rm,Rs)
MLAEQS    R1, R2, R3, R4    ; Equivalent to: if (ZFLAG) R1 = R2*R3 + R4.
                            ; Condition codes are set, based on the result.

; The multiply instruction may be used to synthesize higher precision multiplications.
; For instance, multiply two 32-bit integers and generate a 64-bit result:
MOV       R0, R1 LSR 16     ; R0 (temporary) = top half of R1.
MOV       R4, R2 LSR 16     ; R4 = top half of R2.
BIC       R1, R1, R0 LSL 16 ; R1 = bottom half of R1.
BIC       R2, R2, R4 LSL 16 ; R2 = bottom half of R2.
MUL       R3, R0, R2        ; Low section of result.
MUL       R2, R0, R2        ; Middle section of result.
MUL       R1, R4, R1        ; Middle section of result.
MUL       R4, R0, R4        ; High section of result.
ADDS      R1, R2, R1        ; Add middle sections. (MLA not used, as we need R3 correct).
ADDCS     R4, R4, 0x10000   ; Carry from above add.
ADDS      R3, R3, R1 LSL 16 ; R3 is now bottom 32 product bits.
ADC       R4, R4, R1 LSR 16 ; R4 is now top 32 bits.
```

Notes:

1. R1, R2 are resigters containing the 32-bit integers. R3, R4 are registers for the 64-bit result.
2. R0 is a temporary register.
3. R1 and R2 are overwritten during the multiply.

Load/Store Value from Memory (LDR,STR)

The register load/store instructions are used to load or store single bytes or words of data. The LDR and STR instructions differ from MOV instructions in that they move data between registers and a specified memory address. In contrast, the MOV instructions move data between registers, or move a constant (contained in the instruction) into a register.

The memory address used in LDR/STR transfers is calculated by adding an offset to or subtracting an offset from a base register. Typically, a load of a labeled memory location involves the loading via a (signed) offset from the current PC. Regardless of the base register used, the result of the offset 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 binary immediate value in the instruction, or a second register (possibly shifted in some manner). 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 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.

Hardware Address Translation - The only use of the W bit in a post-indexed data transfer is in non-user mode code where setting the W bit forces the –TRAN pin low for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this pin, as when the MEMC chip is used.

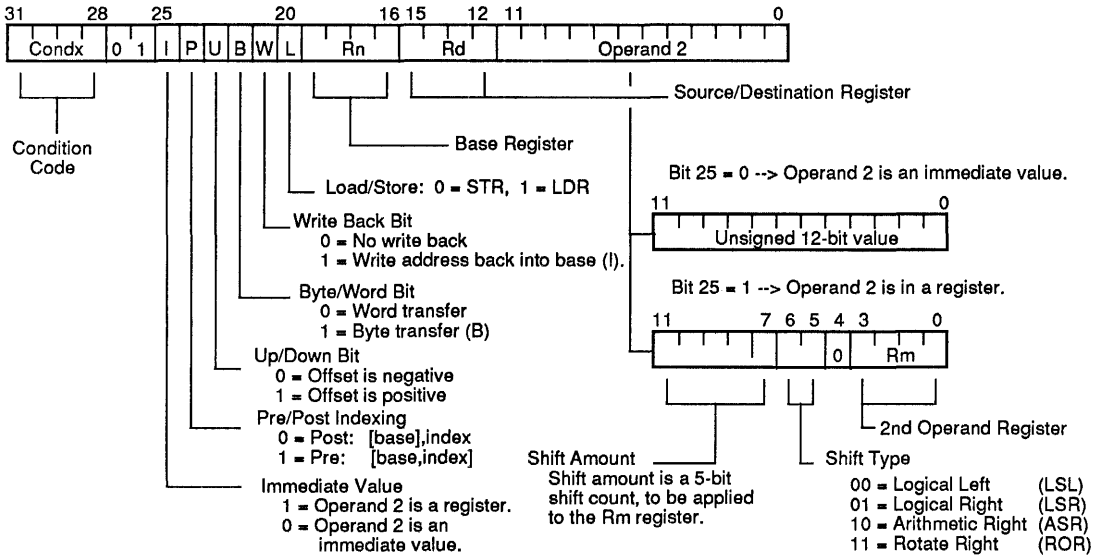
Shifted Register Offset - The eight shift control bits are described in the data processing instructions, but the register specified shift amounts are not implemented in this instruction class.

Bytes and Words - This instruction class may be used to transfer a byte (B=1) or a word (B=0) between a processor register and memory. In the discussion, remember that the VL86C010 stores words into memory with the Least Significant Byte at the lowest address (i.e., LSB first).

A byte load (LDRB) expects the data on bits D7 to D0 if the supplied address is on a word boundary, on bits D15 to D8 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 zeroes.

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

FIGURE 15. LOAD/STORE VALUE FROM MEMORY (LDR,STR)



Note: There is no Rs or shift for the LDR/STR class. That is, the shift amount cannot be contained in a register.

Non-Aligned Accesses - A word load (LDR) should generate a word aligned address. An address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits D7 to D0. External hardware could perform a double access to memory to allow non-aligned word loads, but the VL86C110 Memory Controller does not support this function.

Use of R15 - These instructions will never cause the PSR to be modified, even when Rd or Rn is R15.

If R15 is specified as the base register (Rn), the PC is used without the PSR flags. When using the PC as the base register one must remember that it contains an address eight bytes advanced from the address of the current instruction.

If R15 is specified as the register offset (Rm), the value presented will be the PC together with the PSR.

When R15 is the source register (Rd) of a register store (STR) instruction, the value stored will be the PC together with the PSR. The stored value of the PC will be 12 bytes advanced from the address of the instruction. A load register (LDR) with R15 as Rd will change only the PC, and the PSR will be unchanged.

Address Exceptions - If the address used for the transfer (i.e., the unmodified contents of the base register for post-indexed addressing, or the base modified by the offset for pre-indexed addressing) has a logic one in any of the bits D31 to D26, the transfer will not take place and the address exception trap will be taken.

Note that only the address actually used for the transfer is checked. A base containing an address outside the legal range may be used in a pre-indexed transfer if the offset brings the address within the legal range. Likewise, a base within the legal range may be modified by post-indexing to outside the legal range without causing an address exception.

Data Aborts - A transfer to or from a legal address may still present special cases 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 ABRT pin 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. The instruction can then be restarted and the original program continued.

Syntax:

LDR/STR{cond}{B}{T} Rd,<Address>

where *LDR* means Load from memory into a register.
STR means store from a register into memory.
cond is a two-character condition mnemonic (see Condition Code section).
B If present implies byte transfer, else a word transfer.
T If present, the W bit is set in a post-indexed instruction, causing the –TRAN pin to go low for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.
Rd is a valid register: R0-R15, SP, LK, or PC.
Address Can be any of the variations in the following table.

Address Variants:

Address expression: An expression evaluating to a relocatable address:
<expression> The assembler will attempt to generate an instruction using the PC as a base, and a corrected offset to the location given by the expression. This is a PC-relative pre-indexed address. If out of range (at assembly or link time), an error message will be given.

Pre-indexed address: Offset is added to base register before using as effective address, and offsets are placed within the [ ] pair. Rn may be viewed as a pointer:

[Rn, <expression>]{!} Signed offset of *expression* bytes is added to base pointer.
[Rn, Rm]{!} Add Rm to Rn before using Rn as an address pointer.
[Rn, Rm <shift> count]{!} Signed offset of *Rm* (modified by *shift*) is added to base pointer.

Post-indexed address: Offset is added to base reg, after using base reg for the effective address. Offsets are placed after the [ ] pair:

[Rn],<expression> Expression is added to Rn, after Rn's usage as a pointer.
[Rn], Rm Rm is added to Rn, after Rn's usage as an address pointer.
[Rn], Rm <shift> count Shift the offset in Rm by *count* bits, and add to Rn, after Rn's usage as an address pointer.
[Rn] No offset is added to base address pointer.

where *expression* A signed 13-bit expression (including the sign).
Rm, Rn Valid register names: R0-R15, SP, LK, or PC. If RN = PC, the assembler will subtract 8 from the expression to allow for processor address read-ahead.
shift Any of: LSL, LSR, ASR, ROR, or RRX.
count Amount to shift Rm by. It is a 5-bit constant, and may not be specified as an Rs register (as for some other instruction classes).
! If present, the ! sets the W-bit in the instruction, forcing the effective offset to be added to the Rn register, after completion.

Examples (Pre-Index and Optional Increment):

In each of these examples, the effective offset is added to the value in the Rn (base pointer) register prior to using that value as the effective address. Rn is then updated only if the ! suffix is supplied:

```
STR     R1, [R2, R1]!     ; *(R2+R1) = R1. Then R2 += R1.
STR     R3, [R2]          ; *(R2) = R3.
LDR     R1, [R0, 16]      ; R1 = *(R0 + 16). Don't update R0.
LDR     R9, [R5, R0 LSL 2] ; R9 = *(R5 + (R2<<2)). Don't update R5.
LDREQB  R2, [R5, 5]       ; if (Zflag) R2 = *(R5 + 5), a zero-filled byte load.
```

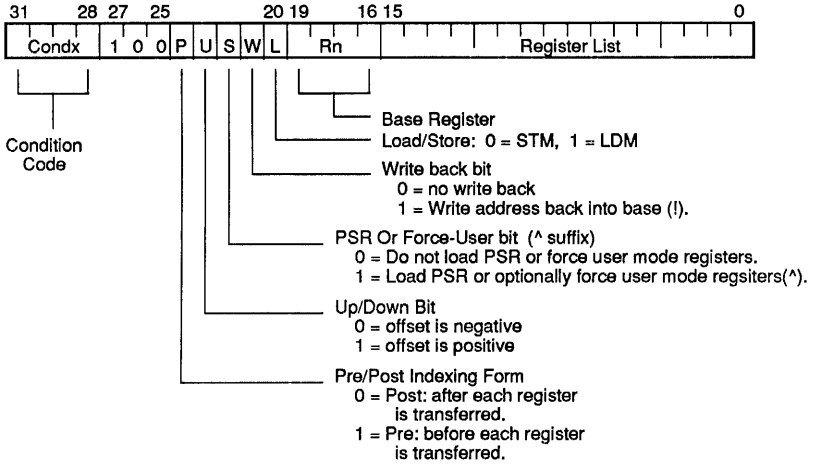
Examples (Post-Index and Increment):
In each of these examples, the effective offset is added to the Rn (base pointer) register after using the Rn register as the effective address. That is, Rn is then updated unconditionally, regardless of any ! suffix.

```
STR     R1, [R2], R1        ; *R2 = R1. Then R2 += R1.
STR     R3, [R2], R5        ; *(R2) = R3. Then R2 += R5.
LDR     R1, [R0], 16        ; R1 = *R0. Then R0 += 16.
LDR     R9, [R5], R0 ASR 3  ; R9 = *R5. Then R5 += (R0 / 8).
LDREQB  R2, [R5], 5         ; if (Zflag) R2 = *R5, a zero-filled byte load, and then R5 += 5.
```

Examples (Expression):
In these examples, the PLACE label is an internal or external PC-relative label, typically created as shown. PC-relative references are precompensated for the 8-byte read-ahead done by the processor. PARMx is a register-relative label, typically created via a DTYPE directive, and assumed to be relative to the LK (R14) register. DATAx is similar, but is presumably defined relative to the SP (R13) register, and GENERAL relative to R0. In any case, they may be located up to ±4096 bytes from the associated base register.

```
        LDR     R0, DATA1       ; SP-relative. Same as: LDR R0, [SP+DATA1].
        STR     R2, PLACE       ; PC-relative. Same as: STR R2, [PC+16].
        LDR     R1, PARM0       ; LK-relative. Same as: LDR R1, [LK+DATA1].
        STR     R1, GENERAL     ; R0-relative. Same as: STR R1, [R0+GENERAL].
        B       Across          ; Skip over the data temporary.
                                ;
PLACE   DW      0               ; Temporary storage area.
Across  • • •                   ; Resume execution.
```

FIGURE 16. LOAD/STORE REGISTER LIST FROM MEMORY (LDM,STM)



The multi-register transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes (push up/pop down, or push down/pop up). They are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

The 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 contained in a 16-bit field in the instruction, with each bit corresponding to a register. A logic one in bit zero of the register field will cause R0 to be transferred, a logic zero will cause it not to be transferred; similarly bit one controls the transfer of R1, and so on.

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. This is illustrated in Figures 17 and 18.

Transfer of R15 - Whenever R15 is stored to memory, the value transferred is the PC together with the PSR flags. The stored value of the PC will be 12 bytes advanced from the address of the STM instruction.

If R15 is in the transfer list of a load multiple (LDM) instruction, the PC is overwritten and the effect on the PSR is controlled by the S bit. If the S bit is zero the PSR is preserved unchanged, but if the S bit is set the PSR will be overwritten by the corresponding bits of the loaded value. In user mode, however, the I, F, M1, and M0 bits are protected from change, whatever the value of the S bit. The mode at the start of the instruction determines whether these bits are protected, and the supervisor may return to the user program, reenabling interrupts and restoring user mode with one LDM instruction.

Transfers to User Bank - For STM instructions the S bit is redundant as the PSR is always stored with the PC whenever R15 is in the transfer list. In user mode the S bit is ignored, but in other modes it has a second interpretation. S = 1 is used to force transfers to take values from the user register bank instead of from the current register bank. This is useful for saving the user state on process switches. Note that when it is so used, write back of the base will also be to the user bank, though the base will be fetched from the current bank. Therefore don't use write back when forcing user bank.

In LDM instructions the S bit is redundant if R15 is not in the transfer list, and again in user mode it is ignored. In non-user mode where R15 is not in the transfer list, S=1 is used to force loaded values into user registers instead of the current register bank. When used in this manner, care must be taken not to read from a banked register during the following cycle; if in doubt, insert a NO-OP. Again, don't use write back when forcing a user bank transfer.

R15 as the Base - When the base is the PC, the PSR bits will be used to form the address as well, so unless all interrupts are enabled and all flags are zero an address exception will occur. Also, write back is never allowed when the base is the PC (setting the W bit will have no effect).

Base Within the Register List - When write back is specified, the base is written back at the end of the second cycle of the instruction. During an 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. An LDM will always overwrite the updated base if the base is in the list.

Address Exceptions - When the address of the first transfer falls outside the legal address space (i.e., has a logic one somewhere in bits 31 to 26), an address exception trap will be taken. The instruction will first complete in the usual number of cycles, though an STM will be prevented from writing to memory. The processor state will be the same as if a data abort had occurred on the first transfer cycle.

Only the address of the first transfer is checked in this way; if subsequent addresses over or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

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 ABRT pin high. This can happen on any transfer during a multiple register load or store, and must be recoverable if the processor is to be used in a virtual memory system.

Abort During an STM - If the abort occurs during a store multiple instruction, the processor 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.

To illustrate the various load/store modes, consider the transfer of R1, R5 and R7 in the case where Rn = 1000H and write back of the modified base is required (W = 1). These 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 1000H unless it was also in the transfer list of the load multiple register instruction. Then it would have been overwritten with the loaded value.

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

Mode Bits - During the execution of LDMs and STMs, the two LSBs of the instruction will contain the (noninverted) mode status bits. These may be used by external hardware to force memory accesses from an alternative bank.

The following figures illustrate the impact of various addressing modes. R1, R5, and R7 are moved to/from memory, where Rn=0x1000, and a write back of the modified base is done (W=1). The figures show the sequence of incrementing "pushes", the addresses used, and the final value of Rn. Without write back, Rn would remain at 0x1000.

Figure 17 illustrates the use of incrementing stack "pushes".

Figure 18 illustrates decrementing "pushes" to the stack based upon Rn.

FIGURE 17. INCREMENTING INDEX

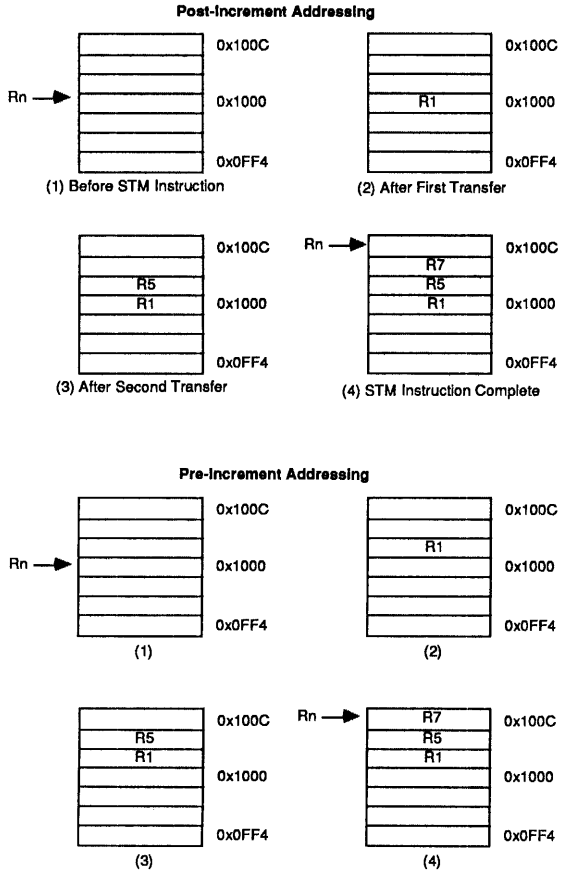
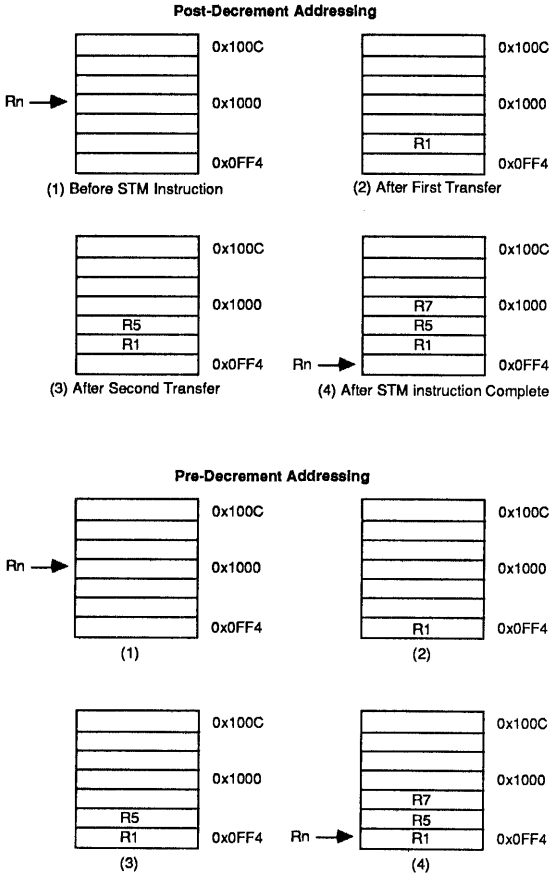


FIGURE 18. DECREMENTING INDEX



Overwriting of registers stops when the abort happens. The aborting load will not take place, nor will the preceding one, but registers two or more positions ahead of the abort (if any) will be loaded. (This guarantees that the PC will be preserved, since it is always the last register to be overwritten.)

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.

Syntax:

LDM|STM{cond}<mode> Rn{!}, <Rlist>{^}

where
- *cond* Is an optional 2-letter condition code common to all instructions.
- *mode* Is any of: IA, IB, DA, or DB.
- *Rn* Is a valid register name: R0-R15, SP, LK, or PC.
- *Rlist* Can be a single register (as described above for Rn), or may be a list of registers, enclosed in { } (eg {R0,R2,R7-R10,LK}).
- *!* If present, requests write back (W=1). Otherwise W=0.
- ^ If present, set S bit to load the PSR with the PC, or force transfer of user bank, when in non-user 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. The names and instruction bit values are:

Function	Mnemonic	L Bit	P Bit	U bit	Operation
Pre-increment load	LDMIB	1	1	1	Pop upwards
Post-increment load	LDMIA	1	0	1	Pop upwards
Pre-decrement load	LDMDB	1	1	0	Pop downwards
Post-decrement load	LDMDA	1	0	0	Pop downwards
Pre-increment store	STMIB	0	1	1	Push upwards
Post-increment store	STMIA	0	0	1	Push upwards
Pre-decrement store	STMDB	0	1	0	Push downwards
Post-decrement store	STMDA	0	0	0	Push downwards

IA, IB, DA, DB allow control of when the memory printer is changed and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

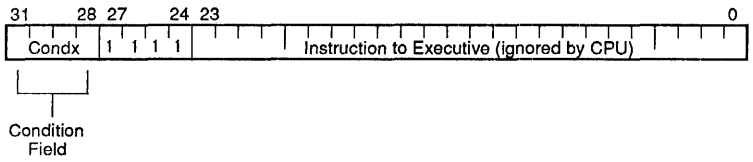
Examples

```
LDMIA   SP!, {R0, R1, R2}      ; unstack 3 registers
STMIA   R2, {R0-R15}           ; save all registers
```

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

```
STMDB   SP!, {R0-R3, LK}       ; Save R0 to R3 for workspace,and R14 for returning.
BL      Subroutine             ; This call will overwrite R14
LDMIA   SP!, {R0-R3, PC}^      ; Restore workspace and return, restoring PSR flags.
```

FIGURE 19. SOFTWARE INTERRUPT (SWI)



Note: The machine comments field in bits 23 - 0 are ignored by the hardware. They are made available for free interpretation by the software executive, and may be found in LSB-first byte order on the stack.

The Software Interrupt (SWI) 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, with execution resuming at 0x08. 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 and PSR are saved in R14_svc upon executing the software interrupt trap with the PC adjusted to point to the word after the SWI instruction. MOVS R15, R14_svc will return to the user program, restore the user PSR and return the processor to user mode.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address.

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

Syntax:

SWI{cond} <expression>

where *cond* Is the two-character condition code common to all instructions.

expression Is a 24-bit field of any format. The processor itself ignores it, but the typical scenario is for the software executive to specify patterns in it, which will be interpreted in a particular way by the executive, as commands.

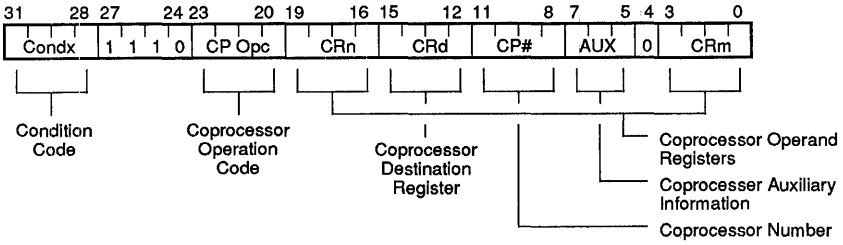
Examples:

```
        acons   Zero=0, ReadC=1, Write1=2       ; Assembler constants.

        SWI     ReadC           ; Get next character from read stream
        SWI     Write1+"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:

```
; Assume that the R13_svc (the supervisor's R13) points to a suitable stack.
        acons   Zero=0, ReadC=1, Write1=2       ; Assembler constants.
        acons   CC_Mask = 0xFC00003             ; Non-address area mask.
08h     B       Super                           ; SWI entry point
        ..
Super   STMOB   SP!,{r0,r1, r2, R14)            ; Save working registers.
        BIC     r1, r14, CC_Mask                ; Strip condx codes from SWI instruction address.
        LDR     R0, [R1, -4]                    ; Get copy of SWI instruction.
        BIC     R0, R0, 0xFF000000              ; Get lower 24 bits of SWI, only.
        MOV     R1, SWI_Table                   ; Get absolute address of PC-relative table.
        LDR     PC, [R1, R0 LSL 2]              ; Jump indirect on the table.
SWI_Table dw    Zero_Action                     ; Address of service routines.
        dw      ReadC_Action
        dw      Write1_Action
Write1_Action                                   ; Typical service routine.
        ..
        LDMIA   R13!,{R0-R2, PC}^               ; Restore workspace, and return to inst after SWI.
```

FIGURE 20. COPROCESSOR DATA OPERATIONS (CDO)



The instruction is executed only if the condition code field is true. The field is described in the Condition Codes section.

This is actually a class of instructions, rather than a single instruction, and is equivalent to the ALU class on the processor. All instructions in this class are used to direct the coprocessor to perform some internal operation. No result is sent back to the CPU, and the VL86C010 will not wait for the operation to complete. The coprocessor could maintain a queue of such instructions awaiting execution. Their execution may then overlap other VL86C010 activity, allowing the two processors to perform independent tasks in parallel.

Coprocessor Fields - Only bit 4 and bits 31-24 are significant to the VL86C010; the remaining bits are used by coprocessors. The above field names are used by convention, but particular coprocessors may redefine the use of any or all fields as appropriate, except for the CP#. For the sake of future family product introductions, it is encouraged that the above conventions be followed, unless absolutely necessary.

By convention, 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, placing the result into CRd.

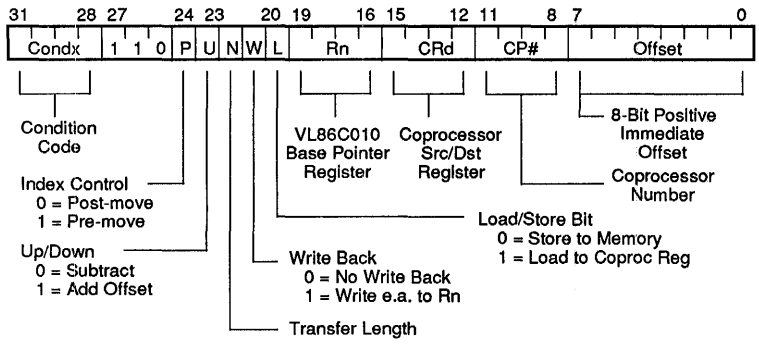
Syntax:

	CDO{cond}	cp#,<expression1>, CRd, CRn, CRm{,<expression2>}
where	*cond*	Is the conditional execution code, common to all instructions.
	cp#	Is the (unique) coprocessor number, assigned by hardware.
	CRd, CRn, CRm	These are valid coprocessor registers: CR0-CR15.
	expression1	Evaluates to a constant, and is placed in the *CP Opc* field.
	expression2	(Where present) evaluates to a constant, and is placed in the *AUX* field.

Examples:

```
CDO    1, 10, CR1, CR7, CR2     ; Request coproc #1 to do operation 10 on CR7 and CR2, putting result into CR1.
CDOEQ 2, 5, CR1, cr2, Cr3, 2    ; If the Z flag is set, request coproc #2 to do
                                ; operation 5 (type 2) on CR2 and CR3, placing the result into CR1.
```

FIGURE 21. COPROCESSOR LOAD/STORE DATA (LDC/STC)



The LDC and STC instructions are used to load or store single bytes or words of data. They differ from MCR and MRC instructions in that they move data between coprocessor registers and a specified memory address. In contrast, the other instructions move data between registers, or move a constant (contained in the instruction) into a register.

The memory address used in LDC/STC transfers is calculated by adding an offset to or subtracting an offset from a base pointer register, Rn. Typically, a load of a labeled memory location involves the loading via a (signed) offset from the current PC. Regardless of the base register used, the result of the offset calculation may be written back into the base register if 'auto-indexing' is required.

Coprocessor Fields - The CP# field identifies which coprocessor shall supply or receive the data. A coprocessor will respond only if its number matches the contents of this field.

The CRd field and N bit contain information which may be interpreted in different ways by different coprocessors. By convention, however, CRd is the register to be transferred (or the first register, where more than one is to be transferred). 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 registers for context switching.

Offsets and Indexing - The VL86C010 is responsible for providing the address used by the memory system for the transfer, and the modes available are similar to those used for the processor's LDR/STR instructions.

Only 8-bit offsets are permitted, and the VL86C010 automatically scales them by two bits to form a word offset to the pointer in the Rn register. Of itself, the offset is an 8-bit unsigned value, but a 9-bit signed negative offset may be supplied. The assembler will complement it to an 8-bit (positive) value and will clear the instruction's U bit, forcing a compensating subtract. The result is a ±256 word (1024 byte) offset from Rn. Again, the VL86C010 internally shifts the offset left two bits before addition to the Rn register.

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 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, 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.

For an offset of +1, the value of the Rn base pointer register (modified, in the pre-indexed case) is used for the first word transferred. Should the instruction be repeated, the second word will go from/to an address one word (4 bytes) higher than pointed to by the original Rn, and so on.

Use of R15 - If R15 is specified as the base register (Rn), the PC is used without the PSR flags. When using the PC as the base register note that it contains an address eight bytes advanced from the address of the current instruction. As with the LDR/STR case, the assembler performs this compensation automatically.

Hardware Address Translation - The W bit may be used in non-user mode programs (when post-indexed addressing is used) to force the -TRANS pin low for the transfer cycle. This allows the operating system to generate user addresses when a suitable memory management system is present.

Address Exceptions - If the address used for the first transfer is illegal, the address exception mechanism will be invoked. Instructions which transfer multiple words will only trap if the first address is illegal; subsequent address will wrap around inside the 26-bit address space.

Note that only the address actually used for the transfer is checked. A base containing an address outside the legal range may be used in a pre-indexed transfer if the offset brings the address within the legal range. Likewise, a base within the legal range may be modified by post-indexing to outside the legal range without causing an address exception.

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 data will be preserved. The coprocessor is partly responsible for ensuring restartability. It must either detect the abort, or ensure that any actions consequent from this instruction can be repeated when the instruction is retried after the resolution of the abort.

Syntax:

<LDC/STC>{cond}{L}{T}{N} cp#, CRd, <Address>{!}

where	*LDC*	means Load from memory into a coprocessor register.
	STC	means store a coprocessor register to memory.
	cond	is a two-character condition mnemonic (see Condition Code section).
	L	If present implies long transfer (N=1), else a short transfer (N=0).
	T	If present, the W bit is set in a post-indexed instruction, causing the –TRAN pin to go low for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.
		N sets the value of bit 22 of instruction.
	cp#	Valid coprocessor number, determined by hardware.
	CRd	Valid coprocessor register number: CR0-CR15.
	Address	Can be any of the variations in the following table.

Address Variants:

Address expression: An expression evaluating to a relocatable address:

<expression> The assembler will attempt to generate an instruction using the PC as a base, and a corrected offset to the location given by the 9-bit expression. This is a PC-relative pre-indexed address. If out of range (at assembly or link time), an error message will be given.

Pre-indexed address: Offset is added to base register before using as effective address, and offsets are placed within the [ ] pair. Rn may be viewed as a pointer:

[Rn]{!}	No offset is added to base address pointer.
[Rn, <expression>]	Signed offset of expression (bytes) is added to base pointer.
[Rn, <expression>]{!}	Signed offset of expression (bytes) is added to base pointer. Then this effective address is written back to Rn.

Post-indexed address: Offset is added to base reg, after using base reg for the effective address. Offsets are placed after the [ ] pair:

[Rn],<expression> Expression is added to Rn, after Rn's usage as a pointer.

where	expression	A signed 9-bit expression (including the sign).
	Rn	*Valid register names: R0-R15, SP, LK, or PC. If Rn* = PC, the assembler will subtract 8 from the expression to allow for processor address read ahead.

Examples (Pre-Index):

In each of these examples, the effective offset is added to the Rn (base pointer) register prior to using the Rn register as the effective address. Rn is then updated only if the ! suffix is supplied. Coprocessor #1 is used in all cases, for simplicity.

```
        STC     1,CR3, [R2]         ; *(R2) = CR3.
        LDC     1,CR1, [R0, 16]     ; CR1 = *(R0 + 16). Don't update R0.
        LDCEQ   1,CR2, [R5, 12]!    ; if (Zflag) CR2 = *(R5 + 12). Then, R5 += 12.
```

Examples (Post-Index):

In each of these examples, the effective offset is added to the Rn (base pointer) register after using the Rn register as the effective address. Rn is then updated unconditionally. Coprocessor #3 is used in all cases, for simplicity.

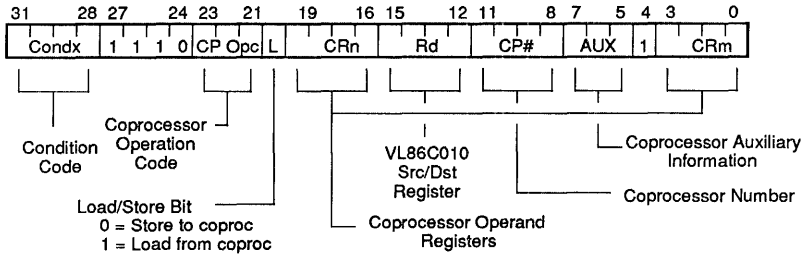
```
        STC     3, CR1, [R2], 8     ; *R2 = CR1. Then R2 += 8.
        LDC     3, CR1, [R0], 16    ; CR1 = *R0. Then R0 += 16.
        LDCEQL  3, CR2, [R5], 4     ; if (Zflag) CR2 = *R5, and then (implicitly), R5 += 4.
                                    ; Use the long option (probably to store multiple words).
```

Examples (Expression):

In these examples, the PLACE label is an internal or external PC-relative label, typically created as shown. PC-relative references are precompensated for the 8-byte read-ahead done by the processor. It may be located up to ±1024 bytes from the associated base register, and must be a multiple of 4 bytes in offset.

```
        STC     3, CR5, PLACE       ; PC-relative. Same as: STC 3, CR5, [PC+8].
        B       Across              ; Skip over the data temporary.
;
PLACE   DW      0                   ; Temporary storage area.
Across  • • •                       ; Resume execution.
```

FIGURE 22. COPROCESSOR REGISTER TRANSFER (MCR,MRC)



The instruction is executed only if the condition code field is true. The field is described in the Condition Codes section.

This is actually a class of instructions, rather than a single instruction, and is equivalent to the ALU class on the processor. Instructions in this class are used to direct the coprocessor to perform some operation between a processor register and a coprocessor register. It differs from the CPD instruction in that the CPD performs operations on the coprocessor's internal registers only.

An example of an MCR usage would be a FIX of a floating point value held in the coprocessor, where the number is converted to a 32-bit integer within the coprocessor, and the result then transferred back to an ARM register. An example of an MRC usage would be the converse: A FLOAT of a 32-bit value in a VL86C010 register into a floating point value within a coprocessor register.

An intended use of this instruction is to communicate control information directly between the coprocessor and the processor PSR flags. As an example, the result of a comparison of two floating point values within the coprocessor can be moved to the PSR to control subsequent execution flow.

Coprocessor Fields - The CP# field is used by all coprocessor instructions to specify which coprocessor is being invoked.

The CP Opc, CRn, CP, and CRm fields are used only by the coprocessor, and the interpretation of these fields is set only by convention; other incompatible interpretations are allowed. The conventional interpretation is that the CP Opc and CP fields specify the operation for the coprocessor to perform, CRn is the coprocessor register used as source or destination of the transferred information, and CRm is the second coprocessor register which may be involved in some way dependent upon the operation code.

Transfers To/From R15: When a coprocessor register transfer to VL86C010 has R15 as the destination, bits 31-28 of the transferred word are copied into the N, Z, C, and V flags, respectively. The other bits of the transferred word are ignored; the PC and other PSR flags are unaffected by the transfer.

A coprocessor register transfer from VL86C010 with R15 as the source register will save the PC together with the PSR flags.

Syntax:

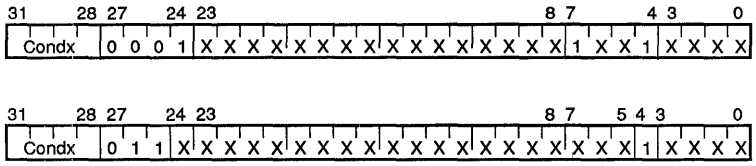
MCR/MRC{cond} CP#,<expression1>, Rd, CRn, CRm{,<expression2>}

where	*cond*	Is the conditional execution code, common to all instructions.
	CP#	Is the (unique) coprocessor number, assigned by hardware.
	Rd	Is the ARM source or destination register.
	CRn, CRm	These are valid coprocessor registers: CR0-CR15.
	expression1	Evaluates to a constant, and is placed in the CP Opc field.
	expression2	(Where present) evaluates to a constant, and is placed in the AUX field.

Examples:

```
MCR    1, 6, R1, CR7, CR2      ; Request co-proc #1 to do operation 6 on
                               ; CR7 and CR2, putting result into ARM's R1.

MRCEQ 2, 5, R1, cr2, Cr3, 2    ; If the Z flag is set, transfer the ARM's R1 reg to the co-proc register (defined
                               ; by hardware), and request co-proc #2 to do oper 5 (type 2) on CR2 and CR3.
```

FIGURE 23. UNDEFINED (RESERVED) INSTRUCTIONS



Note: The above instructions will be presented for execution only if the condition field is true.

2

If the condition is true, the Undefined Instruction trap will be taken.

Note that the undefined instruction mechanism involves offering these instructions to any coprocessors which may be present, and all coprocessors must refuse to accept them by taking CPA high.

Assembler Syntax - At present the assembler has no mnemonics for generating these instructions. If they are adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, these instructions should not be used.

INSTRUCTION SET EXAMPLES
The following examples show ways in which the basic processor instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some), mostly they just save code.

Using Conditional Instructions -

(1) Using conditionals for logical OR, this sequence:

```
        CMP     R1, p           ; If R1=p or R2=q then goto Label
        BEQ     Label
        CMP     R2, q
        BEQ     Label
```

can be replaced by

```
        CMP     R1, p
        CMPNE   Rm, q           ; If condition not satisfied try other test
        BEQ     Label
```

(2) Absolute value

```
        TEQ     R1, 0           ; Test sign
        RSBMI   R1, R1, 0       ; and 2's complement if necessary
```

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

```
        MOV     R2, R0 LSL 2    ; Multiply by 4
        CMP     R1, 5           ; Test value
        ADDCS   R2, R2, R0      ; Complete multiply by 5
        ADDHI   R2, R2, R0      ; Complete multiply by 6
```

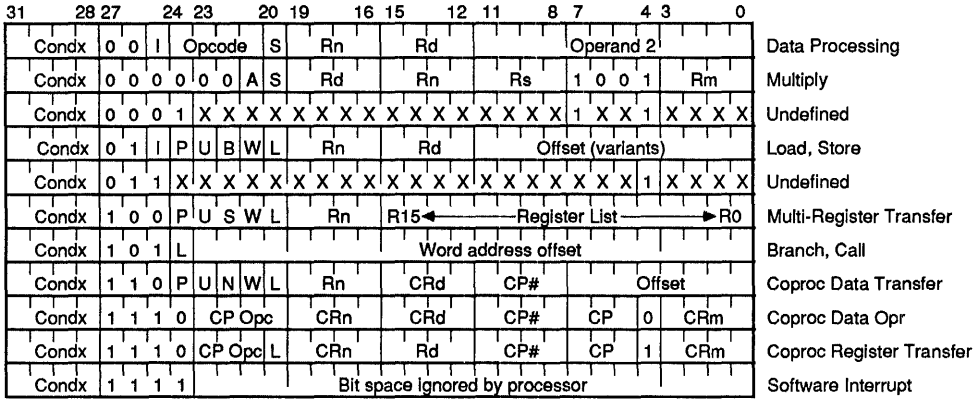
(4) Combining discrete and range tests

```
        TEQ     R2, 127         ; If (R2<>127)
        CMPNE   R2, " "-1       ; Range test and if (R2<' ')
        MOVLS   R2, "."         ; Then, R2 ="."
```

Division and Remainder

```
; Enter with numbers in R0 and R1
        MOV     R4, 1                   ; Bit to control the division
Div1    CMP     R1, 0x80000000          ; Move R1 until greater than R0
        CMPCC   R1, R0
        MOVCC   R1, R1 LSL 1
        BCC     Div1
        MOV     R2, 0
Div2    CMP     R0, R1                  ; Test for possible subtraction
        SUBCS   R0, R0, R1              ; Subtract if ok
        ADDCS   R2, R2, R4              ; Put relevant bit into result
        MOVS    R2, R4 LSR 1            ; Shift control bit
        MOVNE   R1, R1 LSR 1            ; Halve unless finished
        BNE     Div2
; Division result is in R2.
; Remainder is in R0.
```

FIGURE 24. INSTRUCTION SET SUMMARY



Pseudo Random Binary Sequence Generator - It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift register-based generators with exclusive or feedback rather 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). The basic algorithm is Newbit = bit_33 xor bit_20, shift left the 33-bit number and put in Newbit at the bottom. Then do this for all the Newbits needed i.e. 32 of them. Luckily, this can all be done in 5S cycles:

```
; Enter with seed in R0 (32 bits), R1 (1 bit in R1 lsb)
; Uses R2
        TST     R1, R1 LSR 1            ; Top bit into carry
        MOVS    R2, R0 RRX              ; 33 bit rotate right
        ADC     R1, R1, R1              ; Carry into lsb of R1
        EOR     R2, R2, R0 LSL 12       ; (Involved!)
        EOR     R0, R2, R2 LSR 20       ; (Whew!)
; New seed in R0, R1 as before
```

Multiplication by Constant:

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

```
        MOV     R0, R0 LSL n
```

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

```
        ADD     R0, R0, R0 LSL n
```

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

```
        RSB     R0, R0, R0 LSL n
```

(4) Multiplication by 6

```
        ADD     R0, R0, R0 LSL 1        ; Multiply by 3
        ADD     R0, R0 LSL 1            ; and then by 2
```

(5) Multiply by 10 and add in extra number

```
        ADD     R0, R0, R0 LSL 2        ; Multiply by 5
        MOV     R0, R2, R0 LSL 1        ; Multiply by 2 and add in next digit
```

(6) General recursive method for R1 =R0*C,C a constant:

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

```
D=1:    MOV     R1, R0 LSL n
D<>1:   (R1 =R0*D)
        MOV R1, R1 LSL n
```

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

```
D=1:    ADD     R1, R0, R0 LSL n
D<>1:   (R1 = R0*D)
        ADD     R1, R0, R1 LSL n
```

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

```
D=1:    RSB     R1, R0, R0 LSL n
D<>1:   (R1 =R0*D)
        RSB     R1, R0, R1 LSL n
```

This is not quite optimal, but close. An example of its non-optimality is multiply by 45 which is done by:

```
        RSB     R1, R0, R0 LSL 2        ; Multiply by 3
        RSB     R1, R0, R1 LSL 2        ; Multiply by 4*3-1 = 11
        ADD     R1, R0, R1 LSL 2        ; Multiply by $*11+1 = 45
```

rather than by:

```
        ADD     R1, R0, R0 LSL 3        ; Multiply by 9
        ADD     R1, R1, R1 LSL 2        ; Multiply by 5*9 = 45
```

Loading a Word with Unknown Alignment:

```
; Enter with address in R0 (32 bits)
; Uses R1, R2; result in R2.
; Note R2 must be less than R3, e.g. 2, 3
        BIC     R1, R0, 3           ; Get word aligned address.
        LDMIA   R1, {R2,R3}         ; Get 64 bits containing answer.
        AND     R1, R0, 3           ; Correction factor in bytes, not in bits.
        MOVS    R1, R1 LSL 3        ; Test if aligned.
        MOVNE   R2, R2, LSR R1      ; Product bottom of result word (if not aligned).
        RSBNE   R1, R1, 32          ; Get other shift amount.
        ORRNE   R2, R2, R3 LSL R1   ; Combine two halves to get result.
```

Sign Extension of Partial Word

```
        MOV     R0, R0 LSL 16       ; Move to top
        MOV     R0, R0, LSR 16      ; ... and back to bottom
                                    ; (Use ASR to get sign extended version).
```

Return, Setting Condition Codes

```
        BICS    PC, R14,CFLAG       ; Returns, clearing C flag ROM link register.
        ORRCCS  PC, R14, CFLAG      ; Conditionally returns, setting C flag.
```

```
; Above code should not be used except in User mode, since it will reset the interrupt enable flags to
; their value when R14 was set up. This generally applies to non-user mode programming.
; e.g., MOVS PC,R14        MOV PC,R14   is safer!
```

MACHINE CODE INSTRUCTIONS
This chapter describes machine code instructions that are unique to the VL86C010 processor. Each symbolic instruction line is translated into exactly one 32-bit memory word, each aligned on a machine-word boundary.

Appendix A.1 Condition Codes
All instructions executed by the VL86C010 contain a 4-bit field that permits them to be executed only if certain conditions are true. If the specified conditions are not true, the instruction is skipped over. Even an illegally-formatted instruction will be properly skipped over if its condition code field is properly set.

Program Status Register (PSR) - Instructions that use the processor's arithmetic/logic unit set one or more of four status bits. (These are not the same as the 4-bit condition code field of an instruction.) The "program status register" is not really a register of itself, but is a series of bits within the R15 (PC) register. The bits are set to indicate Carry, Zero, oVerflow, or Negative, as follows:

Bit 31	N	is set if bit 31 is set in the result, indicating a negative result.
Bit 30	Z	is set if the result of the operation is zero, all bits reset. Certain instructions (as CMP and TST) do not actually store the result to a destination register, but do alter the status.
Bit 29	C	is set if there was a carry out of bit 31. Logic-only instructions do not use (and cannot generate) this carry. Usually, these set C to zero, or do not alter it.
Bit 28	V	is set if the signs of both operands were identical, but the sign of the result differs from them. It indicates that the 32-bit result register was too short to hold the result.
Bit 27	I	set to 1 to disable the IRQ interrupts.
Bit 26	F	set to 1 to disable the FIRQ interrupts.

Other status bits exist in the PSR, indicating the current interrupt-enable state (bits 27 and 26) and the current processor execution state (bits 1 and 0. The latter may include user, interrupt service, or supervisor modes. These modes are encoded into bits 0 and 1 of the PC, and are discussed page 2-26.

Instruction Condition Code Field - The instruction condition code field bits specify how certain combinations of PSR bits are to be interpreted. Rather than require the programmer to specify the needed combination of status bits, the most useful combinations are encoded into the condition code field. This allows most-often used tests to be performed in one instruction rather than two.

Bits 31 to 28 are encoded according to the table below. Each encoding is represented by a 2-letter suffix that is appended to the base instruction mnemonic. If no such suffix is given, the always encoding is assumed.

Field	Code	Purpose
0000	EQ	Z set (operands are equal)
0001	NE	Z clear (operands are unequal)
0010	CS	C set (1st operand higher or same as 2nd, unsigned compare)
0011	CC	C clear (1st operand lower than 2nd, unsigned compare)
0100	MI	N set (result is negative)
0101	PL	N clear (result is positive or zero)
0110	VS	V set (overflow occurred)
0111	VC	V clear (no overflow occurred)
1000	HI	C set and Z clear (1st operand higher than 2nd, unsigned compare)
1001	LS	C clear or Z set (1st operand lower or same as 2nd, unsigned compare)
1010	GE	N set and V set, or N clear and V set (1st operand greater or equal to 2nd, signed)
1011	LT	N set and V clear, or N clear and V set (1st operand less than 2nd, signed)
1100	GT	Z clear, and either N set and V set, or N clear and V set (1st operand greater than 2nd, signed)
1101	LE	Z set, or N set and V clear, or N clear and V set (1st operand less than or equal to 2nd one, signed).
1110	AL	Always (unconditional execute the instruction)
1111	NV	Never (never execute the instruction)

Appendix A.2 Miscellaneous

Machine-specific information not included elsewhere in this manual is noted in this section.

Word Alignment - All machine-code instructions for the VL86C010 are word-aligned. That is, they must be encoded into memory beginning on a 4-byte boundary. CASM permits most directives to align data which they might create on arbitrary (byte) boundaries. When a machine instruction (opcode mnemonic) is found, CASM forces it to begin on the next higher word boundary. The CLINK linker is also directed to keep it on such a boundary.

Default Condition Code - If no conditional-execution suffix code is appended to an opcode mnemonic, the AL (Always) case is assumed. Illegal (reserved) instruction patterns may be safely encountered by the program counter if their condition code is set to NV (Never); they will never reach the instruction decoder.

Large Immediate Operands - Certain classes of instructions permit the use of immediate constants, that is, constants that are to be loaded and that are specified as a part of the 32-bit machine instruction. For some, an 8-bit constant field is provided, but they permit selected values that are greater than 255. In these cases, any constant may be used that contains a pattern of 1s that span more than 8 contiguous bits.

Regardless where the bits may lie in the 32-bit word to be loaded, they may be shifted or rotated by CASM to store them in the 8-bit constant field of the instruction. CASM then computes the type and number of shifts required to recreate the desired constant. The condition where some bits are located at each end is permitted, as the value can be rotated to place all 1-bits in the least significant part of the instruction.

This type of constant is noted in the instructions that permit them.

Appendix A.3 Reserved (Undefined) Instructions

Several instructions are undefined. They are not currently implemented in the hardware and are reserved for future versions of the processor. The two instructions are:

Bits 27-24 == 0001, and bits 7 and 4 are 1. All other bits are don't-care conditions.

Bits 27-25 == 011, and bit 4 is 1. All other bits are don't-care conditions.

In both cases, the condition field is validly decoded; if bits 31-28 are set to 1111, the instruction will be ignored by the instruction fetch logic.

Appendix A.4 Shifts

For arithmetic-logic instructions where shifts of an operand are permitted, the shift forms in this section are permitted. The source of the shift count may be a 5-bit constant, or may be given in a register, as specified for the individual instruction types.

Appendix A.5 ADC - Arithmetic Add with Carry

ADC adds two 32-bit 2's complement operands, placing the result into a register. A value of +1 is added to the sum if the carry bit was set prior to the instruction; nothing is added to the sum if the carry was previously clear.

The normal use for Add-with-Carry is to compute sums of numbers that are greater than 32 bits in length. The multi-precision add sequence is to ADD the lowest words together (without carry compensation), possibly generating a carry in the process. The next most significant word pair is then added together with ADC, with the carry from the first pair added to the sum. If even more precision is used than two words per operand, they are successively ADC'd together until the most significant word pair has been added. (The same process is used for multi-precision subtracts, but using SUB and SBC.)

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

Intended Usage: Add the upper portions of a multi-word operand pair together.

Operational Function: Rd = Rn + Op2 + Carry

Flags Effected: N, Z, C, V

Syntax: ADC{*condition*}{S} *Rd, Rn, Op2*

where *condition* is an optional 2-character condition code. See the Condition Code section.

S (if present) sets condition codes based on the result.

Rd, Rn are any valid register names, such as R0-R15, PC, SP, or LK.

Op2 is second operand, and may have any of the following forms:

Rm shift Rs

Rm shift expression1

Rm RRX

expression2

Rm is any valid register names, as per *Rd* or *Rn* above, the operand value.

Rs is a register, per *Rd* above, containing a shift count in range of 1..32.

shift is any of: *ASL, LSL, LSR, ASR,* or *ROR*

expression1 is any positive absolute shift count in the range of 1..31.

expression2 is any signed expression shiftable into an 8-bit value.

Examples: Add two 64-bit operands. R0, R1 contain one operand and is to contain the result. R2, R3 contains the other operand.

```
ADD   R1, R3    ; Add LSBs together.
ADC   R0, R2    ; Add R0 = R0 + R2 + Carry (if any).
```

Variations: If a negative constant is specified as the 2nd operand, the 1's complement of it is used, and a SBC is substituted for the *ADC*. This effectively extends the range to 9 bits (including sign), and provides for sign extension to a full 32 bits.

Appendix A.6 ADD - Arithmetic Add
Perform a 32-bit addition of two 2's complement signed numbers. The state of the Carry bit before the addition is ignored, and the result is placed into a designated register. A carry-out from bit 31 will set the Carry flag. If the sum of two numbers of like signs should result in a change of sign in the result, the Overflow (V) bit is set; a carry may or may not occur simultaneously.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated with 2-bit shifts to produce an 8-bit constant.

Intended Usage: Add two operands together, or add together the lower words of a multi-word operand pair.

Operational Function: Rd = Rn + Op2

Flags Effected: N, Z, C, V

Syntax: ADD{*condition*}{S} *Rd, Rn, Op2*

where *condition* is an optional 2-character condition code. See the Condition Code section.
S (if present) sets condition codes based on the result.
Rd, Rn are any valid register names, such as R0-R15, PC, SP, or LK.
Op2 is second operand, and may have any of the following forms:
Rm shift Rs
Rm shift expression1
Rm RRX
expression2
Rm is any valid register names, as per *Rd* or *Rn* above, the operand value.
Rs is a register, per *Rd* above, containing a shift count in range of 1..32.
shift is any of: *ASL, LSL, LSR, ASR,* or *ROR*
expression1 is any positive absolute shift count in the range of 1..31.
expression2 is any signed expression shiftable into an 8-bit value.

Examples:

```
ADD   R0,R0,R2 ASR 2      ; R0 = R0 + (R2/4)
ADD   R5,R4,0x8000        ; R5 = R4 + 32768
```

Variations: If a negative constant is specified as the 2nd operand, the 2's complement of it is used, and a SUB is substituted for the *ADD*. This effectively extends the range to 9 bits (including sign), and provides for sign extension to a full 32 bits.

Appendix A.7 AND - Logical AND
The logical AND operation is performed on two operand words, and the 32-bit result is written to the destination register. For each bit position in the two operands, a test is made to determine that they are both set (1). If so, the same bit position in the destination register is turned on. It is otherwise turned off. The same operation is performed for each of the 32-bit positions.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated with 2-bit shifts to produce an 8-bit constant.

As with all logical operations, no carries are involved between bits in the same register, whether the source or the destination. However, the Carry status flag is set if bit 31 is set in both of the source operand registers.

Intended Usage: Mask selected portions of an operand value to preserve only those bits specified by the second operand. Also, perform the logical AND operation on two words. (To clear only those bits specified by the second operand, use the *BIC* instruction.)

Operational Function: Rd = Rn AND Op2

Flags Effected: N, Z, C

Syntax: AND{*condition*}{S} *Rd, Rn, Op2*

where *condition* is an optional 2-character condition code. See the Condition Code section.

S (if present) sets condition codes based on the result.

Rd, Rn are any valid register names, such as R0-R15, PC, SP, or LK.

Op2 is second operand, and may have any of the following forms:

Rm shift Rs
Rm shift expression1
Rm RRX
expression2

Rm is any valid register names, as per *Rd* or *Rn* above, the operand value.

Rs is a register, per *Rd* above, containing a shift count in range of 1..32.

shift is any of: *ASL, LSL, LSR, ASR,* or *ROR*

expression1 is any positive absolute shift count in the range of 1..31.

expression2 is any signed expression shiftable into an 8-bit value.

Examples:

```
AND   R9, R9, 0xFFFFFF00      ; Same as BIC R9,R9,0xFF.
AND   R2, R1 LSL 2            ; Mask via another register.
```

Variations: If a negative constant is specified as the 2nd operand, the 1's complement of it is used, and a BIC is substituted for the *AND*. This effectively extends the range to 9 bits (including sign), and provides for sign extension to a full 32 bits.

Appendix A.8 B - Branch

Force the program to branch to a new (word-aligned) address. The Program Counter (PC) is kept in the R15 register. The B instruction forces the value in R15 to be the sum of its current value and the instruction's operand field. That addition makes this a PC-relative branch, not an absolute branch. The 24-bit operand field permits a branch to any word address within the processor's address space.

The idea of a "relative" branch is that the processor can jump to any desired offset from its current position, the implication being that code containing the instruction may be moved around at will. This way, there is no need to compensate the operand field of the instruction for the address change if the code is moved around.

Most code using the branch instruction contains other material that may prevent it from being truly position independent, however. For example, an LDC instruction also uses a position-relative addressing scheme, but it may load a constant from memory to be used as an instruction or data pointer. That constant will be the original address of the target instruction or data, and will remain uncompensated for any repositioning of the program. The moral is that position-independent code can be created, with care.

Note that at all times, the program counter (R15) will be 8 bytes ahead, a result of prefetching to fill the processor's 2-word instruction pipeline. CASM automatically compensates for this when computing the offset to the target label. CLINK will do that compensation if the target is in another location counter or is an external label.

Intended Usage: Continue execution from a new address given by a label or an expression.

Operational Function: Jump to PC-relative address.

Flags Effected: (none)

Syntax: B *address_expression*

where *address_expression* may be an expression involving relocatable or external labels, or may be a fully defined (absolute) numeric value. If absolute, the processor will jump to that specific numeric address.

Examples:

```
BEQ   Contin_5        ; Relative branch to CONTIN_5 label.
B     0x3800000       ; Jump into ROM space.
```

Variations: A branch to a fixed address in memory *is* possible, e.g., a jump to 0x1000 or some other fixed address, regardless of any ORG statements used with the CLINK linker. This may be done in either of two ways:

1. Simply ensure that the target expression is an absolute address, without any relocatable labels in it.
2. Compute and load the target address into any register. Then MOV the result from that register into the PC (R15).

CASM recognizes the condition in method #1, and instructs the linker to process the address accordingly.

Appendix A.9 BIC - Bit Clear
Clear those bits in one operand indicated by the bits in the same position in the other operand. The result is placed into the specified destination register.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

As with all logical operations, no carries are involved between bits in the same register, whether the source or the destination. However, the Carry status flag is set if bit 31 is set in both of the source operand registers.

Note: BIC and ORR cannot be used (even in supervisor mode) to set or clear PSR bits. Use TEQP for that purpose.

Intended Usage: Mask out selected bits from the source register (Rn).

Operational Function: Rd = Rn AND 1's-complement-of (Op2)

Flags Effected: N, Z, C

Syntax: BIC{*condition*}{S} *Rd, Rn, Op2*

where
- *condition* is an optional 2-character condition code. See the Condition Code section.
- S (if present) sets condition codes based on the result.
- *Rd, Rn* are any valid register names, such as R0-R15, PC, SP, or LK.
- *Op2* is second operand, and may have any of the following forms:
  - *Rm shift Rs*
  - *Rm shift expression1*
  - *Rm RRX*
  - *expression2*
- *Rm* is any valid register names, as per *Rd* or *Rn* above, the operand value.
- *Rs* is a register, per *Rd* above, containing a shift count in range of 1..32.
- *shift* is any of: *ASL, LSL, LSR, ASR,* or *ROR*
- *expression1* is any positive absolute shift count in the range of 1..31.
- *expression2* is any signed expression shiftable into an 8-bit value.

Examples:

```
BIC   R1, R1, 5        ; Same as AND R1,R1,0xFFFFFFF2
BIC   R0, R0, 1        ; Clear LSB of R0.
```

Variations: If a negative constant is specified as the 2nd operand, the 1's complement of it is used, and a AND is substituted for the *BIC*. This effectively extends the range to 9 bits (including sign), and provides for sign extension to a full 32 bits.

Appendix A.10 BL - Branch with Link
Save the address of the next instruction, and then branch to the address indicated in the instruction. The target address is computed by adding the relative (word) offset given in the operand area of the instruction to the current value in the program counter (R15, or 'PC').

Branch-with-Link (BL) differs from the simple Branch (B) instruction in that it preserves the address of the next instruction in sequence in R14. This permits the routine at the target address to return to that next instruction when it completes its activity.

Intended Usage: Jump to a subroutine, saving address of next instruction for the return. To return from the subroutine, the following are two simple ways to get back:

```
1.    MOVS   PC, R14    ; Restore original status.
2.    MOV    PC, R14    ; Leave current status unchanged.
```

Many other variations to force a return are possible and are permitted.

Operational Function: Save PC in R14, and jump to PC-relative address.

Flags Effected: (none)

Syntax: BL address_expression

where address_expression may be an expression involving relocatable or external labels, or may be a fully defined (absolute) numeric value. If absolute, the processor will jump to that specific numeric address.

Examples:

```
BLGT   READ              ; Call the READ routine.
ACONS  ROM=0x3800000
BL     ROM+IO_WRITE      ; Call subroutine in ROM space.
```

Variations: A subroutine call to a fixed address in memory is possible, e.g., a jump to 0x1000 or some other fixed address, regardless of any ORG statements used with the CLINK linker. This may be done in either of two ways:

1. Simply ensure that the target expression is an absolute address, without any relocatable labels in it.

2. Compute and load the target address into any register. Then MOV the result from that register into the PC (R15).

CASM recognizes the condition in method #1, and instructs the linker to process the address accordingly.

Appendix A.11 CDO - Coprocessor Data Operations
Initiate some data processing action in an attached coprocessor. Actual function of the instruction is implementation dependent. No information (other than for register number and control information) is passed between the CPU and coprocessor. The instruction forces the following items to appear at the coprocessor interface:

- Three coprocessor register-number fields.
- A coprocessor number, specifying which of several coprocessors to activate.
- A 4-bit coprocessor opcode field, indicating the action to be performed.
- An additional unallocated 3-bit field to supply additional information to the coprocessor.

In actual fact, only coprocessor number and the CPU instruction's opcode bits are required by the hardware; all other fields are assigned within CASM by convention only. The assembler will accept information and assign values to the various fields as defined below. As with all coprocessor instructions, depending upon hardware design, they may hang the CPU up if they are executed without there being a hardware coprocessor that can respond to it.

Intended Usage: Force execution of an internal coprocessor opcode operation.

Operational Function: If the condition field evaluates true, instruct the coprocessor to perform the instruction assigned to the indicated coprocessor opcode.

Flags Effected: (none)

Syntax: CDO{*condition*} *cp#, coproc_opc, CRd, CRn, CRm {, expression}*

where
condition is any of the condition codes shown in the **Condition Codes** section.
cp# is an expression giving the coprocessor number, ranging 0..15.
coproc_opc is the coprocessor opcode, an expression in the range of 0..15.
Rn is a valid CPU destination register, R0..R15, SP, LK, or PC.
CRn, CRm, CRM are any valid coprocessor registers, CR0..CR15.
expression is an optional expression in the range of 0..7, of auxiliary information.

Examples:

```
CDONE  1, 6, CR9, CR1, CR0, 7
CDO    0, 13, CR12, CR3, CR3
```

Appendix A.12 CMN - Set Negative Compare

The CMN instruction is to compare an operand against a 2's complement negative value. It is the negative-number counterpart of the CMP instruction. See the Variations section below for the method of processing negative constants. (When comparing against a constant, it is suggested for maintainability and ease of understanding that CMP be used for all compares, letting CASM choose between CMP or CMN based upon the sign of the constant.) Of course, the second operand need not be a constant and may be a register.

This is a "logical" instruction, so no inter-bit carry is permitted in the hardware, and an overflow condition is not possible. The V status bit is, therefore, not altered by the instruction.

Because the only purpose for this instruction is to perform a test, setting the condition codes on the result, the 'S' suffix (save status) is redundant, and is automatically implied by CASM.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

Intended Usage: Comparison against a negative constant.

Operational Function: Rn + Op2 (result not stored)

Flags Effected: N, Z, C, V

Syntax: CMN*{condition}{P}* *Rn, Op2*

where *condition* is an optional 2-character condition code. See the Condition Code section.

P (if present) sets PSR bits based upon bits 28-31 of the ALU result.

Rn is any valid register names, such as R0-R15, PC, SP, or LK.

Op2 is second operand, and may have any of the following forms:

Rm
Rm shift Rs
Rm shift expression1
Rm RRX
expression2

Rm is any valid register names, as per *Rd* or *Rn* above, the operand value.

Rs is a register, per *Rd* above, containing a shift count in range of 1..32.

shift is any of: *ASL, LSL, LSR, ASR,* or *ROR*

expression1 is any positive absolute shift count in the range of 1..31.

expression2 is any signed expression shiftable into an 8-bit value.

Examples:

```
CMN   R0, -23        ; Same as CMP R0, 22
CMN   R10, R2        ; Equiv to CMP R10, (NOT R2)
```

Variations: If a negative constant is specified as the 2nd operand, the 2's complement of it is used, and a *CMP* is substituted for the *CMN*. This effectively extends the range to 9 bits (including sign), and provides for sign extension to a full 32 bits.
An *S* suffix is optional, and is always implied.

When a *P* suffix is used, the those bits of the 32-bit ALU result which map over the PSR bits in R15 are loaded directly into the PSR. This bypasses the usual status store to the PSR.

Appendix A.13 CMP - Arithmetic Comparison

Compare a register against the value in another register or a constant. No register is set with the result, but the flag bits in the PSR are updated accordingly. Constant values may be positive or negative, but consult the below Variations subsection for processing of negative values.

Because the only purpose for this instruction is to perform a test, setting the condition codes on the result, the 'S' suffix (save status) is redundant, and is automatically assumed by CASM.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

Intended Usage: Compare two operands for their relative size to each other.

Operational Function: Rn - Op2 (result is not saved)

Flags Effected: N, Z, C, V

Syntax: CMP*{condition}{P}* *Rn, Op2*

where *condition* is an optional 2-character condition code. See the Condition Code section.

P (if present) sets PSR bits based upon bits 28-31 of the ALU result.

Rn is any valid register names, such as R0-R15, PC, SP, or LK.

Op2 is second operand, and may have any of the following forms:

Rm shift Rs

Rm shift expression1

Rm RRX

expression2

Rm is any valid register names, as per *Rd* or *Rn* above, the operand value.

Rs is a register, per *Rd* above, containing a shift count in range of 1..32.

shift is any of: *ASL, LSL, LSR, ASR,* or *ROR*

expression1 is any positive absolute shift count in the range of 1..31.

expression2 is any signed expression shiftable into an 8-bit value.

Example: A routine to do character range checks and ASCII-hex conversion is given here. R1 holds hex string (upper case), and result goes into R0. Stop at first non-hex character found.

```
Hex     MOV   R0,0            ; Clear result.
Hex10   LDRB  R2,[R1],1       ; Get ASCII character.
        cmp   r2,"0"          ; Check 0-9.
        movcc PC,LK           ; Return to caller.
        cmp   R2,"9"
        subls r2,r2,'0'       ; Convert decimal to binary.
        bls   Hex20
        cmp   r2,"A"          ; Check A-F.
        movcc PC,LK           ; Return to caller.
        cmp   r2,"F"
        movgt PC,LK           ; Return to caller.
        sub   r2,r2,'A'-10    ; Convert hex to binary.
Hex20   add   r0,r2,r0 LSL 4  ; Merge in digit.
        b     Hex10           ; Do next digit.
```

Variations: If a negative constant is specified as the 2nd operand, the 2's complement of it is used, and a *CMN* is substituted for the *CMP*. This effectively extends the range to 9 bits (including sign), and provides for sign extension to a full 32 bits. An *S* suffix is optional, and is always implied. If a *P* suffix is used, the PSRbits are loaded directly from their equivalent positions in the 32-bit ALU result.

Appendix A.14 EOR - Logical Exclusive OR

The logical Exclusive OR operation is performed on two operand words, and the 32-bit result is written to the destination register. For each bit position in the two operands, a test is made to determine that they differ from each other, i.e., only one bit in each pair may be set, and one must be set. If so, the same bit position in the destination register is turned on; it is otherwise cleared. The operation is performed for each of the 32-bit positions.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

As with all logical operations, no carries are involved between bits in the same register, whether it is the source or the destination. The Exclusive OR operation may be viewed as an add operation without inter-bit carries.

Intended Usage: Compute the Exclusive-Or logical function of two operands, saving the results into a destination register.

Operational Function: Rd = (Rn AND NOT Op2) OR (Op2 AND NOT Rn)

Flags Effected: N, Z, C

Syntax: EOR{*condition*}{S} *Rd, Rn, Op2*

where

condition	is an optional 2-character condition code. See the Condition Code section.
S	(if present) sets condition codes based on the result.
Rd, Rn	are any valid register names, such as R0-R15, PC, SP, or LK.
Op2	is second operand, and may have any of the following forms: *Rm shift Rs* *Rm shift expression1* *Rm RRX* *expression2*
Rm	is any valid register names, as per *Rd* or *Rn* above, the operand value.
Rs	is a register, per *Rd* above, containing a shift count in range of 1..32.
shift	is any of: *ASL, LSL, LSR, ASR,* or *ROR*
expression1	is any positive absolute shift count in the range of 1..31.
expression2	is any signed expression shiftable into an 8-bit value.

Examples:

```
EOR   R5, R5,32     ; Complement bit 5.
EOR   R10,R10,r13
```

Appendix A.15 LDC - Load Coprocessor from Memory

LDC loads a coprocessor register from memory. Both a coprocessor and the desired register within it must be specified. This instruction is the coprocessor equivalent to the LDR instruction. As with the LDR, pre- and post-indexing of the Rn CPU register is provided for, and the target address may be a register-relative address.

Intended Usage: Load a coprocessor register from the indicated memory location.

Operational Function: Load the specified CRn register in the indicated coprocessor from memory. Indexing of a CPU register gives the effective memory address.

Flags Effected: (none)

Syntax: LDC{*condition*}{*L*}{*T*} *cp#, CRd, address*{*!*}

where

condition is one of the optional condition test codes described in **Condition Codes**.

N Implies a hardware-dependent function specified by the *N* bit. By convention, *N*=1 implies long transfer. If *N* is missing, a short transfer is indicated.

T Set the *W* bit, indicating that address translation is to take place. The *TRAN* pin is pulled low for the transfer cycle.

cp# is a coprocessor number in the range of 0..15.

CRd is a coprocessor register number, CR0..CR15.

! forces the effective address to be written back to *Rn*, if *Rn* is present.

address can be any of the variations given below:

expression	
[Rn]	(*T* suffix is not allowed)
[Rn, expression]	(*T* suffix is not allowed)
[[Rn], expression	

expression is an expression in the range of -1023 to +1023 (bytes) relative to the current program counter. It is scaled right 2 bits by CASM, and the complement of the sign is placed in the *U*-bit. The 8-bit absolute value if the expression is used in the instruction.

Rn is any valid (CPU) processor register, R0..R15. If R15 is used, the status bits are stripped before usage.

Examples:

```
LDC     1,CR2,[LK,-4]     ; Set CR2 from word after last call.
LDCEQ   2,CR0,[R5],4
LDC     2,cr0,800
```

Appendix A.16 LDM - Load Multiple Registers

From one to 16 registers may be loaded from memory by a single LDM instruction. Any specific register may be included in the register set list; registers in the set need not be contiguous. Sixteen bits in the instruction's operand field indicate which registers are to be loaded. Up to 16 registers may be loaded in one instruction.

Variations in the mnemonic indicate whether the registers are to be loaded in ascending or descending addresses, and whether the base pointer (stack) register is to be incremented/decremented before or after each register gets loaded. The lowest numbered register is always obtained from the lowest address in memory.

As with all instructions, the LDM is only executed if the status specified by the optional conditional code is met.

Intended Usage: Restore multiple registers at one time from a stack.

Operational Function: Perform repeated "pops" via a register designated as a stack base register to the registers supplied in a list. While the value in the stack base register is effectively updated during the transfer, the final value is not written back unless so indicated by the '!' suffix on the register list.

Flags Effected: None, unless the S-bit in the instruction has been set via the '^' caret marker.

Syntax: LDM*{condition}mode Rn{!},{reg_list}{^}*

where

condition is an optional condition, as given in the **Condition Codes** section.

mode is a required mode indicator, taken from the following table.

Rn is any valid register in the range of R0..R15.

! indicates that the updated base address is to be saved back into the Rn register.

^ User mode: ^ is ignored.
Non-User mode: If R15 is in list, PSR is loaded, and any other registers in the list reference the register bank of the *current* mode. Else, any registers in the list reference the user mode register bank.

{reg_list} (braces are required) is a list of registers to be loaded. They may be any of the valid registers R0..R15 separated by commas. A range of registers may also be included by separating them by a dash.

Modes: The above *mode* field must be selected from one of the following codes:

Codes	*Meaning*	*Usage*	*Function*
IB	Increment Before	Pop upwards	Pre-increment load
IA	Increment After	Pop upwards	Post-increment load
DB	Decrement Before	Pop downwards	Pre-decrement load
DA	Decrement After	Pop downwards	Post-decrement load

Other alternative forms for the above codes are supported, for completeness. They are not documented here, and their use is discouraged.

Examples:

```
STMDA   SP!,{R0-R5,LR}      ; Save regs & status.
LDMIB   SP!,{r0-r5,PC}^     ; Restore status and return.
LDMIB   R2,{R3-LR}          ; Restore a bunch.
```

Variations: R15 may be used in the transfer list. If loaded using an LDM, the PC's value will be reduced 12 bytes (3 words) from the value stored in memory, to compensate for the value stored by an STM. If the ^ marker is used, the PSR will be reset to the PSR value which was stored by the STM.

Appendix A.17 LDR - Load Register from Memory

Load a register with the 8-bit or 32-bit value obtained from the designated memory address. The operand address may be specified as relative to any register (including the PC), and either a word or a byte value may be loaded. If a word value is loaded, it must be word aligned, not straddling a word boundary. The ability to specify a base register and an increment or decrement amount is of significant value when accessing arrays of data, or when working with data pointers.

The base register may be offset by a 13-bit (including sign) constant either before or after the transfer. The constant is stored in the instruction in its positive form, and the complement of the original sign is stored in the U-bit field.

Alternatively, the base register may be modified (before or after the transfer) by the value contained in a second register. This modification register's value may optionally be first shifted or rotated rotated from 1 to 31 bits.

LDR differs from MOV OR LEA in that the latter loads values *from another register.* When operating in supervisor mode, the T suffix may be appended (with post-incrementing only) to force the normally untranslated memory address space to be translated. This uses the logical-to-physical address translation tables inside the MEMC memory controller, via the TRAN pin of the processor.

Intended Usage: Load an 8-bit or 32-bit quantity from memory into a register. The memory address may be PC-relative or register relative. (An ordinary program label would be PC-relative). A MOV should be used to load a constant value to the register.

Operational Function: Load register Rd from the effective address.

Flags Effected: (none)

Syntax: LDR{condition}{B}{T} Rd, address {!}

where *condition* is a code given in the section on **Condition Codes**.

B is given to force the loading of an 8-bit byte, rather than a 32-bit word.

T is given (in post-increment mode only) to force an address translation.

Rd is any valid CPU register, R0..R15.

! forces the Rn register to be updated by the value of the offset afterwards.

address is any of the following variations:

Variation	*Effective Address*	*Mode*
[Rn]	Rn	N/A
[Rn, expression]	Rn + expression	Pre-indexed.*
[Rn, Rm]	Rn + Rm	Pre-indexed.*
[Rn, Rm shift count]	Rn + (Rm shifted by *count*).	Pre-indexed.*
[Rn], expression	Rn	Post-increment.
[Rn], Rm	Rn	Post-increment.
[Rn], Rm shift count]	Rn	Post-increment.

Rn is any valid CPU register, R0..R15, and holds the transfer base address.

Rm is any valid CPU register, R0..R15, and holds a (signed) address increment.

expression is an expression in the range of -4095 to +4095.

shift is any shift type indicator: LSL, LSR, ASR, ROR, or RRX

count is any constant in the range of 1..31, and is the shift *count.*

*If ! follows the '] ', then Rn is also incremented, i.e. post increment mode.

Appendix A.17 LDR (Cont.)

Remarks: The "address modifier" is the amount to add to the base transfer address (in Rn). It is added to Rn before the transfer if pre-indexed, or after the transfer if post-indexed. Pre- or post-indexing is determined by where the modifier is found. If it is given *inside* the [ ] brackets, it is a pre-indexed case, and the modifier becomes included in the effective address. If given *outside* of the [ ] brackets, it is a post-indexed case; the modifier comes into play only *after* the transfer has taken place.

Examples:

```
LDR     R1, [R15]
LDREQ   R3, [SP+0x10]!          ; SP = SP+16.
LDR     R5, [R3, R2 SHL 2]
LDR     LK, [LK]
```

Variations: R15 usage has a number of special cases associated with it:

1. PSR is never modified, even when Rd or Rn is the PC.
2. If Rn is R15, the PC is used without any of the PSR flags. Note: it will be advanced by 8 bytes from the current instruction.
3. If PC is used as the offset (Rm) register, the value used *includes* the flags, and thus will be an invalid address unless they are all zero.

Appendix A.18 LEA - Load Effective Address

This pseudo-instruction may be used to load any register with a large constant or an address that is outside of the range of an 8-bit offset (or is unknown). Note that the effective address, not the value stored at that address, is loaded to the designated register.

Intended Usage: Load the effective address of distant locations (or large constants) to a register.

Operational Function: Generate a variant machine instruction to load the desired constant. If necessary, create a forward-reference entry in a literal constant table that is within reach of this instruction. Created instruction may be any of ADD, SUB, MOV, MVN, or LDR.

Flags Effected: None, if address is a forward reference, is in a different location counter, is external, or is outside of 256-byte range. Otherwise, N, Z, C, V.

Syntax: LEA *Rd, expression*

where *Rd* is the (destination) register to be loaded with a value or address, R0..R15.
expression is an expression of any size, absolute, relocatable, or external, that is legal within the assembler.

Example:

```
LEA   R10,Table+20        ; Reverse reference expression.
LEA   R1, Savearea        ; Reverse reference.
LEA   R0, 0x12345678      ; A large constant.
LEA   R9, Forward_Ref     ; Forward reference.
```

Remarks: If the effective address (or value) is within a 256-byte range of the program counter or is relative to some register, a simple MOV, MVN, ADD, or SUB instruction is generated to perform the load of the effective address. If this is not possible, the effective address is computed and is stored as a 32-bit memory word in a "long reach" table. An LDR is generated to load the value to the designated register.

Constants which will be inserted into the long-reach table are accumulated by the assembler, leaving the corresponding LEA unresolved. When the program has been processed to a point where the distance from the farthest unresolved LEA reaches 4096 bytes, a long-reach table is inserted into the assembler source. (Actually, the offset may not reach precisely 4096 bytes, as the size of the table itself is accounted for in the distance.)

The current release of CASM generates a new table entry whenever a forward reference is involved. That is, the entry is made if the target label (or expression) has not yet been defined.

There is one table accumulated for each location counter. If any have not been inserted into the program source by the end of the source file, they are inserted at that time. Whenever a table is inserted into the program source by the assembler, due to the distance from the farthest unresolved LEA, a branch instruction is prefixed to it, so that the processor will avoid the table during execution.

In addition, the programmer may specify where a long-reach table is to be insertped, by using the *REACH* pseudo-instruction. The directive itself is effectively replaced by the generated table. In this case, since the programmer has specified where the table is to be placed, CASM will not first create the bypassing branch instruction. The long-reach table for the currently active LC is inserted. Other tables may be inserted (if they are known to exist) by using NEWLC to switch to a new location counter, and following it with a REACH directive.

Whenever a long-reach table is inserted, a new one is started for the subsequent source code. There may be multiple tables for a single location counter if the program is long enough to warrant them.

Appendix A.19 MCR - Move Coprocessor to CPU

Transfer a register from an attached coprocessor to a CPU register, optionally performing some action within the coprocessor. The instruction forces the following items to appear at the coprocessor interface:

- Two coprocessor register-number fields.
- A coprocessor number, specifying which of several coprocessors must respond.
- A 3-bit coprocessor opcode field, indicating the action to be performed.
- An additional unallocated 3-bit field to supply additional information in.

Only coprocessor number and the CPU instruction's opcode bits are actually required by the hardware; all other fields are assigned within CASM by convention only. The assembler will accept information and assign values to the various fields as defined below. Other than to load Rd from the data bus, the operation of the instruction is entirely implementation dependent.

Intended Usage: Transfer a 32-bit data register from the coprocessor to a CPU register.

Operational Function: If the condition field evaluates true, read the indicated register from the data bus, passing along the other information to the coprocessor.

Flags Effected: (none)

Syntax: MCR{*condition*} *cp#, coproc_opc, Rd, CRn, CRm {, expression}*

where	*condition*	is any of the condition codes shown in the **Condition Codes** section.
	cp#	is an expression giving the coprocessor number, ranging 0..15.
	coproc_opc	is the coprocessor opcode, an expression in the range of 0..7.
	Rn	is a valid CPU destination register, R0..R15, SP, LK, or PC.
	CRn, CRm	are any valid coprocessor registers, CR0..CR15.
	expression	is an optional expression in the range of 0..7, of auxiliary information.

Examples:

```
MCR    1,6, R9, CR1, CR0, 0
MCRLT  0, 0,R12, CR3, CR3, Code-'A'        ; Code is 'A' thru 'G'.
```

Appendix A.20 MLA - Multiply and Accumulate

Perform a 32-bit by 32-bit multiply to yield a 32-bit result. A single 32-bit value is then added to the result. All operands and the result are contained in registers. A modified Booth algorithm is used, and the results are obtained in 16 clock times worst case. If the upper bits of the operands are clear, a truncated cycle is used to return the results faster.

MLA differs from MUL in that it is intended for multiple-precision operands. The typical usage is for accumulating the inner products of the multi-word operands.

Either (or neither) operand may be a signed value, and the resulting sign is correctly processed. When the multiplication of byte values is involved, they are treated as full 32-bit operands, and the result appears in the lowest bits of the Rd register as expected.

Overflow is not possible, as the sign bit from one operand will be redundant in the result. The freed bit is sufficient to hold the data that might otherwise overflow.

The Rd and Rm registers may not be one and the same register, and R15 (PC) may not be used for Rd. The result is not meaningful if Rm = Rd.

The status bits may be set based upon the result, if the S suffix is used.

Intended Usage: Multiply two 32-bit values to produce a 32-bit result, adding in a (partial product) result, typically from a previous multiply.

Operational Function: *Rd = Rm * Rs + Rn*

Flags Effected: N, Z(*C* is scrambled, and *V* is not effected.)

Syntax: MLA *Rd, Rm, Rs, Rn*

where
- *Rd* is the destination register, and is R0-R14, SP, or LK.
- *Rm, Rs* are operand registers, and are R0-R15, SP, LK, or PC.
- *Rn* is a partial-products intermediate addend register, as per Rm.

Examples:

```
MLAS  R1,R2,R3,R4
MLA   R1,R2,R1,R4
```

Variations: The following exception conditions exist:

1. The Rd must not be the same as the Rm register.
2. The Rd may not be R15 (PC).
3. If R15 is used as an operand, it will be displaced on beyond the instruction.
4. When the PC is used as the Rm the PSR flags are included, and the PC is offset +12.
5. When the PC is used as the Rs the PSR flags are ignored, and the PC is offset +8.
6. When the PC is used as the Rn the PSR flags are included, and the PC is offset +8.
7. A 64-bit result can be synthesized using 4 multiplies, using 16-bit partial factors. Each partial factor is the upper or lower portion of a 32-bit operand, and each has the 16 upper bits cleared, permitting an early exit from the multiply after a maximum of 8 clocks each.

Appendix A.21 MOV - Move Register or Constant

Move a 32-bit item from one register to another, or move an 8-bit constant into a register. When an 8-bit constant is supplied as the second operand, the constant may consist of any 8-bit wide pattern in a 32-bit field. It will be rotated to produce an 8-bit constant in the least significant bits of the instruction, but will be re-expanded at execution time to its proper position within the destination register.

MOV can only load positive constants into a register. If a negative value is given, CASM automatically substitutes a MVN opcode into the instruction, and compensates the operand for the 1's complement format used by MVN. (Similarly, CASM will convert a MVN instruction with a negative operand into the equivalent MOV instruction.)

Intended Usage: Move the contents of one register to another, or load a constant into a register.

Operational Function: Rd = Op2

Flags Effected: N, Z, C

Syntax: MOV{*condition*}{S} *Rd, Op2*

where
- *condition* is an optional 2-character condition code. See the Condition Code section.
- S (if present) sets condition codes based on the value of the operand.
- *Rd* is any valid register name, such as R0-R15, PC, SP, or LK.
- *Op2* is second operand, and may have any of the following forms:
  - *Rm shift Rs*
  - *Rm shift expression1*
  - *Rm RRX*
  - *expression2*
- *Rm* is any valid register name, as per *Rd* or *Rn* above, the operand value.
- *Rs* is a register, per *Rd* above, containing a shift count in range of 1..32.
- *shift* is any of: *ASL, LSL, LSR, ASR,* or *ROR*
- *expression1* is any positive absolute shift count in the range of 1..31.
- *expression2* is any signed expression that can be rotated into an 8-bit value in the least significant bits.

Examples:

```
MOVS  PC, LK          ; Same as: (PC, PSR) = R14.
MOV   PC, R14         ; Sub return, previous status.
MOV   R1, 0xC000003F  ; Load big constant.
MOV   R13, -254       ; Load negative constant.
MOV   R1, -1          ; Special case is handled.
MOV   R8, R6 LSR R3   ; R3 has shift count.
```

Variations: If a negative constant is specified as the 2nd operand, the 1's complement of it is used, and a *MVN* is substituted for the *MOV.* This effectively extends the range to 9 bits (including sign), and provides for sign extension to a full 32 bits.

Appendix A.22 MRC - Move CPU Register to Coprocessor

Transfer a processor register to an attached coprocessor, optionally performing some action within the coprocessor. The instruction forces the following items to appear at the coprocessor interface:

- Two coprocessor register-number fields.
- Data from a designated CPU register.
- A coprocessor number, specifying which of several coprocessors to activate.
- A 3-bit coprocessor opcode field, indicating the action to be performed.
- An additional unallocated 3-bit field to supply additional information in.

In actual fact, only coprocessor number and the CPU instruction's opcode bits are required by the hardware; all other fields are assigned within CASM by convention only. The assembler will accept information and assign values to the various fields as defined below. Other than to make Rd available for writing, the operation of the instruction is entirely implementation dependent.

Intended Usage: Transfer a 32-bit data register from the CPU to the coprocessor.

Operational Function: If the condition field evaluates true, place the indicated register on the data bus along with the other control information.

Flags Effected: (none)

Syntax: MRC{*condition*} *cp#, coproc_opc, Rd, CRn, CRm {, expression}*

where
- *condition* is any of the condition codes shown in the **Condition Codes** section.
- *cp#* is an expression giving the coprocessor number, ranging 0..15.
- *coproc_opc* is the coprocessor opcode, an expression in the range of 0..7.
- *Rn* is a valid CPU source register, R0..R15, SP, LK, or PC.
- *CRn, CRm* are any valid coprocessor registers, CR0..CR15.
- *expression* is an optional expression in the range of 0..7, of auxiliary information.

Examples:

```
MRCNE 1, 6, R9, CR15, CR0, 5
MRC   0, 0, R1, CR3, CR3
```

Appendix A.23 MUL - Multiply
Perform a 32-bit by 32-bit multiply to yield a 32-bit result. All operands and the result are contained in registers. A modified Booth algorithm is used, and the results are obtained in 16 clock times worst case. If the upper bits of the operands are clear, a truncated cycle is used to return the results faster.

Either (or neither) operand may be a signed value, and the resulting sign is correctly processed. When the multiplication of byte values is involved, they are treated as full 32-bit operands, and the result appears in the lowest bits of the Rd register as expected.

The Rd and Rm registers may not be one and the same register, and R15 (PC) may not be used for Rd. The result is meaningless if Rm = Rd. The Rn register field is forced to zero for compatibility with future processor family derivatives.

The status bits may be set based upon the result, if the S suffix is used.

Intended Usage: Multiply two 32-bit values to produce a 32-bit result.

Operational Function: *Rd = Rm * Rs*

Flags Effected: N, Z (*C* is scrambled, and *V* is not effected.)

Syntax: MUL *Rd, Rm, Rs*

where *Rd* is the destination register, and is R0-R14, SP, or LK.

Rm, Rs are operand registers, and are R0-R15, SP, LK, or PC.

Examples:

```
MULS  R1,R2,R3
MUL   R1,R2,R1        ; Source & Destination are the same.
```

Variations: The following exception conditions exist:

1. The *Rd* must not be the same as the *Rm* register.
2. The *Rd* may not be R15 (PC).
3. When the PC is used as the *Rm* the PSR flags are included, and the PC is offset +12.
4. When the PC is used as the *Rs* the PSR flags are ignored, and the PC is offset +8.
5. A 64-bit result from multiplying two 32-bit operands may be synthesized by using 4 multiplies, yielding partial products. The four 16-bit factors used in the cross multiply are made up of the upper and lower portions of the original 32-bit operands. By clearing the upper 16 bits of each partial factor, an early exit may be taken by the hardware for each multiply, using only 8 clock cycles each.

Appendix A.24 MVN - Move Complement of Register

This instruction loads the 1's complement of a constant (or of another register) into the destination. The instruction moves a 32-bit item. When an 8-bit constant is supplied as the second operand, the constant may consist of any 8-bit wide pattern in a 32-bit field. It will be rotated to produce an 8-bit constant in the least significant bits of the instruction, but will be rotated at execution time to its proper position within the destination register.

The hardware instruction MVN can only load (the complement of) positive constants into a register. If a negative value is given, CASM will automatically substitute a MOV instruction, using the 1's complement of the operand. (Similarly, CASM will convert a MOV instruction with a negative operand into the equivalent MVN instruction.)

Intended Usage: Load destination register with (1's) complement of a constant or a register.

Operational Function: Rd = 0xFFFFFFFF XOR Op2

Flags Effected: N, Z, C

Syntax: MVN{*condition*}{S} *Rd, Op2*

where *condition* is an optional 2-character condition code. See the Condition Code section.

S (if present) sets condition codes based on the result.

Rd is any valid register names, such as R0-R15, PC, SP, or LK.

Op2 is second operand, and may have any of the following forms:

Rm shift Rs

Rm shift expression1

Rm RRX

expression2

Rm is any valid register names, as per *Rd* or *Rn* above, the operand value.

Rs is a register, per *Rd* above, containing a shift count in range of 1..32.

shift is any of: *ASL, LSL, LSR, ASR,* or *ROR*

expression1 is any positive absolute shift count in the range of 1..31.

expression2 is any signed expression that can be rotated into an 8-bit value in the least significant bits.

Examples:

```
MVN   R12,R5
MVNNV R0, R0          ; A no-operation case.
MVN   R3, R2 ASR 5
MVN   R4, R4 RRX      ; Shift complement thru carry.
```

Variations: If a negative constant is supplied, the value is complemented and a *MOV* is substituted for the *MVN*. This effectively permits a 9-bit constant (including sign) that is sign extended to 32 bits in the destination.

Appendix A.25 ORR - Logical OR
The logical OR operation is performed on two operand words, and the 32-bit result is written to the destination register. For each bit position in the two operands, a test is made to determine if either is set (1). If so, the same bit position in the destination register is turned on. It is otherwise turned off. The same operation is performed for each of the 32-bit positions.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

As with all logical operations, no carries are involved between bits in the same register, whether the source or the destination.

Note: BIC and ORR cannot be used (even in supervisor mode) to set or clear PSR bits. Use TEQP for that purpose.

Intended Usage: Perform a logical OR between equivalent bit positions in the two source registers.

Operational Function: Rd = Rn OR Op2

Flags Effected: N, Z, C

Syntax: ORR{*condition*}{S} *Rd, Rn, Op2*

where
- *condition* is an optional 2-character condition code. See the Condition Code section.
- S (if present) sets condition codes based on the result.
- *Rd, Rn* are any valid register names, such as R0-R15, PC, SP, or LK.
- *Op2* is second operand, and may have any of the following forms:
  - *Rm shift Rs*
  - *Rm shift expression1*
  - *Rm RRX*
  - *expression2*
- *Rm* is any valid register names, as per *Rd* or *Rn* above, the operand value.
- *Rs* is a register, per *Rd* above, containing a shift count in range of 1..32.
- *shift* is any of: *ASL, LSL, LSR, ASR,* or *ROR*
- *expression1* is any positive absolute shift count in the range of 1..31.
- *expression2* is any signed expression that can be rotated into an 8-bit value in the least significant bits.

Examples:

```
ORR   R1,R1,R0          ; R1 = R1 OR R0.
ORRS  R0,R0,32          ; Force ASCII to lower case.
ORR   R0,R1,R3 LSR 4
```

Appendix A.26 RSB - Reverse-Operand Subtract
This instruction is identical in operation to the SUB instruction, except that the operand order is reversed. In SUB, the first operand must be in a register and considerable flexibility is given in addressing the second one. RSB permits the addressing flexibility to effectively be applied to the first operand.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

Intended Usage: Perform a subtraction of two 32-bit operands, or perform the subtraction of the lower words in a multi-precision operand pair. The minuend is obtained from an indexed address. I.e., *Rn* is subtracted from operand 1.

Operational Function: Rd = Op1 - Rn

Flags Effected: N, Z, C, V

Syntax: RSB{*condition*}{S} *Rd, Rn, Op1*

where
- *condition* is an optional 2-character condition code. See the Condition Code section.
- S (if present) sets condition codes based on the result.
- *Rd, Rn* are any valid register names, such as R0-R15, PC, SP, or LK.
- *Op1* is second operand, and may have any of the following forms:
  - *Rm shift Rs*
  - *Rm shift expression1*
  - *Rm RRX*
  - *expression2*
- *Rm* is any valid register names, as per *Rd* or *Rn* above, the operand value.
- *Rs* is a register, per *Rd* above, containing a shift count in range of 1..32.
- *shift* is any of: *ASL, LSL, LSR, ASR,* or *ROR*
- *expression1* is any positive absolute shift count in the range of 1..31.
- *expression2* is any signed expression that can be rotated into an 8-bit value in the least significant bits.

Examples: Subtract R5 from a very large constant.

```
RSB   R5, R5, 0xEA000000
```

Appendix A.27 RSC - Rev-Operand Subtract, Carry

The RSC instruction is identical to the SBC instruction, except that the order of the two operands is reversed. In the SBC, the first operand must be contained in a register while considerable flexibility is given in the addressing of the second. The RSC may be used when the more flexible addressing is needed for the first operand. The "carry" operation is actually a borrow operation.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

Intended Usage: Perform subtract of upper words in a multi-precision operand pair, where the minuend is obtained from an indexed address. I.e., *Rn* is subtracted from operand 1. Carry is also added in (a "borrow" is performed).

Operational Function: Rd = Op1 - Rn - 1 + Carry

Flags Effected: N, Z, C, V

Syntax: RSC{*condition*}{S} *Rd, Rn, Op1*

where *condition* is an optional 2-character condition code. See the Condition Code section.

S (if present) sets condition codes based on the result.

Rd, Rn are any valid register names, such as R0-R15, PC, SP, or LK.

Op1 is second operand, and may have any of the following forms:

Rm shift Rs
Rm shift expression1
Rm RRX
expression2

Rm is any valid register names, as per *Rd* or *Rn* above, the operand value.

Rs is a register, per *Rd* above, containing a shift count in range of 1..32.

shift is any of: *ASL, LSL, LSR, ASR,* or *ROR*

expression1 is any positive absolute shift count in the range of 1..31.

expression2 is any signed expression that can be rotated into an 8-bit value in the least significant bits.

Example: Subtract a 64-bit number in R0,R1 from the 64-bit number 0x3FC00000,00000000

```
RSB   R0,R0,0          ; Handle LSBs.
RSC   R0,R0,0x3FC00000 ; Handle MSBs (with "borrow").
```

Appendix A.28 SBC - Subtract, with Carry

SBC subtracts two 32-bit operands, placing the difference into a register. A value of +1 is subtracted from the difference if the carry bit was clear prior to the instruction; nothing is subtracted from the difference if the carry was previously set.

The normal use for Subtract-with-Carry is to compute difference of numbers that are greater than 32 bits in length. The multi-precision subtraction sequence is to SUB the lowest words together (without carry compensation), possibly generating a carry in the process. The next most significant word pair is then subtracted using SBC, with the carry from the first pair correcting the difference.

If even more precision is used than two words per operand, they are successively SBC'd together until the most significant word pair has been subtracted. (The same process is used for multi-precision addition, but using ADD and ADC.)

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

2

Intended Usage: Multi-precision subtraction.

Operational Function: Rd = Rn - Op2 - 1 + Carry

Flags Effected: N, Z, C, V

Syntax: SBC{*condition*}{S} *Rd, Rn, Op2*

where

condition	is an optional 2-character condition code. See the Condition Code section.
S	(if present) sets condition codes based on the result.
Rd, Rn	are any valid register names, such as R0-R15, PC, SP, or LK.
Op2	is second operand, and may have any of the following forms: *Rm shift Rs* *Rm shift expression1* *Rm RRX* *expression2*
Rm	is any valid register names, as per *Rd* or *Rn* above, the operand value.
Rs	is a register, per *Rd* above, containing a shift count in range of 1..32.
shift	is any of: *ASL, LSL, LSR, ASR,* or *ROR*
expression1	is any positive absolute shift count in the range of 1..31.
expression2	is any signed expression that can be rotated into an 8-bit value in the least significant bits.

Example: Assume that R0,R1 holds a 64-bit integer, as does the R2,R3 pair. Subtract the second pair from the first.

```
SUB   R0, R2,R2     ; Handle the LSBs.
SBC   R1,R1,R3      ; Handle the MSBs (with borrow).
```

Variations: If a negative constant is specified as the second operand, the 1's complement of it is used, and an ADD is substituted for the SUB. This effectively extends the range to 9 bits (including sign), and provides for sign extension to a ful 32 bits.

Appendix A.29 STC - Store Coprocessor to Memory

STC stores the contents of a coprocessor register to memory. Both a coprocessor and the desired register within it must be specified. This instruction is the coprocessor equivalent to the STR instruction. As with the STR, pre- and post-indexing of the *Rn* CPU register is provided for, and the target address may be a register relative address.

Intended Usage: Store coprocessor register to indicated memory location.

Operational Function: Store contents of specified CRn coprocessor register of the indicated coprocessor to memory. Indexing of a CPU register gives the effective memory address.

Flags Effected: (none)

Syntax: STC{*condition*}{*L*}{*T*} *cp#, CRd, address*{*!*}

where *condition* is one of the optional condition test codes described in **Condition Codes**.

N Implies a hardware-dependent function specified by the *N* bit. By convention, *N*=1 implies long transfer. If *N* is missing, a short transfer is indicated.

T Set the *W* bit, indicating that address translation is to take place. The *TRAN* pin is pulled low for the transfer cycle.

cp# is a coprocessor number in the range of 0..15.

CRd is a coprocessor register number, CR0..CR15.

! forces the effective address to be written back to *Rn*, if *Rn* is present.

address can be any of the variations given below:

expression	
[Rn]	(*T* suffix is not allowed)
[Rn, expression]	(*T* suffix is not allowed)
[[Rn], expression	

expression is an expression in the range of -1023 to +1023 (bytes) relative to the current program counter. It is scaled right 2 bits by CASM, and the complement of the sign is placed in the *U*-bit. The 8-bit absolute value if the expression is used in the instruction.

Rn is any valid (CPU) processor register, R0..R15. If R15 is used, the status bits are stripped before usage. If *Rn* is missing, *expression* is assumed to be relative to R15.

Examples:

```
STC   1,CR5,[R1]      ; Load Indirect on R1.
STC   1,CR5,[R2],4    ; As above. Then R1=R1+4.
STC   2,CR7,Label     ; Label same as PC+Label.
```

Appendix A.30 STM - Store Multiple Registers

From one to 16 registers may be stored to memory by a single STM instruction. Any specific register may be included in the register set list, and registers in the set need not be contiguous. Sixteen bits in the instruction's operand field indicate which registers are to be loaded.

Variations in the mnemonic indicate whether the registers are to be stored in ascending or descending addresses, and whether the base pointer (stack) register is to be incremented/decremented before or after each register gets stored. The lowest numbered register is always stored to the lowest address in memory.

As with all instructions, the STM is only executed if the status specified by the optional conditional code is met.

Intended Usage: Save multiple registers at one time onto the system or user stack.

Operational Function: Perform repeated "pushes" via a register designated as a stack base register from the registers supplied in a list. While the stack base register is effectively updated during the transfer, the final value is not written back unless so indicated by the '!' suffix on the base register.

Flags Effected: None, unless the S-bit in the instruction has been set via the '^' caret marker.

Syntax: STM*{condition}mode Rn{!},{reg_list}{^}*

where

condition	is an optional condition, as given in the **Condition Codes** section.
mode	is a required mode indicator, taken from the following table.
Rn	is any valid register in the range of R0..R15.
!	indicates that the updated base address is to be saved back into the Rn register.
^	User mode: ^ is ignored. Non-User mode: Forces reference of user mode register bank. Note: PSR is always stored if R15 is in the list.
{reg_list}	(braces are required) is a list of registers to be stored. They may be any of the valid registers R0..R15 separated by commas. A range of registers may also be included by separating them by a dash.

Modes: The above *mode* field must be selected from one of the following codes:

Codes	*Meaning*	*Usage*	*Function*
IB	**I**ncrement **B**efore	Push upwards	Pre-increment store
IA	**I**ncrement **A**fter	Push upwards	Post-increment store
DB	**D**ecrement **B**efore	Push downwards	Pre-decrement store
DA	**D**ecrement **A**fter	Push downwards	Post-decrement store

Other alternative forms for the above codes are supported, for completeness. These earlier forms are not documented here, and their use is discouraged.

Example: Simulate a conventional push-down stack, where the stack pointer gets updated after each transfer, and pushes downward:

```
STMDA   SP!,{R4,R5,R9-R11,PC}
```

Variations: R15 may be used in the transfer list. If stored using the *STM,* the PC's value will be advanced 12 bytes (3 words) forward of the *STM.*, and the status will be stored with it. A later *LDM* may use the ^ marker to load the PSR with the value which was stored by the *STM.*

Appendix A.31 STR - Store Register to Memory

Store a 32-bit register value to the designated memory address. The operand address may be specified as relative to any register (including the PC), and either a word or a byte value may be stored.

If a word value is stored, it must be word aligned, not straddling a word boundary. The ability to specify a base register and an increment or decrement amount is of significant value when accessing arrays of data, or when working with data pointers.

The base register may be offset by a 13-bit (including sign) constant either before or after the transfer. The constant is stored in the instruction in its positive form, and the complement of the original sign is stored in the U-bit field.

Alternatively, the base register may be modified (before or after the transfer) by the value contained in a second register. This modification register's value may optionally be first shifted or rotated rotated from 1 to 31 bits.

STR differs from a MOV in that the MOV stores a value to another register.

Intended Usage: Store a register to specified (PC-Relative) memory address.

Operational Function: Store a single register at any address within the range of ± 4095 bytes from the current PC (R15), or relative to any other register (such as a data-frame or stack-frame register.

Flags Effected: (none)

Syntax: STR{condition}{B}{T} Rd, address {!}

where *condition* is a code given in the section on **Condition Codes.**

B is given to force the storing of an 8-bit byte, rather than a 32-bit word.

T is given (in post-indexed mode only) to force an address translation.

Rd is any valid CPU register, R0..R15.

! forces the Rn register to be updated by the value of the offset afterwards.

address is any of the following variations:

Variation	*Effective Address*	*Mode*
[Rn]	Rn	N/A.
[Rn, expression]	Rn + expression	Pre-indexed.*
[Rn, Rm]	Rn + Rm	Pre-indexed.*
[Rn, Rm shift count]	Rn + (Rm shifted by *count*).	Pre-indexed.*
[Rn], expression	Rn	Post-increment.
[Rn], Rm	Rn	Post-increment.
[Rn], Rm shift count]	Rn	Post-increment.

Rn is any valid CPU register, R0..R15, and holds the transfer base address.

Rm is any valid CPU register, R0..R15, and holds a (signed) address increment.

expression is an expression in the range of -4095 to +4095.

shift is any shift type indicator: LSL, LSR, ASR, ROR, or RRX

count is any constant in the range of 1..31, and is the shift *count.*

*If ! follows the '] ', then Rn is also incremented, i.e., post increment mode.

Appendix A.31 STR (Cont.)

Remarks: The "address modifier" is the amount to add to the base transfer address (in Rn). It is added to Rn before the transfer if *pre-indexed,* or after the transfer if *post-incremented.* Pre-indexing or post-incrementing is determined by where the modifier is found. If it is given *inside* the [ ] brackets, it is a pre-indexed case, and the modifier becomes included in the effective address. If given *outside* of the [ ] brackets, it is a post-increment case; the modifier comes into play only *after* the transfer has taken place.

Examples:

```
STR     R1, [R15]
STREQ   R3, [SP+0x10]!          ; SP = SP+16.
STR     R5, [R3, R2 SHL 2]
STR     LK, [LK]
```

Variations: R15 usage has a number of special cases associated with it:

1. PSR is never modified, even when Rd or Rn is the PC.
2. If Rn is R15, the PC is used without any of the PSR flags. Remember that it will be advanced by 8 bytes from the current instruction.
3. If PC is used as the offset (Rm) register, the value used *includes* the flags.

Appendix A.32 SUB - Subtract
Subtract one 32-bit operand from another, putting the result back into a register. The first operand must be a register, but the second is permitted a much more general addressing scheme. An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

Intended Usage: Compute the arithmetic difference of two operands.

Operational Function: Rd = Rn - Op2

Flags Effected: N, Z, C, V

Syntax: SUB{*condition*}{S} *Rd, Rn, Op2*

where
- *condition* is an optional 2-character condition code. See the Condition Code section.
- S (if present) sets condition codes based on the result.
- *Rd, Rn* are any valid register names, such as R0-R15, PC, SP, or LK.
- *Op2* is second operand, and may have any of the following forms:
  - *Rm*
  - *Rm shift Rs*
  - *Rm shift expression1*
  - *Rm RRX*
  - *expression2*
- *Rm* is any valid register names, as per *Rd* or *Rn* above, the operand value.
- *Rs* is a register, per *Rd* above, containing a shift count in range of 1..32.
- *shift* is any of: *ASL, LSL, LSR, ASR,* or *ROR*
- *expression1* is any positive absolute shift count in the range of 1..31.
- *expression2* is any signed expression shiftable into an 8-bit value.

Examples:

```
SUB   R1,R1,R2        ; Set R1 = R1-R2
Sub   R0,R0,"A"       ; Subtract a constant from R0.
```

Appendix A.33 SWI - Software Interrupt

Perform a "software interrupt" (system call), changing the processor into supervisor mode. This is equivalent to a subroutine call to the routine whose entry point is branched to by the branch instruction in location 0 x 8 in physical memory. By convention, the action taken by that routine is defined entirely by the system SVC-mode (SWI) handler. The instruction's lower 24-bit field is interpreted by that handler.

The instruction is conventionally used to pass requests to the operating system for I/O transfers and other system-specific operations. When operating in "user" mode, the program will not have access to the I/O space in memory mapped systems; the only recourse is to offload system input/output to the executive.

Intended Usage: Request operating-system or I/O function of the system executive.

Operational Function: Pass a 24-bit field to the system supervisor for interpretation.

Flags Effected: (determined by the supervisor)

Syntax: SWI{*condition*} *operand*

where *condition* is an optional 4-bit code defined in the **Condition Codes** section.
operand is a 24-bit expression that is right-justified in the SWI instruction.

Examples: Predefine certain I/O operations to be done by the operating system. Assume READB reads a byte into bits 0-7 of R, and WRITEB writes the byte constant found in bits 0-7 R0 to some I/O device.

```
acons   READB = 0x10
acons   WRITEB = 0x20
SWI     READB           ; Read byte to R0.
MOV     R0, 'J'
SWI     WRITEB          ; Write 'J' to device.
```

Appendix A.34 TEQ - Set Condition Codes via XOR

Test that the two operands are equal, but without saving any results except for the status bits. This differs from a CMP or CMN in that no overflow or carry is possible. (Carry will be cleared). This is a "logical" instruction, so no inter-bit carry is permitted in the hardware, and overflow is not possible. The V status bit is therefore not altered, however, C is reset by the instruction.

Because the only purpose for this instruction is to perform a test, setting the condition codes on the result, the 'S' suffix (save status) is redundant, and is automatically implied by CASM.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

Intended Usage: Test for bit-wise equality, without regard to relative magnitude.

Operational Function: Rn XOR Op2 (result is not stored)

Flags Effected: N, Z, C

Syntax: TEQ{*condition*}{P} *Rn, Op2*

where	*condition*	is an optional 2-character condition code. See the Condition Code section.
	P	Force PSR loading, directly from 32-bit ALU result.
	Rn	is any valid register names, such as R0-R15, PC, SP, or LK.
	Op2	is second operand, and may have any of the following forms: *Rm shift Rs* *Rm shift expression1* *Rm RRX* *expression2*
	Rm	is any valid register names, as per *Rd* or *Rn* above, the operand value.
	Rs	is a register, per *Rd* above, containing a shift count in range of 1..32.
	shift	is any of: *ASL, LSL, LSR, ASR,* or *ROR*
	expression1	is any positive absolute shift count in the range of 1..31.
	expression2	is any signed expression shiftable into an 8-bit value.

Examples:

```
TEQ R4,5              ; See if R4 contains the value 5.
BEQ Out               ; Jump if it does.
TEQP  R15,0xC0000000  ; Set N,Z. Clear C,V.
```

Variations: An *S* suffix is optional, and is always implied. If a *P* suffix is used, the bits 28-31 and 0-1 of the 32-bit ALU results are stored directly into the PSR bits, rather than the PSR being loaded from the ALU status itself.

Appendix A.35 TST - Set Condition Codes via AND

Test that any of the bits specified by the second operand are set in the source register. This is a "logical" instruction, so no inter-bit carry is permitted in the hardware, and an overflow condition is not possible. The V status bit is therefore not altered by the instruction.

Because the only purpose for this instruction is to perform a test, setting the condition codes on the result, the 'S' suffix (save status) is redundant and is automatically implied by CASM.

An 8-bit constant may be supplied as the second operand. The constant may consist of any 8-bit pattern in a 32-bit field, so long as it may be rotated to produce an 8-bit constant.

2

Intended Usage: Test for nonzero in selected bit field(s). It is a substitute for AND where no result other than the status needs to be retained.

Operational Function: Rn AND Op2 (result is not stored)

Flags Effected: N, Z, C

Syntax: TST{*condition*}{P} *Rn, Op2*

where	*condition*	is an optional 2-character condition code. See the Condition Code section.
	P	Force PSR loading, directly from 32-bit ALU result.
	Rn	is any valid register names, such as R0-R15, PC, SP, or LK.
	Op2	is second operand, and may have any of the following forms: *Rm shift Rs* *Rm shift expression1* *Rm RRX* *expression2*
	Rm	is any valid register names, as per *Rd* or *Rn* above, the operand value.
	Rs	is a register, per *Rd* above, containing a shift count in range of 1..32.
	shift	is any of: *ASL, LSL, LSR, ASR,* or *ROR*
	expression1	is any positive absolute shift count in the range of 1..31.
	expression2	is any signed expression shiftable into an 8-bit value.

Examples: Test R0 to see if bits 1 and 7 are both zero, regardless of the setting of any other bits.

```
TST     R0,0x82
BEQBoth_Zero
BNEEither_Set
```

Variations: An *S* suffix is optional, and is always implied. If a *P* suffix is used, the bits 28-31 of the 32-bit ALU results are stored directly into the PSR bits, rather than the PSR being loaded from the ALU status itself.