Apple Computer, Inc.

APRM / VL2340

32-BIT RISC MICROPROCESSOR

FEATURES

- 32-bit internal architecture
- 32-bit external data bus
- 64M-byte linear program address space
- 4G-byte linear data address space
- Bus timing optimized for standard DRAM usage with page mode operation
- 64+M-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
- 100-pin ceramic pin grid array (CPGA), quad plastic flatpack (QPFP), or MegaCell format in Apple VLSI Library

DESCRIPTION

The VL2340 Apple Proprietary RISC Machine (APRM) is a full 32-bit general-purpose microprocessor designed using reduced instruction set computer (RISC) methodologies. The APRM is an Apple Computer, Inc. owned version of the powerful Acorn RISC Machine (ARM). Applications in which the processor is useful include laser printers, graphics engines, and any other systems requiring fast real-time response to external interrupt sources and high processing throughput.

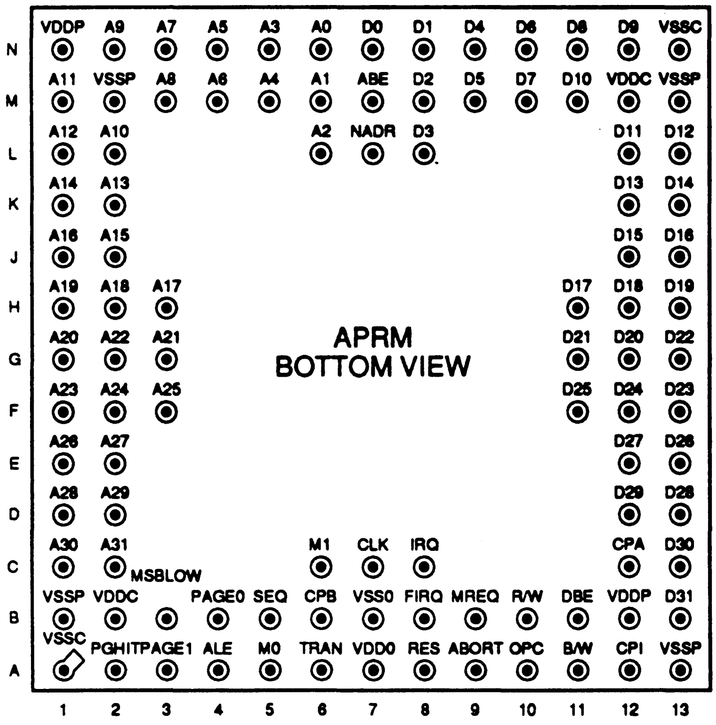
The APRM features a 32-bit data bus, 27 registers of 32 bits each, a load-store architecture, a partially overlapping register set, 1.5 µs worst-case interrupt latency (at 16MHz operation), conditional instruction execution, a 26-bit linear program address space, a 32-bit linear data address space, and an average instruction execution rate of from 12-to-14 million instructions per second (MIPS at 16MHz). 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 directly supported are 8-bit bytes and 32-bit words, with efficient compiler generated support for 16-bit values. The APRM includes support for unaligned access to 32-bit data words.

Using a load-store architecture simplifies the execution unit of the processor, since 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 processsor 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

CERAMIC PIN GRID ARRAY



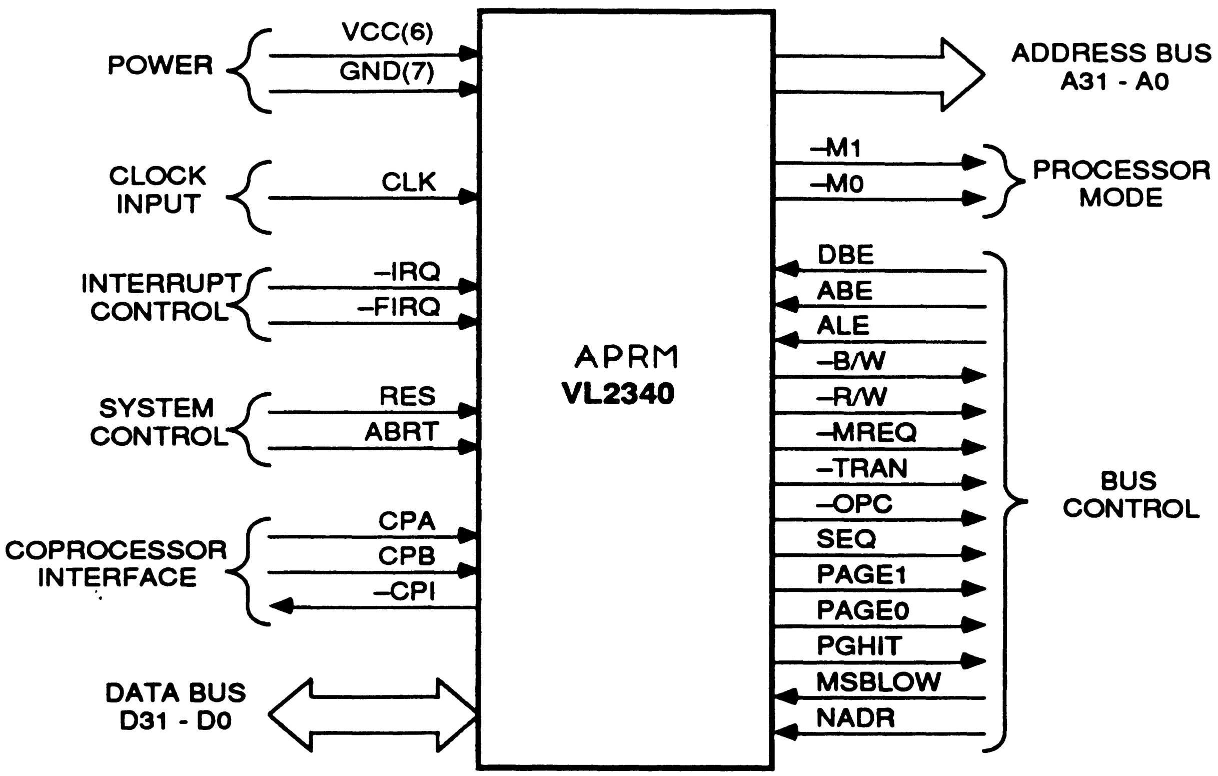
N: VDDP A9 A7 A5 A3 A0 D0 D1 D4 D6 D8 D9 VSSC
M: A11 VSSP A8 A6 A4 A1 ABE D2 D5 D7 D10 VDDC VSSP
L: A12 A10 A2 NADR D3 D11 D12
K: A14 A13 D13 D14
J: A16 A15 D15 D16
H: A19 A18 A17 D17 D18 D19
G: A20 A22 A21 D21 D20 D22
F: A23 A24 A25 D25 D24 D23
E: A26 A27 D27 D26
D: A28 A29 D29 D28
C: A30 A31 MSBLOW M1 CLK IRQ CPA D30
B: VSSP VDDC PAGE0 SEQ CPB VSS0 FIRQ MREQ R/W DBE VDDP D31
A: VSSC PGHIT PAGE1 ALE M0 TRAN VDD0 RES ABORT OPC B/W CPI VSSP
1 2 3 4 5 6 7 8 9 10 11 12 13
APRM
BOTTOM VIEW

ORDER INFORMATION

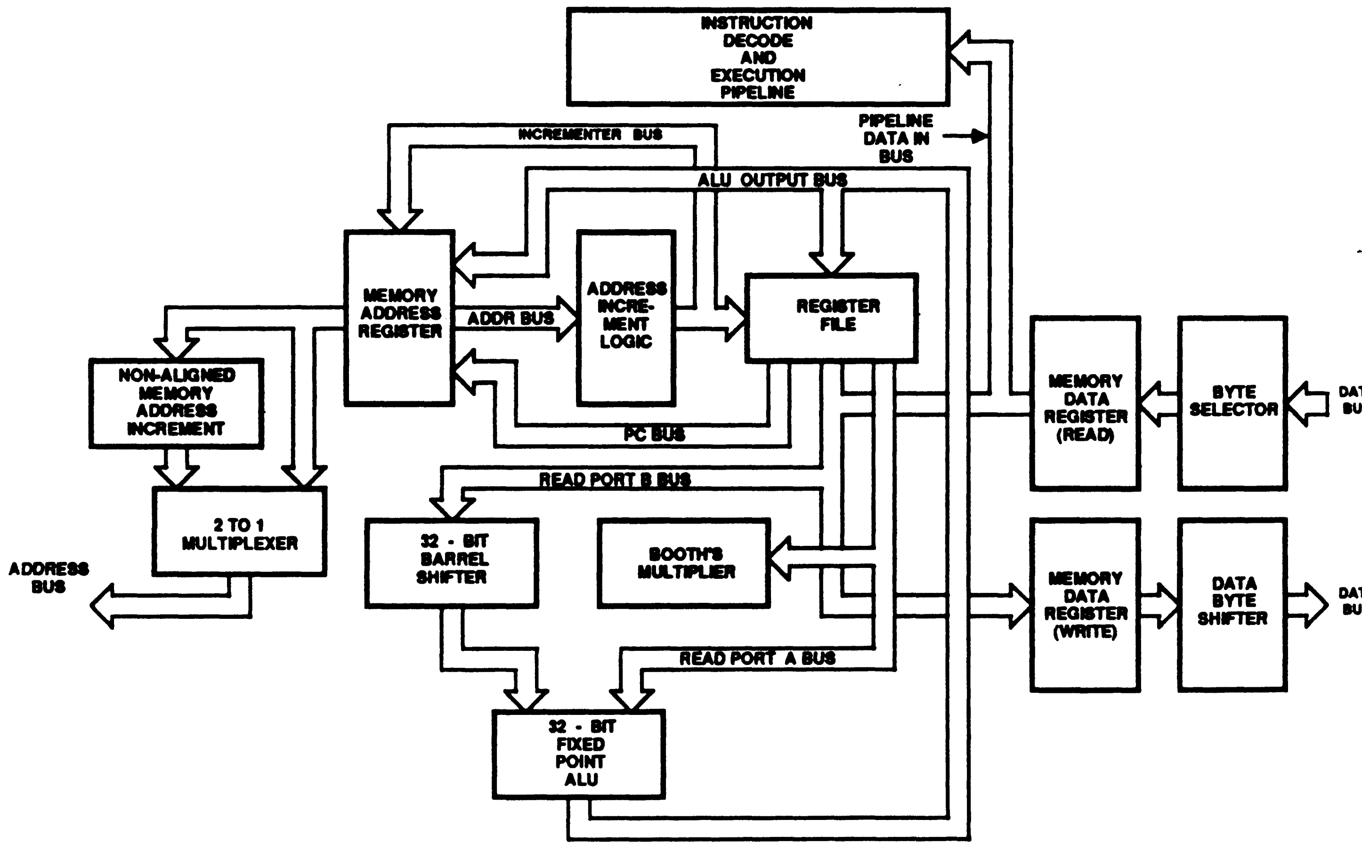
Part Number	Clock Frequency	Package
VL2340-10GC	16 MHz see tCK Min.	Ceramic Pin Grid Array (CPGA)
VL2340-10FC		Plastic Quad Flatpack (PQFP)

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

FUNCTIONAL PIN DIAGRAM



BLOCK DIAGRAM



SIGNAL DESCRIPTIONS

Signal Name	Pin Number(1)	Signal Description
CLK	C7	Processor Clock Input - This input provides the clock to the circuit. The Ø2 internal clock is in phase with this input and the Ø1 internal clock is the nonoverlapping inverse.
–IRQ	C8	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	B8	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	A8	Reset Input - This is the reset signal for the processor. While active, the processor executes no-ops until the signal goes inactive from which point execution starts at the Reset Vector location. This signal is active high.
ABORT	A9	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 to control accesses for protection. The abort signal is active-high.
D31 - D0	See Package	Data31 - Data0 - This is the 32 bit bidirectional data bus used to transfer data to and from the memory. These lines are tri-state and active-high.
DBE	B11	Data Bus Enable Input - This is the asynchronous tri-state control signal for controlling the drivers of the data bus. When asserted the data bus is enabled. This signal is active high.
–B/W	A11	Not Byte / Word Output - This "early warning" (note 2) signal indicates to the memory system that the current fetch is a byte fetch rather than a word fetch. It is asserted during the last portion of the cycle preceding the cycle that requires a byte fetch. When asserted (low) the memory system should deal with bytes. It is active-low. While RES is active -B/W will remain high.
–M1, –M0	C6, A5	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. While RES is active M0 and M1 retain their previous states.

–M1	–M0	MODE
0	0	Supervisor
0	1	FIRQ
1	0	IRQ
1	1	USER

Signal Name	Pin Number(1)	Signal Description
A31 - A0	See Package	Address 31 - Address 0 Outputs - These are the 31 address lines. A0 and A1 are byte addresses and should be ignored during opcode fetech cycles. During opcode fetches, the current mode value may appear on these signals. The address lines are tri-state and active-high.
ABE	M7	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	A4	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.
–R/W	B10	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 an "early warning" (note 2) signal and is active low. While RES is active -R/W will remain low.

SIGNAL DESCRIPTIONS

Signal Name	Pin Number	Signal Description
–MREQ	B9	Next Memory Cycle Start Output - This is an "early warning" (note 2) indicator that is asserted before the processor will start a memory cycle during the next clock phase. This signal is active low. While RES is active -MREQ will remain low.
–TRAN	A6	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	A10	Instruction Fetch Output - This "early warning" (note 2) signal when asserted indicates that the current bus cycle is an instruction fetch. This signal is active-low. While RES is active -OPC will remain low.
SEQ	B5	Next Address Sequential Output - This "early warning" (note 2) 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. While RES is active SEQ will remain high.
–CPI	A12	Coprocessor Instruction (CMOS level output) - When the APRM executes a coprocessor instruction, this output is driven low and the processor will wait for a responsefrom an attached coprocessor device. The action taken is dependent upon the coprocessor response signalled on the CPA and CPB inputs.
CPB	B6	Coprocessor Busy (TTL level input) - An attached coprocessor that is capable of performing the operation which the APRM 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 APRM samples this signal on the falling edge of the Ø1 clock while the –CPI is active (low).
CPA	C12	Coprocessor Absent (TTL level input) - A coprocessor capable of executing the operation currently requested by the APRM (–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.
NADR	L7	Next address (CMOS level input) - When asserted selects the current address plus four for non-aligned memory reads. Also, the appropriate data bytes are latched from the first word from memory. This signal is active high.
PAGE1, PAGE0	A3, B4	Page size (CMOS level inputs) - These two signals are decoded to determine the DRAM page size of the memories used.
MSBLOW	B3	Most Significant byte low (CMOS input) - When asserted this input forces the upper eight address outputs low. This input is active high.
PGHIT	A2	Page hit (CMOS level output) - This early warning signal indicates that the current memory operation is the last address in the active page. This output is active low.

P1	P0	Page Size
0	0	256 words
0	1	512 words
1	0	1024 words
1	1	2048 words

NOTES:
1. Pin numbers are for ceramic pin grid array package only.
2. "Early warning" signals are asserted during the last portion of the cycle preceding the cycle to which they apply.

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. An 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 user pays for the unused silicon required to implement these instructions. He pays 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 RISCprocessors, for good reason. The term MIPS is 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 using compiled QuickDraw routines approximate real conditions. Measurements are based on pixel per second generation, bit-field extraction rates, etc. Running well below its maxium specified clock rates, the APRM, running compiled code, will outperform all popular, commercially available microprocessors running hand crafted code.

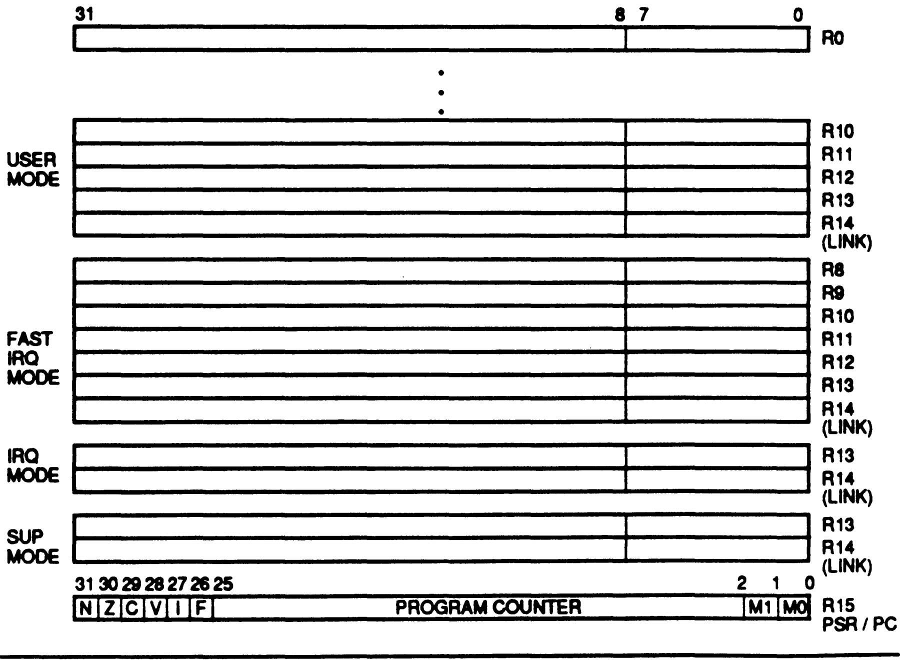
An important parameter to keep constant when benchmarking processors is the memory access times, since not all processors will meet performance claims when working with commodity memories.

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 a one that benchmarks high while using a relatively low clock frequency and a small number of clocks per instruction executed. The APRM possesses these characteristics, giving it the best future evolution path to exploit advances in process technology.

PROGRAMMING MODEL

The APRM contains a large, partially overlapping set of twenty-seven 32-bit registers, although the programmer can access only sixteen registers in any mode of operation. Fifteen of the registers are general purpose; with the remaining twelve 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 APRM. Registers R0-to-R13 are accessible from the user mode for any purpose. The fourteenth 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

FIGURE 1. VL2340 REGISTER MODEL



register at other times. The least significant two bits of the processor status word (PSW) define the current mode of operation.

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 R10-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-R9 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.

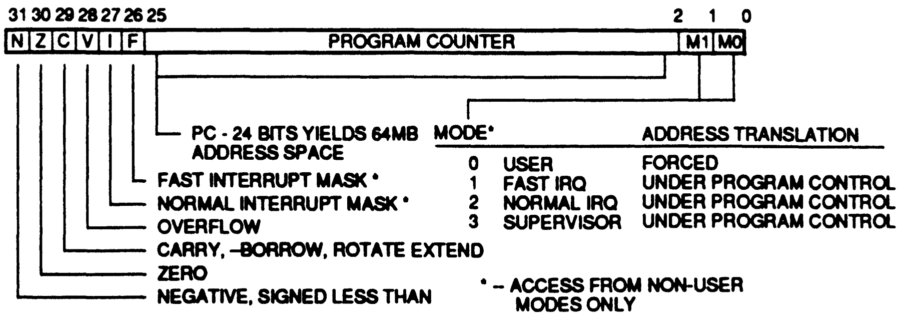
Two registers are dedicated to the supervisor mode and overlay user mode registers R13 and R14 when a supervisor mode switch is made using a software interrupt (SWI) instruction. Operation of these two registers is the same as previously discussed.

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 APRM makes a distinction between user and supervisor modes: the user executes at

FIGURE 2. PROCESSOR STATUS REGISTER


31 30 29 28 27 26 25
2 1 0
N Z C V I F
PROGRAM COUNTER
M1 M0
PC - 24 BITS YIELDS 64MB ADDRESS SPACE
FAST INTERRUPT MASK *
NORMAL INTERRUPT MASK *
OVERFLOW
CARRY, ~BORROW, ROTATE EXTEND
ZERO
NEGATIVE, SIGNED LESS THAN
MODE*
ADDRESS TRANSLATION
0 USER FORCED
1 FAST IRQ UNDER PROGRAM CONTROL
2 NORMAL IRQ UNDER PROGRAM CONTROL
3 SUPERVISOR UNDER PROGRAM CONTROL
* – ACCESS FROM NON-USER MODES ONLY

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.

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 supervisor mode 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 sixteen allowable combinations of the condition code bits are shown in Table 1. These combinations are used for all conditional instruction execution since a conditional branch is nothing more than a jump instruction with conditional execution.

EXCEPTIONS

The APRM 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 APRM is very short because the instruction execution time is typically two clocks, with a maximum of eighteeen (for a load-multiple instruction, loading sixteen registers). Once the processor recognizes an interrupt is pending, the time to begin processing is four clocks making a total worst-case interrupt latency of 22.5 clocks.

In addition to interrupts, five other types of exceptions are supported by the processor. These are data-fetch cycle aborts, instruction-fetch cycle aborts, software interrupts, undefined instruction traps and reset.

The APRM supports a 32-bit linear address space allowing a total of 4G-bytes of physical memory. The total program space is limited to 26-bits of address space, for a total of 64M-bytes used by program execution.

If the abort signal is asserted by the

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

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 instructions 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 a 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.

TABLE 2. EXCEPTION VECTOR MAP

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

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, undefined instruction trap, prefetch or data abort, the IRQ is disabled and the FIRQ unchanged.

INSTRUCTION SET

The APRM 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 at 10 MHz).

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 register, although some bits in R15 can only be changed in particular modes. The source operands can have two

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.

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 thru 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 rewritten. 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 8-bits of the register and zero-extended to a full word. For byte writes the lower 8-bits of the register are replicated onto all four bytes of the data bus. The memory controller should be designed such that only the addressed byte is updated in the memory.

Words are written into the address space as most-significant byte first. That is, the byte at the lowest address will be found left justified in a register and its memory location "BigEndian" fashion. See Appendix 1 for details of word and byte registration.

The APRM 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 first 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 program 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 APRM is offered in two packages, a 100-pin ceramic pin grid array (CPGA) package and a 100-pin quad plastic flatpack (QPFP).

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**		

S implies a sequential cycle.

N implies a non sequential cycle.

* - 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.

EXAMPLES OF THE INSTRUCTION SET

The following examples illustrate methods by which basic APRM 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 ASL 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.

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, refiling the instruction pipeline, and if the branch is with link, R14 is modified (four 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.

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) are 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). The memory interface may be designed such that this internal cycle can be configured to merge with the next cycle into a single memory N-cycle.

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 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 bits, 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).

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. For details of registration during the load operation see Appendix 1.

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.

See Appendix 1 for memory registration details.

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							

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, 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, though 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).

UNDEFINED INSTRUCTIONS AND COPROCESSOR ABSENT

When a Co-Processor detects a Co-Processor 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 14. 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 15. 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 CYCLE OPERATIONS (Cont.)

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 16. 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 Coprocessor To VL86C010 Register	1 S + b I + 1 C
Move From VL86C010 To Coprocessor 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 incremental cycle.

b is the number of cycles spent in the Co-Processor 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	VL2340 Min.	VL2340 Typ.	VL2340 Max.	Units	Conditions
tCK	Clock Period	45	–	10000	ns	
tCKL	Clock Period Low	21	–	10000	ns	
tCKH	Clock Period High	14	–	10000	ns	
tABE	Address Bus Enable	–	–	14	ns	
tABZ	Address Bus Disable	–	–	14	ns	
tALE	Address Latch Fall-Through	–	–	14	ns	
tALEL	ALE Low Time	–	–	10000	ns	See Note 1
tADDRS	CLK Rising Edge To Address Valid Delay	–	–	25	ns	
tADDRN	CLK Falling Edge To Address Valid Delay	–	–	55	ns	
tADRNA	NADR To Address Valid Delay	5	–	18	ns	
tADRMS	MSBLOW To Address Valid Delay	5	–	11	ns	
tAH	Address Bus Hold Time	6	–	–	ns	
tDBE	Data Bus Enable Time	–	–	22	ns	
tDBZ	Data Bus Disable Time	–	–	22	ns	
tDOUT	Data Bus Output Delay	–	–	27	ns	
tDOH	Data Bus Hold Time	6	–	–	ns	
tDIS	Data In Setup Time To CLK	4	–	–	ns	
tDIH	Data In Hold Time To CLK	8	–	–	ns	
tDISN	Data In Setup Time To NADR	2	–	–	ns	
tDIHN	Data In Hold Time To NADR	3	–	–	ns	
tABTS	ABORT Setup Time	18	–	–	ns	
tABTH	ABORT Hold Time	6	–	–	ns	
tIRS	Interrupt Setup Time	4	–	–	ns	See Note 2
tRWD	CLK To –R/W Valid	–	–	31	ns	
tRWH	–R/W Hold Time	5	–	–	ns	
tMSD	CLK To –MREQ And SEQ Delay	–	–	32	ns	
tMSH	–MREQ And SEQ Hold Time	6	–	–	ns	
tBWD	CLK To –B/W Valid	–	–	26	ns	
tBWH	–B/W Hold Time	5	–	–	ns	
tMDD	CLK To -M1, - M0 Valid	–	–	22	ns	
tMDH	M1 - M0 Hold Time	6	–	–	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 interrupt and reset inputs may be asynchronous. This time will guarantee that the interrupt request is latched during this cycle.

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

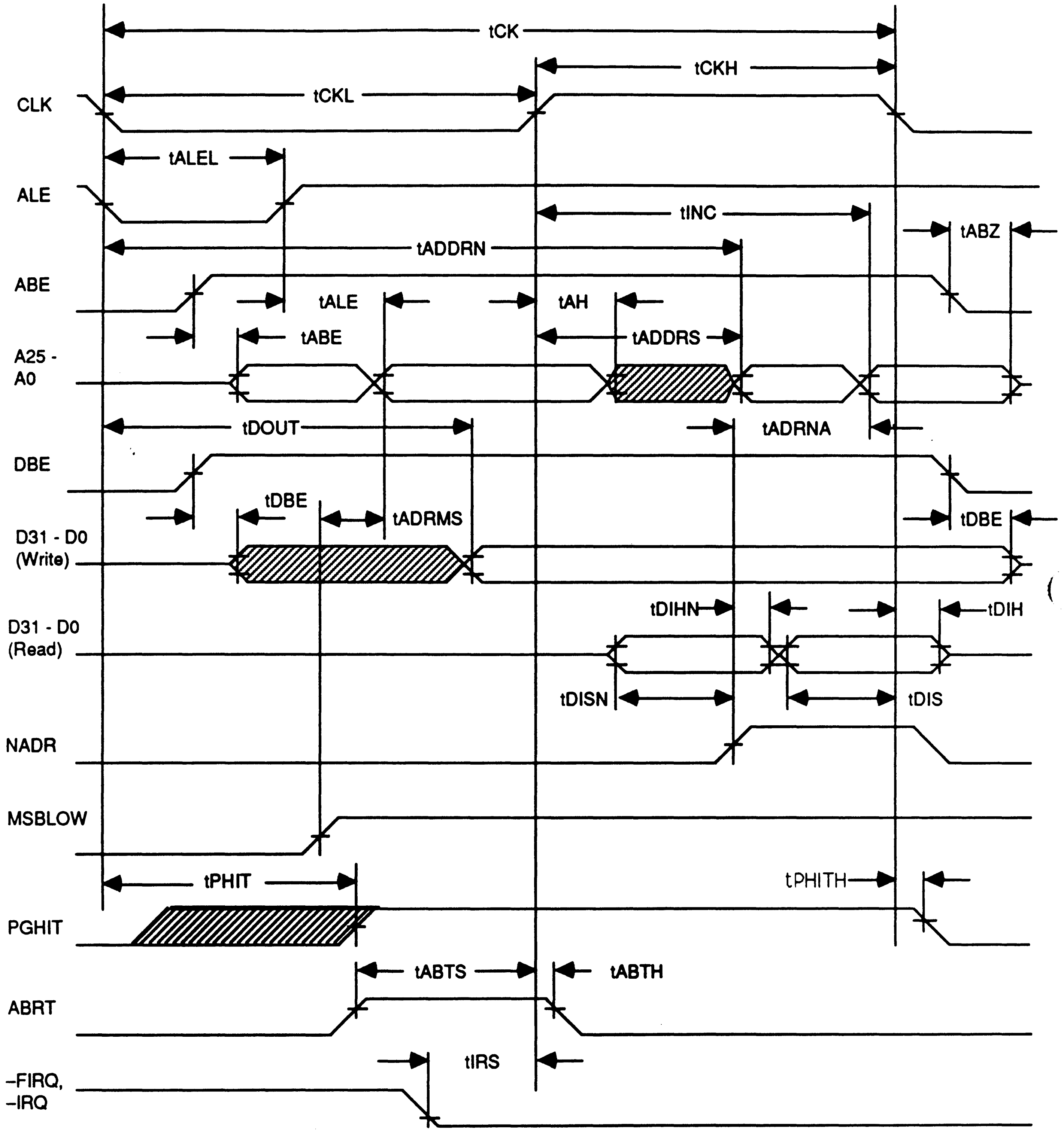
		VL2340				
Symbol	**Parameter**	**Min.**	**Typ.**	**Max.**	**Units**	**Conditions**
tOPCD	CLK To –OPC Valid	–	–	23	ns	
tOPCH	–OPC Hold Time	5	–	–	ns	
tTRMD	CLK To –TRAN Valid	–	–	22	ns	
tTRMH	–TRAN Hold Time	6	–	–	ns	
tTRDD	CLK To –TRAN Valid	–	–	29	ns	See Note 1
tTRDH	–TRAN Hold Time	5	–	–	ns	
tCPS	CPA, –CPB Setup Time	18	–	–	ns	
tCPH	CPA, –CPB Hold Time	6	–	–	ns	
tCPI	CLK To –CPI Delay	–	–	22	ns	
tCPIH	–CPI Hold Time	4	–	–	ns	
tPHIT	CLK To PGHIT Delay	-	–	19	ns	
tPHITH	PGHIT Hold Time	7	-	-	ns	
tINC	CLK To Incremented Address Delay	12	–	65	ns	

Notes:

1. –TRAN will only change during CLK high as the result of a forced translation single data transfer operation while in the User mode. Otherwise it will change during CLK low when the mode change to/from User mode occurs.

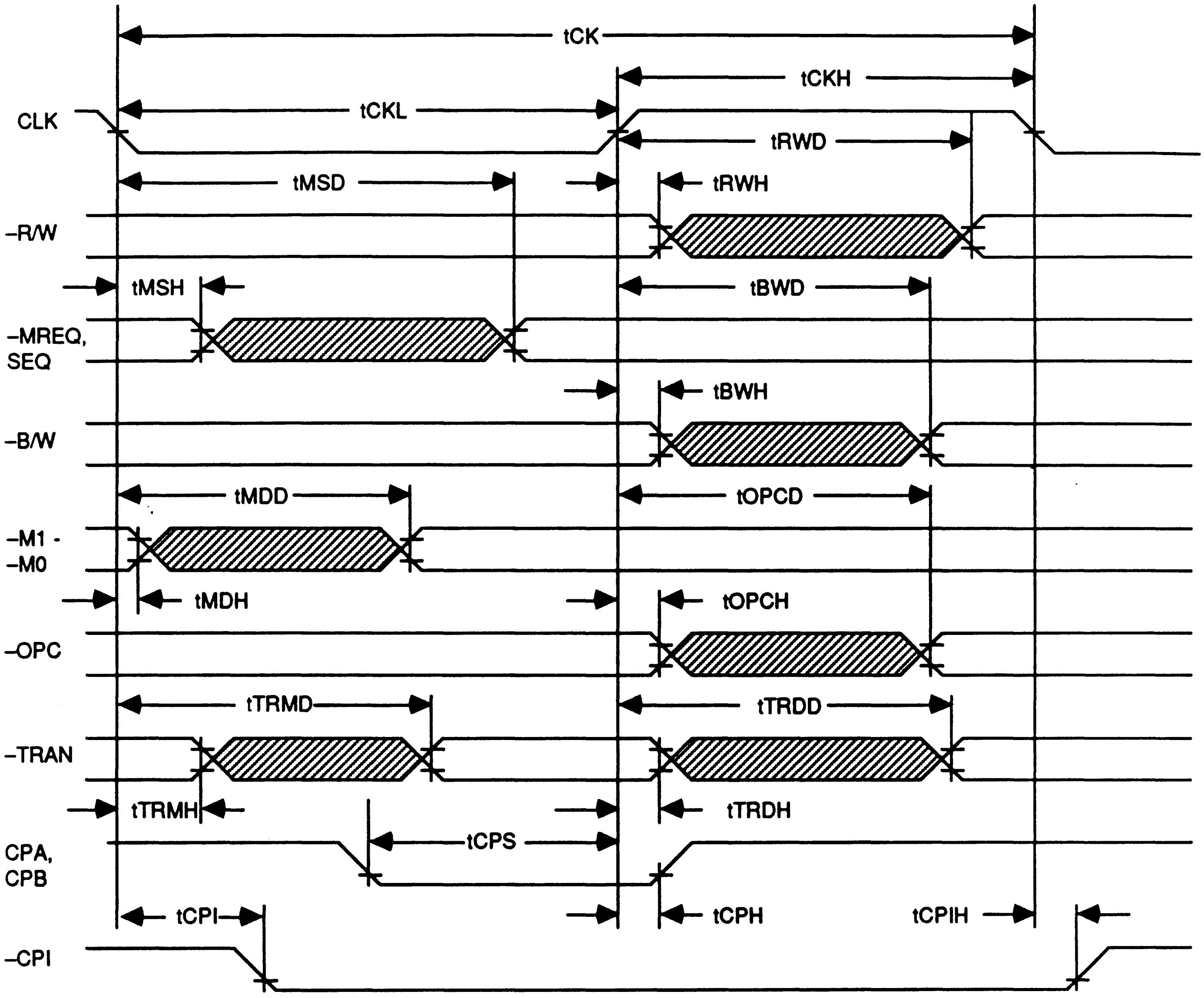
TIMING DIAGRAMS

PROCESSOR DATA BUS



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

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.

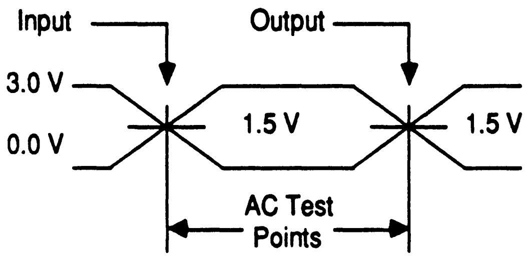
DC CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ± 5%

Symbol	Parameter		Min	Typ	Max	Unit	Conditions
VOHT	Output High Voltage, TTL-DATABUS		VCC–0.75	–	VCC	V	IOH = –5.0 mA
VOLT	Output High 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

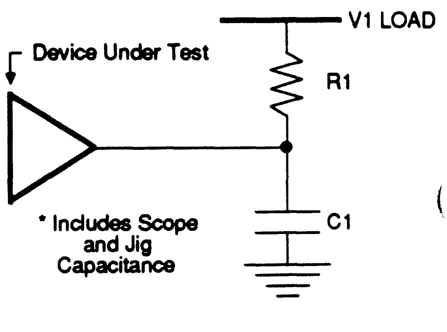
Symbol	Parameter	Min	Max	Unit	Conditions
CI	ClockInput Capacitance (Ø1, Ø2)	–	15	pF	VIN = 0 V (Note2)
	Other Input Capacitance	–	5	pF	VIN = 0 V (Note2)
CO	Output Capacitance	–	8	pF	VOUT = 0 V (Note 2)

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 APRM has a 32-bit data bus and a 32-bit address bus although only 26-bits (64M-bytes) may be used for program space. The processor supports two data types, eight bit bytes and 32-bit words. 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 APRM 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 most-significant-first (MSB first) order. This means that the most significant bytes of a 32-bit word occupies the lowest byte address. The byte address values are illustrated in Figure 5.

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 16.

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 6 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.

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).

FIGURE 5. BYTE SIGNIFICANCE OF APRM

31 24	23 16	15 8	7 0	Word Address Value
Byte Addr. 0000	Byte Addr. 0001	Byte Addr. 0002	Byte Addr. 0003	0000
Byte Addr. 0004	Byte Addr. 0005	Byte Addr. 0006	Byte Addr. 0007	0001

TABLE 17. REGISTER ORGANIZATION

					Typical Use
R0	General				General Usage
R1	General				
R2	General				
R3	General				
R4	General				
R5	General				
R6	General				
R7	General				
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

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.

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 reentrant 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, so that 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 18. The return-from-interrupt sequence is also defined there. This will resume execution of the interrupted code sequence, and restore the original processor state.

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.

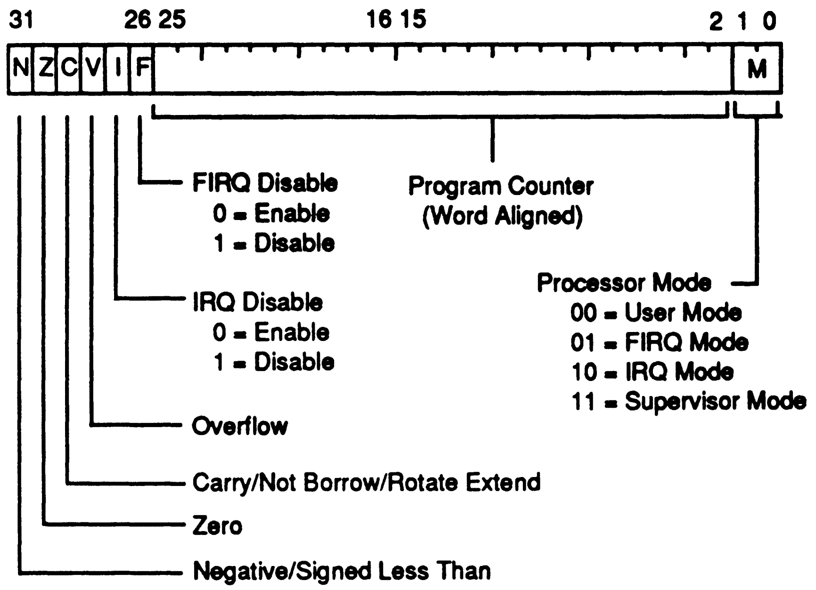
The impact upon execution of an IRQ interrupt is defined in Table 18. The return-from-interrupt sequence is also defined there. This will resume execution of the interrupted code sequence, restore the original processor state, and reenable the IRQ interrupt.

Abort - The ABORT signal comes from an external memory management system, and indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a 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 APRM 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 writeback 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

FIGURE 6. PROGRAM COUNTER AND PROCESSOR STATUS REGISTER



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 18.

The return from Prefetch Abort defined in the Figure 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 Figure.

The abort mechanism allows a demand paged virtual memory system to be implemented when a suitable memory management unit is available in the system. 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 18, as is the method of returning. The indicated return method will return to the instruction following the SWI.

Undefined Instruction Trap - When the APRM 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 APRM 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 APRM will respond as defined in Table 18. The return from this trap (after performing a suitable emulation of the required function), defined in the Figure 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 18. 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 below:

Address	Function
0000000	Reset
0000004	Undefined instruction
0000008	Software interrupt
000000C	Abort (prefetch)
0000010	Abort (data)
0000014	Unused
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

TABLE 18. EXCEPTION TRAP CONSIDERATIONS

Trap Type	CPU Trap Activity	Program Return Sequence
Reset	1. Save R15 in R14 (SVC). 2. Force M1:0 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 M1:0 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 M1:0 to SVC mode, and set I status bit in the PC. 3. Force PC to 0x000010.	Prefetch Abort: SUBS PC, R14,4 ; SVC's R14. Data Abort: MOVS PC, R14,8 ; SVC's R14.
IRQ	1. Save R15 in R14 (IRQ). 2. Force M1:0 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 M1:0 to FIRQ mode, and set I status bit in the PC. 3. Force PC to 0x00001C.	SUBS PC, R14,4 ; FIRQ's R14.

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

1) Reset (highest priority)
2) Data aborts
3) FIRQ
4) IRQ
5) Prefetch abort
6) Undefined Instruction and SWIs (lowest priority)

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

If a data abort occurs at the same time as a FIRQ, and FIRQs are enabled (i.e., the F flag in the PSR is clear), the processor will enter the data abort handler and then immediately proceed to the FIRQ vector. A normal return from FIRQ will cause the data abort handler to resume execution. Placing 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 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, therefore, 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 MMU, 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 count delay entry into the IRQ handling routine for an arbitrary length of time.

The minimum latency 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.

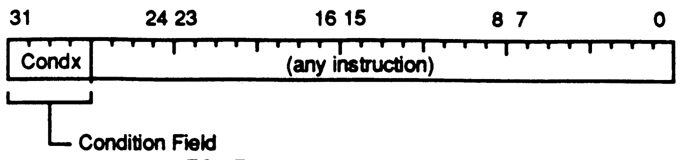
INSTRUCTION SET

All APRM 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 7. CONDITION FIELD



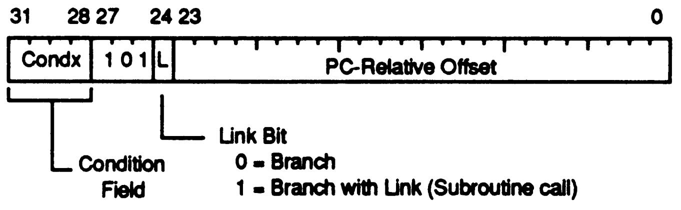
0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or 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

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 program 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 8. 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:

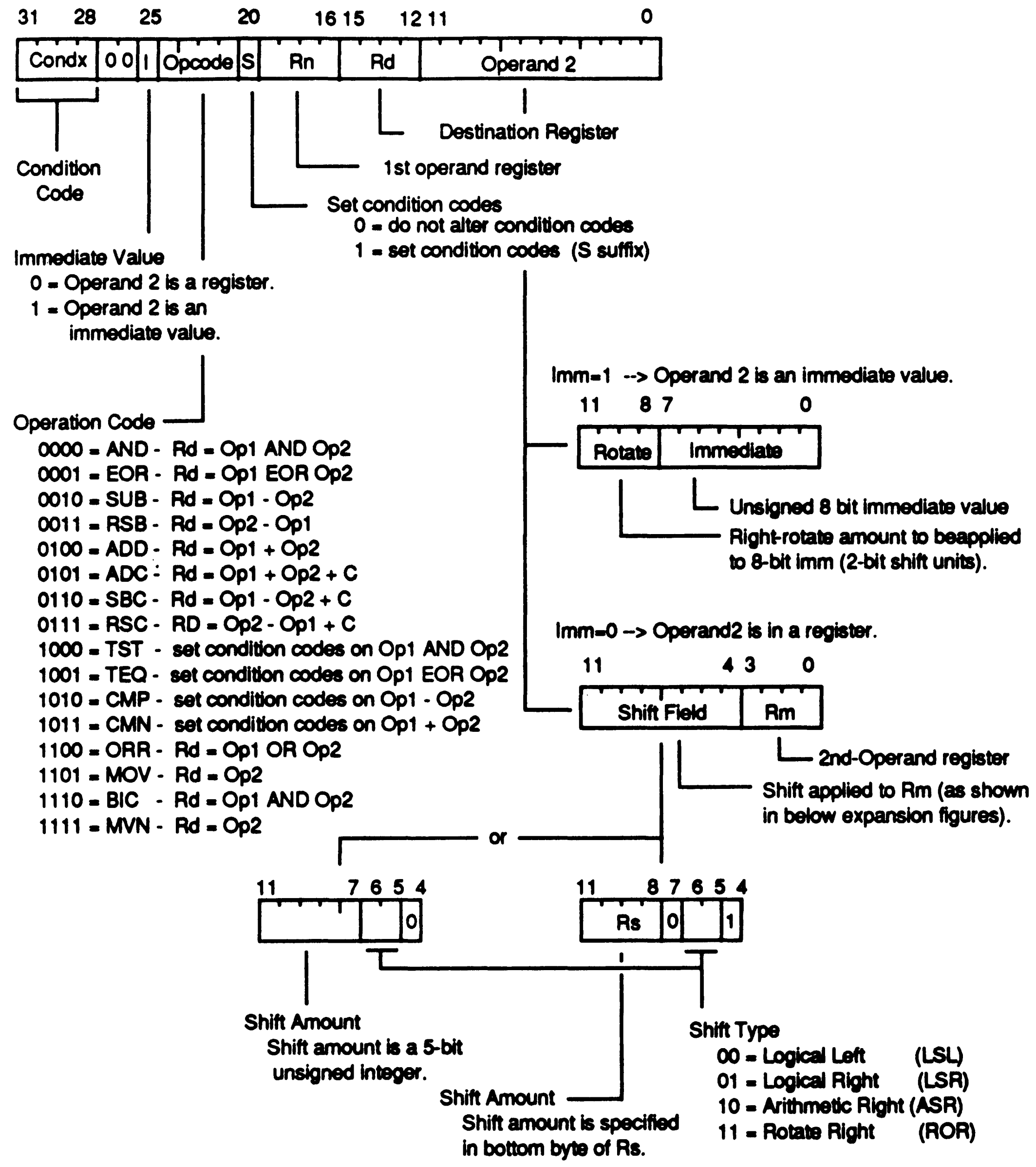
```
Here    BAL     Here          ; Assembles to EAFFFFFE. (Note effect of PC offset)
        B       There         ; Always condition used as default
        CMP     R1,0          ; Compare register one with zero, and branch to Fred if
        BEQ      Fred         ; register one was zero.  Else continue 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 INSTRUCTION

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 eight-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 9. ALU INSTRUCTION TYPES


31 28 25 20 16 15 12 11 0
Condx
0 0
I
Opcode
S
Rn
Rd
Operand 2
Destination Register
1st operand register
Condition Code
Set condition codes
0 = do not alter condition codes
1 = set condition codes (S suffix)
Immediate Value
0 = Operand 2 is a register.
1 = Operand 2 is an immediate value.
Operation Code
Imm=1 --> Operand 2 is an immediate value.
11 8 7 0
Rotate
Immediate
Unsigned 8 bit immediate value
Right-rotate amount to beapplied to 8-bit imm (2-bit shift units).
Imm=0 --> Operand2 is in a register.
11 4 3 0
Shift Field
Rm
2nd-Operand register
Shift applied to Rm (as shown in below expansion figures).
or
11 7 6 5 4
0
11 8 7 6 5 4
Rs
0
1
Shift Amount
Shift amount is a 5-bit unsigned integer.
Shift Amount
Shift amount is specified in bottom byte of Rs.
Shift Type
00 = Logical Left (LSL)
01 = Logical Right (LSR)
10 = Arithmetic Right (ASR)
11 = Rotate Right (ROR)

Operation Code

0000 = AND -	Rd = Op1 AND Op2
0001 = EOR -	Rd = Op1 EOR Op2
0010 = SUB -	Rd = Op1 - Op2
0011 = RSB -	Rd = Op2 - Op1
0100 = ADD -	Rd = Op1 + Op2
0101 = ADC -	Rd = Op1 + Op2 + C
0110 = SBC -	Rd = Op1 - Op2 + C
0111 = RSC -	RD = Op2 - Op1 + C
1000 = TST -	set condition codes on Op1 AND Op2
1001 = TEQ -	set condition codes on Op1 EOR Op2
1010 = CMP -	set condition codes on Op1 - Op2
1011 = CMN -	set condition codes on Op1 + Op2
1100 = ORR -	Rd = Op1 OR Op2
1101 = MOV -	Rd = Op2
1110 = BIC -	Rd = Op1 AND Op2
1111 = MVN -	Rd = Op2

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)

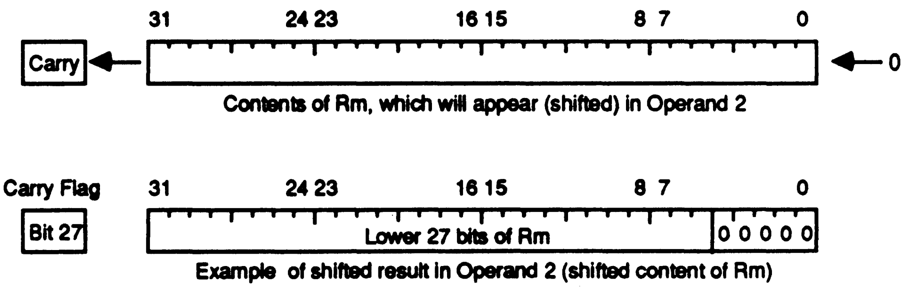
PSR Flags - The operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15) the V flag in the PSR will 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 five-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:

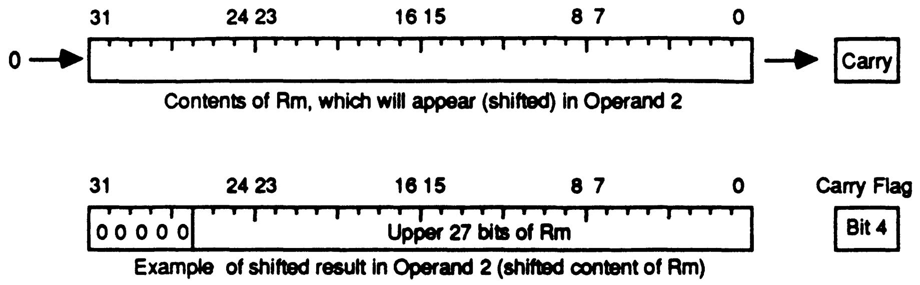
FIGURE 10. LOGICAL SHIFT LEFT (LSL)


31
24 23
16 15
8 7
0
Carry
0
Contents of Rm, which will appear (shifted) in Operand 2
Carry Flag
Bit 27
Lower 27 bits of Rm
0 0 0 0 0
Example of shifted result in Operand 2 (shifted content of Rm)

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:

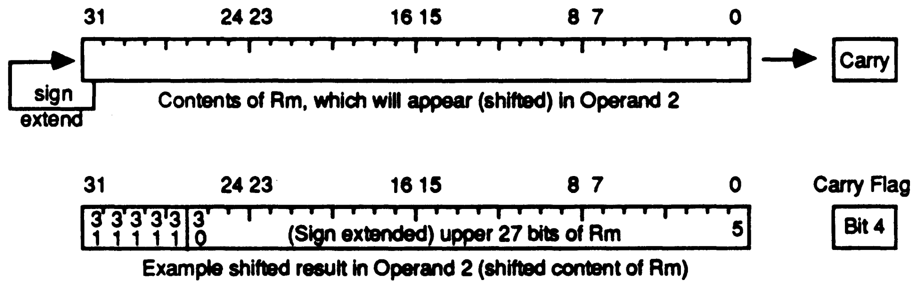
FIGURE 11. LOGICAL SHIFT RIGHT (LSR)


31
24 23
16 15
8 7
0
0
Carry
Contents of Rm, which will appear (shifted) in Operand 2
Carry Flag
0 0 0 0 0
Upper 27 bits of Rm
Bit 4
Example of shifted result in Operand 2 (shifted content of Rm)

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, and 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:

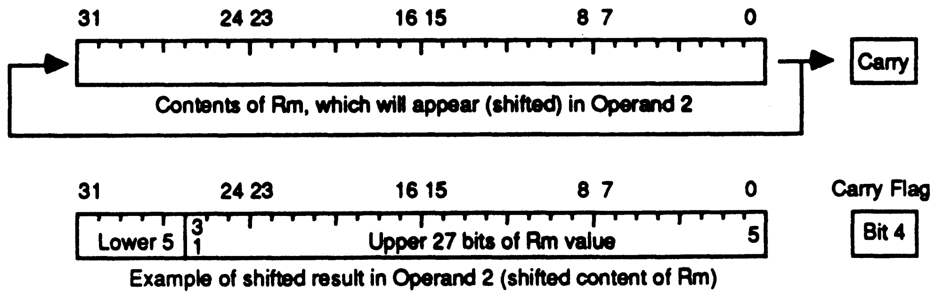
FIGURE 12. ARITHMETIC SHIFT RIGHT (ASR)


31
24 23
16 15
8 7
0
Carry
sign extend
Contents of Rm, which will appear (shifted) in Operand 2
Carry Flag
31 31 31 31 31
30
(Sign extended) upper 27 bits of Rm
5
Bit 4
Example shifted result in Operand 2 (shifted content of Rm)

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.

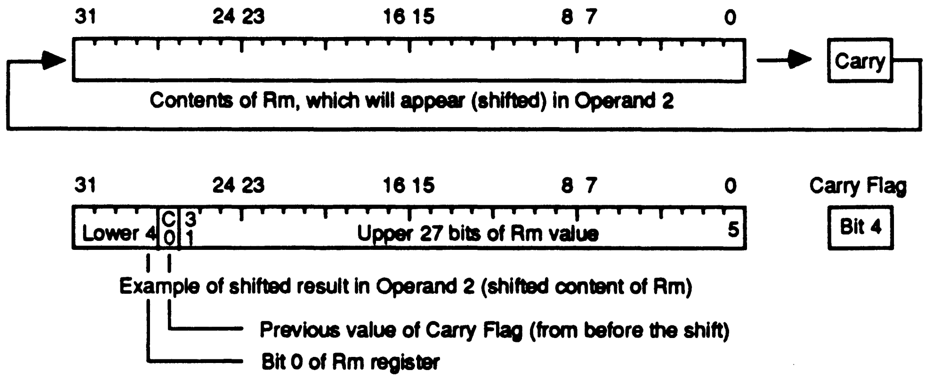
Rotate Right (ROR) operations reuse the bits which "overshoot" in a logical shift right operation, by wrapping them around at the high end of the result. For example, the effect of a ROR 5 is:

FIGURE 13. ROTATE RIGHT (ROR)


31
24 23
16 15
8 7
0
Carry
Contents of Rm, which will appear (shifted) in Operand 2
Carry Flag
Lower 5
31
Upper 27 bits of Rm value
5
Bit 4
Example of shifted result in Operand 2 (shifted content of Rm)

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 14. ROTATE RIGHT EXTENDED (RRX)


31
24 23
16 15
8 7
0
Carry
Contents of Rm, which will appear (shifted) in Operand 2
Carry Flag
Lower 4
C 0
3 1
Upper 27 bits of Rm value
5
Bit 4
Example of shifted result in Operand 2 (shifted content of Rm)
Previous value of Carry Flag (from before the shift)
Bit 0 of Rm register

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 filed with and carry out equal to bit 31 of Rm.
ROR by 32	Result equal to Rm, 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 four-bit unsigned integer which specifies a shift operation on the eight bit immediate value. The immediate value is zero extended to 32-bits, and then subject to a rotate right by twice the value in the rotate field. This enables many command constants to be generated, for example all powers of two. Another example is that the eight 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.

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>

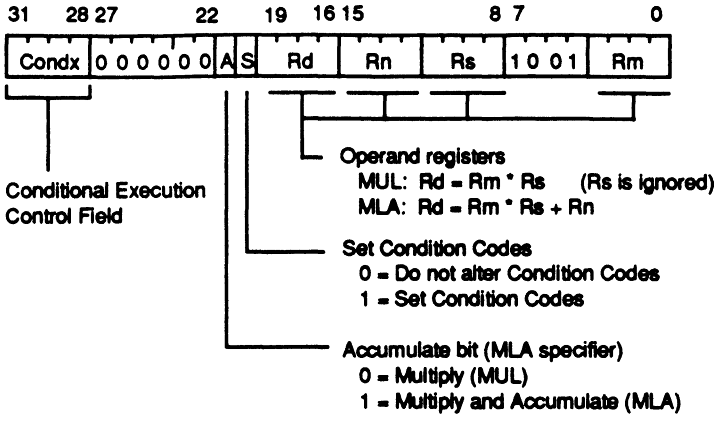
where	*Op2*	Is *Rm{,<shift>}* or, *<expression>*
	cond	Two-character condition mnemonic, see Condition Code section.
	S	Set condition codes if S present (implied for CMP, CMN, TEQ, TST).
	P	Make Rd = R15 in instructions where Rd is not specified, otherwise Rd will default to R0. (Used for changing the 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 15. MULTIPLY, AND MULTIPLY-ACCUMULATE (MUL, MLA)


31 28 27 22 19 16 15 8 7 0
Condx
0 0 0 0 0 0
A
S
Rd
Rn
Rs
1 0 0 1
Rm
Conditional Execution Control Field
Operand registers
MUL: Rd = Rm * Rs (Rs is ignored)
MLA: Rd = Rm * Rs + Rn
Set Condition Codes
0 = Do not alter Condition Codes
1 = Set Condition Codes
Accumulate bit (MLA specifier)
0 = Multiply (MUL)
1 = Multiply and Accumulate (MLA)

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

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

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

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

Operand restrictions - Due to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are 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 also not be R15, as 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 use 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.

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.

A byte load (LDRB) expects the data on bits D31 to D24 if the supplied address is on a word boundary, on bits D23 to D16 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.

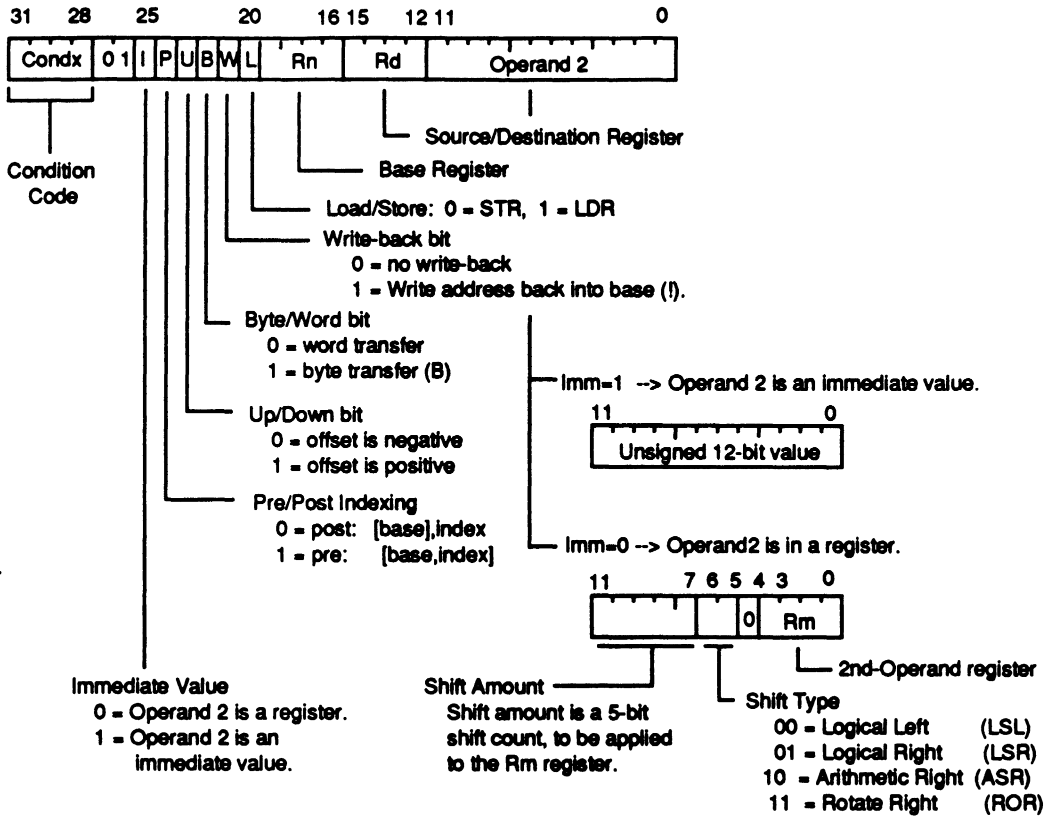
A word load (LDR) will normally generate a word aligned address but may also generate a non-word-aligned address. An address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte position in the data occupies bits D31 to D24. Reference Appendix 1, Table 1.

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

FIGURE 16. LOAD/STORE VALUE FROM MEMORY (LDR,STR)



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

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 on 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.

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-i 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]{!}	No offset is added to base address 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.

where	*expression*	A signed 13-bit expression (including the sign).
	Rm, Rn	A 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 readahead.
	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):

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.

```
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.
```

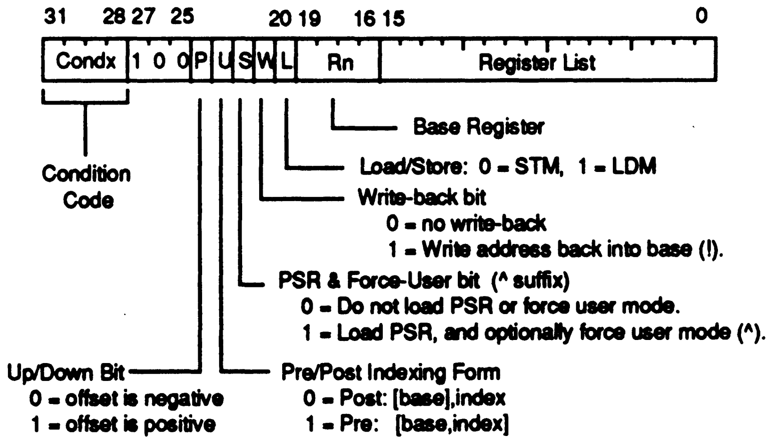
Examples (Post-Index):
In each of these examples, the effective offset is added to the Rn (base pointer) register after using the Rn register as the effective address. Rn is then updated unconditionally, regardless of any ! suffix.

```
        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 17. 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 18 and 19.

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. 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 a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. An LDM will always overwrite the updated base if the base is in the list.

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.

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.

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 writeback, Rn would remain at 0x1000.

Figure 19 illustrates decrementing "pushes" to the stack based upon Rn.

FIGURE 17. INCREMENTING INDEX

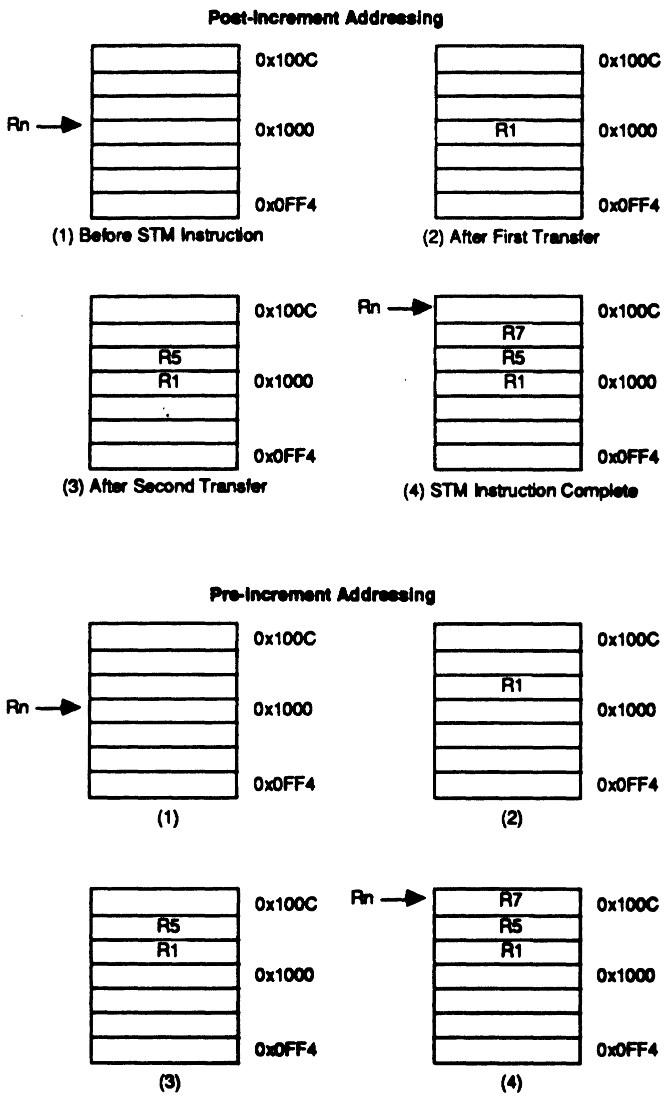
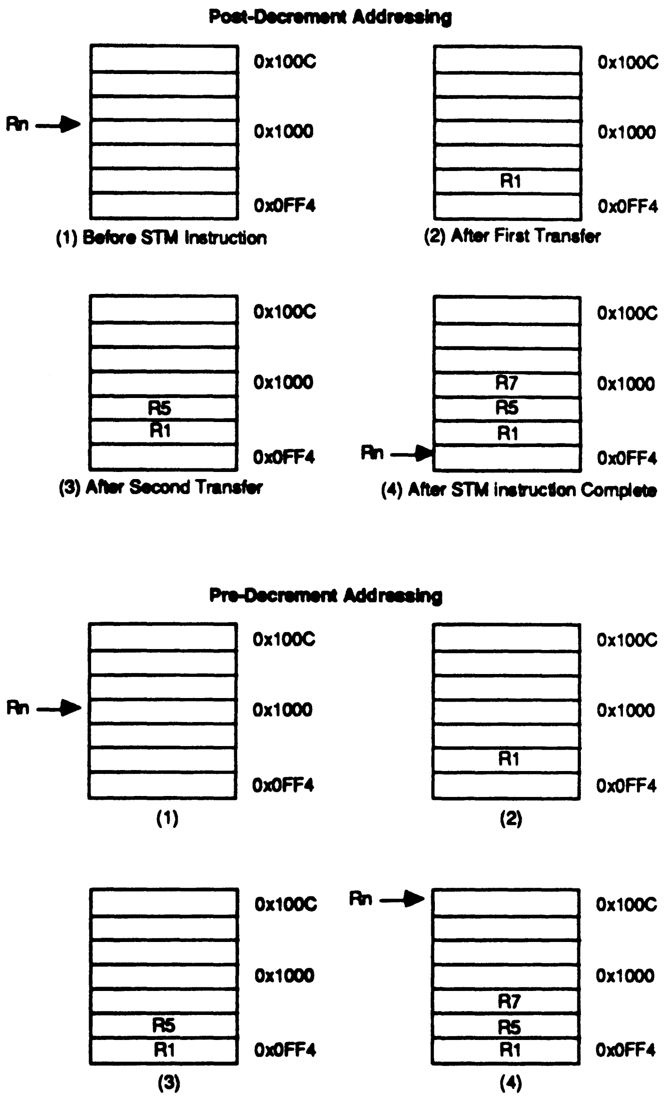

Post-Increment Addressing
Rn
0x100C
0x1000
0x0FF4
R1
R5
R7
(1) Before STM Instruction
(2) After First Transfer
(3) After Second Transfer
(4) STM Instruction Complete
Pre-Increment Addressing
(1)
(2)
(3)
(4)

FIGURE 18. DECREMENTING INDEX


Post-Decrement Addressing
Rn
0x100C
0x1000
0x0FF4
R1
R5
R7
(1) Before STM Instruction
(2) After First Transfer
(3) After Second Transfer
(4) After STM instruction Complete
Pre-Decrement Addressing
(1)
(2)
(3)
(4)

Syntax:

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

where	*cond*	Is an optional 2-letter condition code common to all instructions.
	mode	Is any of: FD, ED, FA, EA, 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 may be used interchangeably: e.g., LDMED performs exactly the same as LDMIB. The name equivalences and instruction bit values are:

Function	Use as Stack	Other usages	L Bit	P Bit	U bit	Operation
Pre-increment load	LDMED	LDMIB	1	1	1	Pop upwards
Post-increment load	LDMFD	LDMIA	1	0	1	Pop upwards
Pre-decrement load	LDMEA	LDMDB	1	1	0	Pop downwards
Post-decrement load	LDMFA	LDMDA	1	0	0	Pop downwards
Pre-increment store	STMFA	STMIB	0	1	1	Push upwards
Post-increment store	STMEA	STMIA	0	0	1	Push upwards
Pre-decrement store	STMFD	STMDB	0	1	0	Push downwards
Post-decrement store	STMED	STMDA	0	0	0	Push downwards

FD, ED, FA, EA indicate whether or not the addressed memory cell has valid data in it (from the previous push or pop), and which direction the stack is to flow. They define the settings of the L, P, and U bits, based on the form of stack required.

The F and E refer to a "full" or "empty" stack cell. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

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

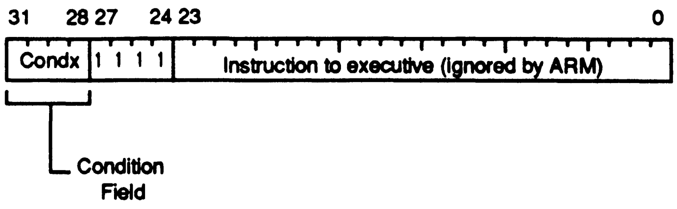
Examples

```
LDMFD       SP!, {R0, R1, R2} ; unstack 3 registers
STMIA       BASE, {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;

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

FIGURE 20. SOFTWARE INTERRUPT (SW)



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 entering 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.
```

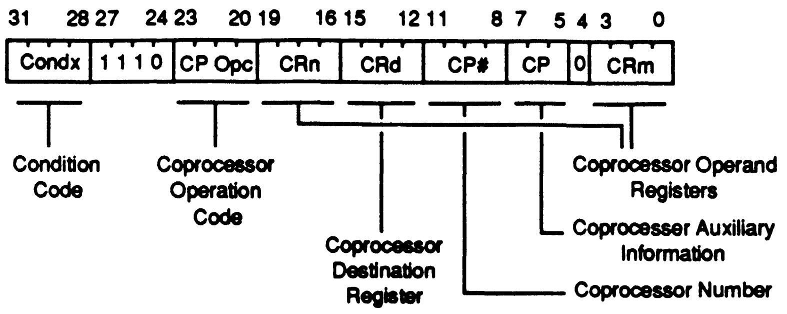
Examples:

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

        SWI             ReadC           ; Get next character from read stream
        SWI             Writel+"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   STMFD   SP!,{r0,r 1, r2)        ; 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.
        ••
        LDM     R13,{R0-R2, PC}^        ; Restore workspace, and return to inst after SWI.
```

FIGURE 21. COPROCESSOR DATA OPERATIONS (CPD)


31 28 27 24 23 20 19 16 15 12 11 8 7 5 4 3 0
Condx
1 1 1 0
CP Opc
CRn
CRd
CP#
CP
0
CRm
Condition Code
Coprocessor Operation Code
Coprocessor Destination Register
Coprocessor Operand Registers
Coprocesser Auxiliary Information
Coprocessor Number

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 APRM. All instructions in this class are used to direct the coprocessor to perform some internal operation. No result is sent back to the APRM, and the APRM 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 APRM activity, allowing the two processors to perform independent tasks in parallel.

Coprocessor Fields - Only bit 4 and bits 31:24 are significant to the APRM; 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.

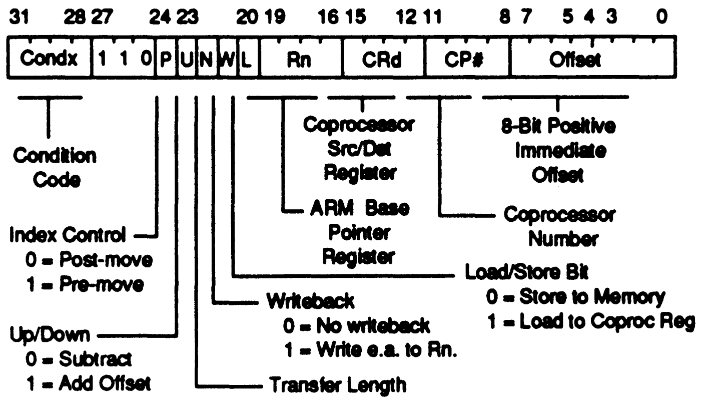
Syntax:

	CPD{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 *CP* field.

Examples:

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

FIGURE 22. 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 APRM 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 APRM's LDR/STR instructions.

Only 8-bit offsets are permitted, and the APRM 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 APRM 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 preindexed 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 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.

Data Aborts - If the address is legal but the memory manager generates an abort, the data abort trap will be taken. The writeback 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} 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-i indexed addressing mode is specified or implied.
	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 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 13-bit expression (including the sign).
Rn	*Avalid register names: R0-R15, SP, LK, or PC. If RN* = PC, the assembler will subtract 8 from the expression to allow for processor address readahead.

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.
```

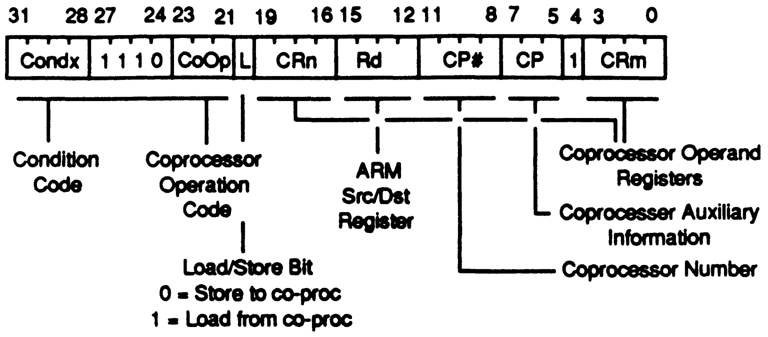
Examples (Post-Index):
In each of these examples, the effective offset is added to the Rn (base pointer) register after using the Rn register as the effective address. Rn is then updated unconditionally, regardless of any ! suffix. Coprocessor #3 is used in all cases, for simplicity.

```
        STC         3, CR1, [R2], R1!     ; *R2 = CR1.  Then R2 += R1.
        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         R2, PLACE             ; PC-relative.  Same as:  STC  R2, [PC+8].
        B           Across                ; Skip over the data temporary.
;
PLACE   DW          0                     ; Temporary storage area.
Across  • • •                             ; Resume execution.
```

FIGURE 23. COPROCESSOR REG 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 APRM. Instructions in this class are used to direct the coprocessor to perform some operation between an APRM 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 APRM register. An example of an MRC usage would be the converse: A FLOAT of a 32-bit value in an APRM register into a floating point value within a coprocessor register.

An important use of this instruction is to communicate control information directly from the coprocessor into the APRM 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 APRM 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 APRM with R1t 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 APRM 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 CP field.

Examples:

```
MCR    1, 10, R1, CR7, CR2        ; Request co-proc #1 to do operation 10 on
                                  ; CR7 and CR2, putting result into APRM's R1.

MRCEQ 2, 5, R1, cr2, Cr3, 2       ; If the Z flag is set, transfer the APRM'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 24. UNDEFINED (RESERVED) INSTRUCTIONS

31 28	27 24	23 8	7 4	3 0
Condx	0 0 0 1	X X X X X X X X X X X X X X X X	1 X X 1	X X X X

31 28	27 25	24 5	4	3 0
Condx	0 1 1	X X X X X X X X X X X X X X X X X X X X	1	X X X X

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

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 SUMMARY

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             ; Discrete test
        CMPNE R2, " "-1                     ; Range test
        MOVLS R2, "."                       ; The,  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          R2nt, 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 25. INSTRUCTION SET SUMMARY

31 28	27 24	23 20	19 16	15 12	11 8	7 4	3 0	
Condx	0 1 I	Opcode S	Rn	Rd	Operand 2			Data Processing
Condx	0 0 0 0 0 0	A S	Rd	Rn	Rs	1 0 0 1	Rm	Multiply
Condx	0 0 0 1	X X X X	X X X X	X X X X	X X X X	1 X X 1	X X X X	Undefined
Condx	0 1 I P	U B W L	Rn	Rd	Offset (variants)			Load, Store
Condx	0 1 1 X	X X X X	X X X X	X X X X	X X X X	X X X 1	X X X X	Undefined
Condx	1 0 0 P	U B W L	Rn	R15 <------ Register List -------> R0				Multi-Register Transfer
Condx	1 0 1 L	Word address offset						Branch, Call
Condx	1 1 0 P	U N W L	Rn	CRd	CP#	Offset		Coproc Data Transfer
Condx	1 1 1 0	CP Opc	CRn	CRd	CP#	CP 0	CRm	Coproc Data Opr
Condx	1 1 1 0	Opc L	Crn	Rd	CP#	CP 1	CRm	Coproc Register Transfer
Condx	1 1 1 1	Bit space ignored by processor						Software Interrupt

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         MOVPC,R14    is safer!
```

Appendix 1 - Differences Between VL86C010 (ARM) And APRM
The modifications made to the ARM in order to create the APRM predominantly affect four instructions; load, load multiple, store, and store multiple. For the load and load multiple instructions both byte and word operations are modified. Only the word functions of the store and store multiple are altered.

The APRM uses the "BigEndian" style byte addressing modes that are the same as the MC680x0 processor family. See Table 1 for examples.

The APRM allows the user to combine segments from two aligned words of data into one nonaligned word oriented as shown in Table 1. The data is loaded via a nine step process which generates 2 complete memory accesses (dbl access). The expected address is generated by the APRM (step 1) the user must notice that the address is nonaligned and freeze the clock (step 2). The user then provides the first word of data (step 3) and brings the NADR signal high (step 4). NADR will latch the appropriate bytes of data from the first word and cause the APRM to output the new address (step 5). This address is the first address incremented by four bytes. The user will provide the second word of data (step 6), deassert NADR (step 7) and restart the clock (step 8). The APRM will take the combination of these two words shifted internally by the proper amount and load them into the destination register (step 9). See Figure 1 for a timing sequence of this procedure.

Table 2 details the shift results for the various address combinations during store operations. External hardware must freeze the processor clock and enable the proper memory lanes for nondestructive un-aligned word stores. NADR functions during double access stores of un-aligned word values to generate the next word aligned address for writing the second data segment.

The APRM also provides 32 address signals although the program area is still limited to the lower 26 bits. Whenever the APRM is performing an opcode fetch the upper six bits are forced low. Increasing the size of the address space made the address exception check unecessary as the old exception areas are now valid memory locations. An input is added, MSBLOW, that when asserted high causes the upper eight bits of the address bus to go low.

A programmable page detector is also added. It can be programmed for 256, 512, 1024, or 2048 word pages. Whenever the next address, if synchronous, would be the last word of the page a new signal called PGHIT would be asserted. See Table 3 for the decodes of the page inputs for the various page sizes.

TABLE 1. SHIFTS FOR LOAD OPERATIONS

B/–W	Byte Address Value (A1, A0)	Data Bus Value (D31-D0)	VL86C010 Shifter (D31-D0)	APRM Shifter (D31- D0)
1	00	11223344	00000044	00000011
1	01	11223344	00000033	00000022
1	10	11223344	00000022	00000033
1	11	11223344	00000011	00000044
0	000 100	11223344 55667788	11223344 55667788	11223344 55667788 no shift or dbl access
0	001 101	11223344 55667788	44112233 no support for dbl access	22334455 with support for dbl access
0	010 110	11223344 55667788	33441122 "	33445566 "
0	011 111	11223344 55667788	22334411 "	44556677 "

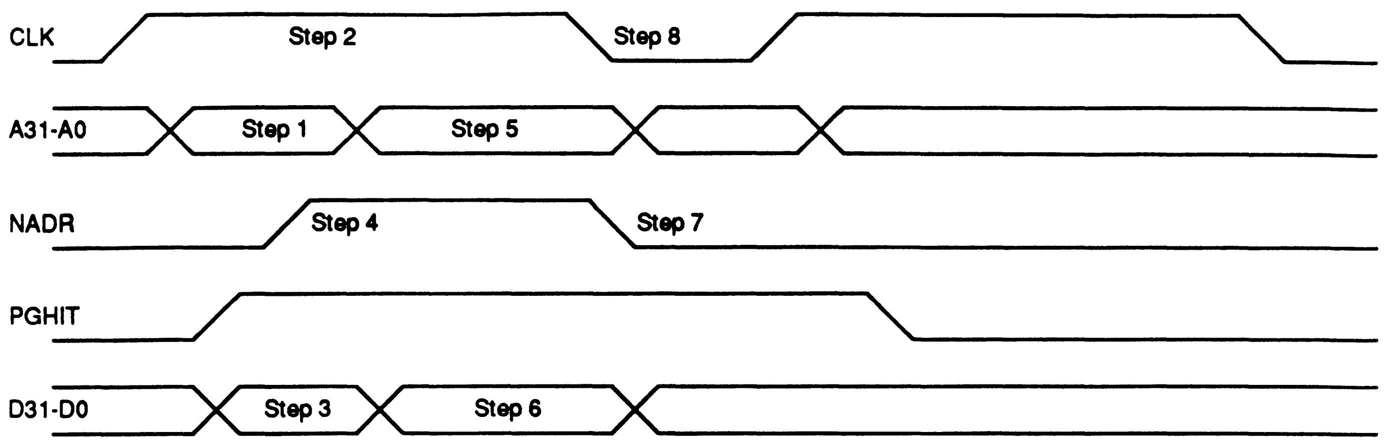
TABLE 2. SHIFTS FOR STORE OPERATIONS

Address Value	Register Data	Output Data at pins
00	11223344	11223344
01	11223344	44112233
10	11223344	33441122
11	11223344	22334411

TABLE 3. MEMORY PAGE SIZE

Page (1,0)	Page Size
00	256 Words
01	512 Words
10	1024 Words
11	2048 Words

FIGURE 1. NONALIGNED MEMORY CYCLES



PACKAGE OUTLINES

100-PIN CERAMIC PIN GRID ARRAY

100-PIN QUAD PLASTIC FLATPACK (QPFP)