32-BIT RISC MICROPROCESSOR WITH CACHE MEMORY

FEATURES

- On-chip 4 Kbyte (1K x 32 bits) cache memory
  - Instructions and data in a single memory
  - 64-way set associative with random replacement
  - Line size of 16 bytes (4 words)
- Compatible with existing support devices
- Upwardly software compatible with VL86C010
- Semaphore instruction added for multiprocessor support
- Full-speed operation up to 20 MHz using typical DRAM devices
- Low interrupt latency for real-time application requirements
- CMOS implementation - low power consumption
- 160-pin plastic quad flatpack package (PQFP)

DESCRIPTION

The VL86C020 Acorn RISC Machine (ARM) is a second generation 32-bit general purpose microprocessor system. The device contains both a general purpose CPU and a full cache memory subsystem in the same package. Several benefits are attained by having the CPU and cache within the same device. First, the processor clock is effectively decoupled from the memory system. This lowers the processor bandwidth demands on the memory and allows most memory cycles to remain on-chip where buffer delays are minimized. Second, a high level of integration is maintained as external components are not required to implement the cache subsystem.

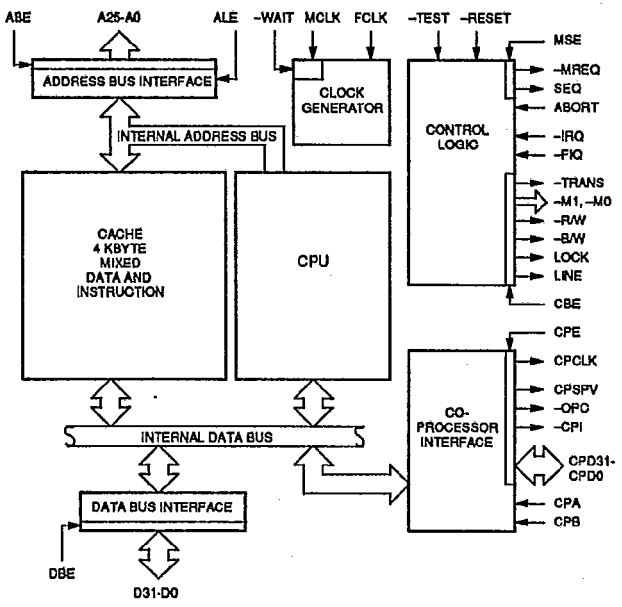
Third, package sizes are reduced as bus widths can remain at reasonable widths. Fourth, memory system design is greatly simplified because most critical timings are handled internally to the device.

The processor is targeted for use in microcomputer and embedded controller applications that require high performance and high integration solutions. Applications where the processor is best applied are: laser printers, graphics engines, network protocol adapters, and any other system that requires quick response to external events and high processing throughput.

Since the VL86C020 typically utilizes only about 14% of the available bus bandwidth, it is particularly well suited to applications where the memory is shared with another high bandwidth device, e.g. a graphics system where the screen refresh occurs from the same memory devices. In addition, systems with more than one processor attached to a single memory system become feasible and are supported with the new semaphore instruction. The instruction performs an indivisible read-modify-write cycle to the memory to allow for management of globally allocated resources reliably.

3

BLOCK DIAGRAM



ORDER INFORMATION

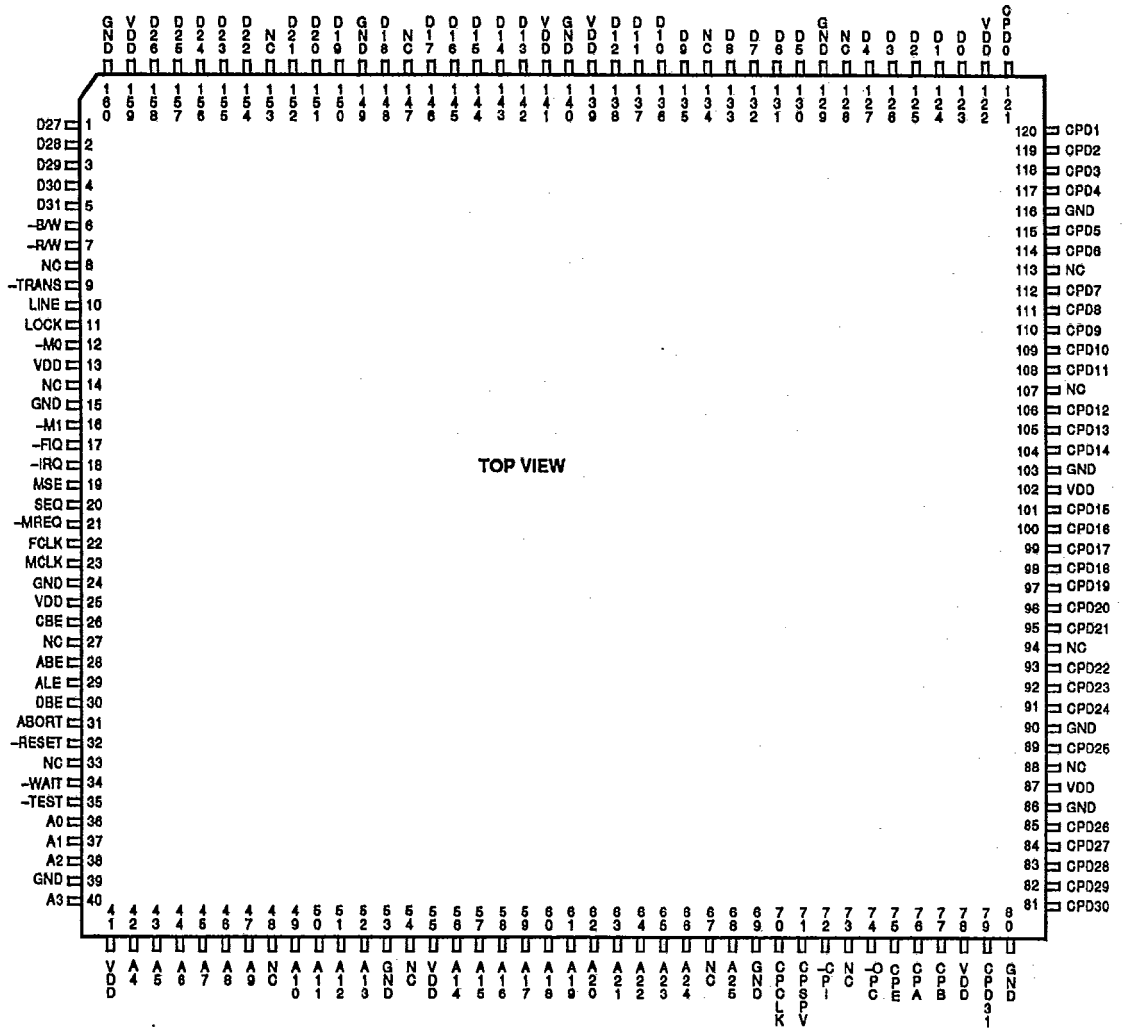
Part Number	Clock Frequency	Package
VL86C020-20FC	20 MHz	Plastic Quad Flatpack (PQFP)
VL86C020-20GC	20 MHz	Plastic Pin Grid Array (PGA)

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

PIN DIAGRAM - PLASTIC QUAD FLATPACK

T-49-17-32

VL86C020



PIN DIAGRAM - PLASTIC PIN GRID ARRAY

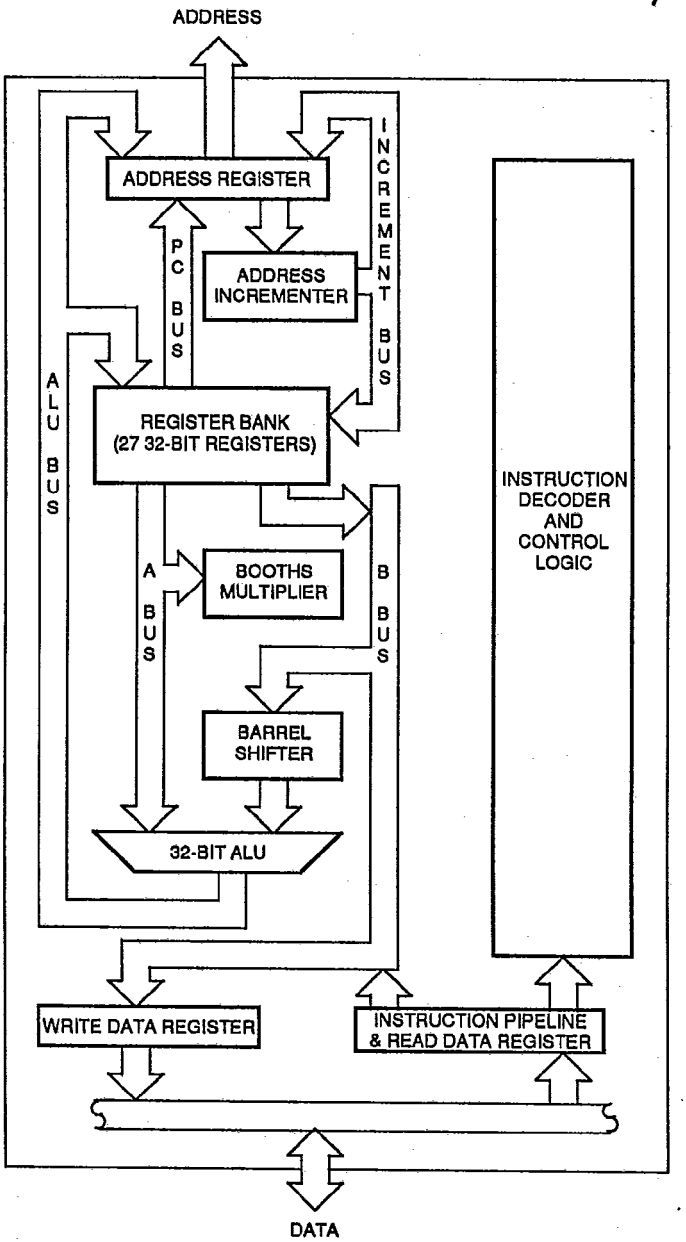
T-49-17-32

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	GND 144	D24 140	D20 136	D19 135	D18 133	D14 129	D13 128	VDD 125	D11 123	D10 122	D7 119	GND 116	D3 114	D0 111	CPD1 108
B	D29 3	VDD 143	D23 139	D21 137	GND 134	D16 131	D15 130	D12 124	D8 120	D6 118	D4 115	D1 112	VDD 110	CPD2 107	GND 104
C	-B/W 6	D28 2	D26 142	D25 141	D22 138	D17 132	VDD 127	GND 126	D9 121	D5 117	D2 113	CPD0 109	CPD3 106	CPD5 103	CPD8 100
D	-TRANS 8	D30 4	D27 1										CPD4 105	CPD7 101	CPD9 99
E	-M0 11	-R/W 7	D31 5	INDEX PIN (INACTIVE)									CPD6 102	CPD10 98	CPD11 97
F	-M1 14	LOCK 10	LINE 9										CPD12 96	CPD13 95	GND 93
G	-FIQ 15	VDD 12	GND 13										CPD15 91	CPD14 94	VDD 92
H	MSE 17	-IRQ 16	SEQ 18					TOP VIEW					CPD16 90	CPD18 88	CPD17 89
J	FCLK 20	GND 22	-MREQ 19										CPD21 85	CPD22 84	CPD19 87
K	MCLK 21	VDD 23	CBE 24										GND 81	CPD24 82	CPD20 86
L	ABE 25	ALE 26	-WAIT 30										CPD26 77	VDD 79	CPD23 83
M	DBE 27	-RESET 29	A1 33										CPD30 73	CPD27 76	CPD25 80
N	ABORT 28	-TEST 31	A2 34	VDD 37	A7 41	A11 45	VDD 49	A18 54	A19 55	A24 60	-OPC 66	CPB 69	VDD 70	CPD29 74	GND 78
P	A0 32	GND 35	A4 38	A6 40	A9 43	A12 46	GND 48	A16 52	A22 58	A23 59	GND 62	-CPI 65	CPE 67	CPD31 71	CPD28 75
Q	A3 36	A5 39	A8 42	A10 44	A13 47	A14 50	A15 51	A17 53	A20 56	A21 57	A25 61	CPCLK 63	CPSPV 64	CPA 68	GND 72

3

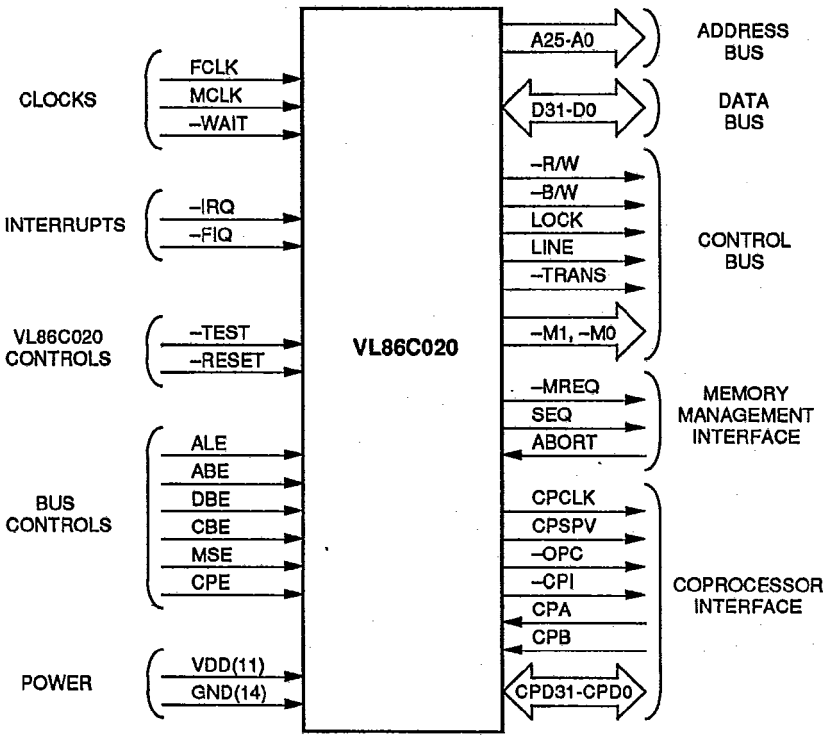
CPU BLOCK DIAGRAM

T-49-17-32



T-49-17-32

FUNCTIONAL DIAGRAM



3

SIGNAL DESCRIPTIONS FOR PLASTIC QUAD FLATPACK

T-49-17-32

Signal Name	Pin Number	Signal Type	Signal Description
A0-A25	42-47, 49-52, 56-66, 68, 36, 38-40	OCZ	Processor Address Bus - If ALE (address latch enable) is high, the addresses change while MCLK is high, and remain valid while MCLK is low; their stable period can be modified by using ALE.
ABE	28	ITP	Address Bus Enable - When this input is low, the address bus drivers (A0-A25) are put into a high impedance state (Note 1). ABE may be left unconnected when there is no system requirement to turn off the address drivers (ABE is pulled high internally - see Note 2).
ABORT	31	IT	Memory Abort - This input allows the memory system to signal the processor that a requested access is not allowed. This input is only monitored when the VL86C020 is accessing external memory.
ALE	29	ITP	Address Latch Enable - This input is used to control transparent latches on the address outputs. Normally the addresses change while MCLK is high. However, when interfacing directly to ROMs, the address must remain stable throughout the whole cycle; taking ALE low until MCLK goes low will ensure that this happens. If the system does not require address lines to be held in this way, ALE may be left unconnected (it is pulled high internally - see Note 2). The ALE latch is dynamic, and ALE should not be held low indefinitely.
-B/W	6	OCZ	NOT Byte/Word - This is an output signal used by the processor to indicate to the external memory system when a data transfer of a byte length is required. -B/W is high for word transfers and low for byte transfers, and is valid for both read and write operations. The signal changes while MCLK is high, and is valid by the start of the active cycle to which it refers.
CBE	26	ITP	Control Bus Enable - When this input is low, the following control bus drivers are put into a high impedance state (Note 1): -B/W, LINE, LOCK, -M1, -M0, -R/W, -TRANS CBE may be left unconnected when there is no system requirement to turn off the control bus drivers (CBE is pulled high internally - see Note 2).
CPA	76	ITP	Coprocessor Absent - A coprocessor which is capable of performing the operation which the VL86C020 is requesting (by asserting -CPI) should take CPA low immediately. The VL86C020 samples CPA when CPCLK and -CPI are both low, the VL86C020 will busy-wait until CPB is low and then complete the coprocessor instruction. If no coprocessors are fitted, CPA may be left unconnected (it is pulled high internally - see Note 2).
CPB	77	ITP	Coprocessor Busy - A coprocessor which is capable of performing the operation which the VL86C020 is requesting (by asserting -CPI), but cannot commit to starting it immediately, should indicate this by taking CPB high. When the coprocessor is ready to start it should take CPB low. The VL86C020 samples CPB when CPCLK and -CPI are both low. If no coprocessors are fitted, CPB may be left unconnected (it is pulled high internally - see Note 2).
CPCLK	70	OCZ	Coprocessor Clock - This pin provides the clock by which all VL86C020 coprocessor interactions are timed. CPCLK is derived from MCLK or FCLK depending on whether the processor is accessing external memory or the cache; the coprocessors must, therefore, be able to operate at FCLK speeds.

SIGNAL DESCRIPTIONS FOR PLASTIC QUAD FLATPACK (Cont.)

T-49-11-32

Signal Name	Pin Number	Signal Type	Signal Description
CPD0-CPD31	121-117, 115, 114, 112-108, 106-104, 101-95, 93-91, 89, 85-81, 79	ITOTZ	Coprocessor Data Bus - These are bidirectional signal paths which are used for data transfers between the processor and external coprocessors, as follows: • For processor instruction fetches (when –OPC = 0), the opcode is sent to the coprocessors by driving CPD0-CPD31 while CPCLK is high. Coprocessor instructions are broadcast unaltered, but non coprocessor instructions are replaced by &FFFFFFFF. • During data transfers from VL86C020 to a coprocessor, the data is driven onto CPD0-CPD31 while CPCLK is high. • During register and data transfers from the coprocessor to VL86C020, CPD0-CPD31 are inputs, and the data must be setup to the falling edge of CPCLK.
CPE	75	ITP	Coprocessor Bus Enable - When this input is low, the following coprocessor bus drivers are put into a high impedance state (see Note 1): CPCLK, CPD0-CPD31, –CPI, CPSPV, –OPC CPE is provided to allow the coprocessor outputs to be disabled while testing the VL86C020 in-circuit, and CPE should be left unconnected for normal operation (it is pulled high internally - see Note 2). If no coprocessor is to be connected to the VL86C020, CPE may be tied low, but CPCLK, CPD0-CPD31, –CPI, CPSPV and –OPC must not be left floating.
–CPI	72	OCZ	NOT Coprocessor Instruction - When VL86C020 executes a coprocessor instruction, it will take this output low and wait for a response from the appropriate coprocessor. The action taken will depend on this response, which the coprocessor signals on the CPA and CPB inputs. –CPI changes while CPCLK is low.
CPSPV	71	OCZ	Coprocessor Supervisor Mode - As instructions are broadcast to the coprocessors on CPD0-CPD31, this output reflects the mode in which each instruction was fetched by the processor (CPSPV = 1 for supervisor/IRQ/FIQ mode fetches, CPSPV = 0 for user mode fetches). The coprocessors may use this information to prevent user-mode programs executing protected coprocessor instructions. CPSPV changes while CPCLK is high.
D0-D31	123-127, 130-133, 135-138, 142-146, 148, 150-152, 154-158, 1-5	ITOTZ	Data Bus - These are bidirectional signal paths which are used for data transfers between the processor and external memory, as follows: • For read operations (when –R/W = 0), the input data must be valid before the falling edge of MCLK. • For write operations (when –R/W = 1), the output data will become valid while MCLK is low.
DBE	30	ITP	Data Bus Enable - When this input is low, the data bus drivers (D0-D31) are put into a high impedance state (Note 1). The drivers will always be high impedance except during write operations, and DBE may be left unconnected in systems which do not require the data bus for DMA or similar activities (DBE is pulled high internally - see Note 2).
FCLK	22	IC	Fast Clock Input - When the VL86C020 CPU is accessing the cache, performing an internal cycle, or communicating directly with the coprocessor, it is clocked with the fast clock, FCLK. This is a free-running clock which is independent of MCLK; the maximum FCLK frequency is determined by the speed of the processor/coprocessor combination.

3

T-49-17-32

SIGNAL DESCRIPTIONS FOR PLASTIC QUAD FLATPACK (Cont.)

Signal Name	Pin Number	Signal Type	Signal Description
–FIQ	17	IT	NOT Fast Interrupt Request - If FIQs are enabled, the processor will respond to a low level on this input by taking the FIQ interrupt exception. This is an asynchronous, level-sensitive input, and must be held low until a suitable response is received from the processor.
–IRQ	18	IT	Not Interrupt Request - As –FIQ, but with lower priority. May be taken low asynchronously to interrupt the processor when the –IRQ enable is active.
LINE	10	OCZ	Line Fetch Operation - This signal is driven high to signal that the CPU is fetching a line of information for the cache. Line fetch operations always read four words of data (aligned on a quad-word boundary), so the LINE signal may be used to start a fast quad-word read from memory. The signal changes while MCLK is high, and remains high throughout the line fetch operation.
LOCK	11	OCZ	Locked Operation - When LOCK is high, the processor is performing a "locked" memory access, and the memory manager should wait until LOCK goes low before allowing another device to access the memory. LOCK changes while MCLK is high, and remains high for the duration of the locked memory accesses (data swap operation).
–M0, –M1	12, 16	OCZ	NOT Processor Mode - These output signals are the inverses of the internal status bits indicating the processor operation mode (–M0, –M1): 11 = User Mode, 10 = FIQ Mode, 01 = IRQ Mode, 00 = Supervisor Mode). –M0, –M1 change while MCLK is high.
MCLK	23	IC	Memory Clock Input - This clock times all VL86C020 memory accesses. The low period of MCLK may be stretched when accessing slow peripherals; alternatively, the –WAIT input may be used with a free-running MCLK to achieve the same effect.
–MREQ	21	OCZ	NOT Memory Request - This is a pipelined signal that changes while MCLK is low to indicate whether the following cycle will be active (processor accessing external memory) or latent (processor not accessing external memory). An active cycle is flagged when –MREQ = 0.
MSE	19	ITP	Memory Request/Sequential Enable - When this input is low, the –MREQ and SEQ cycle control outputs are put into a high impedance state (Note 1). MSE is provided to allow the memory request/sequential outputs to be disabled while testing the VL86C020 in-circuit, and it should be left unconnected for normal operation (MSE is pulled high internally - see Note 2).
–OPC	74	OCZ	Opcode Fetch - –OPC is driven low to indicate to the coprocessors that an instruction will be broadcast on CPD0-CPD31 when CPCLK goes high. –OPC is held valid when CPCLK is low, and changes when CPCLK is high.
–RESET	32	IT	NOT Reset - This is a level sensitive input signal which is used to start the processor from a known address. A low level will cause the instruction being executed to terminate abnormally, and the cache to be flushed and disabled. When –RESET becomes high, the processor will re-start from address 0. –RESET must remain low for at least two FCLK clock cycles, and eight MCLK clock cycles. During the low period the processor will perform dummy instruction fetches from external memory with the address incrementing from the point where –RESET was activated. The address value will wrap around to zero if –RESET is held beyond the maximum address limit.

SIGNAL DESCRIPTIONS FOR PLASTIC QUAD FLATPACK (Cont.)

T-49-17-32

Signal Name	Pin Number	Signal Type	Signal Description
–R/W	7	OCZ	NOT Read/Write - When high this signal indicates a processor write operation; when low, a read operation. The signal changes while MCLK is high, and is valid by the start of the active cycle to which it refers.
SEQ	20	OCZ	Sequential Address - This signal is the inverse of –MREQ, and is provided for compatibility with existing ARM memory systems (VL86C020 has a subset of VL86C010 bus operations; see Memory Interface section).
–TEST	35	ITP	NOT Test - When this input is low, the VL86C020 enters a special test mode which is only used for off-board testing. –TEST must not be driven low while the VL86C020 is in-circuit, but may be left unconnected as it is pulled high internally (see Note 2).
–TRANS	9	OCZ	NOT Memory Translate - When this signal is low it indicates that the processor is in user mode, or that the supervisor is using a single transfer instruction with the force translate bit active. It may be used to tell memory management hardware when translation of the addresses should be turned on, or as an indicator or non-user mode activity.
–WAIT	34	ITP	NOT Wait - When accessing slow peripherals, the VL86C020 can be made to wait for an integer number of MCLK cycles by driving –WAIT low. Internally, –WAIT is ANDed with the MCLK clock, and must only change when MCLK is low. If –WAIT is not used in a system, it may be left unconnected (it is pulled high internally - see Note 2).
VDD	13, 25, 41, 55, 78, 87, 102, 122, 139, 141, 159		Power supply: +5 V
GND	15, 24, 39, 53, 69, 80, 86, 90, 103, 116, 129, 140, 149, 160		Ground
NC	8, 14, 27, 33, 48, 54, 67, 73, 88, 94, 107, 113, 128, 134, 147, 153		No connect

Key to Signal Types:

IC	CMOS-level input
IT	TTL-level input
ITP	TTL-level input with pull-up resistor (Note 2)
OCZ	3-state CMOS-level output
ITOTZ	Bidirectional: 3-state TTL-level output; TTL-level input

Notes:

1. When output pads are placed in the high impedance state for long periods, care must be taken to ensure that they do not float to an undefined logic level, as this can dissipate a lot of power, especially in the pads.
2. The "ITP" class of pads incorporate a pull-up resistor which allows signals with normally high inputs to be left unconnected. The value of the pull-up resistor will fall within the range 10 kΩ - 100 kΩ.

PROGRAMMERS' MODEL

The VL86C020 processor has a 32-bit data bus and a 26-bit address bus. The processor supports two data types, eight-bit byte and 32-bit words, where words must be aligned on four byte boundaries. Instructions are exactly one word, and data operations (e.g. ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words. The VL86C020 supports four modes of operation, including protected supervisor and interrupt handling modes.

BYTE SIGNIFICANCE

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

REGISTERS

The processor has 27 registers (32-bits each), 16 of which are visible to the programmer at any time. The visible subset depends on the current processor mode; special registers are switched in to support interrupt and supervisor processing. The register bank organization is shown in Table 1.

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

TABLE 1. REGISTER ORGANIZATION

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

General Usage

TABLE 2. BYTE ADDRESSING

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

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

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

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

The two private registers allow the IRQ and Supervisor modes each to have a private stack pointer and line 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 re-entrant interrupt handler, R14 should be saved onto a stack in main memory before re-enabling the interrupt. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled.

FIQ - The FIQ (Fast Interrupt Request) exception is externally generated by taking the –FIQ 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 FIQ 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 FIQ synchronizer at the end of each instruction.

The impact upon execution of an FIQ interrupt is defined in Table 3. 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 FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the PC (but note that this is not possible from user mode). If the I flag is clear, the processor checks for a low level on the output of the IRQ synchronizer at the end of each instruction.

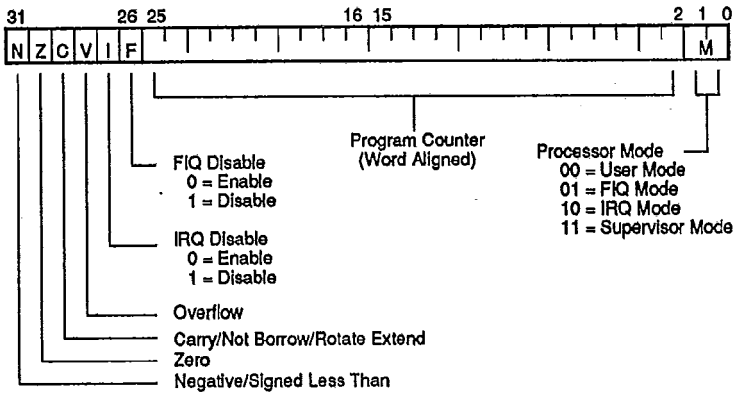
The impact upon execution of an IRQ interrupt is defined in Table 3. The return-from-interrupt sequence is also defined there. This will cause execution to resume at the instruction following the interrupted one, restore the original processor state, and re-enable the IRQ interrupt.

Address Exception Trap - An address exception arises whenever a data transfer is attempted with a calculated address above 3FFFFFFH. The VL86C020 address bus is 26-bits wide, and an address calculation will have a 32-bit result. If this result has a logic one in any of the top six bits, it is assumed that the address is an error and the address exception trap is taken.

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

When an address exception is seen, the processor will respond as defined in Table 3. The return-from-interrupt sequence is also defined there. This will resume execution of the interrupted code sequence, and restore the original processor state.

FIGURE 1. PROGRAM COUNTER AND PROCESSOR STATUS REGISTER



Normally, an address exception is caused by erroneous code, and it is inappropriate to resume execution. If a return is required from this trap, use SUBS PC, R14_svc, 4, as defined in Table 3. This will return to the instruction after the one causing the trap.

Abort - The ABORT signal comes from an external memory management system, and indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. The processor checks for an abort at the end of the first phase of each bus cycle. When successfully aborted, the VL86C020 will respond in one of three ways:

1. 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.)
2. If the abort occurred during a data access (a data abort), the action depends on the instruction type. Data transfer instructions (LDR, STR, SWP) are aborted as though the instruction had not executed. The LDM and STM instructions complete, and if write back is set, the base is updated. If the instruction would normally have overwritten the base with data (i.e. LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.
3. If the abort occurred during an internal cycle it is ignored.

Then, in cases (1) and (2), the processor will respond as defined in Table 3.

The return from Prefetch Abort defined in Table 3 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 Table 3.

The abort mechanism allows a demand paged virtual memory system to be implemented when a suitable memory management unit (such as the VL86C110) is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable the memory manager signals an abort. The processor traps into system software which must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

Software Interrupt - The software interrupt is used for getting into supervisor mode, usually to request a particular supervisor function. The processor

TABLE 3. EXCEPTION TRAP CONSIDERATIONS

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

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

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

response to the (SWI) instruction is defined in Table 3, as is the method of returning. The indicated return method will return to the instruction following the SWI.

Undefined Instruction Trap - When VL86C020 executes a coprocessor instruction or the 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 VL86C020 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 VL86C020 will respond as defined in Table 3. The return from this trap (after performing a suitable emulation of the required function), defined in Table 3 will return to the instruction following the undefined instruction.

Reset - When –RESET 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 3. 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 follows:

Address	Function
0000000	Reset
0000004	Undefined Instruction
0000008	Software Interrupt
000000C	Abort (Prefetch)
0000010	Abort (Data)
0000014	Address Exception
0000018	IRQ
000001C	FIQ

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

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

1. Reset (highest priority)
2. Address Exception, Data Abort
3. FIQ
4. IRQ
5. Prefetch Abort
6. Undefined Instruction, Software Interrupt (lowest priority)

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

If an address exception or data abort occurs at the same time as a FIQ, and FIQs are enabled i.e. the F flag in the PSR is clear, the processor will enter the address exception or data abort handler and then immediately proceed to the FIQ vector. A normal return from FIQ will cause the address exception or data abort handler to resume execution. Placing address exception and data abort at a higher priority than FIQ 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 FIQ latency calculations.

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

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, 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 FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time.

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

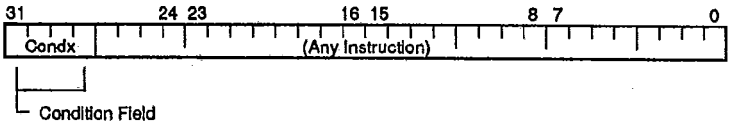
INSTRUCTION SET

All VL86C020 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, i.e. 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 would not be executed.

FIGURE 2. CONDITION FIELD



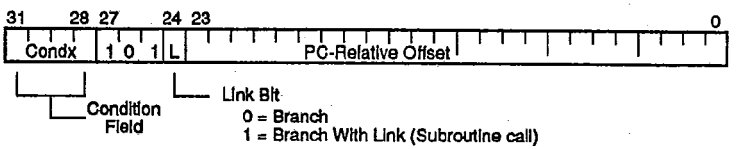
0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned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

Branch and Branch with Link (B, BL) The B, BL instructions are only executed if the condition field is true.

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

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

FIGURE 3. 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)

Assembler 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.

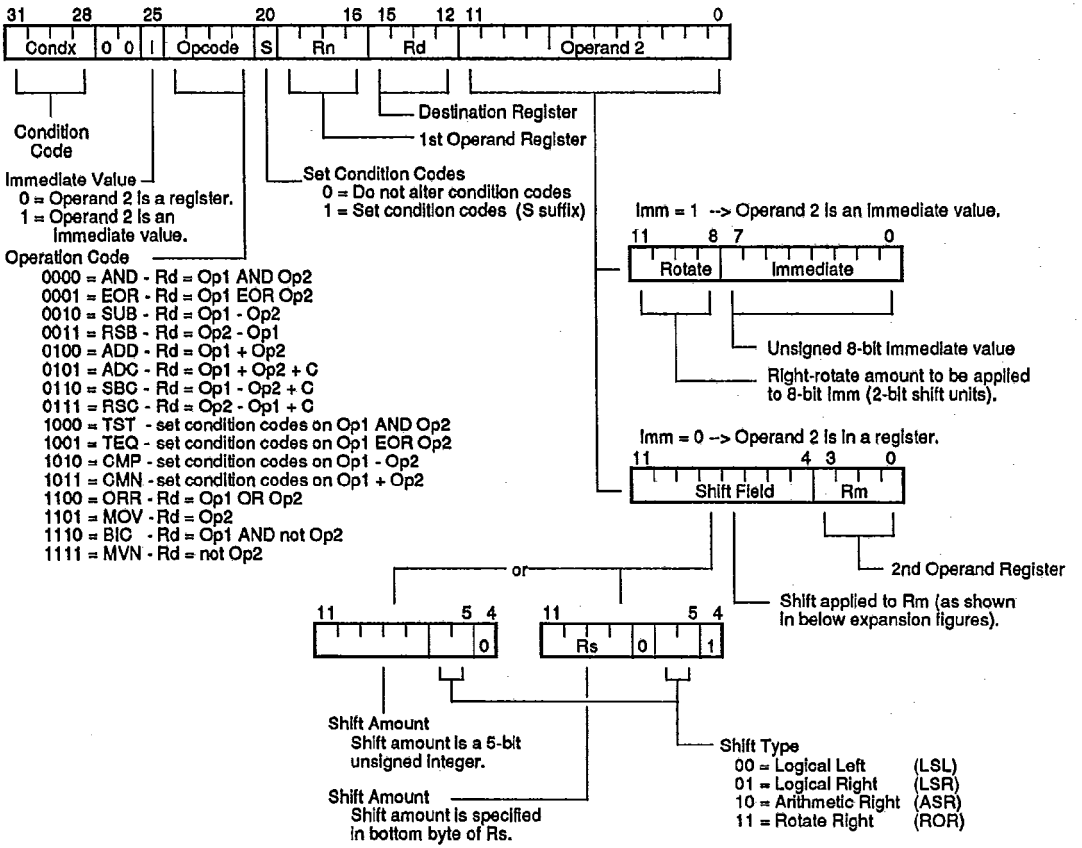
T-49-17-32

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).
```

T-49-17-32

FIGURE 4. ALU INSTRUCTION TYPES



ALU Instructions - The ALU-type instruction is only executed if the condition is true. The various conditions are defined in Condition Field Section.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8-bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the PSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result, and therefore, should always have the S bit set. (The assembler treats TST, TEQ, CMP and CMN as TSTS, TEQS, CMPS and CMNS by default.)

T-49-17-32

DATA PROCESSING OPERATIONS

Assembler Mnemonic	Opcode	Action
AND	0000	Bit-wise logical AND of operands
EOR	0001	Bit-wise logical Exclusive Or of operands
SUB	0010	Subtract operand 2 from operand 1
RSB	0011	Subtract operand 1 from operand 2
ADD	0100	Add operands
ADC	0101	Add operands plus carry (PSR C flag)
SBC	0110	Subtract operand 2 from operand 1 plus carry
RSC	0111	Subtract operand 1 from operand 2 plus carry
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	Bit-wise logical OR of operands
MOV	1101	Move operand 2 (operand 1 is ignored)
BIC	1110	Bit clear (bit-wise AND of operand 1 and NOT operand 2)
MVN	1111	Move NOT operand 2 (operand 1 is ignored)

3

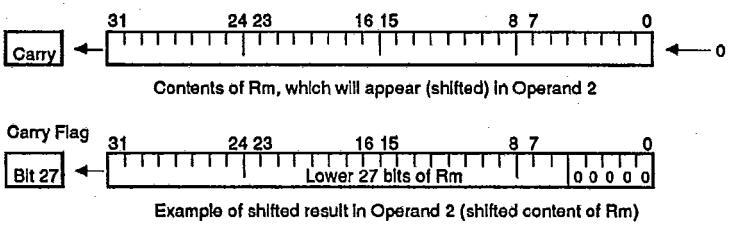
PSR Flags - The operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15), the V flag in the PSR 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 zeros, and the N flag will be set to the logical value of bit 31 of the result.

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or 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 4.

When the shift amount is specified in the instruction, it is contained in a 5-bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm 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 Data Processing Operations above.) For example, the effect of LSL 5 is:

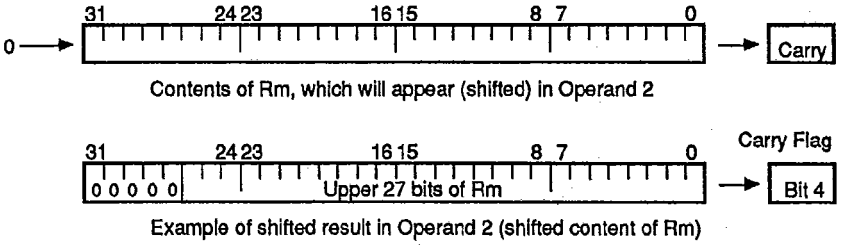
FIGURE 5. LOGICAL SHIFT LEFT (LSL)



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

A Logical Shift Right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR 5 has the effect shown in Figure 6.

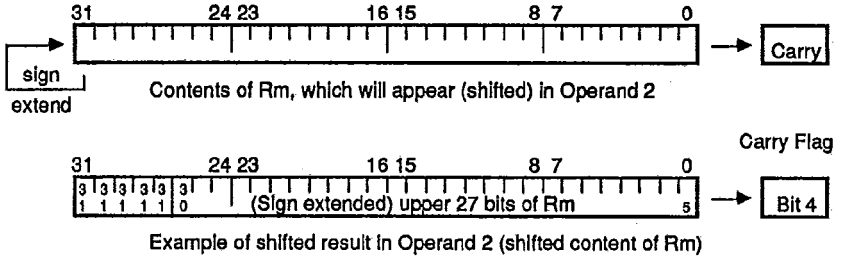
FIGURE 6. LOGICAL SHIFT RIGHT (LSR)



The form of the shift field which might be expected to correspond to LSR 0 is used to encode LSR 32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero. 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 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:

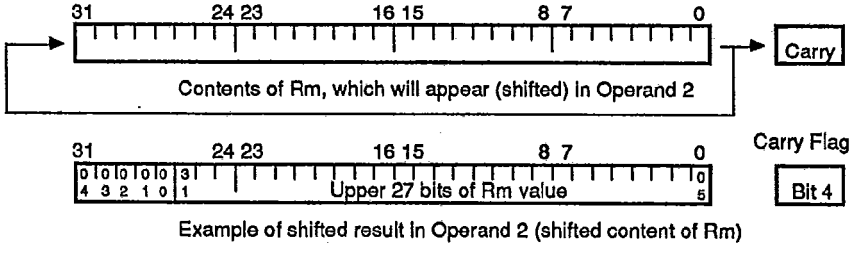
FIGURE 7. ARITHMETIC SHIFT RIGHT (ASR)



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

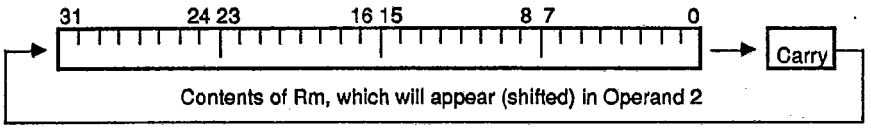
Rotate Right (ROR) operations reuse the bits which "overshoot" in a logical shift right operation by wrapping them around at the high end of the result. For example, the effect of a ROR 5 is:

FIGURE 8. ROTATE RIGHT (ROR)



The form of the shift field which might be expected to give ROR 0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one-bit position of the 33-bit quantity formed by appending the PSR C flag to the most significant end of the contents of Rm:

FIGURE 9. ROTATE RIGHT EXTENDED (RRX)



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

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

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

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

Note: The zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or an undefined instruction.

Immediate Operand Rotation - The immediate operand rotate field is a 4-bit unsigned integer which specifies a shift operation on the 8-bit immediate value. The immediate value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2. Another example is that the 8-bit constant may be aligned with the PSR flags (bits 0, 1, 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, and 29 to the C flag and bit 28 to the V flag. In user mode the other flags (I, F, M1, M0) are protected from direct change, but in non-user modes these will also be affected, accepting copies of bits 27, 26, 1 and 0 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 (R8-R14) during its first cycle. A no-op should be inserted if the next instruction must access a banked register. Accesses to the unbanked registers (R0-R7 and R15) are safe. This restriction is required for the VL86C010 processor and does not apply to VL86C020, but should be adhered to for compatibility.

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 to update those PSR flags which are not protected by virtue of the processor mode.

Setting PSR Bits - It is suggested that TEQP be used to set PSR bits in SVC mode. Because these bits are not presented to the ALU input (even when R15 is the operand), the TEQP's operands replace all current PSR bits. For example, to remain in SVC mode but set the interrupt-disable bits, use a "TEQP PC, 0x C000003" instruction.

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

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

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

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount, the PC will be 8 bytes ahead when used as Rs, and 12 bytes ahead when used a Rn or Rm.

Assembler 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.*

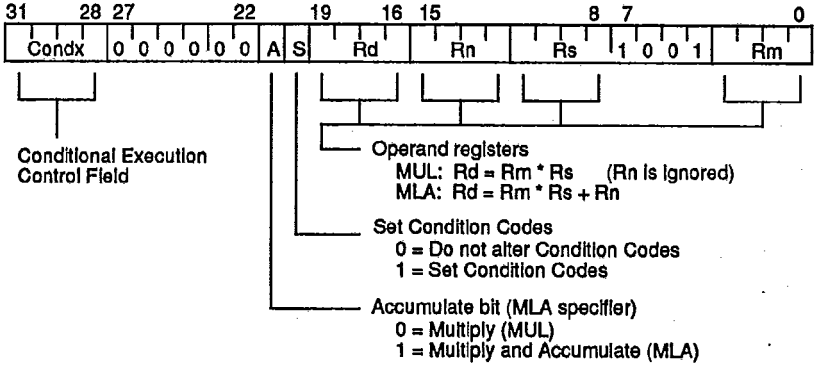
Note: If *<expression>* is used, the assembler will attempt to generate a shifted immediate eight-bit field to match the expression. If this is impossible, it will give an error.

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.
```

T-49-17-32

FIGURE 10. MULTIPLY AND MULTIPLY-ACCUMULATE (MUL, MLA)



3

The multiply and multiply-accumulate instructions use a 2-bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32-bit operands, and may be used to synthesize higher precision multiplications.

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

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

Both forms of the instruction work on operands which may be considered as signed (2'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 previously, R15 must not be used as the destination register (Rd). If it is so used, the instruction will have no effect except possibly to scramble the PSR flags.

R15 As an Operand - R15 may be used as one or more of the operands, though the result will rarely be useful. When used as Rs the PC bits will be used without the PSR flags, and the PC value will be 8 bytes advanced 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 8 bytes advanced 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.

T-49-17-32

Assembler 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 registers 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.

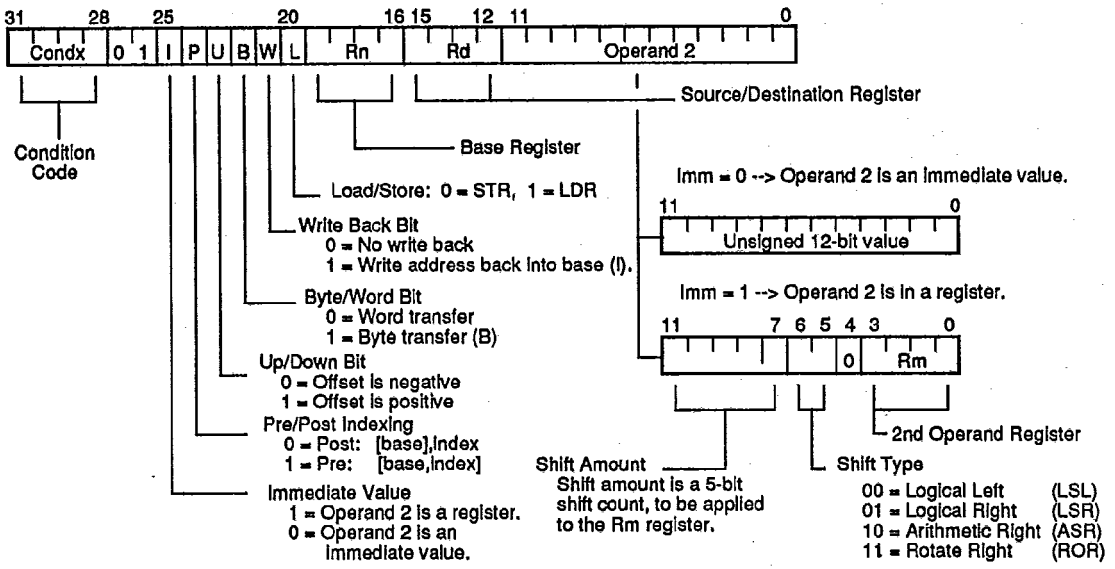
Hardware Address Translation - The only use of the W bit in a post-indexed data transfer is in non-user mode code, where setting the W bit forces the –TRANS pin to go low for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes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

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

Bytes and Words - This instruction class may be used to transfer a byte (B=1) or a word (B=0) between a VL86C020 register and memory. In the discussion, remember that the VL86C020 stores words into memory with the Least Significant Byte at the lowest address (i.e., LSB first).

3

FIGURE 11. SINGLE DATA TRANSFER (LDR, STR)



Non-Aligned Addresses - A byte load (LDRB) expects the data on bits D7 to D0 if the supplied address is on a word boundary, on bits D15 to D8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom eight bits of the destination register, and the remaining bits of the register are filled with zeros.

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.

Non-Aligned Accesses - A word load (LDR) should generate a word aligned address. An address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits D7 to D0. See the below example.

External hardware could perform a double access to memory to allow non-aligned word loads, but the VL86C110 Memory Controller does not support this function.

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

If R15 is specified as the base register (Rn), the PC is used without the PSR flags. When using the PC as the base register one must remember that it contains an address 8 bytes advanced from the address of the current instruction.

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

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

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

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

Data Aborts - A transfer to or from a legal address may still present special cases for a memory management system. For instance, in a system which uses virtual memory, the required data may be absent from main memory. The memory manager can signal a problem by taking the processor ABORT 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 be restarted and the original program continued.

Cache Interaction - When the cache is turned on, a data load operation (LDR, LDRB) will read data from the cache if it is present. If the cache is turned off, or does not contain the required data, the external memory is accessed.

A data store operation (STR, STRB) will always cause an immediate external write to allow the external memory manager to abort the access if it is illegal. If the write operation is not aborted, and the cache contains a copy of data from the address being written to, the cache will be automatically updated with the new byte or word of data. This updating occurs even when the cache is turned off (to maintain cache consistency), but can be disabled by programming the updateable control register appropriately. (See Cache Operation.)

Example: Read two 16-bit values from an I/O port, merging into a 32-bit word.

```
MASK:   DW    0xFFFF
IO_16   DW    0x3100000     ; I/O port address
WORD    DW    0             ; 32-bit result
          .
          .
        LDR   R3, IO_16     ; Get word-aligned source address.
        LEA   R4, BUF       ; Get word-aligned destination address.
        LDR   R0, MASK
        LDR   R1, [R3], 2   ; Fetch even half-word from 16-bit port
        AND   R1, R1, R0    ; Keep lower 16 bits.
        LDR   R2, [R3], 2   ; Fetch 'add' half-word, rotated.
        BIC   R2, R2, R0    ; Keep upper 16 bits.
        ORR   R1, R1, R2    ; Merge even/odd halves.
        STR   R1, [R4], 4   ; Store 32-bit composit.
```

Assembler 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 –TRANS pin to go low for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.
Rd is a valid register: R0-R15, SP, LK, or PC.
Address Can be any of the variations in the following table.

Address Variants:

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

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

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

Examples (Pre-Index and Optional Increment):
In each of these examples, the effective offset is added to the 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.
```

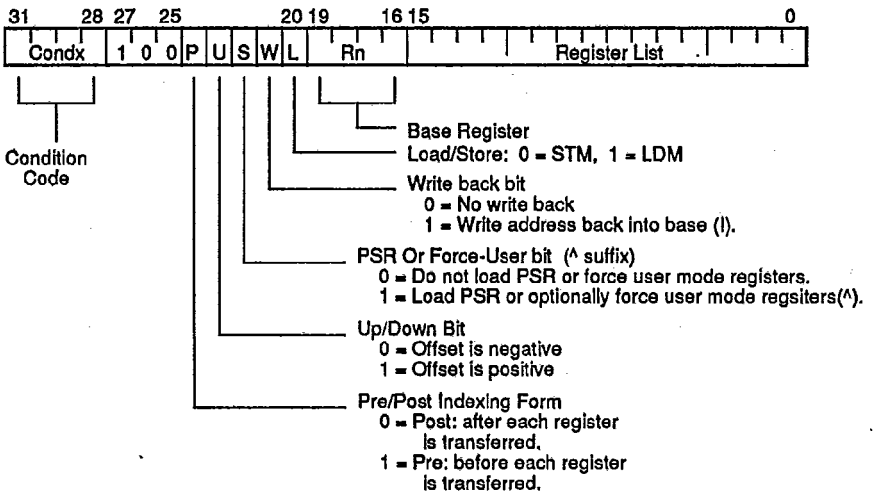
Examples (Post-Index and Increment):
In each of these examples, the effective offset is added to the Rn (base pointer) register after using the Rn register as the effective address. Rn is then updated unconditionally, regardless of any "!" suffix.

```
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 temporarily.
;
PLACE   DW      0               ; Temporary storage area.
Across  ...                     ; Resume execution.
```

FIGURE 12. LOAD/STORE REGISTER LIST FROM MEMORY (LDM,STM)



Multi-Register Transfer (LDM, STM) The instruction is only executed if the condition is true. The various conditions are defined in Control Field Section.

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 1 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 13 and 14.

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, re-enabling 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, do not 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 in to the 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, do not use write back when forcing a user bank transfer.

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

Base within the Register List - When write back is specified, the base is written back at the end of the second cycle of the instruction. During 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.

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

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

Data Aborts - Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the ABORT pin high. This can happen on any transfer during a multiple register load or store, and must be recoverable if VL86C020 is to be used in a virtual memory system.

Abort during STM - If the abort occurs during a store multiple instruction, VL86C020 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 VL86C020 detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

The following figures illustrate the impact of various addressing modes. R1, R5, and R7 are moved to/from memory, where Rn=0x1000, and a write back of the modified base is done (W=1). The figures show the sequence of incrementing "pushes", the addresses used, and the final value of Rn.

Without write back, Rn would remain at 0x1000.

Figure 13 illustrates the use of incrementing stack "pushes".

Figure 14 illustrates decrementing "pushes" to the stack based upon Rn.

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

FIGURE 13. INCREMENTING INDEX

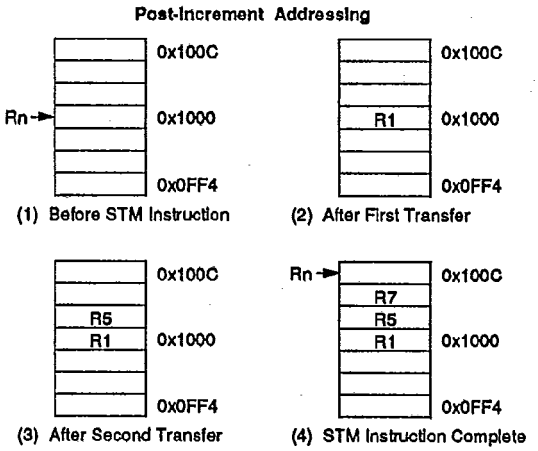
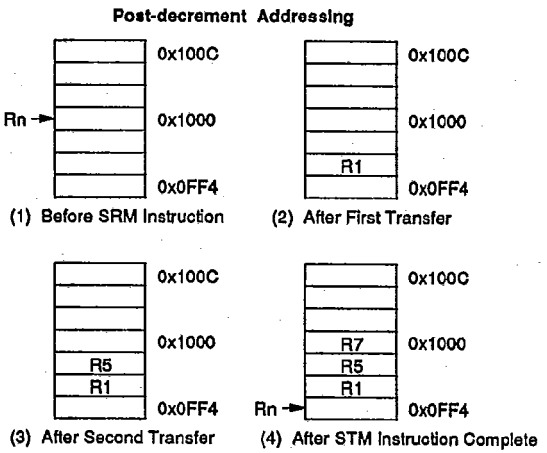


FIGURE 14. DECREMENTING INDEX



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

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

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

With the cache turned on, a block load operation (LDM) will read data from the cache where it is present. When the cache does not contain the required data, the external memory is accessed.

A block store operation (STM) always generates immediate external writes to allow the external memory manager to abort the accesses if they are illegal. The cache is automatically updated as the data is written to memory (provided the area being written to is updateable, see Cache Operation Section).

Assembler Syntax:

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

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.
RIist 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

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

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

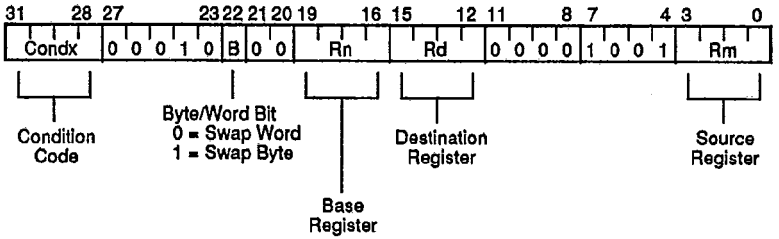
Examples

```
LDMFD   SPI, {R0, R1, R2}     ; unstack 3 registers
STMIA   R2, {R0, R15}         ; save all registers
```

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

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

FIGURE 15. SINGLE DATA SWAP (SWP)



Single Data Swap (SWP) - The instruction is only executed if the condition is true. The various conditions are defined in Condition Field Section.

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

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address (the external memory is always accessed, even if the cache contains a copy of the data). The processor then writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

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

Bytes and Words - This instruction class may be used to swap a byte (B=1) or a word (B=0) between a VL86C020 register and memory.

A byte swap (SWPB) expects the read data on bits 0 to 7, if the supplied address is on a word boundary, on bits 8 to 15 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom eight bits of the destination register, and the remaining bits of the register are filled with zeros. The byte to be written is repeated four times across the data bus. The external memory system should activate the appropriate byte subsystem to store the data (see Memory Interface Section).

A word swap (SWP) should generate a word aligned address. An address offset from a word boundary will cause the data read from memory to be rotated into the register so that the addressed byte occupies bits 0 to 7. The data written to memory are always presented exactly as they appear in the register (i.e. bit 31 of the register appears on D31).

Use of R15 - If R15 is selected as the base, the PC is used together with the PSR. If any of the flags are set, or interrupts are disabled, the data swap will cause an address exception. If all flags are clear, and interrupts are enabled (so the top six bits of the PSR are clear), the data will be swapped with an address 8 bytes advanced from the swap instruction, although the address will not be word aligned unless the processor is in user mode. (M1 and M0 bits determine the byte address).

When R15 is the source register (Rm), the value stored will be the PC together with the PSR. The stored value of the PC will be 12 bytes advanced from the address of the instruction.

When R15 is the destination register (Rd), the PSR will be unaffected, and only the PC will change.

Address Exceptions - If the base address used for the swap has a logic one in any of the bits 26 to 31, the transfer will not take place and the address exception trap will be taken.

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

Cache Interaction - The swap instruction always reads data from external memory, even if a copy is present in the cache. In multi-processor systems, semaphores may be used to control access to system resources; as the semaphores are accessed by more than one processor, the cache copy of a semaphore may be out of date (the cache is only updated if the host CPU writes new data to the external memory). It is, therefore, important always to read the semaphore from the shared external memory, and not the private cache. The write operation of the swap instruction will still update the cache if a copy of the address is present, and updating is enabled (see Cache Operation Section).

Assembler Syntax:

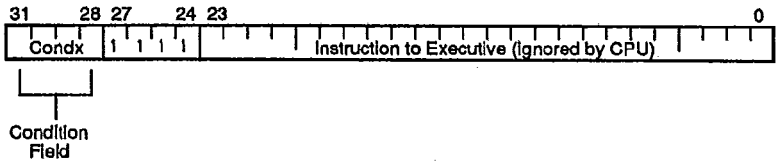
SWP*{cond}{B}* Rd,Rm,[Rn]

where *cond* Two-character condition mnemonic, see section Condition Field
B If B is present then byte transfer, otherwise word transfer.
Rd,Rm,Rn Are expressions evaluating to valid register numbers. Rn is required.

Examples:

```
SWP     R0, R1, [BASE]   ; Load R0 with the contents of BASE, and store R1 at BASE.
SWPB    R2, R3, [BASE]   ; Load R2 with the byte at BASE, and store bits 0 to 7 of R3 at BASE.
SWPEQ   R0, R0, [BASE]   ; Conditionally swap the contents of BASE with R0.
```

FIGURE 16. SOFTWARE INTERRUPT (SWI)



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

The Software Interrupt (SWI) instruction is used to enter supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change, with execution resuming at 0x 08. 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.

Assembler Syntax:

SWI{cond} <expression>

where *cond* Is the two-character condition code common to all instructions.
expression Is a 24-bit field of any format. The processor itself ignores it, but the typical scenario is for the software executive to specify patterns in it, which will be interpreted in a particular way by the executive, as commands.

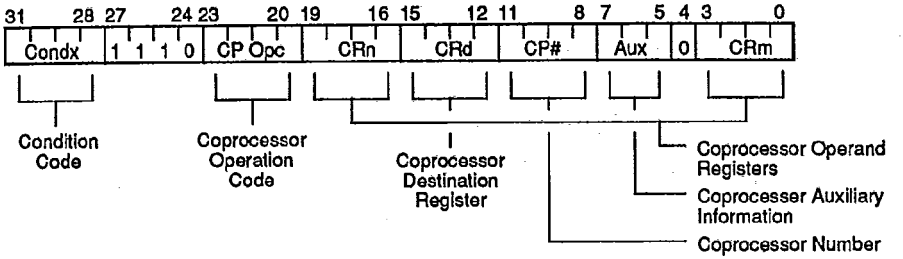
Examples:

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

        SWI     ReadC                       ; Get next character from read stream
        SWI     Write1+"k"                  ; Output a "k" to the Write stream
        SWINE   0                           ; Conditionally call supervisor with 0 in comment field
```

The above examples assume that suitable supervisor code exists. For instance:

```
; Assume that the R13_svc (the supervisor's R13) points to a suitable stack.
        acons   Zero=0, ReadC=1, Write1=2   ; Assembler constants.
        acons   CC_Mask = 0xFC00003         ; Non-address area mask.

08h     B       Super                       ; SWI entry point
        ..
Super   STMFD   SP!,{r0,r1, r2,r14)         ; Save working registers.
        BIC     r1, r14, CC_Mask            ; Strip condx codes from SWI instruction address.
        LDR     R0, [R1, -4]                ; Get copy of SWI instruction.
        BIC     R0, R0, 0xFF000000          ; Get lower 24 bits of SWI, only.
        MOV     R1, SWI_Table               ; Get absolute address of PC-relative table.
        LDR     PC, [R1, R0 LSL 2]          ; Jump indirect on the table.

SWI_Table dw    Zero_Action                 ; Address of service routines.
        dw      ReadC_Action
        dw      Write1_Action

Write1_Action                               ; Typical service routine.
        ..
        LDM     R13,{R0-R2, PC}^            ; Restore workspace, and return to inst after SWI.
```

FIGURE 17. COPROCESSOR DATA OPERATIONS (CDO)



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

Coprocessor Fields - Only bit 4 and bits 31-24 are significant to the CPU; the remaining bits are used by coprocessors. The above field names are used by convention, and 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.

VL86C010 CDO Instruction - The implementation of the CDO instruction on the VL86C010 processor causes a Software Interrupt (SWI) to take the undefined Instruction trap if the SWI was the next instruction after the CDO. This is no longer the case on the VL86C020, but the sequence

```
CDO
SWI
```

should be avoided for program compatibility.

Assembler Syntax:

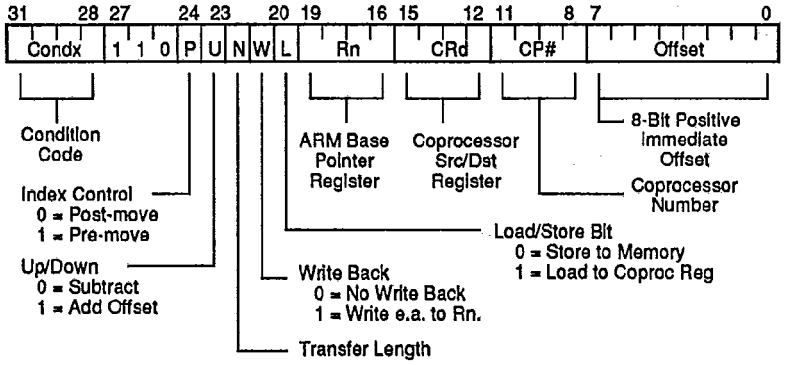
CDO{cond} CP#,<expression1>, CRd, CRn, CRm{,<expression2>}

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

Examples:

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

FIGURE 18. COPROCESSOR DATA TRANSFERS (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 the 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 the registers for context switching.

Offsets and Indexing - The VL86C020 is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are similar to those used for the VL86C020's LDR/STR instructions.

Only 8-bit offsets are permitted, and the VL86C020 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 VL86C020 internally shifts the offset left 2 bits before addition to the Rn register.

The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address. The modified base value may be written back into the base (W-1), or the old base value may be kept (W-0). In the case of post-indexed addressing, the write back bit is redundant, since the old base value can be retained by setting the offset to zero. Therefore, post-indexed data transfers always write back the modified base.

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

Use of R15 - If R15 is specified as the base register (Rn), the PC is used without the PSR flags. When using the PC as the base register note that it contains an address 8 bytes advanced from the address of the current instruction. As with the LDR/STR case, the assembler performs this compensation automatically.

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

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

Note that only the address actually used for the transfer is checked. A base containing an address outside the legal range may be used in a pre-indexed transfer if the offset brings the

address within the legal range. Likewise, a base within the legal range may be modified by post-indexing to outside the legal range without causing an address exception.

Data Aborts - If the address is legal but the memory manager generates an abort, the data abort trap will be taken. The write back of the modified base will take place, but all other processor state data will be preserved. The coprocessor is partly responsible for ensuring restartability. It must either detect the abort, or ensure that any actions consequent from this instruction can be repeated when the instruction is retried after the resolution of the abort.

Cache Interaction - When the cache is on, LDC instructions will attempt to read data from the cache. STC instructions update the cache data if the address being written to matches a cache entry (see Cache Operation Section).

When an STC instruction is executed with the cache turned off, the VL86C020 will drive data onto D31-D0 (provided DBE is high) in the latent cycle preceding the first write operation (latent+active cycle); therefore, no other device should be driving the bus during this cycle.

Assembler Syntax:

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

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

Address Variants:

Address expression: An expression evaluating to a relocatable address:

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

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

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

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

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

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

Examples (Pre-Index):

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

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

Examples (Post-Index):

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

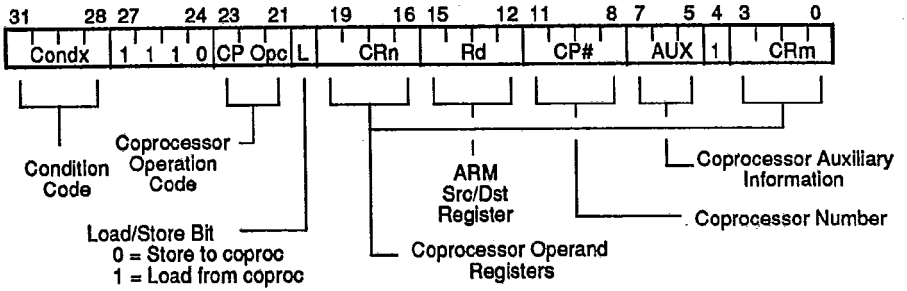
```
STC     3, CR1, [R2], 8!  ; *R2 = CR1. Then R2 += 8.
LDC     3, CR1, [R0], 16  ; CR1 = *R0. Then R0 += 16.
LDCEQL  3, CR2, [R5], 4   ; if (Zflag) CR2 = *R5, and then (implicitly), R5 += 4.
                          ; Use the long option (probably to store multiple words).
```

Examples (Expression):

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

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

FIGURE 19. COPROCESSOR REGISTER TRANSFERS (MRC, MCR)



This 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 VL86C020 processor. Instructions in this class are used to direct the coprocessor to perform some operation between a VL86C020 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 a VL86C020 register. An example of an MRC usage would be the converse: A FLOAT of a 32-bit value in a VL86C020 register into a floating point value within a coprocessor register.

An intended use of this instruction is to communicate control information directly between the coprocessor and the VL86C020 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 transferrred 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 VL86C020 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, and the PC and other PSR flags are unaffected by the transfer.

A coprocessor register transfer from VL86C020 with R15 as the source register will save the PC together with the PSR flags.

Assembler Syntax:

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

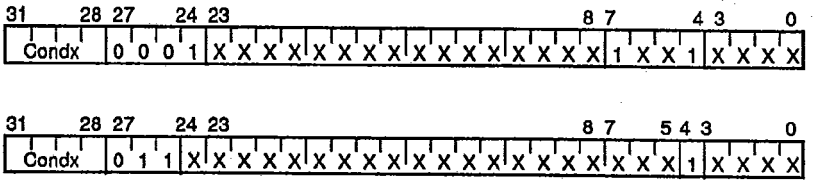
where	*cond*	Is the conditional execution code, common to all instructions.
	CP#	Is the (unique) coprocessor number, assigned by hardware.
	Rd	Is the ARM source or destination register.
	CRn, CRm	These are valid coprocessor registers: CR0-CR15.
	expression1	Evaluates to a constant, and is placed in the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

Examples:

```
MCR    1, 6, R1, CR7, CR2      ; Request coproc #1 to do operation 6 on
                               ; CR7 and CR2, putting result into VL86C020's R1.

MRCEQ 2, 5, R1, cr2, Cr3, 2    ; If the Z flag is set, transfer the VL86C020's R1 reg to the coproc register (defined
                               ; by hardware), and request coproc #2 to do oper 5 (type 2) on CR2 and CR3.
```

T-49-17-32

FIGURE 20. UNDEFINED (RESERVED) INSTRUCTION



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 it by taking CPA high.

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

Instruction Set Examples
The following examples show ways in which the basic VL86C020 instructions can combine to give efficient code. None of these methods save a great deal of execution time (although they may save some), mostly they just save code.

Using Conditional Instructions -

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

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

can be replaced by

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

(2) Absolute value

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

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

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

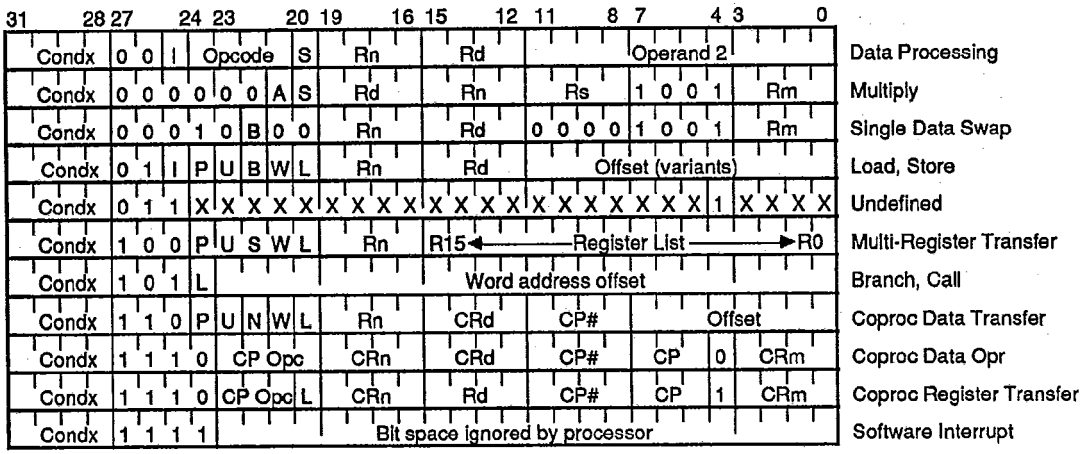
(4) Combining discrete and range tests

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

Division and Remainder

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


FIGURE 21. INSTRUCTION SET SUMMARY



Pseudo Random Binary Sequence Generator - It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift register-based generators with exclusive or feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition). The basic algorithim 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 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 4*11+1 = 45
```

rather than by:

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

Loading a Word with Unknown Alignment:

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

Sign Extension of Partial Word

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

Return, Setting Condition Codes

```
        BICS    PC, R14,CFLAG       ; Returns, clearing C flag ROM link register.
        ORRCCS  PC, R14, CFLAG      ; Conditionally returns, setting C flag.
; Above code should not be used except in user mode, since it will reset the interrupt enable flags to
; their value when R14 was set up. This generally applies to non-user mode programming.
; e.g., MOVS PC,R14        MOV PC,R14   is safer!
```

CACHE OPERATION

The VL86C020 contains a 4 Kbyte mixed instruction and data cache; the cache has 256 lines of 16 bytes (4 words), organized as four blocks of 64 lines (making it 64-way set associative), and uses the virtual addresses generated by the CPU core.

Read Operations - When the CPU performs a read operation (instruction fetch or data read), the cache is searched for the relevant data; if found in the cache, the data is fed to the CPU using a fast clock cycle (from FCLK). If the data is not found in the cache, the CPU resynchronizes to the external memory clock, MCLK, reads the appropriate line of data (4 words) from external memory and stores it in a pseudo-randomly chosen entry in the cache (a line fetch operation).

Write Operations - The cache uses a write-through strategy, i.e. all CPU write operations cause an immediate external memory write. This ensures that when the CPU attempts to write to a protected memory location, the memory manager can abort the operation.

If the cache holds a copy of the data from the address being written to, the cache data is normally automatically updated. In certain cases, automatic updating is not required; for instance, when using the MEMC memory manager, a read operation in the address space between 3400000H-3FFFFFFH accesses the ROMs, but a write operation in the same address space will change a MEMC register, and should not affect the data stored in the cache.

Control Register 4 must be programmed with the addresses of all updateable areas of the processor's memory map (see section Register 4: Updateable Areas Register - Read/Write).

Cache Validity - The cache works with virtual addresses, and is unaware of the mapping of virtual addresses to physical addresses performed by the external memory manager. If the virtual to physical mapping in the memory manager is altered, the cache still maintains the data from the old mapping which is now invalid. The cache must, therefore, be flushed of its old data whenever the memory manager mapping is changed.

Note that just removing or introducing a new virtual to physical mapping (e.g. page swapping) does not invalidate the cache, but that a total re-ordering of the mapping (e.g. process swap) does.

Two methods of cache flushing are supported:

1. Automatic cache flushing. Control Register 5 may be programmed to recognize write operations to certain areas of memory as re-programming the memory manager address mapping. (e.g. write operations to addresses between 3800000H-3FFFFFFH re-program the page mapping in MEMC). When the CPU sees a write operation to one of these disruptive memory locations, the cache is automatically flushed.
2. Software cache flushing. Writing to Control Register 1 will flush the cache immediately.

Automatic cache flushing invalidates the cache unnecessarily on page swaps, but allows all existing ARM programs to be run without modification.

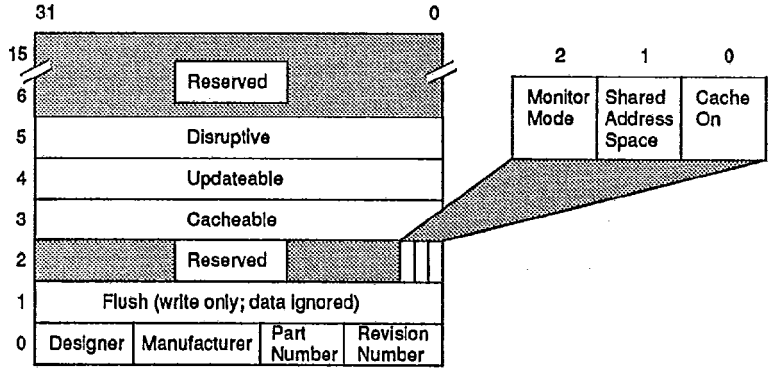
Non-cacheable Areas of Memory Certain areas of the processor's memory map may be uncacheable. For instance, when using MEMC, the area between 3000000H-3400000H corresponds to I/O space, and must be marked as uncacheable to stop the data being stored in the cache. When the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of some data held in the cache.

Control Register 3 must be programmed with the addresses of all cacheable areas of the processor's memory map (see section Register 3: Cacheable Area Register - Read/Write).

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

If, when using MEMC, the Physically Mapped RAM between 2000000H-2FFFFFFH is used to alter the contents of a cacheable virtual address, the cache must be flushed immediately afterwards. This may be performed automatically by marking the Physically Mapped RAM area as disruptive (see Register 5: Disruptive Areas Register).

FIGURE 22. VL86C020 CONTROL REGISTERS



The VL86C020 contains six control registers as shown in Figure 22. These registers are implemented as coprocessor 15, and are accessed using coprocessor register transfer operations, where MRC is a control register read, and MCR is a control register write:

<MCR/MRC>{cond} 15,0,Rd,A3Cn,0

cond	two character condition mnemonic, see section Condition Field.
Rd	is an expression evaluating to a valid ARM register number.
A3Cn	is an expression evaluating to one of the control register numbers.

These registers can only be accessed while the processor is in a non-user mode, and only by using coprocessor register transfer operations. The VL86C020 will take the undefined instruction trap if an illegal access is made to coprocessor 15 (illegal accesses include coprocessor data operations, data transfers and user mode register transfers).

Register 0: Identity Register - Read Only - This is a read-only register that returns a 32-bit VLSI-specified number which decodes to give the chip's designer, manufacturer, part type and revision number:

ID Example: (VL86C020 rev. 0)

Bit 31-Bit 24	Designer code	(=41H - Acorn Computer Ltd.)
Bit 23-Bit 16	Manufacturer code	(=56H - VLSI Technology Inc.)
Bit 15-Bit 8	Part type	(=03H - VL86C020)
Bit 7-Bit 0	Revision number	(=00H - Revision 0)

Register 1: Cache Flush (Write Only) Writing any value to this register immediately flushes the cache.

Register 2: Cache Control (Read/ Write) - This is a three-bit register that controls some special features of the VL86C020:

1. Register Bit(0) - Cache On/Off - If Bit(0) is low, the cache is turned off and all processor read operations will go directly to the external memory. The automatic cache flush and cache update mechanisms operate even when the cache is turned off. This allows the cache to be turned off for a time and then turned on again with no loss of cache consistency.

   If Bit(0) is high, the cache is turned on. Care must be taken that the cacheable, updateable and disruptive registers are correctly programmed before turning the cache on.

2. Register Bit(1) - Separate/Shared User-Supervisor Address Space - the CPU can work with two different memory-mapping schemes:
   a. Shared Supervisor/User Address Space - The memory manager uses the same translation tables for User and Supervisor modes, so the same physical memory location is accessed regardless of processor mode (although the user may only have restricted access). If the memory manager uses this translation system (as MEMC does), Bit(1) must be set high.
   b. Separate Supervisor/User Address Space - The memory manager uses different translation tables for user and supervisor modes, and the processor will access completely different physical locations depending on its mode. If the memory manager uses this translation system, Bit(1) must be set low.
3. Register Bit(2) - Monitor Mode - In normal operation, when the CPU is executing from cache, the external address lines are held static to conserve power, and only coprocessor instructions and data are broadcast on the coprocessor data bus.

   In the software selectable monitor mode, the internal addresses are always driven onto the external address bus, and all CPU instruction and data fetches (whether from cache or external memory) are broadcast on the coprocessor data bus; this allows full program tracing with a logic analyzer. To conserve power, monitor mode forces the VL86C020 to synchronize permanently to MCLK (even for cache accesses).

   Monitor mode is selected by setting Bit(2) high. Normal operation is achieved by setting Bit(2) low (the default on reset).
4. Register Bits 31-3 - Reserved - These bits are reserved for future expansion. When writing to register 2, bit 31-bit 3 should be set low to guarantee code compatibility with future versions of VL86C020. Reading from register 2 always returns zeros in bits 31-3.

When the VL86C020 is reset, all three control bits are set low (cache off, separate user/supervisor space, monitor mode off).

Register 3: Cacheable Area (Read/ Write) - This is a 32-bit register that allows any of the 32, 2 Mbyte areas of the 64 Mbyte processor virtual address space to be marked as cacheable:

Cacheable Areas Register:

Bit 31=1	Data from addresses 3E00000H - 3FFFFFFH is cacheable
Bit 31=0	Data from addresses 3E00000H - 3FFFFFFH is NOT cacheable
"	"
"	"
Bit 0=1	Data from addresses 0000000H - 01FFFFFH is cacheable
Bit 0=0	Data from addresses 0000000H - 01FFFFFH is NOT cacheable

On a cache-miss, if the address is marked as cacheable, a line of data will be fetched from external memory and stored in the cache (when the cache is turned on). If the area is marked as non-cacheable, or the cache is turned off, only the requested byte/word of data will be read from external memory, and it will not be stored in the cache. This register is undefined at power-up, and must be correctly programmed before the cache is turned on.

Register 4: Updateable Areas (Read/Write) - This is a 32-bit register that allows any of the 32, 2 Mbyte areas of the 64 Mbyte processor virtual address space to be marked as updateable:

Updateable Areas Register:

Bit 31=1	Data from addresses 3E00000H - 3FFFFFFH is updateable
Bit 31=0	Data from addresses 3E00000H - 3FFFFFFH is NOT updateable
"	"
"	"
Bit 0=1	Data from addresses 0000000H - 01FFFFFH is updateable
Bit 0=0	Data from addresses 0000000H - 01FFFFFH is NOT updateable

3

Data stored in the cache from areas marked as updateable will be updated when the processor writes new data to that address. This register is undefined at power-up, and must be correctly programmed before the cache is turned on.

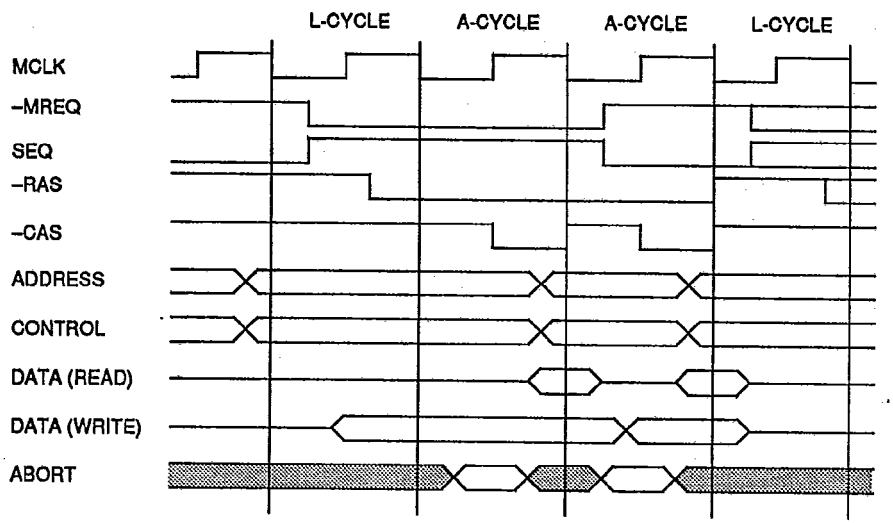
Register 5: Disruptive Areas (Read/Write) - This is a 32-bit register that allows any of the thirty-two, 2 Mbyte areas of the 64 Mbyte processor virtual address space to be marked as disruptive:

If the processor performs a write operation to an area marked as disruptive, the cache will automatically be flushed. This register is undefined at power-up, and must be correctly programmed before the cache is turned on.

Disruptive Areas Register:

Bit 31=1	Data from addresses 3E00000H - 3FFFFFFH is disruptive
Bit 31=1	Data from addresses 3E00000H - 3FFFFFFH is NOT disruptive
"	"
"	"
Bit 0=1	Data from addresses 0000000H - 01FFFFFH is disruptive
Bit 0=0	Data from addresses 0000000H - 01FFFFFH is NOT disruptive

FIGURE 23. VL86C020 MEMORY TIMING



MEMORY INTERFACE

The VL86C020 reads instructions and data from, and writes data to, its main memory via a 32-bit data bus. A separate 26-bit address bus specifies the memory location to be used for the transfer, and a 7-bit control bus gives information about the type of transfer (including direction, byte or word quantity and processor mode).

CYCLE TYPES

The memory interface timing is controlled by the memory clock input, MCLK. Each memory cycle (defined as the period between consecutive falling edges of MCLK) may be either active or latent.

- Active cycles (A-cycles) involve the transfer of data between CPU and memory. The address, control and (for write operations) data buses are valid, and the CPU monitors the ABORT input to check that the current operation is valid.

  Where more than one word of data is to be transferred, consecutive active cycles are used; in this case, each successive transfer will be to/from an address one word after the previous one. At the end of a multiple transfer, when the CPU wishes to access an address which is unrelated to the one used in the preceding cycle, it will request a latent cycle.

- Latent cycles (L-cycles) are flagged when the CPU does not have to transfer any data to/from memory. Typically, this will be because the CPU is fetching data from the internal cache; the CPU must still be clocked with MCLK during latent cycles, since MCLK is used in the resynchronization process.

  The address, control and (for write operations) data buses are all valid during the latent cycle preceding an active cycle; this allows the memory system to start the data transfer during the latent cycle as soon as the following active cycle is flagged (by –MREQ going low).

Active and latent cycles are flagged to the memory system using the –MREQ output. The SEQ output is the inverse of –MREQ, and is provided to allow the VL86C020 to work with the current versions of MEMC. The states encoded by –MREQ and SEQ correspond to the internal and sequential cycles used by the VL86C010 processor, and are shown in the following table.

–MREQ	SEQ	Cycle Type
0	0	(Unused)
0	1	Active
1	0	Latent
1	1	(Unused)

The memory interface has been designed to facilitate the use of DRAM page-mode to allow rapid access to sequential data. Figure 23 shows how the DRAM timing might be arranged to allow the CPU to access two consecutive words of memory.

The address and control signals change when MCLK is high, and apply to the following cycle. Both the address and control buses are valid during the L-cycle preceding the first A-cycle, so the memory system can start the DRAM access by driving –RAS low once the A-cycle has been flagged (by –MREQ being low on the rising edge of MCLK). Since –MREQ remains low during the first A-cycle, the memory system knows that the next cycle will be an access to the consecutive word of memory, and so may leave –RAS low and fetch the next word from the same page of DRAM. Note that the memory system must check that the consecutive access will be in the same page of DRAM before commiting to a page-mode access; if it is not, the memory system must stop the CPU while the new row address is strobed into the DRAM.

The end of the consecutive accesses is denoted when an L-cycle is flagged (by –MREQ being high on the rising edge of MCLK).

When interfacing the VL86C020 to static RAM, L-cycles may be ignored, and RAM accessed only when A-cycles are flagged. The address bus timing may have to be modified (see section on Address timing).

DATA TRANSFER

The direction of data transfer is determined by the state of –R/W.

When –R/W is low, the CPU is reading data from memory, and the appropriate data must be setup on the data bus before the falling edge of MCLK in the active cycle.

When –R/W is high, the CPU is writing data to memory. The data bus becomes valid during the first half of the L-cycle preceding the A-cycle, and remains valid until the A-cycle has completed. In consecutive write operations, the data bus changes during the first half of each A-cycle.

In systems where the VL86C020 is not the only device using the data bus, DBE must be driven low when the CPU is not the bus master. This will prevent the CPU from driving data onto the bus unexpectedly during L-cycles.

BYTE ADDRESSING

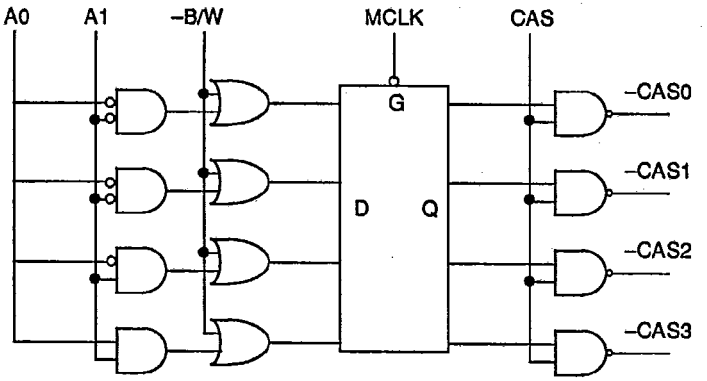
The processor address bus provides byte addresses, but instructions are always words (where a word is four bytes) and data quantities are usually words. Single data transfers (LDR,STR,SWP) can, however, specify that a byte quantity is required. The –B/W control line is used to request a byte from the memory system; normally it is high, signifying a request for a word quantity, but it goes low when the addresses change to request a byte transfer.

When a byte is requested in a read transfer, the memory system can safely ignore the fact that the request is for a byte quantity and present the whole word. The CPU will perform the byte extraction internally. Alternatively, the memory system may activate only the addressed byte of the memory. (This may be desirable in order to save power, or to enable the use of a common decoding system for both read and write cycles.)

If a byte write is requested, the CPU will broadcast the byte value across the data bus, presenting it at each byte location within the word. The memory system must decode address bits A1-A0 to determine which byte is to be written.

One way of implementing the byte decode in a DRAM system is to separate the 32-bit wide block of DRAM into four byte wide banks, and generate

FIGURE 24. BYTE ADDRESSING



the column address strobes independently. (See Figure 24.)

–CAS0 drives the DRAM bank which is connected to D7-D0, –CAS1 drives the bank connected to D15-D8, and so on. This has the added advantage of reducing the load on each column strobe driver, which improves the precision of this time critical signal.

LOCKED OPERATIONS

The VL86C020 includes a data swap (SWP) instruction that allows the contents of a memory location to be swapped with the contents of a processor register. This instruction is implemented as an uninterruptable pair of accesses as shown in Figure 25; the first access reads the contents of the memory, and the second writes the register data to the memory. These accesses must be treated as a contiguous operation by the memory manager to prevent another device from changing the affected memory location before the swap is completed. The CPU drives the LOCK signal high for the duration of the swap operation to warn the memory manager not to give the memory to another device.

3

FIGURE 25. DATA SWAP OPERATION

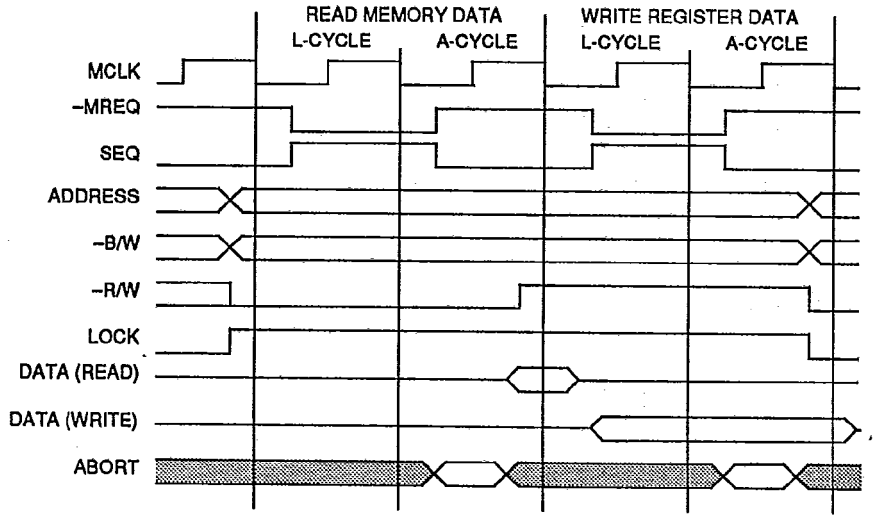
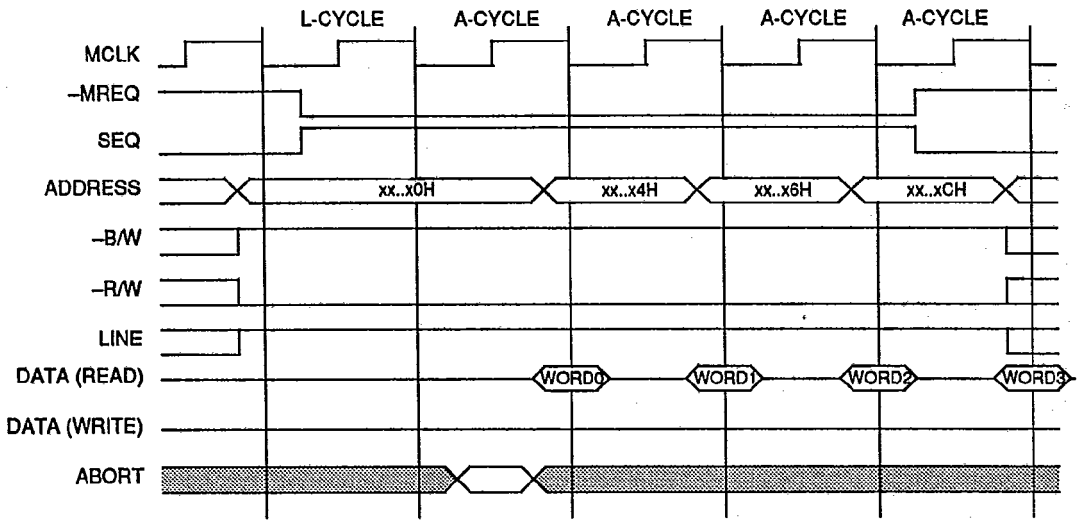


FIGURE 26. LINE FETCH OPERATION



LINE FETCH OPERATIONS
A line fetch operation involves reading exactly four words of data from the memory system into the on-chip cache. The access always starts on a quad-word aligned address (i.e. xx..x0H, xx..x4H or xx..xCH), and consists of one L-cycle followed by four consecutive A-cycles as shown in Figure 26. Line fetch operations may only be aborted during the first access (to address xx..x0H); it is assumed that if the first word of a line is readable, the whole line is readable. The VL86C020 signals a line fetch by driving LINE high for the duration of the five cycle operation.

ADDRESS TIMING
Normally the processor address changes when MCLK is high to the value which the memory system should use during the following cycle. This gives maximum time for driving the address to large memory arrays, and for address translation where required. Dynamic memories usually latch the address on chip, and if the latch is timed correctly, they will work even though the address changes before the access has completed. Static RAMs and ROMs will not work under such circumstances, as they require the address transition must be delayed until MCLK goes low. An on chip address latch, controlled by ALE, allows the address timing to be modified in this way.

In a system with a mixture of dynamic and static memories (which for these purposes means a mixture of devices with and without address latches), the use of ALE may change dynamically from one cycle to the next, at the discretion of the memory system.

VIRTUAL MEMORY SYSTEMS
The CPU is capable of running a virtual memory system, and the address bus may be processed by an address translation unit before being presented to the memory. The ABORT input to the processor is used by the memory manager to inform the processor of addressing faults.

The minimum page size allowed by the VL86C020 is four words (the length of a cache line). Various page protection levels can be suported using the VL86C020 control signals:

- –R/W can be used by the memory manager to protect pages from being written to.
- –TRANS indicates whether the processor is in a user or non-user mode, and may be used to protect system pages from the user, or to support completely separate mappings for the system and the user. In the latter case, the T bit in LDR and STR instructions can be used to offer the supervisor the user's view of the memory.
- –M1-M0 can present the memory manager with full information on the processor mode.

The cache control register must be programmed to implement the appropriate cache consistency mechanism depending on whether the memory manager uses a shared or separate user/non-user translation system (see Cache Operation Section).

STRETCHING ACCESS TIMES
All memory timing is defined by MCLK, and long access times can be accommodated by stretching this clock. It is usual to stretch the low period of MCLK, as this allows the memory manager to abort the operation if the access is eventually unsuccessful (ABORT must be setup to the rising edge of MCLK in A-cycles).

Either MCLK can be stretched before it is applied to the CPU, or the –WAIT input can be used together with a free-running MCLK. Taking –WAIT low has

the same effect as stretching the low period of MCLK, and –WAIT must only change when MCLK is low.

The VL86C020 contains dynamic logic, and relies upon regular clocking to maintain its internal state. For this reason, a limit is set upon the maximum period for which MCLK may be stretched, or –WAIT held low (see AC parameters).

COPROCESSOR INTERFACE

The functionality of the CPU instruction set may be extended by the addition of up to 15 external coprocessors. When a particular coprocessor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions. Adding the relevant coprocessor hardware will then increase the system performance in a software compatible way.

Interface Signals - The coprocessor interface timing is specified by CPCLK, a clock generated by the VL86C020. CPCLK is derived from either MCLK or FCLK depending on whether the CPU is accessing external memory or the cache; the coprocessors must, therefore, be able to operate at FCLK speeds. A coprocessor cycle is defined to be the period between consecutive falling edges of CPCLK. Three dedicated signals control the coprocessor interface, coprocessor instruction (–CPI), coprocessor absent (CPA) and coprocessor busy (CPB).

Coprocessor Present/Absent - The CPU takes –CPI low whenever it starts to execute a coprocessor (or undefined) instruction (this will not happen if the instruction fails to be executed because of the condition codes). Each coprocessor will have a copy of the instruction, and can inspect the CP# field to see which coprocessor it is for. Every coprocessor in a system must have a unique number, and if that number matches the contents of the CP# field, the coprocessor should pull the CPA (coprocessor absent) line low. If no coprocessor has a number which matches the CP# field, CPA will float high, and the CPU will take the undefined instruction trap. Otherwise, the VL86C020 observes the CPA line going low, and waits until the coprocessor flags that it is not busy (using CPB).

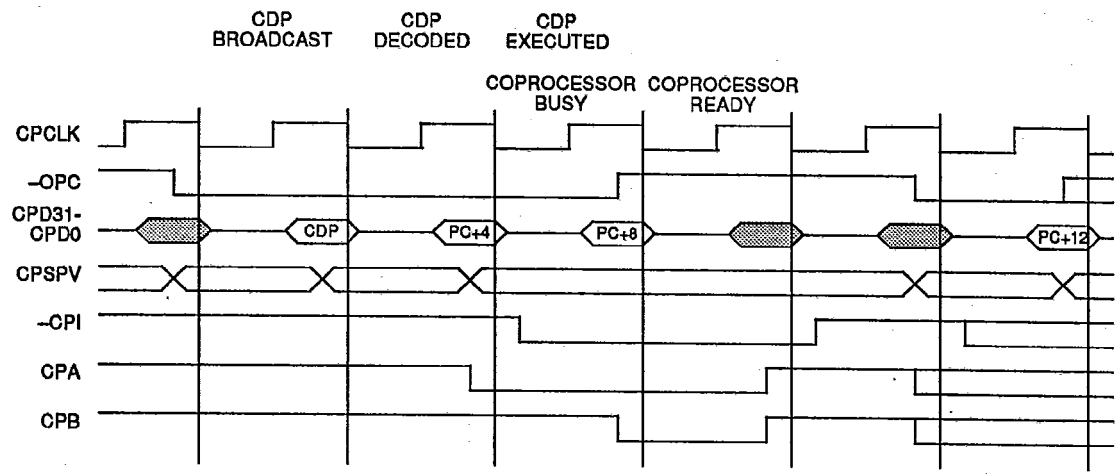
Busy-Waiting - If CPA goes low, the CPU will watch the CPB (coprocessor busy) line. Only the coprocessor which is pulling CPA low is allowed to drive CPB low, and it should do so when it is ready to complete the instruction. The VL86C020 will busy-wait while CPB is high, unless an enabled interrupt occurs, in which case it will break off from the coprocessor handshake to process the interrupt. Normally the CPU will return from processing the interrupt to retry the coprocessor instruction.

When CPB goes low, the instruction continues to completion; in the case of register transfer or data transfer instructions, this will involve data transfers taking place along the coprocessor data bus (CPD31-CPD0) between the coprocessor and CPU. Data operations do not transfer any data, and complete as soon as the coprocessor ceases to be busy.

All three interface signals are sampled by both CPU and the coprocessor(s) on the rising edge of CPCLK. If all three are low, the instruction is committed to execution, and where transfers are involved they will start in the next CPCLK cycle. If –CPI has gone high after being low, and before the instruction is committed, the VL86C020 has broken off from the busy-wait state to service an interrupt. The instruction may be restarted later, but other coprocessor instructions may come sooner, and the instruction should be discarded. An external pull-up resistor is normally required on both CPA and CPB.

3

FIGURE 27. COPROCESSOR DATA OPERATION



Pipeline Following - In order to respond correctly when a coprocessor instruction arises, each coprocessor must have a copy of the instruction. This is achieved by having each coprocessor maintain a copy of the processor's instruction pipeline. If –OPC is low when CPCLK is low, then the CPU will broadcast a processor instruction that cycle. The coprocessors should latch the instruction off CPD31-CPD0 at the end of the cycle (as CPCLK falls) and clock it into their instruction pipelines.

To reduce the number of transitions on CPD31-CPD0, the VL86C020 inspects the instruction stream and replaces all non coprocessor instructions with &FFFFFFFF (which still decodes as a non coprocessor instruction); all coprocessor instructions are broadcast unaltered.

This scheme is disabled when monitor mode is selected, and all CPU instructions and data fetches are broadcast unaltered (see Cache OperationSection).

DATA TRANSFER CYCLES - Once the coprocessor has gone no-busy in a data transfer instruction, it must supply or accept data at the VL86C020 bus rate (defined by CPCLK). The direction of transfer is defined by the L bit in the instruction being executed. The coprocessor is responsible for determining the number of words to be transferred; VL86C020 will continue to increment the address by one word per transfer until the coprocessor tells it to stop. The termination condition is

FIGURE 28. COPROCESSOR DATA TRANSFER (FROM MEMORY TO COPROCESSOR)

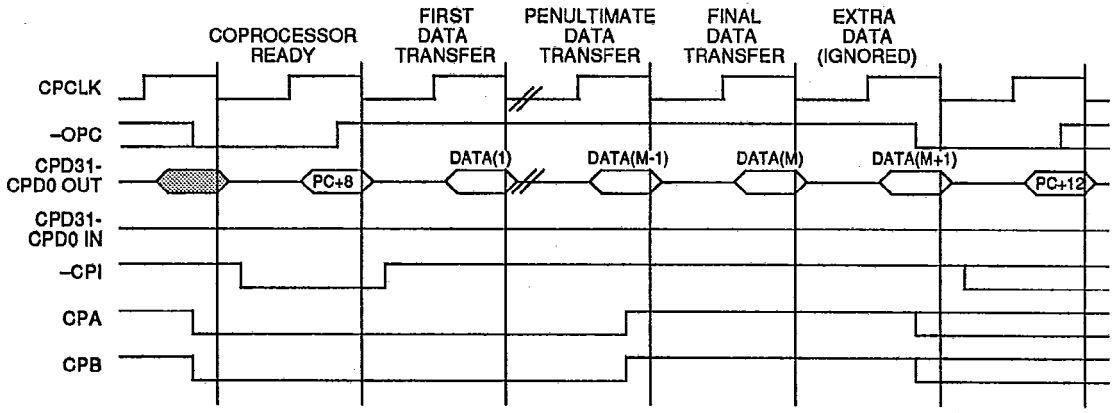
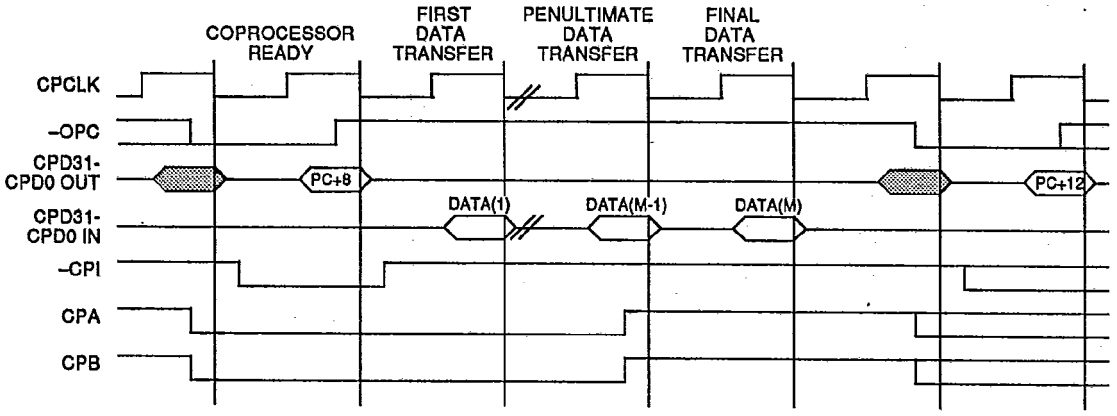


FIGURE 29. COPROCESSOR DATA TRANSFER (FROM COPROCESSOR TO MEMORY)



indicated by the coprocessor releasing CPA and CPB to float high.

The data being transferred to/from memory is pipelined by one cycle within the CPU. In the case of a coprocessor load from memory, this means that the CPU is one word ahead of the coprocessor, and always fetches one extra word of data. This extra fetch will not adversely affect the CPU or the coprocessor, but may cause unexpected faults in the memory system (e.g. if the extra fetch accesses a read-sensitive peripheral).

There is no limit in principle to the number of words which one coprocessor data transfer can move, but by convention no coprocessor should allow more than 16 words in one instruction. More than this would worsen the worst case CPU interrupt latency, since the instruction is not interruptable once the transfers have commenced. At 16 words, this instruction is comparable with a block transfer of 16 registers, and therefore does not affect the worst case latency.

REGISTER TRANSFER CYCLE

Register transfer operations involve the transfer of a single word between the CPU and the appropriate coprocessor along CPD31-CPD0. The transfer takes place in the cycle after the one in which the CPU and the coprocessor committed to the instruction.

PRIVILEGED INSTRUCTIONS

The coprocessor may restrict certain instructions for use in a privileged (non-user) mode only. To do this, the coprocessor may use the CPSPV

3

FIGURE 30. COPROCESSOR REGISTER TRANSFER (LOAD FROM COPROCESSOR)

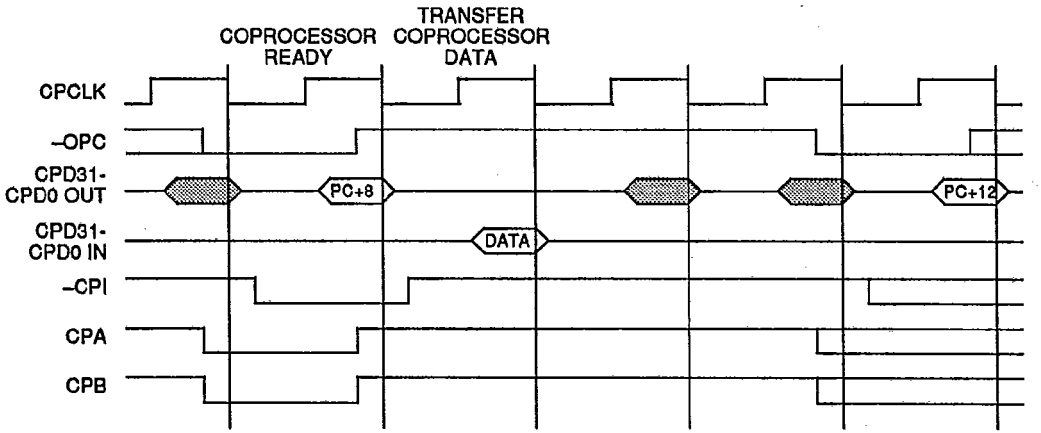
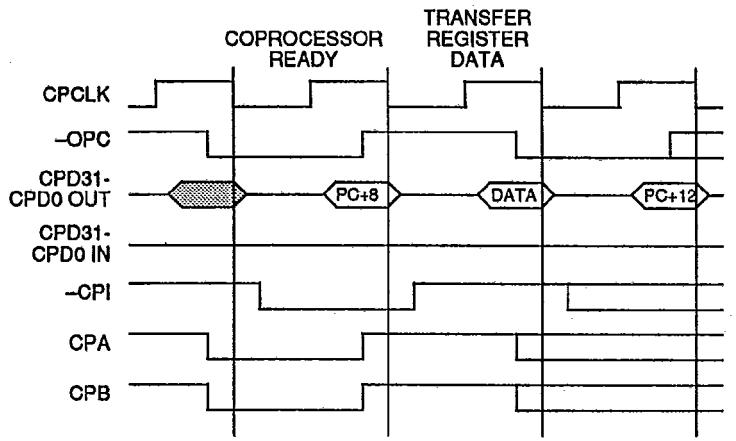


FIGURE 31. COPROCESSOR REGISTER TRANSFER (STORE TO COPROCESSOR)



output of the VL86C020; this signal is valid while CPCLK is low, and applies to the instruction being broadcast during that cycle. When CPSPV is high, the broadcast instruction is privileged.

As an example of the use of this facility, consider the case of a floating point coprocessor (FPU) in a multi-tasking system. The operating system could save all the floating point registers on every task switch, but this is inefficient in a typical system where only one or two tasks will use floating point operations. Instead, there could be a privileged instruction which turns the FPU on or off. When a task switch happens, the operating system can turn the FPU off without saving its registers. If the new task attempts an FPU operation, the FPU will appear to be absent, causing an undefined instruction trap. The operating system will then realize that the new task requires the FPU, so it will re-enable it and save FPU registers. The task can then use the FPU as normal. If, however, the new task never attempts an FPU operation (as will be the case for most tasks), the state saving overhead will have been avoided.

REPEATABILITY

A consequence of the implementation of the coprocessor interface, with the interruptable busy-wait state, is that all instructions may be interrupted at any point up to the time when the coprocessor goes not-busy. If so interrupted, the instruction will normally be restarted from the beginning after the interrupt has been processed. It is, therefore, essential that any action taken by the coprocessor before it goes not-busy must be repeatable, i.e. must be repeatable with identical results.

For example, consider a FIX operation in a floating point coprocessor which returns the integer result to a CPU register. The coprocessor must stay busy while it performs the floating point to fixed point conversion, as the CPU will expect to receive the integer value on the cycle immediately following that where it goes not-busy. The coprocessor must, therefore, preserve the original floating point value and not corrupt it during the conversion because it will be required again if an interrupt occurred during the busy period.

The coprocessor data operation class of instruction is not generally subject to repeatablity considerations, as the processing activity can take place after the coprocessor goes not-busy. There is no need for the CPU to be held up until the result is generated, because the result is confined to stay within the coprocessor.

UNDEFINED INSTRUCTION

The undefined instruction is treated by the CPU as a coprocessor instruction. All coprocessors must be absent (i.e. let CPA float high) when the undefined instruction is presented. The CPU will then take the undefined instruction trap. Note that the coprocessor need only look at bit 27 of the instruction to differentiate the undefined instruction (which has 0 in bit 27) from coprocessor instructions (which all have 1 in bit 27).

VL86C020 INSTRUCTION CYCLES

This section shows the cycles performed by the VL86C020's CPU and coprocessor for all possible instructions. Each class of instruction is taken in turn, and its operation is broken down into constituent cycles.

EXPLANATION OF INSTRUCTION TABLES

Example:

Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0	–CPI	CPA	CPB
1	Read		PC+8	(PC+8)			1	x	x
2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
3	Intnl	–	<= not clocked =>		1	DI (1)	1	1	1
4	Write	N	ALU	DI(1)	<= not clocked =>				
	Read	N	PC+12		1	–	1\)		

Each row in the table represents a single CPU or coprocessor cycle. The cycles which constitute the instruction are numbered from 1 to n.

The OPRTN column shows the CPU operation being performed in each cycle. There are four types of CPU operation as follows:

1. Read: A CPU read operation; the data will be read from the cache if it is present, otherwise an external read or line fetch operation will be necessary.
2. Write: A CPU write operation; VL86C020 always writes data immediately to the main memory.
3. Intnl: An internal operation where the CPU is not transferring data.
4. Trnsf: A coprocessor register transfer where data passes between the CPU and a coprocessor.

The type column gives extra information about the type of operation being performed:

1. Read and write operations may be one of two types, Sequential ("S") or Non-sequential ("N"). A sequential access involves the CPU transferring data with an address that is one word after the preceding access. A non-sequential access is flagged when the current CPU address is unrelated to the one used in the preceding access.
2. Read and write operations normally work on word quantities, but the single data load, store and

swap instructions allow byte quantities to be specified; this is indicated by the symbol "(B/W)" in the type column.

3. The coprocessor register transfer instruction may either transfer data into ("I") or out from ("O") the CPU.

The address and data columns show the contents of VL86C020's internal address and data busses. Note that in normal mode, the internal data bus cannot be observed directly, and the address bus is only observable when the CPU is synchronized to MCLK.

The –OPC, CPD31-CPD0, –CPI, CPA and CPB columns (where shown) indicate the state of the external coprocessor interface. Note that in normal mode CPD31-CPD0 only broadcasts coprocessor instructions and data (see section Pipeline Following). By selecting monitor mode, the internal address bus can be viewed on A25-A0, and all data will be broadcast on CPD31-CPD0.

The final, un-numbered operation in an instruction shows what will happen in the first cycle of the next instruction. Note that the first cycle of an instruction is always an instruction fetch (word read operation), but may be either an N-type or S-type read depending on the previous instruction.

INSTRUCTION TABLES

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

During the second cycle a fetch is performed from the branch destination, and the return address is stored in register 14 if the link bit is set. The first cycle's prefetch data is broadcast on the external coprocessor data bus (there is a one cycle delay between the coprocessor and CPU).

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

3

Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0
1	Read		PC+8	(PC+8)		
2	Read	N	ALU	(ALU)	0	(PC+8)
3	Read	S	ALU+4	(ALU+4)	0	(ALU)
	Read	S	ALU+8		0	(ALU+4)

(PC is the address of the branch instruction, ALU is an address calculated by the CPU, (ALU) is the contents of the 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 affected.)

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 8 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 perform a data transfer).

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 activitiy. When it is the destination, however, 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.

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0
Normal	1	Read		PC+8	(PC+8)		
		Read	S	PC+12		0	(PC+8)
DEST=PC	1	Read		PC+8	(PC+8)		
	2	Read	N	ALU	(ALU)	0	(PC+8)
	3	Read	S	ALU+4	(ALU+4)	0	(ALU+4)
		Read	S	ALU+8		0	(ALU+4)
Shift (RS)	1	Read		PC+8	(PC+8)		
	2	Intnl	–	PC+12	–	0	(PC+8)
		Read	N	PC+12		1	–
Shift (RS), DEST=PC	1	Read		PC+8	(PC+8)		
	2	Intnl	–	–	–	0	(PC+8)
	3	Read	N	ALU	(ALU)	1	–
	4	Read	S	ALU+4	(ALU+4)	0	(ALU)
		Read	S	ALU+8		0	(ALU+4)

Multiply and Multiply Accumulate - The multiply instructions make use of special hardware which implements a 2-bit Booth's algorithm with early termination. During the first cycle the accumulate register is brought to the ALU, which either transmits it or produces zero (according to whether the instruction is MLA or MUL) to initialize the destination register. During the same cycl, 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 algorithm logic. The first operand is shifted right 2 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 PSR flags will be meaningless.)

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0
(RS) = 0, 1	1	Read		PC+8	(PC+8)		
	2	Intnl	–	PC+12	–	0	(PC+8)
		Read	N	PC+12	–	1	–
(RS) >1	1	Read		PC+8	(PC+8)		
	2	Intnl	–	PC+12	–	0	(PC+8)
	•	Intnl	–	PC+12	–	1	–
	m+1	Intnl	–	PC+12	–	1	–
		Read	N	PC+12		1	–

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

Load Register - The first cycle of a load register instruction performs the address calculation. The data is fetched 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 the CPU performs an internal cycle.

The data read may be a byte or word quantity (B/W), and the processor mode may be forced into user mode while the read takes place (depending on the state of the T bit in the instruction).

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.

	Cycle	OPRTN	Type	Mode	Address	Data	–OPC	CPD31-CPD0
Normal	1	Read			PC+8	(PC+8)		
	2	Read	N (B/W)	T	ALU	(ALU)	0	(PC+8)
	3	Intnl	–		PC+12	–	1	(ALU)
		Read	N		PC+12		1	–
DEST=PC	1	Read			PC+8	(PC+8)		
	2	Read	N (B/W)	T	ALU	(ALU)	0	(PC+8)
	3	Intnl	–		PC+12	–	1	(ALU)
	4	Read	N		(ALU)	((ALU))	1	–
	5	Read	S		(ALU)+4	((ALU)+4)	0	((ALU))
		Read	S		(ALU)+8		0	((ALU)+4)
BASE=PC	1	Read			PC+8	(PC+8)		
Write Back	2	Read	N (B/W)	T	ALU	(ALU)	0	(PC+8)
DEST=PC	3	Intnl	–		PC'	–	1	(ALU)
	4	Read	N		PC'	(PC')	1	–
	5	Read	S		PC'+4	(PC'+4)	0	(PC')
		Read	S		PC'+8		0	(PC'+4)
BASE=PC	1	Read			PC+8	(PC+8)		
Writ -Back	2	Read	N (B/W)	T	ALU	(ALU)	0	(PC+8)
DEST=PC	3	Intnl	–		PC'	–	1	(ALU)
	4	Read	N		(ALU)	((ALU))	1	–
	5	Read	S		(ALU)+4	((ALU)+4)	0	((ALU))
		Read	S		(ALU)+8		0	((ALU)+4)

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

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 external memory. There is no third cycle.

The data written may be a byte or word quantity (B/W), and the processor mode may be forced into user mode while the write takes place (depending on the state of the T bit in the instruction).

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

A data abort prevents the base write back.

	Cycle	OPRTN	Type	Mode	Address	Data	–OPC	CPD31-CPD0
Normal	1	Read			PC+8	(PC+8)		
	2	Write	N (B/W)	T	ALU	RD	0	(PC+8)
		Read	N		PC+12		1	RD
BASE=PC	1	Read			PC+8	(PC+8)		
Write Back	2	Write	N (B/W)	T	ALU	RD	0	(PC+8)
	3	Read	N		PC'	(PC')	1	RD
	4	Read	S		PC'+4	(PC'+4)	0	(PC')
		Read	S		PC'+8		0	(PC'+4)

Store Multiple Registers - Store multiple proceeds very much as load multiple (see next section), without the final cycle. The restart problem is much more straightforward here, as there is no wholesale overwriting of registers to contend with.

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0
1 Register	1	Read		PC+8	(PC+8)		
	2	Write	N	ALU	R(A)	0	(PC+8)
		Read	N	PC+12		1	R(A)
n Registers	1	Read		PC+8	(PC+8)		
(n>1)	2	Write	N	ALU	R(A)	0	(PC+8)
	3	Write	S	ALU+4	R(A+1)	1	R(A)
	•	•	•	•	•	•	•
	n+1	Write	S	ALU+.	R(A+n)	1	R(A+n-1)
		Read	N	PC+12		1	R(A+n)

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. The second cycle fetches the first word, and performs the base modifications. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched, and the modification base is moved to the ALU A bus input latch for holding in case it is needed to patch up after 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.

If an abort occurs, the instruction continues to completion, but all register writing after the abort is prevented. 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.

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0
1 Register	1	Read		PC+8	(PC+8)		
	2	Read	N	ALU	(ALU)	0	(PC+8)
	3	Intnl	–	PC+12	–	1	(ALU)
		Read	N	PC+12		1	–
1 Register	1	Read	N	PC+8	(PC+8)		
DEST=PC	2	Read	N	ALU	PC'	0	(PC+8)
	3	Intnl	–	PC+12	–	1	PC'
	4	Read	N	PC'	(PC')	1	–
	5	Read	S	PC'+4	(PC'+4)	0	(PC')
		Read	S	PC'+8	(PC'+8)	0	(PC'+8)
n Registers	1	Read		PC+8	(PC+8)		
(n>1)	2	Read	N	ALU	(ALU)	0	(PC+8)
	•	Read	S	ALU+.	(ALU+.)	1	(ALU)
	n+1	Read	S	ALU+.	(ALU+.)	1	(ALU+.)
	n+2	Intnl	–	PC+12	–	1	(ALU+.)
		Read	N	PC+12		1	–
n Registers	1	Read		PC+8	(PC+8)		
(n>1)	2	Read	N	ALU	(ALU)	0	(PC+8)
incl. PC	•	Read	S	ALU+.	(ALU+.)	1	(ALU)
	n+1	Read	S	ALU+.	PC'	1	(ALU+.)
	n+2	Intnl	–	PC+12	–	1	PC'
	n+3	Read	N	PC'	(PC')	1	–
	n+4	Read	S	PC'+4	(PC'+4)	0	(PC')
		Read	S	PC'+8	(PC'+8)	0	(PC'+8)

Data Swap - This is similar to the load and store register instructions, but the actual swap takes place in cycles two and three. In the second cycle, the data is fetched from external memory (it is always read from the external memory, even if the data is available in the cache). In the third cycle, the contents of the source register are written out to the external memory. The data read in cycle two is written into the destination register during the fourth cycle.

The LOCK output of the VL86C020 is driven high for the duration of the swap operation (cycles two and three) to indicate that both cycles should be allowed to complete without interruption.

The data swapped may be a byte or word quantity (B/W).

The prefetch sequence will be changed if the PC is specified as the destination register.

When R15 is selected as the base, the PC is used together with the PSR. If any of the flags are set, or interrupts are disabled, the data swap will cause an address exception. If all flags are clear, and interrupts are enabled (so the top six bits of the PSR are clear), the data will be swapped with an address eight bytes advanced from the swap instruction (PC+8), although the address will not be word aligned unless the processor is in user mode (as the M1 and M0 bits determine the byte address).

The swap operation may be aborted in either the read or write cycle, and in both cases the destination register will not be affected.

3

	Cycle	OPRTN	Type	Lock	Address	Data	–OPC	CPD31-CPD0
Normal	1	Read		0	PC+8	(PC+8)		
	2	Read	N (B/W)	1	RN	(RN)	0	(PC+8)
	3	Write	N (B/W)	1	RN	RM	1	(RN)
	4	Intnl	–	0	PC+12	–	1	RM
		Read	N	0	PC+12		1	–
DEST=PC	1	Read		0	PC+8	(PC+8)		
	2	Read	N (B/W)	1	RN	PC'	0	(PC+8)
	3	Write	N (B/W)	1	RN	RM	1	PC'
	4	Intnl	–	0	PC+12	–	1	RM
	5	Read	N	0	PC'	(PC')	1	–
	6	Read	S	0	PC'+4	(PC'+4)	0	(PC')
		Read	S	0	PC'+8		0	(PC'+4)

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 the processor enters supervisor mode. 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.

Cycle	OPRTN	Type	Mode	Address	Data	–OPC	CPD31-CPD0
1	Read			PC+8	(PC+8)		
2	Read	N	SPV	XN	(XN)	0	(PC+8)
3	Read	S	SPV	XN+4	(XN+4)	0	(XN)
	Read	S	SPV	XN+8		0	(XN+4)

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

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

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

The coprocessor interface normally operates one cycle behind the CPU to allow time for the instructions to be broadcast. When the CPU starts executing a coprocessor instruction, it busy-waits for one cycle (Cycle 2) while the coprocessor catches up.

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0	–CPI	CPA	CPB
Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
		Read	N	PC+12		1	–	1		
Not Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	1
	•	Intnl	–	PC+8	–	1	–	0	0	1
	n	Intnl	–	PC+8	–	1	–	0	0	0
		Read	N	PC+12		1	–	1		

Coprocessor Data Transfer - Here, the coprocessor should commit to the transfer only when it is ready to accept the data. When CPB goes low, the CPU will read the appropriate data and broadcast it to the coprocessor (if the data is read from the cache, it will be broadcast at FCLK rates). Note that the coprocessor is not clocked while the CPU fetches the first word of data; the data is broadcast to the coprocessor in the next cycle.

During the data transfer, the VL86C020 operates one cycle ahead of the coprocessor, and so always fetches one word more than the coprocessor wants. This extra data is simply discarded.

The coprocessor is responsible for determining the number of words to be transferred, and indicates the last transfer cycle by allowing CPA and CPB to float high.

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

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0	–CPI	CPA	CPB
1 Register Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
	3	Read	N	ALU	DO(1)	<= not clocked =>			1	1
		Read	N	PC+12		1	DO(1)	1		
1 Register Not Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	1
	•	Intnl	–	PC+8	–	1	–	0	0	1
	n	Intnl	–	PC+8	–	1	–	0	0	0
	n+1	Read	N	ALU	DO(1)	<= not clocked =>			1	1
m Registers (m>1) Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
	3	Read	N	ALU	DO(1)	<= not clocked =>			0	0
	4	Read	S	ALU+4	DO(2)	1	DO(1)	1	0	0
	•	•	•	•	•	•	•	•	•	•
	m+3	Read	S	ALU+.	DO(m+1)	1	DO(m)	1	1	1
		Read	N	PC+12		1	DO(m+1)	1		

m Registers	1	Read		PC+8	(PC+8)			1	x	x
(m>1)	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
Not Ready	•	Intnl	–	PC+8	–	1	–	0	0	1
	n	Intnl	–	PC+8	–	1	–	0	0	0
	n+1	Read	N	ALU	DI(1)	<= not clocked =>			0	0
	n+2	Read	S	ALU+4	DI(2)	1	DI(1)	1	0	0
	•	•	•	•	•	•	•	•	•	•
	n+m+2	Read	S	ALU+.	DI(m+1)	1	DI(m)	1	1	1
		Read	N	PC+12		1	DI(m+1)	1		

Coprocessor Data Transfer (from Coprocessor to Memory) - This instruction is similar to the memory to coprocessor data transfer. In this case, however, the VL86C020 operates one cycle behind the coprocessor during the data transfer to give time for data to get through the coprocessor interface. The CPU is halted for a cycle at the start of the transfer while the coprocessor outputs the first word of data, and at the end of the transfer, the coprocessor is halted for one cycle while the CPU writes the last word of data to memory.

3

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD 31-CPD0	–CPI	CPA	CPB
1 Register	1	Read		PC+8	(PC+8)			1	x	x
Ready	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
	3	Intnl	–	<= not clocked =>		1	DI(1)	1	1	1
	4	Write	N	ALU	DI(1)	<= not clocked =>			1	1
		Read	N	PC+12		1	–	1		
1 Register	1	Read		PC+8	(PC+8)			1	x	x
Not Ready	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	1
	•	Intnl	–	PC+8	–	1	–	0	0	1
	n	Intnl	–	PC+8	–	1	–	0	0	0
	n+1	Intnl	–	<= not clocked =>		1	DI(1)	1	1	1
	n+2	Write	N	ALU	DI(1)	<= not clocked =>			1	1
		Read	N	PC+12		1	–	1		
m Registers	1	Read		PC+8	(PC+8)			1	x	x
(m>1)	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
Ready	3	Intnl	–	<= not clocked =>		1	DI(1)	1	0	0
	4	Write	N	ALU	DI(1)	1	DI(2)	1	0	0
	•	•	•	•	•	•	•	•	•	•
	m+2	Write	S	ALU+.	DI(m-1)	1	DI(m)	1	1	1
	m+3	Write	S	ALU+.	DI(m)	<= not clocked =>			1	1
		Read	N	PC+12		1	–	1		
m Registers	1	Read		PC+8	(PC+8)			1	x	x
(m>1)	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	1
Not Ready	•	Intnl	–	PC+8	–	1	–	0	0	1
	n	Intnl	–	PC+8	–	1	–	0	0	0
	n+1	Intnl	–	<= not clocked =>		1	DI(1)	1	0	0
	n+2	Write	N	ALU	DI(1)	1	DI(2)	1	0	0
	•	•	•	•	•	•	•	•	•	•
	m+n	Write	S	ALU+.	DI(m-1)	1	DI(m)	1	1	1
	m+n+1	Write	S	ALU+.	DI(m)	<= not clocked =>			1	1
		Read	N	PC+12		1	–	1		

Coprocessor Register Transfer (Load from Coprocessor) - Here the busy-wait cycles are similar to the previous transfer cycle, but the transfer is limited to one data word, and VL86C020 puts the word into the destination register in the third cycle.

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0	–CPI	CPA	CPB
Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
	3	Intnl	–	<= not clocked =>		1	DI	1	1	1
	4	Trnsf	I	PC+12	DI	<= not clocked =>			1	1
	5	Intnl	–	PC+12	–	1	–	1	1	1
		Read	N	PC+12		1	–	1		
Not Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	1
	•	Intnl	–	PC+8	–	1	–	0	0	1
	n	Intnl	–	PC+8	–	1	–	0	0	0
	n+1	Intnl	–	<= not clocked =>		1	DI	1	1	1
	n+2	Trnsf	I	PC+12	DI	<= not clocked =>			1	1
	n+3	Intnl	–	PC+12	–	1	–	1	1	1
		Read	N	PC+12		1	–	1		

Coprocessor Register Transfer (Store to Coprocessor) - This instruction is similar to a single word coprocessor data transfer.

	Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0	–CPI	CPA	CPB
Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	0
	3	Trnsf	O	PC+12	DO	<= not clocked =>			1	1
		Read	N	PC+12		1	DO	1		
Not Ready	1	Read		PC+8	(PC+8)			1	x	x
	2	Intnl	–	PC+8	–	0	(PC+8)	0	0	1
	•	Intnl	–	PC+8	–	1	–	0	0	1
	n	Intnl	–	PC+8	–	1	–	0	0	0
	n+1	Trnsf	O	PC+12	DO	<= not clocked =>			1	1
		Read	N	PC+12		1	DO	1		

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

	Cycle	OPRTN	Type	Mode	Address	Data	–OPC	CPD31-CPD0	–CPI	CPA	CPB
Ready	1	Read			PC+8	(PC+8)			1	x	x
	2	Intnl	–		PC+8	–	0	(PC+8)	0	1	1
	3	Read	N	SPV	Xn	(Xn)	0	(PC+8)	1	1	1
	4	Read	S	SPV	Xn+4	(Xn+4)	0	(Xn)	1	1	1
		Read	S	SPV	Xn+8		0	(Xn+4)			

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.

Cycle	OPRTN	Type	Address	Data	–OPC	CPD31-CPD0
1	Read		PC+8	(PC+8)		
	Read	S	PC+12	–	0	(PC+8)

Instruction Speeds - In order to determine the time taken to execute any given instruction, it is necessary to relate the CPU read, write, internal and transfer operations to F-cycles (FCLK cycles), L-cycles (Latent MCLK cycles) and A-cycles (Active MCLK cycles).

The relationship between the CPU operations and external clock cycles depends primarily upon whether the cache is turned off or on.

Cache Off - When the cache is turned off, CPU read and write cycles always access external memory. To avoid unnecessary synchronization delay VL86C020 remains synchronized to the external memory when the cache is turned off, so all operations are timed by MCLK. The time taken for each type of CPU operation is as follows:

Operation	Time
N-type Read	L + A
S-type Read	A
N-type Write	L + A
S-type Write	A
Transfer In	L
Transfer Out	L
Internal	L

Key:
L - Latent memory cycle period
A - Active memory cycle period

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the datapath 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 a routine may be calculated from these figures.

Note: This table only applies when the cache is turned off.

If the condition is met the instructions take:

B,BL	1 L + 3 A		
Data Processing	1A	+ 2 L	for SHIFT(Rs)
		+ 1 L + 2 A	if R15 written
MUL,MLA	(m+1) L + 1 A		
LDR	3 L + 2 A	+ 2 A	if R15 loaded/written back
STR	2 L + 2 A	+ 2 A	if R15 written-back
LDM	3 L + (n+1)A	+ 2 A	if R15 loaded
STM	2 L + (n+1)A		
SWP	4 L + 3 A	+ 2 A	if R15 loaded
SWI, trap	1 L + 3 A		
CDO	(b+2) L + 1 A		
LDC	(b+3) L + (n+1)A	+ 1 A	if (n>1)
STC	(b+4) L + (n+1)A		
MRC	(b+4) L + 1 A		
MCR	(b+3) L + 1 A		

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 zero or one takes one cycle. The maximum value m can take is 16.

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

If the condition is not met all instructions take one A-cycle.

Cache On - When the cache is turned on, the CPU will synchronize to FCLK, and attempt to fetch instructions and data from the cache (using FCLK F-cycles). When the read data is not available, or the CPU performs a write operation, the VL86C020 resynchronizes to MCLK and accesses the external memory (using L & A-cycles). The CPU operations are dealt with as follows:

1. Read operations. The CPU will normally be able to read the relevant data from the cache, in which case the read will complete in a single F-cycle.

   If the data is not present in the cache, but is cacheable, the CPU will synchronize to MCLK and perform a line fetch to read the appropriate line (four words) of data into the cache. The CPU will be clocked when the appropriate word is fetched, and subsequently during the line fetch if it is requesting S-type reads or internal operations.

   If the data is not cacheable, the CPU will synchronize to MCLK and perform an external read. If the CPU requests S-type reads, the CPU will remain synchronized to MCLK and use A-cycles to read the appropriate data. The CPU only resynchronizes back to FCLK when the CPU stops requesting S-type reads.

   Note that the swap instruction bypasses the cache, and always performs an external read to fetch the data from external memory.

2. Write operations. The VL86C020 synchronizes to MCLK and performs external writes. When the CPU stops requesting S-type writes, VL86C020 resynchronizes to FCLK.

3. Internal operation. These complete in a single F-cycle (although some are absorbed during line fetches).

4. Transfer operation. These complete in a single F-cycle.

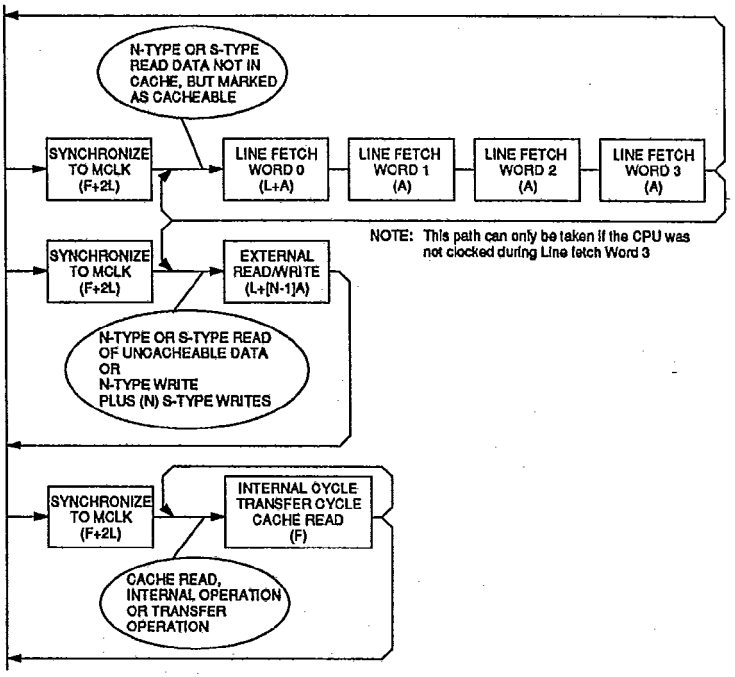
It is not possible to give a table of instruction speeds, as the time taken to execute a program depends on its interaction with the cache (which includes factors such as code position, previous cache state, etc.). In general, programs will execute much faster with the cache turned on than with it turned off.

To calculate the worst-case delay for a particular piece of code, the routine should be written out in terms of CPU cycles. Figure 33 can then be used to calculate the worst-case VL86C020 operation for each CPU cycle.

When using this technique, the following conditions must be assumed:

1. No instructions or data are present in the cache when VL86C020 starts executing the code.
2. A line fetch operation will overwrite any data already present in the cache (i.e., the cache only has one line).
3. All synchronization cycles take the maximum time.

FIGURE 33. WORST-CASE VL86C020 TIMING FLOWCHART



Line Fetch Operation
The CPU is clocked as soon as the requested word of data is available. The CPU will also be clocked if it subsequently requests S-type Read or Internal operations during the remainder of the line fetch.

EXAMPLE:

Consider the following piece of code:

```
;
;   Asssume code runs in a cacheable area of memory, and that
;   Code, Area1 and Area2 are all quad-word aligned addresses.
;
Code
    MOV     R0,Area1        R0 points to data in a cacheable area of memory
    MOV     R1,Area2        R1 points to data in an uncacheable area of memory
    LDR     R7, R0,4        Read data from cacheable area into R7
    LDMIA   R1, {R8-R9}     Read data from uncacheable area into R8 and R9
End
```

Converting the code into CPU cycles gives:

		Cycle	OPRTN Type		Address	Data
		1.0	Read		PC+8	(PC+8) (see Note)
Branch to Code		1.1	Read	N	Code	(Code)
		1.2	Read	S	Code +4	(Code+4)
MOV	R0,Area1	2.1	Read	S	Code+8	(Code+8)
MOV	R1,Area2	3.1	Read	S	Code+12	(Code+12)
LDR	R7,[R0,4]	4.1	Read	S	Code+16	(Code+16)
		4.2	Read	N	Area1+4	(Area1+4)
		4.3	Intnl	–	Code+20	–
LDMIA	RI, {R8-R9}	5.1	Read	N	Code+20	(Code+20)
		5.2	Read	N	Area2	(Area2)
		5.3	Read	S	Area2+4	(Area2+4)
		5.4	Intnl	–	Code+24	–

Note: Cycle 1.0 is the last cycle before the routine is entered, and is not counted as part of the code.

Using the worst-case VL86C020 timing flowchart, the required CPU operations can be converted into CPU operations, and assigned an execution time.

	CPU Operation	VL86C020 Operation	Time
	<wait>	Synchronize to MCLK	(F+2L)
1.1:	Read N (Code)	Line Fetch: (Code)	(L+A)
1.2:	Read S (Code+4)	(Code+4)	(A)
2.1:	Read S (Code+8)	(Code+8)	(A)
3.1:	Read S (Code+12)	(Code+12)	(A)
	<wait>	Synchronize to MCLK	(F+2L)
4.1:	Read S (Code+16)	Line Fetch: (Code+16)	(L+A)
	<wait>	(Code+20)	(A)
	<wait>	(Code+24)	(A)
	<wait>	(Code+28)	(A)

	<wait>	Line Fetch: (Area1)	(L+A)
4.2:	Read N (Area1+4)	(Area1+4)	(A)
4.3:	Intnl	(Area1+8)	(A)
	<wait>	(Area1+12)	(A)
	<wait>	(Code+16)	(L+A)
5.1:	Read N (Code+20)	Line Fetch: (Code+20)	(A)
	<wait>	(Code+24)	(A)
	<wait>	(Code+28)	(A)
5.2:	Read N (Area2)	Extnl Accs (Area2)	(L+A)
5.3:	Read N (Area2+4)	Extnl Accs (Area2+4)	(A)
	<wait>	Synchronize to FCLK	(F)
5.4:	Intnl	Internal Operation	(F)

Adding together the execution times taken for each of the VL86C020 operations gives a worst-case elapsed time for the code:

Maximum execution time = 4 F-cycles + 9 L-cycles + 18 A-cycles

Assuming that MCLK and FCLK both run at 8 MHz:

Maximum execution time = 31*125 ns = 3.875 µs.

COMPATIBILITY WITH EXISTING ARM SYSTEMS

Compatibility with VL86C010 -

The VL86C020 has been designed to be code compatible with the VL86C010 processor. The external memory and coprocessor interfaces are also designed to be usable with existing memory systems and coprocessors. The detailed changes are:

Software changes

1. VL86C020 now contains a single data swap (SWP) instruction. This takes the place of one of the undefined instructions in VL86C010.
2. VL86C020 has a 4 Kbyte mixed instruction and data cache on-chip. This cache should be transparent to most existing programs, although some system software (particularly that dealing with memory management) could be modified slightly to make more efficient use of the cache (see Cache Operation Section).
3. VL86C020 contains a set of control registers that govern operation of the on-chip cache (see Cache Operation Section). These registers must be programmed after VL86C020 is reset in order to enable the cache.
4. The internal timing associated with mode changes has been improved on VL86C020, and a banked register may now be accessed immediately after a mode change (see Data Processing/Writing to R15). However, for compatibility with VL86C010, it is recommended that the earlier restrictions are observed.
5. The implementation of the CDO instruction on VL86C010 causes a software interrupt (SWI) to take the undefined instruction trap if the SWI was the next instruction after the CDO. This is no longer the case on VL86C020 but the sequence

   CDO
   SWI

   should be avoided for program compatibility.

Hardware changes

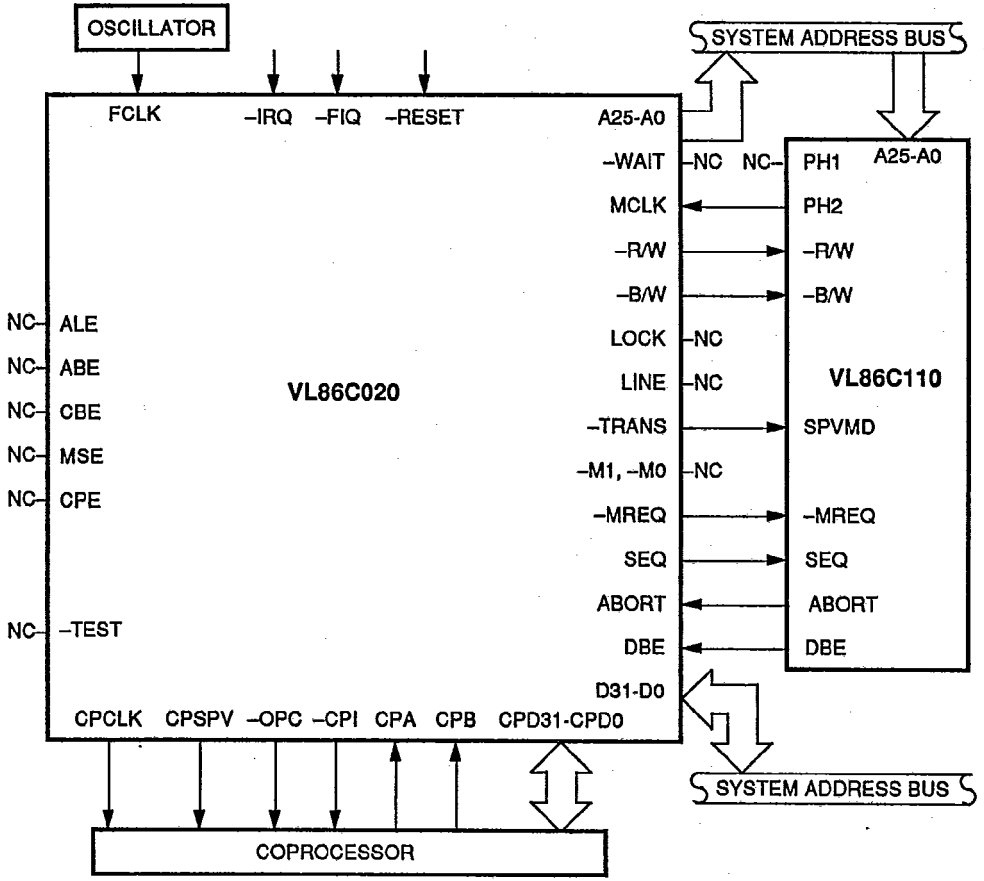
1. VL86C020 is packaged in a 160-pin quad flatpack; VL86C010 uses an 84-pin plastic leaded chip carrier (PLCC) package.
2. VL86C020 does not require non-overlapping clocks for timing memory accesses. When using VL86C020 with MEMC, the PH2 clock output of MEMC should be connected to the MCLK input of VL86C020; the PH1 clock output of MEMC is not used.
3. VL86C020 requires a free-running CMOS-level clock input (FCLK) to time cache accesses and internal operations. FCLK is entirely independent of MCLK.
4. VL86C020 includes two new control signals, LINE and LOCK. These warn of cache line fetch operations and locked swap (SWP) operations respectively.
5. The –TRANS and –M1, –M0 outputs on VL86C010 could change in either (PH2) clock phase. In VL86C020, these outputs only ever change when MCLK is high.
6. The coprocessor interface remains the same, but now operates independently of the external memory using a dedicated bus (CPD31-CPD0). Coprocessors must be able to operate at cache speeds (determined by FCLK).
7. The –OPC output of VL86C020 now applies exclusively to the coprocessor interface, and should not be used in the memory interface.

8. VL86C020 includes pull-up resistors on various control inputs (see Coprocessor Interface Section).

9. To facilitate board level testing, all outputs on VL86C020 can be put into a high impedance state by using the appropriate enable controls (see Coprocessor Interface Section).

Compatibility with MEMC (VL86C110)
The memory interface on VL86C020 is compatible with that used for VL86C010 and the existing MEMC memory controller is suitable. Figure 33 shows how VL86C020 may be connected to MEMC.

FIGURE 33. CONNECTING VL86C020 TO VL86C110 (MEMC)



TEST CONDITIONS
The AC timing diagrams presented in this section assume that the outputs of VL86C020 have been loaded with the capacitive loads shown in the "Test Load" column of Table 4; these loads have been chosen as typical of the system in which the CPU might be employed.

The output pads of the VL86C020 are CMOS drivers which exhibit a propagation delay that increases linearly with the increase in load capacitance. An "output derating" figure is given for each output pad, showing the approximate increase in load capacitance necessary to increase the total output time by one nanosecond.

TABLE 4: AC TEST LOADS

Output Signal	Test Load (pF)	Output Derating (pF/ns)
–MREQ	50	8
SEQ	50	8
–B/W	50	8
LINE	50	8
LOCK	50	8
–M0, –M1	50	8
–R/W	50	8
–TRANS	50	8
A0-A25	50	8
D0-D31	100	8
CPCLK	30	8
CPSPV	30	8
–CPI	30	8
–CPC	30	8
CPD0-CPD31	30	8

General note on AC parameters:
- Output times are to CMOS levels except for the memory and coprocessor data buses (D31-D0 and CPD31-CPD-0), which are to TTL levels.

AC CHARACTERISTICS: TA = 0°C to +70°C, VDD = 5 V ±5%

Symbol	Parameter	Min	Max	Unit	Conditions
tWS	–WAIT Setup to MCLK High	15		ns	
tWH	–WAIT Hold from MCLK High	5		ns	
tWAIT1	–WAIT Low Time		10000	ns	
tABE	Address Bus Enable		30	ns	
tABZ	Address Bus Disable		25	ns	
tALE	Address Latch Open		12	ns	
tALEL	ALE Low Time		10000	ns	Note
tADDR	MCLK High to Address Valid		55	ns	
tAH	Address Hold Time	5		ns	
tDBE	Data Bus Enable		35	ns	(TTL Level)
tDBZ	Data Bus Disable		25	ns	
tDOUT	Data Out Delay		30	ns	(TTL Level)
tDOH	Data Out Hold	5		ns	
tDE	MCLK Low to Data Enable		45	ns	(TTL Level)
tDZ	MCLK Low to Data Disable		40	ns	
tDIS	Data In Setup	8		ns	
tDIH	Data In Hold	8		ns	
tABTS	ABORT Setup Time	40		ns	
tABTH	ABORT Hold Time	5		ns	
tMSE	–MREQ and SEQ Enable		20	ns	
tMSZ	–MREQ and SEQ Disable		15	ns	
tMSD	MCLK Low to –MREQ and SEQ		55	ns	
tMSH	–MREQ and SEQ Hold Time	5		ns	
tCBE	Control Bus Enable		20	ns	
tCBZ	Control Bus Disable		15	ns	
tRWD	MCLK High to –R/W Valid		30	ns	
tRWH	–R/W Hold Time	5		ns	
tBLD	MCLK High to –B/W and LOCK		30	ns	
tBLH	–B/W and LOCK Hold	5		ns	
tLND	MCLK High to LINE Valid		50	ns	
tLNH	LINE Hold Time	5		ns	
tMDD	MCLK High to –TRANS/–M1, –M0		30	ns	
tMDH	–TRANS/–M1, –M0 Hold	5		ns	

Note: To avoid A25-A0 changing when MCLK is high, ALE must be driven low within 5 ns of the rising edge of MCLK.

AC CHARACTERISTICS FOR COPROCESSOR INTERFACE:

Symbol	Parameter	Min	Max	Unit	Conditions
tCPCKL	Clock Low Time		10000	ns	Note 1
tCPCKH	Clock High Time		10000	ns	
tOPCD	CPCLK High to –OPC Valid		15	ns	
tOPCH	–OPC Hold Time	5		ns	
tSPD	CPCLK High to CPSPV Valid		15	ns	
tSPH	CPSPV Hold Time	5		ns	
tCPI	CPCLK High to –CPI Valid		15	ns	
tCPIH	–CPI Hold Time	5		ns	
tCPS	CPA/CPB Setup	45		ns	
tCPH	CPA/CPB Hold	5		ns	
tCPDE	Data Out Enable		10	ns	Note 2, 3
tCPDOH	Data Out Hold	10		ns	
tCPDBZ	Data Out Disable		5	ns	
tCPDS	Data In Setup	10		ns	
tCPDH	Data In Hold	5		ns	
tCPE	Coprocessor Bus Enable		30	ns	
tCPZ	Coprocessor Bus Disable		30	ns	

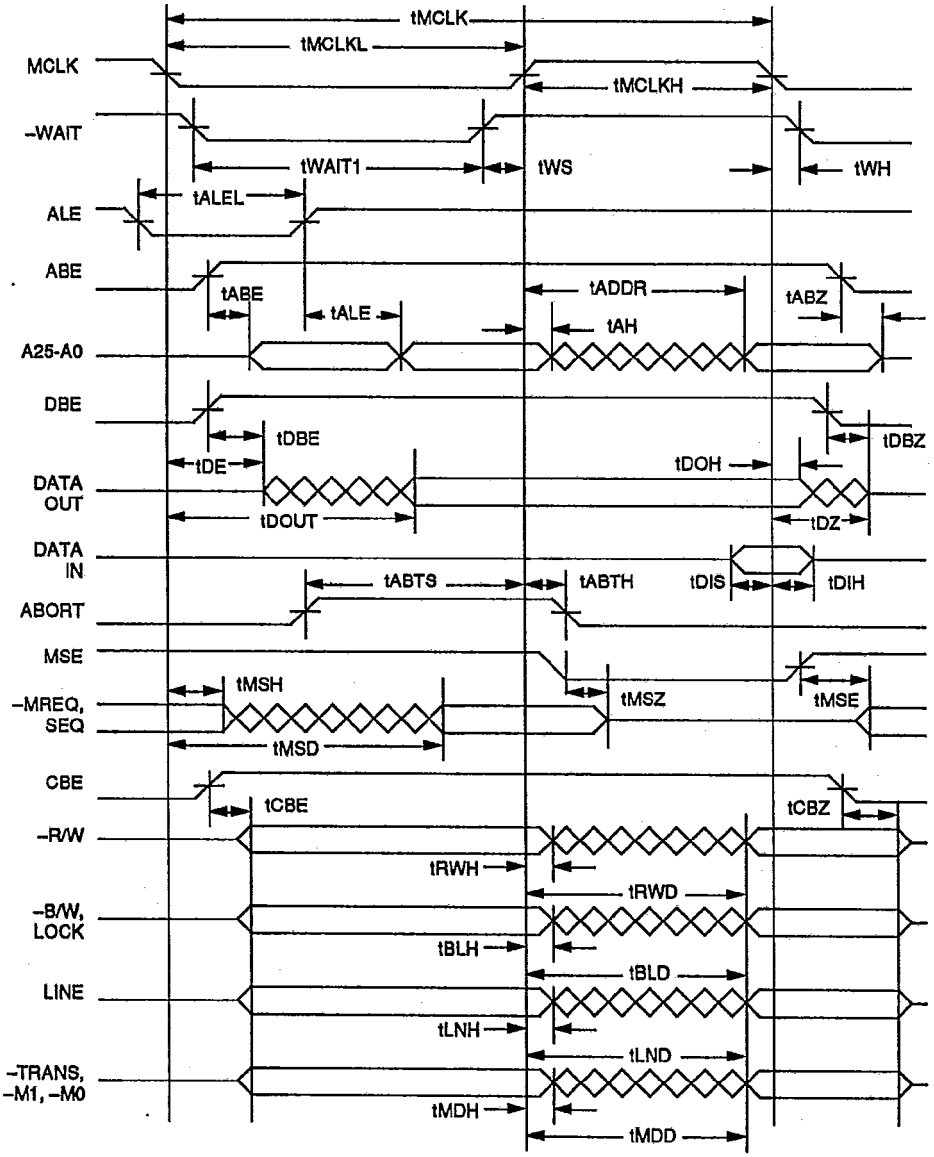
Notes:
1. CPCLK timings measured between clock edges at 50% of VDD.
2. CPD31-CPD0 outputs are specified to TTL levels.
3. The data from VL86C020 is always valid when enabled onto CPD31-CPD0.
4. These timings allow for a skew of 30 pF between capacitive loadings on the coprocessor bus outputs (CPCLK, –OPC, CPSPV, –CPI, CPD31-CPD0).

AC CHARACTERISTICS FOR CLOCKS:

Symbol	Parameter	Min	Max	Unit	Conditions
tMCLK	Memory Clock Period	80		ns	Note
tMCLKL	Memory Clock Low Time	25		ns	
tMCLKH	Memory Clock High Time	25		ns	
tFCLK	Processor Clock Period	50		ns	
tFCLKL	Processor Clock Low Time	23		ns	
tFCLKH	Processor Clock High Time	23		ns	

Note: MCLK timing measured between clock edges at 50% of VDD.

FIGURE 34. MEMORY INTERFACE TIMING



T-49-17-32

FIGURE 35. COPROCESSOR INTERFACE TIMING

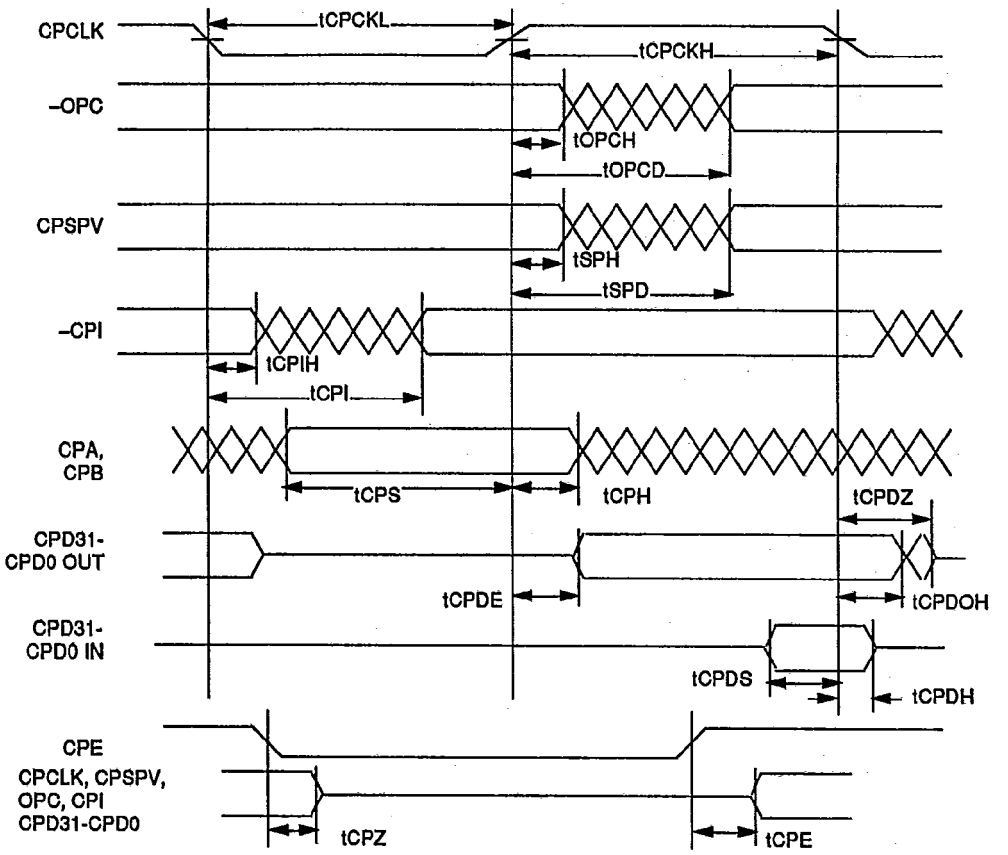
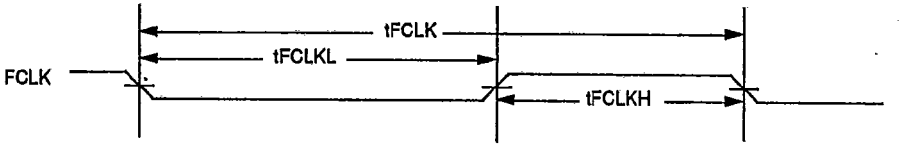


FIGURE 36. FCLK INTERFACE TIMING



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 VDD +0.3 V
Applied Output Voltage	−0.5 V to VDD +0.3 V
Applied Input Voltage	−0.5 V to +7.0 V
Power Dissipation	2.0 W

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

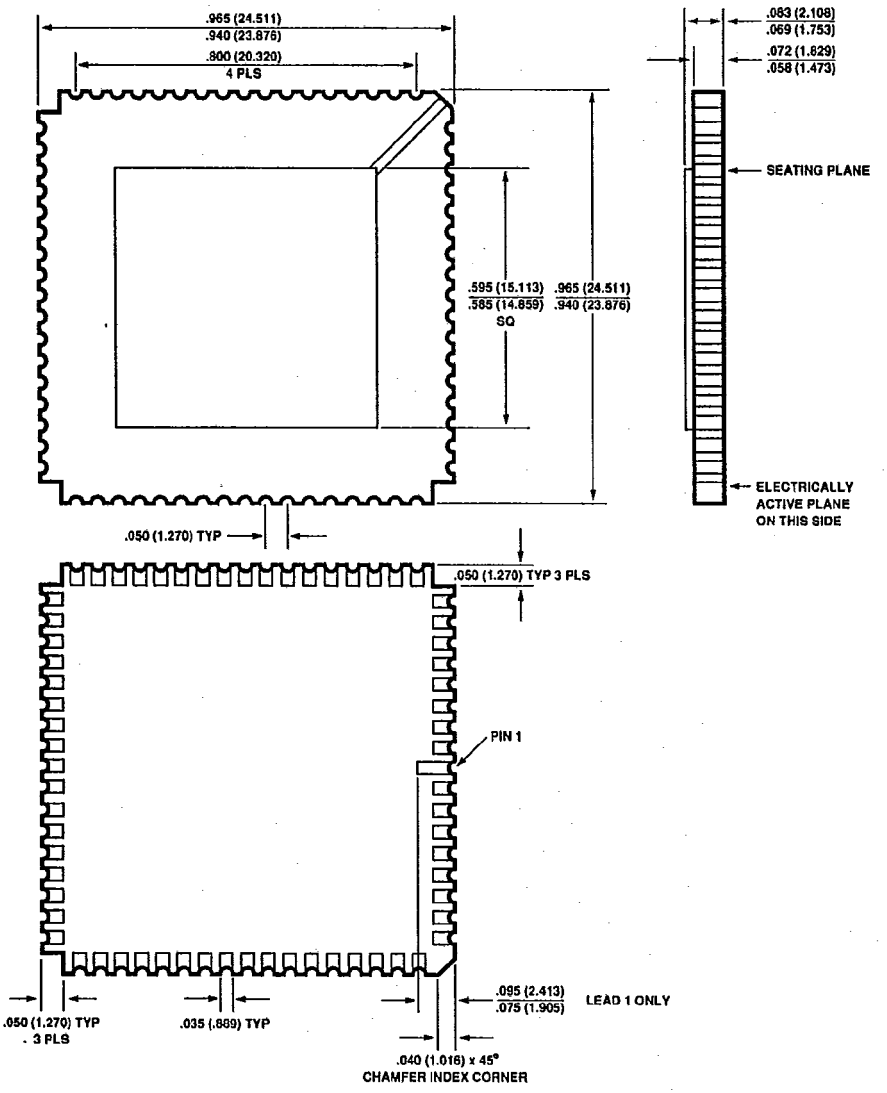
DC CHARACTERISTICS: TA = 0°C to +70°C, VDD = 5 V ±5%

Symbol	Parameter	Min	Typ	Max	Units	Conditions
VDD	Supply Voltage	4.75	5.0	5.25	V	
VIHC	IC Input High Voltage	3.5		VDD	V	Notes 1, 2
VILC	IC Input Low Voltage	0.0		1.5	V	Notes 1, 2
VIHT	IT/ITP Input High Voltage	2.4		VDD	V	Notes 1, 3, 4
VILT	IT/IPT Input Low Voltage	0.0		0.8	V	Notes 1, 3, 4
IDD	Supply Current		200		mA	
ISC	Output Short Circuit Current		160		mA	Note 5
ILU	D.C. Latch-up Current		>200		mA	Note 6
IIN	IT Input Leakage Current		10		µA	Notes 7, 11
IINP	ITP Input Leakage Current		−500		µA	Notes 8, 12
IOH	Output High Current (VOUT=VDD -0.4 V)		7		mA	Note 9
IOL	Output Low Current (VOUT=GND +0.4 V)		−11		mA	Note 9
VIHTK	IC Input High Voltage Threshold		2.8		V	Note 10
VILTT	IC Input Low Voltage Threshold		1.9		V	Note 10
VIHTT	IT/ITP Input High Voltage Threshold		2.1		V	Notes 11, 12
VILTT	IT/ITP Input Low Voltage Threshold		1.4		V	Notes 11, 12
CIN	Input Capacitance		5		pF	

Notes:
1. Voltages measured with respect to GND.
2. IC - CMOS-level inputs.
3. IT - TTL-level inputs (includes IT and ITOTZ pin types).
4. ITP - TTL-level inputs with pull-ups.
5. Not more than one output should be shorted to either rail at any time, and for as short a time as possible.
6. This value represents the DC current that the input/output pins can tolerate before the chip latches up.
7. Input leakage current for the IT, and ITOTZ pins.
8. Input leakage current for an ITP pin connected to GND. These pins incorporate a pull-up resistor in the range of 10 kΩ - 100 kΩ.
9. Output current characteristics apply to all output pads (OCZ and ITOTZ).
10. ICk - CMOS-level inputs.
11. IT - TTL-level inputs (includes IT and ITOTZ pin types).
12. TIP - TTL-level inputs with pull-ups.

PACKAGE OUTLINES

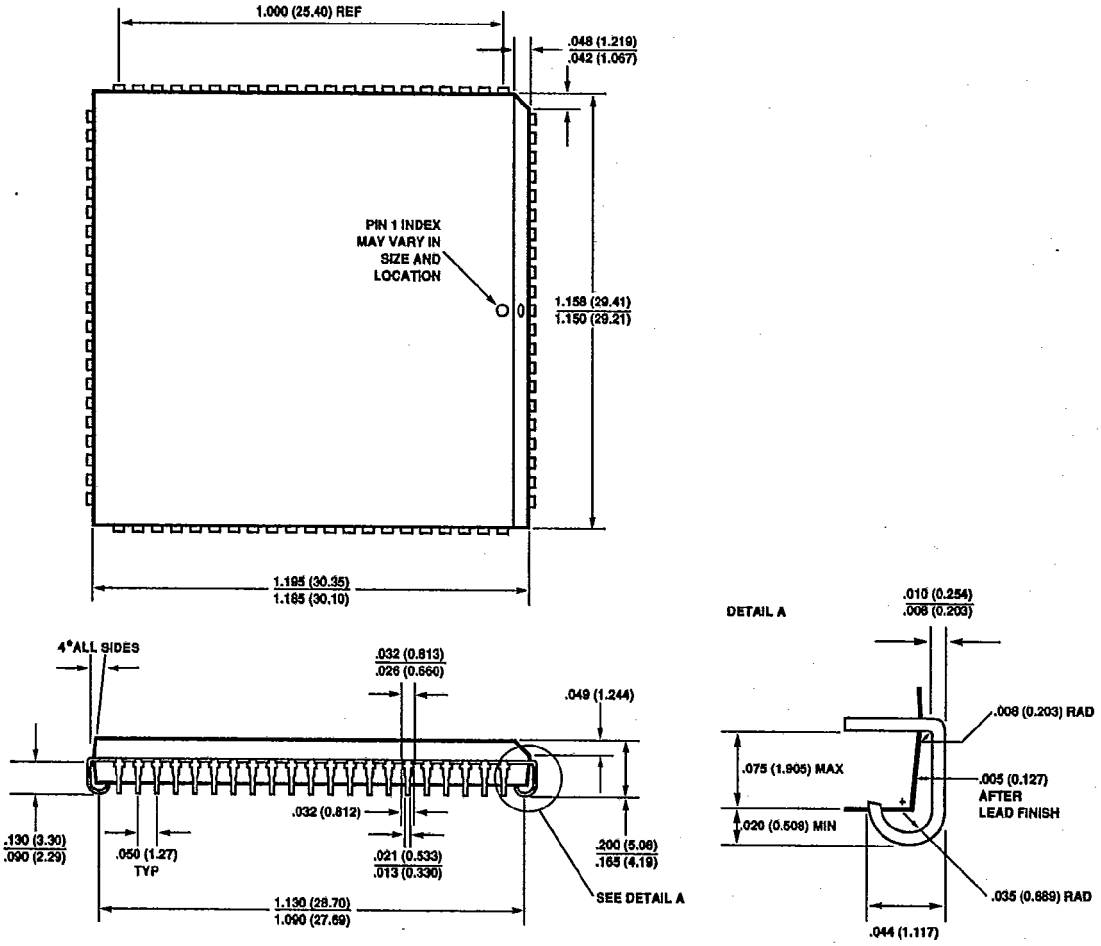
68-PIN PLASTIC LEADED CHIP CARRIER (PLCC)



8

PACKAGE OUTLINES (Cont.)

84-PIN PLASTIC LEADED CHIP CARRIER (PLCC)

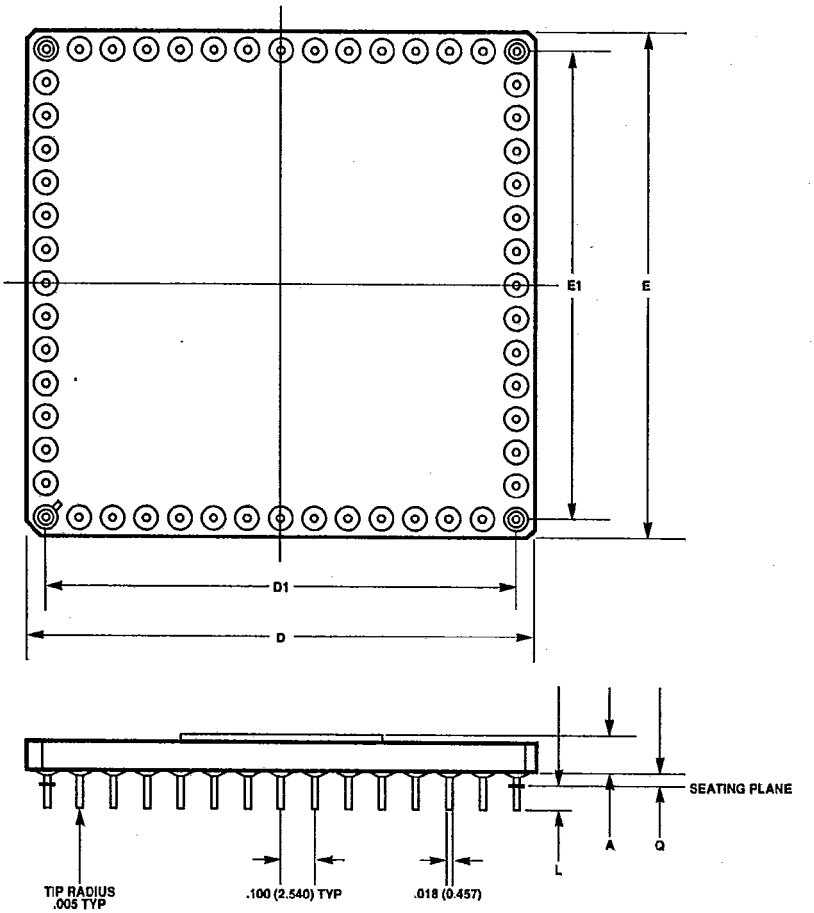


NOTES: UNLESS OTHERWISE SPECIFIED.
1. TOLERANCE TO BE +/- .005 (0.127).
2. LEADFRAME MATERIAL: COPPER.
3. LEAD FINISH: MATTE TIN PLATE OR SOLDER DIP.
4. SPACING TO BE MAINTAINED BETWEEN FORMED LEAD AND MOLDED PLASTIC ALONG FULL LENGTH OF LEAD.
5. MOLDED PLASTIC DIMENSION DOES NOT INCLUDE SIDE FLASH BURR, WHICH IS .010 (0.254) MAX ON FOUR SIDES.
6. CONTROLLING DIMENSIONS ARE METRIC, ALL METRIC DIMENSIONS ARE IN PARENTHESES.

25-60004 4/88

PACKAGE OUTLINES (Cont.)

144-PIN CERAMIC PIN GRID ARRAY



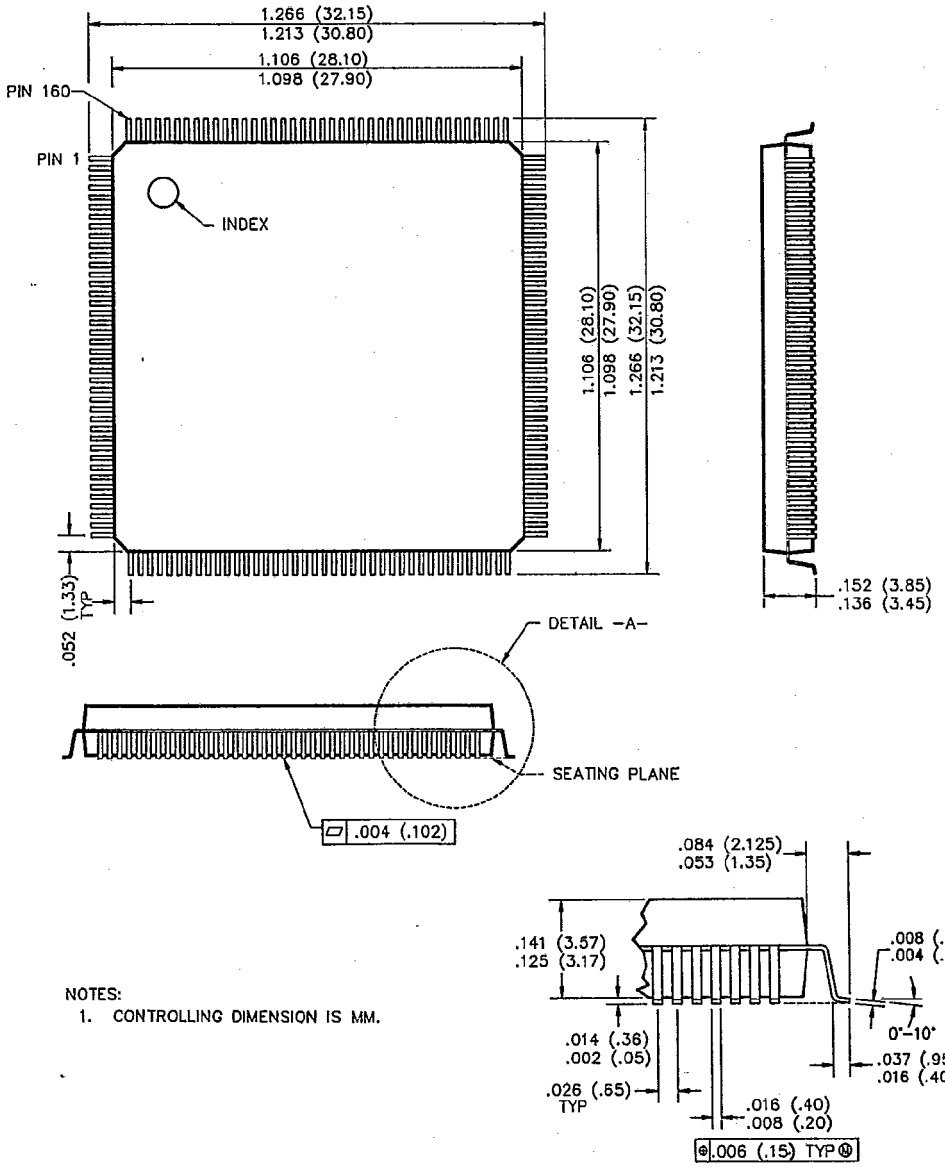
Pin Count	Matrix	Cavity Position	A		D (E)		D1 (E1)		Q	L
			Min	Max	Min	Max	Min	Max	Ref	Ref
144	15 X 15	Up	.0780 (1.981)	.1020 (2.591)	1.559 (39.60)	1.591 (40.41)	1.388 (35.26)	1.412 (35.86)	0.050 (1.270)	0.130 (3.302)

Notes:
1. All dimensions are in inches (mm).
2. Material: Al203
3. Lead Material: Kovar
4. Lead Finish: Gold plating 60 micro-inches min. thickness over 100 micro-inches nominal thickness of nickel

8

PACKAGE OUTLINES (Cont.)

160-PIN CERAMIC PIN GRID ARRAY



NOTES:

1. CONTROLLING DIMENSION IS MM.