VLSI TECHNOLOGY, INC.

ACORN RISC MACHINE (ARM) FAMILY DATA MANUAL

Application Specific
Logic Products Division

Prentice Hall, Englewood Cliffs, New Jersey 07632

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

This book can be made available to businesses and organizations at a special discount when ordered in large quantities. For more information, contact:

Prentice-Hall, Inc.
Special Sales and College Marketing
College Technical and Reference Division
Englewood Cliffs, N.J. 07632

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-781618-9

PRENTICE-HALL INTERNATIONAL (UK) LIMITED, *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL HISPANOAMERICANA, S.A., *Mexico*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
SIMON & SCHUSTER ASIA PTE. LTD., *Singapore*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*

VLSI TECHNOLOGY, INC.

CONTENTS

ACORN RISC MACHINE (ARM) DATA MANUAL — PAGE NUMBER

CONTENTS

PAGE NUMBER

CONTENTS

ACORN RISC MACHINE (ARM) DATA MANUAL

CONTENTS

CONTENTS


VLSI TECHNOLOGY, INC.

PREFACE

This book provides the reader with an in-depth and concise reference on the VLSI Technology, Inc. VL86C010 RISC system product. The RISC microprocessor and three RISC peripherals described in this text are both world-class and international. They were designed in the United Kingdom by Acorn Computer Ltd., using VLSI Technology, Inc. design tools, and are presently manufactured in the United States by VLSI. In addition, under recently signed alternate sourcing agreement, Sanyo, Ltd., will both manufacture the VL86C010 RISC family in Japan and develop derivative product.

In addition to a detailed hardware description of each device, this text extensively examines the software aspect of RISC Architecture. The instruction set is thoroughly explained, with numerous examples shown of programming techniques. Most readers who have some programming experience, whether familiar with existing "standard" microprocessors or not, should quickly understand programming in VLSI RISC system environment.

Except for the cover and VLSI logo, this book was entirely produced using desktop publishing. To maximize the desktop publishing program's usefulness, this text was produced using a preceding minus (–) sign rather than an overbar or asterisk to indicate a complemented signal.


VLSI TECHNOLOGY, INC.

VLSI TECHNOLOGY, INC.

SECTION 1

INTRODUCTION – ACORN RISC MACHINE

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

THE RISC SYSTEM SOLUTION FOR SMALL COMPUTERS

INTRODUCTION

Perhaps the most important topic in the computer industry the past few years has been the emergence of the Reduced Instruction Set Computer (RISC) touted as the next generation of performance oriented architectures. Several different suppliers – both component and system – have announced new computers based on the RISC design methodology. All claim that RISC offers much higher performence than more traditional Complex Instruction Set Computers (CISC). The common denominator among these suppliers has been a systems approach to the CPU design problem, In other words, the CPU is considered as a single unit. When multi-chip solutions are involved (as most ere), interfaces are defined around performance and bandwidth requirements more than functional blocks, the partitioning found in most commercial microprocessors today. Component suppliers often partition their systems around functions, like scalar processor, memory management unit, end floeting point processor. This allows each circuit to be used without the others, meaning that not all components have to be available before sales start. By partitioning around functions, the component suppliers usually sacrifice performance or require other system elements, such as memory, be faster than necessary et a given performance level.

As RISC technology moves from the laboratory into the commercial environment it Is important for system designers to understand these new considerations. When new applications arise that cannot be addressed cost-effectively by CISC architectures, this new technology may provide the only solution. By examining the following system, the designer will become familiar with this new, emerging computer technology and learn how systems can be partitioned around parameters other than functional blocks.

Brief Evolution of CISC and RISC Architectures

Most commercially available computers today should be classified as CISC. Many of these machines have existed for more then a decade, and have their foundation in technology that was radically different from today. When most existing machines began, logic and memory were expensive. In addition, software development was limited by the programming ability of assembler language and lack of efficient high-level language compilers. Early system designers were forced to heavily encode their limited instruction sets to minimize memory requirements of the system. Many processors began, with what was then considered as large, address spaces of 64K words/bytes of memory. Of course 64K words of essembler language code did represent a very large programming effort at the time.

Higher integration in semiconductor technology brought down the high cost of logic and memory. Soon, computer architects found they could build an equivalent system cheaper, with lower power requirements, and having more reliability. Also, Integration allowed them to add enhancements to the instruction set to improve performance of key customer applications for less cost than before. Assembler language progremmers wanted more enriched addressing modes that moved some of the computing functions from software to hardware. In addition, it improved programmer productivity by reducing the number of lines of assembler language necessary to code programs. Less lines per function meant more functions could be coded in the same time - I.e. higher productivity. High-level languages were available but generally were too inefficient to use except in the most complex applications level.

Hardware designers began adding new instructions end addressing modes to meet the programmer requests while remaining compatible with previous generations of software. Soon, system architects realized that they could provide more performance if they could sacrifice backwards compatibility and redefine their instruction sets to exploit new technologies. Instruction complexity had increased to the point where decoding multi-word, multi-format instructions was the limiting factor in processor speed. Unfortunetely, customers had huge investments in software and were reluctant to change to hardware that could not execute their installed base. New architectures were limited to new customers and applications.

High-level language efficiency and hardware performance improved dramatically and became useful for most applications. This helped two areas of concern in computer systems, programmer productivity and program transportability. High-level languages helped progremmers write code that was herdware independent, at least in theory, as compilers stood between the programmer and the execution environment (physical hardware and operating system). Compiler differences and ambiguous language specifications caused some portability problems, but in general it was practical to port programs between machines.

With more high-level language programs being written, hardware suppliers felt pressured to add even more complication to their instruction sets to support compiled code. Many architectures added herdware implementations of high-level constructs like FOR, WHILE, and PROC (procedure calls) directly into the instruction set. The problem arose es to which language to support because each is different, e.g. whether the conditional execution expression is evaluated at the beginning of the loop or the end. As a result, most architectures may support only one language well or ere so general that the compiler cannot exploit them efficiently (Wulf, 1981).

In the mid-seventies computer scientists began to investigate new methods to support all high-level languages more efficiently. It was becoming apparent that most problems were too complex to be written in assembler language and no one high-level language was sufficient to support all applications. From these development efforts came the RISC methodology for CPU design. What constitutes a RISC computer is yet another area of debate, but most emerging machines do have some characteristics in common.

First, most RISC machines are based on single-cycle instruction execution. Unlike their Complex Instruction Set Computers (CISC) counterparts that may take up to 100 minor (clock) cycles to complete complex instructions, the RISC machines instruction set is limited to primitive functions that can execute in a single or extremely few machine cycles. Compiler writers have suggested that it is more efficient to provide primitives to build solutions rather than solutions in the instruction set. When instructions have too much semantic content, a clash occurs between the language and the instruction set (Wulf, 1981) introducing inefficiency and increasing compiler complexity. In addition, single clock execution helps lower interrupt latency, thus making the system more responsive to the asynchronous environment of today's time-shared and/or networked systems.

Another common trait of RISC machines is a load/store architecture providing larger CPU register files. In a load/store architecture, the data processing instructions (logical and numeric functions) can only operate on the CPU registers. A separate set of instructions are used for memory reference that usually support a limited set of addressing modes. Streamlining the addressing modes helps simplify instruction decode, eliminate special-purpose address ALUs, and speed pipeline processing that can be slowed by multi-word address operand fetches. Recent improvements in the global register allocation problem faced by compilers have made efficient use of large numbers of registers possible. In response to compiler improvements, most RISC systems have added larger register files to improve performance. Two factors bring about significant performance increases from added registers: (1) register operations execute much faster, and (2) memory references are reduced because registers can hold temporary results.

In general, RISC machines are tightly coupled to their memory. The simple instruction set translates into a higher effective instruction execution rate, meaning the processors demand a high bandwidth from their memory systems to provide peak performance. In order to provide this bandwidth most, but not all, systems have implemented very sophisticated caching techniques which increase system cost and complexity dramatically.

The VLSI Technology RISC Computer System

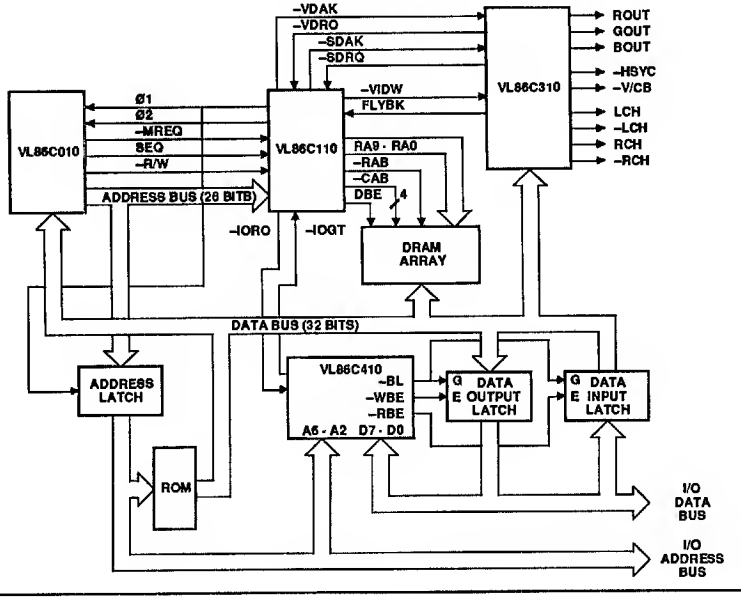
VLSI Technology has a full system solution to the design of a cost-effective, small computer. This system was designed by Acorn Computers Ltd. of Cambridge, United Kingdom, using the VLSI Technology, Inc. CAD system. What makes this system different is its unique method of partitioning the four circuits. Instead of designing the circuits around self-contained functions, this system is partitioned around basic computer fundamentals such as memory bandwidth, die size of all four components, and low-cost packaging available today. Careful attention to these fundamentals has yielded a small computer system that can bring excellent performance to the user at significantly lower cost than ever before. An examination of the system and its alternate form of partitioning will highlight the advantages of a top down design approach to the entire problem, not just CPU optimization.

The computer shown in Figure 1 is partitioned into four circuits: the VL86C010 Acorn RISC Machine (ARM) processor, VL86C110 Memory Controller (MEMC), VL86C310 Video Controller (VIDC), and VL86C410 I/O Controller (IOC). These four circuits together form a full 32-bit microcomputer system with performance in the 5 to 6 million-instructions-per-second (MIPS) range. Somewhat surprising is the fact that the four parts are available in one 84-pin (processor) and three 68-pin packages (JEDEC Type-B or Plastic-Leaded-Chip-Carriers, PLCC) while implementing full 32-bit functions. A more surprising fact is that no part in the system has a die size larger than 230 mils square in VLSI Technology's 2 µm double-layer metal CMOS process which means highly manufacturable circuits are available.

Partitioning The System

Traditionally, component designers viewed a computer system as "centered" around the CPU. The processor was designed in a vacuum, without concern for other elements in the system. The CPU was optimized to be high-performance and then the system designers found that in order to exploit the performance, they had to resort to expensive memory systems or cache sub-systems, increasing the cost

FIGURE 1. RISC SYSTEM BLOCK DIAGRAM



drametically. The CPU made such high demands on the memory that I/O transactions were not sufficiently served. This forced the systems designer to implement ever more complex I/O sub-systems, yet another addition to cost, complexity, and decreased reliability. Even today's most popular personal computers use plug in cards with on-board memory sub-systems for video end deta communicetions.

FIGURE 2. VL86C110 MEMORY CONTROLLER (MEMC) BLOCK DIAGRAM

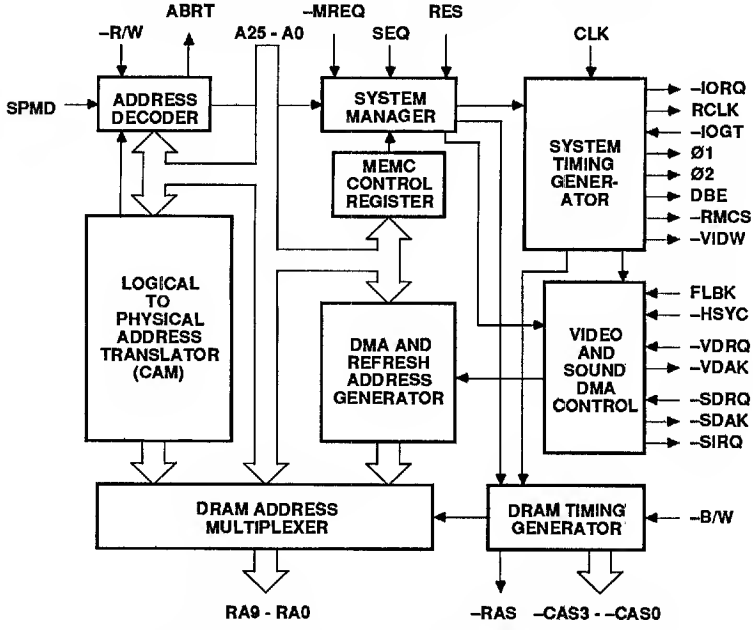
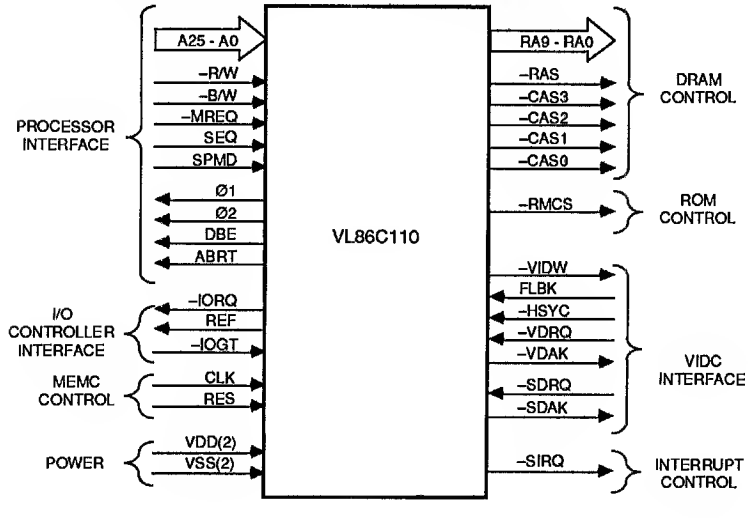


FIGURE 3. VL86C110 MEMORY CONTROLLER (MEMC) PIN DIAGRAM



The requirements for a small computer today, are very much different than even e few years ago. Now users expect a small computer to have capabilities that were only available in minicomputers. Full color displays at resolutions up to 640 by 480, real memory of 1 Mbyte, and networking support are common features demanded by end-users.

The VLSI Technology, Inc. system is "centered" around the memory, with each element designed to use the bandwidth efficiently without making large demands that require premium memory components. The video display is integrated into the design to utilize the main memory for display area, eliminating the need for expensive add-on video cards. The system operates with a 24 MHz clock that yields a basic processor cycle of 8 MHz (125 ns). Even at this speed, the memory system uses inexpensive 120 ns access time page-mode DRAMs.

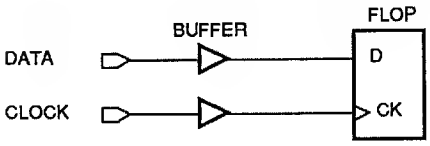
Memory Controller Functions

Since the system is designed around the memory, it is logical that the VL86C110 Memory Controller (MEMC) should be discussed first. Understanding how this part functions provides insight into the other elements end how they are coordinated together.

As the name would indicate, the MEMC generates the timing and control signals required by DRAM. In addition, MEMC acts es the main interface between the other three components by providing the critical timing signals for all elements from a single clock input. Figure 2 shows a block diagram and Figure 3 the functional pin out of the memory controller. It should be noted that MEMC does not have a data bus connection allowing it to be placed in a 68-pin package. To program the internal registers of MEMC, the data is encoded on the address bus during a processor write to the part. While at first this may seem a large overhead, using the simple/fast addressing modes and barrel shifter in the processor, the programmer will find that the address encoding causes very little impact.

The part generates all the timing signals required for interfacing the elements with the memory. High speed timing is generated from a single clock, usually 24 MHz for an 8 MHz processor. All

FIGURE 4. CLOCK SKEW TIMING EXAMPLE



Minimim Setup Time = Flop Setup Time + Data Buffer Maximum - Clock Buffer Minimum

Minimum Hold Time = Flop Hold Time + Clock Buffer Maximum - Data Buffer Minimum

system timing is generated on the MEMC with minimal buffering on the other devices. This scheme minimizes clock skew in the system allowing slower access time memory devices to be used. Figure 4 shows an example of how clock skew occurs in timing paths. Having all buffers on a single chip allows delays to track more closely than the total process variation. As shown by the example, fewer buffers in the path lower the amount of time that data must be valid on the bus, minimizing setup and hold times. Removing the clock buffer will eliminate the difference between the clock buffer delay minimum and maximum times.

The clock is divided by three and used to generate the processor and main system bus reference clocks. The MEMC drives up to 32 memory parts directly in several different configurations. Various configurations provide for up to 4 Mbytes of real memory in the system. The bandwidth of the low-cost DRAM memory is increased through extensive use of page-mode transfers because many memory references in computer systems are sequential in nature. MEMC also provides memory map decoding for I/O and ROM in the system. In order to optimize bandwidth, MEMC will take the ROM chip select active at the beginning of every non-sequential access and remove it if the cycle is not a ROM access making slower ROM accesses more efficient and once again allowing lower-cost ROMs to used.

MEMC supports several key functions in the system that usually have a tendency to impact performance or require faster components, so that this is not the case in this system. If a small computer is to support networking it must provide for multi-tasking and process isolation. MEMC provides full virtual memory support with a Logical-to-Physical Address Translator implemented as a 128 entry content addressable memory (CAM). Logical pages can be 4K, 8K, 16K, or 32K bytes each. RAM memory is always treated as 128 physical pages, meaning that MEMC contains a CAM entry (descriptor) for each physical page in memory. Having a CAM location for every physical page of memory eliminates descriptor thrashing, thus improving system performance. Thrashing occurs when the MMU system has fewer descriptors than physical pages of memory which introduces another source of address translation misses - the data is resident in memory but a descriptor to translate to that page is not available. A descriptor must be taken from another page to point to the requested page.

Many current memory management units contain only a small sub-set of the page tables and must retranslate the logical address whenever a new logical page is referenced (descriptor miss). Translation can take up to several microseconds depending on how many memory cycles must be performed. In this system the address translation is not in the critical path and does not require faster memory than a system that uses physical addresses. No translation takes place on the row address values which are required early in the memory cycle. The mapped address bits are placed into the column address field and are therefore not needed until much later in the cycle. This approach can be taken because the memory is usually configured as a single bank meaning all memories are active when the RAS becomes active regardless. Systems that have more than one bank of DRAM and use this approach would be required to select (bring RAS active) all memory devices on every cycle. Multi-bank memory systems designed in this manner would have much higher power consumption and lose much of the advantage of DRAM technology.

The simple CAM contained in MEMC can support demand paging with some software assistance and it provides a full virtual memory implementation with three levels of access protection efficiently. The goal of virtual memory support in this system was to let programs be written independent of real memory size rather than for multi-user support. Today's most popular PC has suffered recently due to the artificial real memory limitation placed on it by the machine designers.

MEMC contains all the address generators to support DMA activity related to video, cursor, and sound generation. These were placed on this circuit for two reasons. First, it eliminates the need to have the full address bus placed on the video interface circuit. This allows the VIDC to have the full 32-bit data bus and still be packaged in a 68-pin package. Second, this arrangement uses the memory bandwidth more efficiently by reducing synchronization and buffer delays on the memory bus while improving DMA latency. In most systems a DMA operation proceeds as follows: (1) the DMA device requests a transfer, (2) the memory controller synchronizes to the system clock and recognizes the request, (3) processor is signaled to relinquish the bus, (4) processor synchronizes and recognizes the request, (5) processor issues grant to memory controller, (6) memory controller synchronizes and recognizes grant, (7) memory controller issues DMA grant, (8) DMA synchronizes and recognizes grant, (9) DMA device enables address bus drivers, (10) memory controller receives address and multiplexes address to memory devices, (11) memory controller issues data acknowledge, (12) DMA device synchronizes and recognizes acknowledge, and (13) DMA device removes request to end cycle.

MEMC provides the memory arbitration and all address sources in a single

device within the system. This eliminates several levels of pulse synchronizers and buffering delays. When the VIDC signals a DMA request, MEMC only has to recognize the request, disable the processor when appropriate, and enable the address from the internal source. The DMA device has a simple interface to latch the data when the acknowledge signal goes inactive. This interface provides a very efficient DMA capability for read-only devices like video and sound generators. In order to optimize bandwidth usage, MEMC performs four memory cycles per DMA request, one full access taking 250 ns and three sequential page-mode accesses of 125 ns each. Four cycle bursts were chosen for all devices to increase bandwidth but keep bus latency to a reasonable value. Long latency introduces other costly problems that are usually solved with expensive FIFO buffers or other interface hardware that is duplicated in every device that connects to the bus.

RISC Processor Functions

The VL86C010 RISC processor provides the computational element in the system. The processor has a radically reduced instruction set containing a total of only 46 different operations. Unlike most others, all instructions occupy one 32-bit word of memory. In keeping with the tradition of RISC methodology, the processor is implemented as with a single-cycle execution unit and a load/store architecture. The basic addressing mode supported is indexed from a base register, with several different methods of index specification. The index can be a 12-bit immediate value contained within the instruction, or another register (optionally shifted in some

TABLE 1. VL86C010 INSTRUCTIONS

FUNCTION	MNENOMIC	OPERATION	PROCESSOR CYCLES
Data Processing			
Add with Carry	ADC	Rd:=Rn + Shift(Rm) + C	1S
Add	ADD	Rd:=Rn + Shift(Rm)	1S
And	AND	Rd:=Rn • Shift(Rm)	1S
Bit Clear	BIC	Rd:=Rn • Not Shift(Rm)	1S
Compare Negative	CMN	Shift(Rm) + Rn	1S
Compare	CMP	Rn – Shift(Rm)	1S
Exclusive - OR	EOR	Rd:=Rn XOR Shift(Rm)	1S
Multiply with Accumulate	MLA	Rn:=Rm * Rs + Rd	16S max
Move	MOV	Rn:=Shift(Rm)	1S
Multiply	MUL	Rn:=Rm * Rs	16S max
Move Negative	MVN	Rd:=NOT Shift(Rm)	1S
Inclusive - OR	ORR	Rd:=Rn OR Shift(Rm)	1S
Reverse Subtract	RSB	Rd:=Shift(Rm) - Rn	1S
Reverse Subtract with Carry	RSC	Rd:=Shift(Rm) - Rn - 1 + C	1S
Subtract with Carry	SBC	Rd:=Rn - Shift(Rm) - 1 + C	1S
Subtract	SUB	Rd:=Rn - Shift(Rm)	1S
Test for Equality	TEQ	Rn XOR Shift(Rm)	1S
Test Masked	TST	Rn • Shift(Rm)	1S
Data Transfer			
Load Register	LDR	Rd:=Effective address	2S + 1N
Store Register	STR	Effective address:= Rd	2N
Multiple Data Transfer			
Load Multiple	LDM	Rlist:=Effective Address	(n**+1)S + 1N
Store Multiple	STM	Effective Address:=Rlist	(n**+1)S + 2N
Jump			
Branch	B	PC:=PC+Offset	2S + 1N
Branch and Link	BL	R14:=PC, PC:= PC+Offset	2S + 1N
Software Interrupt	SWI	R14:=PC, PC:= Vector #	2S + 1N

*Shift() denotes the output of the 32-bit barrel-shifter. One operand can be shifted in several manners on every data processing instruction without requiring any additional cycles.

** - n is the number of registers in the transfer list.

N denotes a non-sequential memory cycle and S a sequential cycle.

manner). The index can be used in a pre or post-indexed fashion for any method of specification.

Table 1 shows the instructions supported by the processor. These instructions operate only on the CPU internal registers. Only the multiply instruction requires more than one cycle to execute (32 x 32 multiply in 16 clocks worst case) and it is not the limiting factor in interrupt response time. All instructions have conditional execution implementing a type of skip architecture. Unexecuted instructions require a single processor cycle and keep the three-stage pipeline intact. This approach was taken as opposed to the delayed branch approach to simplify the virtual memory page fault recovery process. When the branch and delayed instruction are contained on separate physical pages and a fault occurs on the fetch after the taken branch, the recovery process can be extremely expensive in both software and hardware complexity. Studies have shown that compiled code generated on the VAX averaged three instruction executions between every taken branch (Clark and Levy, 1982). While instruction set differences may cause the number of instructions between branches to vary, the conditional execution helps the processor keep its pipeline intact for forward reference branches of short length.

The VL86C010 supports two types of branch instructions, branch and branch-with-link for subroutine calls. Again, both branch types offer conditional execution. For subroutine calls, the current value of the machine state contained in register 15, program counter and status register, is copied into register 14. Linking subroutine calls through the registers instead of the more traditional memory stack, reduces the call/return overhead. For a single-level linkage, the state is saved within the machine in a single clock and can be restored also in a single clock. For multi-level call sequences, full machine state is contained in a single word, requiring only a single memory reference for stacking.

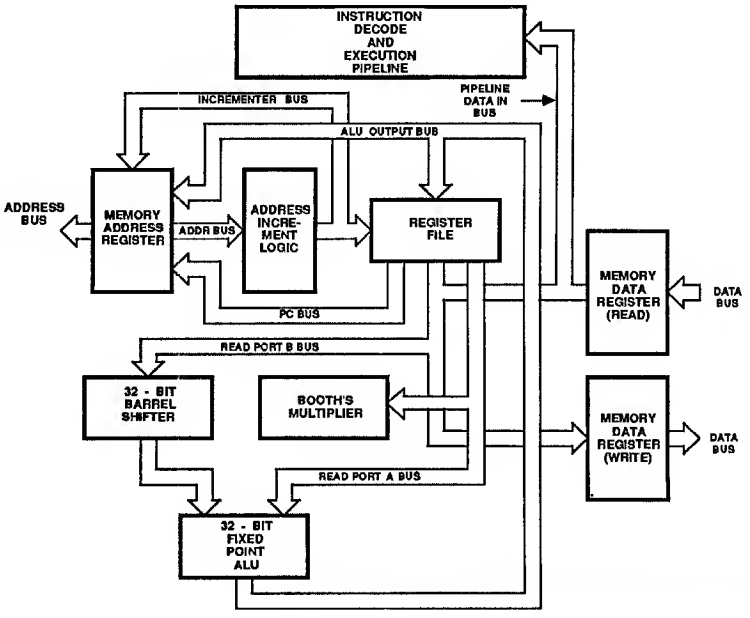
Two types of data transfer instructions are supported for memory references. A single register can be read or written to memory in two clock cycles. In order to exploit sequential memory access modes, the processor also performs load and store multiple operations. For these instructions more than one register is transferred, taking two clocks for the first register and one clock for each additional one. This instruction greatly enhances the processor's ability to move large blocks of memory and context switches that save the entire machine state. A block transfer instruction of all 16 registers is the longest instruction and therefore is the limiting factor in interrupt response time.

Figure 5 shows a block diagram of the processor. Several hardware features are worthy of note. First, by streamlining the instruction set, more silicon area can be dedicated to hardware functions that enhance performance. The VL86C010 contains a full 32-bit barrel shifter that can be used to pre-shift one operand on every processor cycle without additional delay. The barrel shifter increases the performance of shift intensive applications like graphics manipulations significantly. Second, the addition of a memory interface signal (SEQ) to alert the VL86C110 that the next memory address is sequential to the current address. This extra status allows the processor and memory controller to exploit the page-mode capability of DRAMs and obtain higher bandwidth without requiring faster memory devices.

The third major hardware feature is the partially overlapped register file containing 27 locations, although only 16 are visible to the program at any one time. Unlike some other RISC processors, the registers in the VL86C010 overlap across processor modes instead of procedure calls. The processor supports four modes of operation: User, System, Fast Interrupt Request (FIRQ), and Normal Interrupt Request (IRQ). In the User mode the program has 16 (R0 to R15) registers. R15 contains the program counter and status register and R14 is used for subroutine linkage. The other 14 registers are general purpose as is R14 when it is not needed for linkage.

Whenever a mode change is performed, new registers are mapped into the visible space. Two new registers (R13 and R14) are available to the System and IRQ modes respectively. Seven additional registers are available in the FIRQ mode which lowers the processor's interrupt latency. The FIRQ

FIGURE 5. VL86C010 RISC PROCESSOR BLOCK DIAGRAM



mode has a worst case interrupt latency of 22.5 clocks and can be as little as 2.5 clocks. The extra registers can hold DMA pointers and word counts allowing the processor to implement high speed DMA transfers without external controllers, further reducing system cost without significant overhead.

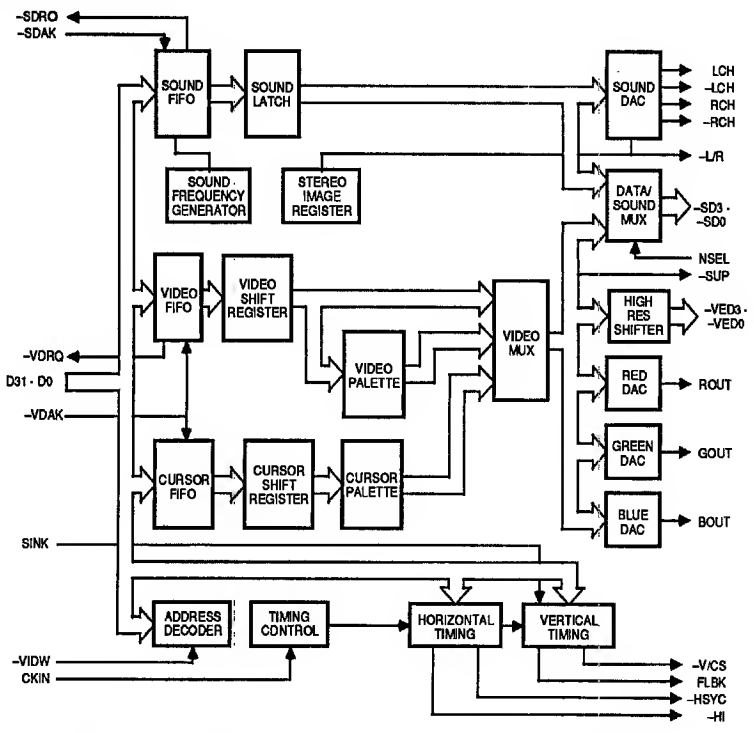
Most other RISC processors overlap the registers across procedure calls, implementing a register stack that is used for local variables and parameter passing. This scheme works well with the C language because C does not allow nested scopes like other languages such as Modula2 and PASCAL. These languages require the program to access variables of all levels that are active at the same time. In addition, the processor must handle the case where the register stack overflows (Hennessy, 1984). Both these problems complicate the processor design and can slow context switching across processor modes. It was determined that the overlap across modes was a more efficient use of chip area for supporting all high-level languages and making the processor more responsive to the asynchronous environment posed by network support. Besides, the large register bank is expensive and can extend processor cycles with extra levels of decode internally.

Video Support In the System

The video support is integrated into the design of the processor system to eliminate add-on video sub-systems and dedicated display memory buffers. The VL86C310 Video Controller (VIDC) provides a highly flexible choice of display formats in both color and high resolution monochrome. Horizontal timing is controlled in units of two pixel times and vertical in units of raster times. Besides performing video operations, the VIDC also can generate high quality stereo sound with up to eight channels of separate stereo position.

Figure 6 shows a block diagram of the video controller. The part accepts video data in a packed pixel format from the memory, serializes the data into pixel information, and presents the data to the color-mapping RAM (video palette) where it is converted to analog values suitable for driving an RGB monitor.

FIGURE 6. VL86C310 VIDEO CONTROLLER (VIDC) BLOCK DIAGRAM

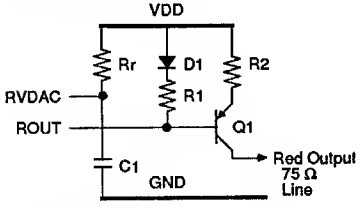


VIDC contains three channels of DMA for interfacing to the video and sound systems but does not generate the addresses directly. For video refresh, the part supports separate DMA channels for video and cursor information. The third DMA channel generates the sound data fetches. Each DMA channel has a dedicated FIFO of four 32-bit words for cursor and sound and 16 words for video. The FIFO depth can be small because of the highly efficient and responsive bus implementation of the system. Each channel uses the four word burst transfers discussed before to exploit the page-mode access mode of DRAMs.

The output of the video FIFO is connected to the video serializer. The pixel rate is programmable at values of 8, 12, 16 or 24 MHz. In addition, the video data format can be selected to be 1, 2, 4, or 8 bits per pixel. Once the video data is serialized, it is presented to the color palette. The palette provides 16 words of 13 bits each, allowing the part to support 256 simultaneous colors from 4096 possible choices or an external video source. The output of the palette is multiplexed with the cursor information and presented to the video DACs for conversion to analog RGB formats. The VIDC can support displays of up to 640 by 480 with 16 colors (high-resolution PC type display) directly without any addition logic. The only external components required are a simple circuit to convert the current sink DAC outputs to an appropriate voltage. A suitable circuit is shown in Figure 7.

The cursor is handled as a separate sprite making its manipulation simple and it is allowed pixel level positioning anywhere on the screen. The cursor is defined as 32-bits wide and any number of rasters high. Cursor information is fetched during horizontal retrace on rasters where the cursor will appear. The cursor sprite can contain up to three different colors from the 4096 palette, with a fourth alternative color of

FIGURE 7. EXAMPLE VIDEO OUTPUT DRIVER

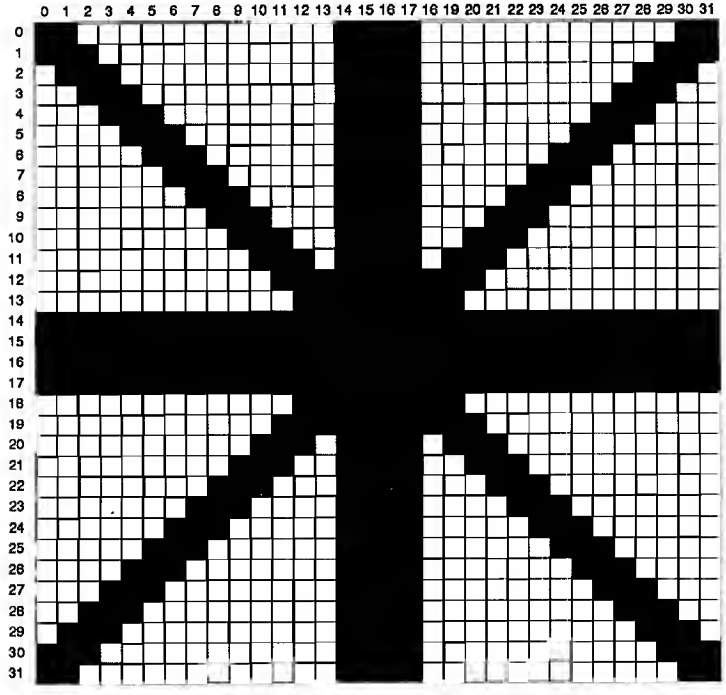


Suggested Component Values
Rr – 10 kΩ
R1 – 330 Ω
R2 – 68 Ω
D1 should have similar characteristics to the emitter-base junction of Q1

transparent. Each pixel that is transparent allows the video information to be displayed instead of the cursor. The background video color "shows through" the cursor. The transparent attribute allows cursors of various shapes to be defined, allowing each application the option of customizing the display to enhance the man-machine interface. Figure 8 shows an example of how a non-rectangular shaped cursor would be defined. Each bit of the cursor sprite can be specified with no limitations as to the number of color changes or length of color fields found in systems that use run-length encoding for data reduction.

Most small computers support some type of sound output as does this system. The difference here is the support for full-stereo sound. Up to eight channels of stereo position are supported yielding very high quality sound. Due to the small die size and large pin count, the addition of stereo sound adds nothing to the cost of the part (perhaps a small test cost increase) if it is not needed. However, the system designers can use this interface to greatly differentiate their machines. Applications programs could be written to exploit the power of the processor to run signal processing algorithms and utilize compressed speech or other sound information to enhance man-machine interfaces or provide other useful functions. This sound capability in conjunction with the VIDC's ability to synchronize to external displays, could provide a highly effective system for the computer-based training market.

FIGURE 8. VL86C310 CURSOR SPRITE EXAMPLE



Supporting I/O Transactions

Input/output control is very important in computer systems. Most component vendors concentrate all their design effort and analysis on the CPU, striving to achieve the highest performance. I/O is left as an after-thought at best, or the I/O sub-system is designed as a special-purpose CPU trying to maximize its performance without regard to the other elements in the system. Interfaces grow complex and establish bottlenecks to system performance or even worse, sub-systems become isolated and difficult to control. For example, many graphics processors proposed in the past few years did not allow the host processor access to the display memory. Software engineers proclaimed this as an unmanageable solution and as a result many component designers reworked their interfaces to provide more control. Addressing I/O and CPU designs at the same time is important because many of today's high performance systems are totally I/O bound, forcing the CPU into idle states, and causing the users to pay for performance they cannot obtain in the execution environment.

The last element in the VLSI Technology, Inc. small computer system is the VL86C410 Input/Output Controller (IOC). The circuit provides a unified environment for I/O related activities such as interrupts and peripheral controllers. This environment simplifies system software and allows the processor to interface easily with existing low-cost peripheral controllers such as VL16C450 Asynchronous Communications Element and VL1772 Floppy Disk Controller. Using these low-cost, mature devices is a key to providing a cost-effective small computer in today's market.

A block diagram of IOC is shown in Figure 9. The part provides the system with several general I/O support functions. The VL86C410 contains four 16-bit counter/timer circuits, two configured as general-purpose timers and two as baud rate generators. One baud rate generator is dedicated to the Keyboard Asynchronous Receiver/ Transmitter (KART) and the other

FIGURE 9. VL86C410 INPUT/OUTPUT CONTROLLER (IOC) BLOCK DIAGRAM

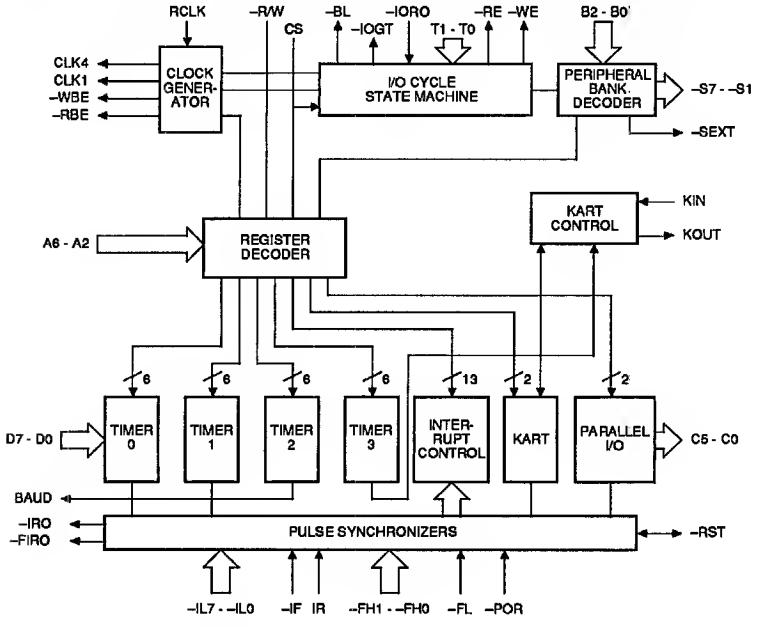
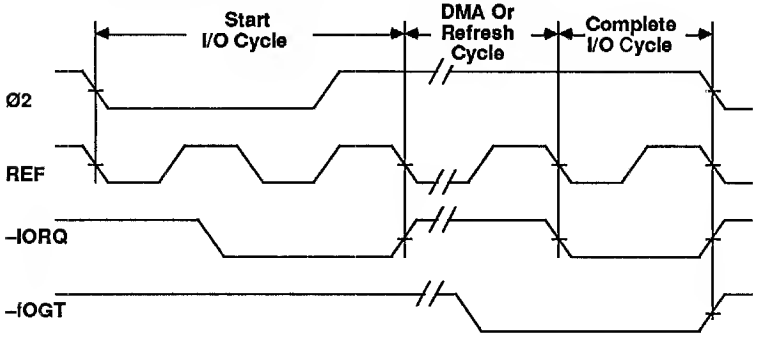


FIGURE 10. VL86C410 INTERRUPTIBLE CYCLE EXAMPLE



controls the BAUD output pin of the device. Timing of external events becomes more important in systems that must support networking and multi-tasking. Most network protocols require nodes to respond within a certain time (three seconds is common) and the initiator node must detect a timeout and invoke error recovery procedures. Multi-tasking operating systems usually require some type of timing interrupt for task control.

The KART section is a simple fixed-format asynchronous bidirectional serial communications link designed basically for keyboard input. The format is fixed with an eight bit character, one start bit, and two stop bits. The clock rate is a standard 16 times the data rate and the transmit and receive clocks are at the same rate and controlled by Timer 3 within IOC. To improve noise immunity, false start bits of less than one-half bit duration are ignored. The KART is ideal for interfacing to the low speed character rate (up to 31K characters/second) from a keyboard but it can be used for other purposes if the format is suitable.

The major task of IOC is the implementation of an efficient interface between the high speed system and the lower speed I/O peripheral controller buses. The system exploits the low-cost peripheral controllers but should not be severely impacted with performance/latency penalties for using them. The part contains six programmable bi-directional I/O pins for implementing special processor control. Interrupts are supported with control for both normal (IRQ) and fast (FIRQ) interrupts through mask, request, and status registers. Sixteen interrupt sources are supported, fourteen level and two edge-triggered, meaning the IOC should have the total interrupt status for most system configurations.

Centralizing the interrupts in this manner reduces polling, improves efficiency, and reduces latency within the system. Fast response time allows the processor to replace expensive dedicated logic with software, lowering the system cost accordingly. Many component vendors demand higher prices for their DMA device than for their CPU. Unfortunately, the CPU is usually idled during DMA transfers because they share the address and data buses to the memory. If the CPU was more responsive, it could provide the transfers without any degradation in system performance and eliminate the expensive DMA hardware.

The peripheral controller cycles are supported with four different lengths for access times. This allows peripheral controllers from various vendors with different bus clocking schemes to be interfaced easily and cheaply without extra logic. Each VL86C410 supports seven peripheral select lines which are independently selectable from the four access cycle times. If more than seven peripheral controllers are needed, multiple IOCs can be used in the system or the select lines can be decoded further externally because the system provides sufficient address set-up time.

In order to maintain low latency on the high speed system bus, the IOC is

designed to allow an I/O cycle to be interrupted by a DMA access on the system bus. Figure 10 shows a timing diagram of this operation. The IORQ is generated by MEMC whenever an I/O access address is detected. The IOC will respond with an IOGT signal when the access is complete. If the MEMC detects a pending DMA request, it removes IORQ and performs the transfer. IOC turns off the buffers that isolate the two buses and continues with the I/O cycle until the MEMC returns the IORQ. Then, the cycle is completed when both the master and slave device parameters have been met. This interruptible I/O cycle eliminates the slower peripheral devices from the system bus latency calculations, improves efficiency, and lowers system cost.

Conclusions

Whenever a system is partitioned, the designers should consider the entire problem as a single coherent entity, optimizing all parts together rather than each separately. The VLSI Technology, Inc. system demonstrates the advantages of partitioning around system bus parameters instead of the more traditional functional, stand-alone blocks. This system exploits low-cost memory and peripheral components while achieving excellent throughput with superior cost/performance ratios. With careful attention, the system designer can eliminate large die sizes and expensive high-pin count packages without sacrificing throughput and achieve superior cost-performance ratios.

References

Clark, D. and H. Levy. "Measurement and analysis of instruction use in the VAX 11/780," In Proceedings of the 9th Annual Symposium on Computer Architecture. ACM/IEEE, Austin, Texas, April 1982.

Hennessy, John L. "VLSI Processor Architecture." IEEE Transactions on Computers, Volume C - 33, Number 12 (December 1984), pp. 1221-1246.

Wulf, William A. "Compilers and Computer Architecture." Computer, July 1981, pp. 41-47.

VLSI TECHNOLOGY, INC.

SECTION 2

VL86C010 32-BIT RISC MICROPROCESSOR

2

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

VL86C010

32-BIT RISC MICROPROCESSOR

FEATURES

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

DESCRIPTION

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

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

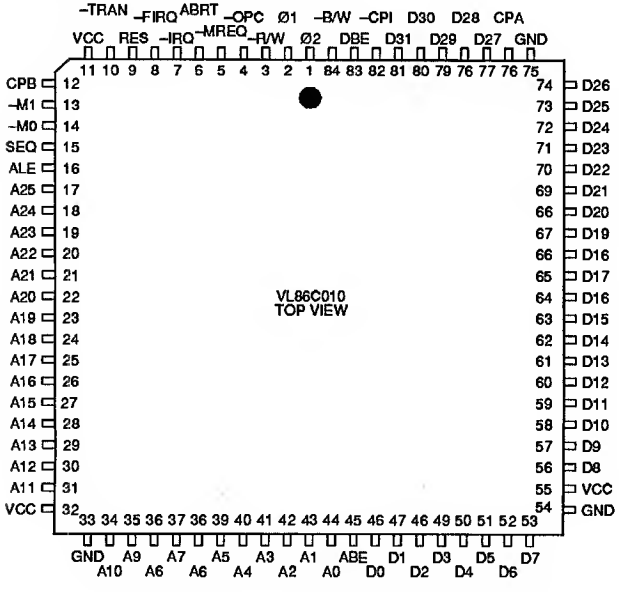
The ability to do pre- and post-indexing allows stacks and queues to be easily implemented in software. All instructions are 32 bits long (aligned on word boundaries), with register-to-register operations executing in one cycle. The two data types supported are 8-bit bytes and 32-bit words.

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

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

PIN DIAGRAM

PLASTIC LEADED CHIP CARRIER (PLCC)

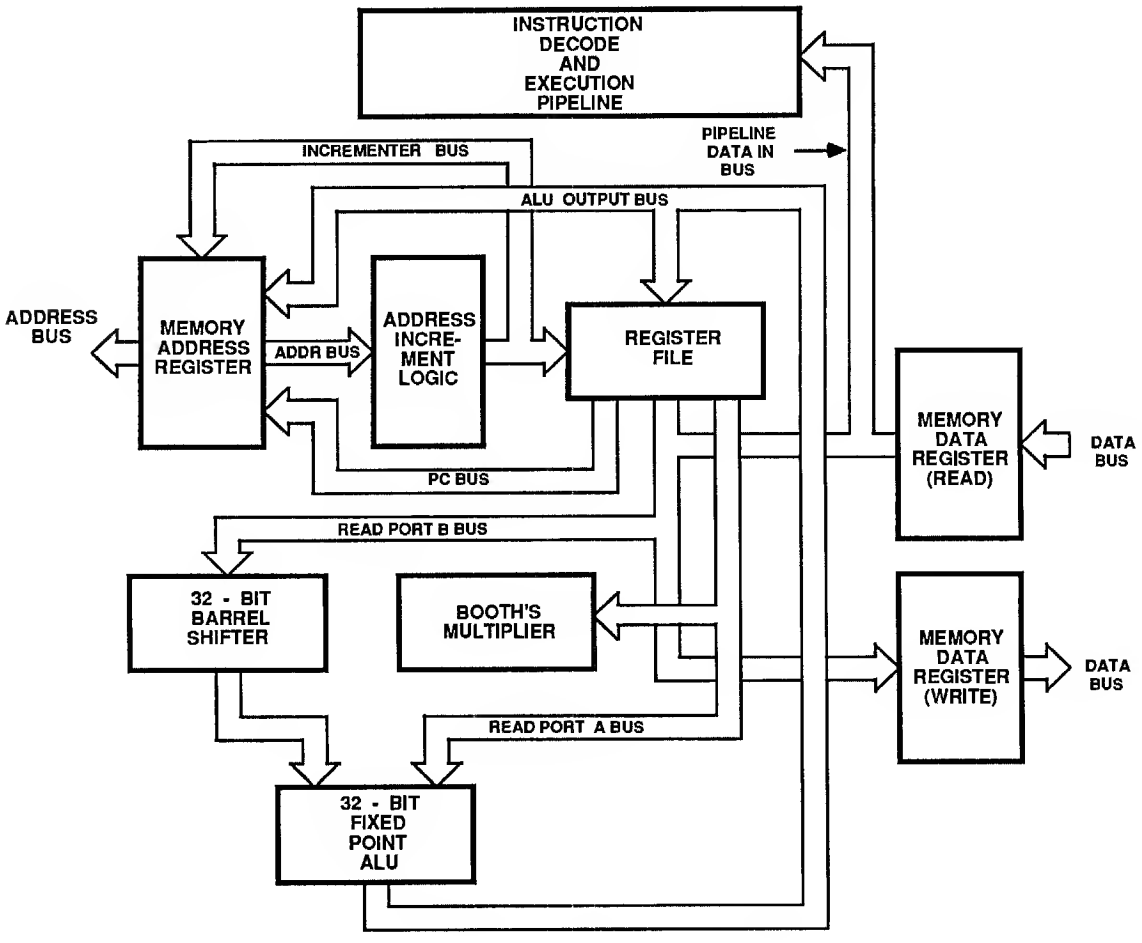


ORDER INFORMATION

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

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

BLOCK DIAGRAM



SIGNAL DESCRIPTIONS

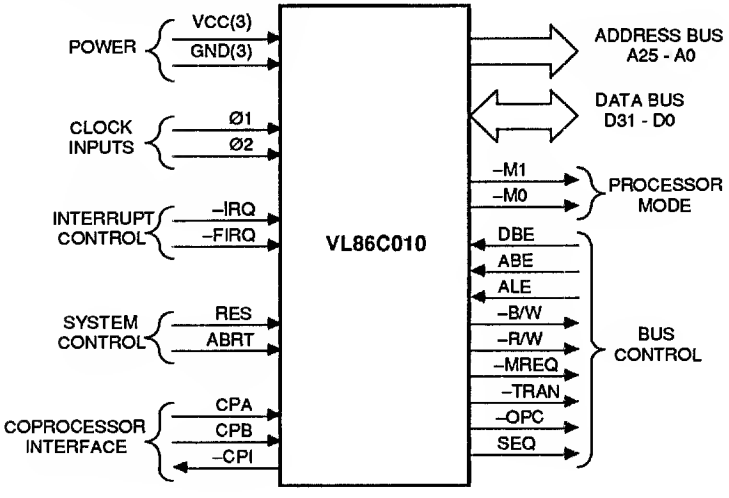
Signal Name	Pin Number	Signal Description
Ø1, Ø2	2,1	Processor Clock Ø1 and Ø2 Inputs - These two inputs provide the clock to the processor. In order to minimize clock skew, these inputs are not buffered internally and therefore must swing monotonically between GND and VCC without overshoot. The clocks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

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

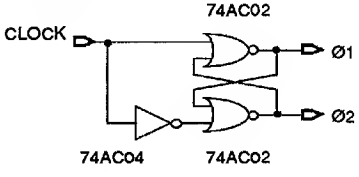
SIGNAL DESCRIPTIONS (Cont.)

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

FUNCTIONAL PIN DIAGRAM



TYPICAL CLOCK GENERATOR



FUNCTIONAL DESCRIPTION

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

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

COMPARISON OF PROCESSORS

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

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

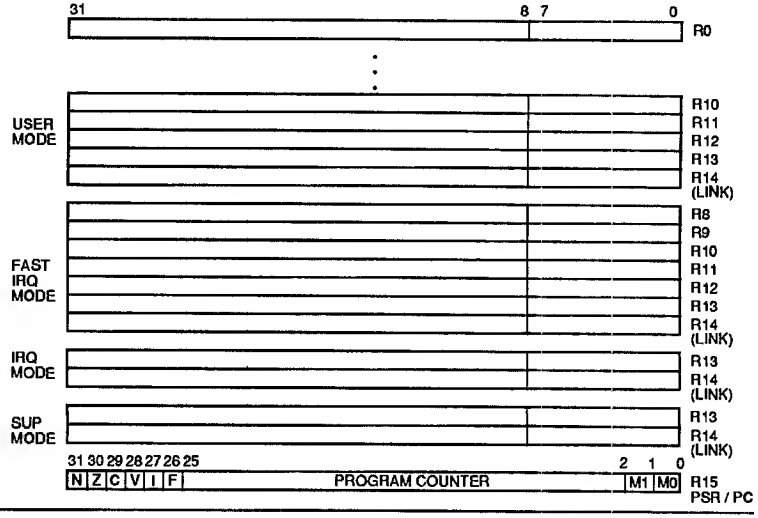
An Important parameter to keep constant when benchmarking processors is the memory access times, since not all processors will meet performance claims when working with commodity memories.

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

PROGRAMMING MODEL

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

FIGURE 1. VL86C010 REGISTER MODEL



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

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

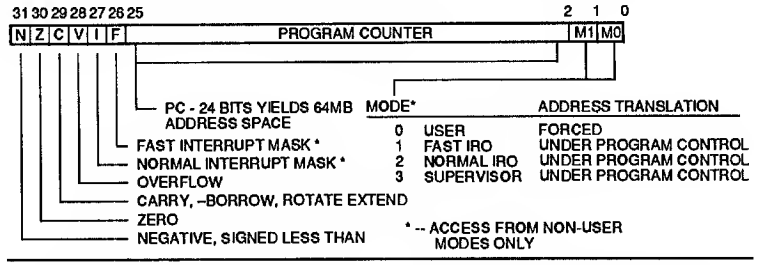
Two registers are dedicated to the supervisor mode and overlie user mode registers R13 and R14 when a supervisor mode switch is made using a software interrupt (SWI) instruction. Operation of these two registers is the same as previously discussed.

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

PROCESSOR STATUS REGISTER

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

FIGURE 2. PROCESSOR STATUS REGISTER



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

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

EXCEPTIONS

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

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

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

TABLE 1. INSTRUCTION CONDITION CODES

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

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

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

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

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

INSTRUCTION SET

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

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

TABLE 2. EXCEPTION VECTOR MAP

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

TABLE 3. DATA PROCESSING INSTRUCTIONS

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

TABLE 4. MEMORY ADDRESSING MODES

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

* Effective Address

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

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

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

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

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

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

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

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

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

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

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

TABLE 5. INSTRUCTION EXECUTION TIMES

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

* - The number of registers transfered in a Load/Store Multiple instruction. If the condition field in an instruction is not true, the instruction is skipped and the execution time is 1S cycle.

** - This is the worst case time. The actual time is a function of the value in the Rs register.

S implies a sequential cycle.

N implies a non sequential cycle.

EXAMPLES OF THE INSTRUCTION SET

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

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

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

By using conditional execution, the routine compresses to:

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

EXAMPLE 2 - ABSOLUTE VALUE

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

EXAMPLE 3 - UNSIGNED 32-BIT MULTIPLY

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

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

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

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

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

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

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

EXAMPLE 7 - MULTIPLICATION BY 6

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

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

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

EXAMPLE 9 - DIVISION AND REMAINDER

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

INSTRUCTION CYCLE OPERATIONS

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

BRANCH AND BRANCH WITH LINK

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

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

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

2

TABLE 6. BRANCH AND BRANCH WITH LINK

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

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

DATA OPERATIONS

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

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

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

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

TABLE 7. DATA OPERATIONS

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

INSTRUCTION CYCLE OPERATIONS (Cont.)

MULTIPLY AND MULTIPLY ACCUMULATE

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

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

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

TABLE 8. MULTIPLY AND MULTIPLY ACCUMULATE

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

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

LOAD REGISTER

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

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

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

TABLE 9. LOAD REGISTER

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

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

INSTRUCTION CYCLE OPERATIONS (Cont.)

STORE REGISTER

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

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

TABLE 10. STORE REGISTER

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

LOAD MULTIPLE REGISTERS

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

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

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

TABLE 11. LOAD MULTIPLE REGISTERS

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

INSTRUCTION CYCLE OPERATIONS (Cont.)

STORE MULTIPLE REGISTERS

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

TABLE 12. STORE MULTIPLE REGISTERS

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

SOFTWARE INTERRUPT AND EXCEPTION ENTRY

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

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

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

TABLE 13. SOFTWARE INTERRUPT & EXCEPTION ENTRY

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

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

COPROCESSOR DATA OPERATION

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

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

TABLE 14. COPROCESSOR DATA OPERATION

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

INSTRUCTION CYCLE OPERATIONS (Cont.)

CO-PROCESSOR DATA TRANSFER (FROM MEMORY TO COPROCESSOR)

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

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

2

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

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

INSTRUCTION CYCLE OPERATIONS (Cont.)

COPROCESSOR DATA TRANSFER (FROM CO-PROCESSOR TO MEMORY)

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

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

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

INSTRUCTION CYCLE OPERATIONS (Cont.)

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

TABLE 17. COPROCESSOR REGISTER TRANSFER (LOAD FROM COPROCESSOR)

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

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

TABLE 18. COPROCESSOR REGISTER TRANSFER (STORE TO COPROCESSOR)

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

INSTRUCTION CYCLE OPERATIONS (Cont.)

UNDEFINED INSTRUCTIONS AND COPROCESSOR ABSENT

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

TABLE 19. UNDEFINED INSTRUCTIONS AND COPROCESSOR ABSENT

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

UNEXECUTED INSTRUCTIONS

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

TABLE 20. UNEXECUTED INSTRUCTIONS

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

INSTRUCTION SPEEDS

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

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

TABLE 21. INSTRUCTION SPEEDS

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

n is the number of words transferred.

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

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

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

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

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

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

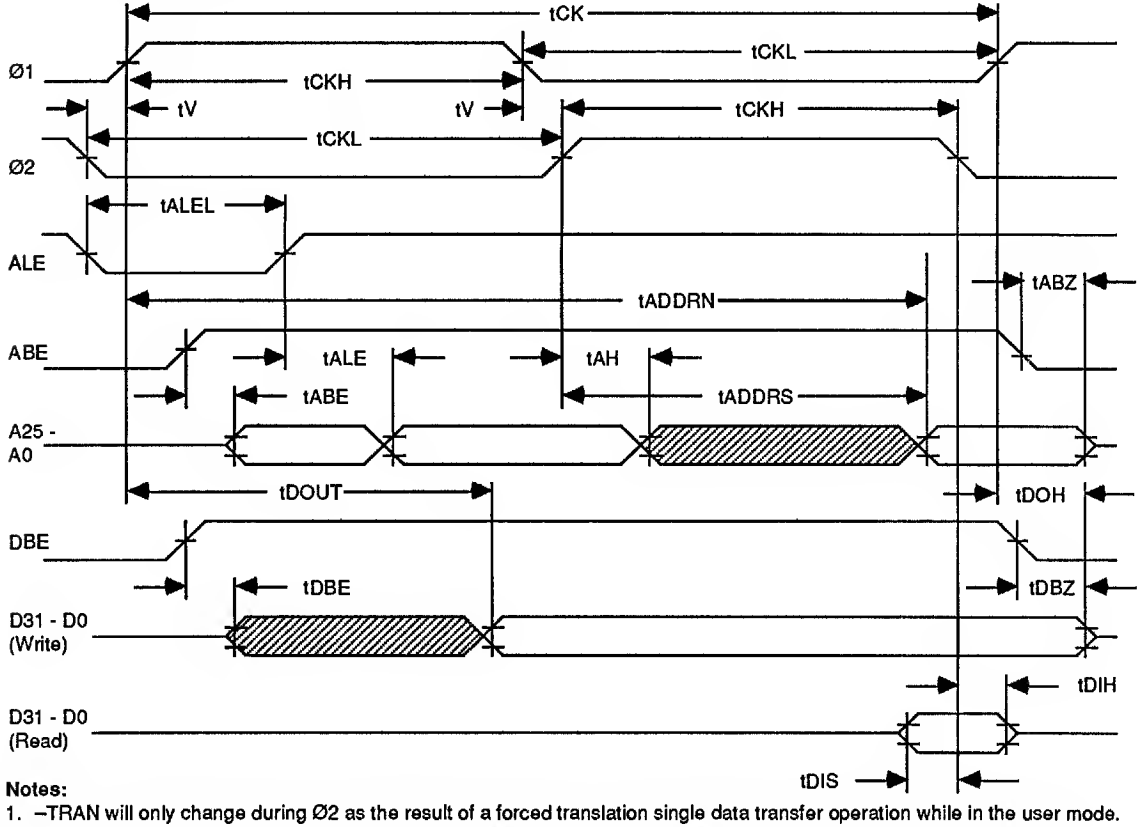
Notes:
1. ALE controls a dynamic storage latch; this parameter is specified to ensure that the stored charge cannot leak sufficiently to generate intermediate logic levels in the associated logic.
2. The Ø1 to address delay only applies to non-sequential cycles, when the address is being calculated in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

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

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

TIMING DIAGRAMS

PROCESSOR DATA BUS

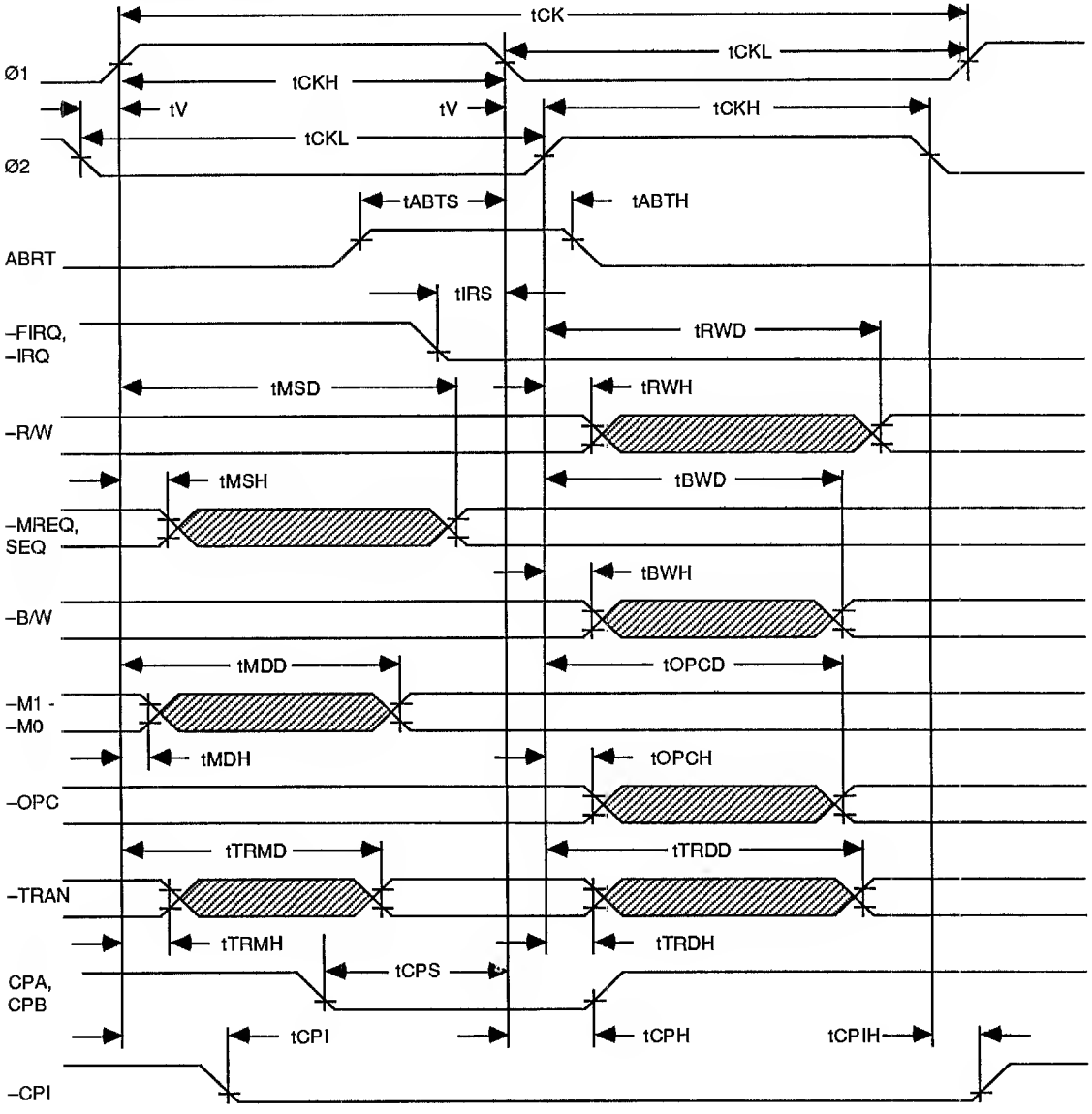


Notes:

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

TIMING DIAGRAMS

PROCESSOR CONTROL SIGNALS



ABSOLUTE MAXIMUM RATINGS

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

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

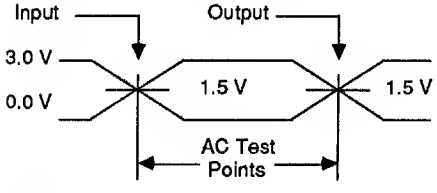
DC CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ± 5%

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

CAPACITANCE: TA = 25°C, f = 1.0 MHz

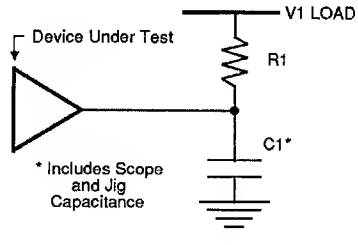
Symbol	Parameter	Min	Max	Unit	Conditions
CI	Clock Input Capacitance (Ø1, Ø2)	–	15	pF	VIN = 0 V (Note2)
	Other Input Capacitance	–	5	pF	VIN = 0 V (Note2)
CO	Output Capacitance	–	8	pF	VOUT = 0 V (Note 2)

FIGURE 3. TEST WAVEFORMS



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

FIGURE 4. TEST LOAD CIRCUIT



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

PROGRAMMERS' MODEL

The VL86C010 processor has a 32-bit data bus and a 26-bit address bus. The processor supports two data types, eight-bit bytes and 32-bit words, where words must be aligned on four byte boundaries. Instructions are exactly one word, and data operations (e.g., ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words. The VL86C010 supports four modes of operation, including protected supervisor and interrupt handling modes.

BYTE SIGNIFICANCE

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

REGISTERS

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

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

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

R14 is used as the subroutine link register and receives a copy of R15 when a Branch and Link instruction is executed. It may be treated as a general purpose register at all other times. R14_svc, R14_irq and R14_firq are used similarly to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within supervisor or interrupt routines.

TABLE 22. REGISTER ORGANIZATION

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

TABLE 23. BYTE ADDRESSING

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

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

IRQ Processing - The IRQ state has two private registers mapped to R14 and R13 (R14_Irq and R13_Irq).

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

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

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

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

EXCEPTIONS

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

The processor handles exceptions by using the benked registers to save state. The old PC end PSR are copied into the appropriate R14, and the PC and processor mode bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 should be saved onto a stack in main memory before re-enabling the interrupt. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled.

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

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

IRQ - The IRQ (Interrupt Request) exception is a normal interrupt caused by a low level on the –IRQ pin. It hes a lower priority than FIRQ, and is masked out when a FIRQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the PC (but note that this is not possible from user mode). If the I flag is clear, the processor checks for a low level on the output of the IRQ synchronizer at the end of each instruction.

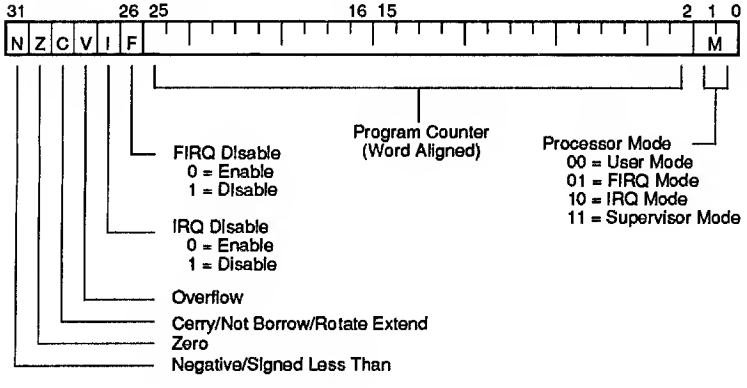
The impact upon execution of an IRQ interrupt is defined in Table 19. The return-from-interrupt sequence is also defined there. This will restore the original processor state and re-enable the IRQ interrupt and will cause execution to resume at the instruction following the interrupted one.

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

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

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

FIGURE 5. PROGRAM COUNTER AND PROCESSOR STATUS REGISTER



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

Abort - The ABRT signal comes from an external memory management system, and indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disk, and considerable processor activity may be required to recover the data before the access can be performed successfully. The processor checks for an abort at the end of the first phase of each bus cycle. When successfully aborted, the VL86C010 will respond in one of three ways:

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

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

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

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

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

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

Software Interrupt - The software interrupt is used for getting into supervisor mode, usually to request a particular supervisor function. The processor response to the SWI instruction is defined in Table 19, as is the method of returning. The indicated return method will return to the instruction following the SWI.

TABLE 24. EXCEPTION TRAP CONSIDERATIONS

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

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

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

Undefined Inatruction Trap - When the VL86C010 executes a coprocessor instruction or an undefined instruction, it offers it to any coprocessors which may be present. If a coprocessor can perform this instruction but is busy at that moment, the processor will wait until the coprocessor is ready. If no coprocessor can handle the instruction, the VL86C010 will take the undefined instruction trap.

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

When the undefined instruction trap is taken, the VL86C010 will respond as defined in Table 19. The return from this trap (after performing a suitable emulation of the required function) defined in Table 19 will return to the instruction following the undefined instruction.

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

Vector Table

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

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

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

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

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

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

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

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

Tsyncmax is 2.5 processor cycles, Tldm is 18 cycles, Texc is three cycles, and Tfiq is two cycles. The total time is 25.5 processor cycles, which is just over 2.5 microseconds in a system using a continuous 10 MHz processor clock. In a DRAM based system running at 4 and 8 MHz (for example, using the VL86C110) this time becomes 4.5 microseconds, and if bus bandwidth is being used to support video or other DMA activity, the time will increase accordingly.

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

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

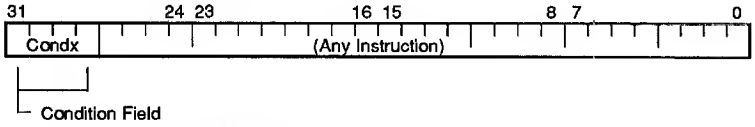
INSTRUCTION SET

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

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

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

FIGURE 6. CONDITION FIELD



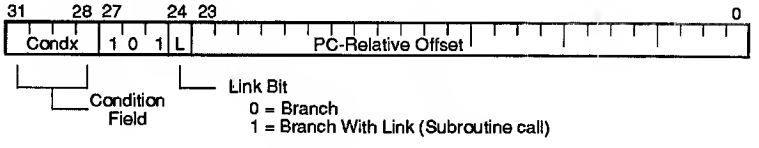
0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = CS - C set (unsigned higher or same)
0011 = CC - C clear (unsigned lower)
0100 = MI - N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned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

BRANCH, BRANCH AND LINK (B, BL)

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

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

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



0 = Branch
1 = Branch With Link (Subroutine call)

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.

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

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

Syntax:

B(L){cond} <expression>

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

cond is a two-character mnemonic as shown in Condition Code section (EQ, NE, VS, etc.). If absent then AL (Always) will be used.

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

Items in ( ) are optional. Items in < > must be present.

Examples:

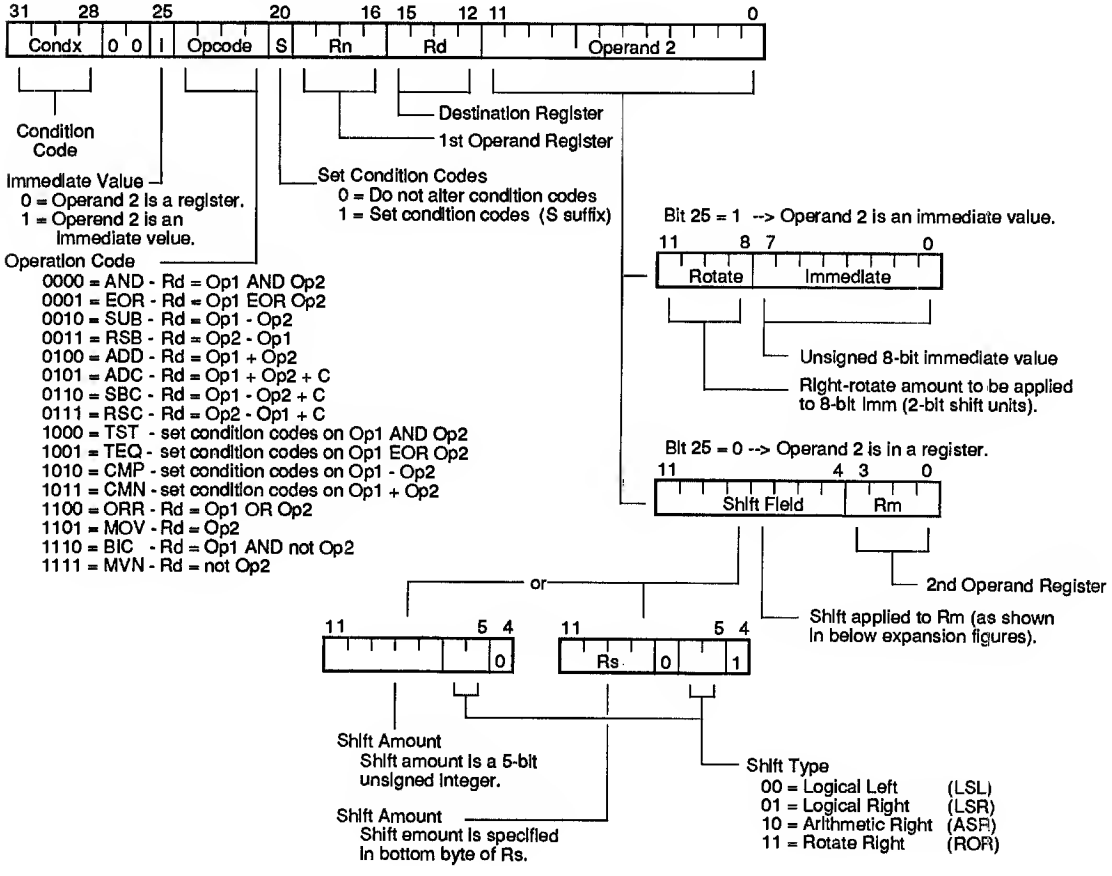
```
Here    BAL     Here        ; Assembles to EAFFFFFE.  (Note effect of PC offset)
        B       There       ; Always condition used as default
        CMP     R1,0        ; Compare register one with zero, and branch to Fred if
        BEQ     Fred        ; register one was zero.  Else continue next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

ALU INSTRUCTIONS

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

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

FIGURE 8. ALU INSTRUCTION TYPES



DATA PROCESSING OPERATIONS

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

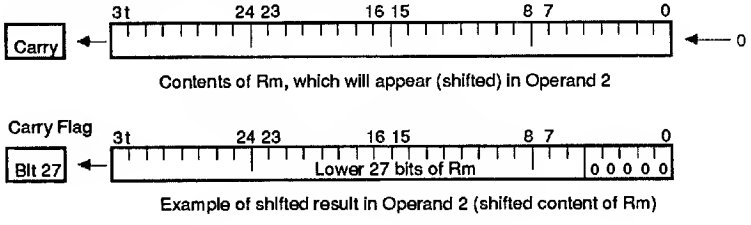
PSR Flags - The operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15) the V flag in the PSR will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

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the PSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

Shifts - When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register as shown in Figure 8.

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

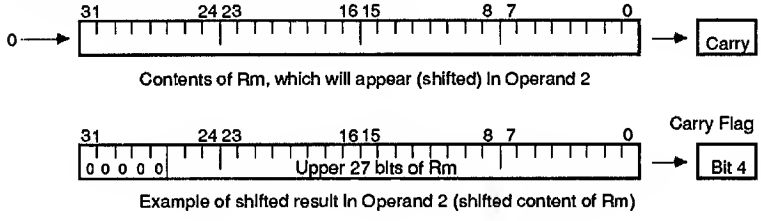
FIGURE 9. LOGICAL SHIFT LEFT (LSL)



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

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

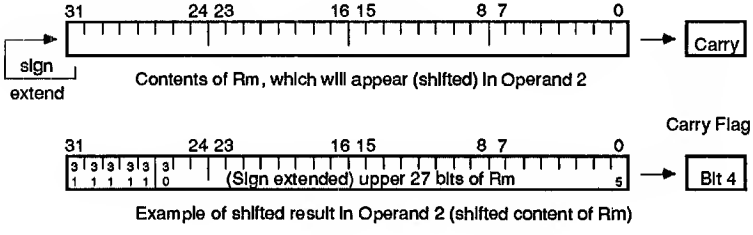
FIGURE 10. LOGICAL SHIFT RIGHT (LSR)



The form of the shift field which might be expected to correspond to LSR 0 is used to encode LSR 32, which has the zero result, with bit 31 of Rm as the carry output. Logical shift right zero is redundant, as it is the same as logical shift left zero. Therefore, the assembler converts LSR 0, ASR 0, and ROR 0 into LSL 0, and allows LSR 32 to be specified.

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

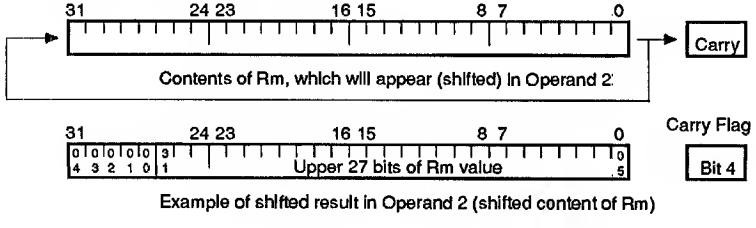
FIGURE 11. ARITHMETIC SHIFT RIGHT (ASR)



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

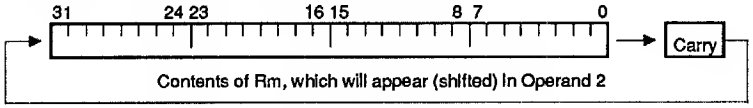
Rotate Right (ROR) operations reuse the bits which "overshoot" in a logical shift right operation by wrapping them around at the high end of the result. For example, the effect of a ROR 5 is:

FIGURE 12. ROTATE RIGHT (ROR)



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

FIGURE 13. ROTATE RIGHT EXTENDED (RRX)



Register-Based Shift Counts - Only the least significent byte of the contents of Rs is used to determine the shift amount. If this byte is zero, the unchenged 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 cerry output.

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

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

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

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

Immediate Operand Rotation - The immediate operand rotate field is a 4-bit unsigned integer which specifies a shift operation on the 8-bit immediate value. The immediate value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many command constants to be generated, for example all powers of two. Another example is that the 8-bit constant may be aligned with the PSR flags (bits zero, one, and 26 to 31). All the flags can thereby be initialized in one TEQP instruction.

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

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

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

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

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

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

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

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

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

Syntax:

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

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

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

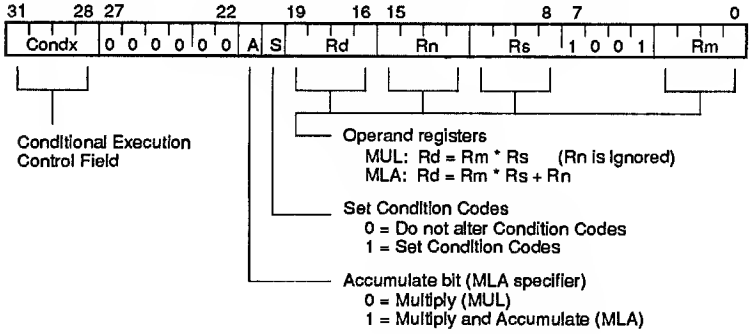
where *Op2*	Is *Rm{<shift>}* or, *<expression>*
cond	Two-character condition mnemonic, see Condition Code section.
S	Set condition codes if S present (Implied for CMP, CMN, TEQ, TST).
P	Make Rd = R15 in instructions where Rd is not specified, otherwise Rd will default to R0. (Used for changing the PSR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

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

Examples:

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

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

SUB         R4, R5, R7 LSR R2   ; Logical Right Shift R7 by the number in the bottom byte of R2, subtract
                                ; the result from R5, and put the answer into R4.
                                : Equivalent to: R4 = R5 - (R7>>R2).

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

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

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

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

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



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

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

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

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

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

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

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

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

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

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

Syntax

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

Notes:

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

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

Examples:

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

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

Notes:

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

Load/Store Value from Memory (LDR,STR)

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

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

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

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

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

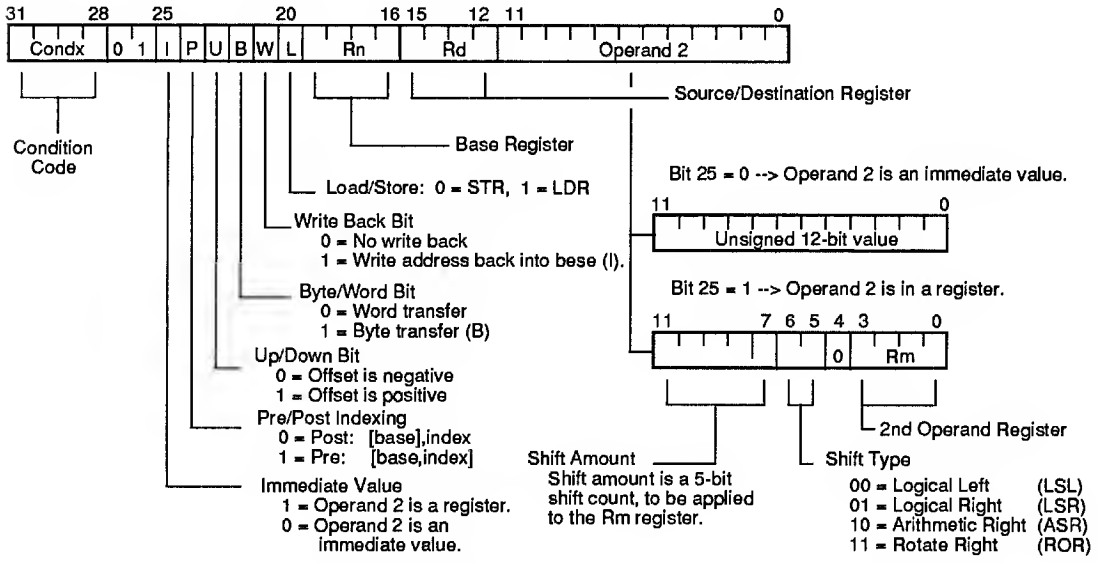
Shifted Register Offset - The eight shift control bits are described in the data processing instructions, but the register specified shift amounts are not implemented in this instruction class.

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

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

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

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



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

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

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

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

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

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

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

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

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

Syntax:

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

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

Address Variants:

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

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

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

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

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

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

Examples (Pre-Index and Optional Increment):

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

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

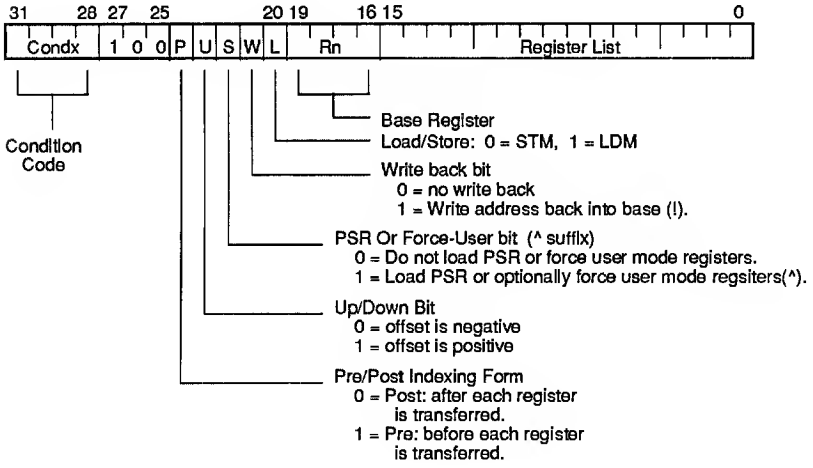
Examples (Post-Index and Increment):
In each of these examples, the effective offset is added to the Rn (base pointer) register after using the Rn register as the effective address. That is, Rn is then updated unconditionally, regardless of any ! suffix.

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

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

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

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



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

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

Addressing Modes - The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. This is illustrated in Figures 17 and 18.

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

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

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

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

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

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

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

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

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

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

To illustrate the various load/store modes, consider the transfer of R1, R5 and R7 in the case where Rn = 1000H and write back of the modified base is required (W = 1). These figures show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 1000H unless it was also in the transfer list of the load multiple register instruction. Then it would have been overwritten with the loaded value.

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

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

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

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

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

FIGURE 17. INCREMENTING INDEX

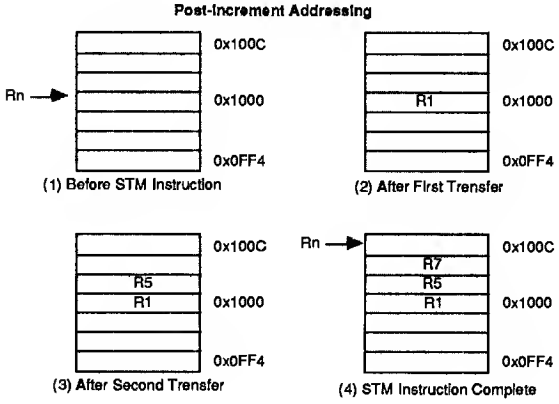
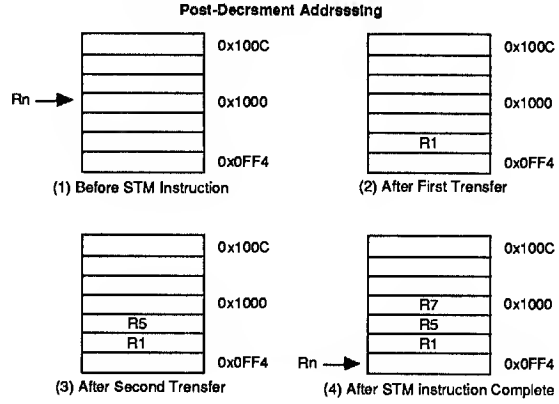


FIGURE 18. DECREMENTING INDEX



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

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

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

Syntax:

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

where
- *cond* Is an optional 2-letter condition code common to all instructions.
- *mode* Is any of: IA, IB, DA, or DB.
- *Rn* Is a valid register name: R0-R15, SP, LK, or PC.
- *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 - There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks, or for other purposes. The names and instruction bit values are:

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

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

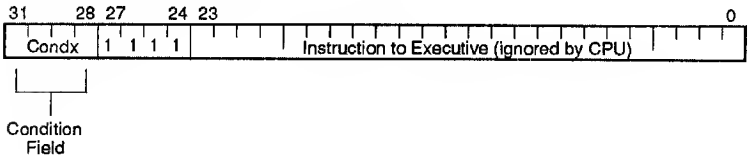
Examples

```
LDMIA   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;

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

FIGURE 19. SOFTWARE INTERRUPT (SWI)



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

The Software Interrupt (SWI) instruction is used to enter supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change, with execution resuming at 0x08. If this address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

Return from the Supervisor - The PC and PSR are saved in R14_svc upon executing the software interrupt trap with the PC adjusted to point to the word after the SWI instruction. MOVS R15, R14_svc will return to the user program, restore the user PSR and return the processor to user mode.

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

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

Syntax:

SWI{cond} <expression>

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

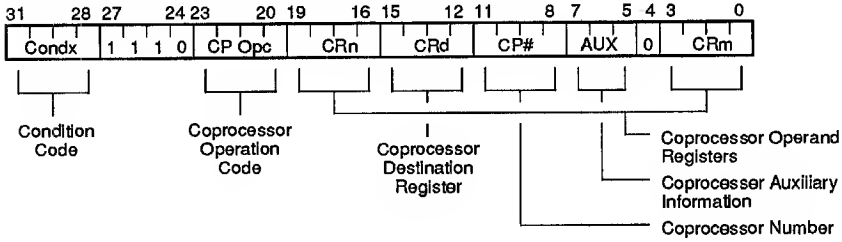
Examples:

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

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

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

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

08h     B       Super                    ; SWI entry point
        ..
Super   STMOB   SP!,{r0,r1, r2, R14}     ; Save working registers.
        BIC     r1, r14, CC_Mask         ; Strip condx codes from SWI instruction address.
        LDR     R0, [R1, -4]             ; Get copy of SWI instruction.
        BIC     R0, R0, 0xFF000000       ; Get lower 24 bits of SWI, only.
        MOV     R1, SWI_Table            ; Get absolute address of PC-relative table.
        LDR     PC, [R1, R0 LSL 2]       ; Jump indirect on the table.

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

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

FIGURE 20. COPROCESSOR DATA OPERATIONS (CDO)



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

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

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

By convention, the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, placing the result into CRd.

Syntax:

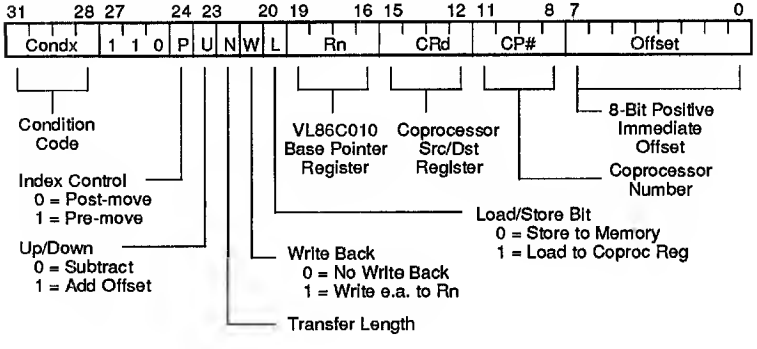
CDO{cond} cp#,<expression1>, CRd, CRn, CRm{,<expression2>}

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

Examples:

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

Syntax:

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

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

Address Variants:

Address expression: An expression evaluating to a relocatable address:

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

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

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

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

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

where expression A signed 9-bit expression (including the sign).

Rn *Valid register names: R0-R15, SP, LK, or PC. If Rn* = PC, the assembler will subtract 8 from the expression to allow for processor address read ahead.

Examples (Pre-Index):

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

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

Examples (Post-Index):

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

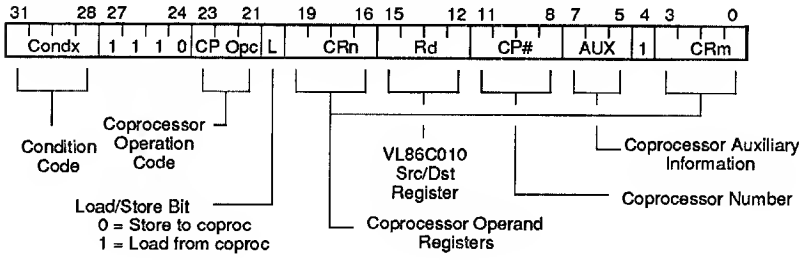
```
        STC     3, CR1, [R2], 8     ; *R2 = CR1. Then R2 += 8.
        LDC     3, CR1, [R0], 16    ; CR1 = *R0. Then R0 += 16.
        LDCEQL  3, CR2, [R5], 4     ; if (Zflag) CR2 = *R5, and then (implicitly), R5 += 4.
                                    ; Use the long option (probably to store multiple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

Examples (Expression):

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

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

FIGURE 22. COPROCESSOR REGISTER TRANSFER (MCR,MRC)



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

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

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

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

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

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

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

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

Syntax:

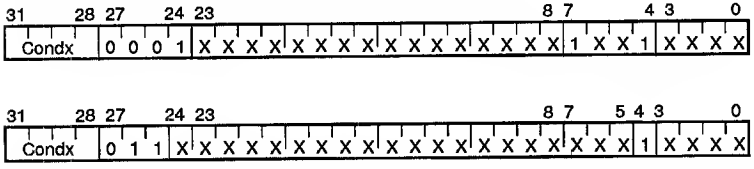
MCR/MRC{cond} CP#,<expression1>, Rd, CRn, CRm{,<expression2>}

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

Examples:

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

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

FIGURE 23. UNDEFINED (RESERVED) INSTRUCTIONS



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

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

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

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

INSTRUCTION SET EXAMPLES

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

Using Conditional Instructions -

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

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

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

(2) Absolute value

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

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

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

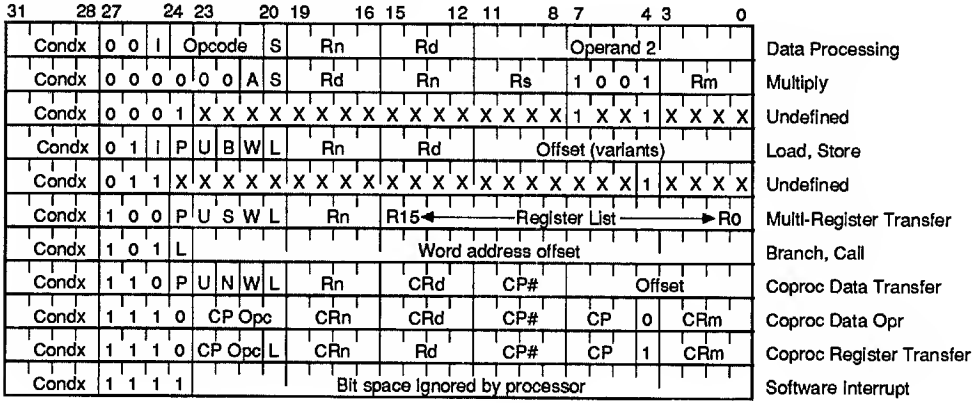
(4) Combining discrete and range tests

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

Division and Remainder

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

FIGURE 24. INSTRUCTION SET SUMMARY



Pseudo Random Binary Sequence Generator - It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift register-based generators with exclusive or feedback rather like a cyclic redundancy check generator. Unfortunately, the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition). The basic algorithm is Newbit = bit_33 xor bit_20, shift left the 33-bit number and put in Newbit at the bottom. Then do this for all the Newbits needed i.e. 32 of them. Luckily, this can all be done in 5S cycles:

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

Multiplication by Constant:

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

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

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

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

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

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

(4) Multiplication by 6

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

(5) Multiply by 10 and add in extra number

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

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

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

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

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

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

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

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

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

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

rather than by:

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

Loading a Word with Unknown Alignment:

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

Sign Extension of Partial Word

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

Return, Setting Condition Codes

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

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

MACHINE CODE INSTRUCTIONS

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

Appendix A.1 Condition Codes

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

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

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

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

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

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

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

Appendix A.2 Miscellaneous

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

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

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

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

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

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

Appendix A.3 Reserved (Undefined) Instructions

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

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

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

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

Appendix A.4 Shifts

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

Appendix A.5 ADC - Arithmetic Add with Carry

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

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

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

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

Operational Function: Rd = Rn + Op2 + Carry

Flags Effected: N, Z, C, V

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

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

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

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

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

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

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

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

Operational Function: Rd = Rn + Op2

Flags Effected: N, Z, C, V

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

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

Examples:

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

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

Appendix A.7 AND - Logical AND

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

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

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

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

Operational Function: Rd = Rn AND Op2

Flags Effected: N, Z, C

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

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

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

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

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

Rm shift Rs

Rm shift expression1

Rm RRX

expression2

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

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

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

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

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

Examples:

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

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

Appendix A.8 B - Branch

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

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

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

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

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

Operational Function: Jump to PC-relative address.

Flags Effected: (none)

Syntax: B *address_expression*

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

Examples:

```
BEQ   ContIn_5        ; Relative branch to CONTIN_5 label.
B     0x3800000       ; Jump into ROM space.
```

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

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

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

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

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

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

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

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

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

Flags Effected: N, Z, C

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

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

Examples:

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

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

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

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

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

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

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

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

Flags Effected: (none)

Syntax: BL address_expression

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

Examples:

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

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

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

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

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

Appendix A.11 CDO - Coprocessor Data Operations

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

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

In actual fact, only coprocessor number and the CPU instruction's opcode bits are required by the hardware; all other fields are assigned within CASM by convention only. The assembler will accept information and assign values to the various fields as defined below.

As with all coprocessor instructions, depending upon hardware design, they may hang the CPU up if they are executed without there being a hardware coprocessor that can respond to it.

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

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

Flags Effected: (none)

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

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

Examples:

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

Appendix A.12 CMN - Set Negative Compare

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

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

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

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

Intended Usage: Comparison against a negative constant.

Operational Function: Rn + Op2 (result not stored)

Flags Effected: N, Z, C, V

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

where
- *condition* is an optional 2-character condition code. See the Condition Code section.
- *P* (if present) sets PSR bits based upon bits 28-31 of the ALU result.
- *Rn* is any valid register names, such as R0-R15, PC, SP, or LK.
- *Op2* is second operand, and may have any of the following forms:
  - *Rm*
  - *Rm shift Rs*
  - *Rm shift expression1*
  - *Rm RRX*
  - *expression2*
- *Rm* is any valid register names, as per *Rd* or *Rn* above, the operand value.
- *Rs* is a register, per *Rd* above, containing a shift count in range of 1..32.
- *shift* is any of: *ASL, LSL, LSR, ASR,* or *ROR*
- *expression1* is any positive absolute shift count in the range of 1..31.
- *expression2* is any signed expression shiftable into an 8-bit value.

Examples:

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

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

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

Appendix A.13 CMP - Arithmetic Comparison

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

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

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

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

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

Flags Effected: N, Z, C, V

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

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

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

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

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

Rm shift Rs

Rm shift expression1

Rm RRX

expression2

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

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

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

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

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

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

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

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

Appendix A.14 EOR - Logical Exclusive OR

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

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

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

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

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

Flags Effected: N, Z, C

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

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

Examples:

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

Appendix A.15 LDC - Load Coprocessor from Memory

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

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

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

Flags Effected: (none)

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

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

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

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

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

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

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

address can be any of the variations given below:

expression

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

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

[[Rn], expression

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

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

Examples:

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

Appendix A.16 LDM - Load Multiple Registers

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

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

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

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

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

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

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

where

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

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

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

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

Examples:

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

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

Appendix A.17 LDR - Load Register from Memory

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

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

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

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

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

Operational Function: Load register Rd from the effective address.

Flags Effected: (none)

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

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

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

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

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

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

address is eny of the following variations:

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

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

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

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

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

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

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

Appendix A.17 LDR (Cont.)

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

Examples:

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

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

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

Appendix A.18 LEA - Load Effective Address

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

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

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

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

Syntax: LEA *Rd, expression*

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

Example:

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

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

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

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

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

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

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

Appendix A.19 MCR - Move Coprocessor to CPU
Transfer a register from an attached coprocessor to a CPU register, optionally performing some action within the coprocessor. The instruction forces the following items to appear at the coprocessor interface:

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

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

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

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

Flags Effected: (none)

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

where

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

Examples:

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

Appendix A.20 MLA - Multiply and Accumulate

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

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

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

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

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

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

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

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

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

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

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

Examples:

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

Variations: The following exception conditions exist:

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

Appendix A.21 MOV - Move Register or Constant

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

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

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

Operational Function: Rd = Op2

Flags Effected: N, Z, C

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

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

S (if present) sets condition codes based on the value of the operand.

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

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

Rm shift Rs

Rm shift expression1

Rm RRX

expression2

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

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

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

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

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

Examples:

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

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

Appendix A.22 MRC - Move CPU Register to Coprocessor

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

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

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

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

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

Flags Effected: (none)

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

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

Examples:

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

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

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

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

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

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

Operational Function: *Rd = Rm * Rs*

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

Syntax: MUL *Rd, Rm, Rs*

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

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

Examples:

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

Variations: The following exception conditions exist:

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

Appendix A.24 MVN - Move Complement of Register

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

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

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

Operational Function: Rd = 0xFFFFFFFF XOR Op2

Flags Effected: N, Z, C

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

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

Examples:

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

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

Appendix A.25 ORR - Logical OR

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

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

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

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

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

Operational Function: Rd = Rn OR Op2

Flags Effected: N, Z, C

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

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

Examples:

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

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

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

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

Operational Function: Rd = Op1 - Rn

Flags Effected: N, Z, C, V

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

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

Examples: Subtract R5 from a very large constant.

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

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

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

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

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

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

Flags Effected: N, Z, C, V

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

where

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

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

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

Appendix A.28 SBC - Subtract, with Carry

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

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

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

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

Intended Usage: Multi-precision subtraction.

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

Flags Effected: N, Z, C, V

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

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

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

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

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

Appendix A.29 STC - Store Coprocessor to Memory

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

Intended Usage: Store coprocessor register to indicated memory location.

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

Flags Effected: (none)

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

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

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

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

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

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

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

address can be any of the variations given below:

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

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

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

Examples:

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

Appendix A.30 STM - Store Multiple Registers

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

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

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

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

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

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

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

where

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

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

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

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

^ User mode: ^ is ignored.
Non-User mode: Forces reference of user mode register bank.

Note: PSR is always stored if R15 is in the list.

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

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

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

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

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

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

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

Appendix A.31 STR - Store Register to Memory

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

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

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

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

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

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

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

Flags Effected: (none)

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

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

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

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

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

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

address is any of the following variations:

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

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

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

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

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

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

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

Appendix A.31 STR (Cont.)

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

Examples:

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

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

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

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

Intended Usage: Compute the arithmetic difference of two operands.

Operational Function: Rd = Rn - Op2

Flags Effected: N, Z, C, V

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

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

Examples:

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

Appendix A.33 SWI - Software Interrupt

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

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

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

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

Flags Effected: (determined by the supervisor)

Syntax: SWI{*condition*} *operand*

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

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

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

Appendix A.34 TEQ - Set Condition Codes via XOR

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

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

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

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

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

Flags Effected: N, Z, C

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

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

Examples:

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

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

Appendix A.35 TST - Set Condition Codes via AND

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

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

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

2

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

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

Flags Effected: N, Z, C

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

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

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

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

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

Notes:

VLSI TECHNOLOGY, INC.

SECTION 3

VL86C020 32-BIT RISC MICROPROCESSOR WITH CACHE MEMORY

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

VL86C020

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

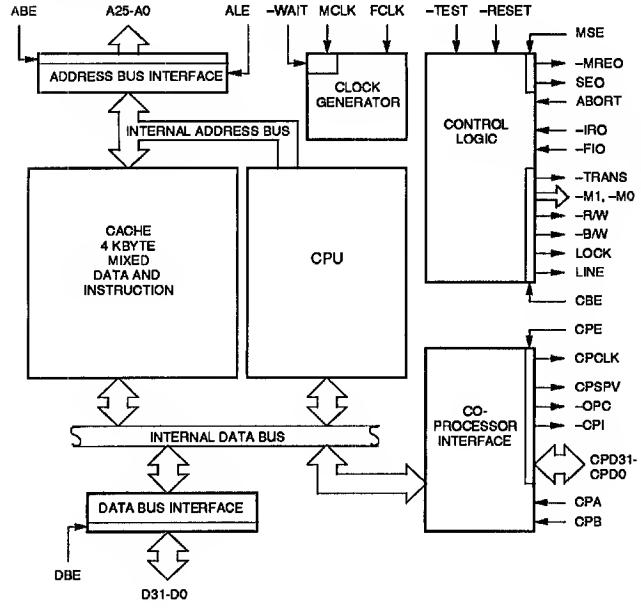
The VL86C020 Acorn RISC Machine (ARM) is a second generation 32-bit general purpose microprocessor system. The device contains both a general purpose CPU and a full cache memory subsystem in the same package. Several benefits are attained by having the CPU and cache within the same device. First, the 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.

BLOCK DIAGRAM

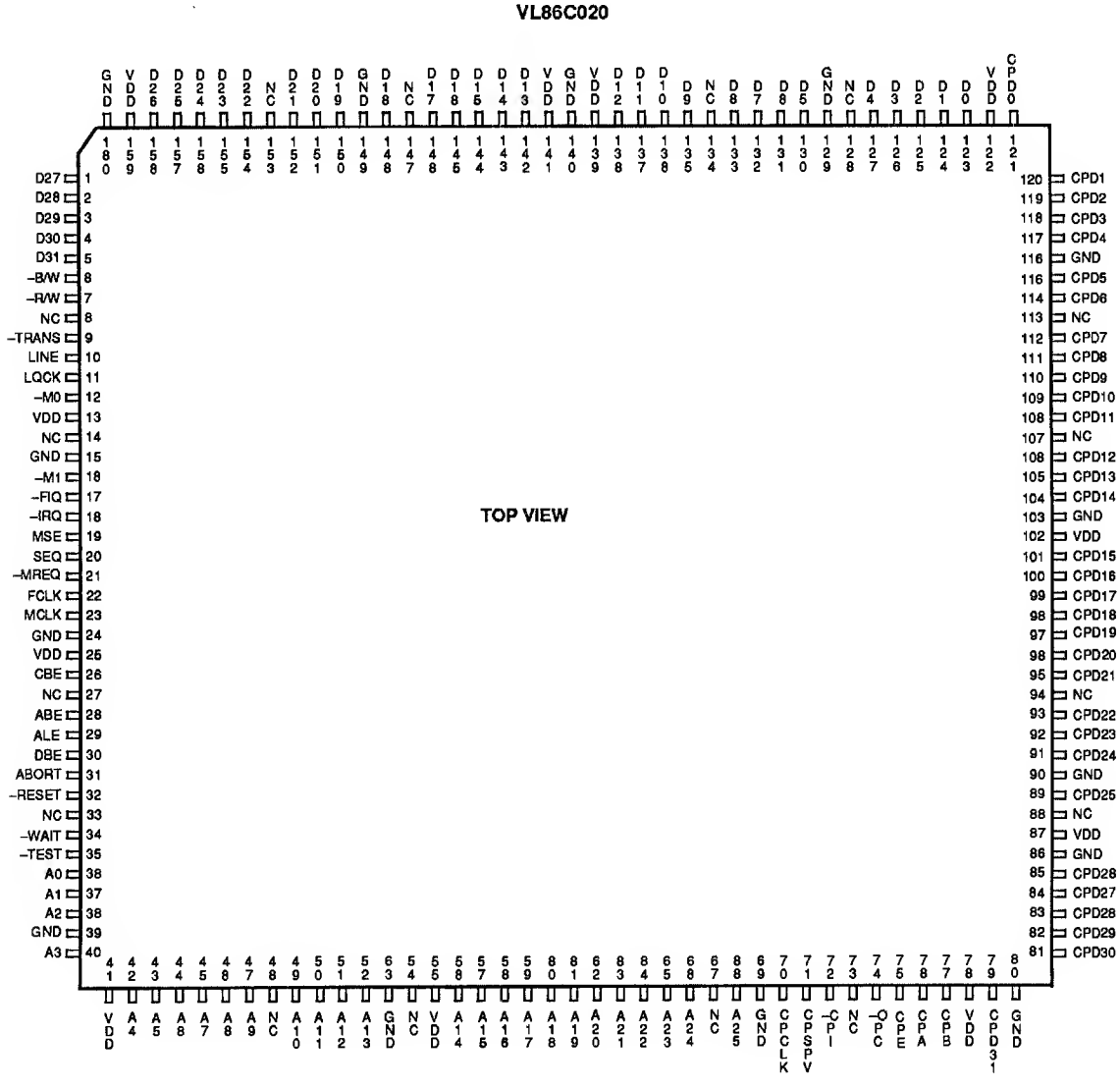


ORDER INFORMATION

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.

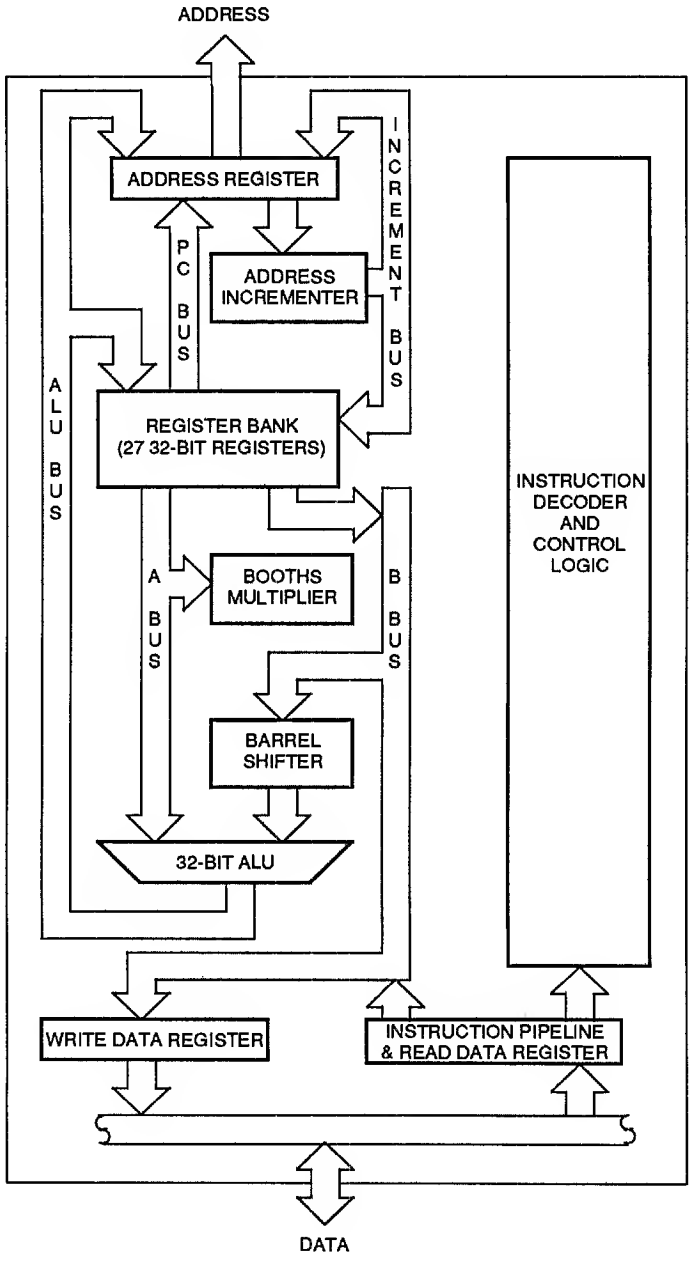
PIN DIAGRAM - PLASTIC QUAD FLATPACK



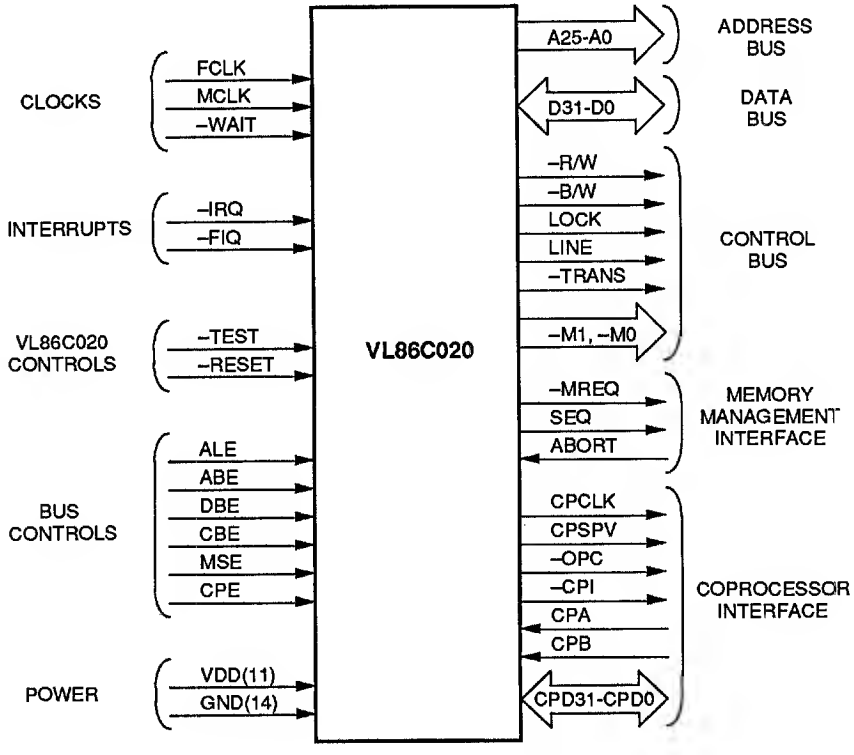
PIN DIAGRAM - PLASTIC PIN GRID ARRAY

	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

CPU BLOCK DIAGRAM



FUNCTIONAL DIAGRAM



SIGNAL DESCRIPTIONS FOR PLASTIC QUAD FLATPACK

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.)

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.

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.)

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				General Usage
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

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.

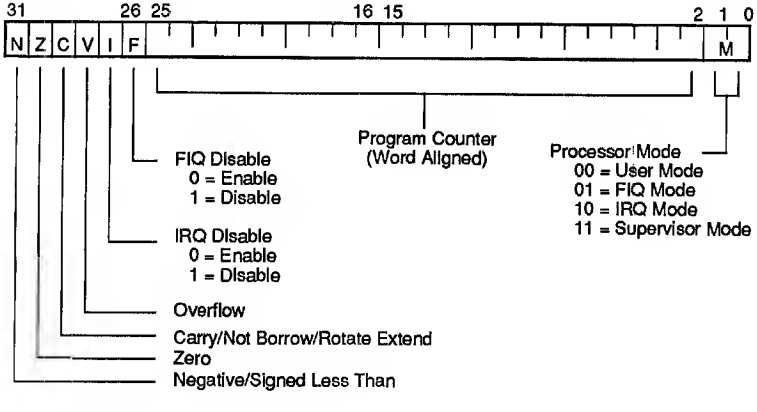
The impact upon execution of an IRQ interrupt is defined in Table 3. The return-from-interrupt sequence is also defined there. This will cause execution to resume at the instruction following the interrupted one, restore the original processor state, and re-enable the IRQ interrupt.

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.

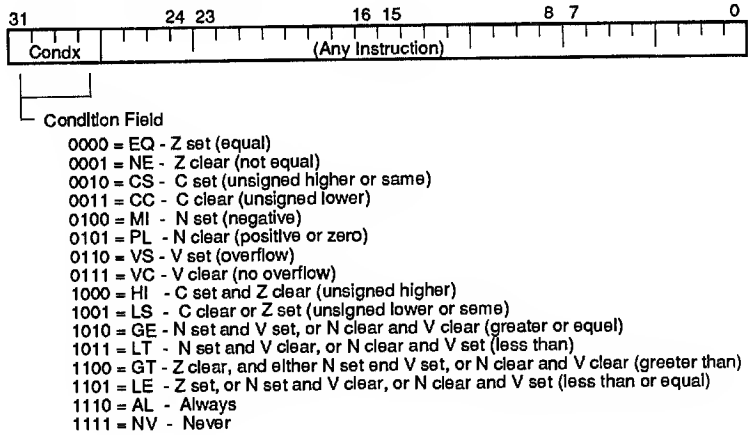
INSTRUCTION SET

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



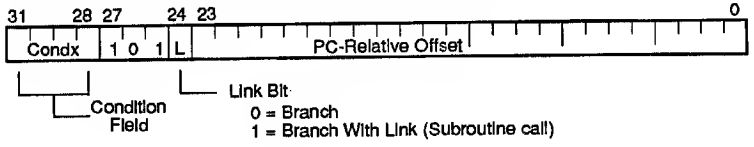
Branch and Branch with Link (B, BL)

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.

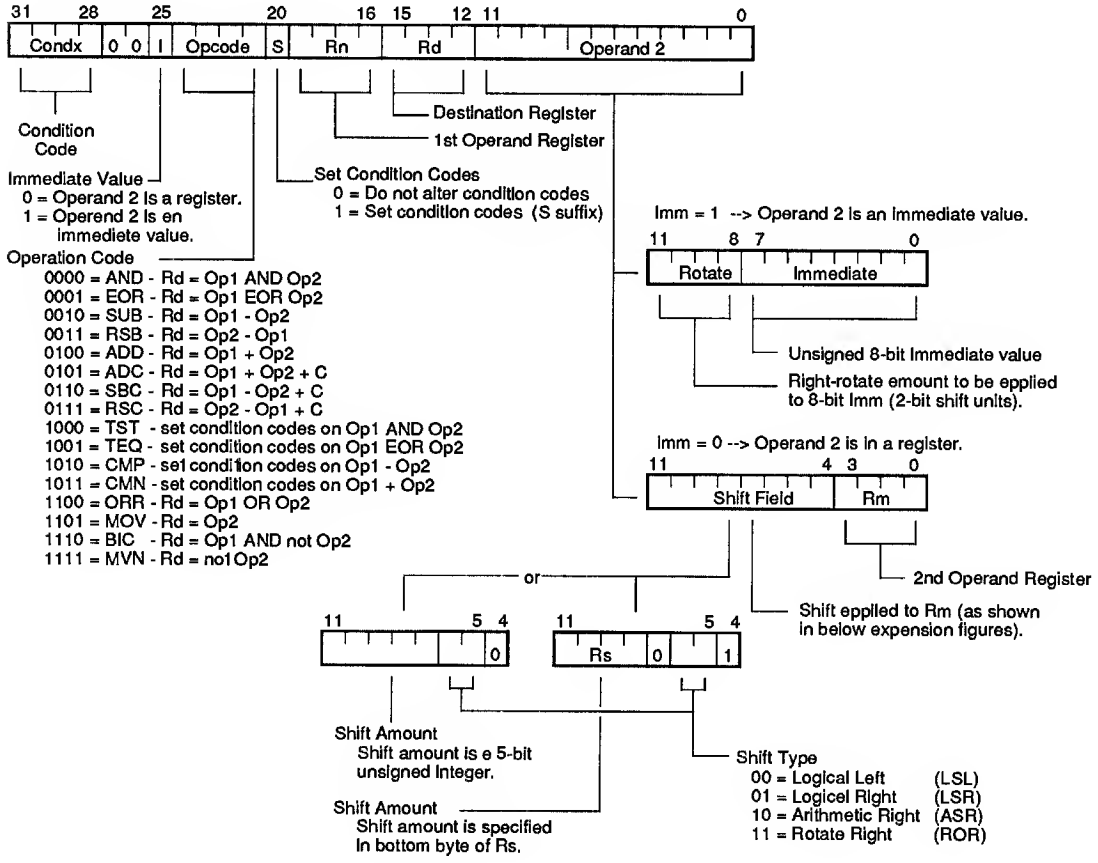
expression is the destination. The assembler calculates the relative (word) offset.

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

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).
```

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.)

DATA PROCESSING OPERATIONS

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

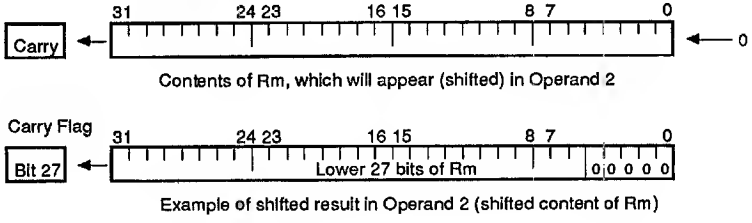
PSR Flags - The operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15), the V flag in the PSR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

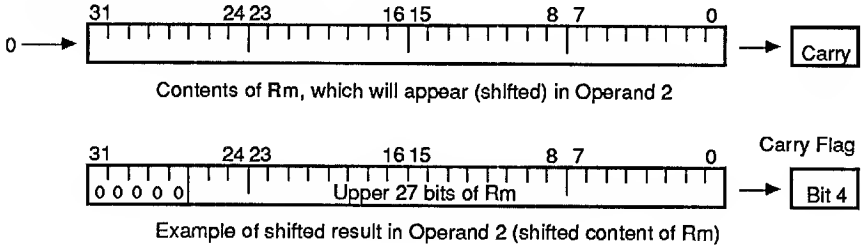
FIGURE 5. LOGICAL SHIFT LEFT (LSL)



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.

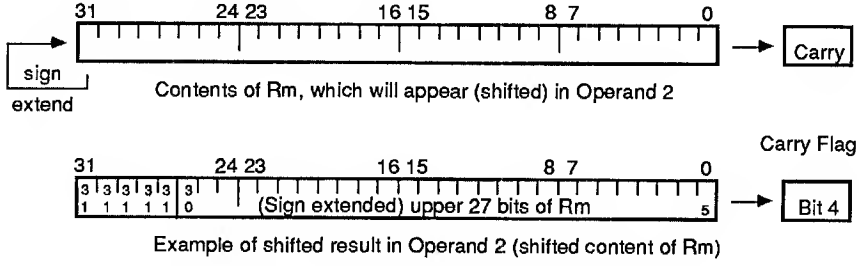
FIGURE 6. LOGICAL SHIFT RIGHT (LSR)



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:

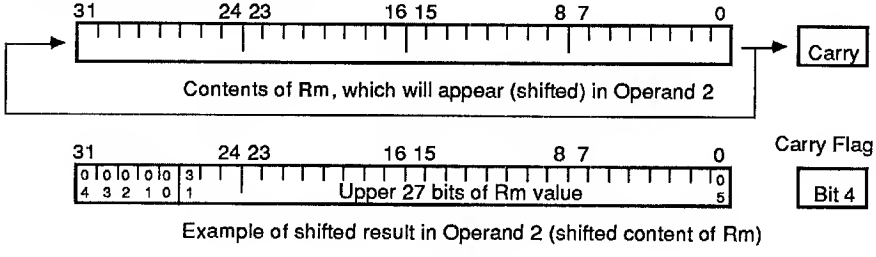
FIGURE 7. ARITHMETIC SHIFT RIGHT (ASR)



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.

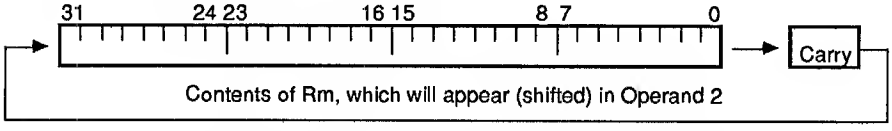
Rotate Right (ROR) operations reuse the bits which "overshoot" in a logical shift right operation by wrapping them around at the high end of the result. For example, the effect of a ROR 5 is:

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 e 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-Besed 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, repeetedly 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; e 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 enebles 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, eccepting 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 epply to VL86C020, but should be adhered to for compatibility.

If the S flag is cleer 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>

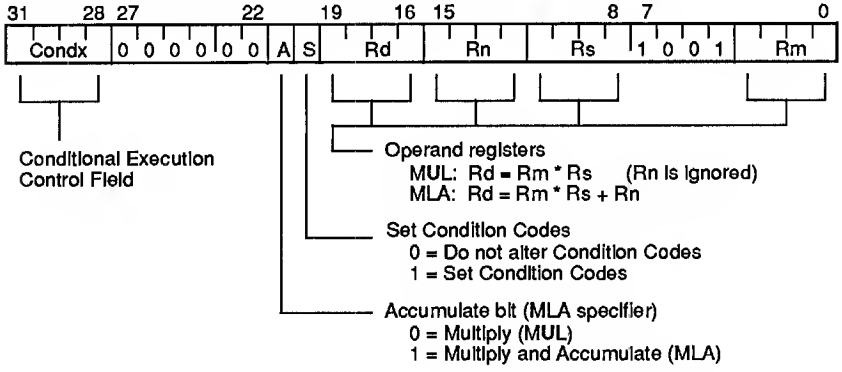
where *Op2*	Is *Rm{<shift>}* or, *<expression>*
cond	Two-character condition mnemonic, see Condition Code section.
S	Set condition codes if S present (implied for CMP, CMN, TEQ, TST).
P	Make Rd = R15 in instructions where Rd is not specified, otherwise Rd will default to R0. (Used for changing the PSR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

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



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

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

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

Both forms of the instruction work on operands which may be considered as signed (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.

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 conditlon 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.

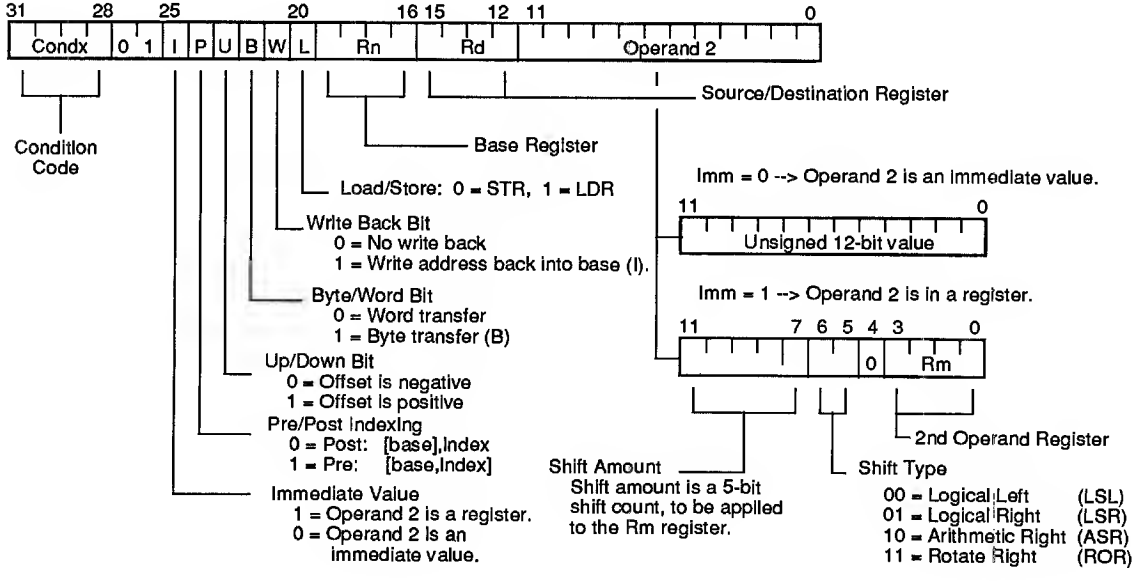
The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant, since the old base value can be retained by setting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

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 I 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 I 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.
```

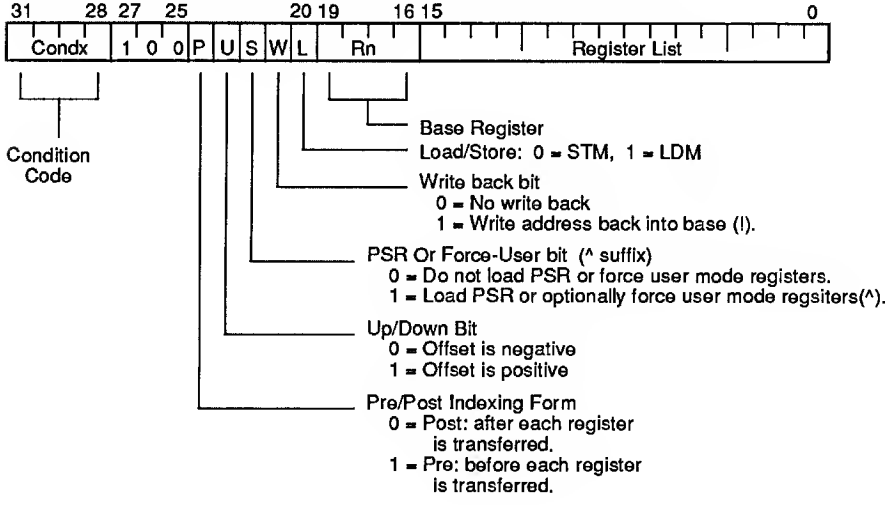
Examples (Post-Index and Increment):
In each of these examples, the effective offset is added to the Rn (base pointer) register after using the Rn register as the effective address. Rn is then updated unconditionally, regardless of any "!" suffix.

```
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

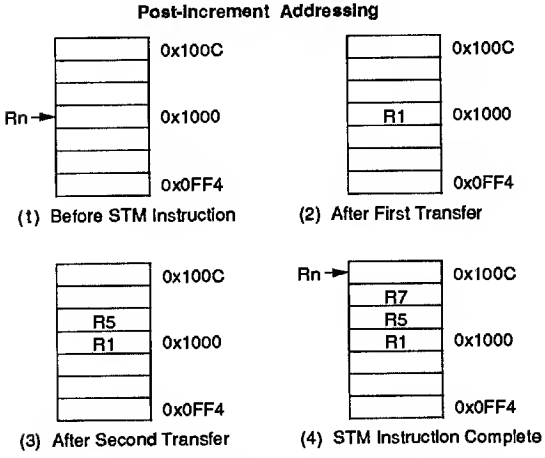
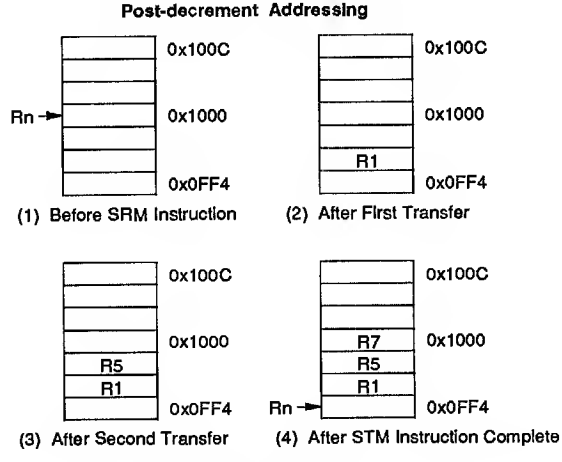


FIGURE 14. DECREMENTING INDEX



Overwriting of registers stops when the abort happens. The aborting load will not teke piece, 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 alweys the last register to be overwritten.)

The base register is restored to its modified velue if write beck 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 ebort trap is taken when the load multiple has completed, and the system software must undo eny base modification (end resolve the ceuse of the abort) before restarting the instruction.

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

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

Assembler Syntax:

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

where *cond* Is an optionel 2-letter condition code common to all instructions.
mode Is any of: FD, ED, FA, EA, IA, IB, DA, or DB.
Rn Is a valid register name: R0-R15, SP, LK, or PC.
Rlist Can be a single register (as described above for Rn), or mey 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 loed 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 upwerds
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 ere not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

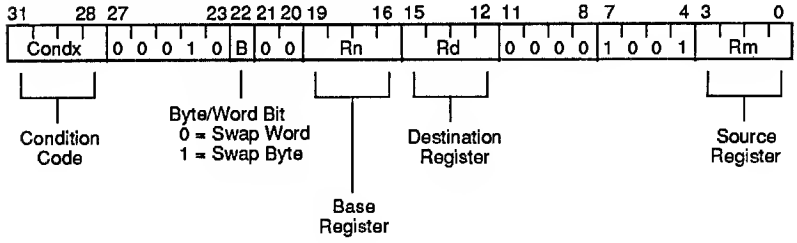
Exemples

```
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}      ; Seve 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:

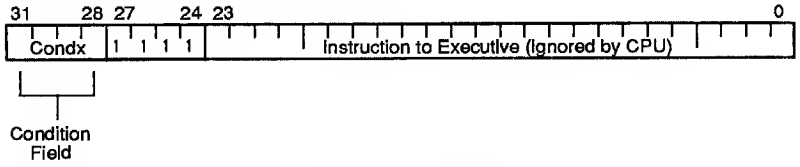
SWP*{cond}{B}* Rd,Rm,[Rn]

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.

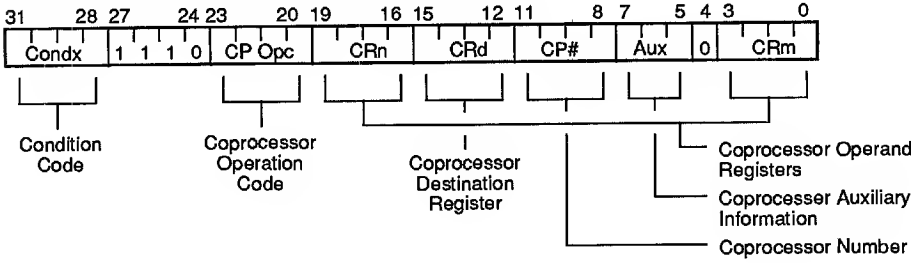
Examples:

```
        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   SPI,{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.

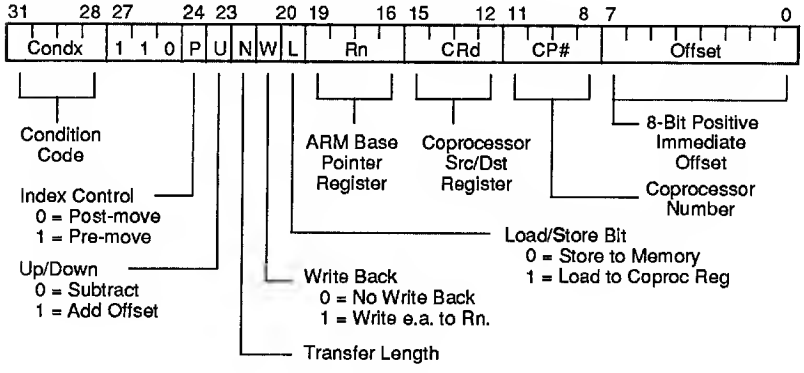
Assembler Syntax:

	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.

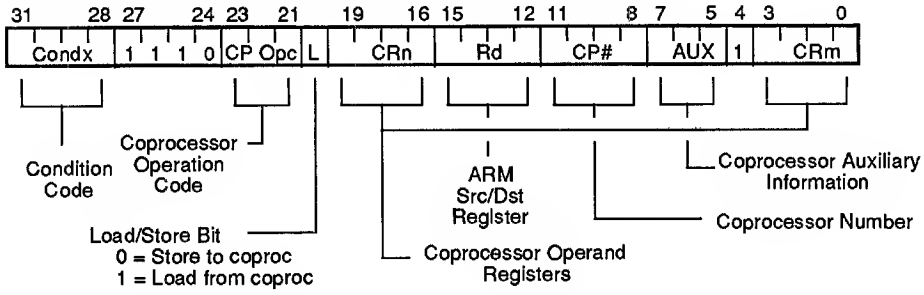
```
        STC     3, CR1, [R2], 8!    ; *R2 = CR1. Then R2 += 8.
        LDC     3, CR1, [R0], 16    ; CR1 = *R0. Then R0 += 16.
        LDCEQL  3, CR2, [R5], 4     ; if (Zflag) CR2 = *R5, and then (implicitly), R5 += 4.
                                    ; Use the long option (probably to store multiple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

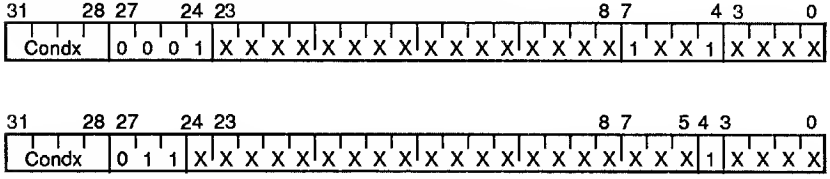
where

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.
```

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
```

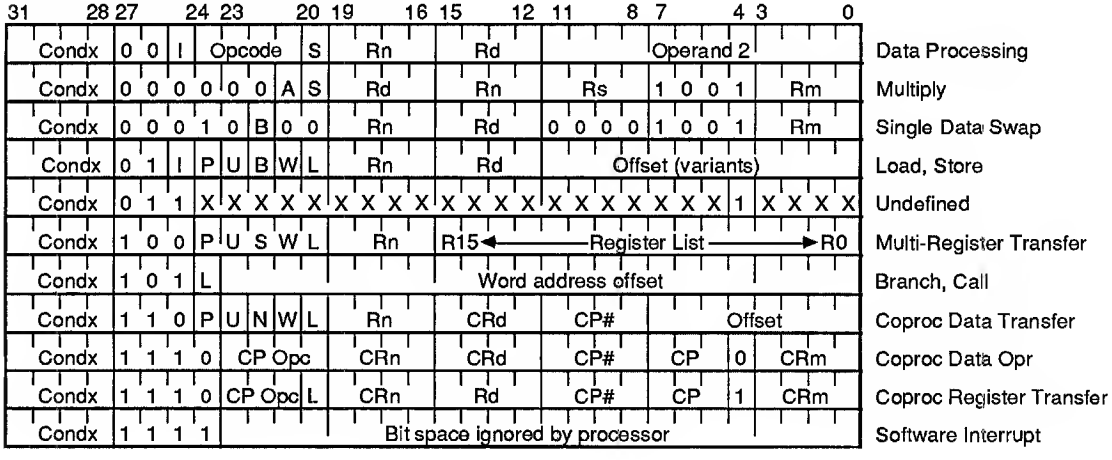
(4) Combining discrete and range tests

```
        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 allgned.
        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.

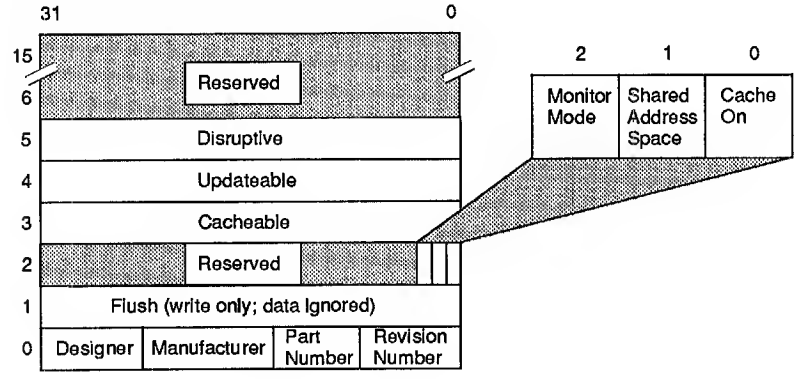
Non-cacheable Areas of Memory Certain areas of the processor's memory map may be uncacheable. For instance, when using MEMC, the area between 3000000H-3400000H corresponds to I/O space, and must be marked as uncacheable to stop the data being stored in the cache. When the processor is polling a hardware 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 eny velue 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 cacheeble
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

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.

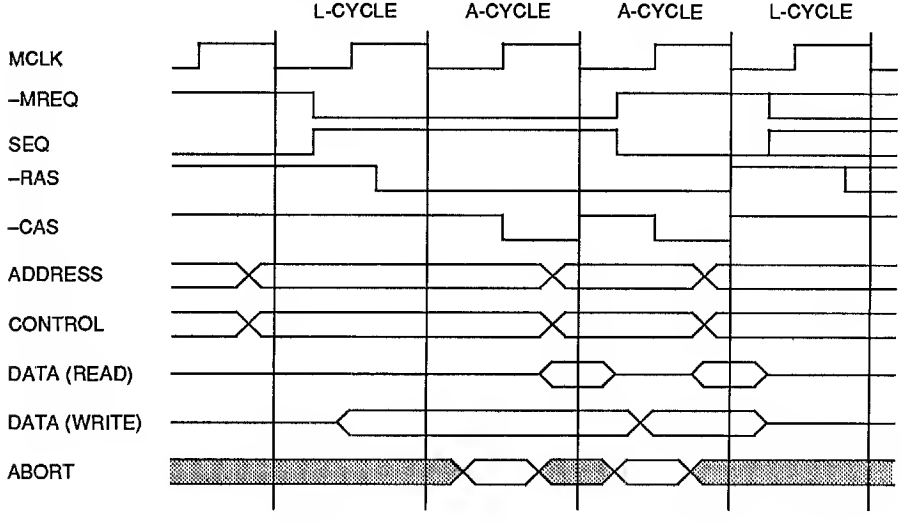
Register 5: Disruptive Areas (Read/Write) - This is a 32-bit register that allows any of the thirty-two, 2 Mbyte areas of the 64 Mbyte processor virtual address space to be marked as disruptive:

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

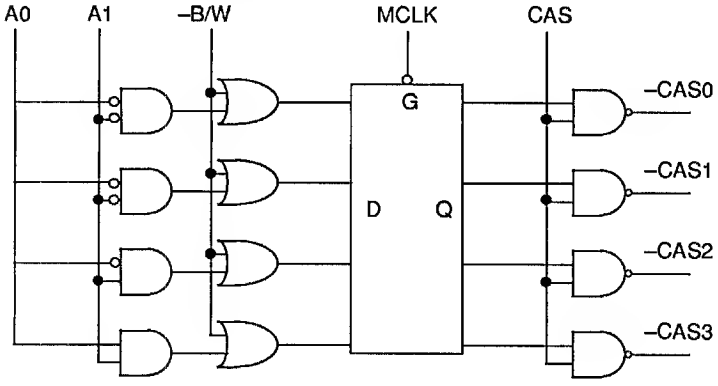
The processor address bus provides byte addresses, but instructions are always words (where a word is four bytes) and data quantities are usually words. Single data transfers (LDR,STR,SWP) can, however, specifiy that a byte quantity is required. The –B/W control line is used to request a byte from the 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.

FIGURE 25. DATA SWAP OPERATION

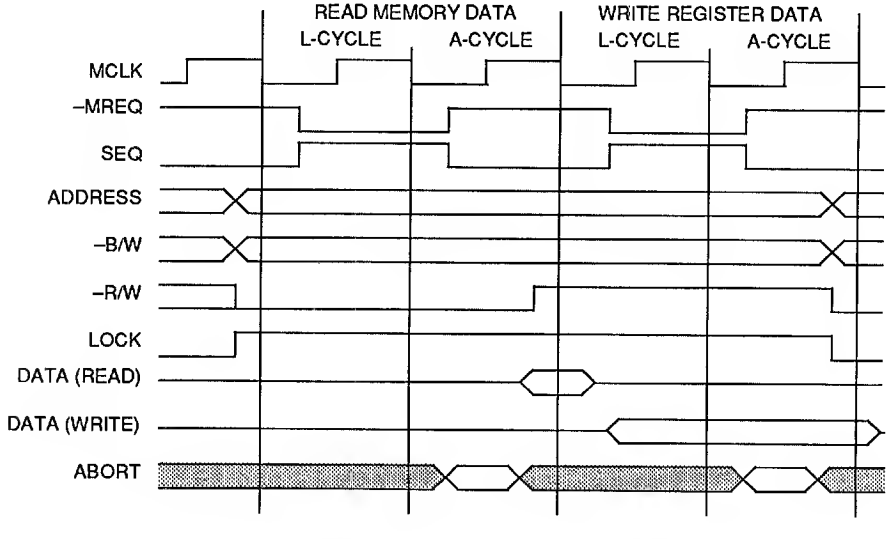
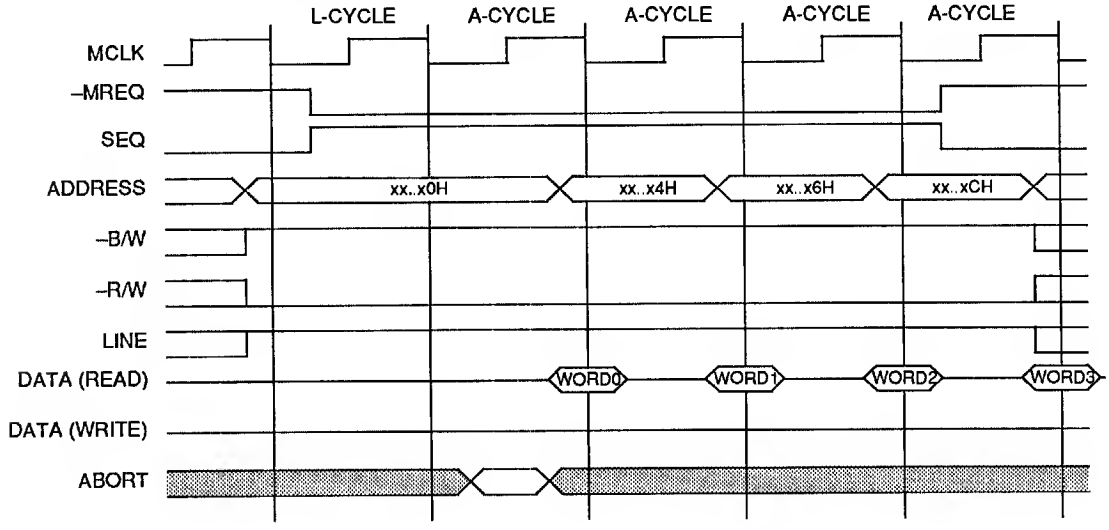


FIGURE 26. LINE FETCH OPERATION



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).

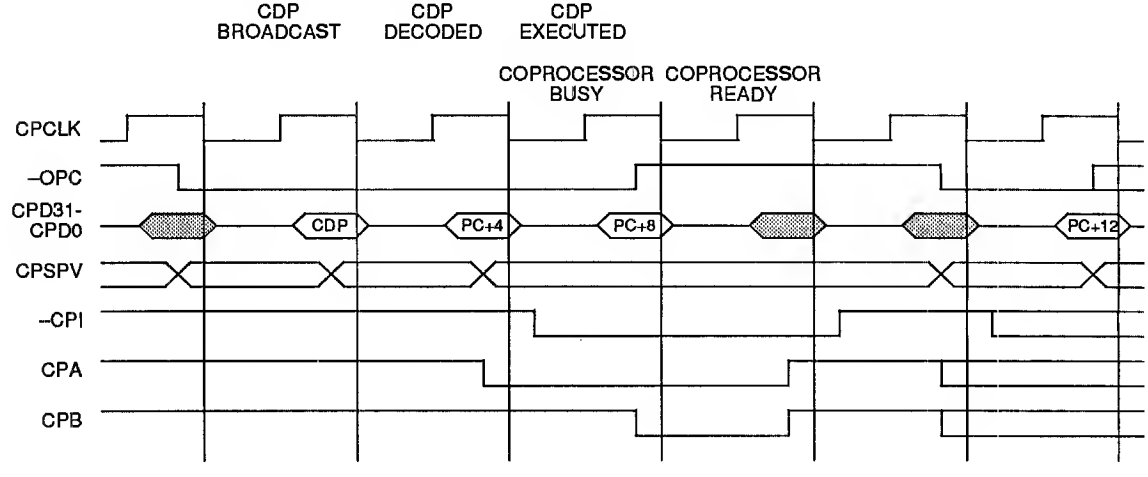
Coprocessor Present/Absent - The CPU takes –CPI low whenever it starts to execute a coprocessor (or undefined) instruction (this will not happen if the instruction fails to be executed because of the condition codes). Each coprocessor will have a copy of the instruction, and can inspect the CP# field 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.

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)

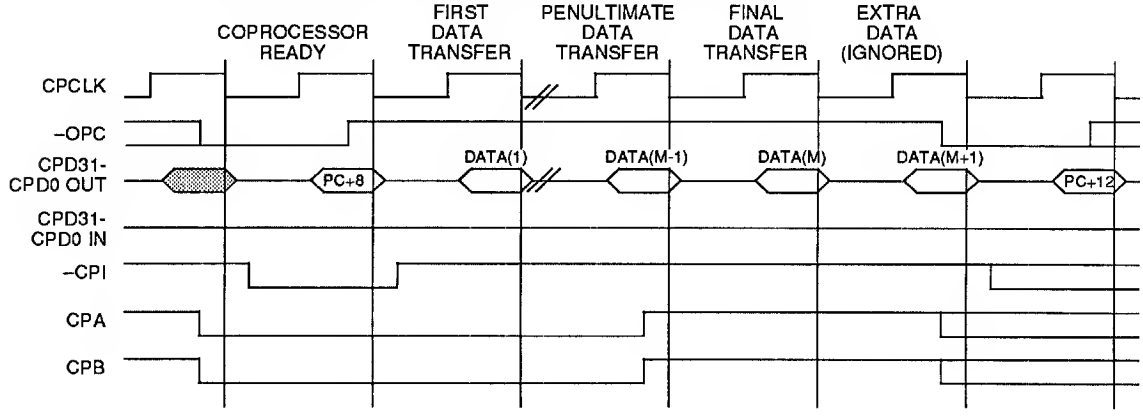
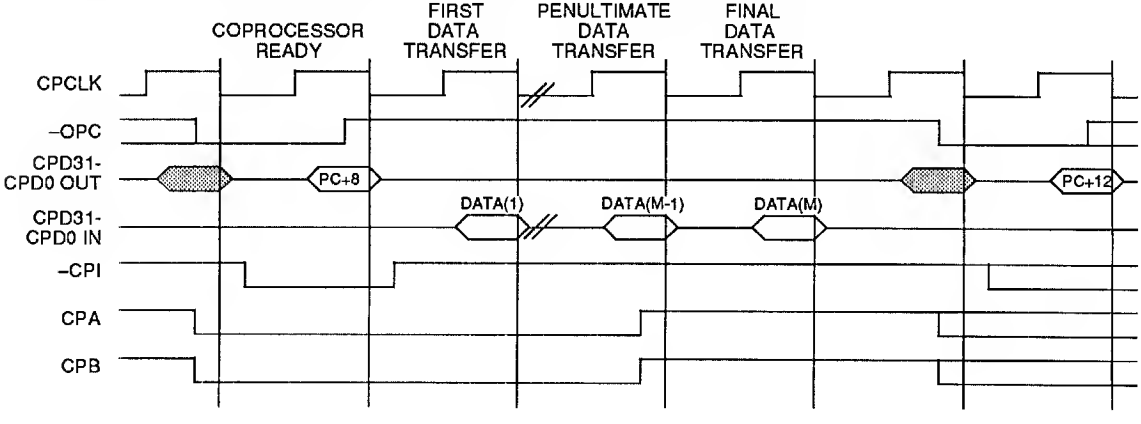


FIGURE 29. COPROCESSOR DATA TRANSFER (FROM COPROCESSOR TO MEMORY)



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

FIGURE 30. COPROCESSOR REGISTER TRANSFER (LOAD FROM COPROCESSOR)

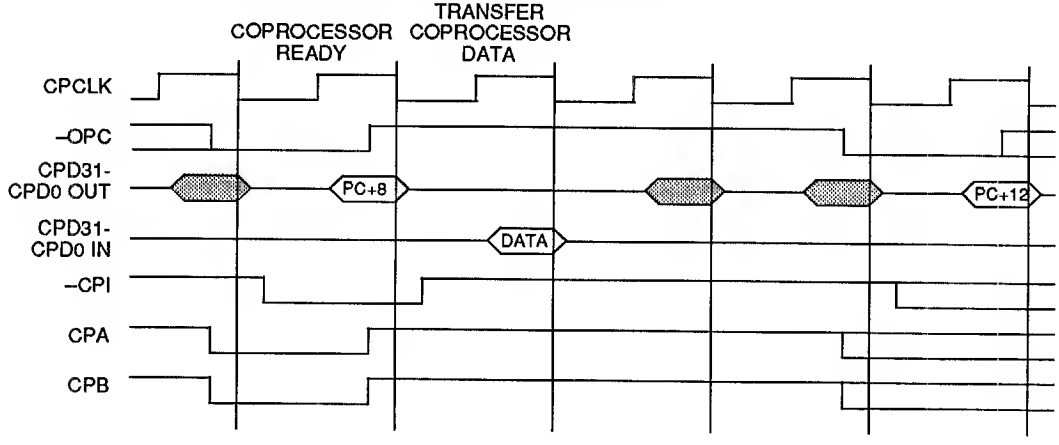
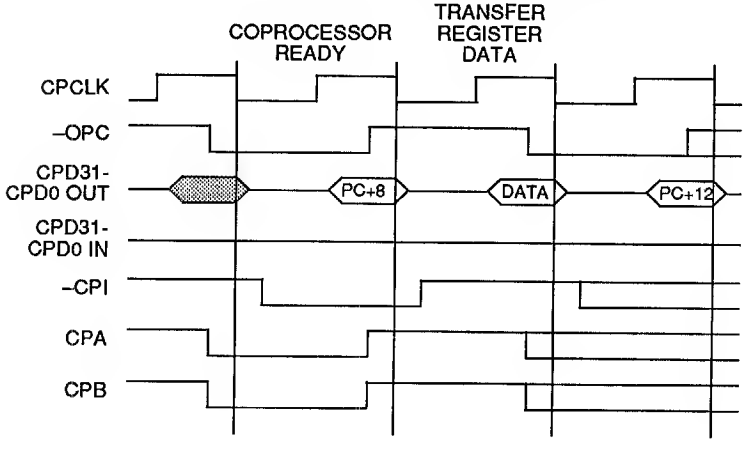


FIGURE 31. COPROCESSOR REGISTER TRANSFER (STORE TO COPROCESSOR)



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 opereting 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 e task switch happens, the operatlng 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-eneble 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 e CPU register. The coprocessor must stay busy while it performs the floeting 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 end 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.

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),	1	Read		PC+8	(PC+8)		
DEST=PC	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.

	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 colcked =>			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.

	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.

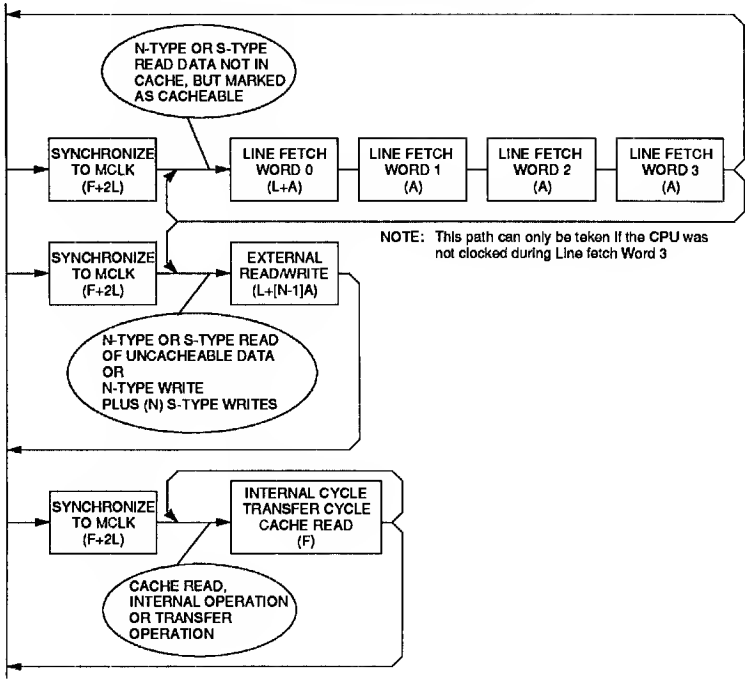
3. Internal operation. These complete in a single F-cycle (although some are absorbed during line fetches).

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.

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.

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.

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

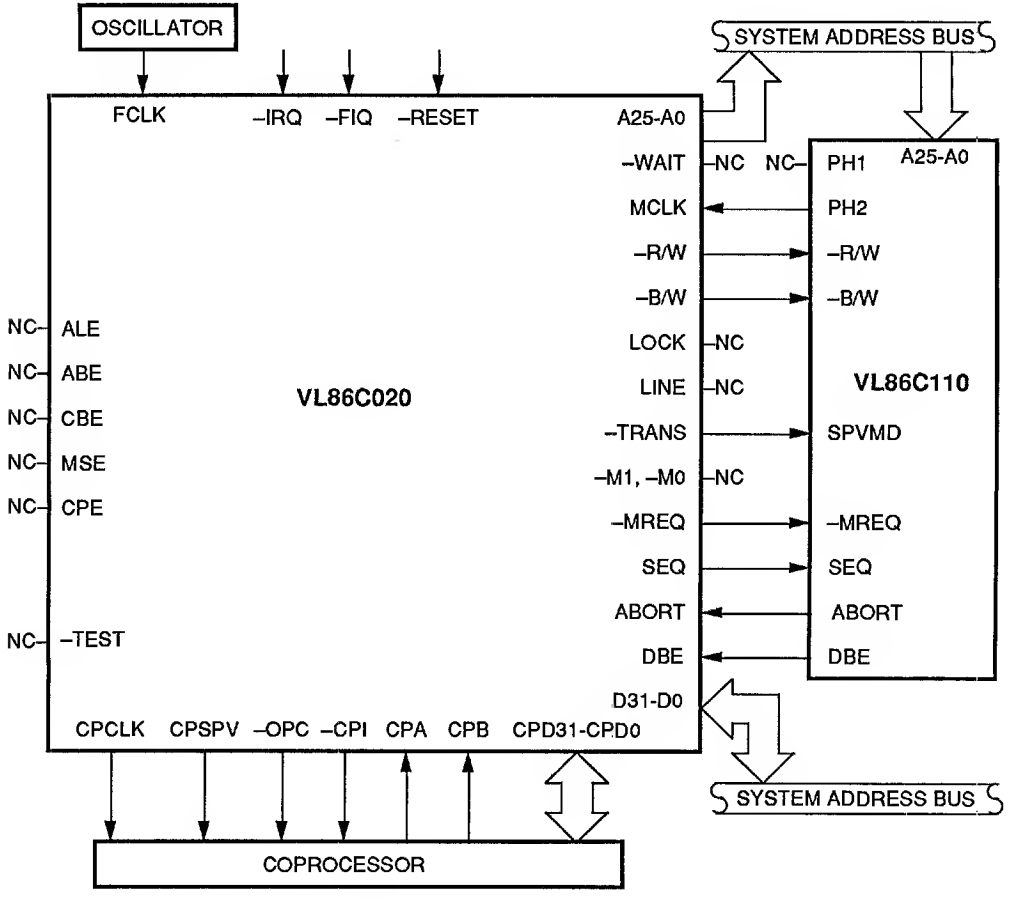
1. VL86C020 is packaged in a 160-pin quad flatpack; VL86C010 uses an 84-pin plastic leaded chip carrier (PLCC) package.
2. VL86C020 does not require non-overlapping clocks for timing memory accesses. When using VL86C020 with MEMC, the PH2 clock output of MEMC should be connected to the MCLK input of 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
–OPC	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

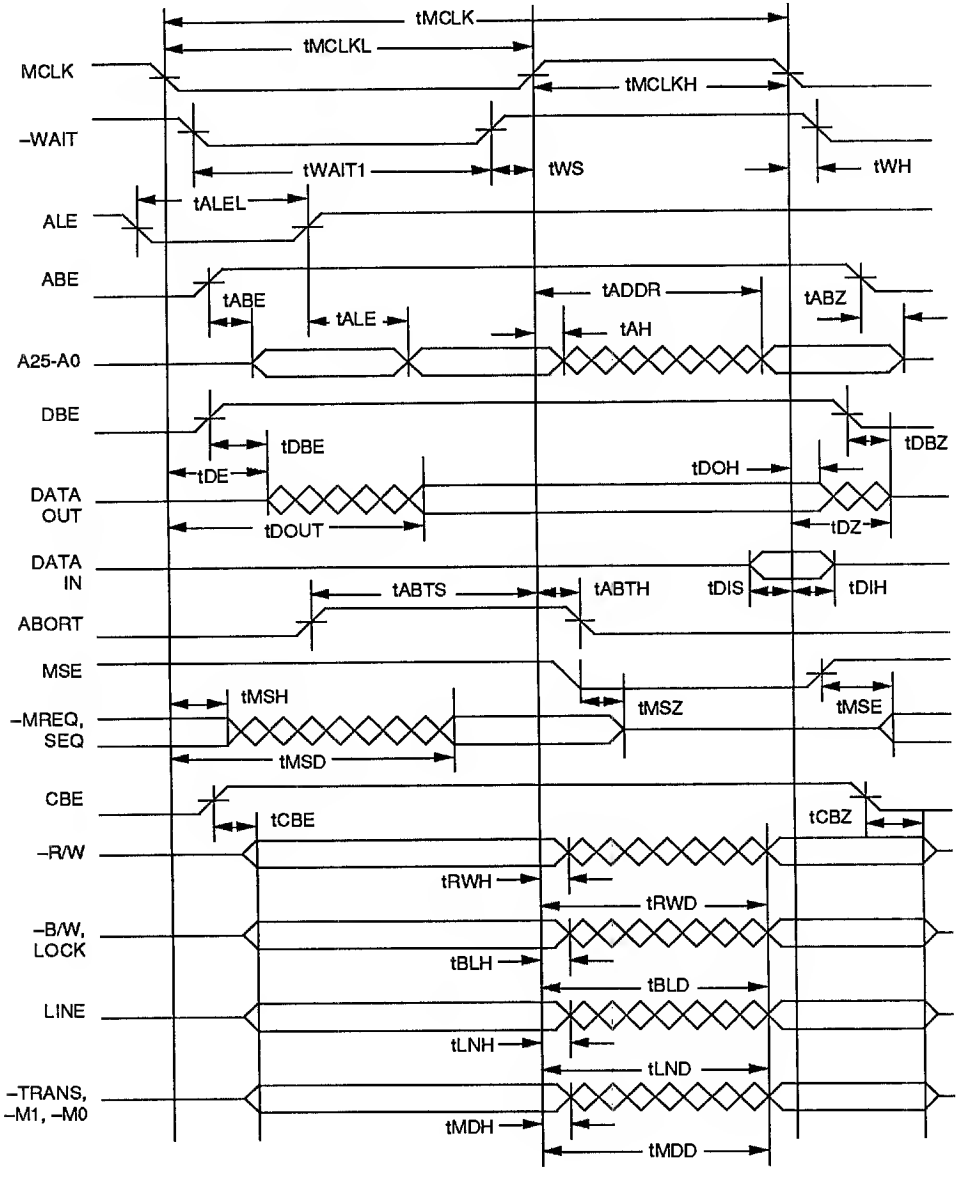


FIGURE 35. COPROCESSOR INTERFACE TIMING

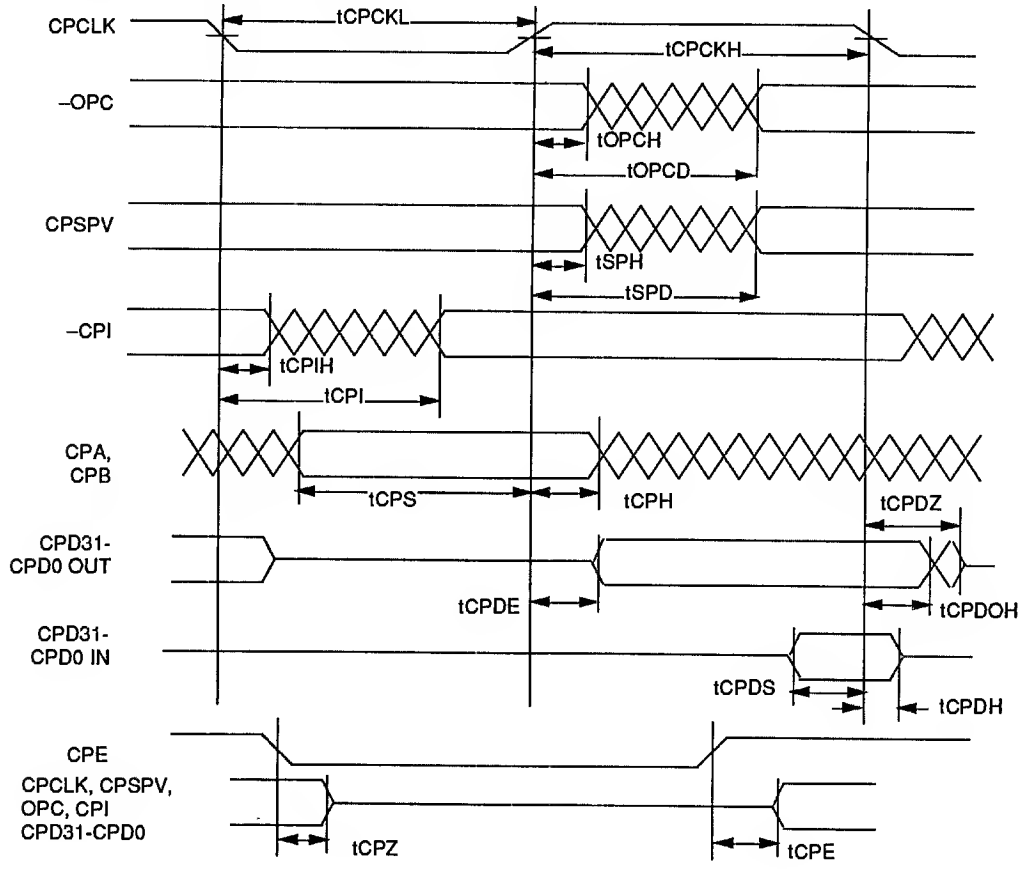
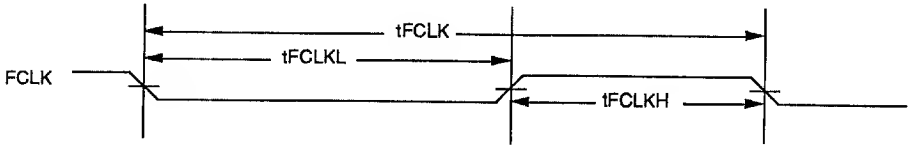


FIGURE 36. FCLK INTERFACE TIMING



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.

Notes:

VLSI TECHNOLOGY, INC.

SECTION 4

VL86C110 RISC MEMORY CONTROLLER (MEMC)

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

VL86C110

RISC MEMORY CONTROLLER (MEMC)

FEATURES

- Drives up to 32 standard dynamic RAMs giving 4 Mbytes of real memory with 1-Mbit devices
- Logical-to-physical address translation (32-Mbyte logical address space) supporting three protection levels:
  - Supervisor Mode
  - Operating System Mode
  - User Mode
- Uses fast page-mode DRAM accesses to maximize bandwidth from commodity memories
- Internal DMA address generators for video, cursor and sound data buffers
- Various ROM speeds supported (access times of 450 ns, 325 ns, 200 ns)
- Provides all critical system timing including processor clocks, –RAS, –CAS, and DMA data transfer strobes
- Arbitrates memory between the processor and DMA systems

DESCRIPTION

The memory controller (MEMC) acts as the interface between the VL86C010 processor and other functions in the system. The four circuits in the RISC family: MEMC, VL86C010, VIDC-video controller, and IOC-I/O controller, can be used to implement a small computer system. MEMC uses a single clock input to derive timing information for the other components.

In addition to providing interface signals to the other controllers, MEMC generates all the control signals for several access times of read-only memory (ROM) plus high-resolution timing and refresh control for dynamic RAM (DRAM). The controller outputs can drive up to 32 memory devices directly in a wide variety of configurations using various architectures of standard DRAMs. A logical-to-physical address translator maps the 4-Mbyte physical memory into the 32-Mbyte logical address space with three levels of protection.

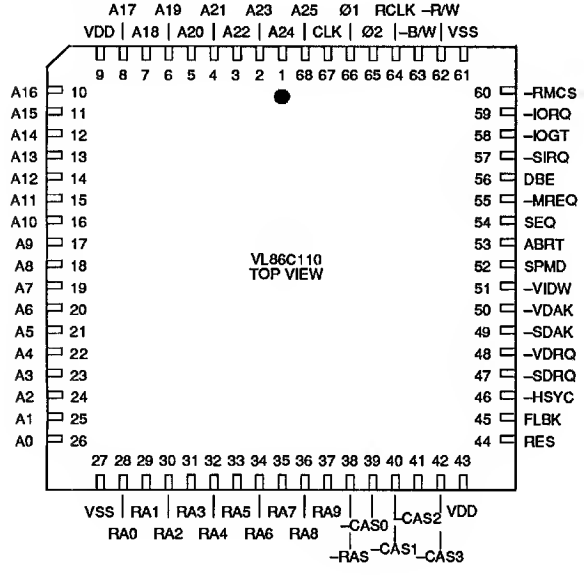
Address translation is performed by a simple 128 entry content-addressable memory (CAM). MEMC provides a descriptor entry for every page of physical memory which eliminates descriptor thrashing (address translation misses) from degrading system performance.

The simple structure allows memory address translation to be performed without increasing required memory access time or decreasing the system clock. MEMC allows virtual memory and multi-tasking operations to be implemented without the usual performance degradation associated with each function. Fast page-mode DRAM accesses are used to maximize memory bandwidth from inexpensive commodity memory devices.

MEMC supports direct memory access (DMA) read operations with three programmable address generators. Video refresh is performed using a circular buffer to enhance scrolling capability plus a separate linear buffer for a cursor sprite. Sound data uses a double buffering system.

PIN DIAGRAM

PLASTIC LEADED CHIP CARRIER (PLCC)

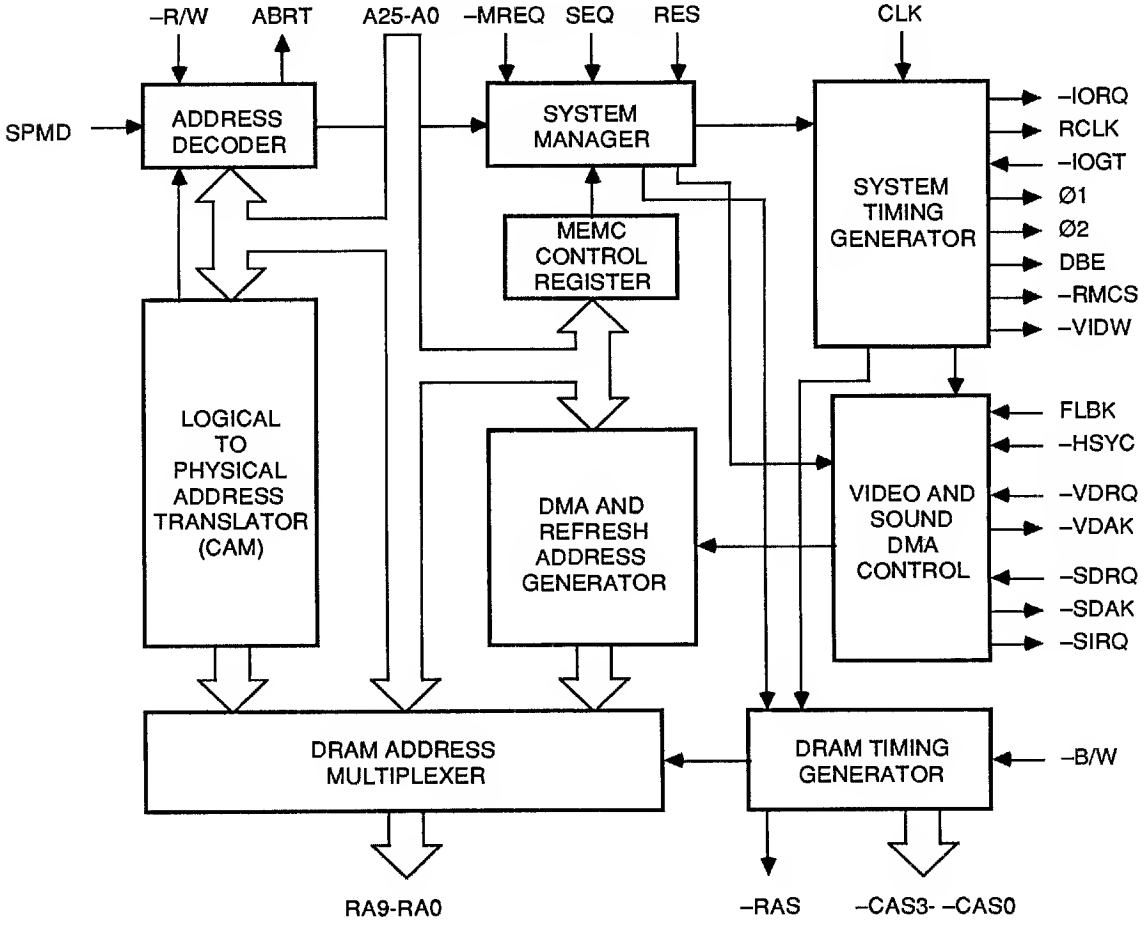


ORDER INFORMATION

Part Number	Bus Clock Frequency	Package
VL86C110-10QC	10 MHz	Plastic Leaded Chip Carrier (PLCC)
VL86C110-12QC	12 MHz	Plastic Leaded Chip Carrier (PLCC)

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

BLOCK DIAGRAM



SIGNAL DESCRIPTIONS

Signal Name	Pin Number	Signal Description
A25-A0	68,1-8, 10-26	Address 25 - Address 0 (CMOS level inputs) - These are the 26 processor address lines that contain the address of the memory reference. When Ø2 is low these signals should contain the address of the current memory reference. When Ø2 goes high, these address pins should be changed to the value for the next cycle. A1 and A0 are byte addresses and are ignored during word transfer cycles. A3 and A2 are decoded to determine sequential access boundaries.
–R/W	62	Not-Read/Write (CMOS level input) - Determines the direction of data flow during the current memory access. When asserted (low) the memory cycle will be a read operation, and if negated (high) a memory write will be performed.
–B/W	63	Not-Byte/Word (CMOS level input) - Determines the size of the data transfer of the memory access (Note 2). When asserted (low) the transfer is byte-wide (8 bits) or negated (high) word-wide (32 bits). When transferring bytes the A1 and A0 address inputs are decoded to determine which 8-bit field is to be referenced. Word transfers are always aligned on word boundaries (A1 = A0 = 0) because A1 and A0 are ignored during word operations.
–MREQ	55	Processor Memory Request (CMOS level input) - Determines whether a memory cycle will be performed during the next access time. When asserted (low) this line indicates that the processor will require either a memory (Note 1) or I/O access during the next cycle time (Note 2). If negated (high), no cycle is required because the the CPU will perform an internal cycle. This input must be valid well before the falling edge of the Ø2 clock signal. Under special circumstances, this signal may affect operation of the current memory cycle. When both –MREQ and SEQ are asserted during a processor internal cycle, MEMC begins a DRAM non-sequential cycle immediately which effectively overlaps the internal cycle with the first half of the non-sequential access time.
SEQ	54	Processor Sequential Access (CMOS level input) - Determines whether the next memory cycle will be a two clock non-sequential (N-cycle) or a one clock sequential (S-cycle) access (Note 2). The VL86C010 processor asserts this signal whenever the address for the next cycle is sequential (current eddress + 4) to the address presently on the bus. When asserted (high), MEMC performs a S-cycle (page-mode) by removing –CAS while retaining –RAS active. This keeps the row address latched in the DRAMs and loads in the new column address. In general, the page-mode access time of most DRAM devices is one-half the random access time. When negated (low), the next memory cycle will be e two clock N-cycle. MEMC removes both –RAS and –CAS at the end of the current cycle, allows the memory to properly precharge, and performs a random-access cycle. This signal must be setup well before the falling edge of the Ø2 clock signal for the same reasons as the –MREQ.
SPMD	52	Supervisor Mode Select (CMOS level Input) - When low, the processor is restricted from eccess to certain areas of the memory map and will be aborted if illegal access attempts are made. SPMD is generally connected to the –TRAN output of the VL86C010 processor. If connected to –TRAN, address remapping is inhibited for all non-user mode transfers.
Ø1, Ø2	66, 65	Processor Clocks (CMOS level outputs) - These signals drive the two phase, non-overlapping clock inputs of the VL86C010 processor. The frequency of these clocks is the master clock (CLK input) frequency divided by three. The Ø2 clock is in phase with the reference clock (RCLK output).
DBE	56	Processor Data Bus Enable (CMOS level output) - Determines when the data bus drivers inside the processor are enabled. When asserted (high) the processor is driving the data bus during e write cycle. This signal should be inverted externally to provide an active low write enable for the Dynamic RAMs to prevent three-state driver contention on the data bus.
ABRT	53	Processor Abort (CMOS level output) - Determines whether the current memory cycle will terminate abnormally. When asserted (high) MEMC has detected either an attempted access to a higher privileged area or a non-existent logical page. Both these conditions will cause an abort of the current memory cycle and exception processing to be invoked by the processor to determine error recovery procedures. When negated the current cycle will terminate normally and processing flow continue under program control.

Notes: 1. The word memory in this context refers to any device mapped into the processor's address space.
2. Some of the processor signals are asserted in the processor cycle preceding that in which they are used.

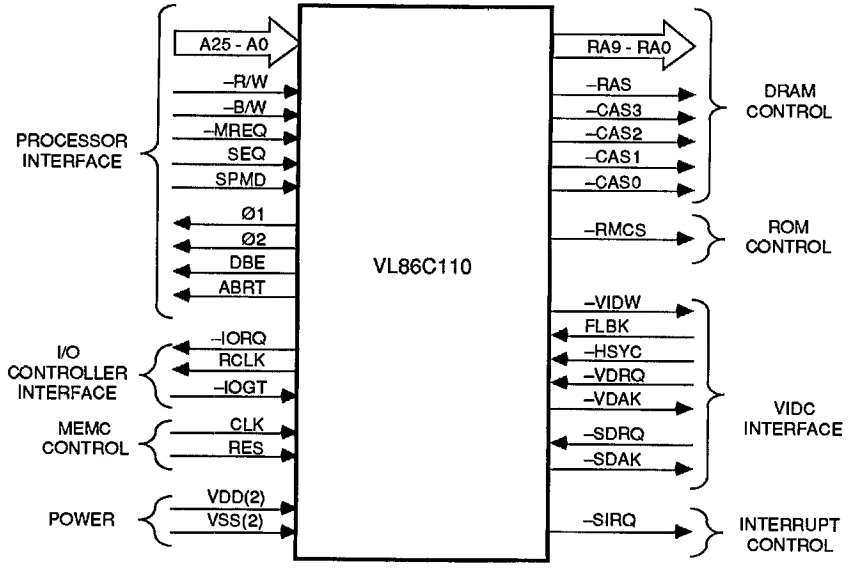
SIGNAL DESCRIPTIONS (Con't.)

Signal Name	Pin Number	Signal Description
–IORQ	59	Input/Output Cycle Request (CMOS level output) - Determines whether the current cycle is a memory or an I/O reference. When asserted (low), MEMC has detected an I/O address and the proper I/O controller should respond. When negated (high), the current cycle is a memory reference.
–IOGT	58	Input/Output Cycle Grant (CMOS level input) - Determines when the current I/O access cycle will terminate. When asserted (low), the selected I/O controller is signaling that the current I/O cycle will end on the next falling edge of the RCLK clock if the –IORQ is still low.
CLK	67	Clock (CMOS level input) - Master input clock used to derive all system timing functions. The input signal should be approximately a 50% duty cycle with full CMOS levels. This clock is divided down internally to obtain the two-phase processor clocks, system reference clock, and the DRAM refresh clock.
RCLK	64	Reference Clock (CMOS level output) - Provides the main reference clock for bus transactions between different devices. RCLK clock is in phase with the Ø2 clock of the processor.
RES	44	Reset (CMOS level input) - Places the MEMC in a known initial state. When asserted (high), MEMC is forced into the following modes: ROM is continually selected with an access time of 450 ns, DRAM page size is 4 Kbytes, operating system mode disabled, sound DMA operations disabled, –SIRQ set low, video/cursor operations unaffected, and –IORQ is held high to prevent I/O controllers from responding to spurious addresses generated during the reset state.
RA9-RA0	37-28	RAM Address Bus (TTL level outputs) - Provide the multiplexed row and column address lines to the DRAM array. Each output is capable of driving up to 32 DRAMs without external buffering. The bit order and logic level of these pins varies according to the page size selected, and are shown in detail in Appendix A.
–RAS	38	Row Address Strobe (TTL level output) - Provides the –RAS timing/control signal to the DRAM array. The falling edge of –RAS strobes the row address on RA9-RA0 pins into the DRAMs. This signal is capable of driving up to 32 DRAMs without external buffering.
–CAS3- –CAS0	42-39	Column Address Strobes (TTL level outputs) - Provide the –CAS timing/control signals to the DRAM array. Each output controls one 8-bit byte of the four byte memory word to support byte writes. The falling edge of a –CAS strobes the column address on RA9-RA0 into the DRAMs. Each signal will drive up to eight DRAMs without external buffering.
–RMCS	60	ROM Chip Select (CMOS level output) - Provides the chip select control signal to the ROM devices. When asserted (low), MEMC has detected a ROM address for the current cycle.
–VIDW	51	Video Controller Write Strobe (CMOS level output) - Provides the register select signal to the video controller (VIDC) device. When asserted (low), MEMC has detected a write request to the video controller. The data should be latched on the rising edge of this signal.
FLBK	45	Video Vertical Flyback (CMOS level input) - Provides vertical timing information from the video section. FLBK is used to time the initalization of the video/cursor DMA address pointers and for refresh control in certain modes. This signal should be taken high while the video retrace is in progress.
–HSYC	46	Video Horizontal Synchronization (CMOS level input) - Provides horizontal timing information from the video section. When asserted (low), the video is in horizontal retrace. Video data requests made while this signal is asserted will obtain data from the cursor data buffer. A video data request made when –HSYC is negated obtains data from the video data buffer.
–VDRQ	48	Video Data Request (CMOS level input) - Provides the synchronization between MEMC and VIDC for video data interface. When asserted (low), the VIDC is requesting either a video or cursor DMA operation for video refresh. Video and cursor requests are distinguished by the level of the –HSYC signal. Requests made during horizontal retrace (–HSYC low) are cursor and all others video.
–VDAK	50	Video Data Acknowledge (CMOS level output) - Provides the synchronization between MEMC and VIDC for video data interface. When asserted (low), MEMC is indicating that the requested video/ cursor data is being fetched from RAM. The data should be latched on the rising edge of –VDAK.

SIGNAL DESCRIPTIONS (Con't.)

Signal Name	Pin Number	Signal Description
–SDRQ	47	Sound Data Request (CMOS level input) - Provides the synchronization between MEMC and VIDC for sound data interface. When asserted (low), VIDC is requesting a sound DMA operation.
–SDAK	49	Sound Data Acknowledge (CMOS level output) - Provides the synchronization between MEMC and VIDC for sound data interface. When asserted (low), MEMC is indicating that the requested sound data is being fetched from RAM. The data should be latched on the rising edge of –SDAK.
–SIRQ	57	Sound Interrupt Request (CMOS level output) - Provides the synchronization between MEMC and the processor for sound data interface. When asserted (low), MEMC is requesting a sound service operation by the processor. The sound DMA address generators interact with interrupt driver software to implement the sound system. –SIRQ is set low on reset.
VSS	61, 27	Digital ground. The digital ground power supply.
VDD	43, 9	Digital power. The digital +5.0 volt power supply.

FUNCTIONAL PIN DIAGRAM



FUNCTIONAL DESCRIPTION

MEMC supports three levels of memory protection:

- Supervisor Mode - Supervisor mode is selected while the SPMD input is held high. This is the most privileged mode, allowing the entire memory map to be freely accessed.
- Operating System Mode (OS) - OS mode is selected by setting a control bit in the MEMC Control Register (which may only be done from supervisor mode). OS mode is more privileged than user mode when accessing logically mapped RAM, but acts as user mode in all other cases.
- User Mode - User mode is the least privileged of the protection modes. Access is allowed only to unprotected pages in the logically mapped RAM and read cycles to the ROM space. No other accesses are allowed.

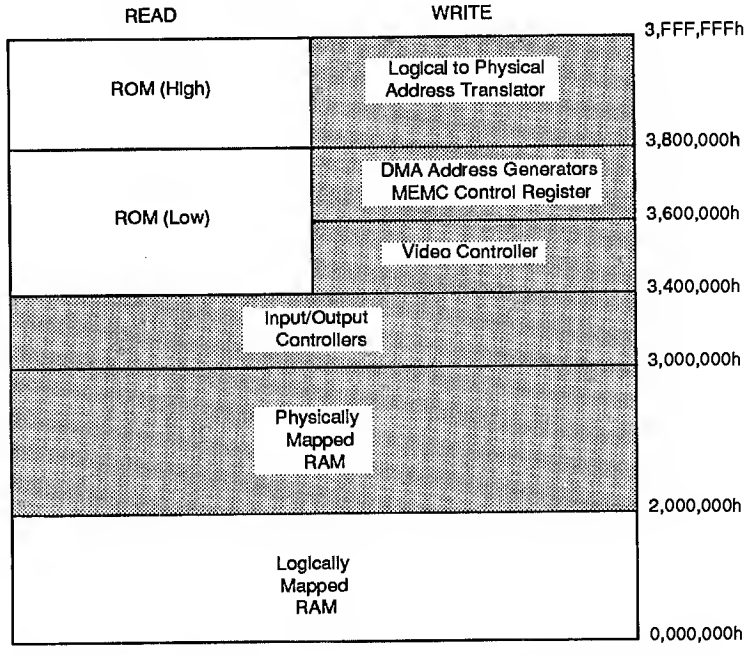
All attempts to access protected addresses from an insufficiently privileged mode (user mode or OS mode) will activate the ABRT line without performing the access.

Memory Pages

MEMC treats the DRAM as a set of 128 sequential physical pages. The page is the fundamental unit of memory used by MEMC, and the page size may be selected as 4, 8, 16, or 32 Kbytes by programming the MEMC control register. Please note the MEMC page unit should not be confused with the page-mode access capability of RAMs.

The page size selection affects the DRAM address multiplexers, so it is essential to choose the correct page size for the amount of memory being controlled. Table 1 shows the page size selection for most DRAM configurations.

FIGURE 1. PROCESSOR MEMORY MAP DECODED BY MEMC



Master/Slave Configuration

A single MEMC will control up to 4 Mbytes of DRAM. A second MEMC can be built into a system to extend the maximum addressable DRAM to 8 Mbytes. The two MEMCs are configured as a Master and a Slave, where the Slave acts purely as a DRAM driver (all DMA operations, I/O Controller interactions, etc. are handled by the Master).

The –B/W input is sampled as RES goes low, and its state determines whether the MEMC will operate in Master (–B/W = 1) or Slave (–B/W = 0) mode. In a single MEMC system, VL86C010 holds –B/W high during reset, so the MEMC is always configured as a Master.

Memory Map

MEMC accepts 26 address lines from the processor, A25 - A0, which are decoded as shown by the memory map in Figure 1. Shaded portions of the memory map are accessible only while MEMC is in the supervisor mode.

Logically Mapped RAM (Read/Write: 0000000h - 1FFFFFFh)

The bottom 32 Mbytes of the memory map consists of logically mapped RAM. MEMC treats this area of the map as a set of contiguous logical pages (there may be 8192, 4096, 2048, or 1024 logical pages depending upon the page size selected).

When a logical page is accessed, the logical-to-physical address translator attempts to convert the logical page number to a physical page number. Provided the mapping exists, and the request is being made in a sufficiently privileged mode, the appropriate physical page will be accessed. If the mapping does not exist, or the access is

TABLE 1. RECOMMENDED PAGE SIZE SETTINGS

Total Amount Of RAM	Page Size	Number Of Physical Pages	Number Of Logical Pages
0.25 Mbytes	4 Kbytes	64	8192
0.50 Mbytes	4 Kbytes	128	8192
1.00 Mbytes	8 Kbytes	128	4096
2.00 Mbytes	16 Kbytes	128	2048
4.00 Mbytes	32 Kbytes	128	1024

made with insufficient privilege, MEMC will signal the processor by setting the abort line high, and the DRAM will not be activated.

The logical-to-physical mapping and protection status of each logical page is undefined at power on, but may be programmed at any time by writing to the logical-to-physical address translator.

Physically Mapped RAM (Read/Write: 2000000h - 2FFFFFFh)
The physically mapped RAM occupies 16 Mbytes of the memory map, and may only be accessed when supervisor mode is selected. The 128 physical pages appear sequentially in this area of the map, with the RAM image being repeated after every 128th page (so that, with a page size of 8 Kbytes, the entire 1 Mbyte of RAM would occur 16 times throughout this area).

Input/Output Controllers (Read/Write: 3000000h - 33FFFFFh)
This area of the map is reserved for I/O Controllers (including IOC). When a Supervisor mode access is made in this memory range, MEMC asserts –IORQ (I/O cycle request), and stops the processor clocks. The I/O cycle terminates when both –IORQ and –IOGT are low on the rising edge of RCLK.

Please note that care must be taken not to access a non-existant I/O Controller, or MEMC will wait indefinitely for an active –IOGT signal that never appears, and the system will stop until RES is asserted.

ROM (Read: 3400000h - 3FFFFFFh)
Read Only Memory may be read freely from any protection mode. The ROM space is divided into two areas:

- Low ROM (4 Mbytes from 3400000h to 37FFFFFh)
- High ROM (8 Mbytes from 3800000h to 3FFFFFFh)

The two ROM areas are distinguished only by the fact that each may be programmed to operate at its own speed. This would allow the high ROM area to contain fast system ROMs, with slower applications ROMs in the low area.

The ROM speeds default to the slowest setting when RES is asserted, and may be altered by reprogramming the MEMC control register.

Video Controller (Write: 3400000h - 35FFFFFh)
A write operation made anywhere in the video controller space (while MEMC is in Supervisor mode) activates the –VIDW output from MEMC.

DMA Address Generators and Control Register (Write: 3600000 - 37FFFFFh)
This address space decodes to some of MEMC's internal registers. The DMA address generators supply the physical RAM address used to obtain data during video, cursor, and sound direct memory access operations. The MEMC Control Register governs a number of the functions of MEMC.

The processor data bus is not connected to MEMC; instead, the internal registers are programmed by encoding the data on the address bus, and performing a write operation with MEMC in supervisor mode. Since most writes to the MEMC registers occur at a fairly low frequency, it was felt that the small amount of overhead incurred encoding register data on the address bus did not justify adding the 32 pins necessary for the data bus interface.

Logical-to-Physical Address Translator (Write: 3800000h - 3FFFFFFh)
The mapping of logical pages to physical pages, and protection mode associated with each mapping, may be controlled by programming the logical-to-physical address translator. The translator is programmed by encoding data in the address lines, and performing write operations in supervisor mode to this area of the memory map.

Effect of Reset
When the RES line is taken high, MEMC initializes to the following state:

- Memory Map - The VL86C010 processor starts executing code from location 000000H after RES goes inactive. To ensure that the processor always finds valid code at this location (which is normally logically mapped RAM), MEMC continually enables ROM.

To restore the normal memory map, the processor must first perform a memory access with the address lines A25 and A24 both low and then perform a memory access with address line A25 high.

These conditions are satisfied when the processor starts executing instructions from location 0000000h, and later jumps to the normal ROM space.

- ROM access times - The ROM access times for both high and low ROM are reset to 450 ns.
- Page sizes - The DRAM page size defaults to 4 Kbytes on reset.
- Operating System mode - The Operating system mode is disabled on reset.
- Direct Memory Access (DMA) operations - Sound DMA operations are disabled by reset, and may be enabled by programming the MEMC Control Register. Video and cursor operations are unaffected by reset.
- Sound Interrupts - The sound interrupt pin, –SIRQ is set low on reset. The interrupt may be removed by initializing the sound DMA buffers in the DMA Address Generators.
- The processor may generate spurious addresses while RES is active high. To avoid accidentally triggering an I/O controller, the –IORQ signal is held high during reset.
- The Test mode (used in functional testing) is disabled by RES. Test mode may be set by programming the MEMC control register, but will crash the system; control is regained by resetting MEMC.

Access Times
A number of devices appear in the processor memory map:

- Dynamic Random Access Memory (DRAM)
- Read Only Memory (ROM)
- Input/Output Controllers
- Video Controller
- MEMC internal registers
  Control Register
  DMA address generators
  Logical to physical address translator

These devices have very different access times, ranging from 500 ns for a slow ROM to 125 ns for DRAMs in page-mode. MEMC provides the processor clocks, Ø1 and Ø2, which are stretched to synchronize the processor with the device it is accessing.

The processor is the default user of the memory and data bus. However, DMA (Direct Memory Access) and refresh operations require control of the DRAM and data bus, so MEMC disables the processor temporarily by placing the processor data bus in the high-impedance state (using DBE), and stretching the processor clocks.

N-cycles and S-cycles
MEMC uses the page-mode access capability of DRAMs, where, once a row address has been strobed into the DRAM, any column in that row may be accessed merely by strobing in the new column address.

This facility is used whenever a number of sequential addresses in the DRAM are to be accessed (either by the processor or during a DMA operation). The first memory cycle in the sequence is a non-sequential (N-cycle) memory cycle (where both the row and column addresses are strobed to the DRAMs). The subsequent memory accesses are sequential (S-cycle) memory cycles (where the previous row address is held, and only the column address is strobed to the DRAMs).

Processor (VL86C010) Interface
Processor cycles - There are two basic types of processor operations:

- Memory access cycles - Where the processor accesses a device in its address space.
- Internal cycles - Where the processor performs an internal operation without access to any external device.

Processor Signals
- Address Bus (A25-A0) - The processor address bus is decoded by MEMC to give the processor access to the various devices.

Much of the processor memory map is only accessible while MEMC is in supervisor mode (SPMD line high).

- Memory Request (–MREQ) - This signal determines whether the next processor cycle will be a memory access or internal cycle.
- Not-Read/Write (–R/W) - Determines the direction of data flow during processor memory access cycles. This signal is ignored during processor internal cycles.
- Not-Byte/Word (–B/W) - Selects a byte (8-bit) or word (32-bit) data transfer during processor memory access cycles. A byte access to physically and logically mapped RAM only enables the appropriate 8-bit block of DRAMs. ROM accesses return a word quantity, regardless of the state of –B/W. This signal is ignored for processor internal cycles.
- Sequential Access (SEQ) - An active high on this line indicates that the processor will generate a sequential address during its next cycle. MEMC uses SEQ to determine whether a fast S-cycle may be used during the next DRAM.
- Supervisor Mode (SPMD) - An active high on this line puts MEMC into supervisor mode, allowing the processor to access restricted areas of the memory map.

Processor Controls
- Processor Clocks (Ø1, Ø2) - MEMC provides the processor with two non-overlapping clocks, Ø1 and Ø2.

Memory access cycles are nominally 250 ns long, (with Ø1 high for about 175 ns, and Ø2 high for around 55 ns), but the following exceptions apply:

- Sequential DRAM accesses - DRAM S-cycles are active (Ø1 high - 55 ns).
- ROM accesses - ROM access times vary from 450 ns to 200 ns. Ø1 is held high long enough to meet the ROM access time requirement.
- I/O Cycles - I/O cycles take a variable length of time to complete. During the longer I/O cycles, the processor is suspended until the I/O controller is ready to complete the cycle by holding Ø2 high. Suspending the processor during its Ø2 phase allows the I/O cycle to be completed much faster when the I/O controller signals the end of the cycle.
- DMA and Refresh operations - DMA and refresh operations have priority over non-sequential processor memory access cycles. A processor non-sequential access is delayed during DMA and refresh operations by disabling the processor data bus drivers (using DBE), and holding the Ø1 clock high until the operation has finished. The processor then continues with its delayed memory access (unless another DMA/refresh operation is pending).

DMA and refresh operations may also occur during long I/O cycles. In this case, the I/O cycle is delayed until the DMA/refresh operation completes.

Internal cycles are always 125 ns long, with both Ø1 and Ø2 high for approximately 55 ns.

- Data Bus Enable (DBE) - Enables the processor data bus during processor write cycles. This signal may also be

TABLE 2. DRAM ADDRESS BUS CONFIGURATIONS

Total Amount Of RAM	Page Size Setting	Typical Configuration	Row /Column Address Connection	Bank Select
0.25 Mbytes	4 Kbytes	8 pcs. 64K x 4 DRAM 32 pcs. 64K x 1 DRAM	RA7-RA0	None
0.5 Mbytes	4 Kbytes	16 pcs. 64K x 4 DRAM	RA7-RA0	RA8
1 Mbytes	8 Kbytes	8 pcs. 256K x 4 DRAM 32 pcs. 256K x 1 DRAM	RA8-RA0	None
2 Mbytes	16 Kbytes	16 pcs. 256K x 4 DRAM	RA8-RA0	RA9
4 Mbytes	32 Kbytes	8 pcs. 1M x 4 DRAM 32 pcs. 1M x 1 DRAM	RA9-RA0	None

inverted externally, and used es a DRAM Write Enable signal.

– Memory Access Abort (ABRT)-Warns the processor that the requested access is illegal (either because an ettempt was made to access a protected address while MEMC was in an insufficiently privileged mode, or en access to a non-existent logical page was attempted).

– **Dynamic RAM Memory (DRAM) Interface**
MEMC interfaces directly to most standard Dynamic RAMs, providing a 10-bit multiplexed RAM address bus, RA9-RA0, a row address strobe, –RAS, and a set of four column address strobes, –CAS3- –CAS0.

DRAM Configurations - The page size setting (in the MEMC Control Register) controls how the DRAM address is presented on the RAM address bus, RA9-RA0, as shown in Table 2.

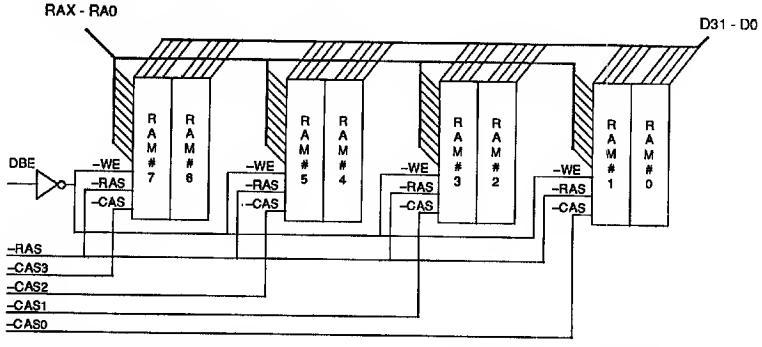
There are three basic dynemic RAM configurations supported by MEMC.

– Thirty-two 64Kx1, 256Kx1, or 1Mx1 DRAMs as shown In Figure 2.

This configuration splits the thirty two 1-bit DRAMs into four blocks of eight bits. Each 8-bit block is controlled by one of the –CAS3 - –CAS0 lines, allowing independent access to any of the byte-wide blocks. The DRAM Write-Enable line is derived by inverting the DBE signal from MEMC. The RAM Address bus, RA9-RA0, and –RAS strobe are routed to all the DRAMs.

– Eight 64Kx4, 256Kx4, or 1Mx4 DRAMs as shown in Figure 3.

FIGURE 3. DRAM CONFIGURATION WITH 8 FOUR-BIT DEVICES



This configuration Is essentially the same as used for the 1-bit wide DRAMs. In this case, only eight chips are required for the full 32-bit data bus.

– Sixteen 64Kx4, or 256Kx4 DRAMs as shown in Figure 4.

The sixteen chips are configured as two parallel banks of eight 4-bit wide DRAMs. One RAM address bit is used as a bank select line (valid at the same time as the column addresses).

Note that a two-of-four decoder is used to derive en Output-Enable and Write-Enable signal for each bank. This insures that only the one bank is activated during any DRAM access.

When only 0.25 Mbyte is used (eight 64Kx4 or 32 64Kx1 DRAMs), the bank select address bit, RA8 is ignored by the DRAMs, so physical pages 64-127 map onto physical pages 0-63.

Byte And Word Accesses
The DRAM is divided into four 8-bit blocks each with a –CAS signal.

The processor byte/word select line, –B/W, selects whether a word (32 bits) or a single byte (8 bits) is to be read from or written to DRAM. During word operations, all four blocks are activated, allowing the full 32 bits to be accessed. During a byte access, the two least significant address bits, A1-A0, select one eight-bit block to be activated, so that only the appropriate 8 bits ere read from or written to the DRAMs.

DRAM Cycles
There are three main types of DRAM eccesses as follows:

– Processor access (fetching instructions and reading/writing data)

– DMA operation (fetching video, cursor or sound data)

– Refresh operation

MEMC uses the page-mode capability of DRAMs, performing sequential accesses (S-cycles) where possible.

FIGURE 2. DRAM CONFIGURATION WITH 32 ONE-BIT DEVICES

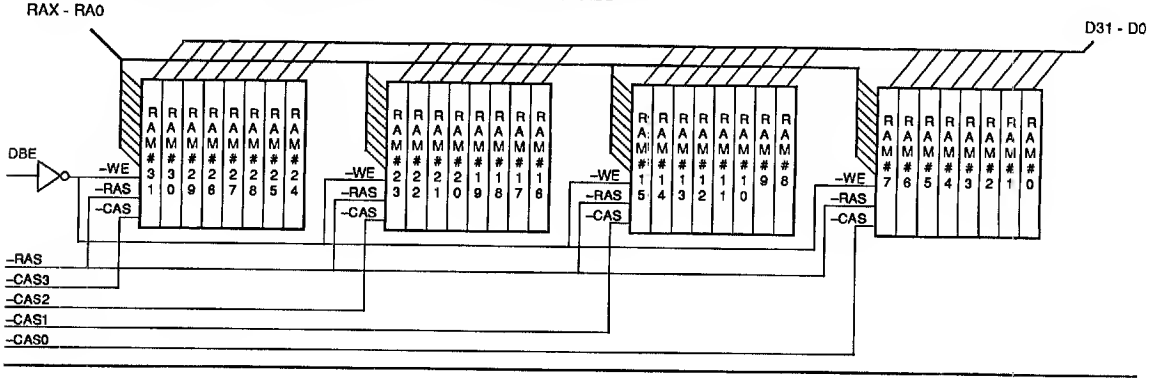
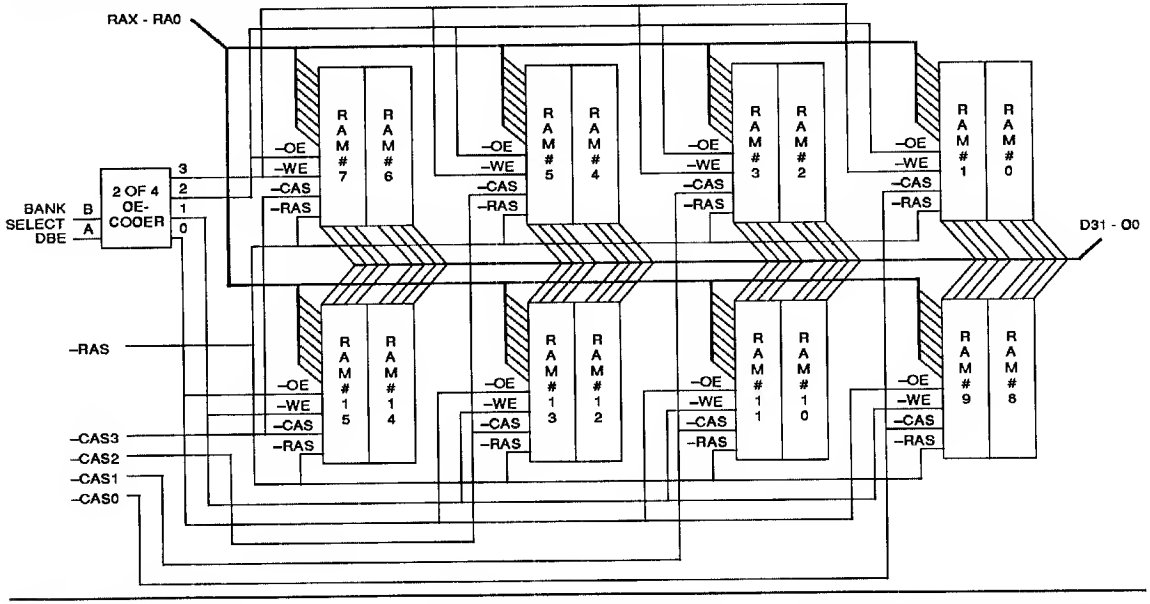


FIGURE 4. DRAM CONFIGURATION WITH 16 FOUR-BIT DEVICES



Processor Accesses

MEMC monitors a processor signal, SEQ, that indicates that the next processor access will be sequential. Please note that MEMC uses the SEQ to detect sequential access, and does not perform its own check. If the SEQ signal is asserted incorrectly, the wrong memory page may be accessed.

If the SEQ signal is low in the processor cycle preceding a DRAM access (SEQ is a pipelined signal), an N-cycle DRAM access will be used. Subsequent DRAM accesses will use S-cycles as indicated by the SEQ signal.

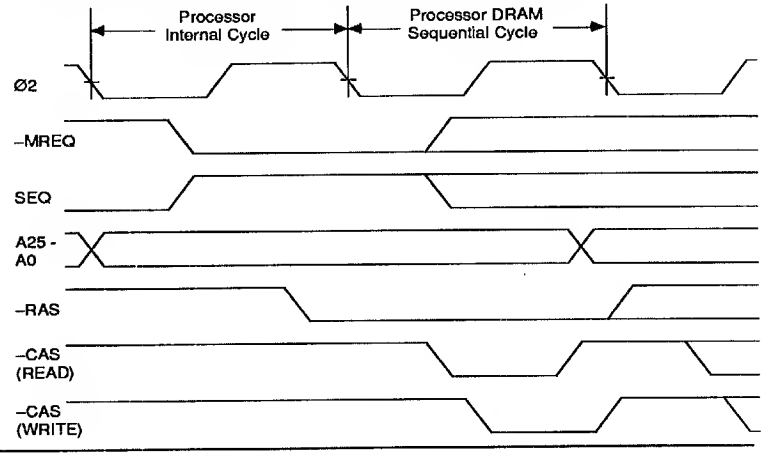
DMA operations may not interrupt the processor if it is about to perform an S-cycle memory access, so the maximum number of consecutive S-cycles is restricted to three. This reduces the worst case DMA latency to an acceptable figure, while retaining the improvement that S-cycles bring.

An N-cycle is forced under either of the following conditions:

- The processor SEQ signal was low in the preceding cycle, indicating that this access would not be to a sequential address.
- The processor address lines A3 and A2 will both be low for the access. This restricts the maximum number of consecutive S-cycles to three.

MEMC optimizes non-sequential DRAM accesses following internal processor cycles as shown in Figure 5. During internal cycles, the processor outputs an address and sets SEQ high, indicating that the address will be valid in the next cycle. In an internal cycle preceding a processor memory access, the -MREQ line will be set low. When MEMC sees SEQ high, -MREQ low, and A25-A0 addressing DRAM during an internal cycle, it starts a DRAM N-cycle immediately. As the row address has already been strobed into the DRAMs, the processor DRAM access can then complete with a DRAM S-cycle. This special operation does not occur if the DRAM address has both A3 and A2 set high (this is a consequence of the multiple S-cycle limiting logic).

FIGURE 5. DRAM ACCESSES FOLLOWING INTERNAL CYCLES



DMA Operations
DMA operations always fetch four words (16 bytes) of data sequentially from the DRAMs. Thus, DMA operations are composed of an N-cycle read followed by three S-cycle reads.

Refresh Operations
A refresh operation is effectively a single N-cycle DRAM read operation, with the exception that the –CASx lines are not strobed low.

DRAM Timing
The –CAS3 - –CAS0 strobes are generated early in read operations, and late in write operations to improve setup and hold times on the data bus. If an ABRT is generated during an N-cycle, the –RAS strobe will be activated as usual, but the –CAS3 - –CAS0 signals are suppressed, effectively disabling the DRAM cycle.

MEMC does not supply a DRAM Write Enable signal directly. A suitable signal may be derived by inverting the DBE output from MEMC.

Read Only Memory (ROM) Interface
In order to minimize the ROM access time, the ROM chip select signal from MEMC, –RMCS, is enabled at the start of every processor burst, and only disabled when the processor address lines have been decoded as addressing another part of the memory map.

External Address Latches
The ROM low order address lines must be latched externally on the rising edge of Ø2 to hold the address stable to the end of each processor cycle. The Ø2 signal must not be loaded too heavily, otherwise Ø1 and Ø2 may overlap, so Ø2 should be buffered with an external inverter to provide a suitable address latching signal.

ROM Speeds
The ROM area of the processor memory map is divided into two sections, high ROM and low ROM. The ROM access time in each area may be independently programmed through the MEMC Control Register. Three ROM access times are available: 450 ns, 325 ns, and 200 ns. When a ROM cycle is performed, the processor clocks are stretched to provide the necessary ROM access time.

MEMC Control Register
The MEMC Control Register is a programmable location that controls the functions of MEMC. The part does not monitor the processor data bus, so the parameters are encoded into the address lines, as shown in Figure 6. The Control Register is programmed by performing a write operation while MEMC is in supervisor mode.

Logical And Physical Page Size
The logical and physical page size must be set to correspond to the type of DRAM connected to MEMC. Page sizes of 4 Kbytes, 8 Kbytes, 16 Kbytes, or 32 Kbytes may be selected. A default page size of 4 Kbytes is selected when RES is asserted.

ROM Access Times
ROM access times of 450 ns, 325 ns, or 200 ns may be selected for each of the two ROM areas (high ROM and low ROM). The ROM access time for both high and low ROM areas is forced to 450 ns when RES is asserted.

Refresh Operations
Video DMA operations address DRAM locations sequentially at regular intervals, effectively refreshing DRAM, but video DMA operations are normally suspended during flyback.

For high resolution displays, the flyback time is shorter than the DRAM hold time, and no data is lost during flyback. Broadcast standard displays have longer flyback times, and extra DRAM refresh must be provided during flyback to retain DRAM integrity.

When no video DMA cycles are requested, all refresh operations must be generated by MEMC. To cover all memory requirements, three refresh modes are available.

Continous Refresh
A refresh operation is performed every 4 µs. The refresh operation uses the DMA video pointer as the refresh address source, incrementing the pointer after use. As this effectively scrambles the video DMA pointer, this mode should never be selected when any video display is being generated.

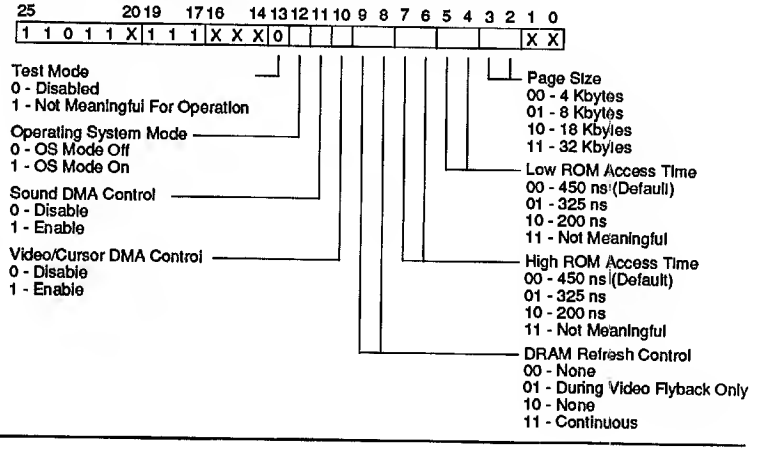
Refresh Only During Video Flyback
A refresh operation is performed every 4 µs while FLBK is active. This mode is selected when a broadcast standard video display is being generated.

No Refresh
This mode of operation disables refresh entirely, relying on video DMA operations to refresh the DRAM. The flyback time of the display (when no video DMA operations are requested) should not exceed the worst case DRAM storage time.

Refresh operations take a single N-cycle. Processor clocks are halted during the operation and the refresh address is strobed into the DRAMs using the –RAS line.

The refresh address is provided using the DMA address generator's video pointer, which is incremented after every refresh operation. Refresh operations have a lower priority than DMA operations and will be delayed if a DMA cycle is in progress when the refresh is attempted.

FIGURE 6. MEMC CONTROL REGISTER DEFINITION



There is no default setting for refresh operations, so the system software must turn on some form of refresh before using the DRAM. The reset condition causes the page size to default to 4 Kbytes. This will alter the RA9-RA0 configuration and break up the display unless 4K pages were being used already. Neither the video enable nor the refresh mode is affected by a system reset.

Direct Memory Access (DMA) Control
The video/cursor and sound DMA operations may be enabled or disabled as required. Assertion of the RES signal disables the sound DMA but does not affect video/cursor operations.

Operating System Mode
When Operating System mode is enabled, the processor may access certain protected logical pages in the logically mapped RAM space. As with all MEMC Control Register parameters, Operating System mode may only be changed while MEMC is in supervisor mode. Operating System mode is disabled when RES is asserted.

Test Mode
Test mode reconfigures MEMC to a known state for functional testing. Test mode must never be selected during normal operation, as it removes all sources of DRAM refresh, and halts the processor. Test mode is disabled when RES is asserted.

Logical-To-Physical Address Translator
The physical RAM is divided into 128 physical pages, which the processor may either access directly through the physically mapped RAM area of the memory map, or indirectly through the logically mapped RAM area (composed of logical pages). The logical-to-physical address translator controls the mapping of logical pages to physical pages, and allows a level of protection to be attached to each logical page.

Page Protection Levels
The logical page protection levels available are shown in Table 3. The protection level is specified by two bits, but two of the four patterns are identical, so only three protection levels are available.

- The lowest protection level (PPL1 = 0, PPL0 = 0) allows the logical page to be freely accessed when MEMC is in any protection mode.
- The medium protection level (PPL1 = 0, PPL0 = 1) allows the logical page to be freely accessed from the Supervisor or OS mode, but prevents write operations from user mode.
- The highest protection level (PP1 = 1, PPL0 = X) allows the logical page to be accessed when MEMC is in supervisor mode, prevents write operations from OS mode, and disallows any user mode accesses.

If the protection mode of MEMC is insufficently privileged to access a protected page, or the logical page being accessed has no physical page mapping, the ABRT line will be taken high to inform the processor that the memory operation was aborted, and the –CAS3 - –CAS0 lines will be held high to ensure the DRAM is not activated.

Address Translator Mapping
The address translator consists of a 128 entry lookup table. Each entry corresponds to a physical page number. A logical-to-physical mapping is made by storing a logical page number in the appropriate entry. Each entry also contains the two-bit page protection level.

When the processor accesses logically mapped RAM, the logical page number is applied to all 128 table entries simultaneously. If one of the entries contains the required logical page number, and the current operating mode of MEMC is sufficiently privileged to overcome the page protection level, the appropriate physical page (the number of the entry that matched) is output to the DRAMs.

If none of the entries matches the requested logical page or a match is found, but the page protection level is too high, the ABRT line is set high, and DRAM access does not complete.

Note that it is possible to store the same logical page number in more than one entry. However, when that logical page is accessed, many entries will claim to match, and an invalid physical page number will result.

Dual MEMC Systems
In a dual MEMC system, the physical RAM is effectively doubled to 256 physical pages, and the Logical to Physical Address Translators in both the Master and Slave MEMCs must be programmed. When programming the Address Translators, A(7) specifies whether the Master or Slave Address translator is being accessed.

Programming The Address Translator
The address translator is programmed by specifying the physical and logical page numbers that are to be associated and the required protection level. As MEMC does not monitor the processor data bus, the information is encoded into the address lines, and conveyed to the address translator by performing a write operation to the calculated address (with MEMC in Supervisor mode). Note that the page size not only affects the number of logical pages available, but also changes the bit order in which the logical and physical page numbers are specified. Diagrams showing how information is encoded into an address for each of the four possible page sizes are shown in Figure 7.

The following points should be noted:
- The address translator is undefined on power up.
- The address translator mappings are not affected by reset, but are effectively scrambled if the page size is changed.
- Only one physical page should be mapped to any given logical page.

TABLE 3. LOGICAL PAGE PROTECTION LEVELS

MEMC Protection Mode	Page Protection Level (PPL1,PPL0)			
	00	01	10	11
Supervisor	Read/Write	Read/Write	Read/Write	Read/Write
Operating System	Read/Write	Read/Write	Read	Read
User	Read/Write	Read	No Access	No Access

FIGURE 7. PROGRAMMING THE LOGICAL-TO-PHYSICAL ADDRESS TRANSLATOR

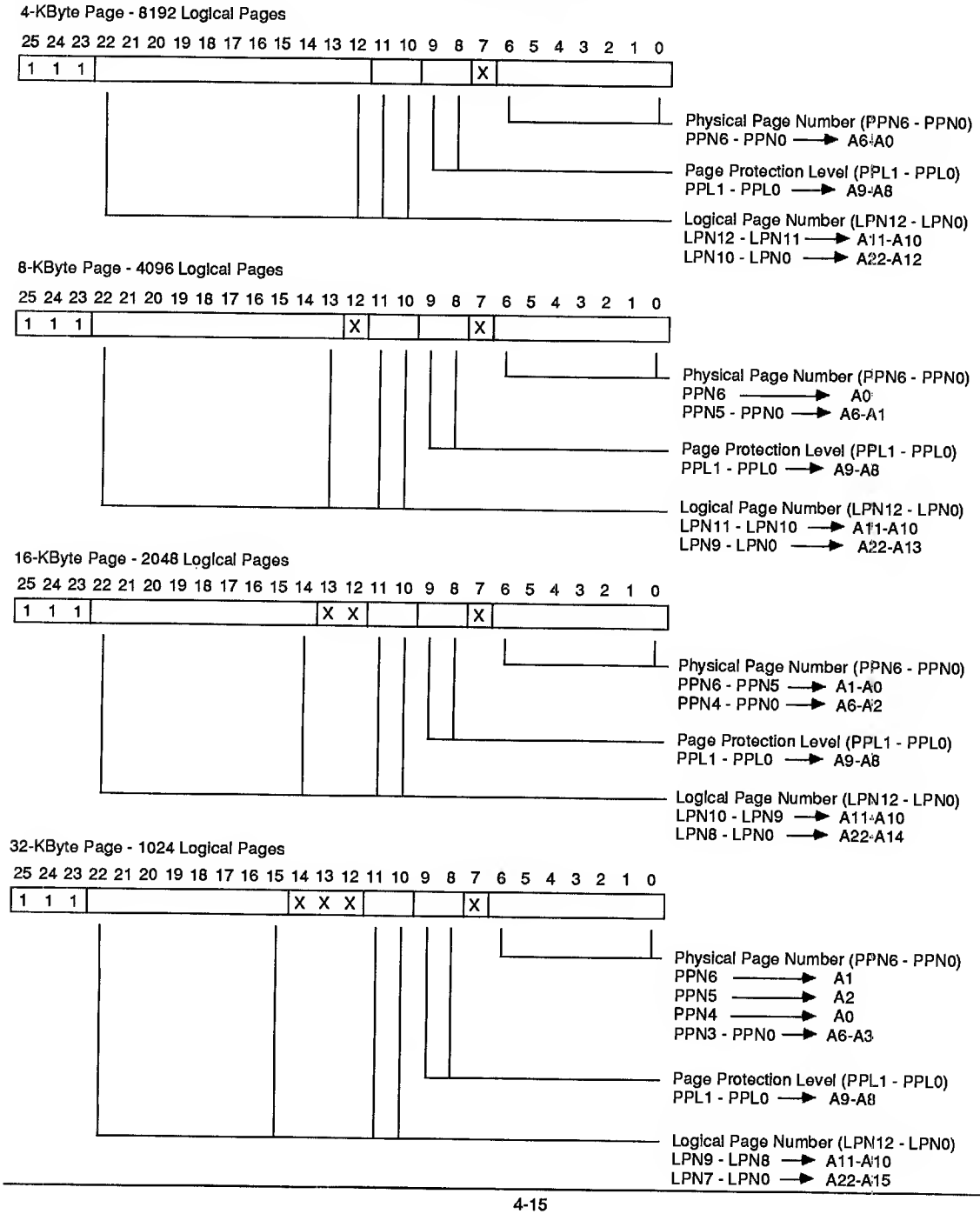


FIGURE 8. ADDRESS GENERATOR REGISTER FORMATS

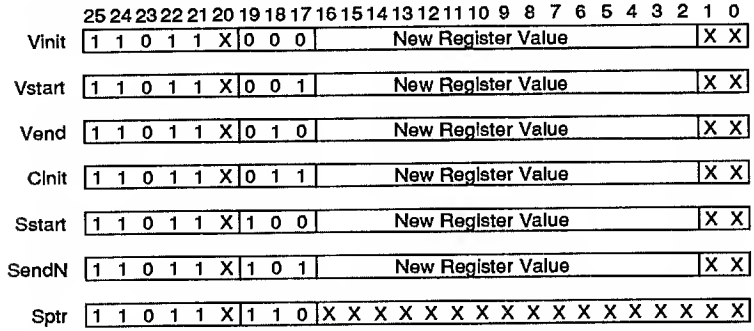
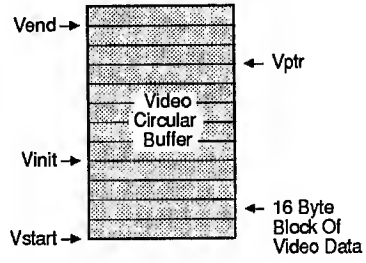


FIGURE 9. CIRCULAR VIDEO BUFFER



DMA Address Generators

The DMA address generators automatically provide addresses during DMA service. These DMA addresses are used to obtain 16 bytes of data from the DRAM (all DMA data must be quad-word aligned). The data is obtained using four DRAM accesses; each access supplies one word (4 bytes) of data which may be latched from the data bus when the appropriate DMA acknowledge line is strobed.

The DMA address generators implement three buffers: video, cursor, and sound. These buffers are defined by registers in the DMA address generators that are programmed by encoding data on the address bus and performing a write operation to the MEMC while in the Supervisor mode. Figure 8 shows how the address is calculated for storage into the DMA address registers.

Notes on programming the DMA address registers:

– The register value is calculated by dividing the physical address by 16.

– The following side effects occur when the sound buffer address generators are programmed: (1) programming the Sstart register sets the next buffer valid flag, and (2) when the Sptr register is programmed the value of the Sstart register is copied into the Sptr register, and the Next Buffer Valid flag is reset.

Video Buffer

This is a circular buffer, as shown in Figure 9. The buffer is a section of memory delimited by Vstart and Vend that contains the video data for a frame. When a video DMA is requested, the address held in Vptr (the video pointer) is used to obtain four consecutive 32-bit words of data (16 bytes) from the DRAM. The –VDAK line is pulsed low as each word of memory data is read.

The Vptr is then incremented ready to point ot the next four words of video data, unless it has reached the end of the buffer (as delimited by Vend), in which case Vptr is reset to the start of the buffer (as defined by Vstart).

The Vinit register contains the address to which Vptr will be initialized just before the new display frame begins (denoted by a high-to-low transition on FLBK), and is thus the address of the first byte of video data for the new frame. Hardware scrolling is effected by reprogramming Vinit.

The processor may program the Vstart, Vinit, and Vend registers when MEMC is in supervisor mode. The Vptr register cannot be altered directly by the processor, but is always reset to the value contained in Vinit before a new video frame is displayed.

The video pointer register, Vptr, doubles as a refresh counter. When a refresh is performed the Vptr address is output and Vptr is incremented (no check is made that Vptr has reached the end of the buffer). Refresh operations alter the contents of Vptr, continous refresh must never be enabled during video DMA operation. The special refresh mode uses Vptr only during flyback, when video DMAs do not occur.

Cursor Buffer

This is a linear buffer, and is shown in Figure 10. The cursor data is contained

FIGURE 10. LINEAR CURSOR BUFFER

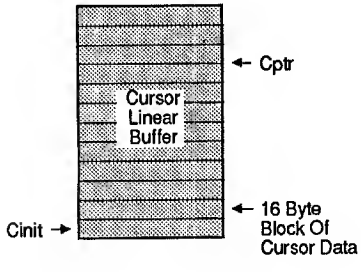
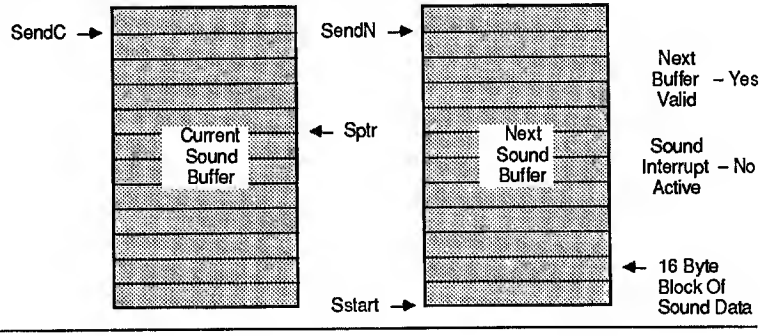


FIGURE 11. DOUBLE BUFFERED SOUND SUPPORT



in a section of memory whose initial (low) address is stored in the Cinit register. While FLBK is high denoting video flyback, Cptr (the cursor pointer) is initialized to address held in Cinit. When a cursor DMA is serviced, the address held in Cptr is sent to the DRAM, and used to obtain four 32-bit words of data (16 bytes). The Cptr is then incremented ready to point to the next four words of cursor data.

Sound Buffer

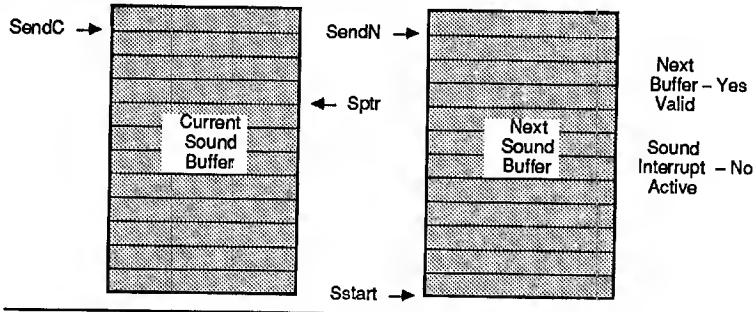
Sound data is divided into areas of memory called sound buffers. The sound system can support any number of sound buffers using a combination of the DMA address generators and interrupt driven software. Sound data is extracted from one buffer at a time, and when each buffer is exhausted, a new sound buffer is used.

The DMA address generators contain information on two sound buffers as follows and is shown in Figure 11.

– Current Sound Buffer
  This is the buffer from which sound data is extracted when a sound DMA is requested. The address of the next 16 bytes of sound data to be supplied in response to a sound DMA request is held in the sound pointer register, Sptr, and the end of the current sound buffer is delimited by the current sound end register, SendC.

– Next Sound Buffer
  This is the buffer of sound data to which the sound pointer, Sptr, will jump when it reaches the end of the current sound buffer. The act of Sptr swapping to the next sound buffer triggers a processor interrupt request (–SIRQ), which should prompt the processor to define the area of memory for the next sound buffer.

FIGURE 14. SOUND BUFFER STATE AFTER NEW START VALUE



The next sound buffer is defined by programming the Sstart and SendN registers. (Note that the processor can only program the start and end address of the next sound buffer). A hardware flag, Next Buffer Valid, is set when the next sound buffer registers have been programmed.

Sound Buffer Operation

– Sound Buffer Swap
  When the sound pointer reaches the end of the current buffer, it swaps to the start of the next buffer (provided the next buffer valid flag is set, indicating the next buffer parameters have been set up). This operation resets the next buffer valid flag, and generates a processor interrupt by taking the –SIRQ (sound interrupt) line low. The SendC and SendN registers swap over, so that the value previously set up in SendN defines the end of the new current buffer. The state after this phase is shown in Figure 12.

– Next Sound Buffer Setup
  The processor should react to the sound interrupt by defining the next sound buffer that the sound system should use. The first part of this process is to define the end address

FIGURE 12 . SOUND BUFFER AFTER SWAP OPERATION

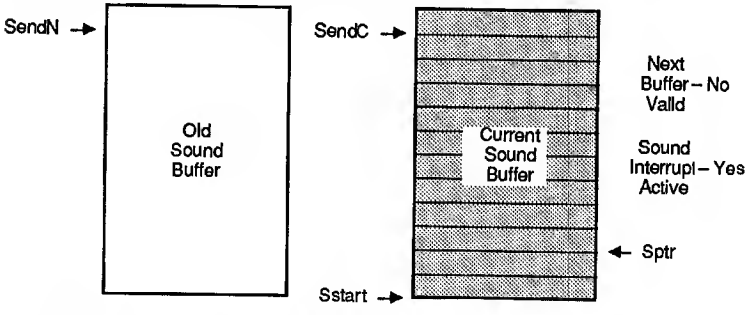


FIGURE 13. SOUND BUFFER STATE AFTER NEXT END VALID

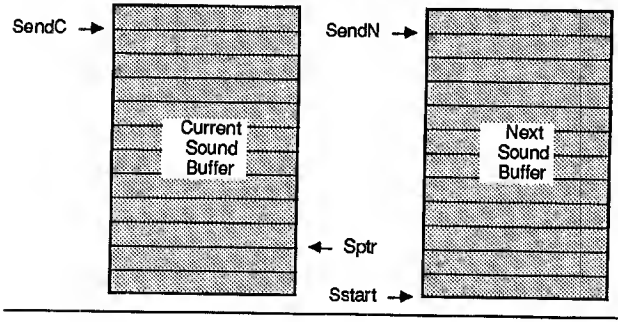


FIGURE 15. INITIAL STATE OF SOUND BUFFERS

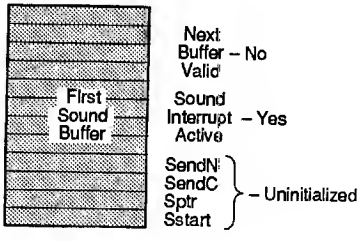


FIGURE 16. SOUND BUFFER STATE AFTER SEND AND SSTART VALID

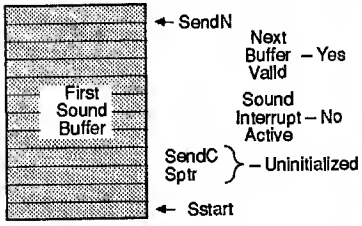
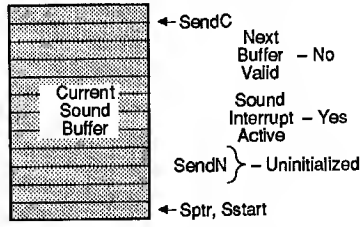


FIGURE 17. SOUND BUFFER STATE WRITE TO SPTR REGISTER



of the next buffer by reprogramming SendN. The state after this action is shown in Figure 13. The processor may now define the start address of the next buffer by reprogramming Sstart. This fully defines the next buffer, setting the next buffer valid flag and clearing the –SIRQ sound interrupt line. This action brings the buffer control to the state shown in Figure 14. The sequence now repeats from the buffer swap event.

If the processor fails to setup a new Sstart value before the sound pointer reaches the end of the current buffer, the sound pointer will swap back to the start of the current buffer (as defined by the old value of Sstart).

Initializing the Sound Buffers

The following procedure is recommended to initialize the sound buffer system (on power up, for example):

– Initial State
After power reset, the sound buffer registers are not defined, and sound DMA operations are disabled as shown in Figure 15. To start the sound system, the processor must first fill the first sound buffer with data. Note that the sound interrupt line is pulled low when RES is asserted.

– Defining the First Buffer
The SendN and Sstart registers are then programmed with the end and start addresses of the first sound buffer. This sets –SIRQ high (inactive) with the state shown in Figure 16.

– Initializing the Sound Pointer
The sound pointer is now defined by performing a write operation (with MEMC in supervisor mode) to the Sptr register. Rather than defining an immediate value to be stored in Sptr, this operation forces a buffer swap operation, copying the contents of Start to Sptr, swapping over SendN and SendC, and setting the –SIRQ low as shown in Figure 17.

The processor may now enable sound DMA operations by reprogramming the MEMC Control Register, and handle the sound interrupt in the usual way to set up the next buffer.

Proceasor/DMA Memory Arbitration

DMA operations read four words from the DRAM. The memory accesses are organized as an N-cycle followed by three S-cycles. As the DMA operation uses the system data bus, the processor must be prevented from performing accesses until the DMA has finished. The processor will not be stopped if it is about to perform a sequential memory access; instead, the DMA operation is postponed until the processor requests an internal cycle or a non-sequential memory access. Excessive DMA hold times are avoided by limiting the maximum number of consecutive processor S-cycles to three.

Processor internal cycles may occur concurrently with DMA operations, but the processor clocks will be halted (Ø1 held high) if it attempts a memory access cycle during DMA. The processor data bus is always disabled during DMA operations by taking DBE low.

The DMA request may arrive while the processor is suspended awaiting the completion of an I/O cycle. In this case, the –IORQ signal is removed, and the processor bus disabled when RCLK next goes low. Once the DMA has completed, the I/O cycle is resumed by setting –IORQ low, and enabling the processor data bus drivers.

DMA Handshaking

Video and cursor DMA operations are assumed to be mutually exclusive and are both requested by taking the –VDRQ line low. The –HYSC line determines whether a video or cursor operation is to be performed. If –HSYC is low when –VDRQ is sent low, a cursor DMA is performed, otherwise a video DMA is executed. A sound DMA operation is requested by taking the –SDRQ line low.

When the DMA operation is performed, the DRAM is read, and an acknowledge line, (either –VDAK for video/cursor DMA, or –SDAK for sound DMA) is strobed low as each word of data is available on the data bus. The rising edge of –VDAK or –SDAK may be used to latch the DMA data from the bus. Some DRAMs disable their data bus drivers before the DMA acknowledge line goes high. In this case, the dynamic storage time of the data bus is sufficient to hold the data valid until it is latched.

The appropriate DMA request (–VDRQ or –SDRQ) should be taken high when the first DMA acknowledge is given unless a consecutive DMA is desired.

The FLBK signal prompts MEMC to initialize the video and cursor buffer pointers. The cursor pointer is initialized during flyback. The video pointer is initialized just after FLBK goes low (inactive) because it (Vptr) is sometimes used as refresh pointer during flyback.

The –VDRQ, –SDRQ, –HSYC, and –SDRQ signals may be asynchronous, so they are all passed through two synchronization latches in MEMC to avoid synchronization errors.

DMA Latencies

Video/cursor DMA requests are higher priority than sound requests, and will be serviced first. All latency calculations shown assume a 24 MHz clock input. The maximum DMA latency from the time a –VDRQ or –SDRQ line is taken low to the first 32 bits of DMA data being read from DRAM is as follows:

– Video/cursor DMA latency
–VDRQ passes through two synchronization latches. The delay through these latches varies from 10 ns to 125 ns, depending on the relative phase of –VDRQ and the internal synchronizing clock. It then requires 187 ns to process the video request,

and prepare to execute a DMA cycle. A further delay of 500 ns is incurred if the processor has just started a worst case uninterruptable DRAM access (N-cycle + three S-cycles without internal cycles) in the preceding 8 MHz clock cycle. Finally, it takes 250 ns for the DRAM N-cycle read operation to supply the first word of video/cursor data. Thus, the minimum and maximum delay from –VDRQ going low to the first word of data available from the DRAMs is:

Minimum video/cursor DMA latency = 10 + 187 + 250 ≈ 450 ns

Maximum video/cursor DMA latency = 125 + 187 + 500 + 250 ≈ 1070 ns

- Sound DMA Latency. The sound DMA latency is similar to the video/cursor DMA latency. However, sound DMA operations have a lower priority than video/cursor DMA operations, and are delayed for 625 ns for every consecutive video/cursor DMA operation that is requested at the same time as, or after –SDRQ goes active low. Thus, the minimum and maximum delays from –SDRQ going low to the first word of data being available from the DRAMs is:

  Minimum sound DMA latency = 10 + 187 + 250 ≈ 450 ns

  Maximum sound DMA latency = 125+187+500+250+(625*DMA V/C) ≈1070 ns+(625*DMA V/C)

  where DMA V/C is the maximum number of consecutive video/cursor DMA operations that may occur while the sound DMA is pending.

Selecting Video or Cursor DMA Operations

The –HSYC signal determines whether a video or cursor DMA is to be performed, and is latched on the high-to-low transition of –VDRQ. The hold time on –HSYC must be sufficient to allow the synchronization latches in MEMC to capture its state. When two or more video/cursor DMA operations occur consecutively, –HSYC is sampled on the falling edge of the penultimate –VDAK acknowledge strobe, and its state determines whether the next DMA operation will fetch video or cursor data.

Flyback Requirements

The video and cursor buffer pointers must be reset between each video frame. The cursor pointer is initialized during flyback (signalled by FLBK high). The initialization takes 250 ns and takes place automatically provided that the following conditions are met:

- FLBK is high (indicating flyback is in operation). It takes up to 250 ns for MEMC to synchronize and process the low-to-high transition of FLBK.
- No DMA or refresh operation is being serviced.
- The processor is performing a non-sequential memory access, but is not writing to the DMA address generator or the MEMC Control Register.

Note that these conditions may be satisfied many times during flyback, and the cursor pointer will be initialized on each occasion. The FLBK signal must remain high long enough to initialize the cursor pointer at least once.

The video pointer is not reset until after FLBK makes a transition from high-to-low (the end of the flyback period). This allows the video pointer to be used as a refresh address register during flyback. The initialization takes 250 ns and occurs automatically provided that the following conditions are satisfied:

- FLBK has made a transition from high-to-low (signaling the end of flyback), and the video pointer has not already been initialized. It takes up to 250 ns for MEMC to synchronize and process the high-to-low transition of the FLBK signal.
- No DMA or refresh operation is being serviced.
- The processor is performing a non-sequential memory access, but is not writing to the DMA Address Generator or the MEMC Control Register.

The delay between FLBK going low and the first video DMA being processed must be long enough to allow the video pointer to be reset.

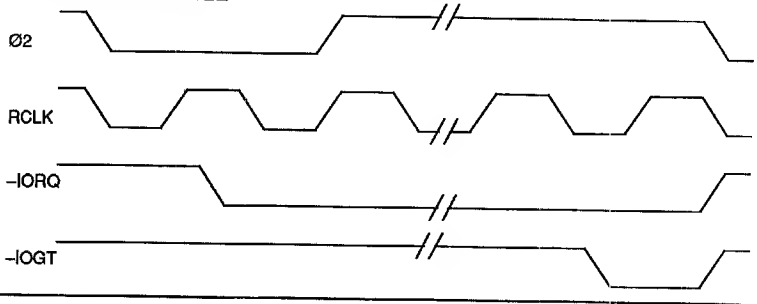
Programming the DMA Address Generators

The DMA address generators are limited to addressing the bottom 0.5 Mbyte of physical memory. However, the processor can use the logical-to-physical address translator to make this 0.5-Mbyte block appear anywhere in the 32-Mbyte logical address space.

MEMC does not monitor the processor data bus, so the DMA address generators are programmed by encoding the data in the address and performing a write operation with MEMC in Supervisor mode. Figure 8 on page 4-16 shows how an address is calculated to store data in the address generator registers. The following points should be noted:

- The address generator has a resolution of 16 bytes (the number of bytes read during a DMA transfer). All DMA buffers must be aligned on 16 byte boundaries. The value stored in the address generator is the appropriate address divided by 16.
- When the sound pointer register, Sptr, is programmed, no immediate value is specified. Instead, a sound buffer swap is forced, copying the value from Sstart to Sptr, and resetting the next buffer valid flag.
- The processor may write to the DMA registers at any time, but multiple, consecutive DMA register write operations (e.g., using the VL86C010 Store Multiple instruction) should never be used, as this may inhibit the initialization of the video pointer register, Vptr.

FIGURE 18. I/O CYCLE



Video Controller (VIDC) Interface
To program the VIDC video controller, the processor performs a write operation anywhere in the video controller address space while MEMC is in supervisor mode. The video controller register number and data are encoded entirely in a 32-bit word which is available on the processor data bus during the write operation (see the VL86C310 data sheet for more detail). MEMC provides a video controller write signal, –VIDW, that latches the information off the data bus.

I/O Controller Interface
The IOC I/O controller provides a unified view of interrupts and peripherals within the VL86C010 based system (see the VL86C410 data sheet for more details). The processor can access a number of I/O controllers through its memory map, and MEMC provides a handshaking control system for processor to I/O controller interactions.

When the processor accesses the Input/Output controller address space (with MEMC in supervisor mode), MEMC starts the I/O cycle by taking –IORQ low, and holding the processor clocks (stretching the processor cycle with Ø2 high). If the –IORQ and –IOGT signals are both low on the rising edge of RCLK, the I/O cycle will end on the next falling edge of RCLK. MEMC then releases the processor clocks, and sets the I/O request line, –IORQ, high; the I/O controller will set the I/O grant line, –IOGT, high and read or write data from/to the processor data bus. An I/O cycle is shown in Figure 18. The cycle starts with –IORQ being taken low. Then follows a number of 8 MHz cycles until the I/O controller is in a position to complete the cycle. The –IOGT line is taken low, and both MEMC and the I/O controller see –IORQ and –IOGT low on the rising edge of RCLK, so the I/O cycles terminates on the next rising edge of RCLK.

DMA and refresh operations are allowed during I/O cycles. To prevent clashes on the data bus, MEMC ensures that the I/O cycle does not end during these operations by taking –IORQ high until they have finished, as shown in Figure 19.

Some I/O cycles may only take a single non-sequential cycle (250 ns). To give the Input/Output controller adequate time to recognize such operations, MEMC produces the first –IORQ early in the I/O cycle (see Figure 20).

The extension of –IORQ only happens at the start of an I/O cycle; if the –IORQ signal is removed during a DMA or refresh operation, it will be reasserted as RCLK goes low.

Care must be taken not to address a non-existent Input/Output Controller, as MEMC will hold the processor clocks indefinitely until a low is seen on the –IOGT line, or RES is set high.

FIGURE 19. I/O CYCLE INTERRUPTED BY DMA OR REFRESH

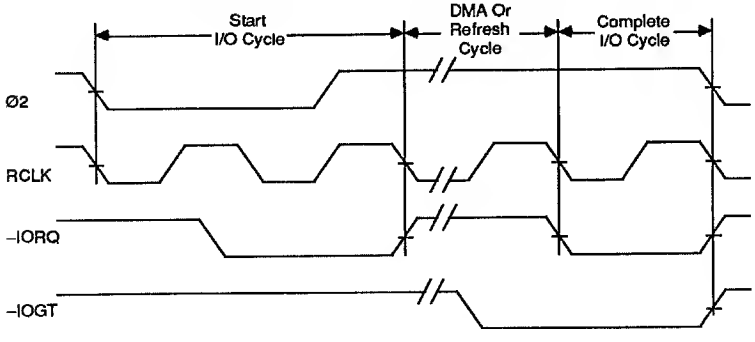
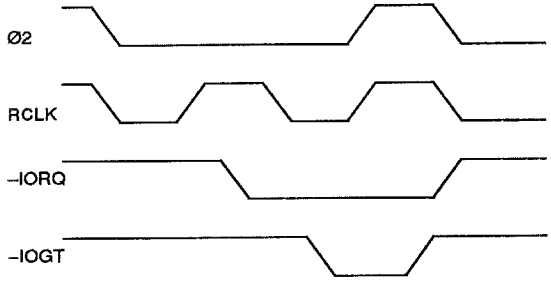


FIGURE 20. FAST I/O CYCLE



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

Symbol	Parameter	10 MHz			12 MHz			Unit	Condition
		Min	Typ.	Max	Min	Typ.	Max		
tCK	Clock Period	100			80			ns	Note 1
tV	Clock Non-overlap	0		10	0		10	ns	@ 1.0 V
tCKL	Clock Low Time	45			38			ns	@ 0.3 V
tCKH	Clock High Time	34			24			ns	@ 4.7 V
tØRF	Ø1, Ø2 Rise/Fall			8			7	ns	Note 1, 2
tØ1L	Ø1 Low to RCLK			7			7	ns	Note 1, 2
tØ1H	RCLK to Ø1 High			8			8	ns	Note 1, 2
tØ2L	Ø2 Low to RCLK	-2			0			ns	Note 1, 2
tØ2H	RCLK to Ø2 High			12			12	ns	Note 1, 2
tAS	A25-A0 Setup (N-Cycle)	130			105			ns	Note 2, 3
tAS	A25-A0 Setup (S-Cycle)	30			25			ns	Note 2, 4
tAH	A25-A0 Hold to RCLK	15			15			ns	Note 2
tRWH	–R/W Hold to RCLK	5			5			ns	Note 2
tDBED	DBE Delay from RCLK			16			15	ns	Note 2
tDBERD	DBE Delay from –R/W			40			40	ns	
tABTD	ABRT Delay			95			50	ns	Note 5
tABTRD	ABRT Delay from –R/W			100			50	ns	Note 5
tABTH	ABRT Hold Time	10			10			ns	Note 2
tMSS	–MREQ & SEQ Setup	15			15			ns	Note 2
tMSH	–MREQ & SEQ Hold Time	15			15			ns	Note 2
tBWS	–B/W Setup to RCLK	50			40			ns	Note 2, 6
tBWH	–B/W Hold Time	5			5			ns	Note 2
tRASS	–RAS Setup to RCLK (N-Cyc)	7			5			ns	Note 2
tRASS	–RAS Setup to RCLK (X-Cyc)	15			13			ns	Note 2
tRAS	–RAS Pulse Width	100			80			ns	Note 7

Notes:
1. 10 MHz operation assumes load on Ø1, Ø2 is reduced to 25 pF.
2. All timings are referenced to RCLK 50% point, with nominal load on all signals. Signals other than RCLK are referenced to high and low levels defined in DC operating conditions section (for inputs) and DC characteristics section (for outputs).
3. This constraint applies to non-sequential cycles (N-cycles) only.
4. This constraint applies to internal cycles (I-cycles) only.
5. This constraint applies only to N-cycles and merged I- and S-cycles.
6. This constraint applies only to cycles where –CAS0- –CAS3 is active.
7. –RAS may be extended by up to 3Trcf for sequential DRAM cycles.
8. These figures apply to read cycles (early –CAS). –CAS will only go low when –RAS is low.
9. These figures apply to write cycles (late –CAS). –CAS will only go low when –RAS is low.

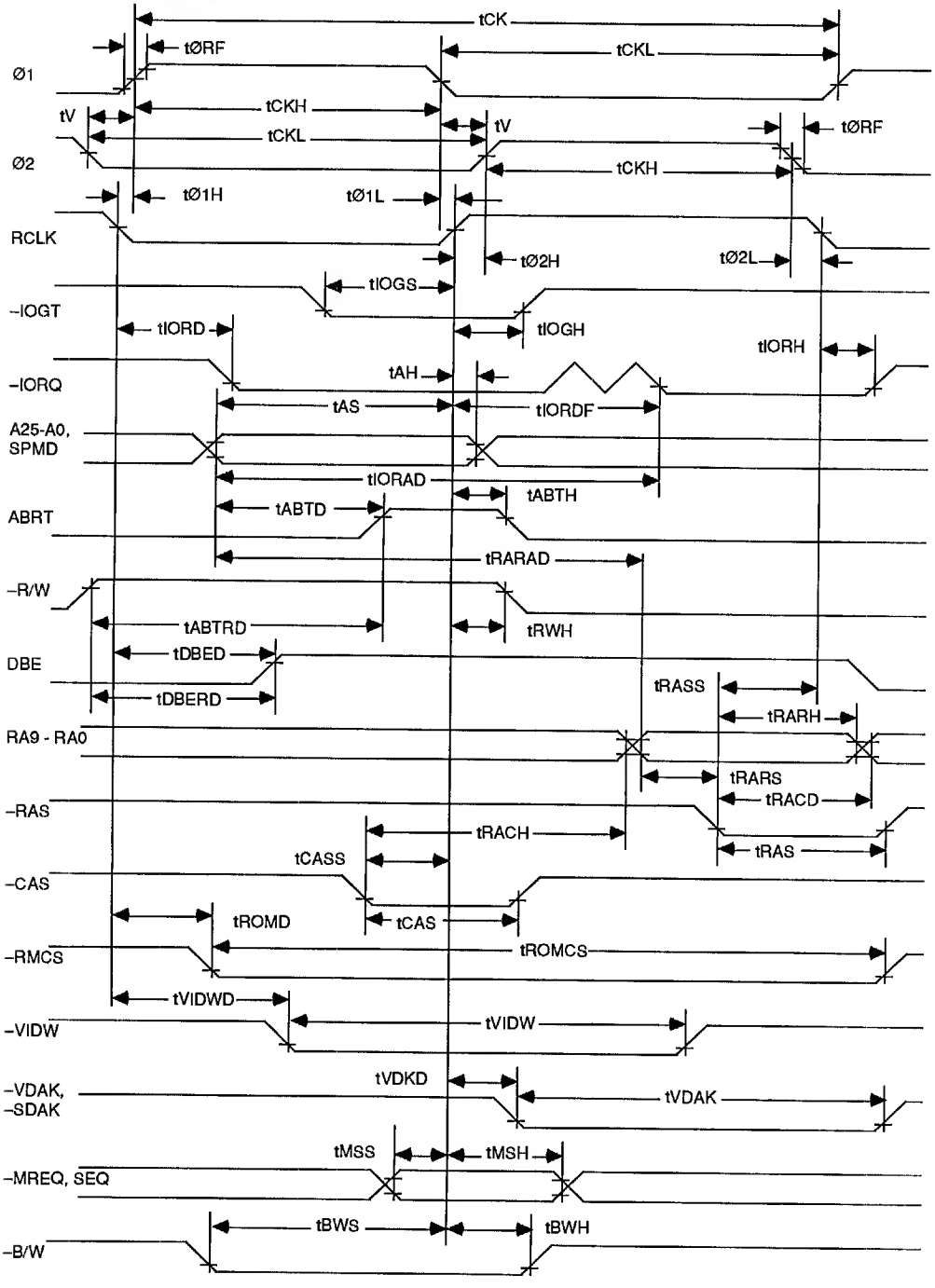
TIMING CHARACTERISTICS (Cont.): TA = 0°C to +70°C, VDD = 5 V ±5%

Symbol	Parameter	10 MHz			12 MHz			Unit	Condition
		Min	Typ.	Max	Min	Typ.	Max		
tCASS	–CAS Setup Read (N-Cyc)	8			2			ns	Note 2, 8
tCASS	–CAS Setup Read (X-Cyc)	20			18			ns	Note 2, 9
tCASS	–CAS Setup Write (N-Cyc)	–5			–5				
tCASS	–CAS Setup Write (X-Cyc)	5			5				
tCAS	–CAS Pulse Width	40			30			ns	
tRARAD	Row RA9-RA0 from A25-A0			65			50	ns	
tRARS	RA9-RA0 Setup to –RAS	20			10			ns	Note 2, 10
tRARH	RA9-RA0 Hold to –RAS	15			10			ns	
tRACD	Column RA9-RA0 from –RAS			35			30	ns	
tRACH	RA9-RA0 Hold to –CAS	20			20			ns	
tIORAD	–IORQ Delay from A25-A0			80			55	ns	Note 2, 11
tIORDF	–IORQ Delay (First)	5		25	5		21	ns	Note 2, 11
tIORD	–IORQ Delay	–2		10	–2		10	ns	Note 2, 12
tIORH	–IORQ Hold Time	–12		0	–12		0	ns	Note 2
tIOGS	–IOGT Setup to RCLK	20			15			ns	Note 2, 13
tIOGH	–IOGT Hold Time	13			13			ns	Note 2, 13
tROMD	–RMCS Delay	0		20	0		15	ns	Note 2
tROMCS	–RMCS Pulse Width	180		395	140		315	ns	Note 14
tVIDWD	–VIDW Delay	–2		13	–1		10	ns	Note 2
tVIDW	–VIDW Pulse Width	50		68	40		58	ns	
tVDKD	–VDAK, SDAK Delay	0		13	0		13	ns	Note 2
tVDAK	–VDAK, SDAK Width	40		55	30		45	ns	

Notes:
10. These figures apply to DMA cycles.
11. These figures apply to the first IORQ of an I/O transaction.
12. These figures apply to the second and subsequent IORQs of an I/O transaction.
13. Only significant when –IORQ is low.
14. These figures apply to a single ROM access. Nibble mode accesses may be one clock period longer than the maximum figure, and RMCS may remain low for multiple consecutive ROM accesses.
15. These times are not measured. The maximum delays are derived from SPICE models of the relevant logic functions, with VLSI slow-slow transistor models, VDD=4.7 volts, VSS=0.1 volts, temperature 100 degrees Centigrade. The minimum hold times are calculated from the same models of the relevant paths, with the time in the table being the slow path time divided by four. All numbers have been rounded to the nearest 5 ns. All numbers are subject to change after device characterization.
16. Output times are to CMOS levels except for the DRAM interface signals (–RAS, –CAS0- –CAS3, RA0-RA9) which are to TTL levels.

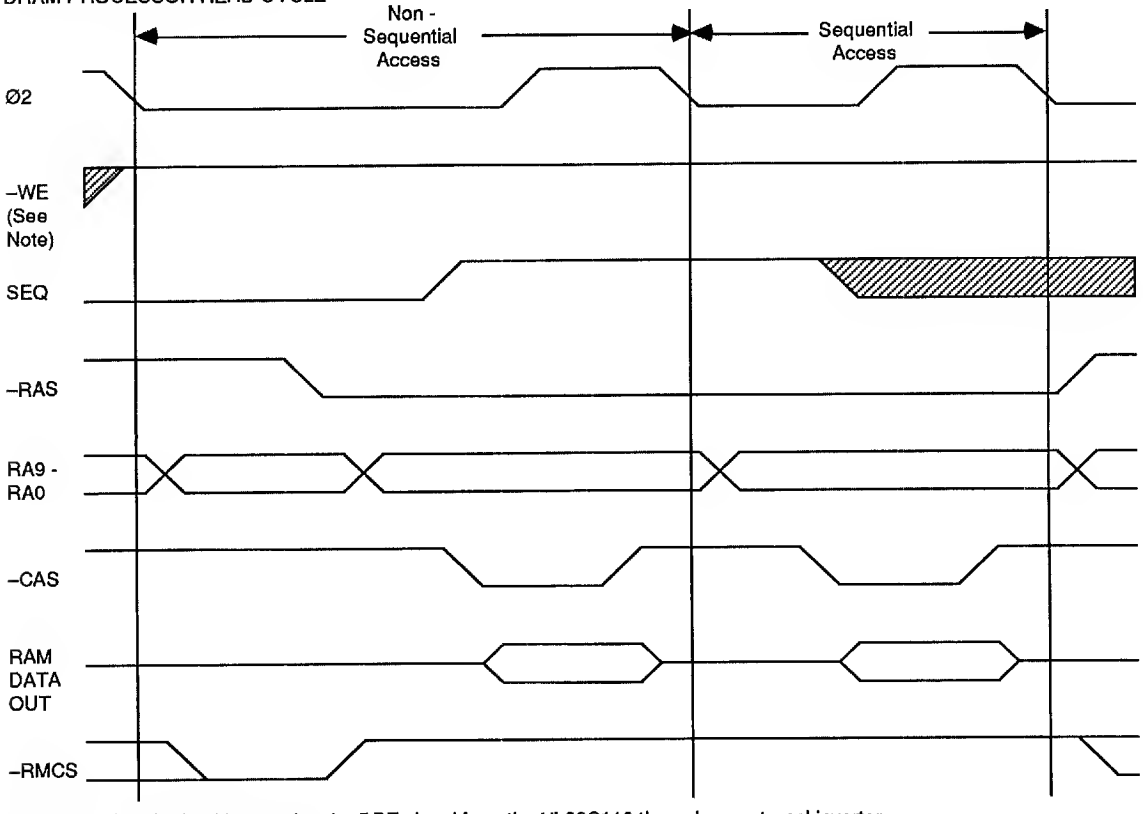
The timing diagrams included in this section represent typical AC waveforms in a MEMC system.

TIMING DIAGRAMS



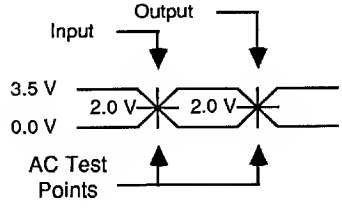
TIMING DIAGRAMS

DRAM PROCESSOR READ CYCLE

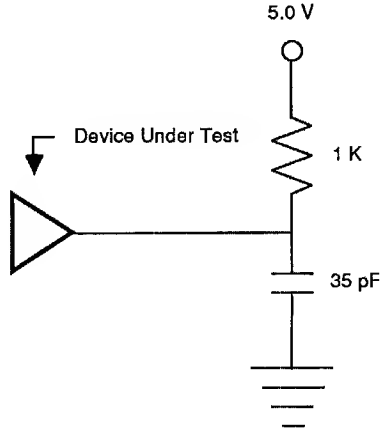


Note: –WE is obtained by passing the DBE signal from the VL86C110 through an external inverter.

A.C. TEST WAVEFORMS

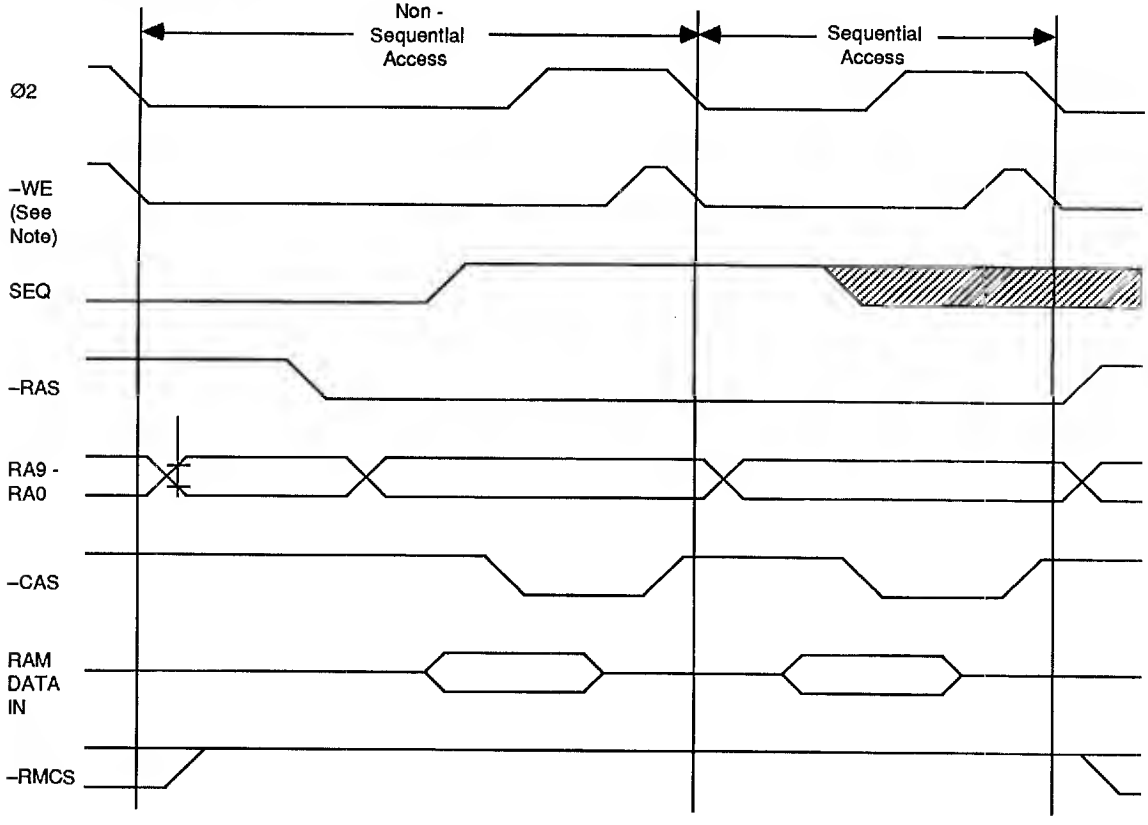


A.C. LOAD CIRCUIT



TIMING DIAGRAMS

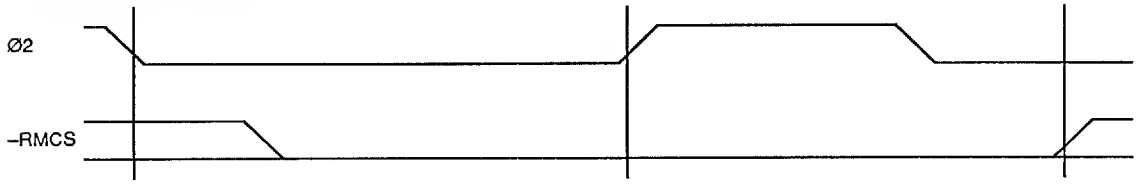
DRAM PROCESSOR WRITE CYCLE



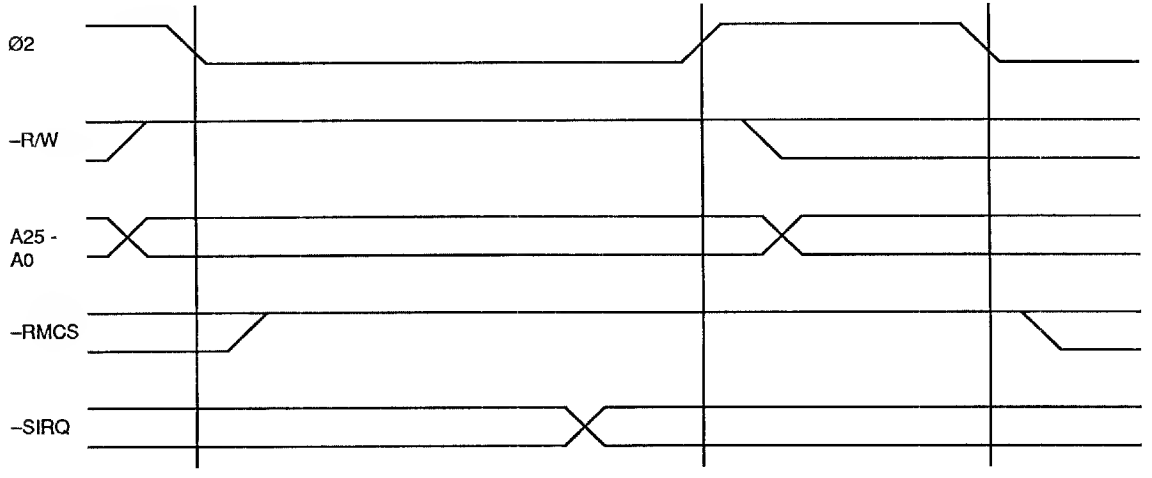
Note: –WE is obtained by passing the DBE signal from the VL86C110 through an external inverter.

TIMING DIAGRAMS

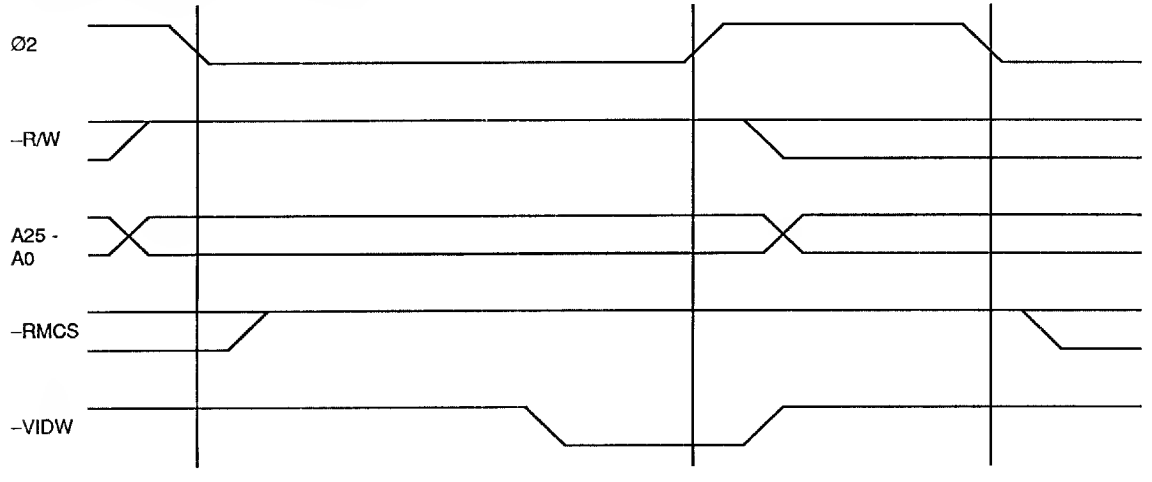
PROCESSOR ROM ACCESS



PROCESSOR MEMC REGISTER ACCESS

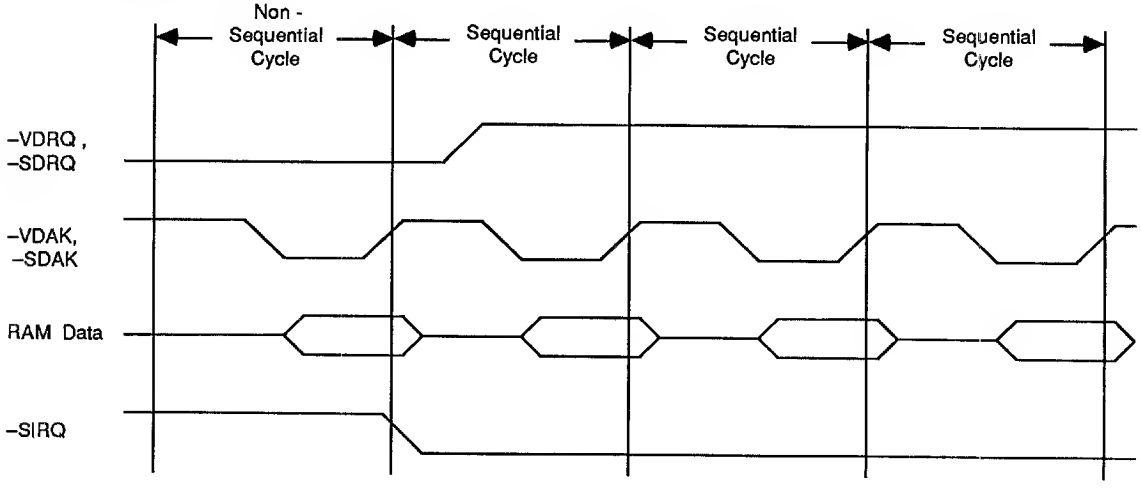


PROCESSOR VIDC REGISTER ACCESS

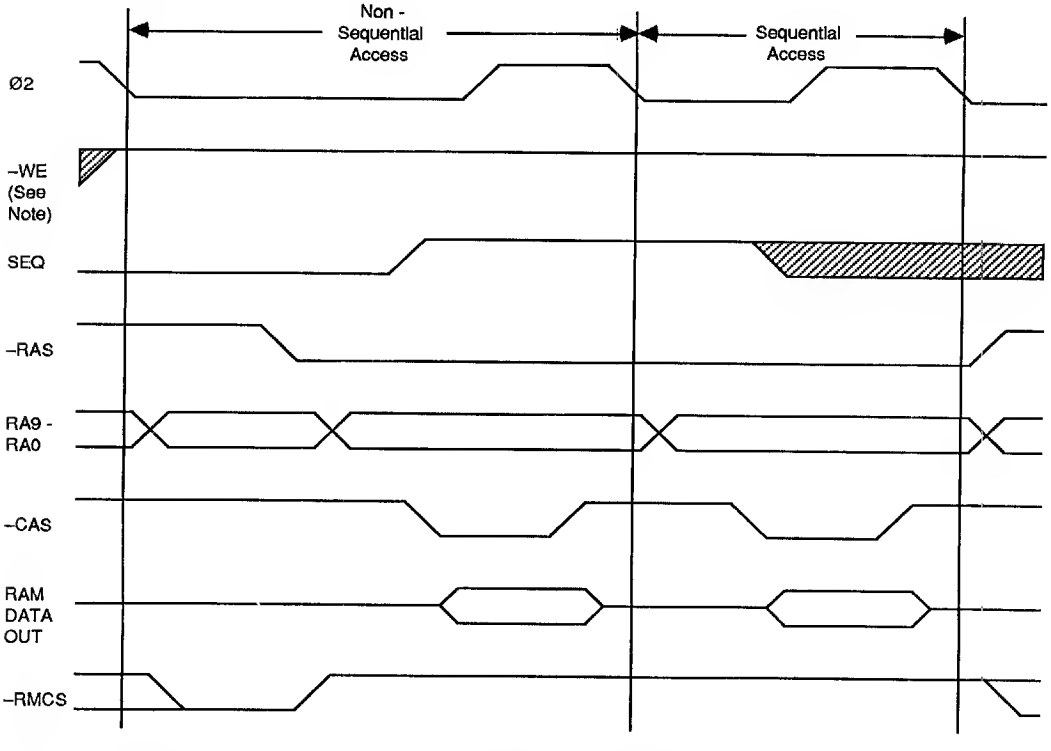


TIMING DIAGRAMS

DMA OPERATIONS



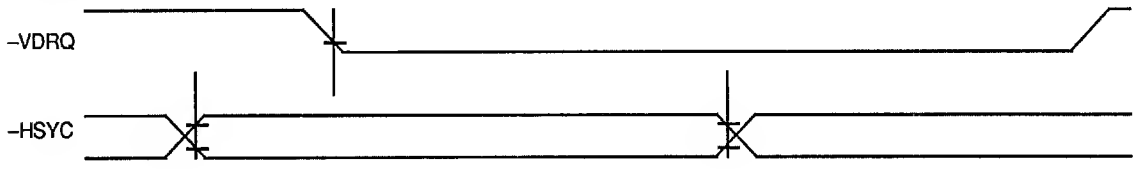
DMA OPERATIONS - DRAM READ CYCLES



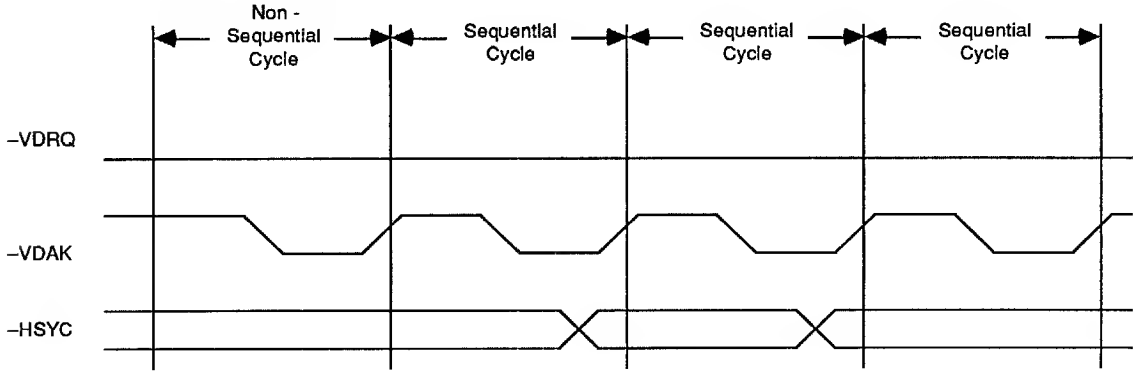
Note: -WE is obtained by passing the DBE signal from the VL86C110 through an external inverter.

TIMING DIAGRAMS

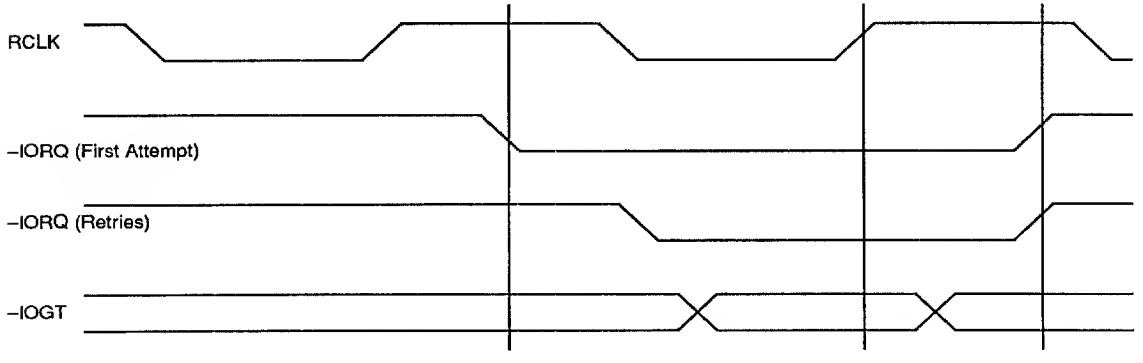
CURSOR/VIDEO DMA REQUEST



CONSECUTIVE CURSOR/VIDEO DMA REQUESTS



I/O CONTROLLER HANDSHAKING



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 Voltag	–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
VOH	Output High Voltage	Ref CIN	VDD-1.25	–	VDD	V	IOH = –10.0 mA
		TTL	2.4	–	VDD	V	IOH = –10.0 mA
		Ø1, Ø2	VDD-0.5	–		V	IOL = –5.0 mA
		CMOS	VDD-1.0	–		V	IOL = –5.0 mA
VOL	Output Low Voltage	Ref CIN			0.4	V	IOH = 10.0 mA
		TTL			0.8	V	IOH = 10.0 mA
		Ø1, Ø2			0.3	V	IOL = 5.0 mA
		CMOS			0.8	V	IOL = 5.0 mA
VIHC	CMOS Input High Voltage		3.5	–	–	V	
VIHT	TTL Input High Voltage		2.4	–	–	V	
VIL	Input Low Voltage CMOS and TTL		–	–	0.8	V	
ILI	Input Leakage Current		–	–	10	µA	VIN = 0 V to VDD
ICC	Operating Supply Current		–	–	70	mA	
IOS	Output Short Circuit Current		–	–	25	mA	See Note.

Note: No more than one output should be shorted to either rail at a time and for no longer than one second duration.

Appendix A - RAM Address Bus

The RAM address bus (RA9-RA0), as output from the multiplexer, is derived from three sources: processor address bus, logical-to-physical address translator, and DMA address generators. When the processor accesses physically mapped RAM, the higher order address bits (A25-A2) define the physical page and the low order bits the word offset within the page. Similarly, when the processor requests access to logically mapped RAM, the logical-to-physical address translator supplies the physical page number (PPN6-PPN0) if the current privilege level is sufficient. The row and column addresses supplied to the DRAM on the RAM addresss bus during processor cycles are a combination of the physical page number (PPN6-PPN0) and the word offset (A14-A2).

During DMA operations the DMA address generators supply a 17-bit word address value within the DRAM address spaca. The DMA operations can only address 512 Kbytes and the upper address bits are forced to zeros which forces the access into the bottom area of the memory map.

Rafresh operations use the Video Pointer register as the address value, while the values on the address bus are a function of the page size selected. They are shown in Figure A-1 for each of the four page sizes.

Please note that the RAM address multiplexer has inverting output drivers and the address values observed are complemented from the input value.

FIGURE A-1. RAM ADDRESS BUS VALUES FOR DIFFERENT PAGE SIZES

4-KByte Page Size

RAM Address Bus

		RA9	RA8	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
Processor Access	Row Address	X	X	–A11	–A10	–A9	–A8	–A7	–A6	–A5	–A4
	Column Address	X	–PPN6	–PPN5	–PPN4	–PPN3	–PPN2	–PPN1	–PPN0	–A3	–A2
DMA And Refresh Operations	Row Address	X	X	–DMA11	–DMA10	–DMA9	–DMA8	–DMA7	–DMA6	–DMA5	–DMA4
	Column Address	X	–DMA18	–DMA17	–DMA16	–DMA15	–DMA14	–DMA13	–DMA12	–DMA3	–DMA2

8-KByte Page Size

		RA9	RA8	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
Processor Access	Row Address	X	–A12	–A11	–A10	–A9	–A8	–A7	–A6	–A5	–A4
	Column Address	X	–PPN5	–PPN4	–PPN3	–PPN2	–PPN1	–PPN0	–PPN6	–A3	–A2
DMA And Refresh Operations	Row Address	X	–DMA12	–DMA11	–DMA10	–DMA9	–DMA8	–DMA7	–DMA6	–DMA5	–DMA4
	Column Address	X	–DMA18	–DMA17	–DMA16	–DMA15	–DMA14	–DMA13	1	–DMA3	–DMA2

16-KByte Page Size

		RA9	RA8	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
Processor Access	Row Address	X	–A12	–A11	–A10	–A9	–A8	–A7	–A6	–A5	–A4
	Column Address	–PPN6	–PPN4	–PPN3	–PPN2	–PPN1	–PPN0	–A13	–PPN5	–A3	–A2
DMA And Refresh Operations	Row Address	X	–DMA12	–DMA11	–DMA10	–DMA9	–DMA8	–DMA7	–DMA6	–DMA5	–DMA4
	Column Address	1	–DMA18	–DMA17	–DMA16	–DMA15	–DMA14	–DMA13	1	–DMA3	–DMA2

32-KByte Page Size

		RA9	RA8	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
Processor Access	Row Address	–A13	–A12	–A11	–A10	–A9	–A8	–A7	–A6	–A5	–A4
	Column Address	–PPN5	–PPN3	–PPN2	–PPN1	–PPN0	–A14	–PPN6	–PPN4	–A3	–A2
DMA And Refresh Operations	Row Address	–DMA13	–DMA12	–DMA11	–DMA10	–DMA9	–DMA8	–DMA7	–DMA6	–DMA5	–DMA4
	Column Address	1	–DMA18	–DMA17	–DMA16	–DMA15	–DMA14	1	1	–DMA3	–DMA2

VLSI TECHNOLOGY, INC.

SECTION 5

VL86C310 RISC VIDEO CONTROLLER (VIDC)

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

VL86C310

RISC VIDEO CONTROLLER (VIDC)

FEATURES

- Pixel rate selectable as 8, 12, 16, or 24 MHz
- Serializes data to 1-, 2-, 4-, or 8- bits per pixel
- 16 x 13-bit words - 4096 color lookup palette
- Three 4-bit DACs (one for each CRT gun)
- Fully programmable screen parameters
- Screen border in any of the 4096 possible colors
- Flexible cursor sprite
- Support for interlaced display format
- External synchronization capability
- Very high resolution monochrome mode support
- High quality stereo sound generation

DESCRIPTION

The Video Controller (VIDC) accepts video data from DRAM under DMA control, serializes and passes it through a color look-up palette, and converts it to analog signals for driving the CRT guns. The chip also controls all the display timing parameters plus the position and pattern of the cursor sprite. In addition, the VIDC includes an exponential DAC and stereo image table for the generation of high quality sound from data in the DRAM.

The VIDC requests data from the RAM when required, and buffers it in one of three first-in, first-out memories (FIFOs). Note that the addressing of the data in RAM is controlled elsewhere in the system (usually in the VL86C110 Memory Controller, MEMC). Data is requested in blocks of four 32-bit words, allowing efficient use of page-mode DRAM without locking up the system data bus for long periods.

The VIDC is a highly programmable device, offering a very wide choice of display formats. The pixel rate can be selected in a range between 8 and 24 MHz and the data can be serialized to either 8-, 4-, 2-, or 1-bit per pixel. The horizontal timing parameters can be controlled to units of 2 pixels, and the vertical timing parameters can be controlled in units of a raster. The color lookup palette which drives the three on-chip DACs is 13-bits wide, offering a choice from 4096 colors or an external video source.

Extensive use is made of pipelining throughout the device.

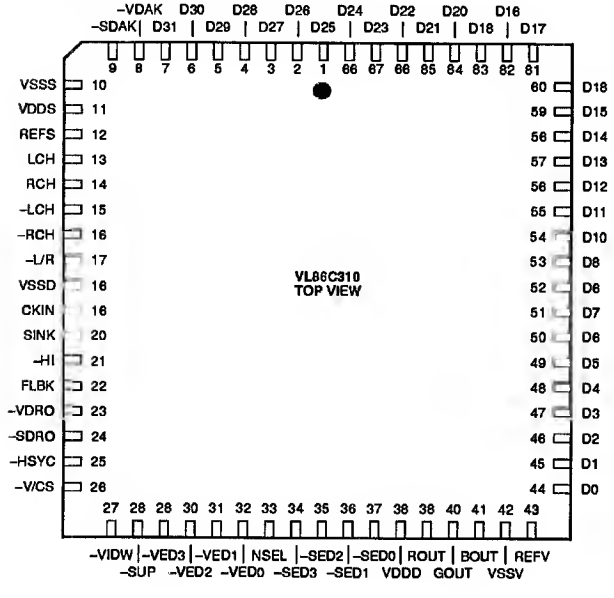
The cursor sprite is 32 pixels wide, and any number of rasters high. Three simultaneous colors (from the 4096 possible) are supported, and any pixel can be defined as transparent, making possible cursors of many shapes. The cursor can be positioned anywhere on the screen.

The sound system implemented on the device can support up to eight channels, each with a separate stereo position.

5

PIN DIAGRAM

PLASTIC LEADED CHIP CARRIER

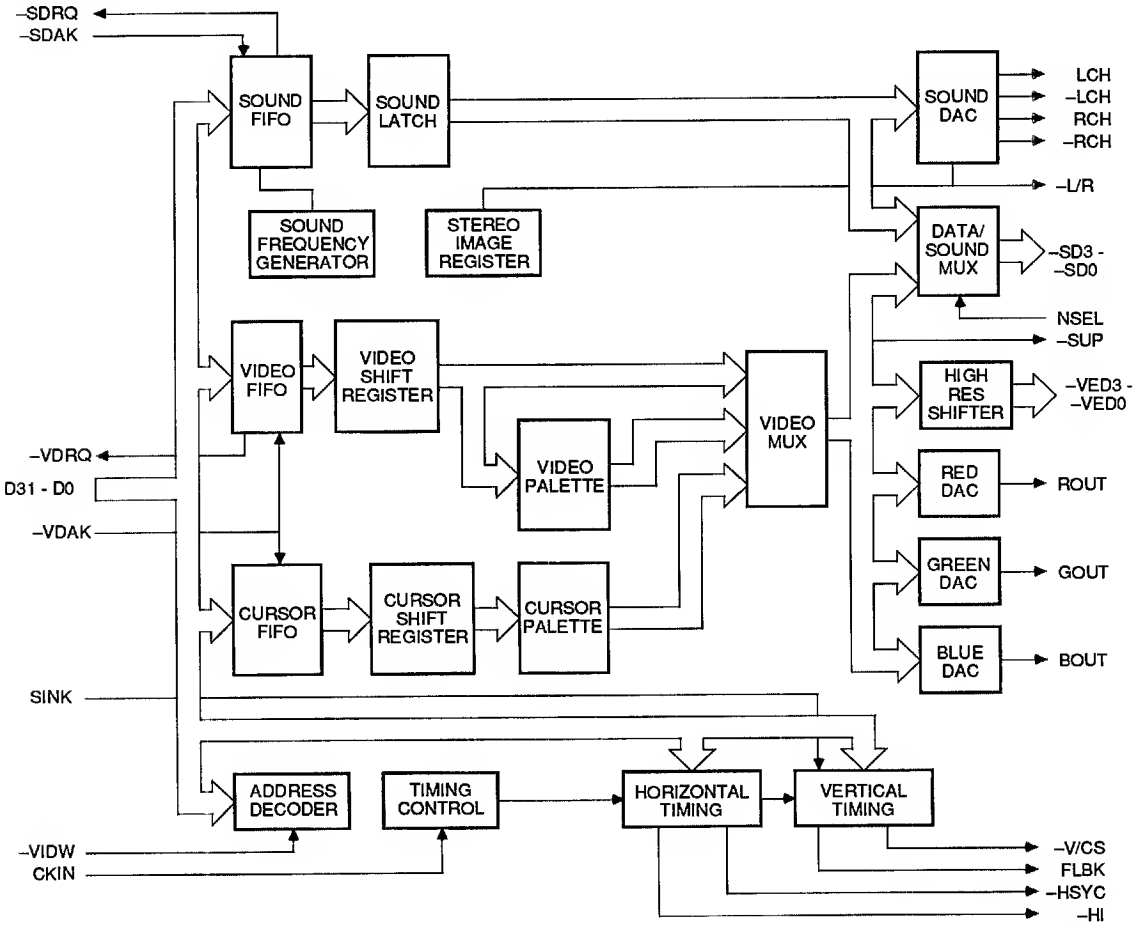


ORDER INFORMATION

Part Number	Clock Frequency	Package
VL86C310-12QC	12 MHz	Plastic Leaded Chip Carrier (PLCC)

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

BLOCK DIAGRAM



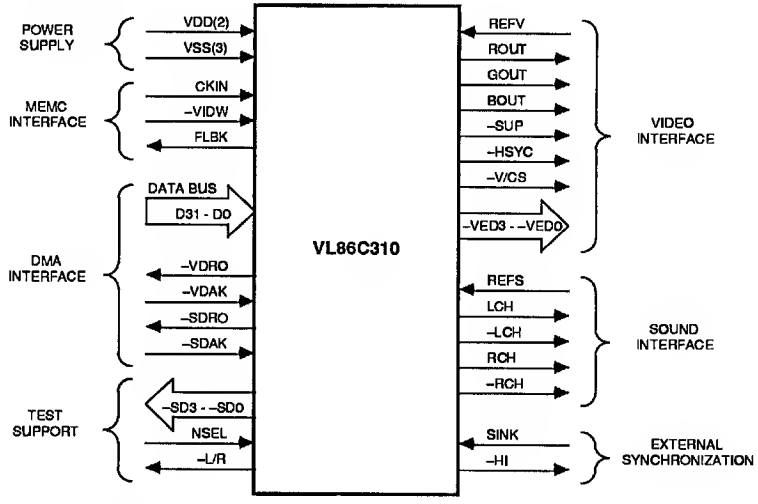
SIGNAL DESCRIPTIONS

Signal Name	Pin Number	Signal Description
CKIN	19	Clock In (TTL level input); Master 24 MHz system clock input - Usually this is the same signal as the VL86C110 Memory Controller (MEMC) uses to generate system timing. Since VIDC resynchronizes all its inputs to this clock reference, these two clocks are not required to be the same frequency, allowing the display frequency to be independent of the processor.
–VIDW	27	Register Write Strobe (TTL level input) - An active low on this line writes data into one of the VIDC registers. The address of the register is supplied on the upper bits, and the data to be written on the lower bits of the data bus. Normally, this signal is generated by MEMC as it is the device that decodes the memory address map in the system.
D31-D0	7-1, 68-44	Data Bus (TTL level inputs) - This 32-bit bus carries data for register writes, video DMA, cursor DMA, and sound DMA, according to which type of data strobe is present.
–VDRQ	23	Video Data Request (CMOS level output) - This signal is driven active (low) when the VIDC requires another block of 16 bytes of video data (when –HSYC is high) or cursor data (when –HSYC is low). It is driven high again by the first valid video data acknowledge, –VDAK.
–VDAK	8	Video Data Acknowledge (TTL level input) - An active low on this signal strobes a data word into the video or cursor FIFO depending on the state of HSYNC when the request was made. Note that a low on –VDRQ signifies a request for four words of data, and so –VDAK must go low four times to service each request.
–SDRQ	24	Sound Data Request (CMOS level output) - This signal is driven low when the VIDC requires another block of 16 bytes of sound data. It is driven high again by the first valid –SDAK.
–SDAK	9	Sound Data Acknowledge (TTL level input) - An active low on this signal strobes a data word into the sound FIFO. Note that a low on –SDRQ signifies a request for four words of data, and so –SDAK must go low four times to service each request.
FLBK	22	Vertical Flyback (CMOS level output) - This signal is driven high when the display is in vertical flyback (retrace). Specifically, it is set high at the start of the first raster which is not display data, although this may be border, (at the bottom of the screen), and is cleared down at the start of the first raster which is display data (at the top of the screen).
SINK	20	External Synchronization pulse (TTL level input) - A high on this signal resets the vertical timing counter, and if interlaced display format is being used, the odd field is selected. The horizontal timing counter, and all other registers are unaffected by this signal.
–HI	21	Horizontal Interlace Marker (Test pin - CMOS level output) - When an interlaced display format is selected this signal is driven low half way along the raster and stays low until the end of each raster. If non-interlaced displays are used, this pin may be used as a programmable timer on each raster.
–SD3 - –SD0	34-37	Multiplexed Sound Data (Test pins - CMOS level outputs) - These pins are used for testing the digital data paths through the chip. Normally, depending on the state of NSEL, they output the inverse of one of the two nibbles of the data byte being fed to the sound DAC, but in test mode three, they output the inverse of the data being fed to the green or blue DACs, again depending on the state of NSEL. For more information on test mode three, refer to the control register section.
NSEL	33	Sound Data Ouput Selector (Test pin - TTL level input) - When this signal is low, the sound data bus port outputs the low nibble of the sound data, or the green DAC data. When NSEL is high, the sound data bus port outputs the high nibble of the sound data, or the blue DAC data.
–L/R	17	Left/Right (Test pin - TTL level output) - This signal is driven low when the sound output is steered to the left output port, and is high when the sound output is steered to the right output port. In test mode three, the pin changes its function, and outputs the sound sampling clock instead.
REFV	43	Video DAC Reference Current (Analog input) - A current must be fed into this pin to set the output current of the video DACs. The full scale output current is 15 times this current. In most applications a resistor from VDD to this pin is sufficient to set the current.

SIGNAL DESCRIPTIONS (Cont.)

Signal Name	Pin Number	Signal Description
ROUT	39	Red Analog output (Analog output) - The output to the CRT guns is in the form of a current sink. Maximum brightness is defined as 15 times the reference current, and "black" is defined as zero current. Level shifting and buffering is normally required to drive the CRT inputs.
GOUT	40	Green Analog output (Analog output) - Same description as for ROUT.
BOUT	41	Blue Analog output (Analog output) - Same description as for ROUT.
–SUP	28	Supremacy output signal (CMOS level output) - This signal is used to control a multiplexer between the output of VIDC and an external source when video mixing is required. If bit 12 of the video or cursor palette for any logical color is set, –SUP is driven low when that logical color is displayed. In this way any logical color can be defined as being supreme or not, on a pixel-by-pixel basis.
–HSYC	25	Horizontal Synchronization pulse (CMOS level output) - This signal is required by some monitors. It is also used by the MEMC to discriminate between cursor and video data requests. The pulse is active low, and the pulse width is programmable in units of two pixels, though there are certain system-related restrictions. See section Restrictions On Parameters.
–V/CS	26	Vertical/Composite Synchronization pulse (CMOS level output) - Depending on bit seven in the control register, this pin can be either the vertical sync pulse, or a form of composite sync pulse. The vertical sync pulse width is programmable in units of a raster and, if selected, is active low. The composite sync pulse is the XNOR of –HSYC and –VSYC.
–VED0		
–VED3 -	39-32	Video External Data output (CMOS level output) - The inverse of the four bits of data which are fed to the red DAC are output on these pins. With an external serializer, this data can be used to produce very high resolution monochrome displays.
REFS	12	Sound DAC Reference Current (Analog input) - A current must be fed into this pin to set the output current of the sound DAC. The full scale output current is approximately 32 times this current. In most applications, a resistor from VDD to this pin is sufficient to set the current.
LCH	13	Left Channel Positive Sound output (Analog output) - The sound output is the form of a current sink which is switched to one of four pins (pins 13-16). The left channel signal is produced by externally integrating and subtracting the two signals, LCH and –LCH. Similarly, the right channel signal is produced by externally integrating and subtracting the two signals RCH and –RCH.
–LCH	14	Left Channel Negative Sound output (Analog output) - See description of LCH.
RCH	15	Right Channel Positive Sound output (Analog output) - See description of LCH.
–RCH	16	Right Channel Negative Sound output (Analog output) - See description of LCH.
VSSD	18	Power (Digital ground) - This pin is the ground supply to the digital circuits in the device.
VSSS	10	Power (Sound ground) - This pin is the ground supply to the sound DAC in the device. It must be connected to the pin VSSD outside the chip.
VSSV	42	Power (Video ground) - This pin is the ground supply to the video DACs in the device. It must be connected to the pin VSSD outside the chip.
VDDD	38	Power (Digital +5 V ±5% supply) - This pin is the positive supply to the digital circuits in the device.
VDDS	11	Power (Sound +5 V ±5% supply) - This pin is the positive supply to the sound DAC in the device. It must be at the same potential as VDDD, and should be decoupled to VSSS. Note that the sound reference current input and the sound analog output currents are all referenced to this signal.

FUNCTIONAL PIN DIAGRAM



FUNCTIONAL DESCRIPTION

Apart from the three 32-bit wide FIFOs (video, cursor, and sound), the VIDC contains 46 write-only registers of up to 13-bits each. In all cases the address of the register is contained in the top 6-bits (31-26) of the data field. Bits 25 and 24 are not used. The actual data bits are distributed among the remaining 24-bits of the data field according to the register in question. The encoding format is shown in Figure 1.

Treating bit 24 as the least significant address bit, the register map is shown in Table 1 on the following page. Note that there are 18 undefined locations. These locations should never be written to as they may actually contain other registers. (Some registers are dual-mapped within the device.)

In order to define the display format, eleven registers must be programmed. Screen parameter definitions are shown in Figure 2 on the following page.

Video Palette Logical Colors 0-FH: Addresses 00-3CH

In 1-, 2-, and 4-bits per pixel mode, data bits D12-D0 define the physical color corresponding to that logical color. The data bus encoding is shown in Figure 3. Figure 4 shows the physical color field specification.

FIGURE 1. DATA BUS ENCODING FORMAT

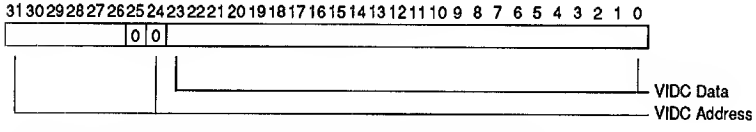


FIGURE 3. VIDEO PALETTE LOGICAL COLOR FORMAT

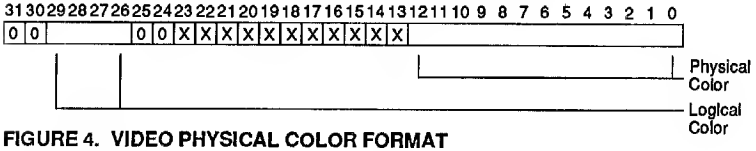


FIGURE 4. VIDEO PHYSICAL COLOR FORMAT

12	11	10	9	8	7	6	5	4	3	2	1	0
SUP	BLUE				GREEN				RED			
D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

D3-D0 define the red amplitude (D0 least significant)
D7-D4 define the green amplitude (D4 least significant)
D11-D8 define blue amplitude (D8 least significant)
D12 defines the supremacy bit for that color

In 8-bits per pixel mode, only 9-bits are defined as shown in Figure 5. The palette outputs define the least significant bits of each color. The most significant bits for each color now come directly from the upper 4-bits of the logical color field, giving the physical data field as shown in Figure 6.

In four and 8-bits per pixel mode, all 16 locations should be programmed. In 2-bits per pixel mode only colors zero, one, two, and three need to be defined. In 1-bit per pixel mode, only colors zero and one need to be programmed.

Border Color Register: Address 40H

In all modes, this register defines the border physical color. The data bus encoding is shown in Figure 7.

D3-D0 define the red amplitude (D0 least significant)
D7-D4 define the green amplitude (D4 least significant)
D11-D8 define the blue amplitude (D8 least significant)
D12 defines the supremacy bit for the border

FIGURE 2. VL86C310 DISPLAY PARAMETERS

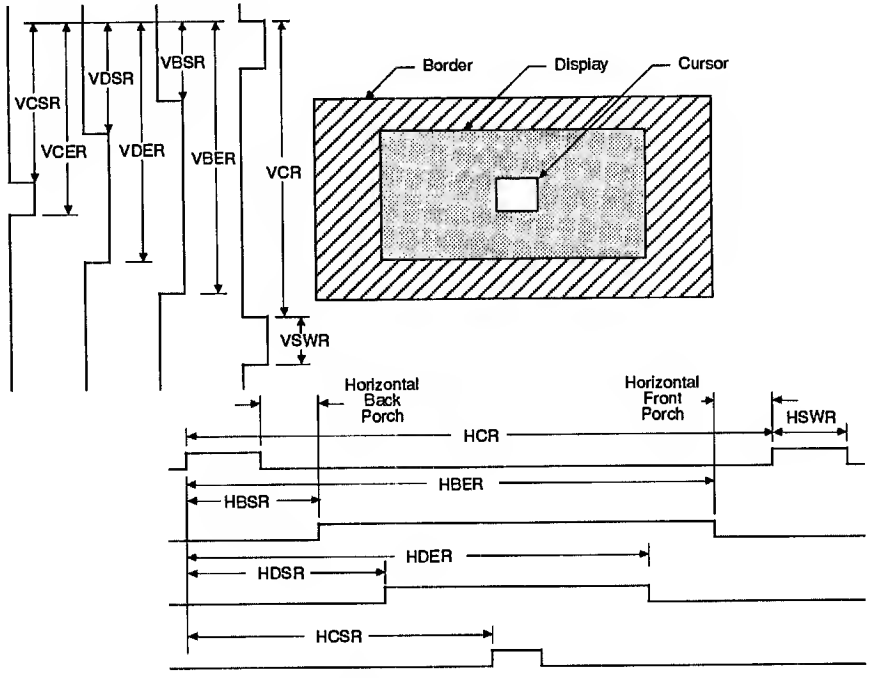


TABLE 1. REGISTER ADDRESS ASSIGNMENTS

Address (Hex)	Register Function	Address (Hex)	Register Function	Address (Hex)	Register Function
00	Video Palette Logical Color 0	44	Cursor Palette Logical Color 1	94	Horizontal Border End Register
04	Video Palette Logical Color 1	48	Cursor Palette Logical Color 2	98	Horizontal Cursor Start Register
08	Video Palette Logical Color 2	4C	Cursor Palette Logical Color 3	9C	Horizontal Interlace Register
0C	Video Palette Logical Color 3	50 - 5C	Reserved	A0	Vertical Cycle Register
10	Video Palette Logical Color 4	60	Stereo Image Register 7	A4	Vertical Sync Width Register
14	Video Palette Logical Color 5	64	Stereo Image Register 0	A8	Vertical Border Start Register
18	Video Palette Logical Color 6	68	Stereo Image Register 1	AC	Vertical Display Start Register
1C	Video Palette Logical Color 7	6C	Stereo Image Register 2	B0	Vertical Display End Register
20	Video Palette Logical Color 8	70	Stereo Image Register 3	B4	Vertical Border End Register
24	Video Palette Logical Color 9	74	Stereo Image Register 4	B8	Vertical Cursor Start Register
28	Video Palette Logical Color A	78	Stereo Image Register 5	BC	Vertical Cursor End Register
2C	Video Palette Logical Color B	7C	Stereo Image Register 6	C0	Sound Frequency Register
30	Video Palette Logical Color C	80	Horizontal Cycle Register	C4 - DC	Reserved
34	Video Palette Logical Color D	84	Horizontal Sync Width Register	E0	Control Register
38	Video Palette Logical Color E	88	Horizontal Border Start Register	E4 - FC	Reserved
3C	Video Palette Logical Color F	8C	Horizontal Display Start Register		
40	Border Color Register	90	Horizontal Display End Register		

FIGURE 5. VIDEO PALETTE DATA ENCODING FOR 8 BITS PER PIXEL MODE

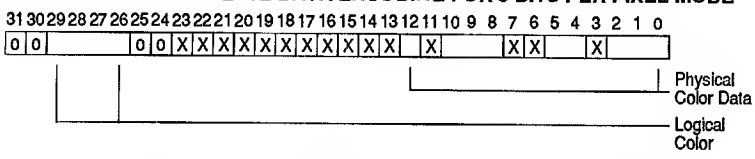


FIGURE 6. VIDEO PALETTE DEFINITION FOR 8 BITS PER PIXEL MODE

12	11	10	9	8	7	6	5	4	3	2	1	0
SUP	BLUE				GREEN				RED			
D12*	L7**	D10*	D9*	D8*	L6**	L5**	D5*	D4*	L4**	D2*	D1*	D0*

* Dn: These bits are from the palette field.
** Ln: These bits are from the logical field.

Cursor Palette Logical Colors 1-3: Address 44-4CH

In all modes, these registers define the physical cursor colors corresponding to the logical colors. Note that cursor logical color 00 is transparent (i.e., no cursor display), and this location is used for the Border Color Register. Figure 8 illustrates the data bus encoding for this register.

D3-D0 define the red amplitude (D0 least significant)
D7-D4 define the green amplitude (D4 least significant)
D11-D8 define the blue amplitude (D8 least significant)
D12 defines the supremacy bit for that cursor color

Stereo Image Registers; Channels 0-7: Addresses 60H-7CH

These eight registers define the stereo image position for each of the eight possible channels as shown in Table 1.

When only four channels are used, registers 4, 5, 6, 7 should be programmed to the same values as registers 0, 1, 2, 3 respectively. If only two channels are used, registers 0, 2, 4, and 6 pertain to one channel, and so should be programmed to the same value, and registers 1, 3, 5, and 7 pertain to the other channel. When only one channel is used, all eight registers should be programmed with the same value. The 3-bit value is defined in Table 2 and data bus encoding is shown in Figure 9.

FIGURE 7. BORDER COLOR REGISTER DATA BUS ENCODING

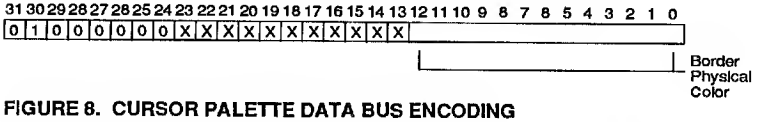


FIGURE 8. CURSOR PALETTE DATA BUS ENCODING

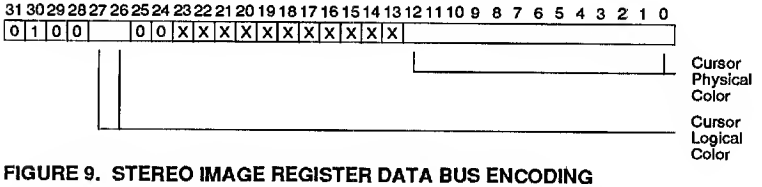


FIGURE 9. STEREO IMAGE REGISTER DATA BUS ENCODING

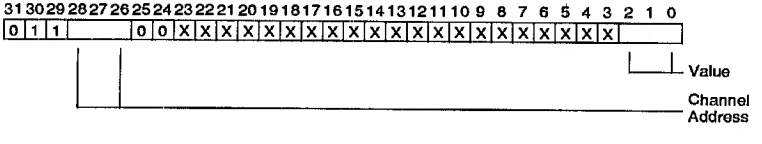


FIGURE 10. HORIZONTAL CYCLE REGISTER DATA BUS ENCODING

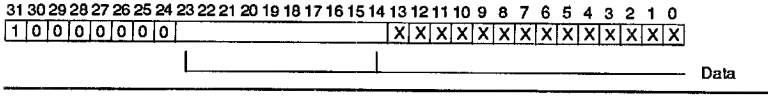


TABLE 2. STEREO IMAGE REGISTER VALUES

Value	Stereo Image Position
0	Undefined
1	100% Left Channel
2	83% Left Channel
3	67% Left Channel
4	Center
5	67% Right Channel
6	83% Right Channel
7	100% Right Channel

Horizontal Cycle Register (HCR): Address 80H

This register defines the period, in units of two pixels, of the horizontal scan (i.e., display time + horizontal retrace time). If N pixels are required in the horizontal scan period, then a value of (N-2)/2 should be programmed into the HCR (N must be even). If interlace display is selected, N/2 must also be even. This is a 10-bit register, with bit 14 the least significant. Data bus encoding is shown in Figure 10.

Horizontal Sync Width Register (HSWR): Address 84H

This register defines the width, in units of two pixel periods, of the horizontal sync pulse. Encoding of the data bus is shown in Figure 11. If N pixels are required in the horizontal sync pulse, then value (N-2)/2 should be programmed into the HSWR. (N must be even.) The minimum value programmed may be 0, but system constraints impose a larger minimum value. See section Restriction On Parameters. This is a 10-bit register, with bit 14 the least significant.

Horizontal Border Start Register (HBSR): Address 88H

This register defines the time, in units of two pixel periods, from the start of –HSYC pulse to the start of the border display. If M pixels are required in this time, then value (M-1)/2 should be programmed into the HBSR. (M must be odd.) Note that this register must

FIGURE 11. HORIZONTAL SYNC WIDTH REGISTER DATA BUS ENCODING

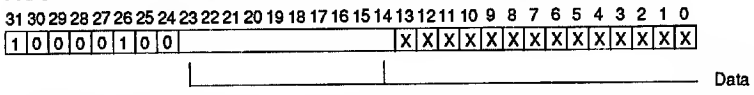


FIGURE 12. HORIZONTAL BORDER START REGISTER DATA BUS ENCODING

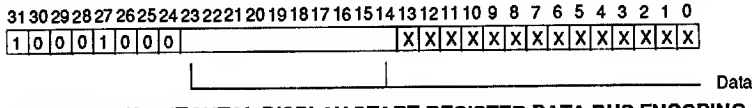


FIGURE 13. HORIZONTAL DISPLAY START REGISTER DATA BUS ENCODING

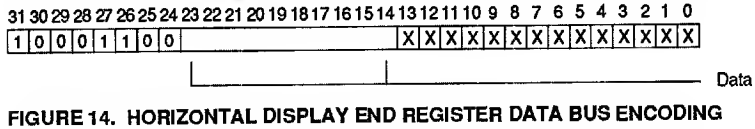


FIGURE 14. HORIZONTAL DISPLAY END REGISTER DATA BUS ENCODING

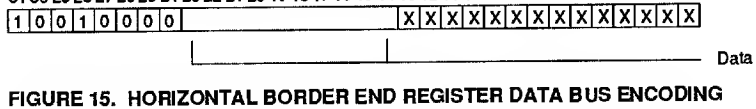


FIGURE 15. HORIZONTAL BORDER END REGISTER DATA BUS ENCODING

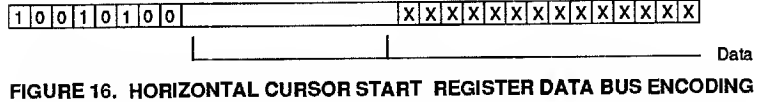
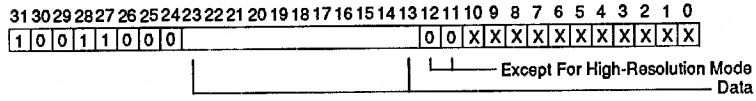


FIGURE 16. HORIZONTAL CURSOR START REGISTER DATA BUS ENCODING



always be programmed, even when a border is not required. If a border is not required, then the value in the HBSR must be such as to start the border in the same place as the display start (i.e, M[HBSR] = M[HDSR]). This is a 10-bit register with bit 14 the least significant. Data bus encoding is shown In Figure 12.

Horizontal Display Start Register (HDSR): Address 8CH

This register defines the time, in units of two pixel periods, from the start of the –HSYC pulse to the beginning of the video display. The value programmed here depends on the screen mode in use. If M pixels are required in this time, then: in 8-bits per pixel mode, the value (M-5)/2 should be program-med into the HDSR; in 4-bits per pixel mode, value (M-7)/2 should be programmed into the HDSR; in 2-bits per pixel mode, value (M-11)/2 should be programmed into the HDSR; in 1-bit per pixel mode, value (M-19)/2 should be programmed into the HDSR. M must be odd in all cases. This is a 10-bit register, with bit 14 the least significant. Data bus encoding for this register is shown in Figure 13.

Horizontal Display End Register (HDER): Address 90H

This register defines the time, in units of two-pixel periods, from the start of the horizontal sync pulse to the end of the video display (i.e., the first pixel which is not displayed). The value programmed here depends on the screen mode used. If M pixels are required in this time, then: in 8-bits per pixel mode, value (M-5)/2 should be programmed into the HDSR; in 4-bits per pixel mode, value (M-7)/2 should be programmed into the HDSR; in 2-bits per pixel mode, value (M-11)/2 should be programmed into the HDSR; in 1-bit per pixel mode, value (M-19)/2 should be programmed into the HDSR. M must be odd in all cases. This is a 10-bit register, with bit 14 the least significant. Figure 14 shows data bus encoding of register values.

Horizontal Border End Register (HBER): Address 94H

This register defines the time, in units of two-pixel periods, from the start of –HSYC pulse to the end of the border display (i.e., the first pixel which is not border). If M pixels are required in this time, then value (M-1)/2 should be programmed into the HBER. [M must be odd.] Again, if no border is required, this register must still be programmed such that M[HBER] = M[HDER]. This is a 10-bit register, with bit 14 the least significant. Data bus encoding for this register is shown in Figure 15.

Horizontal Cursor Start Register (HCSR): Address 98H

This register defines the time, in units of single pixel periods, from the start of the –HSYC pulse to the start of the cursor display. If M pixels are required in this time, then value (M-6) should be programmed into the HCSR. This is normally an 11-bit register, with bit 13 the least significant. Bits 11 and 12 must be zero except in the High Resolution mode.

In this mode, where each 24 MHz pixel is further divided into four pixels, the cursor sub-position can be defined by programming bits 11 and 12 of the HCSR, which will move the cursor position within the 24 MHz pixel. Refer to the High Resolution Mode section.

Note that only the cursor start positon needs to be defined, as the cursor is automatically disabled after 32 pixels. If a cursor smaller than this is required, then the remaining bits in the cursor pattern should be programmed to logical color 00 (transparent). Figure 16 shows the data bus encoding scheme.

Horizontal Interlace Register (HIR): Address 9CH

This register must be programmed if an interlaced sync display is required. Otherwise, it may be ignored. If value L is written into the HCR, the value (L+1)/2 should be written into the HIR. [L is odd.] This is a 10-bit register with bit 14 the least significant. Data bus encoding is shown in Figure 17.

Vertical Cycle Register (VCR): Address A0H

This register defines the period, in units of a raster, of the vertical soan, i.e., display time + flyback time. If N rasters

FIGURE 17. HORIZONTAL INTERLACE REGISTER DATA BUS ENCODING

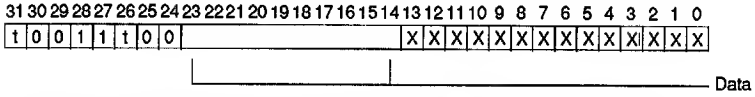


FIGURE 18. VERTICAL CYCLE REGISTER DATA BUS ENCODING

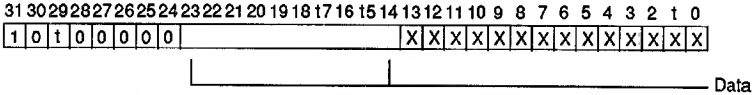


FIGURE 19. VERTICAL SYNC WIDTH REGISTER DATA BUS ENCODING

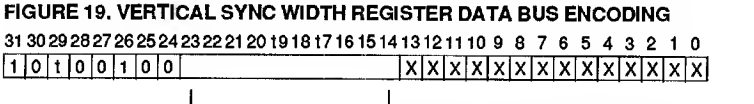


FIGURE 20. VERTICAL BORDER START REGISTER DATA BUS ENCODING

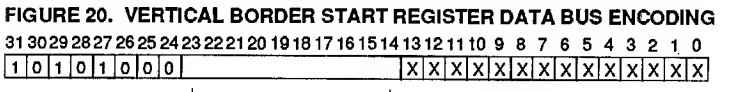


FIGURE 21. VERTICAL DISPLAY START REGISTER DATA BUS ENCODING

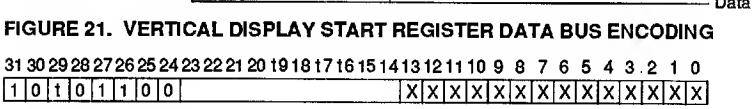


FIGURE 22. VERTICAL DISPLAY END REGISTER DATA BUS ENCODING

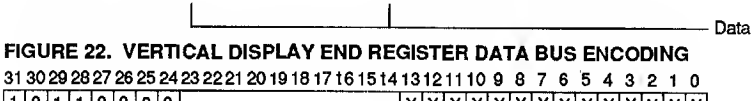
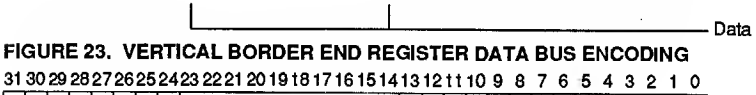


FIGURE 23. VERTICAL BORDER END REGISTER DATA BUS ENCODING



are required in a complete frame, then value (N-1) should be programmed into the VCR. If interlaced display is used, (N-3)/2 must be programmed into the VCR. [N is odd.] Here N is still the number of rasters in a complete frame, not a field. This is a 10-bit register, with bit 14 the least significant. Figure 18 shows the data bus encoding scheme.

Vertical Sync Width Register (VSWR): Address A4H

This register defines the width, in units of a raster, of the –V/CS pulse. If N rasters are required in the vertical sync pulse, then value (N-1) should be programmed into the VSWR. The minimum value allowed for N is 1. This is a 10-bit register, with bit 14 the least significant. Data bus encoding is shown in Figure 19.

Vertical Border Start Register (VBSR): Address A8H

This register defines the time, in units of a raster, from the start of the vertical sync pulse to the start of the border display. If N rasters are required in this time, then value (N-1) should be programmed into the VBSR. If no border is required, then this register must still be programmed, in this case to the same value as the VDSR. This is a 10-bit register, with bit 14 the least significant. Figure 20 shows the data bus encoding.

Vertical Display Start Register (VDSR): Address ACH

This register defines the time, in units of a raster, from the start of the vertical sync pulse to the start of the video display. If N rasters are required in this time, then value (N-1) should be programmed in the VDSR. This is a 10-bit register, with bit 14 the least significant. The data bus encoding is shown in Figure 21.

Vertical Display End Register (VDER): Address B0H

This register defines the time, in units of a raster, from the start of the vertical sync pulse to the end of the video display (i.e., the first raster on which the display is not present). If N rasters are required in this time, then the value (N-1) should be programmed into the VDER. This is a 10-bit register, with bit 14 the least significant. Figure 22 illustrates the data bus encoding.

Vertical Border End Register (VBER): Address B4H

This register defines the time, in units of a raster, from the start of the vertical sync pulse to the end of the border display (i.e., the first raster on which the border is not present). If N rasters are required in this time, then the value (N-1) should be programmed into the VBER. If no border is required, then this register must be programmed to the same value as the VDER. This is a 10-bit register, with bit 14 the least significant. Data bus encoding for this register is shown in Figure 23.

Vertical Cursor Start Register (VCSR): Address B8H

This register defines the time, in units of a raster, from the start of the vertical sync pulse to the start of the cursor display. If N rasters are required in this time, then value (N-1) should be programmed into the VCSR. This is a 10-bit register, with bit 14 being the least significant. Figure 24 shows the data bus encoding for this register.

Vertical Cursor End Register (VCER): Address BCH

This register defines the time, in units of a raster, from the start of the vertical sync pulse to the end of the cursor display (i.e., the first raster on which the cursor is not present). If N rasters are required in this time, then value (N-1) should be programmed into the VCER. This is a 10-bit register, with bit 14 the least significant. Data bus encoding is shown in Figure 25.

5

FIGURE 24. VERTICAL CURSOR START REGISTER DATA BUS ENCODING

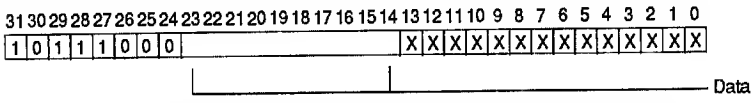


FIGURE 25. VERTICAL CURSOR END REGISTER DATA BUS ENCODING

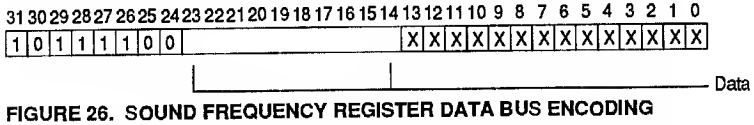
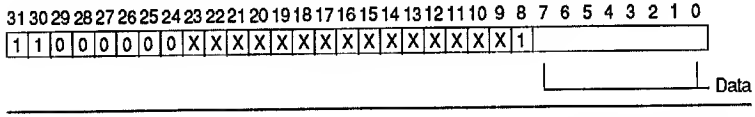


FIGURE 26. SOUND FREQUENCY REGISTER DATA BUS ENCODING



Sound Frequency Register (SFR): Address C0H

This register defines the byte sample rate of the sound data. It is defined in units of 1 µs. If a sample period of N µs is required, then (N-1) should be programmed into the SFR. N may take any value between three and 256. This is a 9-bit register with bit 0 the least significant. Bit 8 in the SFR is used as a test bit, and should always be set to one. When this bit is set to zero, all the internal timing signals are cleared. Figure 26 shows the data bus encoding.

Control Register (CR): Address E0H

This register contains the operating mode controls: a total of 11 bits are defined, and three of these are for test purposes only. Note that bit 8 in the SFR must also be set before the device can operate correctly.

The two bit-pairs for the pixel rate and the bits per pixel selects are defined in Figure 27. The bit-pair to define the point at which the DMA request flag is set is further explained in the Restriction On Parameters section.

To select interlaced sync displays, D[6] in this register must be set as well as setting the correct values in the vertical and horizontal timing registers.

The –V/CS pin on the device can be programmed to output either the vertical sync pulse or the composite sync pulse which is the XNOR of vertical and horizontal sync. Selection is made by D[7].

The remaining 3-bits are for testing the device and are of little interest to the user, but their action is as follows.

In test mode zero (D[14] high, D[15] low), the upper 5-bits of the horizontal counter are clocked by a derivative of the pixel clock.

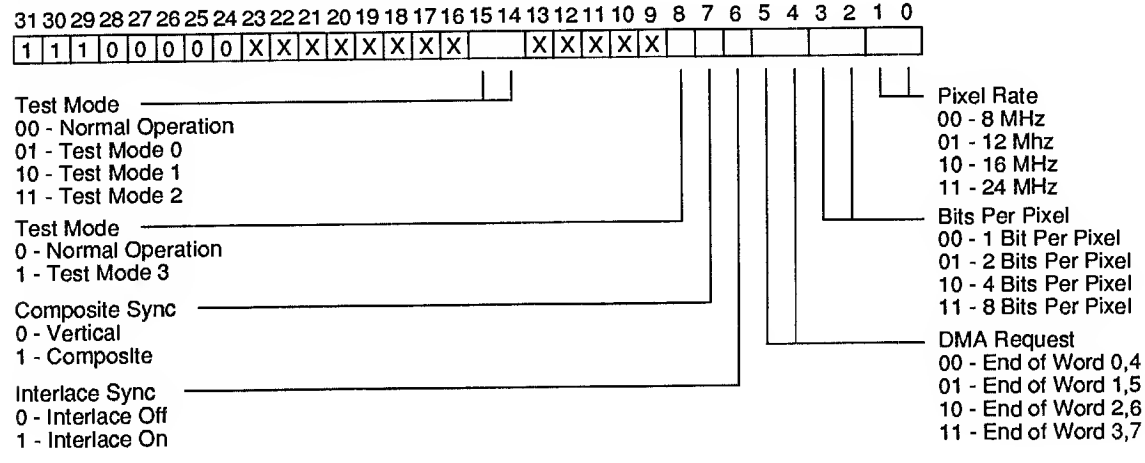
In test mode one (D[14] low, D[15] high) the lower 5-bits of the vertical counter are clocked by a derivative of the pixel clock.

In test mode two (D[14] high, D[15] high), the upper 5-bits of the vertical counter are clocked by a derivative of the pixel clock.

In test mode three (D[8] set), the pin –L/R outputs a signal which is eight times the frequency of the sound byte sampling clock, and the pins SD3-SD0 output the inverse of the data which is fed to the green DAC [NSEL low] or the blue DAC [NSEL high].

Note that the device cannot function properly in test modes zero, one, and two, but test mode three has no effect on the normal operation.

FIGURE 27. CONTROL REGISTER DATA BUS ENCODING



USING THE VIDC

The DMA Interface

The VIDC has three FIFOs into which DMA data is written. The sound FIFO is four 32-bit words deep, and works independently from the the other two FIFOs. The video FIFO is eight 32-bit words deep, and the cursor FIFO is again four 32-bit words deep.

Sound FIFO

Each word of data is strobed into the FIFO on the rising edge of –SDAK. Data is read out of the FIFO into a byte wide latch which then drives the DAC. When the last byte in the FIFO is read into the latch, the signal –SDRQ is driven low, requesting another 16 bytes of data. The signal –SDRQ is driven high when the first –SDAK is received.

The time available to service this data request is dependent on the sound data rate. The minimum value of the SFR is three, which defines a byte-rate of 3 µs. Therefore, the first word must be loaded into the FIFO less than 3 µs after the –SDRQ signal is generated.

Cursor FIFO

The cursor FIFO contains 16 bytes of data, which is enough for two rasters of cursor display. When the VIDC is programmed to display a cursor, –VDRQ is driven low at the same time as –HSYC goes low on the first raster on which the cursor is to appear. Data is loaded into the FIFO on the rising edge of –VDAK. The load cycle must be complete before the –HSYC pulse has ended.

–VDRQ is driven high again when the first –VDAK is received. The cursor may be any number of rasters high, and the cursor FIFO requests data during the –HSYC of every alternate raster on which it is displayed.

Video FIFO

The video FIFO is eight 32-bit words deep, and it is arranged as a circluar buffer. Data must always be loaded into it in blocks of four words, and this FIFO shares the same –VDRQ and –VDAK signals as the cursor FIFO.

To accommodate the vastly different rates at which video data is required in the different modes, and to accommodate different DRAM speeds, the point at which more data is requested can be varied. This is done by bits 4 and 5 in the Control Register.

During the vertical sync pulse, the FIFO is cleared, and the signal –VDRQ is high. After the –HSYC pulse of the first displayed raster, –VDRQ is driven low. Eight words must now be written into the FIFO by driving –VDAK low eight times. This fills the FIFO. –VDRQ is set high again when the fifth –VDAK is received.

Thereafter, the –VDRQ signal is set low whenever the FIFO empties to the point predetermined by bits 4 and 5 in the Control Register. The –VDRQ signal is normally set high when the first –VDAK signal is received. However, if the data request is not serviced quickly, and the FIFO has emptied to the point where another four words have been read out when the first new data word arrives, then the –VDRQ signal will stay low, requesting another four words of data.

The Video DMA Interface

As noted above, the cursor and video FIFOs share the same DMA interface signals. Normally, a –VDRQ low during the –HSYC pulse is a request for cursor data, and –VDRQ low at any other times is a request for video data. Figure 28 shows the relationships graphically.

However, often a video request happens just before the end of a raster requesting data for the next raster. The load cycle for this video request is allowed to overlap the –HSYC pulse, even if a cursor request happens during the –HSYC pulse. Note that in this case the –VDRQ signal may not be driven high between these two cycles. The first cycle must be video data and the second cycle must be cursor data. The cursor load cycle must still be complete before the end of the –HSYC pulse. This is shown in Figure 29.

Figure 30 shows the situation where a cursor is displayed on the first raster of the frame. Note the double video load cycle. The cursor load cycle must not overlap the end of the –HSYC pulse (otherwise data will be loaded into the wrong FIFO), and the first word of video data must be present in the FIFO before the display starts.

Restrictions On Parameters

It is clear from the above that certain restrictions must be applied to the screen parameters, most of which are system dependent. The following paragraphs assume the VIDC is being used in a system with the processor and MEMC and two/one clock page-mode DRAM memory. In this system DRAM cycles consist of an N-cycle (two RCLK clocks) followed by up to three sequential S-cycles (one RCLK clock).

FIGURE 28. VIDEO AND CURSOR DMA OPERATION

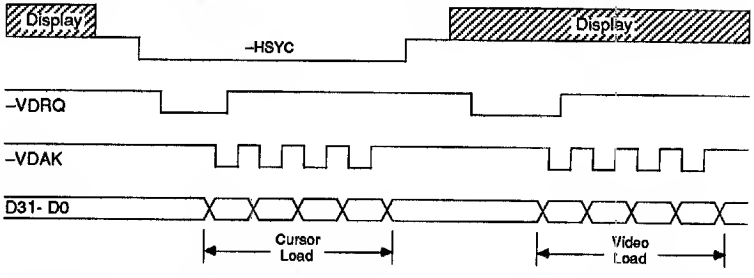


FIGURE 29. VIDEO DMA OVERLAPPING –HSYC PULSE

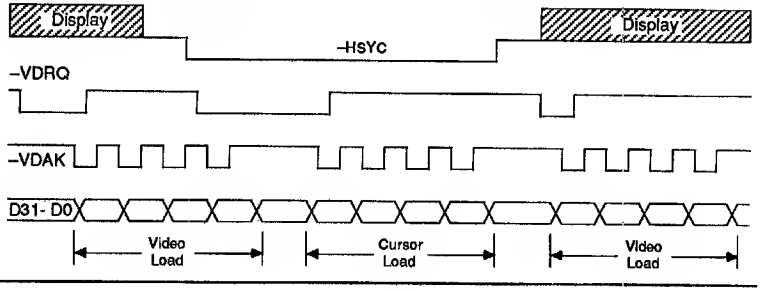
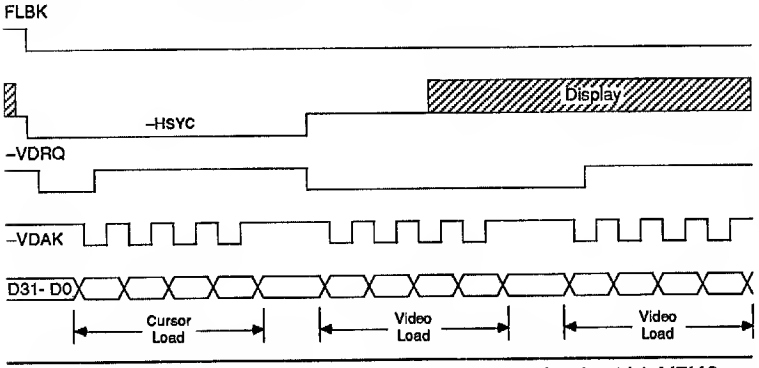


FIGURE 30. CURSOR DMA AT THE BEGINNING OF A FRAME



Hence, a VIDC FIFO load cycle consists of 1N + 3S requiring five RCLK clocks (417 ns at 12 MHz).

FIFO Request Pointer Values (Control Register Bits [4:5])

The video FIFO is a circular buffer, although the core is asynchronous, with a ripple-through time of 150 ns from the top to the bottom. Data is loaded in blocks of four words, and is read out in bytes, starting with byte 0 of word zero and so on. –VDRQ can be set low half way through reading the last byte of any of word 0-3 (and correspondingly 7-4) according to bits 5-4 in the Control Register. In the high resolution video modes where the bytes are being consumed quickly, the request signal must be set at an earlier point than in the low resolution modes. Selections are defined in Table 3.

The request signal –VDRQ should be brought low as soon as the FIFO can accept the four words of data when they arrive. The minimum time between setting the request and receiving the last word of data is 187 ns + 625 ns = 812 ns (at 8 MHz). [The 187 ns figure is the minimum time in which MEMC can start a DMA cycle.] If the FIFO is full at the start, then it will have four words spare 150 ns after the start of word 4. [150 ns is the ripple-through time of the FIFO.] Hence, the request should be made at the first opportunity after (812 ns - 150 ns = 652 ns) before the start of word four. The request can be made halfway through the last byte of any of words 0-3 by programming the Control Register.

TABLE 3. FIFO POINTER SETTINGS

Control Register		–VDRQ Set At End Of Words
Bit 5	Bit 4	
0	0	0, 4
0	1	1, 5
1	0	2, 6
1	1	3, 7

Depending on the video mode in use, data can be read from the FIFO at 1.5, 2, 3, 4, 6, 8, 12, or 16 Mbytes/second.

Figure 31 shows the case for the 16 Mbytes/second mode. The request must be set at the end of words one and five.

Figure 32 shows the case for the 12 Mbytes/second mode. The request must be set at the end of words two and six.

Figure 33 shows 8 Mbytes per second mode. Again, the request must be set at the end of words two and six.

In all other modes, the request should be set at the end of words three and seven.

Horizontal Sync Pulse Width

The –HSYC pulse width must be long enough to allow a complete load of the cursor FIFO. This is made up as follows:

2*[N+3S] (current + cursor cycles) + syncmax + 2*Tprop.

i.e. 2*625 + 312+ 100 = 1662 ns.

FIGURE 31. FIFO OPERATING AT 16 MBYTES PER SECOND

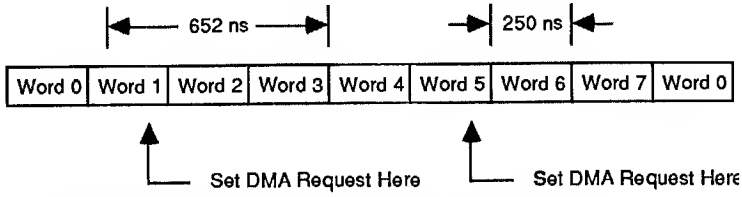


FIGURE 32. FIFO OPERATING AT 12 MBYTES PER SECOND

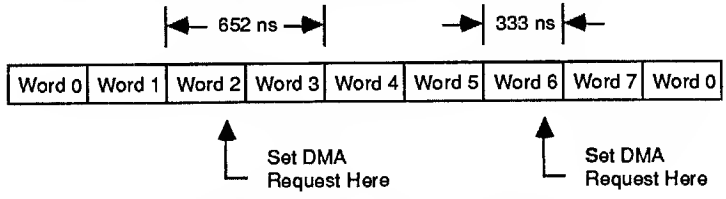
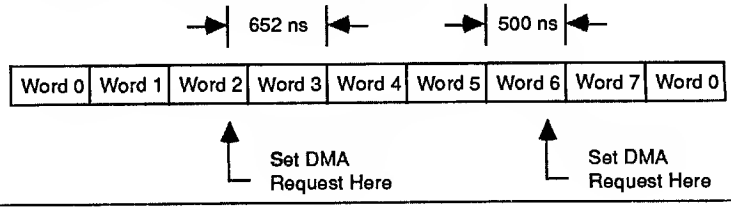


FIGURE 33. FIFO OPERATING AT 8 MBYTES PER SECOND



Syncmax is the maximum time MEMC can take to recognize the DMA request. Tprop is the time taken for the –VDRQ signal to reach MEMC, or the time taken for –VDAK to reach VIDC.

The pulse must also be long enough to allow the processor to write to the DMA address generator (DAG) in the MEMC to reset the screen pointer. This may be as follows:

3*[N+3S] (current + cursor + sound cycles) + DAG write.

i.e. 3*625 + 25D = 2125 ns. Since both these parameters must be met, this larger value must therefore be used.

Horizontal Front Porch Width
The front porch may be zero length. The total time from the end of display to the end of the –HSYC pulse must be more than 1912 ns. As the –HSYC pulse width has to be at least 2125 ns, this does not impose a restriction on the value of the back porch.

Horizontal Front Porch Width
The back porch must be long enough to allow the load of at least one word into the video FIFO before the data is read out again. This is important at the start of the frame because data is required in the bottom of the FIFO at least four pixel-times before the start of display, due to the pipeline delays. Hence the back porch must be greater than:

N+3S+N (current cycle + video N cycle) + syncmax + 2*Tprop + FIFO-ripple + 4 pixels.

i.e. 25D + 375 + 25D + 312 + 1DD + 15D + 4*83 = 1769 ns for 12 MHz displays.

or 25D + 375 + 25D + 312 + 10D + 150 + 4*125 = 1937 ns for 8 MHz displays.

Vertical Sync Pulse and Porch Width
There are no restricitons on the values of the vertical front porch, back porch, or sync width. The Vertical Sync Width Register (VSWR) may be programmed to a value of D which gives a vertical sync width of one raster.

Horizontal Display Width
The number of bits in the pixels of each raster must be a multiple of 128.

Border
The border cannot be disabled. If no border is required, then it should be programmed to start and finish in exactly the same place as the display (both vertically and horizontally).

Cursor Position
The cursor should not be programmed to be outside the display area vertically, but it may be programmed to start or end outside the display area horizontally. Note that the cursor will not be displayed outside the border area either vertically or horizontally.

TABLE 4. SCREEN MODE SUPPORT

Pixel Rate	Bits/Pixel	FIFO Data Rate	Pixel Rate	Bits/Pixel	FIFO Data Rate
24 MHz	8	Not Supported	12 MHz	8	12 Mbytes / Second
	4	12 Mbytes / Second		4	6 Mbytes / Second
	2	6 Mbytes / Second		2	3 Mbytes / Second
	1	3 Mbytes / Second		1	1.5 Mbytes / Second
16 MHz	8	16 Mbytes / Second	8 MHz	8	8 Mbytes / Second
	4	8 Mbytes / Second		4	4 Mbytes / Second
	2	4 Mbytes / Second		2	2 Mbytes / Second
	1	2 Mbytes / Second		1	Not Supported

DISPLAY FORMATS

Screen Modes
Fourteen of the possible 16 display modes are supported (See Table 4).

Data Display
Pixels are displayed starting at the top left hand corner of the screen, with the least significant end of the first word loaded into the FIFO. In 8-bits per pixel mode, bits 0-7 of word zero are the first displayed pixel. In 4-bits per pixel mode, bits D-3 of word zero are the first displayed pixel. In 2-bits per pixel mode, bits 0-1 of word zero are the first displayed pixel. In 1-bit per pixel mode, bit zero of word zero is the first displayed pixel.

Logical Data Fields
In 1-bit per pixel mode, the data field selects the palette at location zero or one. The other 14 locations need not be programmed. In 2-bits per pixel mode, the data field addresses the palette at locations zero through three. The other 12 locations need not be programmed. In 4-bits/pixel mode, the data field addresses the palette at all 16 locations. In 8-bits per pixel mode, the least significant 4-bits drive the palete as in 4-bits per pixel mode, and the most significant four bits drive the most significant bits of the RGB DACs directly.

Physical Data Fields
In 1-, 2-, and 4- bits per pixel mode, the physical data field is shown in Figure 34. In 8-bits per pixel mode, the physical data field is shown in Figure 35. The Dn bits come from the palette field and the Ln bits come from the logical field.

Cursor Format
The cursor, in all video modes, is defined to be 32 pixels wide and any number of rasters high. Any pixel may be defined as being transparent, enabling cursors of any shape to be constructed within the 32 pixel horizontal limit. It is always 2-bits per pixel, with bits zero, one in the first word to be loaded into the cursor FIFO representing the first pixel, etc. The logical cursor pixel bit-pairs are defined in Table 5.

The cursor physical field is exactly as the video physical field in 1-, 2-, or 4- bits per pixel modes.

FIGURE 34. PHYSICAL COLOR FIELD DEFINITIONS

12	11	10	9	8	7	6	5	4	3	2	1	0
SUP	BLUE				GREEN				RED			
D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

FIGURE 35. PHYSICAL COLOR FIELD DEFINITIONS FOR 8 BITS PER PIXEL

12	11	10	9	8	7	6	5	4	3	2	1	0
SUP	BLUE				GREEN				RED			
D12	L7	D10	D9	D8	L6	L5	D5	D4	L4	D2	D1	D0

TABLE 5. CURSOR LOGICAL COLORS

Cursor Bit		
MSB	LSB	Color
0	0	Transparent
0	1	Logical Color 1
1	0	Logical Color 2
1	1	Logical Color 3

Border Field

The border physical field is exactly as the video physical field in 1-, 2-, and 4-bits per pixel modes.

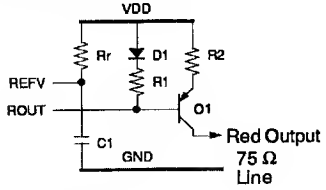
Controlling the Screen On / Off

The simplest method of turning the screen off is to program the Vertical Display End Register (VDER) to be less than the VDSR. This will not generate any video requests, but will display the border color over the whole screen. The border can be turned off either by programming it to physical color black, or by programming the VBSR to be greater than the VBER. Doing the latter will also disable the cursor, though cursor data requests will still be generated. Turning the screen back on should only be done during vertical flyback.

Cursor On / Off

The cursor should be turned off by setting the VCER to be less than VCSR. [Value 0 is suggested.] This will also disable cursor data requests. Turning the cursor on, and moving it around should only be done during vertical flyback to prevent flash.

FIGURE 36. EXAMPLE VIDEO OUTPUT CIRCUIT



Suggested Component Values
Rr – 10 kΩ
R1 – 330 Ω
R2 – 68 Ω
D1 should have similar characteristics to the emitter-base junction of Q1

Writing to the Palettes and Other Registers

The palettes may be programmed reliably at any time, but are best programmed during vertical flyback. Changing the values of other registers should only be done during vertical flyback. The signal FLBK is set high from the start of the first raster after the end of display (though it may still be border), until the start of the first raster which is display.

Video DAC Outputs

The video DAC outputs are in the form of current sinks. Each DAC has a resolution of 4-bits, giving a linear transfer characteristic with 16 values.

A digital input value of four zeros gives zero current sink, and a digital input value of four ones gives the maximum current sink. The magnitude of the output is a function of the video reference input current, with the maximum current sink being 15 times the reference input current.

High Resolution Modes

The four bits of digital data which normally drive the red DAC are available to the user on pins –VED3 through –VED0. This pixel rate bit-stream can be externally serialized to a single bit-stream of four times the VIDC pixel rate. With the VIDC operating at 24 MHz, four bits per pixel mode, 96 MHz bit rates are generated giving very high resolution monochrome displays. Alternatively, with an external DAC, 48 MHz grey-level displays are possible.

Refering to the block diagram, it will be noted that the data passes through the High Res. Shifter block before reaching the pins –VED3 - –VED0. This block enables the cursor to be positioned to any (96 MHz) pixel. Note that this block also inverts the data from the red DAC.

When used in this mode, the VIDC must be programmed to a different set of values. But note that the "normal" analog modes of VIDC are still available simply by reprogramming: the addition of the shifter hardware will not affect the other modes, and the sound system is totally independent from this.

(1) Four-bits per pixel should always be selected.

(2) The programmed VIDC pixel rate is one quarter of the external pixel rate. The vertical timing parameters are unaffected by this as they are defined in units of a raster, but the horizontal timing parameters which are defined in units of two (24 MHz) pixels can only be programmed in units of eight (96 MHz) pixels. There are now four times as many pixels on a line as are actually programmed. For example, if a display of 1024 * 1024 is required, the VIDC should be programmed to generate a display of 256 (horizontal) by 1024 (vertical).

(3) All 16 locations of the video palette should be programmed to a 1:1 logical to physical mapping. Only D[0:3] (red DAC values) need to be programmed, as D[4:11] are ignored. The supremecy bit (D[12]) may be used if required.

(4) D[4,5] in the Border Color Register must be set to zero. D[0:3] and D[12] may also be progremmed if a border is required.

(5) The cursor palette should be programmed to the following values:
cursor color 1 : 10H
cursor color 2 : 20H
cursor color 3 : 30H
Supremacy may also be used.

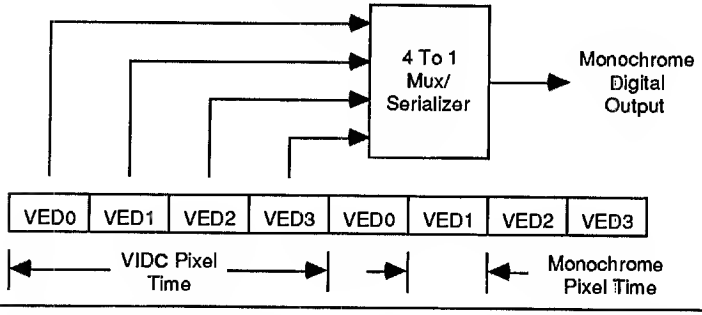
Then the 2-bits which define each cursor pixel are defined in Table 6.

Note that the cursor can only be defined horizontally in units of four (96 MHz) pixels, though it can be positioned anywhere on the screen to within one (96 MHz) pixel. See the section on Horizontal Cursor Start Register for more detail. The hardware should be erranged so that –VED0 is the first bit to be serialized.

TABLE 6. CURSOR COLOR IN HIGH-RESOLUTION MODE

Cursor Bits		
MSB	LSB	Definition
0	0	Trensparent
0	1	Cursor Black
1	0	Do Not Use
1	1	Cursor White

FIGURE 37. HIGH RESOLUTION PIXEL GENERATOR



External Synchronization and Mixing
The VIDC has two signals associated with external synchronization applications: SUP and SINK. SUP is an output which can be used to control an external multiplexer for mixing the VIDC output with that from an external source. All video and cursor logical colors from the palettes and the border color can control SUP. When D[12] in any of the above registers is set and that color is being displayed, SUP is driven low. The output is pipelined and is synchronous with the DAC outputs and the –VED3 - –VED0 signals.

The signal SINK is an input which when driven high resets the vertical counters to the first raster. If an interlaced sync display is being generated, then SINK will reset the counters to the first raster of the odd field. The pulse applied to this pin must be shorter than a raster time. The horizontal counters are not affected by this signal. The horizontal synchronization must be done by phase-locking (or in simple applications, by interrupting) the input clock CKIN. Remember that the sound system is also driven from a derivative of CKIN.

Composite Sync
According to the setting of D[7] in the Control Register, the –V/CS can output a composite sync pulse. This is synthesized from the XNOR of vertical and horizontal syncs as shown in Figure 38.

Interlaced Displays
The VIDC can generate an interlaced sync display. An example of interlace timing is shown in Figure 39. Normally the video data in each field is the same. The VIDC Vertical Cycle Register is set to a value (N-3)/2, where N is the total number of rasters in a frame. There are N/2 raster in the even and odd fields. On raster (N+1)/2, the vertical sync pulse is output and the cycle repeats, but this is now the odd field, so the vertical sync pulse is delayed by half a raster time as defined by the value in the HIR. On the first raster in the odd field which is not vertical sync, a dummy raster is inserted. This makes the odd field N/2 rasters long as well.

FIGURE 38. COMPOSITE SYNC GENERATION

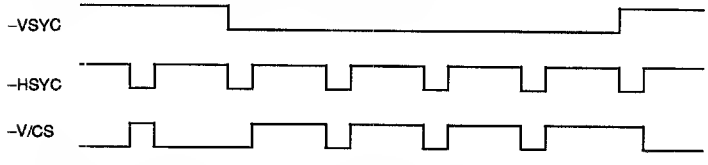


FIGURE 39. INTERLACE DISPLAY TIMING GENERATION

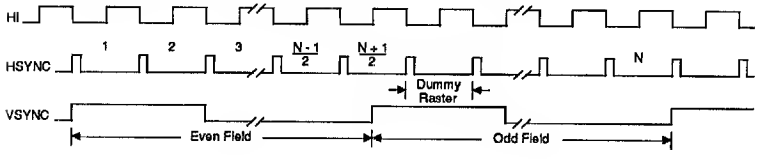
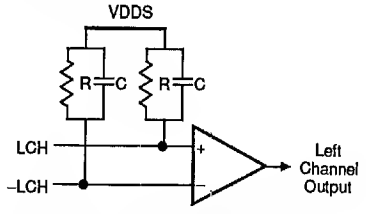


FIGURE 40. SOUND OUTPUT CIRCUIT



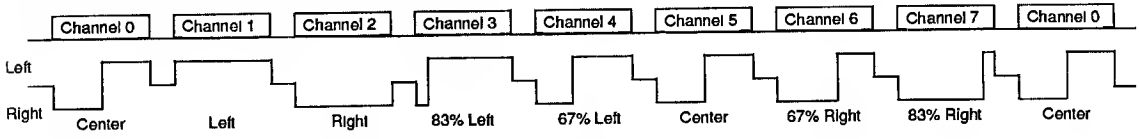
Sound System
The sound system consists of a four word FIFO and byte wide latch which drive a 7-bit exponential DAC. The eighth bit steers the DAC output to one of two pairs of output pins, one pair designated "+", and the other pair designated "-". The sound signal is generated externally by integrating and then subtracting these two pairs of signals. An example circuit is shown in Figure 40. The integration is performed by the capacitor C.

Stereo image is synthesized by time-division multiplexing the output between the "left" and "right" pair of output signals as shown in Figure 41. The first quarter of each sample is muted to allow for DAC settling and deglitching. The stereo image is specified for each channel by programming the corresponding Stereo Image Register.

The system can operate in 1, 2, 4, or 8 channel modes. In 8 channel mode, the channels are sampled sequentially, starting with the first byte of data, which is channel 0; the second byte of data is channel 1 and so on. The external integrating time constant must be long enough to integrate over a complete sample cycle. In 4 channel mode, the fifth byte to be sampled is again channel 0, so Stereo Image Register 4 must be programmed to the same value as Stereo Image Register 0, and so on. In 2 channel mode, Stereo Image Registers 0, 2, 4, and 6 correspond to channel 0 and Stereo Image Registers 1, 3, 5, and 7 correspond to channel 1. In single channel mode, all eight Stereo Image Registers should contain the same value.

5

FIGURE 41. STEREO IMAGE SYNTHESIS



The sample rate is selectable by the SFR in units of 1 µs, with a minimum value of 3 µs. Clearly, in eight channel mode the bytes for each channel are sampled with one-eighth of the frequency of single channel mode for a given value in the SFR.

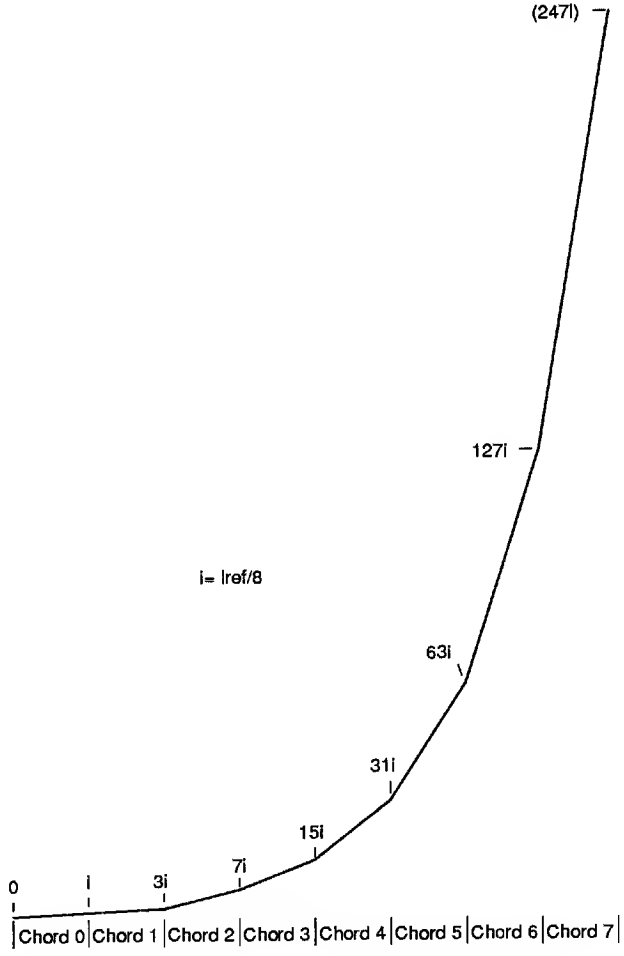
The DAC transfer characteristic consists of eight linear segments (chords). Each chord consists of 16 steps, and the step size in one chord is twice the step in the preceding chord. This gives an approximation to the "µ255 law." The sound data field format is shown in Figure 42.

The outputs are in the form of current sinks. The magnitude of the output is a function of the sound reference input current. The reference current is equal to the step size in the highest chord, which is 8i in Figure 43.

FIGURE 42. SOUND DATA FIELD FORMAT

D7	D6	D5	D4	D3	D2	D1	D0
Chord Select			Point On Chord				Sign

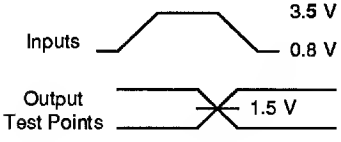
FIGURE 43. SOUND DAC OUTPUT



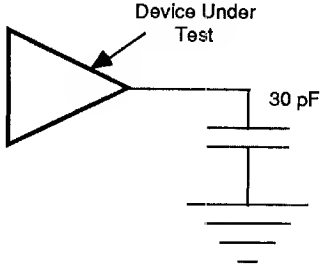
TIMING CHARACTERISTICS: TA = 0°C to +70°C, VDD = +5 V ±5%

Symbol	Parameter		Min	Typ	Max	Units	Conditions
t1	CKIN High		10	–	–	ns	
t2	CKIN Low		13	–	–	ns	
t3	CKIN Frequency		–	–	33	MHz	
t4	Data Setup Time to –VDAK, –SDAK		9	–	–	ns	
t5	Data Hold Time to –VDAK, –SDAK		9	–	–	ns	
t6	–VDAK, –SDAK Pulse Width		15	–	–	ns	
t7	Data Setup Time to –VIDW		10	–	–	ns	See Note 1
t8	Data Hold Time to –VIDW		20	–	–	ns	
t9	–VIDW Pulse Width		20	–	–	ns	
t10	CKIN to –SD3 - –SD0 Delay		–	70	–	ns	See Notes 2, 3
t11	CKIN to –VED3 - –VED0, –SUP Delay		–	70	–	ns	See Note 2
t12	CKIN to –HSYC, –V/CS, FLBK		–	75	–	ns	
t13	CKIN to –HI Delay		–	75	–	ns	
t14	CKIN to ROUT, GOUT, BOUT		–	30	–	ns	See Note 2
t15	Analog Output Rise/Fall Time		–	10	–	ns	See Note 4
t16	NIBSEL to –SD3 - –SD0		–	50	–	ns	
t17	Acknowledge To Request Delay	–SDAK to –SDRQ	–	50	–	ns	See Note 5
		–VDAK to –VDRQ	–	50	–	ns	See Note 5

A.C. TEST WAVEFORMS



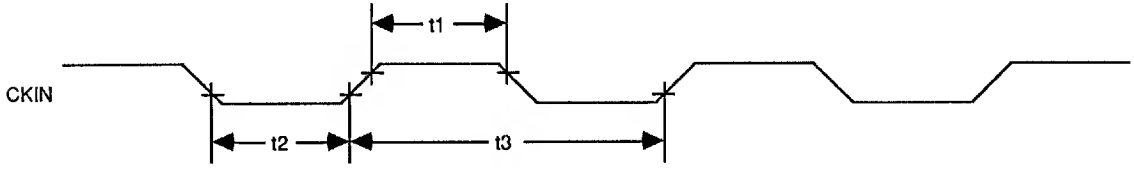
A.C. TEST LOAD CIRCUIT



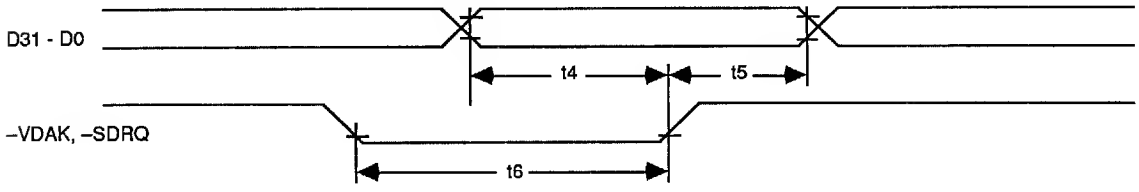
Notes:
1. The data must be setup before –VIDW goes active (low) because the data also contains the register address.
2. For pixel rates of 12 and 24 MHz, the outputs are referenced to the rising edge of CKIN. For pixel rates of 8 and 16MHz, the outputs are alternately referenced to either edge of CKIN.
3. The –SD3 - –SD0 signals are output one pixel time before the corresponding –VED3 - –VED0 due to pipeline differences.
4. Assumes a 5 pF external load.
5. –VDRQ or –SDRQ are cleared by the first –VDAK or –SDAK respectively, as long as no request is pending.

TIMING DIAGRAMS

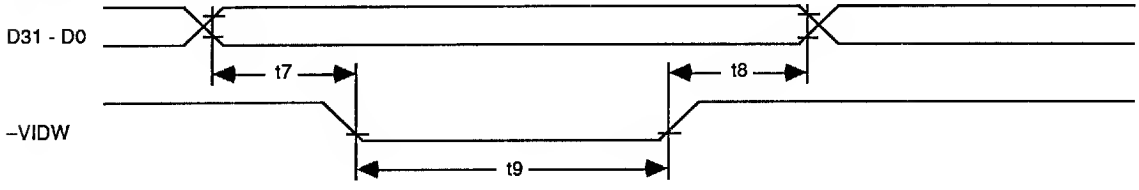
INPUT CLOCK



DMA WRITE CYCLES

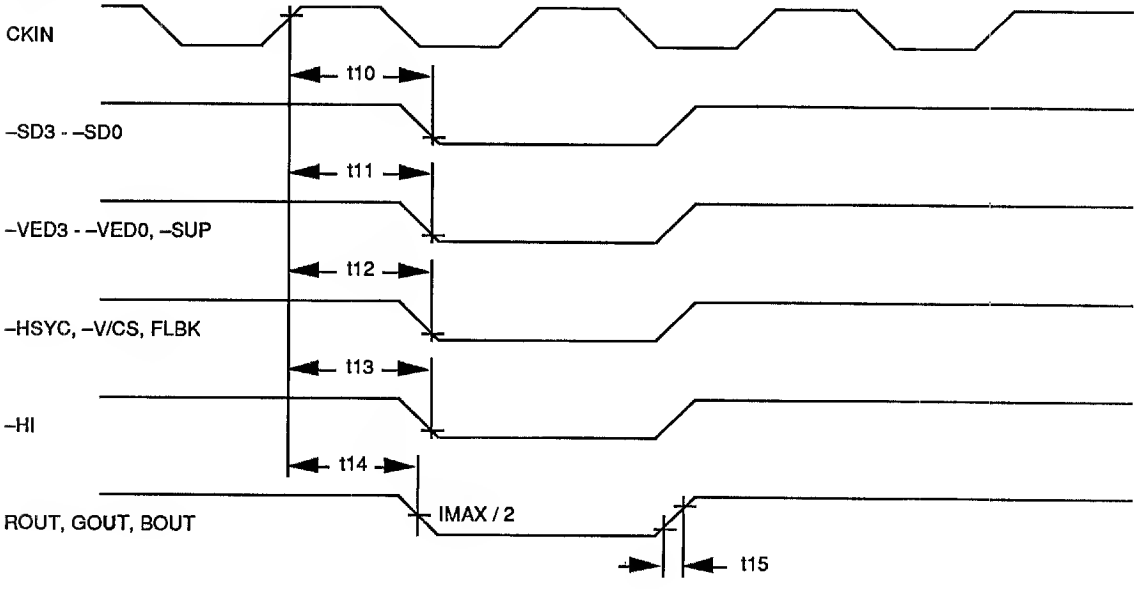


DMA WRITE CYCLES

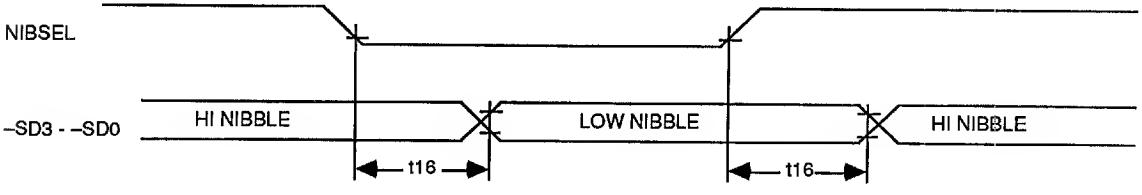


TIMING DIAGRAMS

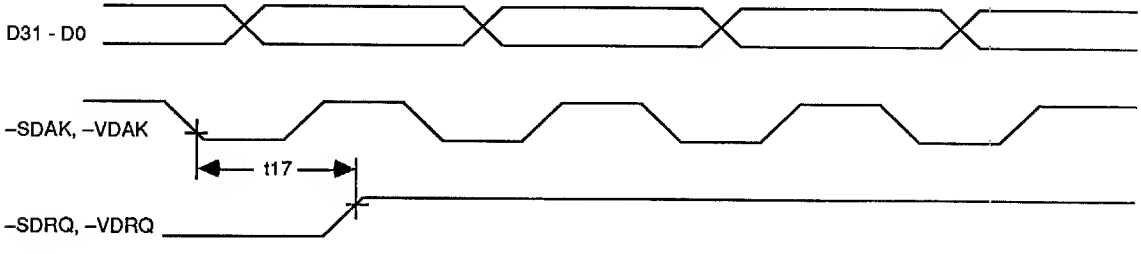
CLOCK OUTPUTS



NIBSEL TIMING



DMA ACKNOWLEDGE CYCLE



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.

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

Symbol	Parameter		Min	Typ	Max	Units	Conditions
VOH	Output High Voltage		VDD - 0.75	–	VDD	V	IOH = 10.0 mA
VOL	Output Low Voltage		–	–	0.4	V	IOL = - 3.0 mA
VIH	Input High Voltage		2.4	–	–	V	
VIHV	Input High, VIDW		3.5	–	–	V	
VIL	Input Low Voltage		0.0	–	0.8	V	
ILI	Input Leakage Current		–	–	10	µA	VIN = 0 V - VDD
ILO	Output Leakage Current		–	–	10	µA	VOUT= 0 V - VDD
ICC	Operating Supply Current		–	–	20	mA	See Note 1
IOS	Output Short Circuit Current		–	25	–	mA	See Note 2
IVOUT	Output Current Video DACs		–	–	– 2.0	mA	
ISOUT	Output Current Sound DAC		–	–	– 2.0	mA	
ADVOL	RVDAC, RSDAC Voltage		–	VDD - 1.3	–	V	See Note 3
ILATCH	Input/Output Latchup Current		200	–	–	mA	See Note 4
VCOMP	Voltage Compliance	Video DACs	–	VDD - 1.7	–	V	IVOUT = – 2.0 mA
		Sound DAC	–	VDD - 1.5	–	V	ISOUT = – 2.0 mA
CCOMP	Current Compliance	Video DACs	–	4.5	–	mA	VOUT = VDD - 0.7
		Sound DAC	–	3	–	mA	VOUT = VDD - 0.7

Notes:
1. Measured at 24 MHz pixel rate. This value does not include any current output by the video DACs.
2. Not more than one output should be shorted to either rail and for no longer than one second.
3. This assumes 10 kΩ resistors to VDD.
4. This value is the current that inputs or outputs can tolerate before the chip latches up. This condition should be avoided to prevent device damage.

VLSI TECHNOLOGY, INC.

SECTION 6

VL86C410 RISC I/O CONTROLLER (IOC)

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

VL86C410

RISC I/O CONTROLLER (IOC)

FEATURES

- Power on reset control
- Four independent 16-bit programmable counters
  - –Two timers
  - –Two baud rate generators
- Bidirectional serial keyboard interface
- Six programmable bidirectional control pins
- Interrupt mask, request and status registers for –IRQ and –FIRQ
- 14 level triggered interrupt inputs
- Two edge triggered interrupt inputs
- Four programmable peripheral cycles
  - – Slow
  - – Medium
  - – Fast
  - – 2 MHz synchronous
- Seven external peripheral selects
- ARM/IO bus interface control
- Expansion bus buffer control

DESCRIPTION

The VL86C410 Input/Output Controller (IOC) is designed to interface to the VL86C010/VL86C110/VL86C310 chip set to provide a unified view of interrupts and peripherals within an Acorn RISC Machine (ARM) based computer. It controls an 8- to 32-bit I/O data bus to which on-board peripherals and any I/O expansions are connected. It provides a set of internal functions, which are accessed without wait states, and programmable speed access to external peripherals.

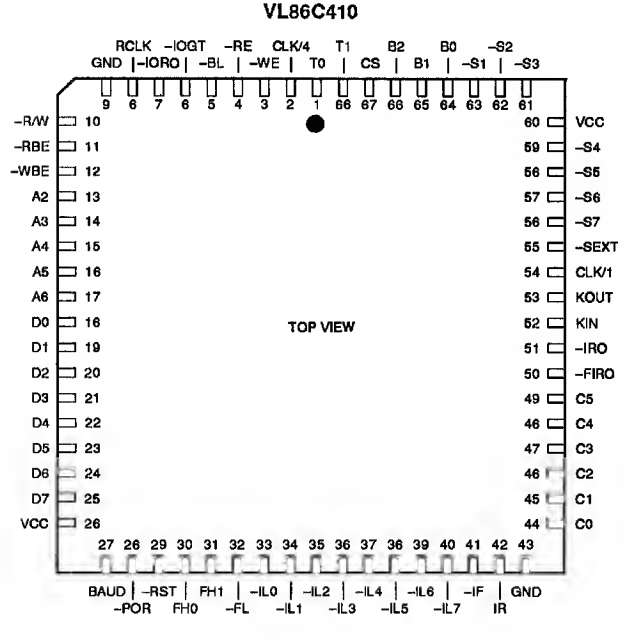
The VL86C410 provides system level I/O with six programmable control pins and a full-duplex, bidirectional serial keyboard interface. To support system timing requirements, the VL86C410 contains four independent programmable counters. Two of these counters are used as baud rate generators. One is dedicated to the keyboard and the other controls the BAUD output pin to generate a free-running clock. The other two counters can be used to generate system timing events.

The IOC serves as the interface between the very high speed RISC system bus and the slower I/O or expansion bus. The part provides all the buffer control required between the two buses. The VL86C410 supports an interruptable I/O cycle that allows the system to use slower, low-cost peripheral controllers such as the VL16C450 Asynchronous Communications Element and VL1772 Floppy Disk Controller without severe latency on the system bus.

Peripheral controllers are supported with 16 interrupt inputs (14 level sensitive and two edge-triggered), seven peripheral select outputs, and four programmable I/O cycle times.

PIN DIAGRAM

PLASTIC LEADED CHIP CARRIER (PLCC)

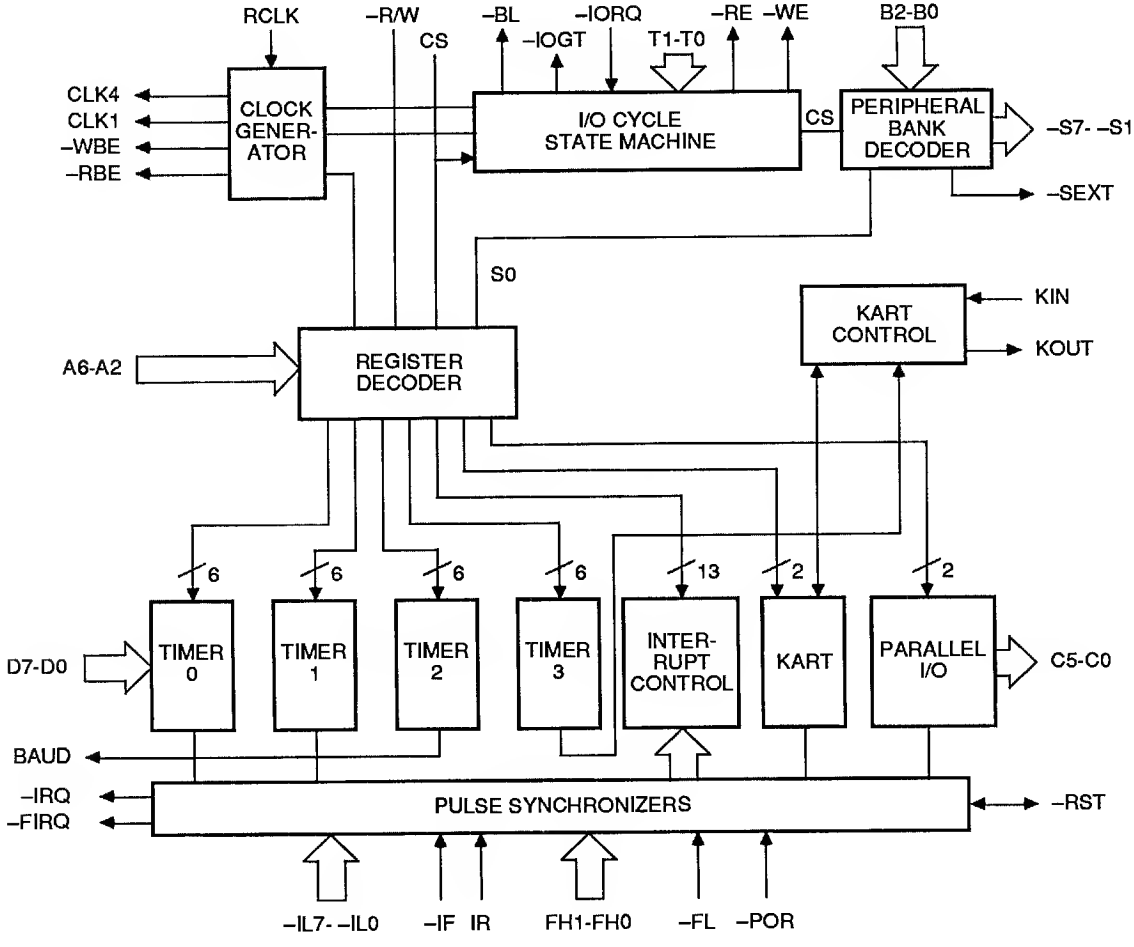


ORDER INFORMATION

Part Number	Clock Frequency	Package
VL86C410-08QC	8 MHz	Plastic Leaded Chip Carrier (PLCC)

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

BLOCK DIAGRAM



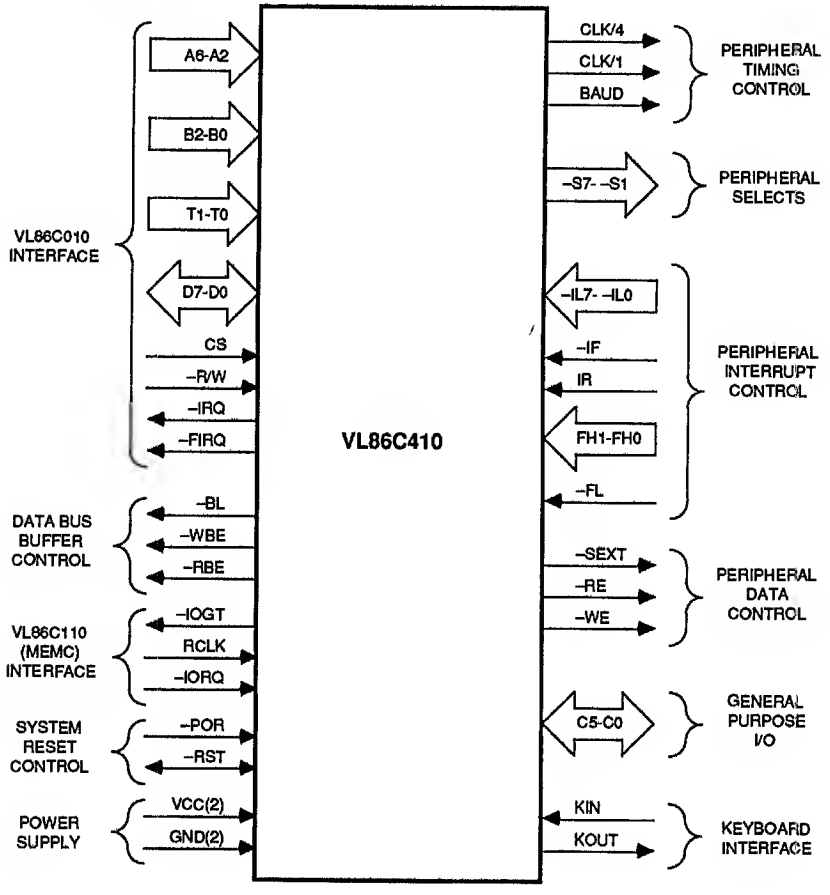
SIGNAL DESCRIPTIONS

Signal Name	Pin Number	Signal Description
RCLK	8	Reference Clock (CMOS Level input) - The main reference clock for bus transactions between different devices. RCLK is in phase with the Ø2 processor clock.
CLK/1	54	Buffered System Clock (CMOS level output) - Provides timing for external peripherals. This is RCLK buffered and inverted.
CLK/4	2	Peripheral Timing Clock (CMOS level output) - Is the RCLK input divided by four. This signal is used to generate timing for external peripheral controllers.
–BL	5	External Data Bus Latch Control (CMOS level output, open-drain) - This signal remains high, making the external data latches transparent, for internal register accesses and when CS is low. It is taken low during a write to latch the processor data, and during a read to latch the peripheral data.
D7-D0	25-18	Processor data bus (CMOS level input/output) - This is the 8-bit bidirectional three-state data bus used to transfer data to/from the attached processor. Normally, these lines would be connected to one byte of the 32-bit VL86C010 data bus through a transparent latch.
–IORQ	7	Input/Output Cycle Request (CMOS level input) - Determines whether the current cycle is a memory or an I/O reference and is usually connected to the corresponding signal of the VL86C110 Memory Controller. When asserted (low), the MEMC has detected an I/O address and the proper IOC should respond. When negated (high), the current cycle is a memory reference or internal processor cycle that does not require the attention of the I/O controllers.
–IOGT	6	Input/Output Cycle Grant (CMOS level output, open-drain) - Determines when the current I/O cycle will terminate. When asserted (low), the VL86C410 is signalling that the current I/O cycle will complete on the next falling edge of the RCLK. An I/O cycle completion is determined when –IOGT and –IORQ are both low on a rising edge of RCLK.
T1-T0	68, 1	Cycle Type 1, 0 (CMOS level inputs) - These signals indicate the type of I/O cycle to be performed; 0 = slow, 1 = medium, 2 = fast, 3 = synchronous.
–R/W	10	Not Read/Write (CMOS level input) - This signal normally is generated by the VL86C010 and determines cycle direction. The signal is low for read or high for write.
–SEXT	55	External Peripheral Select (CMOS level output) - This signal is used to enable an external data buffer. It is active for any IOC external peripheral access. This allows the IOC and any other –IORQ/–IOGT devices to be isolated from the external peripherals.
–S7- –S1	56-59, 61-63	Peripheral Selects (CMOS level outputs) - Indicate address and write data are valid on the bus. The B2-B0 lines are decoded to determine the active select output.
B2-B0	66-64	Bank Select Lines (CMOS level inputs) - Are decoded to determine the I/O access device. A zero value on these lines indicates an access to VL86C410 internal registers. Any other value will decode to one of the external peripheral select outputs.
CS	67	Chip Select (CMOS level input) - When high the IOC will perform either an internal register or external peripheral access cycle. Even when low, the VL86C410 controls the –RBE and –WBE outputs.
A6-A2	17-13	Register Select Lines (CMOS level inputs) - These signals are decoded to determine which internal register to select. Normally, these signals are a latched version of the address outputs from the VL86C010 processor.
–RBE	11	Read Buffer Enable (CMOS level output) - Is taken low during a read of any –IORQ/ –IOGT device including the IOC.
–WBE	12	Write Buffer Enable (CMOS level output) - Is taken low during a write of any –IORQ/ –IOGT device including the IOC.
–RE	4	Read Enable (CMOS level output) - Provides the external peripheral timing strobe. This signal is used to time peripheral read access cycles.

SIGNAL DESCRIPTIONS (Cont.)

Signal Name	Pin Number	Signal Description
–WE	3	Write Enable (CMOS level output) - Provides the external peripheral timing strobe. This signal is used to time the peripheral write access cycles.
–RST	29	Reset (CMOS level open-drain input/output) - Is driven (active) low whenever the –POR input is low to provide the system with a power on reset pulse during cold restart. Once the –POR is inactive, the VL86C410 will monitor the –RST as an input to detect a warm system restart condition.
–POR	28	Power On Reset (Schmitt trigger, active low input) - Is used to generate a reset pulse during power on conditions and to differentiate cold restart from subsequent causes of the reset condition. This signal is usually connected to an RC network.
–IL7- –IL0	40-33	Interrupt (TTL level inputs) - These signals are the –IRQ interrupt active low inputs. These signals are level sensitive.
–IF	41	Interrupt (TTL level input) - This signal provides an edge-sensitive –IRQ interrupt source. An interrupt condition is generated on a falling edge of this input.
IR	42	Interrupt (TTL level input) - This signal provides an edge-sensitive –IRQ interrupt source. An Interrupt condition is generated when a rising edge is detected on this input.
FH1-FH0	31-30	Fast interrupt (TTL level inputs) - These lines provide –FIRQ interrupt sources. An interrupt condition is generated whenever the input is high.
–FL	32	Fast interrupt (TTL level input) - Provides a source of FIRQ interrupt. An interrupt condition is generated whenever this input is low.
C5-C0	49-44	Control (Bidirectional open-drain) - These signals provide six lines of general purpose programmable I/O. The direction of these signals is determined by bits in the Control Register. The outputs can be forced low by programming the Control Register. Otherwise the pins can be treated as inputs.
–IRQ	51	Interrupt Request (CMOS level open-drain) - Provides the interrupt signal to the system processor. When used with the VL86C010, this signal is tied directly to the –IRQ input of the processor.
–FIRQ	50	Fast Interrupt Request (CMOS level open-drain) - Signals the system processor with the fast interrupt condition within the system. When used with the VL86C010, this signal is tied directly to the –FIRQ input of the processor.
BAUD	27	Baud Clock (CMOS level output) - This signal is generated by Timer 2. The output level is toggled by the reload event on the counter. The frequency of the BAUD clock is determined by the equation: f(baud) = 1/(latch + 1) MHz (if the RCLK frequency is 8 MHz) f(baud) max is obtained when the latch value equals one providing a maximum value of 500 KHz.
KIN	52	Keyboard serial data Input signal (TTL level input) - Is connected to the keyboard input controller. A clock of 16 times the data rate is used by the receiver to clock keyboard data into the KART section from this line. The data should be input as LSB first on this pin.
KOUT	53	Keyboard serial data Output (CMOS level output) - This output provides serial data transmission to the keyboard. The data is transmitted with a fixed format of eight bits per character with one start bit and two stop bits. The data appears on this pin with LSB first.
VCC	26, 60	Power Supply - 5 V ±5%
GND	9, 43	Ground

FUNCTIONAL PIN DIAGRAM



FUNCTIONAL DESCRIPTION

If the Bank (B2-B0), Type (T1-T0), Chip Select (CS) and addresses lines (A6-A2) of IOC are joined to the CPU address lines, then the IOC and peripherals are viewed as memory mapped devices. This allows the programmer to specify in a single memory instruction the peripherals to be accessed and the type of timing cycle it requires. The following description of the IOC assumes the use of a VL86C110 (Memory Controller - MEMC) within the system. For further details of the operation of the memory controller and system bus, examine the VL86C110 data sheet. In a typical system, as shown in Figure 1, the VL86C410 space is divided into two halves. The upper half is occupied by the IOC and the lower half is left for additional I/O controllers.

The IOC space is decoded into eight banks, zero through seven, by the B2-B0 lines. The zero bank maps into the internal registers of the VL86C410. The remaining seven banks map onto the seven peripheral select lines, –S7 through –S1 respectively. Each of the seven peripheral banks are further decoded into four types of access by the T1-T0 signals as shown in Figure 2. The type of peripheral access determines the timing of the data transfer cycle.

A particular peripheral device may be accessed by choosing an address where CS is high, B2-B0 decode to the appropriate select, and T1-T0 indicate a timing cycle that suits the accessed device. The remaining low-order address bits may be used to select the register within the device.

Access Speed

While the peripherals appear as memory mapped devices it is not possible for all accesses to be completed with the same access time as main memory. The extra time taken to complete an I/O cycle is expressed as a number of extra RCLK cycles.

Addresses

The pipelined VL86C010 addresses are latched by external buffers to provide valid signals throughout both I/O accesses and ROM reads. The latches are controlled by a VL86C010 clock line which is stretched during slow cycles.

FIGURE 1. RECOMMENDED VL86C410 INTERCONNECTION

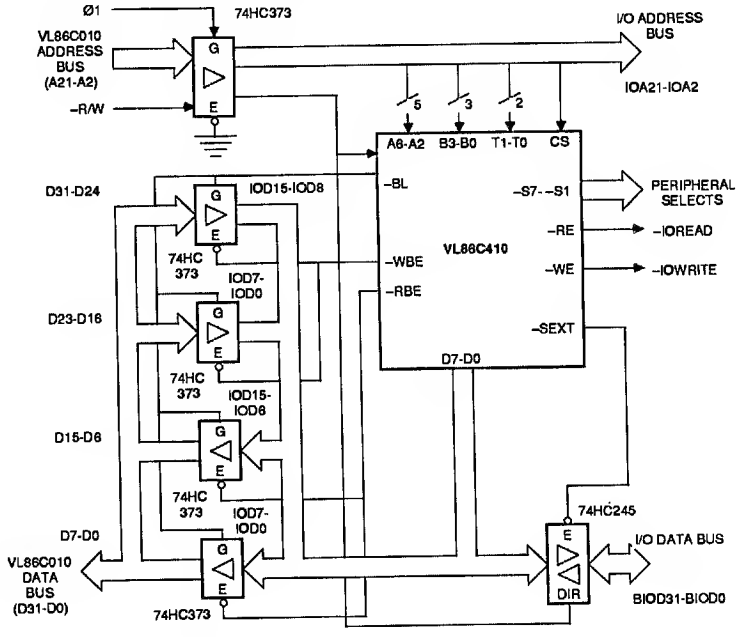
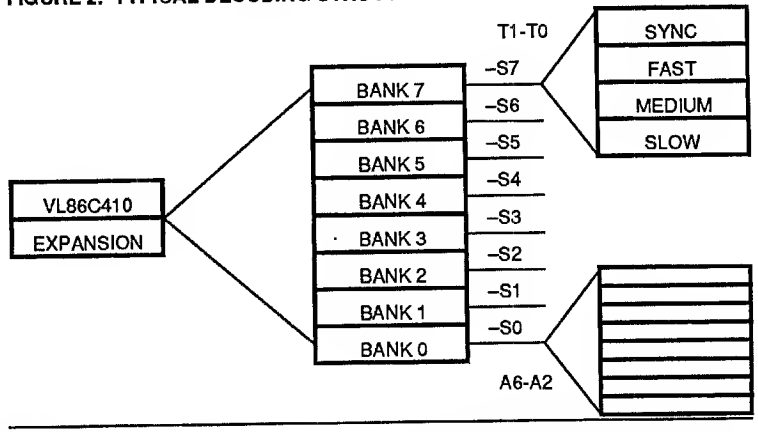


FIGURE 2. TYPICAL DECODING STRUCTURE



Data

The processor data bus is connected to the I/O data bus by a set of latches. These provide two functions. First, they isolate the I/O bus load from the main data bus and second, they allow for the mis-match in speed. These buffers are entirely controlled by the –BL, –RBE, and –WBE lines from IOC.

Internal Registers

All internal registers are accessed with no wait states and accesses take two RCLK cycles to complete. The internal registers are decoded as bank zero so to access them, the B2-B0 lines must all be low and the IOC must be selected by taking CS high. The individual registers are then addressed using the A6-A2 lines. The registers are decoded on word boundaries. The state of T1-T0 lines is ignored. The address values for each internal register are shown in Table 1.

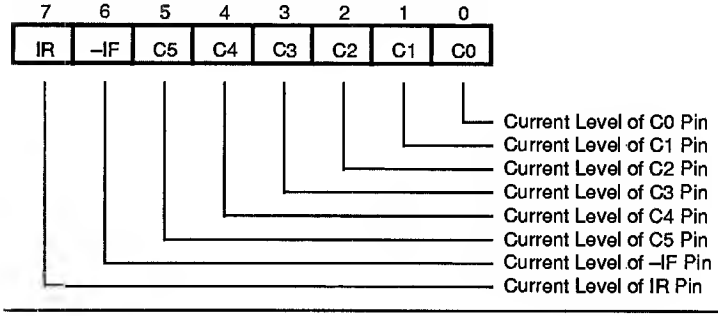
TABLE 1. VL86C410 INTERNAL REGISTER ASSIGNMENTS

Address (Hex)	Read	Write	Address (Hex)	Read	Write
00	Control Register	Control Register	40	T0 Count Low	T0 Latch Low
04	Serial Rx Data	Serial Tx Data	44	T0 Count High	T0 Latch High
08	–	–	48	–	T0 Go Command
0C	–	–	4C	–	T0 Latch Command
10	IRQ Status A	–	50	T1 Count Low	T1 Latch Low
14	IRQ Request A	IRQ Clear	54	T1 Count High	T1 Latch High
18	IRQ Mask A	IRQ Mask A	58	–	T1 Go Command
1C	–	–	5C	–	T1 Latch Command
20	IRQ Status B	–	60	T2 Count Low	T2 Latch Low
24	IRQ Request B	–	64	T2 Count High	T2 Latch High
28	IRQ Mask B	IRQ Mask B	68	–	T2 Go Command
2C	–	–	6C	–	T2 Latch Command
30	FIRQ Status	–	70	T3 Count Low	T3 Latch Low
34	FIRQ Request	–	74	T3 Count High	T3 Latch High
38	FIRQ Mask	FIRQ Mask	78	–	T3 Go Command
3C	–	–	7C	–	T3 Latch Command

Control Register
The control registers allow the external control pins C5-C0 to be read/written and the status of the IR and –IF inputs, prior to level conversion, to be inspected. The C5-C0 bits manipulate the C5-C0 I/O pins of the device. When the control register is read, they reflect the current state of the device pins. When the register value is written with a logic low value, the corresponding output pin is driven low. These outputs are open-drain, and if programmed high the pin is undriven and may be treated as an input. On reset all control register bits are set to logic high; thus C5-C0 will be inputs after reset.

Keyboard Asynchronous Receiver/ Transmitter (KART)
The KART provides an asynchronous serial link, usually to the keyboard. The frame format is fixed with 8-bits per character, one start bit, and two stop bits. It divides into two halves, the receiver and the transmitter. The receive and transmit speeds are the same and programmed using Timer 3.

FIGURE 3. CONTROL REGISTER READ (ADDRESS = 00H)



The VL86C010 accesses the receiver via the Serial Rx Data register. A clock of 16 times the data rate is used by the KART to clock in the serial data from the KIN pin. When a character has been received, the SRx bit is asserted in the IRQ B Status Register to indicate that a data byte is available for reading. False start bits of less than a half bit duration are ignored.

The VL86C010 accesses the transmitter via the Serial Tx Data register. The byte written to the Serial Tx Data register is transmitted serially from the KOUT pin and the STx bit is asserted in the IRQ B Status Register to indicate that the transmission is finished and the Serial Tx Data register may be reloaded.

Serial Tx Data
Writing to this register loads the serial output shift register, clears any outstanding interrupt and starts the transmission. An interrupt is raised when the register is ready to be reloaded. The data format for this register is shown in Figure 5.

FIGURE 4. CONTROL REGISTER WRITE (ADDRESS = 00H)

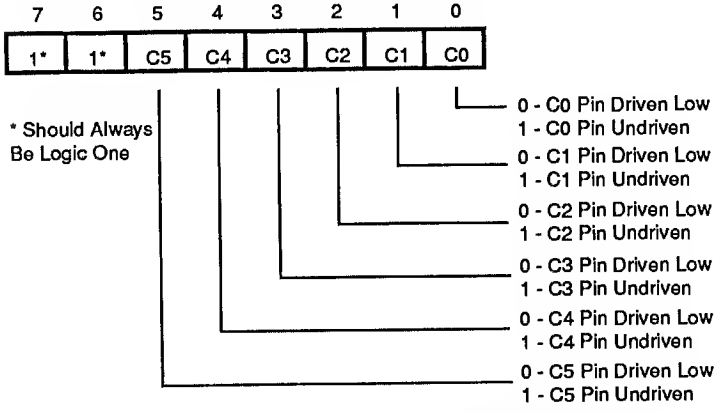


FIGURE 5. SERIAL Tx DATA REGISTER WRITE (ADDRESS = 04H)

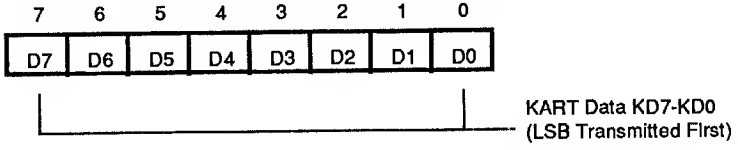


FIGURE 6. SERIAL Rx DATA REGISTER READ (ADDRESS = 04H)

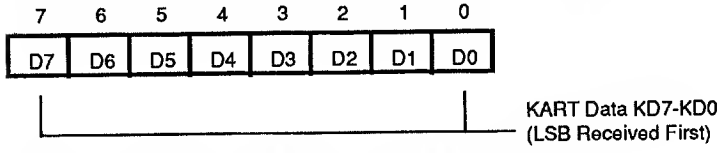
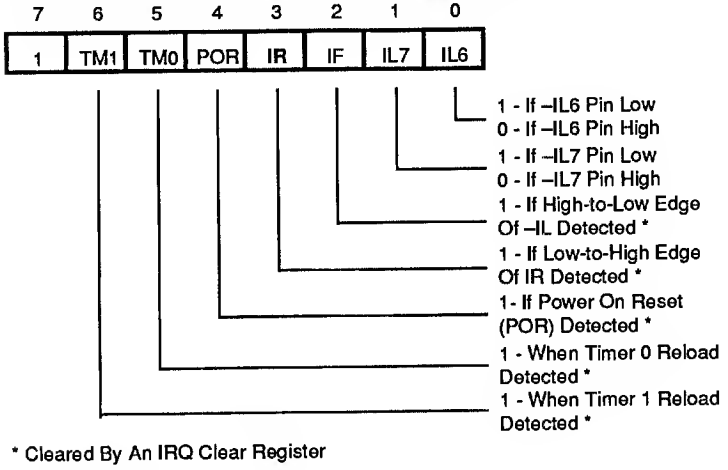


FIGURE 7. IRQ STATUS REGISTER A READ (ADDRESS = 10H)



* Cleared By An IRQ Clear Register

Serial Rx Data
Reading from this register clears any outstanding interrupt and returns the currently received byte as shown in Figure 6. Data is only valid while the SRx Bit is set in the IRQ B status register.

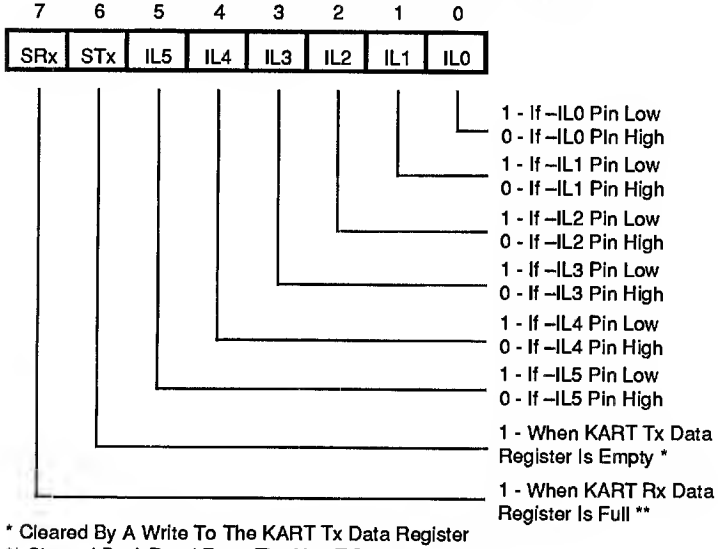
Initialization
After power on, the KART is in an undefined state. The KART is initialized by programming the serial line speed using Timer 3 and performing a read from the Serial Rx Data register, discarding the data byte. This will clear any outstanding receive interrupt and enable the KART for the next reception. Finally the Tx Data register should be written. This will abort any transmission in progress and cause a new one to be started and clear any transmit interrupt.

Receive Interrupt
The receive interrupt is set halfway through the reception of the last data bit. Care should be taken to ensure that the last bit has been received before the Serial Rx data register is read, to prevent this bit being interpreted as the start bit of the next packet.

Interrupt Registers
The VL86C410 generates two independent interrupt requests, –IRQ and –FIRQ. Interrupt requests can be caused by events internal to the device as well as external events on the interrupt or control port pins. The internal sources of interrupt are: timer (TM1-TM0), power on reset, keyboard Rx data available (SRx), keyboard Tx data register empty (STx), and force interrupts. The sources of external interrupts are: IRQ active low inputs (–IL7- –IL0), IRQ falling edge input (–IF), IRQ rising edge input (IR), FIRQ active high inputs (FH1-FH0), FIRQ active low input (–FL), and control port pins C5-C3.

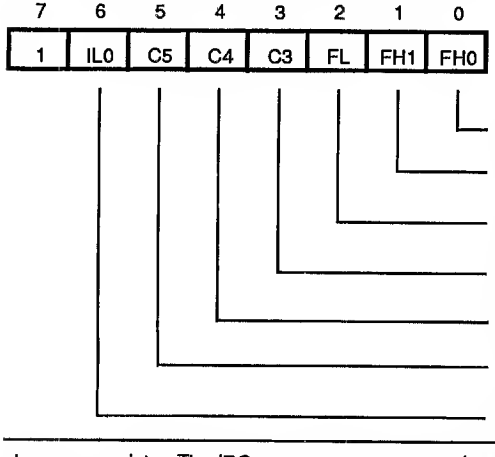
The IOC interrupts are controlled by four types of registers, status, mask, request, and clear. The status registers reflect the current state of the various interrupt sources. The mask registers determine whether the sources may generate an interrupt. The request registers are the logical AND of the status and mask registers, and indicate which sources are actually generating interrupt requests. The clear register allows clearing of interrupt requests

FIGURE 8. IRQ STATUS REGISTER B READ (ADDRESS = 20H)



* Cleared By A Write To The KART Tx Data Register
** Cleared By A Read From The KART Rx Data Register

FIGURE 9. FIRQ STATUS REGISTER READ (ADDRESS = 30H)



where appropriate. The IRQ events are split into two sets of resisters A and B. The mask registers are undefined after power up. There is no priority encoding of the sources.

Interrupt Sense
The –IF, IR, –POR, and TM1-TM0 are edge triggered and therefore are latched by the VL86C410. That is, once one of these sources has caused an interrupt it must be explicitly cleared. An event on one of these sources may be cleared by writing a logic one to the appropriate bit in the clear register. One or many may be cleared in a single operation.

The other interrupt sources are level-sensitive. When one of these sources has caused an interrupt condition, it is cleared by removing the source.

Timers
Four identical 16-bit counters are provided. Two are used as general purpose timers, the third for the keyboard baud rate and the fourth as a general purpose output baud. They have fully programmable start/reload values.

Each counter consists of a 16-bit down counter, a 16-bit input latch (latch low and latch high) and a 16-bit output latch (count low and count high) which contains the value of the counter when the latch command is given. The counter decrements continuously, clocked at RCLK/4. When it underflows, that is decrements to zero, it is reloaded from the input latch and recommences decrementing. The underflow is used to trigger different events depending on the use of the timer. If a counter is loaded with zero it continuously reloads and does not count. If the GO register is written at the same time as the counter underflows an extra clock tick is taken to reload. After power on the state of the counters is unknown.

Latch = latch low + 256 * latch high

Timer 0: General purpose interval timer
Timer 1: General purpose interval timer
Timer 2: External BAUD Pin
Timer 3: KART BAUD rate

Register Actions
–Latch Low - Writing to this updates the low order byte of the input latch.

–Latch High - Writing to this updates the high order byte of the input latch.

–GO - Writing to this causes the counter to be reloaded with the latch value.

–Count Low - This causes the low order byte of the output latch to be read.

–Count High - This causes the high order byte of the output latch to be read.

–Latch - This causes the current value of the counter to be placed in the output latch.

Timers 0 and 1
Two general purpose timers are provided. The underflow event sets a timer interrupt, TM1-TM0 in the IRQ Status A register. The interrupt is cleared via the IRQ Clear register. In order to generate an interrupt after time, Tinterval, the 16-bit value, (latch), to be used is calculated from the following equation:

Tinterval = latch/2 µs

(if the RCLK frequency is 8 MHz)

FIGURE 10. IRQ CLEAR REGISTER WRITE (ADDRESS = 14H)

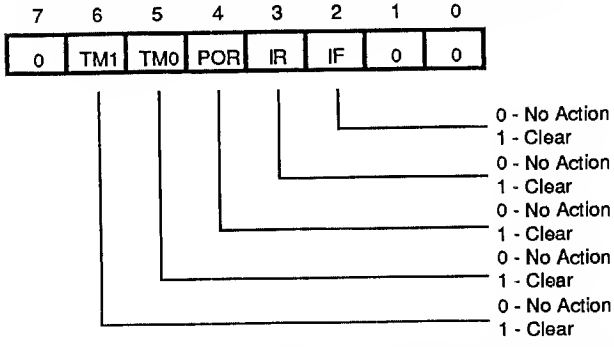


FIGURE 11. INTERRUPT REQUEST REGISTERS READ (ADDRESS = 14, 24, 34H)

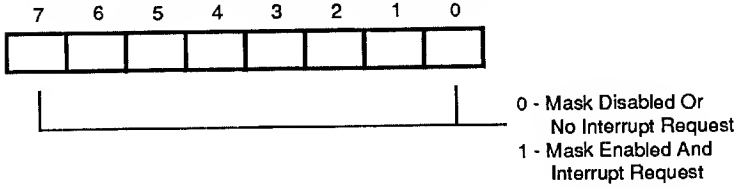


FIGURE 12. INTERRUPT MASK REGISTERS READ/WRITE (ADDRESS = 18, 28, 38H)

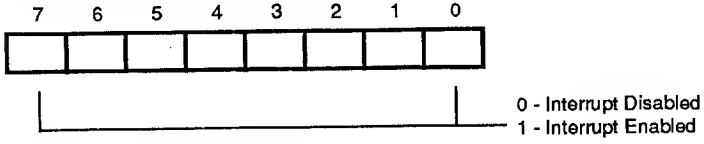
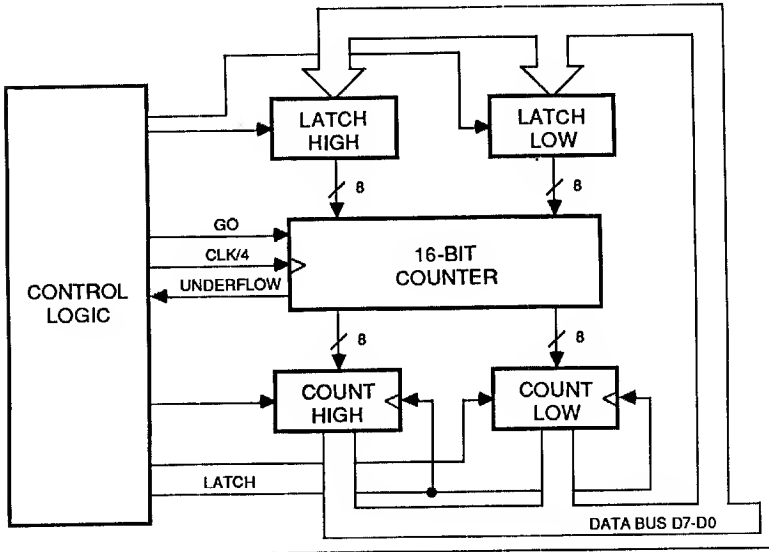


FIGURE 13. TIMER SCHEMATIC



Timer 2 (BAUD)
The timer 2 output is used to drive the BAUD pin. Maximum BAUD rate of 500 KHz is obtained with latch=1.

BAUD rate:=1/(LATCH+1) MHz

Timer 3 (KART)
The speed of the keyboard serial link is programmed via the KBaud registers. The maximum baud rate of 31,250 Hz is obtained for latch=1.

BAUD rate:=1/((latch + 1)*16) MHz

External Peripherals
The IOC provides control for external peripherals which cannot be accessed in a single cycle. A number of differently timed cycles, selected by the T1-T0 lines, are provided. Decoding of the T inputs and length of the various cycles is shown in Table 2. The peripheral cycles are controlled by a small state machine shown in Figure 14. Internal accesses complete in two RCLK cycles and the state machine remains idle. The cycles are timed to two clocks CLK/4 and CLK/1. Two timed data strobes, write enable (–WE) and read enable(–RE) manipulate data.

The number of RCLK cycles an I/O access takes to complete depends on three things: the minimum time for the cycle; the synchronization time; any DMA activity on the VL86C010 bus. The times are expressed as additional cycles over a normal memory access. The first three cycles share common timing and are fixed duration. The last is a square wave synchronized to the CLK/4 output. Examples of the peripheral access timing are shown in Appendix A.

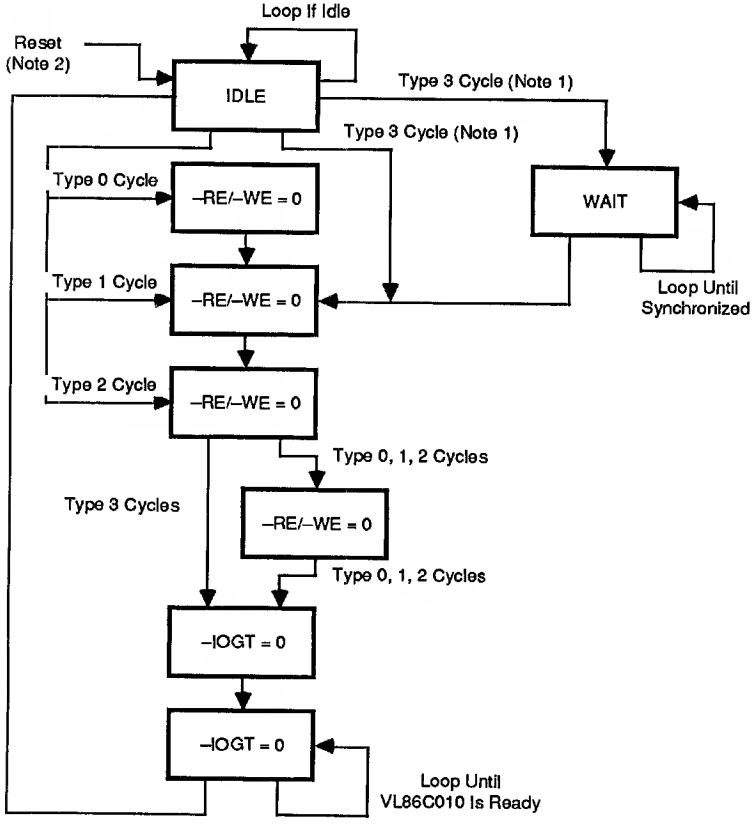
Peripheral Address and Data
The peripheral address and data are not provided by the VL86C410, so their timing is system dependent. The following explanation assumes that the configuration is as shown in Figure 1. Additionally, buffer delays through the VL86C410 can be up to half a RCLK pulse, so there can be a considerable skew between signals generated by the IOC and other sources in the system.

Peripheral select lines (–S7- –S1) are timed at the start of a cycle from –IORQ and disabled at the end of the cycle by the internal state machine.

TABLE 2. I/O CYCLE TIMING SELECTIONS

Cycle State			Minimum Cycle Length (Clocks)	Synchronization Time (Clocks)
T1	T0	Cycle Name		
0	0	Slow	7	0
0	1	Medium	6	0
1	0	Fast	5	0
1	1	Synchronous	5	0, 1, 2, 3

FIGURE 14. I/O CYCLE STATE MACHINE DEFINITION



External Writes
Since the MEMC may perform DMA transfers on the main data bus while an I/O cycle is completed, the write data must be latched to provide valid data throughout the I/O cycle. This is done by taking –BL low at the start of the cycle. It is taken high again at the end of the cycle.

External Reads
To provide fixed duration cycles for the peripherals, the read data is latched by taking –BL low as the –RE strobe is taken high. This allows the peripheral cycle to complete and the data is held in the data latches until the I/O cycle finishes.

Multiple –IORQ/–IOGT Peripherals
The IOC has been designed to allow multiple –IORQ/–IOGT devices to be connected to MEMC. For this reason the –IOGT and –BL lines are open-drain outputs. Even when it is not selected, IOC continues to control the external buffer enables –RBE and –WBE, so additional I/O devices need not generate these signals.

–IOGT Signal
In order for an internal register access to complete in two RCLK cycles the –IOGT signal cannot be logically dependent on –IORQ, which indicates the start of an I/O cycle, because –IORQ becomes valid too late. Therefore, –IOGT is generated from B2-B0 and CS only, and will sometimes be driven low during non-I/O cycles. During peripheral accesses the –IOGT signal is controlled by the state machine.

Reset
The IOC may be reset in two ways: by driving the bidirectional –RST line or the –POR line low. The –POR pin is designed to be connected to an external RC network to ensure that when power is first applied to the IOC, a minimum width reset signal is generated on –RST. A typical circuit is shown in Figure 15. –POR causes an internal latched interrupt to be set to allow system software to differentiate between power on and soft resets, and ensures that peripheral devices have

Notes: 1. Type 3 cycles will go into the wait state unless the cycle starts at the optimal point on the CLK/4 cycle.
2. Reset is a forcing signal to return to IDLE from any state.

had a stable clock for a suitable length of time before being released from reset. The control register is initialized on reset allowing the C5-C0 pins to be set to a known state, high, before the processor commences execution. The power-on reset timing is shown in Figure 16.

FIGURE 15. TYPICAL CIRCUIT FOR –POR CIRCUIT

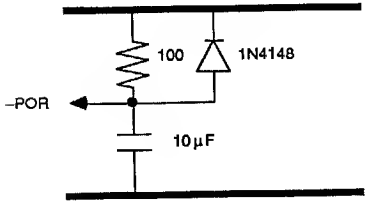
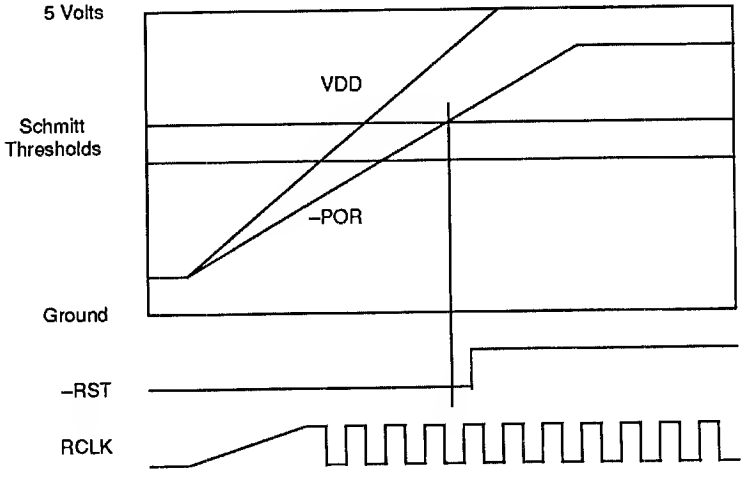


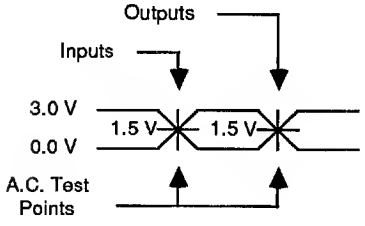
FIGURE 16. POWER-ON RESET TIMING



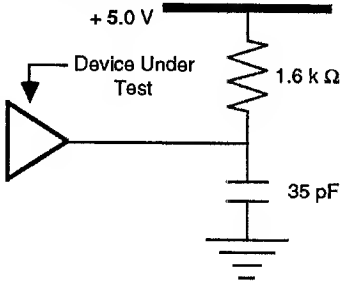
TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter	Min.	Typical	Max.	Units	Conditions
t1	–IORQ to RCLK Setup Time	35	–	–	ns	
t2	–IORQ to RCLK Hold Time	5	0	–	ns	
t3	–IORQ to Data Valid	60	120	–	ns	
t4	Data Hold Time	5	0	–	ns	
t5	B2-B0, –R/W, CS, A6-A2 Setup Time to –IORQ	0	50	–	ns	
t6	B2-B0, –R/W, CS, A6-A2 Hold Time from RCLK	5	10	–	ns	

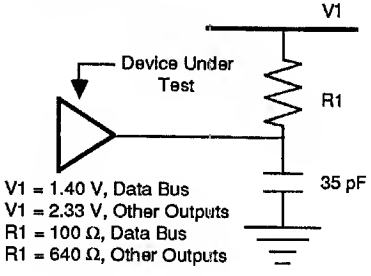
A.C. TEST WAVEFORMS



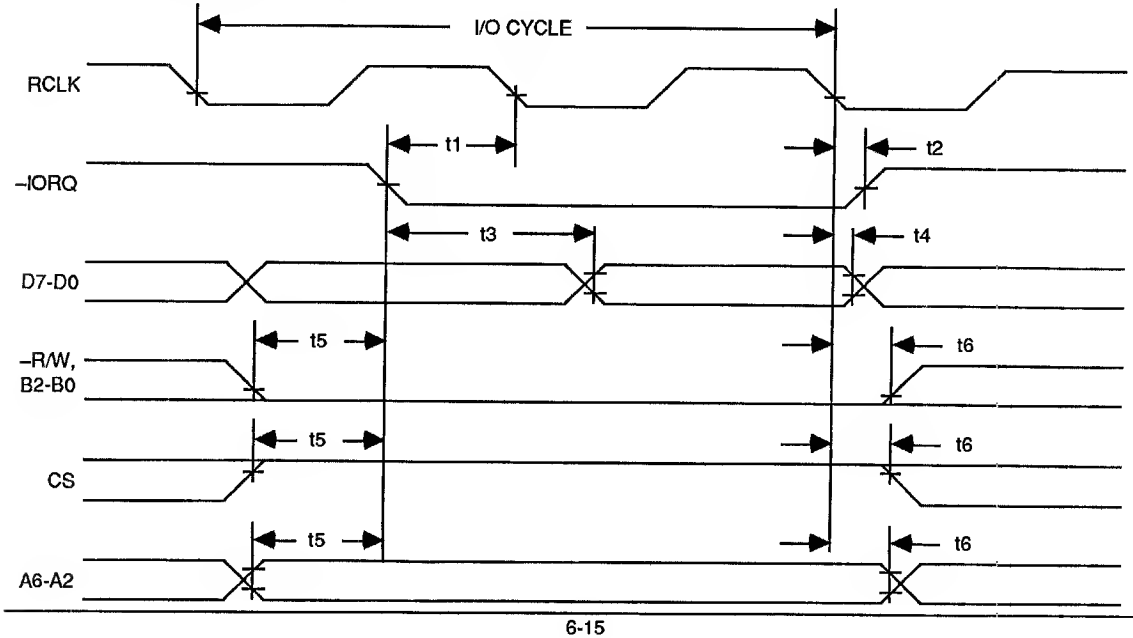
A.C. LOAD CIRCUIT
OPEN-DRAIN OUTPUTS



A.C. LOAD CIRCUIT
OTHER OUTPUTS



TIMING DIAGRAMS
INTERNAL REGISTER READ CYCLES

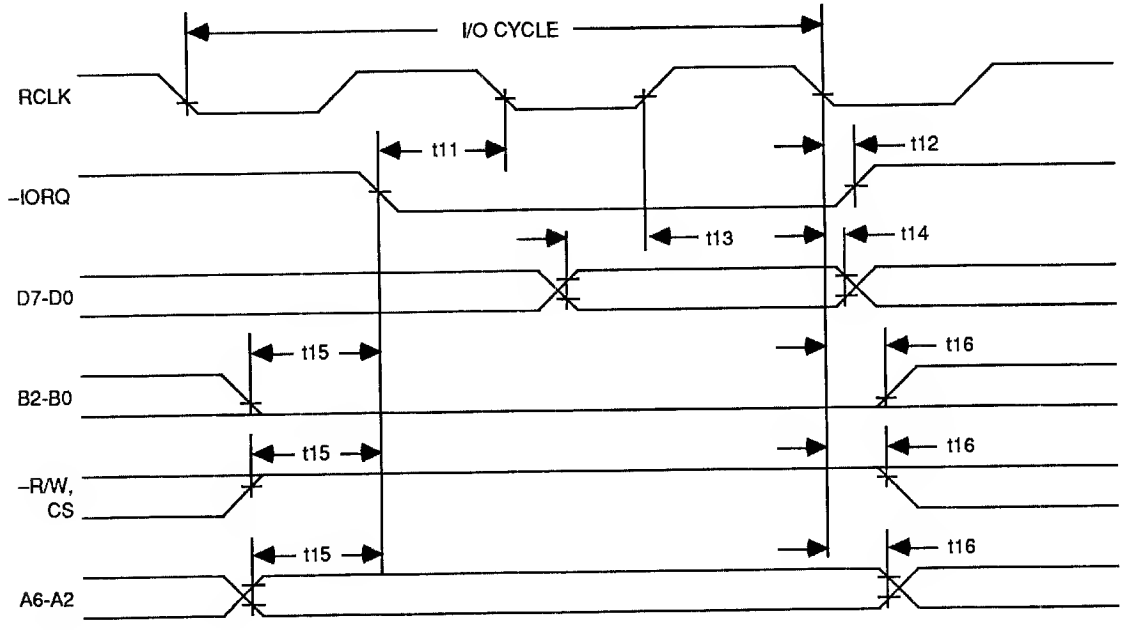


TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter	Min.	Typical	Max.	Units	Conditions
t11	–IORQ to RCLK Setup Time	35	–	–	ns	
t12	–IORQ to RCLK Hold Time	5	0	–	ns	
t13	Data Setup Time	30	20	–	ns	
t14	Data Hold Time	5	10	–	ns	
t15	B2-B0, –R/W, CS, A6-A2 Setup Time to –IORQ	0	50	–	ns	
t16	B2-B0, –R/W, CS, A6-A2 Hold Time from RCLK	5	10	–	ns	

TIMING DIAGRAMS

INTERNAL REGISTER WRITE CYCLES

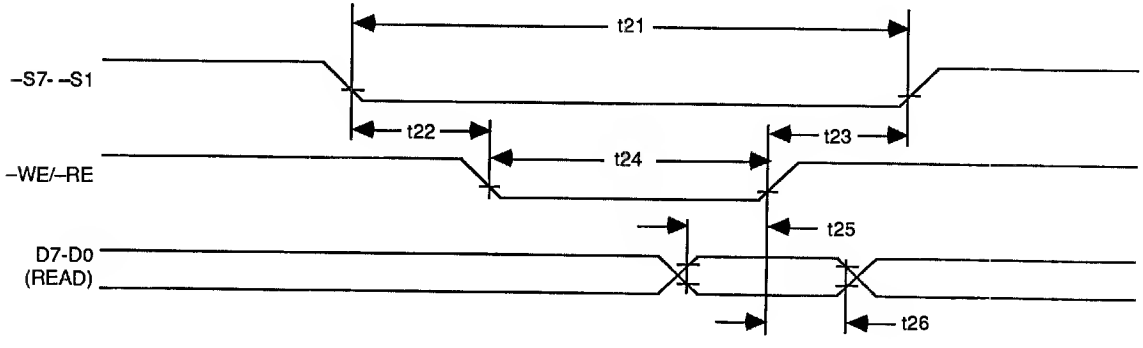


TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter		Min.	Typical	Max.	Units	Conditions
t21	–S7- –S1 Pulse Width	Peripheral Cycle Type 0	625	625	–	ns	
		Peripheral Cycle Type 1	500	500	–	ns	
		Peripheral Cycle Type 2	375	375	–	ns	
t22	–S7- –S0 to –RE/–WE	Peripheral Cycle Type 0	170	187	–	ns	
		Peripheral Cycle Type 1,2	50	62	–	ns	
t23	–RE/–WE to –S7- – S0 Delay		50	62	–	ns	
t24	–RE/–WE Pulse Width	Peripheral Cycle Type 0,1	350	375	–	ns	
		Peripheral Cycle Type 2	230	250	–	ns	
t25	Read Data Setup Time to –RE		–	20	–	ns	See Note 1
t26	Read Data Hold Time to –RE		–	20	–	ns	See Note 1

TIMING DIAGRAMS

CYCLE TYPES 0, 1, AND 2



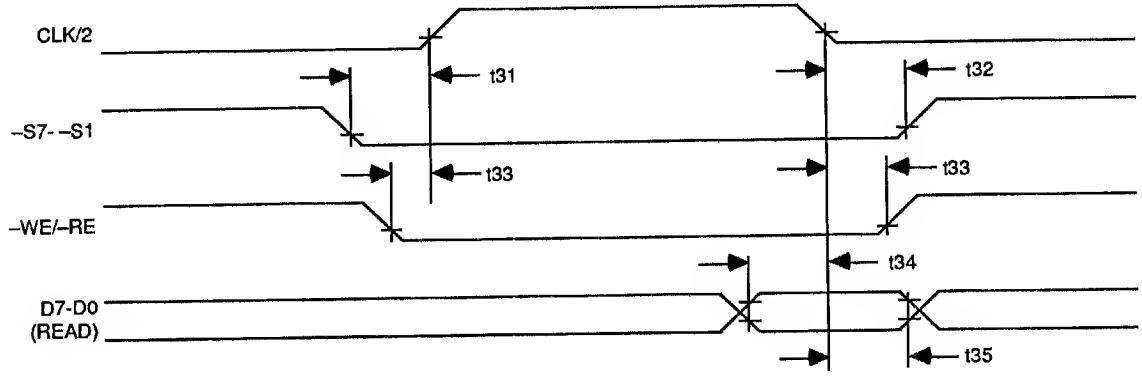
Note: 1. Assumes data is latched by the –BL signal.

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

Symbol	Parameter	Min.	Typical	Max.	Units	Conditions
t31	–S7- –S1 Setup Time to CLK/2	40	30	–	ns	
t32	–S7- –S1 Hold Time to CLK/2	20	10	–	ns	
t33	–RE/–WE to CLK/2 Skew	–	0	10	ns	
t34	Read Data Setup Time	–	20	–	ns	
t35	Read Data Hold	–	20	–	ns	

TIMING DIAGRAMS

CYCLE TYPE 3

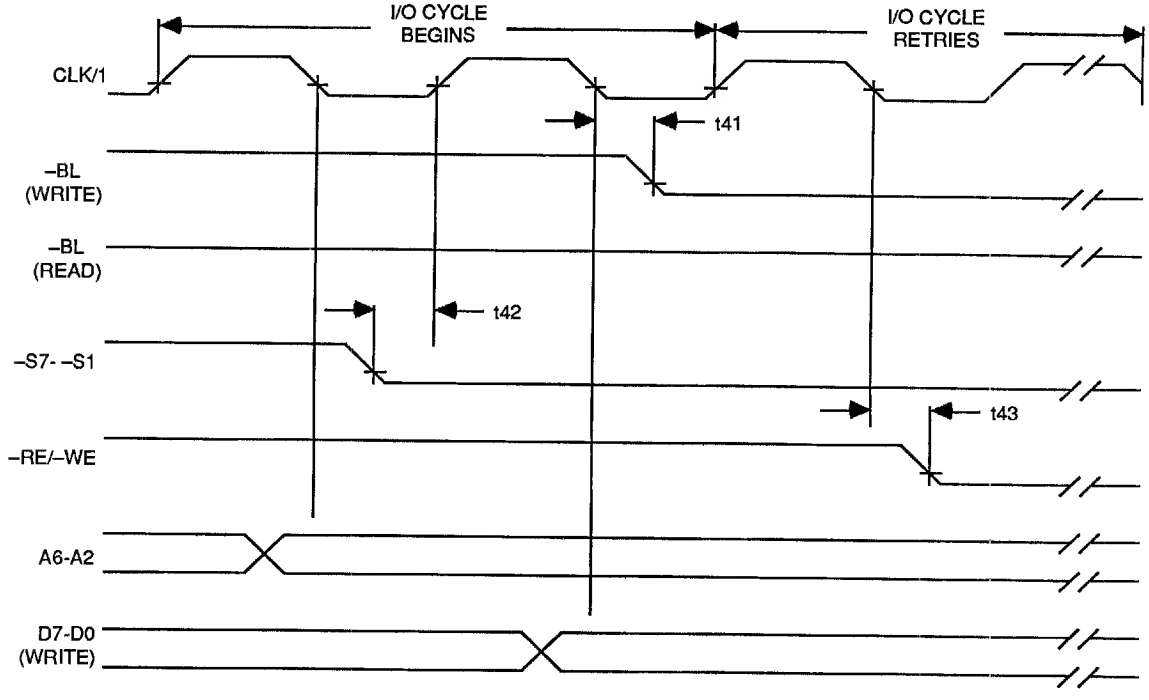


TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter	Min.	Typical	Max.	Units	Conditions
t41	–BL (Write Cycle) Delay	–	15	20	ns	
t42	–S7- –S1 Setup Time to CLK/1	20	10	–	ns	
t43	–RE/–WE Delay	–	30	10	ns	

TIMING DIAGRAMS

CYCLE START (TYPES 0, 1, AND 2)

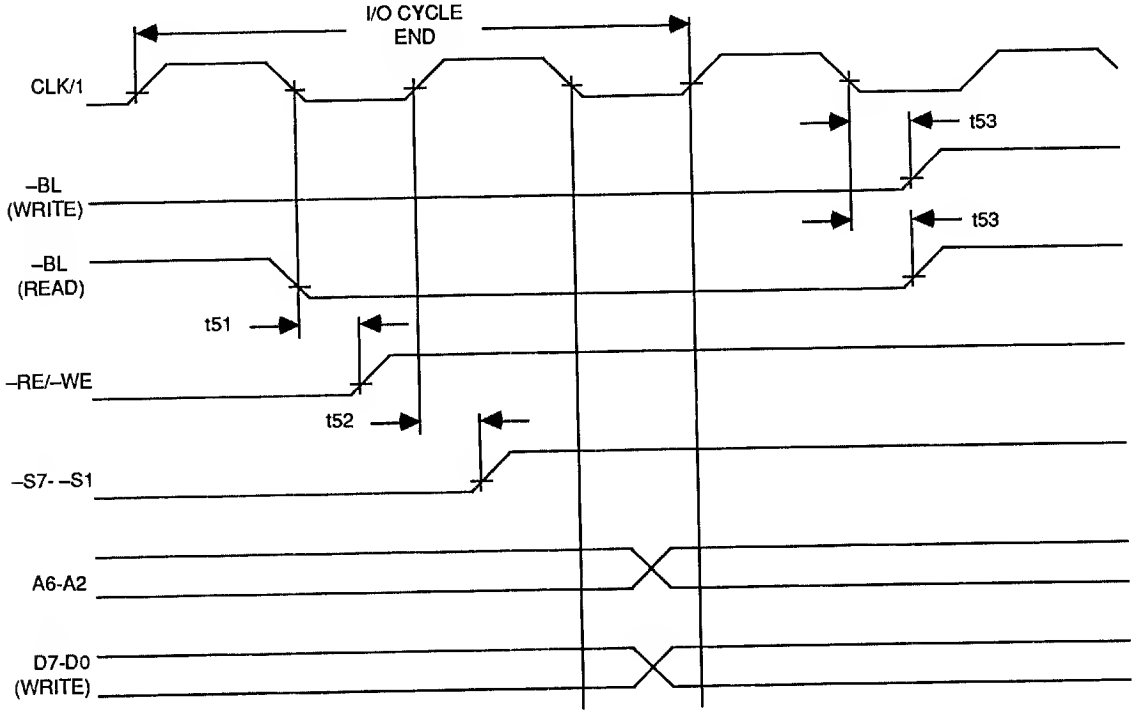


TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter	Min.	Typical	Max.	Units	Conditions
t51	–RE/–WE Delay from CLK/1	–	30	10	ns	
t52	–S7 - –S1 Disable Delay	–	30	10	ns	
t53	–BL Delay	30	30	–	ns	

TIMING DIAGRAMS

CYCLE END (TYPE 0, 1, AND 2)

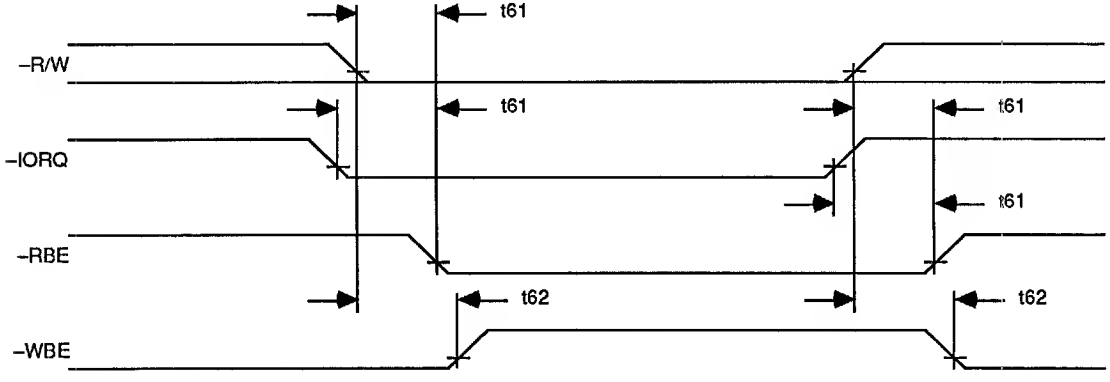


TIMING CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter	Min.	Typical	Max.	Units	Conditions
t61	–RBE Delay from –IORQ or –R/W	–	50	30	ns	
t62	–WBE Delay from –R/W	–	50	30	ns	

TIMING DIAGRAMS

READ AND WRITE BUFFER ENABLES



6

ABSOLUTE MAXIMUM RATINGS

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

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

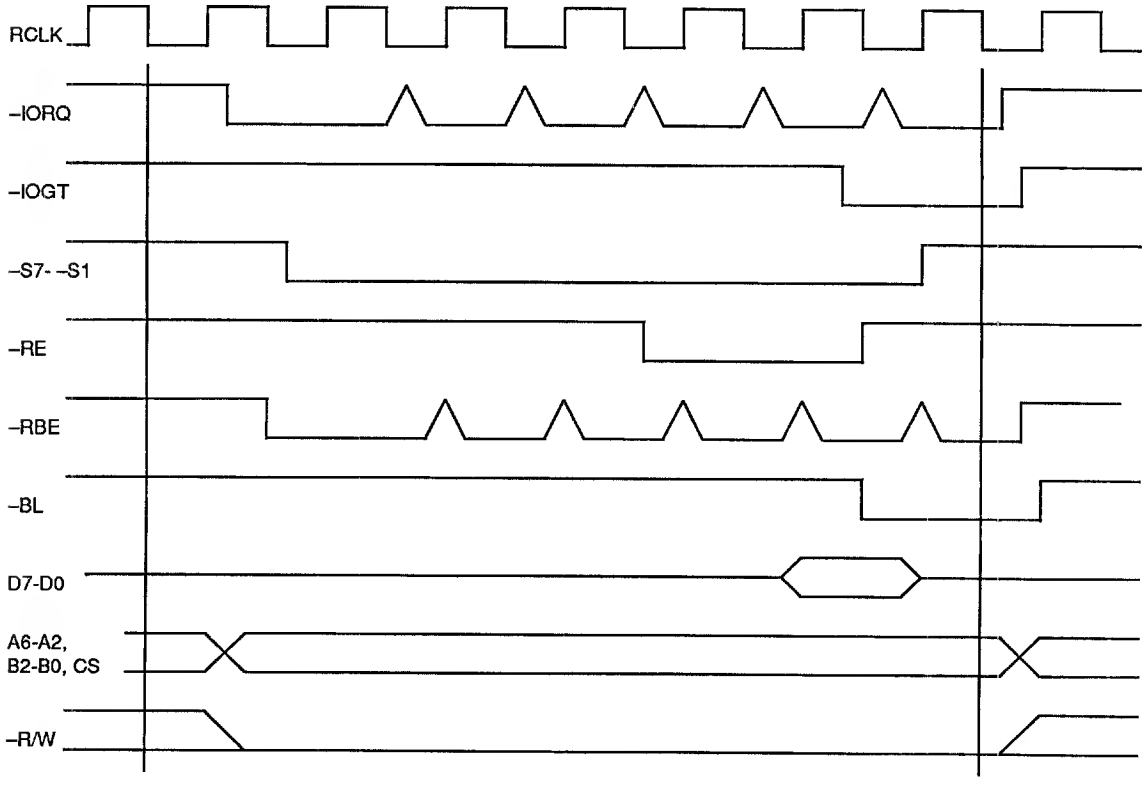
DC CHARACTERISTICS: TA = 0°C to +70°C, VCC = 5 V ±5%

Symbol	Parameter		Min.	Typical	Max.	Units	Conditions
VIH	Input High Voltage	CMOS Inputs	3.5	–	VCC	V	See Note 1
		TTL Inputs	2.4	–	VCC	V	See Note 1
		D7-D0	2.4	–	VCC	V	See Note 1
		C5-C0, –RST	2.4	–	VCC	V	See Note 1
VIL	Input Low Voltage	CMOS Inputs	0.0	–	0.8	V	See Note 1
		TTL Inputs	0.0	–	0.8	V	See Note 1
		D7-D0	0.0	–	0.8	V	See Note 1
		C5-C0, –RST	0.0	–	0.8	V	See Note 1
VOH	Output High Voltage	CMOS Outputs	VCC –0.75	4.2	–	V	IIH = 2.5 mA
		D7-D0	2.4	3.0	–	V	IIH = 10 mA
VOL	Output Low Voltage	CMOS Outputs	–	0.3	0.4	V	IIL = –2.5 mA
		D7-D0	–	0.6	0.8	V	IIL = –10 mA
		C5-C0, –RST	–	0.3	0.4	V	IIL = –2.5 mA
VIHST	Schmitt Trigger Input Rising Edge Threshold	VCC = 4.75 Volts	2.7	–	3.3	V	
		VCC = 5.0 Volts	–	2.8	–	V	
		VCC = 5.25 Volts	–	3.0	–	V	
VIHST	Schmitt Trigger Input Falling Edge Threshold	VCC = 4.75 Volts	1.2	1.5	1.8	V	
		VCC = 5.0 Volts	–	1.7	–	V	
		VCC = 5.25 Volts	–	1.9	–	V	
IOSC	Output Short Circuit Current		–	25	40	mA	See Note 2
IIN	Input Leakage Current		–	–	10	mA	
IDD	Supply Current		–	–	15	mA	At 8 MHz

Notes: 1. All voltages measured with respect to GND pin.
2. Not more than one output should be shorted to either rail at any time, and for no longer than one second.

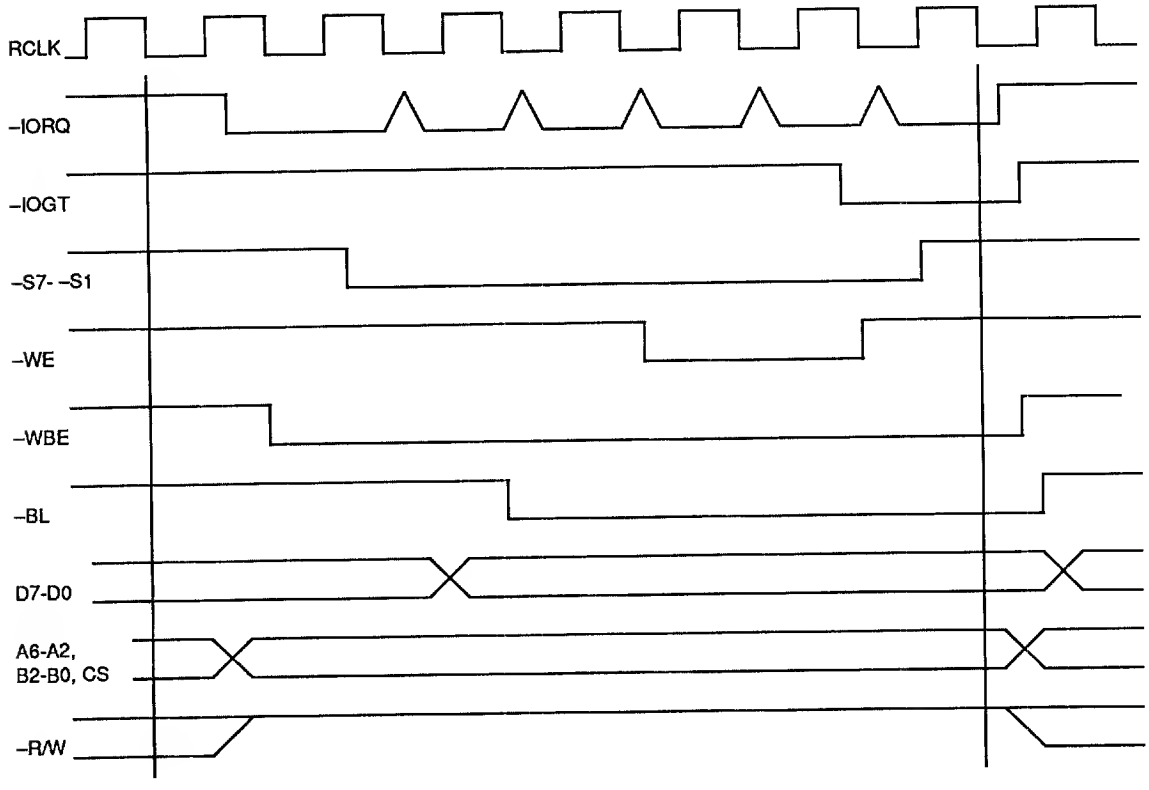
APPENDIX A - 1.

CYCLE TYPE 0 READ

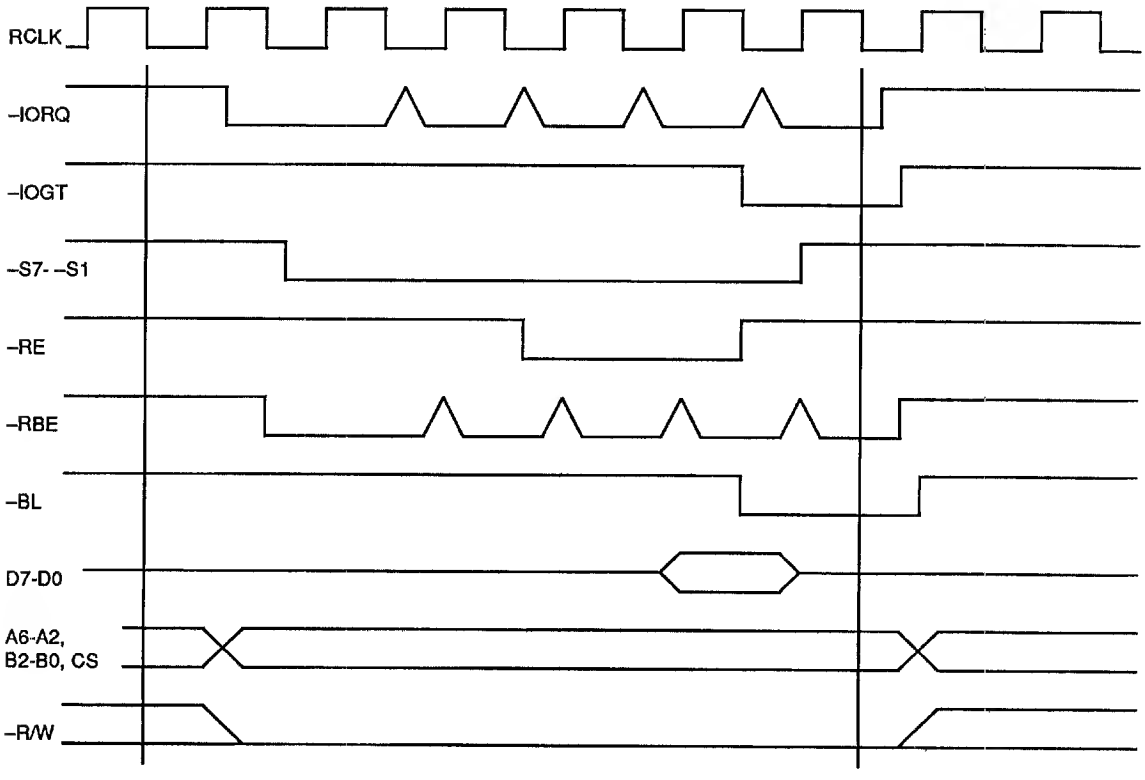


APPENDIX A - 2.

CYCLE TYPE 0 WRITE

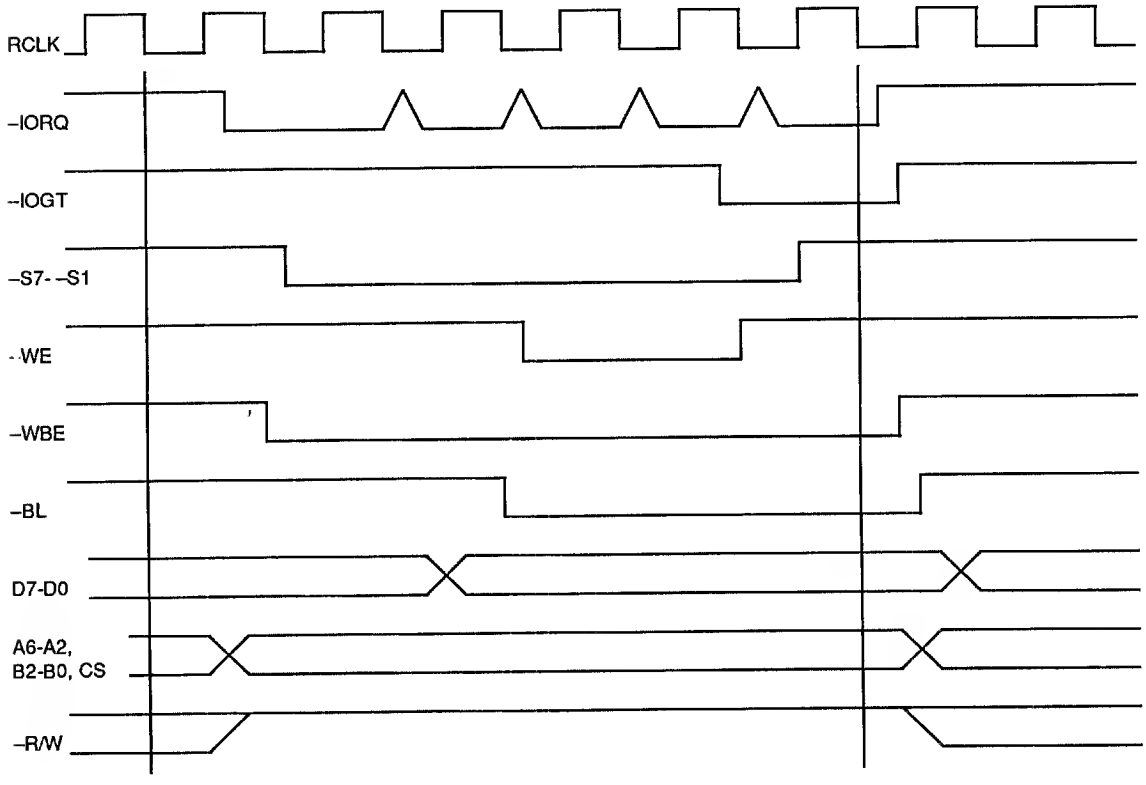


APPENDIX A - 3.
CYCLE TYPE 1 READ



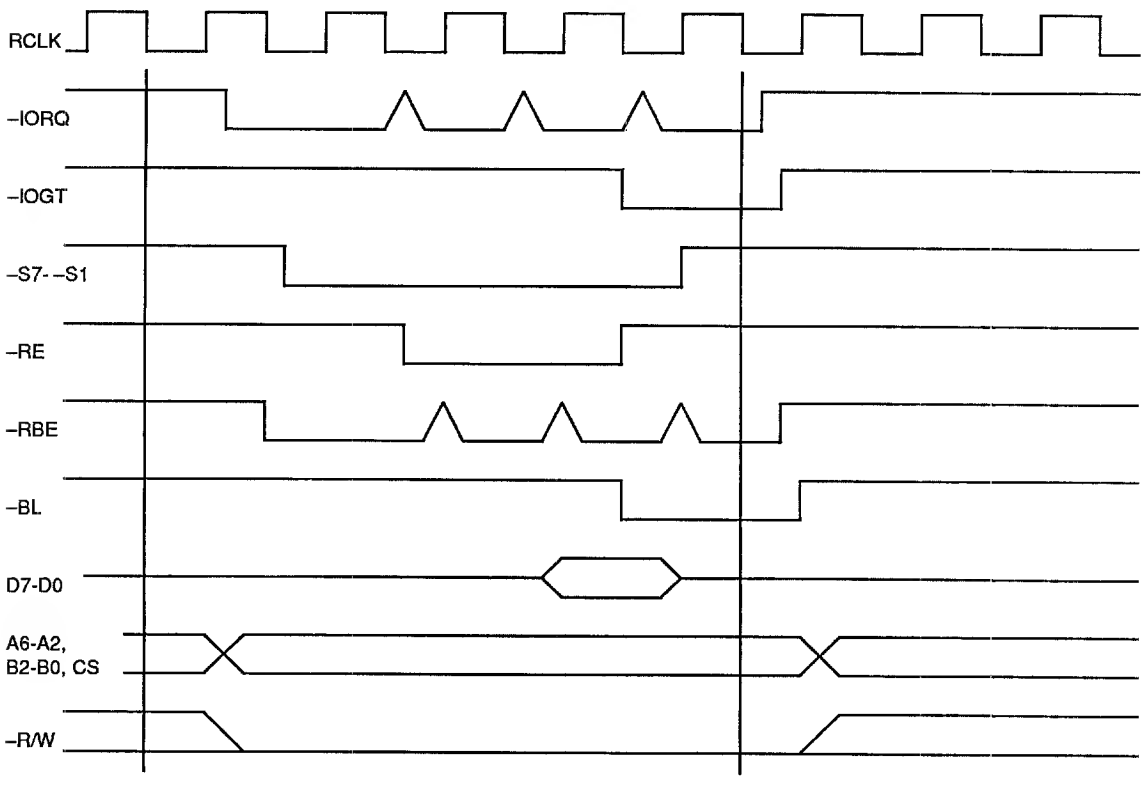
APPENDIX A - 4.

CYCLE TYPE 1 WRITE



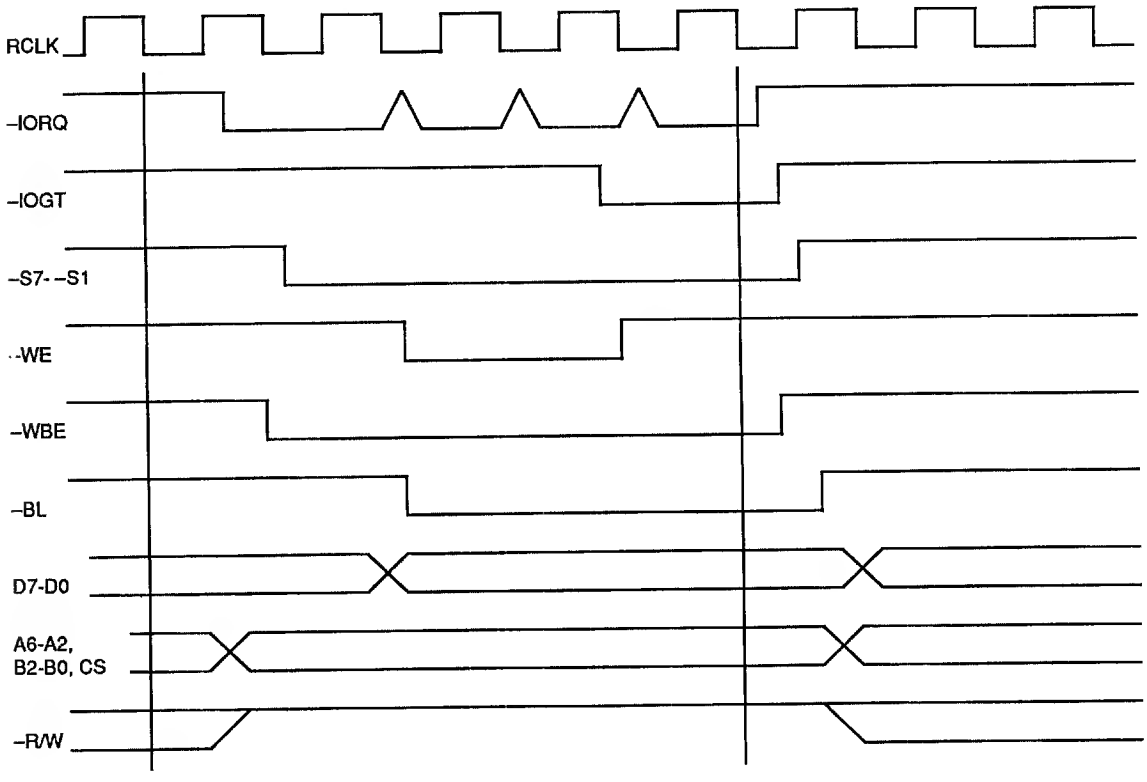
APPENDIX A - 5.

CYCLE TYPE 2 READ

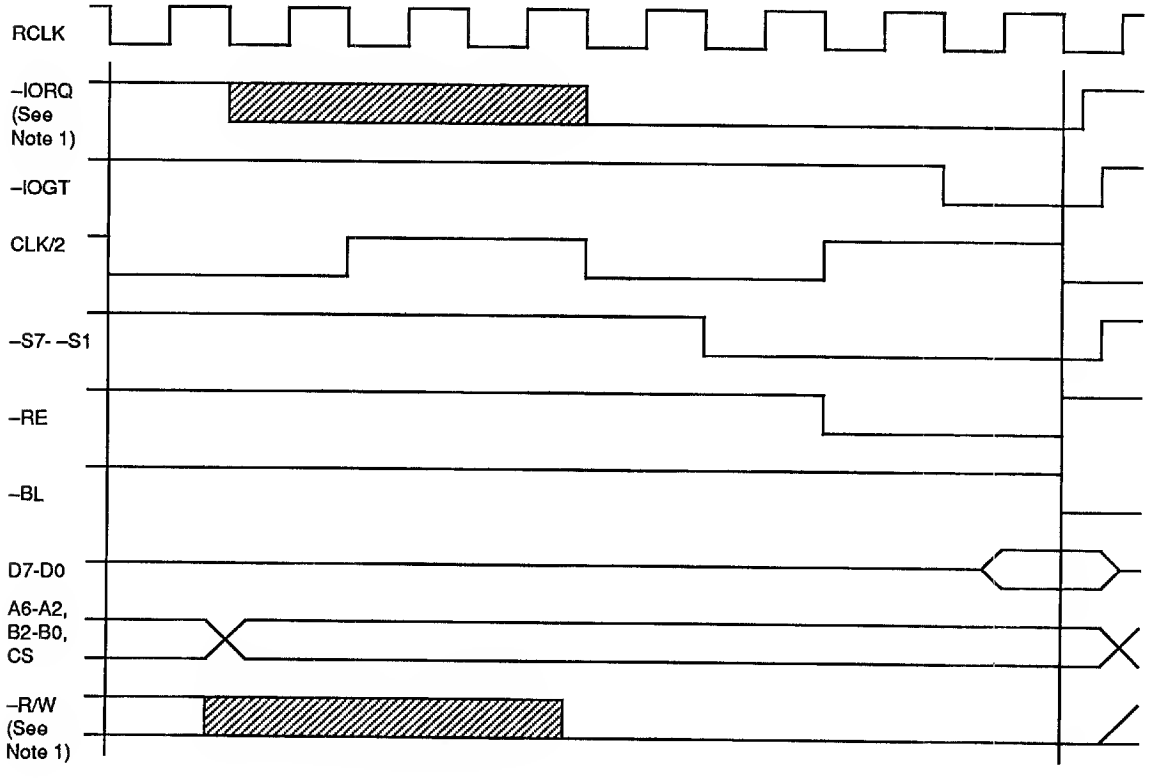


APPENDIX A - 6.

CYCLE TYPE 2 WRITE



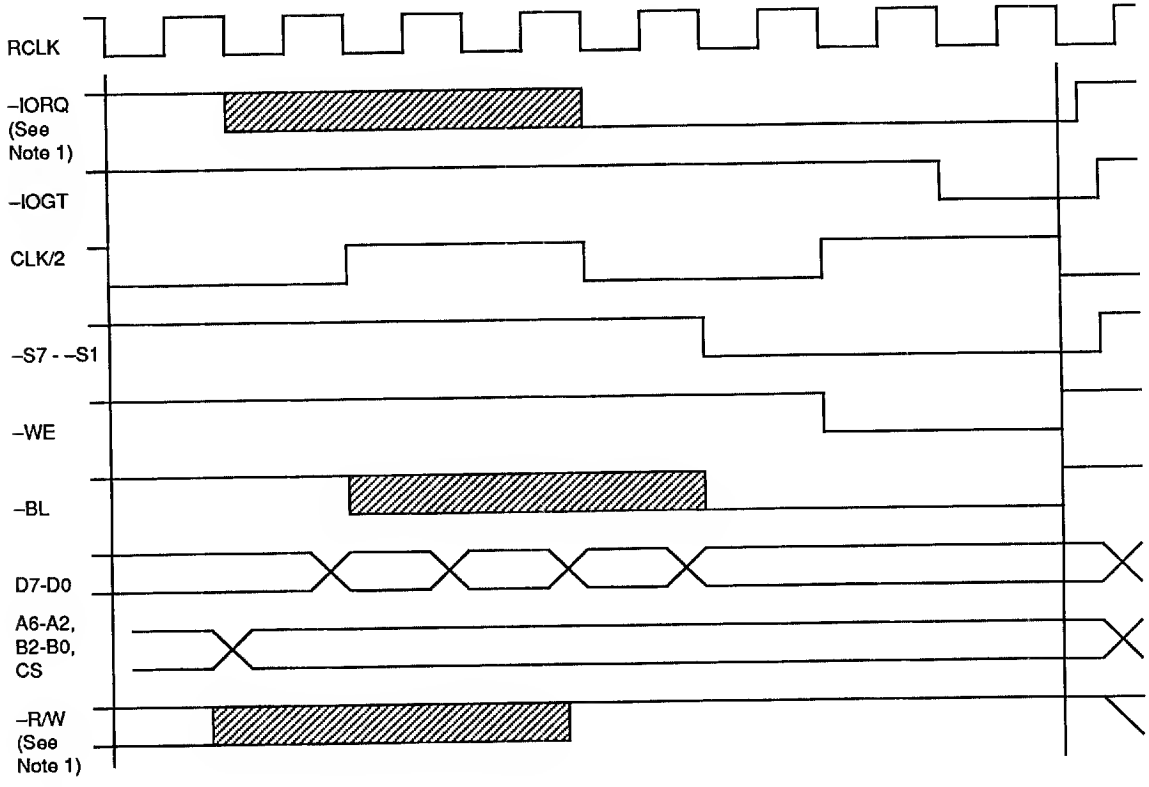
APPENDIX A - 7.
CYCLE TYPE 3 READ



Note: 1. This illustrates the four different sychronization delays represented by the possible –IORQ timings.

APPENDIX A - 8.

CYCLE TYPE 3 WRITE



Note: 1. This illustrates the four different sychronization delays represented by the possible –IORQ timings.

VLSI TECHNOLOGY, INC.

SECTION 7

RISC DEVELOPMENT TOOLS OVERVIEW

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

RISC DEVELOPMENT TOOLS OVERVIEW

BLUE STREAK DEVELOPMENT BOARD

FEATURES

- Hardware and software prototyping vehicle
- 1 MByte or 4 MByte memory
- IBM PC/AT drop-in card
- PC bus-master code
- RISC can access PC memory or PC I/O space
- RS-232C serial port
- Single bootstrap EPROM
- On-board memory manager (MEMC chip)
- Spare socket for 53C90-type SCSI adapter
- Fully supports OC disk and I/O operations
- Includes full source code for RISC monitor programs

DESCRIPTION

The Blue Streak is a PC/AT® add-in cerd that contains a VL86C010, VL86C110, and VL86C410 all operating at 8 MHz. The board is intended as a hardware/software development platform for the processor. The hardware architecture is such that the board is a bus master on the PC expansion bus and therefore the RISC has direct access to the PC memory and I/O space. For PC-to-board communication a simple mail box register is used. The VL86C010 accesses the PC bus under programmed I/O to simulate a DMA channel. An expansion bus is available on a 96-pin DIN connector to allow custom hardware to be attached for prototype development. The VL86C410 provides a full-duplex RS-232 port for downloading code into other target systems. Also on the board (but not supported in beta site versions) is a SCSI interface directly into the RISC system. Full schematics of the board are available to assist customers in interface issues with slower buses. The board is available 1 Mbyte and 4 Mbyte configurations or without memory for customers who can supply their own memory devices.

DEVELOPMENT SUPPORT

Included with the Blue Streak are all programs necessary for interface to the PC and several software development tools such as: debuggers, assemblers, and linkers. Programs are downloaded into the Blue Streak from the PC via the parallel bus. Monitor programs operating in both systems coordinate all I/O activity between the two systems.

Programs can be written in assembler language using the Compiling Assembler™ (CASM™) or the Super-C ANSI C Compiler. CASM is included with the Blue Streak system utilities; Super-C is an additional-cost item.

CASM - CASM supports high-level features like run-time expression evaluation in addition to the traditional macro capability. Structured constructs are also provided.

Super-C - Super-C is a full ANSI standard implementation of the C language for the VL86C010. The VLSI Technology, Inc. developed compiler generates code that is easily placed into ROMs.

LIBR - The object files created by the compiler or assembler may be merged into one or more libraries by the LIBR (librarian) utility program. LIBR is included with CASM.

CLINK - The CLINK linker is compatible with output files from either language. It links modules from both languages together into an executable format, and is included with the CASM assembler.

For beta site releases, CASM, Super-C, LIBR, and CLINK all execute on the PC. Full production releases will support execution on either the PC or Blue Streak.

VBUG - Programs running on the Blue Streak can be debugged using the VBUG Machine Debugger. The VBUG program allows for totally non-intrusive debugging in all processor modes. VBUG supports debug functions such as break pointing, single step, instruction tracing, register manipulation, and memory manipulation.

7

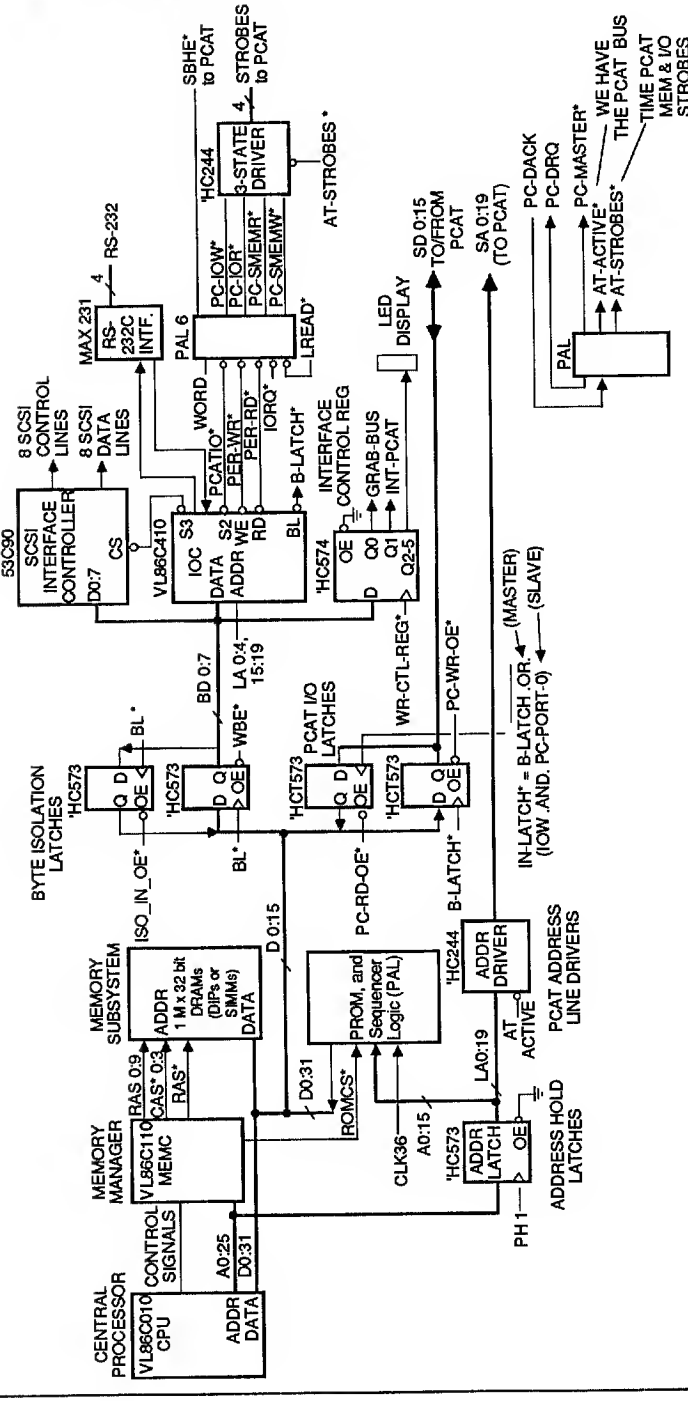
ORDER INFORMATION

Part Number	Description
VL86C010-SB (No memory version) VL86C010-SB3 (1 meg version) VL86C010-SB4 (4 meg version)	Blue Streak Boerd
VL86C010 - DB1	Arm-3 Deughter Cerd
VL86C010-SW1-CASMPC VL86C010-SW1-CASMRS	Compiling Assembler (CASM)™
VL86C010-SW1-SUPCPC VL86C010-SW1-SUPCRS	Super-C ANSI C Compiler
VL86C010-VBUG	VBUG Machine Level Debugger

PC/AT® is a registered trademark of IBM Corporation.

CASM™ and Compiling Assembler™ are trademarks of NIKOS Corporation of Phoenix, Arizona.

BLUE STREAK SYSTEM DIAGRAM



ARM-3 DAUGHTER CARD

DESCRIPTION

This is a daughter card that connects to the Blue Streak board. It contains a VL86C020 processor with 4 Kbytes of instruction and data cache on-chip.

This card contains a PLCC adapter that lets it replace the processor chip on the Blue Streak. The new processor runs at 20 MHz, but uses the same 8 MHz memory subsystem of the unmodified Blue Streak. Most programs then run 2.5 - 3.0 times faster than the original processor, when the cache is enabled. The new board is fully software compatible with the original processor.

COMPILING ASSEMBLER (CASM)

DESCRIPTION

The CASM Assembler provides the ability to program at the machine level effectively and efficiently. Since the processor has fully interlocked pipelines and very simple parallelism, programming in assembler for the VL86C010 is very similar to the more traditional CISC architectures. Performance from the processor does not depend on highly optimized compilers, so the assembly programmer is not required to manage pipeline flows and optimal scheduling strategy as in other RISC processors.

CASM can be used as an ordinary macro assembler or in a compiling mode that generates machine code similar to high-level language statements. Support for listing indentation and structured flow control statements improve programmer productivity.

CASM creates relocatable object modules.

Included with CASM is the CLINK linker. It allows modules to be assembled or compiled independently, and combined into one module for execution. CLINK supports 16 location counters and allows programs to be partitioned for different classes of memory (ROM, RAM, stack, common memory, etc.).

Also included is the LIBR program librarian. This utility merges commonly-used program modules together into a single file. The linker can then automatically search that (library) file for any modules that it needs to complete the construction of a program. This eliminates the requirement to tell the linker the detailed names for common utility modules often used by programs.

DEVELOPMENT ENVIRONMENT

Two versions are available. One that executes on the IBM PC and the other directly on the Blue Streak board. The Blue Streak includes both CASM and CLINK in the basic system. Users who wish to develop code on the IBM PC and download into their target hardware may purchase a cross assembler copy that executes on the PC and produces VL86C010 code.

Modules created on the Blue Streak board may be freely mixed with those created on the PC environment, and vice versa, during the program linking process.

SUPER-C ANSI C COMPILER

DESCRIPTION

The SUPER-C ANSI C Compiler implements the full ANSI specification of the C language for the VL86C010 family processors. The instruction set architecture of the VL86C010 lends itself to efficient compiler implementations and optimization. The compiler uses the conditional execution and condition code control provided by the instruction set to produce optimized code. In addition, efficient register allocation minimizes the number of load/store instructions.

The object code modules produced by SUPER-C are compatible with the CASM and CLINK programs to allow modules written in the high-level language and assembler to be combined.

The runtime libraries follow the ANSI definitions, and support the Blue Streak hardware environment. Source code may be purchased for the libraries so that they may be ported to alternative hardware configurations.

DEVELOPMENT ENVIRONMENT
Two versions are available. One that executes on the IBM PC and the other directly on the Blue Streak board. Users who wish to develop code on the IBM PC and download into their target hardware may purchase a cross compiler copy that executes on the PC and generates VL86C010 code.

Modules created on the Blue Streak board may be freely mixed with those created on the PC environment, and vice versa, during the program linking process.

LIBR LIBRARIAN UTILITY (INCLUDED WITH CASM)

DESCRIPTION

LIBR is a librarian utility that merges software object modules into a single file. The resulting library file is used by the CLINK linker. Placing commonly used functions and modules into a library file minimizes the effort needed to link programs. It also allows programs to be grouped conveniently, such as a different library for different hardware configurations.

DEVELOPMENT ENVIRONMENT
Two versions are available. One that executes on the IBM PC and the other directly on the Blue Streak board. Modules created on the Blue Streak board may be freely mixed with those created on the PC environment, and vice versa, during the library merging process.

ODUMP OBJECT DUMP UTILITY (INCLUDED WITH CASM)

DESCRIPTION

ODUMP is a utility program that extracts and dumps information on an object module to the screen. It may be used to inspect data such as the object file header containing dates, times, source environment, and the like. It is also used to inspect relocation records, displaying them in an easy-to-read manner.

DEVELOPMENT ENVIRONMENT
Only one version is provided, it executes on the PC. It may dump data from modules created on either the PC or on the Blue Streak environments.

VBUG MACHINE LEVEL DEBUGGER

DESCRIPTION

The VBUG program is a machine-level debugger for the VL86C010. It supports software development at the object code level. VBUG allows programs to be loaded into the Blue Streak and controlled via the keyboard. Functions supported include trace, single-step, register examination, and register/memory modification.

Both Step and Step-Over modes are supported for the Single-Step and the Trace commands. Step-Over mode does not perform tracing inside a subroutine that may be called. During both Single Step and Tracing, options may be selected such that each instruction, all 16 registers are displayed. Alternatively, only the registers referenced by the instruction, or only the registers changed by the instruction, may be automatically displayed.

It is possible to trace or single-step in any of the four processor modes, and through transitions from one such mode to another. It is possible, therefore, to trace from User mode into an SWI call (if not using Step-Over tracing).

At all times that VBUG is in control of the keyboard, the user's memory is as it was left. That is, no code is left in the memory after a trace or a Step has been completed. This means that program crashes will not cause debugger code to be left in the user memory areas.

Separate copies are kept of the register environments for each of the possible processor machine states.

ROM areas cannot be traced.

DEVELOPMENT ENVIRONMENT

VBUG is provided with the Blue Streak development board. It is currently only available on Blue Streak as a disk based debugger.

BLUE STREAK FIRMWARE AND PC/AT SHELL (INCLUDED WITH BLUE STREAK BOARD)

DESCRIPTION

The Blue Streak support firmware is comprised of four sections: Bootstrap ROM code, Blue Streak initializer, the RISC-resident monitor, and the PC/AT resident I/O support shell.

The ROM code contains a short program to set up the initial state of the Blue Streak card and to load a (monitor) program from the PC/AT. The initializer program operating in the PC/AT loads the RISC's monitor program from a disk file.

The monitor is a single-tasking program that maintains an operating environment for the user code. It supports both character and disk I/O through DOS, via the PC/AT shell program. Because of the DMA-like bus interface on the Blue Streak card, transfers between the monitor and the shell are very fast.

An interface shell program runs on the PC, and provides I/O services to the RISC's monitor. Both keyboard and disk I/Os are handled, using standard DOS indirection facilities.

The monitor does not support the SCSI adaptor device on the Blue Streak card. Source code is available for all of these programs.

DEVELOPMENT ENVIRONMENT

The bootstrap and the monitor programs execute on the Blue Streak board itself, while the initializer and shell operate on the PC/AT.

Notes:

VLSI TECHNOLOGY, INC.

SECTION 8

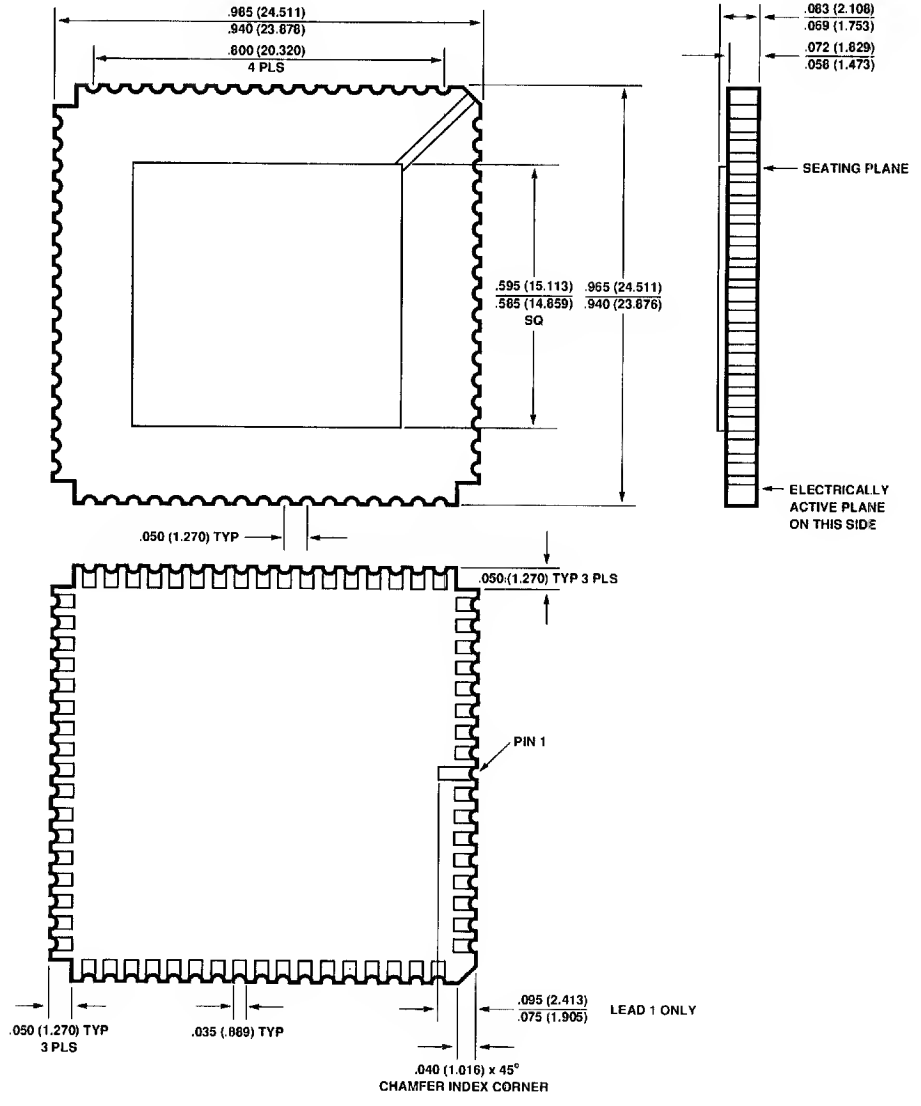
PACKAGING INFORMATION

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

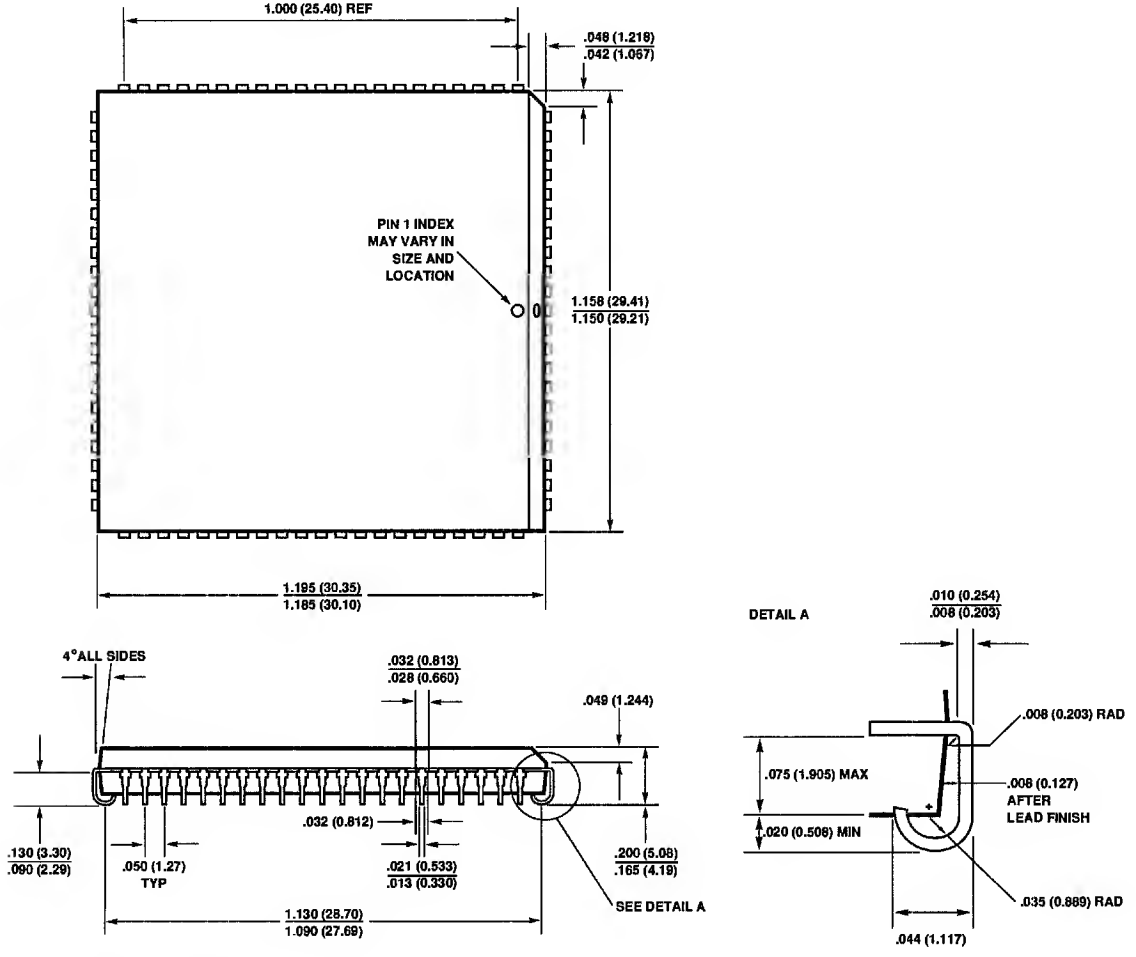
PACKAGE OUTLINES

68-PIN PLASTIC LEADED CHIP CARRIER (PLCC)



PACKAGE OUTLINES (Cont.)

84-PIN PLASTIC LEADED CHIP CARRIER (PLCC)

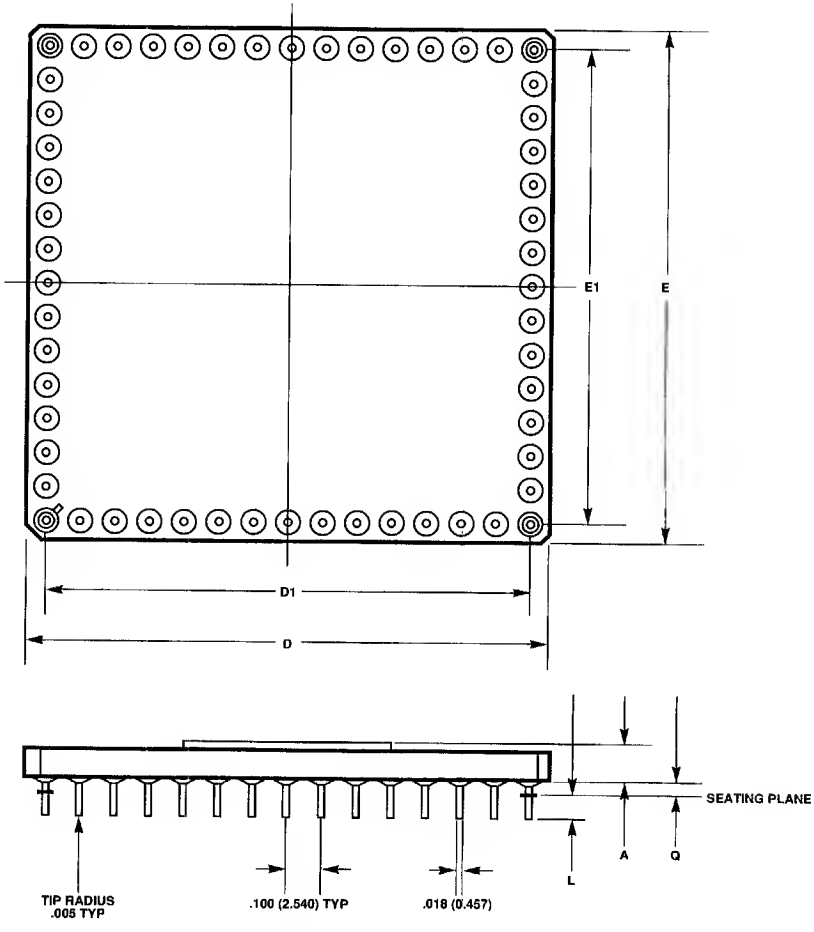


NOTES: UNLESS OTHERWISE SPECIFIED.
1. TOLERANCE TO BE +/- .005 (0.127).
2. LEADFRAME MATERIAL: COPPER.
3. LEAD FINISH: MATTE TIN PLATE OR SOLDER DIP.
4. SPACING TO BE MAINTAINED BETWEEN FORMED LEAD AND MOLDED PLASTIC ALONG FULL LENGTH OF LEAD.
5. MOLDED PLASTIC DIMENSION DOES NOT INCLUDE SIDE FLASH BURR, 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

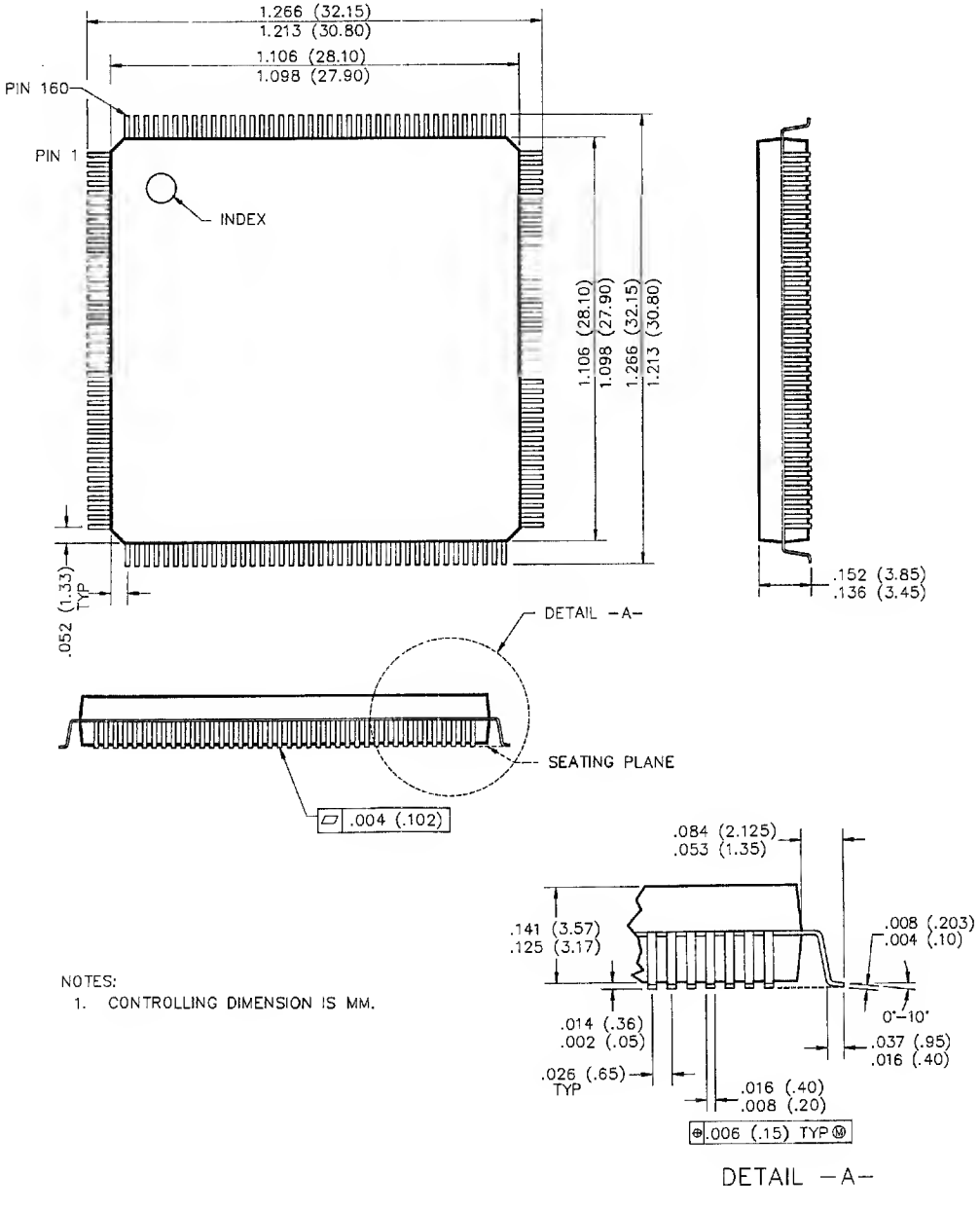


Pin Count	Matrix	Cavity Position	A		D (E)		D1 (E1)		Q	L
			Min	Max	Min	Max	Min	Max	Ref	Ref
144	15 x 15	Up	.0780 (1.981)	.1020 (2.591)	1.559 (39.60)	1.591 (40.41)	1.388 (35.26)	1.412 (35.86)	0.050 (1.270)	0.130 (3.302)

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

PACKAGE OUTLINES (Cont.)

160-PIN CERAMIC PIN GRID ARRAY



NOTES:

1. CONTROLLING DIMENSION IS MM.

VLSI TECHNOLOGY, INC.

SECTION 9

SALES OFFICES, DESIGN CENTERS, AND DISTRIBUTORS

Application Specific
Logic Products Division


VLSI TECHNOLOGY, INC.

VLSI TECHNOLOGY, INC.

SALES OFFICES, DESIGN CENTERS, AND DISTRIBUTORS

VLSI CORPORATE OFFICES

CORPORATE HEADQUARTERS • ASIC AND MEMORY PRODUCTS • VLSI Technology, Inc. • 1109 McKay Drive • San Jose, CA 95131 • 408-434-3100

APPLICATION SPECIFIC LOGIC AND GOVERNMENT PRODUCTS • VLSI Technology, Inc. • 8375 South River Parkway • Tempe, AZ 85284 • 602-752-8574

VLSI SALES OFFICES AND TECH CENTERS

ARIZONA
8375 South River Parkway
Tempe, AZ 85284
602-752-8450
FAX 602-752-6001

CALIFORNIA
2235 Ourne Or
San Joee, CA 95131
408-922-5200
FAX 408-943-9792
TELEX 278007
MAIL
1109 McKay Orive
San Jose, CA 95131

6345 Balboa Blvd., Ste. 100
Encino, CA 91316
818-609-9981
FAX 818-609-0535

30 Corporate Perk, Stes. 100-102
Irvine, CA 92714
714-250-4900
FAX 714-250-9041

FLORIDA
2200 Park Central N., Ste. 800
Pompano Beach, FL 33064
305-971-0404
FAX 305-971-2086

GEORGIA
2400 Pieasant Hill Rd., Ste 200
Ouluth, GA 30136
404-476-8574
FAX 404-478-3790

ILLINOIS
3100 Higgins Rd., Ste. 155
Hoffman Estates, IL 60195
708-884-0500
FAX 708-884-9394

MARYLAND
124 Maryland Rte 3 N.
Millersville, MO 21108
301-987-8777
FAX 301-987-8779

MASSACHUSETTB
261 Ballardvale St.
Wilmington, MA 01887
508-658-9501
FAX 508-658-0423

NEW JERBEY
311C Enterprise Or.
Plainsboro, NJ 08536
809-799-5700
FAX 609-799-5720

TEXAS
850 E. Arapaho Rd., Ste. 270
Richerdson, TX 75081
214-231-8716
FAX 214-669-1413

WASHINGTON
405 114th Ave. SE, Ste. 300
Bellevue, WA 98004
206-453-5414
FAX 206-453-5229

FRANCE
2, Allee des Gareys
F-91124 Palaiseau Cedex
Frence
1-8447.04.79
TELEX visitr 800 759 F
FAX 1-6447.04.80

GERMANY
Rosenkavellerplatz 10
O-8000 Muenchen 81
West Germany
89-9269050
TELEX 521 4279 visid
FAX 89-92690545

HONG KONG
Shui On Centre 28/12
8 Harbor Road
Hong Kong
852-5-865-3755
FAX 852-5-865-3159

JAPAN
Shuwa-Kioicho TBR 8idg., Room 101
5-7 Kojimechi, Chiyoda-Ku
Tokyo, Japan 102
81-3-239-5211
FAX 81-3-239-5215

UNITED KINGDOM
486-488 Midsummer Blvd.
Saxon Gate West, Central Milton
Keyes, MK9 2EO
United Kingdom
09 08/86 75 95
TELEX visiuk 825 135
FAX 09 08/67 00 27

VLSI SALES OFFICES

ALABAMA
2814 Artie St., Ste. 36
Huntsville, AL 35805
205-539-5513
FAX 205-538-8822

CONNECTICUT
60 Church St., Ste. 16
Yalesville, CT 08492
203-265-8698
FAX 203-265-3653

FLORIDA
5955 T. G. Lee Blvd., Ste. 170
Orlando, FL 32822
407-240-9603
FAX 407-240-9605

MINNESOTA
5871 Cedar Lake Rd., Ste. 9
St. Louis Park, MN 55416
612-545-1490
FAX 612-545-3489

NORTH CAROLINA
1000 Park Forty Plaza, Ste. 300
Ourham, NC 27713
919-544-1891/92
FAX 919-544-6667

OHIO
4 Commerce Park Sq.
23200 Chagrin Blvd., Ste.600
Cleveland, OH 44122
216-292-8235
FAX 216-464-7609

OREGON
10300 S.W. Greenburg Rd., Ste. 365
Portland, OR 97223
503-244-9882
FAX 503-245-0375

TEXAS
9800 Great Hills Trail, Ste. 150W
Austin, TX 78759
512-343-8191
FAX 512-343-2759

VLSI AUTHORIZED DESIGN CENTERS

COLORADO
SIS MICROELECTRONICS, INC.
Longmont, 303-778-1887

MAINE
QUAOIC SYSTEMS, INC.
South Portland, 207-871-8244

PENNSYLVANIA
INTEGRATEO CIRCUIT SYSTEMS, INC.
King of Prueeie, 215-265-8690

EIRE AND U.K.
PA TECHNOLOGY
Herts, 78-3-81222

FRANCE
CETIA
Toulon Cedex, 9-42-12005

SOREP
Chateaubourg, 99-623955

NORWAY
NORKRETS AS
Oslo, 47-2360877/8

SWEDEN
NOROISK ARRAYTEKNIK AB
Solna, 8-734 99 35

VLSI SALES REPRESENTATIVES

CALIFORNIA
CENTAUR CORP.
Irvine, 714-261-2123

CENTAUR CORP.
Calabasas, 818-704-1855

CENTAUR CORP.
San Oiego, 619-278-4950

EMERGING TECHNOLOGY
San Jose, 408-283-9368

EMERGING TECHNOLOGY
Orangevale, 918-988-4387

COLORADO
LUSCOMBE ENGINEERING
Longmont, 303-772-3342

IOWA
SELTEC SALES
Cedar Rapids, 319-364-7860

MARYLAND
OELTA III
Columbia, 301-730-4700

NEW YORK
bbd ELECTRONICS
Rochestar, 716-425-4101

OREGON
MICRO SALES
Beaverton, 503-645-2841

UTAH
LUSCOM8E ENGINEERING
Salt Lake City, 801-565-9885

WASHINGTON
MICRO SALES
Bellevue, 206-451-0568

ISRAEL
ROT ELECTRONICS
Tel Aviv, 3-483211-9

SINGAPORE
OYNAMIC SYSTEMS PTE, LTO
Singapore, 011-65-742-1986

VLSI DISTRIBUTORS

United Stetes repreeented by **SCHWEBER ELECTRONICS** except where noted

ALABAMA
Huntsville, 205-895-0480

ARIZONA
Tempe, 602-431-0030

CALIFORNIA
Calabasas, 818-880-9688
Irvine, 714-863-0200
Sacramento, 916-364-0222
San Diego, 619-495-0015
San Jose, 408-432-7171

COLORADO
Englewood, 303-799-0258

CONNECTICUT
Oxford, 203-264-4700

FLORIDA
Altamonte Springs, 407-331-7555
Pompano 8each, 305-977-7511
Tampa, 813-541-5100

GEORGIA
Norcross, 404-449-9170

ILLINOIS
Elk Grove Village, 312-569-3650

IOWA
Cedar Rapids, 319-373-1417

KANSAS
Overland Park, 913-492-2921

MARYLAND
Gaithersburg, 301-596-7800

MASSACHUSETTS
Bedford, 617-275-5100

MICHIGAN
Livonia, 313-525-8100

MINNESOTA
Eden Praire, 612-941-5280

MISSOURI
Earth City, 314-739-0526

NEW HAMPSHIRE
Manchester, 603-625-2250

NEW JERSEY
Fairfield, 201-227-7880

NEW YORK
Rochester, 718-424-2222
Westbury, 518-334-7474

NORTH CAROLINA
Raleigh, 919-876-0000

OHIO
8eachwood, 216-464-2970
Dayton, 513-439-1800

OKLAHOMA
Tulse, 918-622-8003

OREGON
ALMAC ELECTRONICS CORP.
Beaverton, 503-629-8090

PENNSYLVANIA
Horsham, 215-441-0600
Pittsburgh, 412-963-6804

TEXAS
Austin, 512-339-0088
Dallas, 214-247-6300
Houston, 713-784-3600

WASHINGTON
ALMAC ELECTRONICS CORP.
8ellevue, 206-643-9992
Spokane, 509-924-9500

WISCONSIN
New Berlin, 414-784-9020

AUSTRALIA
ENERGY CONTROL
Brisbane, 61-7-376-2955

AUSTRIA
TRANSISTOR GmbH
Vienna, 222-8294010

BELGIUM AND LUXEMBURG
MCAtronix
Angleur, 41-674208

DENMARK
INTERELKO
Karislunde, 3-140700

EIRE AND U.K.
HAWKE COMPONENTS
Sunbury-on-Thames, 1-9797799

OUARNDON ELECTRONICS
Oerby, 332-32651

FINLAND
OY COMOAX
Helsinki, 0-670277

FRANCE
ASAP s.e.
Montigny-le-Bretonneux, 1-3043.82.33

GERMANY
OATA MßOUL GmbH
Munich, 89-4180070

SPEZIAL-ELECTRONIC KG
Bueckeburg, 5722-2030

HONG KONG
LESTINA INTERNATIONAL, LTO
Tsimehatsui, 852-3-7351738

ITALY
INTER-REP S.P.A.
Torino, 11-2165901

JAPAN
ASAHI GLASS CO. LTO
Tokyo, 81-3-218-5854

TEKSEL COMPANY, LTO
Tokyo, 81-3-461-5311

TOKYO ELECTRON, LTO
Tokyo, 81-423-33-8009

KOREA
ANAM VLSI OESIGN CENTER
Seoul, 82-2-553-2106

EASTERN ELECTRONICS
Seoul, 82-2-464-0399

NETHERLANDS
OIOOE
Houten, 3403-91234

SWEDEN AND NORWAY
TRACO AB
Farsta, 8-930000

SOUTH AMERICA - BRAZIL
INTERNATIONAL TRAOE
OEVELOPMENT
Palo Alto, 415-856-6686

SPAIN AND PORTUGAL
SEMICONDUCTORES s.a.
Barcelona, 3-217-23 40

SWITZERLAND
FABRIMEX AG
Zurich, 1-2 5129 29

TAIWAN
PRINCETON TECH CORP.
Taipei, 886-2-717-1439

11/89

9

Notes:

The VLSI Technology, Inc. family of Reduced Instruction Set Computer (RISC) components can perform (and in some cases, outperform) the functions of comparable, conventional microprocessor systems with fewer and smaller components—at a much lower cost. This VLSI RISC-based system also permits a high degree of flexibility, allowing programs originally written for different conventional microprocessors to run without software modification.

This manual contains the hardware and software information necessary to understand and design a highly competitive RISC-based system, using the VLSI Technology, Inc., VL86C010 RISC microprocessor, the VL86C020 RISC with Cache, and their peripheral devices. Hardware and software examples are used extensively throughout the book.

Section 1 illustrates the RISC system solution for desktop computers. Sections 2 and 3 focus on both the hardware and software aspects of the RISC microprocessors. Instruction sets are thoroughly explained, using real-world examples. Section 4 investigates the RISC Memory Controller (MEMC) and its functions in detail. Similar treatment is given to the RISC Video Controller (VIDC) and RISC Input/Output Controller (IOC) in Sections 5 and 6. Section 7 defines the development tools currently available for the system. Section 8 contains the mechanical packaging information.

PRENTICE HALL, Englewood Cliffs, N.J. 07632

ISBN 0-13-781618-9