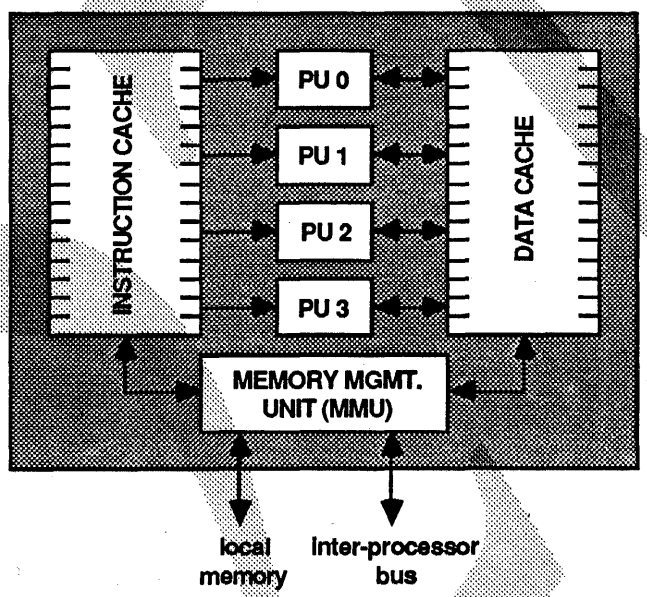
The Antares CPU



An Overview

Revision C
Feb. 6, 1987

The Antares CPU

An Overview

Contents

1. Introduction

1.1 Antares

The Antares CPU (or, simply, Antares) is the processing element of the Id personal computer system, a high-performance personal computer with high-resolution 3D color graphics. This note provides an introduction to the architecture and design of Antares; a detailed specification of the Antares instruction set is provided in a companion document, the Antares Instruction Set Reference Manual.

A typical Id computer system[1] is a multiprocessor comprising several Antares CPUs: one or more CPUs may be assigned to graphics processing, while others are used for application and system processing. These CPUs are connected via an inter-processor bus (IPB); each CPU has its own (local) memory, and can access the (remote) memories of other CPUs via the IPB (Figure 1.1). CPUs coordinate their processing activities via IPB messages. The IPB also connects to a NuBus interface. Both CPU–Memory buses and the IPB are 32-bits wide, and can transfer data at a maximum rate of 32 bits per cycle. The memory of one of the CPUs in the system uses video RAMs to provide an interface to the video subsystem. Higher video rates can be provided by using multiple CPUs, each driving a section of the screen.

Antares (Figure 1.2) is a parallel processor comprising 4 independent and identical 32-bit Processing Units (PUs) which share an instruction cache and a data cache. An on-chip Memory Management Unit (MMU) performs virtual-to-real address translation, initiates and controls transfers between the CPU and local or remote memories, and handles inter-CPU messages. The MMU provides a flat (unsegmented) virtual address space of 1024 million words (4 gigabytes), and accommodates a real memory size of 64 million words. The instruction and data caches are identical: each has a capacity of 4096 bytes, organized as 64 lines of 16 words (64 bytes). Antares caches are architecturally visible: instructions are provided to prefetch, create, flush, and invalidate cache lines.

[1] A minimum system with monochrome display can be constructed with a single Antares CPU.

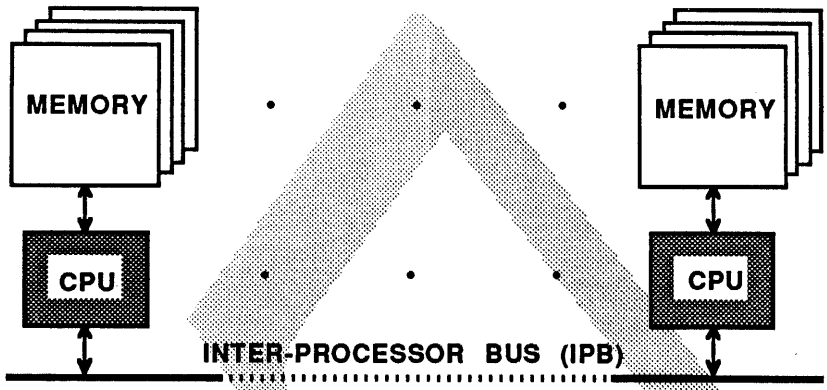


Figure 1.1. Id Multiprocessor System

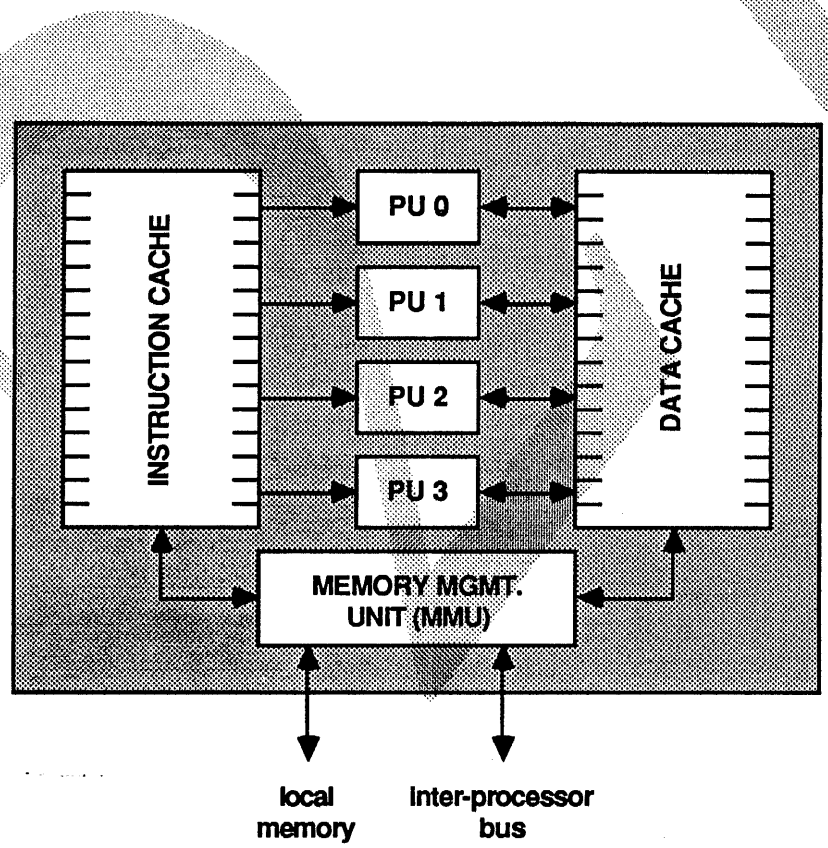


Figure 1.2. Major Elements of the Antares CPU Chip

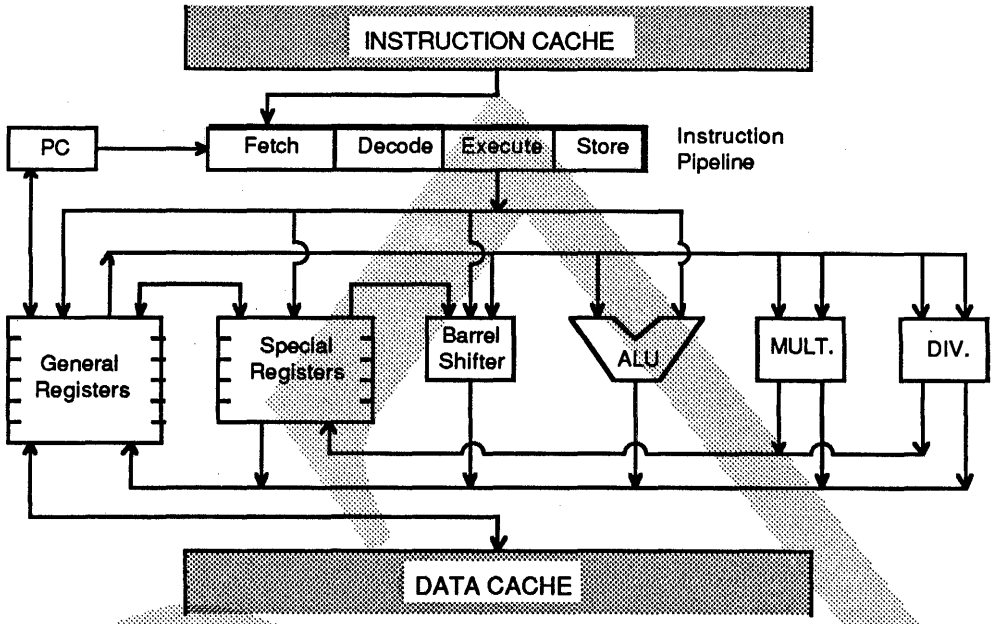


Figure 1.3. Major PU Components

Figure 1.3 shows a simplified diagram of an Antares PU. While all four PUs share access to the same address space, they are otherwise independent. Each PU has its own program counter (PC), instruction pipeline, registers, and arithmetic units, so each PU is capable of executing a different instruction stream. Each PU has 16 private, 32-bit, general-purpose registers (R0 — R15), a private set of special registers (including status and save registers), a full shifter, a 32-bit ALU, and independent multiply and divide units. All 16 general registers can be used to hold data or addresses. Registers R0 — R3 are used as base registers in standard-format base plus displacement mode addressing, and register R4 is used as the link register for branch and link instructions. PUs have a small, register-oriented instruction set in which only load and store instructions access memory and in which most instructions execute in one cycle. Broadcast and semaphore operations are provided to coordinate activities executing on different PUs.

1.2 Parallel Processing

The objective of the Antares design project is the development of a high-performance, single-chip CPU. Given a technology which will provide over a half million transistors on a chip, how can this "real estate" best be exploited to achieve this objective? The primary ingredients of a recipe for a fast, general-purpose, CPU are "big cache, small cycle time", so a large part of the available real estate is allocated for an on-chip cache (Figure 1.4b). To achieve a small cycle time, the processor (PU) implements a simple, general-purpose instruction set; also, for both cost and performance reasons, an on-chip Memory Management Unit (MMU)

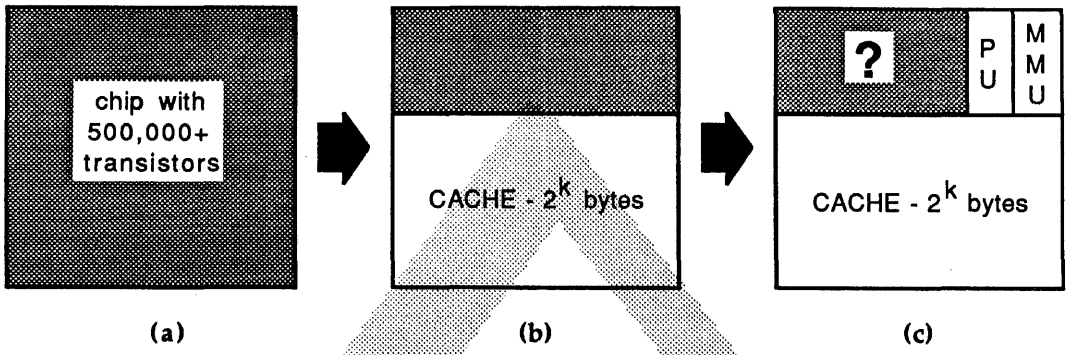


Figure 1.4. Allocating Real Estate

is provided (Figure 1.4c). The question then becomes one of determining how best to use the remaining chip space.

There are several alternatives. The processor design can be extended to provide more instructions and more addressing modes, but it is not clear that this will yield a significant improvement in performance, and the added complexity may even hurt performance by increasing cycle time. A floating point unit could be added, but this would only help floating point applications. Similarly, adding a graphics processor provides a performance improvement for only part of the system's workload. However, adding more processors — three more, for a total of four, in the Antares case — and appropriate facilities for coordinating parallel execution, provides the potential for a substantial performance improvement in all applications. No other alternative offers a potential performance improvement of 4 times (4X) that of the single processor; even if the average improvement is only 2X, no other design alternative offers a comparable across-the-board improvement. The challenge for Antares software development is to realize this potential.

Antares programs can execute in any of several modes of parallel execution. These modes are categorized using (with some liberties) the taxonomy developed by Flynn [1972].

SISD (single instruction stream, single data stream). This mode is uni-, or serial, processing: only one PU executes. Antares typically alternates between intervals of serial and of parallel processing: a single PU initiates (and often participates in) a set of parallel computation activities, and later may accumulate the results of these activities.

SIMD (single instruction stream, multiple data streams). This mode corresponds to the usual view of parallel processing: each PU executes the same operation on different data streams, as illustrated in Figure 1.5, or on different elements of the same data stream. Data access may be ordered or random. In ordered access, inter-PU coordination is implicit, as when each PU operates on every fourth element of a vector. In random access, explicit inter-PU coordination

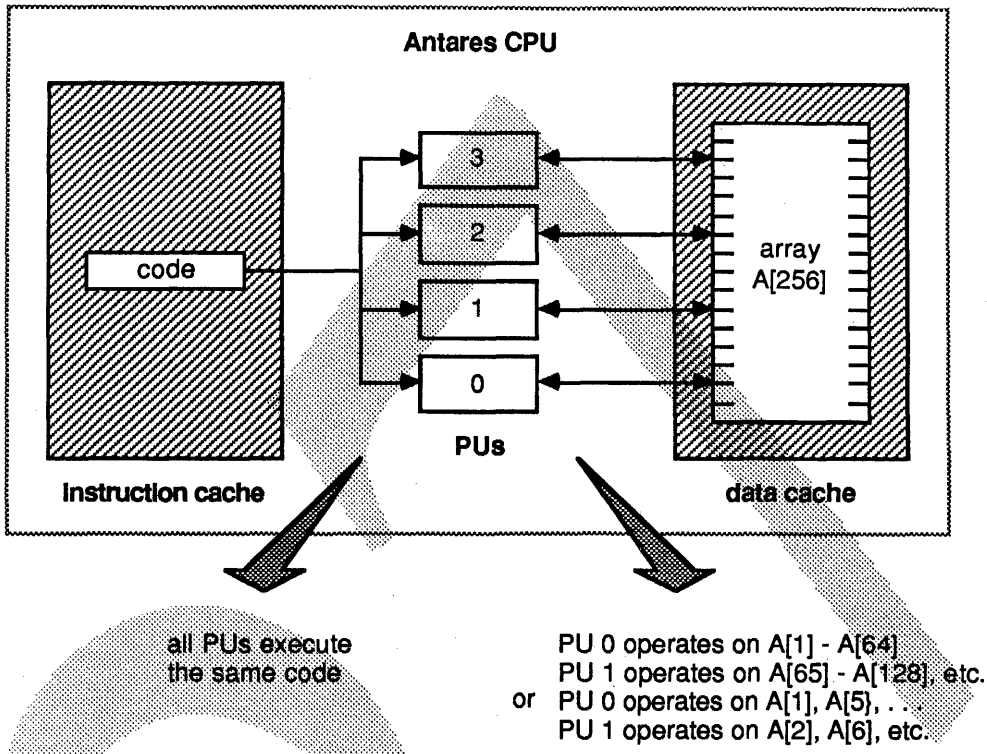


Figure 1.5. SIMD Mode Execution

is required, as when PUs operate concurrently on a linked list or take work from a queue. This coordination is implemented using the Antares semaphore mechanism, which is described in Section 4. This is the easiest form of parallelism to exploit, either with assembly code or by a compiler. For example, the compiler may be able to "unwind" a loop which operates on an array to run on 4 PUs, with each PU operating on every 4th array element. Optimal performance is easily obtained, since all PUs are doing the same work.

As an example of **SIMD** mode execution, consider the common graphics transformation operation (used in scaling, rotation, and translation) which involves the 1 x 4 matrix multiplication

$$[x^* \; y^* \; z^* \; w^*] = [x \; y \; x \; w] \times \begin{vmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{44} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{vmatrix}$$

where

`[x y x w]` = original coordinate set,

`[x* y* z* w*]` = transformed coordinate set,

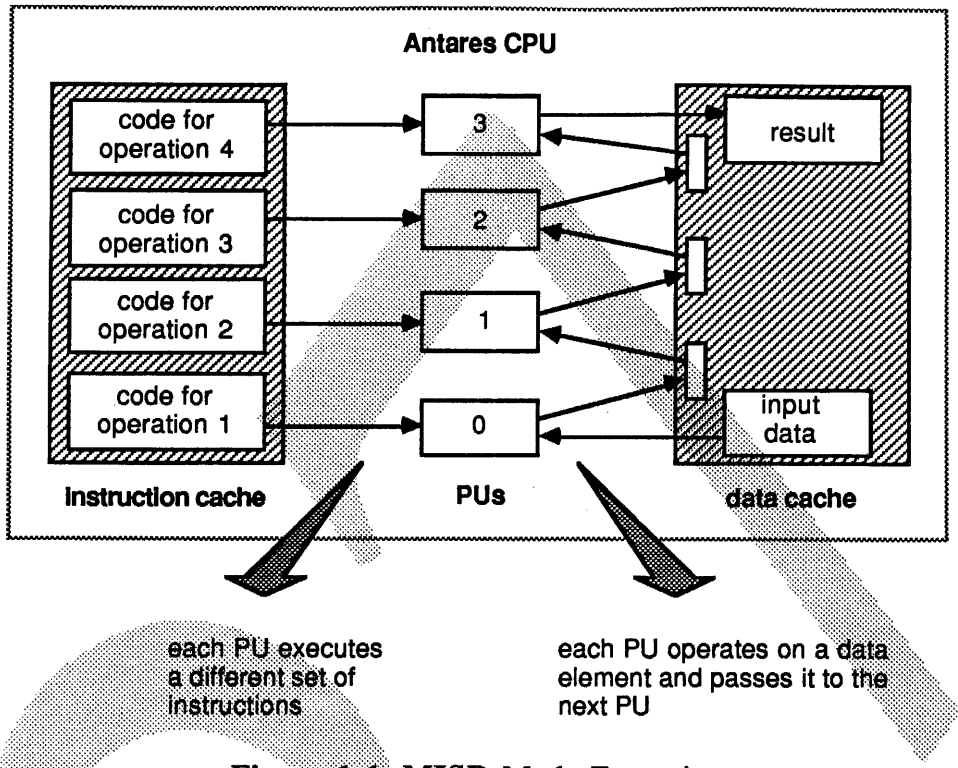


Figure 1.6. MISD Mode Execution

and the

c_{ij} are fixed (pre-computed) for any given transformation. (For any particular transformation, some of the c_{ij} are known to be 0 or 1.)

The matrix product can be written as

$$x^* = xc_{11} + yc_{21} + zc_{31} + wc_{41}$$
$$y^* = xc_{12} + yc_{22} + zc_{32} + wc_{42}$$
$$z^* = xc_{13} + yc_{23} + zc_{33} + wc_{43}$$
$$w^* = xc_{14} + yc_{24} + zc_{34} + wc_{44}$$

In a parallel (**SIMD** mode) implementation of this transformation, PU 0 can be assigned to compute **x***, PU 1 to compute **y***, and so on. Each PU preloads its registers with the appropriate set of constants and, after each *nth* transformation, each PU executes a cache prefetch instruction (Section 3) to prefetch the next line of coordinate data. (Only one prefetch actually takes effect.) By careful scheduling of prefetch and computation operations, very high transformation rates can be realized.

MISD (multiple instruction streams, single data streams). In this mode, each PU executes a different operation on the same data stream element; data is "pipelined" between PUs (Figure 1.6). For example, consider the computation of

$$y = ax^3 + bx^2 + cx + d$$

which might be divided across PUs as follows.

- PU 0: read x, compute $= cx + d$, store x, $cx + d$
- PU 1: compute and store x^2, bx^2
- PU 2: compute $x^3 = x(x^2)$, compute and store ax^3
- PU 3: sum intermediate results to form y and store y

Coordination of these operations would be done via semaphores: for example, one semaphore might be used to transmit x from PU 0 to PU 1, another to transmit x^2 from PU 1 to PU 2, and so on.

This is the hardest form of parallelism to code or for which to compile code, and it is not trivial to balance PU execution times so as to optimize performance. Interpretive programs, such as a Smalltalk byte code interpreter or a 68000 simulator, can use carefully-crafted, hand-coded **MISD** processing to achieve good performance. However, more loosely connected variations of this form are useful in realizing improved (relative to uniprocessing), if not optimal, performance. For example, in processing a linear list, one PU might be assigned to buffer in — prefetch— lines of the list while another PU carries out list element processing.

MIMD (multiple instruction streams, multiple data streams). This mode is analogous to multiprocessing: each PU executes a different and independent set of instructions which operate on different and independent data elements (Figure 1.7). These might correspond to independent expressions within a single statement or to independent statements. It is easy to exploit this form of parallel execution at the assembly code level, and it is not too difficult for the compiler to generate MIMD code. However, it may be hard for the compiler to determine independence (because of pointers, for example), and it also can be hard to obtain optimal performance (allocate comparable work to each PU). The independence problem can be eased somewhat by the use of compiler directives to identify program units which can be executed in parallel.

parallel activity boundaries. At present, parallel execution of compiled code on Antares is not expected to cross procedure boundaries; execution will be serialized (constrained to **SIMD** mode) at procedure call and return points, so that the compiler will not have to maintain multiple stacks. It is possible that certain exceptions may be made (e.g., independent, non-recursive, leaf procedures identified by compiler directive), and critical graphics system and operating system operations may be hand-coded to obtain maximum performance. No explicit support is provided for multi-tasking within an address space (i.e., "light-weight" processes). However, a user state task can execute in parallel with the kernel; an external interrupt will be assigned to an idle PU, if available, so that processing of the interrupt can be done while a user task continues in execution.

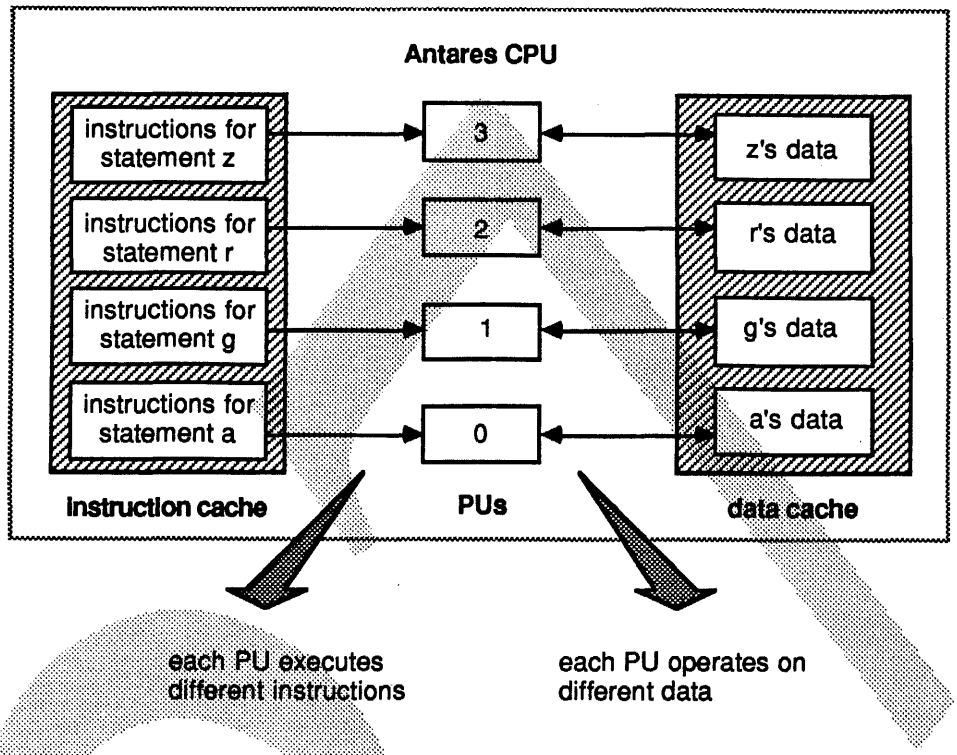


Figure 1.7. MIMD Mode Execution

1.3 Parallelization and Amdahl's Law

One of the highlights of the 1967 Spring Joint Computer Conference was a debate[2] on the subject "The Best Approach to Large Computing Capability" in which proponents of parallel processors (including D. Slotnick of Illiac IV fame) and fast uni-processors argued their respective cases. In that debate, Gene Amdahl [1967] pointed out that the performance of a system with two modes of operation, one high speed and one low speed, will be dominated by the low speed mode. These modes can correspond to vector and scalar operation sequences on a vector computer, or parallel and serial sequences on a parallel computer. This postulate has come be called "Amdahl's Law". A very readable and entertaining discussion of Amdahl's Law is presented by Worlton [1981].

To illustrate, suppose that a workload executes in time T on a single Antares processor (Figure 1.8a). Assume that one-half of this workload can be parallelized to run on 4 PUs, so that execution of this part of the workload is speeded up by a factor of four; the execution time of the other half of the workload is unchanged (Figure 1.8b). The workload execution time reduces to 5T/8, which represents an

[2]This came to be called "the great debate", and was one of the early skirmishes between the *unis* and the *multis*.

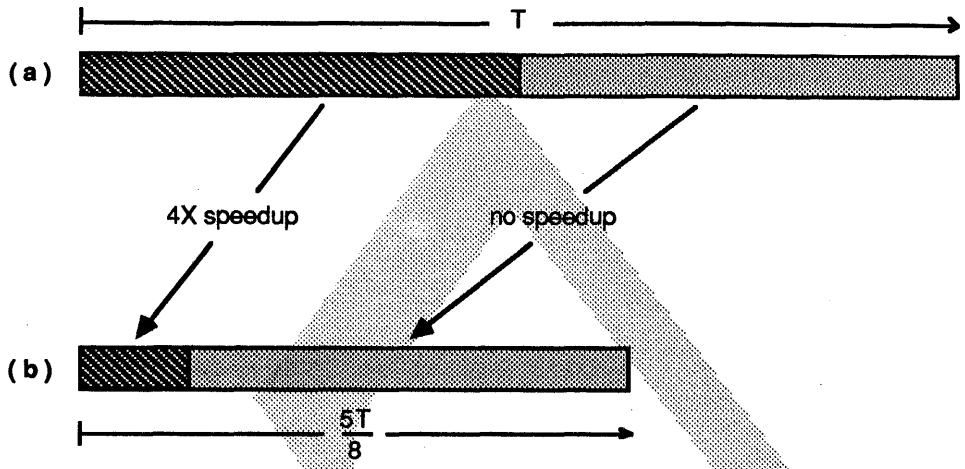


Figure 1.8. An Illustration of Amdahl's Law

overall improvement of only 1.6X. (If execution of the parallelizable part was speeded by a factor of 100, the overall improvement still would be less than 2X.)

Amdahl's Law sometimes is used to downplay the effectiveness of parallel processing. While the potential n-fold improvement of an n-processor system may not be realizable, there are other considerations:

- When single processor performance has been as far as possible using available technology, increased performance for a single workload can be obtained only through parallel processing. While this, as stated, is an over-simplification, it is the rationale for the Cray X-MP as well as for Antares.

- Parallel processing can be effective, even if only part of the workload can be parallelized, if that part is time-critical. In the Id system, for example, the performance achieved via parallelization of the graphics pipeline is crucial in meeting graphics requirements.

Initial versions of Antares software will rely on explicit parallelization to optimize performance of key components; some components will be coded in assembly language, and compiler directives will be used to identify parallelizable code sections to the compiler. These methods will supplement the parallelization done implicitly by the compiler. Improved parallelization should be achieved as experience is gained in exploiting parallelism in software design and as compiler technology evolves. Thus, continuing gains in performance are expected over time. Note the improvement obtainable if the serial part of the workload of Figure 1.8 could be speeded up by just a factor of two .....

2. The Instruction Set

2.1 Instruction Set Design

The Antares PU represents a form of RISC[1] architecture: it has a small, register-oriented, hardwired instruction set in which only load and store instructions access memory and most instructions execute in one cycle. These characteristics simplify instruction decoding and execution control, help reduce cycle time, and make it possible to place 4 PUs, with caches and MMU, on a single chip.

In Antares, almost all instructions are 16 bits — a half word — in length. This differentiates Antares from other RISC processors, in which all or almost all instructions are 32 bits in length. RISC processors use several simple instructions to synthesize operations (such as a string move) which can be done with a single instruction on a CISC[2] processor. This frequently has no direct performance consequence: the operation takes about the same number of cycles in both cases. However, the reduced instruction density of the RISC processor indirectly affects performance by increasing the instruction cache miss rate (or, equivalently, instruction bandwidth). Hennessy [1985] reports a benchmark comparison of the Stanford MIPS processor and the Motorola 68020: the static code size for the MIPS processor was 40% greater than that for the 68020, and the instruction bandwidth was 20% greater. (However, the MIPS processor used only 1/4 as many cycles as the 68020.) Instruction density is of particular concern to Antares, since 4 PUs share the instruction cache and these PUs will not necessarily be executing the same code. Analysis of static and dynamic instruction frequencies shows that a 32-bit instruction length is longer than necessary for many of the most frequent instructions. A significant improvement in instruction density can be achieved by using a 16-bit instruction length or by variable-length instructions (for example, the Fairchild Clipper has 16-, 32-, 48-, and 64-bit instructions). A 16-bit

[1]Reduced Instruction Set Computer: see, for example, Hennessy [1984], [1985], or Patterson [1985]

[2]Complex Instruction Set Computer

standard instruction format was chosen for Antares because of the hardware cost and complexity of handling variable-length instructions. Antares does provide *extended format* load and store instructions in base plus displacement addressing mode to reduce synthesis costs in certain addressing situations.

A second and related motive for a short instruction format, and for the choice of a number of Antares instructions, is the efficient synthesis of higher-level opera - tions. Rather than directly provide floating point arithmetic instructions (unfeasible in any case, at least at present, because of "real estate" limitations) or instructions to facilitate Smalltalk byte code interpretation, Antares tries to provide compact, fast, basic instruction which will permit efficient synthesis of these and other specialized operations.

A short instruction length format restricts the length of immediates, displace - ments, and direct addresses, and requires careful instruction set design and encoding. The Antares instruction set may appear irregular[3] because of its encoding and because displacement and immediate fields vary in length according to their use. However, field sizes have been selected to best match their function and frequency. For example, Antares has 8-bit immediate fields, providing an immediate value range of 1–256, in add, load, and compare immediate instructions (subtract immediate has only a 4-bit immediate field). Studies[4] have shown that about 70% of immediate values are in the range 0–16, about 95% are in the range 0–256, most are positive, and a large proportion of the immediates in the range 16–256 represent character constants (primarily used in loads or compares). A source-code-level study of "CadMac" showed that 73% of its constants (including constant array indices) are in the range 0–16. Thus, most immediates can be specified by a single Antares instruction.

Most long constants are address constants; these, in Antares, are stored as data rather than being imbedded in the instruction stream. For example, the addresses of a procedure's callees may be stored with that procedure's static local variables, or placed in a common directory and accessed via direct addressing. This improves instruction stream density and facilitates prefetching. Similar arguments apply to the (conditional) branch displacement value of ± 256 instructions, to the data address displacement of 64 words[5] for standard-format base plus displacement addressing, and to other instruction set parameters. To the greatest extent possible, instructions are designed so that high-frequency operations can be executed with a single instruction: lower-frequency operations are synthesized by instruction sequences which, because of the short instruction length, tend to require relatively small amounts of instruction space.

---

[3] HP uses the term "precision" to describe the tightly-encoded functional architecture of the HP Spectrum line (Birnbaum and Worley [1985]).

[4] See, for example, Hennessy et al [1982]

[5] A choice based on, among other factors, stack size frequency distributions

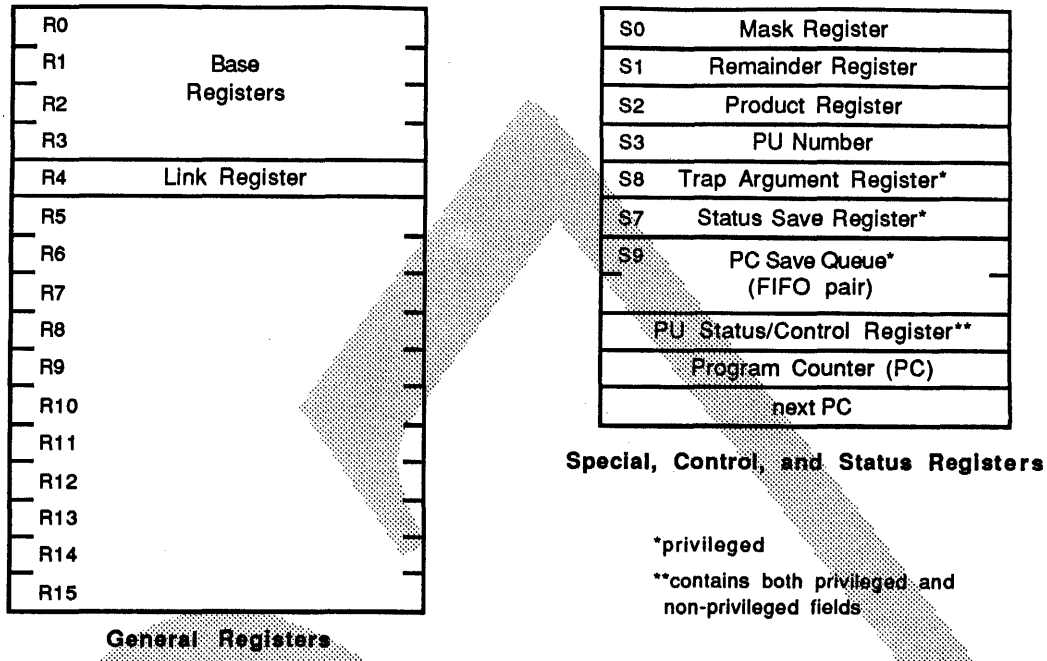


Figure 2.1. PU (Local) Registers

2.2 Programming Model

The Antares programmer deals with a flat (unsegmented) address space of 1024 million words (4096 million bytes), a collection of registers, and a set of instructions whose functions include transferring data between memory and registers and operating on the contents of registers. (The address space model is discussed in Section 5.) Each PU has a set of local registers which, except for broadcast operations, can be accessed only by that PU, and there is a set of global registers which are accessed by all PUs.

The local registers of a PU are shown in Figure 2.1. There are 16 general-purpose registers, R0-R15. All transfers from and to memory are performed by general register load and store instructions. Registers R0-R3 can be used as base registers in standard-format base plus displacement addressing; all 16 registers can be used as base registers in extended-format base plus displacement addressing. Register R4 is used as the link, or return address, register by jump and link instructions.

In addition to general registers, each PU has a set of special, control, and status registers. There are seven local special registers (S0-S3 and S7-S9) and eight global special registers. Special register contents can be read and, in certain cases, written, by Move Special instructions, which transfer data between general and special registers. Values also may be written to special registers as the result of executing other instructions. Access to some special registers is privileged, and can

be effected only in system state. A brief summary of the functions of the local special registers[6] follows; for details, see the Antares Instruction Set Manual.

- **Mask Register** (S0). This register is an implicit operand register of bit field manipulation instructions; it is set with the field position and length by the mask (MSK) instruction.
- **Remainder Register** (S1). A divide instruction stores the remainder in this register.
- **Product Register** (S2). Multiplication of two 32-bit numbers produces a 64-bit product; the high-order bits are stored in this register.
- **PU Number** (S3). A PU can read its number from this hardwired register.
- **Trap Argument Register** (S8). Certain trap operations, such as a data page fault, store an argument in this register.
- **Status Save Register** (S7). On the transfer of control caused by an interrupt or trap, the contents of the PU Status/Control Register (described below) are stored in this register. When a Return From Interrupt instruction is executed, the contents of this register are transferred to the PU Status/Control Register.
- **PC Save Queue** (S9). This is a FIFO register pair. On the transfer of control caused by an interrupt or trap, the contents of the current Program Counter (PC) and next Program Counter are stored in these registers. On returning control after processing the interrupt, the kernel executes a pair of Return From Interrupt instructions to restore the PU Status/Control Register from the Status Save Register and restore the current and next PCs from the PC Save Queue.

Two PCs (current and next) and two PC save registers are required because of the delayed branch instruction execution in Antares (discussed later in this section).

The PU Status/Control Register contains PU status and control information. Figure 2.2 shows this register and identifies certain fields of interest in this overview. Mode bits control various aspects of PU operation, and are set and cleared by Set Mode and Clear Mode instructions. Some mode bits may be modified by a PU operation; for example, execution of a trap instruction clears the user mode and trap enable bits. Antares can perform arithmetic on 8-, 16-, and 32-bit operands, so the condition code field contains 4 carry bits as well as Zero, Negative, and Overflow bits. Flags bits are set during certain PU operations. The register count field is used to store the register count of a load/store multiple

[6]Special register lengths can vary according to function; the numbering of Special Registers may be revised to help decoding.

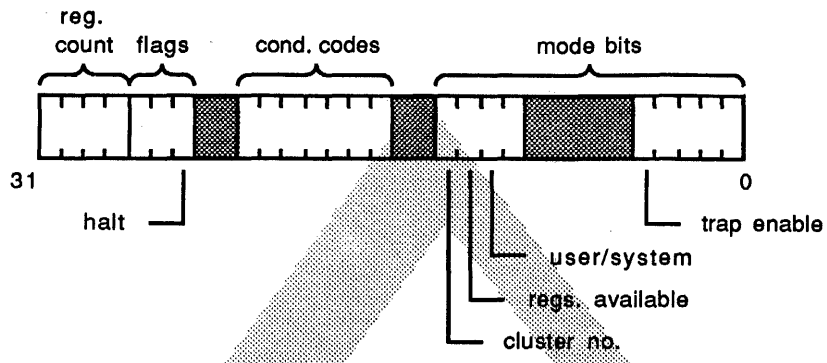


Figure 2.2. PU Status/Control Register

instruction interrupted by a page fault. The "registers available" mode bit and the halt flag are discussed in Section 4. The cluster number mode bit is used in direct address generation, and is described later on.

The single set of global registers (Figure 2.3) is accessible to all four PUs via Move Special instructions. The Test Register is used by diagnostics. The active member of the Semaphore Flag and Prefix Address Register pair is specified by a cluster number. Semaphores are discussed in Section 4.3; the Prefix Address is used in direct address generation (described later in this section). The Event Counter and Selection Registers provide facilities for accumulating performance measures such as cache misses, instructions executed, and PU utilization. The Global Status Register reflects the composite state of each PU (user/system, enabled/disabled, halted, waiting, or running), and is used in deadlock detection (Section 4.4). The Interrupt Argument Register holds an argument accompanying an external interrupt (Section 6). The PTDO and Node No. Register contains the Page Table Directory Origin (Section 5) and the node number of the CPU.

2.3 Addressing and Addressing Modes

Instruction and data addresses in Antares are 32 bits in length. Instruction addressing is in instruction-length units: a relative instruction address (displace - ment) represents a half-word increment, and an absolute instruction address (PC contents or absolute jump address) is a half-word memory address. Instructions are assumed be be aligned on half-word boundaries. Data addresses are word addresses for load and store word instructions, byte addresses for load and store byte instructions. There are three data addressing modes: register, base plus displacement, and direct.

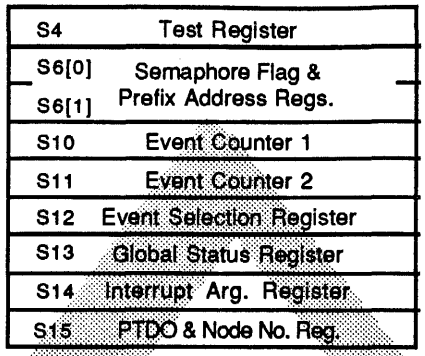


Figure 2.3. Global (CPU) Registers

register addressing. The operand address of a load or store word instruc - tion is the word address contained in bits 0-29 of the specified register: bits 30-31 are ignored. The operand address of a load or store byte instruction is the byte address contained in bits 0-31 of the specified register. Load and store byte instructions are provided only in this form, and are auto-incrementing; the byte address is incremented after the byte has been loaded or stored.

base plus displacement addressing. The operand address of a load or store word instruction is formed by adding the displacement field of the instruction to the contents of the specified base register, and using bits 0-29 of the result as the data word address. For standard-format instructions, the displacement field is 6 bits (displacement value range 1-64), and the base register must be R0, R1, R2, or R3. In extended format, the displacement field is 16 bits (displacement value range 1-65536) and any register may be used as the base register.

direct addressing. The operand address of a load or store word instruction is formed by concatenating the 8-bit displacement field with the prefix address from special register S6[i], where i represents the value of the cluster number in the PU Status/Control Register of the PU executing the instruction (Figure 2.4). The displacement field provides bits 0-7 of the address, S6[i] provides bits 8-29, and bits 30-31 are ignored.

The prefix address defines the start of a 256-word memory region which can be accessed by load and store direct instructions; this region is called *direct address space.* Separate direct address spaces are provided for user and system state, and both user and system can redefine their direct address spaces as desired. The first 8 locations of direct address space are *semaphore* locations; semaphore operations are performed by load and store direct accesses to these locations (see Section 4.3). Semaphore flags are kept with the prefix address in an S6 register and, if desired, can be changed when the prefix address is changed.

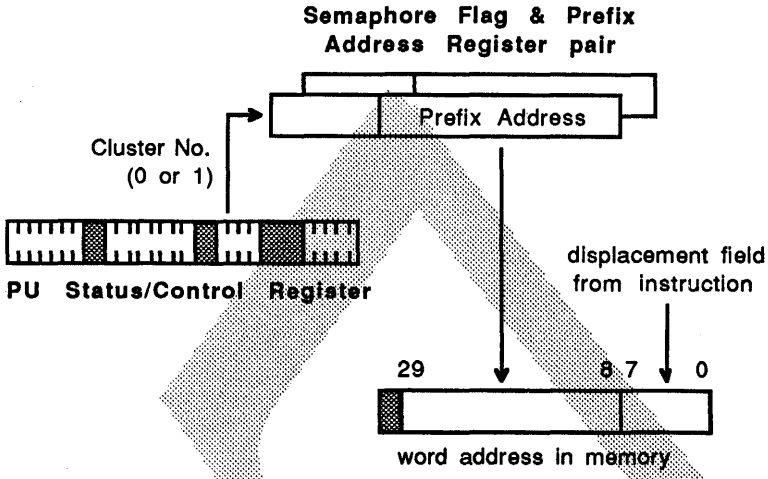


Figure 2.4. Direct Address Generation

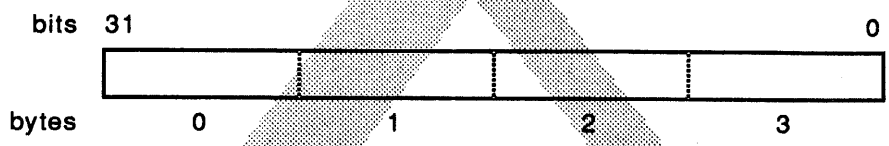
The CPU maintains two sets of semaphore flags and prefix addresses in the S6 register pair. By convention, the semaphore flags and prefix address in S6[0] are used in system state, and the semaphore flags and prefix address in S6[1] are used in user state. Selection is by cluster number. Each PU maintains a cluster number in its PU Status/Control Register; in the initial implementation of Antares, this number is represented by a single bit. The cluster number is assigned a value (0 or 1) by the kernel via Clear Mode and Set Mode instructions; cluster number modification is a privileged operation. The cluster number is forced to 0 when a PU initiates trap or interrupt processing, so the semaphore flags and prefix address used by the system are set by default to those contained in S6[0]. However, the kernel can change the cluster number and use S6[1] to access the user's direct address space.

Other addressing modes can be synthesized as needed: for example, base plus index addressing takes two instructions (and two cycles, but only 32 bits of instruction space).

The use of three different address types — word, half-word (instruction), and byte — makes assembly-level programming of Antares more difficult than would be the case if all addresses were byte addresses. However, multiple address types have performance advantages relative to byte addressing, and it is expected that most Antares programming will be done in a higher-level language; very few programmers will need to be aware of the different address types. One reason multiple address types are used is to make the most efficient use possible of the relatively small (in conventional view) immediate fields of Antares instructions. For example, if only byte addressing was provided, the 8-bit immediate field of the Add Immediate instruction would give an immediate range of only 1-64 for word increments, and the 4-bit immediate field of the Subtract Immediate instruction would give an immediate range of only 1-4 for word decrements. A second reason

is the elimination of index shifting for operations on word arrays which is required if byte addressing only is provided (e.g., @A[i] = @A[0] + i<<2).

data word format. The Antares data word format is shown below. This format is the same as that of the Motorola 68000: Little Endian for bits, Big Endian for bytes (to use the terminology of Cohen [1981]).



Bit 0 is the least significant bit, and bit 31 is the most significant bit for unsigned data or the sign bit for signed data. (Data word formats are further defined by the arithmetic mode specified.)

2.4 Instructions

The Antares instruction set, as currently defined, is summarized in Figure 2.5. The following notation is used in this figure.

Sr, Sr1, Sr2	source register(s)
Base	base/address register
Dst	destination register
Mask	Mask register
Sp	Special register
@R	address contained in R
Imm	immediate operand
Dsp	displacement or direct address
*	Program Counter

A detailed description of the instruction set appears in the Antares Instruction Set Reference Manual; only certain features are noted here.

Operation of most of the general-register load and store instructions was discussed in the preceding section. The **Lcc** (load boolean condition) sets a register to 1 if cc matches the current code and to 0 otherwise: it facilitates optimization of logical expressions and helps mitigate the restrictions on code reordering that condition codes usually impose. Antares provides byte arithmetic operations which operate concurrently on all four bytes of a word, and half-word arithmetic operations which operate concurrently on both half-words of a word (in addition to full-word arithmetic operations). A mode bit in the PU Status/Control Register determines if partial-word arithmetic operates on bytes or half-words. The condition codes in the PU Status/Control Register include four carry flags: 2 or 4 of these may be set as the result of a partial-word arithmetic or compare instruction. The **LDCP** instruction is used to load carry flags, extended to the current operand width, into a general purpose register. Load and store multiple instructions are provided to help keep procedure call and return overhead low. (While the cost of synthesizing these operations is not high in cycles, it is in terms of cache space.)

REGISTER LOAD, STORE, AND MOVE INSTRUCTIONS		
Lcc	->Dst	load boolean on condition
LD	Imm->Dst	load immediate
LDB	@Base->Dst	load byte & increment Base
LDCP	->Dst	load byte or halfword carries
LDM	@Base->Dst	load multiple registers Dst through 1
LDW	Dsp->Dst	load word (direct)
LDW	@Base->Dst	load word (register)
LDW	@Base+Dsp->Dst	load word (base + displacement)
LDW	@Base+Dsp->Dst	load word (base + extended displacement)
MOV	Sr->Dst	move register
MOV	Sp->Dst	move register: special to general
MOV	Sr->Sp	move register: general to special
STB	Sr->@Base	store byte & increment Base
STM	Sr->@Base	store multiple registers Sr through 1
STW	Sr->Dsp	store word (direct)
STW	Sr->@Base	store word (register)
STW	Sr->@Base+Dsp	store word (base + displacement)
STW	Sr->@Base+Dsp	store word (base + extended displacement)
ARITHMETIC INSTRUCTIONS		
ADCP	Sr1+Sr2->Sr1	add register (bytes or halfwords with carries)
ADD	Sr1+Sr2->Sr1	add register (word)
ADD	Sr+Imm->Sr	add immediate
ADDC	Sr1+Sr2->Sr1	add register (word with carry)
ADDP	Sr1+Sr2->Sr1	add register (bytes or halfwords without carries)
CLZ	Sr->Dst	count leading zeroes
DIV	Sr1/Sr2->Sr1	divide
MUL	Sr1*Sr2->Sr1	multiply (word)
MULP	Sr1*Sr2->Sr1	multiply (bytes or halfwords)
NEG	Sr->Dst	negate
SBCP	Sr1-Sr2->Sr1	subtract register (bytes or halfwords with carries)
SUB	Sr1-Sr2->Sr1	subtract register (word)
SUB	Sr-Imm->Sr	subtract immediate
SUBC	Sr1-Sr2->Sr1	subtract register (word with carry)
SUBP	Sr1-Sr2->Sr1	subtract register (bytes or halfwords: carries = 1)
TRANSFER AND COMPARE INSTRUCTIONS		
ADPC	*+1+Sr->Sr	add program counter
Bcc	*+Dsp	branch on condition
CMP	Sr1-Sr2	compare register (word)
CMP	Sr1-Imm	compare immediate
CMPP	Sr1-Sr2	compare register (bytes or halfwords)
JMP	*+Dsp	jump relative
JMP	@Sr	jump absolute
JMPL	@Sr	jump and link (return address -> reg. 4)
TSTF	Sr	test field under Mask
TSTM	Imm	test mode bit number Imm

Figure 2.5. Antares Instruction Set

SHIFT, LOGICAL AND FIELD MANIPULATION INSTRUCTIONS		
AND	Sr1&Sr2->Sr1	and
ANDC	Sr1&¬Sr2->Sr1	and complement
CLRF	Sr->Sr	clear field under (Mask)
DEP	Sr->Dst	deposit field under (Mask)
DSH	Sr2,Sr1	double shift under (Mask)
EXTS	Sr->Dst	extract field under (Mask) & sign extend
EXTU	Sr->Dst	extract field under (Mask)
INS	Sr->Dst	insert field under (Mask)
MSK	Imm1, Imm2	generate mask of length Imm2 at Imm1
MSK	Sr, Imm	generate mask of length Imm at (Sr)
NOT	~Sr->Dst	not (one's complement)
OR	Sr1|Sr2->Sr1	or
SETF	Sr->Sr	set field under (Mask)
SHL	Sr<<Amt->Sr	shift left logical
SHR	Sr>>Amt->Sr	shift right logical
XOR	Sr1^Sr2->Sr1	exclusive or
CACHE CONTROL INSTRUCTIONS		
CDC	@Sr	create data cache line
FDC	@Sr	flush data cache line
IDC	@Sr	invalidate data cache line
IIC	@Sr	invalidate instruction cache line
IICA		invalidate all instruction cache lines
PDC	@Sr	prefetch data cache line
PIC	*+Dsp	prefetch instruction cache line relative
PIC	@Sr	prefetch instruction cache line absolute
UDC	@Sr	update data cache line
VDC	@Sr	validate data cache line
BROADCAST AND CONTROL INSTRUCTIONS		
CLRM	Imm	clear mode bit number Imm
INT	PUmask	interrupt PUs specified by PUmask
ITLB		invalidate translation buffer
RDTX	@Base->Dst	read data cache tag of line indexed by (Base)
RES	PUmask	restart PUs specified by PUmask
RSM	PUmask	resume halted PUs specified by PUmask
RTI		return from interrupt
SEND	Sr->Dst of PUmask	send (Sr) to Dst of PUs specified by PUmask
SETM	Imm	set mode bit number Imm
STRT	@Base->* of PUmask	start halted PUs specified by PUmask at (Base)
TRAP	Imm	trap
WAIT	PUmask	halt, or wait for PUs specified by PUmask to halt

Figure 2.5 (continued). Antares Instruction Set

The address register of load multiple is incremented by the number of registers loaded, and the address register of store multiple is decremented by the number of register stored, to help in stack manipulation.

ADCP, ADDP, MULP, SBCP, and **SUBP** instructions operate on either byte or half-word operands, depending on the current setting of the partial-word arithmetic mode bit; other arithmetic instructions operate on full-word operands. Multiply and divide instructions execute asynchronously; the next sequential instruction is issued in the cycle following issue of the multiply or divide, and instruction issue and execution continues until an attempt is made to read the result register of the multiply or divide. At that point, the instruction attempting to read the result register blocks until the multiply or divide operation completes.

An operation such as field extraction requires 4 operands (source register, field length, field position, and destination register), which is beyond the capacity of the 16-bit instruction format. The Mask register (local Special Register 0) is an implicit operand of Antares bit field manipulation instructions: once it is set with the position and length of a field (by a **MSK** instruction), extract, insert, and test operations all can be done with single instructions. The Mask register also is an implicit operand register of the double shift instruction **DSH. DSH** is used to shift a 64-bit operand split between two source registers Sr1 and Sr2; **DSH** can be used to perform rotation by setting Sr1 and Sr2 to the same register number.

The cache miss rate and miss penalty (delay incurred as the result of a miss) are key factors in determining CPU performance. Antares provides a set of instruc - tions which can help reduce both the miss rate and the miss penalty; these are discussed in Section 3.5.

Broadcast instructions let one PU transmit data or control information simulta - neously to one, two, or three other PUs; the receiving PUs are designated by the PUmask field of the broadcast instruction. These instructions are described in Section 4.2. Set, clear, and test mode instructions operate on the mode bits in the PU Status/COntrol Register. **ITLB, RDTX,** and **IICA** instructions are used to flush the cache and Translation Buffer in task switching; these, and interrupt-related instructions, are discussed in Section 6.

2.5 Nominal Instruction Execution Times

The Antares instruction execution pipeline has four stages: fetch (F), decode (D), execute (E), and store (S). Four different instructions can be in different stages of execution in any cycle. A fifth stage, called store 2 (S2), is used in executing load and store instructions. An Antares PU issues one instruction per cycle unless the pipeline is blocked by a cache miss, a cache bank conflict or a pipeline interlock[7] delay, a register wait condition (e.g., wait for multiply result), or

[7]All Antares pipeline interlocks are hardware controlled (as opposed to, for example, the MIPS processors, which relies on the compiler to generate interlock-free code.)

execution of a synchronous multi-cycle instruction. In the absence of any of these conditions, most instructions execute at a rate of one instruction per cycle. While each instruction takes a minimum of four cycles, three cycles are overlapped by the execution of other instructions. The *nominal execution time* of an instruction is defined as the number of non-overlapped cycles required for its execution in the absence of delays.

Most Antares instructions have nominal execution times of one cycle: exceptions include load and store multiple instructions, load and branch instruc - tions, and the multiply and the divide instructions. Load and store multiple are the only synchronous multi-cycle instructions: these instructions take one cycle for each register loaded or stored. While multiply and divide are multi-cycle instructions, they are asynchronous: their execution is initiated in a single cycle, and subsequent cycles can be overlapped by the execution of other instructions.

The load instruction and taken branch instructions are delayed completion instructions. Load instructions and taken branch instructions require two cycles to execute: the destination register of the load is not available, and the branch is not effected, until the second cycle of the instruction. However, the next sequential instruction is issued immediately after the load or branch is issued, and its execution can overlap the second cycle of the load or branch. If the destination register of a standard-format load is not referenced by the following instruction, the load executes without delay, and its effective execution time is one cycle. (If the following instruction does reference the destination register, it is delayed (via a pipeline interlock) until the load completes.)

The next sequential instruction after a branch always is executed, regardless of whether or not the branch is taken. This instruction stream location is called a *branch shadow*: If no useful work[8] can be done in the shadow of a branch, a **NOP** (e.g., **MOVE** 0->0) instruction should be placed after the branch. When the branch shadow can be usefully filled, the effective execution time of a branch instruction, taken or not taken, is one cycle.

Further information on instruction execution times is given in the Antares Instruction Set Reference Manual. Cache miss delays are discussed in Section 3.4 of this overview.

[8]Several studies have shown that the branch shadow can be filled with a useful instruction at least 70 percent of the time: see, for example, Gross and Hennessy [1982].

3. The Cache

3.1 Introduction

The two principal ingredients of high CPU performance are a small cycle time and a large cache[1]. The simple instruction set of Antares helps keep its cycle time small, and its total cache size of 8K bytes — occupying about 2/3 of the total chip "real estate" — is large for an on-chip cache. However, a cache capacity of 8K is modest, relative to the aggregate speed of the 4 PUs; careful use of the cache, via code and data organization and prefetching and by cache line management, can help realize maximum performance.

This section describes the design of the Antares cache and the timing of cache ↔ memory transfers, discusses the rationale for various cache design decisions, and describes the instructions provided for control of the cache.

3.2 Cache Organization

To provide sufficient instruction and data bandwidth, Antares has separate and independent instruction and data caches. The two caches are identical: each has a total capacity of 4096 bytes, organized as 64 lines of 16 words (64 bytes). All transfers between the CPU and memory are in units of one line. The cache design is 4-way set associative: the 64 lines of a cache are grouped into 16 sets of 4 lines. Every memory line maps into one of these 16 sets, as illustrated in Figure 3.1. For cache access purposes, a virtual word address divides into a word index, which specifies one of the 16 words in a line, a set index, which specifies one of the 16 sets of 4 lines, and a tag. When a line is stored in the cache, its address tag is stored in the tag store location which corresponds to that line. A set of flags also is stored with the tag, including LRU bits, a system/user bit, a valid bit, and a modified bit.

[1]Both are determined by technology; a key factor in determining cycle time is the cache access time. CPU cycle time frequently is determined by the taken branch path length, which includes a cache access for the branch target.

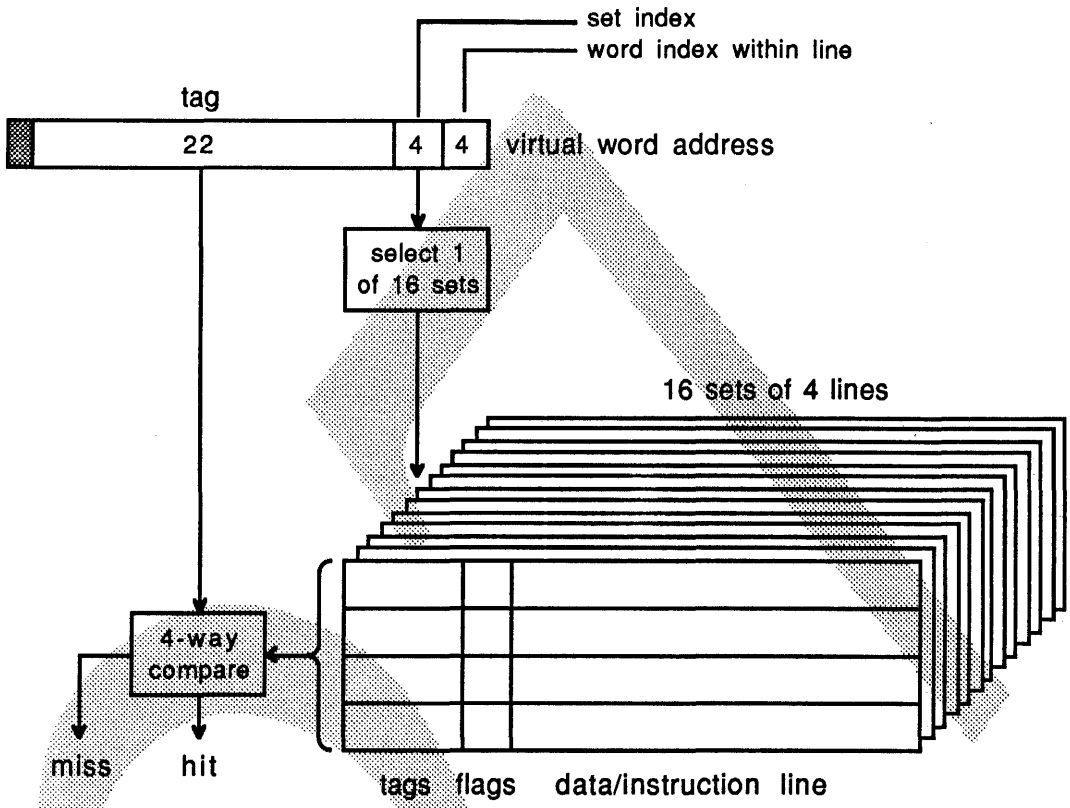


Figure 3.1. Cache Address Mapping

On an instruction fetch, data fetch, or data store access to memory, bits 4–7 of the virtual word address are extracted and used to select 1 of the 16 sets. (Although instruction addressing is on half-word boundaries, fetching of both sequential and branch target instructions is done in units of one word.) The tag (bits 8–29) is compared with the tags of the valid lines in that set. If a match is found, then a cache *hit* has occurred. Bits 0-3 are used to select the word be fetched or stored, the LRU bits are updated, and the modified bit is set if the access is a store. If none of the valid tags in the set match the tag of the current access, a cache *miss* occurs. The LRU bits for the lines in the set are examined to determine which line is to be replaced. If the selected line is modified, then it has to be written to memory. This operation is called a *moveout.* To reduce the time the requesting PU must wait, the line being moved out is placed in a moveout buffer, the missing line is read into the cache from memory (*moved in*) and the requestor activated, and the line in the moveout buffer then written to memory. If the line being replaced is not modified, the missing line simply is read into its cache location. In either case, the memory read is initiated by sending a line missing request to the MMU.

Physically, each cache is organized as four banks of 256 words, as shown in Figure 3.2. A 5 x 4 crossbar switch connects the PUs and MMU to the cache memory banks, and a four-ported tag store permits simultaneous cache access from

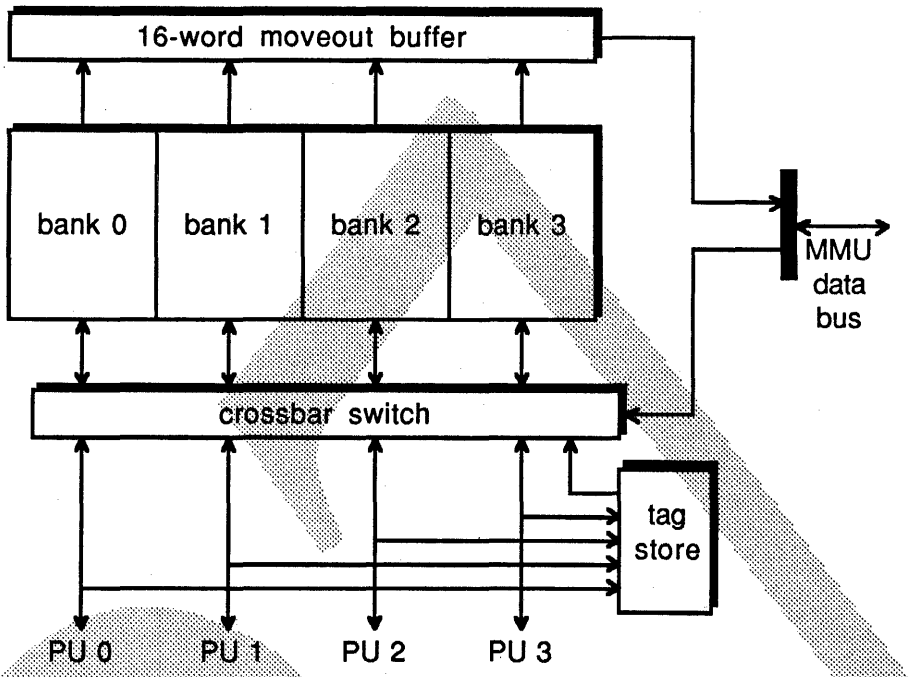


Figure 3.2. Physical Cache Organization

all four PUs. Up to four simultaneous transfers between the cache and PUs or memory can be executed in a cycle, provided that each transfer uses a different bank. When multiple requests are made to one bank in a cycle, one request is selected and the other requestors are blocked. In selecting a request, PUs have higher priority than the MMU; among competing PUs, priority is determined by PU number, with PU 0 having the highest priority.

The cache is divided into banks — interleaved — in order to provide the bandwidth required by multiple PUs. The division into four banks represents a balance between the cost of the crossbar switch and its effect on cycle time, and the performance penalty of bank conflict delays.

Delays caused by cache bank conflicts depend on the PU memory access rate, the number of active PUs, the cache miss rate, and memory addressing patterns. Consecutive words of a line reside in different cache banks: words 0, 4, 8, and 12 reside in bank 0, words 1, 5, 9, and 13 reside in bank 1, and so on. In array operations, assigning each PU to operate on every fourth element of the array results in each PU accessing a different cache bank, substantially reducing bank conflict delays.

3.3 Cache Design Decisions

The overall size of the Antares cache was determined by the available chip "real estate", and its partitioning into separate instruction and data caches was determined

by bandwidth requirements. These partitions were chosen to be equal in size because this is believed to be the best division for small caches[2], and because it was desired to make both caches identical to reduce chip design effort. The physical division of each cache into four banks was based on a trade between the cost of the required crossbar switch and the performance penalty of bank conflicts.

The cache design is store-to, rather than store-through, for performance reasons. Only lines in the data cache can be modified. The same line may be present in both the instruction and the data cache; a store to that line modifies only the line in the data cache. For cost and complexity reasons, cache coherence is left to software: an instruction is provided to flush a cache line.

Several factors influenced the line size decision, including tag store size, line utilization, and miss processing startup cost amortization. For a given size cache, a large line size reduces the number of lines and reduces the amount of tag storage required. At roughly a word per cache line, tag storage represent a substantial fraction of the cache real estate. Line utilization can be viewed as the proportion of words in a line referenced during the time that line is in cache. High line utilization is easier to achieve with small lines than with large ones and, for a given line size, instruction lines have a higher utilization than data lines. (However, the decision to make both caches identical dictated the same line size for both caches.) Small lines also mean that there are more places in the cache into which lines can be mapped. On the other hand, a startup cost is associated with each miss, so that the maximum CPU – memory bandwidth is less for smaller line sizes. In the absence of con - flicts, the time required to move a line into the cache is s + w cycles, where s is the startup cost and w is the number of words in the line. For Antares, s is expected to be on the order of 7 cycles, and a line size of 64 bytes gives a maximum bandwidth 30 percent greater than that of a line size of 32 bytes. Balancing these various factors resulted in the choice of 64 bytes for cache line size.

A set size of 4 was chosen over a set size of 2 to help reduce the miss rate and to reduce the probability that the four PUs might generate a reference pattern which would cause set thrashing.

A cache can be designed to be accessed with virtual addresses or with real addresses. In the latter case, address translation must be done on every memory reference either before or, in some cases, in parallel with, the cache access. This adds some complexity to a single-processor design and can reduce performance by increasing cycle time (although the translation can be pipelined at the cost of additional complexity). In Antares, real addressing of the cache would require a translation mechanism capable of supporting four simultaneous translations, one for each PU. Consequently, the Antares cache is virtually addressed: address translation takes place on on miss processing, which substantially reduces the performance demands on the translation mechanism.

---

[2]Davidson [1987] concludes that the optimal instruction cache size is about 50 percent of total cache capacity for most capacities.

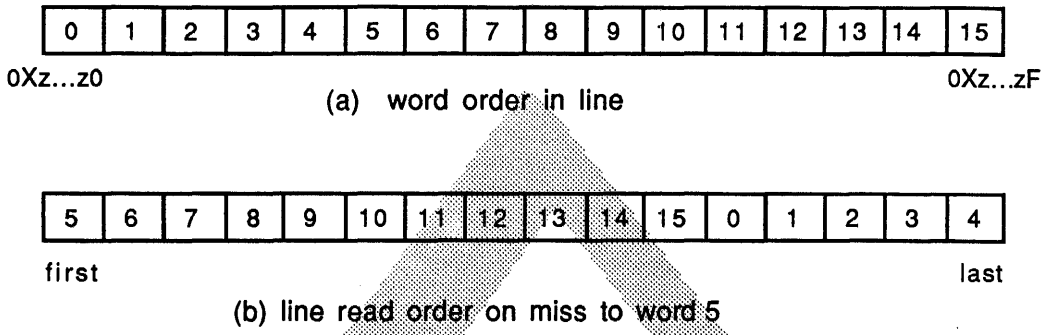


Figure 3.3 Line Rotation on Movein

The major drawback of a virtually addressed cache is the potential synonym problem: multiple copies of the same real memory line may end up in the same cache because of being referenced by different virtual addresses (as a consequence of memory sharing). In Antares, synonym avoidance is left to software. Also, it is possible that a memory line may contain both code and data, and be loaded into both the instruction cache and the data cache at the same time. While this tends to be wasteful of cache space, it should not present data integrity problems, since the contents of the instruction cache cannot be modified. The data in this shared line can be modified, and the line will be properly updated in memory. Note, however, that if the contents of a line are to be modified and subsequently referenced as instructions, the line must be explicitly flushed prior to that reference.

It is possible to define non-cached pages (via an operating system kernel call). This is required for memory-mapped IO. When a load or store instruction refer - ences a word in a non-cached page, that word is directly read from or written to memory by the MMU. References to non-cached data add 8 cycles (ignoring conflicts) to the execution time of load and store instructions.

3.4 Cache Miss Timing

When a PU accesses a word belonging to a line not in the cache, a cache miss occurs, and the missing line is moved in from memory; the MMU translates the virtual address of the line to a real address, reads it from memory, and stores it in the cache. If this line replaces a modified line, the latter must be moved out. To minimize the delay incurred by a PU on a miss, the modified line is written to a moveout buffer (Figure 3.2), the missing line moved in, and the modified line then written to memory. *Rotate* and *forward* operations further reduce the PU's delay. The MMU initiates a line read from memory beginning with the word accessed on the miss, reads the remaining words in the line, and then wraps around to read the first part of the line, effectively rotating the line so that the referenced word is the first word read. For example, if an access to word 5 of a line results in a miss, word 5 is the first word read, as shown in Figure 3.3. When this word is read from memory, it is forwarded to the requesting PU at the same time it is stored in

the cache, so the PU can continue execution without waiting for the movein to complete. Subsequent accesses to that line, however, must wait for movein completion.

The delay incurred by a PU as the result of a cache miss depends on a variety of factors, including cache bank conflicts, MMU busy delays, and translation (TB miss) delays. The timing involved in cache miss processing, assuming a demand miss and no delays, is shown in Figure 3.4 and discussed below. It is assumed that this processing is initiated as a result of a miss on a load or store access to the ith word of a line, and requires a moveout of a modified line. (Instruction cache miss timing is similar except, of course, no moveout can occur.)

A PU issues a load or store request to the cache at the end of the E stage of its pipeline. In the following cycle, which corresponds to the S stage of the PU pipeline, the cache does a tag compare to determine if the line is present. If it is, the accessed word is loaded into or stored from a register in the following cycle (which corresponds to the S2 stage of the pipeline); otherwise, a movein is initiated. The requesting PU is interlocked in its S stage.

In cycle 1 of movein processing, the cache sends a movein request to the MMU. At the same time, it starts to move the (modified) line being replaced into the moveout (MO) buffer. This takes four cycles, and all four cache banks are busy for those four cycles, since 16 words have to be moved. At present, cache accesses for moveout buffer loading are assumed to have higher priority than any other cache access, since this operation must be completed before the first word of the replacement line arrives from memory. Note that moveout buffer loading is overlapped by the startup time for the movein and so does not affect the requesting PU; however, cache accesses by other PUs will be blocked during this period. It may be possible to implement an access priority resolution scheme which will avoid the need for blocking PU access during moveout buffer loading; this will be determined later in the design process.

In cycle 2, the MMU translates the address of the line being moved in via a Translation Buffer (TB) look-up. At the end of this cycle, it initiates memory access for the first word to be moved in — word i — which is not necessarily the first word of the line. The access time for this first word is assumed to be equivalent to 5 cycles; it depends on RAM chip access time and the Antares cycle time. Subsequent words can be read without further delay.

In cycle 8, word i is returned and, simultaneously, stored in the cache and forwarded to the requesting PU (which resumes execution, having incurred a delay of 8 cycles). The remaining words of the line are stored during cycles 9–23. The line is marked valid at the end of cycle 23 and is available for PU access in cycle 24. Store requests for words of the line being moved in are made by the MMU; if access to a cache bank is requested in the same cycle by both the MMU and a PU, the PU is given priority.

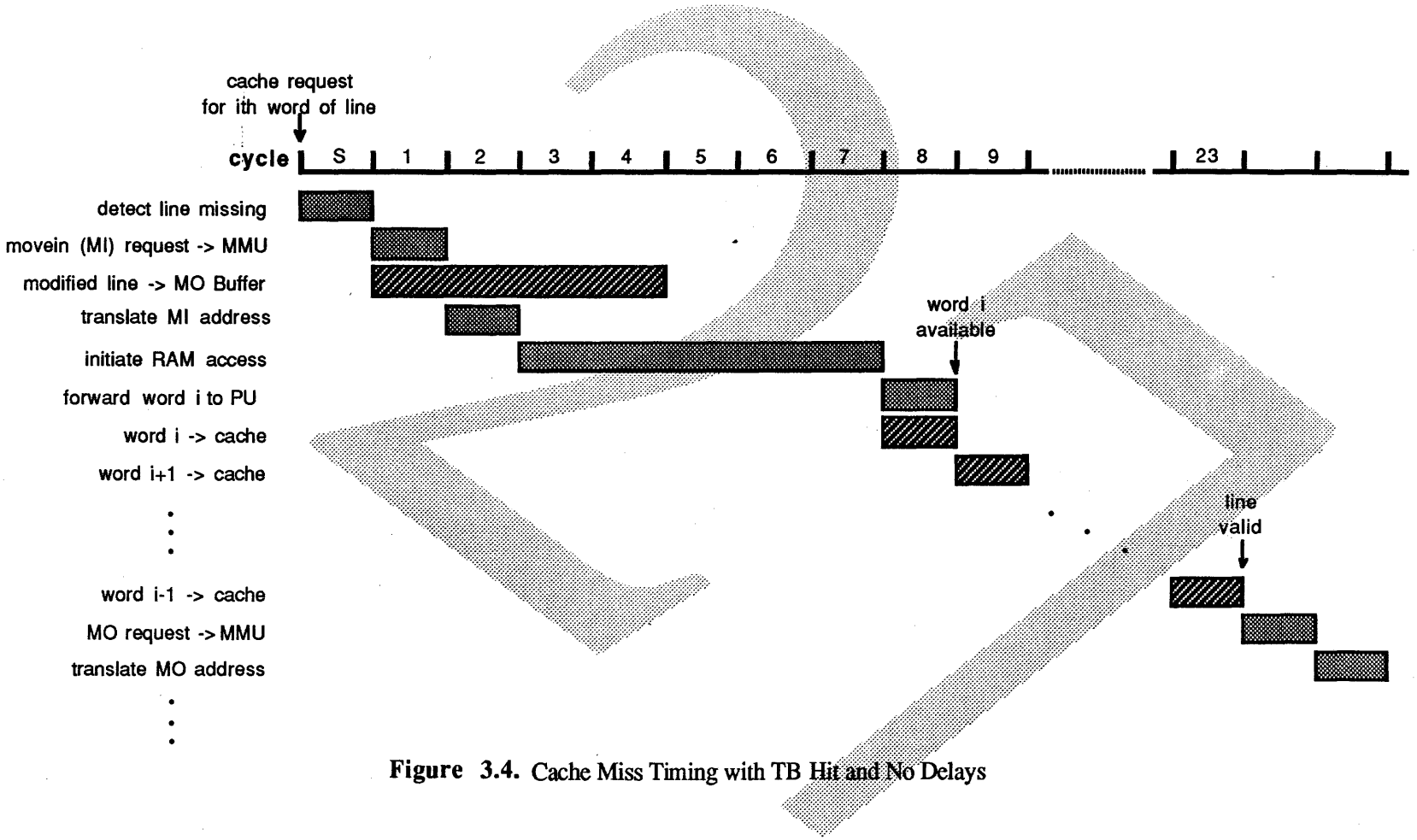


Figure 3.4. Cache Miss Timing with TB Hit and No Delays

When the last word of the missing line has been stored, moveout processing is initiated. This is similar to movein processing: a moveout request is sent to the MMU, the address of the line being moved out is translated, RAM access initiated, and the contents of the MO buffer stored in memory, one word at a time. Note that no cache bank conflicts can occur during the transfer to memory, so the moveout should complete in 23 cycles. The MMU is busy for the duration of a moveout; if either an instruction cache or a data cache miss occurs during this time, the requesting PU will be blocked until the moveout completes.

In the absence of MMU busy, bank busy, and translation delays, a load or store instruction which causes a data cache miss incurs a delay of 8 cycles. This delay is called the *nominal miss penalty*. The instruction stream delay actually resulting from a miss is called the *effective miss penalty*. This often is greater than, but sometimes is less than, the nominal miss penalty. Conflicts and TB misses increase the effective miss penalty. Certain overlap situations reduce it; for example, the actual delay caused by an instruction fetch miss for an instruction which has a divide ahead of it in the pipeline may be less than the nominal miss penalty.

More than one instruction can incur a direct delay as the result of a single miss. The first reference to a missing data line is satisfied after a nominal delay of 8 cycles; a subsequent reference will be blocked until the entire line has been stored in the cache and marked valid, which takes another 15 cycles. The delay incurred as the result of a second reference to a missing line depends, among other things, on the number of instructions between the first and second instructions accessing the line. An example of the timing involved is shown in Figure 3.5, which shows pipeline and cache timings for a sequence of three instructions.

In this example, there is a single **ADD** instruction between two **LDW** instructions which reference words in the same data line. The first **LDW** results in a miss, and interlocks in the S stage of the pipeline for 8 cycles while waiting for its operand word to be forwarded. Because the pipeline is blocked, the **ADD** and **LDW** which follow also are delayed; the **ADD** interlocks in its E stage, and the **LDW** interlocks in its D stage. When the operand word of the first **LDW** is returned, it completes execution. With the pipeline now free, the **ADD** completes and the second **LDW** progresses to its E stage, where it issues a cache request. Because the line is still being moved in and is invalid, this instruction interlocks in its E stage until the line becomes valid and its operand word can be read from the cache, and so incurs a delay of 14 cycles. Additional instructions between between the first and second **LDW**s would reduce this delay.

If the page address of the missing line is not in the Translation Buffer, an additional delay of 8 cycles is incurred if the page table block address is contained in the Directory Buffer; if it is not, a 16-cycle delay is incurred (see Section 5.)

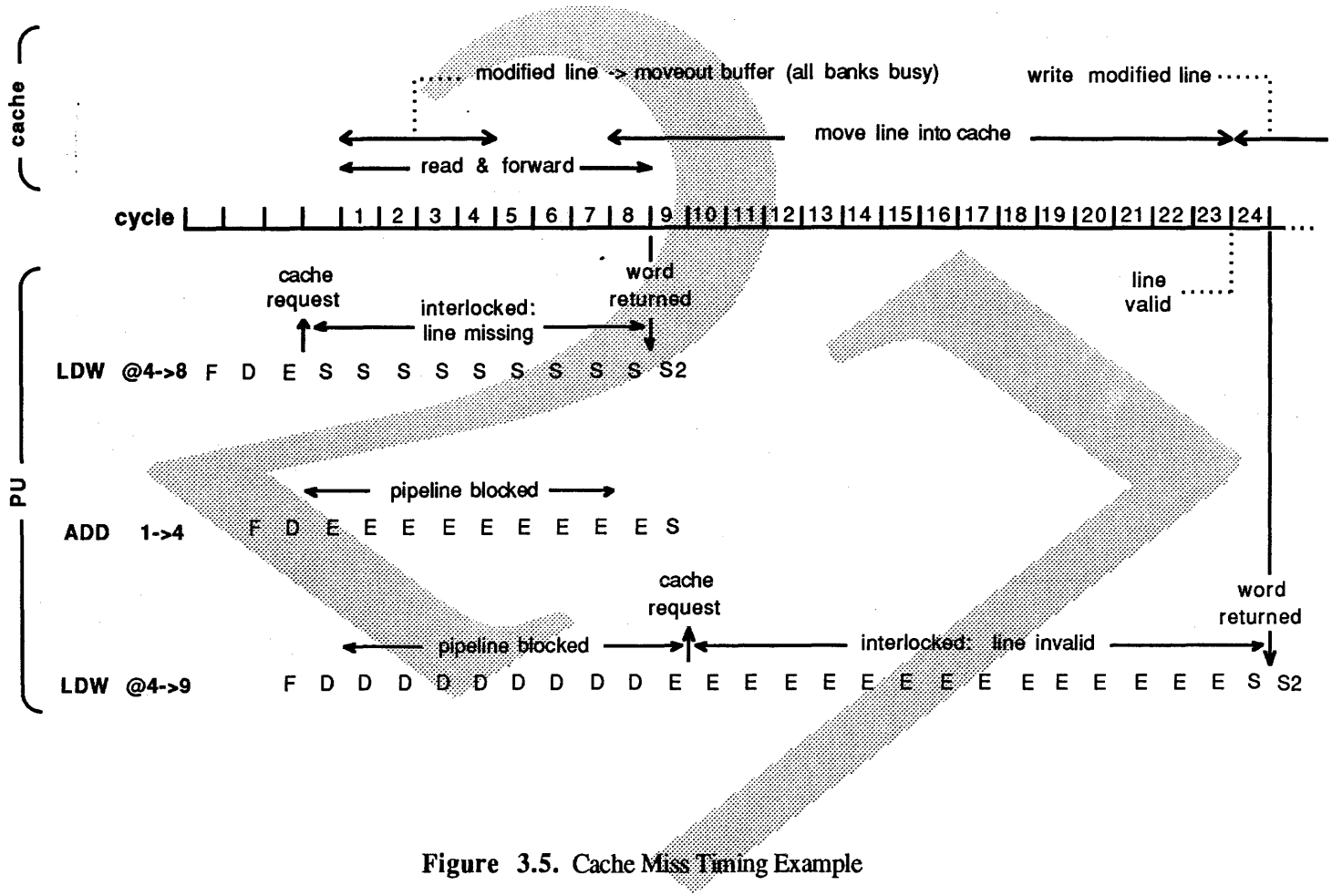
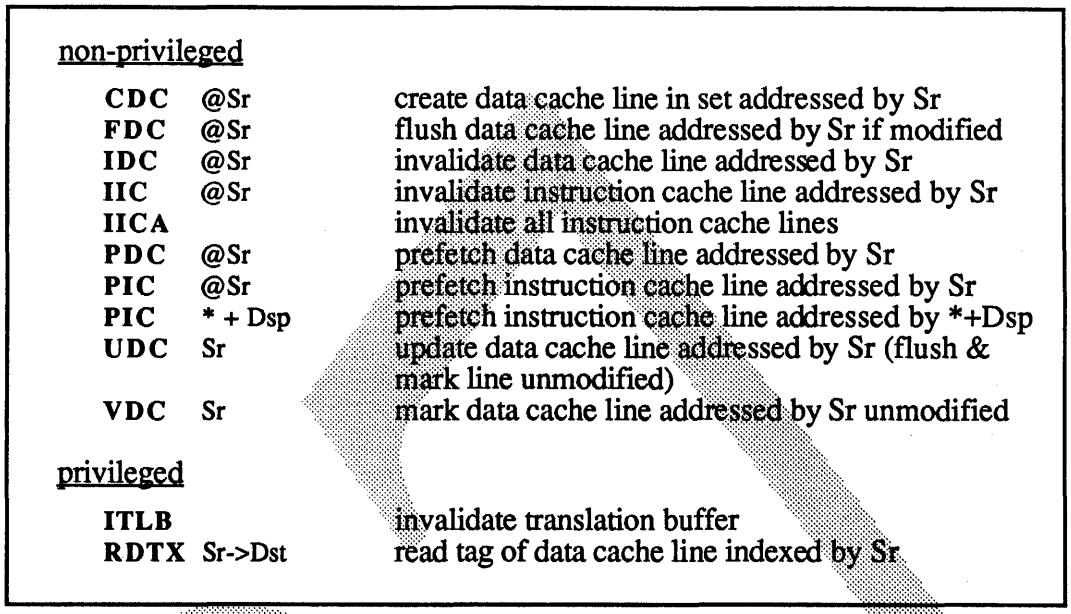


Figure 3.5. Cache Miss Timing Example



non-privileged

CDC	@Sr	create data cache line in set addressed by Sr
FDC	@Sr	flush data cache line addressed by Sr if modified
IDC	@Sr	invalidate data cache line addressed by Sr
IIC	@Sr	invalidate instruction cache line addressed by Sr
IICA		invalidate all instruction cache lines
PDC	@Sr	prefetch data cache line addressed by Sr
PIC	@Sr	prefetch instruction cache line addressed by Sr
PIC	* + Dsp	prefetch instruction cache line addressed by *+Dsp
UDC	Sr	update data cache line addressed by Sr (flush & mark line unmodified)
VDC	Sr	mark data cache line addressed by Sr unmodified

privileged

ITLB		invalidate translation buffer
RDTX	Sr->Dst	read tag of data cache line indexed by Sr

Figure 3.6. Cache and TB Control Instruction Summary

3.5 Cache Control Instructions

In the past, caches have been architecturally invisible, partly because many architectures were defined before caches were routinely incorporated in designs. While the cache miss rate and miss penalty are key factors in determining CPU performance, the user historically has had no direct control over the cache. The trend today is to make the cache visible so that compilers and other software can help improve performance. The Antares instruction set provides nine user-state instructions for this purpose. There also are privileged cache and Translation Buffer (TB) control instructions for use by the kernel in flushing the cache and TB on a task switch or a task termination.

The maximum instruction execution rate of the Antares CPU is one instruction per cycle (per PU); the actual execution rate depends primarily on cache miss delays, and to a lesser extent on cache bank conflict and pipeline interlock delays, and on the relative frequency of multi-cycle instructions. To illustrate, suppose the data cache miss ratio is 0.04 misses per access and the average number of data accesses per instruction is 0.5: the data cache miss rate, then, is 0.02 misses per instruction. If each miss causes a average delay of 11 cycles, then data cache misses add 0.22 cycles to the mean instruction execution time. Instruction cache misses add additional delay cycles (although instruction cache miss ratios usually are lower than data cache miss ratios because of the greater locality of instruction references).

There are two ways to reduce the impact of cache misses on performance: reduce the miss rate, and reduce the miss penalty. The compact Antares instruction set helps reduce the instruction cache miss rate, and the 4-way set associative, full LRU, cache design helps in miss rate reduction. Instructions are provided to invalidate cache lines: when it is known that a line will not be used again, it can be marked invalid and least recently used. This can save displacing another line which may be referenced again. Modified lines which are no longer of use (e.g., lines popped from the stack) can be marked unmodified; this reduces memory traffic delays and helps reduce the miss penalty. When all 16 words in a line are to be written, a create data cache line instruction can be used to avoid bringing in the original contents of that line from memory.

Misses can be divided into two classes: demand misses and prefetch misses. A *demand miss* occurs when an instruction fetch or and operand fetch or store references a line not in the cache; demand miss timing was discussed earlier in this section. Antares provides instructions to prefetch lines into the data and instruction caches; in executing one of these instructions, the PU sends the memory address to the cache and continues fetching and executing instructions. When one of these instructions references a line not in the cache, a *prefetch miss* occurs. While it still takes 23 cycles to complete processing for this miss, it may be possible to overlap part or all of this time with the execution of subsequent instructions, reducing or even eliminating the *effective* miss penalty.

As an example, consider code generation for a procedure call. The actual transfer is effected by an **LDW-JMPL** sequence: the entry address is loaded into a register and a jump and link instruction executed. The compiler can insert an instruction cache prefetch after the load, so that the sequence becomes **LDW-PIC-JMPL**. This adds one cycle to the sequence. The optimizer then tries to move the **LDW-PIC** pair as far back in the instruction stream as it can: every instruction which can be moved ahead of the **LDW-PIC** pair (after the first) represents a potential reduction of one cycle in the miss penalty.

To minimize hardware complexity, prefetch requests are not queued. If a prefetch miss occurs while the MMU is busy with a transfer, the prefetch request simply is discarded.

An instruction is provided to flush (force the writing of) a modified data cache line to memory. This instruction can be used to insure that memory shared between CPUs is updated properly.

Much current-day software, such as that for the Motorola 68000, was not developed with a cache in mind, and frequently produces higher than necessary miss rates when executed on a later CPU which has a cache. Antares software designers have the opportunity to reduce miss rates through careful organization of code and data and through the use of invalidate and create line instructions, and to reduce the effective miss penalty by prefetching.

3.6 Cache Flushing

Cache lines and TB page entries are tagged with their virtual addresses, so that lines and pages in one address space cannot be distinguished from lines or pages with the same address in another address space. Consequently, in switching from one address space to another (see Section 6), the kernel must flush the cache and the TB. Flushing the instruction cache and the TB requires only simple invalidate operations, and instructions (**IICA** and **ITLB**) are provided to perform these operations. Flushing the data cache is more complicated, since modified data cache lines have to be written to memory. To simplify cache control and reduce memory traffic, Antares does not provide a "flush data cache" instruction. Instead, an instruction (**RDTX**) is provided to read the tag associated with a given data cache line (the line effectively is specified by number, 0 - 63). The tag read by this instruction includes the virtual line address, valid bit, modified bit, and system/user bit. The kernel uses this instruction to inspect the data cache contents, and uses other cache control instructions to flush and invalidate selected lines. The kernel, then, can decide on the basis of state and address range what lines must be written out, what lines must be invalidated, and what lines can be left in the cache. On a task termination, the kernel can simply invalidate lines without a moveout.

For a task switch, it is expected that about half the lines in the data cache will be modified and most of these (excepting lines belonging to the kernel) will have to be moved out. The time required to flush the data cache is determined by the moveout time of 24 cycles per modified line, regardless of whether the flush operation is done by hardware or done by software. This operation is easily parallelized, and can be done as efficiently by the kernel as it can be done in hardware.

4. PU Communication and Coordination

4.1 Introduction

Instructions for the initiation and coordination of concurrently-executing activities on Antares PUs can be divided into three classes: broadcast, semaphore, and interrupt. Broadcast instructions permit one PU to broadcast data values and activity starting addresses to other PUs, and let a PU wait for one or more other PUs to complete activity execution. Semaphore instructions are used to transmit data between activities executing on different PUs and to control access to shared data. Interrupt instructions permit a PU in system state to interrupt other PUs, as when preparing for a task switch. This section describes broadcast and semaphore instructions; interrupts are discussed in Section 6.

4.2 Broadcast Instructions

The most common model of execution for an Antares CPU is expected to be an alternating sequence of serial (SISD) and parallel (SIMD or MIMD) activities in which a controlling PU initiating parallel activities on other PUs. Binding of a specific activity to a specific PU usually will be done at compile time. Parallel activities executing on PUs other than the controlling PU terminate by executing a halt instruction. The controlling PU can wait for these activities to complete before initiating a serial activity by executing a wait instruction.

address broadcasting. During serial execution, one PU is executing and the other are halted. To initiate execution of parallel activities, the controlling PU activates one or more waiting (target) PUs via the broadcast instructions

RSM *PUmask*	resume execution at target's current PC

or

STRT *Ri, PUmask*	start execution at address contained in register Ri of the controlling PU

Broadcast instructions have a 4-bit field called the *PUmask* field: bits 0-3 of this field correspond to PU numbers 0-3. Broadcast instructions operate on all PUs whose *PUmask* bit is 1. The resume (**RSM**) instruction causes each target PU — each PU specified by its *PUmask* field — to resume execution at that PU's current Program Counter (PC) address. Halted PUs resume execution immediately. If all target PUs are not halted at the time the **RSM** instruction is executed, the **RSM** instruction blocks; execution on the PU issuing the **RSM** instruction continues only when all target PUs have halted and then resumed. The start (**STRT**) instruction is similar to the resume instruction, except that target PUs resume execution at an address specified by the controlling PU, rather than at their own PC addresses.

Since the controlling PU blocks until all target PUS have halted, a start instruction cannot be used to transfer control on the controlling PU.

data broadcasting. The controlling PU can send a data value to one or more target PUs via the broadcast instruction

SEND *Ri->Rj, PUmask* store value in register Ri of the controlling PU into register Rj of target PU

Execution of a send instruction causes the contents of a register of the controlling PU to be stored in a register of each of the target PUs specified by *PUmask*. The target PUs must be halted for this store to take place, and they remain halted after the store is completed. If all the target PUs are not halted at the time this instruction is executed, the **SEND** instruction blocks until all target PUs have halted and the broadcast value has been stored.

halt and wait operations. A PU halts execution, or waits for other PUs to halt execution, via the instruction

WAIT *PUmask*

If the *PUmask* bit corresponding to the number of the PU executing the wait instruction is set, the instruction unconditionally halts PU execution, and other PUmask bits are ignored; the PU remains halted until reactivated by a resume or start instruction or an interrupt. If the *PUmask* bit corresponding to the number of the PU executing the wait instruction is 0, the PU waits until all the PUs specified by PUmask have halted, and then continues execution. In this case, then, the wait instruction performs a *join* operation.

If the PU's register contents are no longer useful, the PU should set the "registers available" bit in its Status/Control Register (Figure 2.2) via a Set Mode instruction prior to halting to indicate that its registers do not have to be saved. In recognizing an external interrupt, the Antares interrupt mechanism tries to assign processing of the interrupt to a halted PU. The operating system kernel examines the "registers available" bit and skips register saving and restoring if that bit is set.

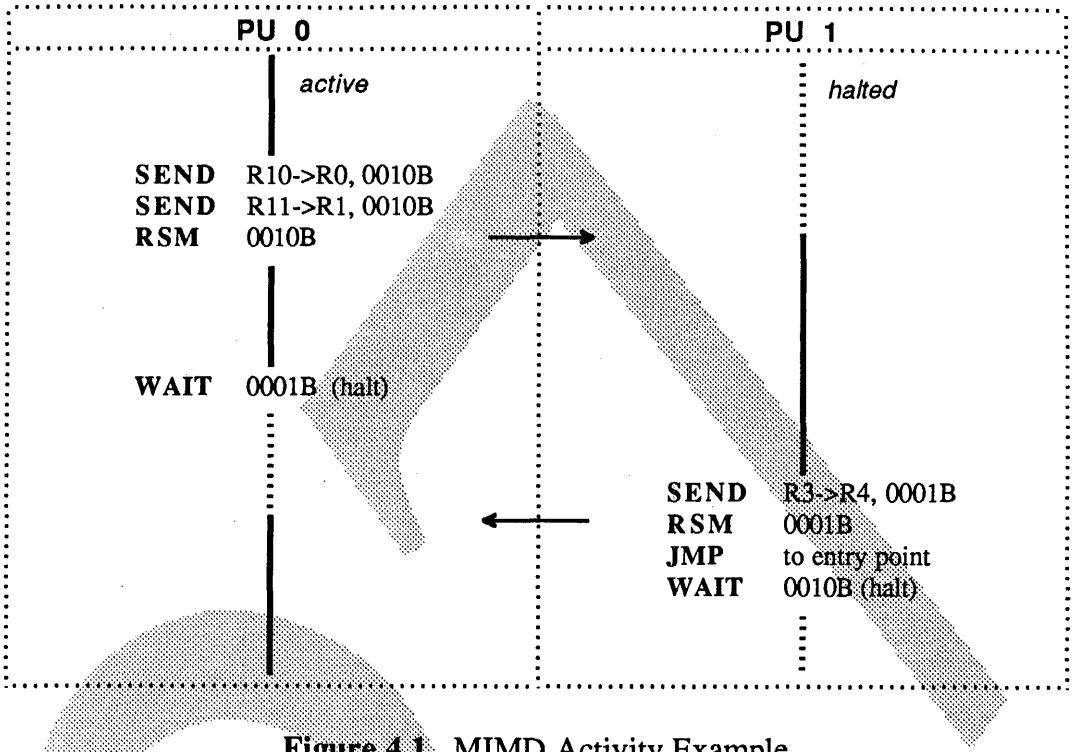


Figure 4.1. MIMD Activity Example

examples. Suppose a program executes its main thread on PU 0, but activates other PUs as needed for floating point arithmetic emulation. Assume PU 1 is used to execute floating point add operations and that, by convention, the floating point add emulation code expects its operands in registers 0 and 1, and returns its result to PU 0's register 4. The instruction sequences used to initiate and terminate this parallel computation might resemble those of Figure 4.1. The execution sequence shown here begins with PU 0 executing and PU 1 halted; PU 1's program counter holds the entry point address of the floating point add emulation code. The operands of the add are contained in registers 10 and 11 of PU 0.

To initiate parallel execution of the add activity, PU 0 executes send[1] instructions to transmit the operands to PU 1, and then executes a resume instruction to activate PU 1. PU 0 continues in execution until it reaches a point where the result of the add is required and then halts by executing a wait instruction with the PUmask bit for PU 0 set. When PU 1 finishes its computation, it issues a send instruction to return its result to PU 0; if PU 0 is not halted at this time, the send instruction blocks until PU 0 does halt. After sending its result to PU 0, PU 1 returns PU 0 to execution by executing a resume instruction. It then prepares for the next add activity by jumping to its entry point and halting.

[1]In instruction fields in this example, a numeric field terminated by a 'B' indicates a binary number: e.g., '0010B' represents 0010_2.

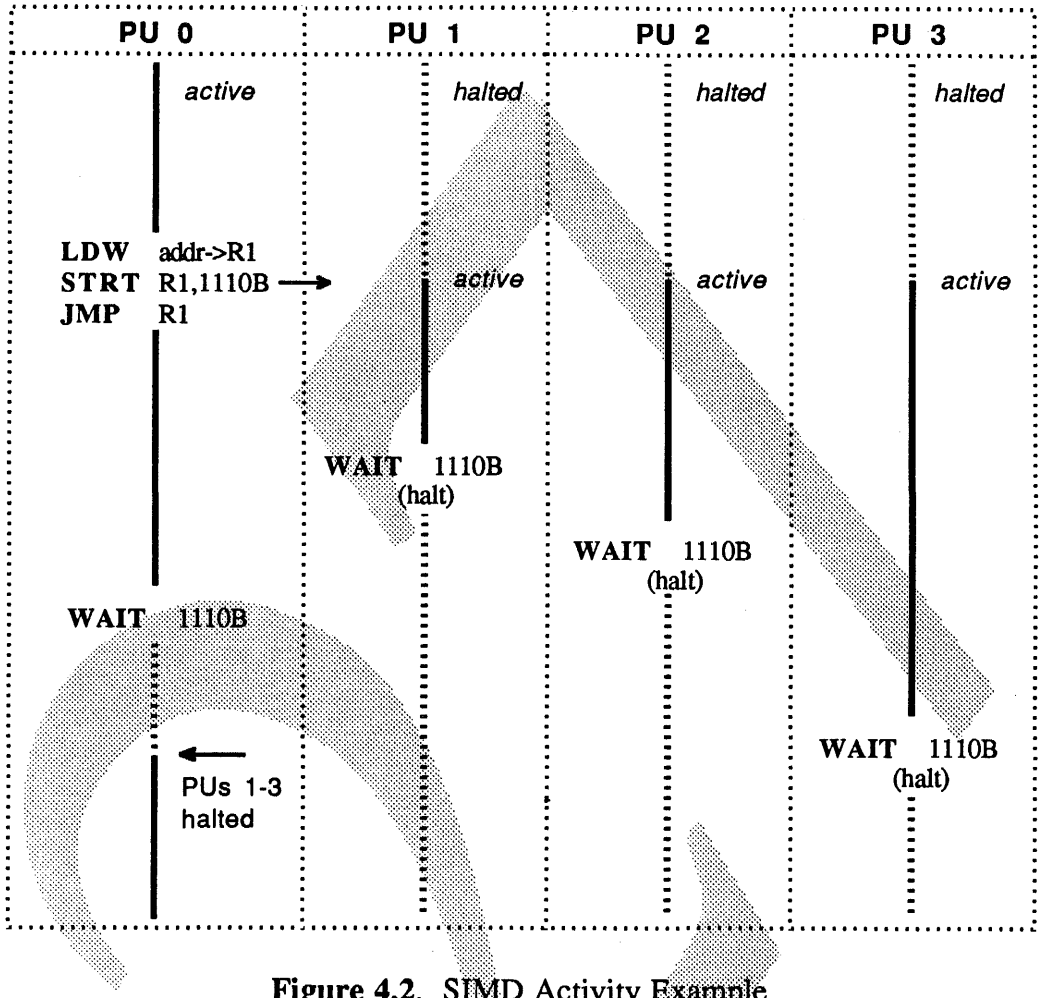


Figure 4.2. SIMD Activity Example

The coordination required to return the result also could be implemented via semaphores, with the advantage that the floating point emulation activity would not need to know which PU invoked it.

Figure 4.2 shows the initiation and termination of an SIMD activity in which all four processors execute the same code. To initiate this activity, PU 0 executes a start instruction with bits set in the *PUmask* field for PUs 1, 2, and 3; this starts these PUs executing at the specified address. PU 0 starts its own execution of the activity via a jump instruction.

While all PUs execute the same code, their execution times for this activity may differ because of data differences and because they incur different delays, such as cache misses. All four PUs, on completing execution of this activity, execute the same wait instruction, WAIT 1110B. This causes PUs 1, 2, and 3 to halt, since their own PU number is specified in the *PUmask* field, and causes PU 0 to suspend execution until each of the other PUs has halted. Thus, synchronization at the end of this activity requires only a single instruction.

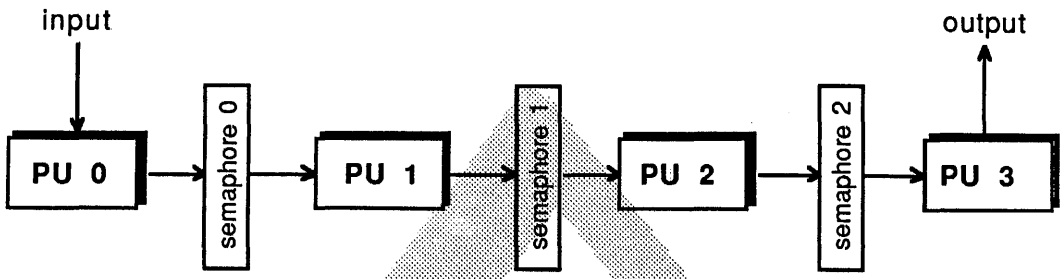


Figure 4.3. Semaphores as Platforms in Pipelined Execution

4.3 Semaphores

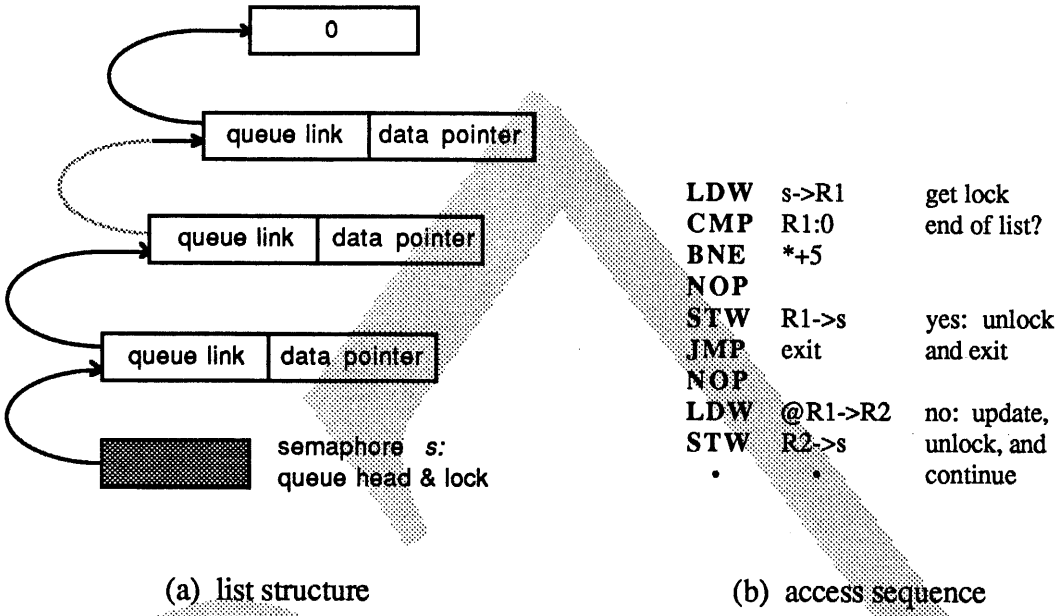
Broadcast instructions are used to transmit data from an active PU to *halted* PUs, and to activate halted PUs. Semaphore operations are used to communicate between and coordinate the activities of *active* PUs.

Antares load/store direct-addressing instructions span a 256-word direct address space. Bits 8-29 of a direct address are called the *prefix address*. The current prefix address is contained in one of the Special Register 6 pair, S6[0] and S6[1]; the cluster number in the PU's Status/Control Register determines which member of this pair is used. Generally, S6[0] is used in system state and S6[1] is used in user state. The address of the word accessed by a load/store direct instruction is formed by concatenating the displacement field of the instruction with the current prefix address. (See the discussion in Section 2.3.)

One use of this space is for inter-PU communication. The first 8 locations of direct address space are *semaphore* locations. Semaphore operations are performed by normal load and store direct[2] accesses to these locations. Associated with each of these locations is a full/empty flag **F**. A store to semaphore location *s* stores a value in that location and sets the **F** flag to full, provided that the **F** flag initially is set to empty. If **F** initially is set to full, the PU executing the store is blocked until **F** is set to empty. A load from semaphore location *s* loads a register with the value from that location and sets **F** to empty, provided that **F** initially is set to full. If **F** initially is empty, the PU executing the load is blocked until **F** is set to full. Semaphore flags also are contained in S6; switching clusters changes both the current prefix address and the current semaphore flag set.

examples. Semaphores are used to transmit data between executing PUs, to control access to data, and to control execution of "critical sections" of code. In the example of Figure 4.1, the result of the floating point add operation performed by PU 1 could have been returned to PU 0 via a semaphore, saving an instruction in

[2]While the contents of these locations may be read or written by register-addressing or base-plus-displacement-addressing load/store instructions, only the direct-addressing instructions perform semaphore operations.



(a) list structure (b) access sequence

Figure 4.4. Data Locking Via a Semaphore

PU 1's activity termination sequence. In MISD — pipelined — execution, semaphores are used as data platforms between pipeline stages, as illustrated in Figure 4.3. Here, each PU performs a different operation on an operand; sema - phores are used both to transmit data between PUs and to synchronize operations. For example, PU 0 reads an operand, performs some operation on it, and stores the resulting value in semaphore 0 via a **STW** Ri->0 instruction. If PU 1 has not yet read the previous value, the **F** flag of semaphore 0 will still be set to full, and the store instruction will block. When PU 1 does read that value (via a **LDW** 0->Ri instruction), the **F** flag for semaphore 0 is set to empty, PU 0's store completes, and PU 0 continues on to read and process the next operand.

Semaphores also are used as locks to control access to data. Suppose all four PUs are executing in SIMD mode, operating on a queue of work maintained in the form of linked list (Figure 4.4(a)). Semaphore *s* is used both to lock the queue and to hold the address of the next element in the queue. When a PU is ready to operate on the next element, it executes the access sequence shown in Figure 4.4(b). If the queue is unlocked, the **F** bit associated with semaphore *s* will be set to full. When a PU executes the **LDW** instruction of the access sequence, the contents of semaphore location *s* are returned to the PU and the **F** bit is cleared, blocking access to the queue by other PUs and so locking it. The queue is unlocked when the PU executes a store to semaphore location s, either after recognizing that the end of the queue has been reached or after removing an element and advancing the queue head to the next element.

activity initiation via semaphores. Since semaphores can be used to pass addresses, as well as data, it may seem that the broadcast instructions are redundant. However, activity initiation-termination sequences based on semaphores typically require execution of 6-10 instructions per PU, depending on the scheme used. This overhead impacts performance both directly and indirectly (by increasing code space). For example, suppose a sequence of 30 instructions can be divided into two independent activities of 15 instructions each. If the cost of parallelizing these activities is an additional 8 instructions, the performance gain[3] is only 30/23 = 1.3X; if the cost is reduced to two additional instructions, the performance gain is 1.75X. Broadcast instructions reduce parallelization overhead or, equivalently, accommodate a finer grain of parallelization than is achievable through semaphores alone.

4.4 Deadlock Detection

The state of each PU is represented by two bits in the Global Status Register (Special Register 13). Both bits are cleared if the PU is executing. One of the two bits is set if the PU is waiting for a semaphore flag to change state, and the other bit is set if a PU is halted. For purposes of deadlock analysis, a PU is considered halted if it has executed a wait instruction with its own number set in the PUmask field, or if it has executed a wait instruction in which the PU mask field specifies other PUs (only), and all the other PUs have not yet halted. If one or the other of these bits is set for all 4 PUs, a deadlock situation is assumed to have occurred and a trap is generated. This trap is non-maskable so that deadlocks occurring in system state can be detected.

[3]Assuming an execution time of one cycle per instruction.

21

5. Address Translation and the MMU

5.1 Address Space Model

Antares provides a flat (unsegmented) address space of 1024 million words (MW), which is mapped into real memory sizes ranging from 1 to 64 MW. There is, at present, no architectural limit on the number of address spaces supported. (A limit may be defined to insure compatibility with later versions which may maintain an address space number as part of the cache line tag; see the discussion of task switching in the next section). The address space model (Figure 5.1) is very simple. The operating system kernel maps into each address space. A 1-MW region at the high end of the address space is allocated for the kernel, and the remaining 1023-MW region is allocated to the user or to parts of the operating system other than the kernel. Traps and interrupts, with the exception of reset, transfer control to an interrupt vector address whose base is the start of the kernel region.

The kernel region is directly mapped, by hardware, to the first million words of real memory. The kernel region is not paged and does not use Translation Buffer (TB) entries; this helps improve the effectiveness of a relatively small TB[1]. Separate prefix addresses for user state and system state provide separate direct-address space (and semaphores) for the user and the kernel. Lines in this non-pageable kernel region are cached in the same way as pageable region lines. The actual amount of real memory allocated to the kernel is determined by its needs; the kernel, at system startup time, assigns the real memory it *doesn't* need in this 1-MW region to allocatable page space. The operating system may use a pageable region, in addition to the hardware mapped kernel region; this region, however, is not specified by the hardware architecture.

[1] Antares resembles MIPS (the MIPS Computer Systems' RISC CPU) in this regard: see DeMoney et al [1986].

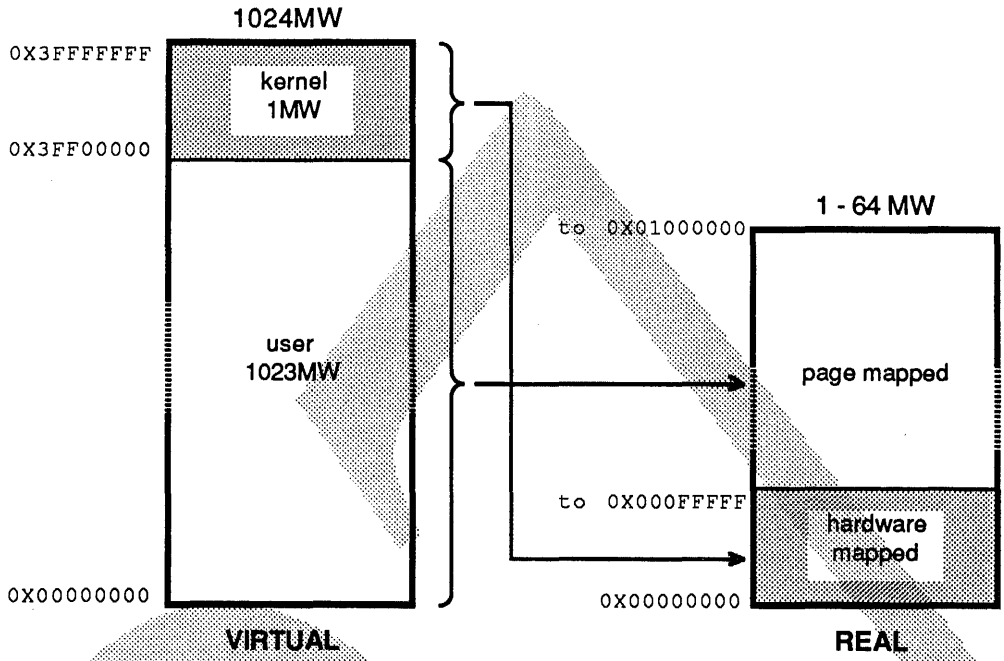


Figure 5.1. Virtual and Real Memory Space Organization

5.2 Virtual Address Format

The format of an Antares virtual address is shown in Figure 5.2. From an address translation viewpoint, a virtual address is the concatenation of a *virtual page address* and a word address within the page; the latter can be divided into a line index and a word index within the line. When a cache miss occurs, the MMU translates the virtual page address into a *real page address*, concatenates the real page address with the line index to form a *real line address*, and reads the corresponding line into the cache.

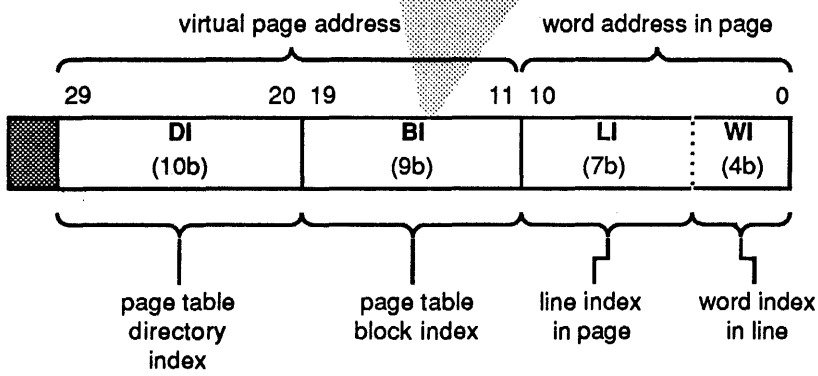


Figure 5.2. Virtual Address Format

In the case of direct addressing, a virtual address is formed by concatenating the current prefix address with the instruction's displacement field. The current prefix address is obtained from one of the Special Register 6 pair, S6[0] or S6[1], as determined by the current cluster number setting. When a trap or interrupt occurs, the current cluster number is set to 0, selecting S6[0]. The prefix address in this register typically is of the form 0x3FFzzz, where 'zzz' is an offset from the start of the kernel region to the start of the kernel's semaphore and directly-addresssable locations.

The page size in Antares is 2048 words (8192 bytes). This size is considered small enough to provide an adequate number of allocatable pages in a minimum real memory configuration and large enough to obtain reasonable Translation Buffer miss rates. Selection of a small page size that would better correspond to typical object sizes was tempting; however, it was decided that a page size small enough to be mapped one-to-one with objects presented significant performance problems[2]. (Blau [1983] reports that the mean size of objects in the standard Berkeley Smalltalk image is 32 bytes, with only 0.3 percent larger than 1024 bytes.) In addition to Translation Buffer size and performance implications, a small page size means large page tables. To keep the amount of memory allocated for page table space within reasonable limits requires a variable-length page table design if pages are small; this would increase the cost and complexity of the MMU.

5.3 Page Table Format

The translation of a virtual page address to a real page address by the MMU is done using a two-level page table constructed and maintained by the kernel. The structure of this table is illustrated in Figure 5.3. The first level is a directory of 1024 entries. The real memory address of the start of a directory is contained in the Page Table Directory Origin (PTDO) register, one of the CPU special registers. This register is loaded via a Move Special instruction when an address space switch occurs. Each directory entry represents a 512-page segment[3] of the virtual address space: if no pages in a segment are allocated (virtually), a flag in the directory entry is set to invalid. The segment at the high end of the address space corresponds to the kernel region. If any page in a segment is allocated, the directory entry contains the starting address of a page table block of 512 entries, one entry for each page in the segment. A page table block entry comprises a set of flags, including an invalid bit, and a real address field. If a real page is bound to the virtual page represented

[2]The VAX, with a page size of 512 bytes, experiences very high TLB miss rates. Clark and Emer [1985] report TLB miss rates in the vicinity of 0.033 misses/instruction for a VAX 11/780 with a 128-entry TLB (these rates however, represent operation in a multi-user environment; also, the split design of the VAX TLB results in a relatively high miss rate for its size).

[3]A segment is defined as the region in a virtual address space represented by a directory entry; it has no architectural definition beyond that.

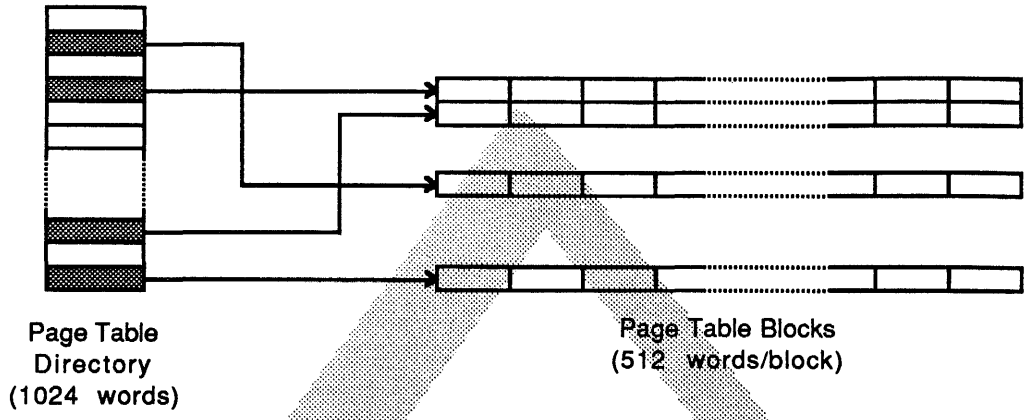


Figure 5.3. Page Table Structure

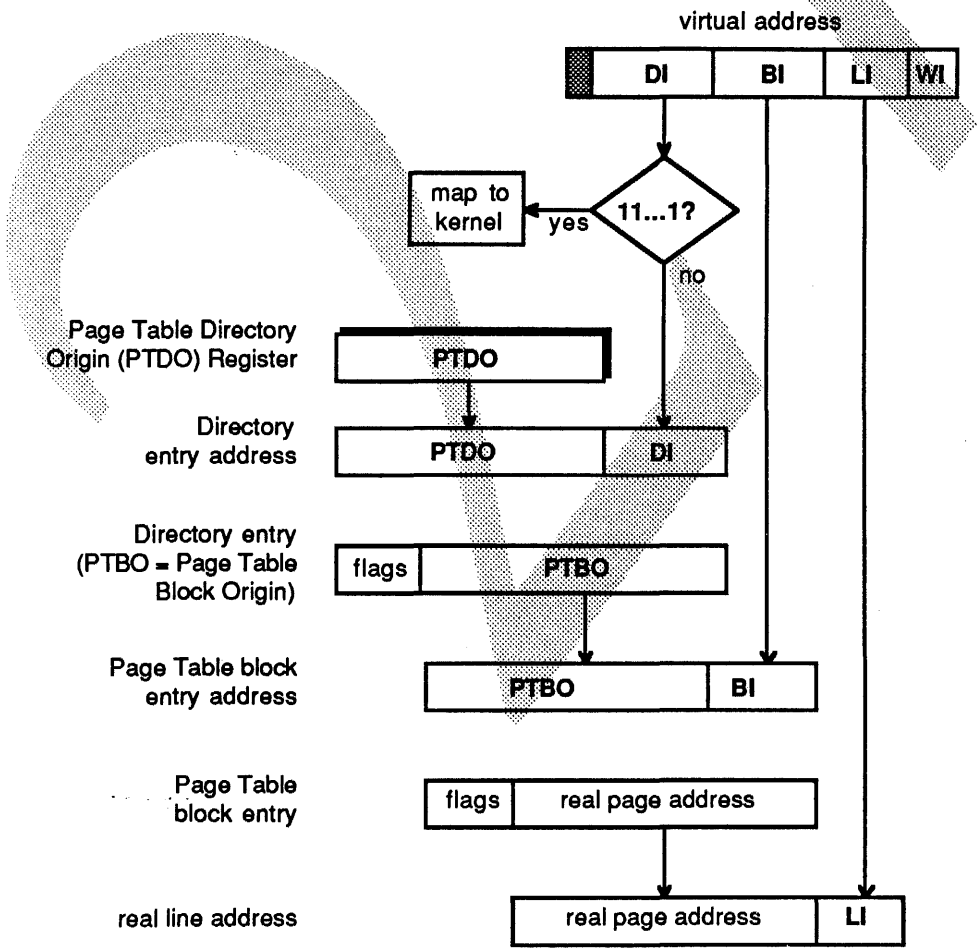


Figure 5.4. Virtual to Real Address Translation

by an entry, the invalid bit is not set and the entry provides the real page address together with access permission flags; the latter are forwarded to the cache for inclusion in the cache line tag. If either the invalid bit in the directory entry or in the page table block entry is set, a page fault interrupt is initiated by the MMU.

The real page address in a page table block entry is the concatenation of a CPU (node) number and a page address within the memory connected to that CPU. The MMU uses the CPU number to determine if the line transfer is to be initiated over the local memory bus or over the inter-processor bus.

The steps in mapping from a virtual address to a real address are illustrated in Figure 5.4. First, the directory index (DI) field is extracted from the virtual address. If this field corresponds to the kernel segment (DI = 0x3FF), mapping is done by hardware. This mapping is done by simply discarding the directory index part of the address and treating the remainder of the address as a real memory address. For a user region address, the DI field is concatenated with the contents of the Page Table Directory Origin to form a directory entry address.

The directory entry provides the page table block origin address: this is concatenated with the block index (BI) field of the virtual address to form the address of the page table block entry. The page table block entry provides the real page address, which is concatenated with the line index (LI) field from the virtual address to form the real line address.

Page table directory and block entries each are one word in length. For simplicity, both the directory and blocks are fixed length, and their addresses are formed by concatenation to avoid the need for addition in the MMU. As a consequence of the latter, directories must start on 1024-word address boundaries and blocks on 512-word boundaries. The division of the virtual page address into a directory index part and a block index part was done so as to minimize page table space requirements for address spaces with 6–12 valid segments.

5.4 The MMU

The translation of a virtual address to a real address is performed by the MMU. The steps in translation of a user region virtual address, and the MMU elements involved, are illustrated in Figure 5.5.

When a cache miss occurs, the virtual address of the missing line is sent to the MMU. The MMU extracts the virtual page address and searches the Translation Buffer (TB) for it. The TB is a small, fully associative[4], cache which holds translations for the *n* most recent MMU page references. in the initial imple - mentation of Antares, *n* is expected to be 16. A TB entry contains a virtual page

[4] i.e., any virtual page address can map into any entry: in a set associative TB, a page entry can map only into a given set.

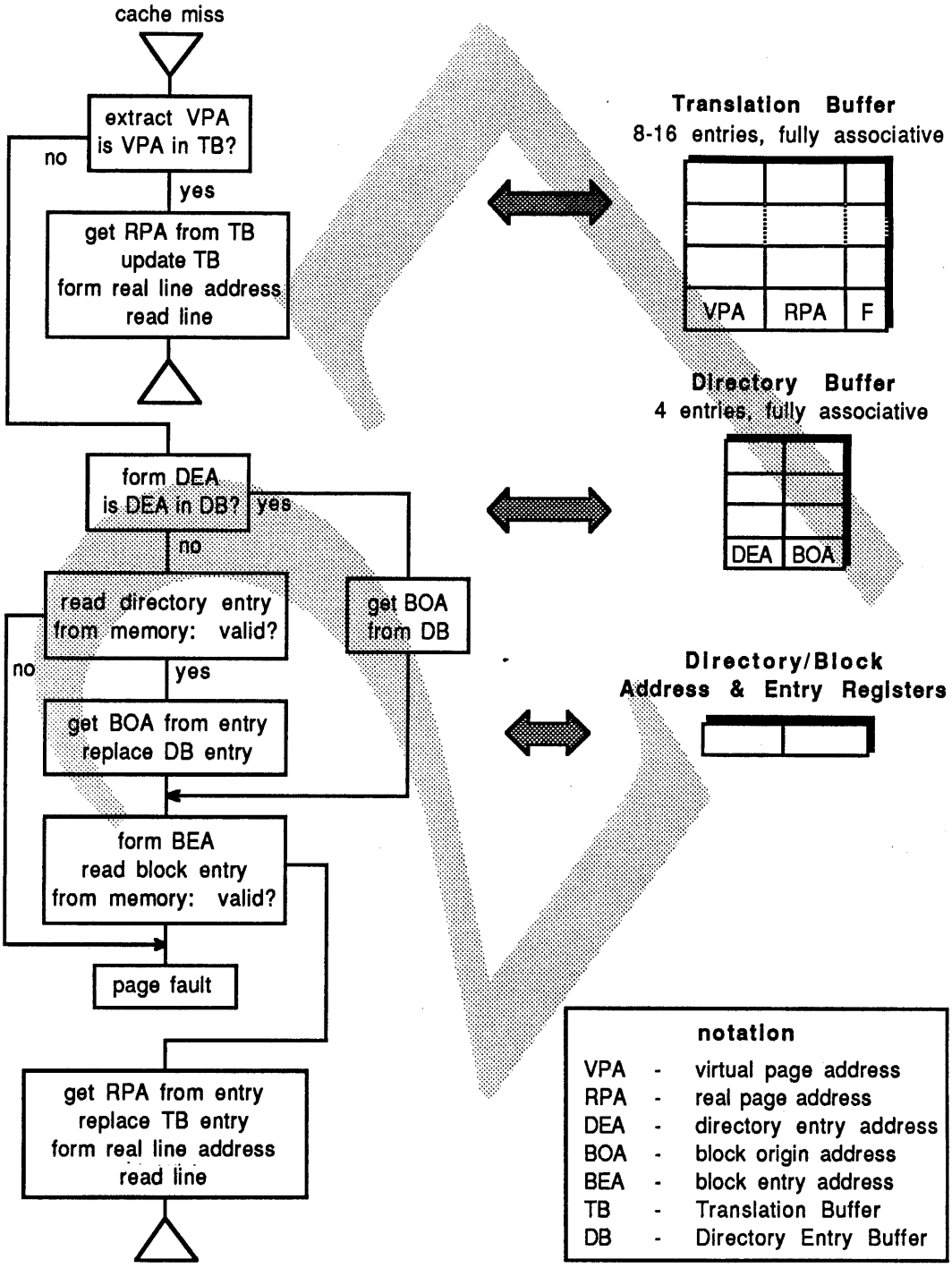


Figure 5.5. Address Translation: MMU Operations and Elements

address, the corresponding real page address, and a set of flags which include flag bits from the page table block entry. If the virtual page address is found in the TB (a TB hit), the real page address is obtained from the entry, the real address of the line is formed, and a line read request sent to (local or remote) memory. The TB entry is established as the MRU (most-recently-used) entry, either by reordering TB entries (there is ample time to do this while waiting for the data transfer to begin) or by setting appropriate flag bits. While the TB is small, it should provide good performance because of its fully associative organization and the relatively large Antares page size, and because it does not have to hold kernel addresses.

If the virtual page address is not found in the TB, the MMU forms the address of the directory entry by concatenating the directory origin address in the PTDO register with the directory index part of the virtual page address, as shown in Figure 5.4. The MMU has a small 4-entry buffer which holds the last four unique directory entry addresses and the page table block origin addresses contained in these directory entries. A four-way comparator determines if the directory entry address is in this buffer: if it is, the page table block origin address is obtained from the buffer. If it is not, the MMU must read the directory entry from memory.

In Antares, directory and page table data is not cached, but instead is read, a word at a time, by the MMU as needed. With a relatively small data cache of 64 lines, directory and page table lines for one code segment and one data segment would take 1/16 of the cache if resident, and there may be several data segments in use (stack, heap, page 0 for direct addressing, etc.). If these lines did not tend to stay resident, then a cache miss on a data or instruction reference would result in additional misses for the directory and page table block lines, substantially increas - ing the cache miss penalty. Caching the page table also would have increased the complexity of the interface between the caches and the MMU. For these reasons, the MMU is designed to read directory and page table block entries from memory as needed, using its own entry address and entry data registers. The small directory buffer significantly reduces the number of memory reads required for directory entries, so that the majority of cache misses which also miss in the TB incur an added penalty of just the 8 cycles required to read the page table block entry.

When a directory entry is read from memory, it is checked to see if it is valid; if it is not, a page fault interrupt is generated. If the entry is valid, it replaces the least recently used entry in the directory buffer.

The block origin address obtained from the directory buffer or read from memory is concatenated with the block index to form the block entry address, and the entry is read and checked. If invalid, a page fault interrupt is generated. Other - wise, the real page address from the entry is used to form the real line address and a read request is initiated for the line. A new TB entry is constructed and inserted in the TB in place of the least recently used entry.

Page table directories and blocks reside in the kernel region but are accessed by the MMU using real memory addresses, not kernel region addresses.

5.5 Page Table Entry Format

A Page Table Directory entry contains a valid bit and, for a valid entry, the real address of the first word of the Page Table Block associated with that entry. A Page Table Block entry comprises the following fields.

length in bits	description
18	real memory address of page. The actual maximum real memory connected to a single Antares CPU will be something less than the 2 gigabytes spanned by an 18-bit page address, so some of these bits can be re-assigned if necessary.
6	node (CPU) number; node numbers may be assign ed to system components in addition to CPUs, such as the NuBus
1	valid/invalid flag
1	read only/read write permission flag
1	system/user flag (protects system pages from user access)
1	cacheable/non-cacheable flag (see Section 5.6)
1	interrupt-on-write flag (see Section 5.6)
3	reserved for user (these bits are available to the operating system for software flags such as "copy-on-write" or "modified page" flags.

The positions of these fields within the one-word Page Table Block entry will be defined later.

5.6 Non-Cacheable Pages

Antares permits a page to be designated as non-cacheable (by having the kernel set the appropriate tag bit in the page table entry for that page). When a load or store access is made to a word in a non-cacheable page, that word is transferred directly between the designated register and local or remote memory. Non-cacheable pages have two principle uses: memory-mapped IO, and inter-CPU messages, both of which involve bypassing the data cache. However, it is possible to bypass the instruction cache by declaring a code page non-cacheable, which is useful in debugging and testing.

Processing of a load or store access to a non-cacheable page is much like miss processing. The cache, on receipt of the load or store request, searches its tag store

for the associated address and, when the address is not found, sends a movein request to the MMU, selects a line for replacement, and prepares to invalidate the selected line. The MMU performs address translation in the usual way and obtains the page table entry for the page containing the referenced word, either from the Translation Buffer (TB) or, in the event of a TB miss, from the page table block. The MMU examines the page table entry tag bits and, on recognizing that the page is non-cacheable, signals the cache to cancel invalidation of the line selected for replacement, and initiates a one-word memory read. The nominal execution time[5] of a load or store to a non-cacheable page is 8 cycles, if the page address is contained in the TB, and either 16 or 24 cycles otherwise (depending on whether or not the Page Table Directory entry is found in the Directory Buffer).

5.7 Inter-CPU Messages

Non-cacheable pages provide the basic mechanism for transmitting messages between CPUs. A page table entry tag bit can be set to specify that a non-cached store to a remote CPU is to be accompanied by an interrupt (valid only if the user/system tag bit is set to system). The operating system allocates *n* pages, where *n* is the number of CPUs in the system, for message sending. (The virtual address region assigned for this use is not hardware bound, but rather determined by the operating system; it may, for example, comprise the *n* virtual pages immediately below the kernel region.) Each CPU, then, has a set of system pages which are common to all its address spaces; the virtual address of each page in this set maps into a real address of one of the other CPUs in the system. This mapping is not defined by the hardware architecture, but is left to the operating system.

The use of non-cacheable pages in conjuction with the interrupt-on-write tag bit for messaging means that the MMU does not have to perform an address comparison to determine if an IPB transfer falls within an address range that is defined as a message range and so requires generation of an interrupt.

As an example, a possible mapping is shown in Figure 5.6. The virtual page in an address space of CPU *i*'s into which that CPU writes to send a message to CPU *j* is called the *outbox page* for CPU *j*. This outbox page for CPU *j* maps into a real memory address in CPU *j*'s memory called the *inbox* page; *i*'s page table entry for this page has the system, non-cacheable, and interrupt-on-write bits sets. CPU *i* sends a message to CPU *j* by writing a word containing the message operation code to word address ß**i* in the outbox page for CPU *j*, where ß is an constant determined by the operating system. The MMU performs the address translation and initiates an IPB transfer with interrupt; the interrupt is presented to the receiving CPU after the message word has been stored in *j*'s memory. *j* becomes disabled on recognition of the interrupt; *j*'s kernel retrieves the message address

[5]If the page is in local memory. Additional cycles will be required if the page is in remote memory and the access is effected via an IPB transfer; exact timing is yet to be determined.

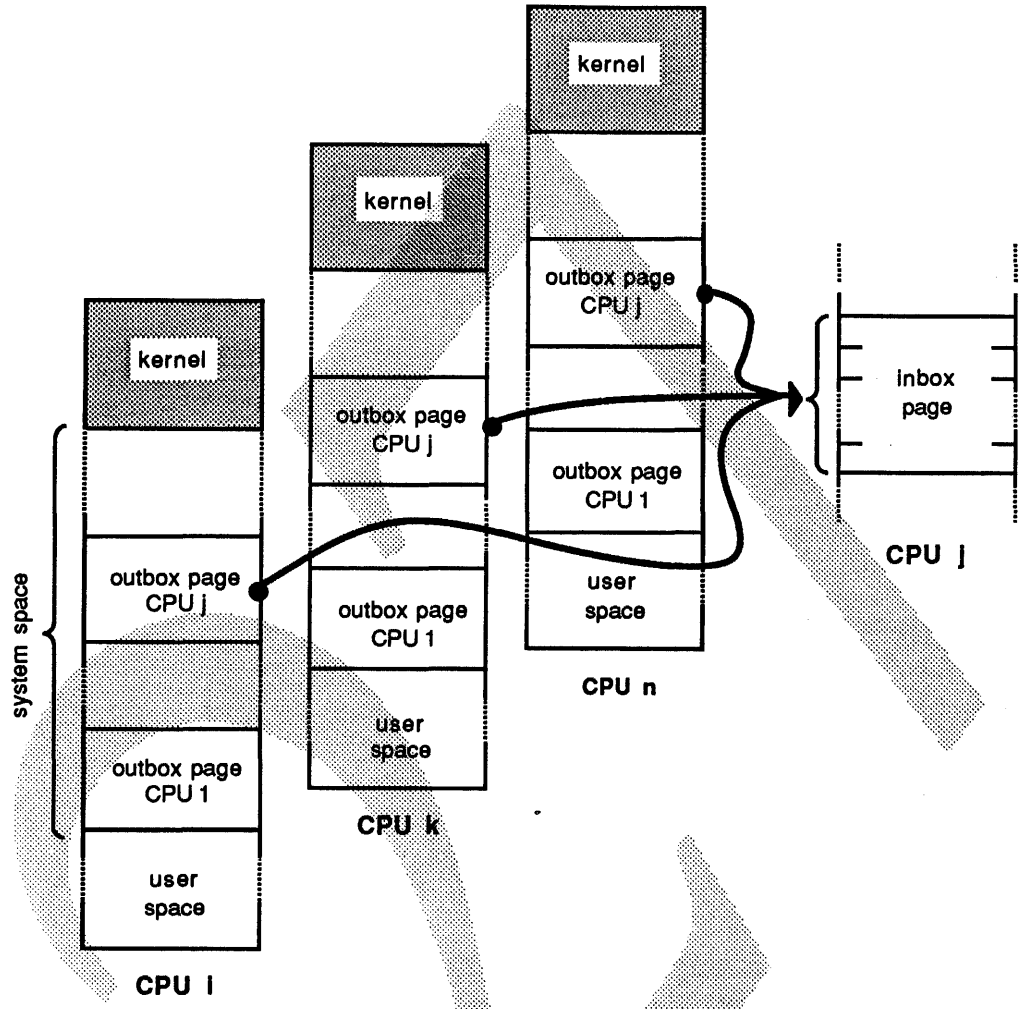


Figure 5.6. An Outbox–Inbox Mapping Scheme

from the Interrupt Argument Register, removes the message from the inbox, and sends an acknowledgement to CPU *i* (in exactly the same way any other message is sent). In this scheme, every CPU in an *n*-CPU system has *n*–1 outbox pages and 1 inbox page; every outbox for CPU *j* maps to the same real page in *j*'s memory.

If CPU *i* sends a message to CPU *j* and CPU *j* is disabled for interrupts, CPU *j*'s MMU will perform the write operation and queue the interrupt until the CPU enables interrupts and the message interrupt can be recognized. Only one message interrupt can be queued; if some other CPU, say CPU *k*, attempts to send a message to CPU *j* while CPU *j* has a queue message interrupt, CPU *k*'s message will be rejected. This rejection is effected via a synchronous negative response to CPU *k*'s IPB transfer; it blocks completion of the STW instruction which initiated

the message, and it cause a "message rejected" trap to be generated on the PU which issued the **STW** instruction.

The kernel decides how to deal with rejected messages. In a small configuration, it simply may reinitiate execution of the **STW** instruction. In a large configuration, it may use some adaptive algorithm to determine an appropriate delay before another attempt to send a message should be made, and perhaps reschedule the executing task if the delay is long.

21

6. Traps, Interrupts, and Task Switching

6.1 Traps and Interrupts

A PU executes in one of two states: system or user. The kernel executes in system state; code outside the kernel region may execute in either system or user state. Following a system reset (or power-on), PUs are placed in system state with one PU placed in execution at a fixed kernel address and the others halted. The kernel effects a transition to user or system task execution by loading global and local special registers with appropriate values and executing a pair of Return From Interrupt (**RTI**) instructions. (Two **RTI** instructions are needed because there are two Program Counters.)

A transition from user or system state task execution to kernel execution occurs as a result of an interrupt or a trap. Currently defined interrupts and traps are summarized in Figure 6.1. A trap results from the execution of a particular instruc - tion, and always is processed by the PU which executed that instruction. Traps can be classified as system calls (execution of a **TRAP** instruction) or as exceptions, which include page faults, access privilege violations, arithmetic errors, illegal operation codes, and rejected inter-CPU messages.

Interrupts usually are caused by events external to the CPU and may be processed by any PU, except for inter-PU interrupts generated by execution of an **INT** or **RES** (restart) instruction, which are processed by the PU(s) specified in the PUmask field of the **INT** or **RES** instruction. All other interrupts can be processed by any PU; the (hardware) interrupt handler assigns processing of an interrupt to a halted PU whenever possible so that interrupt processing can be done concurrently with user/system task execution. An inter-CPU, or message, interrupt occurs when one CPU executes a store instruction which causes a word or a line to be written to a page marked "interrupt-on-write" mapped into the address space of another CPU (see Section 5.7). Antares provides a pair of Event Counters (global special registers S10 and S11) which, under control of the Event Selection Register (S12),

INTERRUPTS	TRAPS
Reset	Arith. Overflow/Divide By 0*
Machine Check	Illegal Operation/Taken Branch*
Restart	Data Access Violation
Power/Temp.	Instruction Access Violation
Inter-PU (INT instr.)	Data Page Fault
Inter-CPU (message)	Instruction Page Fault
Event Counter Overflow*	Message Reject
External	Trap Instruction
* individually enabled trap or interrupt	

Figure 6.1. Interrupt/Trap Summary

can be used to accumulate counts of such things as instructions executed,moveins, moveouts, TB misses, and PU busy cycles. An enable bit in the Event Selection Register permits an interrupt to occur when one of the Event Counters overflows. External interrupts include IO and timer interrupts.

Trap or interrupt processing is initiated as follows. For certain traps, an argument is stored in the PU's Trap Argument Register); for certain interrupts, an argument is stored in the CPU (global) Interrupt Argument Register. The CPU interrupt enable flag (located in the Interrupt Argument Register) is cleared, the contents of the PU's Status/Control Register are saved in the Status Save Register, the Status/Control Register is reset, clearing the trap enable, user, cluster number, and halt bits, the current and next PC are saved in the PC Save Queue, and execution resumed at the interrupt vector address. For all traps and interrupts except machine reset, this is word address x3FF00000 + 8*N, where N is the trap or interrupt number (0-31). (Machine reset causes a transfer to location 0.) The kernel's (software) interrupt handler determines whether or not it needs to save additional state (e.g., register contents) by examining the PU's "registers available" bit in the Status Save Register. Any necessary coordination with other instances of kernel execution can be done via semaphores.

Interrupt/trap processing is controlled by a single (CPU-wide) master enable flag (in the Interrupt Argument Register) and individual PU enable flags (in each PU's Status/Control Register). The possible states of these two flags and the corresponding interpretations are shown in Figure 6.2. When an interrupt can be recognized (CPU enable flag set and a PU enable flag set for at least one PU), a PU is assigned to process the interrupt, the interrupt enable flag is cleared, and the trap enable flag of the selected PU is cleared. When a trap (or an interrupt generated by an **INT** instruction) is recognized, only the PU's trap enable flag is cleared. Once cleared, interrupt and trap enable flags remain cleared until explicitly set. However, the Reset and Machine Check interrupts override the state of the interrupt and trap enable flags. If an interrupt is presented while the CPU is disabled for interrupts, it

CPU enable	PU enable	interpretation
1	1	PU can recognize an interrupt or a trap
0	1	CPU is disabled for interrupts; PU can recognize a trap
0	0	CPU is disabled for interrupts; PU is disabled for traps or interrupts
1	0	PU is disabled for traps or interrupts; CPU is enabled for interrupts (there or may or may not be an enabled PU available)

Figure 6.2. Enable Flags and Their Interpretation

is held until until the CPU becomes enabled. If multiple interrupts are presented, the highest-numbered interrupt is recognized when the CPU becomes enabled; the other interrupts are held until the CPU is enabled once again, and the next-highest-numbered interrupt is then recognized.

Additional details on interrupt/trap processing appear in the Antares Instruction Set Reference Manual.

6.2 Task Switching

The kernel executes a task switch (or, more precisely, an address space switch) by flushing and invalidating the cache, invalidating the TB, setting the address of the page table directory origin for the new address space in the PTDO & Node No. Register, initializing (or restoring) local special registers, restoring general registers if necessary, and executing a Return From Interrupt instruction. In a virtually-addressed cache such as the Antares cache, lines in the cache are tagged with their virtual address; this does not suffice to distinguish a line in one address space from a line with the same address in another address space. (A similar situation exists for the TB.) There are two approaches to dealing with this problem.

First, information can be added to the the cache tag to uniquely identify the line: this information could take the form of an address space number (ASN) or the distinguishing part of the real line address (which substantially increases tag storage space). While adding an ASN to the tag is less demanding in terms of tag storage, the MMU becomes much more complicated. If address space B is active and a line from B replaces a modified line of address space A, the MMU has to retrieve the page tables of address space A in order to translate the virtual address of the modified line prior to its moveout.

Second, the cache can be emptied on an address space switch so that lines from different address spaces cannot be in the cache at the same time. This has two performance costs: the direct cost of the cycles required to carry out the flush and

invalidate operation (including the time required to write modified lines to memory), and the indirect cost of discarding lines which, when the original address space is returned to execution, will have to be brought back in to the cache.

Antares uses the second approach: the cache is flushed on an address space switch. This approach is chosen for hardware simplicity and to minimize tag space. Because of the small cache size of the current design, the performance penalty of flushing is not expected to be severe. In a switch from address space A to B and back without flushing (A → kernel → B → kernel → A), the number of lines of address space A remaining in the cache when A is returned to execution is likely to be small, so that flushing does not greatly increase the number of misses inherent in a task switch. Cache flushing was described in Section 3.6. The Invalidate Translation Buffer (**ITLB**) instruction clears the Directory Buffer as well as the Translation Buffer.

Later implementations of the Antares architecture, with larger caches, probably will adopt a different approach to this problem. The architectural visibility is very low, so different implementations should not present a compatibility problem.

7. References

Amdahl [1967]
Amdahl, G. Validty of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *Proc. AFIPS 1967 Spring Joint Computer Conf.*, Vol. 30, 483-485.

Birnbaum and Worley [1985]
Birnbaum, J. S., and Worley, W. S. Jr. Beyond RISC: High-Precision Architecture. *Hewlett-Packard Journal* (Aug. 1985), 4-10.

Blau [1985]
Blau, R. Paging on an Object-oriented Personal Computer. Proceedings 1983 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems (Aug. 1983), 44-54.

Clark and Emer [1985]
Clark, D. W., and Emer, J. S. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Transactions on Computer Systems 3, 1* (Feb. 1985), 31-62.

Cohen [1981]
On Holy Wars and a Plea for Peace. *Computer 14, 10* (Oct. 1981), 48-54.

Davidson, J. A Split Cache Design for High-Level Language Processors. ACM Transactions on Computer Systems (to appear).

DeMoney et al [1986]
DeMoney, M., Moore, J., and Mashey, J. Operating System Support on a RISC. *Proceedings IEEE 1986 Compcon* (Mar. 1986), 138-143.

Flynn [1972]
Flynn, M. J. Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, 21, 9 (Sept. 1972), 948-960

Gross and Hennessy [1982]
Gross, T. R., and Hennessy, J. L. Optimizing Delayed Branches. *Proceedings 15th Annual Workshop on Microprogramming* (Oct. 1982), 114-120.

Hennessy et al [1982]
Hennessy, J. L., Jouppi, N., Baskett, F., Gross, T. R., and Gill, J. Hardware/Software Tradeoffs for Increased Performance. *Proceedings SIGARCH/SIGPLAN Sym. on Architectural Support for Programming Languages and Operating Systems* (Mar. 1982), 2-11.

Hennessy [1984]
Hennessy, J. L. VLSI Processor Architecture. *IEEE Trans. on Computers 33, 12* (Dec. 1984), 1221–1246.

Hennessy [1985]
Hennessy, J. L. VLSI RISC Processors. *VLSI Systems Design* (Oct. 1985), 22–32.

Patterson [1985]
Patterson, D. A. Reduced Instruction Set Computers. *Comm. of the ACM 28*, 1 (Jan. 1985), 8–21.

Worlton [1981]
Worlton, W. J. A Philosophy of Supercomputing. *Los Alamos National Laboratory Report LA-8849-MS* (June 1981).