MultiTitan: Four Architecture Papers:

MultiTitan Central Processor Unit

MultiTitan Floating Point Unit

MultiTitan Cache Control Unit

MultiTitan Intra-Processor Bus

Digital Equipment Corporation
Western Research Laboratory
100 Hamilton Avenue
Palo Alto, CA 94301

Version of 5 April 1988

1. Introduction to the MultiTitan

This document is a revised collection of four working architecture documents for the MultiTitan project originally published in 1986. The MultiTitan was a research project at the Western Research Lab from mid 1984 through 1987. Because of delays, primarily due to lack of staff in critical stages of the project, and the consequent loss of research potential in several areas of the program, in January 1988 WRL redirected its efforts beyond the MultiTitan.

Since it was research project, it was specifically intended not to be a product in itself, but rather a testbed for many different ideas. Thus ideas proved in the project and experience gained in the project can be useful in future product designs. Research beyond product development is important in that research can afford to try out high payoff but high risk ideas that would be too risky to directly incorporate into products.

2. Research Goals of the MultiTitan

There were four main research areas in the MultiTitan. In general each area has the potential to improve system performance by a factor of 2x or more, however specific aspects of each research area may only contribute a few percent. The four areas were:

- Small-grain multiprocessing
- Architecturally explicit caches
- High performance / low cost floating point
- Software instruction set architectural definition

These four areas will be explained more completely in the next four sections. As well as these four explicit research areas, the MultiTitan provided a testbed for research on VLSI CAD tools and low-latency machine design techniques.

2.1. Small-Grain Multiprocessing

The MultiTitan was to consist of 8 processors. This number was chosen as the most that could be supported by the existing Titan memory and I/O system. We intended to investigate the speedup of single application programs broken up into pieces on the order of 100 instructions. This relatively fine-grain parallelism has wider applicability than large-grain or process-level parallelism, since a system efficient enough for small-grain parallelism can support large-grain parallelism, but not necessarily vice versa. The parallelism within an application was to be scheduled by the compilers, since such small threads could not afford the overhead of a call to the operating system for scheduling. The MultiTitan hardware provided Send and Receive instructions that processors could use to synchronize and transfer cache lines between processors in less time than required for a main memory access. The usual test-and-set instruction was also provided without changing the original Titan memory system. The resulting shared memory model was to be explored with a relatively small number of processors, but the work was intended to be extensible to many processors.

2.2. Architecturally Explicit Caches

Until recently, it has been taught that caches are an implementation technique, and should not be visible in an instruction set architecture. However, with the advent of multiprocessors, much faster individual processors, and relatively constant main memory access times, caches have become a very important component of system performance. For example, a cache miss on a VAX 11/780 takes 0.6 average instruction times to refill, but a cache miss on a Titan takes 10 average instruction times to refill, and the Titan main memory is twice as fast as the 780 memory. This trend seems to be increasing further in proposed future machines with two-level caches. Similarly,

caches have a significant effect on multiprocessor performance. If a process is assigned to a different processor for every quantum of its execution time, it could spend the majority of its time loading the current cache from main memory or its previous processor's cache. Likewise, if a program is decomposed over several processors, and there is much contention for shared data, the program may waste much of its time passing data between caches or invalidating stale cache lines.

First, it seems clear that something that can be the most significant term in a program's performance should not be hidden from the program, but should be visible and controllable by software techniques that can improve the performance of the program. Second, the effort required to hide the caches from the architecture has increased significantly from single-level uniprocessor caches to recent multi-level multiprocessor cache coherency proposals. Finally, if access of shared data between caches is known to be relatively infrequent (less than one sharing access per 100 machine cycles), a hardware cache consistency mechanism which decreases non-shared data performance by only 7% (one additional level of logic in a machine with 15 gate levels per cycle) will be a net performance loss unless its performance for shared data is faster by a factor of 7 over methods which manage cache consistency by cache directives generated by the compiler. These hardware cache consistency methods are also harder to scale to larger numbers of processors and require more design time than a machine without hardware cache consistency, especially since the asynchronous consistency traffic is hard to model exhaustively.

Each MultiTitan processor has a write-back non-snoopy cache. Three cache management instructions are provided for control of the cache by the compiler:

Clear	This instruction allocates a cache line for the specified address. If the line is already allocated, this instruction has no effect. This instruction can be used to prevent normal fetch-on-write of cache lines that will be completely overwritten, hence improving machine performance.
Write-back	This instruction writes back a cache line for the specified address if it is dirty. It has no effect if the line is clean, or if the specified address is not present in the cache. This instruction can be used in cases where data is needed by either another processor or by the I/O system, but it is also required by this processor in the future.
Flush	This instruction removes a cache line for the specified address from the cache and writes it back to main memory if it is dirty. It has no effect if the address is not present in the cache. This instruction is useful when a new version of data must be acquired, so the old version must be discarded so that a new version can be fetched from main memory by the normal cache miss refill mechanism.

2.3. High Performance / Low Cost Floating Point

High performance floating point is becoming increasingly important. This is true even in mid-range and low-end computing where cost is important. In general, scalar floating point performance is more important than vector performance. Many applications do not vectorize, and often more time is spent in the non-vectorizable parts of "vector" benchmarks such as Livermore Loops than in the vectorizable parts when running on vector machines. Clearly if scalar computations could be made almost as fast as vector computations, the distinction between scalar and vector computations would diminish. It would diminish even more if vector support did not require the addition of hardware equal to or greater than the hardware required for the scalar processor. Not only is scalar performance more important than vector performance, but the start-up costs of vector floating-point operations determine the vector length required for efficient operation. Although 100x100 Linpaks have given way to 300x300 Linpaks in supercomputer benchmark popularity, many applications will always have very short vectors. For example, 3-D graphics transforms are expressed as the multiplication of a 4 element vector by a 4x4 transformation matrix. Finally, in scalar operations data dependencies between operations are very important. Being able to perform many scalar floating point operations in parallel is of little use if each one has a high latency.

Three key features distinguish our work in floating point support: a unified approach to scalar and vector processing, low latency floating point, and simplicity of organization.

2.3.1. A Unified Approach to Scalar and Vector Processing

Existing machines that support vectors and use a load/store architecture (i.e., they support only register-to-register arithmetic) provide a separate register set for vectors from scalar data. This creates a distinction between elements of a vector and scalars, where none actually exists. This distinction makes mixed vector/scalar calculations difficult. When vector elements must be operated on individually as scalars they must be transferred over to a separate scalar register file, only to be transferred back again if they are to be used in another vector calculation. This distinction is unnecessary. The MultiTitan provides a single unified vector/scalar floating-point register file. Vectors are stored in successive scalar registers. Each arithmetic instruction contains an operand length field, and scalar operations are simply vector operations of length one.

With this organization, many operations that are not vectorizable on most machines can be vectorized. For example, since the normal scalar scoreboarding is used for each vector element, reduction and recurrence operations can be naturally expressed in vector form. For example, the inner loop of matrix multiplication consists of a dot product in which the elements of a vector multiply must be summed (i.e., a reduction). This can easily be performed without moving the data from the multiply result register with either individual scalar operations, a vector directly expressing the reduction, or the summation expressed as a binary tree of vector operations of decreasing length (e.g., 8, 4, 2, 1). Likewise, the first 16 Fibonacci numbers (i.e., a recurrence) can be computed by initializing R0 and R1 to 1 (Fib_0 and Fib_1) and executing R2 <- R1 + R0 (length 14).

2.3.2. Low Latency Floating Point

Data dependencies increase the value of low latency floating point, as compared to high bandwidth but high latency approaches. Optimizing compiler technology often increases the importance of low latency operations by removing redundant or dead code which would otherwise be executed in parallel with multi-cycle data-dependent operations. In the MultiTitan the latency of all floating-point operations is three cycles, including time required to bypass the result into a successive computation. This is very short in comparison to most machines. (Division is a series of 9 3-cycle operations.) When multiplied by the 40ns cycle time of the MultiTitan, these result in latencies that are only 2-3 times larger than a Cray X-MP, and provide unparalleled scalar performance for a single-chip floating-point unit.

2.3.3. Simplicity of Organization

The MultiTitan floating point is a very powerful yet simple and cost-effective architecture. All floating-point functional units (including scalar/vector floating-point registers) easily fit on one CMOS chip in today's technology. (In the next CMOS technology they could easily fit on the CPU chip.) All floating-point coprocessor operations take the same amount of time, greatly simplifying the scoreboard logic. Sustained execution rates of 20 double-precision MFLOPS with vectorization and 15 MFLOPS without vectorization are attainable.

2.4. Software Instruction Set Architectural Definition

One aspect of the original Titan work was an architecture defined at a software level instead of as hardware object-code compatibility. This software definition of the architecture is called Mahler. All of the compilers available on the Titan produced Mahler instead of machine language, and with very rare exceptions so did any user who wanted assembler-level code.

The Mahler compiler translates from Mahler to the specific (and different) object code for each machine in the Mahler family. The feasibility of this approach is difficult to verify given only one machine in a family; one

research goal of the MultiTitan was to test the flexibility of Mahler. For example, the Titan and MultiTitan have different instruction encodings and substantially different interlocks. The Titan also has more general-purpose registers than the MultiTitan but does not have the MultiTitan's floating-point register set. Finally, the MultiTitan supports vector operations while the Titan does not.

The goal was that the Mahler code for both machines be the same. This goal was attained for most practical purposes. The only changes made to the front end compilers was to implement as double-precision reals those data types that are usually implemented as single-precision reals, because the latter are not supported by the MultiTitan. We would also have needed front-end extensions to exploit the MultiTitan vectors, but this would have required no changes to the Mahler base language generated by the front ends.

The Mahler system, including preliminary results of retargeting to the MultiTitan, is described more fully in WRL Research Report 87/1, "The Mahler Experience: Using an Intermediate Language as the Machine Description" by David W. Wall and Michael L. Powell.

3. Acknowledgements

Many people have contributed to the MultiTitan over the three and a half year history of the project. The following is a list of the people and their contributions:

Bob Alverson	Multiplier design, RSIM enhancements (summer intern).
Joel Bartlett	GPIB and tester software.
Jon Bertoni	Livermore Loops benchmarks.
David Boggs	Uniprocessor system design, Multiprocessor system design.
Anita Borg	MultiTitan Unix locore, proposed operating system structure.
Jeremy Dion	MultiTitan system architecture and design, MultiTitan system simulations, Cache Controller architecture and design, PCB router.
Mary Jo Doherty	Floating-point unit pipeline control and simulations.
Alan Eustace	Floating-point multiplier and reciprocal approximation, schematics tools, CAD environment.
John Glynn	Fab support (at Hudson).
Norm Jouppi	MultiTitan CPU, floating-point, and system architecture; CPU design, timing verification, Magic ports and enhancements, Versatec plotter software, CAD environment.
Chris Kent	MultiTitan system design.
Brian Lee	Floating-point adder (intern).
Jud Leonard	Floating-point algorithms.
Jeff Mogul	Proposed operating system structure.
Scott Nettles	Magic well-checker, Magic under X, fab support.
Michael Nielsen	MultiTitan system simulations.
John Ousterhout	Proposed operating system structure.
Michael Powell	Compilers, operating system architecture, synchronization primitives, benchmark results, SPICE port from VMS.
Don Stark	Resistance extraction, whole-chip power noise simulation, Magic enhancements (summer intern).
Patrick Stephenson	GPIB and tester software (summer intern).
Jeff Tang	Cache Controller design, clock generator, pads, electrical issues, SPICE, fab support.

Silvio Turrini	Floating-point adder.
David Wall	Mahler, instruction-level simulations.
Tat Wong	Process engineer (at Hudson).

Table of Contents

MultiTitan CPU

Norman P. Jouppi
Digital Equipment Corporation
Western Research Laboratory
100 Hamilton Avenue
Palo Alto, CA 94301

Version of 5 April 1988

1. Introduction

MultiTitan is a high-performance 32 bit scientific multiprocessor implemented in CMOS. Each processor consists of three custom chips: the CPU, floating point coprocessor, and external cache controller. They are abbreviated "CPU", "FPU", and "CCU" in this document. This document describes the central processor unit.

Each processor of MultiTitan is similar in many resects to the ECL Titan, but different in several others. MultiTitan is not object code compatible with the ECL Titan. Like the ECL Titan, it is a very simple RISC machine with a branch delay of one. Unlike the ECL Titan, the MultiTitan has hardware support for small-grain parallel processing, vector floating point registers and operations, and a different pipeline and method for handling exceptions. Figure 1-1 is an overview of one MicroTitan processor, while Figure 1-2 illustrates a MultiTitan system consisting of eight MicroTitan processors.

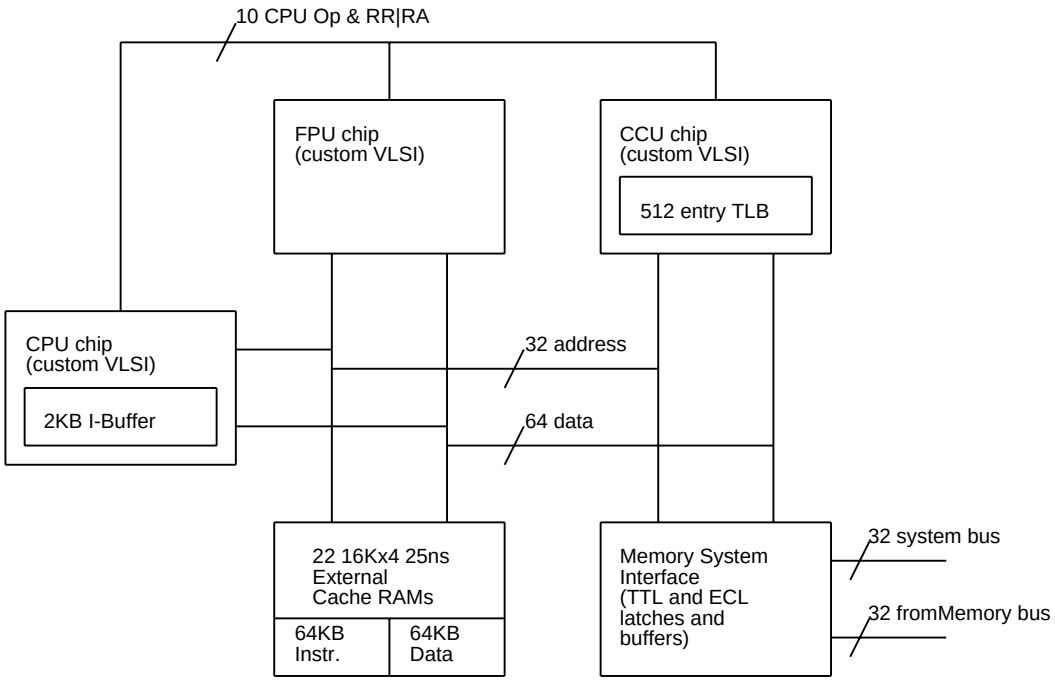


Figure 1-1: Block Diagram of One MicroTitan Processor

Each MicroTitan processor has a large instruction buffer on the CPU chip as well as a large external cache containing both instructions and data. The CPU instruction buffer is a cache containing 512 words, and is direct-mapped with a line size of 4 words. The instruction buffer is addressed with virtual addresses. The 128K byte external cache is partitioned into 64K bytes of data and 64K bytes of instructions, also with a line size of 4 words. Both the data storage and tag storage are constructed from commercial 16Kx4 20ns static RAMs. The external cache is a physically addressed cache, and the TLB access is in parallel with the cache access. In order for this to occur (without sharing of unmapped and mapped bits resulting in restrictions on page placement in main memory) the page size must be greater than or equal to the cache size. The smallest page size given these constraints has been chosen, resulting in a page size of 64K bytes. The external cache is write-back (as opposed to write-through). The instruction buffer and instruction partition of the external cache do not monitor D-stream references, so writing into the instruction stream will have unpredictable effects unless the caches are flushed appropriately. The entire on-chip instruction buffer is cleared by all context switches. The external cache may be cleared on a line by line basis with a cache flush instruction.

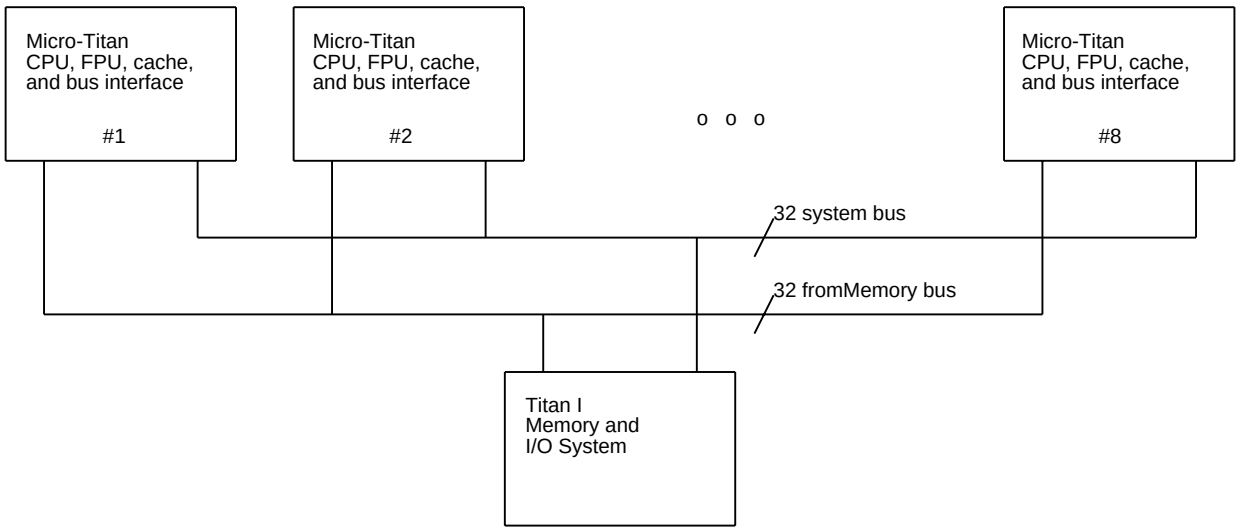
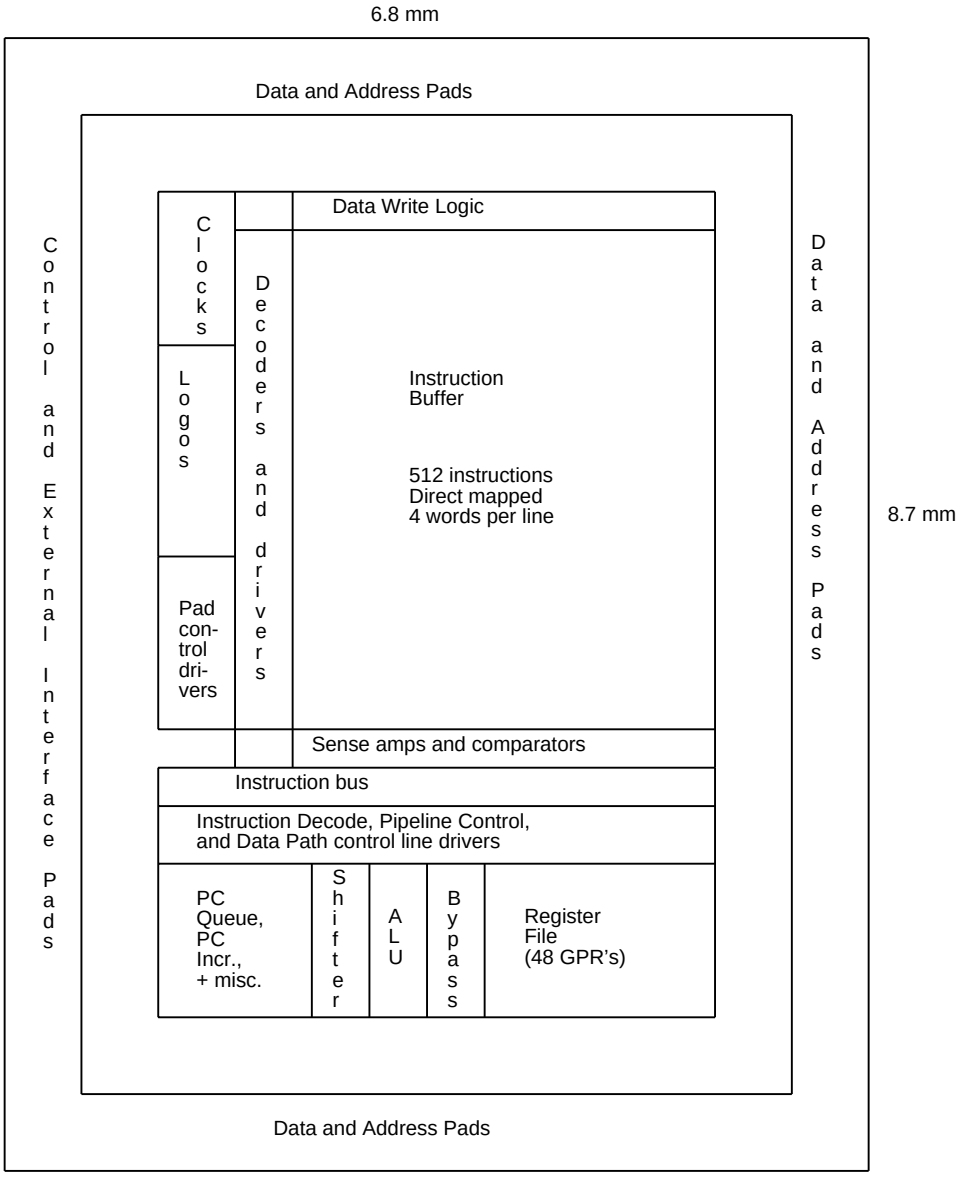


Figure 1-2: MultiTitan System Block Diagram

The CCU handles virtual to physical address translation, interface to I/O devices, low overhead processor to processor communication, interactions with the memory system, and control of the cache during CPU memory references. It provides support for software control of shared data, as opposed to hardware control of shared data (e.g., a snoopy bus). In direct mapped caches (where data store and tag store are implemented with the same speed RAMs), provisional data is available a significant amount of time before hit or miss is known. The CPU takes advantage of this property; it starts using the data a cycle before hit or miss is required from the CCU. Similarly, page faults are not known until after the data may have been written back into the register file. Thus, when a memory reference page faults, the instruction is allowed to complete *in error* before an interrupt occurs. Kernel software resumes execution with the instruction that caused the page fault. Note that this requires all memory references to be idempotent. The details of virtual address translation are orthogonal to the CPU chip itself. Please consult the CCU specification for details.

The floating point coprocessor performs G-format double precision (64 bit) floating point addition, subtraction, multiplication, reciprocal approximation, conversion to and from integer, and single precision (32 bit) integer multiplication. These operations take place concurrently with normal instruction processing of the CPU, except that the CPU and CCU wait for completion of operations when they need a result from the coprocessor. The FPU has 52 general purpose registers, and supports vector arithmetic operations. The CPU chip is the only chip to generate memory addresses. In coprocessor loads and stores the CPU chip generates addresses as if it were a load or store of its own register file, but ships the register address to the coprocessor. The coprocessor then either latches or sends out the data.

The floorplan of the CPU is given in Figure 1-3. The pipeline and CPU organization is given in Figure 1-4. Note that the organization chosen is efficient in its use of datapath resources. For example, only one ALU is required: it is shared between address computations and arithmetic operations, and is used at the same time for both. This plus other efficiencies allows the datapath to be small, even though MicroTitan has more registers than most machines.



Scale: 1/2" = 0.75mm in CMOS-2

Figure 1-3: CPU Floorplan

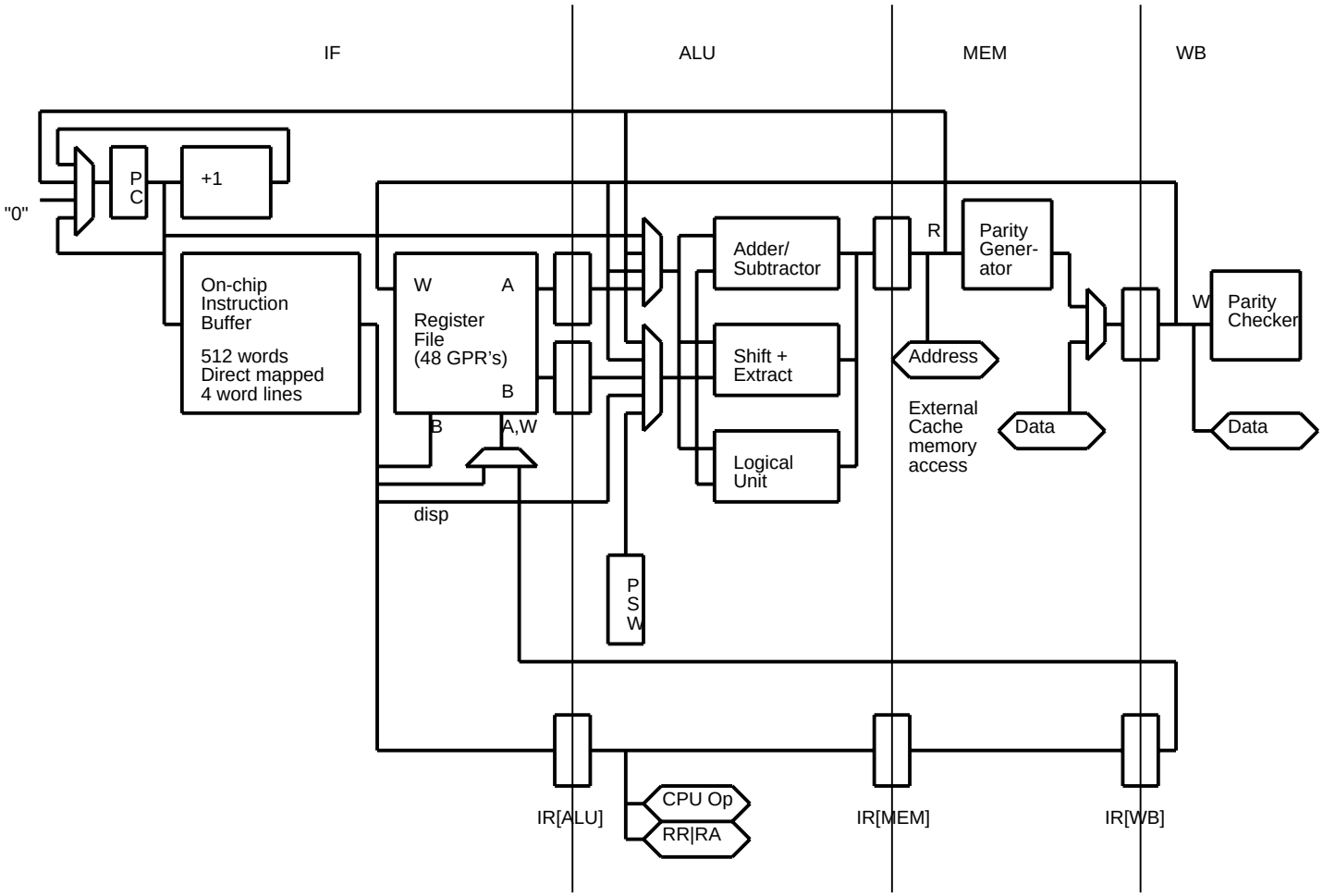


Figure 1-4: CPU Pipeline and Machine Organization

2. Instruction Set Architecture

Several overriding concerns determined the encoding of the instruction set.

First, in order for instruction source registers to be fetched in parallel with decoding of the opcode, the register sources must be in one and only one position in all instructions. Since store instructions read their operand to be stored at the same time as loads write their target, and arithmetic operations write their destination at the same time as loads, both store sources, load destinations, and arithmetic destinations must be in the same place. Some instructions, like add immediate, have only one register source and destination, so this constrains one source and the destination register to be specified by the high order halfword, since the displacement is in the low order halfword. As in the ECL Titan, there is enough encoding space for 64 registers (although only 48 GPR's are implemented). With a four bit opcode we will neatly use the upper halfword: the opcode resides in the highest four bits, followed by the destination (rr) and the first source (ra).

Second, the opcode should be trivial to decode. Thus, the instruction opcodes are arranged so that one or two bits determine most functions. For example, all instructions with 16 bit displacements have as their highest bit "1". As another example, all the instructions with a valid destination register can be covered with two boolean n-cubes. The encodings are given in Figure 2-1.

```
Opcode   Instruction
-------------------------------------
  0      trap
  1      extract
  2      undefined operation
  3      variable extracts
  4      CPU to coprocessor transfer
  5      coprocessor to CPU transfer
  6      coprocessor ALU (+,-,*,/,convert to or from FP)
  7      CPU ALU
  8      undefined operation (reserved for CPU store byte)
  9      test operation (formerly CPU load byte)
 10      coprocessor store
 11      coprocessor load
 12      CPU store
 13      CPU load
 14      conditional jump
 15      add immediate (replaces Titan I jump, RTI, and set oldpc)
```

Figure 2-1: MultiTitan Instruction Opcodes

2.1 CPU Registers

CPU registers are named r0 through r63. The expression "rx" refers to the contents of the register whose number is encoded in the rx field of the current instruction, where rx is either ra, rb or rr. There are two instruction formats, illustrated in Figure 2-2.

CPU registers 48 through 63 are special registers. They cannot be read and written in all circumstances as can general purpose registers. For example, the program counter (special register 63) can only be read as the first source and not as the second. The circumstances in which specials may be used are discussed later. To avoid confusion, the special registers will be referred to by their name and not by their number in this document. They can only be accessed by CPU instructions (not CCU or FPU instructions). The special CPU registers are listed in Figure 2-3. Using special registers as sources for instructions in circumstances other than those in the listed restrictions will return undefined data but will not cause any exception or errors in that or other instructions. Moreover, using the

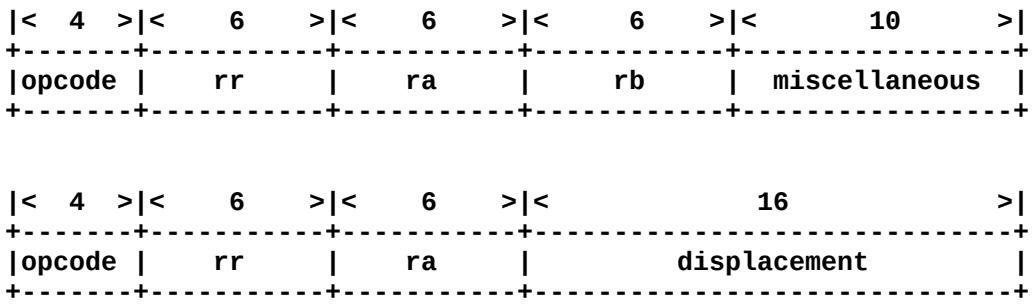


Figure 2-2: Instruction Formats

PSW as a destination, since it is write-only, will also have no effect. Thus the recommended No-op is "PSW := PSW op PSW", where op is any ALU or shift operation.

Number	Name	Restrictions
63	PC	Only for rr in ALU or add imm, ra in all instructions
62	PSW	Only for rb in ALU or var byte (it is read only)
61	PCQE	Only for rb in ALU or var byte, rr in ALU or add imm
60	PCQ	Only for rb in ALU or var byte, rr in ALU or add imm
59-48	reserved for future use	

Figure 2-3: CPU Special Registers

PCQ is a queue of four address: IFpc, ALUpc, MEMpc, and WBpc. When the processor is not in kernel mode, successive values of the pc enter the queue. When a trap occurs, WBpc contains the pc of the instruction which was in its WB pipestage, MEMpc the next instruction, ALUpc the third, and IFpc the address of the instruction in its IF pipestage. If nil instructions are inserted into the pipeline as a result of interlocks or instruction buffer misses, the nil instructions have the same pipestage pc as the next valid instruction in the pipeline. For example, if WBpc, MEMpc, and ALUpc all contain the same value, only ALUpc refers to a valid instruction. Reading PCQ reads WBpc, while writing it writes IFpc. Note that since instruction PC's are duplicated by interlocks or instruction buffer misses, WBpc cannot be used in user mode as the address of a previous instruction. Reading PCQE (PCQExit) reads WBpc, but has the side effect of exiting kernel mode after a branch delay of one instruction.

2.2 Coprocessor Registers

Coprocessors share a 6 bit register address space; the registers are named c0 through c63.

The FPU has 52 GPR's and 3 special registers: FPU PSW, time-of-day clock, and interval timer. They are addressed 0 to 54. The special registers can only be accessed by coprocessor load and store instructions; when accessed by FPU ALU instructions they return the constants 0, 1/2, and 1.

Instructions that access registers in the CCU use coprocessor registers 55 to 63. Access to some registers may cause a CPU interrupt if the process is not running in kernel mode. The specific encoding is given in the CCU Architecture document. It encompasses the functionality shown in Figure 2-4.

```
    Function
-----------------------------------------------------
    Flush cache line
    Clear cache line
    Test and set line in main memory
    I/O operations
    Load/store another processor's cache
    Load/store CCU PSW
    Load/store TLB fault register
    Load/store TLB tag (set 1 through 4)
    Load/store TLB data (set 1 through 4)
```

Figure 2-4: CCU Register Addresses

2.3 Instructions

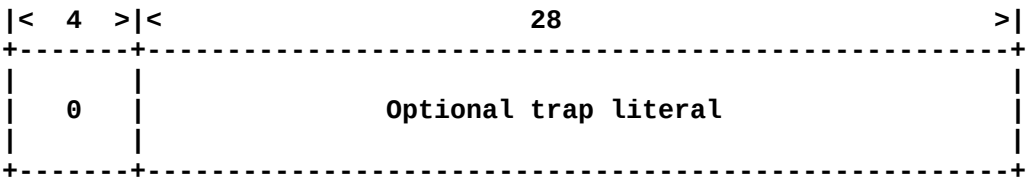
For each instruction, we list its name, its assembly language form, its memory format, and a brief description of its operation. The syntax for the Titan assembler has been extended to provide for the MultiTitan. Note that c0 - c63 denote the coprocessor registers, .. and .: denote variable extracts (bit field and byte respectively).

2.3.1 Trap

TASM Format

```
trap   literal
```

Memory Format



This instruction causes a trap (i.e., software interrupt) in user mode. During kernel mode it is a No-op. The interrupt is asserted duing the instruction's WB pipestage. The optional trap literal is not saved in any CPU register but must be obtained by examining the instruction itself.

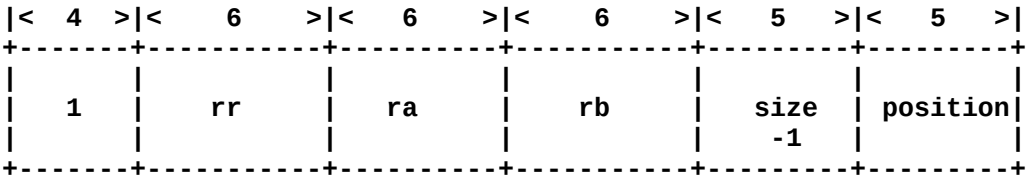
```
Example:        trap;
```

2.3.2 Extract Field

TASM Format

```
rr := ra,rb.[size, pos];
```

Memory Format



Registers ra and rb are concatenated to form a 64 bit word, with ra on the left. A contiguous field is extracted from this quantity, right justified and zero-extended to 32-bits, and stored in register rr.

Field extraction is accomplished by right shifting rb by the value in the position field, filling in the high order bits from ra. Thus a zero in this field implies that rb will appear unshifted in the result, while 31 in the position field implies that most significant bit of rb will be the least significant bit of the result, and all but the most significant bit of ra will appear in the upper 31 bits of the result.

The size of the extracted field is one greater than the value in the size field of the instruction; e.g. zero in the size field of the instruction causes a single bit to be extracted, while 31 in the size field obtains a 32-bit result.

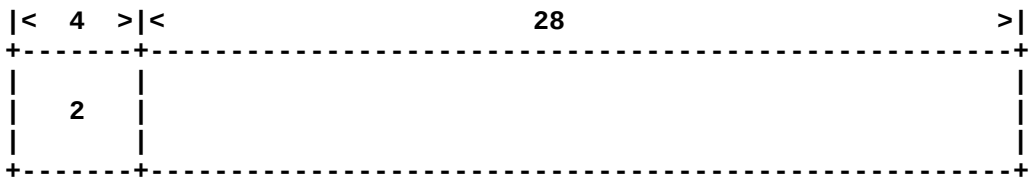
```
Examples:
      r3 := r4,r4.[0,31];        /* puts sign bit of r4 in r3 lsb */
      r3 := r10,r10.[31,4];      /* rotates r10 right by 4 */
      r3 := r7,r7.[7,24];        /* extract high order byte of r7 */
```

2.3.3 Undefined Operation

TASM Format

no TASM format

Memory Format



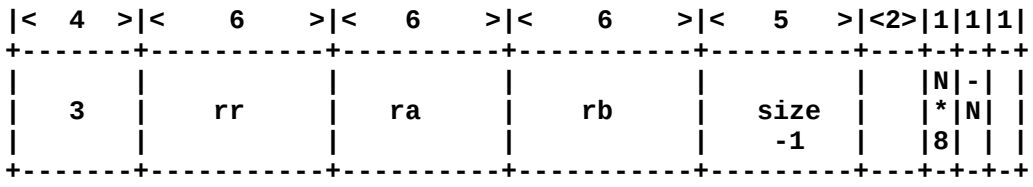
This instruction causes an unpredictable operation. It will not cause an illegal instruction opcode trap, since there is no such trap. However, it may cause a privilege violation, write a register, or write a memory location, but its operation is unknown and implementation dependent. This instruction should never be generated.

2.3.4 Variable Extract

TASM Format

```
rr := (ra,rb)..[size];
rr := (ra,rb)..-[size];
rr := (ra,rb).:[size];  (byte ref)
rr := (ra,rb).:-[size]; (byte ref)
```

Memory Format



A field extract is performed with the position field taken from the least significant 5 bits of the contents of ra. Note that ra is used both as the high order word of the field to be selected from, and as the position count. This position is interpreted according to the N*8 and -N bits. If the -N bit is on, the position is complemented (e.g., in a right bit shift of 30 becomes a right shift of 1). This results in a big endian (i.e., IBM order, where the most significant bit is numbered 0) variable shift being performed. If the -N bit is off, then a little endian right shift is performed (i.e., VAX order, where the most significant bit is numbered bit 31). If N*8 is on, then the position is shifted left 3 bits and filled with zeros. This results in a variable byte right shift. Both the N*8 and -N bits may be asserted at the same time, in which case a big endian byte shift results.

The size of the extracted field is one greater than the value in the size field of the instruction; e.g. zero in the size field of the instruction causes a single bit to be extracted, while 31 in the size field obtains a 32-bit result.

In the notation used to describe the extract field instruction:

```
        if (-N)
        then pos = ~ra
        else pos = ra
        if (N*8)
        then pos  = 8 *(pos mod 4)
        else pos  = pos mod 32
        lbit = 32 + pos - size
        rbit = 32 + pos.
```

Examples:

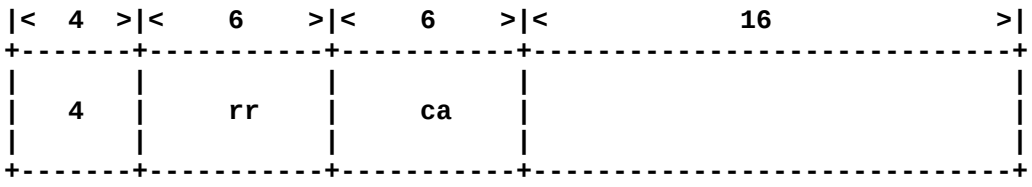
```
          r2 := r3,r4.[0];          /* one bit */
          r2 := r3,r4.-[0];         /* one bit, reversed ordering */
          r2 := r3,r4.:[7];         /* one byte */
          r2 := r3,r4.:-[15];       /* 2 bytes, reversed ordering */
          r2 := r3,r4.:[4];         /* low 5 bits of indicated byte */
```

2.3.5 CPU to Coprocessor Transfer

TASM Format

```
ca := rr;
```

Memory Format



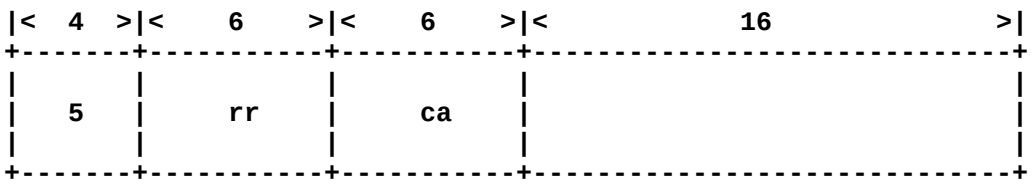
Rr is a CPU register, ca is a coprocessor register. The CPU performs a store instruction, but the CCU does not enable the memory. The CPU outputs rr onto the high order data lines during its WB pipestage (i.e., word "1"). A coprocessor writes the high order word of register ca with the data, and the low order half of register "ca" becomes undefined. This instruction is useful for transferring operands to the FPU for integer multiplies. The register address "ca" plus the opcode are transferred to the coprocessors in the ALU pipestage.

2.3.6 Coprocessor to CPU Transfer

TASM Format

```
rr := ca;
```

Memory Format



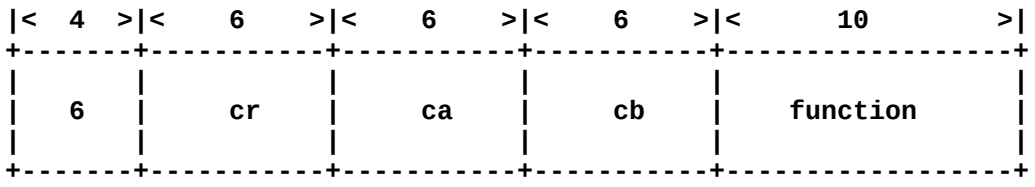
Rr is a CPU register, ca is a coprocessor register. The CPU performs a load instruction, but the CCU does not enable the memory. A coprocessor outputs the high order word of ca onto the high order data lines (i.e., word "1"). The CPU reads the data at the beginning of its WB pipestage. This is useful for obtaining results from the FPU for integer multiplies. It is also used to transfer the result of FPU comparisons to the CPU for testing by conditional branches. If the coprocessor register specified by ca is not yet available due to a computation in progress, the coprocessor will deassert LoadWB until it can output the result. The register address "ca" plus the opcode are transferred to the coprocessors in the ALU pipestage. *This instruction cannot appear in the branch delay slot of any branch immediately following a CPU->Coprocessor transfer. If it does, unpredictable operation may result.*

2.3.7 Coprocessor ALU

TASM Format

```
cr := ca(ALU)cb;
```

Memory Format



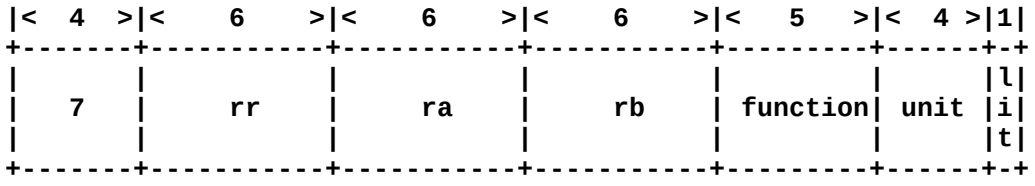
A coprocessor performs an ALU operation. The CPU ships the coprocessor the entire instruction over the address lines during its unused mem pipestage. Please consult the FPU architecture document for more details about this instruction.

2.3.8 CPU ALU

TASM Format

```
rr := ra op rb;
```

Memory Format



The ALU performs boolean or arithmetic operations on the A and B operands, storing the result in register rr. The B operand is register rb. The A operand is register ra if the literal select bit is clear, otherwise it is the ra field of the instruction, zero-extended to 32 bits. The unit field selects a functional unit. Codes with more than one bit set (and hence more than one unit selected) produce the logical AND of the results of the selected units. The unit codes are:

```
0: all one's
1: add and sub
2: comparisons
4: logical (boolean)
8: reserved
```

The function is interpreted depending on the unit specified. The first tables below specify the logical operations performed by each functional unit and how they are encoded. Subsequent tables and information provide implementation specific description of how the functional units work. If checks for arithmetic overflow are enabled, and an arithmetic overflow occurs, then an overflow trap is generated during the instruction's WB pipestage.

```
Add and Subtract: Function Field
Value   Operation
------------------------
0xxV0   a+b             xx denotes 2 "don't care" bits.
1xxV1   b-a             V=1 specifies trap on overflow
0xxV1   a+b+1
1xxV0   b-a-1
```

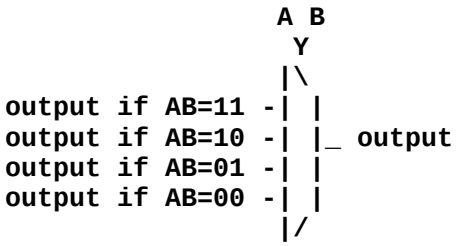
```
        Logical: Function Field
        (Most significant bit of field doesn't matter)
        Hex
        Value   Operation
        ------------------------
        8       and
        1       nor
        E       or
        6       xor
        A       B bus
        C       A bus
        9       eqv
        5       not B
        3       not A
        0       False (0)
        F       True (1)
        2       B and (not A)
        4       A and (not B)
        7       not (A and B)
        B       (not A) or (A and B)            => not (A and not B)
        D       (not B) or (A and B)            => not (B and not A)

        Comparison: Function Field
        (Most significant bit must be 1)
        Hex
        Value   Operation
        -----------------
        1A      a <u  b  \
        1B      a <=u b   \ unsigned
        15      a >u  b   / comparisons
        14      a >=u b  /
        1E      a <   b  \
        1F      a <=  b   \ signed
        11      a >   b   / comparisons
        10      a >=  b  /

Add and subtract function encoding:
        (msb and lsb both on for subtract, both off for add)
        msb: complement A src
             (don't care)
             (don't care)
             trap if overflow detected
        lsb: carry in

Logical function encoding:
        results for each bit in the data path (0<=i<=31):
        msb: (don't care)
             output[i] if A[i]B[i]=11
             output[i] if A[i]B[i]=10
             output[i] if A[i]B[i]=01
        lsb: output[i] if A[i]B[i]=00
```

For example, subtract (without trap on overflow) is selected by asserting unit code 0001b (the b suffix denotes a boolean number) and function code 10001b. The logical unit is a four input multiplexor for each bit (0<=i<=31) in the data path, controlled by the Asrc and Bsrc for that bit. For example, a logical AND is performed when the unit code is 0100b and the function bits are x1000b.



Comparison instructions set the sign bit of rr to 1 if the relation is true, 0 otherwise. The value of the other bits in rr is 1. The compares denoted with a trailing u are unsigned compares, the others are signed compares. Note that there is no ra <> rb or ra = rb instruction. There are four input control lines for comparisons. They are determined by consulting the table below. "S" is the sign of the resulting sum of rb-ra. B[msb] is the sign bit of rb. The comparisons are chosen by specifying the sense of the result for each combination of A[msb] and B[msb] and by providing the carry in. A sense of "1" implies the value in the table (e.g., "S") is complemented.

```
Comparison function encoding:
        msb: must be 1 (for subtract)
             sense of result if A[msb],B[msb] = {1,1}
             sense of result if A[msb],B[msb] = {0,1} or {1,0}
             sense of result if A[msb],B[msb] = {0,0}
        lsb: carry in
```

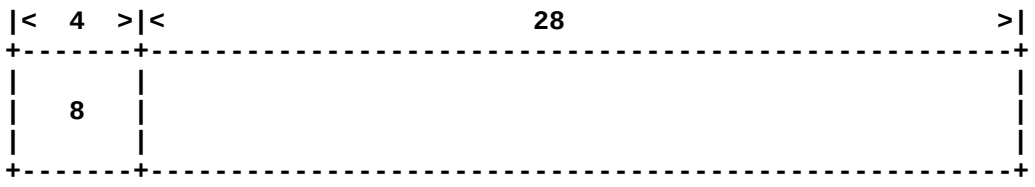
			Function is A cond B							
A[msb]	B[msb]	<u	<=u	>u	>=u	<	<=	>	>=
1	1	~S	~S	S	S	~S	~S	S	S
1	0	B31	B31	~B31	~B31	~B31	~B31	B31	B31
0	1	B31	B31	~B31	~B31	~B31	~B31	B31	B31
0	0	~S	~S	S	S	~S	~S	S	S
Cin		0	1	1	0	0	1	1	0

2.3.9 Undefined Operation (Reserved for CPU Store Byte)

TASM Format

no TASM format

Memory Format



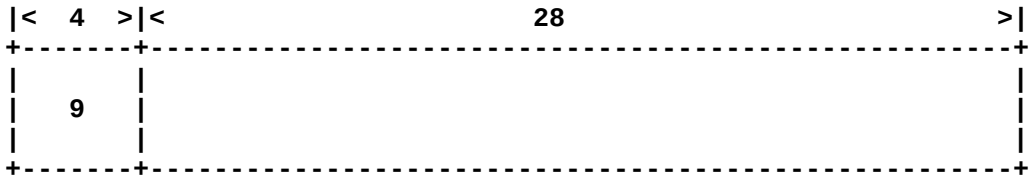
This instruction causes an unpredictable operation. It will not cause an illegal instruction opcode trap, since there is no such trap. However, it may cause a privilege violation, write a register, or write a memory location, but its operation is unknown and implementation dependent. This instruction should never be generated.

2.3.10 Test Operation (Formerly CPU Load Byte)

TASM Format

? TASM format

Memory Format



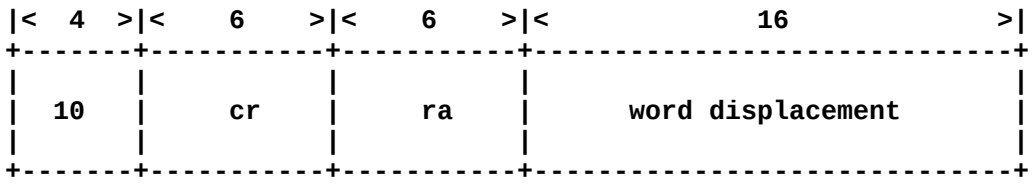
This instruction has the same effect as a Coprocessor ALU instruction. The instruction is placed on the address pins during the MEM pipestage. This is instruction is used is test sequences for the instruction buffer, since it can be set to the complement of a Coprocessor ALU instruction, and both these instructions make themselves visible at the pins.

2.3.11 Coprocessor Store

TASM Format

```
(disp[ra]) := cr;
```

Memory Format



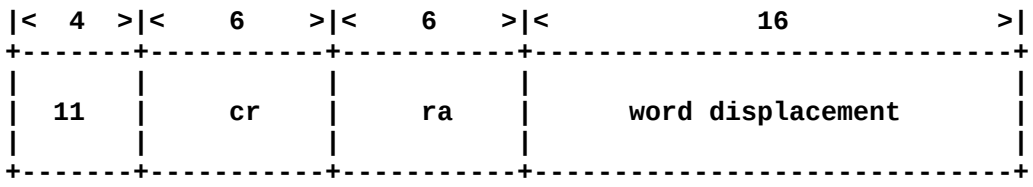
An address is computed by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding register ra. The three low-order bits of the address are ignored (i.e., assumed zero) and register cr of the coprocessor is stored into the 64-bit doubleword at that address. The register address "cr" plus the opcode are transferred to the coprocessors in the ALU pipestage.

2.3.12 Coprocessor Load

TASM Format

```
cr := (disp[ra]);
```

Memory Format



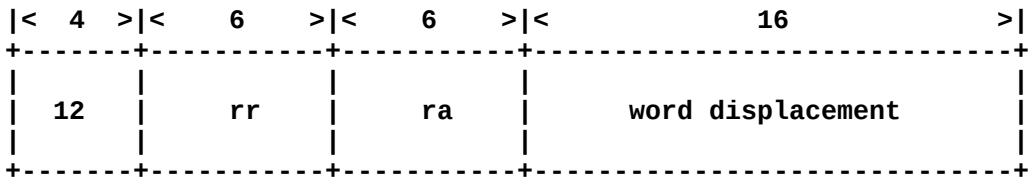
An address is computed by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding register ra. The three low-order bits of the address are ignored (i.e., assumed zero) and the 64-bit doubleword at that address is loaded into register cr of a coprocessor. The CPU sends the register address "cr" along with the opcode to the coprocessors in the ALU pipestage.

2.3.13 CPU Store

TASM Format

```
(disp[ra]) := rr;
```

Memory Format



An address is computed by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding register ra. The two low-order bits of the address are ignored (i.e., assumed zero) and register rr is stored into the 32-bit word at that address. Stores probe the external cache in the MEM pipestage, and send out data to the cache in WB. If a hit has been detected during the probe, the CCU enables writing from the data bus into the RAMs in the second half of the WB pipstage. Interactions with other instructions will be discussed in the timing section.

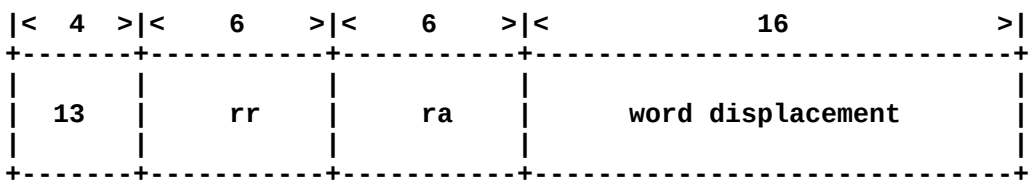
```
Example:        -3[r4] := r5;
```

2.3.14 CPU Load

TASM Format

```
rr := (disp[ra]);
```

Memory Format



An address is computed by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding register ra. The two low-order bits of the address are ignored (i.e., assumed zero) and the 32-bit word at that address is loaded into register rr. *Ra and rr should not be the same or else this instruction is not restartable in the presence of page faults and other interrupts.*

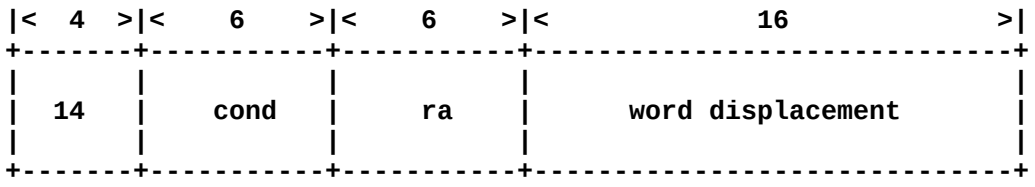
```
Example:        r3 := 1[r4];
```

2.3.15 Conditional Jump

TASM Format

```
if ra cond goto disp
```

Memory Format



An address is computed by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding the pc (the address of the conditional jump instruction). This address is loaded into pc if register ra meets the condition specified by the cond field (otherwise the pc increments normally). This results in a conditional transfer of control to the instruction at the computed address, following execution of the next instruction in line. Note that this instruction requires implicit addressing of the pc (the only such case). It is also the only instruction with three sources: ra, pc, and displacement. The instruction after the branch is always executed.

The table below details the "cond" values for the conditional jump instruction. Following this table there is more information of an implementation nature that details how the conditional jump gets decoded by the MultiTitan.

```
Conditional Jump Values
(The two highest order bits of the cond field are ignored
Note that all comparisons are made against "zero".)
value   utasm
--------------------------------
0       always    goto disp
1       never   goto disp
5       if ra <  goto disp
6       if ra =  goto disp
7       if ra <> goto disp
3       if ra >  goto disp
2       if ra <= goto disp
4       if ra >= goto disp
9       if ra %  goto disp      (if ra odd goto disp)
8       if ra &  goto disp      (if ra even goto disp)
```

The condition is the logical NOR of up to three selected bits: the lsb, the sign bit, and a bit which is true if the word is greater than zero. If the sense bit is on, then the condition is true if the logical OR of the selected bits is one. For example, equality to zero can be tested for by selecting >0 and <0.

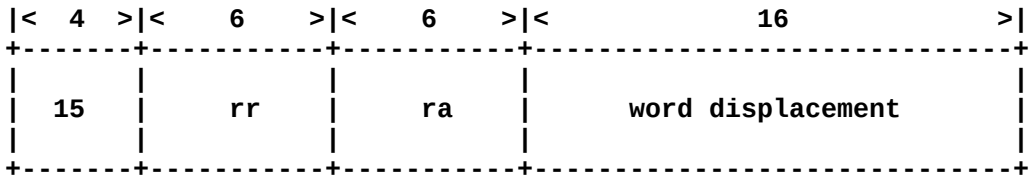
```
COND(ra) is:
        msb: unused
             unused
             select lsb
             select <0
             select >0
        lsb: sense of OR
Example:        if r4 <> 0 goto someLabel;
```

2.3.16 Add Immediate (or Load Address, or ..)

TASM Format

```
                                rr := ra + disp;
                          or    goto disp[ra];
```

Memory Format



An address is computed by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding register ra. The result is stored in register rr. Note that assignments to the PC take effect after executing the next instruction. Add immediate can be used to synthesize many other instructions, but since the displacement is shifted by two bits it can only add or subtract multiples of four.

```
Examples:
    r33:= pc + 2;     /* subroutine call, part 1 */
    pc := pc - 1025;  /* subroutine call, part 2 */
    pcq:= r27;        /* set first element in oldPC queue */
```

3. Exception Architecture

As well as having a simplified instruction set, MultiTitan also has a simplified exception architecture. For example, interrupts, page faults, coprocessor traps, and bus error are all signalled by pulling down a common interrupt line in the cycle *after* the exception. But before individual exceptions and trap handling is described in detail, we must first examine the pipeline and its timing.

3.1 Pipeline Timing

The MultiTitan CPU has a four stage pipeline. The stages are instruction fetch (IF), compute (ALU), memory access (MEM), and result write back (WB). A description of the actions taken in each pipestage is given in Figure 3-1. Each pipestage is broken into four clock phases. One and only one of the clock phases is high at all times, subject to small amounts of skew which may result in zero or two clock phases being high for a few nanoseconds. In the following discussions, *store instruction* will be used generically to refer to CPU store, coprocessor store, and CPU->coprocessor transfer, which all use the data and/or address busses in the WB stage. *Load instruction* will be used to refer to CPU load, coprocessor load, coprocessor->CPU transfer, and coprocessor ALU instructions, which all use the data and/or address busses in the MEM pipestage.

```
Pipestage       Function
-----------------------------------------
IF  p1       PC->IBuf decoders
    p2       IBuf decoders->sense amp outputs
    p3       sense amp outputs->register file decoders
    p4       register file decoders->sense amp outputs

ALU p1       operand sources->functional units
    p2       compute
    p3       compute
    p4       functional units -> result bus

MEM p1       if load or store, memory address computed
                    above->memory, otherwise idle
    p2       cache access
    p3       cache access
    p4       cache access

WB  p1       memory data->functional units if load
    p2       write register file if load, read if store
    p3       memory data->memory if store
    p4

On every pipestage:
    p1       precharge register file, and IBuf bit lines;
                    drive address and store data off chip
    p2
    p3       precharge register file bit lines and R bus
    p4       precharge M bus
```

Figure 3-1: Pipeline Phase-by-Phase Timing

In the absence of exceptional conditions, one instruction is issued every cycle (40 nsec). All instructions commit in their WB pipestage; this implies if an instruction causes an interrupt it can only do so in WB.

Load Interlock — If a CPU register is written by a load instruction, and used as a source in the next instruction, one additional cycle is lost.

Store Interlock — If the instruction following a store class instruction is a load class or store class instruction, one cycle is lost.

Coprocessor->CPU Transfer Interlock — If a Coprocessor->CPU transfer instruction follows a coprocessor load or store, one additional cycle is lost.

CPU->Coprocessor Transfer Interlock — If a Coprocessor->CPU transfer instruction attempts to issue two cycles after a CPU->Coprocessor transfer, one additional cycle is lost. Note that if a CPU->Coprocessor transfer is followed immediately by a Coprocessor->CPU transfer, a store interlock will occur on the first attempted issue of the Coprocessor->CPU transfer, and then the CPU->Coprocessor transfer interlock will occur, increasing the spacing between the two transfers to three.

Instruction Buffer Miss — Any instruction fetch which misses in the instruction buffer will incur at least a two cycle delay, assuming a hit in the external instruction cache. An instruction buffer miss that occurs while a store is in its MEM stage will be delayed one additional cycle. If the missing instruction is on word 3 of the line, or if there is a taken branch in the ALU pipestage, one additional delay cycle will occur at the end of an instruction buffer miss.

External Cache Miss — A load, store, or instruction buffer miss which misses in the external cache results in a delay of 14 cycles. If a load or store instruction immediately follows a load or store instruction that misses, one extra delay cycle will occur.

Coprocessor Stall — Wait for completion of the coprocessor can cost up to 48 cycles.

Trap or Interrupt — Any trap condition costs a three cycle delay until the instruction at the location specified by the upper 28 bits of PSW is executed. This will be followed by a further delay because the I-buffer will miss (since it is cleared on all context switches).

3.2 Program Status Words

Although page faults, coprocessor traps, and bus errors are all signalled via the same interrupt pin, these conditions are differentiated by program status words (PSW's) in each of the CPU, CCU, and FPU. Figure 3-2 gives the format of the CPU PSW. It is read only and reading it is not privileged.

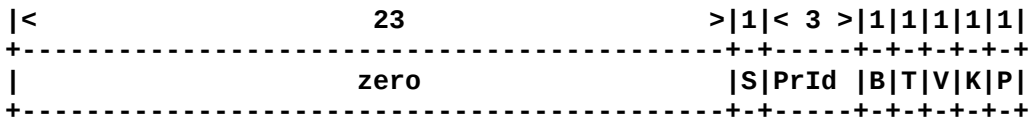


Figure 3-2: CPU PSW format

The B bit is on if the processor is being booted. It is generated internally from the CPU by delaying the Reset input. The PrId field is the hardware processor id. It is directly connected to three external pins and can range from 0 to 7. During boot sequences the upper 28 bits of the PSW (the other bits are zero) is used as a base address; the PrId field allows different processors to execute different code. The PrID field is placed in the PSW so that different processors will execute different cache lines. The S bit is on if the processor is being booted and is requested to save its state before booting. The S bit is a chip input, and is driven from the clock/scan board.

The T bit is '1' if a software trap occurred when the last interrupt was taken. The V bit is '1' if an instruction having an arithmetic overflow trap was in its WB pipestage when the last interrupt was taken. The K bit is '1' if the instruction in the IF pipestage is in kernel mode. In Kernel mode all interrupts are disabled and privileged instructions are enabled. The P bit is '1' if a hard parity error occurred in the cycle *before* the interrupt.

Note that since interrupts are not encoded or prioritized, other interrupts could have occurred at the same time (e.g.,

TLB fault, FP divide by zero, etc.). In order to determine whether other exceptions occurred, the FPU and CCU PSWs must be examined. The encoding of the CCU and FPU PSWs are given in the CCU and FPU architecture documents, respectively.

3.3 Pipeline Advancement

There are seven possible ways in which the pipeline advances from one cycle to the next. Figure 3-3 lists them in increasing order of priority. There are also eight control lines which determine pipeline advancement either directly or indirectly: LoadIF (internal to CPU), LoadALU, LoadMEM, LoadWB, AllowInterrupt, Interrupt, Reset, and Kernel.

```
1) Normal pipeline advance:
        WB := MEM;
        MEM := ALU;
        ALU := IF;
        IF := PC + 4;

2) Branch taken pipeline advance:
        WB := MEM;
        MEM := ALU;
        ALU := IF;
        IF := Branch target;

3) Interlock (LoadIF deasserted; LoadALU, LoadMEM, and LoadWB are not)
        WB := MEM;
        MEM := ALU;
        ALU := NIL;             delay slot injected here
        IF := Recirculate;

4) IBuf miss (LoadALU deasserted; LoadMEM and LoadWB are not):
        WB := MEM;
        MEM := IBM refill;      a load class instr
        ALU := Recirculate;     ALU does not advance
        IF := Recirculate;

5) Memory Reference Retire (LoadMem deasserted, LoadWB is not):
        WB := NIL;
        MEM := Recirculate;
        ALU := Recirculate;
        IF := Recirculate;

6) LoadWB deasserted (i.e., "stall"):
        WB := Recirculate;
        MEM := Recirculate;
        ALU := Recirculate;
        IF := Recirculate;

7) Interrupt: (Interrupt and AllowInt asserted, or Reset asserted)
        WB := NIL;
        MEM := NIL;
        ALU := NIL;
        IF := PC = PSW[31..4];

8) Uninterruptible Stall (LoadWB and AllowInterrupts deasserted):
        WB := Recirculate;
        MEM := Recirculate;
        ALU := Recirculate;
        IF := Recirculate;
```

Figure 3-3: Pipeline Advancement

LoadWB, LoadMEM, LoadALU, and LoadIF control whether each of the four pipestages stalls or advances. If the

advance signal for a later pipestage is deasserted, the advance signals for all previous pipestages must be stalled as well. This is required so that instructions don't overtake (i.e., crash into each other) in the pipeline. For example, if LoadMEM is deasserted, LoadALU and LoadIF must be deasserted as well. (Since the LoadIF signal only exists inside the CPU, it deasserts LoadIF internally when it sees LoadALU deasserted.)

If all signals have their load signal asserted, there is no interrupt, and the instruction in ALU is not a taken branch, all pipestages advance and the new IFpc is IFpc+4 (case 1). If an unconditional branch or a taken conditional branch is exiting its ALU pipestage, all signals have their load signal asserted, and there is no interrupt, then the branch target specified by the branch in ALU is the next IFpc (case 2). An interesting variation on this appears in the case of interlocks (case 3). In an interlock, LoadIF is deasserted, but all the other pipeline advance signals are asserted and there is no interrupt. If a taken branch instruction is in ALU during an interlock, it must advance since LoadALU and LoadMEM are asserted. However, the branch target cannot be immediately loaded into the IF pipestage since the interlock mandates that the instruction in IF be held. Thus interlocks are not allowed if a taken branch is in ALU. The only way to generate an interlock in IF concurrent with a taken branch in ALU is with a CPU->Coprocessor transfer followed by a taken branch with a Coprocessor->CPU transfer in the branch delay slot. This code sequence should never be generated or else unpredictable operation may result.

In the IBuf miss sequence (case 4) the ALU pipestage must not advance into MEM and thereby use the address pins. The IBuf miss sequence usually occurs for two cycles, but may require an additional cycle at the start of the sequence if a store class instruction is in Mem when the IBuf misses. During an instruction buffer miss, a total of two load-class pseudo instructions are inserted into the MEM pipestage in successive non-stall cycles. The first loads the requested doubleword into the cache line and the second loads the non-requested doubleword. The additional cycle delay that occurs when the instruction buffer misses with a store in the MEM pipestage is just a special case of the store interlock. An additional cycle is required at the end of the sequence if the next instruction is not on the same cache line. This is determined (slightly pessimistically) in the miss cycle by checking if the miss instruction is at word 3 of a cache line or if a taken branch is in the ALU pipestage. (In reality, the branch could be to the same cache line, and no extra cycle would be required.) The next instruction must be on the same line as the miss instruction so that the non-requested doubleword of the buffer line can be written into the buffer.

During stalls LoadWB and all earlier Load signals are deasserted (case 6). This results in all pipestages being stalled. If LoadMem is deasserted during an external cache miss but LoadWB is not, this signifies the requested data is on the data bus, and the instruction in WB should be retired (case 5). However, the following instructions may not advance until the memory operation completes.

If Interrupt and AllowInt (allow interrupt) are asserted, an interrupt is taken (case 7). AllowInt (and all the pipeline Load signals) may be deasserted during phase 3 if the CCU is in the process of a memory transaction. Thus AllowInt disables the effects of the Interrupt signal (case 8). To prevent exceptions from being handled, interrupts are ANDed with AllowInt. Inside the CPU the External Reset signal is OR'ed with the result of the AND'ing of AllowInt and Interrupt. Thus, reset has the same effect as an enabled interrupt, except that reset should be asserted at least 7 cycles so that all pipeline state can be reset. Although interrupt receivers must check AllowInt to tell if an interrupt is valid, interrupt sources must check the Kernel signal before generating interrupts. Interrupts should never be asserted in kernel mode, since if it is taken the return addresses will not have been saved in the PCQ.

Figure 3-4 gives the operation of each pipestage given the values of LoadMem, Interrupt, LoadWB, IBuf Miss, and Interlock. Int is actually "Reset or (AllowInt and Interrupt)" in the table.

Control	Int	LoadWB	LoadMEM	LoadALU	LoadIF	
	1	X	X	X	X	nop
WB	0	1	0	X	X	nop
	0	0	0	X	X	WB
	0	1	1	X	X	MEM
	1	X	X	X	X	nop
MEM	0	X	1	0	X	nop
	0	X	0	0	X	MEM
	0	X	1	1	X	ALU
	1	X	X	X	X	nop
ALU	0	X	X	1	0	nop
	0	X	X	0	0	ALU
	0	X	X	1	1	IF
	1	X	X	X	X	nop
IF	0	X	X	X	0	IF
	0	X	X	X	1	next PC

Figure 3-4: Operation of Each Pipestage

3.4 Interrupts

Upon any of a set of special circumstances, the processor interrupts the normal sequence of instruction execution, sets the program status word to kernel mode (interrupts disabled, privileged instructions enabled), disables address translation, and forces the upper 28 bits of the PSW as the new PC. In addition it sets bits in the program status word indicating if a trap instruction or arithmetic overflow was one cause of the interrupt (there may be more than one cause). If an interrupt occurs in kernel mode, the processor follows these same steps. However, since in kernel mode the PCQ does not advance, no record of the trap PC will be saved. Thus, all interrupt generators must check the Kernel bit and only generate interrupts if the Kernel bit is off.

3.4.1 Determining Instructions in Progress

The PCQ records the PC associated with each pipestage. The PCQ is read from WBpc and written to IFpc. When the PCQ is read, WBpc is output and the PCQ shifts by one. Similarly, when IFpc is written the PCQ shifts by one to make room for the new entry. Whenever a nop (i.e., "bubble") is inserted into the pipe, the nil instruction has as its PC the PC of the next valid instruction in the pipeline. Nil instructions can be inserted between WB and MEM during cache misses, between MEM and ALU during instruction buffer misses, and between ALU and IF during interlocks (see Figure 3-5). In fact, in some circumstances there may be as few as two distinct PC’s in all four PCQ registers: in other words, only two valid instructions exist in the pipeline. For example, if an instruction buffer miss occurs two successive nil instructions will be inserted in the MEM pipestage. If the instruction buffer miss faults in the TLB, this will be detected when the first pseudo-load instruction for the I-buffer reload is in its WB pipestage. At this point the instruction in IF is the faulting instruction, and the PC’s in ALUpc, MEMpc, and WBpc are all the PC of the instruction in its ALU pipestage at the time that the instruction buffer miss occurred.

```
Code sequence

 51:  r1 := 0[r2];
 52:  r3 := r1 + r7;
 53:  pc := pc - 43;
 54:  r4 := 2[r3];

                time ->
PCQ Sequence    1       2       3       4       5       6

WBpc            -       -       -       51      52      52
MEMpc           -       -       51      52      52      53
ALUpc           -       51      52      52      53      54
pc in IF        51      52      52      53      54      10
```

Figure 3-5: Operation of the PC Queue during Load Interlock

3.4.2 Parity Errors and Interrupts

The MultiTitan chip set maintains parity on its 64-bit local data bus. This parity is odd byte parity. Checking of parity is controlled by the CheckParity signal, output by the CCU from its PSW. On all CPU load and Cop->CPU transfer instructions the CPU checks the parity of the word written into its register file (see Figure 1-4). If the parity is incorrect, and CheckParity is asserted, the CPU asserts interrupt in the following cycle (assuming AllowInt is asserted). Thus, the parity error is associated with the instruction before the instruction recorded in WBpc (unless a stall was in progress with AllowInt asserted at the time of the interrupt). Since the values of LoadWB at the time of the interrupt is not saved, the instuction causing the parity error can not be determined precisely, and is usually not even one of the instructions in the PCQ. But hard parity errors are not restartable in software anyway, so this impreciseness is of little consequence except to diagnostic programs.

The CPU places proper parity on the driven word of the data bus in CPU stores and CPU->Cop transfers. In order to drive the parity bits at the same time as the data bits, the parity bits must be available when the data is read out of the register file (i.e., the parity bits must be stored in the register file too.) Data is written into the register file from two sources: the local data bus, or a functional unit within the CPU (e.g., ALU, barrel shifter, and compare unit). The local data bus has parity; to provide proper parity in the case of results of CPU functional unit results written into the register file, parity is computed on the R bus during the ALU, shift or extract instruction's normally idle MEM stage. On each write of the register file the parity of the written word is checked. This checks incoming data in the case of a CPU load-class instruction, and also checks the parity generator in the case of a CPU ALU, shift, or extract instruction. Since parity is stored in the register file, parity errors in the register file as well as the external data busses and data cache can also be detected. However, since the CPU does not check the parity of operands read out of the register file, a parity error in the register file is only detected if the erroneous register is written back to main memory or is written into the data cache and loaded back into the register file.

Instructions fetched from the external instruction cache on an instruction buffer miss also have parity since they are fetched over the local data bus. This parity is not checked when data is written into the instruction buffer, but rather when each instruction is read from the instruction buffer. Thus parity errors originating in the instruction buffer are also detected, as well as errors originating in the external data busses and instruction cache. If a parity error occurs on a instruction fetch that hits in the instruction buffer, the hit is forced to be a miss, independent of CheckParity. Then the normal instruction buffer miss sequence fetches the instruction from the external instruction cache. If the parity error originated in the instruction buffer, either through a transient error or a hard error within a RAM cell, the normal IB miss refill bypass will result in correct parity when the instruction is fetched. If a parity error occurs

during the write of the requested word in an instruction buffer miss refill sequence, the error is assumed to be in the data busses or external instruction cache; if CheckParity is asserted the CPU asserts interrupt in the following cycle.

A parity error could also occur in two cases during an instruction buffer miss refill sequence on the instruction following the instruction causing a miss. If the following instruction is in the same doubleword as the missing instruction, it will be written into the instruction buffer at the same time as the missing instruction, and when it is read it will be read from the RAM cells instead of from the bypass path. A parity error in this case could originate either in the instruction buffer of in external circuits. If the following instruction is in the non-requested doubleword, when it is read it will read from the bypass path. A parity error in this case would be independent of the instruction buffer RAM array (i.e., it would probably originate externally). Since these two cases are not distinguished, the CPU just forces a miss on a parity error on the instruction following a miss. Then if the parity error occurs on the refetch of the instruction from off chip, the error will occur during the write of the requested doubleword and will cause an interrupt (assuming AllowInt is asserted). Thus a bit cell error in the instruction buffer, whether transient or due to defects, will simply cause additional instruction buffer misses but will not prevent execution of programs. Up to one cell in every byte of every instruction in the instruction buffer may fail and the chip will still work (albeit more slowly).

3.4.3 Returning from an interrupt

In general, the last two uncommitted "valid instructions" must be restarted when returning from an interrupt. If the last valid instruction is in WB, it should only be restarted in the presense of page faults since in other circumstances the instruction in WB will have already committed. Note that in order to be restarted, load instructions cannot have ra=rr since they will have already written rr.

Although there may be as few as two valid instructions in the pipeline, there may be only one restartable instruction in the pipeline after an interrupt. This occurs in the case of a Coprocessor->CPU transfer that follows immediately after a CPU->Coprocessor transfer. The Coprocessor->CPU transfer instruction will have two successive interlocks. At this point, the address of the CPU->Coprocessor instruction will be in WBpc and the address of the Coprocessor->CPU transfer instruction will be in IFpc, ALUpc, and MEMpc. If an I/O interrupt occurs in this situation, the CPU->Coprocessor instruction in WB will not be restarted, which leaves only one valid instruction to restart. In there is only one valid PC in the PCQ, then there can be no pending branches, and the instructions specified by IFpc and IFpc+4 should be restarted.

Once the PC’s to be restarted have been found, they must be placed back into MEMpc and WBpc, with the first instruction to be restarted placed in WBpc. PCQ may be written by specifying it as the rr field in CPU ALU or add immediate instructions. The write will take place in the MEM pipestage; the PCQ is read in the ALU pipestage. Thus writes to the PCQ will commit one cycle earlier than other instructions, and reads will occur one cycle later. (But since all interrupts must be off to execute these instructions, we are safe.)

To return from an interrupt, the values placed in the PCQ must be transferred to the PC. The transfer must be done by an add immediate instruction with PCQ or PCQE in the rb field and PC in the rr field. Reading the PCQE register reads the PCQ with the side effect of exiting kernel mode. The first transfer of addresses from the PCQ to the PC when exiting kernel mode must use the PCQE register address. The jump to the transferred address and the exit from kernel mode take place after a delay of one instruction, just as other conventional branches have a branch delay of one. Next, PCQ->PC must be executed in the branch delay slot, which restarts the instruction formerly in MEMpc but now in WBpc. The next instruction after this will be the first restarted instruction in user mode. The instruction buffer is automatically flushed on interrupts and return from kernel mode. Thus the restarted instructions will always miss in the instruction buffer. If one of the values restarted from the PC queue causes a translation

buffer fault, an interrupt will occur. However, if the TB fault was on the instruction access of the first restarted instruction, the PC queue will contain an undefined value in ALUpc (and hence also the same value in MEMpc and WBpc due to the instruction buffer miss). If after processing this exception, the values in ALUpc and IFpc were restarted, incorrect operation would result from the undefined address in ALUpc. Therefore for correct operation the first return PC must have a valid translation.

The Kernel signal (i.e., pin) is associated with the instruction in the IF pipestage. It is the responsibility of the coprocessors to track the Kernel bit through the remaining pipestages, just as they track Op and Reg. The ability of an instruction to perform a privileged instruction in a particular pipestage is dependent on the value of the Kernel bit associated with the instruction in that pipestage.

4. Interface Architecture

Figure 4-1 summarizes the CPU pin assignments. Pads for the two words in the data bus doubleword plus the address bus are three-way interleaved. Although an entire data bus doubleword may be read into the CPU, at most one data word is written from the CPU. Similarly, data pad drivers are only active during store instructions, and the address bus is not active when the data bus is active. Thus, since there are at most 8 pads between I/O power and ground pins, at most three of these will be active at the same time.

# of Pins	Type	Function
64	I/O	Data bus
8	I/O	Data byte parity
32	O	Address bus
1	O	Iaddress
1	I	address alternate doubleword (CCU -> CPU)
1	I	check parity enabled
1	O	user/kernel mode
1	I/O	NotInterrupt
1	I	AllowInterrupts
1	I/O	LoadIF
1	I/O	LoadALU
1	I/O	LoadMem
1	I/O	LoadWB
1	I	enable data bus drivers (from CCU)
6	O	Cr (or ca) register address for FPU and CCU
4	O	Instruction opcode for FPU and CCU
3	I	PrId
1	I	SaveState
4	O	phi1..4 output
1	I	power-up reset
1	I	Clock In
1	I/O	Spare
136		signal pads used of 144 available
plus:		
16		I/O Vdd
16		I/O GND
2		internal Vdd
2		internal GND

Figure 4-1: CPU Pinout

For more details on signal function and timing, please consult the MultiTitan Intra-Processor Bus Definition and Timing document.

Table of Contents

List of Figures

MultiTitan Floating Point Coprocessor

Norman P. Jouppi
Digital Equipment Corporation
Western Research Laboratory
100 Hamilton Avenue
Palo Alto, CA 94301

Version of 6 April 1988

1. Introduction

The MultiTitan Floating Point Unit (FPU) is a single chip high performance coprocessor implemented in CMOS. It has multiple pipelined functional units and support for vectors. It can retire one arithmetic operation and one load or store instruction per cycle. It embodies the RISC philosophy in that hardware resources are dedicated to decreasing the latency of frequently used operations as much as possible, while infrequently used operations must be synthesized from these high-performance primitives. MultiTitan has a machine cycle time of 40ns, composed of four equal 10ns clock phases.

This document assumes some familiarity with floating point arithmetic. In particular, knowledge of published floating point implementation techniques such as Booth multiplier recoding, Wallace trees, and division via reciprocal approximation is assumed.[1] Knowledge of Cray-1 or Cray X-MP machine organization is very helpful as well.[2]

1.1 Hardware Resources

There are four functional units in the FPU. Their characteristics are given in Figure 1-1. All floating point computations are performed in 64-bit double-precision G format. G format is the VAX 64-bit floating point format (except for byte order) with a sign bit, 11-bit exponent, and 52-bit fraction.

Unit	Operations Supported	Latency (in cycles)
Register File	2 reads, 1 write, and 1 other read or write per cycle	
	52 64-bit general purpose registers	
	Program Status Word (PSW)	
	Time-of-Day Clock (TOD)	
	Interval Timer	
	Floating-point constants 0, 1/2, 1	
Adder	Addition	3
	Subtraction	3
	Float (integer to floating)	3
	Trunc (floating to integer)	3
Multiplier	Multiplication	3
	Integer multiplication (returns lsw)	3
	Iteration step (2-A*B)	3
Reciprocal	16 bit reciprocal approximation	3

Figure 1-1: FPU Functional Units

The functional units are controlled by a scoreboard and instruction registers. Coprocessor load, store, and transfer instructions are transferred from the CPU over the coprocessor instruction bus. Coprocessor arithmetic (ALU) instructions are wider than the coprocessor instruction bus, so they are transferred over the address bus.

As a reference, the latency of various operations in the Cray X-MP (with a 9.5ns cycle time) are compared with the latency of the functional units in the FPU in Figure 1-2. However, due to the MultiTitan's 4 times slower vector element issue rate, lack of chaining, and less powerful memory subsystem the overall performance can be significantly less than implied by the latency ratios.

---

[1] Hwang, K., "Computer Arithmetic," John Wiley & Sons, New York, 1979.

[2] Siewiorek, D., Bell, G., and Newell, A., "Computer Structures: Principles and Examples, McGraw-Hill, New York, 1982, Chapter 44.

Operation	FPU Latency	X-MP Latency
Addition, Subtraction	120ns	57ns
Multiplication	120ns	66.5ns
Division (reciprocal approx.)	720ns	332.5ns

Figure 1-2: Latency in MultiTitan FPU and Cray X-MP Functional Units

1.2 Data Formats

Two data formats are supported in the FPU: 64-bit double-precision G-format floating point and 32-bit integers. However, the G format is the only data format directly supported by all functional units. Figure 1-3 shows the format for a double-precision floating point number. The *s* bit is the sign of the *fraction* field. The *exp* field is the 1024-biased exponent.

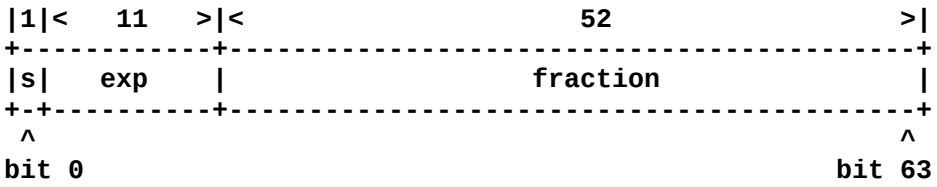


Figure 1-3: 64-Bit, Double-Precision Floating Point G Format

In the G floating point format, the actual fraction is always assumed to be normalized, but the normalizing bit is not present in the physical representation; there is a *hidden* bit. An exponent field of 0 does not signify the most negative exponent, but instead means that the number is assumed to be 0 if the sign and fraction fields are zero, and a reserved operand if the sign bit is 1.

32-bit integers are the same format as used by the CPU. They reside in the high-order half of coprocessor registers, and thus may be moved to and from the CPU with transfer instructions. (Transfer instructions are limited to 32 bits, and must transfer the high-order word since transfers of floating-point numbers to the CPU for comparison against zero must obtain the sign, exponent, and high-order fraction bits.) The Float and Integer Multiply instructions (the only instructions that operate on integers) ignore the low order bits. The Trunc and Integer Multiply instructions (the only instructions that produce integer results) place their results in the high order word.

Bytes and words on the 64-bit data bus are numbered in "Little Endian" order: the most significant half of a doubleword on the data bus corresponds to an odd address (i.e., 1 mod 2), and the sign bit is in byte 7 mod 8.

1.3 FPU Registers

The FPU has a general purpose register set of 52 registers, addressed <0..51> in the MultiTitan coprocessor register space. It also has a PSW (address 52), time-of-day clock (TOD) (address 53), and an interval timer (address 54) which can be used in all FPU instructions except FPU ALU. When these registers are read in FPU ALU instructions, they return the floating point constants 0, 1/2 and 1, respectively. Writes of these registers by FPU ALU instructions have no effect. Note that the FPU shares the coprocessor register address space with the CCU and other undefined coprocessors. The FPU determines that a coprocessor instruction is intended for it by examining the destination register specifier for coprocessor ALU, coprocessor load, and CPU->coprocessor transfers and the source register for coprocessor store and coprocessor->CPU transfers. Coprocessor ALU instructions with an FPU destination but source register specifiers outside the FPU address range will execute with undefined results. Note that CPU->coprocessor and coprocessor->CPU transfer instructions only transfer the high-order word of registers; this is true for the PSW, TOD, and timer as well as the GPR’s.

1.3.1 GPRs

These can be individually loaded, stored, and used in arithmetic operations. They can also be used in vector arithmetic operations. In contrast to other machines, the MultiTitan does not have vector register banks and vector register load/store instructions.

Vector register load/store instructions in a virtual memory environment share many problems with other multi-word memory references present in CISC machines. For example, the vector load can cross a page boundary, and the machine must save enough state to properly restart it. Although vector memory references can result in a significant performance improvement on machines with large memory bandwidth, the MultiTitan has more limited bandwidth than these machines. Also, in many applications the most important advantage of vector instructions is the ability to overlap floating point computations, memory references, and normal loop overhead. In the MultiTitan, this is still possible to a large extent without the use of vector memory references. Once a vector arithmetic operation is begun, the CPU is free to issue loads for future computations, stores for previous results, and loop overhead instructions.

Vector register banks, where registers are grouped into vectors of fixed length and operated on as a group, reduce the op code space required to represent instructions but also limit the flexibility of use of the individual registers. For example, in these static allocation schemes, the user can not select between 8 banks of 64 registers or 64 banks of 8 registers. In MultiTitan, the user can dynamically partition the 52 GPRs through software into any number of 1 to 16 element register groups.

1.3.2 PSW

The FPU program status word (PSW) is read only. It is addressed by coprocessor loads and stores as register 52 in the MultiTitan coprocessor register space. It is loaded every cycle while the CPU is in user mode with interrupt not asserted, and held during kernel mode or when interrupt is asserted. (Interrupt may be asserted on a previous phase 3 without forcing kernel mode on the next cycle if memory transactions are in progress (i.e., LoadMEM is deasserted).) Upon return to user mode it is again latched each cycle. Its format is given in Figure 1-4. The P bit is set if there was a privilege violation (interrupt is also asserted). The only cause of this is a user program that attempts to set the TOD clock or interval timer. This privilege violation trap can be useful when implementing virtual machines. The O bit is set if an arithmetic overflow (including zero divide) occurs during an arithmetic operation (interrupt is also asserted). Rov is the destination register specifier of the element whose result overflowed. If there is more than one arithmetic overflow in a vector ALU instruction, Rov is the destination register specifier of the first to overflow. After an arithmetic overflow all subsequent result elements are discarded (i.e., not written into the register file) until an interrupt is taken. If a computation underflows, the destination is set to zero and *interrupt is not asserted*. The PSW can only be read with a coprocessor store or an FPU->CPU transfer instruction. It is defined as the floating-point constant 0 if read in an FPU ALU instruction. The T bit is set if the interval timer becomes zero.

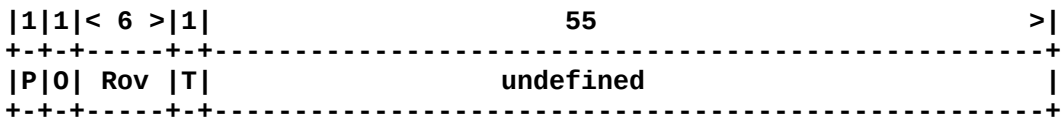


Figure 1-4: FPU Program Status Word

1.3.3 TOD Clock

The time-of-day clock is a 54-bit counter that increments every clock cycle, and is addressed as coprocessor register number 53 by coprocessor loads and stores. It can be read in user or kernel mode, but can only be written in kernel mode. Attempted writes in user mode cause a privilege violation interrupt. It is of sufficient length not to overflow

in 20 years of 40ns clock cycles. The TOD clock can only be read with a coprocessor store or an FPU->CPU transfer instruction, and written with a coprocessor load or a CPU->FPU transfer instruction. It is defined as the floating-point constant 1/2 if used in an FPU ALU instruction. When read with a coprocessor store instruction, the upper 10 bits are zero.

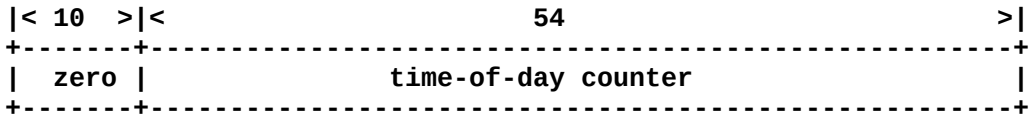


Figure 1-5: FPU Time-of-Day Counter

1.3.4 Interval Timer

The interval timer is a 54-bit counter that increments every clock cycle, and is addressed as coprocessor register number 54 by coprocessor loads and stores. The T bit in the PSW is set and an interrupt occurs when the interval timer increments to zero. The interval timer can be read in user or kernel mode, but can only be written in kernel mode. Attempted writes in user mode cause a privilege violation interrupt. It is of sufficient length to time 20 years of 40ns clock cycles. The interval timer can only be read with a coprocessor store or an FPU->CPU transfer instruction, and written with a coprocessor load or a CPU->FPU transfer instruction. It is defined as the floating-point constant 1 if used in an FPU ALU instruction. When read with a coprocessor store instruction, the upper 10 bits are zero.

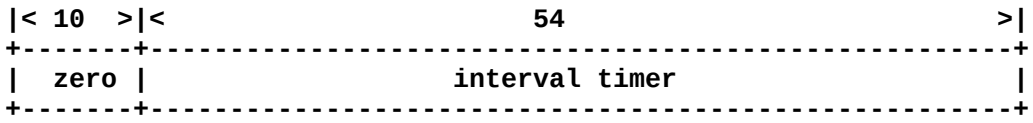


Figure 1-6: Interval Timer

1.3.5 Floating-Point Constants

As described in the previous sections, the floating-point constants 0, 1/2, and 1 are available in FPU ALU instructions as registers 52, 53, and 54 respectively. These constants are useful in synthesizing FPU register-to-register move instructions (Cx <- Cy + 0), since this operation is not present in the basic instruction set. These constants are also useful in providing the round operation by the addition of 1/2 followed by the trunc operation. If a constant is specified as the target of an operation, the operation has no effect but the normal issuing checks and interlocks apply. An interesting FPU NOP is provided by C52 <- C53 + C54, in that it can synchronize the CPU with the issuing of the last element of a previous vector. (The FPU NOP will stall until the last element has issued and left the FPU ALU instruction register. This is useful when only the last result of a long vector is desired, since all vector element results are not reserved immediately, but only as each element issues. Without an intervening FPU NOP between a long FPU vector ALU instruction and a store of the last element, the store would save the old value of the specified register, unless there was an intervening interrupt.)

2. Instruction Set

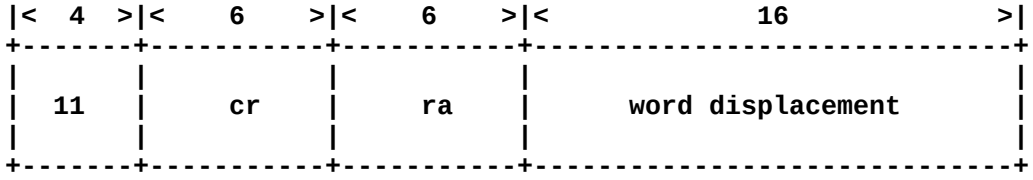
There are two classes of instructions: data transfer (loads/stores/CPU->FPU/FPU->CPU) and arithmetic operations. Coprocessor registers are designated by "c*x*" while CPU registers are designated by "r*x*".

2.1 Coprocessor Load

TASM Format

```
cr := (disp[ra]);
```

Memory Format



Description

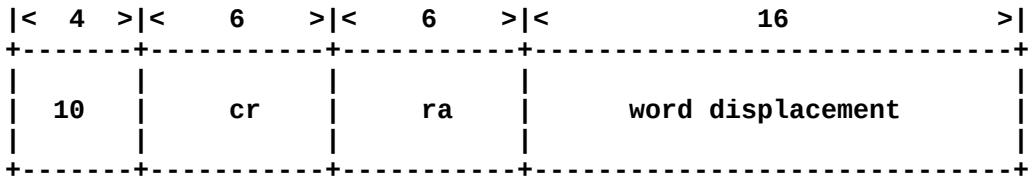
The CPU computes a byte address by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding register ra. The three low-order bits of the address are ignored (i.e., assumed zero) and the 64-bit aligned doubleword at that address is loaded into register cr of the FPU if cr is in the range <0..54>. The CPU sends the register address "cr" along with the opcode to the coprocessors in the ALU pipestage.

2.2 Coprocessor Store

TASM Format

```
(disp[ra]) := cr;
```

Memory Format



Description

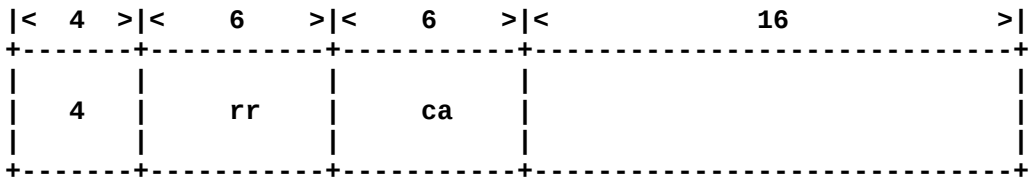
The CPU computes a byte address by left shifting the displacement field of the instruction by two bits, sign-extending it to 32 bits, and adding register ra. The three low-order bits of the address are ignored (i.e., assumed zero) and register cr of the FPU is stored in the 64-bit aligned doubleword at that address if cr is in the range <0..54>. If the coprocessor register specified by cr is not yet available due to a computation in progress, the FPU will deassert LoadWB until it can output the result. The CPU sends the register address "cr" along with the opcode to the coprocessors in the ALU pipestage.

2.3 CPU to Coprocessor Transfer

TASM Format

```
ca := rr;
```

Memory Format



Description

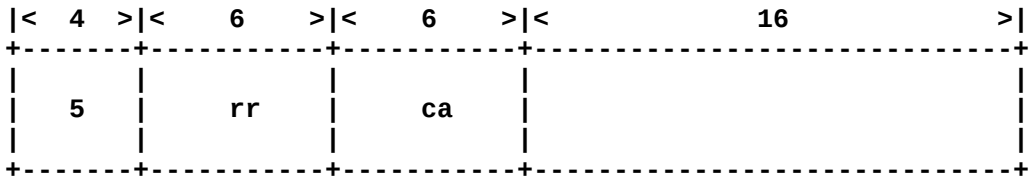
The CPU performs a store instruction, but the CCU does not enable the memory. The CPU outputs rr onto the high order data lines (i.e., word "1") during the WB pipestage. The FPU loads the high order half of register ca with the data put out by the CPU if the register address ca is in the range <0..54>. The contents of the low-order half of register ca are undefined after CPU->Coprocessor transfers. This instruction is useful for transferring operands to the FPU for integer multiplies. The register address ca plus the opcode are transferred to the coprocessors in the ALU pipestage. Note that the FPU pipeline must input data one pipestage later than during loads from memory, and only 32 bits are transferred.

2.4 Coprocessor to CPU Transfer

TASM Format

```
rr := ca;
```

Memory Format



Description

The CPU performs a load instruction, but the CCU does not enable the memory. If ca is in the range <0..54> the FPU outputs the high order half of register ca onto the high order data lines (i.e., word "1"). The CPU reads the data at the beginning of its WB pipestage. This instruction is useful for obtaining results from the FPU for integer multiplies. It is also used to transfer the most significant word of G-format numbers to the CPU for testing by conditional branches. If the coprocessor register specified by ca is not yet available due to a computation in progress, the FPU will deassert LoadWB until it can output the result. The register address ca plus the opcode are transferred to the coprocessors in the ALU pipestage. Note that the FPU pipeline must output data one pipestage earlier than during stores to memory, and only 32 bits are transferred.

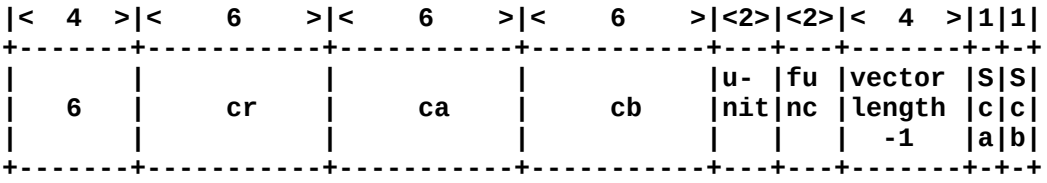
2.5 Coprocessor ALU

TASM Format

```
cr := ca (<function> {,<length> {,sca} {,scb}}) cb;

where <function> is zero, add, sub, trunc, float, fmul, imul or recip;
      <length> is in the range [0..15] and defaults to 0.
```

Memory Format



Description

Coprocessor ALU instructions are transferred from the CPU over the address bus lines at the beginning of the MEM pipestage. The FPU executes coprocessor ALU instructions whose destination is in the range <0..47>. In operations with only one source operand, ca and cb must both be set to the source. In operations with no source operands, ca and cb must be set to the destination register. All FPU ALU instructions are potentially vector instructions; scalar instructions have a vector length of 1. Vector instructions are issued by merely incrementing register fields in the instruction register and issuing the resulting instructions with the same mechanism used for scalar operations. The vector length field specifies the number of elements in the vector, from 1 to 16. The only means of specifying vector length is statically in the instruction itself; there is no dynamically loadable vector length register. After issuing the first instruction in the vector, the vector length field is checked. If it is zero, the instruction is cleared from the instruction register. If it is non-zero, the vector length field is decremented and the appropriate register specifiers are incremented. This instruction is then treated the same as any other instruction newly placed in the instruction register. If the Sca bit is set, register source field ca does not increment (i.e., it is a scalar). If the Scb bit is set, register source field cb does not increment. If both bits are set "vector := scalar op scalar" is performed. Scoreboard hardware already in place for scalar operations allows vector recursion to be performed. For example, one vector instruction could be equivalent to "DO I := 1 to 16 A[I] := A[I-1] + A[I-2]", although this instruction would take three times longer to execute than a non-recursive 16 element vector add instruction.

The unit field selects the functional unit for the computation:

```
Unit, 2 bits:
  0: reserved
  1: adder
  2: multiplier
  3: reciprocal approximation
```

The func field specifies the function requested and is specific to the functional unit selected:

```
Function, 2 bits:
Function is reserved if zero unit selected.
Function if adder unit selected:
  0: addition
  1: subtraction (cb-ca)
  2: float
  3: trunc
Function if multiplier unit selected:
  0: G-format multiplication
  1: integer multiplication
  2: iteration step (2-A*B)
  3: reserved
Function if reciprocal approximation unit selected:
  0: reciprocal approximation
  1-3: reserved
```

The operation of the func and unit fields are summarized below:

```
operation       unit    func
------------------------------
reserved         0       X
add              1       0
subtract         1       1
float            1       2
trunc            1       3
multiply         2       0
integer multiply 2       1
iteration step   2       2
reserved         2       3
reciprocal       3       0
reserved         3       1-3
```

2.6 Operations Synthesized in Software

A number of operations common in other machines are not directly supported in the FPU hardware. These must be synthesized in software using the previously described high-performance primitives.

2.6.1 F, D, and H format floating point

The adder and multiplier obtain their high performance in part through eliminating control logic and being almost exclusively combinatorial logic. Support for other formats would slow down the frequently used double precision G format operations. D format is antiquated, and hardware for H format operations is prohibitively expensive yet unnecessary for most applications. The single precision F format is the most common format besides G format, although there are important applications where it is not present (e.g., the portable C and Berkeley Pascal compilers for the VAX do not generate F format operations). The low latency of the G format operations eliminates the speed advantage enjoyed by F format on most machines. The other F format advantage of compact data storage can obtained by converting G format numbers to and from F format using shift and extract instructions in the CPU. Converts between F and G format were considered for minimal F format support in the FPU, but these operations introduced several complexities inherent in full support for both F and G formats, while yielding F format operations of lower performance than G format. For example, in order to load and store F format numbers without transferring them first to the CPU, single precision coprocessor load and store instructions would be required. However, no available opcodes existed in the CPU instruction set. The use of only double precision floating point format also has historical precedent in machines designed by Seymour Cray.

2.6.2 IEEE Standard Floating Point

There are several features of IEEE standard floating point which can significantly degrade the performance of frequently used operations. Among these are gradual underflow, complicated rounding modes, and good to the last bit multiplication and division. For example, the multiplier array would need to be almost twice as large in order to always correctly compute the least significant bit of the result. Moreover, there is a significant installed base without IEEE standard floating point arithmetic (e.g., VAX, 370, Cray).

2.6.3 Integer Division

This is supported through Float, G format division, and Trunc. (G format division itself consists of an initial reciprocal approximation ROM lookup followed by two Newton iterations.) This obtains about the same performance as integer division implemented by 32 single-cycle 1-bit restoring division steps.

2.6.4 Explicit Comparisons with Conditional Branches

Conditional jumps based on data in coprocessor registers can be efficiently performed given the available instructions. A FPU->CPU instruction can transfer integer results or the most significant word of a G format number to the CPU in one cycle. There it can be tested for less than zero, zero, etc. by CPU conditional branch instructions. To compare of two G format numbers, the numbers can be subtracted one from the other and the result tested for less than zero (e.g., less than), zero (e.g., equality, etc.) as above.

2.6.5 Square Root and Other Functions

ROM look up tables could provide an initial approximation for many functions besides division. These functions are relatively infrequent, and this extra support would not dramatically increase their performance.

3. Implementation

The FPU is implemented in Hudson's CMOS-2 process. This results in a die size of 9.0x10.0mm.The floorplan of the FPU is given in Figure 3-1.

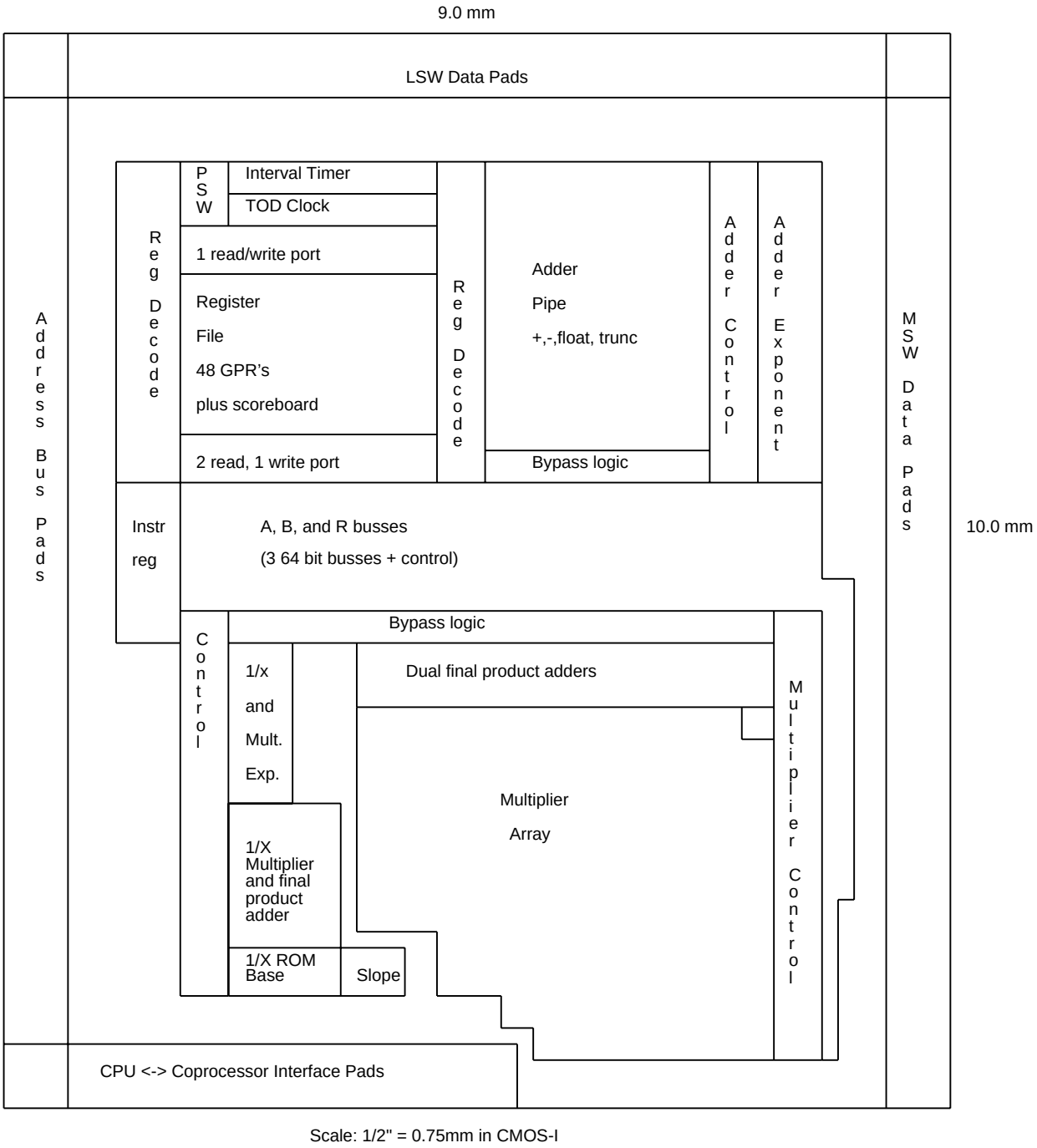


Figure 3-1: FPU Floorplan

Figure 3-2 gives an overview of the microarchitecture of the FPU. The FPU extends the CPU four stage pipeline to a total of seven. The additional pipestages are EX2 (execute stage two; execute stage one is WB), EX3, and PA

(write back ALU result; this is called put away to prevent confusion with WB). The FPU has four major data path blocks: the adder, multiplier, reciprocal approximation unit, and the register file. The register file has four ports: two are read for A and B ALU source operands, the ALU results are written on the R port, and loads/stores/tranfers read or write the memory (M) port. In addition, a time-of-day counter, an interval timer, and the FPU PSW are above the register file. There are seven instructions registers (IR's). IR[ALU] and IR[MEM] hold instructions in the ALU and MEM pipestages, respectively. IR[WB[LST]] holds load/store/transfer instructions in the WB pipestage. Similarly IR[MEM[ALU]] holds ALU instructions in the MEM pipestage, and IR[WB[ALU]] holds ALU instructions in the WB stage. IR[EX2], IR[EX3], and IR[PA] hold ALU instructions in the EX2, EX3, and PA pipestages. All instruction registers except IR[MEM[ALU]] and IR[WB[ALU]] consist of an opcode and one register specifier; IR[MEM[ALU]] and IR[WB[ALU]]contains an entire 32 bit FPU ALU instruction.

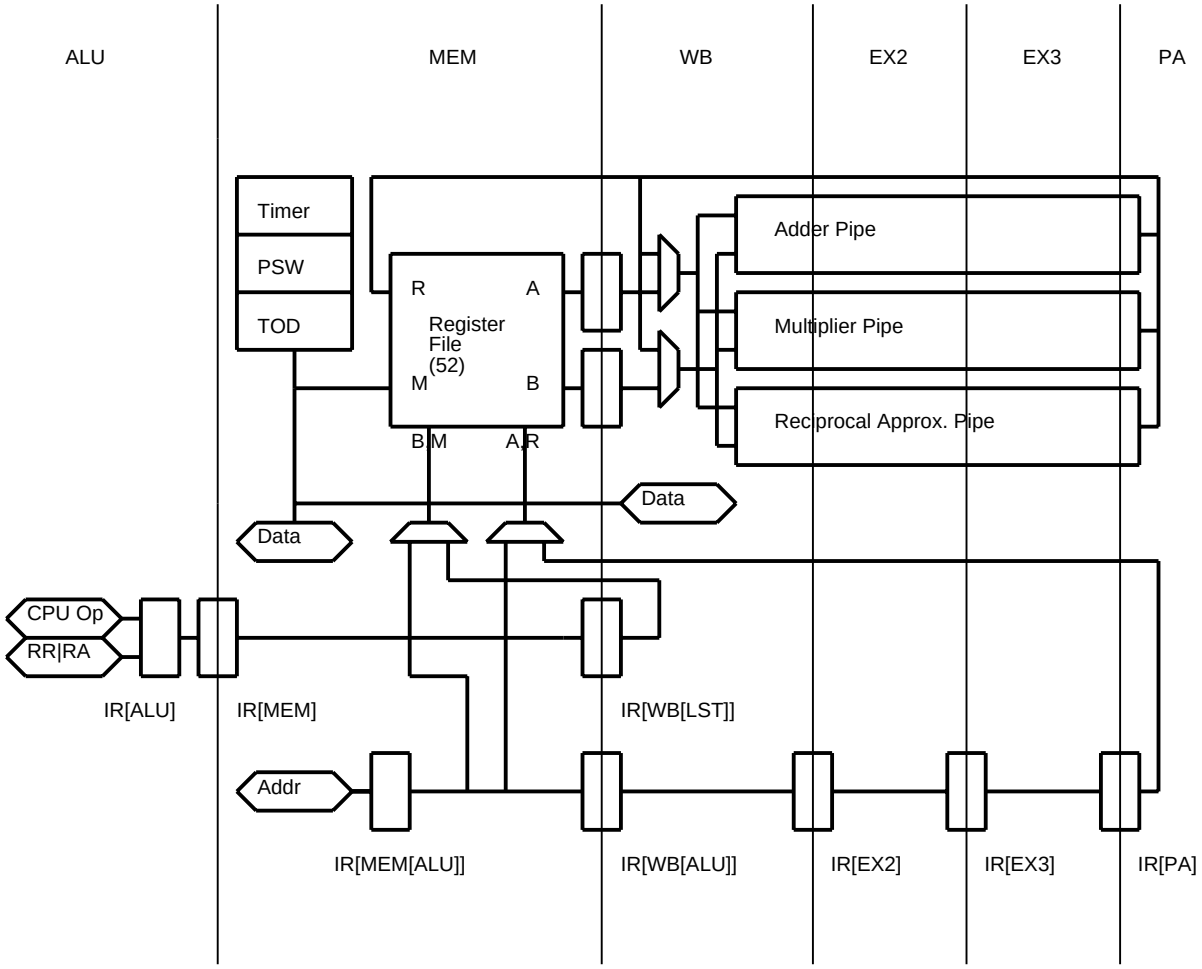


Figure 3-2: Microarchitecture of the FPU

3.1 Adder

The adder[3] performs G-format floating point addition, subtraction, trunc, and float with a three cycle latency. It requires about 15% of the total chip area.

[3]Silvio Turrini has implemented the adder. This section is contributed with his assistance.

3.1.1 Addition and Subtraction

The normal structure of floating point addition involves some of the following seven steps, depending on the values of the operands:

1. Calculation of the absolute value of the exponent difference.
2. Alignment of the fractions by shifting right the fraction of the smaller operand.
3. Addition or subtraction of the aligned fractions.
4. Two's complement the difference if negative.
5. Normalization of the sum or difference.
6. Rounding of the normalized sum or difference.
7. Renormalization if necessitated by rounding.

A number of different methods[4] have been used to reduce the latency of the basic addition and subtraction operation. The adder structure is given in Figure 3-3, and its timing is given in Figure 3-4.

First, the adder implements two major parallel datapaths for the cases where the alignment shift is greater than one or not. If the alignment shift is greater than one, then the normalization step will require at most one bit of shift. Also, large normalization shifts (i.e., greater than one bit) are only required in cases where the alignment shift is less than or equal to one. Thus, the three steps of align, add, and normalize can be reduced to two parallel paths of align and add ("align pipe") or add and normalize ("normalize pipe"). This effectively eliminates a cycle in each of the paths since a barrel shifter requires two clock phases to finish shifting, while the one bit shift may be implemented with either static or domino multiplexors in less than a clock phase. This partition also simplifies the control logic and the sign calculation in each of the cases. Note that this optimization does not increase the number of barrel shifters over that required for a one pipe implementation, but does require an extra adder (which is about half the height of a barrel shifter) and a multiplexor to select between the two pipes. Both major parallel paths finish in two clock cycles and then the path containing the correct result is selected for continuation into the rounding unit.

Second, when subtracting two numbers with equal exponents, the sign of the result is not known until completion of the actual fraction subtraction. To avoid having to perform a two's complement operation on a negative result, two fraction adder/subtractors are implemented so that the positive result is always calculated. This replaces the time required for a two's complement operation with the the time delay of a simple multiplexor.

Third, the rounded result and the renormalized result after rounding are calculated concurrently to further reduce latency.

Fourth, advanced high-speed circuit design techniques are used. For example, the count leading zeroes function (required before the large normalization step) is similar to the propagation of the carry signal in adder circuits; thus similar techniques may be applied to increase its speed. The count leading zero logic uses two levels of group skip circuitry and a pass transistor ripple circuit within bit groups to reduce its count time to two clock phases.

3.1.2 Float and Trunc

The basic hardware required to perform the trunc and float operations corresponds to the floating point adder's align and normalize units. A large saving in chip area is realized by performing these operations within the adder at the expense of increasing the number of cycles required to produce a result from one (trunc) and two (float) to three. During trunc or float the adder generates its own B source operand, which is added to the A source operand to effect

---

[4]Jud Leonard, unpublished communication, "Fast Addition", November 1985.

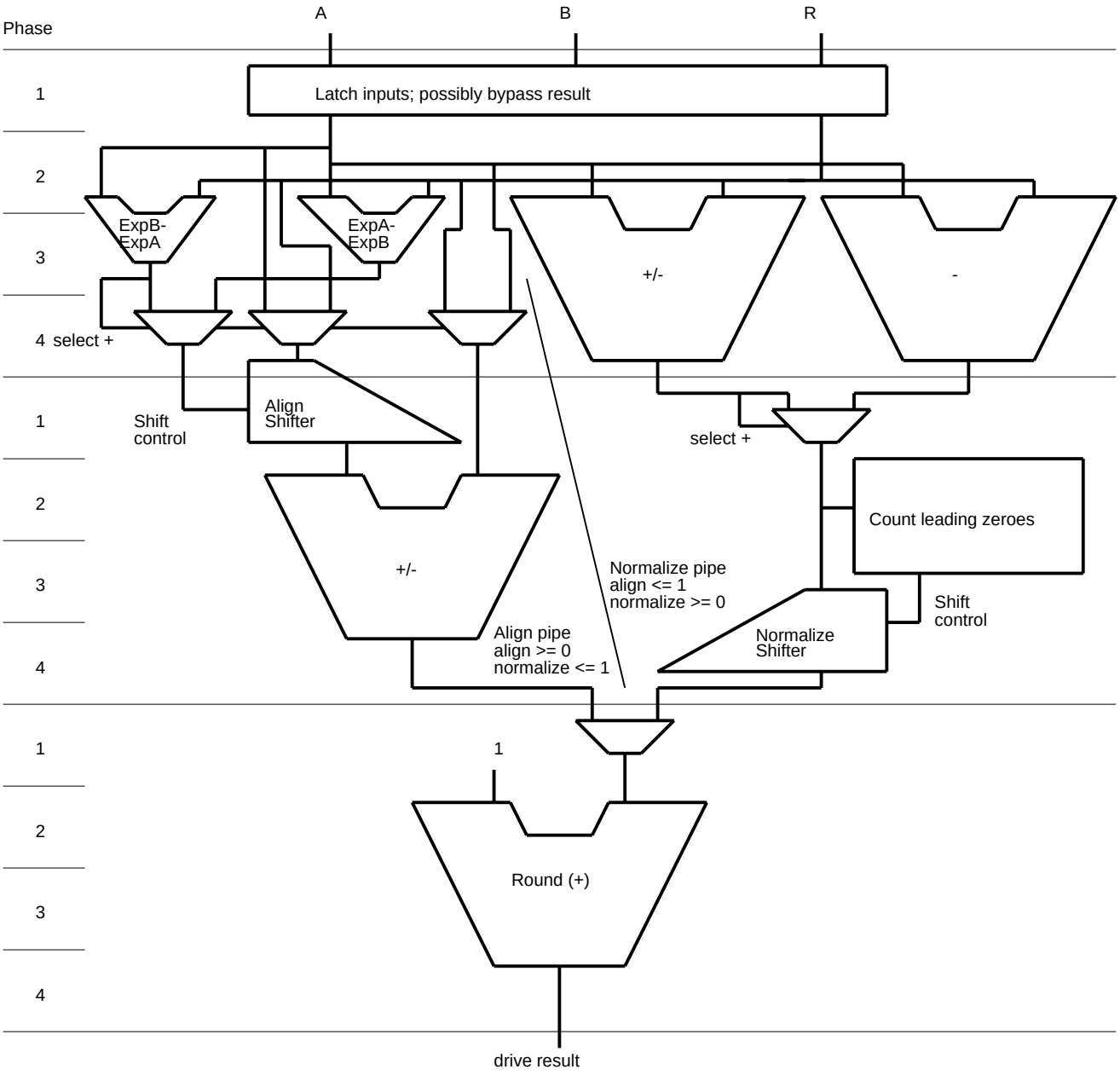


Figure 3-3: Timing vs. Structure of the Adder

its conversion. A small amount of extra hardware is also required to account for the difference between the two's complement integer representation and the sign-magnitude representation of the floating point fraction. Note that an integer number driven from the register file will reside on the high 32 bits of the bus.

3.1.2.1 Float

Converting an integer to a floating point number requires a normalization shift. Setting up the integer and its special operand as shown in Figure 3-5 will cause utilization of the correct parallel path to perform this conversion.

The integer is taken from the high-order half of the A source bus, placed directly into the low order 32 bits of the

cycle	phase	align pipe operation	normalize pipe operation
1	1	operands driven to adder	
	2	exponent comparison	set up add/sub control
	3	exponent comparison	addition
	4	operand alignment	addition
2	1	operand alignment	select positive result
	2	addition	count leading zeros
	3	addition	count leading zeros
	4	addition	normalize
3	1	mux align and normalize pipe output	
	2	round, calculate new exponent	
	3	round	
	4	round	
4	1	drive result	

Figure 3-4: Adder Phase by Phase Timing

	sign	biased exp.	fraction including hidden bit
integer (A source)	int sign	1077	<sign ext.>< integer >
special (B source)	int sign	1077	< 0 >

Figure 3-5: Hardwired Second Operand for Float

fraction, and sign extended to fill the remaining higher order bits. Both operands are given the same exponent value so that the result will be taken from the pipe with the normalization barrel shifter. A biased exponent of 1077 is the correct initial exponent value. The B-A and A-B adders generate both the integer and its two's complement, and the same hardware used in floating point addition selects the positive result.

3.1.2.2 Trunc

Converting a floating point number to an integer requires an alignment shift. Adding the value shown in Figure 3-6 will produce the desired direction of shift.

	sign	biased exp.	fraction including hidden bit
special	sign	1077	< 0 >

Figure 3-6: Hardwired Second Operand for Trunc

This value contains all zeros in its fraction, has a biased exponent of 1077, and has the same sign bit as the number to be converted. If the floating point number is negative then it is subtracted from this value to obtain the necessary two's complement representation, otherwise it is added. Extra hardware is provided to check for integer overflow, since a positive floating point number with an unbiased exponent larger than 31 and a negative floating point number less than -2^{32} can not be converted to a two's complement integer representation. The result is placed onto the high order half of the result bus.

3.2 Multiplier

The multiplier[5] is the largest functional unit, using about 30% of the total chip area. It performs a complete G-format multiplication, an integer multiplication returning the least significant word of the integer product, or an

[5]Alan Eustace has implemented the multiplier. This section is contributed with his assistance.

iteration step (2-A*B) in 3 clock cycles. The multiplier achieves this performance by a combination of different high speed techniques.

First, all summands are generated in the same cycle. This is in contrast with iterative approaches where one or several summands are generated each cycle. This allows the addition of all summands (shifted appropriately) to begin in parallel.

Second, a modified two bit Booth algorithm is used which reduces the number of summands from 53 to 27. A higher order (e.g., three bit) multiplier recoding was not pursued because this would entail the generation and distribution of 3x the multiplicand, which would itself require a clock cycle and an extra metal track (although one metal track might be saved due to fewer summands and offset this). The 2 bit Booth recoding and its distribution to summand generation logic is performed in one clock phase by domino logic.

Third, a new method of combining summands[6] is used that has time equal to that required in a Wallace tree but with wiring approaching a binary tree in complexity. An overview of the method follows. Because interconnect is not free, both in terms of area and delay, the new method attempts to reduce the summands more rapidly than the 3/2 reduction factor found in Wallace trees (see Figure 3-7). For example, an extreme application of this approach would be to use carry lookahead adders to sum each pair of summands; four ranks would reduce the 27 Booth output terms to two, at the cost of making each rank take a full carry lookahead add time -- probably five to six times the delay of the single add cell in each rank of the Wallace tree. However, the fact that each rank has only half as many outputs as inputs makes a dramatic improvement in the vertical interconnect problem. A moderate approach is to add pairs of partial product terms in small "chunks" of N bits, so that each chunk takes 2*N + 1 bits in, (1 for the carry signal into the chunk) and produces N + 1 bits out. The full adder cell in the Wallace tree is the degenerate case of this arrangement with N=1. Figure 3-8 shows an example of this method of summand reduction for N=5. By staggering a group of N chunks, we can arrange that the carries in and out of chunks in the group do not line up, so N rows collapse 2*N+1 terms to N+1 terms. With N=4, for example, we can reduce the 27 Booth partial product terms to 3 in four ranks, then use a single rank of carry-save adder cells, as in the Wallace tree, to reduce these to 2 for the carry lookahead adder. This compares to seven ranks (2 * ((3/2) ^7) > 34) in the Wallace tree, so the propagation delay through the N=4 cell and interconnect would have to be less than 1.4 times the delay of a loaded adder cell to win on performance. This is achieved by the use of an optimized Manchester carry chain within the chunk. Finally, this approach is particularly attractive in CMOS domino logic in which xor gates are relatively expensive, since both the new approach and the Wallace tree require only one 3-input xor per rank. A chunk size of N=5 is implemented in the FPU multiplier array. A chunk size of N=5 is advantageous because it is a prime number, and partial summands that start every four bits do not line up or overlap the chunk carries. Carries out of a chunk appear every 5 bits in a rank, and are treated as 1/5 of a partial summand.

Fourth, the array is "double pumped" to further reduce its area: half the summands are generated on phase 2 and the remainder are generated by the same circuitry on phase 4. This allows the multiplier to start a new multiplication every cycle while cutting the size of the array in half. In other words, the sum of the odd numbered summands is one half clock cycle ahead of the sum of the even numbered summands. This delays the summation of the even summands by half a clock cycle, but this additional latency is made up for by the reduction in wire delay due to a smaller array. Instead of reducing the odd summands and even summands each to a single number, the reduction of the even summands is stopped one reduction earlier than the reduction of the odd summands. At this point the even summands have been reduced to 1 + 4/5 partial sums; these are combined with the 1 + 1/5 partial sum from the odd summands by a carry save adder into two partial sums. These two partial sums are then input into fast 64-bit carry

[6]Jud Leonard, unpublished communication, "Fast Multiplication", November 1985.

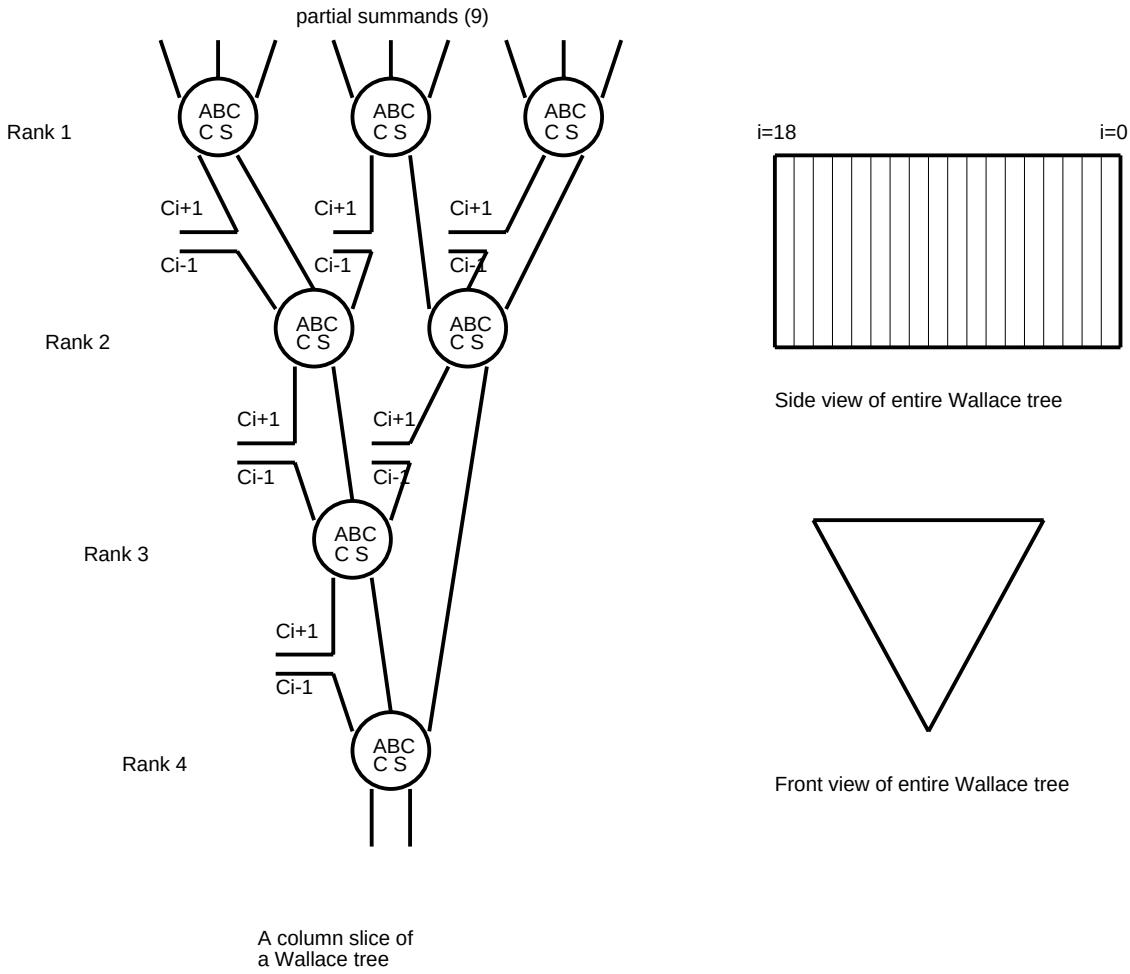


Figure 3-7: Wallace Tree for Reducing 9 18-bit Partial Products

skip adders. The reduction in area obtained by this and the next technique are shown in Figure 3-9.

Fifth, the low order 42 bits of the multiplier array (out of a total of 106) are truncated to further reduce its size by about one half. As an extra benefit from reducing the size of the array, the size of the final carry skip adder is reduced to 64 bits, allowing it to operate significantly faster than the 106 bit adder required in a full array. The maximum error generated by this approach occurs when the truncated portion of the Booth multiplier array contains all ones and is equal to $20.667*2^{-64}$. The minimum occurs when the truncated portion contains all zeros. This truncation can produce errors in the LSB compared to non-truncated approaches, but is correct in over 99.5 percent of all cases. This error can be made more symmetric by adding the average value of the truncated portion. The effect of truncation without compensation is a result fraction at most one lsb smaller than expected from a full multiplier array. With compensation, the resulting fraction can range from one lsb smaller to one lsb larger than expected from a full multiplier array. Unfortunately truncated multiplies do not remain commutative if Booth encoding is used. Since the Booth encoding logic took nearly as much area on the chip as the additional level of adders and significantly increased the complexity of the multiplier array, future implementations would probably not use Booth encoding.

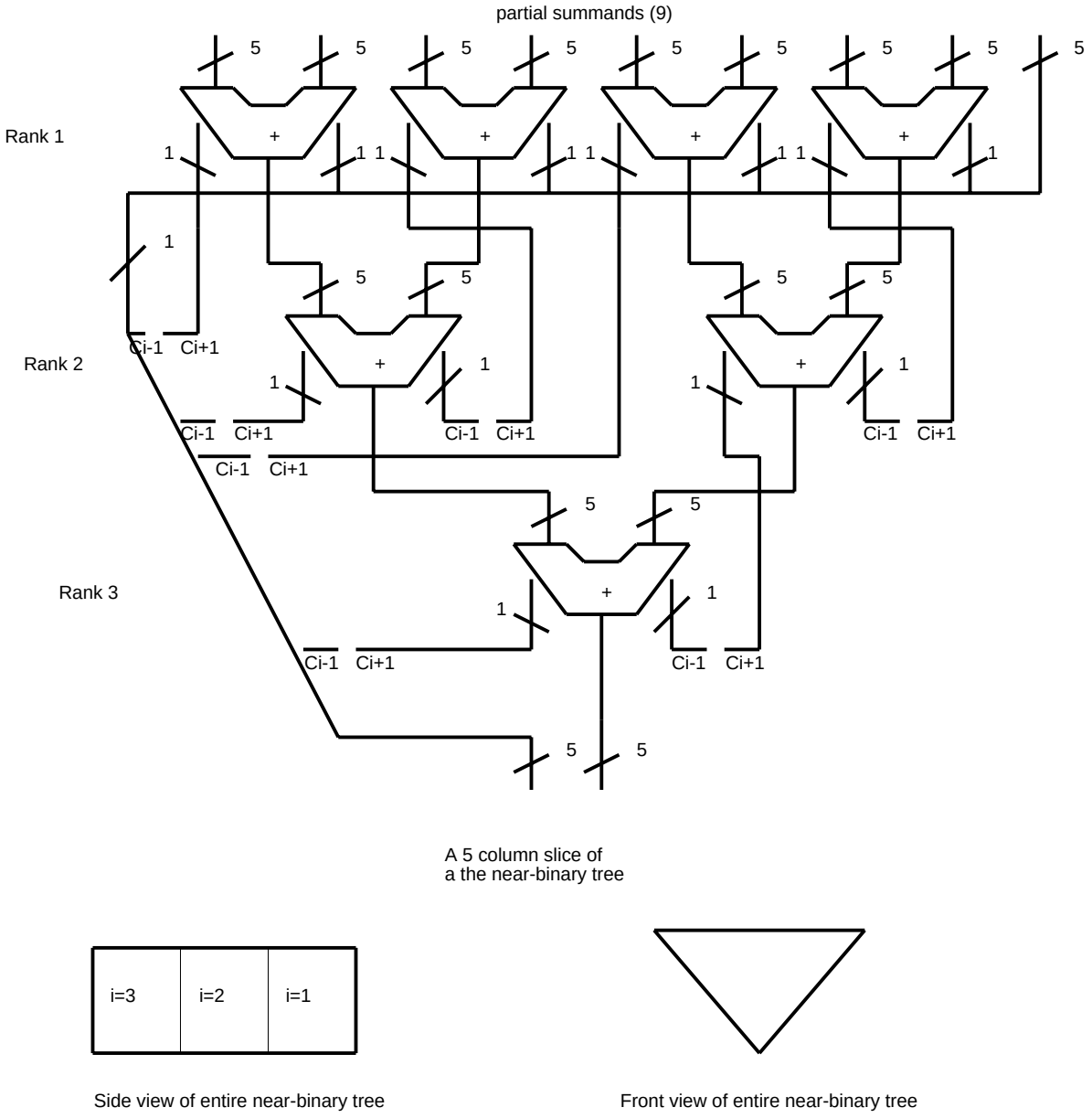


Figure 3-8: New Method for Reducing 9 15-bit Partial Products

Sixth, two carry save adders and two high speed carry skip adders are used to form the final 3 to 1 partial product reduction (see Figure 3-10). The results of a multiplication can be in the range 1/4 <= x < 1. In order to perform rounding at the same time as the final partial product reduction, the final magnitude of the fraction would need to be known in order to add the rounding bit into the correct place. However, this is only known after the final partial product reduction is complete. In the approach implemented, two paths are used to converge from 3 partial products to the final sum. The first path assumes a final sum in the range 1/2 <= x < 1 and adds 1/2 the least significant bit in the unused low order bits of one of the partial products. The second path assumes a final range of 1/4 <= x < 1/2, adds 1/4 the least significant bit, and always performs a one bit normalization shift and decrements the exponent by

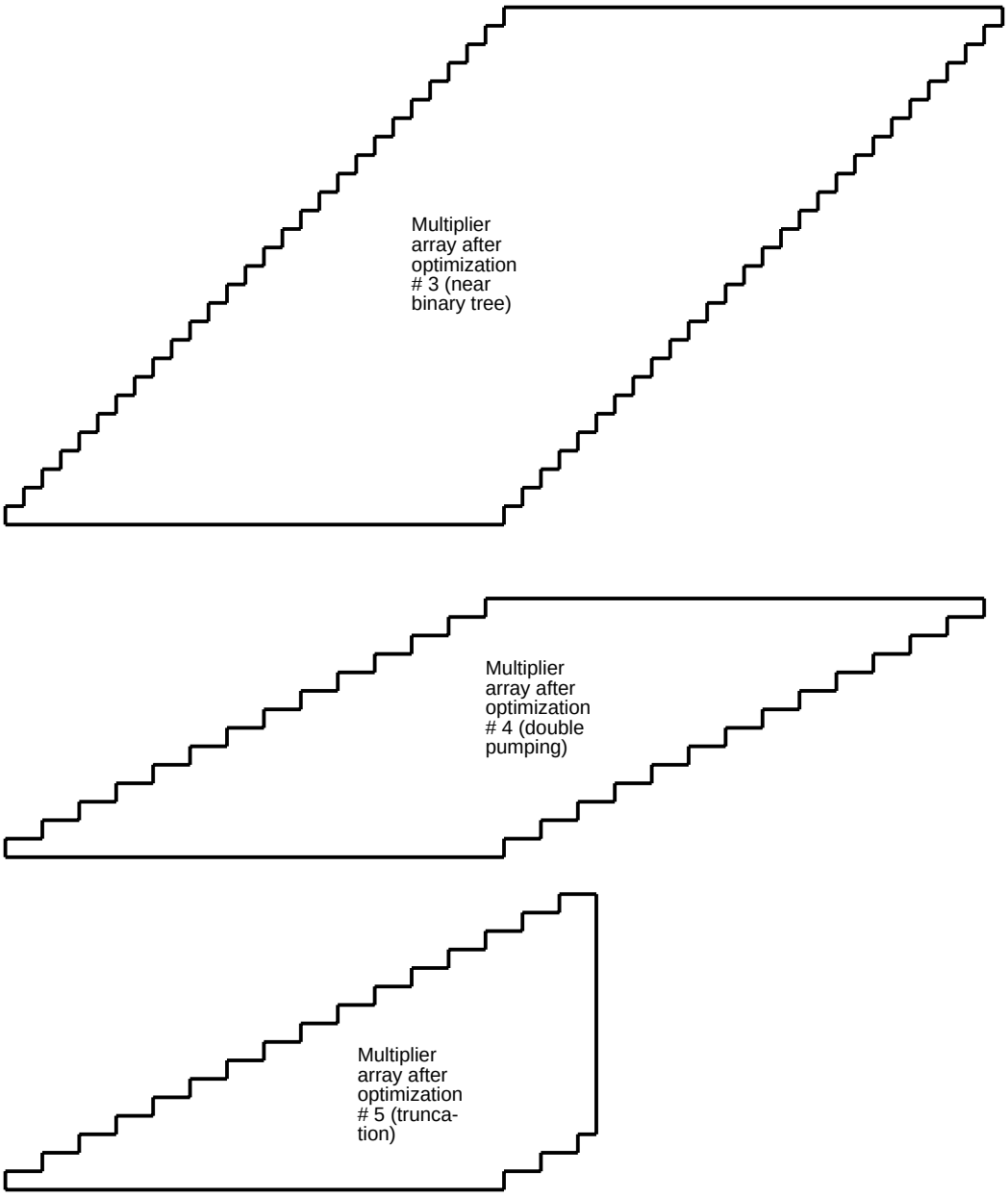


Figure 3-9: Reduction in Multiplier Area from Techniques Four and Five

one. Then the proper rounded sum and its associated exponent are late multiplexed into the output register, according to the table of Figure 3-11. This approach requires one additional carry save adder, but reduces the two serial 64 bit adds to a single 64 bit add. Note that the final rounded product is never greater than or equal to 1. This can be seen as follows. The largest initial fraction is 2^0-2^{-53}. The product of the two largest initial fractions is $(2^0-2^{-53})^2 = 2^0-2*2^{-53}+2^{-53*2} = 2^0-2^{-52}+2^{-106}$. This number will remain less than 1 after rounding since it has a zero in the lsb.

Seventh, high performance circuit designs are used. The timing of the multiplier is given in Figure 3-12.

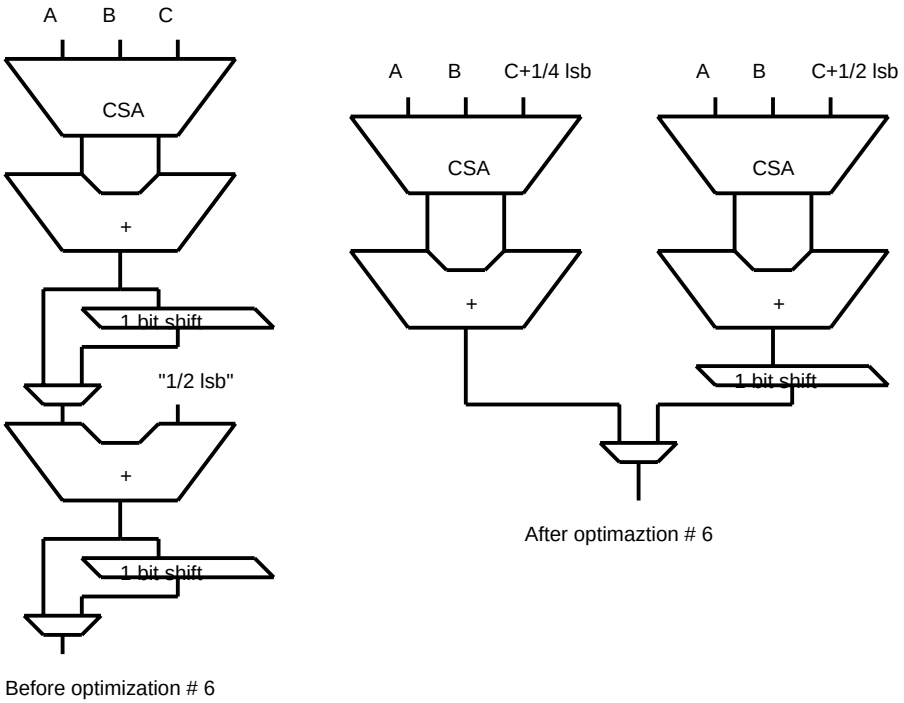


Figure 3-10: Concurrent Final Sum Generation and Rounding

```
Possible outcomes of 1/4 lsb and 1/2 lsb roundings:

Unrounded          Round by Addition of
Product          1/4 lsb          1/2 lsb
-------------------------------------          Range Key:
   L                L                L          "L" = 1/4 <= x < 1/2
   L                L                H          "H" = 1/2 <= x < 1
   L                H                H
   H                H                H

Late Selection Rule:    if (1/4 lsb rounded version in L range)
                        then product := (2 * 1/4 lsb rounded version)
                        else product := 1/2 lsb rounded version;
```

Figure 3-11: Late Selection of Proper Rounded Sum

3.2.1 Integer Multiplication

In this mode, the multiplier takes two 32-bit integers residing in the high order bits of the source operands and multiplies them yielding a 32-bit result. The 32-bit integers are taken off the source busses via different lines than those used for floating point numbers, so that they are repositioned in the part of the multiplier array normally used for the high 32 bits of the fractions. Similarly, on output the low 32 bits of the 64 bit product is placed on the high-order half of the result bus. The multiplier array must be at least 64 bits wide for this operation; thus only ((53*2)-64) => 42 bits of the array may be truncated. Although floating point multiplications operate on unsigned (i.e., sign and magnitude) fractions and integer multiplications take signed operands (i.e., two's complement),

```
cycle  phase      operation
-------------------------------------------------------------------
  1      1     operands driven to multiplier
         2     2 bit Booth recode; generate even summands - odd
         3     14 summand to 7 + 7/5 = 8 + 2/5 reduction  | sum-
         4     8 + 2/5 to 4 + 4/5 reduction               | mands
  2      1     4 + 4/5 to 2 + 4/5 reduction               | are 2
         2     2 + 4/5 to 1 + 4/5 reduction               | phases
         3     1 + 4/5 to 1 + 1/5 reduction               - later
         4     wait for odd summands 2 phases behind
  3      1     CSA the two pumpings (1+1/5+1+4/5 = 3 to 2)
         2     begin final 64 bit add
         3     continue final 64 bit add
         4     complete final 64 bit add
  4      1     late mux and drive result
```

Figure 3-12: Multiplier Phase by Phase Timing

hardware for signed partial products must already be in place for the two bit Booth algorithm, which may add in -1x or -2x the multiplicand. Thus floating point multiplicands are treated as positive quantities with an extra leading "0" bit for the sign. Booth's algorithm correctly handles signed multipliers. Special circuits are provided on the boundaries of the array to implement sign extension without extending the multiplier array itself.[7]

3.2.2 Iteration Step

Each reciprocal approximation iteration consists of two multiplications and a subtraction: $R_i := R_{i-1} * (2 - R_{i-1}*D)$. The iteration step instruction performs a multiplication and a subtraction, yielding the $(2 - R_{i-1}*D)$ part of each iteration. Although most subtractions may require either a large alignment or normalization shift, in the case of a reciprocal iteration step we know the product R*D approaches 1. That is, on each iteration R*D will be in the range 1+e > R*D > 1-e, and s = (2 - R*D) will also be in the range 1+e > s > 1-e. Since the significance of the product fraction is known, it allows the subtraction from 2 to occur concurrently with the reduction of the final partial sum and partial carry into the product.

Iteration step is implemented as follows. First, the Booth recoding is changed to swap +1 multiplicand with -1 multiplicand, and to swap +2 multiplicand with -2 multiplicand. This results in a product of -R*D. Then we add 2 while adding the partial sum, partial carry, and rounding bit. The result is in the range 1+e > (2-R*D) > 1-e. Both fast carry skip adders must be used since the rounding bit must be added into different positions when 1+e > (2-R*D) >= 1 and 1 > (2-R*D) > 1-e. These positions correspond to the 1/2 lsb and 1/4 lsb roundings performed for conventional multiplications. The decision criterion for choosing between the two adders is the same as that for multiplications: if the 1/4 lsb rounded result is normalized then we will chose the 1/2 lsb rounded result, else we will choose the 1/4 lsb rounded result and normalize it with a one bit left shift. Finally, note that the addition of 2 to -R*D can be effected by simply ignoring all bits to the left of $.1*2^1$ in -R*D.

3.3 Reciprocal Approximation

Division[8] in the MultiTitan FPU is performed by multiplying the dividend by the reciprocal of the divisor. The reciprocal approximation unit provides a 18 bit approximation (accurate to 16 bits) to the reciprocal of its input. This is used to start two Newton iterations for a G format reciprocal. The final reciprocal will have both a maximum

[7]These techniques are described in F100183 data sheets.

[8]Alan Eustace has implemented the reciprocal approximation unit. This section is contributed with his assistance.

	Using 1/4 lsb rounding adder: 2-R*D < 1	Using 1/2 lsb rounding adder: 2-R*D >= 1
	s|<. fraction >| exp	s|<. fraction >| exp
2:	001.0000..00..00 * 2^1	001.0000..00..00 * 2^1
R*D:	000.1000..0X..XX * 2^1	000.0111..1X..XX * 2^1
-R*D:	111.0111..1X..XX * 2^1	111.1000..0X..XX * 2^1
2-R*D:	000.0111..1X..XX * 2^1	000.1000..0X..XX * 2^1
rounded 2-R*D:	000.0111..1X..XX * 2^1 or 000.1000..0X..XX * 2^1	000.1000..0X..XX * 2^1

Figure 3-13: Iteration Step Operation

and average error residue from the initial approximation which is less than those present in the multiplier due to the truncation of the multiplier array. A block diagram of the reciprocal unit is given in Figure 3-14. The reciprocal unit requires about 8% of the total chip area.

Once the initial reciprocal approximation R_0 of a divisor *D* is obtained from the reciprocal unit, its accuracy must be increased by two Newton iterations. Each iteration produces a new approximation R_i with twice the accuracy of the last approximation R_{i-1}, by the following computation: $R_i := R_{i-1} * (2 - R_{i-1} * D)$. Once the two iterations are complete, then the dividend must be multiplied by the reciprocal. Note that the multiplier performs $(2 - R_{i-1} * D)$ in a single three cycle operation when in iteration mode. Thus each iteration (for a scalar) requires six cycles. Therefore the initial approximation, two iterations, and final multiply require a total of 18 cycles. However, since the multiplier and reciprocal unit are pipelined this reduces to 6 cycles per element for vectors of length three or more.

The reciprocal approximation fraction is produced by piecewise linear approximation.[9] First, the 8 most significant bits of the fraction (excluding the hidden bit) are used to index a ROM that contains a 18-bit base and an 8-bit slope. The next 8 bits of fraction (local) are multiplied by the slope and added to the base. The ROM logically has 256 word lines and 26 output bits (not including the hidden bit). It is implemented as a ROM with 32 word lines and 208 outputs, with eight words interdigitated on one word line and a multiplexor to select between eight adjacent bit lines. The internal timing of the reciprocal approximation unit is given in Figure 3-15.

The exponent of the reciprocal is the two's complement of the input exponent plus one. This will be shown by first considering what happens when we take the reciprocal of a number with a fraction of one, and then consider normalized fractions. First, the reciprocal of a biased exponent is its two's complement. For example, 2^2 has an exponent of 1000000010, and 2^{-2} has an exponent of 0111111110. Second, for an input fraction in the range $1/2 <= x < 1$, the reciprocal without normalization is in the range $1 < x <= 2$. To normalize it into the range $1/2 <= x < 1$, we need to increment the exponent. (We will also return the largest fraction less than one in the case where the input is 1/2 to prevent the need for a two bit normalization.)

The reciprocal unit will generate an overflow error in two situations: if the initial exponent is zero (i.e., divide by zero), or if the initial exponent is the smallest possible exponent -1023 (i.e., the two's complement plus one of its biased exponent will overflow.)

[9]Jud Leonard, unpublished communication, "Reciprocal Approximation", July 1986.

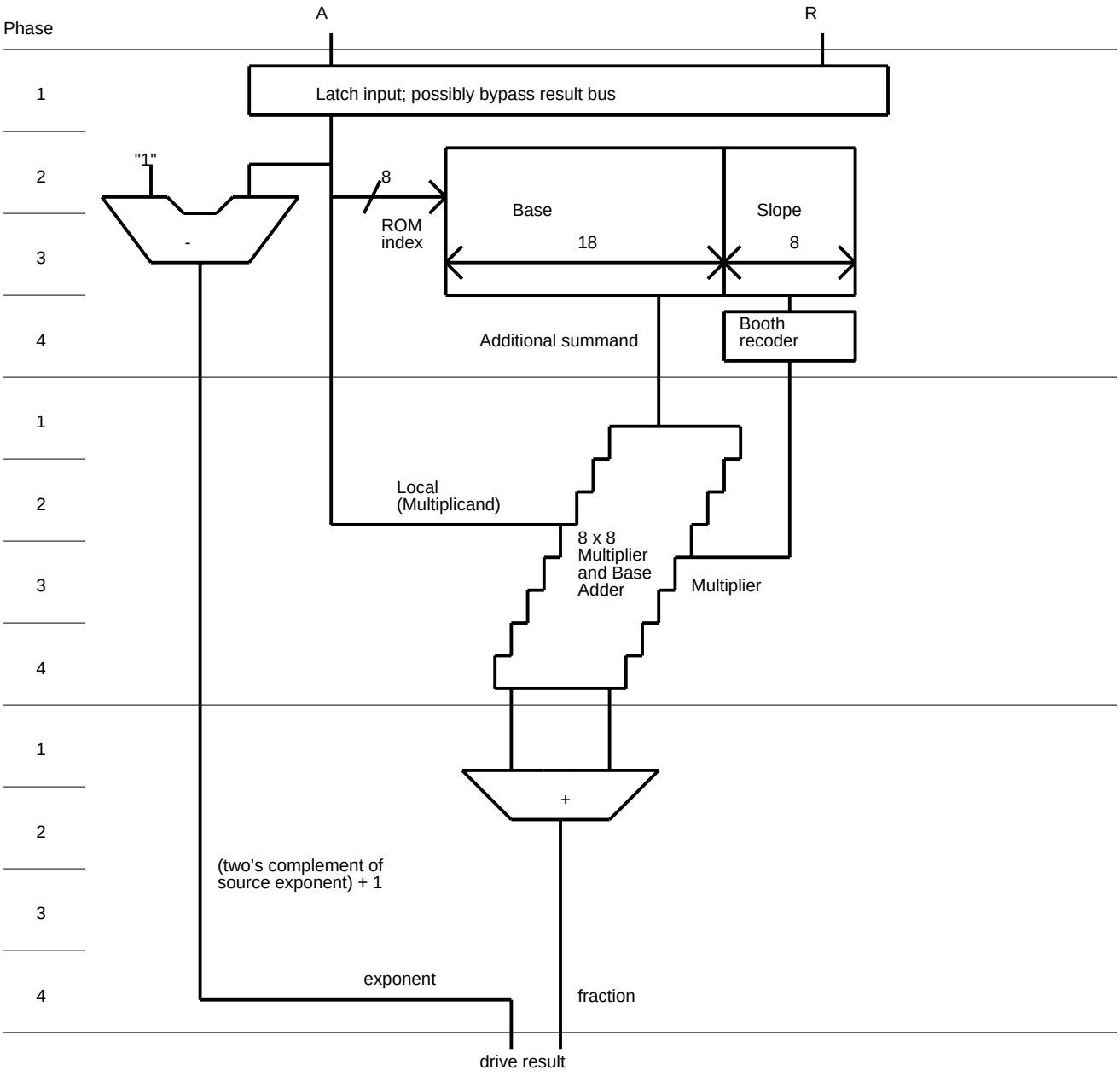


Figure 3-14: Timing vs. Structure of the Reciprocal Approximation Unit

A quadratic approximation that returns 32 good bits was investigated in order to reduce the number of required Newton iterations to one.[10] for optimal methods of approximating the reciprocal function over a wide range of lookup table sizes and polynomial degrees.] Although half the precision of the mantissa is 27 bits, 32 good bits were desired for two reasons. First, with 32 good bits integer division would not require any iteration steps. Second, since the multiplier is truncated to 64 bits, 32 bits are required to provide rounding bits in the non-truncated

[10]See Anderson, Ned, "Minimum Relative Error Approximations for 1/t," Digital Equipment Corporation Technical Report, 1987.

```
cycle   phase   fraction operation      exponent operation
-------------------------------------------------------------------
  1       1             operand driven to reciprocal unit
          2     start ROM access        start 2's compl. + 1
          3     finish ROM access       finish 2's compl. + 1
          4     Booth recode slope      wait
  2       1     4 + base -> 3 reduction .
          2     3 -> 2 CSA              .
          3     begin final sum         .
          4     finish final sum        .
  3       1     wait                    .
          2     .                       .
          3     .                       .
          4     .                       .
  4       1             drive result
```

Figure 3-15: Reciprocal Approximation Phase by Phase Timing

portion of the array. Unfortunately the hardware and time required for the computation were prohibitive. The ROM required for the quadratic approximation method considered would be about eight times that for the linear method. Also, a 12x12 and a 24x24 multiplier would be needed. The approximation would also require four cycles to compute, which would not fit into to the simplified control model with uniform three-cycle latency.

3.4 Register File

The register file contains 52 64-bit general purpose registers, and uses 10% of the total chip area. It logically sits between the 64-bit external data bus and the functional units (see Figure 3-2). The register file logically has four ports *A*, *B*, *R*, and *M*, which are implemented as two ports used twice per cycle. The A and B ALU source operands are read from the *A* and *B* register file ports during phase 4 and unconditionally driven onto the A and B source busses on phase 1. Port *R* is used to write back results of functional units on phase 2. Port *M*, the memory access port, is active on phase 2. It is read during FPU->CPU transfers and FPU stores. It is written in CPU->FPU transfers and FPU loads. During FPU stores and FPU->CPU transfers in which there is concurrently a write of the same register from the R port, the M port will switch to reading the R port in order to implement a bypass. In cycles where a R port write and M port write target the same destination register, the results are undefined.

3.5 Scoreboard and Internal Timing

This section first describes the phase by phase timing of the major busses, signals, and units in the FPU. Then the scoreboard,[11]which is responsible for issuing instructions, is described. Finally, the method used to synchronize the CPU and FPU pipelines is described.

Central to the scoreboard is a *register write reservation table*. This table consists of one bit for each register in the register file which is set when there is an outstanding operation which will write the associated register. This bit is used to prevent subsequent instructions from reading the register before it has been written. This hides the pipeline from the software.

[11]Mary Jo Doherty has implemented the scoreboard and pipeline control logic. This section is contributed with her assistance.

3.5.1 Internal Timing

In contrast to most high performance floating point engines, the FPU has a lock step pipeline like the CPU, greatly simplifying the control logic. For example, since all functional units have a three cycle latency, the functional unit write port to the register file need not be reserved or checked for availability before instruction issue.

The FPU control is split into two parts. The first part manages the operation of FPU loads, stores, and transfers. The second part manages FPU ALU instructions. These two parts of the machine communicate through the register file, register reservation bits, and the inter-chip pipeline control signals. The use of the pipeline control signals will be explained in detail in Section 3.5.2.3, however an overview of their use follows. LoadWB (and hence LoadMEM, LoadALU, and LoadIF) are deasserted by the FPU ALU control section and stall the control section for FPU loads, stores, and transfers (i.e., the LST section). This is used to prevent FPU loads, stores, or transfers from executing before prior FPU ALU instructions have issued. (In the case of vector FPU ALU instructions, only one element must be issued for the stall to be cleared.) There is no dual of this situation to prevent FPU ALU instructions from executing ahead of previous loads, stores, and transfers because they all use the data bus and are serialized by it. Figure 3-16 is a composite timing description for all types of FPU loads, stores, and transfers. Figure 3-17 describes the timing for an FPU ALU instruction.

```
stage    phase      operation
-------------------------------------------------------------
  ALU      1      read CPU opcode bus
           2
           3
           4
  MEM      1
           2      read register if FPU->CPU
                  read register reservation bit
           3      deassert LoadWB if register reserved
           4      drive data bus if FPU->CPU
                  latch data bus (for FPU load)
  WB       1
 (EX1)     2      read or write register for FPU store or load
                  read register reservation bit
           3      deassert LoadWB if FPU store register reserved
                  latch data bus (for CPU->FPU)
                  drive data bus if Store
           4
  EX2      1
           2      write register if CPU->FPU
```

Figure 3-16: FPU Load/Store/Transfer Instruction Timing

Once an ALU instruction gets into the WB pipestage, it is guaranteed to complete. This is required for compatibility with CPU instructions, which also commit in the WB pipestage. Vector instructions that overflow on one element discard all remaining elements after the overflow. The destination register specifier of the first element to overflow is saved in the PSW. Note that vector ALU instructions may continue long after an interrupt. For example in the case of vector recursion (e.g., r[a] := r[a-1] + r[a-2]), if there was an interrupt just after the instruction entered WB the last element would not be written for 48 cycles.

```
stage    phase      operation
-------------------------------------------------------------
  ALU      1      read CPU opcode bus
           2
           3
           4
  MEM      1      coproc ALU instr driven from CPU
           2
           3      set up register file predecoders
           4      read and latch source registers
                  send op code to FU's
                  read register reservation table
                  compute if this was a valid FPU ALU instr
                  compute if bypass required - tell FU's
                  if interrupt, nop instr leaving MEM
  WB       1      drive sources to FU's
 (EX1)            provisionally issue the instruction
                  FU's latch sources or result if source bypass
                  FU's start computing, first FU pipestage
           2      if instr was valid and ready when issued
                  then begin reserve destination reg;
                          increment appropriate register specifiers;
                          decrement vector length
                       end
           3      if ((CPU is sending another ALU instr) and
                     (vector length remaining <> -1))
                     or register interlock
                  then deassert LoadWB
           4
  EX2      1      continue computing, second FU pipestage
           2      .
           3      .
           4      .
  EX3      1      continue computing, third FU pipestage
           2      .
           3      .
           4      .
  PA       1      FU drives result
                  clear reservation bit
           2      write result if IR[PA]<>nop
           3      assert interrupt if IR[PA]<>nop and overflow
```

Figure 3-17: FPU ALU Instruction Phase by Phase Timing

3.5.2 Scoreboard

Five ports are required on the register write reservation table every cycle:

- 2 read with source operands for ALU operations on phase 4
- 1 set with destination for ALU operation issue on phase 2
- 1 cleared for destination of retired ALU operation on phase 2
- 1 read for loads, stores, and transfers on phase 2

Note that ALU destinations are not checked for pending writes; these can only happen when the first write is a dead computation, and since instructions retire in order the live result will overwrite the dead result. However, the reservation bits are read during loads or CPU->FPU transfers; if an ALU operation is in progress with a pending write for that register, the load or CPU->FPU transfer will stall until the ALU operation is complete. Note that this

would be another example of dead code, but if a load was not stalled the load might retire first followed by the write of the dead result. Since dead code should not exist, this check is not necessary. However, it is cheaper to implement than to omit because we must check the scoreboard on stores and FPU->CPU transfers.

Both FPU ALU source specifiers are always checked for pending writes (i.e., unused ALU source specifiers are checked for pending writes in unary FPU ALU operations.) In order to prevent this from causing unnecessary delays, in unary operations the unused cb source specifier should be the same as the ca source specifier.

Of the five required scoreboard ports, all ports are accessed at the same time as their associated data, except for the port that sets a bit on issue of FPU ALU instructions. For example, the ALU source operand reservation bits are read at the same time as the ALU source operands. Moreover, both writes of reservation bits occur on the same phase, and one is a set while the other is a clear. We will take advantage of these restrictions in the following implementation. This implementation has the advantage that it requires only one extra decoder besides those already required for the register file, and in the case of a single reservation bit the decoder area greatly exceeds that for the RAM cell.

Reservation bits are stored as an extra bit on each word in the register file. The R port word line of the extra bit is partitioned into two separate word lines. One segment is controlled by the same word line as the rest of the word. The other is controlled by the destination of the provisionally issued instruction during phase 2 and the usual source operand during phase 4. Since we will never want to write a reservation bit with an arbitrary value, but only set one or clear one, we can do both by single-ended writes. On phase 2 both bitlines are driven low; the true bitline will be used to clear a bit at the same time as the complement bit line is used to set a bit. Where the same register is to be cleared and set, the clear is disabled so that undefined values will not result.

3.5.2.1 Functional Unit Bypasses

Unlike the CPU, each functional unit in the FPU does its own bypassing. If the bypass logic was centralized at the register file, results would have to be put out on the global result bus, then transferred to a global source bus. But since the result bus goes to all functional units, they can select between each source and the result bus based on control signals from the scoreboard. Thus with distributed bypass logic the delay from driving the result to the latching of a source is only one global wire delay, not two.

3.5.2.2 Load/Store/Transfer Bypasses

The results of functional units are driven onto the result bus on phase 1 and written into the register file R port on phase 2. The M port of the register file is also read on phase 2 for loads, stores, and transfers. In cases where the M port is reading the same value as the R port is writing, the new value may not propagate through the register cell in time to prevent false triggering of the sense amps. When identical register addresses are detected on the M and R ports, the read logic on the M port is switched to the R port to make sure the new value is obtained.

3.5.2.3 Vector Result Reservation

The reservation of vector result registers is a difficult problem. Three approaches exist:

1. Reserve all elements of the result register at once before issue of the first element.
2. Handle reservations in software.
3. Reserve each element result register upon issue of the element.

Note that in machines like the Cray-1, vector registers are treated as an indivisible resource and the vector result register reservation problem is simplified to reserving a single resource.

Two difficulties occur if all result registers of a vector operation are reserved at once before issue of the first

element's computation. First, special hardware must be provided above that required for scalar operations to reserve more than one register at a time. Second, additional hardware must be provided to check for prior reservations on all result registers simultaneously, otherwise the vector register reservation may reserve an already reserved register, in which case the second reservation will be lost on the retiring of the first reservation.

Reservation of vector result registers can be handled by code reorganization. In most machines, where floating point operations have relatively long latencies (e.g., 7-30 cycles), scheduling operations well enough to prevent insertion of NOP's is unlikely. Furthermore, for large delays at least three instructions must be inserted (e.g., 1: initialize loop counter, 2: branch if done, 3: decrement loop counter), leading to poor code density. Since all operations in the FPU have a three cycle latency, the longest delay required would be 2 cycles, or two NOPs. This is more attractive. However, the next paragraph presents a more attractive hardware reservation method.

Reservation of individual vector result registers upon issue of each element can provide hardware interlocks with very low cost. By reserving result registers at the issue of each element, the reservation hardware already in place for scalar operations can be used. Unfortunately this causes a synchronization problem: while the elements of the vector are issuing one by one, a load or store instruction may issue and retire. In particular, the register operand of the load or store may be the same as a source or result register operand of a vector element which has not already issued, but whose vector operation was issued before the load or store. In order to prevent out-of-order execution (with non-deterministic results), execution constraints must be placed between the vector instruction and any following loads and stores that issue before every element of the vector has issued. One solution would be to compute the remaining source and result register ranges of in-progress vector instructions each cycle, and compare load and store register operands against these ranges before issuing them. This would add a fair amount of reasonably complex hardware. A far simpler solution is provided by use of the existing inter-chip pipeline control signals.

LoadWB (and hence LoadMEM, LoadALU, and LoadIF) is deasserted in two LST/ALU synchronization situations. Both assume a prior FPU ALU instruction (of one or more elements) is being held in the ALU instruction register, waiting for a pending write of a source operand to complete. Also, in both cases no register addresses are compared, so this approach is simple at the expense of possibly generating unnecessary stalls. First, LoadWB is deasserted when a FPU store enters WB, or an FPU->CPU instruction enters MEM, preventing it from retiring until at least the first element of the prior ALU instruction has issued. Without deasserting LoadWB in this situation FPU stores or FPU->CPU transfers of the ALU destination might execute ahead of the ALU instruction issue. But if the ALU instruction had not yet been able to issue and reserve its result register, the store could execute getting the old value even though it came after the FPU ALU instruction. However once the ALU instruction has issued, the scoreboard will be able to properly handle register dependencies between the ALU instruction and the LST instruction. Second, LoadWB is deasserted when the next FPU load or CPU->FPU transfer instruction enters MEM, preventing the load from writing its result register until at least the first element of the ALU instruction has issued and hence read both its source operands. Without deasserting LoadWB in this situation FPU loads or CPU->FPU transfers might execute ahead of the ALU instruction issue. But if the ALU instruction was waiting for a pending write of one source operand, and the other source operand was the target of the load or transfer, the load could replace its second operand with its new value before its old value was used.

If dependencies occur between elements in a vector other than the first and following loads and stores, the compiler must break the vector into smaller vectors so that the LST/ALU synchronization using LoadWB has effect. However, in most code sequences this will not be necessary: for example, if a vector operation is followed by stores of each result register, the stores can be performed in the same order as the result elements are produced. Only when operands must be stored out of order, or when the first elements of a vector are not stored but later elements are, will the compiler need to break a vector in order for the LST/ALU synchronization via LoadWB to be effective. Note

that if an entire vector was required to issue before loads and stores were honored, most useful overlap of transfers and computations would be precluded.

3.5.3 CPU - FPU Synchronization

The FPU must track the pipeline of the CPU, so that they are synchronized during transfers. It tracks the CPU by observing the LoadALU, LoadMEM, LoadWB, and NotInterrupt signals (see Figure 3-18). Pending source is an internal signal asserted if there is a valid FPU ALU instruction that can not issue because the scoreboard detects a source register is reserved for writing by a previous FPU ALU instruction. Vector in progress (VIP) is an internal signal asserted if after issuing an FPU ALU instruction and decrementing its vector length field its "vector length - 1" field is not -1. Because the FPU can retire both a LST instruction and an ALU operation each cycle, two MEM and two WB instruction registers are required. The instruction register for LST instructions in the MEM pipestage is called MEM[LST]; the instruction register for FPU ALU instructions in the MEM pipestage is called MEM[ALU]. Similarly, the instruction register for LST instructions in the WB pipestage is called WB[LST]; the instruction register for FPU ALU instructions in the WB pipestage is called WB[ALU].

Pipe stage	Load-MEM	NotInterrupt	Load-WB	Load-ALU	Pending source	Vector InProg.	Next stage
ALU	0	X	X	X	X	X	ALU
	1	0	X	X	X	X	nop
	1	1	0	X	X	X	ALU
	1	1	1	0	X	X	ALU
	1	1	1	1	X	X	nop
	1	1	1	1	X	X	opcode bus
MEM	0	X	X	X	X	X	MEM
	1	0	X	X	X	X	nop
	1	1	0	X	X	X	MEM
	1	1	1	0	X	X	nop
	1	1	1	1	X	X	ALU
WB[LST]	0	X	0	X	X	X	WB
	0	X	1	X	X	X	nop
	1	0	X	X	X	X	nop
	1	1	0	X	X	X	WB
	1	1	1	X	X	X	MEM
WB[ALU]	X	X	X	X	1	X	WB
	X	X	X	X	0	1	next element
	0	X	X	X	0	0	nop
	1	0	X	X	0	0	nop
	1	1	0	X	0	0	nop
	1	1	1	X	0	0	MEM
EX2	X	X	X	X	1	X	nop
	X	X	X	X	0	X	WB
EX3	X	X	X	X	X	X	EX2
PA	X	X	X	X	X	X	EX3

Figure 3-18: FPU Pipeline Control

The CPU opcode and register specifier are latched into IR[ALU] on phase 1 of every cycle in which LoadALU and

NotInterrupt are asserted. When latched, the opcode and register specifier correspond to the instruction in the ALU pipestage of the CPU. The instruction in IR[ALU] advances to IR[MEM] on phase 1 in cycles without LoadALU, LoadMEM, LoadWB or NotInterrupt deasserted on the previous phase 4. If NotInterrupt or LoadALU is deasserted on the previous phase 4 IR[MEM] is cleared.

If the instruction in IR[MEM] is a coprocessor ALU instruction, then the CPU will send out the ALU instruction over the address lines this cycle, and it will be latched in IR[MEM[ALU]]. The contents of the address lines are always evaluated as an FPU ALU instruction during the MEM pipestage; however they are only advanced into WB if it is a valid FPU ALU instruction. If the instruction in IR[MEM[ALU]] is an FPU ALU instruction it will advance to IR[WB[ALU]] in the absence of ~LoadWB, ~LoadMEM, pending source and vector in progress. Pending source and vector in progress have higher priority than ~LoadMEM or interrupt; if a source operand is reserved then IR[WB[ALU]] will recirculate. Similarly if a source is not reserved, but a vector has not been fully issued, the next element of the vector will be latched into IR[WB[ALU]] independent of all external conditions. Only when neither pending source or vector in progress are asserted will ~LoadMEM, interrupt or ~LoadWB force IR[WB[ALU]] to nop.

If the instruction in IR[MEM] is a load, store, or transfer instruction, it will advance to IR[WB[LST]] unless LoadMEM, NotInterrupt or LoadWB are deasserted. IR[WB[LST]] is cleared if it is a LST instruction and interrupt is asserted or LoadMEM is deasserted and LoadWB is asserted.

Instructions in IR[WB[ALU]] advance to IR[EX2] unless their source operand is reserved, in which case IR[EX2] is nop.

Instructions in IR[EX2] and IR[EX3] always advance to IR[EX3] and IR[PA], respectively.

3.6 Pinout and External Interface

Figure 3-19 lists the pins of the FPU. For more details on signal function and timing, please consult the MultiTitan Intra-Processor Bus Definition and Timing document.

# of Pins	Type	Function
64	I/O	Data bus
8	I/O	Data bus byte parity
32	I	Address bus
6	I	Cr (or Ca) register address from CPU
4	I	Instruction opcode from CPU
1	I	User/kernel mode
1	I	LoadALU
1	I	LoadMEM
1	I/O	LoadWB
1	I/O	NotInterrupt
1	I	AllowInterrupts
1	I	enable data bus drivers (from CCU)
1	I	check parity enabled
1	I	Power-up reset
1	I	Clock In
4	C	Phi1 though Phi4 output
2	I/O	Spare
130		signal pads used of 144 available
	plus:	
16		I/O Vdd
16		I/O GND
2		internal Vdd
2		internal GND

Figure 3-19: FPU Pinout

4. Sample Applications

4.1 Graphics Transform

This section describes a graphics routine to transform a point by multiplying a vector by a transformation matrix. It is representative of many possible applications for the FPU. The problem and register allocation are given in Figure 4-1. Assume that many points will be transformed by one matrix. Thus the transformation matrix will already be loaded into C0..C15. If the transformation matrix is not loaded, this will require an extra 16 cycles (assuming no cache misses).

```
Problem:

[x y z w] * |a11 a12 a13 a14| = [x' y' z' w']
            |a21 a22 a23 a24|
            |a31 a32 a33 a34|
            |a41 a42 a43 a44|

Register allocation:

[c32 c33 c34 c35] |c0 c4 c8  c12| = [c36 c37 c38 c39]
                  |c1 c5 c9  c13|
                  |c2 c6 c10 c14|
                  |c3 c7 c11 c15|
```

Figure 4-1: Problem Statement and Register Allocation for Graphics Transform

For each element of the initial point vector we will load it and issue a vector floating point multiply of the element by a column in the transform matrix, resulting in a total of 16 result elements. Once these multiplications have been issued, we will start adding together rows of the 4x4 result elements. Each row is added together in a binary tree, and the four trees are summed in parallel using four element vectors. Finally we will store the result vector. Figure 4-2 gives the code sequence and cycle timings for this routine. Each instruction requires one cycle, with two exceptions. First, back-to-back stores require two cycles. Second, arithmetic operations can not issue until a previous vector operation has completely issued all of its elements at a 1 element per cycle maximum rate. There is only one scoreboard stall for data dependencies in the routine. It is assumed that there are no instruction buffer misses during the routine. This example has been run on the MultiTitan simulator[12] and achieves 20 MFLOPS. The total latency in this example is 35*40ns cycles (1.4us), for double precision computations.

4.1.1 Comparison with Other Approaches

Another possible solution to this problem would be to build a custom register file and reciprocal ROM chip that interfaced to Weitek chips for multiplications and addition/subtraction. Two copies of each custom chip would be required in each system; this is because due to pin limitations each custom chip could only connect to half the 64 bit MultiTitan processor data bus, half of the two Weitek 32-bit source operand busses, and half of the Weitek 32-bit result bus. The Weitek 1264 and 1265 do double precision operations; however, in double-precision mode the multiplier chip accepts operands at half its maximum rate. Thus the fastest system would consist of a adder/subtractor chip with two multiplier chips in parallel: one for operations initiated in odd cycles and one for operations initiated in even cycles. This system could transform a point by the previous algorithm in 4.56us.

This approach seems attractive on the surface, but it has some hidden problems. It is attractive because it is only 2.5

[12]Mike Nielsen, unpublished communication, "MultiTitan Simulator and Test Programs",May 1986.

```
Solution:                              Cycles:
-----------------------------------------------------------
/* load and multiply initial vector. */
c32:=(x);                                 1
c[16..19]:=c32*c[0..3];                   1
c33:=(y);                                 1
c[20..23]:=c33*c[4..7];                   3 (issue unit busy)
c34:=(z);                                 1
c[24..27]:=c34*c[8..11];                  3 (issue unit busy)
c35:=(w);                                 1
c[28..31]:=c35*c[12..15];                 3 (issue unit busy)

/* sum rows of products in parallel binary trees. */
c[16..19]:=c[16..19]+c[20..23]            4 (issue unit busy)
c[24..27]:=c[24..27]+c[28..31]            4 (issue unit busy)
c[36..39]:=c[16..19]+c[24..27]            4 (issue unit busy)

/* store result vector. */
(x'):=c36;                                3 (wait for result)
(y'):=c37;                                2
(z'):=c38;                                2
(w'):=c39;                                2
-----------------------------------------------------------
Total latency:                           35
```

Figure 4-2: Code and Cycle by Cycle Timing for Graphics Transform

times slower than the full custom FPU. However, it does require a fair amount of custom design to build the register file and reciprocal ROM chip. Second, the Weitek chips are specified for 60ns cycles; the CPU needs to match this if synchronous design is to be maintained. This would adversely affect all operations. Third, the resulting design is five chips instead of a single FPU chip. Fourth, design time spent building the Weitek interface side of the chip could be more interestingly applied to the design of an on-chip adder/subtractor and multiplier. Fifth, several functions in the MultiTitan FPU would not be performed by the Weitek chips directly, such as trunc, float, and iteration step. Sixth, in other applications the latency of each operation is more important. In these applications a system using Weitek chips would have performance degradations of 4 and 5 for multiplications and addition/subtraction, respectively.

As another point of comparison, the 1986 10-chip Geometry Engine pipeline can transform, clip, and scale points at approximately 90,000 per second, or about 11us apiece. Finally, this application was also implemented with the Weitek 1163, 1164, and 1165 interface, multiplier, and ALU chips hosted by an Intel 80386[13], which required 250 60ns cycles, or 15us, for single precision computations.

4.2 Linpak

Linpak has been run on the MultiTitan uniprocessor simulator.[14] The scalar Linpak performance obtained was 4.1 MFLOPS, while the vector Linpak performance obtained was 6.1 MFLOPS. The scalar performance is approximately 25 times the performance of a VAX 11/780 with FPA.

---

[13]Electronic Design, May 1, 1986, pg 213-219

[14]Mike Powell, unpublished communication, "Linpak Performance", September 1987.

4.3 Livermore Loops

Several Livermore Loops have been on the MultiTitan simulator.[15] The results obtained for one processor are shown in Figure 4-3. The performance of each loop was highly dependent on whether the data referenced by the loop was present in the cache. Simulations were run assuming a cold start cache as well as a cache that had just previously run the loop. These two scenarios differed by over a factor of four in performance. Since the loops are kernels of programs that perform many operations on a data set, the assumption that the loop has just previously run is probably more realistic than assuming that the cache is completely cold upon execution of the loop.

loop	cold cache MFLOPS	warm cache MFLOPS
1	4.3	19.0
2	2.8	17.3
4	2.3	14.5
5	1.8	8.6
7	6.9	23.4
8	6.0	19.9
9	3.6	20.3
10	1.5	7.1
12	1.4	7.9
arithmetic mean	3.4	15.3

Figure 4-3: Uniprocessor Livermore Loops Performance

[15]Jon Bertoni, unpublished communication, "MultiTitan Floating-Point Performance", September 1987.

Table of Contents

List of Figures

MultiTitan Cache Control Unit

Jeremy Dion
Digital Equipment Corporation
Western Research Laboratory
100 Hamilton Avenue
Palo Alto, CA 94301

Version of 7 April 1988

1. Introduction

A MultiTitan is an eight-processor multiprocessor connected to a Titan memory and I/O subsystem. Each of the eight processors consists of the following major pieces:

- a CMOS processor unit (CPU)
- a CMOS floating point unit (FPU)
- a CMOS cache control unit (CCU)
- 128 KBytes of memory used as a per-processor cache, built from 23 16K x 4 KBit static RAMs.
- an interface to the memory system and shared processor bus consisting of a few TTL buffer registers and controlled by the CCU.

In each MultiTitan processor, the principal communication path is the 64-bit data bus with byte parity, to which each of the major units is connected. There is also a 32-bit address bus with no parity which only the CPU drives. Part of this bus addresses the direct-mapped cache, and part is used in memory translation as described in a following section. In transactions with the memory subsystem, the local address and data busses are driven to and from the two 32-bit memory controller busses at appropriate times. See the MultiTitan Bus Specification for more details.

Two notes about terminology:

- the MultiTitan CPU contains an on-chip instruction buffer. The CCU manages external instruction and data caches. In this document, "instruction cache" will mean the instruction half of the external cache.
- 1 word contains 32 bits

1.1 Cache Control Unit Functions

The CCU has the following major functions:

- **Address translation**: The CCU has an on-chip translation buffer containing mappings for 512 pages of 64 KBytes organized as a 256-entry table with a set size of 2. The total memory mapped by the TB is 32 MBytes. Addresses are looked up in this table in parallel with the access to the cache RAMs. If the TB holds no translation for this page, the CCU interrupts the CPU. The CPU can read and write TB entries using privileged instructions.
- **Cache reloading**: In parallel with TB lookups, a translation is read from the tag field of the selected line of the instruction or data cache, and the translations are compared. If the cache line holds the incorrect address, the CCU performs a transaction with the memory system to replace it. The cache policy is write-back, not write-through; that is, lines are written back to main memory only when they need to be replaced in the cache.
- **Cache consistency**: The CCU has instructions to allow the CPU to control the contents of the cache. A line can be flushed back to memory (for instance, before releasing a software lock), to override the write-back cache policy. A line can also be cleared (mapped into the cache, but not read from memory) if the CPU intends to overwrite all its data. Clearing a line is an optimization for program speed and is not necessary for program correctness.
- **Multiprocessor Synchronization**: A test-and-set primitive is provided for coarse-grained synchronization.
- Two operations, Send and Receive, are provided for fine-grain synchronization combined with high-bandwidth data transfer. These operations allow processors to synchronize and transfer cache lines with very low overhead, and are intended for parallelizing small sequences of code within single application programs.
- **I/O**: Using privileged instructions, the CPU can direct the CCU to perform simple I/O interactions with the memory subsystem.
- **Interrupts**: The CCU ensures that external interrupts are eventually taken by the CPU, and provides an instruction which allows a CPU to send an external interrupt to any other processor.

1.2 Address Translation

Translation of virtual addresses is performed as shown in figure 1-1. The data structures involved in address translation are a translation buffer held on the CCU, and a real address cache held in external RAMs. The real address cache is direct-mapped, that is, a cache line index selects exactly one cache line, and there is no associativity. This arrangement allows the data read out of the cache RAMs to be driven back to the CPU before the cache hit or miss signal is computed.

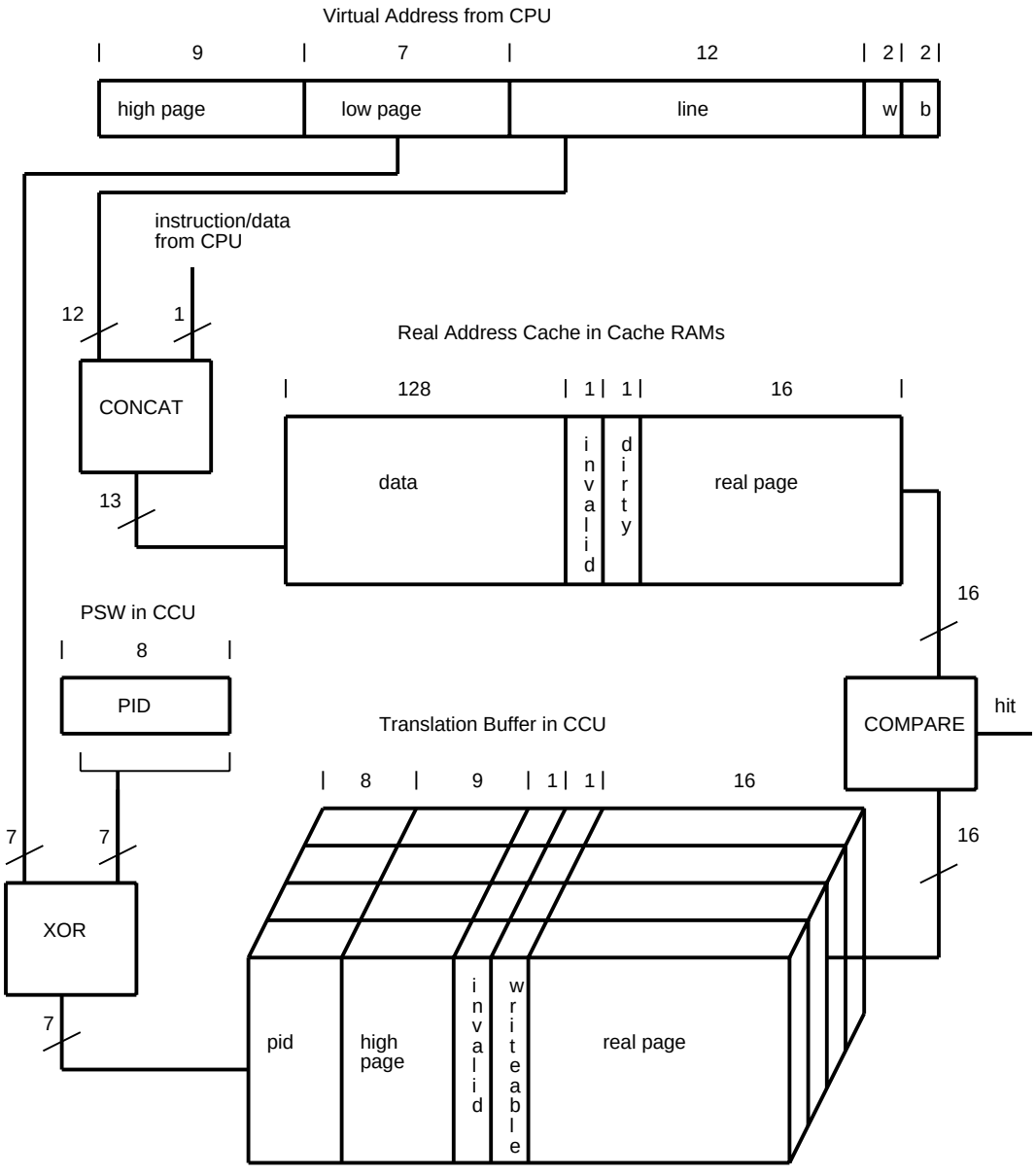


Figure 1-1: Address Translation

All virtual addresses are in bytes. The translation of a virtual address proceeds as follows. The MultiTitan page size is 64 KBytes, so the low order 16 bits give the position within a page, and the high order 16 bits give the page

number. The low order 7 bits of the page number (*low-page* in the diagram) are exclusive-ORed with the low-order 7 bits of the current process id in the CCU status register to get a 7-bit index into the on-chip translation buffer. This index selects a set of 4 translation buffer entries, and if the pid in the CCU status register and the high page bits match one of these entries, and if the invalid bit of this entry is not set, a valid translation exists. If this is a store operation, and the writeable bit is 0, then an interrupt is generated. If the translation buffer has two valid translations for a virtual address, the result of referencing that address is undefined. If no translation exists for this virtual page, the CCU generates a *translation buffer fault*. The total amount of memory mapped by this TB organization is 512 pages of 64 KBytes, or 32 MBytes. Instruction and data entries share the single translation buffer.

Meanwhile, the high 12 *line* bits of the low order 16 address bits are concatenated with the instruction/data reference bit to generate a 13-bit cache line index. Each cache line is 16 bytes long. This is the unit of interaction of the CCU with the memory system. If the selected entry has a translation field that matches the 16 bit translation obtained from the translation buffer, and if the invalid bit of that entry is not set, then the desired line is in the cache. If the line is not in the cache, then it is loaded from real memory automatically by the CCU. If the current occupant is dirty, then it is written out first. Since each cache entry is 16 bytes long, the next two virtual address bits, *w* in the diagram, select the appropriate word of the cache entry. The low order two bits of the virtual address, *b* in the diagram, select a byte within a word. These bits are ignored by the CCU and cache RAMs, which always load doublewords and store single or doublewords[1].

In kernel mode the TB lookup is omitted; this means that the virtual page number translates to the same real page number, and TB faults are never generated. In this mode all pages are writeable. Cache access occurs in the normal way.

Real addresses which result from translation consist of the 16-bit translated page number, the 12-bit line number from the virtual address, and the 2-bit word number from the virtual address for a total of 30 bits. Real addresses are word addresses. All interactions between CCUs and between a CCU and the memory controller use real addresses.

1.3 Pipelining Issues

Address translation in the MultiTitan is pipelined so that the CCU can translate one virtual address per cycle. This means that a sequence of load instructions can execute at a rate of one per cycle. Stores, however, are required to output the virtual address for two cycles, so that a sequence of store instructions can be executed at a rate of one every two cycles.

The pipelining of address translation is shown in Figure 1-2.[2]When a memory reference instruction is in the instruction pipeline, the virtual address emerges from the CPU at the beginning of Phase 1 of the MEM stage. The remainder of this cycle is used for a RAM read, even in the case of a store instruction. The data must be valid at the CPU pins by the end of Phase 4. However, the detection of a cache miss can be delayed until Phase 3 of the next cycle; in the diagram, hit/miss detection corresponds to the value of the loadWB signal. So when a memory reference misses in the cache, the pipeline has already advanced and the instruction stalls in the WB stage, not the MEM stage. This means that the cycle time of the processor can be less than the cache hit/miss decision time, and equal to the cache data access time.

---

[1]Although the logical organization of the external cache has a 128-bit data line as shown in figure 1-1, it is physically implemented as two 64-bit data lines. This is because the per-processor data bus is only 64 bits wide, so all 128 bits of data are never needed simultaneously. So, the most significant bit of the *word* field is also used in addressing the data cache RAMs (but not the tag cache RAMs, as there is only one tag per pair of doublewords). This means that the data RAMs are addressed with 14 bits, leading to full utilization of 16 KBit * 4 SRAMs.

[2]For the timing of inter-chip signals, see the MultiTitan Intra-Processor Bus Specification.

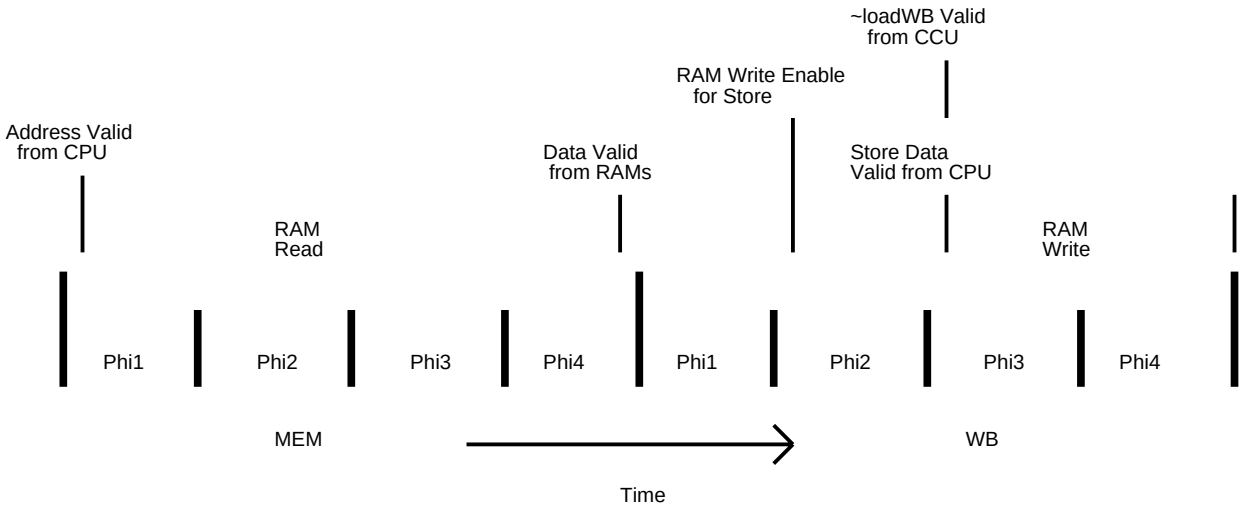


Figure 1-2: Address Translation Pipeline

In the case of a store instruction, the CPU puts its data onto the data bus in Phase 3 of the WB stage, whether or not a miss is signalled in this cycle. However, there is enough time in the CCU between the computation of the hit/miss bit (in Phase 1 internally) and the time that the RAMs must be enabled to write (the CCU pin must be driven in Phase 2 because it fans out to many RAM chips) that the write can be cancelled if the cache missed.

1.4 Bootstrapping

Because the cache state and parity bits are stored in the cache RAMS, after power-up the cache will be in an unknown state. The processor must execute a special instruction sequence to initialize the caches and clear out any potential parity errors. To make this possible, the CCU program status word contains three special bits, the *bootROM*, the *checkParity* and *miss* bits. After a hardware reset, both the *bootROM* bit and the *miss* bit will be set to 1, the *checkParity* bit will be set to 0, and the CPU will begin execution at PC = 0 in kernel mode.

If the *miss* bit in the CCU status register is 1, then the CCU generates a clean miss on every cache access, whether or not the cache tags match the requested virtual address. Thus all memory references bypass the cache, and a cache line is reloaded on each access. This behavior allows the CPU to clear the instruction cache by executing a page of instructions, and the data cache by loading one value per cache line. Both caches can be cleared simultaneously by executing a page of load instructions. Once every cache line has been reloaded, the *miss* bit should be cleared. Note that while the *miss* bit is set, stores instructions will have no effect, so no useful computation can be done.

The *bootROM* bit in the CCU status register is directly connected to an external CCU pin. On processor number 0, this pin is connected to the memory controller signal of the same name. It is unconnected for the other processors. This bit serves the same function as on the Titan, namely to cause the memory controller to read from ROM in low real addresses. It can be used in the same way to copy a bootstrap program from ROM to RAM. See the Titan System Reference Manual for more details.

The translation buffer also is in a random state after a reset, and must be initialized. This is simpler than initializing the cache RAMs, and no special hardware is required. Since the CPU is in kernel mode after a reset with address translation disabled, it can invalidate translation buffer entries in the normal way.

This is the cold-start sequence. There is also hardware provision for restarting after a serious software error. In this case, the current contents of caches, the TB and memory are valid, and should be saved in order that a dump can be taken. The difference between a "cold" and a "warm" start is the *save* signal, which is sent to each CPU and CCU. In the CCU, this bit controls the setting of the *bootROM* and *checkParity* bits in the PSW. When *save* is 0, then *bootROM* and *miss* are both set to 1 during reset. If *save* is 1, then they are set to 0, and the CPU can execute code to flush its register, cache, and TB contents back to memory.

2. Instruction Set

The CPU and its coprocessors communicate by means of the coprocessor registers, a set of 64 registers divided between the FPU (with 51) and the CCU (with 8). In the FPU, these coprocessor registers are floating point or timer registers. In the CCU, some of the coprocessor registers are actual data registers, and some are pseudo-registers which are only useful for the side effects which occur when instructions are issued on them. These pseudo-registers are the mechanism by which the special CCU functions are implemented.

The CPU can issue two kinds of instruction referring to coprocessor instructions; data transfer instructions and ALU instructions. Formats and operation of these instructions are described in detail in the MultiTitan CPU Specification.

The CCU only interprets coprocessor data transfer instructions on its registers, and the result of issuing a coprocessor ALU operation in which any register is a CCU register is undefined. In addition, for some of the pseudo-registers, only certain of the data transfer operations have defined results.

The following figure shows which instructions are implemented for which registers. Instructions marked with an asterisk cause a privilege fault when attempted in user mode. The results of attempting an undefined instruction are undefined.

Reg	Name	CopLoad	CopStore	CpuToCop	CopToCpu
56	TBTag	ReadTBTag*	WriteTBTag*	LoadTBTag*	StoreTBTag
57	TBData	ReadTBData*	WriteTBData*	LoadTBData*	StoreTBData
58	IO	StartIO*	GetCtl*	LoadIO*	
59	Fault		WriteBack		StoreFault
60	PSW		Flush	LoadPSW*	
61	TAS	TestAndSet	Clear	LoadTAS*	StoreTAS
62	Remote	Send	Receive		

Figure 2-1: CCU Instructions

In the following instruction descriptions, all addresses are virtual, and all translation occurs as described in the Address Translation section. Wherever a virtual address is used, it is subject to TB faults, write protect faults and cache misses as described above.

2.1 TBTag Register

The TBTag Register is a register through which translation tags can be read and written. The TBTag register has the format shown below:

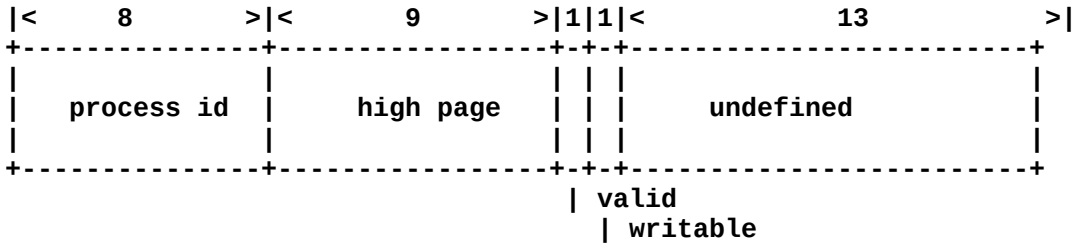


Figure 2-2: Translation Buffer Tag Register Format

process id is the identity of the process for which the translation is being inserted, and *high page* is the most significant part of the virtual address. If *valid* is 1 then this is a valid translation. If *writable* is 1 then user-mode stores into the page are permitted.

2.1.1 LoadTBTag

TASM: ccu_tbtag := rr;
Transfers the contents of the CPU register to the TBTag register. Executable only in kernel mode.

2.1.2 StoreTBTag

TASM: rr := ccu_tbtag;
Transfers the current contents of the TBTag register to the CPU register. ReadTBTag can be used before this instruction to load a translation tag into the TBTag register. Executable only in kernel mode.

2.1.3 ReadTBTag

TASM: ccu_tbtag := (disp[ra]);
Writes a translation tag from the translation buffer into the TBTag register. The virtual address (disp[ra]) is used to select a translation buffer entry according to the following format, and the tag of the selected entry is read into the TBTag register.

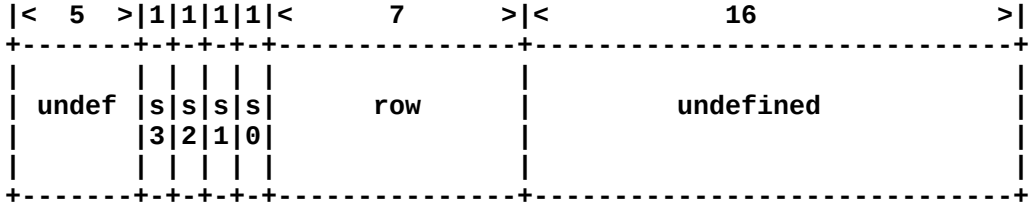


Figure 2-3: Interpretation of a Virtual Address as a TB entry

The row bits of the virtual address is used with the current PID to select a line in the translation buffer as described in the Address Translation section. Bits s0 through s3 of the address select one of the four cache lines at this TB index. Set *i* is selected if bit s*i* is 1. For a ReadTBTag operation, exactly one of the s*i* must be 1.

Executable only in kernel mode.

2.1.4 WriteTBTag

TASM: (disp[ra]) := ccu_tbtag;
Writes the contents of the TBTag register into the the tag part of a translation buffer entry. The entry modified is selected by the virtual address as defined in figure 2-3. All sets for which the set bit is one are written. This can be used to initialize all sets at a particular row simultaneously. Executable only in kernel mode.

Example

```
write a translation buffer tag value
    ccu_tbtag := r1;        { transfer tag value to CCU }
    (0[r2]) := ccu_tbtag;   { write into entry defined r2 }

read a translation buffer tag value
    ccu_tbtag := (0[r2]);  { load with entry defined by r2 }
    r1 := ccu_tbtag;       { and transfer to CPU }
```

2.2 TBData Register

The TBData Register is a register through which the data parts of translation buffer entries can be read and written. The TBData register has the format shown below:

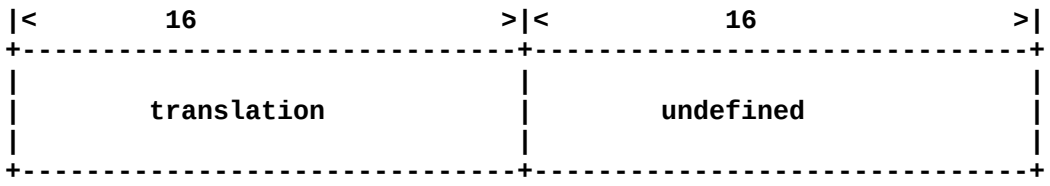


Figure 2-4: Translation Buffer Data Register Format

The *translation* is the real page number which is the translation for virtual pages matching this translation buffer entry. See the Address Translation section for more details.

2.2.1 LoadTBData

TASM: ccu_tbdata := rr;

Transfers the contents of the CPU register to the TBData register. Executable only in kernel mode.

2.2.2 StoreTBData

TASM: rr := ccu_tbdata;

Transfers the current contents of the TBData register to the CPU register. ReadTBData can be used before this instruction to load a translation data value into the TBData register. Executable only in kernel mode.

2.2.3 ReadTBData

TASM: ccu_tbdata := (disp[ra]);

Writes a translation data value from the translation buffer into the TBData register. The translation buffer entry read is defined by the virtual address (disp[ra]) as shown in figure 2-3. Exactly one of the set select bits in the virtual address must be 1. The data of the selected entry are written into the TBData register. Executable only in kernel mode.

2.2.4 WriteTBData

TASM: (disp[ra]) := ccu_tbdata;

Writes the contents of the TBData register into the the data part of a translation buffer entry. The entry modified is selected by the virtual address as defined in figure 2-3. Each set for which the set select bit is 1 is updated, which can be used to initialize several entries simultaneously.Executable only in kernel mode.

Example

```
write a translation buffer data value
    ccu_tbdata := r1;        { transfer data value to CCU }
    (0[r2]) := ccu_tbdata;   { write it into entry defined by r2 }

read a translation buffer data value
    ccu_tbdata := (0[r2]);  { load with entry defined by r2 }
    r1 := ccu_tbdata;       { and transfer to CPU }
```

2.3 I/O Register

The I/O Register contains control information for the next I/O operation for the memory controller. It is write-onely and can be changed by the CPU only in kernel mode. StartIO and GetCtl use the I/O Register contents to perform I/O operations. The I/O Register has the format shown below:

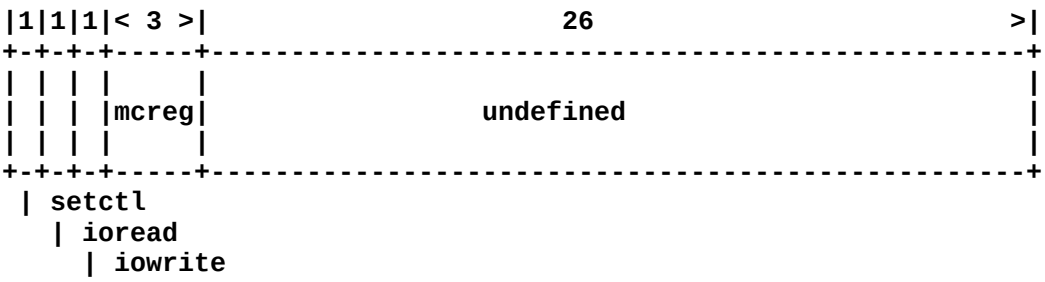


Figure 2-5: IO Control Word Format

- *setctl, ioread, iowrite* determine the operation to be performed.
- *mcreg* is the selected memory controller register for GetCtl and SetCtl operations.

2.3.1 LoadIO

TASM: ccu_io := ra;

This instruction loads the I/O register with a new value. It has no side effects. Executable in kernel mode only.

2.3.2 StartIO

TASM: ccu_io := (disp[ra]);

This instruction performs an IORead, IOWrite, or SetCtl operation with the memory controller. Disp[ra] must be doubleword-aligned, and the word at disp[ra] is used as the data word for the I/O operation. Executable only in kernel mode.

Exactly one of *setctl*, *ioread* and *iowrite* must be 1 or the results of the operation are undefined.

- If the *ioread* bit is set, the value at disp[ra] is used as the address in an IORead operation. This address is sent to the I/O device currently selected in the memory controller status register, and the device responds with a 32-bit data value which is written to the memory controller ioRData register.
- If the *iowrite* bit is set, the value at the word-aligned address disp[ra] is used as the address in an IOWrite operation. This value, along with the 32-bit data value in the memory controller ioWData register, are sent to the I/O device currently selected by the memory controller status register.
- If the *setctl* bit is set, the value at the word-aligned address disp[ra] is written to the memory controller register whose number was in *mcreg* of the CCU I/O register when the **previous** StartIO or GetCtl operation was issued.

2.3.3 GetCtl

TASM: (disp[ra]) := ccu_io;

None of *setctl*, *ioread* or *iowrite* may be set in the I/O register.

The instruction stores the value of a memory controller register in disp[ra], which must be doubleword-aligned. The register is the one whose number was in the *mcreg* field of the I/O register at the time of the **previous** StartIO or GetCtl operation. Executable only in kernel mode.

Example

```
I/O read operation:

ccu_io := r1;            { select getctl, status register }
(0[r2]) := ccu_io;       { null GetCtl to select status register }
ccu_io := r3;            { select setctl, status register }
ccu_io := 8[r2];         { setctl to write the status register }
ccu_io := r4;            { select ioread and status register }
ccu_io := 16[r2];        { StartIO }
ccu_io := r1;            { select getctl, status register }
{ memory controller busy for 3 cycles; next GetCtl will stall }
0[r2] := ccu_io;         { GetCtl to read status register }
r6 := 0[r2];             { load status register }
if r6 < then goto done;  { test whether device heard us }
```

2.4 Fault Register

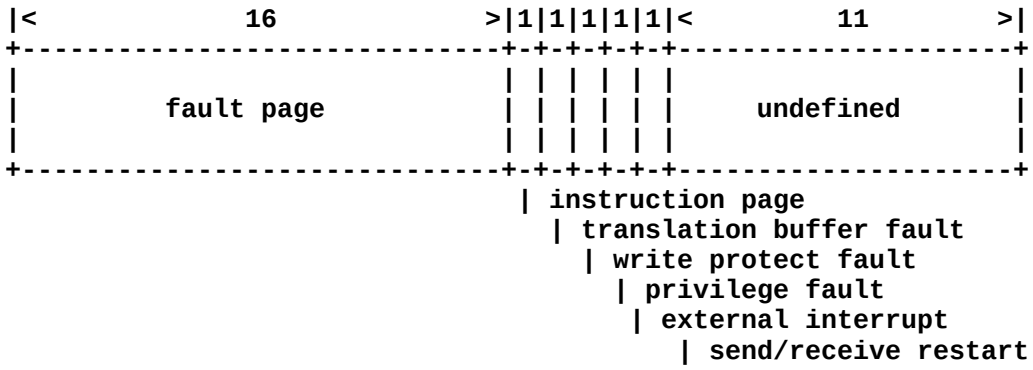


Figure 2-6: CCU Fault Register Format

The fault register records the reason for the most recent CCU fault (if any). The register is read-only by the CPU, and its value is undefined in user mode. It is updated in every user-mode cycle, is held in kernel mode, and is cleared during the kernel exit sequence. In user mode, the CCU interrupts the CPU if any of the four fault bits in the fault register is 1. In kernel mode, the fault register records the state of the last user-mode instruction.

A 1 in a fault bit position means that the fault occurred at the time of the most recent entry into kernel mode. Several of the bits may be set simultaneously.

The possible fault reasons are:

- *translation buffer fault*: an instruction or data reference generated a virtual address for which no translation existed in the CCU's translation buffer. The virtual page for which there was no translation is *fault page*. If *instruction page* is 1 then the instruction currently in the IF stage of the pipeline caused an instruction address translation fault. If the *instruction page* bit is 0, then the instruction in the WB stage of the pipeline caused a data translation fault. If the instruction is a CPU or coprocessor load, then an incorrect result has been written to the destination register. For this reason, CPU loads which

overwrite their base registers must not be generated by software. If the instruction is a CPU or coprocessor store, no cache locations have been modified. For both loads and stores, therefore, software can insert a translation for the missing data page, and restart the WB-stage instruction.

- *privilege fault*: a privileged CCU instruction was attempted in user mode by the WB-stage instruction.
- *external interrupt*: some other processor or the memory controller has requested an interrupt.
- *write protect fault*: attempt to store into a read-only page by the WB-stage instruction. No cache data have been modified. This bit is undefined if *translation buffer fault* is 1.

The *fault page* is undefined when the *translation buffer fault* bit and the *write protect fault* bits are both zero. Otherwise, it defines the virtual page which caused the translation buffer interrupt.

The *send/receive restart* bit is set when a send or receive instruction which has not completed is in the WB-stage of the pipeline. This bit is set when an external interrupt occurs while a send or receive is waiting for a partner to arrive. If it is set, then the WB-stage instruction must be restarted on return from the interrupt. See the interrupt section for more details.

2.4.1 StoreFault

TASM: rr := ccu_fault;
Sets the CPU register to the current value of the fault register.

2.4.2 WriteBack

TASM: (disp[ra]) := ccu_writeback; or (disp[ra]) := ccu_psw;
This operation is provided to allow software to selectively force data in the cache back to memory.

The virtual address disp[ra] is translated and looked up in the cache. If it is not present, or is present but not dirty, the operation has no effect. If the line is present and dirty, the CCU stalls the pipeline, writes the line back to memory, and marks the line clean but valid.

2.5 PSW Register

The CCU Program Status Word Register contains state information for the CCU. It is write-only, and can be changed by the CPU only in kernel mode.

The PSW Register has the following format:

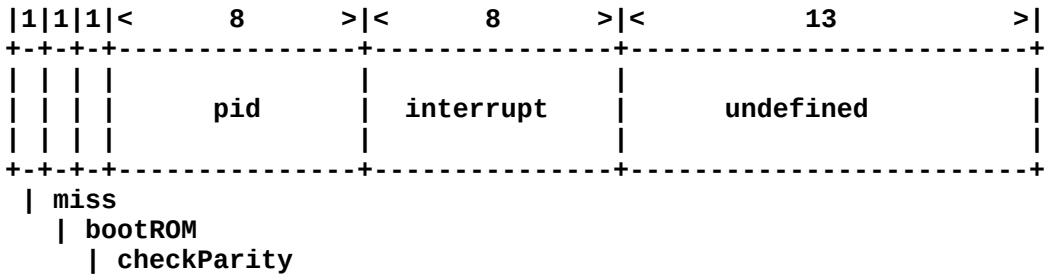


Figure 2-7: CCU PSW Register Format

The *miss* bit selects whether the cache should miss on each reference (miss = 1) or should operate normally. See the

Bootstrapping section above for more details.

The *bootROM* bit selects whether the memory controller should read from ROM in the low-numbered addresses, or whether it should read from RAM. It is connected only for processor number 0, and has no effect for other processors. See the Bootstrapping section above for more details.

The *checkParity* bit selects whether the CPU, CCU, and FPU should check parity on the per-processor data bus and on the cache RAMs. If this bit is 1, parity is checked. If it is zero, parity is not checked. Parity in the CCU is checked on TB and cache entries. In either case, invalid parity is converted into an invalid entry. So, invalid parity on a TB entry will cause a translation buffer fault. Invalid parity on a cache line will cause a cache miss.

The *pid* is the number of the currently running process, which is used in all address translations and in TB reads and writes.

The *interrupt* bits generate external interrupts for other processors. The ms (left) bit is for processor 7 and the ls (right) bit is for processor 0. Each interrupt bit which is 1 generates an external interrupt for the corresponding processor, which will be seen in that processor in the external interrupt bit of the fault register. A processor may set its own interrupt bit. Detection of when the destination processor has taken the interrupt must be by software convention.

2.5.1 LoadPSW

TASM: ccu_psw := rr;
Transfers the CPU register to the PSW register. Executable only in kernel mode.

2.5.2 Flush

TASM: (disp[ra]) := ccu_flush; or (disp[ra]) := ccu_psw;
This operation is provided to allow software to selectively remove data in the cache.

The virtual address disp[ra] is translated and looked up in the cache. If it is not present, the operation has no effect. If it is present but not dirty, the line is invalidated in a single cycle. If the line is present and dirty, the CCU stalls the pipeline, writes the line back to memory, and marks the line invalid.

2.6 Test-And-Set Register

The *TAS* register stores the result of the most recent test-and-set operation. After a test-and-set, the sign bit of this register is 0 if the lock has been claimed, and 1 if attempt to claim the lock failed.

The Test-And-Set Register has the format shown below:

2.6.1 LoadTAS

TASM: ccu_tas := rr;
Transfers the CPU register to the TAS register. Executable only in kernel mode.

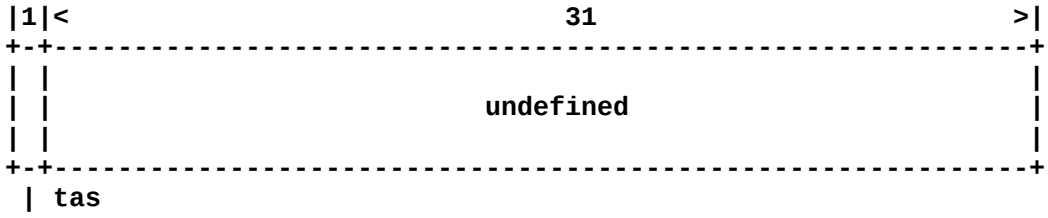


Figure 2-8: Test-And-Set Register Format

2.6.2 StoreTAS

TASM: rr := ccu_tas;
Transfers the TAS register to the CPU register. Executable in user mode.

2.6.3 TestAndSet

TASM: ccu_tas := (disp[ra]);
The word-aligned virtual address disp[ra] defines a data cache line on which a test-and-set is performed. There must be a valid TB translation for the line, or an interrupt is generated. The operation of test-and-set is independent of whether the line is present in the cache or not; this instruction operates only on the quadword in main storage, and does not read or alter a cached copy of any line.

In an atomic memory transaction, a fixed pattern is written to the real memory quadword, and its old value is read into the TAS register. The fixed pattern written has a "1" in the lock bit, which is the sign bit of the doubleword selected by disp[ra]. The remaining 127 bits of the line are undefined, and should not be used by software. Software convention must ensure that all test-and-set operations on a given line use the same real address of the sixteen possible choices.

The sign bit of the TAS register is set to the old value of the lock bit in the line in memory. Thus it is 0 if the lock bit was 0 but is now 1 (and the lock has been claimed) and 1 if the lock bit was 1 (and the lock has not been claimed). To clear a lock, a 1 should be written to the lock bit and flushed to memory.

Example

```
claim a lock:
     ccu_tas := (lock);       {test and set; no flush needed}
     r2 := ccu_tas;           {get result of test and set}
     if r2 >= then goto locked;

release a lock:
     r1 := 0;             { lock clear when lock bit = 0 }
     (lock) := r1;        { clear lock bit }
     ccu_flush := (lock); {flush cache line back to memory}
```

2.6.4 Clear

TASM: (disp[ra]) := ccu_clear; or (disp[ra]) := ccu_tas;
The virtual address disp[ra] defines a quadword which is mapped into the data cache and marked clean. If the same quadword is already mapped, or if a different quadword is mapped and clean, then the operation takes a single cycle. If a different quadword is mapped and dirty, it is written back to memory first. In user mode, an interrupt is generated if there is no TB translation for this address, or if it is not writeable. After execution of this instruction, reading data from this line will return undefined results.

2.7 Remote Register

The Remote Register is a pseudo-register which can neither be read nor written. It is useful only for the side effects of Send and Receive which are encoded as load and store on this register.

2.7.1 Send

TASM: ccu_remote := (disp[ra]);
This instruction stalls the processor until a quadword has been sent to some other processor.

The virtual address disp[ra] defines the cache line which is to be send. The address need not be quadword-aligned. The address must have a valid translation in the TB, and be writeable, or an interrupt is generated. The CCU stalls its CPU, and if the line is not present in the cache, it is read from memory in the normal way. The CCU now repeatedly broadcasts the translated doubleword address on the system bus. After each broadcast in which no receiver signals a match, the CCU waits for the arrival of a new receiver before retrying. Matching is done after translation on real doubleword addresses, not virtual byte addresses. More than one CCU may receive the line, but no indication will be given as to how many receivers there were, nor which processors received the line. On completion, the line will have been copied to the receiving caches, and will be marked invalid in the sending cache. While the CCU is waiting in user mode for a new receiver to arrive the processor can be interrupted.

2.7.2 Receive

TASM: (disp[ra]) := ccu_remote;
This instruction stalls the processor until a quadword has been received from some other processor

The quadword-aligned virtual address disp[ra] defines the cache line which is to be received. The address must have a valid translation and be writeable, or an interrupt is generated. If the cache line is not mapped in the cache, then the equivalent of Clear on this line occurs (the current occupant is written to memory if necessary, and the line is mapped; if the line is already mapped, clean or dirty, nothing happens). On beginning this instruction, the CCU deasserts the *~receive* line on the shared system bus for one cycle to indicate to other CCUs executing Send that a new listener has arrived. The CCU executing Receive now stalls its CPU and waits for Send broadcasts to occur on the system bus. Each time one occurs, it matches the translated real doubleword address from the Receive instruction with the address on the system bus. Matching is done after translation on real doubleword addresses, not virtual byte addresses, and translation happens in the standard way as described above. When a match occurs, an entire quadword line will be copied into the receiving cache. On completion, the received line will be marked dirty. While the CCU is waiting in user mode for a sender to transmit, the processor can be interrupted.

3. Interrupts

For the various reasons defined in the Fault Register section, the CCU may interrupt the processor pipeline. After the interrupt is dealt with, the operating system software must decide which of the four available PC's in the user pipeline should be restarted. The default condition is that the WB-stage instruction has completed, and should not be restarted. The CCU defines the following exceptions to this rule by the state of the Fault Register:

- if the *translation buffer fault* bit is set, and the *instruction page* bit is not set, the WB-stage data reference did not complete because of a missing address translation. The instruction should be restarted.
- if the *write protect* bit is set, the WB-stage store did not take effect. If the user program is resumed, the WB-stage instruction must be restarted.
- if the *external interrupt* bit is set and the *send/receive restart* bit is set, then a send or receive instruction in the WB pipeline stage was interrupted by an external processor interrupt before the cache-to-cache transfer had occurred. It must be restarted on return to the user program.

Since several interrupt conditions may be simultaneously presen (for instance, a send instruction may provoke a write protect violation simultaneously with an external interrupt), the WB-stage instruction should be restarted if any of the above three conditions is true. In all other circumstances, the WB-stage instruction should not be restarted.

4. Instruction Timing

The following table shows the time taken by the CCU special functions in the absence of other pipeline stalls and memory system contention.

Operation	Cycles	
Cache Miss	14	whether cache line is clean or dirty
ReadTBTag	1	
WriteTBTag	1	
LoadTBTag	1	
StoreTBTag	1	
ReadTBData	1	
WriteTBData	1	
LoadTBData	1	
StoreTBData	1	
StartIO	3	
GetCtl	3	
LoadIO	1	
StoreFault	1	
Flush	1	if line is not dirty
	8	if line is dirty
Clear	1	same line or other clean line is mapped
	8	if different dirty line is mapped
LoadPSW	1	
TestAndSet	14	
LoadTAS	1	
StoreTAS	1	
Send	9	if receiver is waiting
Receive	12	if sender is waiting

5. Cache Control Unit Organization

Figure 5-1 shows the floorplan of the CCU. Most of the chip is occupied by the Translation Buffer, which has about the same storage capacity as the CPU Instruction Buffer, and is similarly organized. The remainder of the translation data path is below the translation buffer. The CCU registers are at the lower left, and the lower part of the chip is occupied by three state machines which sequence interactions with the memory controller and other CCU's.

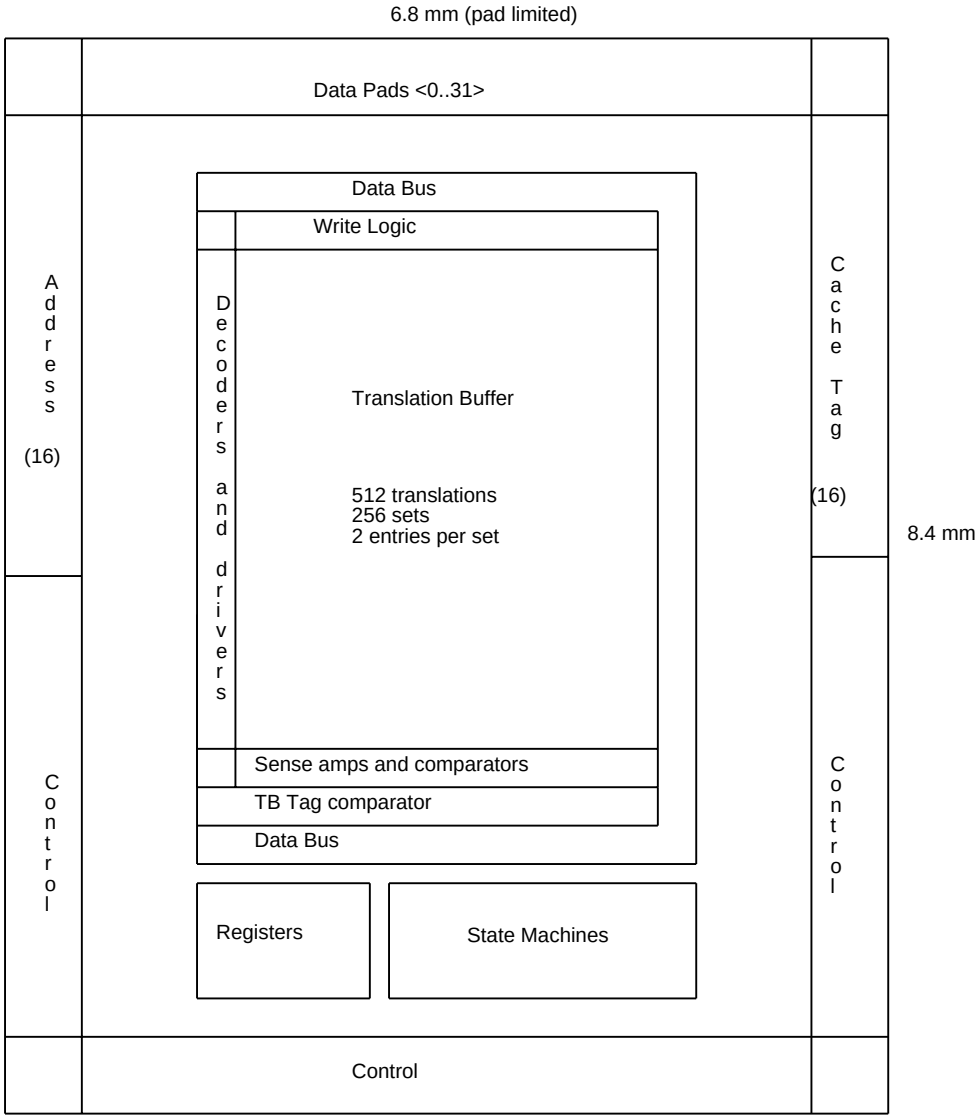


Figure 5-1: CCU Floor Plan

6. Cache Control Unit Pins

76		localBus	per-processor bus
32	IO	bus	ms half of the 64-bit localBus
4	IO	parity	byte parity
16	I	address	page number from the virtual address
1	I	insAddr	address is PC
1	O	altAddr	address other half of cache line
1	I	wordInPair	word bit from the virtual address
1	I	reset	global reset
1	I	save	save processor state to memory on reset
1	I	clock	master clock
1	IO	interrupt	coprocessor interrupt for CPU
1	IO	allowInterrupt	interrupts are allowed now
1	IO	loadwb	load wb stage
1	O	loadmem	load mem stage
1	I	loadalu	load alu stage
1	I	loadif	load ifetch stage
1	I	kernel	CPU in kernel mode
4	I	opCode	instruction opcode
6	I	register	instruction register
1	O	checkParity	whether bus parity should be checked or not
20		racTag	real address cache tag
16	IO	realPage	real page number
2	I	parity	real page parity
1	IO	valid	cache line valid
1	IO	dirty	cache line dirty
4		racCtl	controls for real address cache
2	O	~dataCs	don’t enable cache data
1	O	~we	don’t write data and tags
1	O	~tagCs	don’t enable line tags
13		globalCtl	controls for global interprocessor bus
1	O	busReq	request to use global bus
1	I	busMaster	bus master
8	IO	~interrupt	no interrupt from other processors
1	IO	send	CCU starting send broadcast
1	IO	~receive	no CCU starting receive instruction
1	IO	~match	no send/receive address match
10		memCtl.toMem	controls to memory controller
1	O	memRead	start read request
1	O	memWrite	start write request
1	O	specialOp	do a non-memory operation
1	O	ioRead	do an I/O read operation
1	O	ioWrite	do an I/O write operation
1	O	setCtlReg	do a SetCtl operation
3	O	ctlReg	memory controller register to select
1	O	bootROM	memory controller reads from ROM
3		memCtl.fromMem	controls from memory controller
1	I	memRdy	ready for a new memory request
1	I	ioRdy	ready for a new I/O request
1	I	dataRdy	first data word ready
13		intfCtl	TTL interface chip controls
2	O	fromMemCk	clocks for 374 fromMem register
1	O	~fromMemOE	output enable for 374 fromMem register
1	O	globalAddrCk	clock 646 address register
1	O	~globalPageOE	output enable page register
1	O	~halfLineOE	output enable for halfline register

```
     2  O       globalDataCk   clock globalBus register
     2  O       ~globalDataOE  output enable for globalBus reg
     1  O       outward        direction of transfer in globalBus reg
     2  I       lineMatch      receive/send octal address comparators.
---------
139   total

in a 176-pin package with 144 signal pins
```

Table of Contents

List of Figures

MultiTitan Local & Global Bus Definition and Timing

David Boggs
Jeremy Dion
Michael J.K. Nielsen
Digital Equipment Corporation
Western Research Laboratory
100 Hamilton Avenue
Palo Alto, CA 94301

Version of 7 April 1988

1. Local Bus

The MultiTitan Local bus is the group of signals that interconnect the CPU, FPU, CCU, cache rams, and memory interface of one processor in a MultiTitan; see figure 1-1.

1.1 Pipeline Stages

The CPU has a four-stage instruction pipeline. The pipeline stages affect the local bus signals in the following ways:

IF Instruction fetch stage. The CPU uses the PC to read the on-chip instruction buffer, decodes the register file addresses, and reads two register file operands. The instruction's opcode and one of its register numbers are driven onto the Op and Reg buses; its PSW.kernel bit is driven to the Kernel signal.

ALU Arithmetic and logical unit stage. Performs arithmetic operations, logical operations, shift operations, branch comparisons, and computes branch PCs. Instructions in this stage have no external effects.

MEM Memory operation stage. A load- or store-class instruction drives its effective address onto the Address bus. An instruction buffer load (as opposed to an operand load or store) asserts InsAddr. Cached memory data must return by the end of this cycle on the Data bus, or else WB must be held in the next cycle. Coprocessor ALU instructions are themselves driven onto the Address bus.

WB Result write-back stage. Arithmetic, logical, and shift instructions write results into the register file but have no external effects. Load instructions write the register file with memory data latched at the end of the last cycle. Store instructions read the register file and drive memory data onto the Data bus. Trap instructions assert ~Interrupt here.

1.2 Pipeline Control

The pipeline control signals coordinate the pipelines inside the CPU, FPU, and CCU on a cycle-by-cycle basis. The signals are driven by one or more of the chips and received by all.

At the end of each cycle, each pipe stage does one of three things:

- **load** the stage, so that in the next cycle, the instruction from the previous stage will be in this stage.
- **hold** the stage, so that in the next cycle, the same instruction will be in this stage.
- **kill** the stage, so that in the next cycle, a null instruction will be in this stage.

Each chip internally decodes the pipeline control signals every cycle to decide on one of these three actions for each of its pipeline stages as shown in Figure 1-3.

In normal operation, all pipeline stages **load** on each cycle, so a new instruction starts down the pipeline on every cycle. However, due to interactions between consecutive instructions in the pipeline, or to external causes such as cache faults, delays must sometimes be injected at some point in the pipeline. When this happens, the pipeline breaks into three parts: upstream of the *bubble*, all stages **hold**; at the stage where the bubble is to be injected, that stage **kills**; downstream of the bubble, all stages **load**. By holding upstream stages, and by advancing downstream stages, a **killed** stage can be created at any point in the pipeline. These actions are encoded in the four control signals: LoadIF, LoadALU, LoadMEM and LoadWB.

Resets and interrupts kill all pipe stages and jump to low memory. In the absence of resets and interrupts, the four

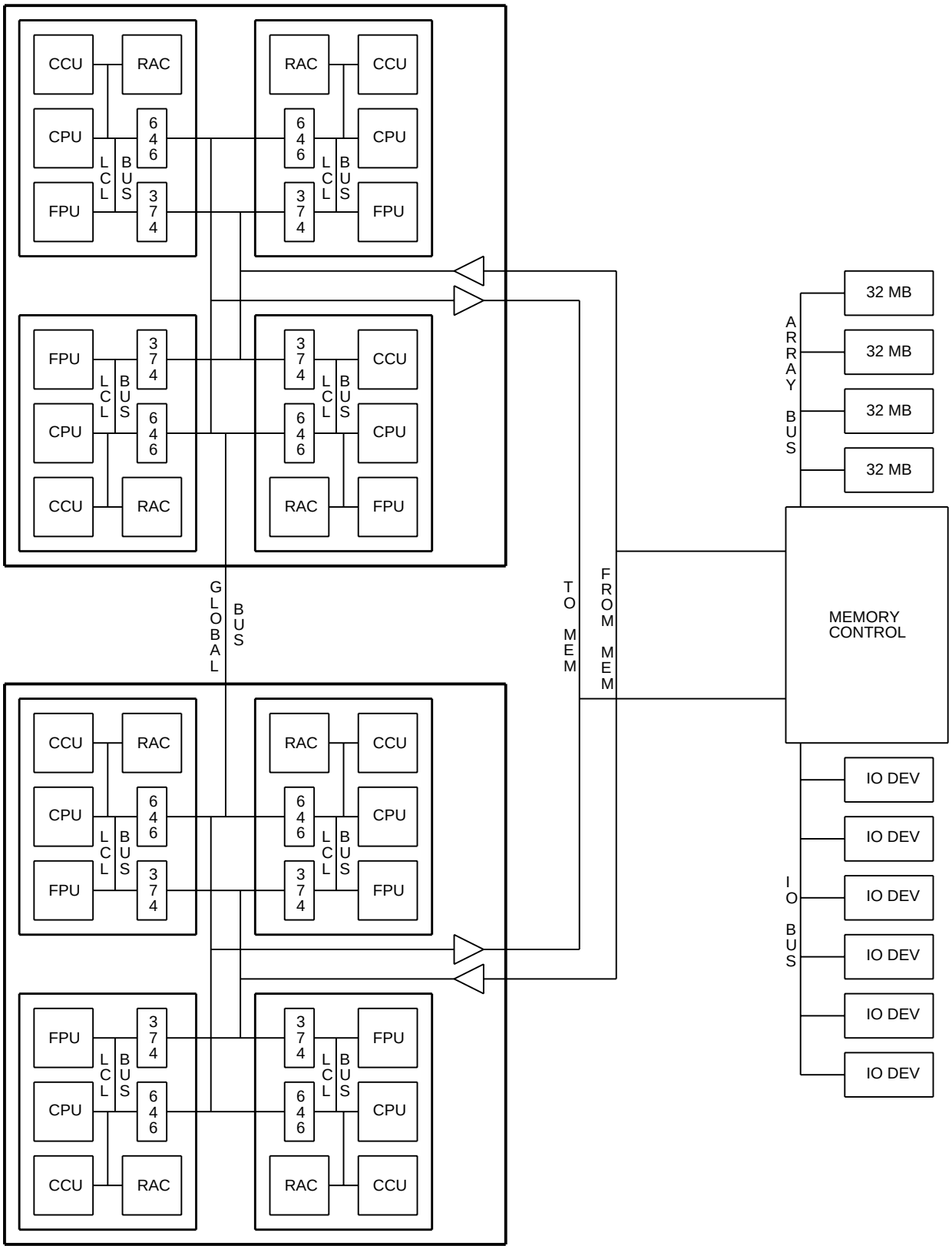


Figure 1-1: System Buses

LoadXXX signals are combined to produce an appropriate action for each stage. For $stage_i$, the appropriate action can be determined by examining $Load_i$ and $Load_{i-1}$, the signal for the next stage upstream.

$Load_{i-1}$	$Load_i$	Action for $stage_i$
F	F	hold
F	T	kill
T	F	(illegal)
T	T	load

Figure 1-2: Generation of pipeline actions from load signals

Note that there is one illegal state: an instruction leaves $stage_{i-1}$ but isn't loaded into $stage_i$. Whenever $Load_{i-1}$ is true, $Load_i$ must also be true. This simplifies decoding the LoadXXX signals into pipeline actions. Figure 1-3 shows how to generate the kill, hold, and load actions for a stage from the pipeline control signals.

```
Let Abort = Reset + (AllowInt * Interrupt).

Abort LoadIF LoadALU LoadMEM LoadWB |  IF  ALU  MEM   WB   cause
-----------------------------------|---------------------------------
  T      X      X       X       X   | kill kill kill kill  interrupt
  F      F      F       F       F   | hold hold hold hold  cache miss
  F      F      F       F       T   | hold hold hold kill  miss retire
  F      F      F       T       T   | hold hold kill load  IBuff miss
  F      F      T       T       T   | hold kill load load  interlock
  F      T      T       T       T   | load load load load  normal case

ALU:
    kill =  Abort +  LoadALU * ~LoadIF
    hold = ~Abort * ~LoadALU
    load = ~Abort *  LoadIF

MEM:
    kill =  Abort +  LoadMEM * ~LoadALU
    hold = ~Abort * ~LoadMEM
    load = ~Abort *  LoadALU

WB:
    kill =  Abort +  LoadWB  * ~LoadMEM
    hold = ~Abort * ~LoadWB
    load = ~Abort *  LoadMEM
```

Figure 1-3: Next States from Pipeline Control Signals

The truth table above summarizes the normal combinations of the pipeline control signals. When an interrupt happens, all pipe stages kill. When an external cache misses, the CCU holds the entire pipeline until data is available from memory. At the end of a cache miss sequence, the CCU sometimes has to kill WB. During an instruction buffer miss sequence, the CPU kills MEM for several reasons. When an instruction interlock is detected, the CPU separates the instructions by killing ALU. In the normal case, all pipe stages load and the entire pipeline advances.

Note that the Kernel bit is not in the next state equations. Kernel is driven by the instruction in IF and must be forwarded through WB, during which time it prevents assertion of ~Interrupt. The CCU may deassert AllowInt in the same phase that the CPU or FPU may assert ~Interrupt. Therefore, each stage of each chip must look at AllowInt to decide whether or not to believe the ~Interrupt signal.

1.3 Memory System Issues

The IF stage drives the Kernel signal and the Op and Reg buses. The ALU stage does not drive any external signals. The MEM stage drives the Address bus and receives the Data bus. The WB stage drives the Address and Data buses.

MicroTitan's sixteen opcodes can be divided into memory- and nonmemory-class instructions, and the memory class can be further divided into load- and store-class instructions. When a memory-class instruction is in the MEM pipe stage, it drives the Address bus and receives the Data bus. Additionally, a store-class memory instruction drives the Address and Data buses when it is in the WB stage. A store instruction reads the target location during MEM and then writes it during WB. The cache drives the "old" data onto the Data bus during MEM and the CPU drives the "new" data onto the bus during WB. Trap, both Extracts, CPU ALU, Conditional Jump, Add Immediate, and the undefined opcodes are non-memory class instructions. Cop to CPU transfer, Cop Load, CPU Load, and Cop ALU are load-class memory instructions. CPU to Cop transfer, Cop Store, and CPU Store are store-class memory instructions. Cop ALU is a load-class instruction because it uses the Address bus during MEM to transfer itself to a coprocessor.

A data reference starts when a memory-class instruction advances to MEM. An instruction fetch starts when InsAddr is asserted; they come in pairs and the effective address is the IF-stage PC both times. The missed half-line is written first; in the next cycle the CCU asserts AltAddr to invert Address[3] during the second write. (Actually, the CCU detects the start of an IBuff sequence when MEM is killed and WB doesn't contain a store instruction; this happens one phase before InsAddr.)

Figure 1-4 shows what happens when a memory-class instruction follows a store-class instruction in the executing program. A store-class instruction drives the Address bus in MEM and WB and a memory-class instruction drives the Address bus in MEM. Their uses of the Address (and Data) buses conflict and so the CPU inserts a null instruction between them by killing ALU. The second memory instruction stays in IF, a null instruction materializes in ALU and the store instruction advances to MEM.

```
IF        store  load   load    --     --    --
ALU        --    store  null   load    --    --
MEM        --     --    store  null   load   --
WB         --     --     --    store  null  load

PipeCtl   load  killALU load   load   load  load
Address    --     --    store  store  load   --
```

Figure 1-4: Store interlock

Figure 1-5 shows the simplest case when an instruction fetch misses in the CPU's IBuff and hits in the external cache. The ALU stage holds and IF's fetch is discarded. Then the CPU inserts two load-class pseudo-instructions into MEM. These instructions drive the PC onto the Address bus while asserting InsAddr and load the IBuff half-lines from the Data bus. IF fetches the missed instruction, ins2, while WB writes it. In the next cycle, WB writes the alternate IBuff half-line while IF is fetching ins3.

```
IF        ins1  miss   miss   ins2   ins3   --    --    --
ALU        --   ins1   ins1   ins1   ins2  ins3   --    --
MEM        --    --    IBuf1  IBuf2  ins1  ins2  ins3   --
WB         --    --     --    IBuf1  IBuf2 ins1  ins2  ins3

PipeCtl   load  load killMEM killMEM load  load  load  load
Address    --    --     PC     PC    ins1  ins2  ins3   --
```

Figure 1-5: Simple IBuff miss

Figure 1-6 shows what happens when an IBuff miss happens and a store instruction is in MEM. The CPU wants to insert two IBuff-load instructions into MEM, but it must insert a null instruction first or else there will be a conflict on the Address bus between the store in WB and the first IBuff-load in MEM (like the store interlock above).

```
IF        store ins1  miss   miss    miss    ins2    --    --   --
ALU        --   store ins1   ins1    ins1    ins1   ins2   --   --
MEM        --    --   store  null    IBuf1   IBuf2  ins1  ins2  --
WB         --    --    --    store   null    IBuf1  IBuf2 ins1 ins2

PipeCtl   load  load  load  killMEM killMEM killMEM load  load load
Address    --    --   store  store    PC      PC    ins1  ins2  --
```

Figure 1-6: IBuff miss with store in MEM interlock

Figure 1-7 shows what happens when the instruction after an instruction that misses in the IBuff could possibly be in a different cache line. There are two cases when this could happen. If the instruction that missed was in the last word of a cache line, then incrementing the PC will cross a cache line boundary. If the instruction that missed follows a jump instruction, then the next PC could be anything and it is pessimistically assumed to address a different cache line. The conflict causing the interlock occurs on the IBuff address lines. IF refetches the missed instruction in the same cycle that it is being written by the first IBuff-load. In the next cycle, the second IBuff-load writes the other IBuff half-line and IF fetches the instruction after the one that missed. IF's PC drives the IBuff address lines directly, so IF must be held if it tries to cross a cache-line boundary before the second IBuff-load completes, or else the wrong cache line will be written. If IF is held, ALU must also be held (or else the address of a taken branch in ALU would be lost), so the null is inserted by killing MEM.

```
IF        jump  miss  miss    ins2    ins2    --    --    --
ALU        --   jump  jump    jump    jump   ins2   --    --
MEM        --    --   IBuf1   IBuf2   null   jump  ins2   --
WB         --    --    --     IBuf1   IBuf2  null  jump  ins2

PipeCtl   load  load killMEM killMEM killMEM load  load  load
Address    --    --    PC      PC     null   jump  ins2   --
```

Figure 1-7: IBuff miss with refill interlock

Figure 1-8 shows the simplest case when a load instruction misses in the external cache. The instruction advances to WB where the CCU holds it, stalling the entire pipe. Many cycles pass (shown by a column of *'s) during which the CCU references main memory. The requested half-line is on the Data bus during the last *'ed cycle, and the missed load instruction's destination register is written during the last held cycle

```
IF        load  ins    --    --    --   *  --    --
ALU        --   load  ins    --    --   *  --    --
MEM        --    --   load  ins   ins   * ins    --
WB         --    --    --   load  load  * load  ins
                                       *
PipeCtl   load  load  load  load  hold  * hold  load
Address    --    --   load  ins   load  * load   --
```

Figure 1-8: Simple cache miss

Figure 1-9 shows what happens when the instruction following one that misses in the external cache is also a memory-class instruction. In the cycle before releasing the pipe, the CCU kills WB. This allows the Address bus to switch back to the effective address of the load instruction after the miss (i.e. load2)

Figure 1-10 shows how the CPU forms a Local bus address. The IF-stage Program Counter is output when an IBuff miss happens. A memory-class instruction adds a register to its displacement field, forming an *effective address*,

```
IF        load1 load2  --    --    --    *  --    --    --
ALU        --   load1 load2  --    --    *  --    --    --
MEM        --    --   load1 load2 load2 * load2 load2  --
WB         --    --    --   load1 load1 * load1 null  load2
                                        *
PipeCtl   load  load  load  load  hold  * hold killWB load
Address    --    --   load1 load2 load1 * load1 load2  --
```

Figure 1-9: Cache miss followed by a memory instruction

which it outputs when it is in MEM. A store instruction continues to output its effective address when it is in WB. A load instruction held in WB because of a cache miss outputs its effective address.

The Address bus is always driven by the CPU; it is not tri-state. The low-order address bits (word within page) directly drive the cache rams (and the CPU's internal instruction buffer). The CCU can cause the CPU to invert Address[3] by asserting the AltAddr signal. A memory reference transfers 128 bits of data and this is also the cache line size. However, the memory data bus is 32 bits wide and the cache data bus is 64 bits wide. So the CCU makes two cache references to read or write a line, and the references are separated by one cycle. The "alternate" half-line is written first, then the "requested" half-line (containing the missed address) is written.

The CPU latches the Data bus at the end of phase 4 of each cycle and writes the destination (register file or IBuff) during phase 2 of the next cycle, regardless of any pipeline holds. When a cache misses, the instruction in WB is held, causing it to repeatedly write back its result. At the end of the miss sequence, when the requested half-line is on the Data bus and is being written into the cache, the CCU holds the pipe for one more cycle. During that last held cycle, the instruction in WB writes its destination once more, this time with the data latched in the previous cycle, when the requested half-line was on the bus.

To reduce di/dt and improve noise margins, the CPU drives only 32 bits (plus parity) of the 64-bit Data bus in any cycle. During a CPU to COP transfer instruction, the CPU drives the high half of the Data bus. During a CPU store instruction, Address[2] determines which half of the Data bus is driven. During a COP store instruction, all 64 bits of the Data bus are driven (by the FPU or the cache rams). During a COP to CPU transfer instruction, the FPU drives the high half of the Data bus.

1.4 Signal Definitions

Control signals are treated as boolean variables. A high-true signal is positive when it is asserted or true, and ground when it is deasserted or false. A low-true signal is ground when it is asserted or true, positive when it is deasserted or false, and has "~" as the first character of its name.

LoadIF Load the IF stage. High-true open-collector with external pull-up resistor. Driven and received by all three chips. When false (i.e. somebody grounding it), the IF-stage PC is not updated and on the next cycle the same instruction will be refetched. When true (i.e. nobody grounding it), IF's PC is either incremented or loaded from the branch address computed by the jump instruction in ALU.

LoadALU Load the ALU stage. High-true open-collector with external pull-up resistor. Driven and received by all three chips. When false (i.e. somebody grounding it), the instruction in ALU is held there on the next cycle. When true (i.e. nobody grounding it), ALU should either be killed or loaded in the next cycle from the current contents of IF, depending on LoadIF.

LoadMEM Load the MEM stage. High-true open-collector with external pull-up resistor. Driven by FPU and CCU, received by all three chips. When false (i.e. somebody grounding it), the

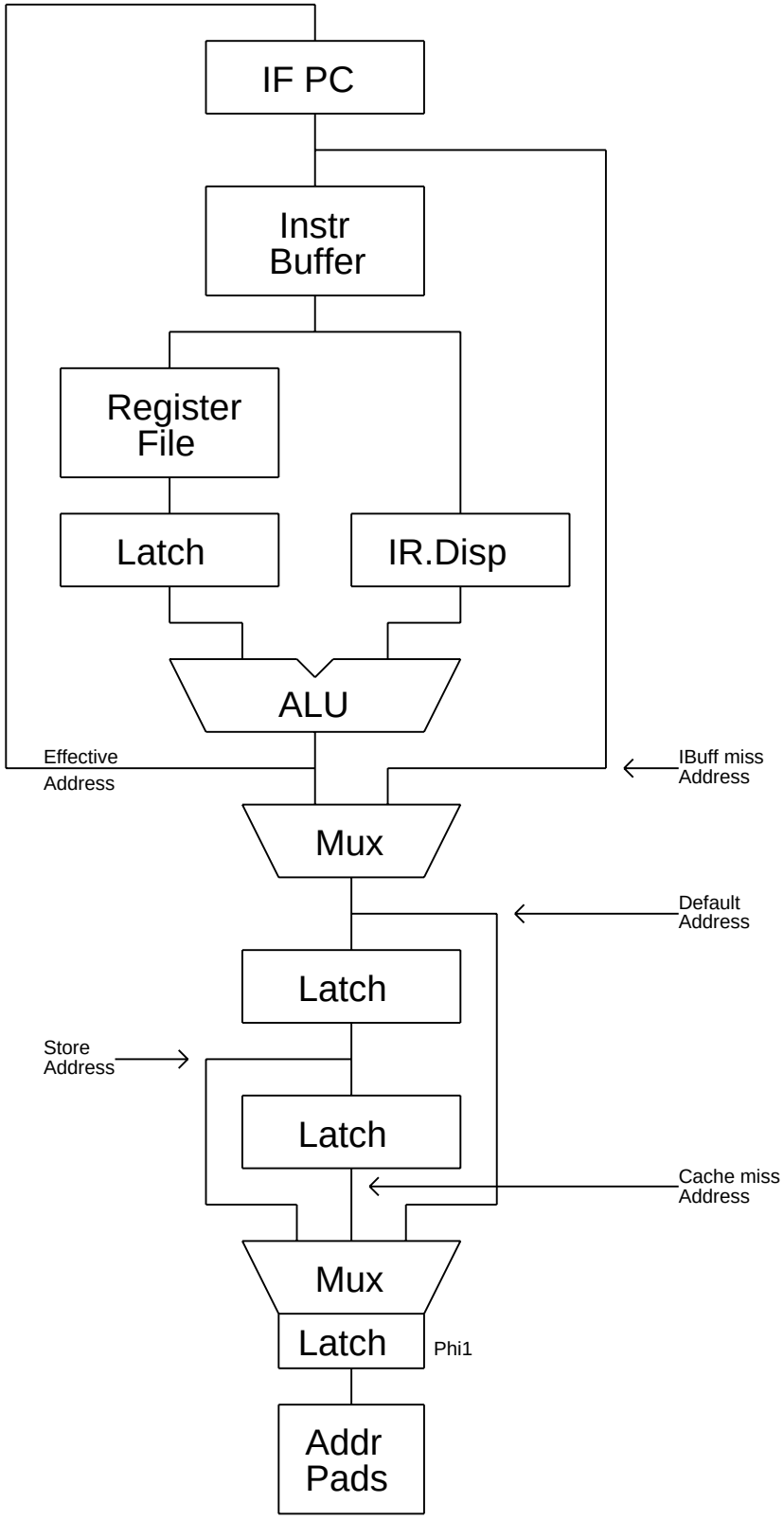


Figure 1-10: CPU Address paths

instruction in MEM is held there on the next cycle. When true (i.e. nobody grounding it), MEM should either be killed or loaded in the next cycle from the current contents of ALU, depending on LoadALU.

LoadWB Load the WB stage. High-true open-collector with external pull-up resistor. Driven by FPU and CCU, received by all three chips. When false (i.e. somebody grounding it), the instruction in WB is held there on the next cycle. When true (i.e. nobody grounding it), WB should either be killed or loaded in the next cycle from the current contents of MEM, depending on LoadMEM.

~Interrupt Interrupt the instruction stream. Low-true open-collector with external pull-up resistor. Driven and received by all three chips. (Interrupt and AllowInt) kills all pipeline stages, sets the PC to low memory, and puts the CPU in kernel mode. ~Interrupt must not be asserted when a kernel-mode instruction is in any pipe stage.

AllowInt Allow interrupts. High-true totem-pole. Driven by the CCU, received by all. While AllowInt is false, ~Interrupt is ignored. De-asserted by the CCU during multiple-cycle interactions with the memory controller to prevent an interrupt from killing a memory reference in MEM or WB.

Kernel Kernel mode. High-true totem-pole. Driven by the CPU from the PSW.kernel bit of the instruction in IF. Received by coprocessors to track the kernel bit in the instruction pipeline. ~Interrupt must not be asserted when a kernel-mode instruction is in any pipe stage.

Op Instruction opcode. High-true totem-pole, 4-bit bus. Driven by the CPU with the opcode of the instruction in IF. Received by coprocessors to track instructions in the pipeline.

Reg Instruction operand register. High-true totem-pole, 6-bit bus. Driven by the CPU with either RR or RA of the instruction in IF. Received by coprocessors to address internal registers.

Address 32-bit virtual byte address. High-true totem-pole. A load- or store-class instruction in MEM drives its effective address onto the Address bus. A store instruction in WB also drives its effective address onto the Address bus. A coprocessor ALU instruction in MEM is itself driven onto the address bus by the CPU for decoding by the relevant coprocessor.

InsAddr Instruction address. High-true totem-pole. Driven by the CPU to the CCU and cache rams. This signal starts a memory reference to reload the instruction buffer. It is directly used as an address bit to the cache rams, thereby separating the instruction from the data cache. InsAddr has the same timing as an address bus bit.

AltAddr Address alternate double word. High-true totem-pole. Driven by the CCU only to the CPU. The CPU always XORs the address[3] bus signal with the AltAddr signal. The CCU asserts AltAddr to access the alternate doubleword of a cache line.

Data 64-bit data bus with byte-parity. High-true tri-state. Driven by the CPU, CCU, FPU, cache rams, and the memory interface to transfer data.

DataEnable Enable Data bus drivers. High-true totem-pole. Driven by the CCU to the CPU and FPU. Deasserted by the CCU to disable the CPU and FPU Data bus drivers. Prevents instructions stalled in the pipeline during a cache miss from driving the Data bus, which the CCU uses to speak to main memory.

CheckParity Check Data bus parity. High-true totem-pole. Driven from the CCU's PSW; received by

all three chips. When asserted, a parity error sets a bit in the PSW of the chip detecting the error, causing an interrupt.

Reset Processor reset. High-true totem-pole. Asserted by the clock/scan module to restart the processor. Reset kills all pipeline stages, sets the PC to a low address, and puts the CPU in kernel mode. Used by the CPU as bit 7 of the PC after a reset. This signal allows a distinction to be made between a trap and a power-on reset.

Save Save processor state over reset. High-true totem-pole. Asserted by the clock/scan module in conjunction with reset. Prevents the CCU from invalidating the caches during a reset. Used by the CPU as bit 8 of the PC after a reset. This signal allows a distinction to be made between resets after power-on, when memory should be cleared, and resets after crashes, when memory should be preserved.

Clock Master clock oscillator. TTL-level sine wave at four times the cycle rate. A 100 MHz master clock frequency yields a 40 ns cycle time.

1.5 Timing

All chips operate synchronously using the same four-phase non-overlapping clocks, called Phi1 through Phi4. The rising edge of the Reset signal synchronizes the phase generators in the three chips. Each clock phase is nominally 10ns, and a cycle of four clock phases is nominally 40ns. The boundary between one cycle and the next occurs between the end of Phi4 and the start of Phi1.

Most output signals come from latches with less than half a phase of delay between latch output and chip pin. Most input signals go to latches with less than half a phase of delay between chip pin and latch input. A signal can be driven by one chip and received by the other two chips in one phase.

Figures 1-11 and 1-12 show the Local bus timing. Reset falls at the start of Phi3, and should be sampled at the end of Phi3 or Phi4. ~Interrupt, LoadMEM and LoadWB precharge during Phi2 and should be sampled at the end of Phi3 or Phi4. The CCU should drive AllowInt at the start of Phi3; the CPU and FPU sample it at the end of Phi3. The CPU can't drive LoadIF and LoadALU until the start of Phi4, so coprocessors should sample these two signals at the end of Phi4. The CPU samples the pipeline control signals at the end of Phi3 and drives what it saw back out during Phi4 and Phi1. Coprocessors should drive LoadIF and LoadALU starting in Phi3, since the CPU (when it isn't driving them) samples them at the end of Phi3. During Phi4, all chips compute in parallel the hold, load, or kill action for each pipeline stage. (Since LoadIF and LoadALU are arriving during this phase, the calculation for the ALU and MEM stages are incomplete.) The next state of the instruction pipeline is known early in Phi1 and it advances in all chips in parallel (with some late multiplexing to fix up the late arrival of LoadIF and LoadALU). The new value of PSW.kernel for the instruction being fetched is driven at the start of Phi1. The CPU fetches a new instruction from IBuff during Phi1-3, and drives the Opcode and Register fields at the start of Phi4. The Address bus is driven as early as possible in a cycle to allow the maximum time for cache ram access. InsAddr, which is used as an address bit for the cache rams, has the same timing as the Address bus. AltAddr, which affects Address[3], is sampled at the start of Phi1, so it must be driven at the start of Phi4 of the previous cycle. The CPU samples the Data bus at the end of Phi4 of every cycle and writes what it got into a register or IBuff half-line during Phi2 of the next cycle if WB contains a load-class instruction. Write data to the cache should be valid as early in Phi3 as possible, and should remain stable through the end of Phi4. DataEnable is de-asserted by the CCU in Phi1 to prevent the CPU or FPU driving the Data bus in this cycle (it must also stall the pipeline).

Signal	Driven	Sampled	Notes
Reset	Start phi1	special	assert
Reset	Start phi3	end phi3	deassert
~Interrupt	start phi3	end phi3	
AllowInt	start phi3	end phi3	
LoadIF	start phi4	end phi4	CPU driving
LoadALU	start phi4	end phi4	CPU driving
LoadIF	start phi3	end phi3	FPU+CCU driving
LoadALU	start phi3	end phi3	FPU+CCU driving
LoadMEM	start phi3	end phi3	
LoadWB	start phi3	end phi3	
Kernel	start phi1	end phi4	
Op bus	start phi4	end phi4	
Reg bus	start phi4	end phi4	
Address	start phi1	end phi4	
InsAddr	start phi1	end phi4	
AltAddr	start phi4	end phi1	
Data (Read)	start phi1	end phi4	
Data (Write)	start phi3	end phi4	
DataEnable	start phi1	phi1	
CheckParity	start phi3	end phi3	

Figure 1-11: Local bus timing table

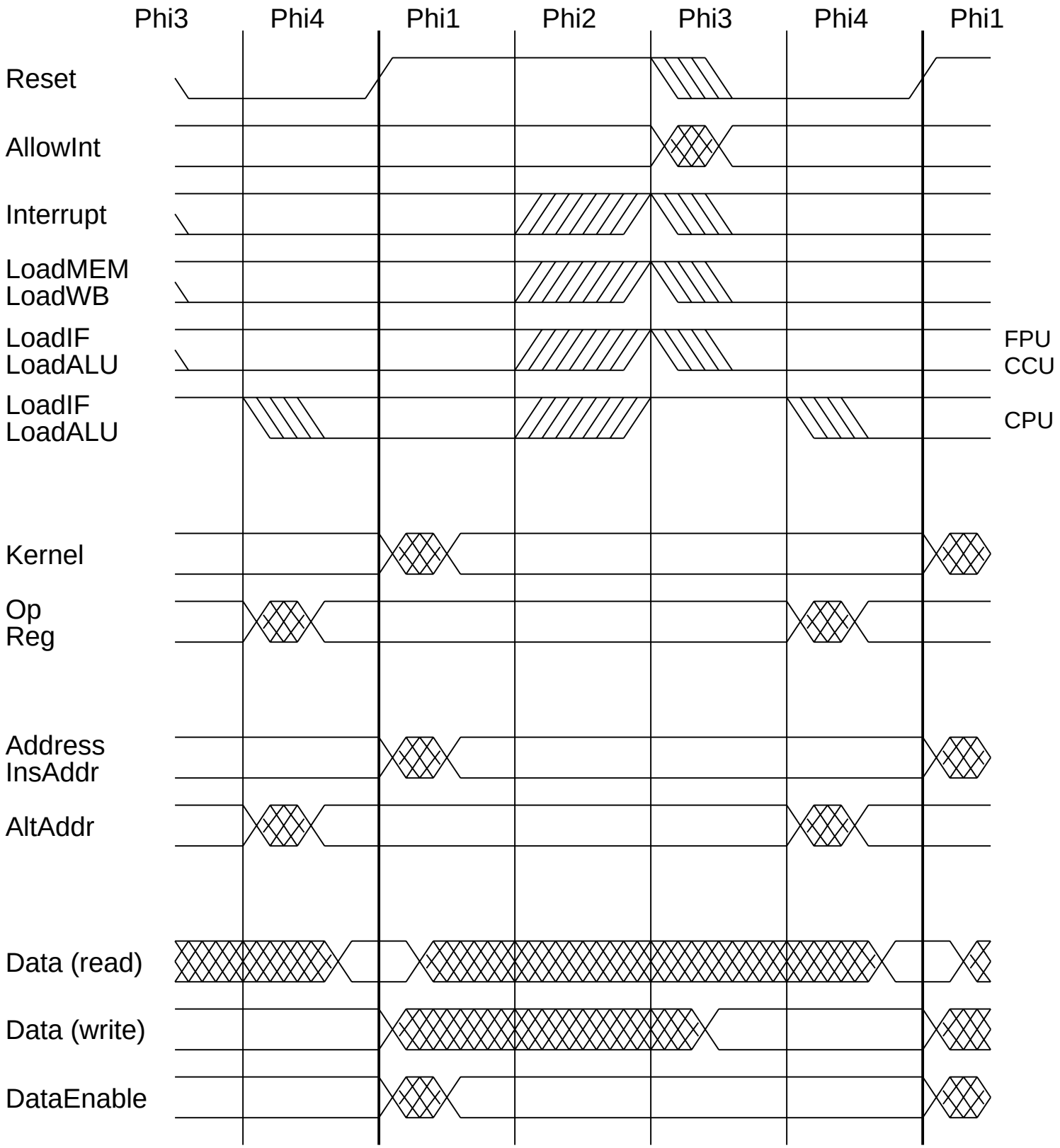


Figure 1-12: Local bus timing diagram

2. Global Bus

The MultiTitan Global bus is the group of signals that interconnects the CCUs and main memory in a multiprocessor; see figure 1-1.

2.1 Bus Access Control

Access to the Global bus is controlled by a finite state machine which implements a quasi round robin service policy. Each processor has a request and a grant line to the bus controller. A processor gains control of the Global bus by asserting its request line and waiting for its grant line to be asserted. A processor relinquishes control of the bus by deasserting its request line one cycle before it stops using the bus. The bus controller latches the request lines each cycle when it is idle. Once one or more asserted request lines are latched, the controller becomes busy, stops latching requests, and grants bus access to the latched requests in priority order. When all of the latched requests have been granted, the controller becomes idle and starts latching the request lines again. This algorithm is simple to implement and fair under heavy load.

Figure 2-1 shows how this works for three requestors. (This drawing assumes zero combinatorial and propagation delays.) Three arbitration cycles are shown: first, all three simultaneously request access; then as soon as they are served they request again; finally, 0 and 2 request again but 1 does not. Requestor 0 has highest priority, so the order of service is 0, 1, 2, 0, 1, 2, 0, 2.

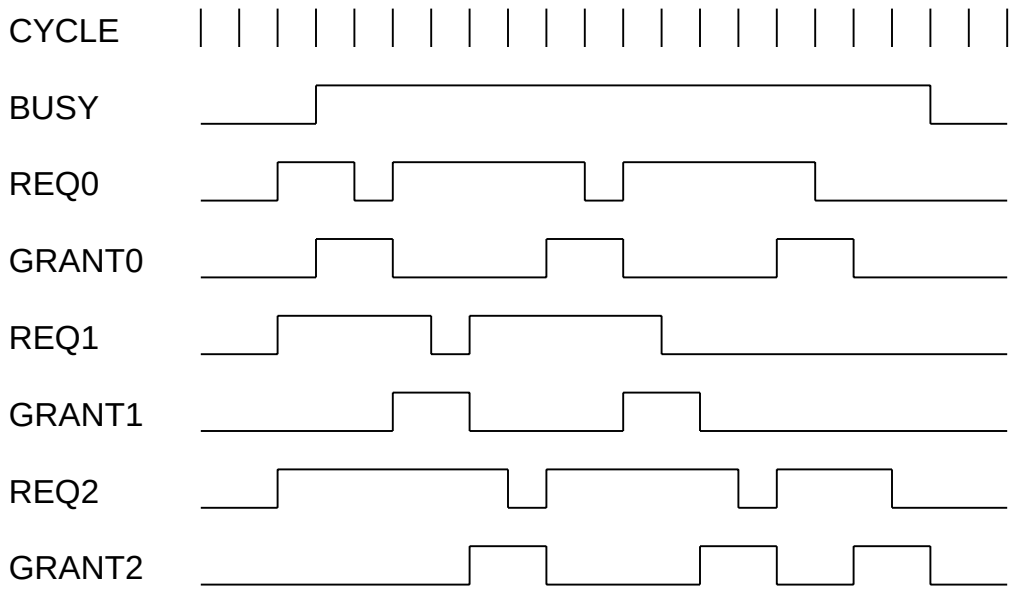


Figure 2-1: Global bus access

2.2 Remote Send and Receive

The CCU implements a pair of interruptible blocking instructions with which processors can exchange cache lines at low cost. A CCU executing a *receive* instruction stalls its CPU and asserts the ~Receive signal to notify senders of its presence. A CCU executing a *send* instruction stalls its CPU, becomes master of the Global bus and then broadcasts the instruction's effective address and cache line data while asserting the Send signal. If another CCU blocked in a receive instruction matches the send address, it loads a cache line and asserts the ~Match signal. Senders and receivers that match stop stalling their pipelines. Each time ~Receive is asserted, stalled senders contend for the Global bus and rebroadcast their addresses and cache lines, hoping that the new receiver will assert ~Match. Multiple senders and one receiver may share an address, but only the first successful sender will transfer

data and stop stalling. Multiple receivers and one sender may share an address, but the sender can't tell which receivers stopped stalling.

Figure 2-2 shows a cache line being exchanged over the global bus. The sending processor requests bus control and it is granted on the next cycle. The Send signal is asserted as the remote address is driven onto the global data bus. Four 32-bit words are driven onto the data bus in consecutive cycles. At the end of the second data cycle, the sender samples the ~Match signal.

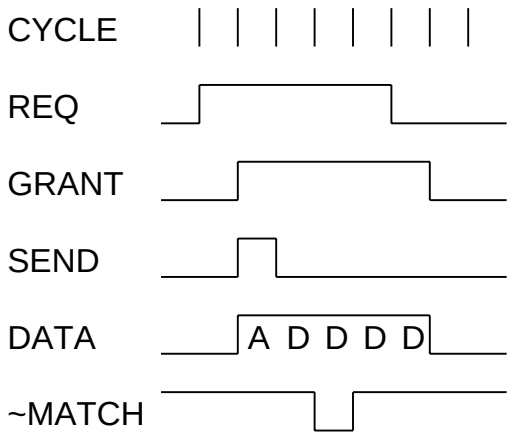


Figure 2-2: Exchanging a cache line between processors

2.3 Memory Control

The MultiTitan will use the (ECL) Titan memory and I/O system. The memory controller maintains the memory arrays and serves as an interface between the processors and the I/O adaptors. The memory controller supports from one to four memory modules of 32M bytes each, performing the ECC generation and checking as well as ram refresh functions. All memory transactions are in units of four 32-bit words.

The memory controller performs read, read/write, and write operations to service processor cache clean miss, dirty miss and flush operations, respectively. During a dirty miss, the write data is received from the processor during the ram read access time to minimize cache miss overhead.

The Titan memory controller performs direct memory access (DMA) read and write operations for I/O adaptors. It also performs 32-bit reads and writes to I/O adaptor registers in response to processor I/O instructions. See the Titan System Manual for more details.

Figure 2-3 shows a memory read/write cycle. The CCU requests global bus control and it is granted on the next cycle. Since MemRdy is true, MemRead is immediately asserted to start the read reference, and the read address is driven onto the global bus (and repeated onto the ToMem bus) in the next cycle. MemWrite is asserted in the cycle after MemRead to start an overlapped write cycle as well, and the write address is driven to memory in the next cycle. After the write address come the four 32-bit words to be written and then the global bus is released. Some time later, when the read data is available, the memory controller asserts DataRdy and drives four 32-bit words onto the FromMem bus.

Figure 2-4 shows a memory read-only cycle. The CCU requests global bus control and it is granted on the next cycle. Since MemRdy is true, MemRead is immediately asserted to start the read reference, and the read address is driven onto the global bus in the next cycle. Global bus control is released in the next cycle, and some time later,

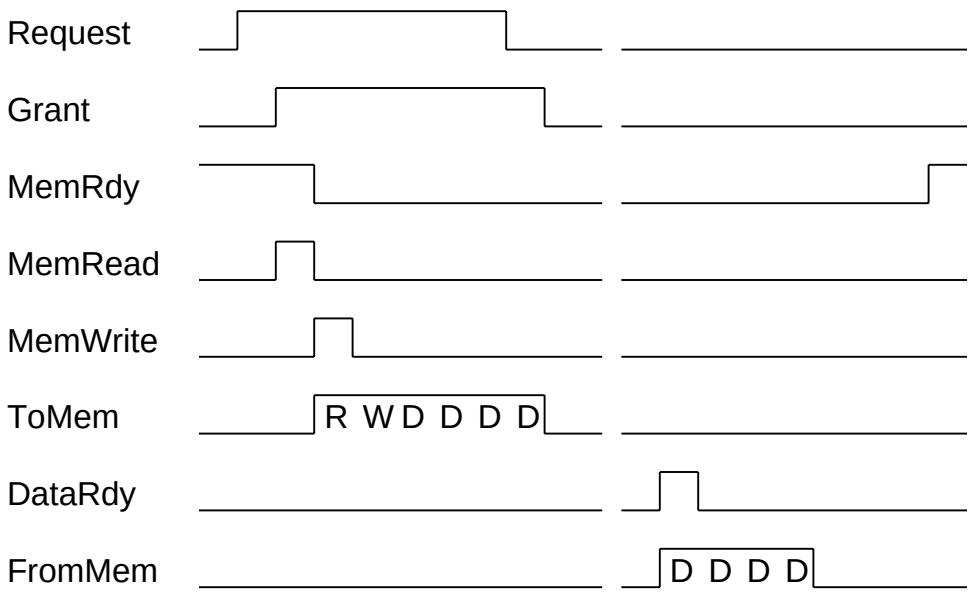


Figure 2-3: Memory Read/Write cycle

when the read data is available, the memory controller asserts DataRdy and drives four 32-bits words onto the FromMem bus.

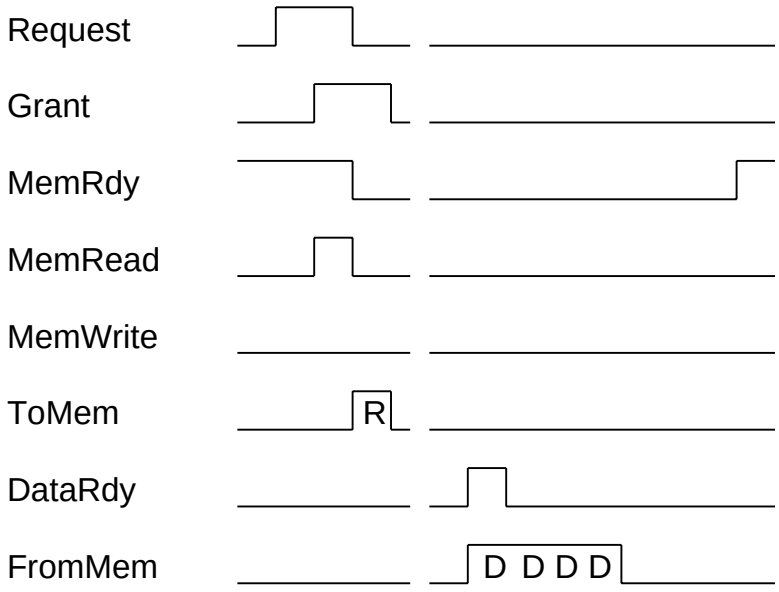


Figure 2-4: Memory Read-only cycle

2.4 Signal Definitions

Control signals are treated as boolean variables. A high-true signal is positive when it is asserted or true and ground when it is deasserted or false. A low-true signal is ground when it is asserted or true, positive when it is deasserted or false, and has "~" as the first character of its name.

BusReq Request Global Bus control. High-true totem-pole. Eight of these, one per processor, are received by the bus access control machine. Asserted by a CCU when it wants to control the Global bus. Deasserted one cycle before the CCU relinquishes control.

BusGrant Grant Global Bus control. High-true totem-pole. Eight of these, one per processor, are driven by the bus access control machine. At most one of these eight signals is asserted at any time. The CCU corresponding to the asserted bit is master of the Global bus and may drive Data, Send, and the memory control signals.

Data multiplexed address and data with byte parity. High-true tri-state 32-bit bus. Driven by the bus master CCU and received by the memory controller during a cache reference; received by other CCUs during a remote reference.

~Interrupt Interrupt a processor. Low-true open-collector 8-bit bus with external pull-up resistor. Asserted to cause an external interrupt to a processor. The CCU corresponding to the asserted bit of the global ~Interrupt bus asserts the ~Interrupt signal on its Local bus and records an external interrupt in its PSW. Memory system interrupts (errors and device interrupts) assert ~Interrupt[0].

Send Send a remote address. High-true totem-pole. Asserted by the Global bus master when it executes a Send instruction. The remote address and cache line are broadcast on the global bus; the ~Match signal is asserted if another CCU matches the address.

~Receive Receive a remote address. Low-true open-collector with external pull-up resistor. Asserted by a CCU when it begins to execute a Receive instruction. Sending CCUs should gain global bus control, rebroadcast their remote addresses, and check for ~Match asserted.

~Match Match a remote address. Low-true open-collector with external pull-up resistor. Asserted by a CCU when it matches the remote address sent by the current bus master.

MemRead Start memory read reference. High-true tri-state. Asserted by the Global bus master to start a memory read. The read address is driven onto the Global bus in the next cycle.

MemWrite Start memory write reference. High-true tri-state. Asserted by the Global bus master to start a memory write. This should only be asserted if MemRead was asserted in the previous cycle. The write address is driven onto the Global bus in the next cycle.

MemRdy Memory control is ready to start a memory reference. High-true totem-pole. Driven by the memory controller and received by the global bus master. The global bus master must continue to assert MemRead and drive the Address bus until MemRdy is true. MemRdy goes false when the controller accepts a command. The memory controller stays busy after a reader has released the Global bus. This signal prevents the next bus master from starting another memory reference until the controller has finished the previous reference.

DataRdy Memory read data is ready. High-true totem-pole. Driven by the memory controller and received by whichever CCU last started a memory read reference. Asserted during the cycle that the first word of memory read data is on the FromMem bus.

IORdy Memory control is ready to start an I/O reference. High-true totem-pole. Driven by the memory controller and received by the global bus master. IORdy goes true one cycle after the memory controller drives the FromMem bus with the selected control register value.

IOInst Perform an I/O operation. High-true tri-state. Driven by the global bus master and received by the memory controller. When asserted, the memory controller drives the contents of one of its internal registers onto the FromMem bus. The control register was specified by the CtlReg bus value during the previous assertion of IOInst. Zero or one of IORead, IOWrite, and SetCtlReg may also be asserted.

IORead Start an I/O read reference. High-true tri-state. Driven by the global bus master and received by the memory controller. When asserted with IOInst, an I/O read reference is started.

IOWrite Start an I/O write reference. High-true tri-state. Driven by the global bus master and received by the memory controller. When asserted with IOInst, an I/O write reference is started and the value on the ToMem bus is written to an IO device.

setCtlReg Load a controller register. High-true tri-state. Driven by the global bus master and received by the memory controller. When asserted with IOInst, a memory controller register is loaded from the value on the ToMem bus.

CtlReg Control register number. High-true 3-bit tri-state bus. Driven by the Global bus master and received by the memory controller. This value is remembered each time IOInst is asserted, and used to select the controller register returned on the FromMem bus the next time IOInst is asserted.

ToMem Data bus to memory. High-true 32-bit bus with byte parity. The global Data bus is repeated to the memory controller as the ToMem bus.

FromMem Data bus from memory. High-true 32-bit bus with byte parity. Driven by the memory controller and received by all processors. Data from the memory and IO system arrive over this bus.

2.5 Timing

Figure 2-5 shows the Global bus timing. The Global bus is long and crosses the backplane so its signals are driven early in Phi1 and sampled at the end of Phi4.

Signal	Driven	Sampled
BusReq	start phi3	end phi4
BusGrant	start phi1	end phi4
Data	start phi1	end phi4
~Interrupt	start phi1	end phi4
Send	start phi1	end phi4
~Receive	start phi1	end phi4
~Match	start phi1	end phi4
MemRead	start phi1	end phi4
MemWrite	start phi1	end phi4
MemRdy	start phi1	end phi4
DataRdy	start phi1	end phi4
IORead	start phi1	end phi4
IOWrite	start phi1	end phi4
IORdy	start phi1	end phi4
SpecialOp	start phi1	end phi4
SetCtlReg	start phi1	end phi4
CtlReg	start phi1	end phi4
From Mem	start Phi1	end phi4

Figure 2-5: Global Bus Timing Constraints

Table of Contents

List of Figures