JULY 1989

WRL Research Report 89/9

Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU

Norman P. Jouppi

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	WRL-Techreports@decwrl.dec.com
CSnet:	WRL-Techreports@decwrl.dec.com
UUCP:	decwrl!wrl-techreports

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU

Norman P. Jouppi

July, 1989

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

This paper describes the architectural and organizational tradeoffs made during the design of the MultiTitan, and provides data supporting the decisions made. These decisions covered the entire space of processor design, from the instruction set and virtual memory architecture through the pipeline and organization of the machine. In particular, some of the tradeoffs involved the use of an on-chip instruction cache with off-chip TLB and floating-point unit, the use of direct-mapped instead of associative caches, the use of a 64-bit vs. 32-bit data bus, and the implementation of hardware pipeline interlocks.

This is a preprint of a paper that will be presented at the
16th Annual International Symposium on Computer Architecture,
IEEE and ACM, Jerusalem, Israel, May 28-June 1, 1989.
An early draft of this paper appeared as WRL Technical Note TN-8.

1. Introduction

The MultiTitan is a high-performance general-purpose 32-bit microprocessor developed at Digital Equipment Corporation's Western Research Lab (DECWRL). Each processor consists of three custom chips: the CPU, floating point coprocessor (FPU), and external cache controller (CCU). The CPU has been implemented in a 1.5u CMOS technology with 179,390 transistors and runs with a 40ns cycle time [6]. In this paper we will discuss architectural and organizational tradeoffs made in the design of the system.

Each processor of MultiTitan is similar in some respects to the DECWRL Titan [10]. The Titan was built from 100K ECL MSI parts, and was developed at the lab from 1982 to 1986. Like the ECL Titan, the MultiTitan is a very simple RISC machine with a branch delay of one. However, the MultiTitan is not object-code-compatible with the ECL Titan. Unlike the ECL Titan, the MultiTitan has hardware support for fine-grain parallel processing, unified vector/scalar floating point registers and operations, and a different pipeline and method for handling exceptions. Figure 1 is an overview of one MultiTitan processor.

The MultiTitan instruction set is very simple and has a regular encoding. The machine has a 4-stage pipeline: instruction fetch (IF), execute (EX), memory access (MEM), and write back (WB). The resources and timing of the pipeline are shown in Figure 2. The primary principle followed in the design of the the machine is:

> *Sustained performance must be maximized while the ratio of peak performance to sustained performance must be minimized. This is best accomplished by minimizing the latency of all operations as much as possible within a simple and regular framework.*

Obviously sustained performance must be maximized, because that is what is delivered to the user. Peak performance should be minimized, since

> *Peak performance is meaningless except as an indicator of machine cost.*

In other words, high peak performance requires high hardware cost. Thus to utilize the hardware most efficiently, the ratio of sustained performance to peak performance must be as close to 1 as possible.

In general, the lowest possible latency provides the best possible performance. The latency of an operation is the time from its initiation until its completion. In contrast, bandwidth corresponds to the peak rate at which operations may be issued or retired but says nothing about

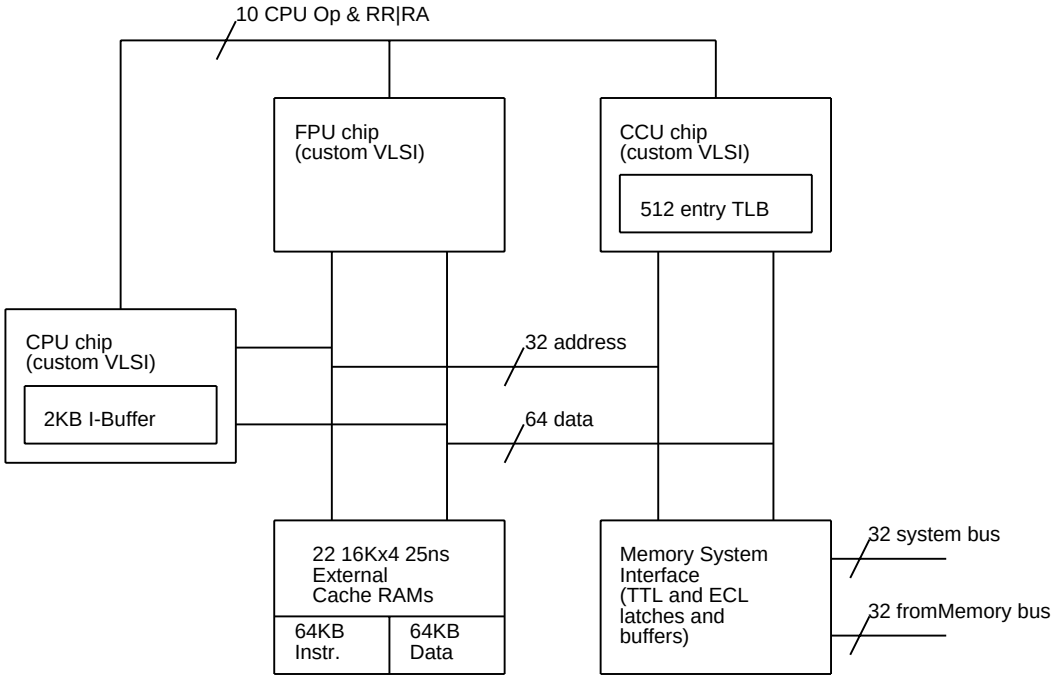

Figure 1: One MultiTitan processor

how long each operation takes. Bandwidth is quoted when peak MFLOPS or the clock frequency of a microprocessor is given. By heavily pipelining a machine, the bandwidth of the pipeline is increased, but the latency of operations when measured in cycles is also increased. Latency is important because instruction-level parallelism is limited and data dependencies exist between instructions. If a large bandwidth is provided at high latency, data dependencies and limited parallelism will force the machine to stall waiting for the results of operations.

Finally, the latency must be minimized within a simple and regular framework. If the machine is made complex and irregular, the global machine control and overall machine organization can be adversely affected. This can slow down the basic cycle time or other operations in the machine, outweighing the result of the original speedup.

This paper follows the structure of the MultiTitan pipeline. Section 2 discusses issues related to the IF pipestage such as the large on-chip instruction cache and hardware interlocks. The implementation of integer multiply and divide by the coprocessor and the use of a single adder in the CPU are discussed with other EX pipestage issues in Section 3. MEM pipestage issues such as the TLB and cache organization are the subject of Section 4. Issues involving the WB pipestage are presented in Section 5. Section 6 summarizes the major tradeoffs presented in the paper.

2. IF Pipestage Tradeoffs

In this section we describe how the primary design principle was applied to the IF pipestage. Excess latency before the execute stage is visible in stalls when branches are taken. Since about 1 in every 15 instructions are taken branches in our benchmarks, each extra cycle required to fetch an instruction can slow the machine down by 1/15, or about 7%.

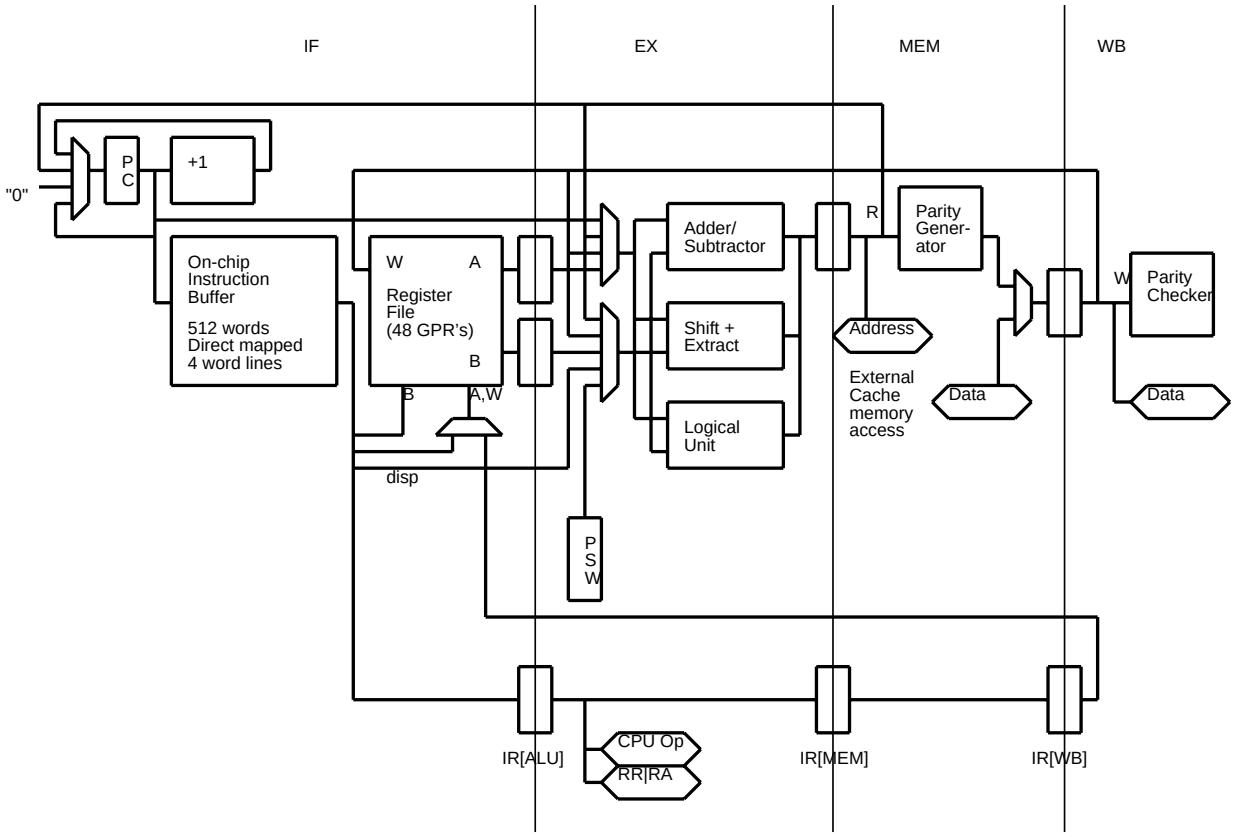

Figure 2: CPU pipeline and machine organization

2.1. What should be put on-chip?

At the level of integration available for the MultiTitan, several options are possible for what to place on the CPU chip. Along with the datapath and instruction decode, other recent machines have placed either the TLB on-chip [8, 11], the floating-point unit on-chip [3, 12], or an instruction cache on-chip [1, 5]. The choice of what to put on-chip in the MultiTitan was based on minimizing the average latency. The average latency of a particular operation must be weighted by the frequency of execution of that operation to obtain its effect on the overall machine performance. For example, in a RISC machine running at one cycle per instruction, an instruction must be fetched every cycle. Loads and stores usually account for around 30% of all instructions. Floating-point operations rarely account for more than 25% of all operations even in numeric applications.

If the instruction cache is placed on-chip and is a virtual instruction cache, then instruction accesses do not require address translation on a hit. Then the most frequent operation (instruction fetch every cycle) is on-chip, and the less frequent operations (load or store and their associated address translation) incur the increased latency of going off-chip. If a separate FPU chip has its own set of registers, then loads and stores to the FPU chip can take place directly from the data cache, with the same latency as an FPU on the CPU chip. Only when transfers between the floating-point registers and CPU registers are required is additional latency incurred.

Luckily, transfers between CPU and FPU registers are rare, occurring only for conversions between integer and floating-point values and implicitly or explicitly when branches must be made on the result of floating-point compares. Our simulations showed transfers between CPU and FPU register sets or vice versa to be less than 3% of the instructions executed on the Livermore Loops (1-12), Whetstones, and other floating-point intensive benchmarks.

Of course this level of analysis is an oversimplification. Second-order effects, such as whether the instruction cache can be made large enough to have a good hit rate (and hence a low average latency) often dominate. Also, care must be taken to count additional latency only when it really affects the machine, such as counting instruction-fetch latency only during branches. Similar questions must be raised for the other possibilities as well. For example, could an on-chip TLB be made large enough to have a low average latency (i.e., high hit rate)? Similarly, would an on-chip FPU have enough transistors available to exploit potential parallelism in floating-point operations, such as a full multiplier array instead of an n-bit at a time algorithm?

Next the simplicity and regularity of the system must be considered. For example, if the TLB is put on-chip, will an off-chip backup TLB still be necessary? If so, then the on-chip TLB does not reduce the parts count on the board. If the instruction cache is on-chip, this will reduce the I/O requirements of the chip. If the instruction cache and data cache are both off-chip, then the data and address busses will either have to be time multiplexed (which is likely to add latency and impose an upper bound on performance), or separate busses must be provided (which implies a very high pin count). The signal integrity of high pin count packages is worse than that of low pincount packages, which further increases the latency of going off-chip. If the instruction cache is placed on-chip, and the external cache is a mixed cache, then an entire set of cache RAMs chips can be eliminated over the case where both the instruction and data caches are off-chip and are accessed with time-multiplexed busses. Finally, care must be taken not to add all sorts of mechanisms (e.g., branch target buffers) to decrease latency without verifying that they don't increase the latency of global control or other operations more than they have saved.

In the case of the MultiTitan, the chip area we had available was enough to build a TLB with a reasonable hit rate. However, this area was not enough for very high performance floating-point support. We decided to put the instruction cache on-chip instead of the TLB primarily to reduce chip pin I/O bandwidth requirements. By also reducing the requirement for two independent external caches to a single external cache, putting the instruction cache on chip eliminated more parts from the board than moving a TLB from a custom CCU chip on-chip or by eliminating the custom FPU chip. Thus, for the MultiTitan, we found that placing the instruction cache on the CPU chip would have a better effect on performance and simplify the system more than placing anything else on-chip.

2.2. Organization of the Instruction Cache

The decision to put an instruction cache on-chip is based on its hit rate which depends on its size and organization, so these decisions must be made concurrently. But given that an instruction cache was the best thing to put on-chip, what is the best organization of the cache? This was decided according to the primary design principle. In other words, we want to minimize the average instruction fetch latency. This is given by the sum of its hit latency and its miss latency times its miss rate (i.e., Equation 1).

$$\tau_{average} = \tau_{hit} + \rho_{miss} * \tau_{miss} \quad (1)$$

There are many contributors to latency in an instruction cache access. Three of these are extensively discussed in the literature: the hit rate, the cache line refill time, and the access time of the cache memory arrays. The hit rate of the cache itself depends on the overall size as well as the line size. The overall size of the cache was fixed by the space available on the die to 2K bytes. If the cache was made larger than this not only would the die size become prohibitively large, but the access time of a memory array twice as large is also longer than that of the smaller array. The line size is an important factor in determining the miss rate of an instruction cache. Since taken branches are about 1 in every 15 instructions, if we have just branched to a location the chances are very good that we will execute at least the next four instructions. Therefore, making the instruction cache refill fewer than 4 words on a miss is a bad idea since the machine will execute more instructions than that in a row on average. For example, a machine that fetched only one word on a miss would have four cache misses to execute 4 consecutive instructions, while a machine that fetched back four words on a miss would have only one or at most two misses. Although for a 2K byte cache an 8-word line would provide a better hit rate than a 4-word line, in the MultiTitan we chose a line size of four words. This brings up another observation:

Real caches are determined by the RAM sizes available and the system context and are intentionally non-optimal from the viewpoint of simulation studies.

In particular, three aspects of the MultiTitan system dictated the decision for a four-word line. First, the MultiTitan was targeted to run on the Titan memory and I/O system, and the Titan memory system returns four words on all accesses. If the off-chip mixed cache had a four-word line to match the memory system, then it would also be simpler and more regular for the on-chip cache to have a four-word line as well. Second, if the off-chip cache had an access time of one cycle, then the time lost on a miss to get the required instruction would be 2 cycles (IF fails from miss, fetch instruction, and IF succeeds). During this time 4 words could be fetched over a 64-bit data bus. Third, based on the aspect ratio of the available on-chip space and the RAM cell itself, 160 cells could be easily driven on one word line. This corresponds to four words plus a tag.

The cache associativity also affects latency. For example, by increasing the associativity the average access latency may be reduced a few percent due to a higher hit rate. However, a much more important effect is present in a direct mapped cache (see case *b* of Figure 3). In a direct-mapped cache where the data and tag stores are made of the same memory, the data is available at the same time as the tag. In many cases the tag comparison is a significant fraction of the cache access time. In a direct-mapped cache the processor can start using the data before hit or miss is computed. This reduces the latency of the cache access in the case of a hit to that of the RAM access time. In an associative organization (see case *a* in Figure 3), the machine does not know which data to use until the tag comparison and the multiplex between the sets is complete. The reduced latency provided by the direct-mapped cache organization far outweighs the small increase in average latency from its lower hit rate for the cache size of interest.

Another common organization of caches is a set of buffers with a set of corresponding associative tags, as in the CRAY-1 [2] or MIPS-X [1] (see case *c* and *d* of Figure 3 respectively). These organizations are just extreme cases of caches with very long lines and associativity equal to the number of lines in the cache. There are two cases to consider for this organization. These

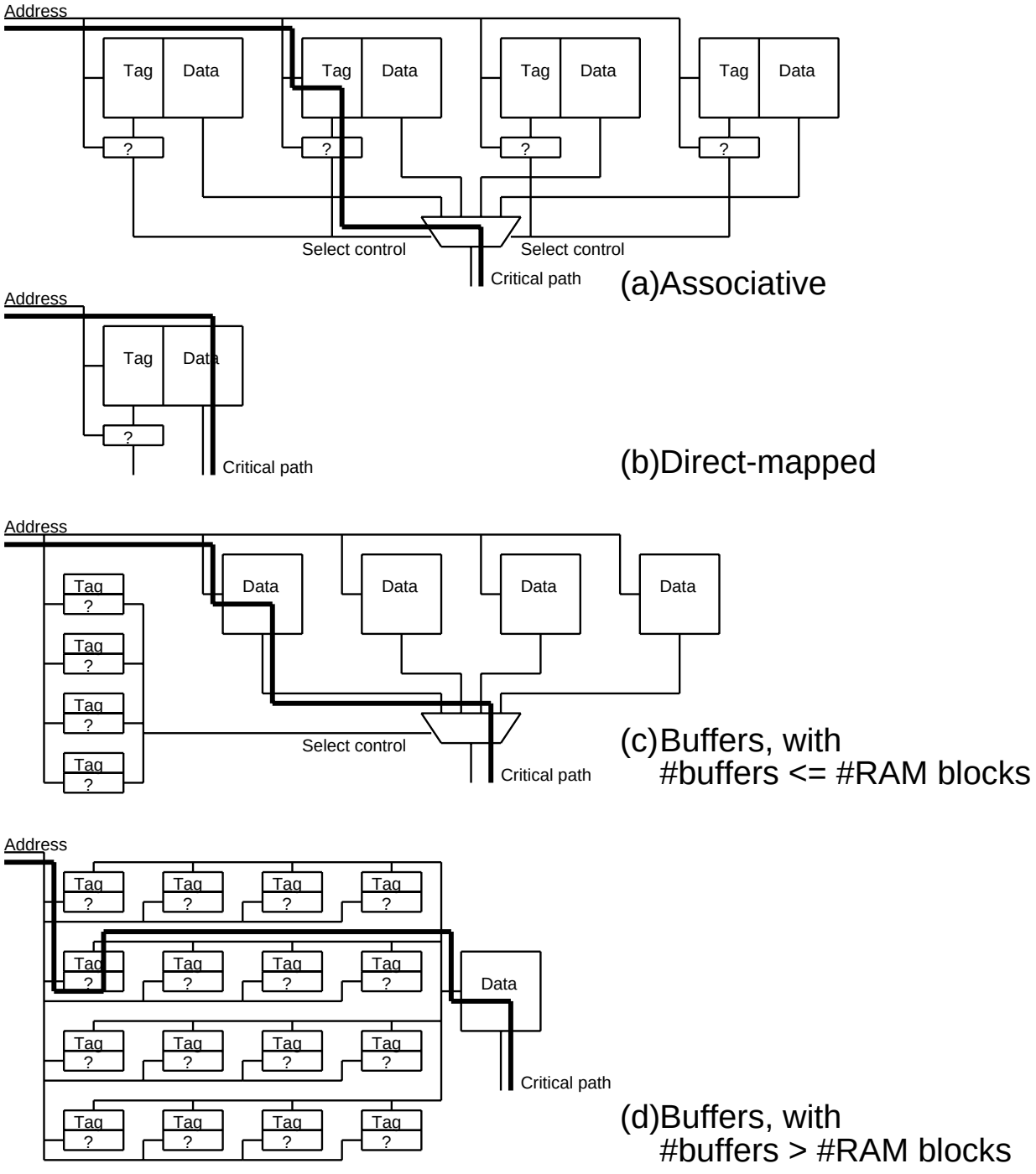

Figure 3: Effective latency of cache organizations

two cases differ based on the relationship between the buffer size and the size of the basic RAM building block used to construct the buffer. If the size of each buffer is greater than or equal to the basic building block used to build the buffer (e.g., 16-entry buffers made from 16x4 bit RAMs as in the case in the CRAY-1), the different buffers may be indexed immediately in parallel based on the low order bits of the instruction address. This is because each chip stores data from at most one buffer, and all of the chips can be cycled in parallel. The tag comparison is

also performed at the same time and can set up the multiplexors while the buffer access is still in progress. This method gives an access time for hits that is not much larger than that of the memory building block. However, the total number of tags available is limited to the total cache size divided by the size of each buffer. This limit on the number of tags (e.g., 4 in the CRAY-1) can have a significant negative impact on the hit rate. For example, three small code fragments that each cross a buffer address boundary would require two buffers each, for a total of six tags. In situations like this thrashing would result between the code fragments, and each of the large buffers would be poorly utilized. To circumvent this problem, the number of tags can be increased beyond the number of memory building blocks. Hence, data from several buffers will be stored in each RAM block. This was done in MIPS-X, where the cache is organized as 32 buffers of 16 words each. However, since the RAM itself only accesses four words in parallel, the tag comparison must be performed in order to generate the address of the block within the RAM to be accessed. This puts the tag comparison in series with the RAM access, and yields a latency about equal to that of the conventional associative case for a hit.

In summary, we have discussed four basic ways to organize a cache. Two methods put the latency of a tag comparison in series with the RAM access, while two put the tag comparison in parallel with the RAM access. Of the two parallel methods, one has a poorer hit rate than the other due to the limited number of tags available. For this reason, we chose a direct-mapped organization (method *b*) in the MultiTitan because its average latency is the lowest.

2.3. 64-bit Data Busses

As was mentioned in the previous section, the MultiTitan has a 64-bit data bus. This improves the performance of both double-precision floating-point benchmarks and machine performance during instruction cache misses. The costs and benefits of a 64-bit data bus for instruction cache refill will be quantified in this section.

The cost of a single 64-bit data bus and a 32-bit address bus shared by instructions and data is fairly low. It requires only 50% more pins than a 32-bit address and 32-bit data bus shared by instructions and data, and 25% fewer pins than separate 32-bit address and 32-bit data busses for both instruction and data references. The MultiTitan CPU is in a package with 140 signal pins and 36 power pins. Not all pins on the package are used since the die size is limited by the required perimeter of the bonding pads. The data bus also has byte parity, so together with the address bus they account for 102 of the 136 pins used. The remaining 34 pins are easily sufficient to handle all other I/O requirements of the chip.

Another cost of extra pins is increased power supply noise from simultaneously driving large numbers of outputs. In the case of the CPU chip, however, at most 32 outputs are driven at a time. (The CPU does not perform double-word stores.) When the CPU executes a store instruction, it places the data to be stored on the proper place on the 64-bit external data bus, so no external multiplexors are required in the path of the data.

Combined with the external 1-cycle cache, the 64-bit refill path reduces the time to refill a 16-byte line to 2 cycles. To get a better understanding of the implications of a two-cycle miss, consider the case where the CPU executes straight-line code with each instruction executed once and only once. Then the CPU will incur a 2-cycle miss on every line of code executed. The resulting performance of the CPU will only be degraded by 50% over the case where the CPU

hits on every access to the instruction cache. For realistic code sequences that do not miss on every line, the cost is much lower. Table 1 gives the miss rate of six programs, and the time required to execute them with 32-bit refill and with 64-bit refill of the 16-byte line. The first three programs are large applications that are in use daily at our lab. The second three are popular benchmark programs. For the real programs, the ability to fetch instructions at a rate of two per cycle reduces the CPI (cycles per instruction) burden of the on-chip instruction cache from 0.168 CPI to 0.084 CPI. A reduction of this magnitude would be insignificant for a complex-instruction-set machine that took 10 cycles to execute an instruction on the average. However, on a machine that executes an instruction every 1.25 cycles on the average, a 0.084 CPI improvement is fairly important. Since the maximum issue rate of the machine is 1 instruction per cycle, an improvement of 0.084 CPI is a reduction of the stall cycles by 30%. Unlike the real programs, the three benchmark programs spend almost all of their time in small loops. This gives them miss rates that are two orders of magnitude better than the real programs.

		Refill CPI burden	
benchmark	**miss rate**	**64-bit**	**32-bit**
ccom	5.4%	.108	.216
PCBroute	5.1%	.103	.206
TimingVerify	2.1%	.042	.083
real programs	4.2%	.084	.168
Linpack	0.03%	.0006	.0012
Livermore	0.05%	.0009	.0019
Stanford	0.01%	.0002	.0003
benchmark avg.	0.03%	.0006	.0011

Table 1: Performance improvement with 64-bit refill

One other possibility for reducing the CPI burden of instruction cache misses is a prefetch mechanism. For example, a machine that refilled cache misses over a 32-bit data bus might continue prefetching beyond the missed instruction until another miss occurred or the end of a very long cache line was reached. However, if the prefetcher only fetches 32 bits per cycle, the instruction fetch stage can easily becomed starved for instructions. This is because 30% of the instructions typically executed are loads or stores, and stores use the bus for two cycles (i.e., probe then write). Moreover, coprocessor ALU instructions (of which floating-point coprocessor arithmetic is the most important member) also use the address bus for one cycle when they are transferred between the CPU and the FPU. Thus, it is not uncommon for 50% of the bus cycles to be already occupied by instruction execution. Therefore even if the MultiTitan had a prefetch mechanism, a 64-bit bus would be necessary to allow the prefetcher to keep up with the instruction fetch stage, at least on the average.

2.4. Where did the Instruction Decode Stage Go?

In Section 2.2 a direct-mapped cache organization was chosen because it provided the data (at least provisionally, subject to the tag comparison registering a hit) faster than any other method. In the MultiTitan CPU many useful operations are overlapped with the instruction cache tag comparison. In fact, the entire contents of the instruction decode and register operand fetch stages of some machines are performed in parallel with the instruction cache tag comparison in the MultiTitan. This is possible because of the simplicity of the MultiTitan instruction set and

the simplicity of the CPU organization. All control signals can be generated by the instruction decode using at most two levels of four-input logic and an inverter. A total of 74 gates are required in all of the instruction decode logic.

The MultiTitan instruction set consists of two instruction formats (see Figure 4). Registers to be fetched in the instruction fetch stage appear in the same place in both formats, so decoding of the format is not required before the access of the register file begins. For example, the register file always fetches the registers specified by the bits in the ra and rb positions, even if the instruction is in the immediate format. Similarly rr is always accessed in the WB pipestage, whether written for ALU operations and loads or read for stores. The encoding of the instruction fields was chosen to simplify decoding as much as possible. Although the opcode is four bits, many control functions can be determined by examining only a subset of the opcode bits. For example, all instructions in the immediate format have the most significant bit of their opcode equal to 1, while those in the register format have 0 as their most significant bit. All control information is decoded in the instruction fetch stage. Control bits that are used in later pipestages are delayed by shift registers until they are needed.

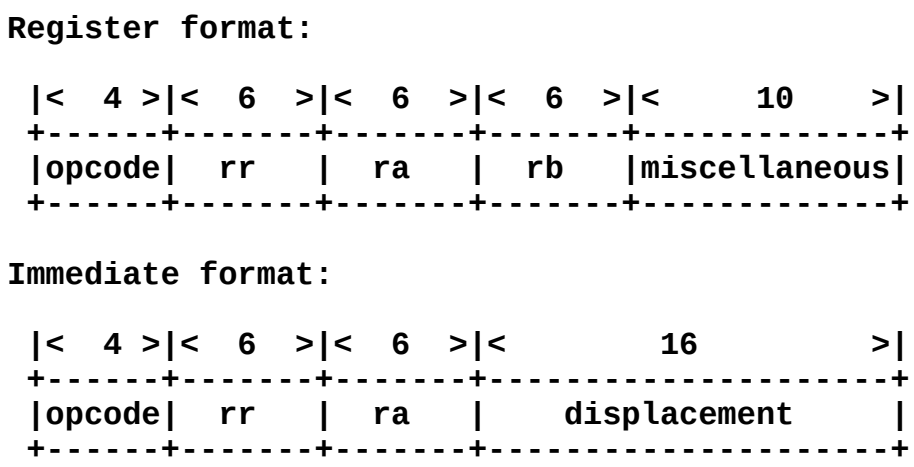

Figure 4: MultiTitan instruction formats

Table 2 shows the branch frequency for a number of programs. Since there is no instruction decode pipestage, one cycle is saved on every taken branch, or about 7% of the instructions. Assuming there is only one branch delay slot in the architecture, this improves the performance of the machine by approximately 7%. The improvements as a result of second-order factors, such as simpler resulting control logic or fewer PC queue entries are harder to quantify, but can be as significant. Although some machines have more than one branch delay slot, the second slot is not usually usefully filled without the use of branch "squashing" techniques [9], which add to the complexity of the machine.

2.5. A Machine with Interlocked Pipeline Stages

In the Stanford MIPS project [4], it was decided to implement all interlocks in software by inserting NOPs in the code. Interlocks were implemented in software because hardware interlocks were thought to adversely affect machine performance due to added control complexity. As an experiment in the MultiTitan, all interlocks were put in hardware.

benchmark	percent of instr. executed: uncond-itional	cond-itional	total taken	percent cond. taken
ccom	3.2	9.5	8.9	60%
PCBroute	3.7	9.7	8.2	46%
TimingVerify	3.5	7.7	7.3	50%
Linpack	0.1	2.3	1.7	69%
Livermore	0.0	4.7	4.4	94%
Stanford	2.6	13.1	12.9	79%
average	2.2	7.8	7.2	66%

Table 2: Frequency of branches

The four interlocks present in the CPU pipeline and detected by the MultiTitan are described below. In the following discussions, *store class instruction* will be used generically to refer to CPU->coprocessor transfer, coprocessor store, and CPU store, which all use an external bus in the WB stage. *Load class instruction* will be used to refer to CPU load, coprocessor load, coprocessor->CPU transfer, and coprocessor ALU instructions, which all use an external bus in the MEM pipestage. The FPU is responsible for stalling the machine if the result of a floating-point computation is requested before it is ready.

Load Interlock
: If a CPU register is written by a load instruction, and used as a source in the next instruction, one cycle is lost. There is no delay required between a coprocessor load and the use of the load data by a coprocessor.

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 cycle is lost.

CPU->Coprocessor Transfer Interlock
: If a Coprocessor->CPU transfer instruction attempts to issue two cycles after a CPU->Coprocessor transfer, one 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.

Table 3 gives the frequency of occurrence of these interlocks in several programs. Our three production programs are quite similar in their behavior, perhaps because they primarily use linked data structures. The numeric benchmarks in general make heavy use of array structures. The behavior of Linpack and Livermore are quite similar except for store and transfer interlocks. The inner loop of this version of Linpack has two integer multiplies used for array addressing calculations. Each multiply requires two transfers to the coprocessor and one to return the result of the multiply. These transfers are a bottleneck and cause transfer interlocks between themselves. The Stanford benchmark suite contains a wide range of programs, and so the frequency of interlocks in it is between that of the numeric benchmarks and our production programs.

	frequency of interlocks as a percent of all instructions executed		
benchmark	load	store & transfer	total
ccom	10.4	6.7	17.1
PCBroute	11.6	8.8	20.4
TimingVerify	12.7	8.6	21.3
Linpack	0.2	10.7	10.9
Livermore	0.6	1.0	1.6
Stanford	8.2	4.0	12.2
average	7.3	6.6	13.9

Table 3: Frequency of MultiTitan CPU interlocks

In the MultiTitan all interlocks are checked concurrently with the cache tag comparison. Since the instruction formats are so simple, the logic to detect interlocks is also very simple and fast. Like the instruction decode logic, at most two levels of logic and an inverter are required for interlock detection, with the exception of load interlocks which also require a pair of 6-bit register specifier comparators. Besides the comparators, there are only 22 gates and 5 latches required for detecting all interlocks in the CPU. This is about 18% of the total gate count of the control logic, which itself uses only 1.0% of the chip's transistors.

Based on the results from the MultiTitan, full hardware interlocks in a simple and regular machine provide a 14% improvement in code density for low cost. In particular, since interlock detection was performed in parallel with instruction decode, register fetch, cache parity checking, and cache tag comparison, interlock detection did not increase the cycle time of the machine. As a second-order effect, since the MultiTitan loses 8.4% of its cycles to instruction buffer refills, a code density improvement of 14% should result in approximately 1.2% fewer instruction cache miss refill cycles.

2.6. IF Pipestage Summary

During the design of the IF pipestage, we tried to follow our design principle as closely as possible. The timing of the resulting pipestage is summarized in Figure 5. It is hard to imagine an IF pipestage with an organization that results in lower latency.

3. EX Pipestage Tradeoffs

Several tradeoffs were made during the design of the MultiTitan regarding the execute pipestage. The two most important tradeoffs were the placement of EX operations in the pipeline, and the support provided for integer multiplication and division.

3.1. Where should the EX pipestage be?

To support memory refenences, after the instruction fetch pipestage we need a pipestage for address calculation and after that another to access the cache for loads or stores. Given this structure, the next decision to be made is where to execute ALU operations. Two reasonable choices exist.

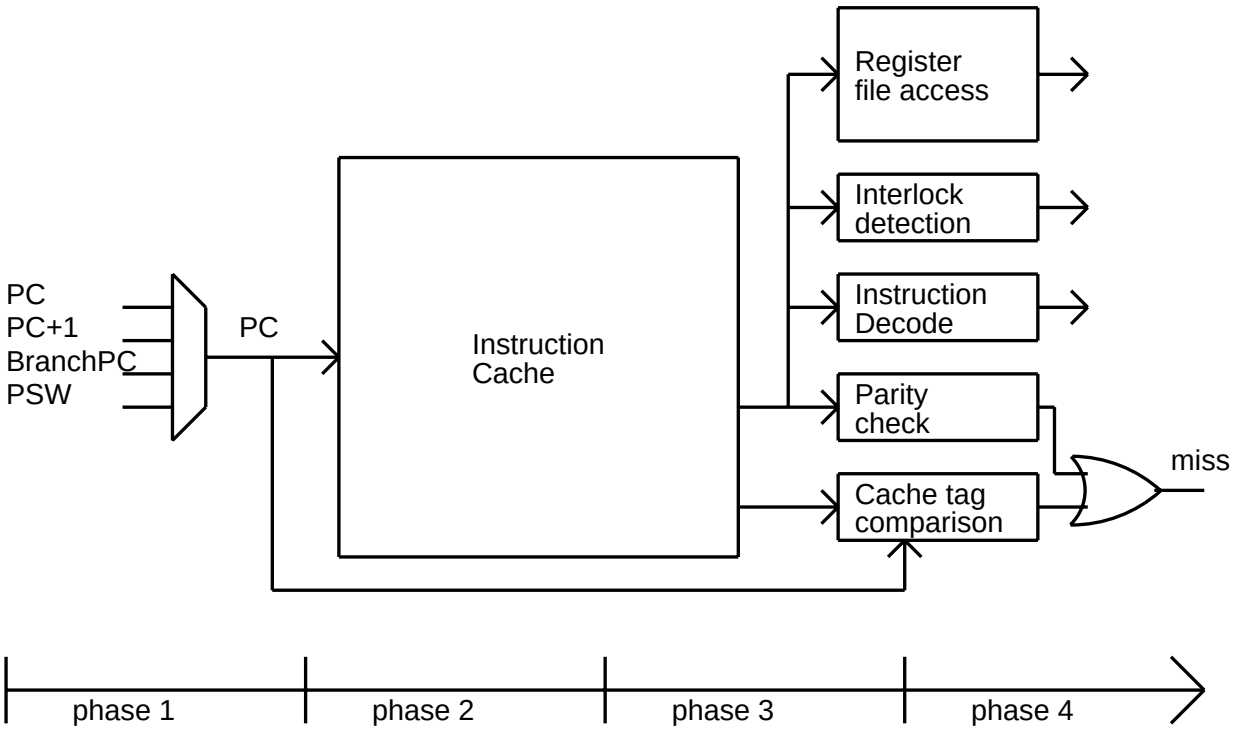

Figure 5: IF pipestage timing summary

The first option is that ALU operations could be performed in the same stage as addressing calculations (see Figure 6). If ALU operations are performed immediately after IF, the results of load instructions will not be available in time for an ALU operation that immediately follows the load. This results in a one cycle interlock in cases where another instruction can not be found to fill the load delay slot. This delay slot is not present for stores, since stores do not have a "result" accessible by ALU operations in a register-to-register machine. Similarly there is not an interlock for coprocessor loads and stores (e.g., floating-point loads and stores). This is because in the MultiTitan coprocessor ALU instructions are transferred to the coprocessor in the MEM pipestage over the address bus, and coprocessor operations begin execution in the CPU's WB pipestage. Coprocessor loads returning data at the end of the MEM pipestage then complete in time to begin a coprocessor ALU instruction in the WB pipestage. Thus if ALU operations are executed at the same time as address calculations, a 1 cycle interlock would occur between CPU load instructions and ALU operations that use the result of the load, but not for any other combinations of operations.

```
Load: IF      Ad/EX   MEM   -+ WB
                              |
                      IF      | Ad/EX | MEM     WB
                              |
ALU:                  IF      +>Ad/EX   MEM     WB
```

Figure 6: ALU operations in the address pipestage

As a second option ALU operations could be performed at the same time as memory references (see Figure 7). This has the advantage that there is no load delay cycle after CPU loads. However, the extra latency of addition and cache access (i.e., of memory reference

instructions) versus that of an addition alone (i.e., for ALU instructions) appears in another place. If ALU operations are performed at the same time as cache accesses, then ALU operations that compute values used in a later address calculation cannot execute as the instruction before the instruction with the address calculation without a one cycle interlock. Note that unlike the first option, this restriction applies to all memory references, whether loads or stores, and whether for the CPU or coprocessor.

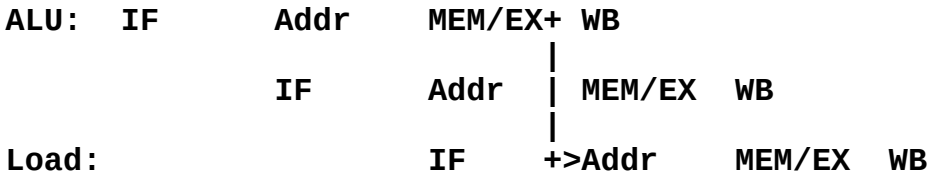

Figure 7: ALU operations in the MEM pipestage

These two pipeline organizations are quite close in performance. Although the second option applies to all memory references, many address calculations are based on relatively constant values (e.g., the stack pointer is fixed for the execution of a procedure). For base addresses that have not been recently calculated this interlock does not occur. The first option, however, causes an interlock in a higher percentage of the cases where it applies. This is because when a load is issued the data is usually required for another operation within a few cycles at most. To quantify the performance implications of both options, we ran a series of simulations of machines based on the two pipeline organizations (see Table 4).

	frequency of interlocks	
benchmark	load	address
ccom	11.6	13.1
PCBroute	10.4	7.4
TimingVerify	12.7	9.9
Linpack	0.19	0.34
Livermore	0.58	1.01
Stanford	8.2	7.9
average	7.3	6.6

Table 4: Load interlocks vs. address interlocks

Based on this table, it appears either executing ALU operations concurrent with addressing calculations or with memory references results in performance within 1% of each other. The next tradeoff to quantify are their relative implementation costs.

The big advantage of performing ALU operations at the same time as addressing calculations is that it allows the use of a single adder/subtractor for both. This is because in a load/store architecture ALU operations and addressing calculations are mutually exclusive. Besides performing address calculations and ALU operations, branch addresses can also be calculated in the same adder as well. This means that the machine only requires one adder/subtractor (and an incrementer for the program counter). Besides cutting down on the number of transistors and area required, the use of a single adder reduces the number of operand busses, bypass multiplexors, and control required in comparison to a machine with multiple adders. By reducing the area and complexity of the machine, we reduce the latency of communication across the machine for many different signals, both data and control. This means that the single adder machine should have a faster cycle time than a machine with multiple adders.

3.2. Where should integer multiplication and division be performed?

There are many different ways to provide support for integer multiplication and division. Some of the options we considered are listed in Figure 8. The most major choice is between performing the operations in the CPU itself versus in the FPU.

Integer multiplication:	cycles
In the CPU, 2 bits per cycle:	16
In the CPU, 4 bits per cycle:	8
In the CPU, 8 bits per cycle + 1 cycle:	5
In the FPU, transfer interlocks:	9
In the FPU, no transfer interlocks:	6
In the FPU, one constant operand, and no interlocks:	5

Integer division:	cycles
In the CPU, 1 bit per cycle:	32
In the CPU, 2 bits per cycle:	16
In the FPU, via reciprocal approx:	21

Figure 8: Integer multiplication & division tradeoffs

Performing integer operations in the FPU has the advantage that integer operations can use high-performance structures (e.g., a full multiplier array) already in place for the support of high-performance floating-point operations. These structures will have much better performance than any structure we could afford to place on the CPU. Although the FPU structures have lower latency, if integer operations are to be performed in another chip the time required to transfer operands and results between the chips must be added to the latency of the basic operations. If we perform these operations in the FPU with already-existing transfer and coprocessor ALU instructions, they add no hardware to the CPU. In contrast, if integer operations are performed in the CPU, the CPU datapath must be augmented with special hardware to support multiplication and division. Besides making the data path larger and slowing other operations down, this increases the design time of the CPU.

For these reasons in the MultiTitan we decided to support integer multiplication only in the coprocessor. The coprocessor provides an integer multiplication operation that returns the 64-bit product of two 32-bit integers in 3 cycles. Table 8 shows that the resulting integer multiplication times are equivalent to those available when performed in the CPU with a significant amount of hardware support (i.e., 4-8 bits per cycle). In many code situations the transfers to and from the coprocessor can be scheduled in order to avoid interlocks, resulting in an integer multiplication time of 6 cycles.

There is even less support for integer division in the MultiTitan than for integer multiplication. There is no integer division operation in the coprocessor, and no floating-point division operation either. Instead floating-point division is supported by a reciprocal approximation instruction followed by a series of Newton-step iteration instructions and floating-point multiplies. Integer division is performed by transferring the operands over to the coprocessor, converting them to floating-point values, performing a floating-point division via reciprocal approximation, convert-

ing the floating-point result back to an integer, and then transferring the result back to the CPU. Even with this long series of operations, performance between that of 1 bit per cycle and 2 bit per cycle CPU hardware is provided by the coprocessor. This performance is provided at no additional hardware cost over that required for floating-point operations.

4. MEM Pipestage Tradeoffs

In the MEM pipestage the MultiTitan performs a cache access for memory reference instructions. Two interesting tradeoffs made in the MEM pipestage during the design of the MultiTitan were the size of the external data bus and the size and organization of the external cache.

4.1. 64-bit external data busses

A 64-bit external data bus can dramatically improve the performance of double-precision floating-point applications at relatively low cost. (The implementation cost of 64-bit data busses was discussed in Section 2.3.) Table 5 shows the performance improvement in four programs derived simply from the ability to perform 64-bit coprocessor loads and stores. The four programs were chosen to cover a wide range of floating-point intensive applications. For each benchmark the percentage improvement obtained was estimated by increasing the execution time of the benchmarks on a MultiTitan with a 64-bit data bus by an additional cycle for each coprocessor load and by two additional cycles for each coprocessor store. This assumes that the two coprocessor stores required to store a 64-bit quantity will have a store interlock between them.

benchmark	%loads	%stores	%improved
vector Linpack	38.1	20.3	61.9
Livermore loops:			
unrolled scalar	23.2	19.6	46.1
rolled scalar	12.4	8.4	24.7
Whetstones	9.4	6.0	14.3
average	20.8	13.6	36.8

Table 5: Improvement from 64-bit loads and stores

The MultiTitan FPU provides a simple vector capability [7]. Programs that use the vector hardware have increased needs for load/store bandwidth because computations can be effectively overlapped with loads and stores. The availability of vector operations reduces the inner loop of Linpack to not much else besides loads and stores. For the vector Linpack benchmark, the performance obtainable with 64-bit data busses is over 60% greater than that given 32-bit data busses. Scalar benchmarks vary in their load/store bandwidth requirements depending on whether their loops are unrolled by the compiler. For example, an unrolled version of the Livermore loops improves 46% when moving from 32-bit to 64-bit data busses, but the non-unrolled benchmark only improves by about 25%. Finally, some floating-point benchmarks such as Whetstones have a smaller percentage of floating-point loads and stores and benefit correspondingly less.

4.2. Choosing the external cache size and organization

Just as was the case for the on-chip instruction cache in Section 2.2, the size and organization of the external cache was dictated primarily by the RAM sizes available and by the system context.

There are two ways to structure a cache with regard to writes: write-through and write-back. A write-through cache sends all writes on to the memory system. A write-back cache only writes to main memory when a dirty line is replaced or flushed. The external cache of the MultiTitan is a write-back cache. A write-back cache was chosen over a write-through cache for several reasons:

- A write-back cache is a simpler design than a write-through cache of similar performance. A write-through cache requires a write buffer and its associated control logic if good performance is to be maintained, while a write-back cache does not need a write buffer.
- A write-back cache generates less bus traffic than a write-through cache, which was important since the MultiTitan was targeted to be an 8-processor multiprocessor.
- We planned to use the Titan memory system. A write-back cache was used in the Titan, so the Titan memory system was tailored for operation with a write-back cache. Also, software to manage the write-back cache (e.g., flush I/O buffers to main memory) was already in place.

Since the external cache is a write-back cache, the cache must be probed (i.e., checked for a hit) before it can be written. This is because the cache contains a unique copy of some data, so we must probe it before we write the cache so that no unique data is lost. This read (i.e., probe) of the cache takes place in the MEM pipestage for stores, just as load instructions read the cache in the MEM pipestage. If the probe of a store is successful, then the write of the store is performed in the WB pipestage.

The external cache of the MultiTitan is a physically addressed cache. Since it appeared that a physically addressed cache was not any harder to implement in a system than a virtually addressed cache, we chose a physically addressed cache to make the software easier. In order to obtain the fastest possible cache access, the address mapping was performed in parallel with the cache access by a TLB. To allow pages to be placed in any page frame in main memory, we restricted ourselves to a cache organization that only indexed the cache with unmapped address bits (i.e., page offset bits). This allows the TLB map to proceed in parallel with the cache access (see Figure 9), but requires the page size to be as large as the cache.

The TLB is implemented on the cache controller chip (CCU). The TLB has 512 entries, organized four-way-set associatively. With 64K byte pages this allows 32M bytes to be mapped at one time, which is much larger than most machines. For example, the WRL Titan TLB can map 4M bytes, and the R2000 [8, 11] TLB can map 256K bytes at one time. Unlike the direct-mapped nature of the caches, the TLB was made associative for several reasons. First, the TLB miss refill is performed in software. With our current software on the WRL Titan, this takes over 2,000 cycles, which is more than two orders of magnitude larger than a cache miss. Thus, while a program that uses two arrays of data that map to the same direct-mapped cache location might miss on every data reference and run up to 14 times slower, similar behavior with the TLB could result in programs running 2,000 times slower. A 4-way set associative organization allows the executing code fragment and three operands (e.g., A[i] := B[i] + C[i]) to overlap with-

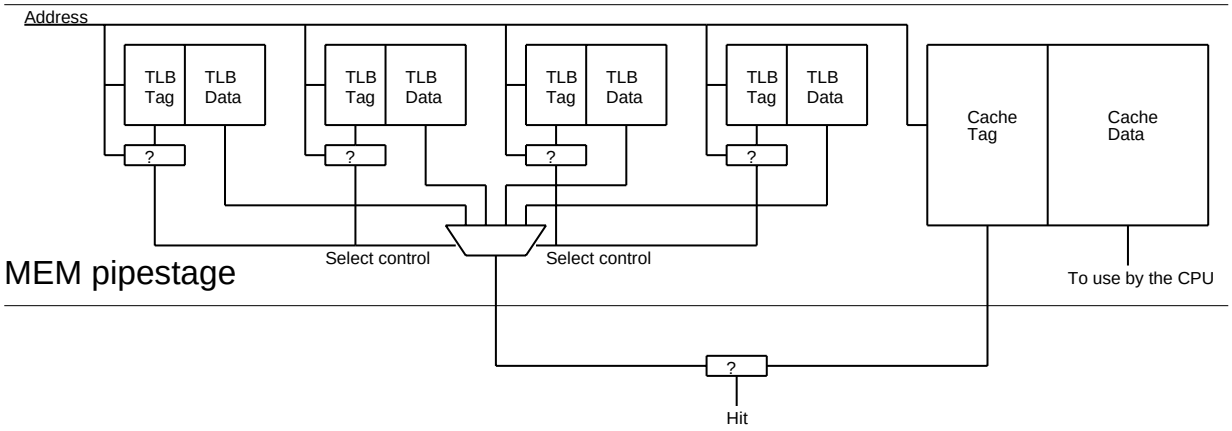

Figure 9: MEM pipestage timing summary

out thrashing in the TLB. Second, since the TLB and its comparators are on the same chip, and the custom TLB RAM access is faster than that of the external RAM parts, a four-way set-associative TLB can be built that operates in time for the comparison with the cache tag.

The MultiTitan external cache is a direct-mapped cache. Just as with the on-chip instruction cache, a direct-mapped organization was chosen because it has the shortest latency. In the external cache, the RAM access is performed in the MEM pipestage, but the tag comparison is not complete until well into to WB pipestage. By this time the data returned from a load instruction will have been written into the register file and possibly bypassed into an ALU operation which is almost complete. This allows the cycle time of the machine to be much closer to that of the fundamental RAM access time than if an associative organization was employed. Reduced cycle time is a very significant factor since it results in increased performance for all instructions, not just loads and stores.

In order for the cache latency to be as low as possible, we decided that only one row of 4-bit-wide RAM chips would be used for the cache. This minimized the capacitance of the address bus and improved the performance of the data bus relative to implementations with multiple chips per bit. The capacitance of the address bus can also account for a significant fraction of the cache access latency. By using 4-bit-wide chips, the loading on the address lines was cut to 1/4 that present if one-bit-wide memory chips were used. Since the largest fast static RAMs available when the MultiTitan was being designed were 16Kx4 20ns CMOS parts, this resulted in a 128K byte cache. (64 bits of data bus / 4 bits per chip = 16 chips for data, 16 chips x 8K bytes per chip = 128K bytes).

If the external cache were a mixed instruction and data cache, the resulting system would have 128K byte pages since the TLB map was in parallel with the cache indexing. This was judged to be a little too large, so the external cache was partitioned into a 64K byte instruction and a 64K byte data section. This partition required no extra chips: the high order address bit of the chips

was merely connected to a pin on the CPU which specified whether an access was an on-chip instruction cache miss access or a data reference.

Table 6 gives the results of simulations of a split cache consisting of two 64K byte segments, versus a mixed 128K byte cache. The conventional wisdom on mixed versus split resources is that a single shared resource of a given size is always better than two private resources each of 1/2 size. This is the observed behavior for most programs, but the PC board router had better performance with a split cache than a mixed cache. This is because the external cache is a direct mapped cache, and providing separate instruction and data sections provides a measure of added associativity in the cache. In other words, with a split cache data references and instruction references that map onto each other can coexist in the cache, whereas they can thrash between each other in a mixed cache. However, for most programs, the mixed cache performed better than the split cache. This was especially true for the numeric benchmarks. These have large data sets and spend most of their time in small loops. For example, a 100x100 Linpack has an 80K byte array. This fits in a 128K byte mixed cache but does not fit in the 64K byte data side of a split cache, so its split performance is much worse than its mixed performance. Since the numeric programs spend much of their time in small loops, the external instruction cache is rarely used by the numeric benchmarks.

Miss cost for all configurations is 14 cycles

	split: two 64KB			mixed	split/
benchmark	instr	data	total	128KB	mixed
ccom	.208	.045	.253	.229	1.10
PCBroute	.039	.114	.153	.209	0.73
TimingVerify	.013	.020	.033	.020	1.65
Linpack	.0001	.192	.192	.031	6.19
Livermore	.0004	.095	.095	.020	4.75
Stanford	.001	.001	.002	.002	1.00
average	.044	.078	.121	.085	1.42

Table 6: Split vs. mixed external cache CPI burden

Although the mixed cache clearly performs better than the split cache, in the MultiTitan we implemented the split cache. This is because the overall difference between the two methods averaged over the six benchmarks above is only .036 CPI. At our design target of 25 Mhz, this means that the split cache machine is less than 1 MIP slower (i.e., 3.6%) than the machine with a mixed cache. It was felt that the reduction in page sizes would result in better net overall system performance even though the cache performance was somewhat lower.

5. WB Pipestage Tradeoffs

All MultiTitan instructions commit in the WB pipestage. For example, even though ALU operations are computed two pipestages before WB, they are not written into the register file until WB. By having all instructions commit in WB, the pipeline control of the machine becomes very regular and is simplified. Another implication of the uniform commit of instructions in WB is that instructions that enter the WB pipestage will write their result registers, even if the result is incorrect. For example, load instructions write the register file in the WB pipestage before it is known whether or not they will have a page fault. This means that in order to recover from page faults, the base register for a load cannot be the same as its target. If it is and a page

fault occurs, the ability to calculate the address of the page fault will be lost. Floating-point operations also commit in the WB pipestage, even though they only begin execution the WB pipestage. Thus once a floating-point operation begins, it is guaranteed to complete no matter what types of interrupts may occur in the machine. Because the latency of all floating-point operations is three cycles, floating-point operations that abort will abort precisely relative to all other floating-point operations. Details of other floating-point tradeoffs are beyond the scope of this paper.

6. Conclusions

In this paper we presented some of the tradeoffs made during the design of the MultiTitan CPU. These tradeoffs were made to achieve the highest sustained performance with the lowest peak performance. In particular, many of the tradeoffs involved minimizing the latency of operations while simplifying the organization of the machine. Moreover, these tradeoffs were primarily driven by the available technology and system context.

The first IF pipestage tradeoff considered was the basic system partitioning of the design with regards to the CPU chip. By putting the instruction cache on-chip instead of the FPU or the TLB, we were able to maximize system performance while simplifying the system design. Second, a direct-mapped on-chip instruction cache organization was chosen because it had the lowest average latency when hit and miss latencies are combined with the probability of a miss. Third, by using a 64-bit bit data bus the refill latency was reduced by two cycles, improving the performance of the machine on our production programs by 8% at low cost. Fourth, by placing interlock detection, instruction decode, register fetch, and cache parity checking in parallel with the instruction cache tag comparison, the pipestage normally used for instruction decode in most machines could be eliminated. By eliminating this pipestage, the latency of branches is reduced, and hence the machine performance directly improves by 7% in machines with a single branch delay slot. The improvements as a result of second-order factors, such as simpler resulting control logic or fewer PC queue entries, are harder to quantify, but can be as significant.

The first tradeoff about the EX pipestage was where it should be in the machine. Based on simulations, placing it in the same pipestage as memory reference address calculations generates about the same number of interlock cycles as placing it concurrent with cache access. However, by combining the EX pipestage with the address calculation pipestage, the machine hardware was reduced by an adder and the bus, bypass, and control structure of the machine was simplified. Finally, in the MultiTitan we relied on high-performance hardware in the FPU chip for support of integer multiplication and division. This resulted in similar or better performance than methods with augmented CPU hardware but at negligible additional hardware cost.

Two MEM pipestage tradeoffs were discussed. First, the use of 64-bit busses were shown to dramatically improve double-precision floating-point performance by up to 60%. Second, the external cache was designed as a 128K byte direct-mapped cache, partitioned with 64K bytes for instructions and 64K bytes for data. This decision was driven primarily by the available static RAM technology and for the system desire to avoid 128K byte pages.

The most important feature of the WB pipestage is that it is when all instructions commit. This has implications for CPU load instructions (i.e., rr <> ra) and for floating-point operations. This regular commit framework helped permit a small, fast, and regular pipeline control structure to be designed.

7. Acknowledgements

Jeremy Dion was the first to simulate some of the cache and TLB tradeoffs, and was the chief designer of the CCU. Special thanks to David W. Wall for his help in providing the parameterizable MultiTitan simulator and code optimizer that was used for some of the tradeoffs in this paper. Many other people contributed in one form or another to the MultiTitan project, too many to list here. Jeremy Dion, John Ousterhout, Richard Swan, and David W. Wall provided many helpful comments on an early draft of this paper.

References

[1] Chow, P., and Horowitz, M.
Architectural Tradeoffs in the Design of MIPS-X.
In *The 14th Annual Symposium on Computer Architecture*, pages 300-308. IEEE Computer Society Press, June, 1987.

[2] Cray Research Inc.
The CRAY-1 Computing System Reference Manual.
Chippewa Falls, WI, 1976.

[3] Dobbs, C., Reed, P., and Ng, T.
Supercomputing on Chip.
VLSI Systems Design :24-33, May, 1988.

[4] Hennessy, J., Jouppi, N., Baskett, F., Gross, T., and Gill, J.
Hardware/Software Tradeoffs for Increased Performance.
In *First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2-11. IEEE Computer Society Press, March, 1982.

[5] Hill, M., et. al.
Design Decisions in SPUR.
Computer :8-22, November, 1986.

[6] Jouppi, N. P., Tang, J. Y. F., and Dion, J.
A 20 MIPS Sustained 32b CMOS Microprocessor with 64b Data Busses.
In *The 36th International Solid-State Circuits Conference*, pages 84-85. IEEE Solid State Circuits Council and the University of Pennsylvania, February, 1989.

[7] Jouppi, N. P., Bertoni, J., and Wall, D. W.
A Unified Vector/Scalar Floating-Point Architecture.
In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 134-143. IEEE Computer Society Press, April, 1989.

[8] Kane, G.
MIPS R2000 RISC Architecture.
Prentice-Hall, 1987.

[9] McFarling, S., and Hennessy, J. L.
Reducing the Cost of Branches.
In *The 13th Annual Symposium on Computer Architecture*, pages 396-403. IEEE Computer Society Press, May, 1986.

[10] Nielsen, M. J. K.
Titan System Manual.
Technical Report 86/1, Digital Equipment Corporation Western Research Lab, September, 1986.

[11] Rowen, C., et. al.
RISC VLSI Design for System-Level Performance.
VLSI Systems Design :81-88, March, 1986.

[12] Sachs, H., and Hollingsworth, W.
A High Performance 846,000 Transistor Unix Engine: The Fairchild Clipper.
In *Proceedings IEEE International Conference on Computer Design: VLSI in Computers*, pages 342-346. IEEE Computer Society Press, October, 1985.

WRL Research Reports

''Titan System Manual.''
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

''Global Register Allocation at Link Time.''
David W. Wall.
WRL Research Report 86/3, October 1986.

''Optimal Finned Heat Sinks.''
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

''The Mahler Experience: Using an Intermediate Language as the Machine Description.''
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

''The Packet Filter: An Efficient Mechanism for User-level Network Code.''
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

''Fragmentation Considered Harmful.''
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

''Cache Coherence in Distributed Systems.''
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

''Register Windows vs. Register Allocation.''
David W. Wall.
WRL Research Report 87/5, December 1987.

''Editing Graphical Objects Using Procedural Representations.''
Paul J. Asente.
WRL Research Report 87/6, November 1987.

''The USENET Cookbook: an Experiment in Electronic Publication.''
Brian K. Reid.
WRL Research Report 87/7, December 1987.

''MultiTitan: Four Architecture Papers.''
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

''Fast Printed Circuit Board Routing.''
Jeremy Dion.
WRL Research Report 88/1, March 1988.

''Compacting Garbage Collection with Ambiguous Roots.''
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

''The Experimental Literature of The Internet: An Annotated Bibliography.''
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

''Measured Capacity of an Ethernet: Myths and Reality.''
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

''Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.''
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

''SCHEME->C A Portable Scheme-to-C Compiler.''
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

''Optimal Group Distribution in Carry-Skip Adders.''
Silvio Turrini.
WRL Research Report 89/2, February 1989.

''Precise Robotic Paste Dot Dispensing.''
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

''Simple and Flexible Datagram Access Controls for Unix-based Gateways.''
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.''
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.''
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

''A Unified Vector/Scalar Floating-Point Architecture.''
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.''
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.''
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.''
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

''Leaf: A Netlist to Layout Converter for ECL Gates.''
Robert L. Alverson and Norman P. Jouppi.
WRL Research Report 89/12, July 1989.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

Table of Contents

List of Figures

List of Tables