Proceedings of CS292i: Implementation of VLSI Systems

TABLE OF CONTENTS

SPUR ARCHITECTURE DESIGN RATIONALE

The SPUR Hardware Design Team
Randy H. Katz, Editor
Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT: The SPUR ("Symbolic Processing Using RISCs") Project is a research effort aimed at applying reduced instruction set computer concepts to the support of LISP programming environments. In addition, the project extends previous Berkeley RISC efforts in the direction of multiple processor and co-processor support. The objective of the project is to build a complete prototype *computer system*, including extensive software development. This document records the implementation issues and decisions made during the design of the hardware prototype.

KEY WORDS AND PHRASES: Reduced Instruction Set Computer, Tightly Coupled Multiprocessor, Tagged Architecture, LISP Machine, Floating Point Coprocessor;

1. Introduction

This document presents a description of and rationale for the design of the SPUR ("Symbolic Processing Using Riscs") multiprocessor. SPUR is intended to be a workstation-sized computer system incorporating custom-designed processor nodes and off-the-shelf memory and I/O boards. Its unique features include: (1) a custom-designed VLSI processor chip, implementing a 40-bit tagged architecture and a load-store/register-register instruction set that is well-suited for pipelined execution, (2) a VLSI memory/cache management chip, implementing a multiprocessor cache consistency protocol as well as virtual memory management, (3) a custom designed floating point coprocessor that, combined with software routines, implements the I.E.E.E. standard, and (4) an unusually large (128K bytes) cache associated with each processor node. Necessity is the mother of invention, and other features of the design have been influenced by implementation constraints. For example, to avoid the hardware design of a translation buffer, page table entries are buffered in the processor cache. Evaluation studies have shown that this is actually more effective than systems with conventionally sized translation buffers! Another example is the SPUR bus, which is based upon a dual system bus, one for cache to cache communication and another for communication among the caches and global memory and I/O devices. A large software development effort to create the associated operating and programming systems is also underway.

The SPUR design has evolved from previous VLSI processor design efforts undertaken by faculty and graduate students at the University of California, Berkeley over the past five years [PATT85]. Led by Professors David Patterson and Carlo Sequin, the first generation processor was called RISC [PATT82]. The processor implemented a "reduced instruction set," i.e., one with a small number of instructions, each of which could be executed in a single processor cycle. This made possible a simple and fast controller design. The pipelined implementation of the architecture, called RISC-II, was able to achieve an execution rate of 3 million instructions per second. The other architectural feature that distinguished the Berkeley RISC processor from other reduced instruction set computers (e.g., Stanford MIPS [HENN84] and IBM 801 [RADI83]) was its large on-chip register file. This made it possible to reduce the number of off-chip data references, significantly improving the processor performance [PATT82].

The RISC architectures were influenced by conventional programming languages, such as C and PASCAL. The second generation processor, called SOAR ("SmallTalk on a RISC"), applied RISC-concepts to the radically different programming style of object-oriented systems [UNGA84]. SOAR's most significant architectural enhancement is its use of tagged words to encode type information, and a trap mechanism to support dynamically typed data in SmallTalk-80. The tags also include "generation bits" for implementing an efficient garbage collection scheme, called *Generation Scavenging* [UNGA85]. SOAR designs have been completed in NMOS and P-Well CMOS technologies, and chips have been submitted for fabrication.

SPUR is the latest generation of VLSI processor designs undertaken at Berkeley, and the most ambitious to date [KATZ85a, KATZ85b]. It extends the previous efforts along several dimensions. First, the processor is being designed as a component of a complete multiprocessor system, including operating system and programming environments. Second, an on-chip instruction buffer (an instruction cache) will improve processor performance by further reducing off-chip references. Third, a companion memory management chip is also being designed for implementation in VLSI. Fourth, a coprocessor interface will be supported by the processor chip, and a companion floating point coprocessor is being designed to explore that interface.

A key hypothesis of the project is that RISC architectures can provide high performance for LISP applications while maintaining compatibility with conventional procedural languages. (We expect the operating system to be written in C.) There already exist a number of successful implementations of architectures that support "symbolic processing," such as the Symbolics 3600, the LMI Lambda, and the Texas Instruments Explorer. These machines have been customized for LISP support through microcode. On the other hand, SPUR is a general purpose processor, executing a reduced instruction set directly.

The rest of this paper is organized as follows. In the next section, the important design constraints are described. Section 3 describes the design of the processor, while section 4 describes the combined memory management unit/cache controller/bus controller, and section 5 discusses the coprocessor interface and the floating point unit. The processor board design is described in section 6. Conclusions and status is given in section 7.

2. Design Constraints

Our objective is to build a complete multiprocessor computer system. To accomplish this goal, we must place some constraints on the design to reduce its scope and complexity. In particular, we have limited the design effort to the level of the processor node, leveraging off existing memory and I/O devices whenever possible.

(1) THE DESIGN EFFORT IS LIMITED TO A PROCESSOR BOARD.

Off-the-shelve memory boards and I/O controllers will be used, rather than designing these ourselves. This means that the system must be built around an existing and reasonably popular system bus. We have chosen the Texas Instruments NuBus for a number of reasons. First, the bus is a state-of-the-art microcomputer bus, based on a simple protocol that supports multiword transfers of 32-bit words, with a 37.5 Mbyte/second peak bus bandwidth[1]. Second, the form factor of the board conforms to the Eurocard format, which is emerging as an industry standard. Backplanes,

[1]This is achieved only with the maximum transfer blocksize. We expect to achieve a maximum peak rate of 35.5 Mbytes/second.

cabinets, and other "hardware" are readily available from several sources, and the boards themselves are generously sized (approximately 11" by 14"). Third, the bus electronics requires only one of the three 96-pin DIN connectors available on the board, leaving the remaining two connectors for user defined buses. The rack mount NuMachine provides a 21-slot card cage with a mixture of NuBus and MultiBus slots, providing a prototype processor like SPUR with access to NuBus native memory and peripheral boards as well as the large number of peripherals available for the MultiBus.

This constraint has effected the processor node's interface to the system bus. The SPURBUS is actually implemented as a two tightly coupled buses (Memory and Cache) implementing the same protocol and sharing the same arbitration logic. A SPURBUS transaction is implemented by tandem transactions on the underlying buses (they **cannot** be used independently). The Memory Bus, implemented directly by the TI NuBus, provides a communications path between the processors and shared resources such as global memory and I/O devices. It uses physical addresses. The other bus, called the Cache Bus, provides a parallel communications path among the processors' caches. Virtual addresses are used on this bus. Note that the SPUR processor caches are accessed with virtual addresses, thus eliminating virtual memory translation overhead on cache accesses. Maintaining cache coherency is much easier with virtual addresses on the system bus side, since the cache can be accessed with the same hardware from either side. Otherwise, special reverse translation hardware is needed to map physical bus addresses into virtual addresses that can be used to access the cache. A system with virtual addresses throughout would be ideal, but would require the design of our own memory boards with on-board translation hardware.

(2) **PROCESSOR NODE CHIP COUNT IS KEPT SMALL**

The processor node design avoids SSI logic wherever possible by placing as much logic as possible on the custom VLSI chips. However, to obtain the amount of local processor node memory we need for adequate performance, we must make use of commercial RAM technology. Some driver circuitry, comparison logic, and so on must be placed off-chip because of pin limitations on our chip packages. The two major custom designs involve the processor and a companion memory/cache management unit with on-chip bus interface. A third design is a floating point co-processor. The cache size is 128K bytes, organized as 4K x 32 byte transfer blocks. The cache is implemented by 64 16Kx1 SRAM chips for the cache data, 4 4Kx4 SRAM chips for the physical tags, 12 4Kx4 RAM chips for the virtual tags, and 2 4Kx4 RAM chips for the state bits.

3. Central Processor Design

3.1. Instruction Buffer

One of the major departures from RISC II is the incorporation of a prefetching instruction buffer on the processor. The instruction buffer is organized as a 128 word (512 byte) cache, grouped into 16 blocks of 8 words each. Valid bits are associated with each word, thus making the subblock size one word. The small number of blocks makes block selection fast. We chose the large blocksize because it makes prefetching more effective while simultaneously reducing the size of the address tag array.

The prefetching algorithm proceeds as follows. If a processor instruction fetch misses in the instruction buffer, a "miss opcode" is delivered to the pipeline and a memory access is initiated for the target instruction. The instructions following the target are prefetched, as long as they fall within the same block and as long as such fetches do not result in misses in the off-processor cache. In-progress prefetching is aborted immediately if the processor makes another instruction fetch that misses in the instruction buffer.

3.2. Execution Pipeline and Control

An ideal pipeline organization for a register-oriented machine has three stages: instruction fetch, register read and execute, and register write. However, LOAD/STORE instructions take longer because of memory access delays to the on-board cache. RISC II adopted a three stage pipeline structure for REGISTER/REGISTER instructions and a four stage "stretched" pipeline for LOAD/STORE, i.e., instructions already in the pipeline were stalled while a load or store waited for completion of the memory access.

The SPUR pipeline is a departure from RISC II in that all instructions use the same four stage pipeline:

(1) Instruction Fetch
(2) Register Read/Execute OR Effective Address Computation
(3) Memory Access (if load/store operation)
(4) Register Write

The pipeline control is simplified because all instructions behave the same way. However, the disadvantage is slightly more complicated forwarding logic: the register read stage may need the result of either of the TWO previous instructions, neither of which will have written back to the register file in time.

A conservative four phase non-overlapping clocking scheme has been selected for the SPUR datapath. Each stage is divided into four phases of 25 ns each, with a 10 ns non-overlap time, yielding an estimated cycle time of 140 ns. The scheme is well suited for the register file, which is read and written once per cycle. Precharged bit lines and decoders are used to achieve reasonable performance while avoiding complicated sense amp circuit design.

3.3. CPU Control

The control signals and their sequencing for all instructions have been determined, but the detailed implementation approach is unresolved at this time. We are investigating a two level decoding scheme, in which a master control finite state machine (most likely implemented by a PLA) sequences through high level operations that are locally decoded into detailed control signals (most likely implemented by random logic).

3.4. Exceptions, Faults, and Interrupts

Exceptions are unusual conditions detected inside the CPU or the FPU (e.g., integer overflow, window overflow, etc.). Faults are unusual conditions outside of the CPU/FPU that occur during the execution of an instruction, such as a page fault. Interrupts are unusual conditions outside of the CPU/FPU that occur asynchronous to the execution of an instruction, such as an I/O interrupt.

An exception appears as though the currently executing instruction has been replaced by a CALL instruction. The write stage of the executing instruction that caused the exception is cancelled.

4. Memory/Cache Management Unit Design

Cache memory is an implementation technique for reducing main memory access time, by placing a high speed memory buffer close to the processor. This is particularly important in a tightly coupled multiprocessor system such as SPUR, since our pathway to memory is through a relatively slow microcomputer bus. Futhermore, a cache can reduce the system bus bandwidth required by a processor. In a multiprocessor system, this reduction will often be more important than the reduction in main memory access time [GOOD83]. [KATZ85b] showns that the cache must be very large for reasonable multiprocessor performance to be achieved. Thus, a large cache (128K bytes in the current design) is associated with each processor node. The SPUR programming model is a single, large, potentially shared memory; there is no programmer managed local memory. However, special instructions have been included in the instruction set for manipulating the contents of the cache.

4.1. Cache Coherency Protocol

A multiprocessor system with caches must maintain *cache coherency*, i.e., all processors must have a consistent view of memory. In other words, all cached copies of a memory block must contain the same data, and main memory must eventually be updated with changes made to cache blocks. One common solution is to implement a *write-through* strategy. Processor writes are written through to main memory, and as a side effect, copies of the block in other caches are invalidated. Thus, the memory and cached versions of a block are always identical. In particular, no two copies of a cached block can be different, and thus consistency is maintained.

The problem with adopting such a strategy is that each processor write requires a bus transaction, thus consuming more bandwidth of the system's most critical resource. The effective miss ratio is equal to the fraction of accesses that are writes – approximately 15 - 30%. These miss ratios are much worse than what could be achieved for a uniprocessor with the size of the cache we are implementing. Thus, we have have developed a new cache consistency protocol, called *Berkeley Ownership*, whose goal is to minimize the additional bus traffic needed to maintain cache consistency.

The underlying principle of the Berkeley Ownership Protocol is that a cache must explicitly *own* a memory block before it is permitted to update the block. The protocol guarantees that a memory block has only one owner at a time. It also makes it possible for the cache controller to know when it can update its cache without needing to communicate with other caches or with memory. However, the owner is responsible for eventually copying the block back to memory.

Since memory blocks are now stored in the cache, instead of written through, the owner must also respond to read requests made by other caches on behalf of their processors. A distinction is made between acquiring a block for reading, and acquiring it with the intention of ultimately updating it. Ownership is transfered on the latter operation. The details of the protocol are reported in [KATZ85a].

The advantage of the Berkeley Protocol over other similar protocols is that it attempts to transfer blocks between caches directly whenever possible, and only updates memory when a dirty block is replaced in the cache. For example, the Synapse Protocol [FRAN84] performs many transfers through main memory. A read request is first aborted by the owning cache, which then gives up ownership while transfering the block back to memory. A short time latter, the restarted read request will receive the block from main memory. In the Berkeley Protocol, the block is transfered to the requesting cache in a single bus transaction. Our studies have shown that for several representative reference traces, the Berkeley Protocol reduces bus traffic by from 20 -- 30% over other similar protocols.

One might observe that sharing of data across processors in existing multiprocessor applications is rare, and thus unworthy of hardware support. By placing shared data in non-cacheable pages and/or providing some processor instructions for atomic test-and-set and cache flushing, data sharing can be managed by software. For example, access to shared data is through critical sections enforced by software: (1) locks are acquired at the beginning of the section, (2) the data is accessed, (3) when complete, the cache is flushed to make memory consistent again, and (4) the locks are released. However, SPUR has been designed as a vehicle for experimenting with developing shared data applications. As such, we do not want to make it expensive or difficult for programs to share data among processors. Thus, we have adopted the more complicated hardware solution. We have been able to show that for write-shared data, the additional bus utilization associated with our protocol is no worse than with non-cacheable pages, and is significantly better for read-shared data. In addition, flushing caches of our size would significantly reduce performance.

The arbitration logic of the system bus guarantees that at most one of the possible masters of the bus has control of it at a time. Since every processor is connected to the bus, bus mastership can be used as a system wide semaphore for synchronization purposes. A significant advantage of the Berkeley Protocol is that it makes possible an atomic test-and-set operation in the cache, thus avoiding bus arbitration and transfer latency in many cases. When a test-and-set operation is applied to a memory block, the block is read and held until it has been updated by the cache controller. In other words, the cache controller refuses to give up the block until the test-and-set has completed.

4.2. Virtual Memory Management

For high performance, it is useful if a cache can be addressed with virtual addresses from the processor side. This allows it to be accessed directly from the processor, without adding the additional latency of virtual-to-physical address translation to the cache access time. However, a virtually addressed cache suffers from the *synonym problem* in that two virtual addresses may map to the same physical address. Synonyms are a serious problem for cache update unless the virtual addresses can be constrained to map into the same cache entry.

The usual solution is to place translation between the processor and the cache (or in parallel with cache access), and to access the cache with physical addresses. As mentioned above, this is undesirable from a performance viewpoint. The problem is solved in SPUR by outlawing synonyms in the address space. The unit of sharing is not the page, but the segment. The system supports 256 active segments, each of which is one gigabyte in length. A global virtual address is 38 bits: 8 bit segment identifier plus a 30 bit byte offset. A given process can access four segments. The processor generates short virtual addresses that are 32 bits: 2 bit short segment identifier plus a 30 bit byte offset. The cache controller contains software maintained control registers that map the short segment identifier into a full global segment identifier. The cache is addressed with global virtual addresses. The placement of pages within segments is the same for all processes which make use of that segment, although they may use different short segment identifiers.

Because of the size of the address space, the segment page tables themselves are quite large. Even with 4K byte pages, a page table requires 256K entries. Assuming 4 bytes per entry, this would represent a megabyte of contiguous memory per segment!

The most obvious way to avoid contiguous allocation of physical memory is to place the page tables in virtual memory, allowing them to be paged. To make it possible to bootstrap the mapping process, a small "root" portion of the page tables are maintained in physical memory. The mapping mechanism works as follows. The requested word's virtual address is used to form the offsets into the various tables; the entries always contain a presence bit and a physical address. *Segment page tables* are placed in contiguous virtual memory. It is likely that they will be allocated by the operating system in a segment reserved for system data. A *root page table map* is located in a fixed location in physical memory. Each entry (one per segment) holds 256 pointers to *page table pages*. Each page table page holds 1024 *page table entries*. In turn, each page table entry maps 4096 bytes of address space ($2^8 * 2^{10} * 2^{12} = 2^{30}$). The root page table map, which must remain resident in physical memory, only requires 256K bytes of contiguous physical memory to begin the mapping process for the 2^{38} byte (256 gigabyte) address space. The two level scheme allows page tables to be allocated incrementally.

Our method for avoiding contiguous allocation of page tables leads to another potential problem: the cost of performing the multiple layers of translations. In the worst case, it may be necessary to first access the root page of the system virtual space, then the page of the system virtual space page table referenced by the root page, then the root page of the user segment's page table in the system virtual address space, then the page table entry for the referenced page, and finally the block on the referenced page itself.

The SPUR cache is accessed with virtual addresses, thus virtual memory translation is only necessary in the rare case of a cache miss. Nevertheless, intermediate translations must be cached if reasonable performance is to be achieved. Conventional systems cache their translations in a special append only associative buffer called the Translation Lookaside Buffer or *TLB*. One of the unique features of SPUR is that page table entries have virtual addresses and are thus cacheable just like any other data We avoid using special purpose hardware by using the processor's cache for translation entries as well as conventional data and instructions. Our studies [RITC85] have shown that our caches are sufficiently large so that there is little interference among instructions/data and page table entries: placing page table entries in the cache has a negligible affect on cache miss ratios. An advantage of the approach is that the page table entries fall under the same cache consistency protocol as any other cached data. Thus, changes to page table entries are guaranteed to leave all copies of the page table in a consistent state. Furthermore, the caches are large enough so references to page table entries rarely miss in the cache.

Since SPUR does not have a single translation buffer, but rather a buffer distributed among the processors' caches, we need a distributed algorithm for maintaining the REFERENCE and DIRTY status of pages. Normally, this information would be maintained in the TLB. Performance would be seriously affected if each data reference also required a cache look up and write to set the REFERENCE and DIRTY status information in the page table entries. Observe first that REFERENCE information is used as a hint to the operating system to approximate "least recently used" page replacement with "not recently used" replacement. We conjecture that some inaccurracy in the REFERENCE information can be tolerated without significant effect on performance. In SPUR, the page table entry REFERENCE bit is checked whenever a reference to a block on a page results in a miss. If the bit is not already set, the cache controller traps to software to set the bit. Inaccuracies can be introduced after a page table entry's reference bit is reset by the operating system. Blocks already in the cache may continue to be referenced, but the REFERENCE bit is not set because the accesses are not missing in the cache.

In a similar way, DIRTY bit information is a performance hint to the operating system. In the simplest case, all replaced pages that are writable could be written back to disk (read-only pages need not be written). This may not be such an awful option as it sounds, especially if large amounts of physical memory are available and pages are rarely replaced. However, SPUR does provide a mechanism for recording the DIRTY status of pages. It works as follows. Each cache entry contains two status bits: one to indicate that the page that contains the block is dirty, the other to indicate that the block itself is dirty (the latter status bit would be needed anyway, to determine which cache entries to write back to memory on a cache flush). When a block is first read into the cache, its associated page table entry must first be accessed. The PTE's page dirty bit is copied by the cache controller is copied into the cache state's page dirty bit. When a write is directed to a cached block for which the page dirty bit is NOT already set, the cache controller traps to software to set the PTE's page dirty bit. Once the page is marked as dirty, subsequently accessed blocks will have the cache state's page dirty bit set correctly. The first update to a block that had been accessed before the first page write will trap to set the PTE dirty bit, even though this is redundant. Our simulations have indicated that such redundant traps occur relatively infrequently [RITC85].

4.3. Bus Controller and Interrupt Mechanism

The bus controller interfaces the SPURBUS and the processor cache controller finite state machine. Some complexity is introduced because the bus clock, upon which the bus controller FSM is based, and the processor clock, which controls the cache controller FSM, are asynchronous. Some interface signals are handled by fully interlocked exchanges, while others are handled by latched signals with polling.

The interrupt register maintained by the bus controller is controlled by an asynchronous handshake. It is examined by the cache controller to determine whether an interrupt should be signalled to the processor. On the other hand, the bus controller may set it in response to memory mapped interrupts occuring on the SPURBUS. The cache controller asserts an inhibit line while reading the register, releasing the line when the read completes.

The bus controller deals with all interfacing to the backplane, while the cache controller interfaces with the on-board data paths. For example, the processor cache controller generates the block address, but the bus controller generates the rest of the address that indicates the bus transfer blocksize. The interface between the two is based on a request/acknowledge handshake. The interface between the bus controller and the snoop is based on wakeup/timeout. That is, the bus controller wakes up the snoop controller when a snoopable event is detected on the bus, and waits for the reply, aborting if time has run out. This is done because a snoop response can only be bounded; it is not possible to know in advance how long the response will take.

4.4. Cache Manipulation Instructions

5. Co-Processor Interface and Floating Point Unit

5.1. Co-Processor Interface

VLSI technology has not yet achieved levels of integration that would allow a complete computer system to be fabricated on a single chip. The approach taken by commercial microprocessors has been to package some system functionality in a chip separate from the central processor, and to integrate that functionality into the system through a coprocessor interface. For example, the DEC MicroVAX II system employs a floating point coprocessor in conjunction with the main cpu. When the cpu encounters a floating point point instruction, the floating point unit takes over execution while the cpu is suspended. Alternatively, the Motorola 68020/68881 coprocessor interface permits the cpu to continue executing non-floating point instructions during floating point operations, but only under special conditions. Typically, there is a significant amount of overhead consumed in resynchronizing the asynchronous execution. The commercial co-processor interfaces either make it expensive to communicate with the co-processor or do not attempt to exploit the parallelism available by having more than one execution unit in the system.

The primary goals of the SPUR co-processor interface is to provide a very low overhead communications interface to the co-processors, while making it possible for both the processor and its co-processors to be in simultaneous execution. This is accomplished by incorporating a separate instruction bus into the system that provides a communications path between the processor and the co-processors. This bus is necessary because the processor has its own on-chip instruction buffer and does instruction prefetching. Thus, it is not sufficient for the co-processors to merely monitor instructions entering the processor from the processor board cache. As soon as an instruction enters the central processor's execution pipeline, it is forwarded out along the instruction bus to the co-processors. The instruction set format has been formulated to make it easy for the processor's decode stage to determine that it has a co-processor instruction. In such cases, the execution pipeline performs no functions for this instruction (this is not quite true for conditional branches that test conditions in the co-processors).

Perhaps the greatest performance advantage of the co-processor interface comes from how operands are provided to the co-processors. Most co-processor interfaces provide explicit instructions for moving co-processor operands from the processor's registers to the co-processor. In many cases this involves staging the data into the processor only to move it to the co-processor, with no processor computations performed on the data other than traffic control. In SPUR, loads and stores are executed identically for the processor or a co-processor. Consider a load operation. Effective address computations are performed by the processor and sent to the processor board cache. When data is returned, it is latched into the appropriate co-processor's register file, rather than the processor's.

Further, the interface has been formulated to allow simultaneous execution of the processor and the co-processor. Consecutive co-processor operations force the processor to stall only in the case that the co-processor is still busy executing a co-processor instruction.

5.2. Floating Point Unit

Because of pin limitations, we have not chosen to implement a completely general co-processor interface for SPUR, although the issues remain under investigation. The design has been tailored for the floating point co-processor. For each floating point instruction encountered by the processor, a co-processor op code and three register specifiers are transmitted to the floating point unit for decode and execution. The processor will stall if the floating point unit is busy, and cannot yet accept a new instruction.

From the viewpoint of communications, very high performance can be achieved with the floating point unit because (1) the pathway between the cache and the fpu is 64-bits, and (2) the fpu register is dual ported. The wide pathway between memory and the fpu makes it possible to transfer data into and out of the fpu with a minimum number of instructions. The dual ported register file organization makes it possible to overlap floating point loads and stores while the fpu is doing computation. To keep the pipeline implementation straightforward, the code must be appropriately scheduled at compile time to avoid pipeline hazards. This is often called *software pipeline interlocks*.

6. Processor Board Design

The processor board contains the three chips that constitute the processor (CPU, memory management unit/cache controller, floating point co-processor) and the cache state, tag, and data RAMs. The cache is a virtual address cache organized into 4096 direct mapped blocks of 32 bytes each, for a total size of 128K bytes. To provide fast responses to system as well as processor bus actions, the cache state and virtual tags are dual ported. This is implemented by providing two sets of RAMs that can be read independently, but must be written at the same time. Even though the cache is accessed with virtual addresses, the tag store also includes a set of single ported physical tags. This is used to replace blocks from the cache to global memory without invoking memory translation.

Much of the SSI logic on the board deals with multiplexing data between the SPUR bus, the cache, and the processor and floating point unit. The cache data is organized into 16K x 64 bit words. When read or written from the system bus side, the appropriate high or low order 32-bit portions of the cache word are accessed.

The bus on the processor side is 64 bits wide. Loads and stores on behalf of the floating point unit access the full 64 bits. The processor accesses 40-bit or 32-bit words depending upon whether it is executing in tagged or non-tagged mode. In non-taged mode, the desired word is placed on the low order 32 bits of the processor bus. This requires some multiplexing logic on the board for the 32 bit word in the high order portion of the cache word. In tagged mode, the low order 40 bits of the addressed cache word are accessed by the processor. Thus, in non-tagged mode, the cache can hold 32K x 32 bit words, and can be accessed by even or odd processor word addresses. In tagged mode, the cache holds 16K x 40 bit words, and only even addresses are output by the processor. Tagged and non-tagged blocks can be co-resident in the cache.

We considered end-to-end generation and checking of parity. This would involve parity checking (on reads) and generation (on writes) between the processor and the cache, and between the cache and the SPUR Bus. While the NuBus supports optional word parity checking, we subsequently discovered that the existing memory or I/O boards for the NuBus ignore parity altogether. Thus, we have decided not to do parity checking on the processor board.

7. Summary and Conclusions

We have described the architectural design rationale of an ambitious multiprocessor workstation project currently underway at the University of California, Berkeley. At this point, the major architectural decisions have been made, pieces of the implementation have been completed, but considerably more work is necessary to complete the circuit level design and implementation. We expect to have a prototype workstation for debugging purposes by the summer of 1986.

Even at this stage of the project, several important architectural contributions have been made: (1) a new cache coherency protocol has been defined and is actually being implemented within a custom VLSI cache controller chip, (2) a virtual memory mechanism, based on cacheable page table entries, has been shown to be superior in performance to more conventional translation buffer mechanisms, (3) a coprocessor interface has been defined that permits concurrent and cooperative execution of a floating point coprocessor with very low overhead, and (4) a "reduced" floating point unit has been designed that provides excellent I.E.E.E. Standard floating point performance in a straight forward implementation.

8. Acknowledgements

A project of the size and complexity of SPUR cannot be undertaken without the help of many dedicated individuals. The core architectural design team consists of B. K. Bose, Susan Eggers, Garth Gibson, Paul Hansen, Mark Hill, Shing Kong, George Taylor, and David Wood. Joan Pendleton, primary microarchitect of the NMOS SOAR chip, made many key contributions to the early machine specification, based on her experiences with SOAR. Glenn Adams, John Keller, Corinna Lee, Daebum Lee, Scott Ritchie, Trudy Stetzler, and Tom Wisdom are masters students associated with the project. Chien Chen, Rich Rudell, Li-fai Pan, Albert Wang, Don Webber, and Sinohe Villapando contributed to the project through their participation in an intensive implementation seminar offered in Spring 1985. We made extensive use of the Berkeley CAD environment, made possible through the work of Professors Richard Newton, John Ousterhout, and Alberto Sangiovanni-Vincentelli and their students. The SPUR hardware design and implementation is being directed by Professors David Hodges, Randy Katz, and David Patterson. The operating system design effort is being lead by John Ousterhout, while the LISP system is being designed by a team led by Richard Fateman and Paul Hilfinger. Research funding has been provided by the Defense Advanced Research Project Agency, under the Infrastructure Program of the Department of Defense's Strategic Computing Initiative, administered by Paul Losleben.

9. References

[FRAN84] Frank, S., "Tightly Coupled Multiprocessor System Speeds Memory Access

Times," *Electronics,* (January 12, 1984).

[GOOD83] Goodman, J., "Using Cache Memories to Reduce Processor--Memory Traffic," 10th Annual Symposium on Computer Architecture, Trondheim, Norway, (June 1983).

[HENN84] Hennessy, J. L., "VLSI Processor Architecture," *IEEE Transactions on Computers,* V C-33, N 12, (December 1984).

[KATZ85a] Katz, R. H., S. J. Eggers, D. A. Wood, C. L. Perkins, R. G. Sheldon, "Implementating a Cache Consistency Protocol," 12th Annual Symposium on Computer Architecture, Boston, MA, (June 1985).

[KATZ85b] Katz, R. H., S. J. Eggers, G. A. Gibson, P. M. Hansen, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, D. A. Wood, D. A. Patterson, "Memory Hierarchy Aspects of a Multiprocessor RISC: Cache and Bus Analyses," U. C. Berkeley Computer Science Report No. UCB/CSD 85/221, (January 1985).

[PATT82] Patterson, D. A., C. H. Sequin, "A VLSI RISC," *IEEE Computer Magazine,* V 15, 9, (September 1982).

[PATT85] Patterson, D. A., "Reduced Instruction Set Computers," *Comm. ACM,* V 28, N 1, (January 1985).

[RADI83] Radin, G., "The 801 Minicomputer," *IBM Journal of Research and Development,* V 27, N 3, (May 1983).

[RITC85] Ritchie, S. A., "TLB for Free: In-Cache Address Translation for a Multiprocessor Workstation," U. C. Berkeley Computer Science Division Report No. UCB/CSD 85/233, (May 1985).

[UNGA84] Ungar, D., R. Blau, P. Foley, D. Samples, D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," *Proc. 11th Annual Symposium on Computer Architecture,* Ann Arbor, MI, (June 1984).

[UNGA85] Ungar, D., Ph.D. Dissertation, U. C. Berkeley, in preparation.

SPUR Instruction Set Architecture

George S. Taylor

Computer Science Division
Electrical Engineering and Computer Sciences Department
University of California
Berkeley, California 94720

Version 9: September 12, 1985

1. General features

The SPUR instruction set architecture extends RISC-I and RISC-II to include runtime tag checking and floating point arithmetic. Tag checks are performed in parallel with data operations in order to run Lisp programs faster. Floating-point arithmetic gives SPUR comparable performance on arithmetic programs to what it achieves on non-arithmetic ones.

SPUR retains a number of features from earlier Berkeley RISC processors:

- Only load and store instructions reference memory.
- Addresses are expressed in terms of bytes, although loads and stores transfer four-byte words aligned on four-byte boundaries.
- All instructions are four bytes long.
- Overlapping register windows are provided with 32 registers visible in each window.

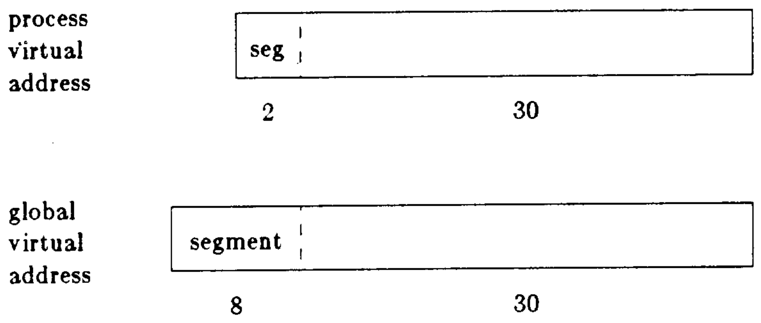
SPUR's extensions to RISC-II include the following:

- The general purpose registers are 40 bits wide: 6-bit type tag + 2-bit generation number + 32-bit data.
- The 32-bit virtual address space of each process is divided into four segments to allow sharing among processes on a per-segment basis.
- There are 15 floating-point registers, which are separate from the integer registers and wide enough to hold IEEE standard extended precision floating-point numbers.
- Integer and floating-point instructions can execute in parallel.

2. Virtual address space

The SPUR system supports 38-bit global virtual addresses composed of an 8-bit segment number and a 30-bit byte offset within each segment. A process uses 32-bit virtual addresses composed of a 2-bit short segment number and a 30-bit byte offset. A process can access four of the 256 global virtual segments by mapping from each of its short segment numbers to one of the global segment numbers. The mapping makes it possible to share segments among processes or to have processes that are entirely independent of each other.

The kernel segment is short segment number 0 (implicit in the CALL_KERNEL instruction and the trapping mechanism). This does not mean that all processes share a single kernel segment, however, since the mapping from short segment 0 into a global segment number is determined by the four segment registers.



2.1. Model for sharing instructions and data

A shared object exists at a single location within SPUR's global virtual address space. Virtual synonyms are not allowed. However, SPUR cân still support dynamic linking. All of the objects shared among a group of processes exist in one or two segments that are shared by all of the processes. We chose not to support the fine-grain level of sharing avaiable in Multics[]. That would have required a much larger number of segments or the ability to handle virtual synonyms.

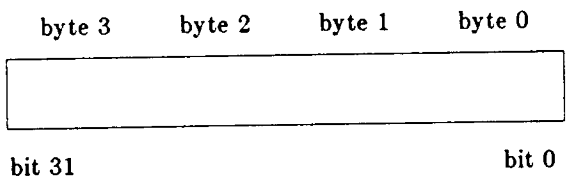
3. Data types

SPUR memory consists of four-byte words and eight-byte doublewords. These are aligned on four byte and eight byte boundaries, respectively, in terms of byte addresses.

data types in memory	
four-byte:	integer single precision floating-point
eight-byte:	double precision floating-point extended precision floating-point (two parts) tagged Lisp pointers tagged Lisp immediate integers and characters

3.1. Bit and byte numbering convention

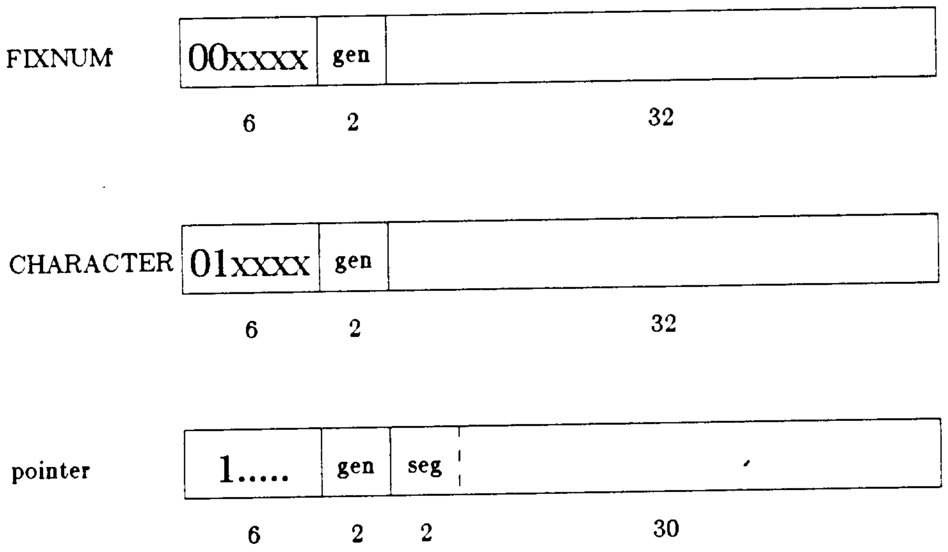
Bits and bytes are numbered in the same order. The most significant bit of an integer is bit 31 and the least significant bit is bit 0.



3.2. Tagged immediates and pointers

SPUR represents Lisp pointers, immediate integers and immediate characters using five bytes: one for a tag and four for the pointer or data. The tag byte consists of a 6-bit type tag and a 2-bit generation number. This number is the basis for a "generation scavenging" garbage collection algorithm [Ungar84].

Operations on the tag and data are logically independent, that is, no information moves between the two parts by carry propagation or any other implicit mechanism. The generation number for immediate FIXNUMs and immediate CHARACTERs is normally 0 (oldest) so that generation exception mechanism is simple to implement. See section 6.1.

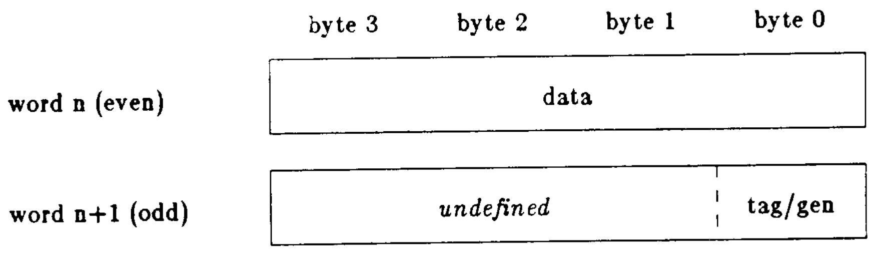


xxxx -- don't cares
gen -- generation tag
seg -- short segment number

3.3. Memory storage for tagged words

SPUR stores the five-byte tagged words in eight bytes of memory (aligned on an eight-byte boundary). This means that only the integer registers and the hardware directly connected to the CPU data bus have to deal with five-byte quantities. The rest of the system is organized around four-byte or eight-byte words.

The tag and data are stored in memory in the following way:



The advantage of this ordering is that all of the address bits are valid for the data part of the tagged word. Memory does not have to distinguish LD_32 from LD_40 since the processor ignores the tag byte that comes in from memory on a LD_32 instruction.

The upper three bytes in word $n+1$ are undefined after a ST_40 instruction.

3.4. Tag values

The SPUR Common Lisp compiler and runtime system use the following 19 tag values:

hex	
00-0f	immediate FIXNUM
10-1f	immediate CHARACTER
	MISC_TYPE
	BIT_VECTOR
	INTEGER_VECTOR
	GENERAL_VECTOR
	STRING
	FUNCTION
	ARRAY
	NIL
	CONS
	SYMBOL
	GC_FORWARD
	BIGNUM
	RATIO
	COMPLEX
	SHORT_FLOAT
	LONG_FLOAT
	EXTENDED_FLOAT

The hardware knows about the tags for FIXNUM, CHARACTER, CONS, NIL and the six types of numbers given last in the table. The meanings of the other tags are by software convention only.

NIL and CONS should have a Hamming distance of 1. The six types of numbers should be in a contiguous group of tag values (for instance 38-3f) for the benefit of the *eql* compare condition defined in section 6.3.

3.5. Floating point

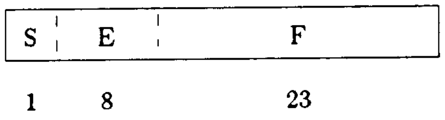
SPUR supports the IEEE 754 standard for binary floating point arithmetic [IEEE83]. The details of SPUR's implementation are described in [Lee85].

Floating point numbers can be represented in memory in three different formats: single, double and extended. Single and double precision floating point numbers are converted to extended precision when they are loaded into registers. All arithmetic operations produce an extended precision result in a register.

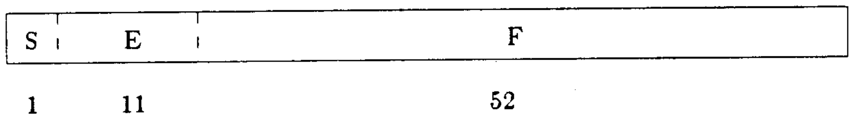
The memory formats are as follows:

S	sign
E	exponent
F	fraction
T	type tag
R	rounding bits

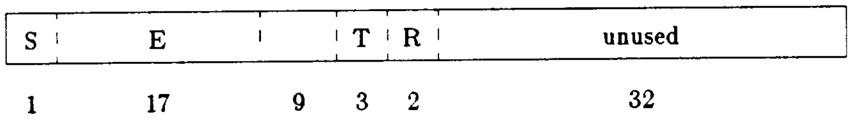
single



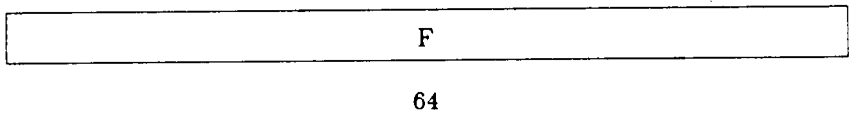
double



extended1



extended2



Six types of floating point numbers are encoded in each precision:

zero
denormalized number
normalized number
infinity
quiet Not-a-Number
signaling Not-a-Number

The IEEE standard requires that exponents be represented in excess-127 notation for single precision and excess-1023 notation for double precision. In both of these formats, the normalized fraction's leading bit is not stored (hence the name "hidden bit"). The hidden bit plus the fraction form a positive number whose value is equal to or greater than 1.0 and less than 2.0.

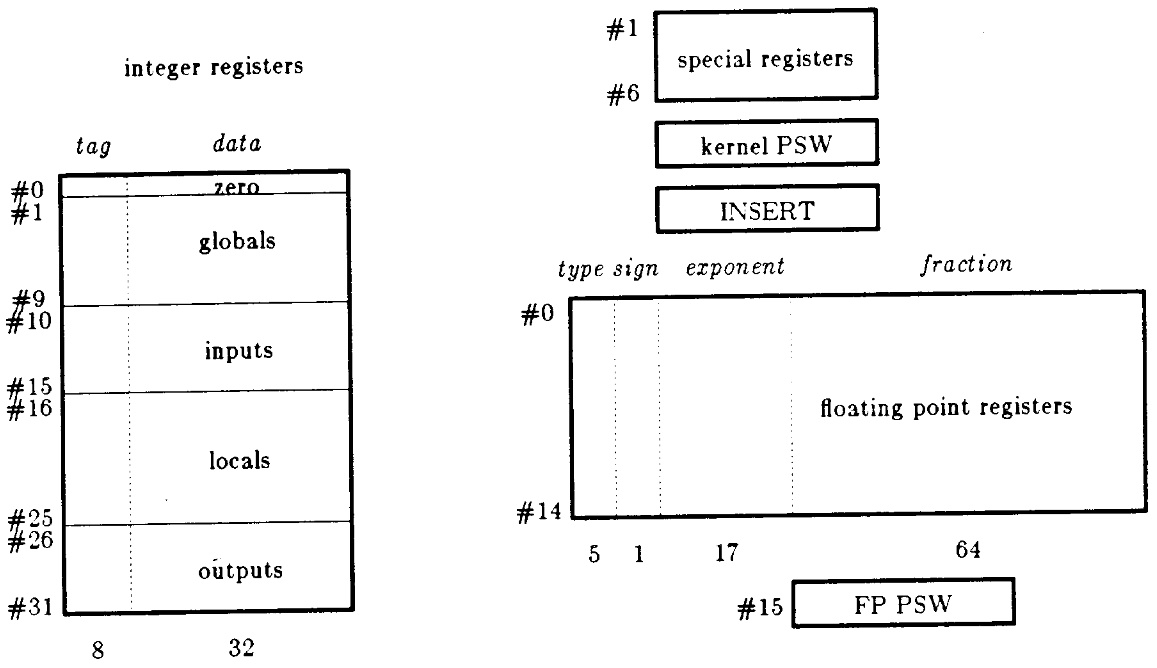
The standard does not prescribe the way that extended precision numbers must be represented. SPUR's extended precision exponent and fraction are stored in an expanded form, similar to how they appear in the middle of the floating point hardware data paths. This allows a software trap handler to modify a number when certain exceptions occur, achieving the same effect that hardwired control or microcode would in a system that encoded extended precision in the minimum number of bits. The expanded fraction has an explicit leading bit. The expanded exponent has 17 bits rather than the minimum 15. The exponent is represented in *2's complement - 1* notation rather than excess notation because all three precisions need to be converted to a common internal format. The excess notation does not work as well for this purpose because it implies a different bias for each precision. 2's complement - 1 is the same as two's complement, except that the numbers are biased down by one (-1 stands for 0, 0 stands for 1, and so on.)

There is additional information in the five bits that make up the data type field and the rounding tag field. The three-bit data type field encodes which of the six categories listed above the number belongs to. This field is assigned a value after every load from memory and after every register-to-register operation. One of the advantages of having it is that LD_SGL and LD_DBL instructions do not need to convert the exponent of a number in certain categories to a proper extended precision exponent. The fact that the conversion was not completed in these cases can be determined from the data type field.

Two bits record enough information so that the last rounding of the number could be undone. The floating-point standard requires that a system such as SPUR which delivers all of its results in extended precision must be able to mimic the operation of a system that has only single precision or only double precision arithmetic. The rounding tag provides enough information so that a software routine can unround an extended precision result and re-round it as if the result had been produced directly in single precision or double precision.

4. Registers

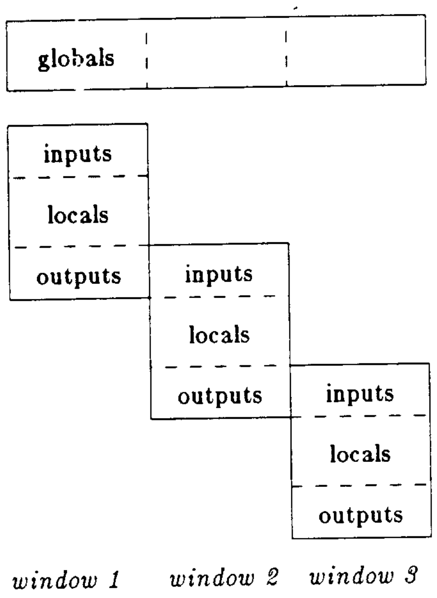
The registers are divided into three groups: integer, special and floating point.



4.1. Integer registers

The integer registers are organized in overlapping windows. Arguments placed in the output registers of one window appear in the input registers of the next window, after the program executes a CALL instruction. In a similar way, return values placed in the input registers of one window appear in output registers of the previous window, after the program executes a RETURN instruction. The local registers in each window are unique, while the global registers remain the same across all windows. R0 contains zero when read and does nothing when written.

The register windows are organized in the following way:



4.2. Special registers

The special registers consist of the six registers listed in the table below, the kernel process status word (KPSW) and the INSERT register. The KPSW is separate from the other registers because it can be written only when a process is in kernel mode. The INSERT register is separate because it is written from a different point in the data path than the other registers.

The special registers cannot be used directly in operations, but they can be copied to and from the integer registers. The UPSW and KPSW contain mode bits and trap enable bits that control whether traps occur in response to exceptions, faults and interrupts detected by the hardware. The CWP and SWP are used to determine when a window overflow or underflow has occurred. A program can read the PC in order to calculate PC-relative addresses. The FPU_PC locates the offending instruction when a floating-point exception occurs while the CPU and the FPU are executing instructions in parallel. Since the CPU has gone past the floating-point instruction that caused the exception, there is no other way to determine where that instruction was. The WRITE_PC is read by the first instruction of a trap handler in order to get the location of the instruction next after the one that caused the trap. The INSERT register provides the third operand for the INSERT instruction, since only two operands can be read in a single cycle. More details are given in the description of individual instructions in section 6.

number	name	width	function	description
1	UPSW	<05:00>	RW	user process status word
2	CWP	<09:07>	RW	current window pointer
3	SWP	<31:03>	RW	saved window pointer
4	PC	<31:02>	R	program counter
5	FPU_PC	<31:02>	RW	last FPU instruction initiated
6	WRITE_PC	<31:02>	R	PC of instruction that is writing its result this cycle
	INSERT	<01:00>	RW	byte shift count for insert instruction
	KPSW	<23:00>	RW	kernel process status word

<we will change the numbering to 0-5>

4.2.1. UPSW

bits	description	
05-03	trap enables:	tag exceptions integer overflow exception co-processor exception
02	illegal opcode exception on co-processor instructions	
01	parallel execution with co-processor	
00	co-processor number	

We may keep the floating-point rounding mode in the UPSW as well as the FP PSW.

We once planned to copy the opcode of a trapping instruction into the UPSW, but we decided instead that trap handlers should recover the PC of the trapping instruction, use that to recover the instruction itself, and finally extract the opcode from the instruction.

4.2.2. KPSW

There are separate virtual address enable bits for I-fetch and D-fetch so that bootstrap code in PROM has the option to access data with either physical or virtual addresses. The trap enable bits are discussed in section 6.

bits	description	
23-16	processor ID	
12-08	trap enables:	all traps interrupt error fault normal fault window overflow/underflow exception
05	previous mode (kernel = 1)	
04	current mode (kernel = 1)	
03	virtual address I-fetch	
02	virtual address D-fetch	
01	instruction buffer enable	
00	instruction prefetch enable	

Example -- changing from physical to virtual addressing mode:

```
                                                                running in kernel mode

                 rd_kpsw      temp1
                 and          temp2, temp1, ib_enable_mask      save ib_enable
                 and          temp1, temp1, ~(ib_enable_mask)   disable ib
                 or           temp1, temp1, virtual_addr_mask
                 jump         virtual_target                    delayed jump
                 wr_kpsw      temp1                             change address mode
virtual_target:  invalidate_ib
                 or           temp1, temp1, temp2
                 wr_kpsw      temp1                             restore ib_enable
```

4.3. Floating point

The fifteen floating-point registers hold numbers in the 87-bit extended precision format described in section 3.5.

The fields of the FP PSW are:

bits	description	
14-12	type tag for operand 1	
11-09	type tag for operand 2	
08-04	exception flags:	operand trap result overflow result underflow result inexact
04	write destination register when exception occur	
03	enable inexact exception	
02	enable other exceptions	
01-00	rounding mode	

5. Instruction formats

SPUR has seven instruction formats. The opcode and the register specifiers are in the same positions in all formats. The three-register format (RRR) is used for loads, register to register operations, special register operations and FPU operations. The register-register-immediate format (RRI) is used for loads and register to register operations.

5.1. Effective addresses

SPUR does not support PC-relative addresses for data, although the PC can be copied into a general-purpose register to accomplish the same effect. Load instructions have two addressing modes: register + register and register + signed 14-bit displacement. Store instructions have only the register + displacement mode so that a two-port register file is sufficient to implement them.

The effective address is a byte address in all cases. Its two or three low order bits are ignored by memory, but a carry can propagate from these bits into the higher part of the address during the addition. Address computations never cause arithmetic overflow.

Lisp implementation note: Unlike SOAR, there is no tag check to require that an effective address is a the sum of a pointer and a FIXNUM. SPUR Lisp checks the type of a pointer explicitly before using it to reference memory. Distinguishing a pointer from an immediate is not a sufficient check.

5.2. Long constants

Constants longer than 14 bits are loaded from a constant table rather than by a sequence of load-immediate instructions. SPUR does not have an instruction to load a constant into the high order part of a register, as RISC-II did.

Lisp implementation note: The first word of a Lisp function's code object is a pointer to its constants table. When a function is called, it begins executing at the third word of its compiled code object. By reading the PC into a register, the function can load the pointer to the constants table into a local register.

RRR

opcode	dest	src1	0	src2	unused
7	5	5	1	5	

RRI

opcode	dest	src1	1	immediate
7	5	5	1	14

store

opcode	imm high	src1	1	src2	imm low
7	5	5	1	5	9

compare RR

opcode	cond	src1	0	src2	word offset
7	5	5	1	5	9

compare RI

opcode	cond	src1	1	short imm	word offset
7	5	5	1	5	9

compare tag

opcode	cond	src1	tag imm	word offset
7	5	5	6	9

jump, call

op	word address within current segment
4	28

6. Instructions

The SPUR instruction set is listed in Tables ? through ? in Appendix A.

6.1. Loads and stores

The two primary load instructions are LD_40 and LD_32. CXR is the same as LD_40 except that it also performs a tag check. TEST_AND_SET is the same as LD_32 except that it also performs an atomic read-modify-write in the cache.

CXR stands for "car" or "cdr", the Lisp names for the two parts of an element of a list. The car of an element is the data represented by that element. The cdr points to the next element.

FPU loads and stores are similar to normal loads and stores. On an FPU load, the CPU calculates the effective address and the cache responds with the data. The only difference is that the FPU receives the data instead of the CPU. The timing is the same as if the CPU had taken the data. There is no provision for extra delay.

The TO_FPU and FROM_FPU instructions provide direct 32-bit transfers between the integer and floating point registers. The primary purpose is to speed the transfer of operands for an integer multiply or divide operation. The alternative would be store the data into memory and then load it into the FPU. From the CPU's point of view, these instructions are identical to LD_32 and ST_32 in every respect.

6.1.1. Read for Ownership and Read Anyway

One feature of SPUR's cache consistency protocol is that a process can read and write its private data with the same number of bus transactions that it would use in a system without cache consistency. To do this, the process must gain ownership of the cache line containing the private data the first time that the data is read. On the other hand, a process should avoid gaining ownership of shared data unless it intends to do a write. To distinguish these types of access, SPUR has duplicate sets of load instructions. LD_40, CXR, LD_32 and the floating-point load instructions come in two versions -- one called *read* and the other called *read for ownership.*

The LD_32 instruction has yet a third option. LD_32_*read anyway* reads a word from memory even if the page table entry says that the page is not available in memory. This instruction is used by the code that flushes a page from main memory to disk.

6.1.2. Generation exception

The generation tag exception should occur if a pointer to an object in a newer space is stored into a location in an older space. In terms of the ST_40 instruction, this means that Rs2 is a pointer and its generation number is newer than Rs1's. But a simpler tag check is just to compare the generation numbers without checking Rs2's type. The two definitions are equivalent if we assign the oldest generation number to immediate FIXNUMs and CHARACTERs. This also means that storing a pointer into an immediate will cause an exception. Rs1's tag can be checked explicitly if it matters.

Without the generation tag check, SPUR would need a five instruction sequence in front of every store that might cause a generation exception.

```
rd_tag
and (mask all but generation bits)
rd_tag
and (mask all but generation bits)
cmp_trap
```

6.2. Register operations

6.2.1. Arithmetic and logical

The arithmetic instructions are ADD, SUB and ADD with no trap (ADD_NT). ADD_NT is used for address calculations and to copy the tag and data from one register to another without causing a tag exception. Implicit address calculations within a load or store instruction do not cause overflow. ADD_NT allows us to perform explicit address calculations without overflow, also.

The logical operations are bitwise AND, OR and XOR. Integer multiplication, division and remainder are carried out by converting the operands to floating point, performing the operation and converting the result back to integer form. See section 12.

6.2.2. Shift

Allows static or dynamic shift control. Shift 0,1,2 or 3 bits left and 0 or 1 bits right.

Alternative that was rejected: SLL, SLL 2, SLL 3, SRL and SRA, all with compile-time shift counts.

6.2.3. Bytes and characters

Character and string operations are performed with byte INSERT and EXTRACT instructions. Insert is a two-instruction sequence so that the insertion point can be specified at compile time or runtime. We decided not to optimize for compile time shift distances.

We considered LD_BYTE and ST_BYTE instructions to transfer one byte directly between memory and a register. LD_BYTE was defined to set the tag to CHAR because Lisp defines each byte of a STRING to be a CHAR. C and Pascal disable tag traps, so the tag's value would not matter. EXTRACT sets the tag value to FIXNUM. Therefore LD_BYTE is equivalent to the sequence LD_32, EXTRACT, WR_TAG.

6.2.4. Tags

RD_TAG is similar to EXTRACT. WR_TAG is similar to INSERT.

6.3. Compare and branch

Compare and branch are combined into one instruction. SPUR does not have condition codes.

Two types of comparisons: data and tag. Data comparisons are between two registers or between a register and a five-bit immediate constant. Tag comparisons are between either two registers or a register and a constant.

There are two types of delayed branches. The CMP_BR_DELAYED instructions always execute the following instruction, whether or not the branch is taken. The CMP_BR_TRUE instructions always fetch the following instruction, but they only execute it if the branch is taken. If the branch is not taken, the following instruction behaves as a NOP.

We have the CMP_TRAP instructions because of the short PC-relative offset field in CMP_BR instructions. CMP_TRAP is an easier way to handle error conditions. The offset field is left open for a number to pass to the trap handler.

If the test condition is true (and trapping on the CMP_TRAP exception is enabled), then these instruction cause a trap in exactly the same way that a subtract instruction would cause an integer overflow trap.

6.4. Special registers

WR_KPSW is separate from WR_SPECIAL instruction so that the opcode alone tells whether to check for kernel mode.

WR_SPECIAL changes its target register so that the change is visible to the following instruction.

6.5. Call, return and jump

CALL and JUMP change the PC to an absolute word address in the current segment. The 28 bits of word offset limit the reach of a CALL or JUMP to one quarter of the address space. CALL_KERNEL and JUMP_REG are used to transfer control across segments.

To call a function whose address is in a register, SPUR uses an instruction sequence rather than a single "call_register" instruction. The sequence costs three extra cycles and adds three more instructions to the code. It also requires the allocation of a temporary global register or output argument register.

```
                call_reg    Ryy          ; target address in Ryy
                nop
continue:                                ; return here

target:         ...
                return      r10 + 8
                nop

                move        Gxx ← Ryy    ; target address in Ryy
                call        go
                nop
continue:                                ; return here

go:             jump_reg    Gxx
                nop

target:         ...
                return      r10 + 8
                nop
```

Here the assumption is that the CALL instruction saves its own address in R26.

We expect to use CALL_ABS much more frequently than CALL_REG in both C and Lisp.

JUMP_REG and RETURN add Rs1 to RC to form a byte address. The addition is included because there is no advantage in terms of either time or layout from taking the target address before the ALU instead of after it.

6.6. Floating point

7. Traps

We divide the unusual conditions that arise at runtime into four groups. Unusual conditions detected inside the CPU are called exceptions. Examples are the integer overflow, tag check and window overflow exceptions. Unusual conditions detected by the FPU are also called exceptions, i.e, the floating point exceptions.

Exception	Definition
A	Rs2 generation > (younger than) Rs1 generation
B	floating point
C	Rs1.tag != CONS or NIL
D	integer overflow
E	! (both tags == FIXNUM)
F	kernel mode
G	! (both tags == FIXNUM or both tags == CHAR)
H	true cmp_trap condition
I	window overflow
J	tags equal and in range hex 38-3F, and data not equal
K	window underflow

Notes:	
generation	two-bit field: 3 is youngest, 0 is oldest

All of the other unusual conditions occurring outside the CPU are called faults and interrupts. Faults occur in response to the execution of an instruction, while interrupts are asynchronous events that come from outside the processor that receives them. An example is an I/O interrupt. Faults must be handled immediately, while interrupts can be postponed until a convenient time.

The faults are divided into two groups. "Normal faults" include access violations, instruction page faults and data page faults. "Error faults" mean that there is a hardware problem. To respond to error faults, the CPU will change to the physical addressing mode.

The CPU responds to exceptions, faults and interrupts by taking a vectored trap. The trap vector consists of a trap base address concatenated with the trap type field. In order to service events external to the CPU, the trap handling software reads special external registers (with LD_CACHE or FROM_CPU instructions) to get more information.

There is a priority ordering for how to respond when more than one unusual condition occurs at the same time. For instance, while the CPU is waiting for the cache to respond to a LOAD instruction, it (the CPU) will not immediately respond to the interrupt line. Instead, it waits until after the data valid line or the normal fault line becomes active, so that the cache operation will not be left hanging.

The effect of an exception that causes a trap is to make the offending instruction behave as if it were a CALL instruction (with one or two NOP cycles inserted). The write to the offending instruction's destination register is blocked.

All traps are taken during an instruction's third cycle. This means that only one instruction can cause a trap (or be the victim of a trap) in any given cycle.

List of all exceptions, faults and interrupts.

1	error fault
2	normal fault
3	interrupt
4	floating point exception
5	window overflow exception
6	window underflow exception
7	illegal opcode exception
8	kernel mode exception
9	integer overflow exception
10	"CONS or NIL" tag exception
11	"FIXNUM" tag exception
12	"FIXNUM or CHAR" tag exception
13	"eql" tag exception
14	generation exception
15	CMP_TRAP exception

Priority of traps:

	trap type (priority)	description
highest	0	error fault (bus fault, debugging): change to physical addressing mode
	1	window overflow/underflow (CALL, CALL_KERNEL, RET, RET_KERNEL)
	2	normal faults (instruction page fault, data page fault, access violation)
	3	interrupt (I/O, timer, uart)
	4	floating point exception
	5	illegal opcode, kernel mode exception (WR_KPSW, DISABLE), "cons or nil" tag exception (CXR, TAG_CMP, TAG_CMP_TRAP)
	6	integer overflow (ADD, SUB)
	7	tag exceptions that recover operands in the same way "fixnum" ADD, SUB, logical, shift "fixnum or char" CMP_BR, CMP_TRAP "eql" CMP_BR, CMP_TRAP
	8	generation exception (ST_40):recovers its operands differently
lowest	9	CMP_TRAP, TAG_CMP_TRAP or CP_CMP_TRAP with TRUE test condition

7.1. Trap enable mechanism

There are eight trap enable bits, five in the KPSW and three in the UPSW. (Others will be defined in the FP PSW.)

7.2. Interrupts

Each processor board contains a memory-mapped interrupt register that any other processor can write. When certain bits are set in this interrupt register, the interrupt line to the CPU goes active.

8. Timing

An instruction can begin every cycle, even though some instructions require more than one cycle to complete. Software is responsible for scheduling instructions so that delayed results are available before they are used.

8.1. Delayed execution

Five groups of instructions have delayed execution: loads, compares, calls, returns and jumps. A delayed load means that the memory data is not available in the target register until the second instruction following the load. The instruction immediately after a load cannot use the target register as either a source or a destination. Using it as a source provides unpredictable data, depending on whether a trap occurs. Using it as a destination does not make sense.

Compare_and_branch instructions also take effect after a one cycle delay. The next instruction is always fetched. However, depending on the type of CMP_BR (one of three types), the next instruction may be turned into a NOP by the hardware.

8.2. Changing the current window pointer

The instruction next after a CALL or RETURN writes its result into a register in the new window, but this next instruction can read its operands only from the global registers or the special registers. The reason for this restriction is so that the processor will not have to restore its operand forwarding registers to their previous state if the instruction after a CALL or RETURN causes an instruction page fault.

8.3. Parallel execution of co-processor instructions

9. Kernel mode

Program control goes to a known location in all cases when the mode changes from user to kernel. The mode is changed by a CALL_KERNEL instruction or when a trap occurs for the following reasons:

10. Prohibited instruction sequences

```
             load   rX, address
followed by  op     rX, ...

             load   rX, address
followed by  op     rY, rX, ...
```

```
    cmp_br_true, tag_cmp_br_true, fpu_cmp_br_true
    call or return

    test_and_set
    load or store
```

11. Integer multiply, divide and remainder

We use the floating point multiply and divide operations to perform integer multiply, divide and remainder. The instruction sequence for integer multiply is:

			cycles
replace	mul	rcc, raa, rbb	
with	to_fpu	raa, f1	1
	to_fpu	rbb, f2	1
	float	f1, f1	3
	float	f2, f2	3
	fmul	f3, f1, f2	6
	fix	f4, f3	3
	from_fpu	rcc, f4	1
	nop		1
		total	19

If the integer operands are in memory rather than the integer registers, replace the "to-fpu" instructions with "ld_int".

The code sequence for integer division is similar. To meet the definition of division in most high-level languages, additional code is inserted to save the floating point rounding mode, change it to "round-toward-zero", and restore it. One possible code sequence would be:

```
from_fpu            rt1, fpu_psw
nop
and                 rt2, rt1, rounding-mode-mask
or                  rt2, rt2, round-toward-zero-mask
to_fpu              rt2, fpu_psw
...
code-for-division
...
to_fpu              rt1, fpu_psw
```

If a floating-point divide takes 18 cycles, then an integer divide will take about 40 cycles.

12. Questions

(1) Are 34 tag values enough? We could have more if FIXNUM and CHARACTER were defined by more than two bits. A: 34 is enough.

(2) Does RETURN_KERNEL need to copy the previous mode bit into the current mode bit?

13. Decisions to make

(1) Should CALL and JUMP contain a PC-relative offset instead of an absolute address? Absolute addresses were introduced to make one-cycle jumps and calls possible, but now all jumps and calls take two cycles. The advantage of a PC-relative offset is that it makes code position- independent (except for calls to separately compiled modules).
Answer: Keep absolute addresses because they simplify the hardware and we are willing to patch the code at load time.

(2) Should floating point exceptions cause traps as soon as possible or should they wait until the next floating point instruction? This matters only when the CPU and FPU are executing instructions in parallel.

(3) Should the type tag be in the low order six bits of the tag byte?

(4) Which floating point convert instructions should we have: a pair for double rounding, a pair for single rounding, or all four? Answer: Only the pair for double rounding.

(5) Do we need the ENABLE and DISABLE instructions? Alternative:

```
        rd_kpsw          rd
        or          rd, rd, mask
        wr_kpsw          0, rd,    0
```

Answer: No.

(6) Does changing the CWP or SWP with WR_SPECIAL cause window overflow or window overflow exceptions?
Answer: Whether it does or does not is undefined.

(7) Should a WR_SPECIAL to the PC or the WRITE_PC be a nop or should it remain undefined?

14. Instructions considered and rejected

multiply, divide, remainder
  Convert to floating point.

find first one
  Don't need it because SPUR has hardware floating-point.

add with carry, subtract with carry
  Use 31-bit rather than 32-bit segments for extended precision arithmetic.

reverse subtract
  Subtracting a register from a constant is an infrequent operation.

load multiple, store multiple
  Not significantly faster than sequence of loads and stores.

load byte, store byte
  Load byte requires an extra multiplexer on the input data path.

Store byte requires byte enable control lines in the cache.

load immediate into high-order bits of a register

Long constants are infrequent.

Load a word from a constant table in memory instead.

call register

Much less frequent in compiled Lisp than interpreted Lisp.

Can substitute a code sequence that includes jump_register.

rotate

We don't know of any high-level language constructs that make use of it.

shift 0-31

Low frequency of use in Berkeley C programs measured.

extract bit field

We decided not to provide special support for bit fields.

and_not, or_not, xor_not

Originally thought to help graphics programs, but idea not followed up.

operand shadow registers

We thought these might complicate the data path, but now we think not.
The only problem is to control them properly during a trapping sequence.

store substitute result into destination of trapped instruction

Would make the register file decoders more complicated and slower.

cdr coding

We prefer to use more memory in order to have only the standard type
of list representation. Lists are only part of a Lisp program space;
code and arrays probably account for larger portions.

invisible pointers

We don't want to trap on data just after it is read from the cache
or attach another controller to the cache.

compare and branch false

Save total number of instructions to implement and debug.

Can sometimes use compare-and-branch-true instead.

15. How to run C, Fortran and Pascal programs on SPUR

Disable tag exceptions. Use load_32 and store_32 instead of load_40 and store_40.

16. References

[Lee85]
Corinna Lee, "Internal Description of the SPUR Floating-Point Unit"

[IEEE83]
"IEEE Standard 754 for Binary Floating-Point Arithmetic"

[Ungar84]

16. Changes from Version 8 to Version 9

(1) Substitute the term "floating-point unit" (abbreviated FPU) for "co-processor" (CP) throughout the document. We decided that CP and CPU were too often confused.

(2) Change CALL, CALL_KERNEL and JUMP from one-cycle execution to delayed execution.

This means that all PC changes have delayed execution. One advantage is that the hardware and software trap handling mechanisms become simpler. Normally the current PC and next PC are saved when a trap occurs. When the trap handler has finished its work, it returns to original program with a *return to current PC, jump_reg to next PC* sequence. But if a page fault occurs on the instruction fetch for a one-cycle CALL, then there is no next PC in the pipeline because the next PC comes directly out of the CALL instruction itself. We haven't fetched that instruction yet because of the page fault. So the trap handler has to examine the instruction that would have been fetched and return in different ways depending on the opcode. A second advantage is that one-cycle execution was difficult to implement.

We expect the cost of this change due to call instructions to be small because their frequency is low. The frequency of unconditional jumps is higher, so the cost will depend on how often we can fill the delayed slot.

(3) Add a register-immediate form for the CMP_BR_DELAYED, CMP_BR_TRUE and CMP_TRAP instructions. No new opcodes are required because one value of the register vs. immediate operand selector (bit 14) was unused.

This new form compares data in a register with a 5-bit unsigned constant in the s2 field. The dynamic Lisp measurements that we have so far show that CMP_BR R,I accounts for 5% of all instructions.

(4) Remove the CMP_BR_FALSE, TAG_CMP_BR_FALSE and FPU_CMP_BR_FALSE instructions.

We considered eliminating both CMP_BR_TRUE and CMP_BR_FALSE, but after going through Dain Samples' code scheduler, the three types of conditional branches appear with the following (static) frequencies:

CMP_BR_DELAYED	5 %
CMP_BR_TRUE	80 %
CMP_BR_FALSE	15 %

We expect CMP_BR_TRUE to be even more frequent in the dynamic statistics because it is the usual branch instruction at the bottom of a loop.

Most delayed conditional branches will be followed by NOPs unless we keep at least one of the "cancelling" varieties. This result contrasts sharply with the one reported by the MIPS people, who said that they put useful instructions after a delayed branch 90% of the time in Pascal programs. It also contrasts with the RISC-II C compiler that put a useful instruction after about 50% of the delayed branches. What we're finding is that filling the slots after delayed loads (which RISC-II did not have) consumes almost all of the instructions that we might have put after the compares. (Why wasn't this also true for MIPS?)

Removing CMP_BR_FALSE alone does not save the hardware from implementing the conditional NOP capability, but it reduces the number of opcodes. The lack of CMP_BR_FALSE may cost us an extra cycle at runtime and it may increase the code size by one instruction for each instance where it would have been used. If the compiler is smart enough, the code size and number of cycles could remain the same. CMP_BR_FALSE is the easiest instruction for the compiler to generate because it does not involve a delayed instruction slot.

We think that CMP_TRAP will be used more often than CMP_BR_FALSE to check for error cases.

Example:

```
old:        CMP_BR_FALSE    cond        unlikely
            next1
            next2
            ...
merge:      ...

unlikely:   unlikely1
            ...
            jump            merge
--------------------------------------------------
new:        CMP_BR_TRUE     not(cond)   Lnext2
            next1
            unlikely1
            ...
            jump            merge

Lnext2:     next2
            ...
merge:      ...
```

Another example (if-then-elseif-elseif type of case statement):

```
old:  CMP_BR_FALSE   cond   target1
      CMP_BR_FALSE   cond   target2
      CMP_BR_FALSE   cond   target3
      next1
```

new:

(5) Remove the FPU_CMP_TRAP instruction.

FPU compares are used less often than integer compares to check for error conditions.

(6) WR_SPECIAL will change the special register in cycle 2. The next instruction after a WR_SPECIAL CWP cannot read its operands from the input, output or local registers. Otherwise, there are no restrictions.

Our first idea was that WR_SPECIAL would change the register in cycle 4, but there is no advantage in waiting past cycle 3. The problem with changing the special register in cycle 2 is that the change must be blocked if the previous instruction takes a trap in its cycle 3. In addition, the instruction before the WR_SPECIAL must be able to complete its register write without interference from the WR_SPECIAL.

Kong suggested a way to back up the change in case of a trap. We can also isolate the instruction before a WR_SPECIAL CWP by passing the window number along with the destination register number through the pipeline.

We could write the special registers in cycle 3 and make WR_SPECIAL into either a two-cycle instruction or a delayed instruction. *<Explain why this would not make the hardware any simpler than it is for writing in cycle 2.>*

Code sequence for store-byte:

```
ST_BYTE    Rx, addr    <=>    LD_32         Rtemp, addr
                              WR_SPECIAL    INS, addr
                              INSERT        Rtemp, Rtemp, Rx
                              ST_32         Rtemp, addr
```

This sequence is one cycle shorter than it would be if WR_SPECIAL changed the register later than cycle 2.

(7) Calls and returns change the CWP at the same time that WR_SPECIAL CWP does. Consequently, the next instruction following cannot read the input, output or local registers.

(8) The RETURN_TRAP, JUMP_REG sequence at the end of a trap handler must read the second return address from a global register. Therefore, one global register must be left open by software convention for use by trap handlers.

(9) WR_KPSW writes the KPSW in cycle 2 and delays the next instruction by one cycle. WR_KPSW also resets the instruction buffer since we might be changing between the virtual and physical address spaces.

The instruction fetch for the instruction after a WR_KPSW has to be repeated after the write has taken place.

If the I-buffer needs to be invalidated, this is done by software before the WR_KPSW instruction.

(10) CALL_KERNEL is similar to WR_KPSW. It changes to kernel mode in cycle 2 and delays the next instruction fetch by one cycle.

We could ignore this problem if instruction fetches did not check the page protection bits, but we do not plan to distinguish instruction fetches from data fetches, and the protection bits are checked on data fetches.

(11) The UPSW, CWP, SWP and FPU_PC special registers are writeable, but the PC and the WRITE_PC are read-only. The effect of writing to either of these registers is undefined.

(12) Remove the CVTS_UNRND and CVTD_UNRND floating point instructions. These will be handled by instruction sequences instead.

The "unrounding" converts are not expensive to implement in hardware, but we prefer to have only one pair of convert instructions -- the ones that perform a "double-rounding". The impact of doing the unrounding converts in software is that SPUR will have to evaluate all right-hand-side expressions in extended precision. This goes against the traditional way to evalute expressions in Fortran. Mimicking the behavior of machines that do not have extended precision will be slow, but we do not think that this is important. (Professor Kahan thinks we will fare badly on comparisons with the National and Weitek chips.)

(13) Add the RETURN_TRAP instruction.

We have to enable all traps (KPSW bit 12) at the same time that we return from a trap handler. Otherwise, the window overflow handler could get stuck in an infinite loop. This is partly a consequence of our decision to increment the window pointer in the process of taking a trap.

(14) The window overflow stack will grow up instead of down.

(15) Add the INVALIDATE_IB instruction.

This function is not easy to encode as part LD_CACHE or ST_CACHE, as we had previously planned to do.

(16) Traps vector to addresses 000010x0 rather than addresses 000020x0. There is no reason for traps to go to a different page than CALL_KERNEL does. It goes to address 00001000.

(17) The trap vector addresses are 16 bytes apart rather than 8 because we no longer have one-cycle JUMP instructions.

(18) Compare conditions "always" and "never" will not cause an exception even if the tags are not "fixnum" or "character". The primary use for these conditions is expected to be "CMP_TRAP always ..." This instruction should be easier to use if it can cause only one type of exception.

(19) Add the FMOV instruction. Adding floating point zero to a number is not equivalent because the sign of -0 + +0 depends on the rounding mode.

(20) Add the RD_INSERT and WR_INSERT instructions. The INSERT register will not be in the same location as the other special registers, so it is convenient to have separate opcodes to read and write it.

Decisions:

(1) There will be two fault pins on the CPU, so that the error faults which change the processor to physical mode are distinguished from the normal faults which do not.

(2) TO_FPU and FROM_FPU will be retained if the impact on the board design is small. The impact on the CPU and FPU chips has to do with the opcode PLA, but nothing else.

	CPU	FPU
TO_FPU	reads rs2 as if ST_32	writes rd as if LD_INT

FROM_FPU	writes rd as if LD_32	reads rs2 as if ST_INT

(3) We rejected the idea of delayed stores (with the restriction that a memory operation could not follow a store.) Stores will remain two-cycle instructions, implemented by causing a stall in the third cycle. This means the subsequent instruction will be in its execution cycle when the stall occurs.

17. Changes from Version 7 to Version 8

(1) Remove the LD_BYTE and ST_BYTE instructions.

LD_BYTE was deleted to simplify the path between the data pads and the register file inside the CPU chip.

ST_BYTE was deleted to simplify the cache implementation. In the current cache this means that there are fewer chip enable lines. At one time, there were larger hardware requirements having to do with parity. ST_BYTE would have required five extra pins on the CPU data bus and there was a conflict between the word parity used by the NuBus and the byte parity implied by ST_BYTE. But these reasons no longer matter because we eliminated parity checking.

Sample code sequences are:

```
old     ld_byte    Rxx, Ryy, RC           Rxx<07:00> <- M [Ryy + RC]
        nop

new     ld_32      Rxx, Ryy, RC
        nop
        extract    Rxx, Rxx, Ryy or RC    if only one of Ryy or RC
                                          has non-zero low order bits

new     ld_32      Rxx, Ryy, RC
        add_nt     Rzz, Ryy, RC
        extract    Rxx, Rxx, Rzz          if both Ryy and RC have
                                          non-zero low order bits


old     st_byte      Rxx, Ryy, RC      Rxx<07:00> -> M [Ryy + RC]

new     ld_32        Rzz, Ryy, RC
        wr_special   INS, Ryy, RC      INS <- (Ryy + RC)<01:00>
        insert       Rzz, Rzz, Rxx
        st_32        Rzz, Ryy, RC
```

(2) Remove the BYTE_INPUT and BYTE_OUTPUT instructions.

The device registers that are mapped into the physical address space can be read and written with LD_32 and ST_32, provided that the page table entries are marked "non-cacheable". We originally thought that some of the device registers would be at byte addresses that were not word-aligned.

(3) Remove the NIL instruction.

To substitute for this instruction, we will copy the NIL value from a global register to each of the local registers that a function uses, one at a time. There are two reasons to set registers to NIL. One is to initialize local variables after entering a function. We should be able to avoid NILing variables that are defined before they are used, The other reason is to clean up local pointer variables before returning from a function so that the next call will not begin with stale pointers in its register frame. Stale pointers could cause the garbage collector to save more state than necessary. We expect that this won't be a significant problem.

(4) Remove the shadow registers and the FIXUP instruction.

These features speed up the trap handler when it needs to recover a trapped instruction's operands or when it needs to store a substitute result into a trapped instruction's destination. Lisp programs that perform non-integer arithmetic (floating point, bignums, rational numbers, complex numbers) will take this type of tag trap.

One way to avoid the overhead of taking a trap and recovering the operands would be to use declarations in Lisp programs that perform a significant amount of floating point or other non-integer arithmetic.

(5a) Remove the CALL_REG instruction.

In some cases, it is necessary to call a function whose address is in a register. An alternate code sequence is to put the target address into a global register, then execute a CALL to a JUMP_REG instruction.

```
            add_nt      Gxx, Ryy, 0   target address in Ryy
            call_abs    go
comeback:   jump        next          return here
go:         jump_reg    Gxx
            nop
next:       ...

target:     target code
            return      r10 + 4
            nop
```

Here the assumption is that a CALL saves its own address in r26.

We expect to use CALL_ABS much more frequently than CALL_REG in both C and Lisp. The advantage of deleting CALL_REG is that we will then have only CALLs that take one cycle (CALL_ABS and CALL_KERNEL). This will make it easier for the hardware to save PC's during a trap in the same way that it saves the return PC on a CALL.

(5b) Remove Trap K -- FUNCTION trap on CALL_REG.

(6) Change WR_SPECIAL from Sd ← RC to Sd ← Rs1 + RC.
Change WR_KPSW from KPSW ← RC to KPSW ← Rs1 + RC.

Two reasons: (a) there is no advantage in avoiding the ALU operation, and (b) this makes the ST.BYTE instruction sequence simpler.

(7) STOREs will take two cycles (not visible to the programmer).

(8) CMP_TRAP, TAG_CMP_TRAP, CP_CMP_TRAP will not have a separate trap base register. The compare-trap exception will vector off of the same trap base address as all of the other exceptions.

(9) Change the EQL compare condition.

old definition:	TRUE if tags equal and data equal

FALSE if tags not equal
FALSE if tags both FIXNUM or CHAR and data not equal
TRAP if tags equal (but not FIXNUM or CHAR) and data not equal

new definition: TRUE if tags equal and data equal
TRAP if tags equal and any type of number, but data not equal
FALSE otherwise

(10) Remove the floating point REM instruction. We found an acceptable way to perform argument reduction in software.

(11) Remove the integer MUL, DIV and REM instructions.

We will convert integer operands to floating point, do a floating point multiply or divide, and then convert the answer back to integer form. Integer overflows will be detected on the final convert. The multiply product will be exact if it does not overflow. Division by zero will be detected during the floating point division operation. We need to think more about correct rounding of integer quotients. Our current scheme for changing the rounding mode during integer division and then restoring the rounding mode to its previous value may be slow. Integer multiplies will take about 20 cycles.

(12) The instruction (call it X) following a CALL or RETURN can write to any register in the new window, but it can only read from the global registers or the special registers. Our previous statement that X could read from registers in the old window was incorrect. The reason is that if the instruction fetch for X causes a page fault, then the CWP will change before X is re-executed after the fault is serviced. If no page fault occurs, then the CWP would have its old value at the time that X reads its operand registers.

(13) Add the SYNC instruction. This causes the CPU to stall until either the FPU data valid line or the FPU exception line goes high.

(14) Add the floating point NEGATE and ABSOLUTE VALUE instructions.

Decisions:

(1) We decided to retain the generation exception check on ST_40 instructions. The two-bit wide comparator for this check does not affect the critical path. If we removed this tag exception, every ST_40 that might put a pointer to new space into an old space location would need five extra instructions to compare the generation tags (whereas SOAR needed only one extra instruction to make the check in software rather than hardware). We think that the frequency of stores that need to be checked may be twice as high for Lisp as it was for SOAR. Thus the cost of taking this hardware out may be ten times greater for SPUR than it would have been for SOAR.

Storing into a large data array in old space will need to be handled by a higher-level mechanism that this tag trap. Otherwise, a long list of pointers will be added to the garbage collection root list and execution speed will be severely degraded.

LOAD								
Op-code	Instruction	Operands	Cache Op	Action		Delay	Excep-tion	Notes
				Rd.data ←	Rd.tag/gen ←			
	LD_40 [RO]	Rd,Rs1,RC	R [RO]	M [(Rs1 + RC) & ~07]	M [(Rs1 + RC) | 04]	yes	-	1
	CXR [RO]	Rd,Rs1,RC	R [RO]	M [(Rs1 + RC) & ~07]	M [(Rs1 + RC) | 04]	yes	C	1
	LD_32 [RO] [RA]	Rd,Rs1,RC	R [RO] [RA]	M [(Rs1 + RC) & ~03]	FIXNUM, OLDEST	yes	-	2
	TEST_AND_SET	Rd,Rs1,RC	TS	M [(Rs1 + RC) & ~03]	FIXNUM, OLDEST	yes	-	1,3
				M [(Rs1 + RC) & ~03]<00> ← 1				
	LD_CACHE	Rd,Rs1,RC	any	data lines	FIXNUM, OLDEST	yes	-	4
				cache op = (Rs1 + RC)<04:00>				
				address lines = (Rs1 + RC)<31:00>				
	FROM_FPU	Rd,0,Rs2	NA	FPU Rs2<63:32>	FIXNUM, OLDEST	yes	-	5
	LD_SGL [RO]	Rd,Rs1,RC	R [RO]	FPU Rd ← M [(Rs1 + RC) & ~03]		yes	-	
	LD_DBL [RO]	Rd,Rs1,RC	R [RO]	FPU Rd ← M [(Rs1 + RC) & ~07]		yes	-	
	LD_EXT1 [RO]	Rd,Rs1,RC	R [RO]	FPU Rd ← M [(Rs1 + RC) & ~07]		yes	-	6
	LD_EXT2 [RO]	Rd,Rs1,RC	R [RO]	FPU Rd ← M [(Rs1 + RC) & ~07]		yes	-	7
	LD_INT [RO]	Rd,Rs1,RC	R [RO]	FPU Rd<63:32> ← M [(Rs1 + RC) & ~03]		yes	-	

STORE							
Opcode	Instruction	Operands	Cache Op	Action	Delay	Excep-tion	Notes
	ST_40	Rs2,Rs1,SC	W	Rs2.data → M [(Rs1 + SC) & ~07]	-	A	1
				Rs2.tag/gen → M [(Rs1 + SC) | 04]			
	ST_32	Rs2,Rs1,SC	W	Rs2.data → M [(Rs1 + SC) & ~03]	-	-	2
	ST_CACHE	Rs2,Rs1,SC	any	cache op = (Rs1 + SC)<04:00>	-	-	4
				address lines = (Rs1 + RC)<31:00>			
				data lines = Rs2.data, Rs2.tag/gen			
	TO_FPU	Rs2,0,Rd	NA	Rs2.data → FPU Rd<63:32>	yes	-	8
	ST_SGL	Rs2,Rs1,SC	W	FPU Rs2 → M [(Rs1 + SC) & ~03]	-	-	-
	ST_DBL	Rs2,Rs1,SC	W	FPU Rs2 → M [(Rs1 + SC) & ~07]	-	-	-
	ST_EXT1	Rs2,Rs1,SC	W	FPU Rs2 → M [(Rs1 + SC) & ~07]	-	-	6
	ST_EXT2	Rs2,Rs1,SC	W	FPU Rs2 → M [(Rs1 + SC) & ~07]	-	-	7
	ST_INT	Rs2,Rs1,SC	W	FPU Rs2<63:32> → M [(Rs1 + SC) & ~03]	-	-	-

REGISTER

Opcode	Instruction	Operands	Action		Delay	Exception	Notes
			Rd.data ←	Rd.tag/gen ←			
	ADD_NT	Rd,Rs1,RC	Rs1.data + RC	Rs1.tag/gen	-	-	9
	ADD	Rd,Rs1,RC	Rs1.data + RC	Rs1.tag/gen	-	D,E	
	SUB	Rd,Rs1,RC	Rs1.data - RC	Rs1.tag/gen	-	D,E	
	AND	Rd,Rs1,RC	Rs1.data and RC	Rs1.tag/gen	-	E	
	OR	Rd,Rs1,RC	Rs1.data or RC	Rs1.tag/gen	-	E	
	XOR	Rd,Rs1,RC	Rs1.data xor RC	Rs1.tag/gen	-	E	
	SLL	Rd,Rs1,RC	Rs1.data << RC<01:00>	Rs1.tag/gen	-	E	
	SRA	Rd,Rs1,RC	Rs1.data >> RC<00>	Rs1.tag/gen	-	E	
	SRL	Rd,Rs1,RC	Rs1.data >> RC<00>	Rs1.tag/gen	-	E	
	RD_TAG	Rd,Rs1,0	<31:08> ← 0 <07:00> ← Rs1.tag/gen	FIXNUM, OLDEST	-	-	
	EXTRACT	Rd,Rs1,RC	<31:08> ← 0 <07:00> ← byte RC<01:00> of Rs1.data	FIXNUM, OLDEST	-	-	
	WR_TAG	Rd,Rd,RC	unchanged	RC.data<07:00>	-	-	
	INSERT	Rd,Rs1,RC	byte INS<01:00> of Rd.data ← RC<07:00> other bytes copied from Rs1 to Rd	Rs1.tag/gen	-	-	10

SPECIAL READ/WRITE

Opcode	Instruction	Operands	Action		Delay	Exception	Notes
			Rd.data ←	Rd.tag/gen ←			
	RD_SPECIAL	Rd,Ss1,0	Ss1	FIXNUM,OLDEST	-	-	
	RD_INSERT	Rd,0,0	INSERT	FIXNUM,OLDEST	-	-	
	RD_KPSW	Rd,0,0	KPSW	FIXNUM,OLDEST	-	-	
	WR_SPECIAL	Sd,Rs1,RC	Sd ← Rs1.data + RC		yes	-	11
	WR_INSERT	0,0,RC	INSERT ← RC<01:00>	-	-		
	WR_KPSW	0,Rs1,RC	KPSW ← Rs1.data + RC reset instruction buffer		-	F	12
	SYNC	0,0,0	stall until FPU is not busy		-	-	
	INVALIDATE_IB	0,0,0	invalidate all instruction buffer entries		-	-	

number	name	width	description
1	UPSW	<05:00>	user process status word
2	CWP	<09:07>	current window pointer
3	SWP	<31:03>	saved window pointer
4	PC	<31:02>	program counter
5	FPU_PC	<31:02>	last FPU instruction initiated
6	WRITE_PC	<31:02>	PC of instruction that is writing its result this cycle

COMPARE AND BRANCH						
Opcode	Instruction	Operands	Action	Delay	Exception	Notes
	CMP_BR_DELAYED	cond,Rs1,CC,offset	if (Rs1 cond CC) PC ← PC + sign-extended offset<10:02>	yes	G,J	
	TAG_CMP_BR_DELAYED	cond,Rs1,TC,offset	if (Rs1.tag cond TC) ...	yes	C	
	FP_CMP_BR_DELAYED	cond,Rs1,Rs2,offset	if (FP Rs1 cond FP Rs2) ...	yes	B	
	CMP_BR_TRUE	cond,Rs1,CC,offset	if (Rs1 cond CC) PC ← PC + sign-extended offset<10:02> else make next instruction into NOP	yes	G,J	
	TAG_CMP_BR_TRUE	cond,Rs1,TC,offset	if (Rs1.tag cond TC) ...	yes	C	
	FP_CMP_BR_TRUE	cond,Rs1,Rs2,offset	if (FP Rs1 cond FP Rs2) ...	yes	B	
	CMP_TRAP	cond,Rs1,CC,unused	if (Rs1 cond CC) cmp_trap exception occurs	-	G,J,H	13
	TAG_CMP_TRAP	cond,Rs1,TC,unused	if (Rs1.tag cond TC) ...	-	C,H	13

COMPARE CONDITIONS					
Instruction	Encoding	Name	Meaning	Exception	Notes
CMP_BR_DELAYED,	00	eq	data eq	G	
CMP_BR_TRUE,	01	neq	data ne	G	
CMP_TRAP	02	gt	data gt signed	G	
	03	le	data le signed	G	
	04	ge	data ge signed	G	
	05	lt	data lt signed	G	
	06	ugt	data gt unsigned	G	
	07	ule	data le unsigned	G	
	08	uge	data ge unsigned	G	
	09	ult	data lt unsigned	G	
	0A	always	always	-	
	0B	never	never	-	
	0C	eq_tag	tags eq	-	
	0D	ne_tag	tags ne	-	
	0E	eq_40	tags eq and data eq	-	
	0F	ne_40	tags ne or data ne	-	
	1E	eql	tags eq and data eq	J	14
	1F	neql	tags ne or data ne	J	14
TAG_CMP_BR_DELAYED,	0C	eq_tag	tags eq	-	
TAG_CMP_BR_TRUE,	0D	ne_tag	tags ne	-	
TAG_CMP_TRAP	1C	endp	tags eq	C	
	1D	nendp	tags ne	C	
FP_CMP_BR_DELAYED,	bit<24>		Invalid Exception if relation is unordered	B	
FP_CMP_BR_TRUE	bit<23>		unordered	B	
	bit<22>		less than	B	
	bit<21>		equal	B	
	bit<20>		greater than	B	

CALL and JUMP						
Opcode	Instruction	Operands	Action	Delay	Excep-tion	Notes
x0--x7	JUMP	28-bit addr	PC ← PC<31:30> @ addr<29:02>	yes	-	
	JUMP_REG	0,Rs1,RC	PC ← (Rs1 + RC) & ~03	yes	-	
y0--y7	CALL	28-bit addr	CWP ← CWP + 1 window	yes	I	15
			R10 (new window) ← PC			
			PC ← PC<31:30> @ addr<29:02>			
	CALL_KERNEL	0,0,0	CWP ← CWP + 1 window	yes	I	15,16
			R10 (new window) ← PC			
			PC ← hex 00001000 (2nd page)			
			KPSW<mode> ← 1 (kernel mode)			
	trap		CWP ← CWP + 1 window	-	I	17
			R10 (new window) ← PC			
			PC ← hex 00001000 @ TT<07:04>			18
			KPSW<all traps> ← 0 (disable traps)			
			disable certain traps depending on TT			
			change to kernel mode depending on TT			
			change to physical mode depending on TT			
			reset instruction buffer			
	RET	0,Rs1,RC	PC ← (Rs1 + RC) & ~03	yes	K	15
			CWP ← CWP - 1 window			
	RET_TRAP	0,Rs1,RC	PC ← (Rs1 + RC) & ~03	yes	K	15
			CWP ← CWP - 1 window			
			KPSW<all traps> ← 1 (enable traps)			
	RET_KERNEL	0,Rs1,RC	PC ← (Rs1 + RC) & ~03	yes	K	15,16
			CWP ← CWP - 1 window			
			KPSW<mode> ← 0 (user mode)			
			KPSW<all traps> ← 1 (enable traps)?			

FLOATING-POINT						
Opcode	Instruction	Operands	Action	Delay	Excep-tion	Notes
	FADD	Rd,Rs1,Rs2	FPU Rd ← FPU Rs1 + FPU Rs2	-	B	
	FSUB	Rd,Rs1,Rs2	FPU Rd ← FPU Rs1 - FPU Rs2	-	B	
	FMUL	Rd,Rs1,Rs2	FPU Rd ← FPU Rs1 * FPU Rs2	-	B	
	FDIV	Rd,Rs1,Rs2	FPU Rd ← FPU Rs1 / FPU Rs2	-	B	
	FMOV	Rd,Rs1,0	FPU Rd ← FPU Rs1	-	?	
	FABS	Rd,Rs1,0	FPU Rd ← FPU Rs1 with zero sign	-	?	
	FNEG	Rd,Rs1,0	FPU Rd ← FPU Rs1 with opposite sign	-	?	
	FLOAT	Rd,Rs1,0	FPU Rd ← convert to extended (FPU Rs1<63:32>)	-	-	19
	FIX	Rd,Rs1,0	FPU Rd<63:32> ← convert to integer (FPU Rs1)	-	B,D	
	CVTS	Rd,Rs1,0	FPU Rd ← convert to single (FPU Rs1)	-	B	
	CVTD	Rd,Rs1,0	FPU Rd ← convert to double (FPU Rs1)	-	B	

d	number in dest field of an instruction
s1	number in src1 field of an instruction
s2	number in src2 field of an instruction
Rd	contents of register d
Rs1	contents of register s1
Rs2	contents of register s2

CXR	car or cdr
RC	Rs2 or 14-bit immediate constant
SC	14-bit store constant
CC	Rs2 or 5-bit unsigned immediate constant
TC	6-bit tag constant
FPU	floating-point unit
TT	trap type
Cache Ops:	R -- read
	RO -- read for ownership
	RA -- read anyway (ignore fact that PTE says page is unavailable)
	TS -- test and set
	W -- write
	NA -- no access

Notes:

1 Address must point to a cacheable page (otherwise either an error or undefined.)

2 Address may point to either a cacheable or non-cacheable page.
If non-cacheable then a word is read from memory or a memory-mapped device register.

3 Cannot be followed by a load or store instruction.

4 Bits <04:00> appear on both the cache op and address lines.

5 Direct FPU to CPU transfer. Identical to LD_32 (including delayed arrival of the data) for the CPU.
Identical to ST_INT for the FPU.
The cache ignores the bogus address that the CPU sends out.

6 Sign, exponent and extra information fields of an extended precision floating point number.

7 64-bit fraction field of an extended precision floating point number (including hidden bit).

8 Direct CPU to FPU transfer. Identical to ST_32 for the CPU.
Identical to LD_INT (including delayed arrival of the data) for the FPU.
Rd can be specified because it forms part of the SC field.
The cache ignores the bogus address that the CPU sends out.

9 ADD with no traps

10 INSERT depends on the value of the INS special register. The INS register's value
is assigned with a WR_SPECIAL instruction.

11 No delay when writing the INS, SWP or UPSW registers.
One cycle delay when writing the CWP (meaning that the next instruction can read
only from the global registers or the special registers other than the CWP).

We have not decided whether the hardware will prohibit writing the PC, WRITE_PC and FPU_PC.
In the meantime, writing to the PC has undefined effects.
Writing to the WRITE_PC or the FPU_PC serves no useful purpose.
Whether writing to the CWP or the SWP can cause a window overflow or underflow is undefined.

12 The next instruction is delayed by one cycle so that the KPSW change will
have taken effect.

13 The cmp_trap exception (H) is ignored if one of the other exceptions (C, G or J) occurs at the same time.

14 "eql" is TRUE if the tags are equal and the data are equal,
causes an exception if the tags are equal and in range hex ??-?? (any type of number), but the data are not equal
is FALSE otherwise

15 The instruction that executes next after a CALL or a RETURN can read its operands only from
the global registers or the special registers.

16 The timing of the user/kernel mode change does not matter for CALL_KERNEL,
but it does matter for the instruction fetch after a RET_KERNEL.

17 When a trap is taken in response to a fault, an interrupt, a CPU exception or an FPU exception,
the processor follows the sequence described by this dummy "trap" instruction.

18 Whether to assign the trap base address to fixed location and whether to include the opcode
as part of the trap vector are under discussion.

19 FLOAT takes an integer in the high order half of the fraction part of an FPU register as its operand.
The conversion is undefined if the low order half of the fraction is non-zero.

Design Notes for the SPUR Processor

Chien Chen
Shing Kong
Daebum Lee
Trudy Stetzler

University of California, Berkeley
CS 292i
VLSI Testing and Implementation

1. Introduction

This report is a summary of the hardware design for the SPUR processor chip (referred to as CPU henceforth). An overview of the CPU is shown in Figure_1-1. The estimated area of the CPU (including control logic and routing) is 7000 lambda high by 7000 lambda wide. The goal of this report is to summarize the specifications for the CPU.

An interesting aspect of the CPU register array and instruction buffer is the uniform pipeline stages. Because the pipeline stages are uniform, the only control signal (other than clocks) required by the arrays is a "stall" signal, which is used in the latches to maintain the current state.

This report is divided into three chapters. Chapter 2 explains the selection and operation of the four stage pipeline used in the CPU. Then the clocking specification, which is a direct result of the pipeline used, is explained. Chapter 3 briefly explains the clock generation circuitry required for the CPU.

2. Pipeline and Clocking Specification

2.1. Pipeline Organization

The basic ideas behind the SPUR pipeline organization are very similar to the three-stage pipeline of RISC-II. [Kate83] In an ideal three-stage pipeline, each instruction takes three cycles to finish. However, the load instruction, whose frequency of occurrence is about 15% in RISC-II, takes four cycles to complete.

There are many ways to handle this 4-cycle instruction. The option shown in Figure_2-1 is used in RISC-II and SOAR [Kate83]. This option requires suspending the pipeline for one cycle after a load instruction. This suspension of the pipeline causes less than one instruction to be executed per cycle. The major reason for suspending the pipeline is due to the single port memory. This means that only one memory access can be in progress at any time. As a result, the memory access of instruction1 (the load instruction) cannot be overlapped with the instruction fetch of instruction3.

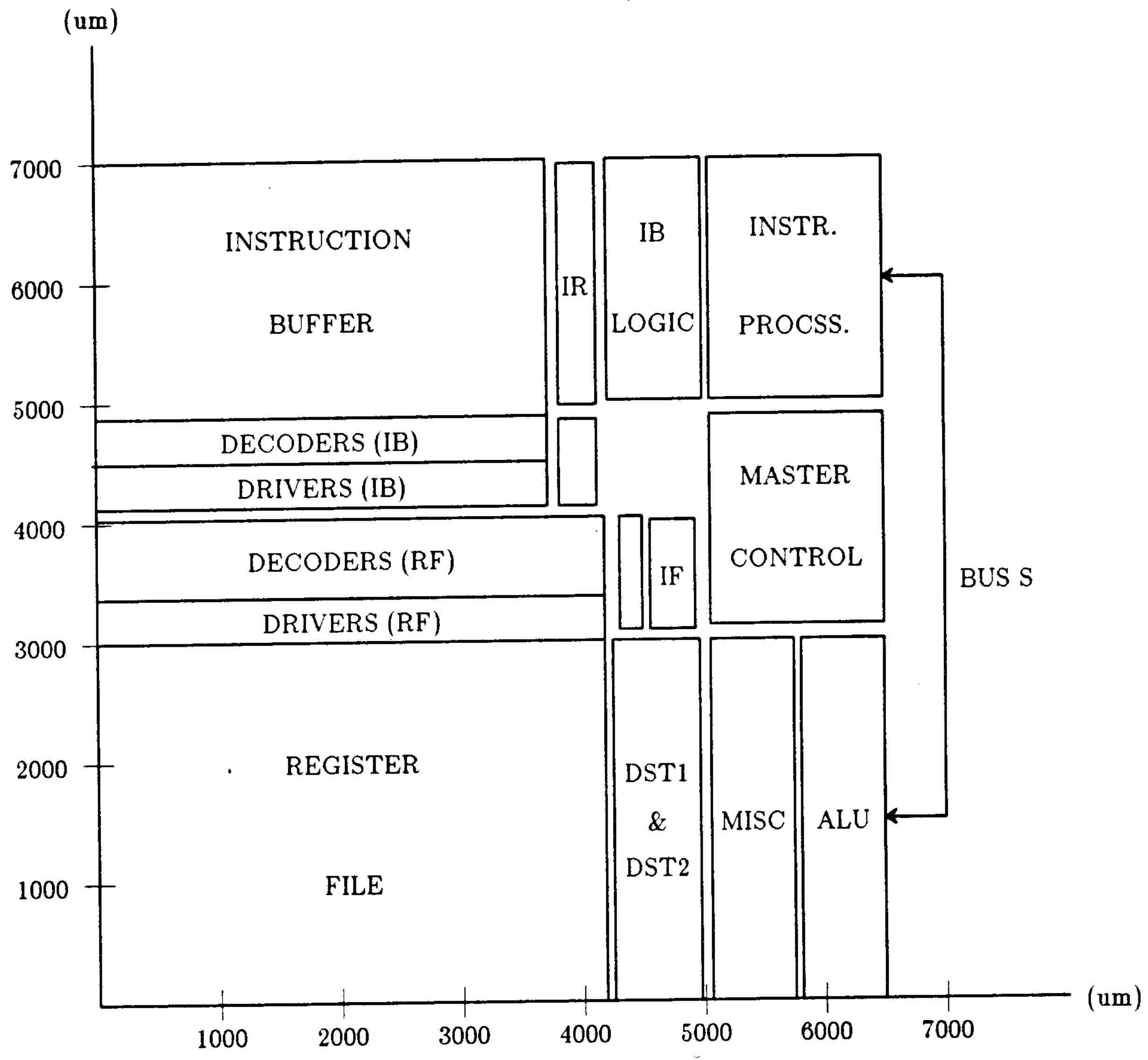


Figure 1-1: Floor Plan of CPU

SPUR avoids this suspension of the pipeline by including an internal instruction buffer on the CPU chip. The CPU then effectively sees two memory ports, one for instructions and one for data. The instruction fetch cycle can then proceed in parallel with the memory access cycle, thereby avoiding the suspension of the pipeline. The instruction buffer allows the two pipeline options shown in Figure_2-2a and b. Both options require that the instruction immediately following a LOAD does not depend on the value being loaded. Both options also require double internal forwarding, although the first option only requires it after a LOAD (internal forwarding is shown by the arrows in Figure_2-2). When the destination of one instruction is the source of the next instruction, the result of that instruction is internally forwarded from the temporary latch. The temporary latch keeps the result to be written into the destination register until a later cycle.

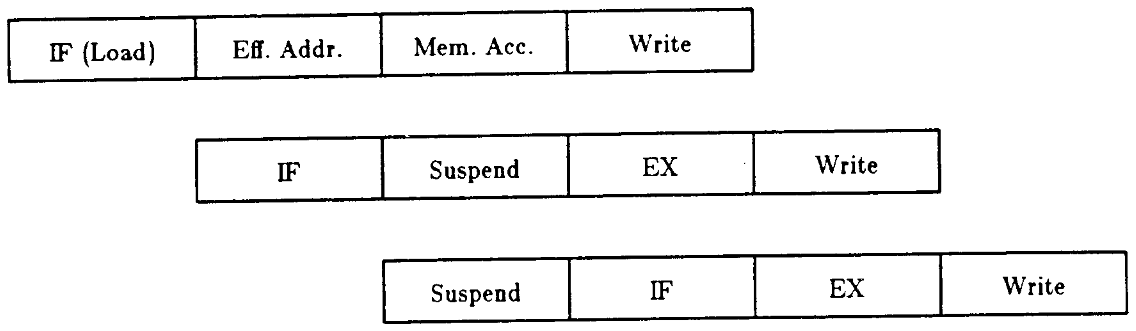


Figure 2-1: RISC-II Pipeline

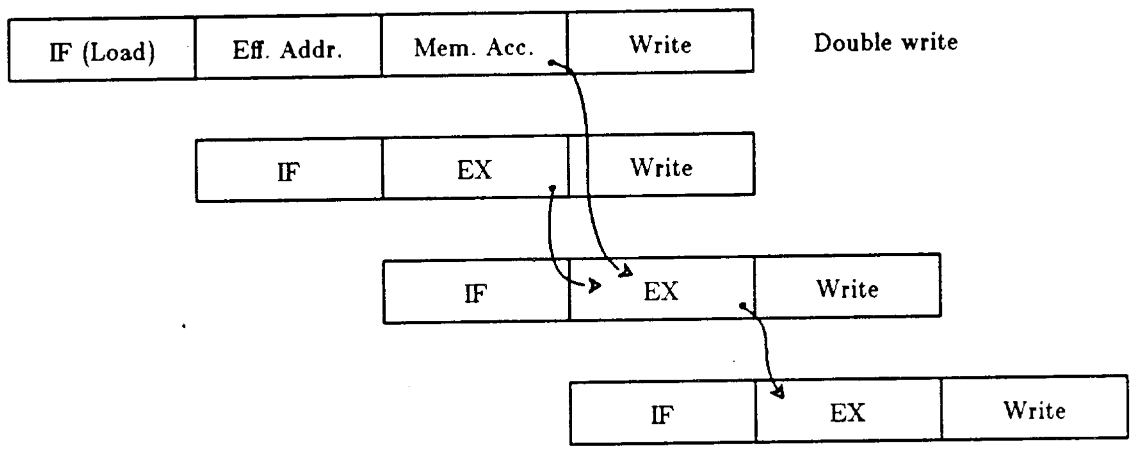


Figure 2-2a: Option 1

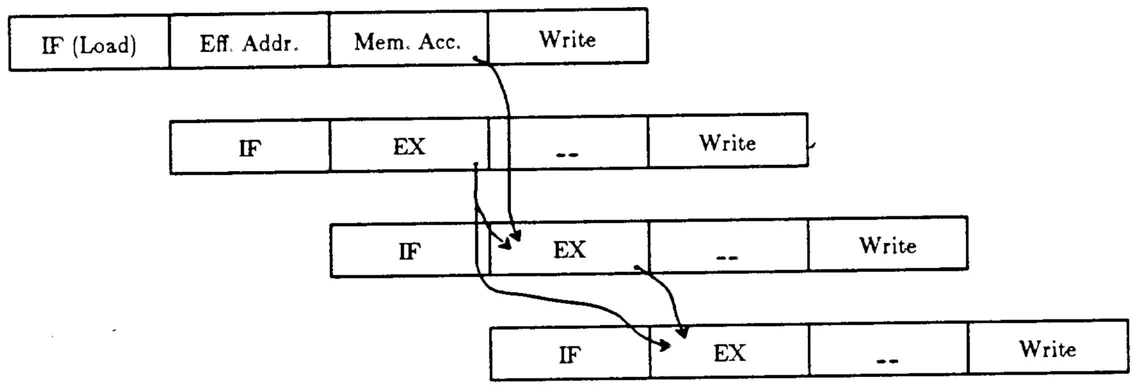


Figure 2-2b: Option 2

The major difference between these two options is the requirement of dual-ported writing capability in the register file. The first option requires two writings of the register file in a single cycle. After extensive circuit simulation, it was found that a register file with dual port writing is not feasible for this application because:

(1) Using a pseudo static, dual port 9T memory cell is safe and easy but requires more control lines. This results in a larger cell area.

(2) Single port writing with a 6T static RAM cell is possible by bootstrapping the word line while writing, but it requires unreasonably large bootstrap capacitance as well as huge bit line drivers if more than one register is written simultaneously.

(3) Other dynamic cells require a refresh cycle once every 1ms or 2ms.

Therefore, the second option was chosen for the SPUR processor. This option requires a dummy pipeline stage to be inserted into all instructions when LOAD is writing into the register file. This dummy cycle not only eliminates the double write requirement on the register file, but it also makes the control of the processor more uniform because all instructions now have the same number of stages. In addition, adding the extra stage to delay the write by one cycle does not degrade the performance of the CPU since the result of each ordinary instruction is still available to the next instruction through internal forwarding.

2.2. Clocking Specification

The clocking scheme chosen for the SPUR processor is a direct result of its register file operation, which is in turn a direct result of the pipeline operation. As shown in Figure_2-3, the pipeline selected uses the register file twice per cycle, writing from instruction1 and reading from instruction3

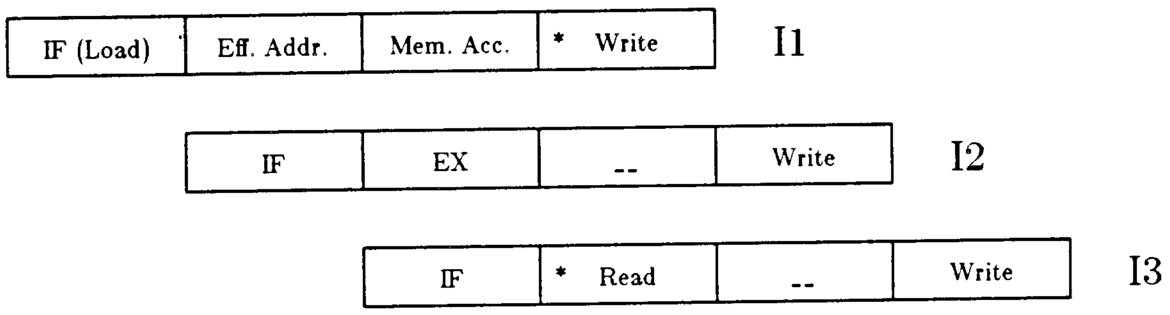
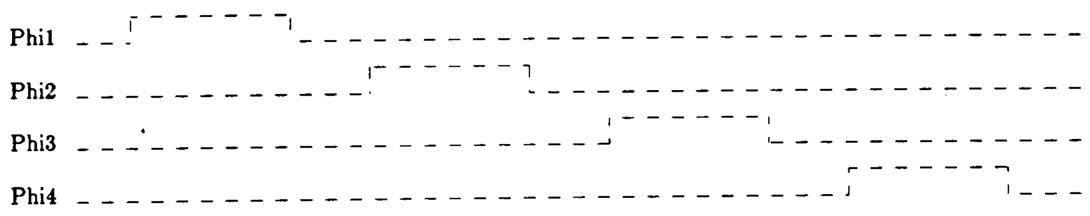


Figure 2-3: Read and Write in One Cycle

To avoid conflicts, at least two phases in each cycle are needed. In order to reduce read access time and to avoid using sense amplifiers, the bit lines are precharged prior to reading. Moreover, in order to keep the writing of register cells fast and safe, the bit lines are also "pre-driven" to their proper values. Due to the two additional phases required for the register file, the four phase clocking scheme shown in Figure_2-4 is being implemented.

In order to ensure that the word line is discharged after read (phi1) and write (phi3), non-overlap is needed between phi1 and phi2, and between phi3 and phi4. Otherwise, pre-



	Phil		Phi2		Phi3		Phi4	
Reg. File	Read		---		Write		Precharge Bitline	
Decoder	Precharge		Decode for WR		Precharge		Decode for RD	
Bitline	Discharge (read)		Predriven		Driven for WR		Precharge to 1	
Wordline	Driven for RD	(1)	0		Driven for WR	(1)	0	

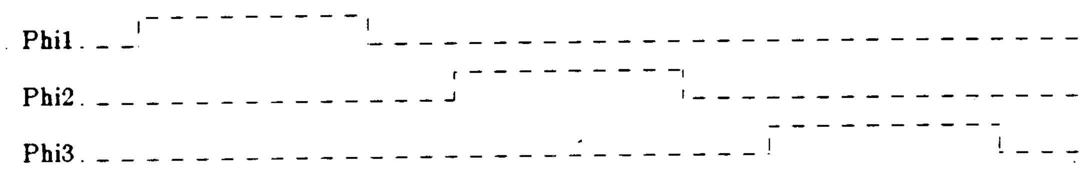
(1) Discharge to 0

Figure 2-4: Four-phase Non-overlapping Clock

drive (phi2) and precharge (phi4) will accidentally destroy the register file contents. As far as the register file operations are concerned, non-overlapping is not needed between phi2 and phi3, nor is it needed between phi4 and phil. However, the instruction buffer operation, which is similar to the register file operation, is read in phi2 and written in phi4. Therefore, it requires non-overlap between phi2 and phi3, and between phi4 and phil. This four phase clocking scheme is well suited for dynamic decoders since two decodings can be done by a dynamic decoder as long as they are not in consecutive phases (due to precharging the decoders). Therefore, in the four-phase approach, only one dynamic decoder is needed.

The four-phase clocking scheme employed in this application is superior to the other clocking schemes. A three-phase scheme is shown in Figure_2-5. The write phase (phi2) will take longer than in the four phase approach since the bit lines are not pre-driven. Consequently, the net gain will not be an entire phase as one might have suspected. Also, two sets of dynamic decoders or one set of static decoders is needed since decoding must be done in consecutive phases (phil and phi3).

An alternative is the two-phase clocking scheme shown in Figure_2-6. During the non-overlap time between phases, the word lines are being discharged. The bit lines cannot be precharged at this time since this might destroy a register's contents if the word line is not discharged to a low enough level. Since the bit lines are neither precharged nor pre-driven, both the read and write phases will take longer than the



	Phi1		Phi2		Phi3	
Reg. File	Read		Write		Precharge Bitline	
Decoder	decode for WRITE		----		decode for READ	
Bitline	Discharge conditionally		driven for WRITE		Precharge to 1	
Wordline	driven for READ	(1)	driven for WRITE	(1)	0	

(1) Discharge to 0

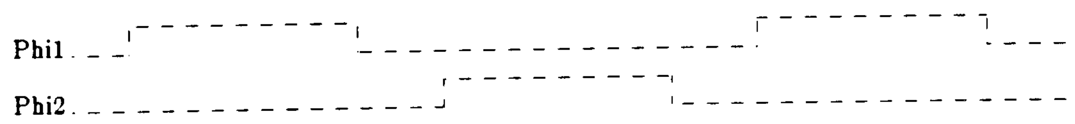
Figure 2-5: Three-phase Non-overlapping Clock

corresponding phases in the four phase scheme. Also, since the register cell's pull-up transistor is small, and the bit line capacitance is high, sense amplifiers might be necessary for reading. Again, either two sets of dynamic decoders or one set of static decoders is needed since decoding must be done in consecutive phases.

3. Clocking Generation

To eliminate clock skew between the four phases of the clock, it is desirable to bring on chip only one phase of the four phase clock and generate the other three phases from this phase. The functional diagram in Figure_3-1 shows the basic inputs and outputs of the clock generation circuit. The reference frequency is provided by a crystal oscillator with a 50% duty cycle running at either 10MHz (100ns) or 6.67MHz (150ns). Phi1 is assumed to be generated off chip, and represents one phase of the desired four phase clock. The outputs of the circuit are the four phases of the clock.

The circuit used to generate the phases of the clock must be process and temperature independent. Therefore, a second-order digital phase-locked loop (PLL) is used to adjust the frequency of a voltage controlled oscillator (VCO) to the given reference frequency. The digital PLL consists of the four major blocks shown in Figure_3-2. The elements of the system are a phase/frequency detector, a loop filter, an inverting amplifier, and a VCO. It should be noted that the inverting amplifier is used only as a buffer stage. The



	Phi1		Phi2		Phi1
Reg. File	Read		Write		Read
Decoder	decode for WRITE		decode for READ		decode for WRITE
Bitline	Driven by reg. cell		Driven by write driver		Driven by reg. cell
Wordline	driven for READ	(1)	driven for WRITE	(1)	driven for READ

(1) Discharge to 0 or drive to proper value for the next phase

Figure 2-6: Two-phase Non-overlapping Clock

VCO is simply an oscillator whose frequency is proportional to an externally applied voltage. The phase/frequency detector compares the reference frequency and the frequency of the VCO. A signal based upon the difference between the two frequencies is output to the charge-pump loop filter. The loop filter integrates the signal into the control voltage used to adjust the VCO frequency so that it approaches the reference frequency. This is an iterative process, with the VCO frequency nearing the reference frequency with each cycle. The PLL constantly compares the two voltages to provide any compensation needed during operation.

The four phases of the clock are produced by a process independent analog delay line. The delay line and VCO are made from the same basic delay cell and use the same control voltage. Therefore, once the VCO and reference frequencies are matched, the delay line has a known delay per stage which is the same as the VCO. For example, assume the reference frequency is 10MHz. Then the VCO, which contains 10 stages of delay cells, is at a high output for 50ns and a low output for 50ns. Therefore, it is running with a 5ns per stage delay. This delay per stage is the same for the delay line, which contains 20 stages. Taps are taken off of the delay line every five stages to uniformly divide the entire delay of twenty stages into four equal delays as shown in Figure_3-3. Phi1 is fed into the delay line with the desired high time of the phase. The high time must be less than one-fourth of the overall clock cycle to provide a non-overlap time between phases. After five stages of delay (or 25ns for this example), phi2 is obtained. As shown in Figure_3-3, phi2 is identical in shape to the input waveform, but delayed by one-fourth of the total clock cycle. Phi3 is obtained after ten delay stages (50ns after phi1), which is

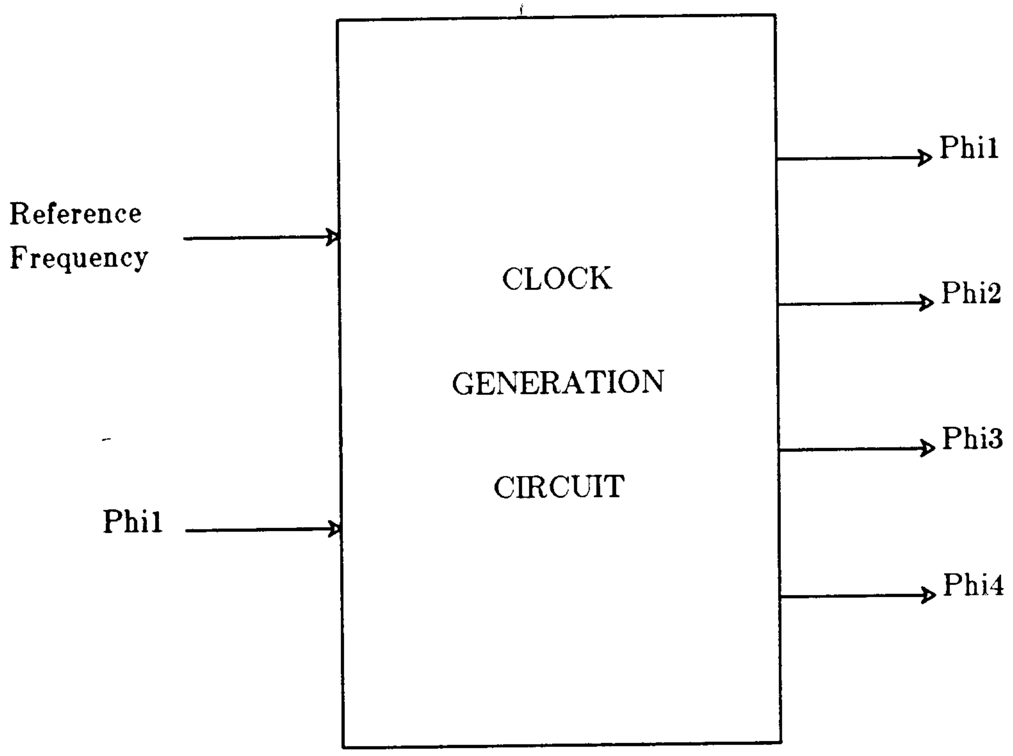


Figure 3-1: Functional Diagram of Clock Generator

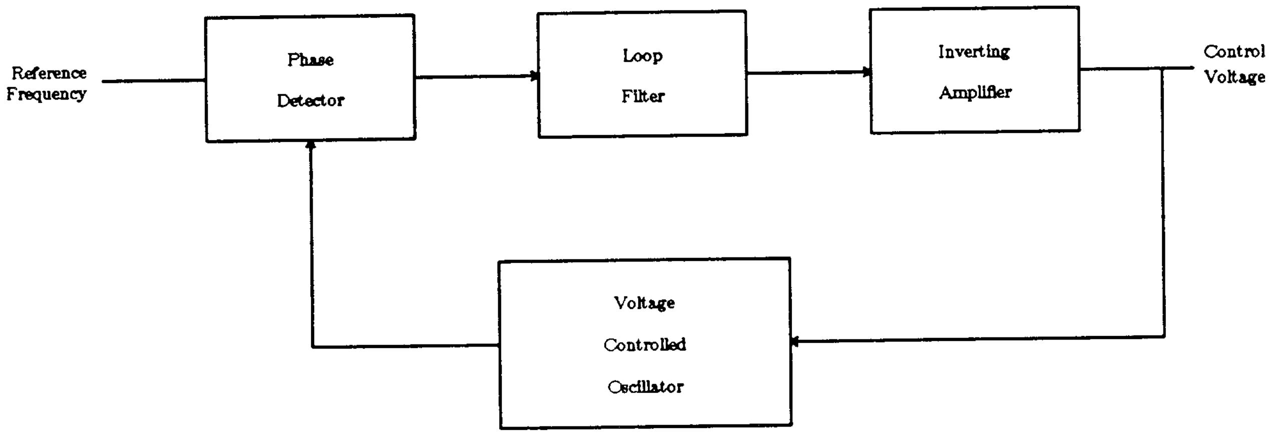


Figure 3-2: Phase-Locked Loop System

one-half of the clock cycle. Phi4 is obtained after fifteen delay stages (75ns after phi1 for this example). As shown in Figure_3-3, after twenty delay stages, an entire clock cycle of delay has occurred and phi1 is obtained. In order to keep the loading and driving abilities of all phases as close as possible, Phi1 is being delivered to the rest of the chip from this

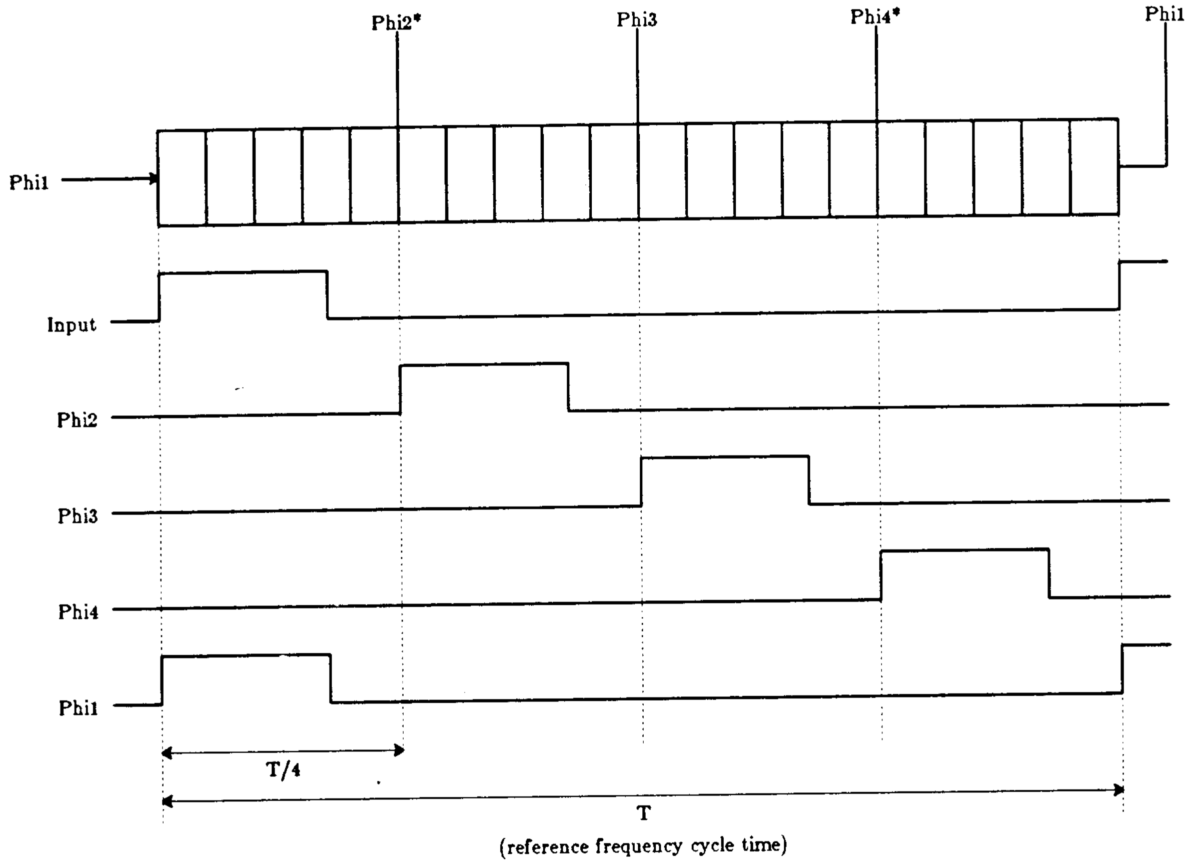


Figure 3-3: Twenty Stage Analog Delay Line

point rather than the initial point where it is brought on chip. This step will help to prevent clock skew between phases. Although the inversions of phi2 and phi4 are obtained from the delay line, there is no additional delay in order to generate the "true" values of the phases. They are generated form the clock drivers in the same way that the complemented values are generated for phi1 and phi3. Figure_3-3 showed the true values of the phases since it is easier to visual the delay if the waveform being considered is identical to the input, rather than its complement. It should be noted that if the reference frequency in the above example had been 6.67MHz, then the delay per stage would have been 7.5 ns. The delays between rising edges of the clock phases are therefore automatically divided into one-fourth of the total clock cycle.

The distribution of the clock will be done in two stages. First, the clock generator is buffered to drive a 3pf load in 2ns. This load approximates the expected routing and gate capacitance. Second, there are distributed buffers before the clock enters main circuit blocks, such as the register array. This method of clock distribution is beneficial in two ways. The two stages of buffering permit the use of optimally scaled buffers for the second stage. Also, the clock signal is regenerated before being applied to the circuits.

In the event the clock generator does not work, the clock can be generated off-chip. The simplest method of off-chip clock generation would be to use 4 one-shots to generate the four phases from the reference frequency. Since the pulse widths of the one-shots are set by a resistor and capacitor, they can be varied extensively by using a potentiometer for the resistor. Alternatively, discrete logic gates could be used to generate a delay based on the reference frequency generated by the crystal. Then phi1, which is already being generated off-chip, can be fed into the delay circuit to obtain the other four phases.

SPUR Cache Controller Datapath Description

Sinohe Villalpando
Tom Wisdom

CS 292-I project report
Computer Science Division - EECS
University of California
Berkeley, California 94720

June 25, 1985

1. Introduction

This paper describes the design and implementation of the SPUR Cache Controller Datapath. Section 2 describes the floorplan. Section 3 describes the the major blocks in the datapath including the register cells, shifter, counters, and interrupt logic. Section 4 describes control of the datapath and Section 5 concludes with a timing description.

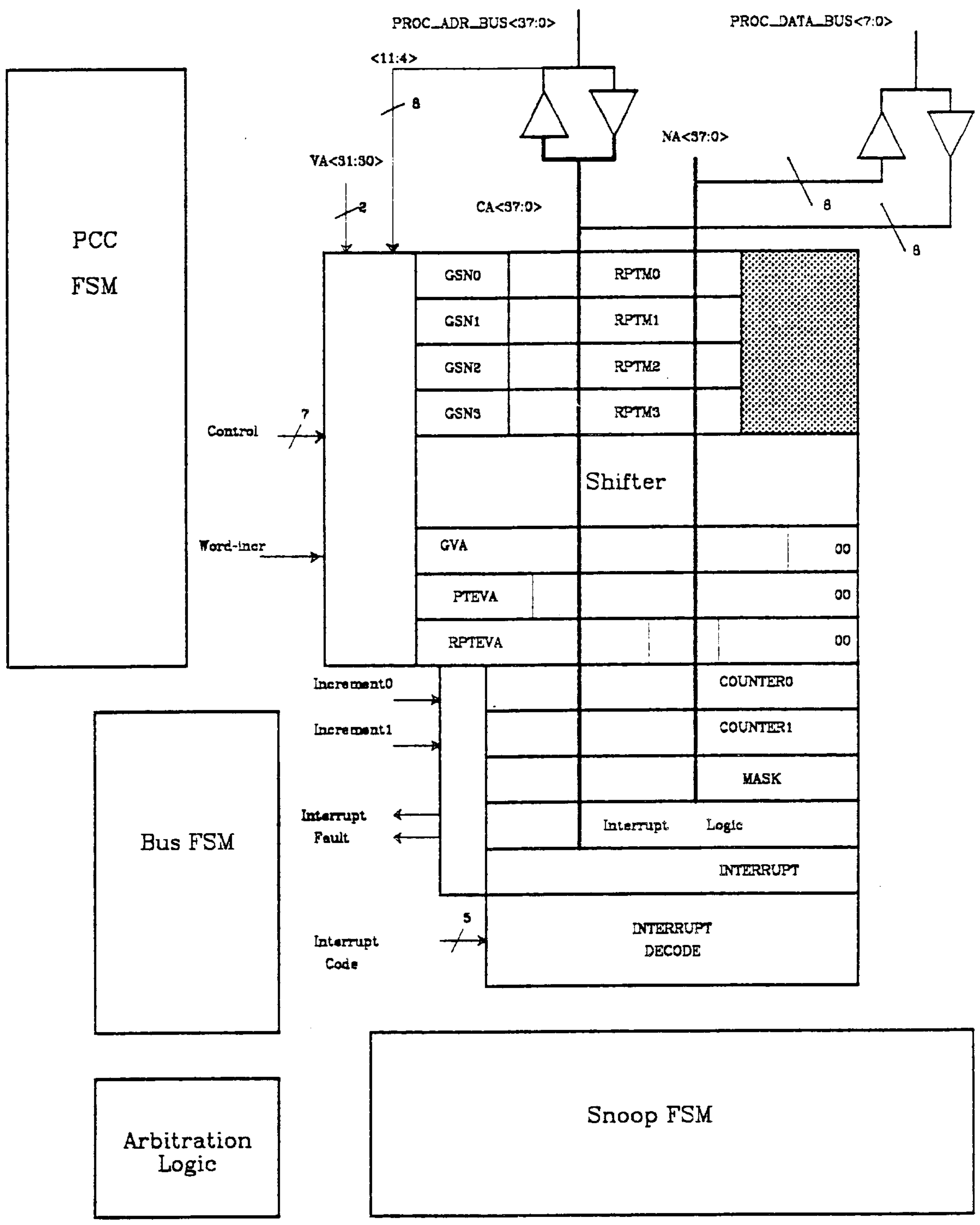
2. Floorplan

The floorplan of the Cache Controller is shown below. Two major busses run vertically in metal-2. The CURRENT ADDRESS (CA) bus is driven by either the PROCESSOR ADDRESS BUS (Proc_Adr_Bus), the PROCESSOR DATA BUS (Proc_Data_Bus) buffer, or a selected register in the datapath. The NEXT ADDRESS (NA) bus is driven by the Shifter and drives the Proc_Data_Bus output buffer and the inputs to the datapath registers.

Control lines are run horizontally in metal-1 along with Vdd and GND lines. The number of control lines has been reduced by running a single Store Register (STR) line for each register. This STR line is ANDed with a Byte Select line for each byte of a register to creat the "load" signal for that byte. The Byte Select lines are run vertically in the datapath. This prevents having to run a separate STR line for each byte of a register through the whole register. Control lines for the Shifter are reduced in half for each shift operation by using n-channel pass gates.

The Finite State Machine PLA's are placed around the datapath according to the control signals which are needed by the datapath. The Processor Cache Controller (PCC) FSM is placed next to the address decoders for loads and stores, and to generate the horizontal control signals for virtual memory operations. The Bus Controller FSM is placed near the Interrupt Registers and near the PCC since there is a lot of communication between them. The Snoop FSM communicates primarily with the Bus Controller and so is placed near it. Another concern is the access to the pins on the chip. The FSM's are spread out around the edge of the chip since most of the pins connect to the FSM's.

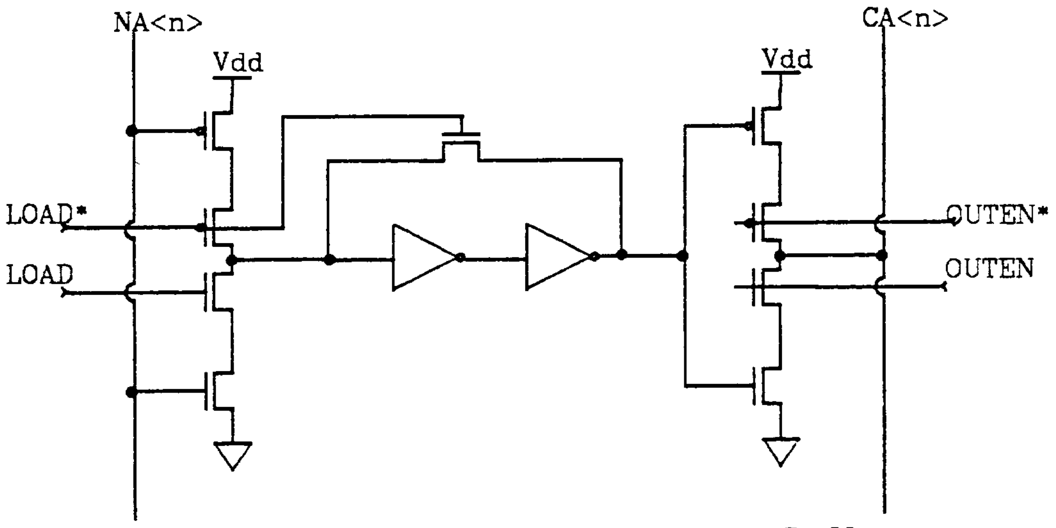
Cache Controller Floorplan



3. Datapath Blocks

3.1. Register Cells

The register cells for RPTEVA, PTEVA, GSN, and RPTM are the same. They are loaded from the NA bus and are stored onto the CA bus. The schematic for the standard register cells is shown below.



Standard Register Cell

The signals needed to either load or store the cells are the same, but these control signals are generated differently. The control signals to either load or store a register are generated by the control circuitry on the left of the registers. The individual byte within the register is selected by the byte select signal running vertically.

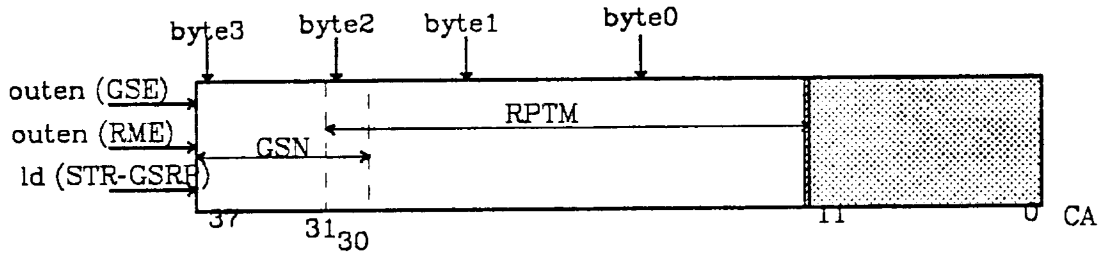
The registers can be loaded from either the processor or from other registers in the data path (see the Appendix for address mapping). In both cases the data must pass through the Shifter before being loaded into the appropriate register from the NA bus. All of the registers' bytes can be read or written by the processor in 8-bit bytes. Certain variable-length bit fields of the registers are loadable from other registers. Data from another register within the datapath is loaded with a "latch" signal. Data from the processor data bus is loaded with a "load" (via processor STORE REGISTER instruction) signal. In both cases, the load and the latch signal may be combined with the byte select signals to select the appropriate bits to be loaded.

3.1.1. GSN and RPTM Registers

The GSN registers are located to the left of the RPTM registers. The GSN registers and the RPTM registers are 8-bits and 20-bits wide respectively. The RPTM registers consists of 3 bytes. These 3 bytes must be loaded independently, thus 3 different load signals will be required to do the loading into the RPTM registers. The GSN and RPTM registers are mapped to a single physical address from the processor's view. A single load signal (**ld(STR_GSN_RPTM)**) is asserted when the processor does a store register operation to this address.

Since there are 4 bytes and they must be loaded separately, the 4 signals are generated to the left of the byte where they are needed. These load signals are generated by ANDing the **LD** signal with the byte select signal on each of the bytes.

A control signal floorplan for the registers GSN and RPTM is shown below:



GSN and RPTM registers

The placement of the registers in the datapath is indicated by the numbers below the boxes. When any of these bytes need to be loaded, the **LD** signal is asserted. This **LD** signal is ANDED with each of the byte select signals, and the output of the AND gates is used as the load signal for the standard register cells on each byte.

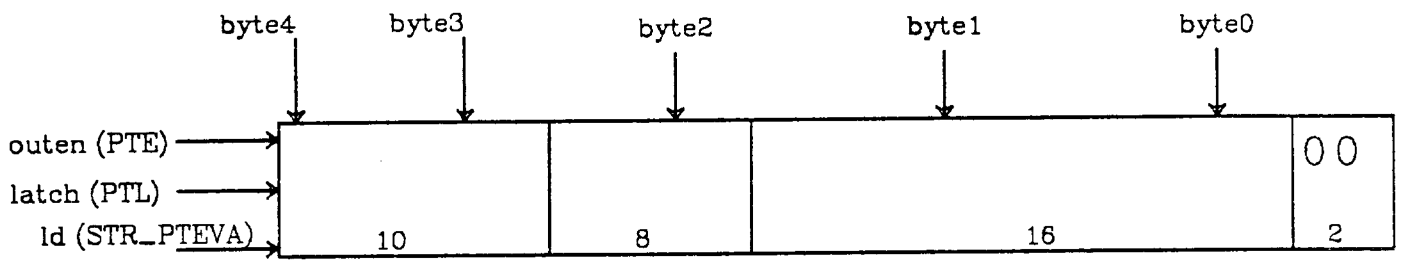
The GSN registers require a separate **OUTEN** signal than that of the RPTM registers because on a virtual address request, the processor will be driving the lower 30 bits of the CA bus, and the RPTM registers should not be driving it at the same time. During this time, the upper 8 bits of the CA bus will be driven by one of the GSN registers.

3.1.2. PTEVA Register

The PTEVA register is 38 bits wide. When read, all the bits are output enabled into the CA bus. The **OUTEN** and **OUTEN*** signals for all the register cells in the PTEVA register are the same.

The lower 2 bits of the PTEVA register are always 0, and do not need to be loaded. Because the data of the lower 28 bits to be loaded into the PTEVA register come from the processor address bus, they are all available at one time, and they can all be loaded into the PTEVA register at the same time. Only one signal, **PTL** (and **PTL***) is required by all these bits. However, to load the bytes in the PTEVA register with data coming from the processor data bus, they must be loaded in 8-bit bytes.

A floor plan of the PTEVA register with the control signals is shown below.

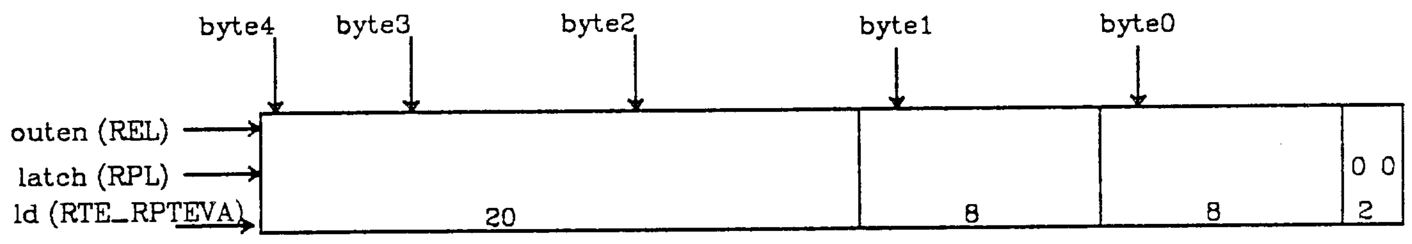


PTEVA register

3.1.3. RPTEVA Register

The RPTEVA register is 38 bits wide. It also output enables all of its register cell data onto the CA bus at once.

The lower two bits of the RPTEVA are always zero. There is no need for loading data in them. Bits 3 to 17 come from the PTEVA register. After being passed through the shifter, they can be loaded simultaneously. Bits 18 to 37 come from the processor data bus and they come in 8-bit bytes.

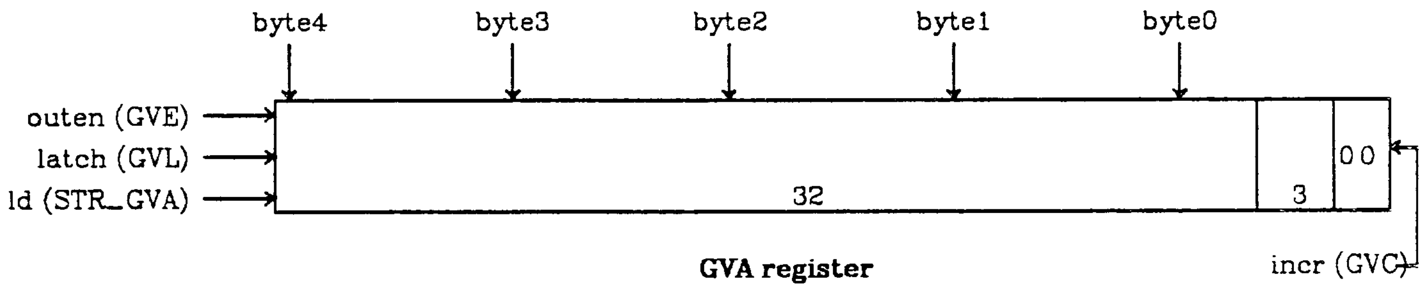


RPTEVA register

3.1.4. GVA Register

The GVA register is also 38 bits wide. Only 1 output enable signal (**GVE**) is needed to read this register. To latch the register, only 1 load signal is required (**GVL**). Bits 0 to 29 come from the processor address bus. The 8 upper bits come from one of the GSN registers. When the global virtual address is needed, the processor provides the lower 30 bits. These bits, together with the 8 bits of the GSN register present on the CA bus, are then latched into the GVA register directly from the CA bus. This is different from the other registers which are latched from the NA bus with data from the shifter. The reason for the difference is that the shifter is being used to generate the value for the PTEVA register at the same time. The consequence is that the GVA register is not directly modifiable (processor STORE) by the processor. It can be read by the processor, but the only way to modify it is with a normal cache access.

The lower 3 bits of the GVA (bits 2-4) are really a counter. This 3 bit counter is controlled by the BUS FSM during cache block updates. Since the BUS FSM runs off the BUS clock, there is some circuitry to synchronize the counter control signal.



GVA register

3.2. SHIFTER

The shifter in the cache controller datapath is a special purpose shifter. Because there is no need to shift every bit of the CA bus into every bit of the NA bus, the operations of the shifter are limited to those needed by the cache controller datapath.

All of the operations of the shifter are listed below:

CA <37:10> ===> NA <27:10>
CA <37:24> ===> NA <5:0>
CA <31:24> ===> NA <7:0>
CA <23:16> ===> NA <7:0>
CA <15:8> ===> NA <7:0>
CA <7:0> ===> NA <7:0>
CA <5:0> ===> NA <37:32>
CA <7:0> ===> NA <31:24>
CA <7:0> ===> NA <23:16>
CA <7:0> ===> NA <15:8>

Except for the first operation of the shifter shown above, the shifter performs byte shifting for processor loads and stores to the cache controller registers. Only the bits required will be shifted and driven on to the NA bus. The other bits on the input CA bus can have any value without affecting the resultant output on the NA bus.

The shifter is formed by the CA and the NA bus running perpendicular to each other. There are n-channel pass transistors from the CA to the NA bus at some of their intersections. When a shift operation is needed, the pass transistors will be made active at the appropriate intersections.

Because there are only 10 different shift operations needed, only 10 control signals are needed in the shifter. If need for more shift operations arises, the shifter can accommodate these by adding additional n-channel pass transistors with their required control signals. This will increase the height of the shifter but will not affect its width.

3.3. COUNTERS

The two counters in the cache controller's datapath will be located below the RPTEVA registers and above the interrupt logic registers.

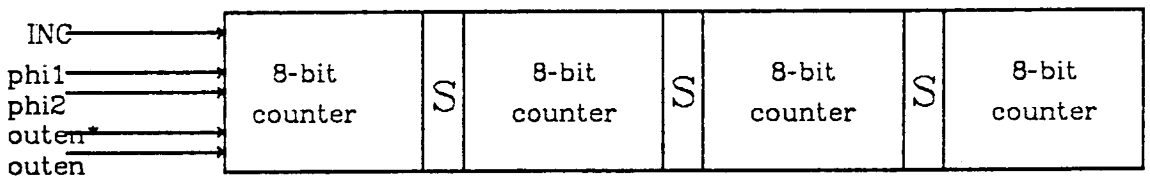
Since the counters are only to be read, and reading is only 8-bits at one time, they will be output enabled into the CA bus. The NA bus does not need to go through the counters. The bits of the counter cell will all be output enabled into the CA bus at the same time. Only one output enable (and its complement) signal is needed for the counters as control signal. The counter works with a two-phase nonoverlapping clock cycle.

The counters are 32-bits wide. They are formed by four pipelined 8-bit counters. The overflow signal of an 8-bit counter is saved in a save register, and used as the increment signal of the next higher order 8-bit counter on the next clock cycle.

Because the time for a signal to propagate from bit 1 to bit 32 of a counter is long, the counters were made of 8-bits counters so that it is possible to read the 8 lower bits of the counter while the signal is still propagating through the other 24 bits.

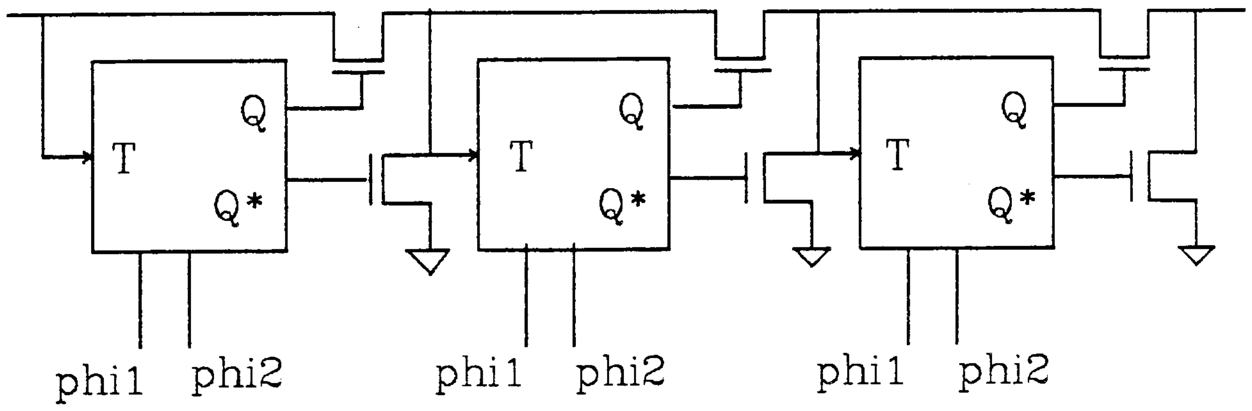
The save register between each 8-bit counter is used to save the overflow bit. The bit stored will be used by the next higher order 8-bit register. It is because of this that we can read a 8-bit register even though the signal has not been propagated through the next higher order 8-bit register. The save register also serves as a buffer for the propagate signal.

A block diagram of the 32 bit counter is shown below with the needed signals for its operation.



32 BIT COUNTER

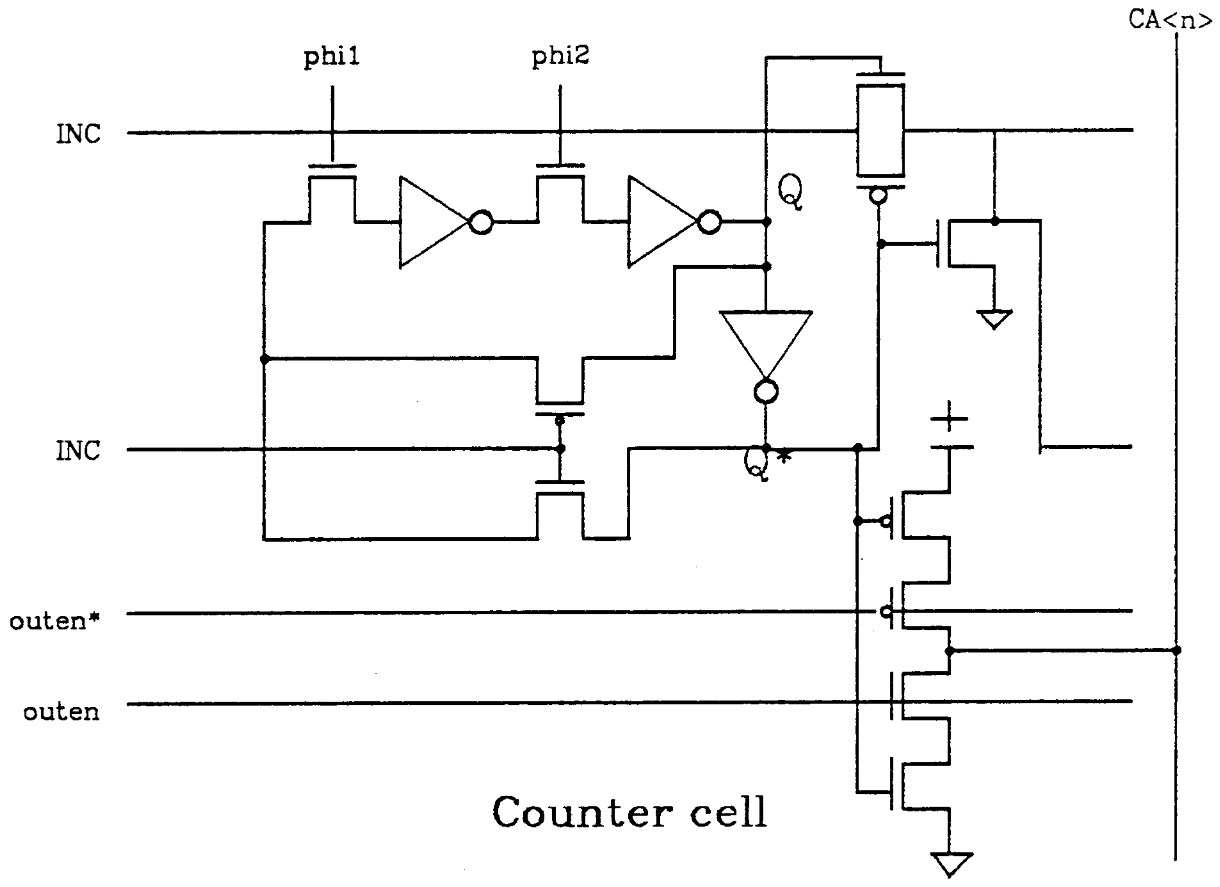
The counter cells are made of a toggle register which is toggled depending on the propagate signal coming to it from the previous lower order counter cell. The propagate signal can either be propagated or killed by a counter cell depending on the state of the cell.



3 bit counter

A signal is propagated if all the bits on the past cells are at a value 1. This means that the first cell with a value "0" will toggle its value on the next phi2 clock phase. This cell will then kill the propagate signal so that no higher order bit cells are toggled.

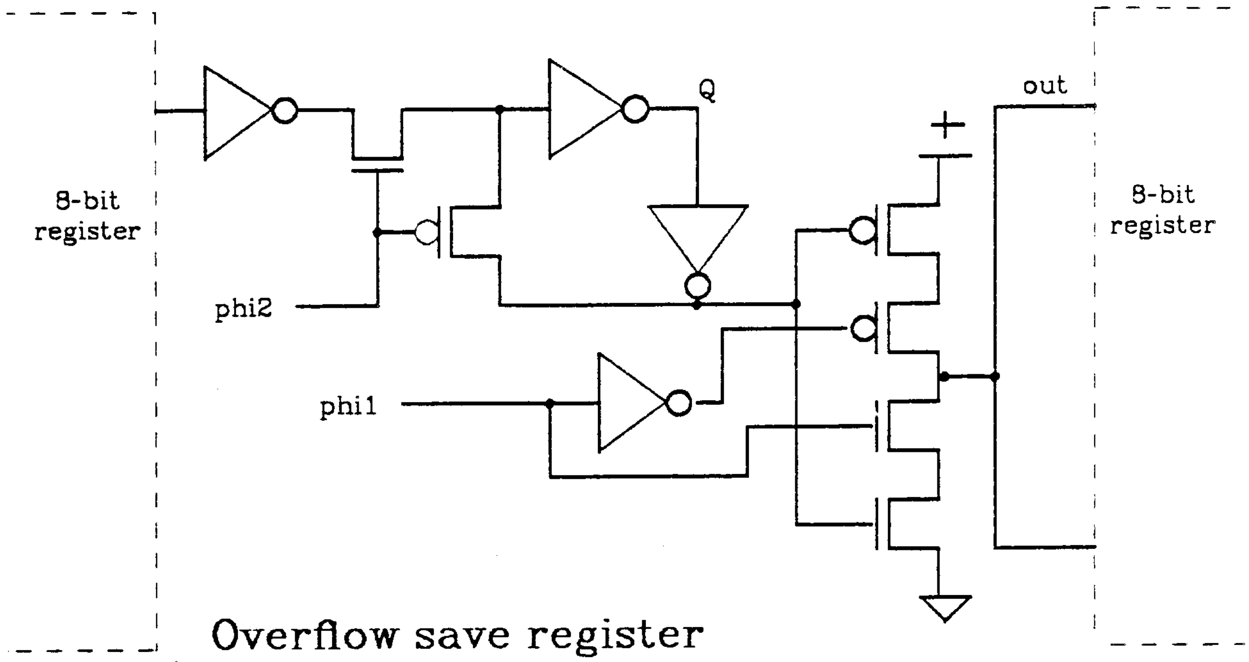
The circuit schematic of a counter cell looks like this:



Counter cell

The cmos latch is used to read the data bit of the cell on to the CA bus.

The register used to save the overflow bit of the 8-bits counters is shown below.



Overflow save register

The cmos latch is used to provide a strong propagate signal for the next 8 counter cells.

The requirement of the counter cells is that the increment signal must be made active during phi1 of the clock cycle. The data will be updated during phi2 of that same clock cycle. Data cannot be read until after phi2 of the clock cycle when the counter was incremented. This is necessary to obtain the new updated data on the counters. The data must have been propagated through the 8-bits to be read. It is better that the cells are read on either phi1 or phi2 of the next clock cycle. But since back to back increments can occur, the cells should only be read during phi1.

Data is read out of the counters by the **outen** and the **outen*** signals. By this, the data of the 32 bits of the counter will be on the CA bus. The data will go through the shifter and only the 8 appropriate bits will be transffered to the NA bus. Note that if an increment signal comes in, on the next clock cycle the propagate signal will have traveled through the first 8-bit counter and it will also be loaded into the save register. At this time, the 3 higher order 8-bit registers will not have been updated. If the counter needs to be read, the output enable signal and its complement will read the data on all the 32 bits of the counter into the CA bus. Only the 8 lower bits will be correct because the others will not have been updated yet. However, only the 8 lower bits will be transferred to the 8 lower bits of the NA bus. On the next clock cycle, the bit stored on the first save register will be used as the increment on the next 8-bit counter and the signal will propagate through the counter and the overflow signal will be stored in the next save register. At this time, the full 32 bits of the counter will be read into the CA bus, but this time the bits of the second least significant 8-bit counter will be shifted into the 8 lower bits of the

NA bus. The same procedure will be repeated for the other two 8-bit registers.

The problem of reading a value which has not been updated yet on the counter is eliminated if the counter is read first by its 8 least significant bits, and then by the second 8 least significant bits, and so on. The updating of the counter cells is done by placing a high value on the increment signal of the least significant 8-bit counter during the phi1 clock phase.

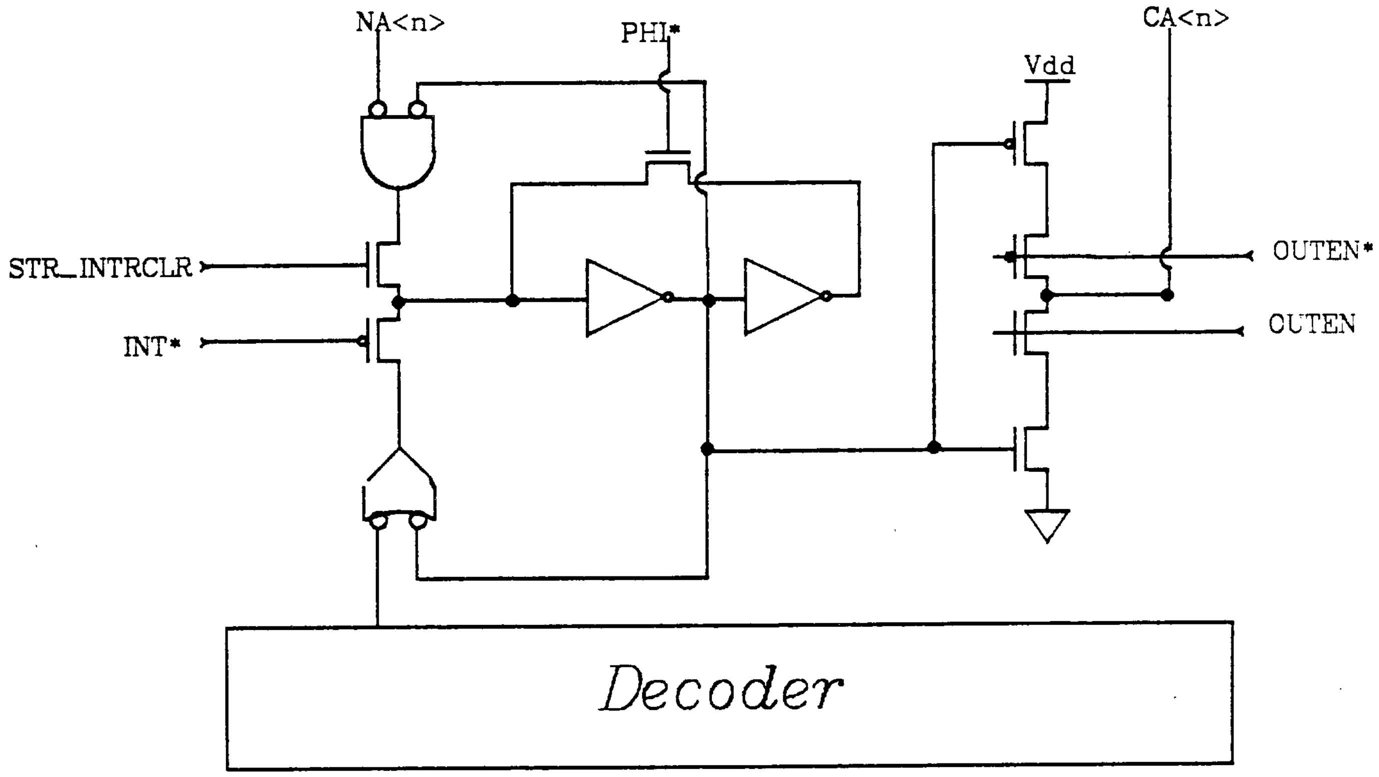
The cells of the counter are the ones that determined the pitch of the datapath. The width of the 32-bit counter, including the save registers is 1196 lambda. All the other registers on the data path are to be pitch matched to the the counters.

3.4. INTERRUPT LOGIC

The SPUR interrupt mechanism consists of a mask register, an interrupt register, and a decoder. The decoder is explained separately. The interrupt and the mask registers are described next.

The interrupt register is cleared by performing the AND function of its present value with that of a byte clearing pattern that is created by a processor store instruction to the interrupt clear register. The interrupt register is set by the ORing of its stored data bit with that coming out of the decoder.

The control circuitry will determine which function is to be performed by sending a signal, either **STR_INTCLR** or **INT***, on the write phase of the clock cycle.



Interrupt Register Cell

To reset or set the interrupt register cell, negative logic on the incoming bits is required. The input to the bit to be either set or reset, needs to be active low. Thus, if a bit is to be cleared, a 1 must be present on the byte clearing pattern. To leave it as it is, a 0 must be present on the byte clearing pattern.

The output of the decoder is 0 for the bit that needs to be set to 1, and all others are 1. The decoder has been designed using NOR array structure to implement this function.

The interrupt register cell is made out of four bytes. Four different **LD** and **LD*** signals and one pair of **OUTEN** and **OUTEN*** signals are required for control.

The mask register cell is a standard register cell. It is used to AND its bit with that of the interrupt register. It consists of four bytes, thus four different **LD** and **LD*** signals and one **OUTEN** and **OUTEN*** signal are required as control.

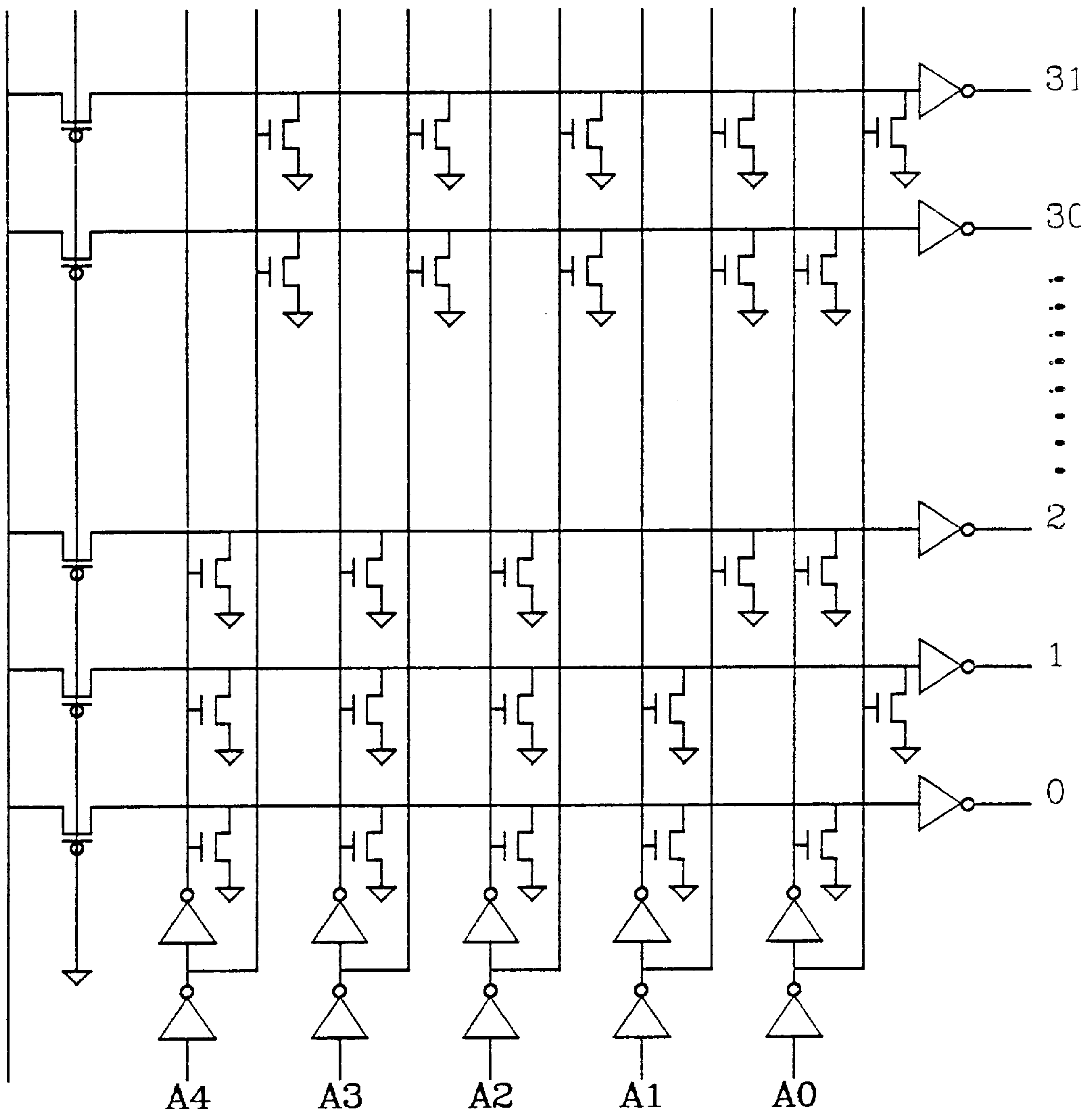
Both the mask register cell and the interrupt register cell are loaded from the NA bus and they both output enable their data to the CA bus. There is a single decoded **LD** and **LD*** signal for each register which is ANDed with the byte select signals to generate the four different load signals for each register.

3.5. INTERRUPT DECODER

The interrupt decoder is a 5-bit to 32-bit NOR array decoder. Only the output to be selected will be low, and all the others will remain high. The interrupt logic has been designed so that the bit to be selected must be active low, and all the others must remain high.

The decoder is made up of a core of n-channel transistors with p-channel transistors as pull-ups. The p-channels pull-ups are pseudo-nmos logic. Their gates are always connected to ground, thus the pull-up transistors are always turned on.

A circuit schematic of the decoder is shown below.



Interrupt decoder

The inverters at the output of the decoder are used because the interrupt logic requires negative logic.

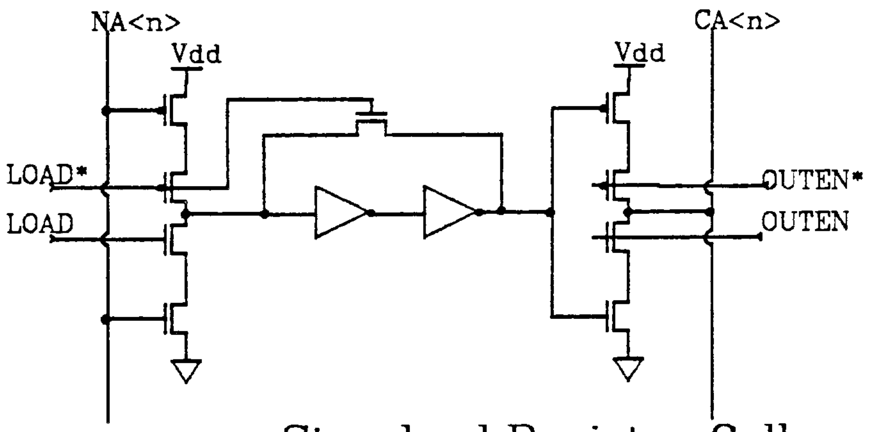
3.6. Basic Cells

The following is a brief description of the different cells that are used in the cache controller datapath. The circuitry for the control of the datapath has not been implemented. The following cells are the ones that make up the core of the datapath

The cache controller datapath consists of the GSN, RPTM, GVA, PTEVA, and RPTEVA registers; the MASK and INTERRUPT registers; the counters; the interrupt decoder; and the shifter.

3.6.1. Standard Cell

The basic cell used for the GSN, RPTM, GVA, PTEVA, and RPTEVA registers is the same. This is the standard register cell shown below.

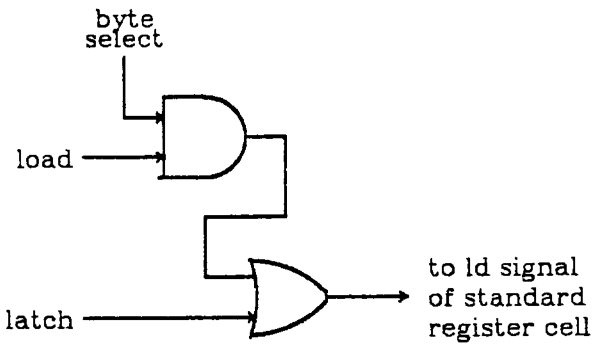


Standard Register Cell

All the registers use this same standard cell, but the controlling signals: **LD**, **LD***, **OUTEN**, and **OUTEN*** are generated differently for each register by the control circuitry. The GVA cell is different in that the NA bus is replaced with the CA bus, since its is loaded from the CA bus.

3.6.2. Byte Select

The selection of a particular byte within a register is done by the use of a cell that will be placed next to that byte. The byte selection cell is shown next.

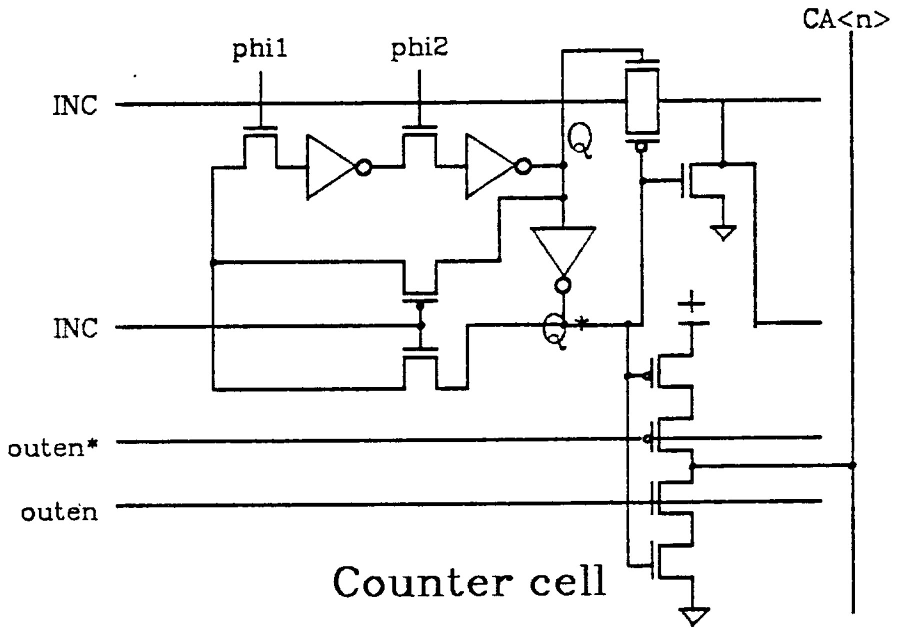


Byte Select Cell

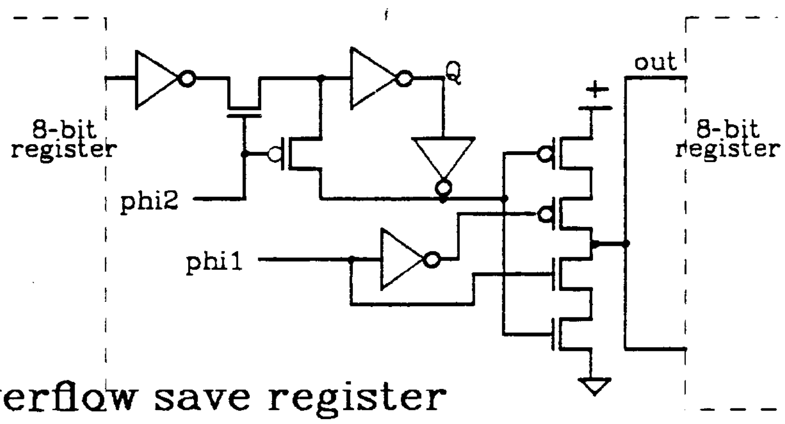
In some registers such as GSN and RPTM the OR gate is not needed because data is loaded into the registers only from the processor data bus. The latch signal is not needed in these registers.

3.6.3. Counter Cells

The counter contains two different cells. They are the counter-cell and the overflow-save-register cell. They are both shown below.



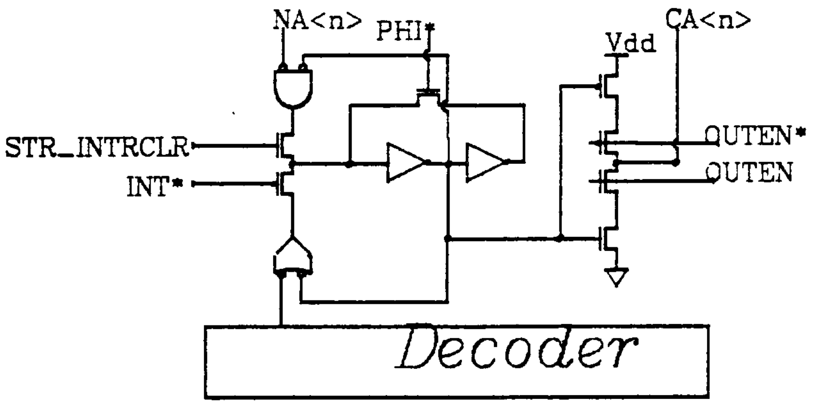
Counter cell



Overflow save register

3.6.4. Interrupt Cell

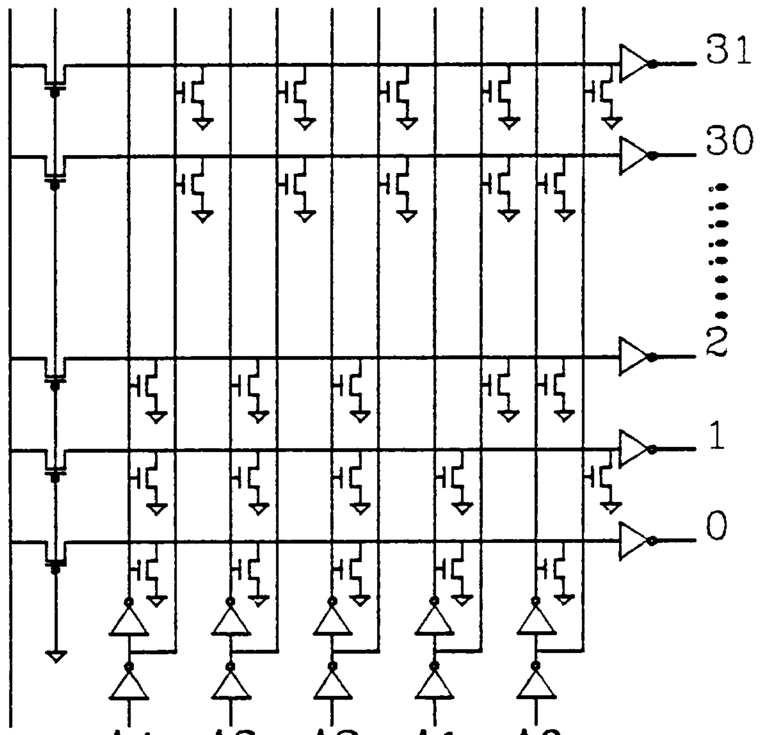
The interrupt logic register and the mask register are combined into one cell called the interrupt register. Only one cell is needed to perform both functions: interrupt and clear. The interrupt register cell is shown below.



Interrupt Register Cell

3.6.5. Decoder

The interrupt decoder is a pseudo n-mos decoder with an inverter used on each of the outputs



Interrupt decoder

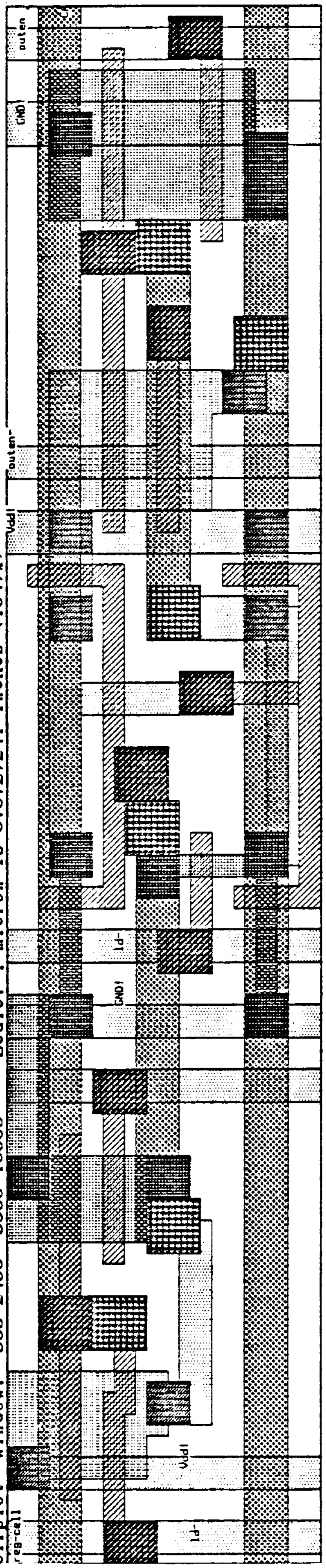
3.7. LAYOUTS

The counter cell is one of the most important cells of the cache controller datapath architecture because it determines the pitch of the cells in the horizontal direction. The layout of the counter cell is shown in the next page. The interrupt cell and the standard register cells layouts are also shown. However, because these are not pitch matched to the counter cells, the layout of these two cells will be redesigned.

The width of the datapath is 1400 lambda not including the control circuitry. The height cannot be determined until the shifter and interrupt decoder are laid out. The estimate is approximately 2500 lambda (if 2 counters are used) not including the FSM's or Bus logic.

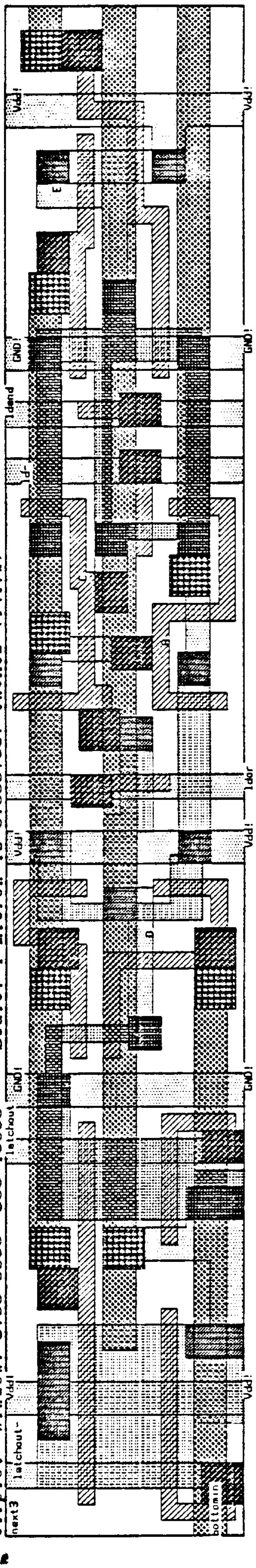
The following pages show the layouts of the cells in the datapath. All dimensions are in lambda. A layout of a full 32-bit counter, including the overflow-save-registers, is included because it determines the width of the datapath.

cifplot• Window: -500 2400 -3900 10600 --- Scale: 1 micron is 0.0727241 inches (1847x)



Standard Register Cell
height 145λ
width 29λ

oifplot* Window: 3700 6600 -300 18600 --- Scale: 1 micron is 0.0557937 inches (1417x)

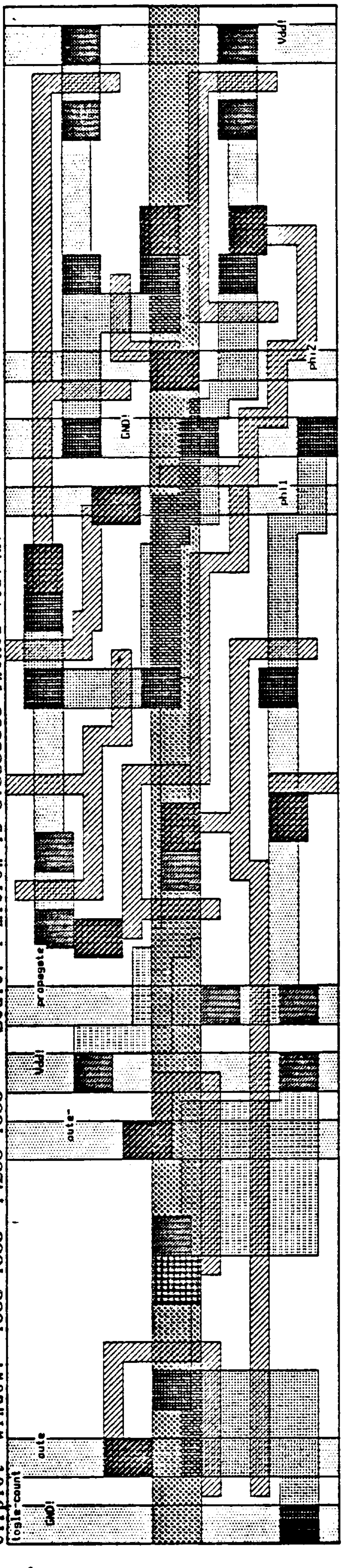


Interrupt Register Cell

height 189 λ

width 29 λ

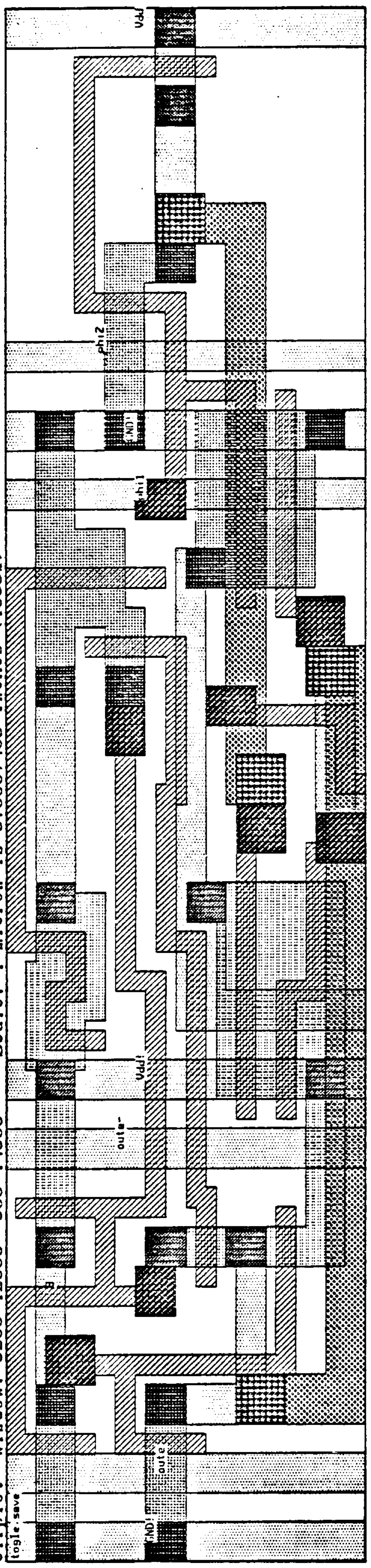
cifplot* Window: -1800 1600 -14200 1800 --- Scale: 1 micron is 0.0659063 inches (1674x)



Counter Cell

height 160 λ

width 34 λ

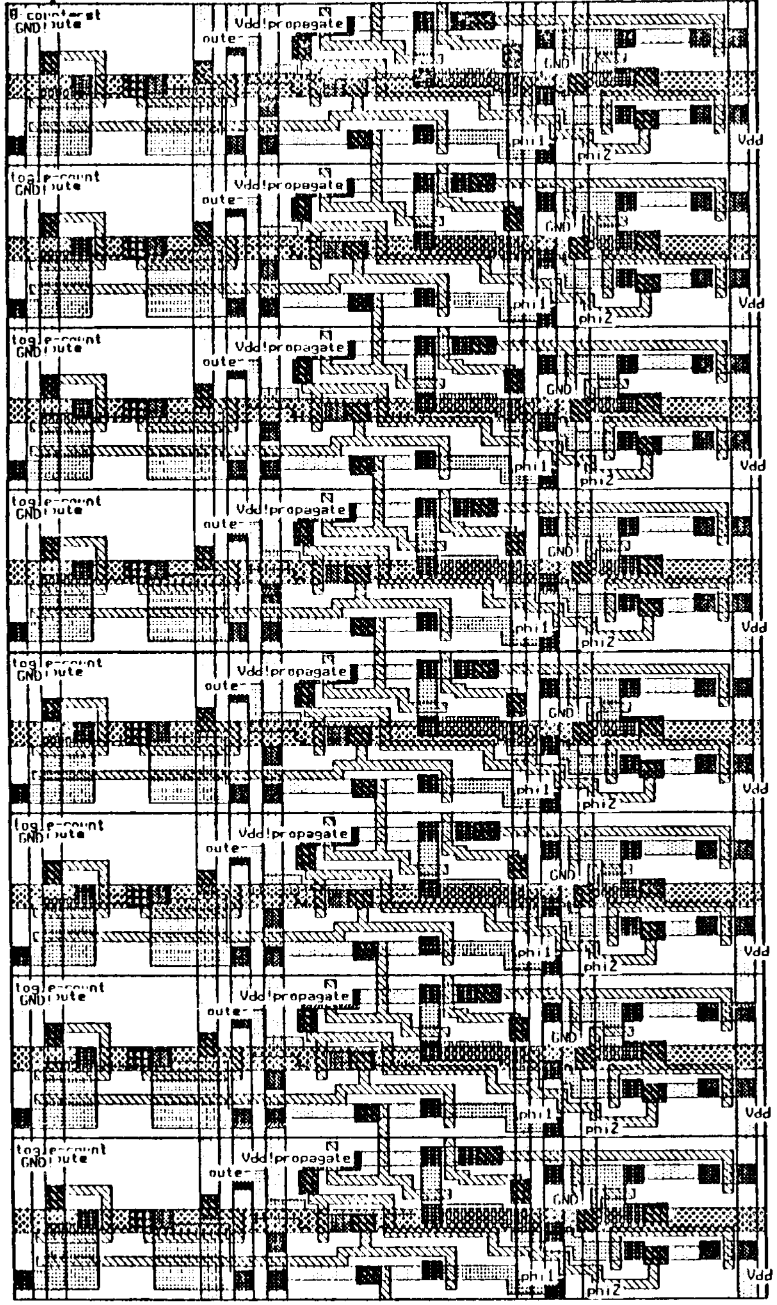


cifplot* Window: 9200 12800 -900 14900 --- Scale: 1 micron is 0.0667405 inches (1695x)

Overflow - Save Register

height 160λ
width 36λ

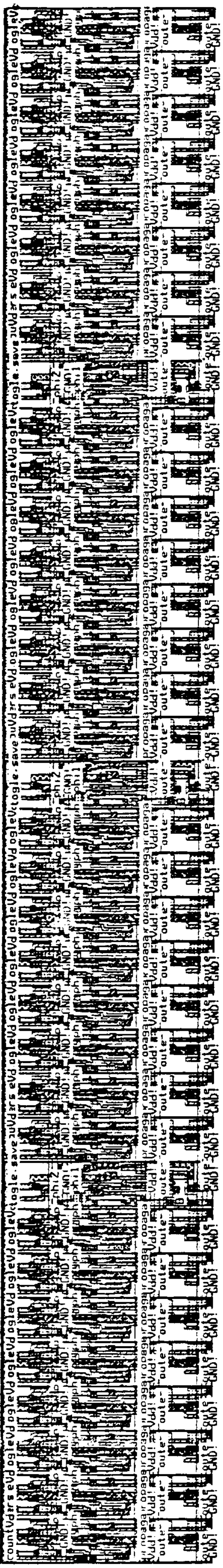
cifplot* Window: -3500 23700 -2200 13800 --- Scale: 1 micron is 0.025 inches (635x)



8-biT CounTer

height 160 λ
width 272 λ

cifplot* Window: -11600 4400 -11700 107900 --- Scale: 1 micron is 0.00881689 inches (224x)



32-BIT COUNTER

height 160 λ
width 1196 λ

4. Datapath Control

The control signals for the CC datapath shown below (table 1.) are generated by the PCC FSM. Instead of generating each signal independently, they are derived from the STATE bits (VAref, PTEref, etc.). For example, The signal **S10** for shifting 10 bits is generated by ORing **VAref** and **PTEref**. The STATE bits could be reduced further by encoding the 7 states into 3 bits. But this would require a longer delay in the control signal path and the savings is only 4 bits of output from the FSM.

The Global Virutal Enable (**GVE**) and Count (**GVC**) are generated by the Bus Controller FSM to control address generation for cache block updates and replacements. Since the Bus FSM runs on a different clock than the PCC, these signals must be synchronized to the PCC clock.

4.1. Control Signal Descriptions

ADR - Processor Address Direction: 0 for input to data path, 1 for output.
0: PROC_ADR_BUS<29:0> => CA<29:0>.
1: CA<29:0> => PROC_ADR_BUS<29:0>.

GSE - GSN Register Output Enable. PA<31:30> selects which one.
1: GSNx<7:0> => CA<37:30>.

RME - RPTM Register Output Enable. PA<31:30> selects which one.
1: RPTMx<16:0> => CA<27:12>.

GVE - GVA Register Output Enable onto CA<37:0>.
1: GVA<37:0> => CA<37:0>.

PTE - PTEVA Register Output Enable onto CA<37:0>.
1: PTEVA<37:0> => CA<37:0>.

REH - RPTEVA Register High Part Output Enable.
1: RPTEVA<37:12> => CA<37:12>.

REL - RPTEVA Register Low Part Output Enable.
1: RPTEVA<11:0> => CA<11:0>.

GVL - GVA Register Latch.
1: CA<37:0> => GVA<37:0>.

GVC - GVA Register Increment.
1: GVA<4:2> + 1 => GVA<4:2>.

PTL - PTEVA Register Latch.
1: NA<27:0> => PTEVA<27:0>.

RPL - RPTEVA Register Latch.
1: NA<17:0> => RPTEVA<17:0>.

S10 - Shift Right 10 bits.
1: CA<37:10> => NA<27:0>.

STR - STORE Register byte from Processor. PROC_ADR_BUS specifies which register and which byte.
1: CA<7:0> => NA<x+7:x>. -- Shift byte x bits to appropriate position.
NA<x+7:x> => Reg<x+7:x>. -- Load addressed register.

LDR - LOAD Register byte to Processor. PROC_ADR_BUS specifies which register and which byte.
1: Reg<x+7:x> => CA<x+7:x>. -- Enable addressed register.
CA<x+7:x> => NA<7:0>. -- Shift byte x bits Right to LSB position.

INT - INTERRUPT Request Available. Update INTREG with result.

Control Signal Truth Table															
STATE	ADR	GSE	RME	GVE•	PTE	REH	REL	GVL	GVC•	PTL	RPL	S10	STR	LDR	INT
VAref	0	1	0	0	0	0	0	1	0	1	0	1	0	0	0
PTEref	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0
RPTEVAref	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
RPTEPAref	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0
STOREop	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
LOADop	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
INTreq	0	X	X	0	0	0	0	0	0	0	0	X	0	0	1

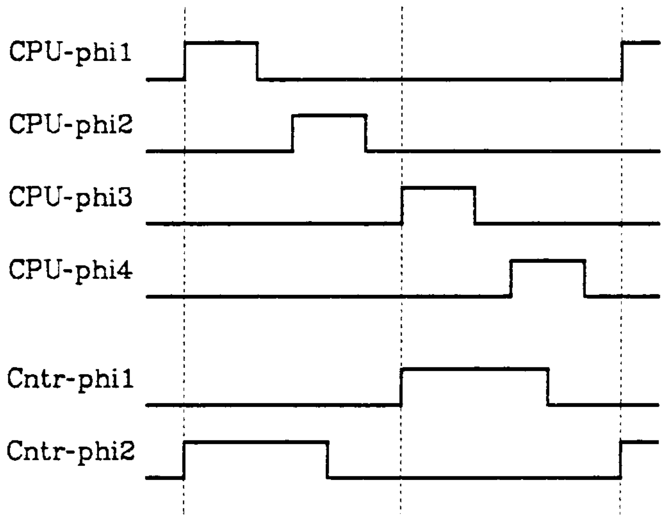
Table 1. Control Signal Truth Table

5. Datapath Timing

One major goal in the design of the cache controller (CC) datapath is to reduce the timing constraints of the circuitry. Since it was not known whether the CPU (processor) four phase clock could be generated accurately on the CC chip and since fewer functions need to be performed during a clock cycle, a static approach was implemented. The registers are all pseudo-static since they are fed back on themselves when not being loaded. The loading or latching of the registers will occur within a specified phase (the 'write' phase) of the clock cycle to make sure that the inputs are stable. The only dynamic circuit in the datapath is the counters.

The counters need to be read by the CPU with a Load Register instruction. The counter data is not always available since a two-phase clocking scheme is used. The counter data can be read any time except during counter phase-2 since it is being updated by the latest increment. To synchronize this restriction with the CPU's need to read the data during CPU phase-3, counter phase-2 will occur during the first half of the CPU clock cycle and counter phase-1 will occur during the second half. This allows the CPU to read the counters while they are being updated. The counter clock phase can be longer than a CPU clock phase as long as it occurs during the correct "half" of the CPU clock cycle.

• These control signals are generated by the Bus Controller FSM which operates off of a separate Bus Clock.



Counter Timing

Appendix

The table below describes the address map for the SPUR CC registers. These registers can only be accessed a byte at a time. Each byte is given a unique address. The address has been broken up into fields which make it easier for the programmer to access the registers.

Physical Address Assignment				
Reg Name	PA<11:9> "type"	PA<8:7> "no."	PA<6:4> "byte"	PA<3:0>
GSN0	000	00	011	XXXX
GSN1	000	01	011	XXXX
GSN2	000	10	011	XXXX
GSN3	000	11	011	XXXX
RPTM0	000	00	000	XXXX
	000	00	001	XXXX
	000	00	010	XXXX
RPTM1	000	01	000	XXXX
	000	01	001	XXXX
	000	01	010	XXXX
RPTM2	000	10	000	XXXX
	000	10	001	XXXX
	000	10	010	XXXX
RPTM3	000	11	000	XXXX
	000	11	001	XXXX
	000	11	010	XXXX
PTEVA	010	00	000	XXXX
	010	00	001	XXXX
	010	00	010	XXXX
	010	00	011	XXXX
	010	00	100	XXXX
RPTEVA	011	00	000	XXXX
	011	00	001	XXXX
	011	00	010	XXXX
	011	00	011	XXXX
	011	00	100	XXXX
INTMASK	100	00	000	XXXX
	100	00	001	XXXX
	100	00	010	XXXX
	100	00	011	XXXX
INTCLR	101	00	000	XXXX
	101	00	001	XXXX
	101	00	010	XXXX
	101	00	011	XXXX
INTREG	110	00	000	XXXX

The Design of a Floating-Point Processor Unit

Glenn Adams, B.K. Bose, Li-Fan Pei, Albert Wang

Department of EECS, U.C. Berkeley

ABSTRACT

This report presents the design of a VLSI Floating Point Unit (FPU) for a LISP-based multi-processor system. The FPU implements the IEEE standard for the essential operations, supporting single, double and extended format operands. Whatever is not done on chip, is handled by supporting software. A four-phase clocking scheme is used, with cycle time of 140ns. Targeted times for ADD, MPY and DIV are 1.0us,1.5us and 3.0us respectively. A dual-ported register file is implemented on chip to allow parallelism between floating point operations and operand loads. The coprocessor interface is designed to minimise overhead in information transfer. The chip is being designed in 2 micron, N-well, double-metal CMOS LOCOS technology. Most of the major modules of the datapath have been designed and laid out so far, and these will be described here. Much work remains, including the completion of the datapath, writing a simulator in SLANG and generating the control PLAs, global routing, and verification of the entire design.

1. Introduction

The Floating Point Unit discussed below aims at meeting the following design objectives :

- fast, single-chip VLSI design
- efficient interface
- simultaneous execution of CPU and FPU
- overlap of FPU LOADs with FPU operations
- execution of the most common functions efficiently; the rest done in software
- follow the IEEE standard (P 754)

1.1. Data Format

Data in Single, Double and Extended formats are accepted by the FPU. The individual representations are shown in Fig.1.1. When operands are loaded into the FPU, the data is unpacked into the internal 80-bit format before writing into the register file. This is to keep the datapath uniform and regular, and not have to treat portions of it specially, depending upon operand type. A few extra bits associated with each operand are also saved; for example, whether all exponent bits are 0 or 1, and information about rounding. Before a STORE instruction, though, an explicit CONVERT instruction is necessary to send the data out in the format desired.

1.2. Timing

A four-phase clocking scheme is adopted, with each phase 25ns in duration, and non-overlap guard time of 10ns. This gives a total cycle time of 140ns. Instructions are latched onto the FPU at the end of phi3 of a cycle, and the appropriate data, if valid, is available from the cache in the same phase, two cycles later. The data is converted to the internal format before being stored in the register file in the same phase of the next cycle.

While the FPU is executing an operation, the CPU may load in new operands in parallel. This is possible with a register cell with two READ and WRITE ports. Register access conflicts are avoided by restrictions on the code generator, and appropriate insertion of NOPs. Once an operation is completed, the FPU sets two flags, one indicating that it is free to accept new instructions and the other indicating whether there was an exception.

1.3. Operations

The functions that the FPU performs, includes: loading data from memory and storing data in memory; floating point add, subtract, multiply, divide; generating absolute value and negation; conversion to desired formats and different kinds of compares.

1.4. Interface

The FPU is designed to have an efficient interface with the rest of the system. The details of the interface are presented in a separate document, and will not be duplicated here. The key features of the FPU interface are :

1) The interface is synchronous with the rest of the system. Instructions are sent to both CPU and FPU at the same time, and decoded independantly.
2) There is a 64-bit data path between memory and FPU. Data transfer is identical for CPU and FPU in terms of timing; the CPU provides the memory address, and either processor proceeds with a data transfer, depending on a control signal.
3) The CPU and FPU can continue execution in parallel. The CPU does not send out valid FPU instructions without making sure that the FPU is free to accept them. Also, at the end of an FPU instruction, the FPU indicates whether the instruction concluded normally or with an exception, and the CPU takes appropriate action.

4) While the FPU is executing an operation, the CPU can continue to load the FPU register file with more operands. This overlap is made possible by using a dual-ported register file, allowing reading or writing of 2 operands simultaneously.

1.5. Special Cases and Exceptions

There are several operands which need to be detected, and treated specially. These are 0, infinity, denormalized numbers, NaNs and Integers. Exceptions that need to be indicated are exponent overflow and underflow, divide by zero, inexact (rounding done), invalid operation and trapping NaNs. Denormalized numbers are detected and flagged, and the software normalizes and sends out the number again. The FPU indicates exceptions by raising the EXC line, and expects the exception handler to take care of it. To make sure that the CPU does not inadvertently ignore the exception when servicing an interrupt or doing a process switch, the FPU-SYNCH instruction is present to make sure that the CPU knows about an FPU exception (even if it does not attend to it just then), before going away to service another call.

2. Functional Description

The exponent, fraction and multiply/divide sub-systems are described below. Specific functions are described, decomposition into smaller modules is shown and the interaction between these modules and between sub-systems is presented.

2.1. Exponent Datapath

2.1.1. Overview

The exponent handling datapath must provide support for all arithmetic instructions. The tasks of this unit may be divided into two groups, with one group representing the tasks for an add/subtract instructions and the other group representing the multiply/divide instructions. The tasks required for add/subtract instruction support are:

1) Determination of the operand with the greater (i.e. result) exponent. Computation of the difference between exponents.

2) Adjustment of result exponent to reflect adder/subtractor normalization.

The multiply/divide instructions require the following support tasks:

1) Computation of the sum/difference of exponents to generate the preliminary result exponent.

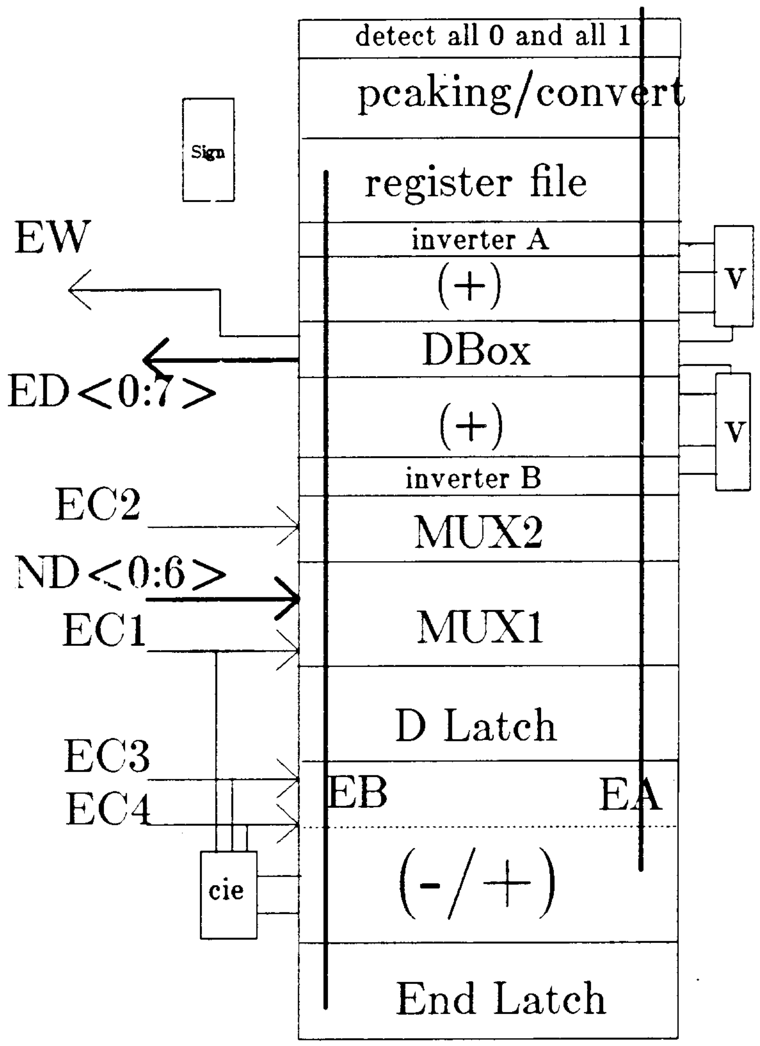
2) Adjustment of the preliminary result exponent to reflect adder/subtractor normalization.

The similarity of these requirements means that much of the hardware that is used is the same for all four instructions. Thus, the exponent datapath is a straightforward sequence of two 16 bit arithmetic components. One component computes the absolute difference of operand exponents and determines which operands has the greater exponent. The other component is an adder/subtractor that computes the final result exponent and in the case of multiply/divide finds the preliminary result exponent as well. The datapath components for the exponent sub-system is shown in Fig.2.1

2.1.2. Exponent Bias

This datapath would be extremely straightforward were it not for the fact that the exponents are biased quantities. Inside the chip, exponents are stored as 16 bit positive numbers with a bias of two to the 15th minus one. A problem arises in that whenever exponents are added or subtracted, their biases are added or subtracted as well. Thus, the result has its bias either doubled or set to zero depending on what operation was performed. Clearly, the exponents cannot be operated on with the bias present. However, simply removing the bias from the operands and restoring it to the result is too difficult if the bias is two to the 15th minus one (it would require extra adds and subtracts).

The solution to this problem is to subtract two to the 15th from each exponent and to interpret all exponents in the datapath as 16 bit two's complement numbers with a bias of -1. Subtracting two to the 15th can be done by simply inverting bit 15 of the operands; inverting the same bit of the result restores the original bias. Since the bias is now only -1, it is now possible to keep the correct bias for intermediate results by manipulating the carry input of the two's complement add or subtract. To see this, recall that the result bias (without adjustment) of an add operation would be $-1 + -1 = -2$, while for a subtract it would be $-1 - -1 = 0$. Thus, to restore the bias the result should be incremented if an addition took place and decremented if a subtraction was performed. Ordinarily the carry input is asserted only on subtraction, thus both cases will be handled correctly if the carry input is inverted. In this way, the correct bias is maintained for all intermediate results in this datapath; there is no need to worry about incorrect bias accumulation.



Exponent Floor Plan

Fig. 2.1

2.1.3. Exponent Difference Section

As mentioned above, the exponent datapath must compute the absolute (positive) difference between the two operands and determine which exponent is greater. This information is sent to the fraction box in three signals, these being:

1) A single line that is asserted if and only if the right operand exponent is greater than the left operand exponent.
2) A seven bit magnitude that is the shift amount for decimal point alignment.

3) A single line that indicates the difference between exponents was greater than 128.

Since these signals are in the critical path of the add/subtract instructions, the logic to generate them must be made as fast as possible. Thus, this section computes both differences (left minus right and right minus left) in parallel. Note that since differences are required, the result bias of the subtraction (zero) is correct; the carry inputs to the subtractors need not be manipulated. Since the signs of the operands may be different, the signal that indicates the greater exponent is a function of the signs of the operands and the signs of the differences. If the signs are different, then the left operand is greater than the right if and only if the right hand operand is negative. If the operand signs are the same, then the greater exponent signal is the sign of the right hand (right minus left) difference. Once this signal is generated, it is used to select the correct (positive) exponent difference to send to the fraction bit. The lower seven bits are sent as is, the upper nine are ORed together to form the difference greater than 128 signal.

2.1.4. Result Exponent Calculation

Although the calculation for the result exponent is different for the two groups of instructions; the same adder/subtractor may be used. For the add/subtract instruction, the result exponent can be computed in one step by adjusting the greater exponent by the normalizing distance. For a multiply/divide instruction, a preliminary result must first be generated by taking the sum/difference of operand exponents. That result is then sent through the same adder/subtractor again to adjust for the normalizing distance. Since the control for this second iteration of the multiply/divide is the same as the control for the add/subtract case, only the control for the first iteration will be described here.

For multiply/divide, the left operand to the adder/subtractor is always the left operand exponent, however for add/subtract the exponent for the greater operand is desired. Thus the left operand multiplexor always selects the left exponent unless the right exponent is greater and an add/subtract instruction is being executed. The right hand operand is the right exponent for multiply/divide, otherwise the normalizing distance from the fraction unit must be used. Now, the add/subtract control signal to the adder/subtractor must be multiplexed as well. This signal is simply the multiply/divide control signal if a multiply or divide instruction is executing, for an add or subtract this signal comes from the fraction unit. Note also that the bias of the normalizing distance is zero, hence the bias needs correction if and only if we are computing the preliminary result of the exponent for the multiply/divide instruction. In this way, the unit computes the correct result exponent for the add/subtract instructions or the correct preliminary result exponent for the instructions multiply and divide. It is a simple matter to later adjust this preliminary result by the final normalizing distance to get the correct result for multiply/divide.

2.1.5. Slang Implementation and Status

At the time of this writing, the slang simulator module for this unit is complete and awaits final testing. Although the details of the four main arithmetic instructions have been completely worked out, the role of the exponent box in the remainder and compare instructions is still not quite clear.

2.2. Fraction Datapath

2.2.1. Overview

The fraction datapath is in essence a 64 bit add/subtract unit. Its use however is not limited to the implementation of the floating point add and subtract instructions. Indeed, every floating point instruction (except for loads and stores) uses some portion of this datapath during its execution. However, the datapath has been designed primarily with floating point add/subtract in mind. Consequently, the datapath contains the following components:

1) Operand decimal point alignment
2) Adder/Subtractor
3) Rounding of Result
4) Normalization and Exponent Adjustment

For all instructions, the input values are interpreted as unsigned magnitudes and the result is also considered to be positive. These components are arranged in straight sequential order, thus it is possible to view the entire unit as a large piece of combinational logic. That is precisely the view that will be taken in this description. A simplified view of the fraction datapath is shown in Fig.2.2.

2.2.2. Operand Alignment

An addition or subtraction operation begins when the 64 bit values are latched into the inputs from the register file. The fraction datapath cannot begin work on these operands immediately; it must wait for the exponent unit. The exponent unit must determine the operand with the greater exponent and the magnitude of the exponent difference. After this is finished, the fraction unit swaps the operands if the operand with the lesser exponent is in the left (A) position. The operand now in the A (greater exponent) position may pass directly to the adder/subtractor. The decimal point of the right operand must be aligned with the left operand before addition can take place, however. This is accomplished using a right shifter whose shift amount is controlled by the exponent difference of the two operands.

Fraction Datapath

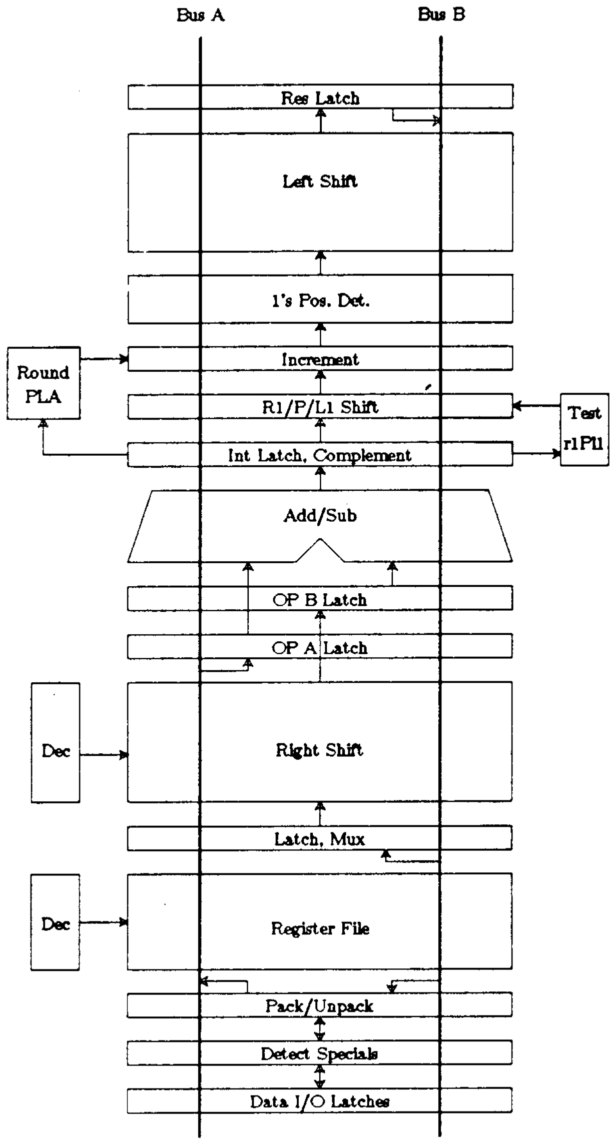


Fig. 2.2

When the operand is shifted, the two least significant bits that were shifted out of the operand become the guard (G) and round (R) rounding bits. The rightmost round bit is generated by ORing the rest of the bits that were shifted out, hence it is called the sticky (S) bit. Because the lost precision of the operand can be condensed, the aligning shifter need only handle a shift amount of up to 66. Shifts of greater than 66 all result in a zero operand, G and R bits as well as a sticky bit that is zero if and only if the entire operand was zero. In this way, each input is made ready for the adder/subtractor.

2.2.3. Adder/Subtractor

Up to now, the operands to the adder/subtractor are assumed to come from the input conditioning section described above. However, the multiplier and divider also use this datapath; for those instructions the operands enter directly to the inputs of this component. In all cases, although each operand is presented to the adder/subtractor in unsigned form, this component is implemented as a two's complement adder. When performing a subtraction at the unit[1] the operands with the lesser exponent is taken to be the subtrahend. Subtraction is therefore performed by complementing the left operand and adding it to the right operand. This way the rounding bits do not have to participate in the subtraction, thus the width of the adder/subtractor remains at 65 bits.[2] As a result, most of the time the result of the subtraction will be a negative number; an addition will always result in a positive number. The only exception to this rule is the case where the operands of a subtraction instruction have the same exponent. In this case the relative magnitudes of the operands are not known.

The output of the adder is in all cases is a 69 bit two's complement value. The value consists of a sign bit, two bits to the left of the decimal point, 63 bits to the right of the decimal and three rounding bits. After the addition/subtraction is performed the result must be returned to positive magnitude form. The first part of this operation is done immediately after the adder/subtractor; the result is inverted if it is negative. This is implemented by exclusive-ORing all of the 68 rightmost bits with the intermediate sign bit. Thus, the final output of this unit is a 68 bit positive number. The number is correct if the result of the adder/subtractor was positive; otherwise the value must be incremented to finish complementing it.

2.2.4. Rounding and Normalization

At this point, the basic operations that remain to be performed are rounding, normalization and incrementing. Also, the exponent of the result must be adjusted in order to reflect any normalization that was done. Normalization here means to bring the uppermost one in the number to the bit position just to the left of the decimal point. This requires that the value be shifted any amount from right by one to left by 66. Since rounding must be done to the normalized number; it would seem that normalization would be the first thing to do in this section. That is however NOT correct because the number may still need to be incremented because it is negative. The incrementing may cause the uppermost one to move left one position; that would require another adjustment to the exponent. Furthermore, the rounding may also require that the number be incremented even if it was originally positive. Therefore, the primary design goals for this section of the datapath were to use one incrementer for both rounding and complementing and to make only one adjustment to the exponent.

[1] This is NOT the same as executing a subtract instruction.

[2] It is made one bit larger to accomadate the multiplier/divider.

The solution that satisfies the above design criteria exploits the fact that different actions must be taken based on the value of the input. It can be shown that only those number that require a shift amount of zero or one (left or right) need to be rounded. (DESIGN NOTE: Although it should be clear that shifting by more than two zeroes the rounding bits, the case shift left by two is not obvious. It can be shown however that numbers in the range $(0.25 < x < 0.5)$ must have resulted from a subtraction where the lesser exponent operand was shifted by no more than one. Hence all one's are shifted out of the rounding bits during the normalization in this case.) These numbers are therefore treated differently from those numbers that require only normalization involving a left shift of greater than one.

First, the input to this section is tested to see if the required shift amount for normalization is right by one, zero, left by one or léft by greater than one. This is done by looking for the uppermost one in the two positions to the left of the decimal and the uppermost position to the right. If the shift amount is greater than one, all of the bits of that number are passed to the incrementer latch; no rounding is done. There all 67 bits (only one bit exists to the left of the decimal now) are incremented if the number is negative. If the shift magnitude is zero or one, the number is normalized immediately by a unit that shifts right by one, passes the number or shifts it left by one. The uppermost 63 bits of this result are sent to the incremeter; the four least significant bits (L, G, R, and S bits) are sent to the rounding pla. This pla returns the new value of the L bit and zeroes the rounding bits. It also returns a signal to increment the upper 63 bits if necessary. Note that this requires that the incrementer be split into two parts consisting of 63 and 4 bits respectively. Also, a multiplexor is required to choose the correct version of the four least significant bits; it is controlled by the signal that indicates a shift left by greater than one.

After the incrementing is complete, the number consists of at most 66 bits and is almost ready for the final priority encode and normalization step. First, however, the possibility of an overflow on the increment must be handled. This is done by placing the carry output of the incrementer back into the uppermost bit of the number and adjusting the exponent accordingly (exponent adjustment is covered later). Now the number is passed to a priority encoder; this being a circuit that finds the amount of left shift required to normalize the value. Here the shift amount can range from zero to 66; a zero shift will always result for those numbers that were normalized already. The encoder generates the shift amount in both a "one-hot" encoding that controls the shifter and a binary encoding that is used to adjust the result exponent. The normalizing shifter generates the final result for the fraction unit.

In addition to generating the final result, this section must also adjust the result exponent to reflect all normalization that was done. Normalization occurs in three places of this section; these being the initial shifter (zero or one), the incrementer overflow or the priority encoder. The first two sources are mutually exclusive of the third; the correct exponent adjustment will come from the priority encoder only in the case where the shift test amount (before the increment) is greater than one. The value from the priority encoder must always be subtracted from the result. In the other case, the shift amount must be generated using separate logic; they are a function of the shift test amounts and the incrementer overflow signal. Two results come from this section; the magnitude of the amount and a signal that indicates whether to add or subtract that value from the exponent.

Last but not least, some discussion of the details of the rounding pla must be made. First the four least significant bits of the operand are collapsed into three by ORing the lowermost two bits together. These three bits then select a rounding output value based upon the two bits of rounding mode (from the control unit), the sign of the final result and the sign of the intermediate result. The sign of the intermediate result modifies the truth table to reflect the results of incrementing those four bits before rounding. The sign of the final result modifies the rounding mode that is applied to produce answer. The truth table for the pla is given in Table 2.1.

2.2.5. Slang Implementation and Status

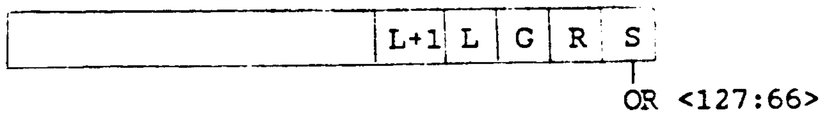
As mentioned above, the entire add/subtract unit may be thought of as one piece of combinational logic. This is the view that was taken when generating the slang module for the device. This view made the implementation of the module very straightforward and did not sacrifice any usefulness of the results. However, one complication did arise in the storing and performing operations on large (greater than 32 bit) quantities. The Franz Lisp implementation of arbitrary precision integers (bignums) does not have a sufficient set of arithmetic and logical functions, nor can such functions be readily written. Furthermore, functions that handle both single and arbitrary precision integers are very difficult to write. Thus the fraction datapath was represented as a list of integers, and an arithmetic/logical functions package was written to operate on these lists. Although most functions were fairly easy to write, a few (such as shifting by an arbitrary amount) could not be implemented cleanly. It should be noted that the problems that forbid the use of arbitrary precision integers stem not from the slang simulator but from the Franz Lisp interpreter itself.

At the time of this writing, all of the functional details with the possible exception of the rounding pla have been resolved. The slang simulator has been written, and the correctness of this functional description will be tested as soon as an "exhaustive" set of test cases can be found.

Table 2.1

ROUNDING:

IEEE style beautiful roundling, using Guard, Round and Sticky bits.



1<n<.5	...	L1 :	G → L′	R → R′	S → S′
1<n<2	...	P :	L → L′	G → R′	R+S → S′
2<n<4	...	R1 :	L+1 – L	L – R′	G+R+S – S

True sign of result				Rounding scheme			
				0	+	–	Nearest (Even)
S	L′	R	S′	L	L	L	L
0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	0
0	0	1	1	0	1	0	1
0	1	0	0	1	1	1	1
0	1	0	1	1	INC	1	1
0	1	1	0	1	INC	1	INC
0	1	1	1	1	INC	1	INC
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	0	1	0
1	0	1	1	0	0	1	1
1	1	0	0	1	1	1	1
1	1	0	1	1	1	INC	1
1	1	1	0	1	1	INC	INC
1	1	1	1	1	1	INC	INC

2.3. Multiply/Divide Datapath

The Multiply/Divide sub-system is shown simplified in Fig.2.3.

Floor-plan of the Multiplier/Divider

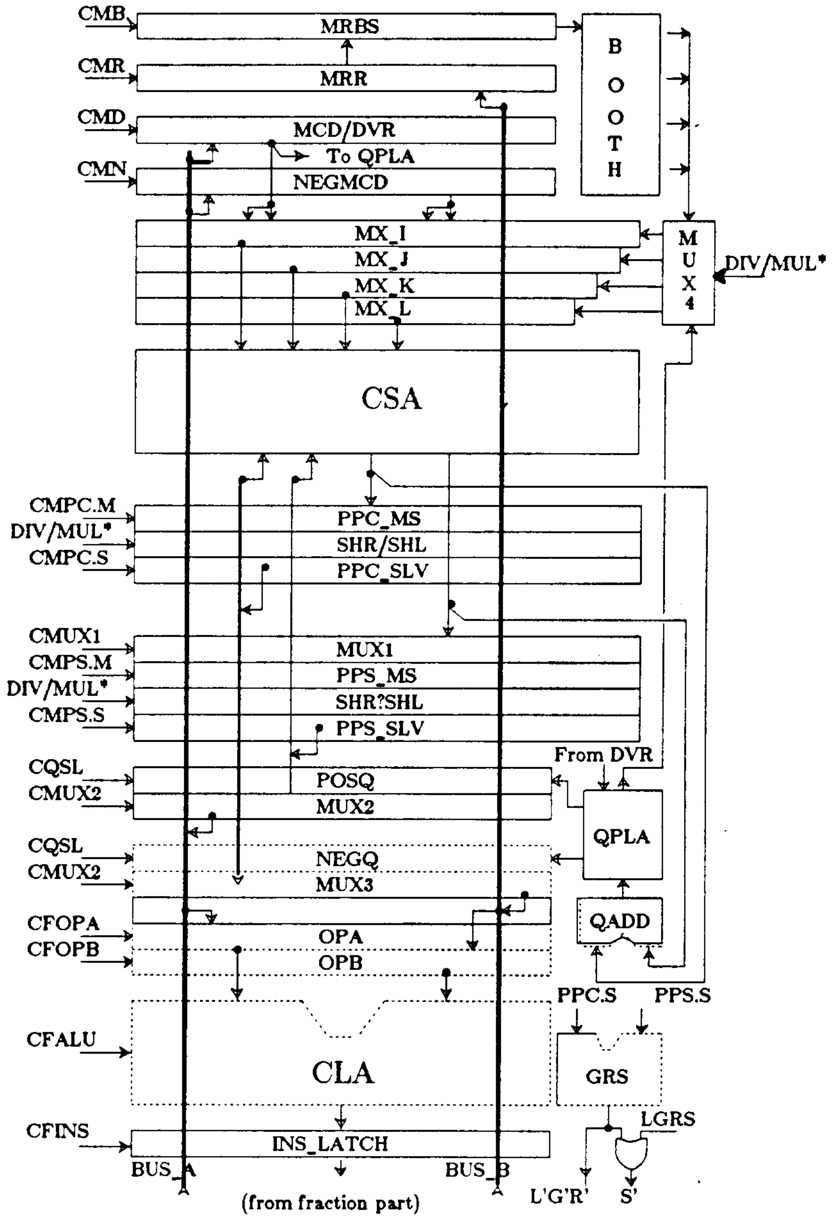


Fig. 2.3

2.3.1. Algorithm

The floating point unit chip uses a version of Booth's algorithm for multiply. For both multiply and divide operations, the algorithm consists of a loop in which bits from one operand (the multiplier or divisor) control the addition of the other operand to the partial results (partial product or remainder). In both cases this addition is performed by a carry-save adder in order to decrease the loop execution time. Also, in each scheme the final results from the unit are generated in a non-binary encoding and must be accumulated to form the true final answer. Because of the similarity between each algorithm, it was possible to share much of the hardware required for each operation. Specifically, within this unit the operand selection multiplexors, carry save adder and and partial result latches are common to both the multiplier and divider. The adder/subtractor in the fraction unit is also used in each operation to perform the final result accumulation. Since chip area is so limited, this hardware sharing was absolutely necessary in order to meet the design goals for the chip.

2.3.2. Multiplier

Although at the highest level the multiply and divide algorithms appear similar, many of the implementation details are completely different. To begin operations, the multiplier latches in the two operands (multiplier and multiplicand) from the register file. The unit, however, also requires the two's complement of the multiplicand in order to function. Thus, the multiplicand is simultaneously latched in at the subtractor input latch of the fraction unit. The adder/subtractor in the fraction box is then used to complement the multiplicand. Note that the complement requires at least two clock phases to generate, furthermore the operand busses are only available on alternate cycles. Thus, it requires an entire processor cycle (four phases) to latch in the operands required for multiplication.

Since the multiplier generates eight bits at every step, the control unit must first select an entire byte of the multiplier for processing. Those eight bits are then recoded into four groups where each group contains four bits in a "one hot" encoding. Each group then controls a multiplexor that selects one of four versions of the multiplicand; the versions being the multiplicand, the multiplicand shifted left by one (multiplied by two), or the complement of the multiplicand shifted or unshifted. A single encoding group can select the correct version of the multiplicand for two bits of result generation. Thus the four groups control four such multiplexors, with each multiplexor shifted to the left by two bits from its predecessor. In this way, the partial results for one step of the loop are generated.

This partial result must then be added to the accumulated results of the previous steps to form the complete result for this step. A tree of four carry-save adder rows is required for this task. Since four two bit partial results are generated at once, the carry-save adders must be 73 bits wide. The entire tree reduces six operands (the four partial results and the partial products from the previous step) to the sum and carry vectors of the current partial product. Although four rows of carry-save adders are required, the order of addition has been rearranged so that the critical path through the tree contains only three adders. This is accomplished by placing the six input operands into two groups of three and adding them in parallel. The four vectors from that level are then accumulated in sequence to generate the two final result vectors.

Once the result vectors have been generated, they are latched into the master of the partial product accumulator latches. A new cycle begins on the next phase when the slave of the accumulator is loaded. The partial product vectors must be shifted to the right by eight bits before they can returned to the carry-save adder for the next cycle. The eight bits shifted out of each result vector are the least significant bits, however they still cannot simply be thrown away. Instead, the two eight bit vectors are added together to form the rounding bits for that step.[3] When this addition is performed in the last step of the multiply, the result is sent to the rounding bits in the fraction box. The mapping of these bits is, however, not straightforward. The leftmost bit of this adder actually belongs in the L (bit 63) position of the fraction unit datapath, not in the rounding bits. Thus, the second from leftmost position of the adder corresponds to the guard bit of the fraction unit. Furthermore, the multiply unit does NOT produce a bit for the round (R) bit of the fraction bit; using bit five from the adder is not correct. Instead, a zero is placed in the round position and the rightmost six bits of the adder are ORed with the previous value of the sticky bit to form the new sticky bit. Finally, the carry output from this adder must enter into the formation of the next partial product. To do this, the carry bit is inserted into the carry save adder at the rightmost position on the lowest level of the tree. Thus, the precision generated at each step of the multiply loop is condensed but not lost.

Up to now, no mention has been made of the timing of the multiply loop. Since the generation of multiplicands and the conditioning of partial product vectors are data independent, these functions are pipelined. Thus, for a given cycle the shifting and rounding of the partial product vectors is done in parallel with multiplicand selection. Each pipelined function is estimated (conservatively) to take two clock phases, and two phases of latency required to set up the multiplicand selection of the first cycle. Note that no extra hardware is required to form this pipeline; this is because the control unit byte selector (hence the multiplicand selection) is synchronized with the partial product vector latch. However, all of the hardware that is in the loop must be implemented using fully restoring logic (fully static CMOS). A total of 22 clock phases is required therefore for the multiply/divide unit to perform its intended task.

This does not imply, however, that the multiply instruction requires only 22 clock phases. The partial product vectors and the rounding bits generated by the last multiply cycle must be added together in the fraction unit in order to generate the final answer. Now, the partial product vectors are each 65 bits wide; this (NOT the add/subtract instruction!) sets the upper bound on the width of the fraction unit adder/subtractor. However, the final result still is the range ($1 < x < 4$), thus no more logic is required in the fraction unit to support the multiply instruction.

[3] Actually, only five of those bits are significant. Five bit adder cells are however no faster than eight bit cells.

2.3.3. Divider

As mentioned above, the divider uses the same accumulator/selector loop as the multiplier. However the similarity between the units ends there, for while the multiplier loop generates eight bits of result per step, the divider can only produce two. Furthermore, the divide algorithm generates increasing precision with each step rather than increasing significance. Also, accumulator/selector loop in the divider loop generates the partial remainder, not the partial quotient. The algorithm therefore begins by loading the partial remainder accumulators (previously called the partial product accumulators) with the dividend and a zero respectively. The other operand (divisor) plays the same role as the multiplicand did in the multiplier. Thus, the divisor and its complement are generated and loaded using the same process described for the multiplicand loading.

The actual divide cycle begins with the topmost eight bits of the partial remainder vectors being added together. The topmost six bits of this result constitute an estimate of the entire partial remainder at that step. Doing eight bits of addition rather than just six significantly increases the precision of this estimate and does not increase execution time. The remainder estimator is then fed along with the four most significant bits of the divisor into a pla called the quotient box. The quotient box pla produces the two bits of quotient for this step. Although it only generates a two bits of quotient the pla still has a large[4] number of minterms; the complexity of this pla dictates the upper bound of result bits per cycle.

Since the data fed into the pla is only an estimate of the partial remainder, the data output is only an estimate of the partial quotient. Thus, although it really only contains two bits of significance, the quotient pla output is encoded as a signed magnitude integer in the range from negative two to positive two. The result is signed because the quotient estimate from the previous step may have been too high; in this case the partial remainder and the next quotient bits are both negative. The pla output is also decoded before being used in the divisor selection unit. The rest of the loop proceeds in the same manner as the multiplier, with the exceptions being that the partial remainder vectors are shifted left by two bits and that no rounding of the partial remainder takes place. Also, since only two bits are generated on each cycle, only the topmost of the four selection multiplexors and CSA are actually used.

The quotient pla output bits are then assigned to one of two latches based on the sign of the output. Thus, the quotient is encoded in a vector of positive estimates and a vector of negative estimates. These latches are actually shift registers; the new data is always placed at the two rightmost position and the entire quantity shifted to the left by two bits. Each vector is encoded as a 68 bit quantity (65 + 3 rounding bits), however the uppermost bit of the result of the first quotient cycle is always zero.[5] To form the final result, the negative estimate vector must be subtracted from the positive estimate vector. This subtraction is performed by sending each value to the fraction unit. However, one cannot send all 68 bits of both quantities to the fraction unit; the fraction unit expects rounding bits from

[4] Exact number should be inserted here!!

[5] Since a simple loop is used, total time for divide unit is 68 clock phases.

one operand only. Rather than enlarge the adder for this special case, the three rounding bits are subtracted before being sent to the fraction unit. Also, the final result must be decremented if the sign of the partial remainder is negative. This is accomplished by implementing the rounding bit subtractor as a two's complement adder and using the complement of the partial remainder sign bit as the carry input. Thus, to compute the final result the fraction unit is sent two 65 bit operands, the three rounding bits and the carry output of the three bit subtractor. The carry output of the three bit subtractor cascades into the carry input of the fraction unit adder/subtractor.

2.3.4. Status

At the time of this writing, almost all of the details of the function of this unit have been resolved. It has not been determined, however, exactly how the partial remainder vectors are to be sent to the fraction unit to generate the result for a remainder instruction. Also, the slang simulation for this unit is not yet complete.

3. Logic & Circuit Design of Major Modules

Logical design and circuit implementation notes of some major blocks of the data-path are presented. Wherever appropriate, area and expected performance are also mentioned. Circuit design notes for the Xerox CMOS process are summarised in Appendix A.

3.1. A Fast Adder

In the floating point chip, various adders are needed in different parts of the data path. Specifically, a 64-bit adder is needed in the data path of the fraction part. Exponent data path needs a general alu which is based on the 16-bit adder. Two additional subtractors, which are also based on the adder, are needed in order to determine the exponent difference. And, finally, an 8-bit adder is required in the multiply/divide unit.

As one can see, the usage of the adders is different and therefore the requirement is also different. Some may require high speed, as in the subtractor, and the size of it is not as important. In other places, the size may become the first consideration. Because of the limited amount of time and man power, we decided to design one adder for all the needs in the FPU with the best compromise between the speed and area. As a result, we chose to use a carry-lookahead, Brent-Kung scheme because of its high speed, relatively small area, and fairly regular layout. The reason we chose static instead of dynamic logic is the timing requirement to determine the exponent difference. We need to get the difference as fast as possible, hopefully within one phase, and it is not possible to 'hide' the precharge time, needed for a dynamic scheme, anywhere.

3.1.1. The Implementation of Brent-Kung Adder

This is a kind of carry-lookahead adder. The tasks are distributed at different stages. Before describing each stage in detail, some general schemes needs to be mentioned which are used throughout the adder.

CMOS static logic is used for the design. According to the study done by Shong, this scheme has higher speed when the fan-in of the circuit is small. In the adder, the fan-in is usually only three.

Some optimizations are done on each gate. The optimization involves finding a path in the pull down net work which results in a minimum number of seperation zones and maximum sharing of source and drain regions. The second thing is to find the corresponding path in the pull up net work which is the dual of the pull down network.

In the Brent-Kung adder, the tasks are distributed into different stages. If static CMOS logic is used, an inverter has to be added to each stage, resulting in twice as many gate delays. For this reason, we invert the logic every other stages, using De Morgan's theorem. The B-K adder is a particularly nice application of this scheme because it has clearly identified small blocks and every one of them can be partitioned into a stage, with no conflict. This way, we reduce the number of gate delays to a minimum.

As pointed out earlier, the B-K adder has sevaral stages each of which perform particular functions. We will now describe each one of them in detail.

3.1.1.1. The p and g generation stage

The first thing is to generate p's and g's, the carry propagate and generate signals. This is fairly straightforward. We implement the following equations:

p = a XOR b
g = a AND b

For carry lookahead, we only have to compute p by OR of a and b. The XOR is used because in the final stage of the adder, we won't have to seperately compute the a XOR b when evaluating the sum. Figure 1.1 shows the layout of this stage - pgg.

3.1.1.2. The Carry Lookahead Stage

The second stage is to evaluate the P and G terms. The whole scheme is based on the B-K operation, which is:

$$(g1,p1) \circ (g2,p2) = (g1+p1g2,p1p2)$$

The purpose is to allow parallel computation of the P's and G's. Figure 3.1a shows the circuit block for this operator, and figure 3.1b is the actual implementation if we use this without modifications.

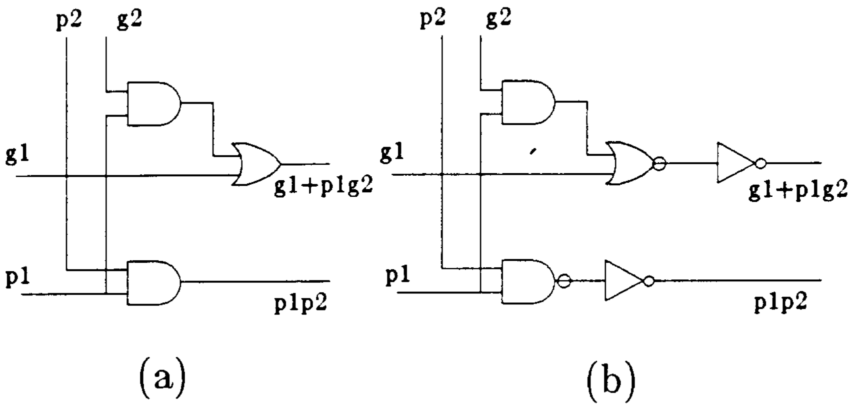


Fig.3.1 PGG : p and g gernerator

To reduce the number of stages, as described before, we invert the logic every alternate stage. This results in two new blocks representing the modified B-K operators. They are shown in figure 3.1a and 3.1b. Fig. 3.1a is the complement of the original operator and 3.1b is the same circuit with inverted logic.

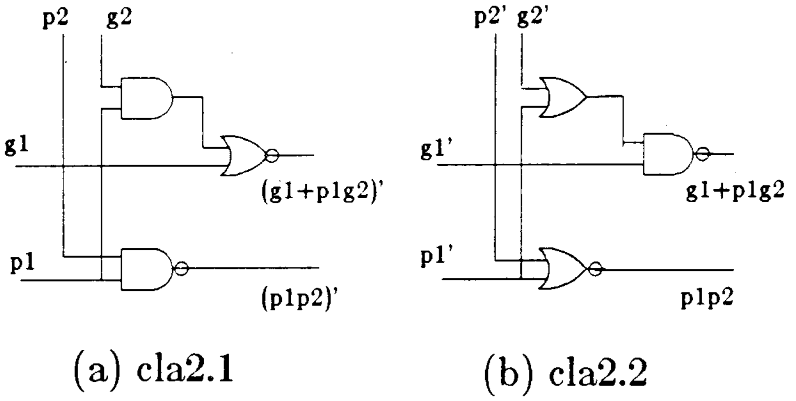
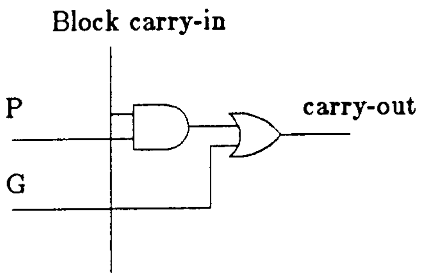


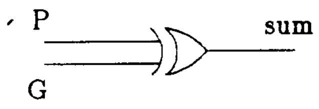
Fig.3.2 CLA2 : carry lookahead, true & inverted logic

3.1.1.3. The Carry and Sum Evaluation Stage

After getting the P's and G's, the next stage is to evaluate the carry in for each bit. The final stage is to compute the sum. Both these blocks are simple and figures 3.2 and 3.3 show them. Putting all the stages together, we get the adder. Figure 3.4 is the logic diagram of an 8-bit B-K adder. Figure 3.5 is the floor plan of a 16-bit adder using the modified B-K operator and inverted logic in every other stage.

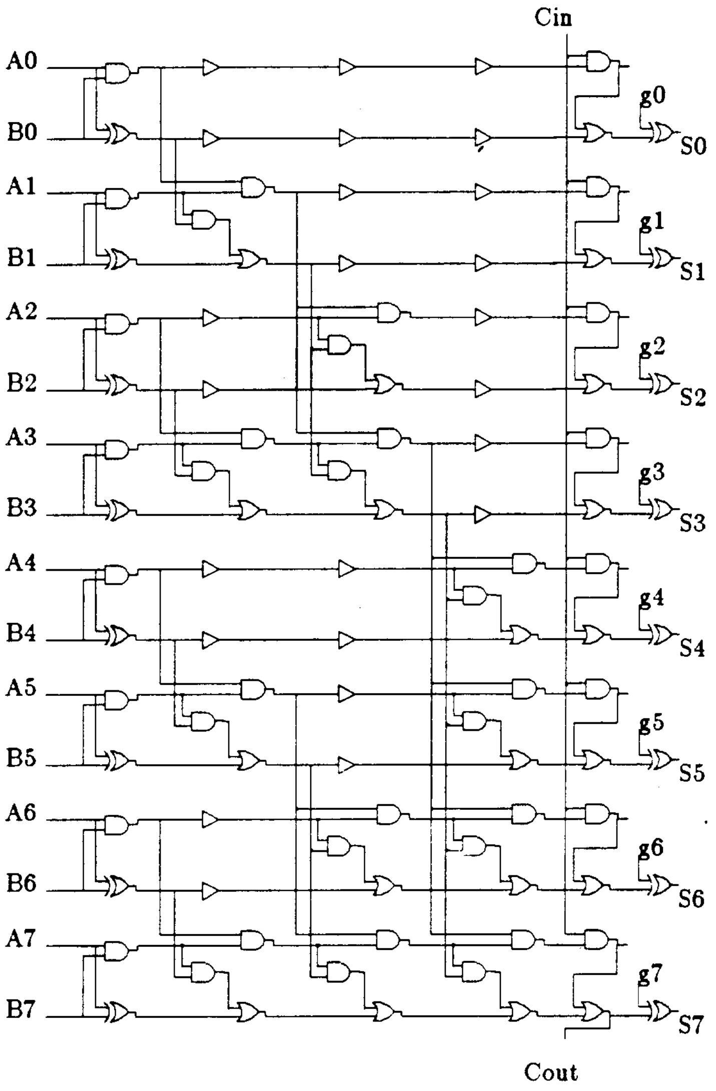


(a) carry evaluation



(b) sum evaluation

Fig. 3.3 carry evaluate & sum



8-bit carry lookahead adder

Fig. 3.4

pgg	cla1	cla1	cla1	cla1	eval2	sum
pgg	cla2.1	cla1	cla1	cla1	eval2	sum
pgg	cla1	cla2.2	cla1	cla1	eval2	sum
pgg	cla2.1	cla2.2	cla1	cla1	eval2	sum
pgg	cla1	cla1	cla2.1	cla1	eval2	sum
pgg	cla2.1	cla1	cla2.1	cla1	eval2	sum
pgg	cla1	cla2.2	cla2.1	cla1	eval2	sum
pgg	cla2.1	cla2.2	cla2.1	cla1	eval2	sum
pgg	cla1	cla1	cla1	cla2.2	eval2	sum
pgg	cla2.1	cla1	cla1	cla2.2	eval2	sum
pgg	cla1	cla2.2	cla1	cla2.2	eval2	sum
pgg	cla2.1	cla2.2	cla1	cla2.2	eval2	sum
pgg	cla1	cla1	cla2.1	cla2.2	eval2	sum
pgg	cla2.1	cla1	cla2.1	cla2.2	eval2	sum
pgg	cla1	cla2.2	cla2.1	cla2.2	eval2	sum
pgg	cla2.1	cla2.2	cla2.1	cla2.2	eval3	sum

16-bit CarryLookahead Adder
(Floor Plan)

Fig. 3.5

3.2. Data flow in the Exponent Unit

This part does all the exponent operations for the floating point unit. The main data path is 16-bits wide (15-bit exponent plus one bit for denormalized numbers) with some simple blocks placed on both sides to evaluate carry-in's, detect overflows, and hold the carry-out for control purposes. The following is a list of functions supported in the current implementation.

(1) To find the difference of two exponents by doing subtractions, EA-EB and EB-EA, and detect certain conditions of the subtraction results. First of all, it will detect whether overflow or underflow has occurred. Next is to check

whether the result is greater than 64 (the fraction width). If none of the above conditions occurred, the 7-bit difference is sent to the fraction part in the form of an absolute value.

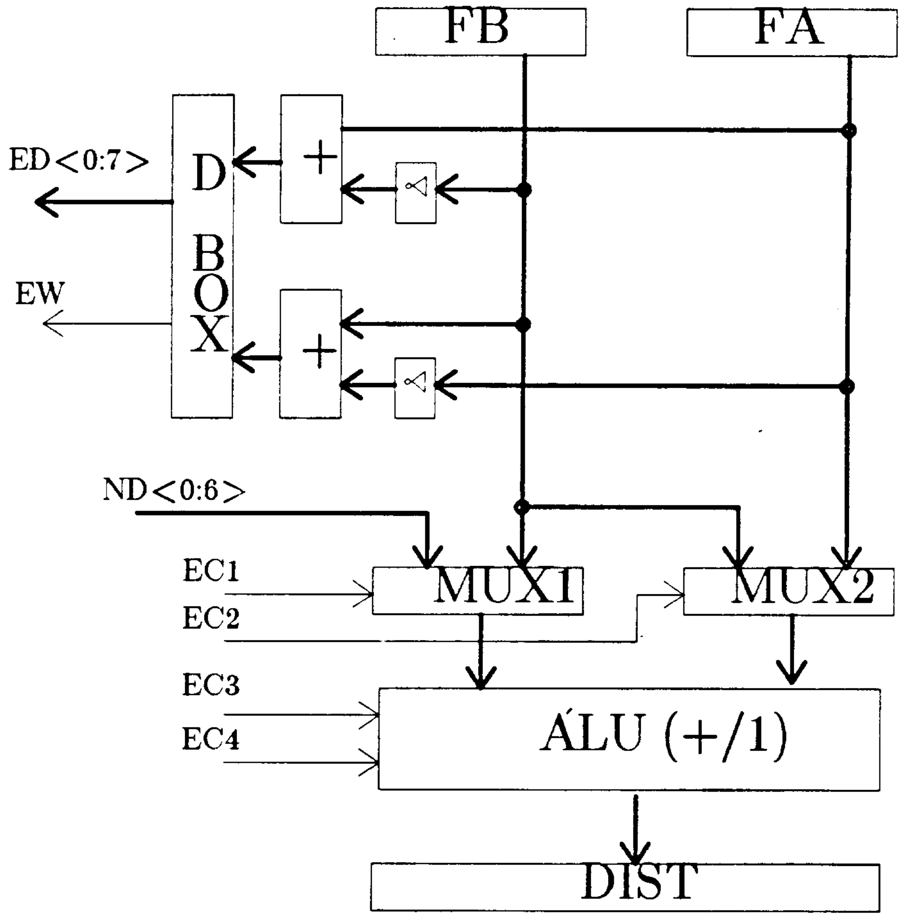
(2) Add two exponents. This is intented for the multiply instruction.

(3) Subtract one exponent from the other. This is for the divide instruction.

(3) Subtract/Add the normalizing distance to the larger exponent. This is for floating point add and subtract.

There are two seperate functional block of the exponent part. The first one is to determine the exponent difference. The second is the general add/subtract unit with some mux's and latches.

Fig. 3.6 is the block diagram of the exponent part, indicating direction of the data movement. Inputs, outputs, and control signals, and how to use them to do the different functions, are presented in figure 3.7.



Exponent Block Diagram

Fig. 3.6

3.2.1. Internal Representation of Exponent

The internal format for exponent is in 2's complement minus one form. The advantage of using this is that converting from external format to internal format is straightforward, allowing fast and easy conversion, and at the same time retaining the advantages of 2's complement arithmetic. Because of this choice, all the operations have to be considered and the carry-in of the adder has to be set to the correct value in order to produce the proper results. All exponents are represented in the 2's complement minus one form and, on the other hand, all the data transfer between the exponent part and the fraction part is in 2's complement format. This fact complicates the the logic. Table 3.1 lists all the functions performed by the exponent part, corresponding carry-in values, and the format of the final result.

3.2.2. Exponent Difference Calculation

Determining the exponent difference is the first step in floating point add and subtract. Therefore, the speed of this operation is very important to the speed of the whole add/subtract instruction. We sacrifice some chip area in order to achieve high speed. Subtracting hardware is duplicated so the computation of EA-EB and EB-EA can be done concurrently. The positive difference will be picked and passed to the Dbox which will pass the low order 7 bits and the OR of the higher order 8 bits to the fraction part. Figure 3.8 is the logic for Dbox and Fig. 3.9 shows the floor plan of the difference part.

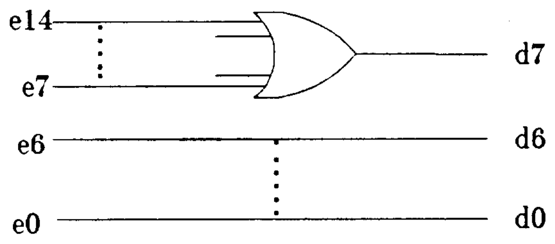
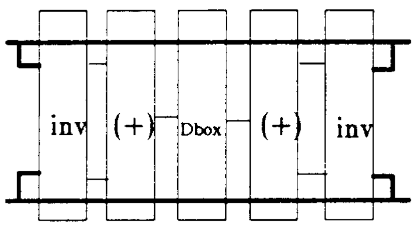


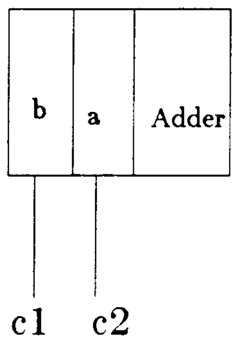
Fig. 3.8 Difference Box

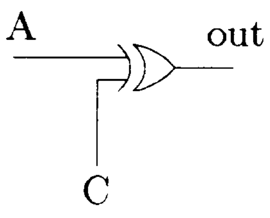


Floorplan for Difference Computation
Fig. 3.9

3.2.3. General ALU

This ALU need only do the addition and subtraction. Using the ADD to do subtraction involves inverting every bit of one of the operands and setting the carry-in bit to 1. If we have a control line that controls whether to invert an operand or not, we get the general ALU. Figures 3.10 and 3.11 show how to build the general ALU using the adder and controlled inverters.





ALU

Fig. 3.10

C1	C2	func
0	0	A+B
0	1	A-B
1	0	B-A
1	1	-A-B

ALU Function Control

Fig. 3.11

3.2.4. Exponent Timing

Instead of giving the exact timing and phase assignment, some rules are presented. These, in combination with the control logic, will determine the exact timing of the exponent part. The rules are:

(1) Precharge the bus one phase before putting the data on it.

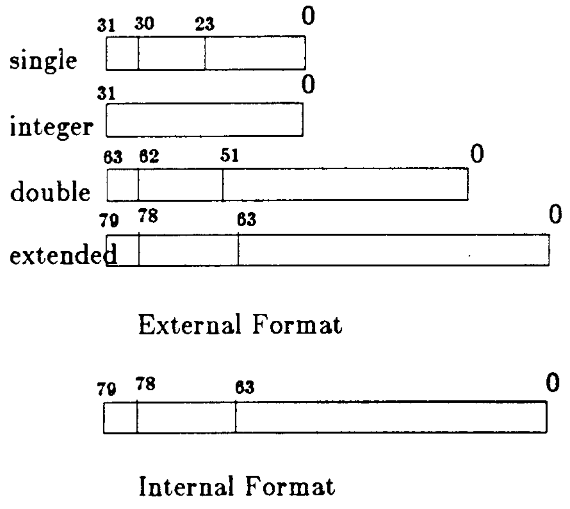
(2) Latch the inputs to the ALU prior to ADD block operation.

(3) The results of the ALU have to be latched before any changes to the input occur. This is because the adder is fully static.

(4) In the difference part, the selection of a positive number is done by the AND of a control bit and a sign bit. There should be some time between when the input is stablized and the control is enabled. This time is the delay of the ADD block and an invert gate delay. Otherwise, the Vdd and GND may be connected through two pass gates in the Dbox.

3.3. Packing and Unpacking Data

This part deals with the conversion between the external and internal representations of data. It will also detect exception conditions. Because there are only four types of data, there is no need to use a full shifter. In stead, each circuit is designed particularly to convert a specific data format. Figure 3.12 shows the difference in terms of number of bits. Figure 3.13 shows the different representations.



Internal & External Data Format

Fig. 3.12

	Fraction	Exponent
single	sign-magnitude	excess 2^7 -1
double	sign-magnitude	excess 2^{10} -1
extended	sign-magnitude	excess 2^{14} -1
integer	2's complement	NA
internal	sign-magnitude	2's complement - 1

Fig. 3.13 Data Representation

In the following description:

eed<n:m> - represents exponent external data bits n to m
eid<n:m> - represents exponent data bits n to m
fed<n:m> - represents fraction external data bits n to m
fid<n:m> - represents fraction data bits n to m

Conversions from external format to internal format:

1) Single

eid<0:6> = eed<0:6>
eid<7:14> = eed<7>'
fid<40:63> = fed<0:23>

fid<0:39> = 0

2) Double

```
eid<0:9>   = eed<0:9>
eid<10:14> = eed<10>'
fid<12:63> = fed<0:51>
fid<0:11>  = 0
```

3) Extended (ed<0:14> --> id<0:14>)

```
eid<0:14>  = eed<0:14>
fid<0:63>  = fed<0:63>
```

4) Integer

```
eid<0:14>  = 32
fid<31:63> = fed<0:31>
```

Conversions from external format to internal format:

1) Single

```
eed<0:6>   = eid<0:6>
eed<7:14>  = eid<14>'
fed<40:63> = fid<0:23>
if eid<7:14> is not same,  exception
```

2) Double

```
eed<0:9>   = eid<0:9>
eed<10:14> = eid<14>'
fed<12:63> = fid<0:51>
if eid<10:14> not same sign,  exception.
```

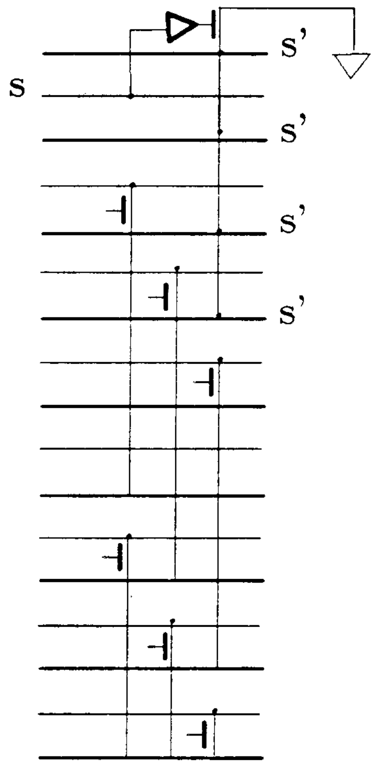
3) Extended (ed<0:14> --> id<0:14>)

```
eed<0:14>  = eid<0:14>
fed<0:63>  = fed<0:63>
```

4) Integer

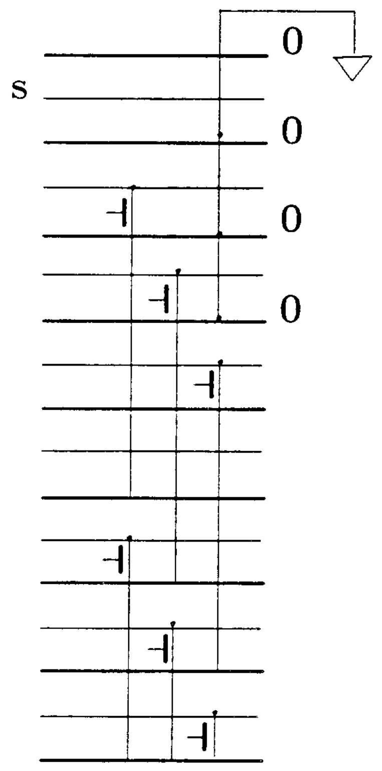
```
fed<31:63> = fid<0:31>
```

The next two figures show how the conversion is done. Figure 3.14 is for the exponent part and figure 3.15 is for the fraction part. Note the gates at the end of the block.



exponent unpacking

Fig. 3.14



fraction unpacking

Fig. 3.15

3.4. The Register File

The FPU has 16 externally addressable, 80-bit registers. One is reserved for control and status, and the rest are for storing operands and results. As mentioned earlier, an incoming operand is converted to an internal format consisting of 1 sign bit, 15 exponent bits and 64 fraction bits. A few extra bits are also associated with each internal representation. These include extra exponent bits for denormalized numbers, flags to indicate if all exponent bits are 0s or 1s, and also some bits to indicate rounding information. The efective register width thus increases to nearly 90 bits.

A requirement for the register file is that it be able to read two operands simultaneously. Also, to allow for overlapped FPU operations and LOADs, it is necessary that two operands be written at the same time. Thus we need a dual-ported register file, with two sets of decoders for the two operands.

Fig. 3.16 shows the circuit for the register cell. It contains 9 active devices. Four are transmission gates providing read and write access to the two data busses A and B. The rest of the cell is a pseudo-static latch. During READ, the latch feedback path is closed, while for WRITE it is open. Thus there is no fight between input data and what is in the cell.

The data busses A and B are precharged to increase speed and reduce area and power of bus drivers. Since the high level at X is degraded by the access transmission gates, the ratio of inverter I is skewed, with a stronger pull-down device than the normal ratio. Inverter II is larger than I to provide greater drive to allow for fast discharge of the data busses.

Register Cell

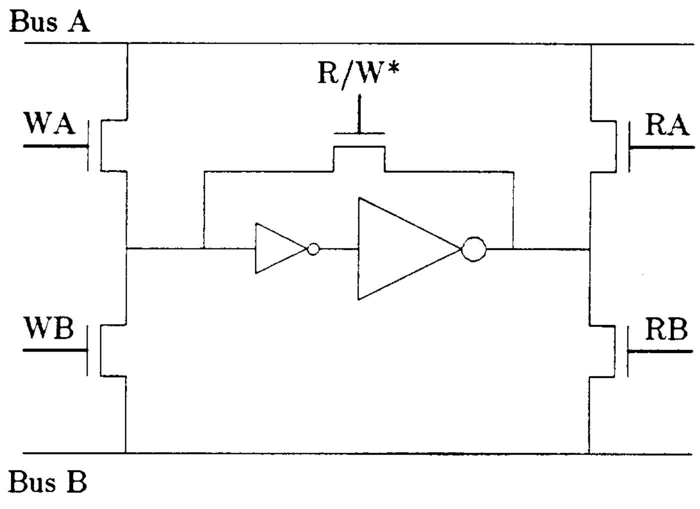


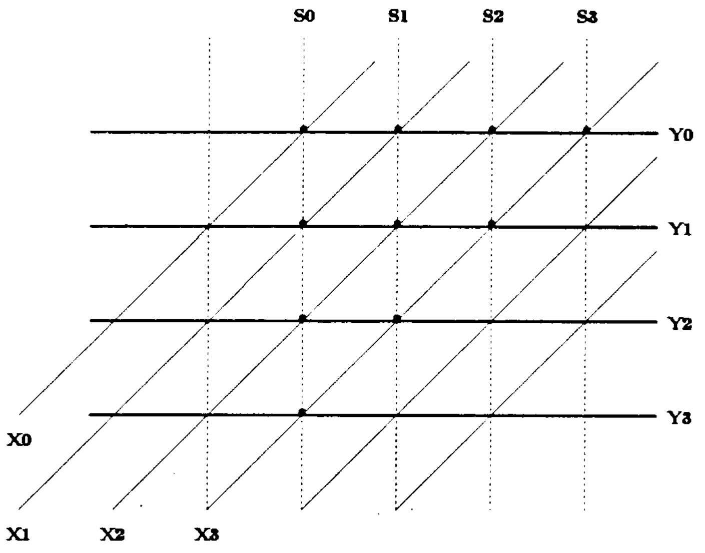
Fig. 3.16

3.5. Bi-directional Barrel Shifter and Decoder

The fraction datapath requires two variable-length shifters. A right shifter is required almost at the beginning of an ADD/SUB operation to align the fraction with the smaller exponent to the fraction with the greater exponent. Again, at the very end of the operation, a left shift is necessary to normalize the fraction result. Both shifters need to be able to shift the fraction by a number ranging from 0 to 66.

Barrel shifters with large shift length are expensive in silicon, and given the restrictions on chip dimensions, it is clear that we cannot afford to have two such wide shifters. We have built a single, bi-directional shifter, where the input and output busses are reversed for right and left shift.

Fig. 3.17 shows the structure of the shifter. both symbolically and schematically. Busses X and Y are the two data busses, and Bus S is the decoded shift distance. Both data busses are precharged to avoid having a full complimentary matrix of transistors. A more common configuration for this kind of shifter has the input and output busses running perpendicular to each other, and the shift control lines diagonally traversing the shifter matrix. With our configuration, the shift control lines are shorter, thus having less parasitic capacitance, and hence faster for the same size transistor matrix.



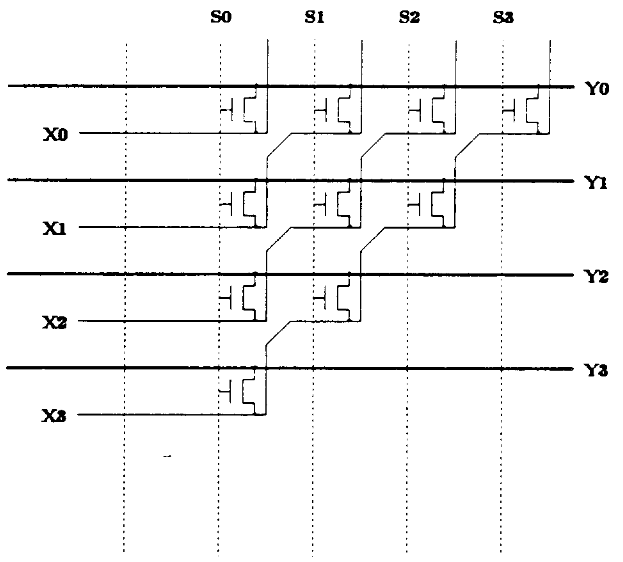


Fig. 3.17

Fig.3.18 shows the schematic for decoding the shift distance. It is just like a precharged NOR decoder in effect, but in practice uses a single level of pre-decode with a pseudo-NOR stage following it. The 'NOR' gate transistors are connected between two adjacent rows instead of between each row and ground. This reduces the total number of decoder transistors by about half, allowing the devices to be about twice as large, thus cutting down the decode time significantly.

3.6. One's Position Detector

After an ADD/SUB instruction, the intermediate result has to be normalized. To do this, we need to detect the position of the leading '1' in the intermediate result. This information needs to be transferred to the exponent unit so the result exponent can be appropriately modified. Also, this information has to be decoded and used to control the left shift distance.

The three most significant bits of the output indicate which byte of the 8 bytes in the intermediate result has the leading '1'. The remaining three bits of the output indicate which bit in the above byte has the leading '1'. The implementation of this combinational logic piece is in dynamic CMOS, using alternating N channel and P channel sections. Since there are large fan-in AND and OR gates, we chose alternating N and P sections to avoid long series strings of devices, as would happen in a standard 'domino' implementation. The worst case delay is 7 gate delays, and is estimated at about 20ns.

3.7. The Multiplier

The multiplication of two 64-bit fractions is implemented in 8 iterative steps. Each iteration implements a 64 bit(MCD) * 8 bit (MR) multiplication. In each iteration, four overlapped triplets of multiplier bits (9 bits) are decoded by the modified 'BOOTH' decoder. Four multiplicand multiples of magnitude +2MCD, +1MCD, -1MCD, -2MCD and 0 are needed per iteration. The multiplicand is always a positive fraction. The -1MCD (2's complement of MCD) is generated by the ALU in the fraction datapath, before the beginning of the iterations and is latched in the latch NMCD. The +2MCD and -2MCD are implemented by wired shift left by 1 bit of the +1MCD and -1MCD respectively.

The four overlapped triplets of multiplier pairs generate four multiples of the multiplicand. They are added to the partial 'sum' and partial 'carry' terms of the previous iteration using the carry save adder array (CSA). Note that the four multiples of the MCD are shifted left 2 bits with respect to each other, depending on the significance of each multiplier triplet. The partial 'sum' and 'carry' are shifted left 8 bits and 7 bits respectively, when looping them back to be the new inputs of the CSA in the next iteration. Since there are negative as well as positive operands, both in 2's complement form, the MX and CSA array must be fully sign-extended to the left.

The bit width of the CSA, MX and PPS/PPC are 73 bits (see Fig.3.21). After right shifting 8 bits, the sum of the 2 partial products will be 65 bits (64 bits magnitude and 1 sign bit). If there are only 72 bits of the PPS, the sign bit will be lost and the sigh extension will be incorrect. The sticky bit is set if there is at least one non-zero bit in the right-shifted 64 least significant bits of the product. The guard and round bits are the 2 most significant bits of the lower half of the 128 bit product. The G,R and S bits are generated by an 8-bit adder and an OR gate (for S), used at every iteration of the multiply loop. (Fig.3.22)

The CSA tree is configured in a rather irregular way in order to let the data pass through only three stages of the CSA, instead of 4. Thus two rows of the CSA

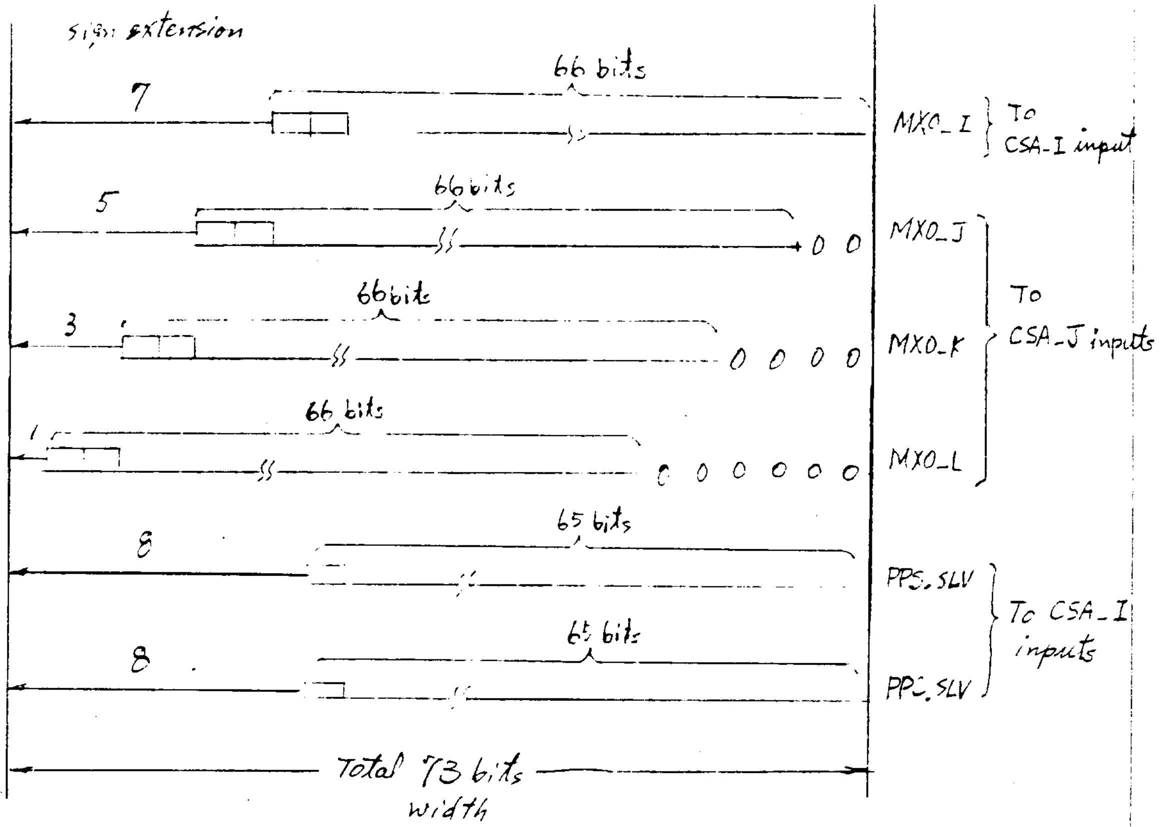


Fig. 3.21 CSA inputs

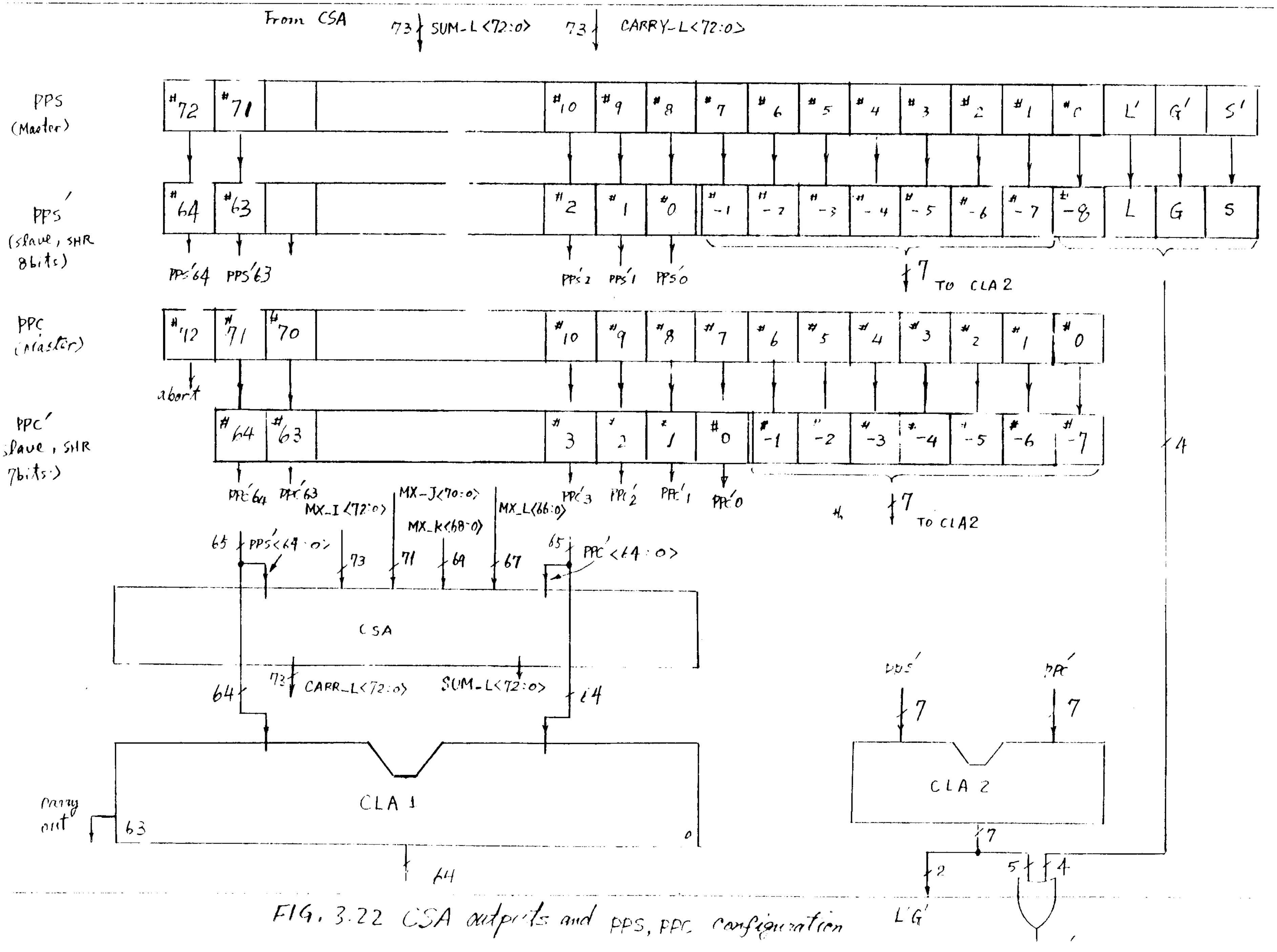


FIG. 3.22 CSA outputs and PPS, PPC configuration

W

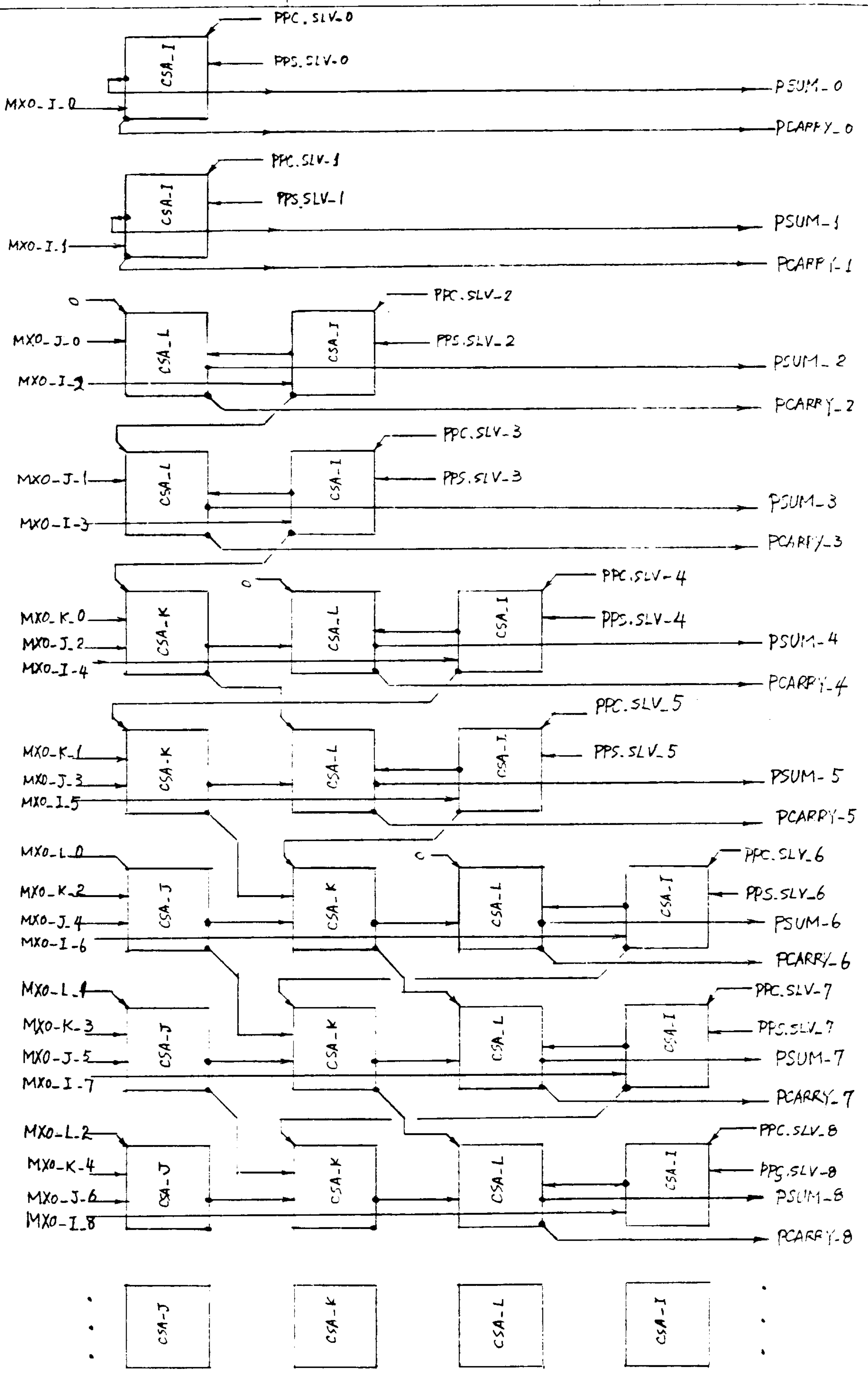


FIG. 3.23 CSA interconnect (Rightest 9 bits)

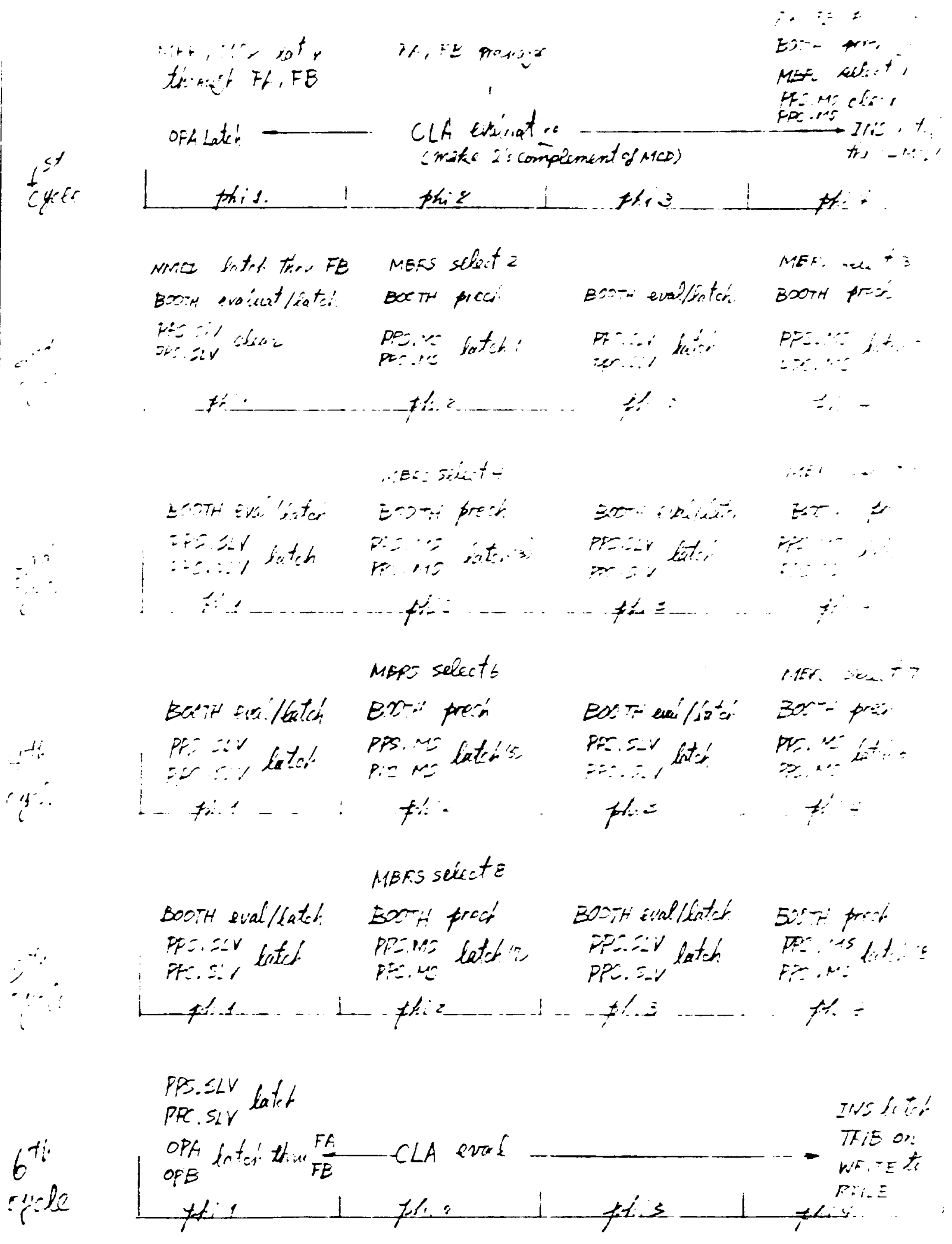


FIG. 3.24 [illegible] of the Multiplication

TABLE 3.2 Control signals of the Multiplier

No	Symbols of Control signals	Function	Timing* cycle #1	cycle #2	cycle #3	cycle #4	cycle #5	cycle #6
1	CMR	Latch multiplier	ϕ_1					
2	CMD	Latch multiplicand (positive)	ϕ_1					
3	CMND	Latch negative multiplicand		ϕ_1				
4	CMBS<1>	Select 1st byte of the multiplier	ϕ_4					
5	CMBS<2>	" 2nd " "		ϕ_2				
6	CMBS<3>	" 3rd " "		ϕ_4				
7	CMBS<4>	" 4th " "			ϕ_2			
8	CMBS<5>	" 5th " "			ϕ_4			
9	CMBS<6>	" 6th " "				ϕ_2		
10	CMBS<7>	" 7th " "				ϕ_4		
11	CMBS<8>	" 8th " "					ϕ_2	
12	CMPC. MS	Latch PPC (Master)		ϕ_2 ϕ_4	ϕ_2 ϕ_4	ϕ_2 ϕ_4	ϕ_2 ϕ_4	
13	CMPC. SLV	Latch PPC (slave)		ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1
14	CMPS. MS	Latch PPS (Master)		ϕ_2 ϕ_4	ϕ_2 ϕ_4	ϕ_2 ϕ_4	ϕ_2 ϕ_4	
15	CMPS. SLV	Latch PPS (Slave)		ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1
16	CMQSL	Quotient latch and shift left 2 bits	NOP	NOP	NOP	NOP	NOP	NOP
17	CMPO	Output the 'Sum' and 'Carry' of the Product to CLA						ϕ_1
18	CMQO	Output the " " " Quotient to CLA						
19	MUL/DIV	Multiplication (Low) or Division (High)	←		LOW			→
20	CLRP	Clear PPS and PPC register (Master)	ϕ_4					
21	COPA	Latch operand A of the CLA	ϕ_1					ϕ_1
22	COPB	Latch operand B of the CLA						ϕ_1
23	COMPA	CLA make 2's complement of operand A	ϕ_1 to ϕ_4					
24	CINS	Latch the instant result of CLA	ϕ_4					ϕ_4
25	CWRB	Write the instant result to B_BUS						ϕ_4
26	CRFWRB	Register file write-in from B_BUS.						ϕ_4
27	CBRC	Latch BOOTH recode		ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1 ϕ_3	ϕ_1 ϕ_3	
28	CADD	CLA make addition operation						ϕ_1 thru ϕ_4
29	CMDV/$\overline{PPS}$	Mux dividend or PPS.MS	←		LOW			→

* Note: ϕ_i means the control is high during ϕ_i.

tree compute in parallel. See Fig. 3.23.

3.7.1. Interfacing with the Fraction Datapath

The Multiply/Divide subsystem shares some hardware with the fraction part. These are :

* 64-bit carry-look-ahead adder
* latches for operands A and B and 2's complement of MCD
* tri-state driver for writing -1MCD onto Bus B
* the A and B data busses and the register file

The reading of the multiplier and positive and negative multiplicand are as follows :

1) The multiplier and multiplicand are sent out on busses A and B from the register file to latches MRR and MCD simultaneously, during the first cycle of the execution. At the same time, the multiplicand is latched in the OP A latch and converted to 2's complement by the 64-bit adder.
2) After the completion of the eight 64 * 8 bit multiplication iterations, the 'sum' and 'carry' terms of the product are sent to two latches using busses A and B, and are added using the fraction unit adder. The result, latched in the RES latch, is then written back to the register file.

3.7.2. Timing and Control

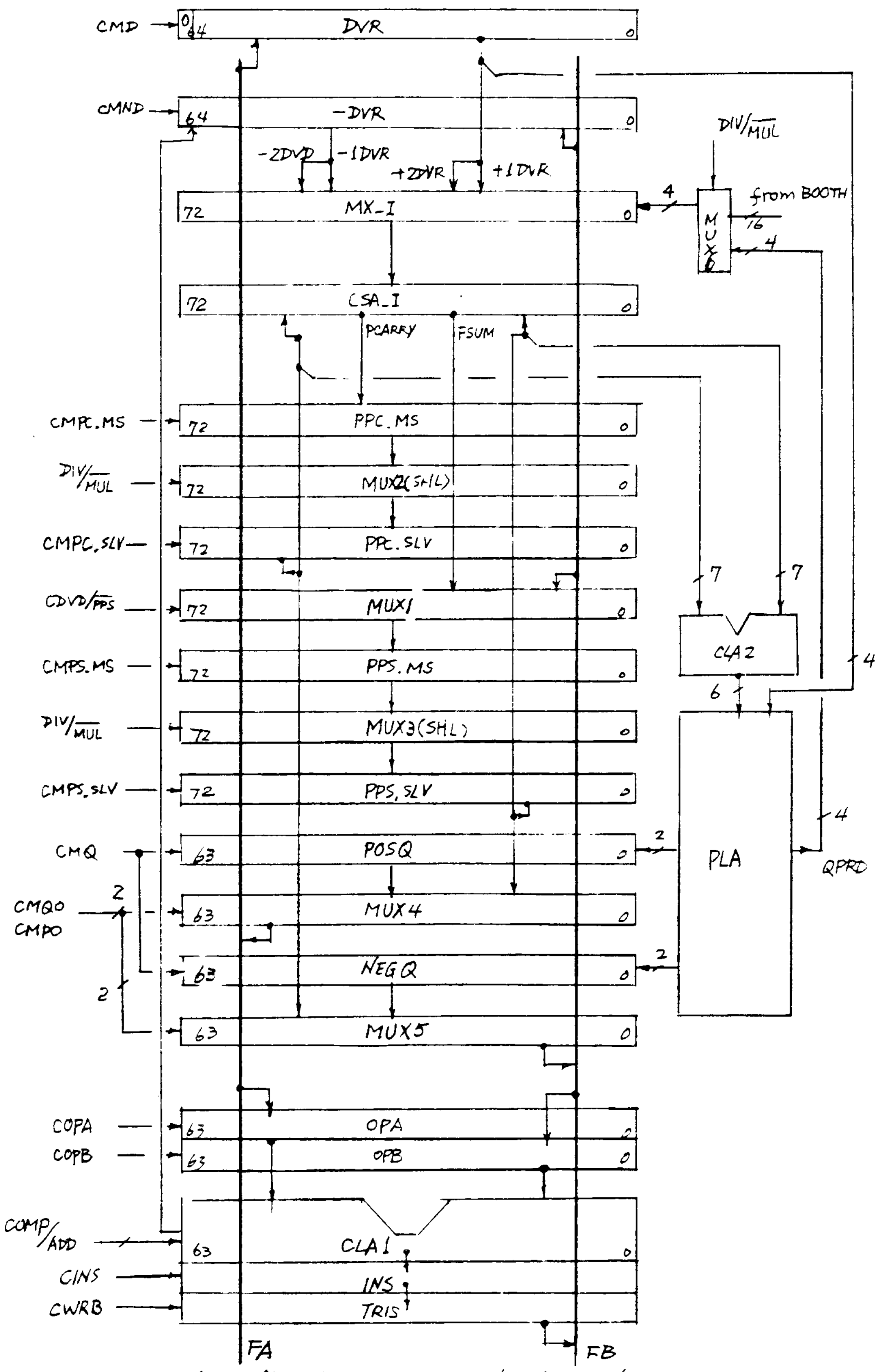
The timing scheme is determined by the following conditions:

1) The data busses are both precharged during phases 2 and 4 of every cycle, and data is valid in phases 1 and 3.
2) Bus A is available during the first execution cycle of the multiplication, but not available for internal operations in the other cycles. This is because that bus is used by external memory to load data into the register file. Bus B is available during all cycles of the execution of the multiply instruction.
3) The propagation delay of the 64 * 8 inner loop is approximately 40ns. Hence it is possible to execute the loop twice in one machine cycle. The timing of the activities of each block is illustrated in Fig. 3.24. Table 3.2 is the truth table of the control signals for multiplication.

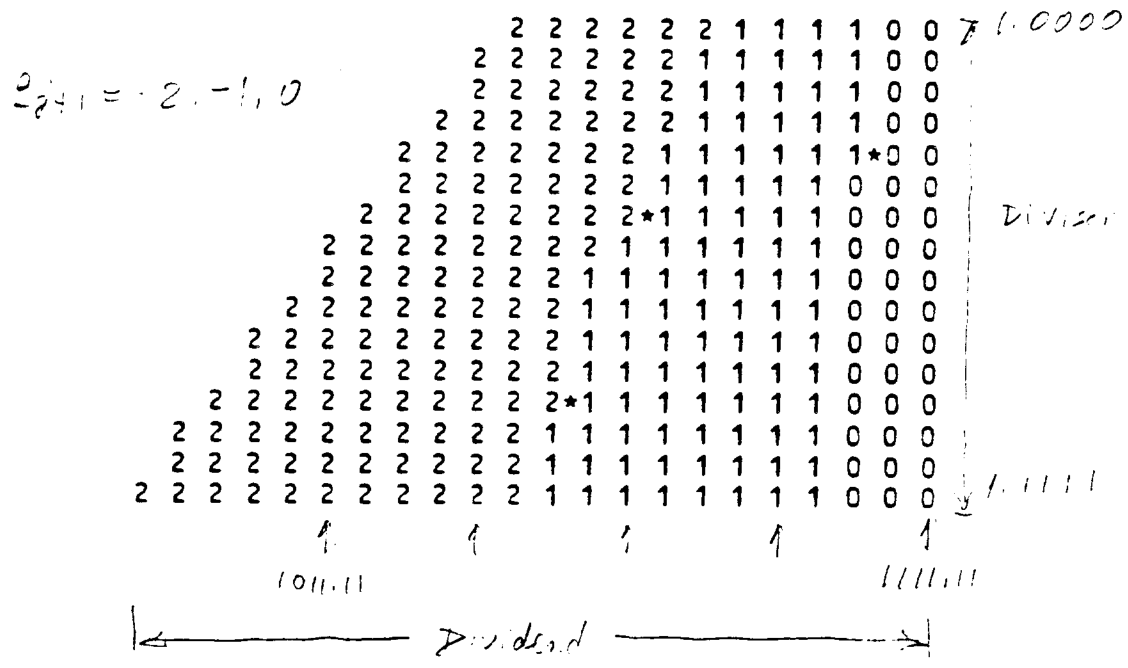
3.8. The Divider

Fig. 3.25 shows the schematic diagram of the DIVIDE unit. It is clear that much of the hardware is shared between the multiply and divide units.

The hardware design for fraction division is based on the radix four, non-restoring division algorithm using estimates of divisor and partial remainder. The radix four quotient digits are expressed ,using redundant representations

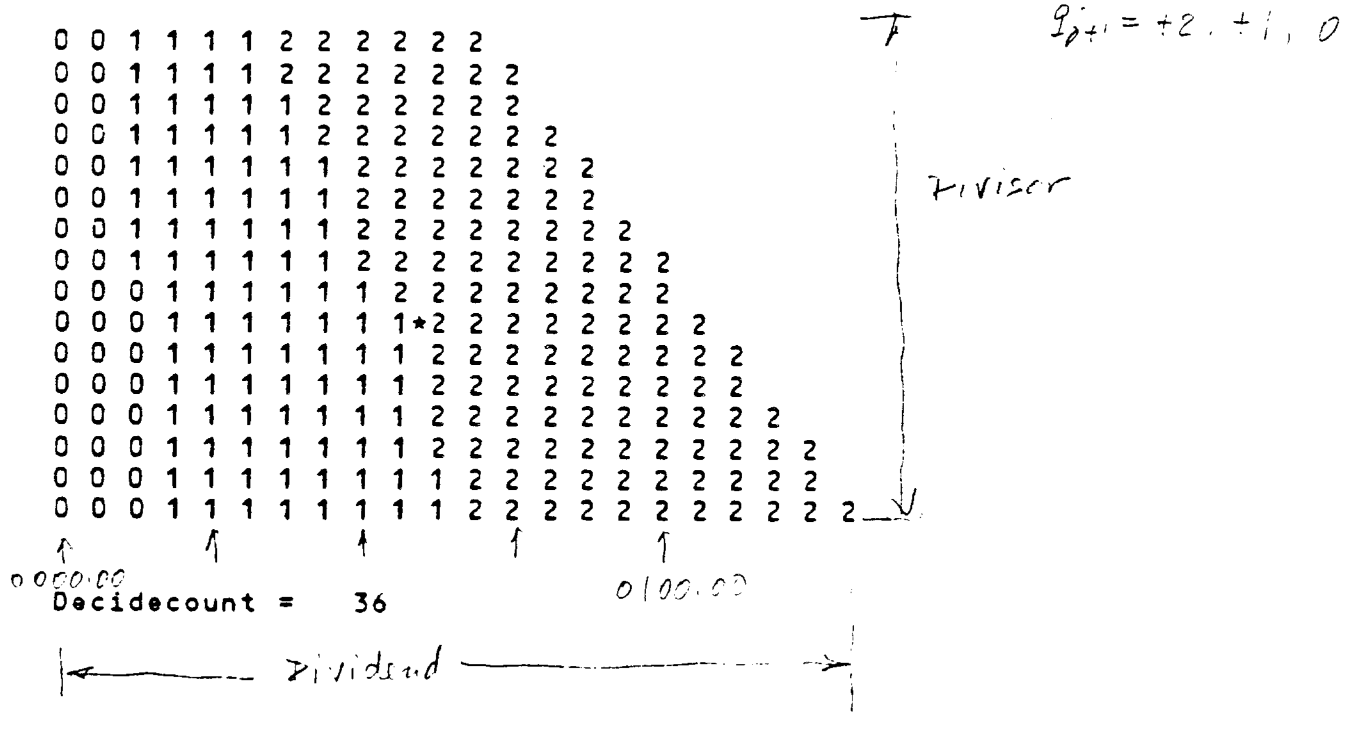


Schematic diagram of the Divider
FIG. 3.25



```
                    2 2 2 2 2 2 1 1 1 1 0 0
                  2 2 2 2 2 2 1 1 1 1 1 0 0
                  2 2 2 2 2 2 1 1 1 1 1 0 0
                2 2 2 2 2 2 2 1 1 1 1 1 0 0
              2 2 2 2 2 2 2 1 1 1 1 1 1*0 0
              2 2 2 2 2 2 2 1 1 1 1 1 0 0 0
            2 2 2 2 2 2 2 2*1 1 1 1 1 0 0 0
          2 2 2 2 2 2 2 2 1 1 1 1 1 1 0 0 0
          2 2 2 2 2 2 2 1 1 1 1 1 1 1 0 0 0
        2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 0 0 0
      2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 0 0 0
      2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 0 0 0
    2 2 2 2 2 2 2 2 2 2*1 1 1 1 1 1 1 0 0 0
  2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0 0
  2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0 0
2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 0 0 0
```

```
Mar 29 14:37 1985  2641og9_2.0out Page 2
```



```
0 0 1 1 1 1 2 2 2 2 2 2
0 0 1 1 1 1 2 2 2 2 2 2 2
0 0 1 1 1 1 1 2 2 2 2 2 2
0 0 1 1 1 1 1 2 2 2 2 2 2 2
0 0 1 1 1 1 1 1 2 2 2 2 2 2 2
0 0 1 1 1 1 1 1 2 2 2 2 2 2 2
0 0 1 1 1 1 1 1 2 2 2 2 2 2 2 2
0 0 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 1*2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
Decidecount =    36
```

FIG. 3.26

TABLE 3.3. Control Signals of the Divider

NO	Symbols	Functions	Timing cycle# 1	cycle# 2	cycle# 3	cycle# 4 to cycle# 16	cycle# 17	cycle# 18	cycle# 19
1	CMR	Latch MR							
2	CMD	Latch positive Divisor	ϕ_1						
3	CMND	Latch negative Divisor		ϕ_1					
4	CMBS<1>	Not use							
5	CMBS<2>								
6	CMBS<3>								
7	CMBS<4>								
8	CMBS<5>								
9	CMBS<6>								
10	CMBS<7>								
11	CMBS<8>								
12	CMPC.MS	Latch PPC (Master)		ϕ_2 ϕ_4	ϕ_2 ϕ_4	same as cycl# 3	ϕ_2 ϕ_4		
13	CMPC, SLV	Latch PPC (slave)	ϕ_3	ϕ_3	ϕ_1 ϕ_1	for cycle# 4 to cycl# 16	ϕ_1 ϕ_3	ϕ_1	
14	CMPS, MS	Latch PPS (Master)	ϕ_4	ϕ_2 ϕ_4	ϕ_2 ϕ_4		ϕ_2 ϕ_4		
15	CMPS. SLV	Latch PPS (Slave)	ϕ_3	ϕ_1 ϕ_3	ϕ_3 ϕ_3		ϕ_1 ϕ_3	ϕ_3	
16	CMQSL	Latch & Shift Left 2 bits of Quotient		ϕ_2 ϕ_4	ϕ_2 ϕ_4		ϕ_3 ϕ_4		
17	CMPO	Output the PPS and PPC of the remainder							ϕ_1
18	CMQO	Output the Pos & Neg Quotient to CLA						ϕ_1	
19	$\overline{MUL}$/DIV	MUL (Low) or DIV (High)	←			HIGH			→
20	CLRP	Clear PPS, PPC (Master)	ϕ_2						
21	COPA	Latch operand A of CLA (Divisor)	ϕ_1					ϕ_1	ϕ_1
22	COPB	Latch operand B of CLA						ϕ_1	ϕ_1
23	COMPA	Making 2's Comp of opA by CLA	ϕ_1 thru ϕ_4						
24	CINS	Latch instant result of CLA	ϕ_4					ϕ_4	ϕ_4
25	CWRB	Write inst. result to BUS-B						ϕ_4	ϕ_4
26	CRFWRB	Write the BUS-B into ~~reg~~. file						ϕ_4	ϕ_4
27	CBRC	Not use							
28	CADD	Add OPA and OPB by CLA						ϕ_1 thru ϕ_4	ϕ_1 thru ϕ_4
29	CMDV/$\overline{PPS}$	Mux Dividend or PPS.MS	ϕ_4						

-2, -1, 0, +1 and +2, and the partial remainders are irredundant. This redundancy of the expression of the quotient digits permits less precision in comparing the divisor and partial remainder to select a quotient digit. Atkins has presented the theory of high-radix non-restoring division. The required precision in inspecting the partial remainder and the divisor can be determined using the P/D plot method suggested by Freiman. It was found that 7 bits of the partial remainder and four bits of the divisor are needed for determining the next quotient digit. The quotient digits are generated by a PLA, which is provided input from the truncated partial remainder (7 bits) and the truncated divisor (4 bits plus an implicit MSB). The truth table of the PLA is shown in Fig. 3.26.

Since the hardware for creating -2MCD, -1MCD etc. is already there, this is shared between multiply and divide to generate the partial remainder from the divisor and the quotient digit. See Fig. 3.25.

The division is done iteratively, with two quotient bits computed per iteration.The hardware loop for generating the next remainder in parallel with generating the next quotient digit is :

1) A 7-bit carry-lookahead-adder, which generates the truncated partial remainder from the 'sum' and 'carry' terms of the previous partial remainder and four bits of the truncated divisor. The three-bit output represents one of the possible values of the quotient digit, out of -2, -1, 0, +1 and +2. The outputs of the PLA are further decoded into two 2-bit sets corresponding to the groups -2, -1, 0 and +2, +1, 0. These two 2-bit numbers are fed into registers POSQ (positive quotient) and NEGQ (negative quotient) respectively, and shifted left two bits per iteration. The 3-bit output of the PLA are also decoded into another 4-bit set which is used to control a mux (MX-I) to make the product of the next quotient digit and the divisor (-2DVR, -1DVR, 0, +1DVR, +2DVR) for generating the next partial remainder. The iterative relation is:

$$p_{j+1} = r\,p_j - q_{j+1}\,d$$

j	index of the recursive loop <31:0>
p_j	partial remainder obtained in the jth loop
p_0	dividend
p_{31}	final remainder
q_{j+1}	quotient digit after the jth loop
d	divisor
r	radix (r=4, rp => shift p left 2 bits)

2) One row of the CSA array is used for adding the '-q_{j+1} d' term with the partial remainder, 'rp'.

3) Master-slave registers PPS and PPC are used to keep the updated partial remainder in separate 'sum' and 'carry' terms.

4) Shift registers, POSQ and NEGQ, are used to keep the positive and negative digits of the quotient respectively, and shift left two bits every iteration.

The critical path of the quotient/partial remainder generation loop includes the following: an 8-bit carry-lookahead-adder, a PLA with approximately 20 minterms, a 2:1 mux, a 4:1 mux, one stage of CSA and the PPS/PPC latch. The loop delay is estimated to be 40 ns, and so we expect to run the loop twice per machine cycle. Thus the inner loop for 64 bits takes 32 iterations or 16 cycles. A few cycles are needed before and after loop evaluation for getting divisor and dividend, forming the negative divisor, adding POSQ and NEGQ to get the quotient and adding PPS and PPC to get the remainder.

A mux (MUX-1) is added in front of PPS to latch in the dividend from data Bus B. There are two muxes inserted between master and slave stages of PPS and PPC. These two muxes are used to shift the contents of these latches. Shift length for 'sum' is 8 bits and for 'carry' is 7 bits, when multiplying. However, when dividing, 'sum' is shifted left 2 bits, and 'carry' is shifted left 3 bits. This is done by muxes MUX_2 and MUX_3.

In order to make the division loop as fast as possible, the output of the first row of the CSA array is routed directly to the inputs of MUX_1 and PPC.MS. The control signals and timing for divide are shown in Table 3.3. Circuit implementation details of important sections of the multiply/divide unit are enclosed in Appendix B, together with relevant design notes. A SPICE simulation of the CSA is included in Appendix C.

4. Chip Floorplan and Power Supply Routing

The tentative floorplan of the FPU is shown in Fig. 4.1. The total chip size is estimated to be .8cm * 1cm (.8 square cm.) The area of the fraction datapath is estimated to be 4500u * 5500u and of the mul/div datapath is tentatively 2500u * 5500u. A strip immediately above the main datapath, 500u wide, is reserved for decoders, buffers, and other miscellaneous logic. The exponent datapath is expected to be 3500u * 2000u. Totalling all this, the datapath area is about .5 square cm., about 60% of total chip area. With about 22% of the chip area reserved for peripheral circuits, we are left with about 18% chip area for control and global routing.

Fig. 4.2 shows power supply routing in the FPU. Since we are restricted in the X direction, we decided to run data lines in parallel with power supply lines in that direction, all in second layer metal. Control lines run orthogonally in first layer metal. Because of the high bit rates and small rise and fall times of external signals, inductance between pins can be a severe performance limitation. According to present estimates, we may need to reserve as many as 15 VDD and GND pads, to minimise this inductance problem, and enable us to run at speed.

FPU Floorplan

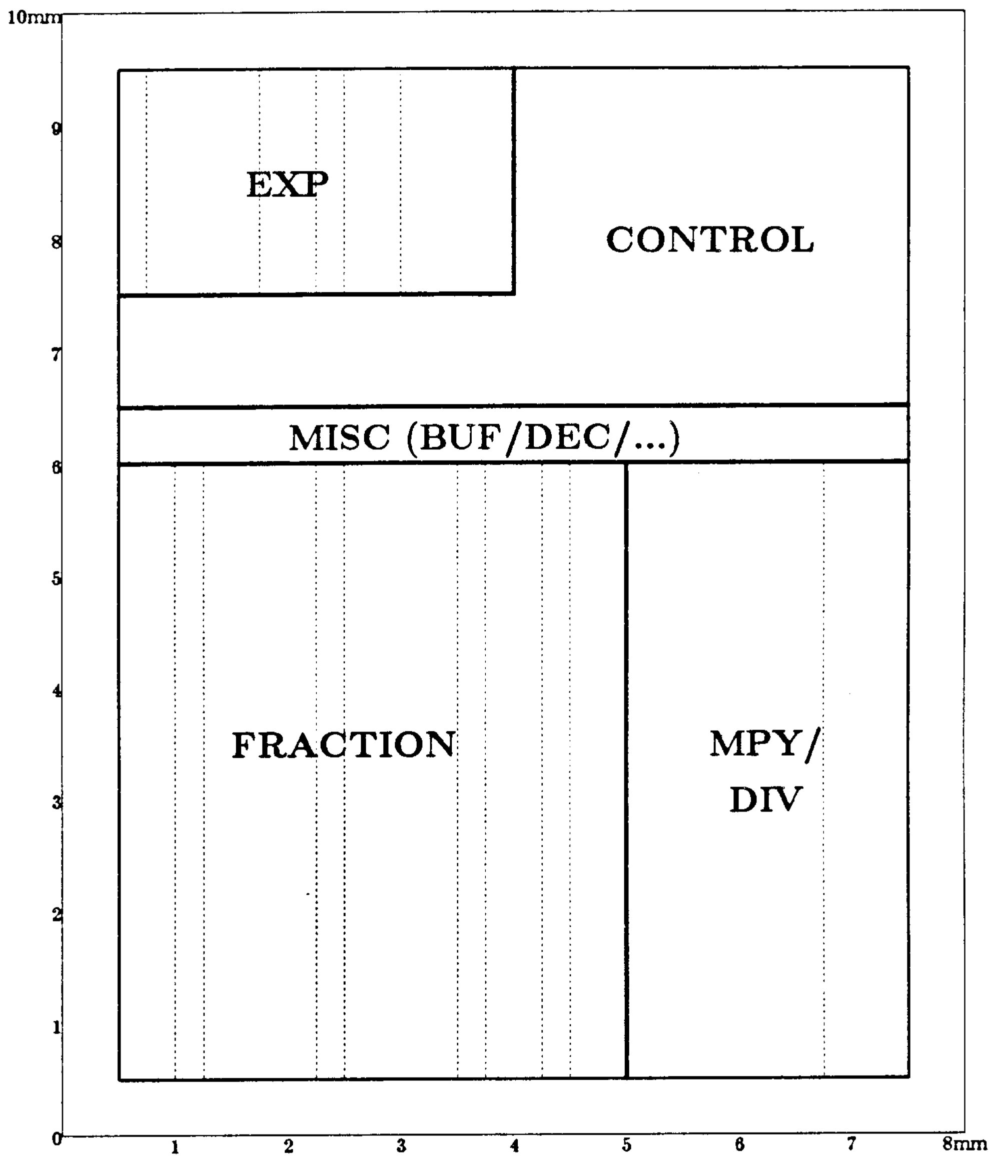


Fig. 4.1

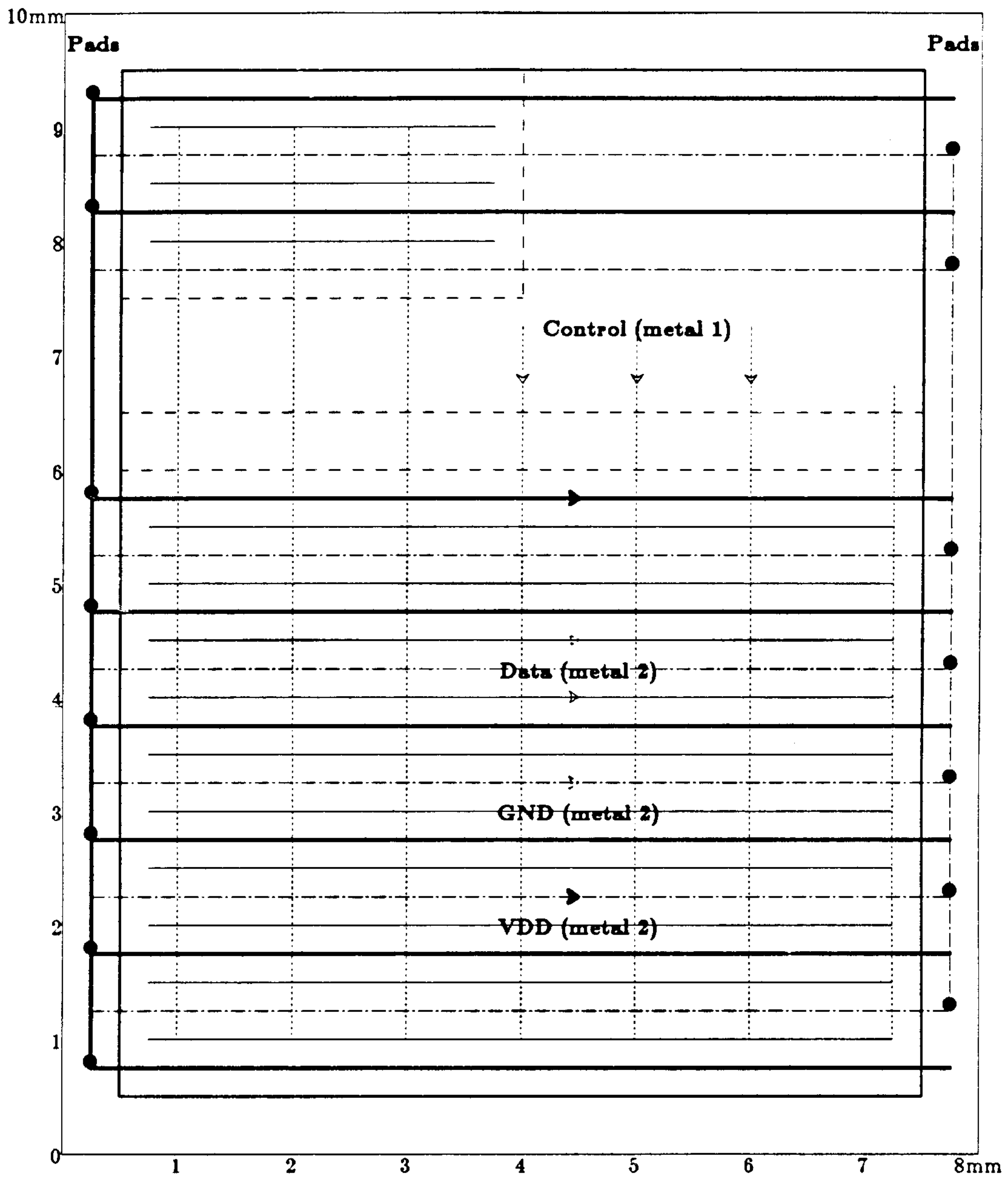


Fig. 4.2 Power and Signal distribution

5. Design Status

We still have to do detailed SPICE simulations on some of the cells that are process-sensitive and speed-critical. Once the individual blocks are ready, they will have to be connected together, and then tested for functional correctness using ESIM, and also timing estimates will have to be verified using CRYSTAL.

A description of data flow in the FPU is being coded in SLANG. Once it is complete, we will have to verify functional correctness of the datapath. Then the timing information will have to be included in the SLANG description, and when the whole system is verified, the control PLAs have to be generated, and laid out.

When the datapath and the control sections are complete, global routing will have to be done, and diagnostics run on the extraction of the laid-out circuit to see if the circuit indeed matches the already verified simulator.

We plan to build at least two test chips and send them out for fabrication, to analyse key blocks in the design and get invaluable feedback about our target pocess. One will be a 64-bit adder/subtractor with a 16-bit Brent-Kung carry-lookahead scheme; the other will be the multiply/divide unit.

6. Summary

The design of a VLSI circuit to perform floating-point operations is described above. It conforms to the IEEE standard, and supports the different operand types, rounding modes, special cases etc, that the standard demands. The estimated number of cycles for ADD, MPY and DIV are 4, 7 and 20 respectively. The area of the chip is estimated to be 0.8 square cm. Much work has been accomplished this past semester. Major modules of the datapath have been designed and laid out. But, as expected with a project of this magnitude, much work still remains to be done.

SPUR COPROCESSOR INTERFACE DESCRIPTION

Paul M. Hansen

Computer Science Division of EECS
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

This report describes the SPUR coprocessor interface. The interface provides enhanced performance potential by allowing parallel operations between the SPUR processor and SPUR coprocessors. A decoupled control and execution architecture allow data transfers to proceed while coprocessor functions are performed. Implicit and explicit synchronization mechanisms provide the programmer complete control and flexibility. On-chip coprocessor register files and a wide data path between the memory and coprocessor minimize data transfer overhead. An intelligent interface control unit provides parallel decoding of instructions for maximum performance. Other coprocessor functions applicable to signal processing, workstation graphics, and so forth are being considered, but will not be reported here.

1. INTRODUCTION

The SPUR CPU is a custom VLSI 32 bit general purpose host targeted to support Lisp and other high-level language software environments. The RISC-like architecture provides high performance for a wide range of applications.

Traditional von Neumann computer architectures have achieved enhanced performance by adding optional hardware to perform tasks that are usually executed in software. These devices include such things as attached processors, array processors, floating point accelerators, data channels, graphics display processors, etc., and are called *coprocessors*. Thus, a coprocessor is an optional piece of hardware that replaces a piece of software for a higher level of performance.

Many peripheral devices as well as more closely coupled coprocessors fall in this general category. It is nevertheless important to recognize a distinction between standard peripheral hardware devices and *tightly coupled* coprocessors: the programming model for the coprocessor differs from that of peripheral devices. Standard peripheral hardware usually appears to the programmer as a set of registers in the memory space of the main processor. The programmer must consider the communication protocol and implement the interface between the peripheral and the device in software.

The tightly coupled coprocessor on the other hand adds additional instructions and generally additional registers and data types which are not directly supported by the main processor architecture. Dedicated coprocessor instructions allow the programmer to utilize the coprocessor capabilities. However, certain interactions needed between the main processor and the coprocessor (i.e., the communications protocol) are implemented in hardware and are transparent to the programmer. Thus, the coprocessor can extend the functions provided to the user without appearing as hardware external to the main processor. This provides a more uniform programming model from a user point of view.

The SPUR system employs an optional special purpose device for floating point arithmetic. We refer to this as the SPUR Floating Point Unit, or simply FPU. In the general case, it would seem logical to refer to all signals and mnemonics related to the coprocessor to be designated "CP". Other applications are being considered besides floating point arithmetic, but this report will focus on the FPU. Thus, to avoid confusion between the CPU and CP designations, the coprocessor interface signals, blocks, modules and functions will be designated with the "FPU" prefix (even though, as mentioned, the interface will support other devices).

Section 2 of this report provides a brief overview of the SPUR coprocessor interface and functions. Section 3 provides a greater degree of detail and timing diagrams for various operations, instructions, and CPU <---> FPU interaction.

2. FLOATING POINT COPROCESSOR INTERFACE OVERVIEW

From the assembly language programmers point of view, the SPUR FPU consists of 15 read/write 80-bit operand registers and one read/write control/status register (nominally 64 bits). All arithmetic operations involve three registers: two source and one destination.

2.1. Instructions

As of this writing, approximately 30 operations are defined for floating point arithmetic and general coprocessor functions. These are listed in Table 1 at the end of the report.

2.2. Control Flow

The FPU coprocessor will employ two function units: the bus interface control unit (BICU) and the execution unit (EU). The pipelined architecture of the FPU is identical to the CPU. On phi3 of every cycle, the FPU BICU accepts and decodes the INSTRUCTION BUS fragment conveyed on *fpuOpcode* lines, and initiates operation in the subsequent cycle if it is an FPU operation. This continues until cycle n (n depends on the particular instruction being executed). The *fpuBusy* signal is asserted in cycle (n-1) to indicate when the FPU EU is done and allow instruction overlap. (See section 2.6 for complete definitions of signals.)

Under normal circumstances, CPU and FPU instructions execute in parallel. This parallelism is controlled in two possible ways: (1) explicit: the *fpuParallel* bit in the UPSW may be set, which will prevent overlap of CPU and FPU operation instructions, and (2) implicit: the *fpuBusy* line will prevent the CPU from issuing FPU operation instructions if the FPU is still in the execution phase of a previously issued instruction. In all cases where overlap is prevented, the CPU stalls until the *fpuBusy* line is not asserted.

2.3. Data Flow

Data flows between the FPU and the SPUR data cache memory under direct control of the CPU. The data path to the cache is 64 bits wide. Double precision operands are loaded in one cycle. As well, loads may proceed in parallel with FPU operation, since the FPU register file is dual ported. The FPU pipeline is similar to the CPU pipeline: the load instruction requires the fetch, effective addresses calculation, memory access and register write cycles before the operand is ready for use. However, there is no operand forwarding in the FPU, so loaded data is not ready for use as an operand in the FPU until the third instruction following the load.

Two instructions allow loading the CPU registers directly from FPU registers, and vice versa without passing through cache memory. This is useful for transferring integer operands and status and control information.

2.4. Performance

Initial analyses indicate the timing for fundamental algebraic instructions supported by the FPU as shown in Table 2.

Studies comparing the SPUR FPU with commercial microprocessor-based systems employing VLSI floating point coprocessors indicate that the SPUR-FPU combination can execute the Berkeley Loops between 5 and 15 times faster than other systems [Han85]. The main performance advantages come from (1) the dual ported register file allowing data loads during FPU operation, (2) the overlapped execution of the FPU and CPU, and (3) very efficient algorithms and hardware structures for the four operations implemented on-chip: add, subtract, multiply, and divide.

2.5. Programming Interface

The FPU effectively adds new data types, new registers, and new instructions to the CPU For the most part, the coordination of the processor-coprocessor operation is handled by the programming languages and coprocessor interface automatically. The architecture is Load/Store, with arithmetic operations between FPU registers. The hardware is invoked directly by the programmers instructions, and no recompilation is necessary for systems which are not equipped with an FPU. Simple link-time command arguments direct the loading of algebraic routines in the absence of the FPU. One bit in the UPSW causes the CPU to trap if an FPU is not available in the system.

2.6. Hardware Interface

As a coprocessor in the SPUR system, the FPU is connected as shown in Figure 1. Figure 2 shows the logical interconnections between the CPU and FPU. The CPU and FPU both employ a 4-phase non-overlapped clocking scheme as illustrated in Figure 3. The interface signals fall into three groups: (1) new instruction valid, opcode specifier, coprocessor identification, and coprocessor suspend (2) register specifiers, and (3) coprocessor status.

2.6.1. CPU to FPU Signals

The signals between the CPU and FPU which provide control are described next. Many of the details of operation are contained in section 3 of this report: Coprocessor Interface Details.

fpuOpcode: 7 bits. This specifies the opcode of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

fpuRS1: 5 bits. This specifies the Source One register of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

fpuRS2: 5 bits. This specifies the Source Two register of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

fpuRD: 5 bits. This specifies the Destination register of the instruction which the CPU broadcasts to all coprocessors. The CPU starts driving these lines at the beginning of phi3.

fpuNewInstr: 1 bit. Asserted by the CPU whenever a valid instruction is issued. Similar to *fpuOpcode*, the CPU starts driving this signal at the beginning of phi3.

fpuID: 1 bit coprocesser identifier provided by the CPU to identify its target coprocesser. This bit is part of UPSW and is available to all coprocessors. The CPU may change this line only during phi1. (SPUR can have at most two coprocessors in this version. Later versions will allow multiple coprocessors.)

fpuSuspend: 1 bit. Asserted by the CPU to stall the FPU's operation. The CPU may change the value of this signal only during phi4.

2.6.2. FPU to CPU Signals

The signals between the FPU and CPU which provide status are described next. Many of the details of operation are contained in section 3 of this report: Coprocessor Interface Details.

fpuBusy: 1 bit. Asserted by the active coprocessor (in this case, the FPU) to indicate that the FPU is busy. The CPU reads (latches in) this bit during phi3. Therefore, the FPU must assert this signal before phi3 of the 2nd cycle (1st execution cycle) in a FPU's instruction.

fpuExcep: 1 bit notifies the CPU that an error condition exists in one of the coprocessors. This line is shared by all coprocessors because only one coprocessor can be active at any given time. The CPU reads (latches in) this signal during phi1. The *fpuExcep* signal is valid only when *fpuBusy* is unasserted.

fpuBrT_F: 1 bit. True/false line for FPU compare instructions. The CPU reads (latches in) this bit during phi3. The *fpuBrT_F* line is valid only if the *fpuBusy* is unasserted.

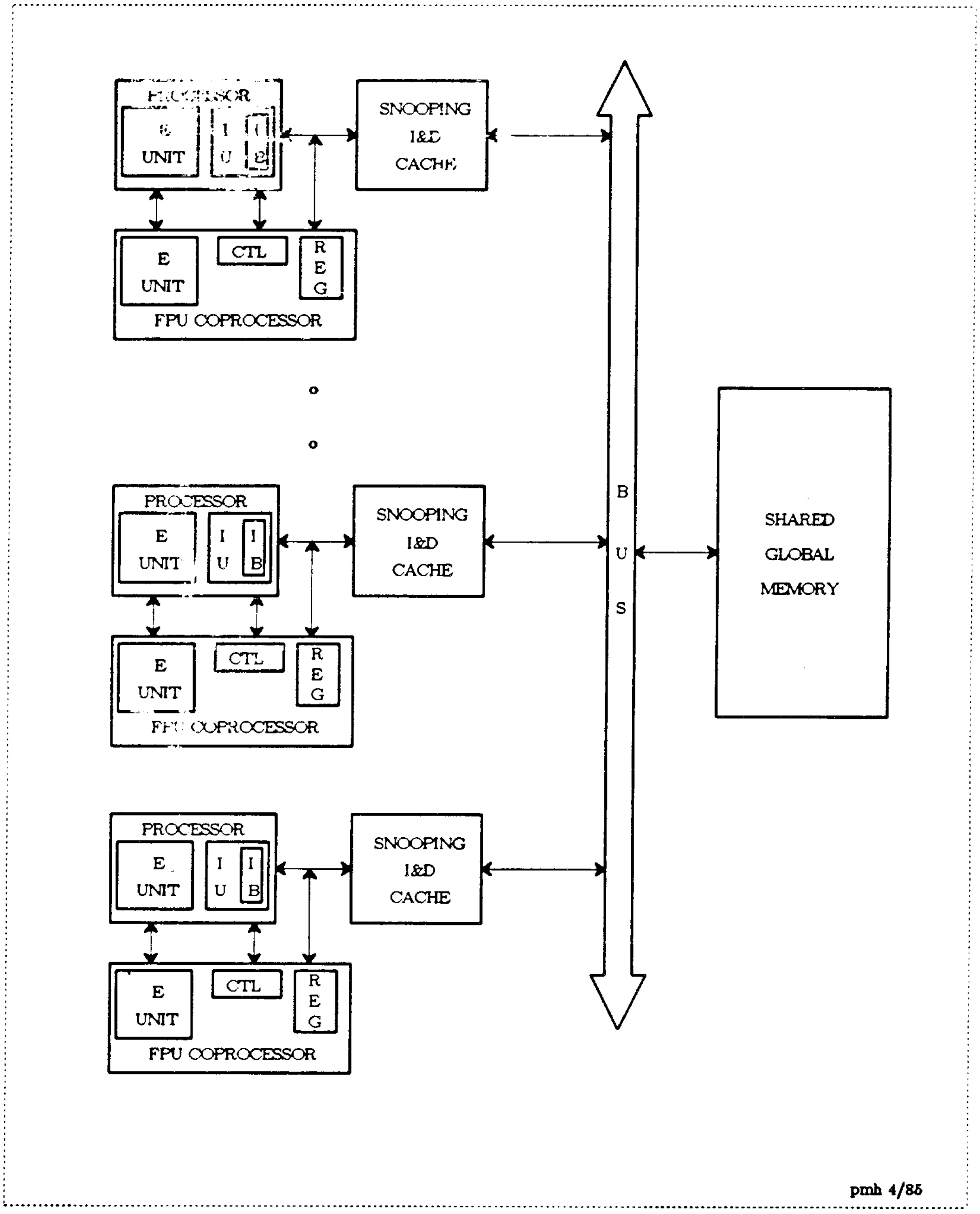


Figure 1. The UC Berkeley SPUR Multiprocessor System.

The processor is a single chip with an on-board instruction buffer (IB). The floating point unit (FPU) is tightly coupled to the CPU via the local processor bus. The cache controller is integrated on one chip with off chip tag and data RAMs. The caches work together to implement a cache consistency protocol, described in [KEP]. Shared memory and I/O devices are accessible through the system bus.

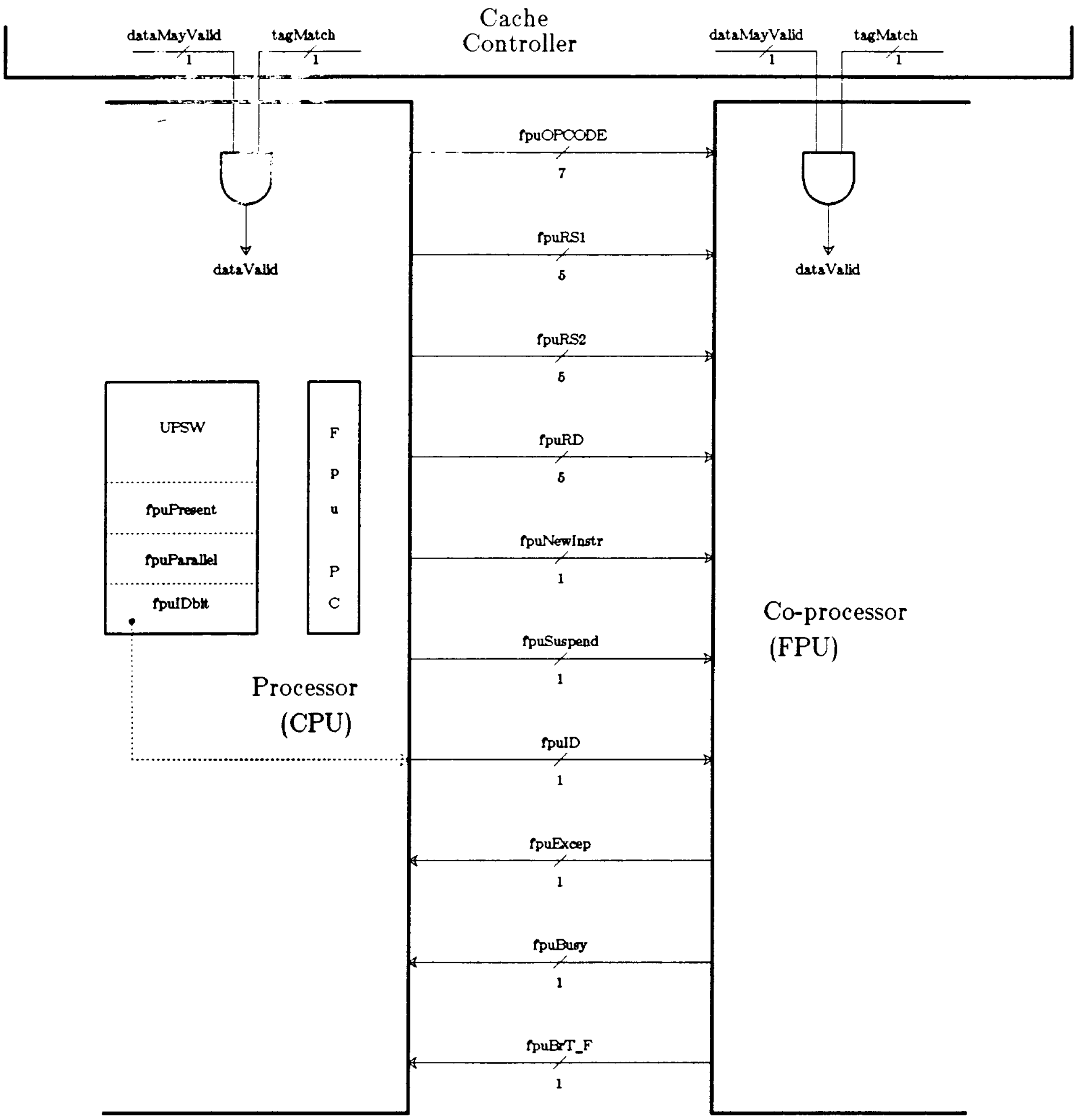


Figure 2. The SPUR Coprocessor Interface (FPU).
Signals dataMayValid and tagMatch come from the cache controller, as shown, to both the CPU and FPU.

Besides the signals above, the CPU must also have the following hardware to support coprocessors:

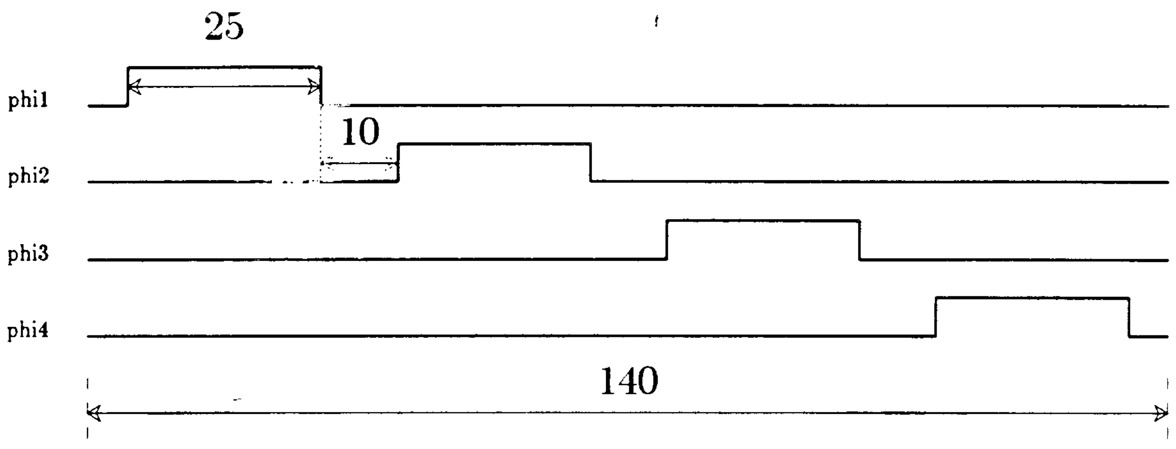


Figure 3. The SPUR system clocking scheme.
Clocking is 4-phase, non-overlapping as shown, with 25 nsec high levels, 10 nsec underlap low levels yielding a 140 nsec total cycle time.

fpuIDbit: This bit is sent out to the coprocessors via the *fpuID* signal line to inform which coprocessor is selected by the CPU. In the current implementation, this is 1 bit (maximum number of coprocessors is two) and is part of the UPSW.

fpuParallel: 1 bit. When asserted, it enables concurrent operation (parallel mode) of the processor (CPU) and the active coprocessor (FPU). If this bit is unasserted, concurrent operation is prohibited (forcing sequential mode). Concurrent operation is described later.

fpuPresent: 1 bit. When asserted, it indicates to the CPU that an FPU device is available in the system. When not asserted, the CPU traps to runtime routines to emulate floating point hardware operations.

fpuPC: A special register which stores the address of the last FPU instruction the CPU issues. Due to concurrent operation of the CPU and FPU, the PC inside the CPU may not be pointing at the FPU instruction that causes the exception.

2.7. Floating Point Unit Micro-Architecture

The description of the internal architecture and structure of the floating point unit is discussed in another report in this Technical Memo. The following sections of this report describe in detail the interaction between the CPU and FPU relative to normal processing, exception and interrupt handling, concurrent operation, and so forth. Much of the material and diagrams included here appeared earlier as part of an internal working document: *Chapter 7. Coprocessor Interface - Processor's Perspective*, by Shing I. Kong, UC Berkeley, August 1985.

3. COPROCESSOR INTERFACE DETAILS

3.1. Sending Instructions to Coprocessors and Suspending Operation

Every instruction the CPU fetches from its internal instruction buffer is sent to all co-processors via *fpuOpcode, fpuRS1, fpuRS2,* and *fpuRS3.* The active coprocessor, which is selected by the *fpuIDbit* must decode every new instruction it receives to determine whether or not the instruction is intended for it. However, due to occasional CPU pipeline suspension, the CPU may not fetch one instruction per cycle and thus may not have one instruction to broadcast every cycle. Two reasons may exist for CPU pipeline suspension:

1. The FPU is busy with a previous operation and the CPU must to wait until it becomes free.
2. Other reasons that are NOT related to FPU operation. These include cache miss, multiple cycle instructions, and so forth.

Whenever the CPU's pipeline is suspended because the FPU is busy, the FPU operation cannot be suspended. Otherwise, deadlock may result. On the other hand, whenever the CPU's pipeline is suspended due to reasons mentioned above in the second category, certain FPU activities must be suspended to prevent it from advancing to an inconsistent state if a trap occurs. Thus, FPU operations can be suspended by the CPU with the *fpuSuspend* signal. (Note: The FPU does not have to suspend everything as soon as it receives the *fpuSuspend* signal. It must simply remain in a state where certain instructions can be killed if a trap occurs and before writing to internal registers.) This is illustrated in Figure 4, which shows that the CPU asserts the *fpuSuspend* signal during phi4 before the first suspended cycle and disasserts the *fpuSuspend* signal during phi4 before the first normal cycle after suspension.

There are two other things in Figure 4 worth pointing out before continuing the discussion:

1. Ifet(FPU) means FPU is in Ifet cycle and NoOp(CPU) means CPU is in NoOp cycle.
2. Depending on whether I1 is a CPU instruction or an FPU instruction, the Exec cycle of I1 can either be in the CPU or the FPU.

The CPU asserts the signal *fpuNewInstr* to tell the coprocessor that new values for *fpuOpcode*, *fpuRS1*, *fpuRS2*, and *fpuRS*3 are available and valid. During CPU pipeline suspension, no new instructions are issued to the FPU. This is also illustrated in Figure 4 which shows signal *fpuNewInstr* unasserted zero during the CPU pipeline suspension.

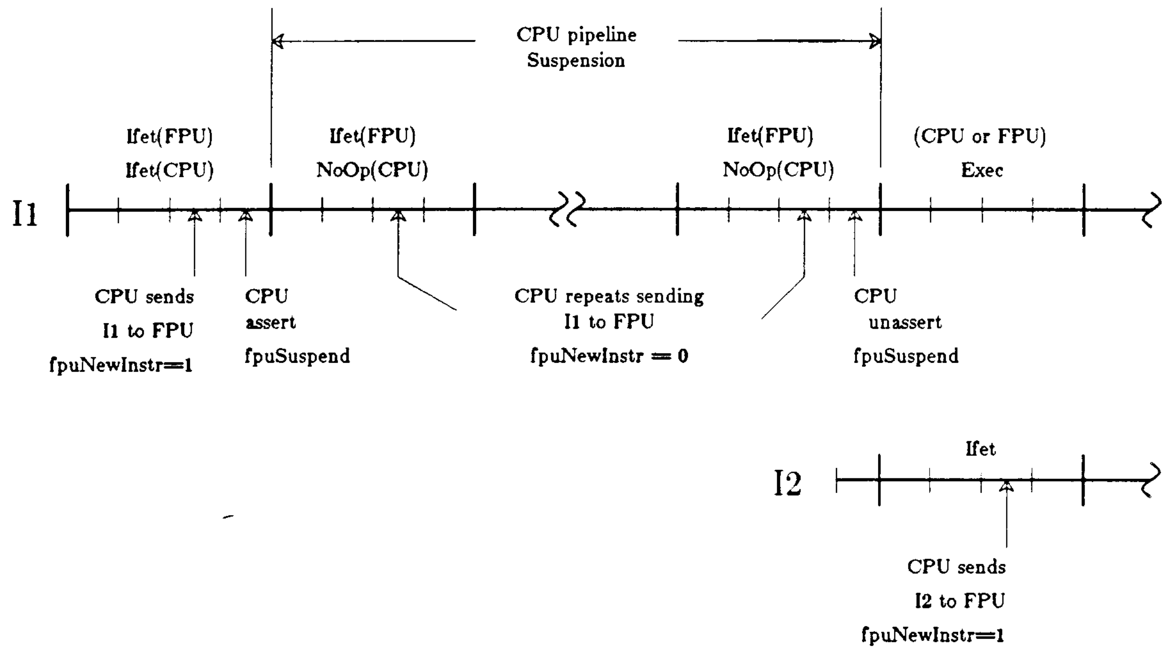


Figure 4. Illustration of fpuSuspend signal.

In Figure 4, also notice that during CPU pipeline suspension, the CPU continues broadcasting the last instruction it issued (I1 in Figure 4) before suspension. Thus, the FPU can continue latching in the instruction (I1 is always available to the FPU during CPU pipeline suspension) until the suspension is over. In other words, the Ifet cycle of I1 in the FPU is being repeated until the CPU pipeline suspension is over. This is illustrated in Figure 4 by the repeating Ifet(FPU) cycles of I1.

One final note: an instruction buffer miss does not cause the CPU pipeline to be suspended. This is handled by using an internal MISS instruction (detailed in the report by S. Kong).

3.2. Concurrent Operation of CPU and FPU

The CPU and FPU can function in two different modes: parallel and sequential. In parallel mode, the *fpuParallel* bit in CPU's UPSW is asserted and concurrent operations of the CPU and FPU are allowed. The concurrent operation of the CPU and FPU can be summarized as follows:

1. After the FPU load or FPU store is issued, the CPU and FPU can continue to execute either CPU or FPU instructions. (Note: There must be at least one FPU instruction between the FPU store and the FPU instruction that produces the result to be stored. The pseudo FPU instruction SYNC may be used for this purpose.)

2. After the FPU compare instruction is issued, the CPU cannot execute any other instruction until the FPU compare instruction is done.
3. After an FPU instruction is issued which is not in the above two categories, the CPU can continue to execute CPU instructions, or FPU load or FPU store instructions. No new FPU operation instructions are allowed to begin until the previous FPU operation instruction is done.

In sequential mode, the *fpuParallel* bit is not asserted and concurrent operations of CPU and FPU are disallowed. All instructions must be executed sequentially. After the CPU issues an FPU instruction, the CPU cannot continue to execute any instructions until the FPU instruction is done. In other words, all FPU instructions are treated like FPU compare mentioned in Case 2 above.

In the above discussion, the phrase "instruction is done" is used without any explanation of how it is detected. In this FPU interface, the FPU will disassert the *fpuBusy* signal whenever the FPU instruction is done. This is illustrated more clearly in the next section with timing diagrams.

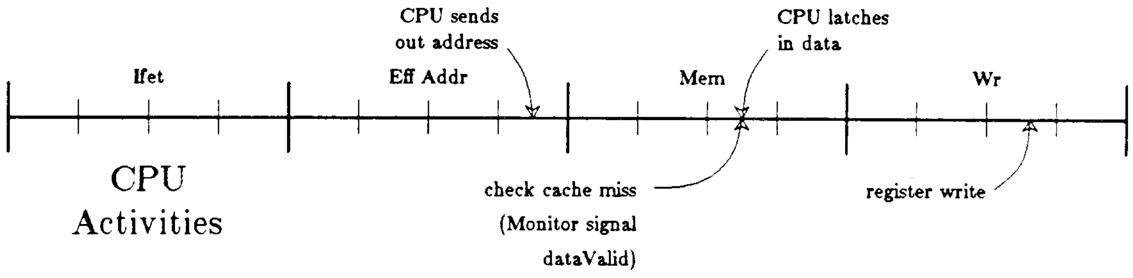
3.3. Timing of Coprocessor (FPU) Instructions

3.3.1. Timing of the FPU Load & Store

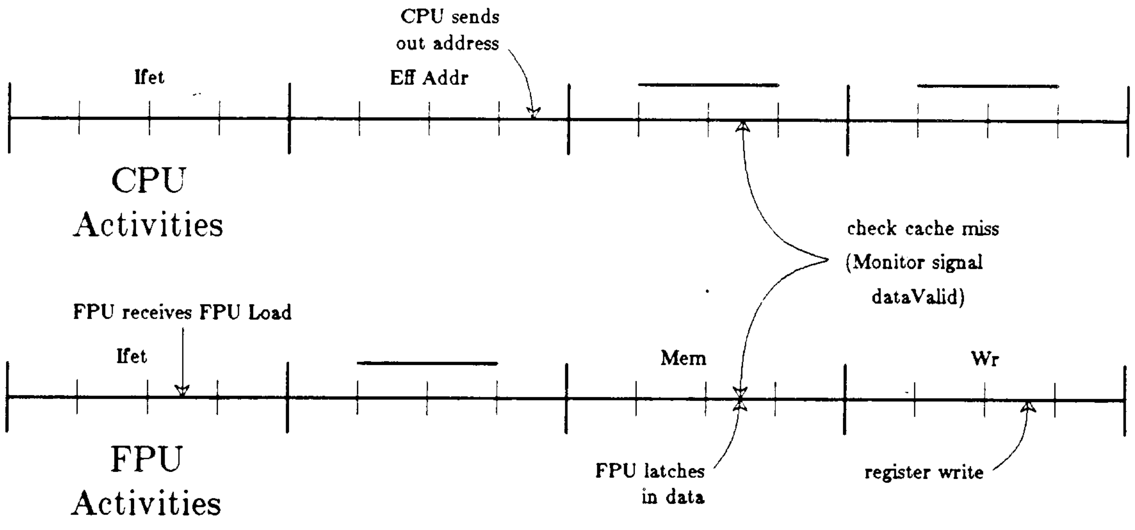
Figure 5b shows the timing of the FPU load instruction. The only obvious difference between this and a regular CPU load (Figure 5a) is that the FPU latches in the data instead of the CPU. Similarly the only obvious difference between a regular CPU store (Figure 6a) and the FPU store (Figure 6b) is that the FPU sends out the data instead of the CPU. The goal here is to make the FPU behave identically to the CPU: receive input data and send output data during the same times the CPU would. This essentially makes the FPU transparent to the cache controller.

One important point shown in Figures 5 and 6 is that in a FPU load or store instruction, both the CPU and FPU are responsible for checking cache miss. Both the CPU and FPU must monitor the dataMayValid and tagMatch signals to form their own dataValid line (see Figure 2) This is necessary because a cache miss requires both the CPU and FPU to do something special. The CPU has to suspend the pipeline (as discussed in the cache controller section). The FPU, on the other hand, has to repeat its memory access cycle and suspend all FPU activities that are related to FPU instructions received after the FPU memory access instruction.

This is illustrated in Figure 7, which shows an FPU load, which causes a cache miss, and is followed by another FPU load instruction. For the more complicated case where a CPU load or store instruction is followed by an FPU load or store instruction, please refer to Section 3.4.1. Notice that in Figure 7,



Part a. CPU LOAD

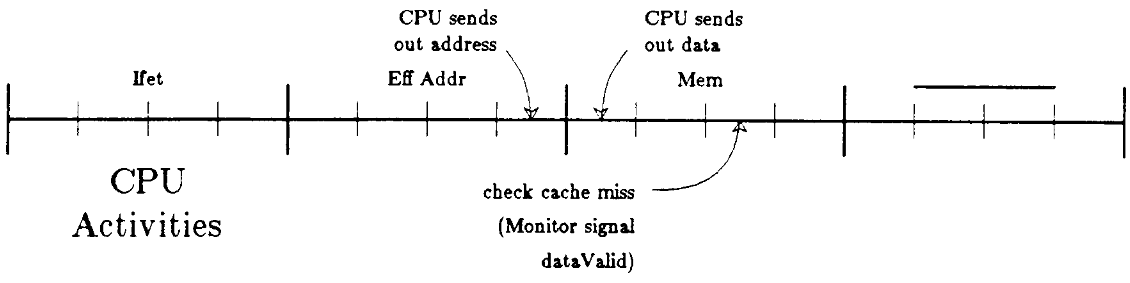


Part b. FPU LOAD

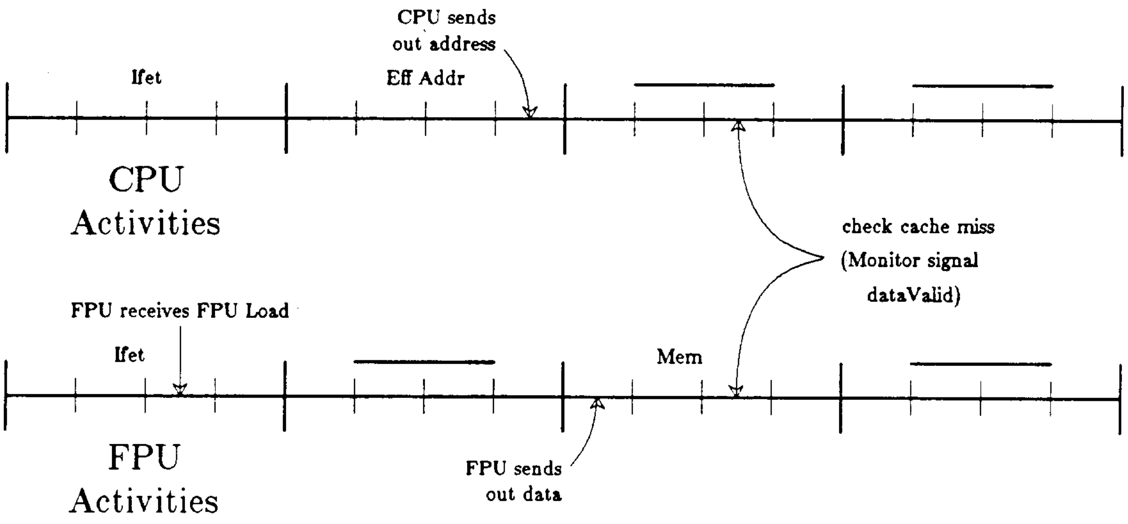
Figure 5. CPU and FPU load instruction timing.

the CPU does not assert the *fpuSuspendfP signal during the CPU pipeline suspension, because the FPU cannot stall its memory access operation. The FPU, however, must suspend all FPU activities that are related to FPU instructions received after the FPU memory access instruction, until the pending cache miss is serviced. For example in Figure 7, the second FPU load instruction is suspended until valid data is received for I0. Any FPU instruction received by the FPU before I0, however, can continue its execution.*

Furthermore, the FPU load and store instructions do not cause *fpuBusy* to be asserted. If there is no cache miss, the FPU load and store instructions are both four cycle instructions, the same as all other CPU instructions. In the case of



Part a. CPU Store



Part b. FPU Store

Figure 6. CPU and FPU store instruction timing.

a cache miss, the *fpuBusy* line must also remain low so that the FPU load and store can be handled like CPU load and store.

In Section 3.2, it was shown that the FPU load or store instruction can be treated the same as a CPU instruction as far as concurrent operation of the CPU and FPU is concerned. Here it is shown that the timing of the FPU load or store is very similar to regular CPU load. Therefore, it may be advantageous to give the FPU load and store an opcode similar to other CPU instructions as compared to an opcode similar to other FPU instructions. Even though both the CPU and FPU monitor the dataValid line, only the CPU has to monitor the pageFault or busFault lines. This is explained in Section 3.4.

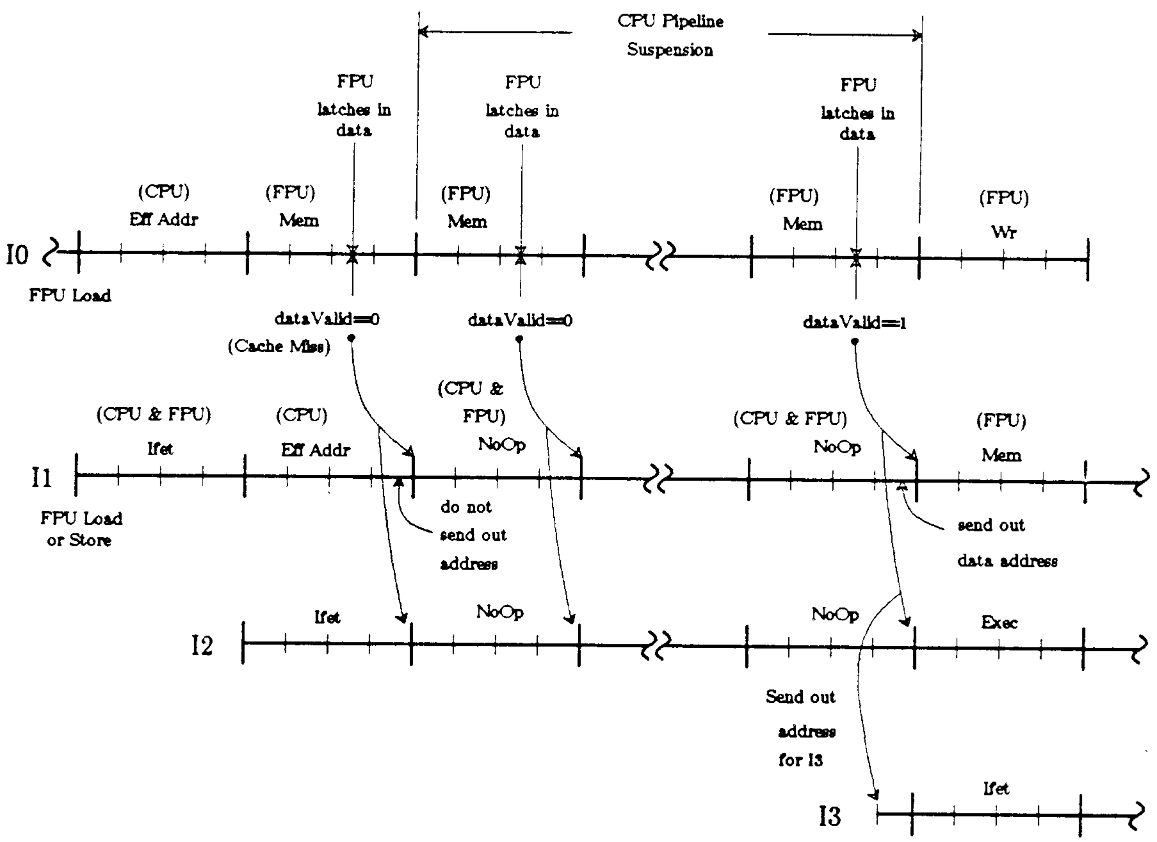


Figure 7. FPU cache miss followed by FPU load or store.

3.3.2. Timing of FPU Compare Instruction

The timing of the FPU compare instruction is shown in Figure 8. For simplicity, it is assumed that when this instruction is launched, the FPU is not busy (*fpuBusy* = 0). The case where the FPU is busy (*fpuBusy* = 1) is similar to the case discussed in Section 3.3.3.

Assuming the FPU compare instruction takes more than one cycle to finish, the *fpuBusy* signal will be asserted in the following phi3 to indicate that the FPU is busy doing the comparison and the *fpuBrT_F* signal is therefore not valid. Since no instruction can be executed until the FPU compare is done (see Section 3.2), the CPU pipeline is suspended until the FPU disasserts the *fpuBusy* line to indicate that the comparison is done and *fpuBrT_F* is valid. There are several things worth noticing in Figure 7.

1. During the CPU pipeline suspension, the *fpuNewInstr* signal is not asserted and the instruction I1 is broadcast repeatly to the FPU.
2. The address of instruction I2 is a function of the *fpuBrT_F* signal. Depending on the signal *fpuBrT_F*, I2 can either be I1 + 4 (each instruction

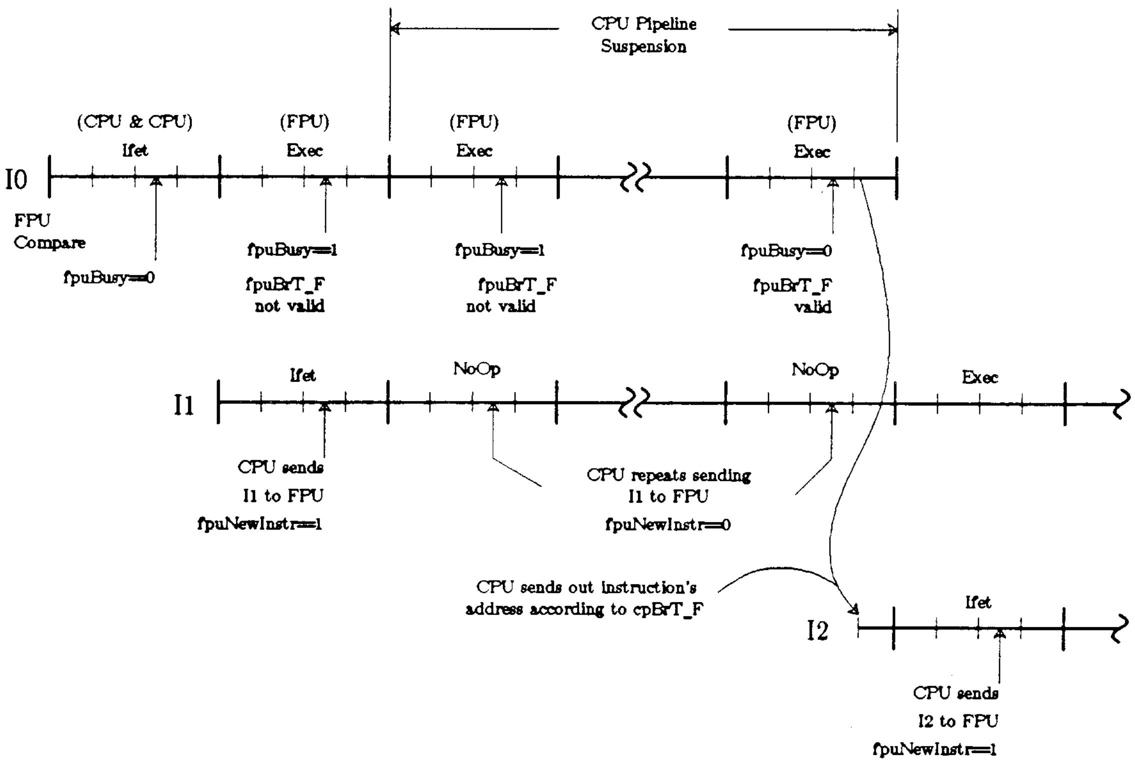


Figure 8. FPU compare instruction timing.

is 4 bytes long) or be the target address provided by I0.

3. During the CPU pipeline suspension, the FPU operation is not suspended (*fpuSuspend* remains unasserted throughout), otherwise a deadlock results.

3.3.3. Timing of Other FPU Instructions

The timing of an FPU instruction, which is neither an FPU load, an FPU store, nor an FPU compare instruction is shown in Figure 9. In this figure, I0 is the FPU instruction that caused the FPU to be busy (*fpuBusy* = 1).

Figure 9 shows the more complicated case in which the *fpuParallel* bit is on (parallel mode - see Section 3.2) and I1 is another FPU instruction. Since the CPU reads the *fpuBusy* signal during phi3 of I1's Ifet cycle, in order to stop I1 in time, the FPU must assert this signal by phi3 of the 1st execution cycle of I0. Furthermore, notice the following:

1. Both I0 and I1 have more than one Exec cycle. This is possible because these Exec cycles are performed by the FPU.
2. During the CPU pipeline suspension, the *fpuNewInstr* signal is disasserted and the instruction I1 is broadcast repeatly to the FPU.

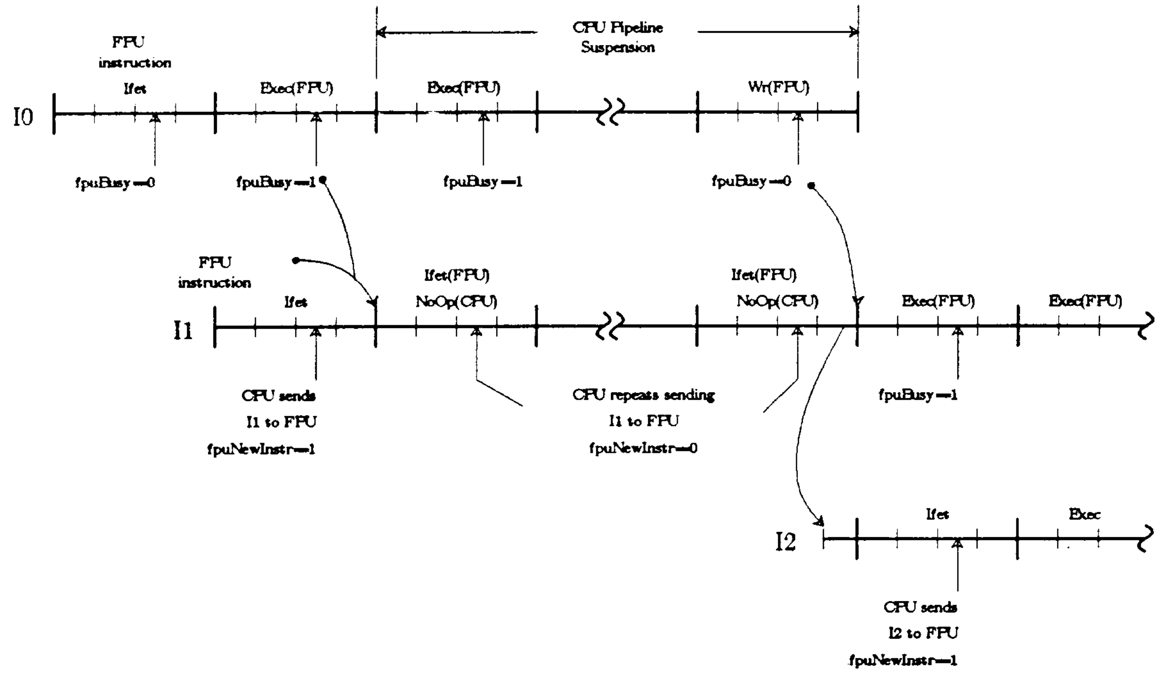


Figure 9. General FPU instruction timing.

3. During the CPU pipeline suspension, the FPU operation is not suspended (*fpuSuspend* remains unasserted throughout), otherwise a deadlock results.
4. Instruction I2, which is not an FPU instruction, is allowed to continue (does not cause the CPU pipeline to suspend) because it is assumed that the *fpuParallel* bit is asserted and I1 is not an FPU compare instruction.

3.4. How Special Cases Are Handled By This Interface

3.4.1. CPU Cache Reference Followed By FPU Cache Reference

A CPU cache reference followed by an FPU cache reference (or any combination of the two) can be handled by the FPU as long as the FPU monitors the dataValid and *fpuSuspend* signals. This is illustrated in Figure 10.

Figure 10a shows the simplest case where neither instruction I0 nor I1 causes a cache miss. The FPU will not be mislead by the first assertion of the dataValid signal as long as the FPU knows when to start checking the dataValid signal. This is a requirement even for the simplest case when an FPU load or store instruction is not preceded by any other cache reference instruction. As indicated in Figures 5, 6, and 7 (I1 in Figure 7), without any pending cache miss, the FPU does not start checking the dataValid signal until two cycles (third cycle overall) after it receives the FPU load or store instruction.

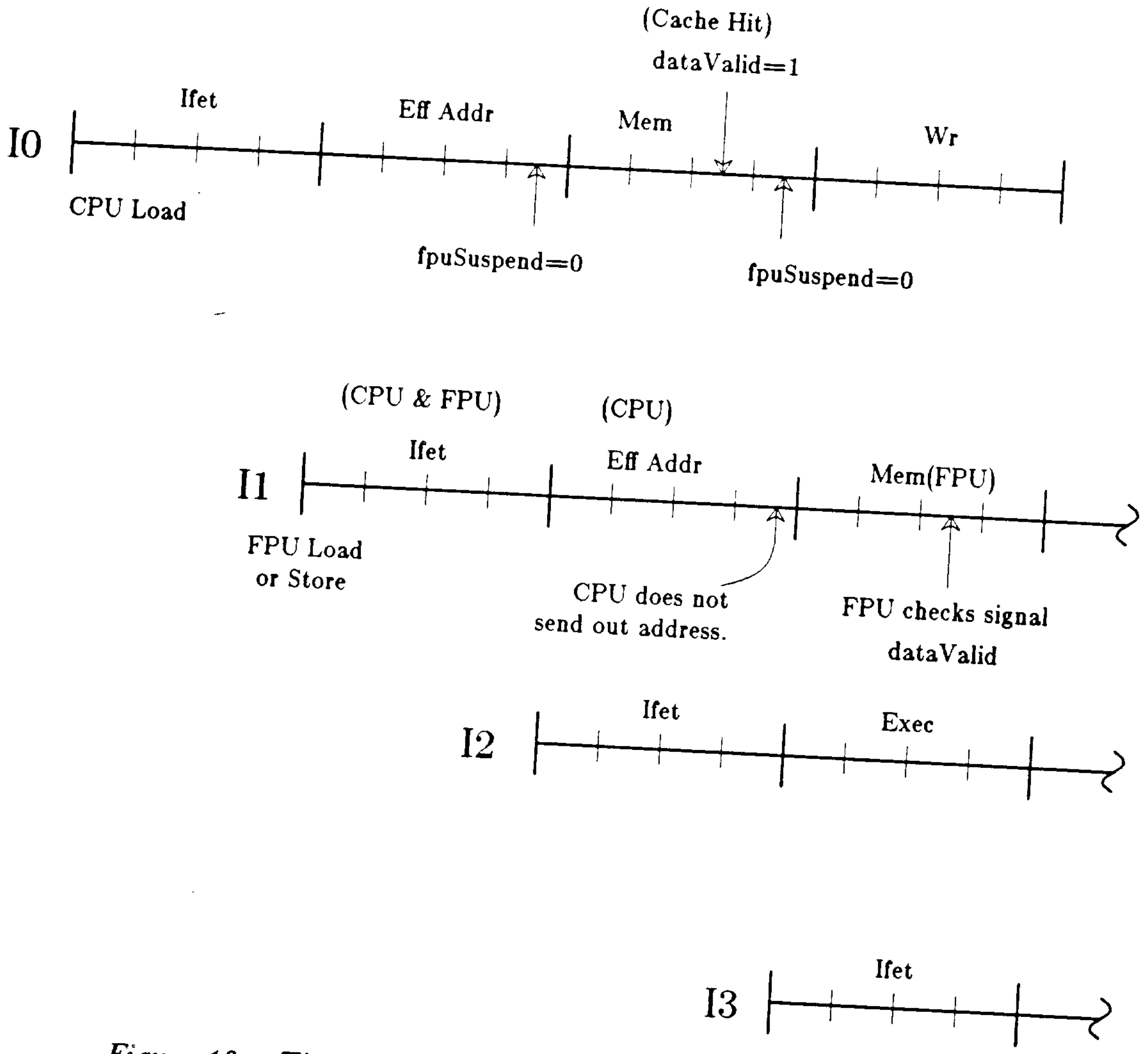


Figure 10a. Timing for CPU load followed by FPU load or store.

Figure 10b shows the more complicated case where the CPU load instruction causes a cache miss. Due to the pending cache miss caused by the CPU load, the FPU cannot start looking at the dataValid line until two cycles after it receives the FPU load instruction. This problem is solved easily by using the *fpuSuspend* signal. The CPU pipeline is suspended due to the cache miss caused by I0. Since the CPU pipeline suspension is caused by I0 (which is not an FPU instruction) the *fpuSuspend* signal is asserted (see Section 3.1).

By comparing I1 in Figure 10a and I1 in Figure 10b, it is obvious that the *fpuSuspend* signal stops the FPU from getting into I1's memory access cycle (Mem) until the pending cache miss is serviced. Since I1 is not in its memory access cycle, the FPU does not look at the dataValid signal and will not be mislead by the first assertion of the dataValid signal.

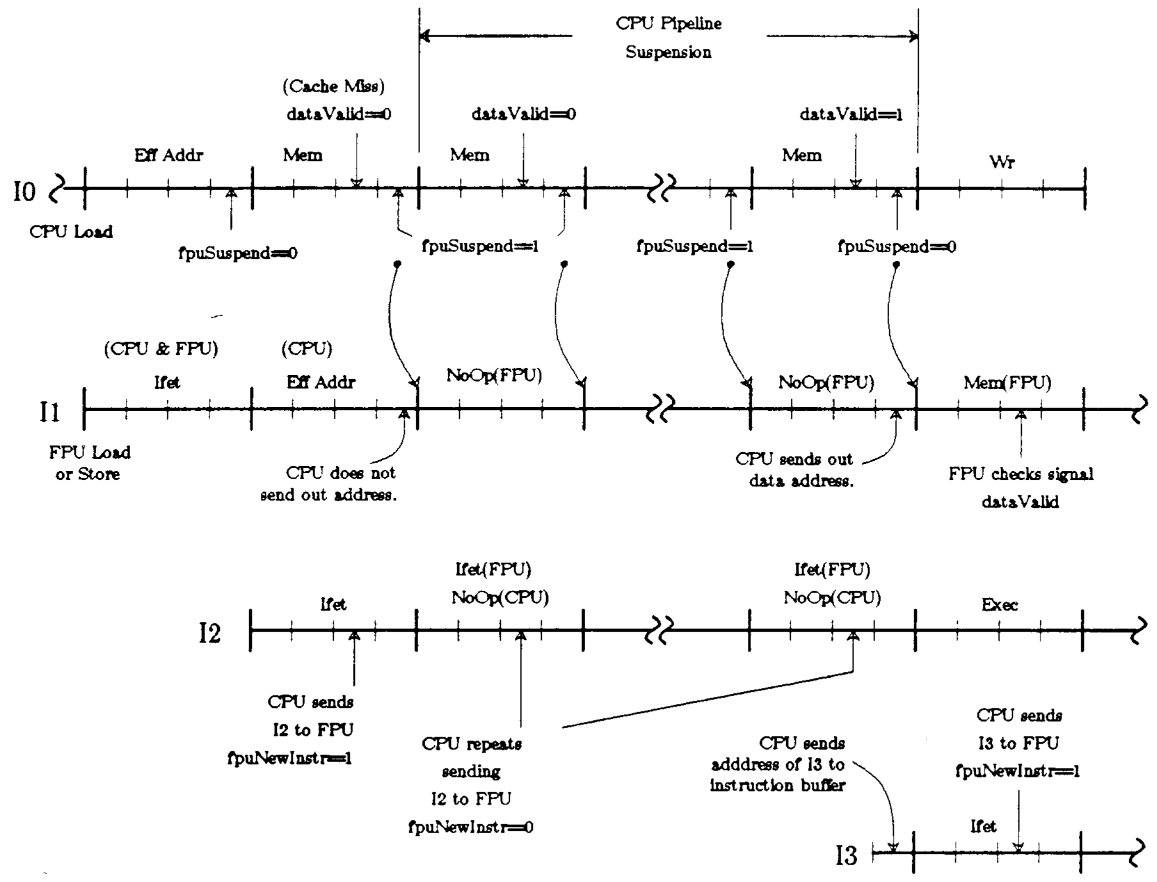


Figure 10b. Timing for CPU cache miss followed by FPU load or store.

Finally, as pointed out in Section 3.3.1, if I1 in Figure 10b causes a cache miss, the CPU pipeline will again be suspended, but the *fpuSuspend* signal won't be asserted. The FPU controller must repeat the memory access cycle of I1 until valid data is received (dataValid = 1) and suspend all all other activities in the FPU.

By using the *fpuSuspend* signal, the FPU does not have to understand the CPU load or store instruction. Also, the FPU does not have to match each assertion of dataValid with each cache reference. All the FPU has to do is mind its own business and keep an eye on the *fpuSuspend* signal. That is:

1. If *fpuSuspend* is asserted, suspend FPU's activities, (Note: The FPU does not have to suspend everything as soon as it receives in which certain instructions, if they happen to be FPU instructions, can be killed (see Section 3.4.2) if a trap occurs.) otherwise,
2. Start looking for valid data by monitoring the dataValid signal two cycles after it receives the FPU load or store instruction.

3. If dataValid is not asserted, repeat the memory access cycle and suspend all FPU activities that are related to FPU instructions received after the FPU memory access instruction until dataValid is asserted.

Figure 10c shows the case in which the FPU load or store instruction results in a page fault or a bus fault (see the discussion on the cache controller chip). The FPU need not monitor the pageFault or busFault lines because the FPU knows a pageFault or busFault has occurred when it receives the internal (internal to the CPU) TRAP_CALL instruction, The only thing that can cause a trap when the CPU pipeline is suspended is a page fault or bus fault. Figure 10c only shows why the FPU does not have to monitor the fault lines. For a detailed discussion of what the FPU has to do after receiving the TRAP_CALL instruction, please refer to Section 3.4.2.

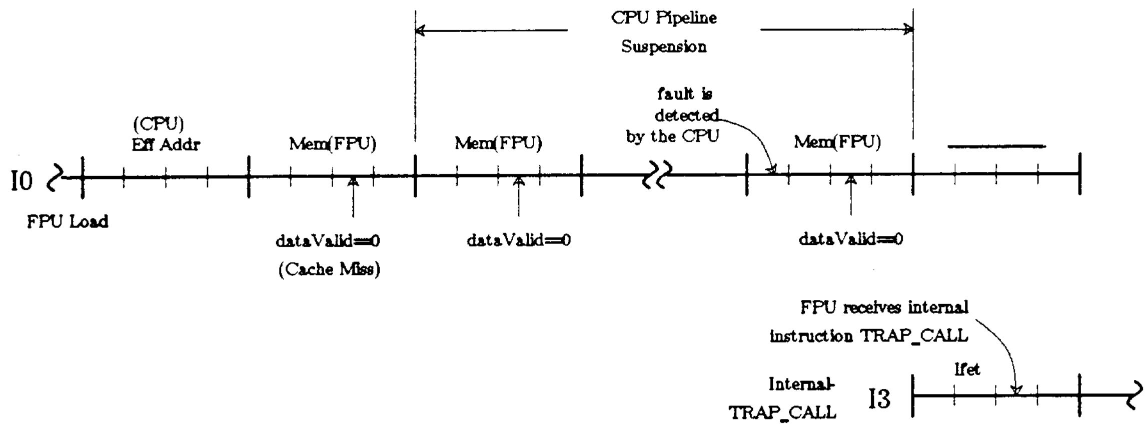


Figure 10c. Timing of FPU cache miss which results in a fault.

3.5. FPU's Response to CPU Trap

The FPU knows the CPU is taking a trap when it receives the internal (internal to the CPU) TRAP_CALL instruction. This section discusses the procedure the FPU must follow after it receives the TRAP_CALL instruction.

Inside the CPU, there is a slight difference between a trap caused by a page fault or bus fault and a trap caused by something else: a page fault or bus fault causes a trap during CPU pipeline suspension while the other conditions can only cause a trap when the CPU pipeline is not suspended. How the FPU should response to TRAP_CALL depends on whether the page or bus fault is caused by an FPU cache access, or by a CPU cache access. Therefore, there are three different cases to consider:

1. The CPU takes a trap because the CPU cache reference results in a page or bus fault,

2. The CPU takes a trap because the FPU cache reference results in a page or bus fault,
3. The CPU takes a trap for reasons other than 1 or 2 above. This will be referred to as "regular trap" for the rest of this section.

Case 3 described above is the simplest and is illustrated in Figure 11a. When the FPU receives the internal instruction TRAP_CALL, all it has to do is check whether the last instruction it received before receiving TRAP_CALL (in Figure 11a, this is I1) was an FPU instruction. If that is the case, then that instruction must be killed. Otherwise, no action has to be taken.

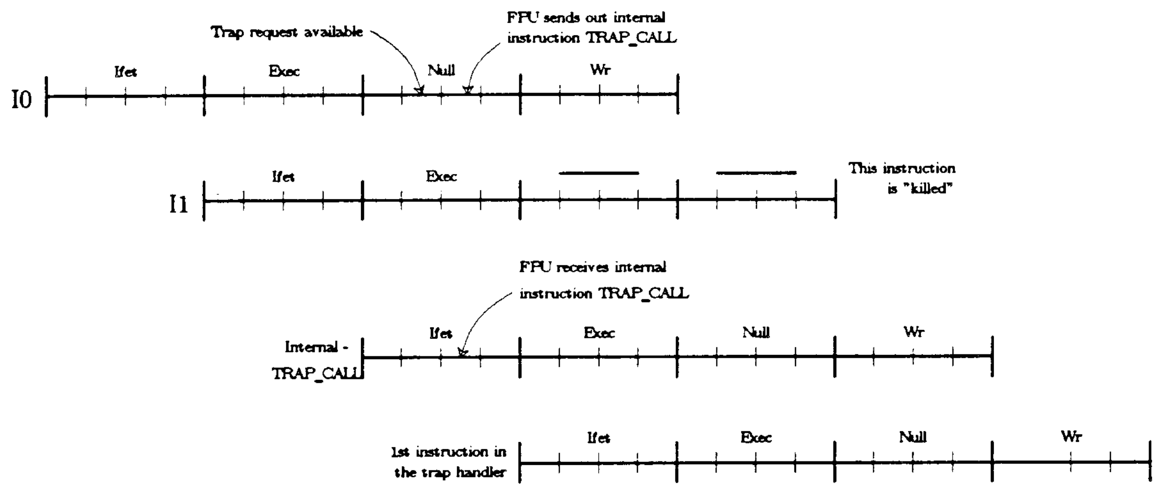

Trap request available
FPU sends out internal instruction TRAP_CALL
I0
Ifet
Exec
Null
Wr
I1
Ifet
Exec
This instruction is "killed"
FPU receives internal instruction TRAP_CALL
Internal - TRAP_CALL
Ifet
Exec
Null
Wr
1st instruction in the trap handler
Ifet
Exec
Null
Wr

Figure 11a. A regular TRAP timing sequence.

Figure 11b illustrates Case 1 in which the CPU takes a trap because a CPU cache reference results in a page or bus fault. As discussed in Section 3.1, whenever the CPU pipeline is suspended for a CPU instruction, FPU operation is also suspended. In other words, during CPU pipeline suspension, no FPU instruction can complete its execution even if the CPU-FPU pair operates in parallel mode. I1, which is shown in Figure 11b as a FPU instruction, and thus won't finish its execution during the CPU pipeline suspension and can be killed easily when FPU receives the TRAP_CALL instruction. Any FPU instruction received before I1, however, must be allowed to finish.

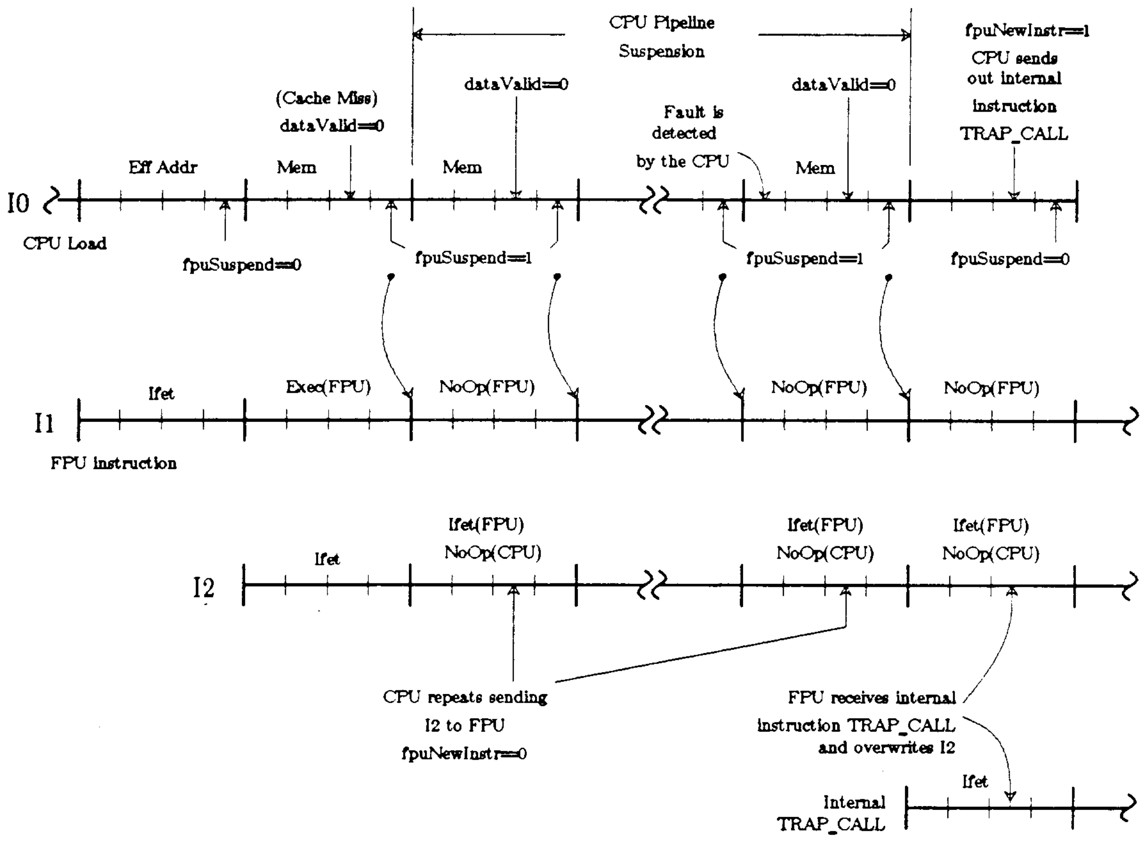


Figure 11b. Timing of trap caused by CPU page or bus fault.

Notice that in Figure 11b the *fpuSuspend* signal is still asserted when the FPU receives the TRAP_CALL instruction. As discussed in Section 3.1, as long as *fpuSuspend* is asserted, the Ifet cycle of the I2 is being repeated by the FPU. Consequently, when the FPU receives the TRAP_CALL, the FPU still considers I2 to be in its Ifet cycle. As far as the FPU is concerned, instruction I2 is overwritten by the TRAP_CALL instruction. The last instruction the FPU received before receiving the TRAP_CALL is, in effect, still I1. Therefore the same hardware, which is used in Case 1 to kill I1 in Figure 11a, can be used here for Case 1 as long as the hardware ignores the empty cycles when the FPU is suspended by the *fpuSuspend* signal.

Figure 11c illustrates Case 2 in which the CPU takes a trap because the FPU cache reference results in a page or bus fault. The CPU pipeline is suspended due to the cache miss, and the FPU must suspend all execution activities. Consequently, during an FPU cache miss, no FPU instruction can complete its execution even if the CPU-FPU pair operates in parallel mode. This is discussed in detail in Section 3.3.2. I1, which is shown in Figure 11c as a FPU instruction, thus won't finish its execution during an FPU cache miss and can be killed easily when the FPU receives the TRAP_CALL instruction.

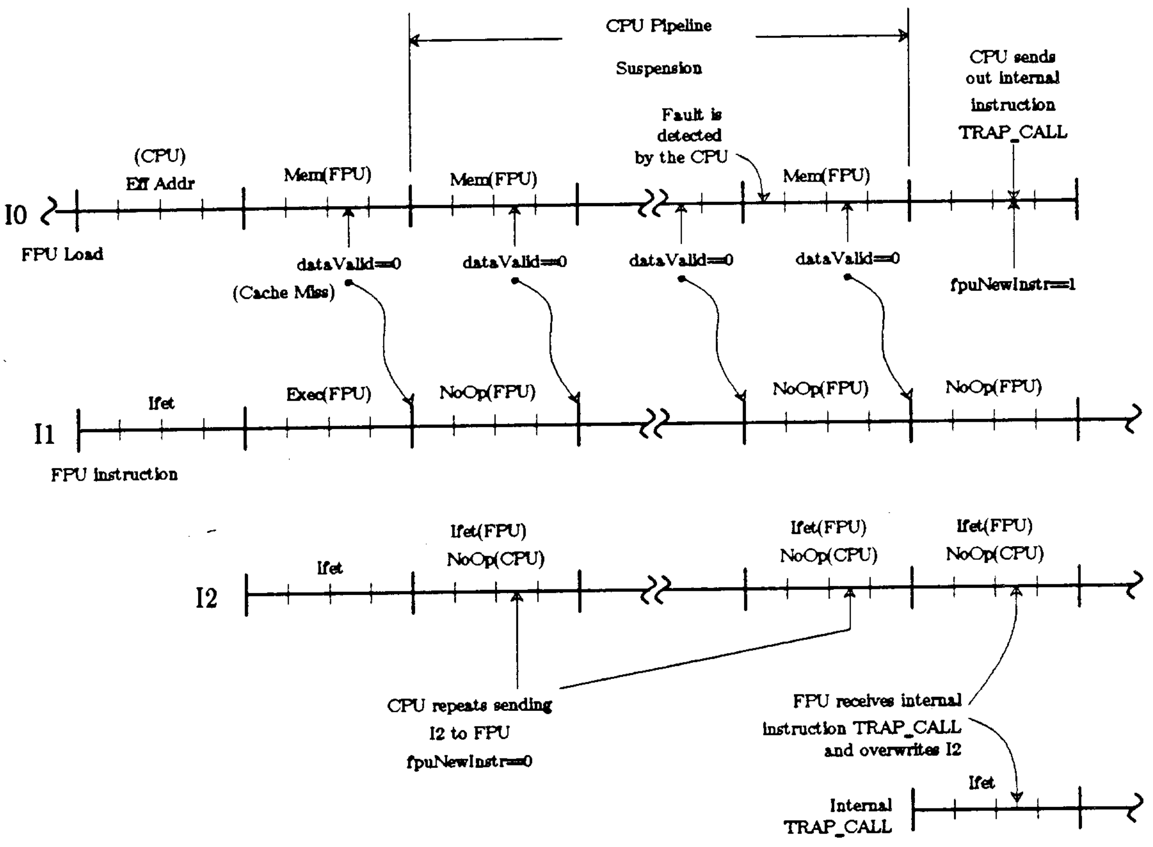


Figure 11c. Timing of trap caused by FPU page or bus fault.

Since the FPU is not looking at the page fault or bus fault line, it does not know a fault has occurred until the FPU receives the TRAP_CALL instruction. Therefore in Figure 11c, when the FPU receives the TRAP_CALL instruction, the FPU still considers I2 is in its Ifet cycle. Once again, the TRAP_CALL instruction has overwritten instruction I2 and I1 is still, in effect, the last instruction the FPU received before receiving the TRAP_CALL. Therefore the same hardware, which is used Case 1 to kill I1 in Figure 11b, can be used here for Case 2 as long as the hardware ignores the empty cycles when most of the FPU activities are suspended due to an FPU cache miss.

3.6. Internal MISS Instruction

Besides the internal TRAP_CALL instruction, the FPU must also understand the internal MISS instruction, which is used to handle instruction buffer misses. Upon receiving this instruction, the FPU should not do anything. Thus, the FPU must treat this instruction the same way it treats all others CPU (integer) instructions.

3.7. Exception Handling in the CPU-FPU

Figure 12 shows a most critical case, where two FPU instructions are in series and the first one causes an exception. In this case, the CPU must respond to the exception before the CPU updates the FpuPC incorrectly. When the CPU responds to the exception by taking a trap, the FpuPC must contain the address of I0 - the FPU instruction that caused the exception.

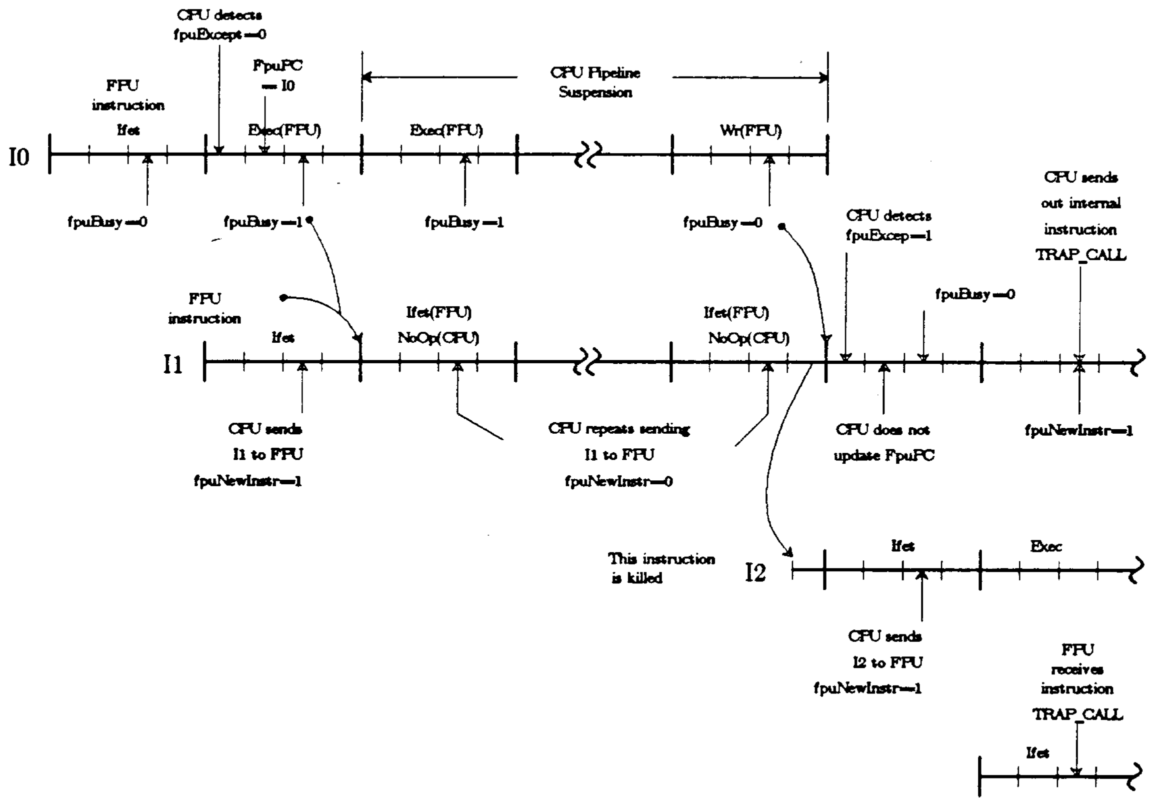


Figure 12. Timing of FPU exception.

Figure 12 should be compared with Figure 9, which shows the case where two FPU instructions are in series but none causes an exception. The following two assumptions are made in Figure 12:

1. The CPU looks at the *fpuExcep* line only when it encounters another FPU instruction.
2. The CPU, if no exception has occurred, updates the FpuPC during phi2 of the first execution cycle of an FPU instruction. (Note: The FpuPC is likely to be a master slave register. The master will latch in every instruction address. The slave, however, is updated only if the instruction turns out to be

a FPU instruction.)

In Figure 12, when the CPU encounters the FPU instruction I1, the CPU checks for a pending FPU exception. In Figure 12, the pending exception is caused by I0. The FpuPC must therefore contain the address of I0 when exception is serviced. To prevent the CPU from updating the FpuPC to the address of I1, the CPU must read the *fpuExcep* during phi1 of I1's first execute cycle. I1's address is saved automatically by the internal TRAP_CALL instruction because the CPU considers I1 the instruction that caused the trap. Instruction I2 is killed because it is the instruction immediately following the "trapped" instruction.

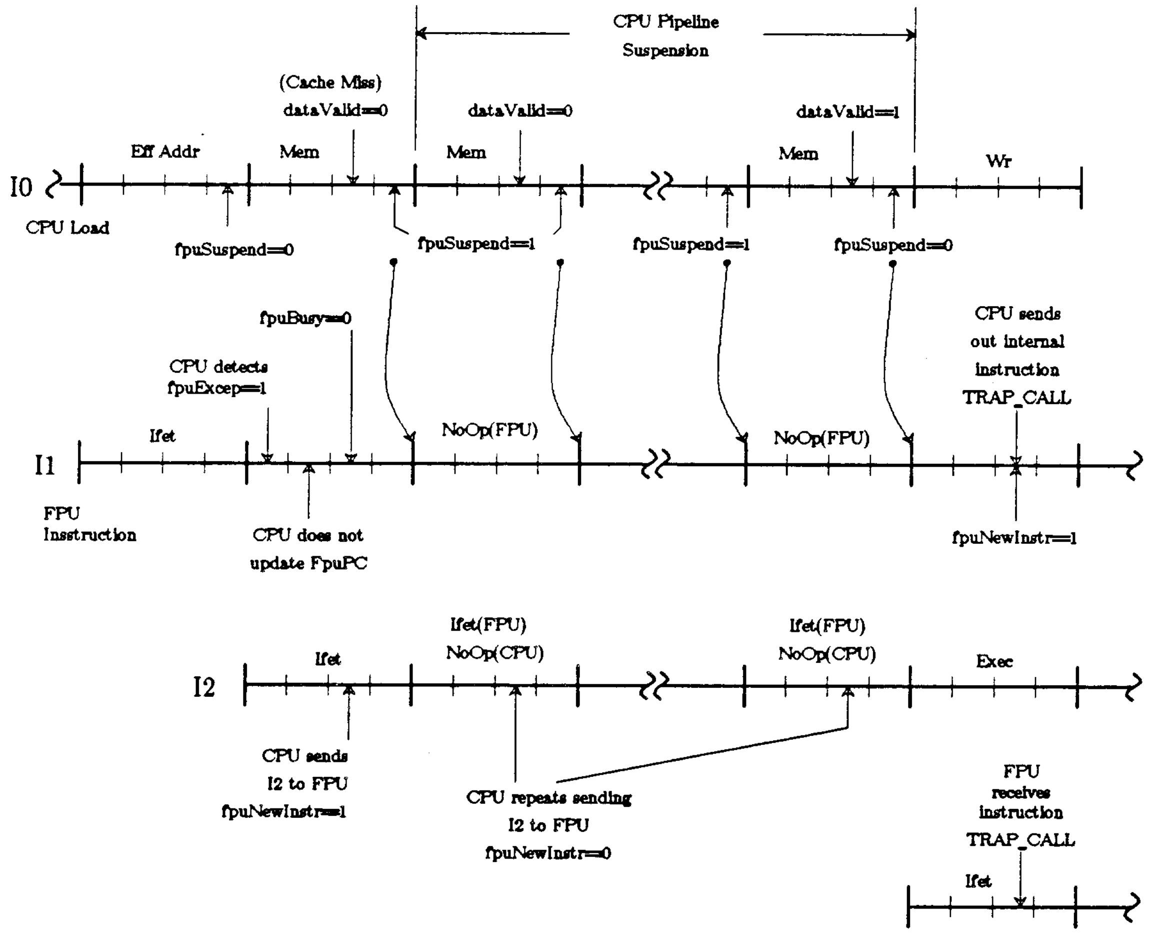


Figure 13. Timing of CPU cache miss followed by FPU exception.

A very interesting question can be raised in Figure 11b, and 11c: what happens if I1 is an FPU instruction and there is a pending exception caused by an earlier FPU instruction. As explained in Section 3.4.2, the I1 is simply killed. Therefore, instruction I1 will not detect the pending exception because the page or bus fault has killed I1. The pending exception will either be detected in the trap routine (if there is at least one FPU instruction inside the trap handle)r or will be detected when I1 is re-executed after the trap. Finally, if I0 in Figure 11b and 11c does not cause a page or bus fault, I1 will still cause a trap. However, that trap will not occur until the CPU pipeline suspension is over. This is illustrated in Figure 13.

4. REFERENCES

[Han85] P. M. Hansen, Coprocessor Architectures for VLSI, Phd Qualifying Examination Proposal, Berkeley, CA 94710, May 3, 1985.

[KEP] R. H. Katz, S. Eggers, C. Perkins, R. Sheldon and D. Wood, Implementing a Cache Consistency Protocol, 12th Annual Symposium on Computer Architecture, Boston, MA, (June 1985)..

Table 1. SPUR FPU Coprocessor Instructions

ARITHMETIC OPERATIONS				
Instruction Syntax		Instruction Semantics		
FADD	Rd,Rs1,Rs2	CP Rd	<-	CP Rs1 + CP Rs2
FSUB	Rd,Rs1,Rs2	CP Rd	<-	CP Rs1 - CP Rs2
FMUL	Rd,Rs1,Rs2	CP Rd	<-	CP Rs1 * CP Rs2
FDIV	Rd,Rs1,Rs2	CP Rd	<-	CP Rs1 / CP Rs2
FABS	Rd,Rs1,0	CP Rd	<-	CP Rs1 with sign = 0
FNEG	Rd,Rs1,0	CP Rd	<-	CP Rs1 with inverted sign
FLOAT	Rd,Rs1,0	CP Rd	<-	convert to extended (CP Rs1)
FIX	Rd,Rs1,0	CP Rd	<-	convert to integer (CP Rs1)
CVTS	Rd,Rs1,0	CP Rd	<-	convert to single (CP Rs1)
CVTS_UNRND	Rd,Rs1,0	CP Rd	<-	convert to single (CP Rs1)
CVTD	Rd,Rs1,0	CP Rd	<-	convert to double (CP Rs1)
CVTD_UNRND	Rd,Rs1,0	CP Rd	<-	convert to double (CP Rs1)

LOAD COPROCESSOR REGISTERS				
Instruction Syntax		Instruction Semantics		
LD_SGL	Rd,Rs1,RC	CP Rd	<-	M [(Rs1 + RC)
LD_DBL	Rd,Rs1,RC	CP Rd	<-	M [(Rs1 + RC)
LD_EXT1	Rd,Rs1,RC	CP Rd	<-	M [(Rs1 + RC)
LD_EXT2	Rd,Rs1,RC	CP Rd	<-	M [(Rs1 + RC)
LD_INT	Rd,Rs1,RC	CP Rd	<-	M [(Rs1 + RC)

STORE COPROCESSOR REGISTERS				
Instruction Syntax		Instruction Semantics		
ST_SGL	Rs2,Rs1,SC	CP Rs2	->	M [(Rs1 + SC)
ST_DBL	Rs2,Rs1,SC	CP Rs2	->	M [(Rs1 + SC)
ST_EXT1	Rs2,Rs1,SC	CP Rs2	->	M [(Rs1 + SC)
ST_EXT2	Rs2,Rs1,SC	CP Rs2	->	M [(Rs1 + SC)
ST_INT	Rs2,Rs1,SC	CP Rs2	->	M [(Rs1 + SC)

CPU <—> COPROCESSOR REGISTERS				
Instruction Syntax		Instruction Semantics		
FROM_CP	Rd,Rs1,Rs2	CP RD.data	<-	CP Rs2<31:00>
TO_CP	Rs2,Rs1,Rd	Rs2.data	->	CP Rd

COMPARE AND BRANCH ON COPROCESSOR CONDITIONS		
Instruction Syntax		Instruction Semantics
CP_CMP_BR_D-	cond,Rs1,Rs2,offset	if (CP Rs1 cond CP Rs2)
CP_CMP_BR_TRUE	cond,Rs1,Rs2,offset	if (CP Rs1 cond CP Rs2)
CP_CMP_BR_FALSE	cond,Rs1,Rs2,offset	if (CP Rs1 cond CP Rs2)
CP_CMP_TRAP	cond,Rs1,Rs2	if (CP Rs1 cond CP Rs2)
CP_CMP_BR_xxx	bit<24>	Invalid Exception if relation unordered
CP_CMP_TRAP	bit<23>	unordered
	bit<22>	less than
	bit<21>	equal
	bit<20>	greater than

Table 2. SPUR FPU Coprocessor Instruction Cycle Times

ARITHMETIC OPERATIONS		
Instruction		Cycles (operation only)
FADD	Rd,Rs1,Rs2	3
FSUB	Rd,Rs1,Rs2	3
FMUL	Rd,Rs1,Rs2	6
FDIV	Rd,Rs1,Rs2	9
FABS	Rd,Rs1,0	1
FNEG	Rd,Rs1,0	1
FLOAT	Rd,Rs1,0	1
FIX	Rd,Rs1,0	1
CVTS	Rd,Rs1,0	3
CVTS_UNRND	Rd,Rs1,0	3
CVTD	Rd,Rs1,0	3
CVTD_UNRND	Rd,Rs1,0	3

Draft

SpurBus Specification

Garth Gibson
Computer Science Division
Electrical Engineering and Computer Science
University of California, Berkeley

Table of Contents

Table of Figures

Table of Tables

1. Introduction

This documents specifies the design of a shared system bus for a synchronous multiprocessor based on shared memory. The bus is called the SpurBus and it is part of the SPUR, Symbolic Processing Using RISCs, project.

SPUR is a multiprocessor workstation. Each processor is a RISC, Reduced Instruction Set Computer, with a tagged architecture for supporting the LISP programming environment and an instruction buffer to reduce the instruction traffic across the chip borders. Each node in the multiprocessor contains a processor, a large cache, a floating point coprocessor and a cache controller. The caches are kept consistent by a "snooping cache protocol" [Katz85a] implemented in the cache controller and across the bus.

The goal of SPUR is to provide a low cost, fast uniprocessor with additional processors available for research efforts into shared memory multiprocessing. The number of processors is small (6 to 10) so that a low cost interconnect, the system bus, will be able to supply the required memory bandwidth. The choice of a system bus is influenced by the focus of our research: a multiprocessor LISP programming environment. The system bus is well-understood, flexible, reliable, a convenient point of serialization for synchronization and monitoring and, most important, its design can be borrowed from existing microcomputer system buses.

The Texas Instruments NuBus was selected for our basis design. The NuBus is described in detail in this document. A subset of it is extended to provide for the cache coherency and virtual memory mechanisms in SPUR. The extensions are transparent to the NuBus subset because the SPUR project will be using commercial memory and I/O boards using the NuBus protocol. The resulting design is called the SpurBus.

This specification is organized so that the SpurBus is presented as modifications to the NuBus. Section 2 presents the NuBus in detail including features that the SpurBus does not support. These absent features are discussed in section 3. The new features of the SpurBus are introduced and detailed in section 4. Finally the details of the mechanical backplane are presented in section 5.

2. NuBus Specifications

The description of the NuBus in this document is drawn from the Texas Instruments document, "NuMachine NuBus Specification," part number TI-2242825-0001, published in 1983. Both documents are subject to change without notice.

2.1. Overview

Figure 2.1 shows an overview of a NuBus configuration. Each board has a NuBus interface connected to the backplane. The NuBus provides each interface with a group of data transfer lines (address/data and control), a group of arbitration lines, a central clock and an unique identifier.

Some of the major characteristics of the NuBus are:

synchronous
: All events are synchronized by a central clock; however, the number of cycles between events is usually unspecified. In this way events are asynchronous in bus cycle units.

bandwidth
: The 10 MHz nominal cycle time coupled with block transfers gives a peak transfer rate of 37.5 Mbytes/sec.

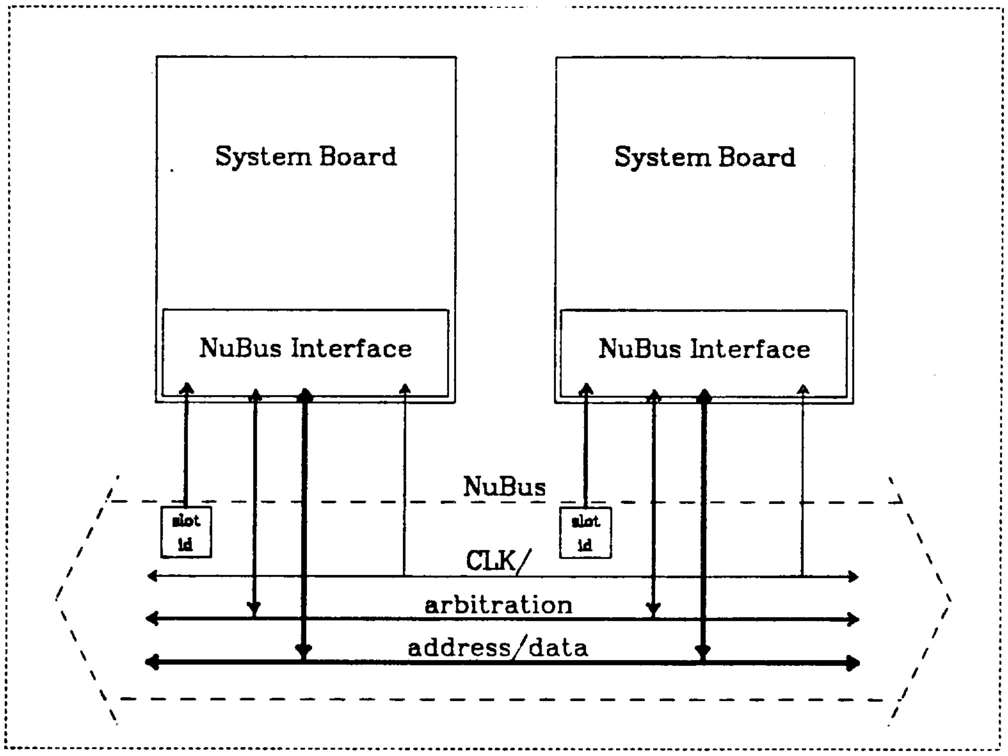


Figure 2.1: NuBus Block Diagram
The NuBus provides 32 multiplexed address/data lines, some control, arbitration, clocking and unique identifiers to each backplane slot.

simple
: The only operations on the bus are reads and writes. Interrupts and IO transfers are memory mapped into a large physical address space (4 GBytes)

small consumption of backplane lines
: Multiplexing address and data lines helps limit signal lines to a total of 49. With power and ground lines included, only one 96 pin backplane connector is required by the NuBus.

distributed configuration
: Each of the 16 possible bus ports are provided with backplane identification. This eliminates board "DIP" switches and allows distributed, parallel arbitration rather than daisy chain bus grants.

multiprocessor support
: The fair arbitration policy and the capability to lock the bus across distinct transfers provides basic and effective multiprocessor support.

2.2. Connector Pins Description

All NuBus signals use negative logic; that is, the asserted state has a low voltage on the line and the unasserted state has a high voltage on the line. This is indicated in Table 2.1 by the suffix "/" on each signal name. To try to reduce confusion, values referred to in this document will be given as high or low and will represent the voltage of the negative logic signal. This means that high value is a logic zero and a low value is a logic 1. This document will also use the terminology "asserted" and "unasserted" to indicate movement of data. An asserted signal will usually be a signal driven low; however, the address/data lines are asserted whenever they are driven high or low.

The NuBus defined backplane lines can be grouped as Clock, Control, Address/Data, Arbitration, Parity, Utility, Reserved and Power and Ground. Table 2.1 shows these groups, their member signals, the connector pins they require and the electrical protocol of each. The Clock is described in the next section, Utility is describe in its own section, Control, Address/Data and Parity are described in the section on data transfer and Arbitration is described in its own section. The electrical protocols are input-only, tri-state (drive as output, receive as input and short circuited) and open-collector (value is the logic AND of all ports driving the signal as output).

2.3. System Timing

The NuBus clock, shown in Figure 2.2, has a nominal cycle time of 100 nsec. It has a 75% duty cycle; that is, 75 nsec unasserted and 25 nsec asserted phases. The low to high transition (clock assertion edge) triggers changes in bus signals. The high to low transition (clock deassertion edge) triggers bus signal sampling. The duty cycle of the central clock is asymmetric to provide increased time for propagation and setup time while preserving enough time between signal sample and change to avoid skew problems. Figure 2.3 shows the setup, hold, propagation, assertion and release times relative to the clock phases.

2.4. Physical Memory Map

The NuBus defines a large physical address space (4 GBytes). Each bus port, called a slot, has some of this memory mapped to it. In total the top 16th (256 Mbytes) of the physical address space, called slot space, memory maps the 16 possible bus ports. Figure 2.4 shows this organization of physical memory as it applies to a board in slot 13.

The board at each bus port interprets reads and writes to its slot without constraint. Processor boards desiring interrupt capabilities associate interrupt registers with slot space addresses. They may also provide debug or status information through slot space addresses. Memory boards may provide their RAM through slot space and IO boards can provide control, status and data ports

GROUP	SIGNAL	# OF PINS	PROTOCOL
Clock	CLK/	1	input-only
Control	START/	1	tri-state
	ACK/	1	tri-state
	TM0/	1	tri-state
	TM1/	1	tri-state
Address/Data	AD<31..0>/	32	tri-state
Parity	SP/	1	tri-state
	SPV/	1	tri-state
Arbitration	ARB<3..0>/	4	open-collector
	RQST/	1	open-collector
Utility	RESET/	1	open-collector
	ID<3..0>/	4	input-only
	Total Signals	49	
Reserved	RSVD/	1	
Power/Ground	+5	11	
	-5	8	
	+12	2	
	-12	2	
	GND/	23	
	Total Lines	96	

Table 2.1: NuBus Backplane Lines
The NuBus defines the use of one 96 pin backplane connector for 49 signal lines plus power and ground. The protocol of a signal refers to its driving characteristics.

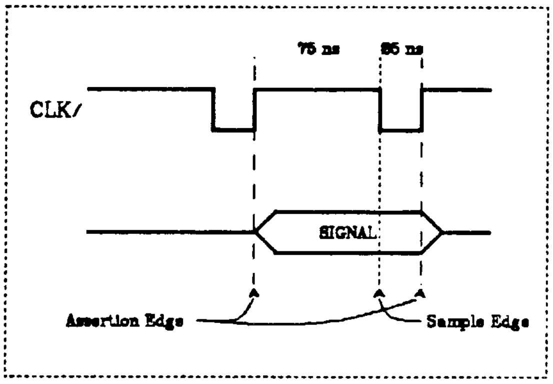


Figure 2.2: System Clock
The NuBus has a synchronous design. The central clock, CLK/, has a 75% duty cycle separating its sample and assertion edges.

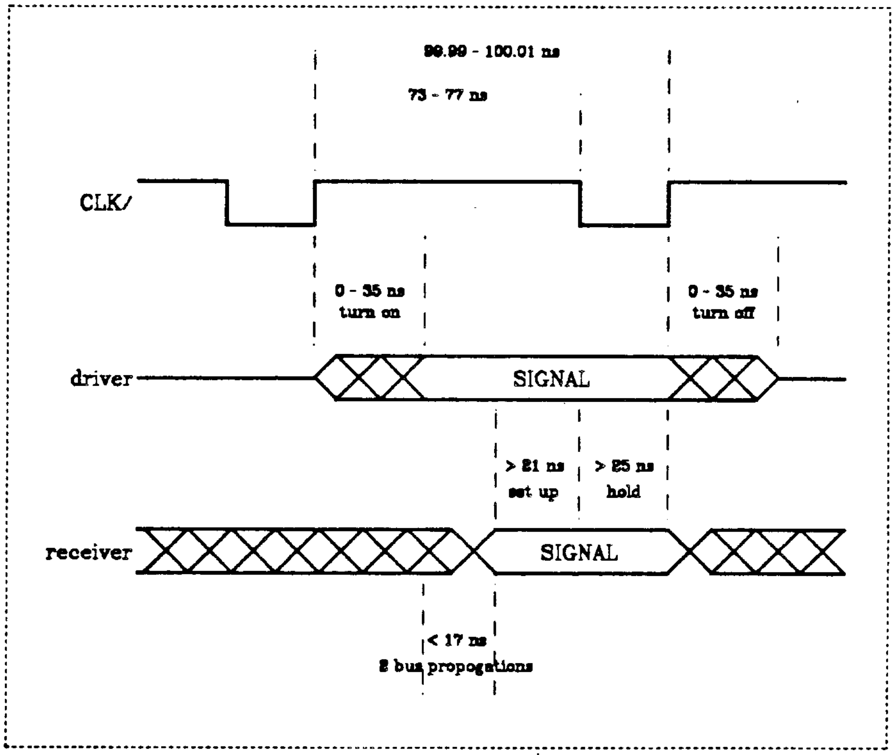


Figure 2.3: Signal Timing
The selection of the 75% duty cycle provides for greater relative emphasis on propagation and setup times while preserving freedom from bus skew problems.

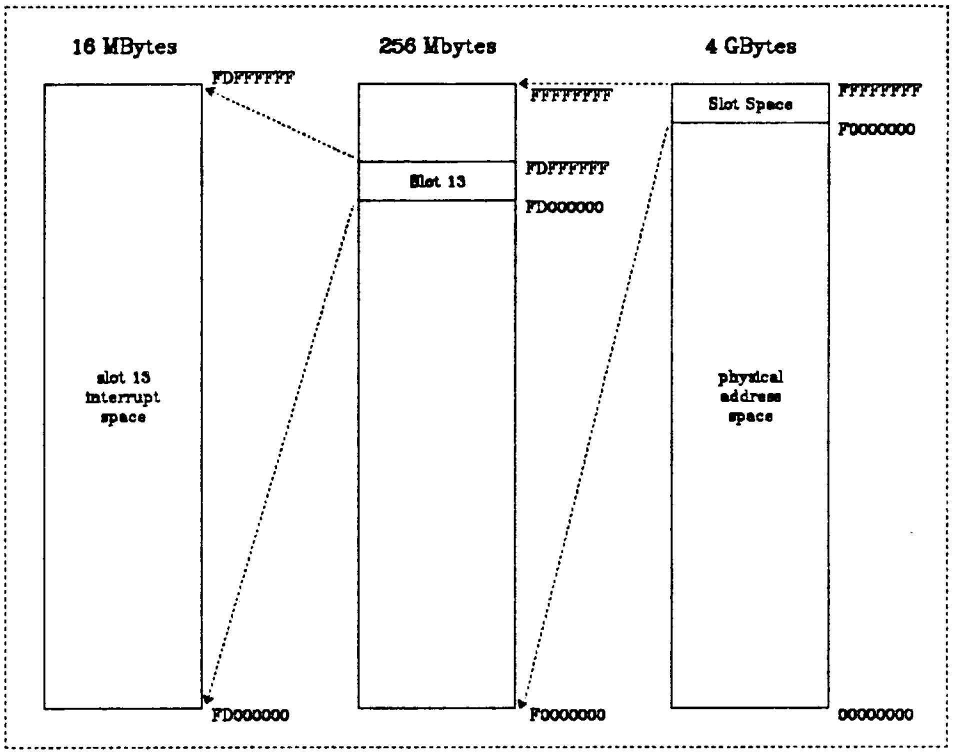


Figure 2.4: Physical Memory Layout
NuBus physical memory maps the top 256 MB to its 16 slots, 16 MB each. This slot space is used to memory map interrupt and I/O transfers.

through slot space. All boards should provide identification (board type, revision, name, etc.) and configuration control registers through slot space. The remaining physical memory (non-slot space) is not constrained by the NuBus specification. Typically, memory boards will allow their RAM addresses to be remapped to arbitrary locations (if the RAM of different memory boards is required to be contiguous in physical memory).

2.5. Utility Signals

RESET/

This open-collector signal is used to synchronize the state of the system. If it is asserted for one cycle all NuBus interfaces are initialized. This is referred to as a Bus Reset. If it is asserted for more than one clock period, then it should be asserted for at least a millisecond (10000 cycles) and the entire system is initialized to the power-up state. This is called a System Reset.

ID<3..0>/

These four bits name, in binary, the backplane slot that the reading board occupies. In addition to uniquely naming each slot these lines are used for NuBus arbitration (see section 2.7). These lines are also used to determine whether a physical address is mapped to the local slot (addresses F(ID)XXXXXX, X any hex digit are mapped to the local slot).

2.6. Data Transfer

The NuBus defines read and write data transfers. These may move bytes, halfwords, words, doublewords, quadwords, octwords and doubleoctwords. All units must be aligned in physical memory according to their size (more precisely, starting address modulo size is zero). For the small units, the byte and halfword units, the data is returned as if the word enclosing them was read. It is up to the board or processor receiving the unit to justify the data as it wishes. For these sizes and the word size, one transfer across the backplane is all that is needed. These are referred to as word reads and word writes. For the larger sizes, multiple transfers across the backplane are needed. These are referred to as block transfers. This section describes word and block transfers in detail.

Before a NuBus interface may begin a data transfer it must become the bus master. This is the process of arbitration. Once a master, the interface must still wait for the possible current transfer to complete. When it has obtained mastery and seen the end of the last master's transfer then the interface may initiate a new transfer.

To avoid system deadlock when board or backplane errors occur, the system is required to provide a "watchdog timer". This timer is started at the beginning of each transaction, reset each time a word is successfully transferred and cancelled at the end of a transaction. If it fires at any time during a transfer, then an error is very likely and the watchdog generates a transfer termination in error.

2.6.1. Control Signals

The signals START/, ACK/, TM0/, TM0/, AD<5..0>/ and SPV/ are data transfer control signals.

START
: The START signal originates data transfers and triggers arbitration. When asserted, it has a one cycle duration. This cycle is referred to as a START Cycle. During a START cycle all possibly addressed boards, slaves to the current bus master, examine the remaining control lines and the address lines to determine whether they are effected. Each START cycle is paired with exactly one ACK cycle.

ACK The ACK signal terminates data transfers and triggers the current arbitration winner to take over bus mastery. It also has a one cycle duration and this cycle is referred to as an ACK cycle. During an ACK cycle the last word of data transferred is valid and the remaining control lines contain transfer status. There is exactly one ACK cycle for each START cycle.

TM0 and TM1
: These are multi-purpose control lines. During a START cycle TM1 carries the transfer type (unasserted=read, asserted=write) and TM0 determines whether the transfer unit size is a byte (asserted=byte transfer) as shown in Table 2.2. During an ACK cycle they carry transfer status information as shown in Table 2.4. Between a START and ACK cycle of a block transfer TM0 is used to strobe word transfers (asserted means data lines valid for non-final word transfer).

AD<1> and AD<0>
: During a START cycle these are used for control instead of addressing. Conveniently, if the transfer unit is a byte, these can be interpreted as the byte address. If the transfer unit is not a byte they describe units of word, low halfword, high halfword and block transfer as shown in Table 2.2. When block transfer is specified, AD<5..2> may also carry control information.

AD<5..2>
: During a START cycle of a block transfer at least some of the low order address bits carry no information because blocks are aligned. For example, the smallest block is a doubleword and is doubleword aligned. So its low order address bits (AD<2..0>) are 000. NuBus block transfer encoding, shown in Table 2.3, specifies that if AD<2> is unasserted then the transfer unit is a doubleword and AD<5..3> are valid address bits. However, if AD<2> is

TM1/	TM0/	AD<1>/	AD<0>/	Transfer
L	L	L	L	Write Byte 3
L	L	L	H	Write Byte 2
L	L	H	L	Write Byte 1
L	L	H	H	Write Byte 0
L	H	L	L	Write Halfword 1
L	H	L	H	Write Block
L	H	H	L	Write Halfword 0
L	H	H	H	Write Word
H	L	L	L	Read Byte 3
H	L	L	H	Read Byte 2
H	L	H	L	Read Byte 1
H	L	H	H	Read Byte 0
H	H	L	L	Read Halfword 1
H	H	L	H	Read Block
H	H	H	L	Read Halfword 0
H	H	H	H	Read Word

Table 2.2: Transfer Mode Encoding
The type and size of a transfer unit are encoded in the Transfer Mode (TM) lines and the low order address lines during a START cycle.

asserted, then the transfer unit must be bigger than a doubleword and AD<3> should be examined. This algorithm is applied to each of AD<3> and AD<4> in turn. If AD<4> is asserted, then AD<5> is examined, but it must be unasserted because the largest transfer unit defined is 16 words long.

SPV The signal SP carries an even word parity bit on any cycle that SPV is asserted. The AD lines are the only lines covered by this parity mechanism.

2.6.2. Single Transfer Operations

A single transfer operation is composed of a START cycle specifying a byte, halfword or word transfer unit and its address followed by and ACK cycle specifying the data and the transfer status. Many cycles may pass between these two events, as long as

AD<5>/	AD<4>/	AD<3>/	AD<2>/	# of Words	Address
X	X	X	H	2	AD<31..3> 000
X	X	H	L	4	AD<31..4> 0000
X	H	L	L	8	AD<31..5> 00000
H	L	L	L	16	AD<31..6> 000000

Table 2.3: Block Size Encoding
Transfer units must be aligned according to their size in memory. This forces some of the low order bits of an address to zeroes. The encoding of block sizes larger than one word takes advantage of these address bits with known values.

the watchdog timeout does not occur. During the START cycle the transfer size is specified as shown in Table 2.2. If the operation type is read, TM1 is unasserted; if it is write, TM1 is asserted. For byte transfers TM0 is asserted and AD<1..0> used for addressing. For halfword transfers TM0 is unasserted, AD<0> asserted and AD<1> used for addressing. For word transfers TM0 is unasserted and AD<1..0> used for addressing (that is; unasserted).

2.6.2.1. Byte, Halfword and Word Read Protocol

In Figure 2.5, the protocol for single transfer read operations is shown.

R(1) The bus master drives the address and control lines (TMx, SPV and START) to signal a START cycle. The TMx and AD<1..0> lines indicate the transfer unit size. TM1 is unasserted and AD<1..0> are not asserted and unasserted respectively.

F(1) All potentially addressed slaves sample the control and address lines to determine if they are affected.

R(2) The master releases the address and control lines and waits for the ACK cycle.

R(n) The selected slave drives the data and control lines (TMx, SPV and ACK) to signal an ACK cycle. The TMx lines indicate the transfer completion status.

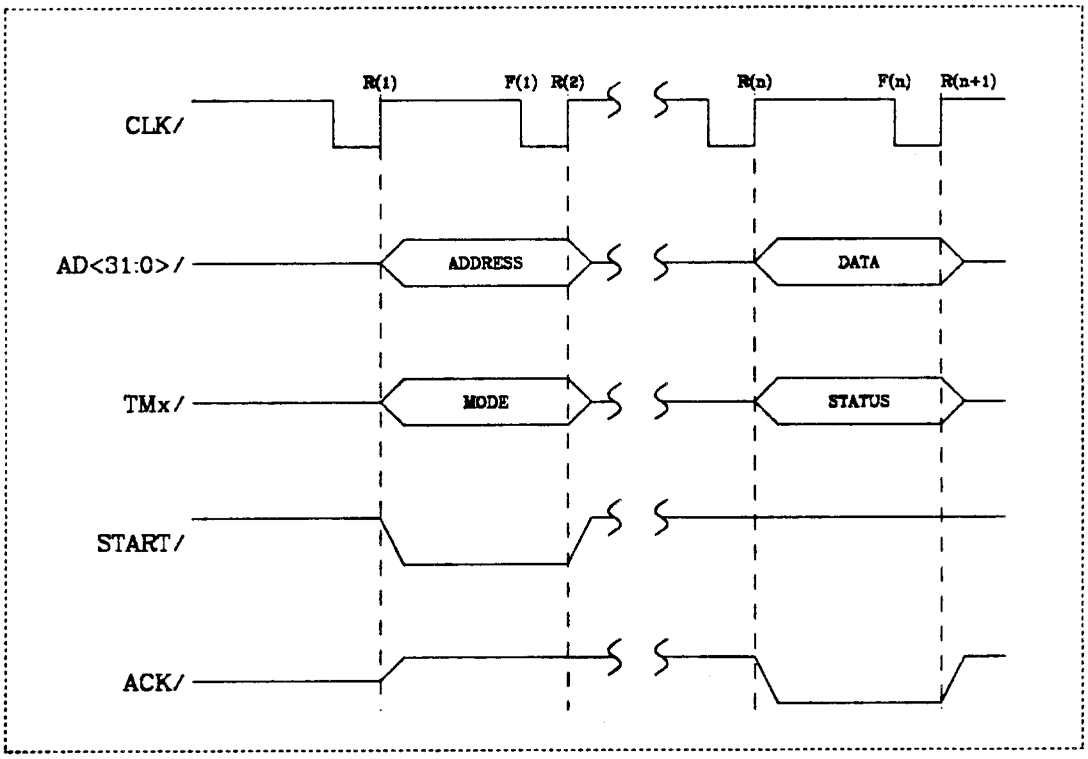


Figure 2.5: Word Read Protocol
Single transfer reads are composed of an address/mode cycle and a data/status cycle. The duration of a single transfer read can be many cycles as long as it is less than the system timeout period.

F(n) The master samples the data, status and ACK lines on all falling edges until it sees the ACK cycle.

R(n+1)
The selected slave releases the data and control lines for the next master to use.

(1) The parameter "n" will be less than the system timeout because a watchdog on the NuBus interface will generate an ACK cycle with a "timeout" status whenever the transfer takes too long.

2.6.2.2. Byte, Halfword and Word Write Protocol

The single transfer write operation protocol is shown is Figure 2.6.

R(1) The bus master drives the address and control lines (TMx, SPV and START) to signal a START cycle. The TMx and AD<1..0> lines indicate the transfer unit size. TM1 is asserted and AD<1..0> are not asserted and unasserted respectively.

F(1) All potentially addressed slaves sample the control and address lines to determine if they are affected.

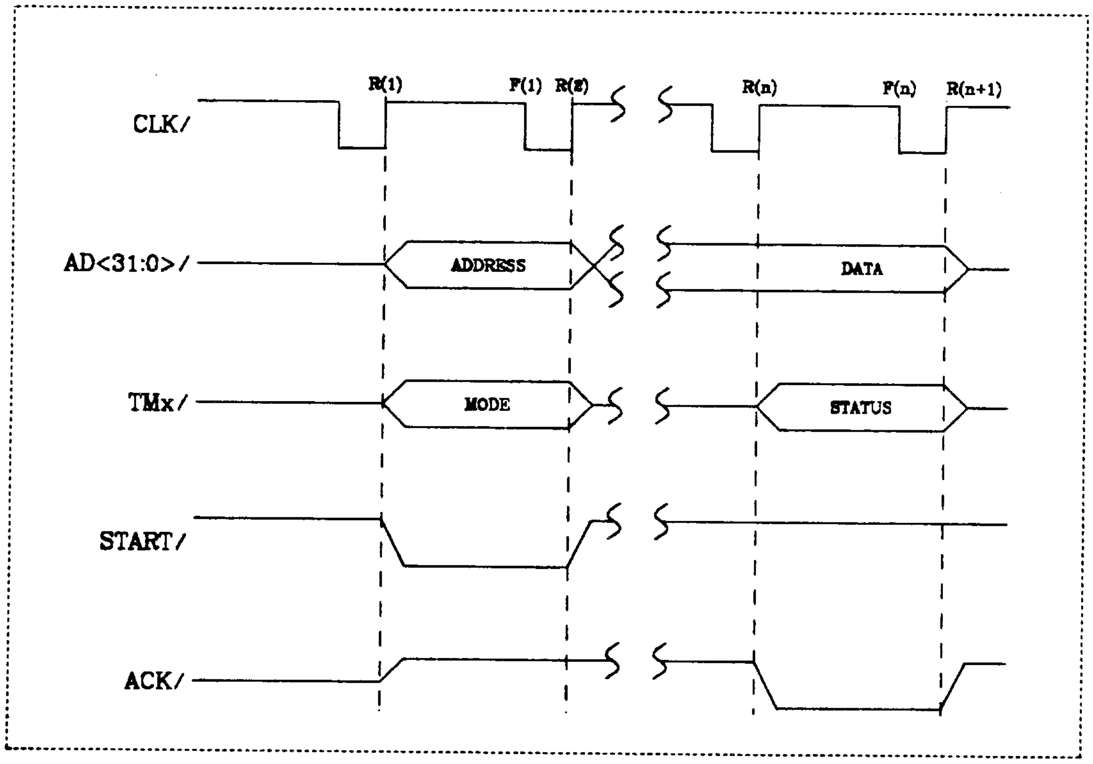


Figure 2.6: Word Write Protocol
A single transfer write operation appears very similar to a single transfer read. The important distinction is that the data must be valid on the first cycle after the address has been issued.

R(2) The master releases the control lines, drives the data and SPV lines and waits for the ACK cycle.

R(n) The selected slave drives the control lines (TMx and ACK) to signal an ACK cycle. The TMx lines indicate the transfer completion status. The slave may sample the data lines on or before the ACK cycle.

F(n) The master samples the status and ACK lines on every cycle until is sees the ACK cycle.

R(n+1)
The selected slave releases control lines and the current master releases the data lines for the next master to use.

(1) The parameter "n" will be less than the system timeout because a watchdog on the NuBus interface will generate an ACK cycle with a "timeout" status whenever the transfer takes too long.

2.6.3. Multi-Transfer Operations

A multi-transfer operation, commonly called a block transfer, is composed of a START cycle specifying a transfer unit size and its address followed by enough intermediate data strobes to transfer all but the last word of the transfer unit and finally an ACK cycle specifying the final word of data and the transfer status. The data in a block transfer is contiguous in physical memory and aligned according to the unit size. Many cycles may pass between each word transfer, as long as the watchdog timeout does not occur. During the START cycle the operation type and transfer size are specified. If the operation type is read, TM1 is unasserted; if it is write, TM1 is asserted. Because it is a block transfer TM0, AD<1> and AD<0> are, respectively, unasserted, asserted and unasserted. The transfer size is specified by one or more of the AD<5..2> lines as follows: if AD<2> is unasserted then transfer 2 words, else if AD<3> is unasserted transfer 4 words, else if AD<4> is unasserted transfer 8 words, else demand AD<5> to be unasserted and transfer 16 words.

2.6.3.1. Block Transfer Read Protocol

The protocol of a block transfer read operation is shown in Figure 2.7.

R(1) The master drives the address lines (AD<5..0> with control), SPV if SP is valid, drives TM0 and TM1 high and START low.

F(1) All potentially addressed slaves sample the address and control lines to determine whether they are affected.

R(2) The master release the address, TMx and START lines and begins waiting on the TM0 and ACK lines. The slave will drive TM0 unasserted.

R(n) The slave drives the data, SPV and TM0 lines. TM0 is asserted to indicate that a word, bit not the last word, is valid on the data lines.

F(n) The master is continually sampling the data, TMx and ACK lines looking for TM0 asserted and ACK unasserted. This is called an intermediate ACK cycle. The data lines contain a intermediate word on the transfer unit.

R(n+1)
The slave drives TM0 unasserted and begins to form the next word to transfer.

NOTE:
These intermediate stages (n) are repeated B-1 times where the size in words of the transfer unit is B.

R(b) The slave drives the data, SPV, TMx (with a status code) and ACK lines.

F(b) The master continually samples the data, TMx and ACK lines looking for intermediate or final ACK cycles. On the ACK cycle, the data lines contain the last word of the transfer unit and the TMx lines contain the transfer status. If this cycle occurs before the master believes that the last word is due then the master must detect this and realize that its transfer is over (in error).

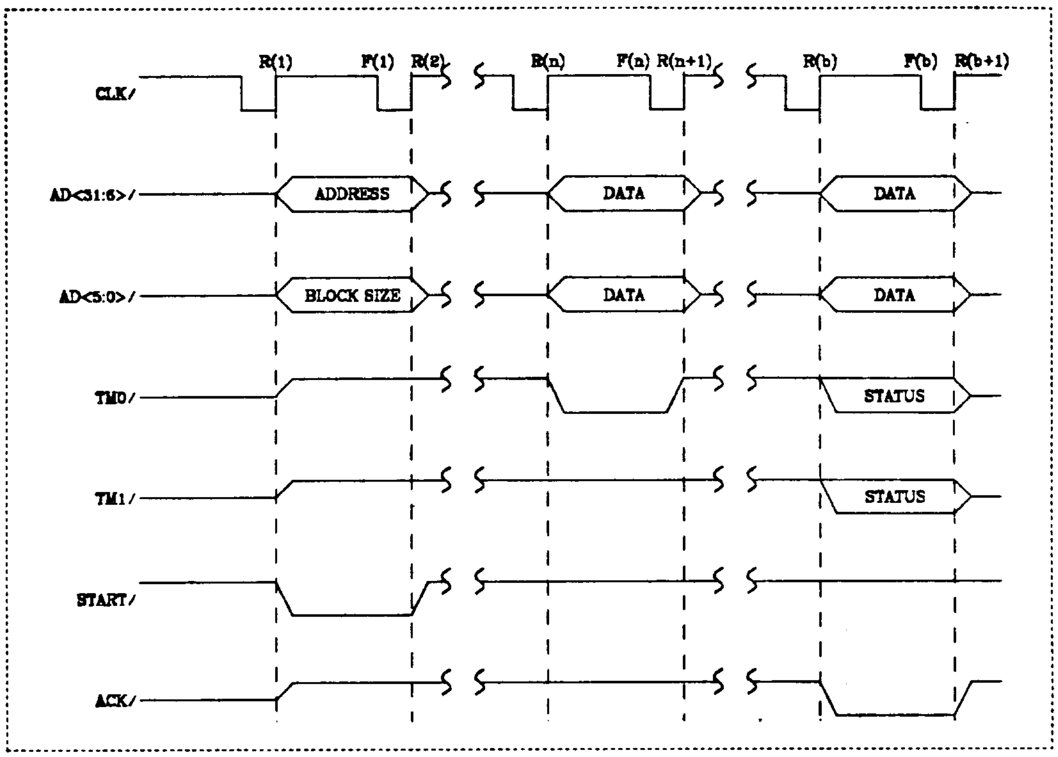


Figure 2.7: Block Read Protocol
Block transfers use the TM0 line to strobe intermediate words across the backplane. The slave controls the data and mode lines during a read. Each completed word transfer resets the system watchdog timer, so a block transfer can last much longer than a single word transfer.

R(b+1)
The slave releases the data and control lines for the next master to use.

(1) The parameter "n" will be less than the system timeout because a watchdog on the NuBus interface will generate an ACK cycle with a "timeout" status whenever the transfer takes too long.

(2) The parameter "b" will be less than the product of the transfer unit size in words, B, and the system timeout constant referred to above.

2.6.3.2. Block Transfer Write Protocol

The protocol for a block transfer write operation is shown in Figure 2.8.

R(1) The master drives the address lines (AD<5..0> with control), SPV if SP is valid, drives TM0 high, TM1 low and START low.

F(1) All potentially addressed slaves sample the address and control lines to determine whether they are affected.

R(2) The master release the TMx and START lines, drives the data lines with the first word of the transfer unit and begins waiting on the TM0 and ACK lines. The

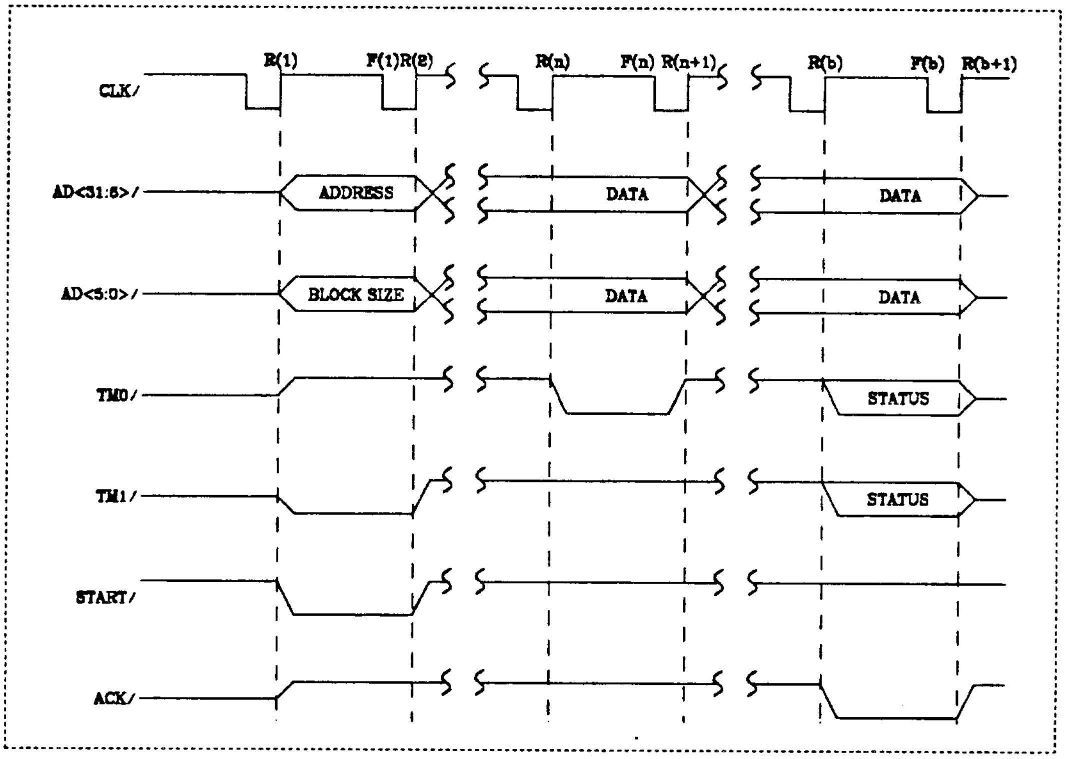


Figure 2.8: Block Write Protocol
A block write transfer is very similar to a block read transfer. The important difference, like the single word transfer case; is that the master retains control of the data lines and must be able to provide the next data word on the cycle after an intermediate acknowledge or address transmission.

slave will drive TM0 unasserted.

R(n) The slave drives the TM0 line. It samples the data lines on or before this cycle.

F(n) The master is continually sampling the TMx and ACK lines looking for an intermediate ACK cycle. On the intermediate ACK cycle the slave may sample the data.

R(n+1)
The slave drives TM0 unasserted. The master must change the value on the data lines to the next word of the transfer unit.

NOTE:
These intermediate stages (n) are repeated B-1 times where the size in words of the transfer unit is B.

R(b) The slave drives TMx (with a status code) and ACK lines. It must sample the data lines on or before this cycle.

F(b) The master continually samples TMx and ACK lines looking for intermediate or final ACK cycles. On the ACK cycle, the TMx lines contain the transfer status. The slave may sample the data lines on the ACK cycle. If this cycle occurs before the master believes that the last word is due then the master must detect this and realize that its transfer is over (in error).

R(b+1)
The slave releases the control lines and the current master releases the data lines for the next master to use.

(1) The parameter "n" will be less than the system timeout because a watchdog on the NuBus interface will generate an ACK cycle with a "timeout" status whenever the transfer takes too long.

(2) The parameter "b" will be less than the product of the transfer unit size in words, B, and the system timeout constant referred to above.

2.6.4. Transfer Completion Status Codes

The status of each transfer according to the slave is reported to the master using the mode lines (TMx) during the ACK cycle. Table 2.4 shows the encoding and status types.

Successful Completion
This indicates that the slave believes the transfer is successfully complete.

Bus Timeout Error
The NuBus specifies that a system timer (probably located with the system clock) limit the duration between START and ACK cycles of a single transfer operation and between each transfer of a multi-transfer operation. If this timer expires, the watchdog logic will generate an ACK cycle with the "bus timeout error" status.
The typical reason for a bus timeout is that the address transmitted during a START cycle is unimplemented by the active slaves. The timeout value should be large enough that any valid operation will complete (ie., substantially longer than a reasonable transfer duration).
The system timer is also intend to discover SpurBoards that arbitrate and obtain the bus, but don't use it (generate a START cycle). To do this, the system timer should also start whenever arbitration selects a winner.

Try Again Later
If the addressed slave is unable to respond to the transfer request for a transient reason, it can generate a "try again later" ACK cycle. The master's request is not in error and should be retried later.

Unspecified Error
This is the catch-all error status. Typical reasons for an "unspecified error" are: bus parity error in transit or memory parity/ECC error.

TM1/	TM0/	Status Message
L	L	Successful Completion
L	H	Unspecified Error
H	L	Bus Timeout Error
H	H	Try Again Later

Table 2.4: NuBus Status Codes
During the ACK cycle of any transfer the mode lines pass an overall status code. The "try again later" code is intended for bus-to-bus interface board usage.

2.6.5. Idle Cycles

An idle cycle is any cycle during which START and ACK are asserted. When this occurs no information is passed on the mode or data lines. This type of cycle is intended to reinitiate bus arbitration. The primary situation where this is required occurs when the bus is requested and obtained, but the master does not require a transfer.

2.6.6. Interrupts

Interrupts are not given special attention in a NuBus system. They are implemented as memory mapped write transfers. The transfer unit size is unspecified, so considerable volumes of data can be passed. The destination of an interrupt write transfer must be an address that a processor board will recognize as interrupt space. Typically part of a board's slot space will be used for this. Software can construct interrupt priorities and levels according the address of the interrupt write transfer.

2.6.7. Parity

Parity can be used to protect the integrity of the address/data lines during any transfer. A pair of lines are defined to implement the optional use of even word parity. One line, SP/, is unasserted if an even number of the lines AD<31> to AD<0> are asserted; otherwise, it is asserted. The other line, SPV/, is asserted if SP/ is valid. Parity is optional on a cycle by cycle basis. Either masters or slaves may choose to ignore it, but must drive SPV/ unasserted if they are driving the AD<31..0> lines.

The following policies should be used when parity errors are detected:

During a START cycle
: All slaves should ignore the transfer. The system timer will reinitiate arbitration if the master doesn't.

During transfers of a read operation
: The master should ignore the data it receives and treat the transfer as in error once the ACK cycle arrives.

During transfers of a write operation
: The slave should generate an ACK cycle with an unspecified error status code.

2.7. NuBus Arbitration

NuBus arbitration is distributed across boards that act as bus masters. It uses five "*open-collector*" backplane lines[1] (RQST/ and ARB<3..0>/). Allocation of backplane bandwidth through NuBus arbitration prevents the "starvation" of any particular requestor. In this sense this scheme is "fair". In the limiting case, each requestor will receive the same fraction of bandwidth.

The NuBus arbitration mechanism resolves all contention between bus masters. When a bus master requires the bus it begins by entering arbitration for bus ownership. All masters that enter arbitration at the same time form a "wave". One master at a time will be selected and gain bus ownership. Within a contest the physical position of backplane boards is used to select a winner. The winner uses, then releases the bus. The winner of the next contest within the original wave then gets the bus. This continues until all masters in the original wave have been given the bus and the last has released the bus. At this point a new wave is allowed to form. Because a new wave cannot begin until all members of the previous have been served, it is guaranteed that a high priority master cannot preempt a lower priority master.

[1] An open-collector line implements a "wired-AND" logic function. The line will take a low value if any master drives it low and it will take a high value if all masters drive it high.

2.7.1. Arbitration Specification

A bus master enters a wave by pulling the RQST/ line low (asserting RQST/). This is only allowed when RQST/ is not currently asserted; that is, a wave is formed by those masters that assert RQST/ on the same clock cycle.

A schematic for the arbitration logic on each bus master board is shown in Figure 2.9. In this diagram the line arb/ and GRANT are internal to the bus master board. The arb/ line indicates the master's desire to use the bus and the GRANT line indicates that bus ownership has been attained. This logic corresponds to the following equations (where ARBx is the inverted value of ARBx/, ARBxo is the driven value and ARBxi is the read value):

$$\begin{aligned}
ARB3o &= arb * ID3\\
ARB2o &= arb * ID2 * (ID3 + ARB3/i)\\
ARB1o &= arb * ID1 * (ID3 + ARB3/i) * (ID2 + ARB2/i)\\
ARB0o &= arb * ID0 * (ID3 + ARB3/i) * (ID2 + ARB2/i) * (ID1 + ARB1/i)
\end{aligned}$$

When a master enters arbitration by asserting RQST/ it activates the arbitration logic on its board. The slot ID of the board is driven onto the ARB<3..0>/ lines and the boards determine the highest priority slot ID in parallel (slot zero has the lowest priority). After two clock cycles the arbitration logic on all boards must have settled and the winner is the board whose slot ID is still on the ARBx lines.

The winner of arbitration will be the next master to initiate a transaction, but it may not be able to this immediately. Arbitration may be overlapped with the last winner's last (or only) transfer so the new winner may have to wait for this transfer to complete. At this point the new winner may begin a transfer. Once it has begun a transfer, it can release ownership of the bus; that is, re-initiate arbitration among the masters that lost to it.

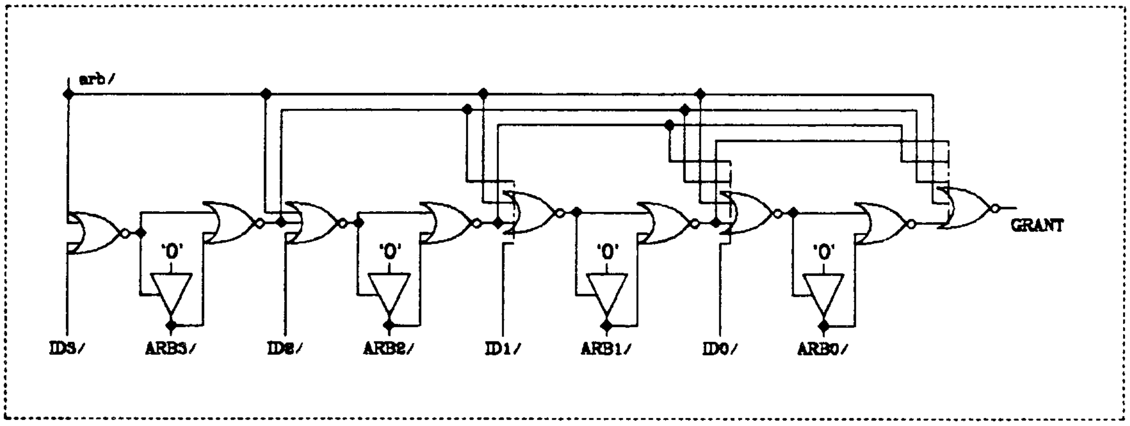


Figure 2.9: Possible Arbitration Schematic
An implementation of the distributed arbitration logic is shown here. The IDx lines are backplane supplied and unique to each bus port. They contain the binary coded slot number of that bus port. The open-collector ARBx lines are used to selected the slot number of the next bus master from those that are asserting arb/. The winner will see GRANT asserted. Circuit design must guarantee that GRANT has settled within two clock cycles (200 nsec) of the assertion of arb/.

After a winner has been selected in arbitration, the winner and losers alike continue to assert RQST/ to prevent new requestors from entering the wave. The losers will examine the ARBx lines again two cycles after the next START cycle they see on the bus. If the current winner stops driving its slot ID onto the ARBx lines by the end of the START cycle then a new winner will be selected. If the current winner is going to stop driving its slot ID at the end of a START cycle it must release the RQST/ line at the beginning of that START cycle. This ensures that boards waiting to start a new contest may enter arbitration during the transfer of the last winner in the current contest.

2.7.2. Bus Locking

An important feature of arbitration mechanisms is the provision of multi-transfer transactions. A good example of this is the operation "test-and-set". In this operation a memory location is read and a integer value of 1 is written back into it without any intervening operation by any other processor. This is used to construct a variety of multiprocessor synchronization mechanisms.

The NuBus arbitration mechanism specifies that a new winner within a wave will be selected two cycles after each START cycle during that arbitration sequence. Usually the current winner will withdraw from arbitration on the START cycle it generates after winning. This corresponds to a single transfer per transaction (period of bus ownership). However, the current winner may not withdraw from the arbitration on its START cycle. Because it is the current winner and because no new contender is allowed into the arbitration until all have withdrawn, the continued presence of the current winner ensures that it will win again. In this way the current master can complete many consecutive transfers. In general it is not good policy for a processor to hold the bus for too long because it increases the overall average time every processor is blocked waiting for the bus.

2.7.3. Bus Parking

When there are a small number of bus masters in the system or when requests are generally rare but clustered, it is often the case that the last user of the bus will be the sole member of a wave. When this is the case it will be held up two cycles arbitrating without competition. This adds unnecessary latency to the transfer duration.

In the NuBus specification "bus parking" is defined to allow this problem to be reduced. A bus master is parked on the bus if after withdrawing from arbitration there is no other contender (RQST/ becomes unasserted). This parked master remains parked until a new wave is formed. While it is parked it may begin a transfer without arbitrating (note that it must still look at RQST/ to determine if it is still parked on the bus). If a wave begins after the parked master has begun a transfer the selected winner will wait for the current transfer to end and the parked master will realize that it is no longer parked when it next wants to use the bus.

2.7.4. Arbitration Timing

The timing of the arbitration logic is shown by example in Figure 2.10. The most important details are that the ARBx lines are driven beginning on the sample edge instead of the assertion edge and that the arbitration logic must settle within 200 nsec after starting.

In this example the bus masters on boards in slot 5, 10 and 15 want to use the bus. Masters 5 and 10 go after the bus together and master 15 comes after the bus a few cycles later. At the time masters 5 and 10 become interested in the bus it is idle; that is, the last transfer is finished and the RQST/ line is unasserted.

F(0) Masters 5 (binary coded 0101) and 10 (binary coded 1010) notice that RQST/ is unasserted and begin driving the ARBx lines with their respective IDx lines immediately (note that the ARBx lines are asserted on the sample clock not on

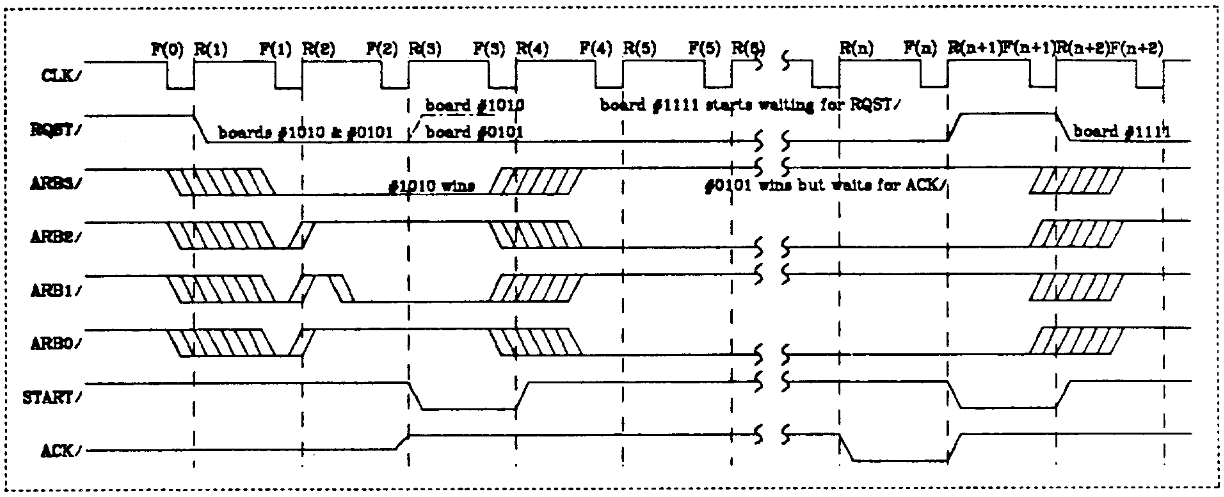

F(0) R(1) F(1) R(2) F(2) R(3) F(3) R(4) F(4) R(5) F(5) R(6) R(n) F(n) R(n+1) F(n+1) R(n+2) F(n+2)
CLK/
RQST/
ARB3/
ARB2/
ARB1/
ARB0/
START/
ACK/
board #1010
boards #1010 & #0101
board #0101
board #1111 starts waiting for RQST/
board #1111
#1010 wins
#0101 wins but waits for ACK/

Figure 2.10: Arbitration Timing Example
The behaviour of the arbitration logic is demonstrated with this sample contest between boards #10 and #5 followed by a contest containing only board #15. The bus is inactive at the time that the first contest begins. Board #10 wins after two cycles, begins its transfer and releases bus ownership. This prompts #5 to arbitrate against itself, win and then wait for the end of the current transfer. When #10 is finished with the bus, #5 starts its transfer and releases bus ownership. This prompts #15 to begin to arbitrate against itself.

the assertion clock). Using the sample arbitration logic shown in Figure 2.9, driving the ARBx lines corresponds to asserting the arb/ line on their respective boards. The ARBx lines will be changing as the logic settles but must be stable early enough so that the internal signal GRANT correctly reports the winning master after two cycles.

R(1) Masters 5 and 10 both assert RQST/ (form a wave) to stop further masters from entering arbitration late (specifically, master 15 will be blocked until both 5 and 10 release RQST/).

F(1) Arbitration continues.

R(2) Arbitration continues; that is, both masters continue to assert their arb/ and RQST/ lines.

F(2) The arbitration period is over. Masters 5 and 10 look at their respective GRANT lines to see who is the winner. Master 10 sees an asserted GRANT and master 5 sees an unasserted GRANT. Master 10 is the winner. Both masters continue to assert RQST/ and their own arb/ lines.

R(3) Master 10 begins the START cycle of its transfer. Since it does not want to lock the bus it releases its assertion of RQST/, but it continues to assert its arb/ line. Master 5 continues to assert RQST/ and its arb/ line.

F(3) Master 10 releases its local arb/ line which releases its assertion of the ARBx lines. Master 5 detects the START cycle and begins waiting for this arbitration to settle.

R(4) Master 5 continues to assert RQST/ and its arb/ line.

F(4) Arbitration continues. Master 15 wants to use the bus and looks at RQST/. Since RQST/ is asserted master 15 is not allowed to arbitrate. It will poll RQST/ each cycle until it is unasserted.

R(5) Arbitration continues; that is, master 5 continues to assert RQST/ and its arb/ line.

F(5) This arbitration is over. Master 5 sees its GRANT asserted and is the winner. Master 15 sees RQST/ still asserted.

R(6) Master 5 would like to begin its transfer, but master 10's transfer is not finished. Master 5 will poll the ACK/ line each sample edge waiting for the current transfer to end. It will continue to assert RQST/ and its arb/ line so that it holds bus ownership until it can begin a transfer.

R(n) Master 10 finishes its transfer by asserting ACK/. Master 5 is still asserting RQST/ and its arb/.

F(n) Master 5 sees the ACK cycle. Master 15 still sees RQST/ asserted.

R(n+1)
Master 5 releases RQST/. This allows RQST/ to become unasserted. Master 5 also begins the START cycle of its transfer.

F(n+1)
Master 5 releases its arb/ line. Master 15 sees RQST/ unasserted so it asserts (immediately) its arb/ line. It also sees the START so it will monitor for the ACK to track the current transfer.

R(n+2)
Master 15 asserts RQST/ to form a contest with itself only. This contest will be won in two cycles and master 15 will proceed much as master 5 did after it won.

3. NuBus Subset in SpurBus

The SpurBus design is based on the NuBus specifications; however, it doesn't implement the full protocol. The subset selected eliminates portions of the NuBus design that the SPUR system does not use. The SpurBus subset applies to SpurBoards only; NuBus protocols not supported in SpurBus may still exist in a SPUR system.

3.1. One and Eight Word Transfer Sizes

Data transfers have two possible transfer unit sizes: 1 word and 8 words (respectively, 4 bytes and 32 bytes). The 8 word unit corresponds to a SPUR cache line size and typical program activity will use this size unit. The 1 word size is provided for direct processor operations on physical memory or device registers (bypassing the cache).

3.2. No Bus Parity

The SpurBus does not generate or detect parity on the bus. Refer to the SPUR Design Rationale document [???] for a discussion of this decision.

3.3. Dataless Interrupts

The SpurBus interrupts do not pass information after the START cycle. All information is contained in the address passed during the START cycle of a write operation. The ACK cycle is used to delay the transfer (or rather the beginning of the next transfer) until the receiving SpurBoard has recorded the interrupt.

A SpurBoard recognizes addresses in its slot space as interrupts if they fall within the low 1 MB; that is, if the address begins "F(id)0" (hex) where "id" is the Slot ID of the SpurBoard. This space folds around 16 unique interrupt addresses, all word aligned; that is, the four bits AD<5,2> are used to fix the interrupt type. The transfer unit size of a SpurBus interrupt is a word.

3.4. No Bus-based Test-and-Set

The SPUR processor cache controller design provides test-and-set operations on data obtained by the normal read for ownership cache operations. Because this operation is entirely carried out on data in the cache (or brought into the cache for this operation) the bus multi-transfer facilities are not used.

4. NuBus Extensions for SpurBus

The SPUR project will use the NuBus design; however, its cache consistency and virtually addressed caches make demands on the bus that the NuBus does not attend to well. An extended NuBus design, called the SpurBus, supports the needs of SPUR more fully.

4.1. Snooping Caches' Datapath

The SPUR system employs "snooping caches" [Katz85a???] to provide cache consistency across multiprocessor caches. In essence this means that each cache in the system must monitor all bus transfers and be prepared to provide data from its RAMs whenever it has the most up-to-date copy. This is implemented with state bits on each cache line that distinquish states "invalid", "owned private", "owned shared" and "unowned". The concept of ownership is used to determine when a cache may write its copy of the data without informing the other caches. There is at most one cache in the system that owns a block. When a block is not owned by a cache, it is implicitly owned by memory.

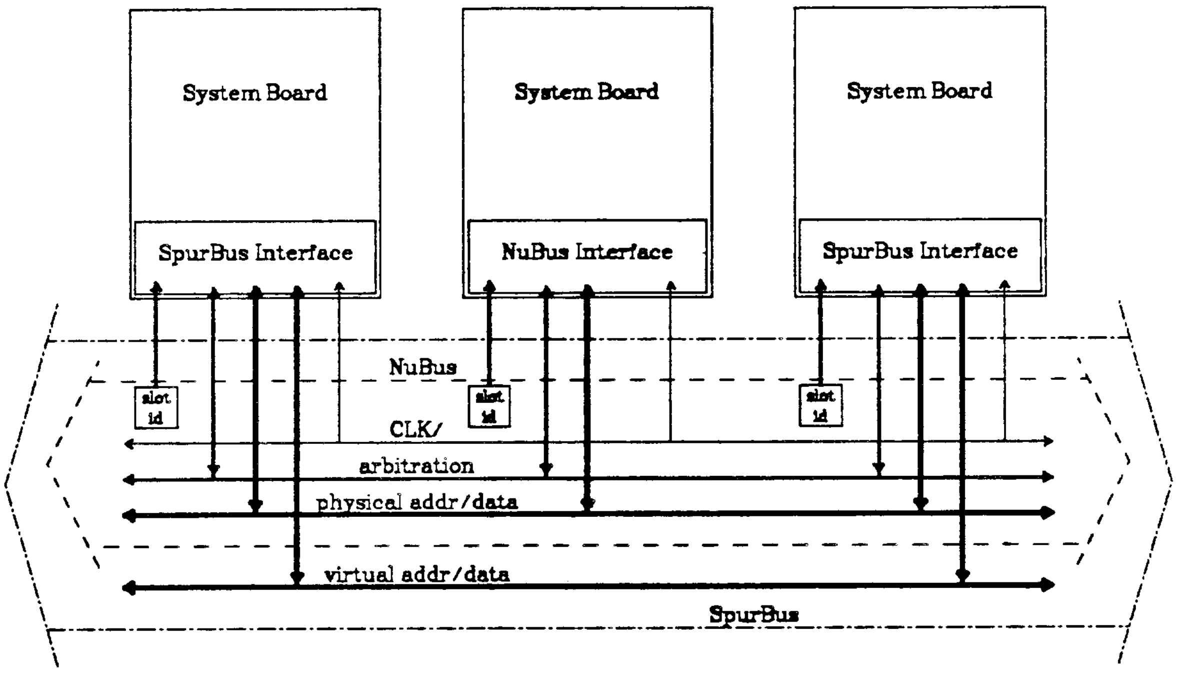


Figure 4.1: SpurBus Block Diagram
The SpurBus extends the NuBus design with a second data path. This second path is used by SpurBoard snooping caches to pass shared data lines amongst themselves. This path has its own control for transfers, but it is synchronized to the NuBus data path arbitration and START cycle.

The implementation of a snooping cache requires bus bandwidth, the ability to monitor bus events and the ability to cancel memory operations. The NuBus specification and the memory boards that are currently available for NuBus racks (1/2 MB boards and plans for 2 and 8 MB boards) do not make any allowances for cancelling operations. The addition of a second data path in SpurBus allows inter-cache operations to occur in parallel with memory operations without loss of bandwidth. This way memory operations do not need to be cancelled; they can be allowed to finish without delaying the inter-cache operation.

This second path also allows the inter-cache address to be different than the memory address. This is important because it allows the SpurBoard cache to be accessed on virtual addresses and the memory system to use physical addresses. During any read on the SpurBus the virtual address is sent on the inter-cache bus and the physical address is sent on the NuBus.

4.2. SpurBus Signals

CLASS	GROUP	SIGNAL	# OF PINS	PROTOCOL
Shared	Clock	CLK/	1	input-only
	Arbitration	ARB<3..0>/	4	open-collector
		RQST/	1	open-collector
	Utility	RESET/	1	open-collector
		ID<3..0>/	4	input-only
NuBus	Address/Data	AD<31..0>/	32	tri-state
	Parity	SP/	1	tri-state
		SPV/	1	tri-state
	Control	START/	1	tri-state
		ACK/	1	tri-state
		TM0/	1	tri-state
		TM1/	1	tri-state
SpurBus	Address/Data	VAD<32..0>/	33	tri-state
	Control	VACK/	1	tri-state
		VTM0/	1	tri-state
		VTM1/	1	tri-state
		Total Signals	85	
Shared	Reserved	RSVD/	1	
	Power/Ground	+5	22	
		-5	16	
		+12	4	
		-12	4	
		GND/	46	
		Total Lines	178	

Table 4.1: SpurBus Backplane Lines
The SpurBus extends the NuBus backplane needs by 36 signal pins and 46 power/ground pins. The resulting 178 pins use the better part of two 96 pin connectors.

The SpurBus extension to the NuBus design, shown in Figure 4.1, calls for an additional 36 backplane lines, shown in Table 4.1. These are the address/data lines, VAD<32..0>, and control lines, VACK, VTM0 and VTM1, for the inter-cache datapath.

The use of the inter-cache datapath is synchronized with the NuBus datapath. This means that only one set of arbitration logic and only one START line are needed. The master of the bus always performs a NuBus operation and can optionally perform an inter-cache operation at the same time.

Some of the control for a NuBus operation applies to the inter-cache operation. Both datapaths read or write together; that is, the read/write characteristic of a NuBus operation applies to both paths. Also, snooping reads only apply to cache lines which are 32 byte blocks.

4.3. Snoop Operations

Three operations are needed to support snooping caches. Two of these are modifications on the NuBus block transfer read operation. The third is a non-NuBus operation for invalidating cache lines.

4.3.1. Snooping Reads

"Snooping reads" are variations on the NuBus block transfer read. In the START cycle of these operations the NuBus describes a regular 32 byte read and the

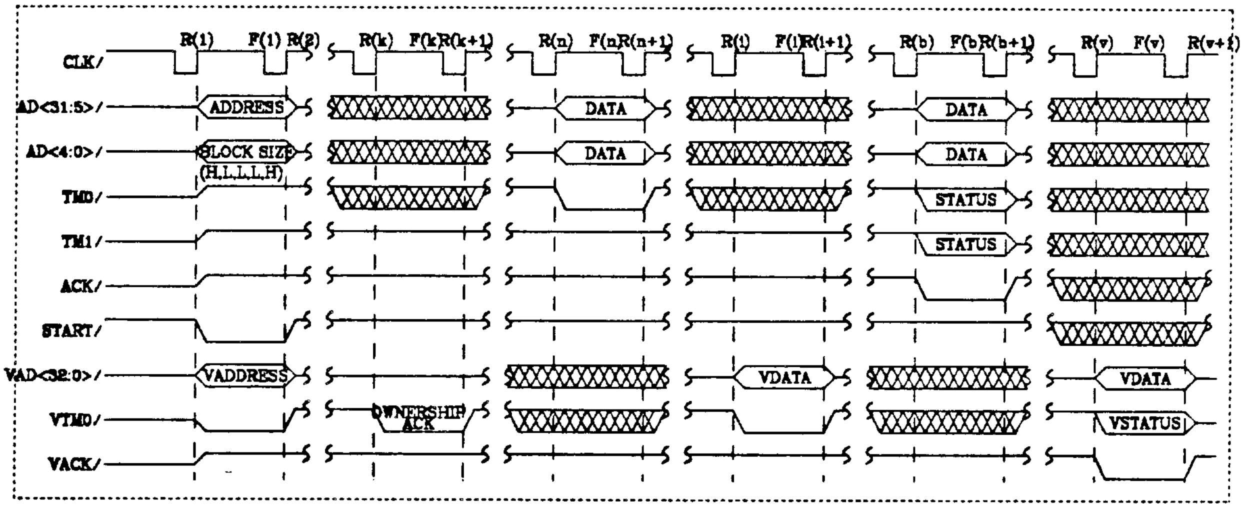


Figure 4.2: Snooping Read Protocol
A SpurBus snooping read may result in a cache-to-cache transfer of a shared data cache line on the inter-cache data path. If any cache intends to use this path, then only one will and it will generate an ownership acknowledgement indicating that it will reply within a timeout guaranteed to be shorter than the NuBus data path transfer. The two transfers proceed with distinct control and may end at different times. The address sent on the inter-cache bus is the virtual address of the line in the master's cache (less the block offset bits).

inter-cache bus qualifies this for the benefit of the other caches as non-snooped, snooped for a shared copy (called "read shared") or snooped for a private copy (called "read for ownership"). The NuBus operation is unaffected; the inter-cache bus carries the modifiers and the potential inter-cache data.

Figure 4.2 shows the protocol for snooping reads.

R(1) The master drives AD<31..5> with a cache line physical address (in units of aligned 32 byte blocks) and VAD<32..0> with the virtual address of the same cache line (also in units of aligned 32 byte blocks). It drives the control lines AD<4..0> to high, low, low, low and high respectively and TM0, TM1 and VTM0 to high, high and low respectively. If the operation is read for ownership, it also drives VTM1 low. Finally, to make this a START cycle, it drives START low.

F(1) All potentially addressed slaves on the NuBus sample the AD<31..0> and TMx lines to determine whether they are affected. All SpurBoards sample the AD<4..0>, TMx, VAD<32..0> and VTMx lines and check their cache line states to determine whether they are affected and should reply with a cache line of data.

R(2) The master releases the ADx, TMx, VADx, VTMx and START lines. It will now wait on the TM0, ACK and VTMx lines.

NOTE:
The transfers on the NuBus datapath and inter-cache datapath are decoupled. In the following descriptions, two events are sequential if they apply to the same datapath; otherwise, they may occur together or in either order. The one exception to this is that an OACK cycle, if one is to occur, must not occur after an ACK cycle.[2]

R(k) If a cache on a slave SpurBoard decides to respond with a cache line on the inter-cache datapath it asserts VTM0 for a cycle. This is called an OACK cycle (ownership ack).

F(k) The master samples the VTMx lines and determines that a inter-cache transfer will take place. This implies that data arriving on the NuBus datapath to be thrown away. The master then starts sampling the VADx, VTMx and VACK lines for data and completion.

F(k+1)
The responding slave cache drives VTM0 unasserted and begins to form the first word of the line.

R(n) The addressed NuBus slave drives the ADx and TM0 lines. This is one of the first 7 words of the 8 word block.

F(n) The master captures the data from the ADx lines if it has not decided to throw away this data in preference for inter-cache data.

R(n+1)
The NuBus slave drives TM0 unasserted and begins to form the next word of to transfer.

NOTE:
These NuBus intermediate stages (n) are repeated B-1 times where the size of the transfer unit in words is B.

R(i) The responding cache drives the VADx and VTM0 lines. This is one of the first 7 words of the 8 word block.

F(i) The master captures the data from the VADx lines. It will keep this data in preference over data from the NuBus datapath.

R(i+1)
The responding cache drives VTM0 unasserted and begins to form the next

[2] This is a source of trouble if the NuBus slave generates an error ACK before it would normally be possible to terminate normally (8 cycles to transfer 8 words). All SpurBoard caches currently snooping must see this abnormal ACK, terminate snooping and be ready to snoop again on the next bus cycle.

word to be transferred.

NOTE:
These inter-cache intermediate steps (i) are repeated B-1 times where the size of the transfer unit in words is B.

R(b) The addressed NuBus slave drives the ADx lines with the last word of the line, the TMx lines with a status code and the ACK line.

F(b) The master captures the final word of the data and the status code.

R(b+1)
The NuBus slave releases the ADx, TMx and ACK lines for the next NuBus master to use. There can be a complication here if the inter-cache transfer operation is not finished. This is covered in section (?????).

R(v) The responding cache drives the VADx lines with the last word of the line, the VTMx lines with a status code and the VACK line. The status code is shown in Table 4.3. It includes information to tell the receiving cache whether the cache line transferred is dirty (memory has not been updated after a cache store into this line).

F(v) The master captures the final word of the data and the status code.

R(v+1)
The responding cache releases the VADx, VTMx, and VACK line for the next master to use.

(1) The parameter "k" is restricted to be less than or equal to the eighth cycle after the START cycle (ie., k <= 9). After this point the master assumes that the cache line it is looking for will come from memory only.

(2) The parameter "n" is constrained to be less than the system timeout on transfers. If the timeout fires the transfer will be aborted by a "timeout" ACK cycle generated by the system logic.

(3) The parameter "i" should be constrained by an inter-cache transfer timeout in the same way that the NuBus datapath transfers are constrained.

(4) The parameter "b" must be less than the product of the line size, B, in words and the NuBus system timeout.

(5) The parameter "v" should be less than the product of the line size, B, in words and the inter-cache transfer timeout.

4.3.2. Write For Invalidate

The third operation needed to support snooping caches is called "write for invalidation". It is used by a cache that wants all other caches to invalidate (throw away) their copies of a cache line. It requires a bus operation, but it doesn't correspond to a parallel NuBus operation. The SpurBus defines a NuBus write operation to pass the write for invalidation. This uses a NuBus write of one word to a physical address in the slot space of the generating board at a non-interrupt address. The snoop operation is sent during the START cycle and the operation lasts long enough for all caches to complete it then the master generates an ACK cycle to itself.

The protocol for an write for invalidation operation is shown is Figure 4.3.

R(1) The master drives the ADx lines with "F(id)F00000" (hex) where "id" is the Slot ID of the master. This is a dummy physical address that falls into the master's own slot space. This guarantees that no NuBus slave will respond to this operation. The master will complete this operation itself after an appropriate delay. Also driven by the master during the START cycle is TM0, VTM0 and the VADx lines. This makes the operation a snooped word write. The VADx lines carry the virtual address of a cache line to be invalidated by other snooping caches.

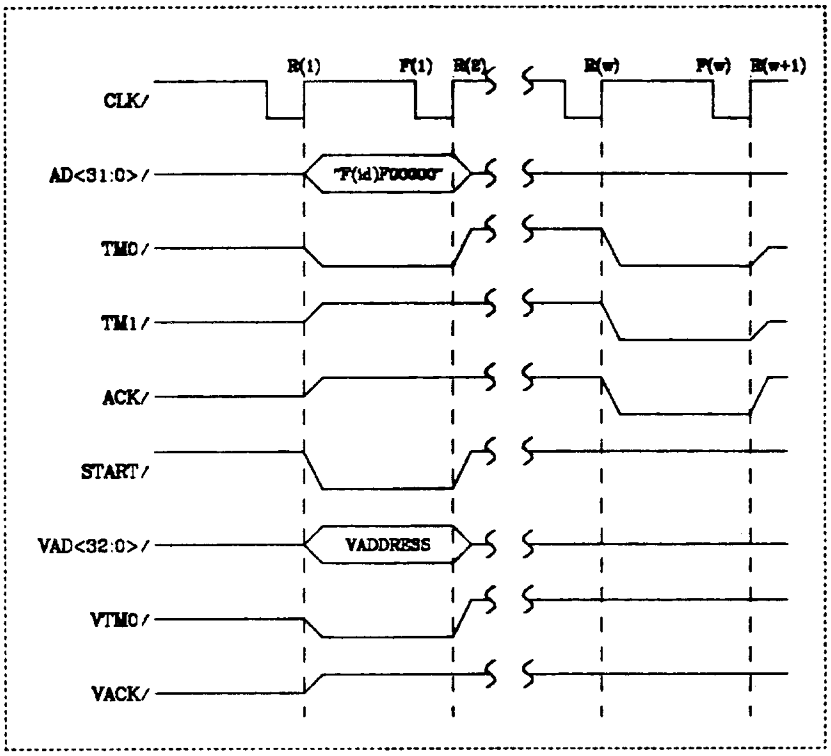


Figure 4.3: Write For Invalidation Protocol
A SpurBus write of invalidate operation employs a NuBus word write back into the generating board's slot space (but not into interrupt locations). The invalidation information is the virtual address of the cache line to invalidate and is carried on the inter-cache data path. The duration of the operation is controlled by the generating board and must be large enough that all other caches have time to complete the invalidation.

F(1) All SpurBoard snooping caches determine that a write for invalidation is active and look into their caches to see if they have the line to be invalidated. They will not drive any backplane lines in this operation.

R(2) The master releases the ADx, VADx, VTMx and TMx lines. It will delay a fixed period of time before completing the operation.

R(w) The master completes its own operation by driving a successful status code on the TMx lines during a ACK cycle.

(1) The duration of the write for invalidate operation is selected to guarantee that all snooping caches have time to complete their own invalidations, but it must be less than the system timeout.

4.4. Inter-cache Datapath Control

The inter-cache datapath is composed of address/data lines, VAD<32..0>, and acknowledgement line, VACK, and two multi-purpose control lines, VTM0 and VTM1.

VAD<32..0>

The inter-cache data path is VAD<31..0>. It has the same characteristics as the NuBus data path except that it is used to move cache lines according to the snooping cache protocols.
The inter-cache data path is wider than the NuBus data path because of SPUR virtual addressing. Global virtual addresses in the SPUR system are 38 bits wide. Since cache lines are aligned on 32 byte boundaries the low order 5 bits are always zero and are not transferred. So inter-cache addresses are 33 bits.

VACK

The VACK line on the inter-cache datapath corresponds to the NuBus ACK line. When transfers occur on the inter-cache datapath (when an ownership ACK occurs on a snooping read) VACK terminates the operation. A cycle that has VACK asserted is called a VACK cycle.

VTM0 and VTM1

These two control lines parallel the transaction mode lines on the NuBus datapath.
During a START cycle VTM0 is asserted if the inter-cache datapath is being used. If a read is taking place then VTM1 indicates the type of snooping required as shown in Table 4.2. VTM1 asserted indicates a "read for ownership" and VTM1 unasserted indicates a "read shared" operation.
After the START cycle of a snooping read caches on boards other than the current bus master have a window of time to determine whether they should provide the requested data. This window is eight cycles long. If a cache decides to provide the data it will assert VTM0 for a cycle. This is called an OACK (ownership ack) cycle.
After an OACK cycle VTM0 is used to strobe data across the inter-cache datapath in the same way that TM0 is used on the NuBus datapath. When the last word of the cache line is transferred VACK is asserted and VTMx carry

VTM0/	TM1/	VTM1/	Inter-cache Operation
H			None
L	L		Write for Invalidation
L	H	H	Read for Shared Copy
L	H	L	Read for Private Copy

Table 4.2: Snooping Operation Encoding
During the START cycle of any NuBus operation the usage of the inter-cache datapath is determined from the VTM0, VTM1 and TM1 lines.

VTM0/	VTM1/	Status Message and Cache Line Status
L	L	Successful Completion; Line Dirty
L	H	Successful Completion; Line Clean
H		Unspecified Error

Table 4.3: Snooping Operation Status Codes
During the VACK cycle of a snooping read on the inter-cache datapath VTM0 carries a success/error status code. If VTM0 indicates success, VTM1 indicates whether the data that came across the inter-cache bus matches its memory image (clean) or has been written since memory was last updated (dirty).

operation status information as shown in Table 4.3.[3] In the status code VTM0 indicates success (asserted) or error (unasserted) and when VTM0 indicates success VTM1 indicates cache line clean/dirty state information. A cache line is "dirty" if the memory copy of this data is not up-to-date (ie. a store into this block has not yet propagated to the memory); otherwise, it is "clean". During a write for invalidation operation (indicated by a NuBus write with snooping activated) VTM0 and VTM1 are not used after the START cycle and VACK is not used at all.

4.4.1. Late Finishing Inter-cache Operation Complication

The SpurBus design expects inter-cache operations to be faster than the NuBus operations. Unfortunately, this is not ensured.

If the NuBus portion of a snooping read completes before the inter-cache portion then it is possible for the NuBus datapath to be reused immediately. This occurs because arbitration is overlapped with data transfer and the winner of arbitration delays until the ACK cycle of the current operation before beginning the next.

SpurBoards, participants in SpurBus, are required to delay a subsequent operation until both the ACK and VACK cycle are complete (if the inter-cache datapath was used). This means that the only operations that might be started before the inter-cache transfer completes are those that are not SpurBus. These will not use the inter-cache datapath so no collisions will occur; however, there is still the complexity of two distinct simultaneous operations. The only foreseen operations on the NuBus during an inter-cache operation are device interrupts (such as DMA complete).

[3] There needs to be a timeout on inter-cache transfers paralleling the system timeout on transfers on the NuBus datapath.

5. Mechanical Specifications

5.1. Boards

The mechanical structure of NuBus/SpurBus boards follows the "Eurocard" design. They are triple height IEC boards (9U high where U = 1.74in) and have triple depth (11.024in).

5.2. Backplane Connectors

There are three 603-2-IEC-C096-M connectors mounted on each board. These are referred to as P1, P2 and P3 from top to bottom respectively. NuBus defines the pins on P1. SpurBus additionally defines some of the P2 pins. See Tables 5.1 and 5.2 for the pin assignments.

Pin/Row	Col. A	Col. B	Col. C	Pin/Row	Col. A	Col. B	Col. C
1	-12	-12	RESET/	17	AD23/	GND	AD22/
2	GND	GND	GND	18	AD25/	GND	AD24/
3	SPV/	GND	+5	19	AD27/	GND	AD26/
4	SP/	+5	+5	20	AD29/	GND	AD28/
5	TM1/	+5	TM0/	21	AD31/	GND	AD30/
6	AD1/	+5	AD0/	22	GND	GND	GND
7	AD3/	+5	AD2/	23	GND	GND	RSVD/
8	AD5/	-5	AD4/	24	ARB1/	-5	ARB0/
9	AD7/	-5	AD6/	25	ARB3/	-5	ARB2/
10	AD9/	-5	AD8/	26	ID0/	-5	ID1/
11	AD11/	-5	AD10/	27	ID3/	-5	ID2/
12	AD13/	GND	AD12/	28	ACK/	+5	START/
13	AD15/	GND	AD14/	29	+5	+5	+5
14	AD17/	GND	AD16/	30	RQST/	GND	+5
15	AD19/	GND	AD18/	31	GND	GND	GND
16	AD21/	GND	AD20/	32	+12	+12	CLK/

Table 5.1: P1 Connector Assignments
The P1 connector on a NuBus board has these pin assignments. The connector is type 603-2-IEC-C096-F. Column B is optional, allowing a 64 pin minimum power subset.

Pin/Row	Col. A	Col. B	Col. C	Pin/Row	Col. A	Col. B	Col. C
1	-12	-12	-	17	VAD23/	GND	VAD22/
2	GND	GND	GND	18	VAD25/	GND	VAD24/
3	-	GND	+5	19	VAD27/	GND	VAD26/
4	-	+5	+5	20	VAD29/	GND	VAD28/
5	VTM1/	+5	VTM0/	21	VAD31/	GND	VAD30/
6	VAD1/	+5	VAD0/	22	GND	GND	VAD32/
7	VAD3/	+5	VAD2/	23	GND	GND	GND
8	VAD5/	-5	VAD4/	24	-	-5	-
9	VAD7/	-5	VAD6/	25	-	-5	-
10	VAD9/	-5	VAD8/	26	-	-5	-
11	VAD11/	-5	VAD10/	27	-	-5	-
12	VAD13/	GND	VAD12/	28	VACK/	+5	-
13	VAD15/	GND	VAD14/	29	+5	+5	+5
14	VAD17/	GND	VAD16/	30	-	GND	+5
15	VAD19/	GND	VAD18/	31	GND	GND	GND
16	VAD21/	GND	VAD20/	32	+12	+12	-

Table 5.2: P2 Connector Assignments
The P2 connector on a SpurBus board has these pin assignments. The connector is type 603-2-IEC-C096-F. Column B is optional, allowing a 64 pin minimum power subset.

Pin/Row	Col. A	Col. B	Col. C	Pin/Row	Col. A	Col. B	Col. C
1	-	-	-	17	-	-	-
2	-	GND	-	18	-	-	-
3	-	GND	-	19	-	GND	-
4	-	-	-	20	-	-	-
5	-	+5	-	21	-	-	-
6	-	+5	-	22	-	-	-
7	-	+5	-	23	-	GND	-
8	-	-	-	24	-	-	-
9	-	-	-	25	-	-	-
10	-	-	-	26	-	-	-
11	-	-	-	27	-	-	-
12	-	GND	-	28	-	+5	-
13	-	-	-	29	-	-	-
14	-	-	-	30	-	GND	-
15	-	-	-	31	-	GND	-
16	-	GND	-	32	-	-	-

Table 5.3: P3 Connector Assignments
The P3 connector on a NuBus board should have these pin assignments. The connector is type 603-2-IEC-C096-F.