Titan System Manual

Michael J. K. Nielsen
Digital Equipment Corporation
Western Research Laboratory
100 Hamilton Avenue
Palo Alto, CA 94031

2 August 1988

Acknowledgments

David Goldberg wrote the original version of the architecture and instruction set verbal descriptions in this document. Jud Leonard wrote the original coprocessor and Titan I/O bus specifications. Russell Kao wrote the original memory and I/O system descriptions. David Boggs provided the TNA description. I merged these separate documents into one system manual and provided the detailed description of the instruction set and its execution, as well as updating, correcting, and enhancing the general description and Titan I/O bus description.

Mike Nielsen

Preface: Titan History and Players

The Titan project was begun as the initial project of the Western Research Laboratory in April of 1982. The most obvious milestones were the following:

April	1982	Project begins
March	1983	CPU logic simulation
October	1984	CPU executing with *toy* memory system
May	1985	CPU executing with real memory system
June	1985	I/O starts working
December	1985	Complete system with all I/O running Unix

These are the major hardware milestones. These were accompanied by a much longer list of software milestones that were very much in parallel with, sometimes slightly ahead, sometimes slightly behind, the hardware milestones.

The CPU is partitioned into four large boards. Neil Wilhelm designed the data path, instruction cache, and data cache trio while Jud Leonard designed the floating point coprocessor board. Russell Kao designed the main memory system and the two control boards and the array board for that memory system. Jud Leonard designed the I/O bus that was also implemented by the memory controller boards. Earl Devenport designed the packaging and was instrumental in debugging. Mike Nielsen debugged and redesigned so that everything eventually played together.

There were a number of other contributors, especially in software, but the people above deserve special mention for their hardware efforts.

Forest Baskett

1. Introduction

This document describes the hardware architecture, software interface, instruction set, I/O bus, and I/O adaptors of the Titan system. Chapter 2 provides an overview of the Titan system organization and a description of the function of the various modules within a system. Chapter 3 presents the special registers in the processor and memory controller that are available to the operating system to manage processes and the memory and I/O systems. The Titan instruction set is described in Chapter 4 in sufficient detail for compiler writers. Chapter 5 presents the logical, electrical, and physical specifications of the Titan I/O bus. Chapters 6, 7, 8, 9, and 10 briefly describe the Clock/Scan Module, Titan Memory Adaptor, Titan Disk Adaptor, Titan Network Adaptor, and Titan Fiber Adaptor, respectively.

The document corresponds to revision 3 of the processor, and revision 2 of the memory and I/O system.

2. Hardware Architecture

Titan is a high-performance, 32-bit scientific workstation, consisting of a central processor, memory, disk storage, and network interface. The first implementation of Titan is in 100K ECL, with a 45 nanosecond cycle time and 13 cycle cache miss penalty. The central processor has a four-stage pipeline with a peak instruction issue rate of one per cycle. Due to cache misses and pipeline stalls, a new instruction is typically issued every 1.5 cycles.

System organization and interconnection is shown in Figure 2-1. This logical partitioning of the system into modules is also reflected in the physical partitioning of the system into boards. The following sections briefly describe the function of each module within the system.

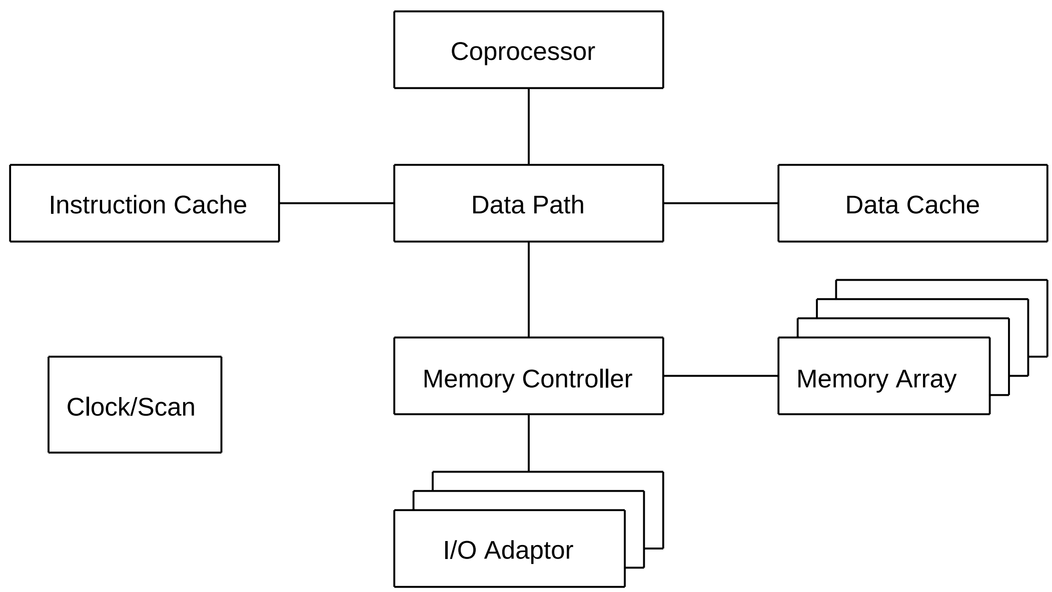

Figure 2-1: Titan System Architecture

2.1 Data Path

The data path module implements the register file, operand fetch, execution units, memory controller interface, and control logic portions of the processor.

2.1.1 Pipeline Stages

Stage	Function
IF	Instruction fetch
DO	Instruction decode and operand fetch
EX	Function unit execution
WR	Result put away

Figure 2-2: Titan Pipeline Stages

The Titan processor implements a load/store architecture to access its data cache; all other instructions use register operands. Figure 2-2 names the four stages of the Titan processor pipeline. The four stage pipeline allows a new instruction to be issued every cycle, with every instruction proceeding sequentially through the pipeline. Cache hits complete in one cycle, so every instruction completes in four cycles in the absence of stalls. If a cache miss occurs, the entire pipeline is stalled until the cache miss is resolved. Pipeline stalls due to resource contention stall only the earlier stages of the pipeline until the resource contention ends.

Execution of four instructions is shown in Figure 2-3, illustrating the overlapped execution of instructions. During the DO stage of instruction 2, the value of R1 is in the WR stage, and the value of R2 is in the EX stage. Bypass paths exist that allow the DO stage to obtain operands from these stages, so that instruction 2 is not stalled. However, the address calculation during the DO stage of instruction 3 does not have a bypass path from the EX stage and must stall for one cycle. Refer to Section 4.16 for further discussion of pipeline stalls.

```
0: r1 := (Base)      IF  DO  EX  WR
1: r2 := (Offset)        IF  DO  EX  WR
2: r3 := r1 + r2             IF  DO  EX  WR
3: r4 := (0[r3])                 IF  DO  DO  EX  WR
```

Figure 2-3: Overlapped Instruction Execution

Figure 2-4 shows a simplified representation of the data path pipeline along with the stage boundaries. The figure omits the stage bypass paths, special registers and control logic.

The IF stage is primarily in the instruction cache. The data path provides new instruction cache address register (ICAR) values in the case of branches, otherwise, the ICAR increments every cycle.

The DO stage contains the instruction decoder, register file, and operand fetch logic. The register file is organized as 4 banks of 64 registers. Normally, only one bank of registers is available to a given process; multiple banks allow rapid context switches between processes. There are two identical sets of register files to allow parallel reads of two different registers. The operand fetch logic selects operands from the register file, special registers, or pipeline bypass busses and loads the A register and the B register.

The EX stage of the data path contains an ALU, a shifter, the coprocessor interface, and the data cache interface. The ALU performs 32-bit integer add, subtract, and logical functions. The shifter can perform a 0- to 31-bit shift of a 64-bit operand followed by a 1- to 32-bit masking operation, as well as byte extract functions. The functional units get their operands from the A and B registers, and load their result into the R register.

The WR stage writes either the register file, special registers, or the data cache with the contents of the R register. The register file is multiplexed between the DO and WR stages with the DO stage performing a read on the first half of a cycle, and the WR stage performing a write on the second half of a cycle.

Note that each set of register file RAMs maintains byte parity. The register files are not initialized when the processor is rebooted, so all registers must be written to initialize their parity before CPU parity checking is enabled.

2.1.2 Instruction Sequencing

When Titan executes a branch instruction the instruction pipeline is not stalled or flushed. Instead the next instruction in line is executed normally, and if the branch condition is satisfied the branch takes effect on the instruction following it. As an illustration, if (*w x y z*) is a sequence of instructions in memory, and if *w* is an unconditional branch to instruction z, the actual execution sequence will be (*w x z*). If in addition *x* is a branch to *w*, then it will be (*w x z w*). This behavior is common in microengine instruction sets, but not normally seen in macrocode.

The delayed branch results because the IF stage will already have fetched the instruction after the branch before the DO stage has determined that it is in fact a branch. Rather than stalling the pipeline when a branch is detected, the

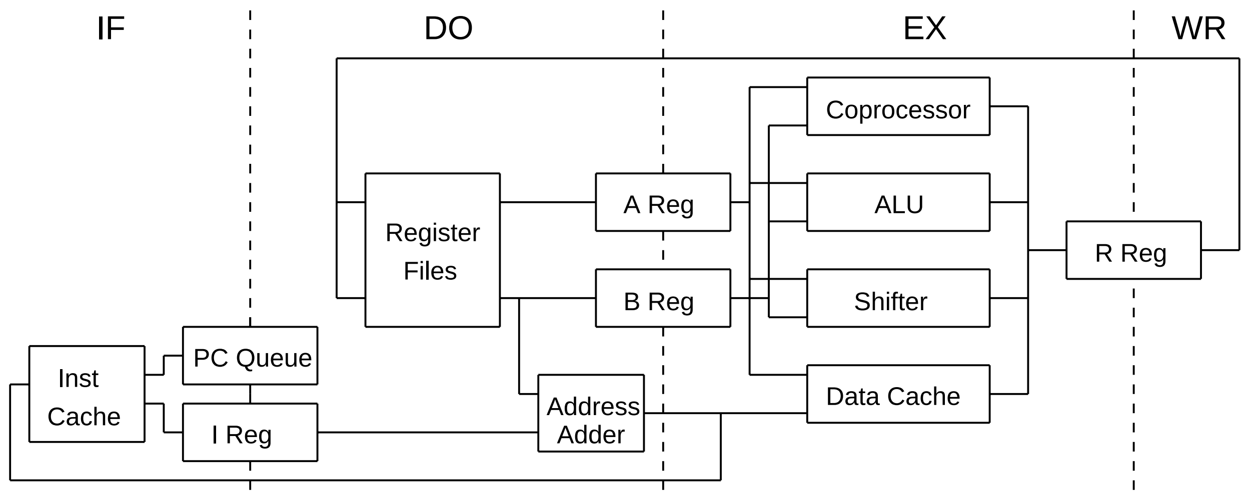

Figure 2-4: Simplified Data Path Pipeline

instruction after the branch is allowed to continue through the execution pipeline. Compilers can generally schedule useful instructions in delayed-branch slots, avoiding any pipeline-induced branch penalty, or at worst, place a *null* instruction in the delayed-branch slot.

To allow restarting the instruction stream after interrupts, there is a register containing the address of the instruction in each pipeline stage as shown in Figure 2-5. The PC is mapped to register R0 for operand fetches, allowing pc-relative addressing; writes to R0 are discarded. When an interrupt occurs, the WR stage completes, the addresses of the instructions in the EX and DO stages are saved for the operating system, and the IF stage is aborted. Use of this *pc queue* is discussed further in Sections 4.2.2, 4.3, 4.4, and 4.13.

Register	Stage
ICAR	IF
PC	DO
PC1	EX
PC2	WR

Figure 2-5: Pipeline Instruction Address Registers

2.1.3 Memory Interface

The data path is connected to the memory controller via two uni-directional, 32-bit data busses. The data path checks parity of the bus transferring data from the memory controller to the data path; the memory controller checks parity of the bus transferring data from the data path to the memory controller. The control logic for the caches, and thus the control signals to and from the memory controller, is implemented on the instruction cache module.

2.2 Caches

There are separate, independent data and instruction caches that operate in parallel. Each cache contains a translation buffer (TB) and a real address cache (RAC). The Titan page size is 4K bytes; 256 lines of 4 32-bit words.

The real address cache is 16K bytes in size, organized as 4 sets of 256 lines of 4 words, is write-back, and uses a random replacement policy. The real address cache is managed by the hardware, with the exception of the privileged flush instruction to force write-back and invalidation of the caches as required for process switches and DMA for

I/O.

The translation buffer contains 1024 virtual-to-real page-number translations, organized as 2 sets of 512 translations. The translation buffer is managed by the operating system, via a translation fault register (TFR) that records the virtual page number that caused the translation fault, and special instructions to read and write the translation buffer. Each translation buffer tag contains a writable-bit that enable data pages to be marked read-only, causing store instructions to those pages to generate traps.

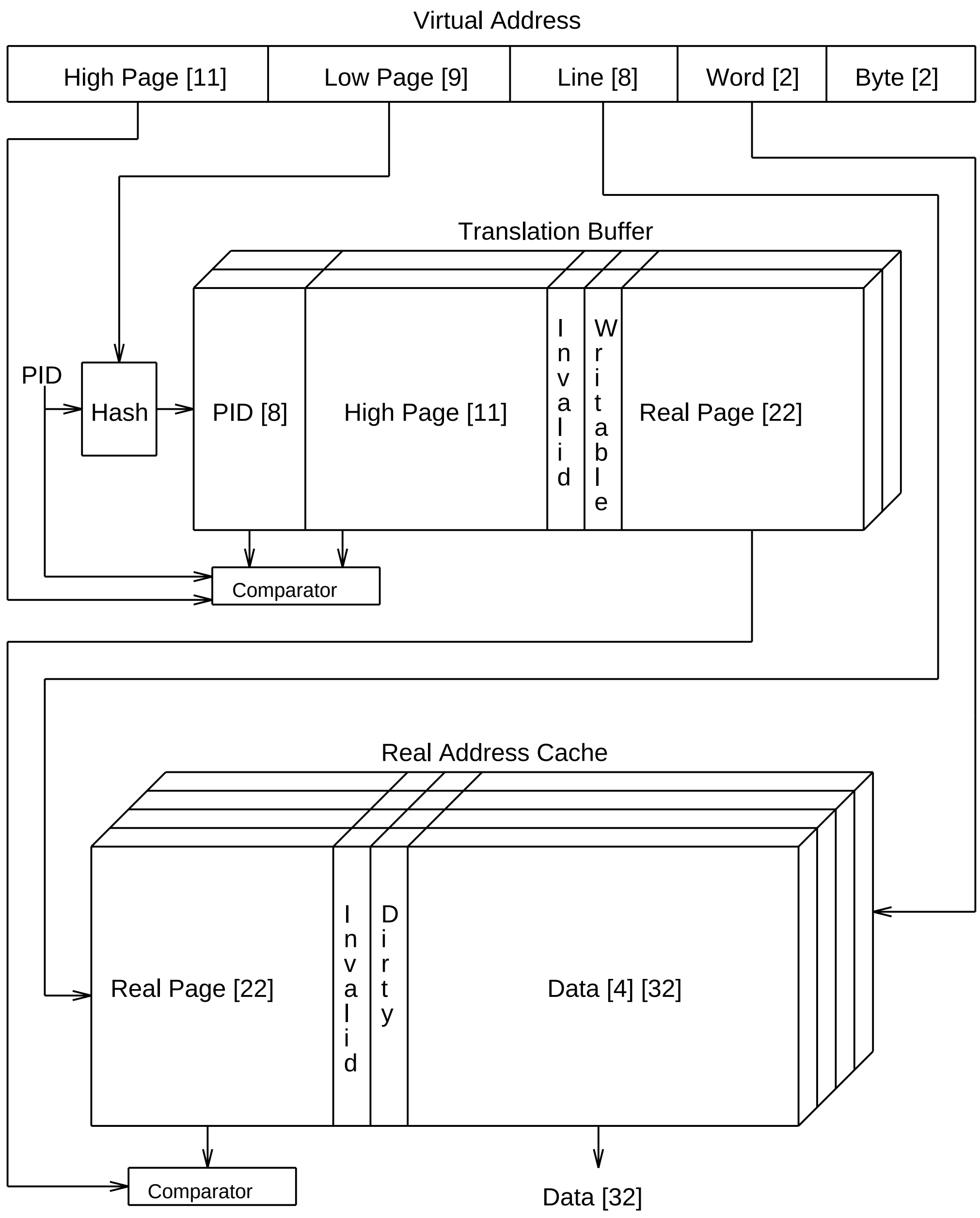

Figure 2-6: Titan Address Translation

2.2.1 Byte Versus Word Addressing

To facilitate use of software developed for byte-addressed machines, the Titan effective address calculation treats the quantity in the address register as a byte address. However, internally, the Titan is implemented as a word-addressed machine, i.e., the caches, memory controller, and I/O adaptors all deal with word addresses. During effective address calculations, the processor shifts the address register contents right by two bits to convert it from a byte address to a word address before adding the 16-bit, signed displacement. This implies that the two most significant bits of the resulting effective address will always be zero (ignoring negative addresses). Furthermore, since the hardware discards the two least significant bits of the address register, all instructions, and all data referenced by a single load or store instruction, must be word aligned.

This implies that the Titan has a 2^{32}-byte virtual address space, and a 2^{34}-byte real address space.

In this document, all addresses and discussions of addresses will be presented as byte addresses, unless otherwise noted.

2.2.2 Address Translation

Memory addresses generated by most programs are virtual, and the behavior specified by an instruction referencing memory occurs only if a valid translation exists in the appropriate translation buffer, and no protection violation occurs. When a reference faults, the instruction is suppressed and a trap occurs so that kernel software can establish a valid translation buffer entry and resume execution of the program.

Figure 2-6 show the virtual-to-real address translation and cache addressing. The Titan page size is 4K bytes, so the low-order 12 bits of an address are the position within a page, and the high-order 20 bits are the page number. The *low-page* field is used to select a pair of page table entries. If the PSW *pid* field[1] and the *high-page* field matches one of these entries, and if the *invalid* bit of this entry is not set, a valid translation exists. If the translation buffer has two valid translations for a virtual address, the result of referencing that address is undefined.

Meanwhile, the *line* field, is used to index into the cache. If one of the four selected entries has a tag field that matches the 22-bit real page number obtained from the translation buffer, and if the invalid bit of that entry is not set, then the desired data is in the cache. Each cache entry is 4 words long, so the *word* field is used to select the appropriate word of the cache entry. The *byte* field is not used by the hardware.

This description is simplified in one respect. If mapping occured exactly as described, there is a potential problem, since the translation buffer only has a set size of 2. If many processes reference the same virtual addresses, the translation buffer would suffer a low hit rate. Consequently, the *low-page* field of an address is hashed with the *pid* field of the PSW to compute the index into the translation buffer. Figure 2-7 shows the hashing function used.

```
low-page and 256  +  (low-page + 4 * pid) mod 256
```

Figure 2-7: TB Hashing Function

2.2.3 Cache Parity

The translation buffer tag and data RAMs, and the real address cache tag and data RAMs, have byte parity. The translation buffer entries are not initialized when the processor is rebooted, so all entries must be written before CPU parity checking is enabled. The real address cache entries are automatically initialized as part of the processor reboot sequence.

---

[1]Refer to Section 3.2.

2.3 Coprocessor

The Titan coprocessor performs integer multiply and divide, and floating point arithmetic operations. These operations take place concurrently with normal instruction processing, except that the Titan processor stalls to wait for the coprocessor to finish computing a result that it needs.

Coprocessor operations use as operands a pair of processor registers and an accumulator in the coprocessor itself. The accumulator consists of a sign bit, a 16-bit exponent register, and a 64-bit fraction register. Integer operations in the coprocessor manipulate the fraction register as an unsigned quantity. Floating point operations affect the entire accumulator.

The coprocessor also contains a time-of-day clock and an interval timer. The time-of-day clock is incremented every machine cycle. The interval timer generates an interrupt up to 20 milliseconds in the future. The operating system can use these timers to maintain system clocks.

Refer to Section 4.12 for further discussion of floating point formats and coprocessor operations. Refer to Section 4.16 for coprocessor instruction timing.

2.4 Memory Controller

The memory controller maintains the memory arrays and serves as an interface between the processor and the I/O adaptors. The memory controller supports from 1 to 4 memory modules of 32M bytes each, performing the ECC generation and checking as well as RAM refresh functions. In the Titan, all memory transactions are in units of 4 word lines. Thus a single memory module is comprised of 4 memory array boards that operate in parallel.

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

On the Titan I/O bus side, the memory controller performs DMA read or write operations for I/O adaptors. The memory controller also performs reads or writes to I/O adaptor registers in response to processor I/O instructions. The memory controller maintains several registers for this purpose, discussed further in Section 3.8. The Titan I/O bus supports up to 7 I/O adaptors. Refer to Chapter 5 for detailed description of the I/O bus.

2.4.1 Memory configuration

The 32-bit real address space consists of 512 32MB modules, each of which may or may not be occupied by physical memory. Vacant modules will always read zeros regardless of what is written to them, while occupied slots behave as memory. The current memory controller implements a 25-bit physical word address space, i.e., modules [0..3].

Since the memory controller ignores the high-order 7 bits of real addresses, references to modules [4..511] map over modules [0..3]. However, address parity included in the ECC computation causes single-bit ECC errors if memory is written in one group of 4 modules and read in another group of 4 modules.

Figure 2-8 lists the module real address ranges.

In order to determine which module slots are populated, bootstrap code can write the module number into the first word of each of the 4 module address ranges. When those words are then read in a second pass, those slots retaining the correct slot number are backed up by physical memory. The data cache must be explicitly flushed via the flush

Module	I/O Word Address	CPU Byte Address
0	[0000000..07FFFFF]	[0000000..1FFFFFF]
1	[0800000..0FFFFFF]	[2000000..3FFFFFF]
2	[1000000..17FFFFF]	[4000000..5FFFFFF]
3	[1800000..1FFFFFF]	[6000000..7FFFFFF]

Figure 2-8: Memory Module Address Ranges

instruction before performing the second pass.

Note that double and single bit ECC error halts and interrupts should be disabled during the memory configuration poll. After module population has been determined, those modules should be written in their entirety before enabling ECC.

The process of reading a vacant slot will cause a single bit error to be detected. Writes to vacant slots cause no error indication. Because the parity of the line address is included in the ECC code, if the same region of memory is written using one address and read using a different address a single bit ECC error may be detected although the data will be correct.

2.4.2 Bootstrap Prom

If the *rom* bit in the program status word is asserted, then the low 128K bytes of the address space are mapped into the Boot-Prom for CPU and DMA read references. Writes are unaffected by the *rom* bit; they modify the main memory shadowed by the Boot-Prom.

Accesses to the Boot-Prom are extremely restricted. The only permissible accesses are non-overlapped reads and writes. IO and CPU activity must not overlap, and the CPU must not perform dirty misses in which the read operation accesses the Boot-Prom.

The contents of the Boot-Prom should be transferred into memory in a loop that reads one page (4K bytes) into the CPU data cache and then flushes that page of the data cache back out into memory. Flushing after each page is read into the data cache will prevent dirty misses from occurring.

Note that the scan-chain built into all Titan modules allows diagnostic programs to *boot* directly from pre-loaded main memory, bypassing the Boot-Prom.

2.4.3 I/O Configuration

Every I/O adaptor is required to respond with an adaptor type code when an I/O read is performed to it at address FFFFFFFF. This allows the operating system to poll all seven I/O slots and determine the number and type of I/O adaptors present. Figure 2-9 shows the data returned by currently implemented I/O adaptors.

Type Code	I/O Adaptor
00000000	Empty slot
00000001	Reserved
00000002	Disk (MSCP/SDI)
00000003	Reserved
00000004	Network (Ethernet)
00000006	Fiber (100Mbs Manchester)

Figure 2-9: I/O Adaptor Types

Note that I/O bus parity error halts and interrupts should be disabled during the I/O configuration poll because reading empty slots will cause I/O bus parity errors.

2.4.4 I/O Lock

A special case occurs when an I/O read operation is performed on nonexistent slot 0. Because the memory system does not support atomic read-modify-write operations for I/O adaptor DMA, the I/O bus includes a *lock* signal in order to implement mutual exclusion between entities on the I/O bus. When an I/O read to slot 0 is performed with an odd value in the I/O write data register, the processor tests and attempts to acquire the lock in one atomic operation. The old value of the lock is recorded in the LSB of the I/O read data register and the lock is acquired only if it is free. When an I/O read to slot 0 is performed with an even value in the I/O data register, the processor tests the lock and if it is in possession of the lock, releases it.

2.5 Clock/Scan

The clock/scan module distributes the system clock to all other modules in the system as well as providing test and diagnostic access to processor, memory, and I/O modules via *scan-chains* and clock *single-step* functions. The clock/scan module has a simple Ethernet interface to allow remote restart of a Titan, as well as manipulation of the system's internal state for diagnostic purposes.

In addition to the Ethernet interface, the clock/scan module supports maintenance panel reset and auto-boot switches, as well as a halt led that is on if either the processor or memory controller is halted.

Other than clocks and diagnostic scan signals, the clock/scan module drives only the processor reset signal. There is no direct processor or memory controller access to the clock/scan module.

Refer to Chapter 6 for detailed discussion of clock/scan module functions.

3. Software Interface

3.1 Kernel/User Mode

In many operating systems, there is a small subset of code that is responsible for managing especially critical low-level hardware functions. We refer to this code as the kernel, and in Titan, it is responsible for many functions which are performed by microcode sequences in conventional machines. In particular, it receives and dispatches interrupts from I/O devices, handles the transitions between processes (including the user/operating system switch), maintains the contents of the address translation buffers, and ensures the coherence of cache and main memory contents when DMA I/O is performed. It is expected that the core of the operating system will run in kernel mode, which is a state with traps disabled and privileged instructions enabled, while the bulk of the operating system will run in user mode with privileged instructions enabled. A user program can change its state to kernel mode by executing the trap instruction.

3.2 Processes

At any instant, there is only one process running. It has a 32-bit virtual memory address space and 64 registers. A page of memory may be flagged as read-only, and this can only be modified by a privileged instruction. We expect that there are a number of processes actively working on behalf of a user, and that the processor will be switched frequently from one process to another. Therefore, the processor hardware incorporates 4 sets of registers which can be assigned dynamically to active processes in order to avoid saving and restoring them on every context change. Furthermore, each address translation stored in the translation buffers is tagged with an 8-bit process identification code, so as to minimize the frequency that the translation buffers need to be flushed. The switching of register sets and process identification is performed entirely under kernel software control.

3.3 Program Status Word

```
|1|1|1|1|1|<   8    >|<2>|<2>|1|1|1|1|1|1|1|1|1|<    6    >|
+-+-+-+-+-+----------+---+---+-+-+-+-+-+-+-+-+-+-----------+
|r|p|k|p|p|          |   |   |i|d|e|i|i|d|w|c|u|           |
|o|r|r|r|a|   pid    |ra |rb |m|m|x|l|t|t|p|o|s|  literal  |
|m|e|n|v|r|          |   |   |p|p|t|l|f|f|v|p|r|           |
+-+-+-+-+-+----------+---+---+-+-+-+-+-+-+-+-+-+-----------+
```

Figure 3-1: Program Status Word

The processor state is controlled by a 32-bit program status word as shown in Figure 3-1. It can be read and written only with privileged instructions. The fields are:

rom — If set, then the lowest physical addresses are mapped to the Boot-Prom. This is used when booting the processor.

pre — This is the *pre-kernel* bit. When written, the value will be taken on by the kernel bit on the next cycle.

krn — If set, the processor is in kernel mode; external and coprocessor interrupts are ignored and privileged instructions are enabled. Writing into *krn* has no effect; its value is the value that the pre-kernel bit had on the previous cycle.

prv — If set, privileged instructions are enabled; they will not cause an illegal instruction trap. Privileged instructions are always enabled in kernel mode, no matter what the state of *prv*.

par — If set, processor parity checking is **not** performed.

pid — The current process id, used in the translation buffer address hashing for virtual-to-real

	address translation.
ra	This field indicates to which of the four register banks instruction fields *ra* and *rc* refer.
rb	This field indicates to which of the four register banks instruction register field *rb* refers. It is possible to transfer information from one bank to another with a register move instruction if PSW *ra* and *rb* fields are different.
imp	If set, virtual-to-real address translation is **not** performed for instruction references.
dmp	If set, virtual-to-real address translation is **not** performed for data references.
ext	If set, an external (memory controller) interrupt is pending.
ill	If set, a special, set pc-queue, kernel exit, or flush instruction was executed in user mode without privileged instructions enabled. The *ill* bit is also set if an abort, undef1, or undef2 instruction is executed in user mode.
itf	If set, an instruction translation fault occurred.
dtf	If set, a data translation fault occurred.
wpv	If set, a store to a read-only page occurred.
cop	If set, a coprocessor arithmetic trap occurred and/or an interval timer interrupt is pending.
usr	If set, a user trap instruction was executed.
literal	If *usr* is set, the *literal* field contains the the *literal* field from the user trap instruction. If the *usr* bit is clear, the value of the *literal* field is undefined.

Note that more than one of the *ext*, *ill*, *itf*, *dtf*, *wpv*, *cop*, or *usr* bit can be set, indicating that multiple trap conditions occurred.

Note that only the *rom*, *pre*, *prv*, *par*, *pid*, *ra*, *rb*, *imp*, *dmp* fields can be written. The *krn* bit shadows the *pre* bit. The *ext*, *ill*, *itf*, *dtf*, *wpv*, *cop*, *usr*, and *literal* fields are updated by the processor during every user mode cycle, and held during every kernel mode cycle.

Note that all registers of all register banks and all translation buffer entries should be written to initialize their parity before enabling processor parity checking. All real address cache entries are automatically written during the processor reset sequence initializing their parity.

3.4 Processor Reset

When the processor is reset, the *rom*, *krn*, *pre*, *par*, *imp*, and *dmp* bits are set, the *prv* bit is cleared; other fields are undefined. The startup code should write the PSW as its first operation after a processor reset to initialize the *pid*, *ra*, and *rb* fields.

The ICAR is set to 00000000 and the processor executes a cache clear sequence that successively invalidates every line of the RAC, writing both the tag and data entries for each line of all 4 RAC sets in parallel. The processor then starts executing instructions at byte address 00001000 (I/O word address 00000400). Thus the startup code should start at address 00001000.

During this cache clear sequence, the processor asserts a reset signal to the memory controller, causing it to reset itself and the I/O adaptors. The cache clear sequence lasts a minimum of 1024 cycles.

The startup code should then write all registers of all banks and invalidate all translation buffer entries to initialize their parity. Processor parity checking should then be enabled.

The startup code should then determine the number and type of I/O adaptors present and enable CPU and I/O bus parity checking.

After determining the amount of physical memory, the startup code should write all of the physical memory to initialize ECC. ECC correction should then be enabled.

3.5 Traps

Upon any of a set of special circumstances, the processor interrupts the normal sequence of instruction execution, and forces 00000000 as the new ICAR. Thus the operating system interrupt handler starts at real address 00000000.

The PSW *krn*, *pre*, *imp*, *dmp* bits are set, other fields are not changed. The *ext*, *ill*, *itf*, *dtf*, *wpv*, *cop*, and *usr* bits should be used to determine appropriate trap and interrupt handling. The PSW may have more than one of these bits set if multiple traps occurred.

Note that user programs can still start at virtual address 0 as mapping is automatically disabled when an interrupt or trap occurs.

The pc-queue (PC2 and PC1) is frozen with the address of the two instructions that were aborted by the trap or interrupt. The kernel exit instruction restarts the pipeline from these saved addresses. Two addresses are required in case the instruction previous to the trap point was a branch instruction. Refer to Sections 2.1.2, 4.2.2, 4.3, 4.4, and 4.13 for further discussion of the pc-queue.

3.6 Processor Halt Conditions

If an instruction translation fault, data translation fault or write protection violation occurs when the processor is in kernel mode, then the processor halts. Executing an abort, user trap, undef1 or undef2 instruction when the processor is in kernel mode also causes the processor to halt. If the PSW *par* bit is cleared and a processor parity error occurs, the processor halts regardless of kernel/user mode.

Once the processor halts, it must be externally reset.

3.7 Coprocessor Registers

Coprocessor registers are discussed in Section 4.12.

3.8 Memory Controller Registers

The memory controller is the only hardware component that interconnects the processor, main memory, and I/O adaptors. Three classes of communication occur between these three:

- I/O adaptors access main memory via DMA reads and writes.
- The processor accesses main memory via clean miss, dirty miss, and flush cache events.
- The processor accesses control registers of the I/O adaptors via IoRead and IoWrite special instructions.

The memory controller contains a number of control registers related to the I/O and memory systems that are described below. Special I/O instructions allow the processor to access the memory controller to manipulate the memory and I/O systems. Refer to Sections 4.2.14, 4.2.15, 4.2.16, and 4.2.17 for discussion of instructions to manipulate the memory controller registers. Figure 4-8 lists the memory controller register addresses.

3.8.1 I/O Address Register

```
|<                              32                            >|
+--------------------------------------------------------------+
|                           IoAddress                          |
+--------------------------------------------------------------+
```

Figure 3-2: I/O Address Register

The IoAddress register is for diagnostic purposes only. This register contains the address sent by the last IoWrite or IoRead instruction. The IoAddress register is read only.

Note that I/O addresses are interpreted independently by each I/O adaptor to select internal registers or memory. There is no relation between I/O addresses and memory addresses.

3.8.2 I/O Read Data Register

```
|<                              32                            >|
+--------------------------------------------------------------+
|                           IoReadData                         |
+--------------------------------------------------------------+
```

Figure 3-3: I/O Read Data Register

The IoReadData register contains the data received in response to the last IoRead instruction. The contents of this register are destroyed by IoWrite instructions, and is typically read immediately after issuing an I/O read. This register is read only.

3.8.3 I/O Write Data Register

```
|<                              32                            >|
+--------------------------------------------------------------+
|                          IoWriteData                         |
+--------------------------------------------------------------+
```

Figure 3-4: I/O Write Data Register

The IoWriteData register holds the data to be transmitted to an I/O adaptor by the IoWrite instruction, and is typically written just before issuing the I/O write. This register is both readable and writable by the processor.

Note that when I/O device drivers specify addresses to I/O adaptors, they must be word addresses. Device driver software must explicitly convert byte addresses to word addresses.

3.8.4 I/O Status Register

```
|1|<                           28                     >|< 3 >|
+-+----------------------------------------------------+-----+
|a|                            0                       | slot|
+-+----------------------------------------------------+-----+
```

Figure 3-5: I/O Status Register

The *slot* field of the IoStatus register specifies which of the 7 I/O slots (numbered 1 to 7) will be affected by subsequent IoRead and IoWrite instructions. Refer to Figure 5-12 for the position of I/O slots in the backplane. This field may be read or written by the processor.

The *a* (ack) field contains the value of the I/O bus *ack* signal at the end of the last IoRead or IoWrite instruction.

The act bit is set by an I/O adaptor as an indication that it has received and processed an I/O request. Certain adaptors can enter a state in which they are busy and are temporarily unable to process new requests. In this case these adaptors may ignore the request and return a zero ack bit. Programs driving these adaptors are responsible for testing the ack bit after each IoRead or IoWrite operation and retrying the operation if the ack bit is zero. The a field is read only.

3.8.5 Event Register

```
|1|<     7     >|<                 20                 >|<  4  >|
+-+-------------+--------------------------------------+-------+
|0|   ioInt     |                  0                   | hErrs |
+-+-------------+--------------------------------------+-------+
```

Figure 3-6: Event Register

The event register is used to report I/O interrupt requests and certain error conditions detected by the hardware. The *ioInt* field contains the value of each of the 7 I/O adaptor interrupt request lines. Slot 1 is represented by leftmost bit; slot 7, the rightmost. The *ioInt* field is read only. The *hErrs* field is used to report hardware errors as shown in Figure 3-7. The *hErrs* bits may be cleared by writing a word containing ones in the corresponding bit positions. Note that the *hErrs* bits are not automatically cleared when the machine is reset; the operating system should clear them as part of its initialization by writing 0000000F to the event register.

Position	Description
bit 3	Single bit memory error
bit 2	Double bit memory error
bit 1	CPU bus parity error
bit 0	I/O bus parity error

Figure 3-7: Hardware Error Bits

3.8.6 Enable Register

```
|1|<     7     >|<           15          >|1|<  4  >|<  4  >|
+-+-------------+-------------------------+-+-------+-------+
|0|   ioIntEn   |            0            |c|errHalt| errInt|
+-+-------------+-------------------------+-+-------+-------+
```

Figure 3-8: Enable Register

If a bit in the event register is set and the corresponding bit in the enable register *errInt* field is set then the CPU receives an external interrupt request. If a hardware error occurs and the corresponding bit in the *errHalt* field of the enable register is set then the memory controller will halt; this will cause the processor to stall indefinitely when the next cache miss occurs. If the correctionEnable *c* bit is set then memory error correction is enabled. Note that ECC generation and checking is always performed, the *c* bit only controls whether or not correction is applied to read data. If an I/O adaptor generates an interrupt, and the corresponding slot has its *ioIntEn* bit set, the memory controller generates an external interrupt in the processor. Note that the IoEvent register *ioInt* field always reflects the state of I/O adaptor interrupts whether or not a given slot has interrupts enabled. The enable register is initialized to all zeros during the hardware bootstrap sequence. The enable register may be read and written by the processor.

For example, to enable interrupts for I/O slots 1 and 2, enable ECC correction, halt on parity errors and double-bit ECC errors, and interrupt on single-bit ECC errors, write C0000178 to the enable register.

3.8.7 Error Log Register

```
|<      8      >|<      8      >|<           16            >|
+---------------+---------------+---------------------------+
|    ~hiSynd    |    ~loSynd    |          errCnt           |
+-------------------------------+---------------------------+
```

Figure 3-9: Error Log Register

Main memory ECC is done on a half line basis. Main memory accesses are to a full line (4 words). If, during a memory read access, a double- or single-bit error is detected in either of the half lines, the syndrome bits of the even and odd address half lines are recorded in *hiSynd* and *loSynd*, respectively. The *errCnt* field contains the number of memory errors which have occurred since it was last reset. The *errorCount* is reset when the single-bit-memory-error bit in the event register is cleared. The ErrorLog register is read only.

```
         \ Synd[2:0]
Synd[5:3] \
           \   0    1    2    3    4    5    6    7
            +---------------------------------------
          0 |     C1   C2    0   C4    1    2
            |
          8 | C8    8    9   10   11   12   13
            |
         16 | C16  14   15   16   17   18   19
            |
         24 | 20   21    4         5
            |
         32 | C32  24   25   26   27   28   29
            |
         40 | 22   23    6         7
            |
         48 |  3   30   31
            |
         56 |
```

Figure 3-10: ECC Syndrome Decode

Figure 3-10 shows the decoding of the syndrome bits for a half-line. Note that the syndrome bits are complemented in the ErrorLog register, and that this table applies to the uncomplemented syndrome. For single-bit ECC errors, the least significant 6 bits of the syndrome indicate which bit is incorrect; the bits prefixed with a 'C' are ECC check bits. The Synd[6] bit is a parity bit over 32 data bits. It indicates which word of the half-line had the ECC error, and corresponds to the address[1] bit. The Synd[7] bit is a parity bit over 64 data bits, 7 check bits and 30 address bits. When a single bit error occurs and Synd[5:0] is equal to 000000, then one of three things happened: 1) a single-bit error in check bit 7, 2) a single-bit error in check bit 6, or 3) an address error: a location responded to more than one address. Synd[6] distinguishes between cases 1 and 2. Figure 3-11 tabulates the syndrome values for single-bit errors in the check bits.

Check Bit	Syndrome	Use
0	0x81	C1
1	0x82	C2
2	0x84	C4
3	0x88	C8
4	0x90	C16
5	0xa0	C32
6	0xc0	Word
7	0x80	Parity

Figure 3-11: Check Bit Syndromes

3.8.8 Error Address Register

```
|<                        30                        >|1|1|
+----------------------------------------------------+-+-+
|                      lineAddr                      |0|w|
+----------------------------------------------------+-+-+
```

Figure 3-12: Error Address Register

LineAddr records the address of the last double- or single-bit memory error. *W* is 1 if an I/O adaptor made the request and 0 if the processor made the request. This register is read only.

4. Instructions

There are two instruction formats, as shown in Figures 4-1 and 4-2. The fields labeled *ra*, *rb*, and *rc* refer to registers, where *a*, *b*, and *c* are integers in the range 0-63. Register r0 is special in that when it is read, it returns the value of the program counter (the virtual address of the instruction referencing r0) and when it is written, the data being written is discarded.

```
|<  4  >|<     6     >|<     6     >|<              16               >|
+-------+-----------+-----------+-------------------------------+
|opcode |    ra     |    rb     |         displacement          |
+-------+-----------+-----------+-------------------------------+
```

Figure 4-1: Load/Store/Branch Instruction Format

For the load/store/branch instruction format, an effective address is calculated as follows:

Effective-address **:= (rb >> 2) + displacement;**

The value of register *rb* is right-shifted by two bits so that software can maintain addresses of byte quantities, where the byte within a word is indicated by the least significant two bits of the address. Loads or stores of word data that is not word-aligned must be handled in software with multiple loads or stores. Similarly, all instructions must be word aligned in memory. The *displacement* is sign-extended to 32 bits. This results in a 30-bit virtual word address.

```
|<  4  >|<     6     >|<     6     >|<     6     >|<      10       >|
+-------+-----------+-----------+------------+----------------+
|opcode |    ra     |    rb     |     rc     |  miscellaneous |
+-------+-----------+-----------+------------+----------------+
```

Figure 4-2: Alu/Shifter/Coprocessor Instruction Format

For the alu/shifter/coprocessor instruction format, the result is calculated as follows:

rc := ra miscellaneous rb

Register *rc* can be the same as either *ra* or *rb*, as *ra* and *rb* are read before *rc* is modified. The function units decode *miscellaneous* to determine the function to apply on *ra* and *rb*.

For each of the following instruction descriptions, we list its name, its Titan assembly language (tasm) form, its memory format, a brief description of its operation, any restrictions that apply to the use of the instruction, and an indication of what occurs during each pipeline stage.

Mnemonic	Stage	Register
PSW	-	Program status word
ICAR	IF	Instruction cache address register
ITFR	IF	Instruction translation fault register
PC	DO	Program counter
PC1	EX	Program counter
PC2	WR	Program counter
AR	DO/EX	A operand register
BR	DO/EX	B operand register
RR	EX/WR	Result operand register
DCAR	EX	Data cache address register
DTFR	EX	Data translation fault register
AC	EX	Coprocessor accumulator

Figure 4-3: Pipeline Register Mnemonics

In the instruction descriptions, various processor pipeline registers will be used to explain the execution of the instruction. Figure 4-3 lists the mnemonics and register descriptions. Refer to Section 2.1 for further description of the pipeline registers.

```
Notation  Operation
a := b    Assign value of b to a
a + b     Arithmetic sum of a,b
a >> b    Shift a right by b bits
a << b    Shift a left by b bits
a | b     Logical OR of a,b
a & b     Logical AND of a,b
(e)       Evaluate e first
```

Figure 4-4: Arithmetic Notation

Unless otherwise noted, all numeric values are in hexadecimal in the instruction descriptions. In some figures, C-style arithmetic notation is used, as shown in Figure 4-4.

When addresses are shown in examples or text, they will be byte addresses as specified by programs. Note that addresses specified to I/O adaptors must be word addresses; e.g., in I/O write instructions. Device driver software must explicitly convert byte addresses to word addresses.

In examples of TASM code, unconditional branches (*gotos*) are shorthand notations for subroutine jumps with register *ra* equal to r0; discarding the current instruction address. Similarly, the *null* instruction is a shorthand for an alu instruction that assigns to r0; discarding the alu result. The <*number*> (and <*!number*> notation indicates that the instruction must (or must not) reside in word *number* of a memory line.

4.1 Abort

TASM Format

```
abort [ra, rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<              16              >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   0   |    ra     |    rb     |               0               |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

If the abort instruction is executed in user mode, the illegal instruction trap bit will be set in the PSW, PC2 will contain the address of the abort instruction, PC1 will contain the address of the instruction in execution sequence after the abort instruction.

If abort is executed in kernel mode, then the processor halts with the PC containing the address of the instruction in execution sequence after the abort instruction, and AR, BR with the contents of *ra*, *rb*.

Execution

DO	AR := ra, BR := rb
EX	trap
WR	-

4.2 Special Instructions

TASM Format

```
ra := special opCode[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    6   >|1|< 3 >|<    6    >|
+-------+-----------+-----------+----------+-+-----+-----------+
|       |           |           |          |w|     |           |
|   1   |     ra    |     rb    |     0    |r| op  |    misc   |
|       |           |           |          |t|     |           |
+-------+-----------+-----------+----------+-+-----+-----------+
```

Description

Special instructions allow manipulation of the PSW, pc-queue, cache translation buffers, and I/O system. The *wrt* bit specifies whether it is a read-class or write-class special operation. The *op* field specifies the hardware resource to be manipulated as shown in Figure 4-5. The *misc* field specifies resource specific operations as shown in Figure 4-6 to select a resource for reading, and Figure 4-7 to enable resource writes. The I/O special instructions obtain the I/O register from the *rb* field as shown in Figure 4-8.

Op	wrt=0	wrt=1
0	PSW	Inst Cache
1	PC-Queue	Data Cache
2	GetCtl	PSW
3	Inst Cache	SetCtl
4	Data Cache	IoRead
5	-	IoWrite
6	-	-
7	-	-

Figure 4-5: Special Instruction Resource Encoding

The following sections present specific encodings for the commonly used special instructions. Most other encodings are redundant or only useful in conjunction with scan-chain-based diagnosis of the processor.

```
|<1>|< 1 >|<  2 >|< 2 >|
+---+-----+------+-----+
|   |     |      |     |
| 0 | tag | data | cam |
|   |     |      |     |
+---+-----+------+-----+
```

Value	Tag	Data	CAM
0	TLB Col[0]	TLB Col[0]	TLB Data
1	TLB Col[1]	TLB Col[1]	TLB Tag
2	-	ICAR	RAC Tag
3	-	ICAR	TFR

Figure 4-6: Cache Read Misc Field Encoding

```
|< 2 >|<  1  >|<  1   >|<  1  >|<  1   >|
+-----+-------+--------+-------+--------+
|     |       |        |       |        |
|  0  | ~tag0 | ~data0 | ~tag1 | ~data1 |
|     |       |        |       |        |
+-----+-------+--------+-------+--------+
```

Figure 4-7: Cache Write Misc Field Encoding

Rb	I/O Register	Access	Function
01	IoAddress	R	I/O address
10	IoReadData	R	I/O read data
20	IoWriteData	RW	I/O write data
30	IoStatus	RW	I/O status
08	Event	RW	Hardware errors and I/O interrupts
28	Enable	RW	Halt, ECC, and interrupt enables
18	ErrorLog	R	ECC syndrome, memory error count
38	ErrorAddress	R	Memory error address

Figure 4-8: I/O Special Instruction Rb Field Encoding

Restrictions

The special instructions all effect operation of the PC register causing it to be invalid two instructions after the special instruction. For this reason, load, store, branch, pseudo call, and flush instructions with register *rb* equal to r0 must not reside in this instruction slot.

Special instruction don't have the hardware resource interlocks provided for most other instructions, and consequently must not have pipeline stalls during some stages of their execution. Specific constraints are listed for each instruction.

Special instructions should only be executed with interrupts disabled, i.e., in kernel mode.

Executing a special instruction in user mode causes an illegal instruction trap.

4.2.1 Read Program Status Word

TASM Format

```
ra := special ReadStatus[r0];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     ra    |     0     |     0     |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

ReadStatus loads the program status word into register *ra*.

Restrictions

The instruction following this one must not be a conditional branch that tests register *ra*.

The instruction executed two cycles later must not read r0.

Figure 4-9 shows the recommended instruction sequence.

```
r1 := special ReadStatus[r0];
null;
null;
```

Figure 4-9: Read PSW Instruction Sequence

Execution

DO	AR := PSW
EX	RR := AR
WR	ra := RR

4.2.2 Read PC-Queue

TASM Format

```
ra := special ReadPcQ[r0];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |    ra     |     0     |     43    |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

ReadPcQ loads PC2 into register *ra*. The pc-queue is advanced so that executing the instruction again will read PC1. The values loaded into PC of the pc-queue are undefined.

Restrictions

The instruction following this one must not be a conditional branch that tests register *ra*.

The instruction executed two cycles later must not read r0.

Figure 4-10 shows the recommended instruction sequence.

```
r1 := special ReadPcQ[r0];
null;
null;
```

Figure 4-10: Read PC-Queue Instruction Sequence

Execution

DO	BR := PC2, PC2 := PC1
EX	RR := BR
WR	ra := RR

4.2.3 Write Program Status Word

TASM Format

```
ra := special WriteStatus[r0];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     ra    |     0     |    280    |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

Writes the program status word with the contents of register *ra*. The effect of changing the PSW doesn't take effect until 3 cycles after the instruction is issued, with the exception of the kernel bit, which doesn't change value until 4 cycles after the instruction is issued. In particular, reading the PSW immediately after changing it returns the old value.

Restrictions

Due to pipeline constraints, the write PSW instruction must not have an instruction cache miss during its WR stage or the PSW will not be written. Therefore is is recommended that the instruction always be aligned at word 0 of a memory line and be followed by either 3 null instructions, or the kernel exit sequence.

Note that the trap bits and literal field of the PSW cannot be written.

```
        @1000x;
        r1 := (KernelStatus);
<0>     r1 : special WriteStatus[r0];
        null;
        null;
        null;

Next:   ...

KernelStatus: !68006000x;
```

Figure 4-11: Initialization of the PSW

It is possible to change from kernel mode to user mode by writing a value with the pre-kernel bit deasserted to the PSW. Instruction execution continues in sequence in this case, at *Next* in Figure 4-11. The operating system may enable interrupts in this fashion. As with the normal kernel sequence, if instruction mapping is enabled, the first user mode instruction (at *Next*) must not generate an instruction page fault. Refer to Section 4.3 for discussion of the normal kernel exit sequence.

When the processor is reset, the pid and register bank fields are not initialized. Operating system startup code should write the PSW before executing any other instructions as shown in Figure 4-11.

Execution

DO	AR := ra
EX	RR := AR
WR	PSW := RR

4.2.4 Read Instruction Translation Fault Register

TASM Format

```
ra := special ReadInstTFR[r0];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |    ra     |     0     |    C3     |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

When a virtual address is referenced that does not have a valid translation, a trap is caused and the page number of the offending virtual address is saved until kernel mode is exited. ReadInstTFR writes the contents of the ITFR into register *ra*. The format of the ITFR is shown in Figure 4-12.

```
|<           22           >|<       10       >|
+--------------------------+------------------+
|                          |                  |
|       page number        |    undefined     |
|                          |                  |
+--------------------------+------------------+
```

Figure 4-12: Translation Fault Register Format

Restrictions

Reading the ITFR modifies the ICAR, so the instruction must be followed by a goto. The instruction sequence shown in Figure 4-13 is recommended.

```
r1 := special ReadInstTFR[r0];
goto 1[r0];
null;
```

Figure 4-13: Read Instruction TFR Sequence

Execution

DO	ICAR := undefined
EX	RR := ITFR
WR	ra := RR

4.2.5 Read Data Translation Fault Register

TASM Format

```
ra := special ReadDataTFR[r0];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     ra    |     0     |    103    |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

When a virtual address is referenced that does not have a valid translation, or a store instruction to a page that is not writable is executed, a trap is caused and the page number of the offending virtual address is saved until kernel mode is exited. Read data TFR writes the contents of the DTFR into register *ra*. The format of the DTFR is shown in Figure 4-12.

Restrictions

The instruction following this one must not be a conditional branch that tests register *ra*.

The instruction executed two cycles later must not read r0.

The instruction sequence shown in Figure 4-14 is recommended.

```
r1 := special ReadDataTFR[r0];
null;
null;
```

Figure 4-14: Read Data TFR Sequence

Execution

DO	-
EX	RR := DTFR
WR	ra := RR

4.2.6 Read Instruction Translation Buffer Tag Entry

TASM Format

```
ra := special ReadInstTlbTag0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     ra    |     rb    |   C1/D1   |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions read the instruction cache TLB tag for columns 0 or 1 into register *ra*. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

```
|<    8     >|          13          >|1 |1|< 3  >|<2>|< 4 >|
+------------+----------------------+--+-+------+---+-----+
|    pid     |          tag         |~v|w|parity| m |  0  |
+------------+----------------------+--+-+------+---+-----+
```

Figure 4-15: TLB Tag Entry Read Data

The format of the data read from the TLB tag is shown in Figure 4-15; the *pid* field corresponds to the PSW pid field, the *tag* field is the high order 13 bits of the virtual word address, the *v* field is 0 if the translation is valid, the *w* field is 1 if the page is writable[2], the *parity* field is the odd parity of the high order 23 bits of the data, and the *m* field is encoded as shown in Figure 4-16.

m	Columns Matched
0	None
1	Col[0]
2	Col[1]
3	Col[0] and Col[1]

Figure 4-16: TLB Tag Match Bits Encoding

Restrictions

The instruction preceding this one must not modify register *rb*.

The ICAR is modified, so the instruction must be followed by a branch.

Figure 4-17 shows the recommended instruction sequence.

Execution

DO	ICAR := (rb >> 2)
EX	RR := TlbTag[ICAR]
WR	ra := RR

[2]This applies only to the data cache.

```
{r1 has address, r2 gets tag}
null;
r2 := special ReadInstTlbTag0[r1];
goto 1[r0];
null;
```

Figure 4-17: Read Instruction TLB Tag Entry Instruction Sequence

4.2.7 Read Data Translation Buffer Tag Entry

TASM Format

```
ra := special ReadDataTlbTag0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |    ra     |    rb     |  101/111  |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions read the data cache TLB tag for columns 0 or 1 into register *ra*. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

The format of the data read from the TLB tag is as shown in Figure 4-15 and discussed in Section 4.2.6.

Restrictions

The instruction preceding this one must not be a store or modify register *rb*.

The instructions following this one must not be a conditional branch that tests register *ra*.

The instruction executed two cycles later must not read r0.

Figure 4-18 shows the recommended instruction sequence.

```
{r1 has address, r2 gets tag}
null;
r2 := special ReadDataTlbTag1[r1];
null;
null;
```

Figure 4-18: Read Data TLB Tag Entry Instruction Sequence

Execution

DO	DCAR := (rb >> 2)
EX	RR := TlbTag[DCAR]
WR	ra := RR

4.2.8 Read Instruction Translation Buffer Data Entry

TASM Format

```
ra := special ReadInstTlbData0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     ra    |     rb    |   C0/C4   |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions read the instruction cache TLB tag for columns 0 or 1 into register *ra*. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

```
|<         22         >|<  3   >|<    7    >|
+---------------------+--------------------+
|  real page number   | parity |     0     |
+---------------------+--------------------+
```

Figure 4-19: TLB Data Entry Read Data

The format of the data read from the TLB tag is shown in Figure 4-19; the *real page number* field is the real page number that will be presented to the memory controller, and the *parity* field is the odd parity of the *real page number*.

Restrictions

The instruction preceding this one must not modify register *rb*.

The ICAR is modified, so the instruction must be followed by a branch.

Figure 4-20 shows the recommended instruction sequence.

```
{r1 has address, r2 gets data entry}
null;
r2 := special ReadInstTlbData1[r1];
goto 1[r0];
null;
```

Figure 4-20: Read Instruction TLB Data Entry Instruction Sequence

Execution

DO	ICAR := (rb >> 2)
EX	RR := TlbData[ICAR]
WR	ra := RR

4.2.9 Read Data Translation Buffer Data Entry

TASM Format

```
ra := special ReadDataTlbData0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     ra    |     rb    |  100/104  |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions read the data cache TLB tag for columns 0 or 1 into register *ra*. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

The format of the data read from the TLB tag is shown in Figure 4-19 and discussed in Section 4.2.8.

Restrictions

The instruction preceding this one must not be a store or modify register *rb*.

The instruction following this one must not be a conditional branch that tests register *ra*.

The instruction executed two cycles later must not read r0.

Figure 4-21 shows the recommended instruction sequence.

```
{r1 has address, r2 gets data entry}
null;
r2 := special ReadDataTlbData0[r1];
null;
null;
```

Figure 4-21: Read Data TLB Data Entry Instruction Sequence

Execution

DO	DCAR := (rb >> 2)
EX	RR := TlbData[DCAR]
WR	ra := RR

4.2.10 Write Instruction Translation Buffer Tag Entry

TASM Format

```
r0 := special WriteInstTlbTag0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     0     |     rb    |  207/20D  |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions write the instruction cache TLB tag for columns 0 or 1 with the contents of RR. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

```
|<    8     >|          13         >|1 |1|<     9     >|
+------------+---------------------+--+-+-------------+
|    pid     |         tag         |~v|w|      0      |
+------------+---------------------+--+-+-------------+
```

Figure 4-22: TLB Tag Entry Write Data

Figure 4-22 shows the format of the data written into the tag entry; the *pid* field is the process id of the virtual address space being mapped and it must be the same as the current PSW pid field, or the TLB hashing will produce incorrect translations, the *tag* field is the high 13 bits of the virtual word address (i.e., high 11 bits of *rb*), the *~v* bit is 0 to create a valid translation or 1 to invalidate the entry, and the *w* bit is 1 to make the page writable or 0 to make it read-only[3]. The least significant 9 bits must be zero, or processor parity checking will not work correctly.

Restrictions

The TLB uses the value of RR to write the tag entry, so an ALU or load instruction must immediately precede this instruction to load RR properly.

For the instruction cache only, the tag entry must be complemented before writing.

The ICAR is modified, so the instruction must be followed by a branch.

Figure 4-23 shows an instruction sequence for writing the TLB tag for column 0.

```
{r1 has tag entry, r2 has virtual address}
r0 := not r1;
r0 := special WriteInstTlbTag0[r2];
goto 1[r0];
null;
```

Figure 4-23: Instruction TLB Tag Entry Write Sequence

[3]The *w* bit applies only to the data cache.

Execution

DO	ICAR := (rb >> 2)
EX	TlbTag[ICAR] := RR
WR	-

4.2.11 Write Data Translation Buffer Tag Entry

TASM Format

```
r0 := special WriteDataTlbTag0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     0     |     rb    |  247/24D  |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions write the data cache TLB tag for columns 0 or 1 with the contents of RR. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

Figure 4-22 shows the format of the data written into the tag entry, and Section 4.2.10 discusses the fields.

Restrictions

The TLB uses the value of RR to write the tag entry, so an ALU or load instruction must immediately precede this instruction to load RR properly.

The write TLB tag instruction must not have an instruction cache miss during its EX stage or the tag entry will not be written correctly.

The instruction executed two cycles later must not read r0.

Figure 4-24 shows an instruction sequence for writing the TLB tag for column 1.

```
    {r1 has tag entry, r2 has virtual address}
<!1>r0 := r1;
    r0 := special WriteDataTlbTag1[r2];
    null;
    null;
```

Figure 4-24: Data TLB Tag Entry Write Sequence

Execution

DO	DCAR := (rb >> 2)
EX	TlbTag[DCAR] := RR
WR	-

4.2.12 Write Instruction Translation Buffer Data Entry

TASM Format

```
r0 := special WriteInstTlbData0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     0     |     rb    |  20B/20E  |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions write the instruction cache TLB data for columns 0 or 1 with the contents of RR. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

```
|<        22        >|<        10        >|
+--------------------+--------------------+
|  real page number  |         0          |
+--------------------+--------------------+
```

Figure 4-25: TLB Data Entry Write Data

Figure 4-25 shows the format of the data written into the tag entry; the *real page number* field is the real page to be used for this virtual page. The least significant 10 bits must be zero, or processor parity checking will not work correctly.

Restrictions

The TLB uses the value of RR to write the tag entry, so an ALU or load instruction must immediately precede this instruction to load RR properly.

For the instruction cache only, the data entry must be complemented before writing.

The ICAR is modified, so the instruction must be followed by a branch.

Figure 4-26 shows an instruction sequence for writing the TLB data entry for column 0.

```
{r1 has tag entry, r2 has virtual address}
r0 := not r1;
r0 := special WriteInstTlbData0[r2];
goto 1[r0];
null;
```

Figure 4-26: Instruction TLB Data Entry Write Sequence

Execution

DO	ICAR := (rb >> 2)
EX	TlbData[ICAR] := RR
WR	-

4.2.13 Write Data Translation Buffer Data Entry

TASM Format

```
r0 := special WriteDataTlbData0/1[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     0     |     rb    |  24B/24E  |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

These instructions write the data cache TLB data for columns 0 or 1 with the contents of RR. Register *rb* contains the address that selects one of the 512 rows of the TLB. The row is hashed and extracted from the address in the normal fashion, i.e., addresses [0..FFF] reference row 0, addresses [1000..1FFF] reference row 1, etc. Note that the TLB hashing discussed in Section 2.2.2 is in effect at all times, so the PSW pid field should be set to the process of interest before manipulating the TLB.

Figure 4-25 shows the format of the data written into the tag entry; Section 4.2.12 discusses the field values.

Restrictions

The TLB uses the value of RR to write the tag entry, so an ALU or load instruction must immediately precede this instruction to load RR properly.

The write TLB tag instruction must not have an instruction cache miss during its EX stage or the data entry will not be written correctly.

The instruction executed two cycles later must not read r0.

Figure 4-27 shows an instruction sequence for writing the TLB data for column 1.

```
        {r1 has tag entry, r2 has virtual address}
    <!1>r0 := not r1;
        r0 := special WriteInstTlbData1[r2];
        null;
        null;
```

Figure 4-27: Data TLB Data Entry Write Sequence

Execution

DO	DCAR := (rb >> 2)
EX	TlbData[DCAR] := RR
WR	-

4.2.14 Read I/O Control Register

TASM Format

```
ra := special GetCtl[ioReg];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     ra    |   ioReg   |     80    |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

This instruction reads the contents of the I/O control register specified by the **previous** I/O instruction into register *ra*. The *ioReg* field specifies the I/O control register to be used for the **next** I/O instruction. Figure 4-8 lists the encodings for the *ioReg* field.

Note that all special instructions update the I/O control register address in the memory controller; a dummy GetCtl instruction is typically used before the actual GetCtl instruction as shown in Figure 4-28.

```
r0 := special GetCtl[Event]; {select event register}
null;                        {can't have adjacent I/O insts.}
r1 := special GetCtl[Event]; {read event register into r1}
null;                        {no cond. branch on r1 here}
null;                        {no pc-relative addressing here}
```

Figure 4-28: I/O Control Register Read Sequence

Restrictions

I/O instructions may not be adjacent to other I/O instructions.

Interrupts should be disabled during I/O instruction sequences.

If a GetCtl is used to read a register written by a SetCtl instruction, the GetCtl and SetCtl instructions must be separated by at least two instructions.

The instruction following a GetCtl may not be a conditional branch that tests register *ra*.

The GetCtl must select the same I/O control register as it is reading, or the preceding instruction must not be a load, store, or special instruction and the GetCtl must not reside in word 3 of a memory line.

Execution

DO	BR := (I/O Control Address), I/O Control Address := rb
EX	RR := BR
WR	ra := RR

4.2.15 Write I/O Control Register

TASM Format

```
ra := special SetCtl[ioReg];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     0     |   ioReg   |    2C0    |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

This instruction writes the contents of the I/O control register specified by the **previous** I/O instruction with the contents of RR. The *ioReg* field specifies the I/O control register to be used for the **next** I/O instruction. Figure 4-8 lists the encodings for the *ioReg* field.

Note that all special instructions update the I/O control register address in the memory controller; a dummy GetCtl instruction is typically used before the SetCtl instruction as shown in Figure 4-29.

```
r0 := special GetCtl[Enable];{select enable register}
r0 := r1;                    {load RR}
r0 := special SetCtl[Enable];{write r1 into enable register}
null;
null;                        {no pc-relative addressing here}
```

Figure 4-29: I/O Control Register Write Sequence

Restrictions

Section 4.2.14 lists additional restrictions on the use of I/O instructions.

Execution

DO	I/O Control Address := rb
EX	(I/O Control Address) := RR
WR	-

4.2.16 Read I/O Adaptor Register

TASM Format

```
ra := special IoRead[ioReg];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     0     |   ioReg   |    300    |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

This instruction reads the contents of the I/O adaptor register specified by the value of RR into register *ra*. The *ioReg* field specifies the I/O control register to be used for the **next** I/O instruction. Figure 4-8 lists the encodings for the *ioReg* field. The read is applied to the I/O adaptor in the slot specified by the IoStatus register.

```
        {r1 has slot, r2 has I/O adaptor register address}
        r0 := special GetCtl[Status];    {select status register}
        r0 := r1;                        {select I/O slot}
        r0 := special SetCtl[Status];    {write r1 into status}
        r0 := r2;                        {select adaptor register}
        r0 := special IoRead[Status];    {read adaptor register}
        null;
    <!3>r3 := special GetCtl[IoReadData];{get status}
        null;
        r4 := special GetCtl[IoReadData];{get read data}
        null;
        null;                            {no pc-relative here}
```

Figure 4-30: Read I/O Adaptor Register Sequence

Figure 4-30 shows a typical instruction sequence for reading an I/O adaptor register. The sequence selects the I/O slot, issues an IoRead and gets the operation status and read data. Note the alignment restriction on the GetCtl to read the status register as discussed in Section 4.2.14. Also note that this GetCtl will stall the processor until the memory controller completes the IoRead operation.

Restrictions

Section 4.2.14 lists additional restrictions on the use of I/O instructions.

Execution

DO	I/O Control Address := rb
EX	(I/O Address Register) := RR
WR	-

4.2.17 Write I/O Adaptor Register

TASM Format

```
ra := special IoWrite[ioReg];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    16   >|
+-------+-----------+-----------+-----------+
|       |           |           |           |
|   1   |     0     |   ioReg   |    340    |
|       |           |           |           |
+-------+-----------+-----------+-----------+
```

Description

This instruction write the contents of the I/O adaptor register specified by the value of RR. The *ioReg* field specifies the I/O control register to be used for the *next* I/O instruction. Figure 4-8 lists the encodings for the *ioReg* field. The write is applied to the I/O adaptor in the slot specified by the IoStatus register.

```
{r1 has slot, r2 has address, r3 has data}
r0 := special GetCtl[Status];    {select status register}
r0 := r1;                        {select I/O slot}
r0 := special SetCtl[IoWriteData];{write r1 into status}
r0 := r3;                        {set write data}
r0 := special SetCtl[IoWriteData];{write r3 into wdata}
r0 := r2;                        {select I/O register}
r0 := special IoWrite[Status];   {write adaptor register}
null;
r3 := special GetCtl[Status];    {get status}
null;
null;                            {no pc-relative here}
```

Figure 4-31: Write I/O Adaptor Register Sequence

Figure 4-31 shows a typical instruction sequence for writing an I/O adaptor register. The sequence selects the I/O slot, sets the write data, issues an IoWrite, and gets the operation status. Note that the GetCtl to get the operation status will stall the processor until the memory controller completes the IoWrite operation.

Restrictions

Section 4.2.14 lists additional restrictions on the use of I/O instructions.

Execution

DO	I/O Control Address := rb
EX	(I/O Address Register) := RR
WR	-

4.3 Kernel Exit

TASM Format

```
r0 := kernelExit disp[r0];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<             16              >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   2   |     0     |     0     |         displacement          |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

The kernel exit instruction loads the ICAR with the contents of PC2, plus the sign-extended *displacement*, advances PC1 into PC2, and advances the address of the kernel exit into PC1. Like all branch-class instructions, the instruction following the kernel exit is executed before the branch takes effect.

```
{r1 contains user PSW, pc-queue previously loaded}
100:<0>r1 := special writeStatus[r0];   DO EX WR
104:   r1 := (UserR1[r0]);                 DO EX WR
108:   r0 := kernelExit 0[r0];                DO EX WR
10C:   r0 := kernelExit 0[r0];                   DO EX WR
PC2:   -                                            DO EX WR
```

Figure 4-32: Return From Kernel Instruction Sequence[4]

The typical return-from-kernel instruction sequence is shown in Figure 4-32. The PSW is written with the user status, generally turning mapping on and kernel mode off. Due to pipelining, the PSW does not reflect the new status until the D0 stage of instruction 10C, and the kernel bit of the PSW does not clear until the D0 stage of the first user instruction. Refer to Section 4.2.3 for further discussion of the write PSW instruction. The two kernel exit instructions reload the processor pipeline with the address of the user instructions aborted when the user process trapped or was interrupted. The user instructions will be fetched with mapping enabled, and executed in user mode.

Note that it is possible to exit kernel mode without manipulating the pc-queue; refer to Section 4.2.3.

Restrictions

The user instruction referenced by PC2 at the start of the return from kernel sequence must have a valid instruction translation. Due to pipeline constraints, if this instruction doesn't have a translation, the resulting translation trap would happen before reloading of the PSW trap bits can occur, and before the pc-queue returns to a stable state. Therefore, the operating system must insure that there is a valid translation for the address in PC2. The address in PC1 can cause a translation fault, and need not be checked.

The kernel exit instruction will cause an illegal instruction trap if executed in user mode.

---

[4]The *<0>* notation indicates that the instruction must be aligned at word 0 of a memory line. This is a constraint of the write PSW instruction; refer to Section 4.2.3.

Execution

DO	ICAR := (PC2 >> 2) + disp, PC2 := PC1
EX	-
WR	-

4.4 Set PC-Queue

TASM Format

```
r0 := pseudoCall disp[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<              16              >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   3   |     0     |    rb     |         displacement          |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

The effective address is loaded into the pc-queue. Since the address must pass through the ICAR to reach the pc-queue, the pseudo call instruction must be followed by a branch instruction as shown in Figure 4-33.

```
    { r10 is the address of the user process' pc-queue }
<!2>r0 := pseudoCall 0[r10];
    goto  1[r0];
    null;
<!2>r0 := pseudoCall 1[r10];
    goto 1[r0];
    null;
```

Figure 4-33: PC-Queue Load Instruction Sequence

Restrictions

Due to pipeline constraints, the pseudo call instruction must not have an instruction cache miss during its EX stage or the pc-queue will not be written properly. For the code sequence shown in Figure 4-33 this means that the pseudo call instructions must not reside in word 2 of a memory line.

If the pseudo call instruction is executed in user mode, an illegal instruction trap occurs.

Execution

DO	ICAR := (rb >> 2) + disp
EX	PC := ICAR
WR	PC2 := PC1, PC1 := PC

4.5 Flush Cache

TASM Format

```
flush disp[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<              16               >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   4   |     0     |     rb    |         displacement          |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

The flush instruction invalidates lines in the instruction and data caches starting at the effective address through the end of the page containing the effective address. If matching data cache lines are dirty, they are written out to memory before being invalidated.

```
{ r1 contains the address to start flush }
r1 := r1,r1.[3,29];      {clear least significant 3 bits}
r1 := r1,r1.[29,32];
null;                    {no hardware address interlock}
flush 1[r1];
goto 1[r0];              {ICAR modified, reload}
null;
```

Figure 4-34: Flush Instruction Sequence

Restrictions

Due to an implementation constraint, the effective address should be a multiple of 8 plus 1, or the processor may never exit the flush instruction.

The flush instruction modifies the ICAR, and must be followed by a branch instruction.

The preceding instruction must not be a store or modify register *rb* as there is no hardware resource interlock.

If executed in user mode, flush causes an illegal instruction trap.

Execution

DO	ICAR := (rb >> 2) + disp
EX	flush (ICAR), ICAR := ICAR + 4
WR	-

4.6 Load

TASM Format

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

Memory Format

```
|<  4  >|<    6    >|<    6    >|<            16              >|
+-------+-----------+-----------+-----------------------------+
|       |           |           |                             |
|   5   |     ra    |     rb    |        displacement         |
|       |           |           |                             |
+-------+-----------+-----------+-----------------------------+
```

Description

The word at the effective address is written into register *ra*.

If data mapping is enabled and no data translation buffer entry exists, a data translation trap occurs. PC2 will contain the address of the load instruction, PC1 will contain the address of the instruction following the load instruction in execution sequence. Destructive loads of the form:

```
r1 := (0[r1]);
```

will not modify register *r1* and can be reexecuted after resolving the page fault.

If the load is immediately after a store, the processor will stall one cycle. If register *rb* is modified by the immediately preceding instruction, the processor will stall one cycle. When possible, compilers should reorder instructions to minimize such stalls.

Execution

DO	DCAR := (rb >> 2) + disp
EX	RR := (DCAR)
WR	ra := RR

4.7 Store

TASM Format

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

Memory Format

```
|<  4  >|<    6    >|<    6    >|<              16               >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   6   |     ra    |     rb    |          displacement         |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

The contents of register *ra* is written to the effective address.

If data mapping is enabled and no data translation buffer entry exists, a data translation trap occurs. PC2 will contain the address of the store instruction, PC1 will contain the address of the instruction following the store instruction in execution sequence.

If data mapping is enabled, a valid data translation buffer entry exists, but writes are not enabled to the page, a write protection trap occurs. PC2 will contain the address of the store instruction, PC1 will contain the address of the instruction following the store instruction in execution sequence.

If either of the above traps occur, the store into the cache is suppressed.

If register *rb* is modified by the immediately preceding instruction, the processor will stall one cycle. If the store is immediately followed by a load or store, the processor will stall one cycle. When possible, compilers should reorder instructions to minimize such stalls.

Execution

DO	DCAR := (rb >> 2) + disp
EX	(DCAR) {Probe}
WR	(DCAR) := ra

4.8 Subroutine Jump

TASM Format

```
ra := goto disp[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<              16              >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   7   |    ra     |    rb     |         displacement          |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

The ICAR is loaded with the effective address. PC is written into register *ra*. Due to the delayed branch, the next instruction in execution sequence, e.g., the instruction after the branch, is executed before execution commences at the effective address.

```
1FFC: r62 := goto 0[r1]; {r1 = 4000}
2000: null;
```

Figure 4-35: Branch Destination Translation Trap

If the instruction in execution succession of the branch instruction causes an instruction translation trap, PC2 will contain the address of the branch instruction, PC1 will contain the address of the instruction causing the trap. In Figure 4-35, if address 2000 causes a translation trap, PC2 will contain 1FFC and PC1 will contain 2000.

If the branch destination causes an instruction translation trap, PC2 will contain the address of the instruction in execution succession of the branch instruction, PC1 will contain the branch destination. In Figure 4-35, if address 4000 causes a translation trap, PC2 will contain 2000, and PC1 will contain 4000.

In both cases, the instruction stream can be restarted with the existing pc-queue after resolving the translation fault.

If register *rb* is modified by the immediately preceding instruction, the processor will stall one cycle. When possible, compilers should reorder instructions to minimize such stalls.

Execution

DO	ICAR := (rb >> 2) + disp, BR := PC
EX	RR := BR
WR	ra := RR

4.9 Conditional Jump

TASM Format

```
if ra <cond> goto disp[pc];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<             16              >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   8   |     ra    |   cond    |         displacement          |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

The branch destination is computed by sign-extending *displacement* to 32 bits, and adding the value of PC (the address of the conditional jump instruction). This address is loaded into the ICAR if register *ra* meets the condition specified by *cond*, otherwise the ICAR increments normally. Due to the delayed branch, the next instruction in line is always executed.

The *cond* field encoding is shown in Figure 4-36. Register *ra* is tested for the specified relation to zero.

Relation	Encoding
=	00
<>	08
<	10
>=	18
>	20
<=	28
odd	30
even	38

Figure 4-36: Conditional Jump Condition Encoding

The state of the pc-queue after instruction translation traps is analogous to subroutine jump translation traps described in Section 4.8.

Restrictions

The immediately preceding instruction must not be a special instruction that modifies register *ra*.

The instruction two cycles earlier must not be a special instruction or PC will be invalid.

Execution

DO	if ra <cond> ICAR := (PC >> 2) + disp
EX	-
WR	-

4.10 Extract Field

TASM Format

```
rc := ra,rb.[shift:mask];
```

Memory Format

```
|<  4  >|<    6    >|<   6    >|<    6    >|<   5   >|<   5   >|
+-------+-----------+----------+-----------+---------+---------+
|       |           |          |           |         |         |
|   9   |    ra     |   rb     |    rc     | mask-1  | ~shift  |
|       |           |          |           |         |         |
+-------+-----------+----------+-----------+---------+---------+
```

Description

Registers *ra* and *rb* are concatenated to form a 64-bit word, with *ra* as the most significant 32 bits. Field extraction is accomplished by right shifting the double word quantity (*ra*,*rb*) *shift* bits and retaining the rightmost *mask* bits of the result in register *rc*; *shift* is in the range [0..31], *mask* is in the range [1..32].

Execution

DO	AR := ra, BR := rb
EX	RR := ((ra<<32 | rb) >> shift) & ((1<<mask)-1)
WR	rc := RR

4.11 Alu

TASM Format

```
rc := ra <op> rb;
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<    6    >|<    7      >|1|1|1|
+-------+-----------+-----------+-----------+-------------+-+-+-+
|       |           |           |           |             |s|s|l|
|   A   |     ra    |     rb    |     rc    |  function   |h|b|i|
|       |           |           |           |             | | |t|
+-------+-----------+-----------+-----------+-------------+-+-+-+
```

Description

The ALU performs boolean or arithmetic operations on the AR and BR operands, storing the result in register *rc*. The BR operand is register *rb*. The AR operand is register *ra* if the *literal* bit is zero, otherwise it is the *ra* field of the instruction zero-extended to 32 bits.

```
|<     4     >|<1>|<1>|<1>|
+-------------+---+---+---+
|             | ~ | c | u |
|   select    | c | m | n |
|             | i | p | s |
+-------------+---+---+---+
```

Figure 4-37: ALU Function Fields

When both the *sh* and the *sb* bits are zero, the 7-bit *function* field has the encoding shown in Figure 4-37. The *select* field determines the ALU function as shown in Figure 4-38, the *~ci* bit is the inverted carry-in for arithmetic ALU functions, the *cmp* bit enables the ALU sign corrector for performing comparisons, and the *uns* bit enables unsigned comparisons.

Select	Operation
0	<unsupported>
1	<unsupported>
2	<unsupported>
3	<unsupported>
4	ra + rb + ~ci
5	rb - ra + ~ci
6	ra - rb + ~ci
7	-ra + ~ci
8	ra eqv rb
9	ra xor rb
A	ra or rb
B	rb
C	not ra
D	ra
E	ra and rb
F	0

Figure 4-38: ALU Select Codes

While many combinations are possible for the 7-bit *function* field, most are redundant or uninteresting. Figure 4-39 lists the encodings for commonly encountered operations. Subtraction and unary negation are two's complement. The comparison instructions set the sign bit of *rc* to 1 if the relation is true, 0 otherwise. The value of the other bits of *rc* is undefined. The compares denoted with a trailing *u* are unsigned compares, the others are signed compares. Note that there is no equality or inequality comparison, but xor can be used instead. Appendix I discusses operation of the ALU sign corrector in more detail.

Function	Operation
24	ra + rb
30	ra - rb
38	-ra
2A	ra > rb
2B	ra >u rb
2E	ra >= rb
2F	ra >=u rb
32	ra < rb
33	ra <u rb
34	ra <= rb
35	ra <=u rb
40	ra eqv rb
48	ra xor rb
50	ra or rb
58	rb
60	not ra
68	ra
70	ra and rb

Figure 4-39: ALU Function Encodings

If *sh* is set then the ALU instruction is transformed into a field extract instruction, in which a fixed-size field is extracted from a variable location within a word. The TASM format is shown in Figure 4-40.

```
rc := ra,rb.(size:b);   {bit-field extract}
rc := ra,rb.(size:o);   {byte extract}
```

Figure 4-40: TASM Variable/Byte Extract Format

The encoding of the field size in the 7-bit *function* field of the instruction is shown in Figure 4-41. The extract size is in the range [1..32]. The shift count for the 64-bit (ra,rb) pair is specified by the least significant 5 bits of *ra*, which can be a literal. As with the extract instruction, the complement of the desired shift count should be specified. The shift count is in the range [0..31].

```
|<     5     >|<1>|<1>|
+-------------+---+---+
|             |   | i |
|   size-1    | 0 | n |
|             |   | v |
+-------------+---+---+
```

Figure 4-41: Extract Size Encoding

If the *sb* bit is set, the field extract is forced to a byte boundary. The byte position within the word is specified by the least significant 2 bits of *ra*, which can be a literal. The *inv* bit controls the order of bytes within a word as shown in Figure 4-42.

```
|< 8 >|< 8 >|< 8 >|< 8 >|     ra  inv | byte     ra  inv | byte
+-----+-----+-----+-----+     --  --- | ----     --  --- | ----
|     |     |     |     |     0    0  |  A       0    1  |  D
|  A  |  B  |  C  |  D  |     1    0  |  B       1    1  |  C
|     |     |     |     |     2    0  |  C       2    1  |  B
+-----+-----+-----+-----+     3    0  |  D       3    1  |  A
```

Figure 4-42: Byte Extract Position

Execution

DO	AR := ra, BR := rb
EX	RR := function(ra, rb)
WR	ra := RR

4.12 Coprocessor Instructions

TASM Format

```
rc := ra,rb(<result>,<func>,<op>)
```

Memory Format

```
|<  4  >|<   6   >|<   6   >|<   6   >|<  3  >|<  3  >|<   3  >|1|
+-------+---------+---------+---------+-------+-------+--------+-+
|       |         |         |         |       |       |        | |
|   B   |   ra    |   rb    |   rc    |result |  op   |  func  |0|
|       |         |         |         |       |       |        | |
+-------+---------+---------+---------+-------+-------+--------+-+
```

Description

The coprocessor instruction writes the value of the coprocessor *accumulator* (AC) register into register *rc* in the format specified by *result*. The instruction uses the 64-bit (*ra*,*rb*) pair as the coprocessor *operand* and starts the operation specified by *func*, interpreting the *operand* in format *op*.

The coprocessor supports integer, single-precision floating point, double-precision floating point, and miscellaneous status and system clock functions. The integer functions provide 32-bit multiply and divide and 64-bit add and subtract support for the processor. The floating point functions provide add, subtract, multiply, and divide support for the processor. The miscellaneous functions provide time-of-day clock and interval timer support for the processor.

```
|1|<  8  >|<        23         >|
+-+-------+---------------------+
|s|  exp  |      fraction       |
+-+-------+---------------------+
```

Figure 4-43: 32-Bit, Single-Precision Floating Point F Format

Figure 4-43 shows the memory format for a single-precision floating point number. The *s* bit is the sign of the *fraction* field. The *exp* field is the 128-biased exponent. Figure 4-44 shows the memory format for a double-precision floating point number. The *s* bit is the sign of the *fraction* field. The *exp* field is the 1024-biased exponent.

```
|1|<   11   >|<                    52                    >|
+------------+--------------------------------------------+
|s|   exp    |                  fraction                  |
+-+----------+--------------------------------------------+
```

Figure 4-44: 64-Bit, Double-Precision Floating Point G Format

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

```
|1|<    16    >|<2 >|<                   62                    >|
+--------------+----+-------------------------------------------+
|s|    exp     | 01 |                 fraction                  |
+-+------------+----+-------------------------------------------+
```

Figure 4-45: Internal Coprocessor Floating Point Format

Figure 4-45 shows the internal coprocessor format for floating point numbers. The fraction register has its binary point between the most significant and the second most significant bits. The bit that was hidden in the external representation is present in the second most significant position of the fraction register. The exponent register holds an unbiased two's complement exponent.

Loading a non-zero single-precision floating point number from register *ra* to the AC sets the most significant bit of the fraction register to 0, the next bit to 1, and copies the succeeding bits from the fraction field of register *ra*. The exponent register will hold the value of the exponent field of *ra* minus 128, sign extended to 16 bits.

Loading a non-zero double-precision floating point number from registers (*ra*,*rb*) to the AC sets the most significant bit of the fraction register to 0, the next bit to 1, and copies the succeeding bits from the fraction field of registers (*ra*,*rb*). The exponent register will hold the value of the exponent field of (*ra*,*rb*) minus 1024, sign extended to 16 bits.

Loading a zero single- or double-precision floating point number sets the sign bit, exponent register, and all bits of the fraction register to zero.

Loading a double fixed number from registers (*ra*,*rb*) causes all 64 bits to be placed into the fraction register without change. The exponent register is unaffected. The sign bit is set from the most significant bit of register *ra*.

Value	Result	Format
0	sign	ac sign and various status bits
1	exp	exponent, sign extended to 32 bits
2	inth	most significant word of the fraction
3	intl	least significant word of the fraction
4	dph	most significant word of a double value
5	dpl	least significant word of a double value
6	sp	single precision value
7	null	zero

Figure 4-46: Result Field Encoding

The coprocessor instruction waits for completion of the previous coprocessor operation, if it is not already complete. A portion of the result in the coprocessor AC is stored in register *rc*, as selected by the result format field as shown in Figure 4-46. In the same cycle that a result is written to register *rc*, registers *ra* and *rb* are passed to the coprocessor for use in an operation selected by the *func* field as shown in Figure 4-47. The *op* field specifies the format of the (*ra*,*rb*) *operand* as shown in Figure 4-48. Register *rc* is written after reading *ra* and *rb*, so even if *ra* or *rb* selects the same register as *rc*, the old value of the register will be passed to the coprocessor before the coprocessor's previous result is written.

Value	Function	Action
0	load	ac <- operand
1	add	ac <- ac + operand
2	subtract	ac <- ac - operand
3	rsubtract	ac <- operand - ac
4	misc	miscellaneous
5	multiply	ac <- ac * operand
6	divide	ac <- ac / operand
7	rdivide	ac <- operand / ac

Figure 4-47: Function Field Encoding

Value	Operand	Format
0	i	integer
1	-	reserved
2	-	reserved
3	-	reserved
4	f	f input, f result rounding
5	fg	f input, g result rounding
6	gf	g input, f result rounding
7	g	g input, g result rounding

Figure 4-48: Operand Field Encoding

In Figure 4-48 the *result rounding* refers to rounding after calculations; *f* rounding is into a 23-bit fraction, *g* rounding is into a 52-bit fraction. The input format applies to the *operand*. Figure 4-49 show conversion of a g-format number to f-format using this feature. Note that for the *load* function, the *fg* operand type is mapped to a miscellaneous function as discussed below.

```
{g-number in r1,r2}
r0 := r1,r2(null,load,gf);
r3 := r0,r0(sp,nop);
```

Figure 4-49: Conversion of G-Format to F-Format

For *i* operands, the *operand* and the AC fractions are 64-bit, two's complement integers for the add, subtract, and rsubtract functions. For the multiply function, the contents of register *ra* and *inth* of the AC fraction are two's complement integers generating a 64-bit product into the AC fraction. For the divide function, *ra* is an unsigned divisor, and the AC fraction is an unsigned dividend; register *rb* must be zero to get a quotient in AC fraction *intl* bits, and remainder in AC fraction *inth* bits. For the rdivide function, the *operand* is an unsigned dividend, and *inth* of the AC fraction is an unsigned divisor; *intl* of the AC fraction must contain zero to get a quotient in *intl* of the AC fraction and remainder in *inth* of the AC fraction.

Integer multiplication is signed and generates a 64-bit product. If one is interested only in the low-order word of product, signed and unsigned multiplication are equivalent, and overflow can be detected by examination of the high-order word; in unsigned arithmetic the high word must be zero, in signed arithmetic the high word must be copies of the sign of the low word. If the program requires the full signed product, it should use the result returned. If the unsigned product is required, it can be obtained by adding the multiplier to the high word of product if the multiplicand msb is set, and adding the multiplicand to the high word of product if the multiplier msb is set.

Integer division is unsigned; the hardware computes 32-bit quotient and remainder from a 64-bit dividend (numerator) and 31-bit divisor (denominator), subject to the following restrictions:

- Software must check for division by zero.
- Software must check for overflow; the high word of dividend is greater than or equal to the divisor.
- Software must ensure that the low-order word of the divisor is zero.
- The remainder may need correction. If the quotient is even (least significant bit is zero) and the remainder returned is not zero, then the divisor must be added to the remainder to obtain the correct result.

Integer division leaves the quotient in the low-order word of the AC, and the remainder in the high-order word. Signed division may be obtained by taking the absolute values of both divisor and dividend, then negating the quotient if the divisor and dividend signs were different and negating the (corrected) remainder if the dividend was negative.

Value	Function	Action
50	fix	AC fraction is shifted to place the binary point at the right of bit 63. If AC sign is set, the fraction is negated.
48	floatf	Ra and Rb are interpreted as a two's complement, fixed-point number with the binary point between Ra and Rb. Their value is converted to floating point, loaded into AC, and rounded to single precision.
58	floatg	Ra and Rb are interpreted as a two's complement, fixed-point number with the binary point between Ra and Rb. Their value is converted to floating point, loaded into AC, and rounded to double precision.
68	scale	Ra is interpreted as an integer and added to the exponent of AC.
78	ldExp	Ra is interpreted as an integer and is assigned to the exponent of AC.
08	wrSign	The sign bit of AC is loaded from bit 0 of Ra.
18	rdTod	The current time of day value is placed in the AC fraction.
30	wrInt	In kernel mode, Ra and Rb are interpreted as the time of day when the next interval timer interrupt should occur. Ignored in user mode.
38	wrTod	In kernel mode, Ra and Rb are interpreted as the current time of day. The interval timer value is lost, and it requests an interrupt immediately. Ignored in user mode.
28	nop	The coprocessor state is not changed.

Figure 4-50: Miscellaneous Coprocessor Functions

Figure 4-50 shows the miscellaneous coprocessor functions that provide floating point to fixed point conversion, direct loading of the AC fields, and maintenance of the time-of-day clock and interval timer.

The time-of-day register is a 64-bit register that counts time in machines cycles. When the interval timer is loaded with a time between 1 and 2^19 - 1 cycles more than the current value in the time-of-day register, an interrupt occurs when the time-of-day reaches that time. An interrupt also occurs whenever the time-of-day register is written.[5] Note that the interval timer must be serviced within 2^19 cycles

[5] If the processor is in kernel mode at the time, this interrupt is suppressed as are all interrupts. Writing the interval timer after writing the time-of-day register will clear the interrupt.

(approximately 20 milliseconds) after it generates an interrupt, or the time-of-day register will not operate properly.

There are three kinds of floating exceptions:

1. Division by zero is detected at the time a floating point divide instruction is requested, and causes a trap immediately. The requested operation does not occur.

2. Reserved operand is detected at the time such an operand is presented to the coprocessor, and causes a trap immediately. The requested operation does not occur.

3. Range error (overflow and underflow) is detected at the time an out-of-range result is requested by the processor, which will often be the next coprocessor instruction following that which generated such a result. However, range error occurs when the exponent value is not in the range that can be represented in the requested result format, and so depends on what is requested. Range error cannot occur except on instructions which obtain results in floating point formats.

```
|< 1 >|<1 >|< 25 >|< 1 >|< 1  >|< 1  >|< 1 >|< 1 >|<1>|
+-----+----+------+-----+------+------+-----+-----+---+
|     |    |      |     |      |      |     |     |   |
| neg | nz |  0   |abort|gRange|fRange|rsvOp|noDiv|int|
|     |    |      |     |      |      |     |     |   |
+-----+----+------+-----+------+------+-----+-----+---+
```

Figure 4-51: Coprocessor Status Register

The exact type of exception that occurred can be obtained by reading the coprocessor status register. Figure 4-51 shows the format of the status register, where the fields are:

neg	AC is negative
nz	AC fraction is non-zero
abort	Coprocessor instruction was aborted
gRange	AC can’t be represented in G format
fRange	AC can’t be represented in F format
rsvOp	Operand was reserved operand
noDiv	Divide by zero attempted
int	Interval timer interrupt pending

Note that the *gRange* and *fRange* bits are status bits that are set any time the AC can’t be represented in the corresponding format. Therefore, occurrence of a range error requires that the *abort* bit is set and the coprocessor instruction requested a *sp* result and the *fRange* bit is set, or the coprocessor instruction requested a *dph* or *dpl* result and the *gRange* bit is set.

Note that the *abort* bit is cleared by all coprocessor instructions; reading the coprocessor status after traps should be the first coprocessor instruction executed in the trap handler. Writing the status register sets the values of the *acNeg*, *rsvOp*, and *noDiv* bits, and clears the *abort* bit.

The *int* bit continually reflects the state of the interval timer and may change at any time.

The double fix operations never cause any exceptions. To do integer addition with overflow detection, do a double fix addition and then test the least significant bit of the high half of the fraction register.

When servicing coprocessor interrupts/traps the following procedure is suggested:

1. If *int* bit set, update interval timer and restart process.

2. If *abort* bit set and any of the *fRange*, *gRange*, *rsvOp*, or *noDiv* bits are set, then cause an arithmetic exception.

Note that if floating point underflows are being coerced to zero, then in step 2 above, the *fRange*/*gRange* bit should be checked against the exponent to see of the AC should be loaded with zero and the process restarted.

Restrictions

Coprocessor instructions that require multiple cycles to complete must not be immediately followed by another coprocessor instruction, or erroneous results may occur. Refer to Section 4.16 for coprocessor instruction timing.

4.13 User Trap

TASM Format

```
trap literal;
```

Memory Format

```
|<  4  >|<                  22                  >|<    6   >|
+-------+----------------------------------------+----------+
|       |                                        |          |
|   C   |                   0                    | literal  |
|       |                                        |          |
+-------+----------------------------------------+----------+
```

Description

This instruction causes a user trap. The user trap bit and *literal* are set in the program status word. The processor enters kernel mode and jumps to address 00000000. PC2 will contain the address of the trap instruction. PC1 will contain the address of the instruction in execution sequence after the trap instruction. To restart the user instruction stream, the pc-queue must be manually advanced (or the trap instruction would be reexecuted). To advance the pc-queue, reload it so that PC2' = PC1, and PC1' = PC1 + 1.

Restrictions

The processor halts if a trap instruction is executed in kernel mode.

Execution

DO	PSW := literal
EX	trap
WR	-

4.14 Load Address

TASM Format

```
ra := disp[rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<              16              >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|   D   |    ra     |    rb     |             disp              |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

The effective address is written into register *ra*. Note that effective addesses are word addresses (*(rb >> 2) + disp*).

If register *rb* is modified by the immediately preceding instruction, the processor will stall one cycle. When possible, compilers should reorder instructions to minimize such stalls.

Execution

DO	DCAR := (rb >> 2) + disp
EX	RR := DCAR
WR	ra := RR

4.15 Undefined Instruction

TASM Format

```
undef1/2 [ra, rb];
```

Memory Format

```
|<  4  >|<    6    >|<    6    >|<             16              >|
+-------+-----------+-----------+-------------------------------+
|       |           |           |                               |
|  E/F  |    ra     |    rb     |               0               |
|       |           |           |                               |
+-------+-----------+-----------+-------------------------------+
```

Description

If the undefined instructions are executed in user mode, the illegal instruction trap bit will be set in the PSW, PC2 will contain the address of the undefined instruction, PC1 will contain the address of the instruction in execution sequence after the undefined instruction.

If one of the undefined instructions is executed in kernel mode, then the processor halts with the PC containing the address of the instruction in execution sequence after the undefined instruction, and AR, BR with the contents of *ra*, *rb*.

Execution

DO	AR := ra, BR := rb
EX	trap
WR	-

4.16 Instruction Timing

A new instruction is issued every cycle unless one of the following conditions occurs:

Address interlock	If a register is written by some instruction, and used in an address calculation by the next, one additional cycle is lost. However, this does not apply to register r0 (the program counter).
Store cycle	If the instruction following a store is a load or store, one additional cycle is lost.
Trap/interrupt	Any trap condition costs a three cycle delay until the instruction at location 00000000 is executed.
Cache miss	Any instruction fetch, operand load or store which misses the cache results in a delay of about 13 cycles.
Coprocessor	Wait for completion of coprocessor operations can cost up to 58 cycles, depending on the operation in progress. Figure 4-52 shows the number of cycles required to deposit a result in the accumulator, including issuing the coprocessor instruction itself.
I/O instruction	Variable number of cycles if an I/O read or write is in progress when a new I/O instruction is issued. Subject to undefined number of I/O bus and I/O adaptor select stall delays.
Flush instruction	Every clean line requires 3 cycles, every dirty line requires about 13 cycles.

	single float	double float	double fix
addition	4*	4*	1
multiplication	8	18	6
division	29	58	34
float	3	3	-

*** In special cases these may take either 3 or 5 cycles.**

Figure 4-52: Coprocessor Timing

To the extent possible, compiliers should try and reorder instructions to minimize address interlock, store cycle, and coprocessor stalls.

5. Titan I/O Bus

The Titan I/O bus provides a high-speed connection between the I/O adaptors and main memory. It carries control and status information during the execution of CPU I/O instructions, and otherwise is available for DMA traffic initiated by the I/O adaptors.

The I/O bus exists only on the backplane in the Titan system cabinet. It cannot be extended off the backplane, and only one can be provided per processor.

The remainder of this chapter uses the following terminology:

Word	refers to a unit containing 32 bits of data and 4 bits of parity.
Transfer	identifies the movement of one word in one cycle between an I/O adaptor and the memory controller.
Transaction	describes a sequence of cycles, including arbitration, address and data transfers.
Input	refers to transfers from an I/O adaptor to the memory controller.
Output	refers to transfers to an I/O adaptor from the memory control.
DMA	describes transactions initiated by an I/O adaptor.

From the perspective of an I/O adaptor, the terms "CPU", "memory", and "memory controller" are equivalent. The adaptor communicates with them over bus wires that actually connect only to the memory controller, which switches data between the I/O bus lines and the CPU’s cache data lines or the memory data lines.

The Titan I/O bus carries four types of transaction:

CPU input	The CPU sends a word, called *address*, to the adaptor, which responds with a word called *data*.
CPU output	Two words, called *address* and *data*, are sent to an I/O adaptor by the CPU.
DMA input	The I/O adaptor sends a word containing a real address to the memory controller, followed by four words to be stored at the given address and succeeding locations. The address should contain zeros in the least significant two bits.[6]
DMA output	The I/O adaptor sends a word containing a real address to the memory controller, which responds with the four words at the given address and succeeding locations. The address should contain zeros in the least significant two bits.[7]

The I/O bus is comprised of five logical groups of signals as shown in Figure 5-1. The five groups provide DMA bus arbitration, control of CPU input and output cycles, transfer of data, diagnostic scan, and clocks.

DMA Control	I/O Control	Data	Scan	Clocks
dmaRequest	select	inWord	diagStop	sysClk
dmaWr	cmdWr	outWord	diagScan	we422
noArb	cmdStall		scanIn	we474
dmaTransfer	ack		scanOut	reset
lock	interrupt		scanMode	

Figure 5-1: Titan I/O Bus Signal Groups

Figure 5-2 lists the number of each signal in the bus, their direction, their impedence, the number of pins to a given

[6]The memory controller accepts addresses with the least significant two bit non-zero. However, only one line of main memory is written; the least significant two bits specify the modulo order of that the four words are presented to the memory controller, e.g., word 1, word 2, word 3, and then word 0.

[7]The memory controller accepts addresses with the least significant two bit non-zero. However, only one line of main memory is read; the least significant two bits specify the modulo order of that the four words are presented to the I/O adaptor, e.g., word 2, word 3, word 0, and then word 1.

adaptor, and their function. As stated above *in* is from the I/O adaptor to the memory controller.

Signal	Bits	Dir	Ohms	Pins	Function
dmaRequest	7	in	50	7	Bus arbitration
dmaWr	1	in	25	1	DMA data is to memory
noArb	1	out	50	1	Suppress arbitration
dmaTransfer	7	out	50	1	DMA data control
lock	1	both	25	1	Mutual exclusion
select	7	out	50	1	Adaptor selection by CPU
cmdWr	1	out	50	1	Command data is to adaptor
cmdStall	1	in	25	1	CPU must wait for adaptor data
ack	1	in	25	1	Adaptor accepts command
interrupt	7	in	50	1	Interrupt request by adaptor
inWord.data	32	in	25	32	Data from adaptor
inWord.parity	4	in	25	4	Byte parities
outWord.data	32	out	50	32	Data to adaptor
outWord.parity	4	out	50	4	Byte parities
reset	1	out	50	1	Boot time initialization
diagStop	1*	out	50	1	Adaptor diagnostic mode select
diagScan	7*	out	50	1	Scan adaptor in diagnostic mode
scanMode	1*	out	50	1	Scan in progress
scanIn	1*	out	50	1	Input to adaptor's scan chain
scanOut	7*	in	50	1	Output of adaptor's scan chain
clk	7*	out	50	1	System clock
we422, we474	14*	out	50	2	RAM write pulses
	---			--	
	145 bus wires			97 per slot	

* To/from clock/scan board, not memory controller

Figure 5-2: Titan I/O Bus Signals

5.1 Arbitration

Bus arbitration is on a strict priority basis, with distributed arbitration. Each adaptor asserts the *dmaRequest* signal assigned to it when it needs a transaction, and monitors *noArb* and the *dmaRequest* signals for higher priorities. If no higher priority request is asserted, and *noArb* is not asserted, the adaptor wins the arbitration. *NoArb* is asserted by the memory controller from the cycle following arbitration until it is ready to begin another transaction. In the first cycle of a DMA transaction, the adaptor sends an address, and *dmaWr* if this is to be a write transaction. The memory controller accepts the address in this cycle, and asserts the adaptor's *dmaTransfer* line to begin the movement of data between the adaptor and controller.

For a memory write, the adaptor puts the first word of data onto the *inWord* lines immediately after sending the address, and sends the second word in the cycle following that in which it sees *dmaTransfer*; the third and fourth words are transfered in the immediately succeeding cycles.

For a memory read, the memory controller puts the first word of data on the bus in the same cycle as it asserts *dmaTransfer*, and sends the second, third, and fourth words in the following cycles. This definition is intended to allow the memory controller to begin another operation while waiting for read data from the array; each adaptor generally waits for completion of one of its own transactions before requesting another.

It is possible to partially overlap some DMA cycles **after** the memory controller has asserted *dmaTransfer*. For example, while an I/O adaptor is writing the last word of a DMA write transaction onto the bus, it can start bus arbitration for a pending DMA transaction. Similarly, once an I/O adaptor receives *dmaTransfer* for a DMA read transaction (and starts reading data off the *outWord* bus), it can start a pending DMA transaction because the *inWord* bus is idle.

The connections from *dmaRequest* signals to I/O adaptor logic are made through backplane wiring, so the adaptor need not include logic to decide which *dmaRequest* line to assert or which ones to monitor; the adaptor asserts a standard pin, and monitors 6 others.

5.2 CPU Commands

To perform a CPU operation, the memory controller asserts *noArb* and one of the *select* lines (and *cmdWr* if this is a write to the adaptor) during a cycle immediately following one in which none of the *dmaRequests* were asserted. The adaptor selected must accept the address in that cycle, but may (generally will) assert *cmdStall* in the following cycle to indicate that it is either not ready to accept CPU write data or not ready to return CPU read data. The actual data transfer occurs in the first cycle in which *cmdStall* is not asserted; thus, if the transaction is a CPU write, the memory controller must hold the data on the bus until the end of a cycle in which *cmdStall* is not asserted. If it is a CPU read transaction, the controller must use the data on the bus in that cycle in which *cmdStall* is not asserted.

In a CPU operation, the memory controller asserts select for one cycle, but holds *noArb* asserted until the cycle following deassertion of *cmdStall*.

5.3 Retry

Ordinarily, the I/O adaptor acknowledges each CPU transaction by raising *ack* in the data transfer cycle (that is, when it is not asserting *cmdStall*).

In some cases, an I/O adaptor may be in a state in which it is unable to respond to CPU commands for an extended period of time (more than 1 microsecond -- this limit is chosen to prevent overruns on DMA activity), or in which acceptance of a CPU command would create a deadlock. In such a case, the adaptor does not raise *ack* in response to *select*, and the CPU aborts the transaction as if there were no adaptor present. The adaptor may assert *cmdStall* while deciding whether to *ack*. The memory controller records *ack* after each CPU transaction. It is up to software to test this bit after any transaction with a adaptor whose design permits this case, and retry the command if necessary.

5.4 Locking

Certain I/O adaptors may require mutual exclusion from other adaptors and/or the CPU during short sequences of memory operations. Such a adaptor may arbitrate (assert its *dmaRequest*) for a mutually exclusive access only during a cycle immediately following one in which *lock* was not asserted. When it wins arbitration, it asserts *lock* until its need for exclusion is passed. This allows non-mutex transactions to proceed, but excludes other contenders for the lock. Adaptors are expected to use this primitive mechanism to control access to higher-level interlocks in memory whenever the period of exclusion might exceed 5 microseconds. Adaptors which do not require mutual exclusion need not implement this signal.

When the memory controller receives the IoRead instruction which tests and sets the lock, it waits for a cycle in which no *dmaRequest* is set. If *lock* is asserted on the bus in that cycle, the controller returns that fact to the requesting instruction. Otherwise, it asserts *noArb* followed by *lock* on the bus, and returns the fact that *lock* was not set, keeping *lock* on the bus until cleared by software. Programmers are urged to ensure that interrupts are disabled before the lock set, and not re-enabled until after it is cleared. Refer to Sections 2.4.4, 3.8, and 4.2.16 for further disussion of locking and IoRead instructions.

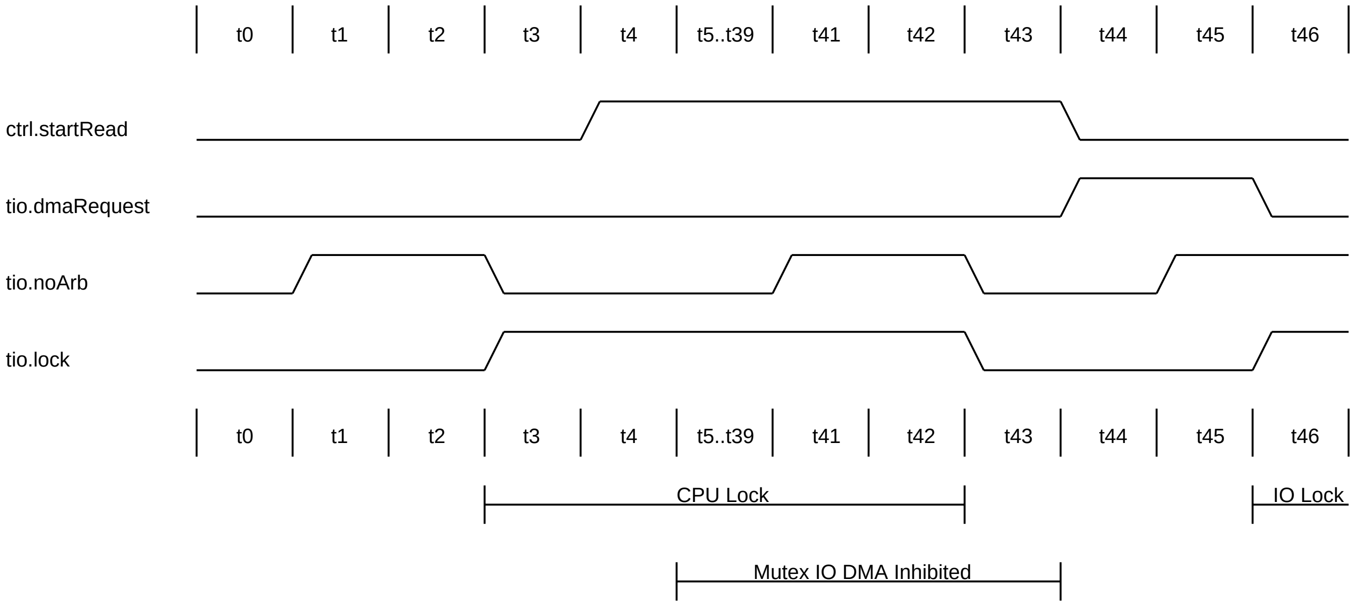

Figure 5-3: I/O Lock Example

Figure 5-3 shows an example of the use of the *lock* signal. *ctrl.startRead* is an internal I/O adaptor signal that indicates that the adaptor is ready to perform a DMA read transaction. *tio.dmaRequest* is the adaptor's I/O bus DMA request signal. *tio.noArb* and *tio.lock* and the I/O bus signals.

t1	The CPU performs an IoRead instruction on the *lock* slot to set the lock.
t2	*tio.lock* was deasserted so the CPU acquires the lock.
t3	The IoRead instruction ends and the memory controller asserts the acquired CPU lock on the bus.
t4	The adaptor is ready to perform a DMA read transaction. The CPU continues to hold the lock.
t5..t40	The adaptor honors the lock and suppresses its DMA request. The CPU continues to hold the lock.
t41	The CPU performs an IoRead instruction on the *lock* slot to clear the lock.
t42	The CPU held the lock, so the memory controller clears the lock.
t43	The IoRead instruction ends and the memory controller deasserts the *tio.lock* signal.
t44	The adaptor observes that the lock has ended and asserts its DMA request signal.
t45	The adaptor wins the bus arbitration.
t46	The adaptor asserts the *tio.lock* signals to inhibit the CPU from gaining the lock until the adaptor has finished a sequence of DMA transactions (not shown).

5.5 Addresses

In DMA transactions, the address presented by the adaptor to the memory controller is a real memory address which identifies the first of a block of four 32-bit words to be transferred. In CPU transactions, the address presented to the adaptor by the memory controller is a 32-bit value generated by the software (specifically, it has not been translated or otherwise modified by hardware). Its interpretation is entirely at the discretion of the adaptor, and independent of the interpretation of addresses by other adaptors. Simple adaptors, for example, may use a few bits to select the adaptor control registers and other bits as a command, ignoring most of the address; a frame buffer might use a large portion of the address to indicate a particular pixel.

5.6 Identification

To simplify the problem of software initialization, each adaptor on the Titan I/O bus responds with a unique identification pattern when it is read by a CPU transaction at its address of all ones (FFFFFFFF). These patterns are listed in Figure 5-4. See also Section 2.4.3.

Type Code	I/O Adaptor
00000000	Empty slot
00000001	Reserved
00000002	TDA (MCSP/SDI Disks)
00000003	Reserved
00000004	TNA (Ethernet)
00000006	TFA (Fiber optic)

Figure 5-4: Titan I/O Adaptor Identifiers

5.7 Parity

All transfers carry odd byte parity on the *inWord* or *outWord* lines; there is no checking on the remaining signals. The memory controller is responsible for asserting correct parity on the *outWord* lines whenever it is sending an address or data to an adaptor, and the adaptor is expected to check received parity and report any errors detected when it is selected. Adaptors are responsible for asserting correct parity on the *inWord* lines whenever they are driving data onto the bus (parity need not be correct while the adaptor asserts *cmdStall*, or if it does not assert *ack*). The memory controller checks parity of data and addresses received from the adaptors.

These rules imply:

- Bus parity is incorrect when no driver is enabled.
- No adaptor should report a bus error except as a result of a transfer it thought was directed to it.
- Adaptors and the memory controller are permitted to drive incorrect parity onto the bus while they have stall asserted.

5.8 Interrupts

An adaptor requests an interrupt by (synchronously) asserting its *interrupt* line; it deasserts the line when software in the CPU issues I/O instructions to the adaptor that clear the condition which caused the interrupt.

5.9 Scan

Each adaptor must connect its bus registers in a scan chain, with *scanIn* as the shift input and *scanOut* as the output. When *diagStop* is asserted without *diagScan*, these registers should be forced into *hold*, and when both *diagStop* and *diagScan* are asserted, shifting should occur synchronous to the system clock. It is desireable for the scan chain to include as much as possible of the adaptor state, but probably excludes that portion of the adaptor which is not on the system clock and/or not 100k ECL.

Note that when *diagStop* is asserted, *reset* and RAM write pulses should be suppressed by I/O adaptors. The I/O bus reset signal is generated by the memory controller and is part of its internal, scannable state. Therefore, during a scan of the memory controller, the I/O bus reset signal will pulse. This should **not** reset I/O adaptors. Similarly, during scan of an I/O adaptor, its internal control signals will temporarily take on random values, possibly causing spurious writes of internal RAMs if those control signals are directly gated with the backplane write enable signals.

Figure 5-5 shows a typical diagnostic stop and scan of an I/O adaptor. When *diagStop* is asserted, the I/O adaptor should hold its internal state. Every clock cycle that both *diagStop* and *diagScan* are asserted the I/O adaptor should shift its internal state by one bit. Scan of the entire internal state occurs in a sequence of 8-bit scans, terminated by a final length-mod-8-bit scan if the total scan chain length is not an integral multiple of 8 bits. The external host controlling the scan operation *knows* the length of a given adaptor's scan chain and instructs the clock/scan module accordingly.

If adaptors wish to include logic in their scan chain that require a clock cycle to convert between *normal* and *scan* modes, the *scanMode* signal should be used to put such logic in its scan mode on the rising edge of *scanMode* and back to its normal mode on the falling edge of *scanMode*.

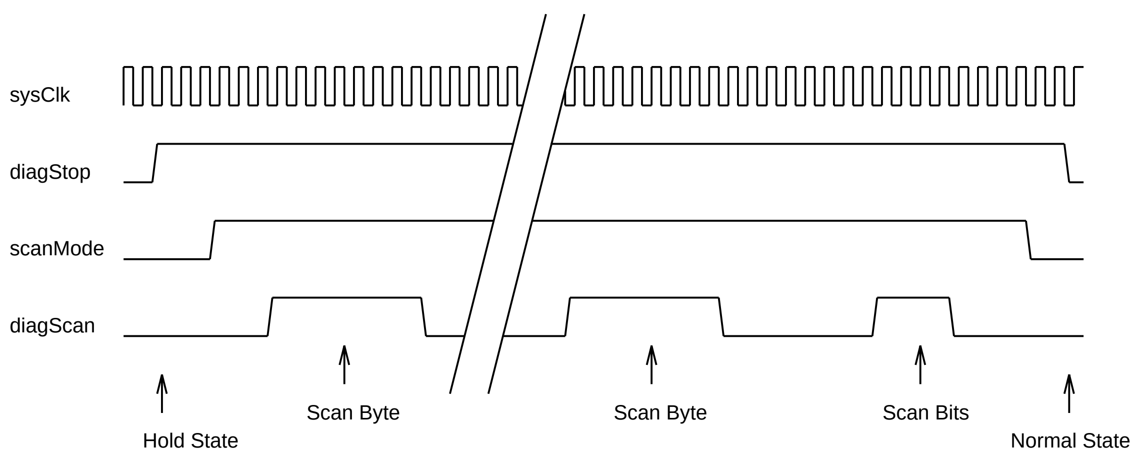

Figure 5-5: Diagnostic Stop and Scan of Adaptor

5.10 Conventions

The Titan I/O bus uses ECL 100k devices for all its signals, which are received directly by the parallel data inputs of 100141's clocked by the system clock. Bus discipline does not permit logic between backplane signals and the register, nor more than one receiver per signal, but does permit a small amount of logic before the drivers.

Bus stubs may be no more than 1.5 inches, which implies that bus driver and receiver chips must be along the connector edge of the board. All bus signals use 50 ohm transmission lines, but those which have multiple drivers must be terminated at both ends, so they require 25 ohm drivers, obtained by 100123, (preferred) or by paralleling the outputs of a 100113 (where inversion is required). Other signals may be driven by any ECL 100k output.

Bus interfaces are required to provide at least 2.0 ns minimum propagation and wire delay from system clock to bus pins, and no more than 15.0 ns maximum delay. *Select* and *dmaTransfer* signals are radially distributed (a separate signal goes to each adaptor), and therefore have less stringent timing constraints: the memory controller must assert them to the bus no later than 20 ns, and the adaptor can expect them to stabilize by 25 ns, allowing a small amount of logic before a register.

5.11 Clocks

The system clock is distributed by a common 100113 driver to all adaptors on the I/O bus through individual equal-length traces in the backplane. Adaptors are expected to buffer the clock with a 100113 before distribution on the board; this provides 8 copies in each polarity, and when considered with the minimum propagation delay spec above, avoids the need for deskew adjustments.

Nominal clock net length on an I/O adaptor is 22.0 inches. Note that this is total clock net length on the adaptor, e.g., if the backplane connector to a 100113 input net is 2.0 inches in length, the 100113 output to load lengths would be 20.0 inches.

If a TTL clock is needed, it should be derived from the clock pin by a 100125.

In addition to the system clock, each I/O slot receives signals from the clock module designated as ~we422 and ~we474, suitable for buffering and use as write pulses for 100422 (256x4) and 100474 (1024x4) rams, respectively. The system clock is symmetric, with 20 ns high and 20 ns low. The write pulses are timed with respect to the falling edge (middle) of the cycle as shown in Figure 5-6.

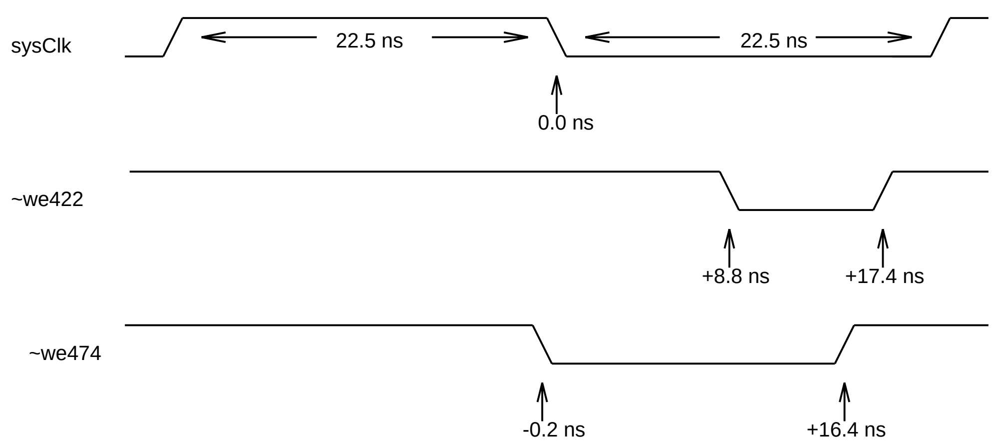

Figure 5-6: Write Pulse Timing

5.12 Termination

Each adaptor must provide 50-ohm termination for its own copy of the *select*, *dmaTransfer*, and *clk* signals. In addition, each single-slot adaptor must provide 83 uncommitted terminators on backplane pins (specified in Figures 5-15, 5-16, and 5-17) so that if it is plugged into the last slot, they can terminate the bus. The backplane jumpers I/O bus signals needing termination to nearby pins of slot 26 (IO1) that are assigned to be terminator pins. This implies that slot 26 (IO1) must always be populated.

The memory controller terminates the *inWord*, *interrupt*, *dmaRequest*, *cmdStall*, *dmaWr*, *ack*, and *lock* signals.

To facilitate implementation of two-slot I/O adaptors, slots 20 (IO7) and 21 (IO6), slots 22 (IO5) and 23 (IO4), and slots 24 (IO3) and 25 (IO2) have their terminator pins connected between the two slots of each pair by the backplane. Furthermore, the 108 pins not assigned as bus signals, terminators, or grounds in connector blocks 1 and 2 are jumpered between these slot pairs. This provides up to 191 connections between the two boards of the I/O adaptor without using external cables. This also implies that such adaptors can only be positioned in one of these three slot pairs.

5.13 Cpu Write Transactions

A CPU write transaction is shown in Figure 5-7. Signal names with a *tio* prefix are backplane signals; signal names with a *state* prefix are adaptor internal FSM outputs; signal names with a *ctrl* prefix are adaptor internal control signals.

In this example, the I/O bus and the adaptor are idle before the memory controller starts the CPU write.

t1	The memory controller selects the adaptor for a CPU write by asserting *tio.select* and *tio.cmdWr*. It also asserts *tio.noArb* to inhibit DMA activity. The adaptor's bus interface then generates the *ctrl.ioWriteAddr* internal signal and prepares to latch the address that the memory controller is driving onto the *outWord* bus.
t2	The memory controller deasserts *tio.select* and *tio.cmdWr*, and drives the contents of its *IoWriteData* register onto the *outWord* bus. The adaptor's bus interface latches the address and checks its parity. The adaptor's bus FSM enters *state.ioWrite*, and asserts *tio.cmdStall* because it cannot complete the write in one cycle.
t3	The memory controller continues to drive the write data onto the *outWord* bus. The adaptor continues processing the CPU write transaction.
t4	The memory controller continues to drive the write data onto the *outWord* bus. The adaptors completes processing the CPU write transaction and deasserts *tio.cmdStall*. It also asserts *tio.ack* to indicate that the transaction was successfully completed.
t5	The memory controller sees *tio.cmdStall* deasserted and completes the CPU write transaction, latching the value of *tio.ack* in its *IoStatus* register. It also deasserts *tio.noArb* to allow DMA activity to resume. The adaptor's bus interface leaves *state.ioWrite*.

If the bus had not been idle, extra cycles would have occurred before *tio.select* was asserted in t1. The memory controller only asserts *tio.select* if the previous cycle has *tio.noArb* deasserted, and all of the DMA request signals deasserted.

If the adaptor can complete the CPU write in one cycle, *tio.ack* would be asserted in t2, and *tio.cmdStall* would never have been asserted.

The adaptor should check parity of the write data on the cycle that it latches it off the bus.

5.14 Cpu Read Transactions

A CPU read transaction is shown in Figure 5-8. Signal names with a *tio* prefix are backplane signals; signal names with a *state* prefix are adaptor internal FSM outputs; signal names with a *ctrl* prefix are adaptor internal control signals.

In this example, the I/O bus and the adaptor are idle before the memory controller starts the CPU read.

t1	The memory controller selects the adaptor for a CPU read by asserting *tio.select* and deasserting *tio.cmdWr*. It also asserts *tio.noArb* to inhibit DMA activity. The adaptor's bus interface then generates the *ctrl.ioReadAddr* internal signal and prepares to latch the address that the memory controller is driving onto the *outWord* bus.
t2	The memory controller deasserts *tio.select*. The adaptor's bus interface latches the

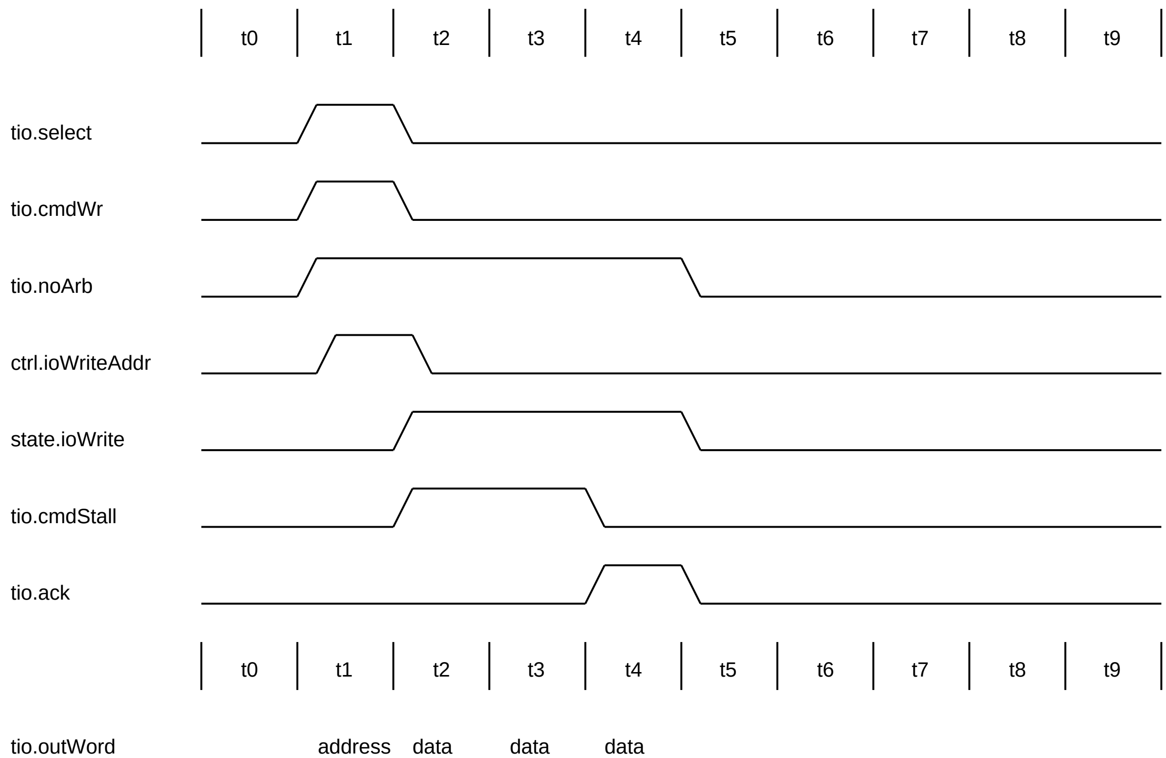

Figure 5-7: Cpu Write Timing

	address and checks its parity. The adaptor's bus FSM enters *state.ioRead*, and asserts *tio.cmdStall* because it cannot complete the read in one cycle.
t3	The adaptor continues processing the CPU read transaction.
t4	The adaptors completes processing the CPU read transaction, drives the read data onto the *inWord* bus, and deasserts *tio.cmdStall*. It also asserts *tio.ack* to indicate that the transaction was successfully completed.
t5	The memory controllers sees *tio.cmdStall* deasserted and completes the CPU read transaction, latching the value of *tio.ack* in its *IoStatus* register. It also deasserts *tio.noArb* to allow DMA activity to resume. The adaptor's bus interface leaves *state.ioRead*.

If the bus had not been idle, extra cycles would have occurred before *tio.select* was asserted in t1. The memory controller only asserts *tio.select* if the previous cycle has *tio.noArb* deasserted, and all of the DMA request signals deasserted.

If the adaptor can complete the CPU read in one cycle, *tio.ack* would be asserted in t2, and *tio.cmdStall* would never have been asserted.

5.15 DMA Read Transactions

A DMA read transaction is shown in Figure 5-9. Signal names with a *tio* prefix are backplane signals; signal names with a *state* prefix are adaptor internal FSM outputs; signal names with a *ctrl* prefix are adaptor internal control signals.

In this example, the I/O bus and the memory controller are idle before this adaptor starts the DMA read.

t1	The adaptor requests a DMA read of its bus interface by asserting *ctrl.startRead*.

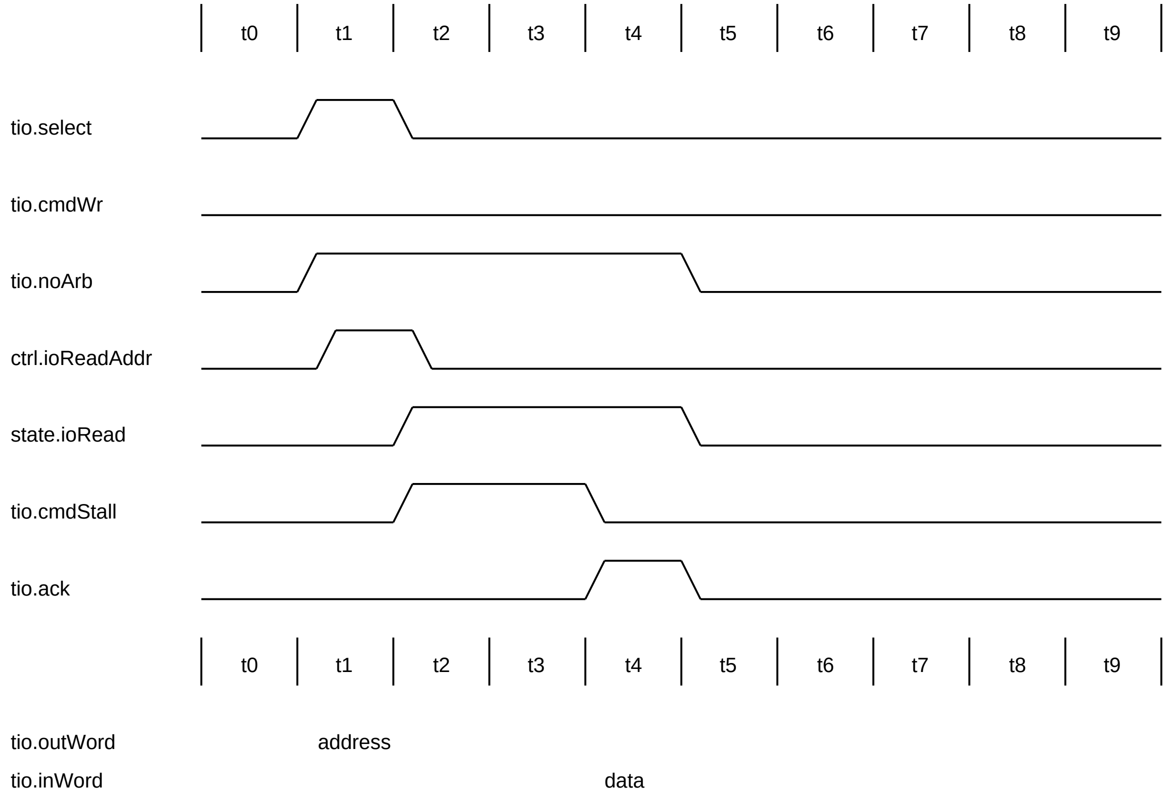

Figure 5-8: Cpu Read Timing

t2 — The adaptor's bus FSM enters *state.readAddress* and asserts *tio.dmaRequest*. The adaptor's bus interface also asserts *ctrl.readBusy* to inhibit other internal bus requests.

t3 — In the previous cycle, *tio.dmaRequest* was asserted (and no higher priority DMA requests are asserted since the bus is idle) and *tio.noArb* was deasserted. Therefore, the bus interface generates *ctrl.dmaAck*, which enables driving the read address onto the *inWord* bus.

t4 — The adaptor's bus FSM leaves *state.readAddress* and enters *state.readWait*. It also stops asserting *tio.dmaRequest*. The memory controller latches the read address from the *inWord* bus and starts processing the read cycle.

t5 — The memory controller continues processing the read cycle.

t6 — The memory controller continues processing the read cycle.

t7 — The memory controller continues processing the read cycle.

t8 — The memory controller deasserts *tio.noArb* to allow other adaptors to start bus arbitration.

t9 — The memory controller continues processing the read cycle.

t10 — The memory controller continues processing the read cycle.

t11 — The memory controller continues processing the read cycle.

t12 — The memory controller drives word 0 of the line onto the *outWord* bus and asserts *tio.dmaTransfer*. The bus interface deasserts *ctrl.readBusy*, allowing pending DMA requests from the adaptor to start.

t13 — The bus interface leaves *state.readWait* and enters *state.readData0*. The bus interface latches word 0 off the bus, checks parity, and provides it to the adaptor. The memory controller drives word 1 of the line onto the bus.

t14 — The bus interface leaves *state.readData0* and enters *state.readData1*. The bus interface latches word 1 off the bus, checks parity, and provides it to the adaptor. The memory controller drives word 2 of the line onto the bus.

t15	The bus interface leaves *state.readData1* and enters *state.readData2*. The bus interface latches word 2 off the bus, checks parity, and provides it to the adaptor. The memory controller drives word 3 of the line onto the bus.
t16	The bus interface leaves *state.readData2* and enters *state.readData3*. The bus interface latches word 3 off the bus, checks parity, and provides it to the adaptor.

If the bus had not been idle, extra cycles would have occurred before *ctrl.dmaAck* was asserted in t3. If the memory controller had not been idle (CPU activity or DMA write completing), extra cycles would have occurred before *tio.dmaTransfer* is asserted in t12.

5.16 DMA Write Transactions

A DMA write transaction is shown in Figure 5-10. Signal names with a *tio* prefix are backplane signals; signal names with a *state* prefix are adaptor internal FSM outputs; signal names with a *ctrl* prefix are adaptor internal control signals.

In this example, the I/O bus and the memory controller are idle before this adaptor starts the DMA read.

t1	The adaptor requests a DMA write of its bus interface by asserting *ctrl.startWrite*.
t2	The adaptor's bus FSM enters *state.writeAddress* and asserts *tio.dmaRequest*. The adaptor's bus interface also asserts *ctrl.writeBusy* to inhibit other internal bus requests.
t3	In the previous cycle, *tio.dmaRequest* was asserted (and no higher priority DMA requests are asserted since the bus is idle) and *tio.noArb* was deasserted. Therefore, the bus interface generates *ctrl.dmaAck*, which enables driving the write address onto the *inWord* bus.
t4	The adaptor's bus FSM leaves *state.writeAddress* and enters *state.writeData0*. It also stops asserting *tio.dmaRequest*. The memory controller latches the write address from the *inWord* bus. The bus interface drives word 0 of the line onto the *inWord* bus.
t5	The memory controller asserts *tio.dmaTransfer* to indicate that its ready for the write data. The bus interface continues to drive word 0 onto the *inWord* bus. The bus interface deasserts *ctrl.writeBusy*, allowing pending DMA requests from the adaptor to start (though they would not gain access to the bus as *tio.noArb* is still asserted).
t6	The memory controller latches word 0 off the *inWord* bus. The bus interface leaves *state.WriteData0* and enters *state.WriteData1*. The bus interface drives word 1 onto the *inWord* bus.
t7	The memory controller latches word 1 off the *inWord* bus. The bus interface leaves *state.WriteData1* and enters *state.WriteData2*. The bus interface drives word 2 onto the *inWord* bus.
t8	The memory controller latches word 2 off the *inWord* bus. The memory controller deasserts *tio.noArb* so that a new DMA arbitration can occur on the following cycle. The bus interface leaves *state.WriteData2* and enters *state.WriteData3*. The bus interface drives word 3 onto the *inWord* bus.
t9	The memory controller latches word 3 off the *inWord* bus and starts processing the write cycle. The bus interface leaves *state.WriteData3*.

If the bus had not been idle, extra cycles would have occurred before *ctrl.dmaAck* was asserted in t3. If the memory controller had not been idle (CPU activity or DMA write completing), extra cycles would have occurred before *tio.dmaTransfer* was asserted in t5.

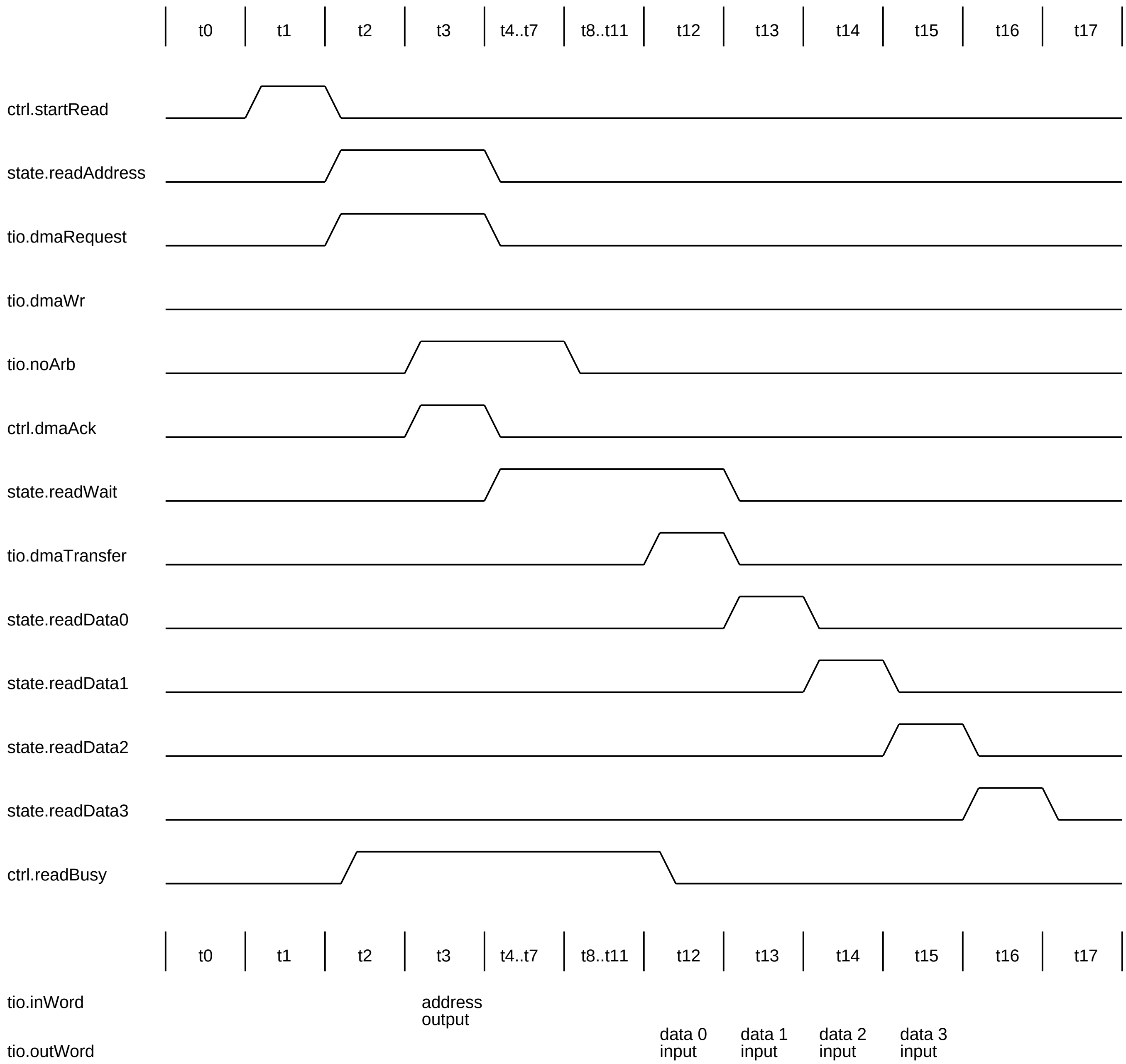

Figure 5-9: DMA Read Transaction

5.17 Physical Dimensions

Figure 5-11 shows a component-side view of an I/O adaptor. The board size is approximately the same as a DEC extended-hex board. Using a 600 mil by 1300 mil grid of 24-pin ECL parts and termination SIPs, this provides 200 IC/SIP positions. I/O adaptor boards are 12 layer printed circuit boards, generally comprised of 6 signal layers and 6 supply/ground planes.

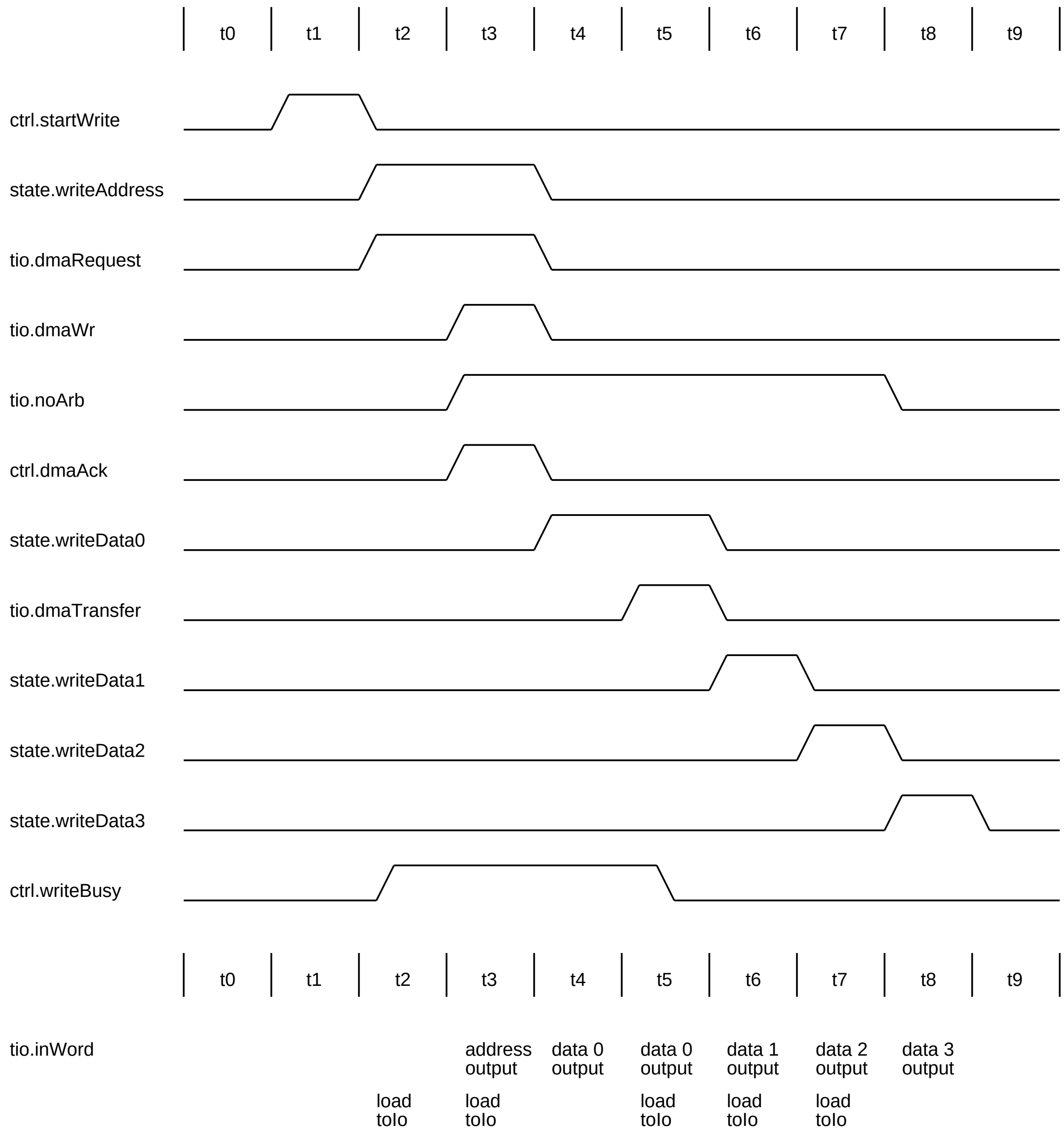

Figure 5-10: Dma Write Timing

5.18 Backplane

Figure 5-12 shows a pin-side view of the 26-slot Titan backplane with the processor connectors, P3..P7 and P10..P14, the three connector blocks for each backplane slot, and the DC power rails. Above the drawing of the backplane, the slots numbers and module types are listed for each backplane slot. The backplane is a 10 layer printed circuit board, comprised of 6 signal layers and 4 ground planes.

Figure 5-13 lists the mnemonics used for the module types. For the memory array modules, the numeric suffix

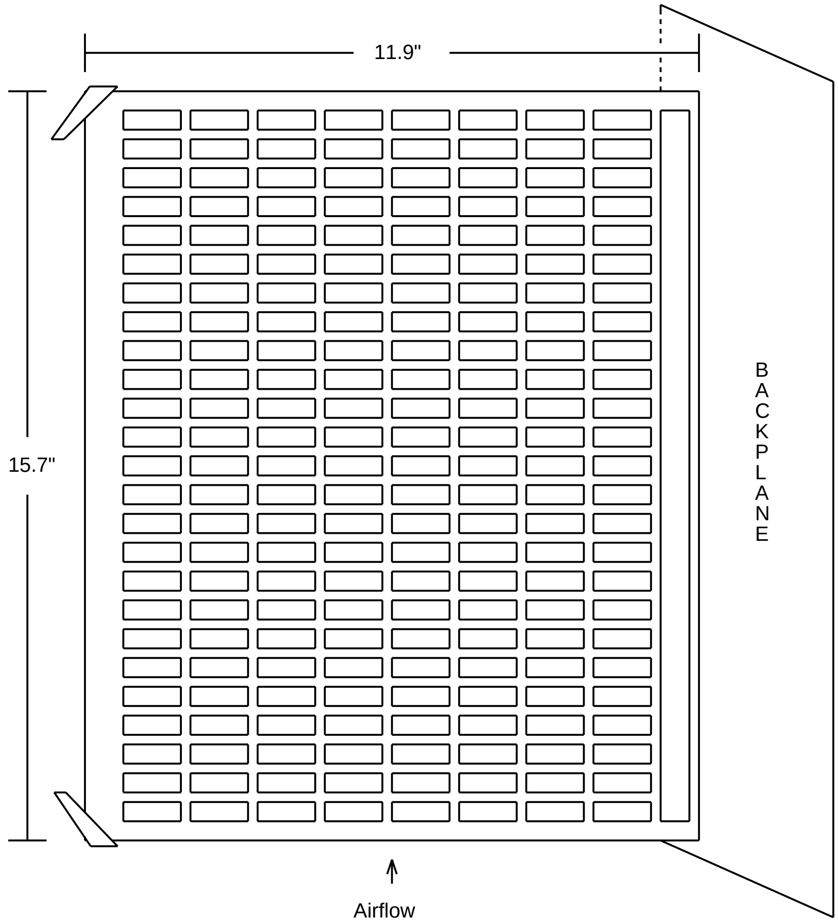

Figure 5-11: I/O Adaptor [Component-Side View]

indicates the memory module number [0..3] and the word of the memory line within that module [0..3]. For example, the 32-bit word at physical address 00000002 is in MA02, and the word at physical address 00100003 is in MA23. However, all memory array boards are identical, the backplane slot implicitly specifies the memory module and line word. For the I/O adaptor modules, the numeric suffix idicates the bus arbitration priority of the slot [1..7]; with slot IO1 having the highest priority, and slot IO7 having the lowest priority.

Pins of a given backplane slot's connectors are identified by a four digit number of the form BRRC. The most significant digit is the connector block [1..3]. The middle two digits are the connector row number, [01..40] for blocks 1 and 3, and [01..50] for block 2. The least significant digit is the connector column, [1..3] for memory array modules, [1..4] for all other modules. Figure 5-14 shows the number of the four corner pins of a four column, block 1 connector as viewed from the pin side of the backplane. As another example, referring to Figure 5-12, the lower, right-hand corner pin of slot 19's (MA00) block 2 connector is 2503.

The power rails are positioned to give distributed grounds, for signal integrity, and to move ECL circuitry near connector block 1 and TTL circuitry near connector block 3. The supply return rails (0 V) are not attached to the backplane ground planes, to decrease noise coupling between supply- and signal-ground systems. The ground systems are connected via the ground and supply-return planes of memory and I/O boards.

Cooling air moves from bottom to top, from block 3 to block 1. Because of its temperature compensation, ECL

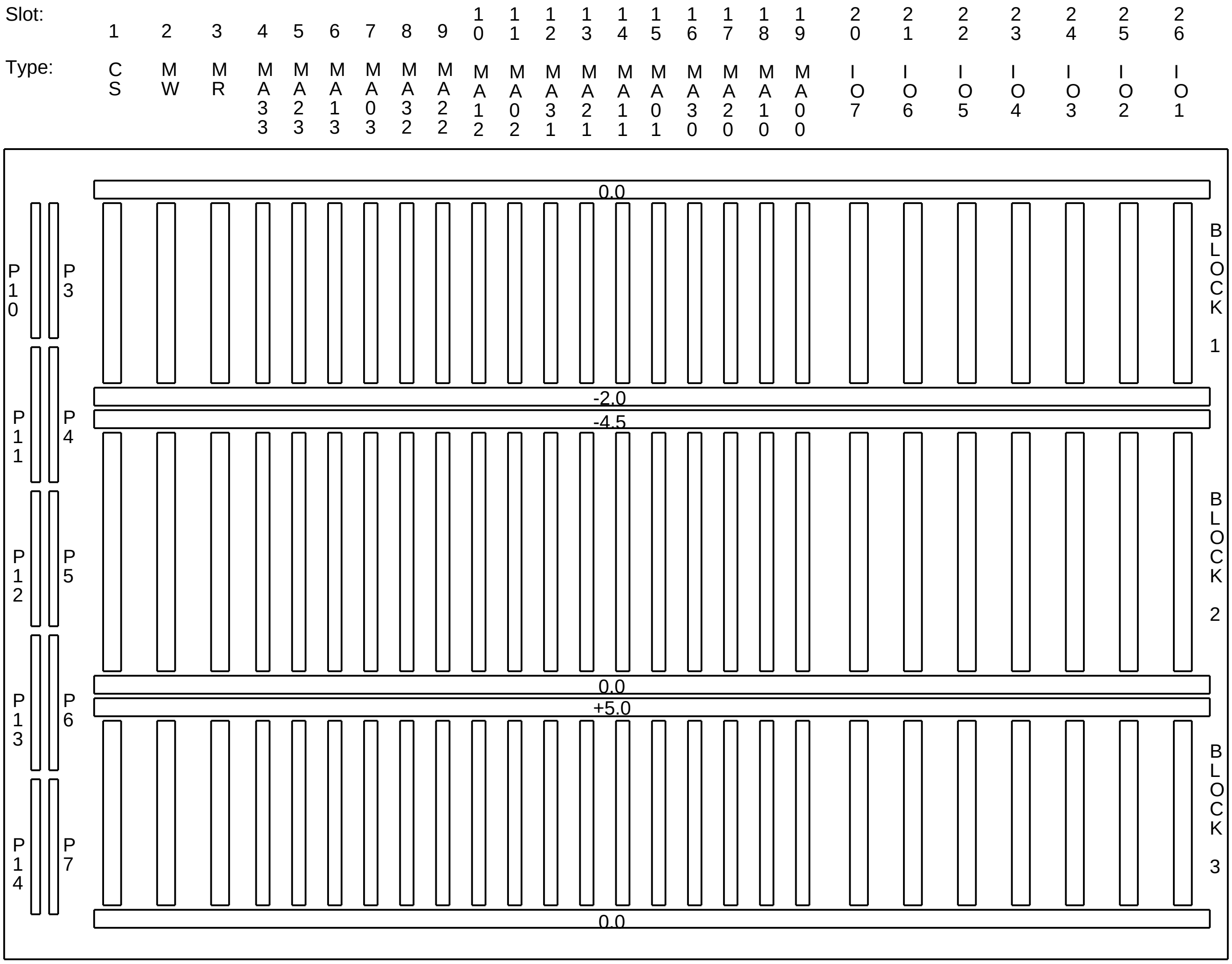

Figure 5-12: Backplane Pin-Side View

Number	Type	Description
1	CS	Clock/Scan
1	MW	Memory Write Controller
1	MR	Memory Read Controller
16	MAMW	Memory Array For Module ’M’, Word ’W’
7	IOP	I/O Adaptor At Priority ’P’
--		
26		Total Slots

Figure 5-13: Backplane Module Quantity and Types

should be more tolerant of elevated temperatures than TTL.

Figure 5-1 show the groups of signals that form the Titan I/O bus. Assignment of these signals to connector blocks 1, 2, and 3 is shown in Figures 5-15, 5-16, and 5-17.

For the *outWord.data* and *inWord.data* busses, bit 31 is the most significant, and bit 0 is the least significant. E.g., if *outWord.data* has the value 80000000 then *outWord.data[31]* is 1 and all other bits are 0. *outWord.parity[3]* is for bits *outWord.data[31..24]*; *outWord.parity[2]* is for bits *outWord.data[23..16]*; etc.

```
                C  C  C  C
                o  o  o  o
Pin 1011        l  l  l  l    Pin 1014
        \                    /
         \      1  2  3  4  /
          \                /
                .  .  .  .  Row 01

                .  .  .  .  Row 02

                .  .  .  .  Row 03


Block 1

                .  .  .  .  Row 38

                .  .  .  .  Row 39

                .  .  .  .  Row 40
               /          \
              /            \
             /              \
Pin 1401                     Pin 1404
```

Figure 5-14: Backplane Connector Pin Numbering [Pin Side]

Signals are all assigned in columns 3 and 4, where the air loop of the connector pin is shortest. This minimizes inductance at the bus interface.

Terminators which will be used only in slot 26 (IO1) are assigned the remaining pins near signals which will need termination. Note that slot 26 (IO1) must be populated, or the bus will not be properly terminated. Adaptors which need two slots, and therefore cannot use slot 26, may use these pins for communication between even and odd slots.

Pins are aligned with appropriate package pins subject to the constraint to columns 3 and 4, and minimizing the length of the longest stubs. No interface chip pin is more than 1.8 inches from the via at which the backplane pin is connected. (There is an additional half inch or so through the connector block to the backplane.)

Grounds assigned in columns 1 and 3 have pin numbers equal to 4 mod 6, and in columns 2 and 4 have pin numbers equal to 1 mod 6. This places grounds no farther than 0.2" from any signal, and gives one ground for every five signals.

Most of connector block 3 is unused by the Titan I/O bus, and unconnected on the backplane, making it available for use with push-on headers to ribbon cables for bulkhead connections. Pins 340X to 335X have low-current auxilliary voltages that may be needed for special interfaces as shown in Figure 5-18. The supply return (0 V) pins are tied to the backplane supply return (0 V) rails. The GND pins are tied to the backplane ground planes; for use by I/O cables that need a logic signal ground. The FRAME pins are tied to the cabinet; for use by I/O cables that need a *drain* wire.

The backplane connections for DMA request signals are performed as shown in Figure 5-19, to provide the fixed-priority bus arbitration of the I/O slots.

Pin	Col 1	Col 2	Col 3	Col 4
101	jump01	gnd	scanIn	gnd
102	jump02	term01	outWord.data[30]	outWord.data[31]
103	jump03	term02	outWord.data[28]	outWord.data[29]
104	gnd	term03	gnd	outWord.data[27]
105	jump04	term04	outWord.data[26]	inWord.data[31]
106	jump05	term05	outWord.data[24]	outWord.data[25]
107	jump06	gnd	term06	gnd
108	jump07	term09	term08	term07
109	jump08	term10	inWord.data[29]	inWord.data[30]
110	gnd	term11	gnd	inWord.data[28]
111	jump09	term12	inWord.data[27]	inWord.data[25]
112	jump10	term14	term13	inWord.data[26]
113	jump11	gnd	term15	gnd
114	jump12	term18	term17	term16
115	jump13	term19	inWord.data[23]	inWord.data[24]
116	gnd	term20	gnd	inWord.data[22]
117	jump14	term21	inWord.data[21]	inWord.data[20]
118	jump15	term24	term23	term22
119	jump16	gnd	term25	gnd
120	jump17	term26	outWord.data[22]	outWord.data[23]
121	jump18	term27	outWord.data[20]	outWord.data[21]
122	gnd	term28	gnd	outWord.data[19]
123	jump19	term29	outWord.data[18]	inWord.data[19]
124	jump20	term30	outWord.data[16]	outWord.data[17]
125	jump21	gnd	term31	gnd
126	jump22	term34	term33	term32
127	jump23	term35	inWord.data[17]	inWord.data[18]
128	gnd	term36	gnd	inWord.data[16]
129	jump24	term37	inWord.data[14]	inWord.data[15]
130	jump25	term40	term39	term38
131	jump26	gnd	term41	gnd
132	jump27	term43	term42	outWord.data[15]
133	jump28	term44	outWord.data[13]	outWord.data[14]
134	gnd	term45	gnd	outWord.data[12]
135	jump29	term46	outWord.data[11]	inWord.data[13]
136	jump30	term47	outWord.data[09]	outWord.data[10]
137	jump31	gnd	outWord.data[08]	gnd
138	jump32	term48	inWord.data[11]	inWord.data[12]
139	jump33	term49	inWord.data[09]	inWord.data[10]
140	gnd	term50	gnd	inWord.data[08]

Figure 5-15: Block 1 Signal Assignments

Pin	Col 1	Col 2	Col 3	Col 4
201	jump34	gnd	reserved	gnd
202	jump35	term51	reserved	reserved
203	jump36	term52	outWord.data[06]	outWord.data[07]
204	gnd	term53	gnd	outWord.data[05]
205	jump37	term54	outWord.data[03]	outWord.data[04]
206	jump38	term55	outWord.data[02]	inWord.data[07]
207	jump39	gnd	outWord.data[01]	gnd
208	jump40	term56	term81	outWord.data[00]
209	jump41	term57	term82	reserved
210	gnd	term58	gnd	inWord.data[06]
211	jump42	term59	inWord.data[04]	inWord.data[05]
212	jump43	term60	inWord.data[03]	inWord.data[01]
213	jump44	gnd	inWord.data[02]	gnd
214	jump45	term61	reserved	reserved
215	jump46	term62	reserved	reserved
216	gnd	term63	gnd	inWord.data[00]
217	jump47	term64	inWord.parity[2]	inWord.parity[3]
218	jump48	term65	inWord.parity[0]	inWord.parity[1]
219	jump49	gnd	reserved	gnd
220	jump50	term66	reserved	scanMode
221	jump51	term67	outWord.parity[2]	outWord.parity[3]
222	gnd	term68	gnd	outWord.parity[1]
223	jump52	term69	outWord.parity[0]	cmdWr
224	jump53	term70	dmaTransfer[dev]	select[dev]
225	jump54	gnd	we422[dev]	gnd
226	jump55	term71	we474[dev]	sysClk[dev]
227	jump56	term72	dmaReq[4]	dmaReq[5]
228	gnd	term73	gnd	dmaReq[3]
229	jump57	term74	dmaReq[2]	dmaReq[1]
230	jump58	term75	dmaReq[0]	cmdStall
231	jump59	gnd	noArb	gnd
232	jump60	term76	diagStop	diagScan[dev]
233	jump61	term77	scanOut[dev]	reset
234	gnd	term78	gnd	ack
235	jump62	term79	interrupt[dev]	dmaWr
236	jump63	term80	lock	dmaRequest[dev]
237	jump64	gnd	term83	gnd
238	jump65	jump76	jump87	jump98
239	jump66	jump77	jump88	jump99
240	gnd	jump78	gnd	jump100
241	jump67	jump79	jump89	jump101
242	jump68	jump80	jump90	jump102
243	jump69	gnd	jump91	gnd
244	jump70	jump81	jump92	jump103
245	jump71	jump82	jump93	jump104
246	gnd	jump83	gnd	jump105
247	jump72	jump84	jump94	jump106
248	jump73	jump85	jump95	jump107
249	jump74	gnd	jump96	gnd
250	jump75	jump86	jump97	jump108

Figure 5-16: Block 2 Signal Assignments

Pin	Col 1	Col 2	Col 3	Col 4
301				
302				
303				
304				
305				
306				
307				
308				
309				
310				
311				
312				
313				
314				
315				
316				
317				
318				
319				
320				
321				
322				
323				
324				
325				
326				
327				
328				
329				
330				
331				
332				
333				
334				
335	gnd	gnd	gnd	gnd
336	+15 V	+15 V	+15 V	+15 V
337	0 V	0 V	0 V	0 V
338	Frame	Frame	Frame	Frame
339	-15 V	-15 V	-15 V	-15 V
340	-5.2 V	-5.2 V	-5.2 V	-5.2 V

Figure 5-17: Block 3 Signal Assignments

Pin	Voltage	Max Current
340X	-5.2 V	10 A
336X	+15 V	10 A
339X	-15 V	10 A
337X	0 V	-
335X	GND	-
338X	FRAME	-

Figure 5-18: Auxilliary Low-Current Power

```
                Memory controller request lines
                1   2   3   4   5   6   7
                |   |   |   |   |   |   |
                |   |   |   |   |   |   |
Slot 20         x   x   x   x   x   x   IO7 dmaRequest
low priority    |   |   |   |   |    \_
                |   |   |   |   |      \
Slot 21         x   x   x   x   x   nc  IO6 dmaRequest
                |   |   |   |    \_____
                |   |   |   |          \
Slot 22         x   x   x   x   nc  nc  IO5 dmaRequest
                |   |   |    \_________
                |   |   |              \
Slot 23         x   x   x   nc  nc  nc  IO4 dmaRequest
                |   |    \_____________
                |   |                  \
Slot 24         x   x   nc  nc  nc  nc  IO3 dmaRequest
                |    \_________________
                |                      \
Slot 25         x   nc  nc  nc  nc  nc  IO2 dmaRequest
                 \_____________________
                                       \
Slot 26         nc  nc  nc  nc  nc  nc  IO1 dmaRequest
high priority
```

Figure 5-19: DMA Request Connections

6. Clock/Scan Module (CSM)

The clock/scan module (CSM) provides remote boot/halt and diagnostic scan facilities for the Titan via the Ethernet. It also monitors boot switches and drives a halt LED on the local maintenance panel .

6.1 Packet Format

The CSM packet format is shown in Figure 6-1. This format applies to both received and transmitted packets. The protocol has been designed to require the minimum amount of microcode, and to leave as many decisions as possible to higher level software. The host computer initiates transactions; the CSM never initiates communications. Once the CSM receives a packet it will execute the command and acknowledge with a reply packet.

```
     +---------------+
0000 | DestAddress   |
     +---------------+
0006 | SrcAddress    |
     +---------------+
000C | Type          |
     +---------------+
000E | SeqNumber     |
     +---------------+
000F |               |
     | Header        |
001F |               |
     +---------------+
0020 | Version       |
     +---------------+
0021 | ScanInModeA   |
     +---------------+
0022 | ScanInBytes   |
     +---------------+
0023 | ScanInBits    |
     +---------------+
0024 | Go            |
     +---------------+
0028 | ScanOutModeA  |
     +---------------+
0029 | ScanOutBytes  |
     +---------------+
002A | ScanOutBits   |
     +---------------+
002B | EndModeB      |
     +---------------+
002C | EndModeA      |
     +---------------+
002D | SystemStatus  |
     +---------------+
002E | Reserved      |
     +---------------+
0030 |               |
     | ScanData      |
007F |               |
     +---------------+
```

Figure 6-1: CSM Packet Format

The packet fields are:

Dest Address	Ethernet destination address.
Src Address	Ethernet source address.
Type	Ethernet packet type; must be 803A.
SeqNumber	Sequence number.
Header	Header for higher-level protocols.
Version	CSM microcode version, currently 3.
ScanInModeA	Value for *ModeA* register during scan-in.
ScanInBytes	Number of bytes to scan-in from packet.
ScanInBits	Number of bits to scan-in from packet (after ScanInBytes).
Go	Bit vectors for number of single-step cycles.
ScanOutModeA	Value for *ModeA* register during scan-out.
ScanOutBytes	Number of bytes to scan-out from packet.
ScanOutBits	Number of bits to scan-out from packet (after ScanOutBytes).
EndModeB	Value for *ModeB* register after scan operations.
EndModeA	Value for *ModeA* register after scan operations.
SystemStatus	Value of CPU and memory controller halt signals.
ScanData	Up to 80 bytes of scan data.

The CSM microcode implements the following algorithm:

1. Discard the packet if its a broadcast.
2. Check the *Type* field and discard the packet if the type is incorrect.
3. If the sequence number and the Ethernet source address are the same as that of the last request packet, discard this request and retransmit the last reply.
4. Copy the request *SrcAddress* into the reply *DestAddress* field.
5. Copy the request *DestAddress* into the reply *SrcAddress* field.
6. Copy the request *Header* into the reply *Header* field.
7. Write the reply *Version* field.
8. If the *ScanInModeA* field is equal to 0F, then skip the scan-in phase. Otherwise, assert the Titan I/O bus *scanMode* signal, write *ScanInModeA* into the *ModeA* register, scan-in 8**ScanInBytes*+*ScanInBits* bits, and deassert the *scanMode* signal.
9. Write the 4 bytes of the *Go* field into the clock single-step register.
10. If the *ScanOutModeA* field is equal to 0F, then skip the scan-out phase. Otherwise, assert the *scanMode* signal, write *ScanOutModeA* into the *ModeA* register, non-destrutively scan-out 8**ScanOutBytes*+*ScanOutBits* bits, and deassert the *scanMode* signal.
11. Write the *ModeB* register with *EndModeB*.
12. Write the *ModeA* register with *EndModeA*.
13. Write the value of the system halt signals into the reply packet *SystemStatus* field.
14. Transmit the reply packet.

Figure 6-2 shows the format of the *ModeA* register. The *select* field determines the Titan module to use for scan operations as listed in Figure 6-3. The *EndModeB* field must have the *Select* field of the register set to the Null module (F), or spurious scanning of other modules will result. The ~*StopCPU*, ~*StopMC* ~*StopNS*, and ~*StopIO* bits control the diagnostic stop signal to the CPU modules, memory controller, non-stop (RAM refresh) memory controller, and IO modules, respectively. The *~StopNS* bit should not be asserted (set to 0) for more than a few milliseconds, or main memory contents will be lost due to the absense of RAM refresh cycles.

```
|<  4   >|<   1    >|<   1   >|<   1   >|<   1   >|
+--------+----------+---------+---------+---------+
|        |          |         |         |         |
| Select | ~StopCPU | ~StopMC | ~StopNS | ~StopIO |
|        |          |         |         |         |
+--------+----------+---------+---------+---------+
```

Figure 6-2: CSM Mode A Register

Select	Module
0	Data Path
1	Inst Cache
2	Data Cache
3	Coprocessor
4	Memory Controller
5	Non-stop Memory Controller
6	I/O Slot 7
7	I/O Slot 6
8	I/O Slot 5
9	I/O Slot 4
A	I/O Slot 3
B	I/O Slot 2
C	I/O Slot 1
D	Reserved
E	Reserved
F	Null Module

Figure 6-3: CSM Mode A Register Select Encoding

The *ModeB* register, shown in Figure 6-4, controls scanning and the *Boot* signal. The *ScanMode* bit controls the state of the Titan I/O bus *scanMode* signal. The *Boot* bit controls the state of the *boot* signal sent to the CPU. The *ScanCount* field specifies the number of clock cycles to assert the *diagScan* signal to the selected module. The encoding of the *ScanCount* field is shown in Figure 6-5.

```
|<   1    >|< 1  >|<2>|<    4    >|
+----------+------+---+-----------+
|          |      |   |           |
| ScanMode | Boot | 0 | ScanCount |
|          |      |   |           |
+----------+------+---+-----------+
```

Figure 6-4: CSM Mode B Register

ScanCount	DiagScan Cycles
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8..F	0

Figure 6-5: CSM Mode B Register ScanCount Encoding

The *Go* register, shown in Figure 6-6, allows single- and burst-stepping of the *clock*. In reality, for every set bit of the *SingleStep* field of the register, the diagnostic stop signals are deasserted for one cycle, effectively single-stepping the clock. For example, the value 01 would single-step the clock, the value 0F would deassert the diagnostic stop signals for 4 contiguous cycles.

```
|<1>|<    7     >|
+---+------------+
|   |            |
| 0 | SingleStep |
|   |            |
+---+------------+
```

Figure 6-6: CSM Go Register

Figure 6-7 shows the format of the *SystemStatus* field of the reply packet.

```
|<   1    >|<     1    >|<6>|
+----------+------------+---+
|          |            |   |
| McHalted | ~CpuHalted | 0 |
|          |            |   |
+----------+------------+---+
```

Figure 6-7: CSM System Status Format

7. Titan Memory Adaptor (TMA)

The Titan Memory Adaptor (TMA) allows Titan main memory to be read and written via the Ethernet. All transactions are initiated from the Ethernet end. The Titan cannot interact with this board except to acknowledge and clear interrupts generated by the board.

7.1 Packet Format

TMA request packet format is shown in Figure 7-1. The fields are:

Dest Address	Ethernet destination address.
Src Address	Ethernet source address.
Type	Ethernet packet type.
Header	Higher-level protocol header, e.g., sequence number, etc.
Null	Unused padding bytes.
Command	TMA command.
DMA Address	DMA transaction base address; should be quadword aligned (least significant two bits zero).
Write Data	Variable-length write data. The length is truncated to a multiple of 4 bytes. There must always be at least 1 word of write data, **even** if it is a read command.

```
    +--------------+
  0.| Dest Address |
    +--------------+
  6.| Src Address  |
    +--------------+
 12.| Type         |
    +--------------+
 14.| Header       |
    +--------------+
 32.| Null         |
    +--------------+
 35.| Command      |
    +--------------+
 36.| DMA Address  |
    +--------------+
 40.|              |
    | Write Data   |
551.|              |
    +--------------+
```

Figure 7-1: TMA Request Packet Format

The *command* byte format is shown in Figure 7-2. If the *interrupt* bit is set, the TMA will assert its Titan I/O bus interrupt signal. If the *Write* bit is set, the **following** packet will provide up to 128 words of write data to be written at *DMA Address* plus 128.

```
|<6>|<    1    >|<  1  >|
+---+-----------+-------+
|   |           |       |
| 0 | Interrupt | Write |
|   |           |       |
+---+-----------+-------+
```

Figure 7-2: TMA Command Byte Format

The TMA always sends a reply packet with the 128 words of data read from memory starting at the *DMA Address*. The reply packet format is shown in Figure 7-3. The TMA exchanges the Ethernet source and destination address fields, and copies the type and header fields from the request packet.

```
     +--------------+
   0.| Src  Address |
     +--------------+
   6.| Dest Address |
     +--------------+
  12.| Type         |
     +--------------+
  14.| Header       |
     +--------------+
  32.|              |
     | Read Data    |
 543.|              |
     +--------------+
```

Figure 7-3: TMA Reply Packet Format

7.2 Byte Order

The board was designed to place low address bytes in the *most* significant part of a word. This is the byte ordering convention adopted by the Titan project. The VAX places low address bytes in the **least** significant part of a word. Hence, care must be exercised in formating the address and data fields of a packet if that program runs on a VAX (as opposed to a Titan).

For example, if the program communicating with the TMA is executing on a VAX, and the DMA addresses are manipulated as 32-bit integers in the program, they must be byte-swapped before insertion into a request packet.

7.3 I/O Registers

The TMA does not have any I/O registers. Consequently, performing I/O reads or I/O writes will always have the *ack* field of the IoStatus register deasserted. However, I/O operations do have the side-effect of deasserting the TMA interrupt signal.

7.4 I/O Adaptor Identifier

The TMA does not support an I/O adaptor identifier register.

7.5 Interrupts

If the *command* byte of a request packet has the *interrupt* bit set, the TMA asserts interrupt. Any I/O read or write of the TMA deasserts the interrupt.

8. Titan Disk Adaptor (TDA)

The Titan Disk Adaptor (TDA) implements a DEC Mass Storage Control Protocol (MSCP) server for Serial Disk Interface (SDI) storage devices. The TDA supports multiple, asynchronous MSCP commands and responses for one to four SDI devices.

8.1 I/O Registers

The interface between the device driver and the TDA consists of 256 32-bit words physically located on the TDA board. The layout of these registers is shown in Figure 8-1.

```
    31                                    0
     +-------------------------------------+
   0.|                                     |
     |                                     |
     \      MSCP Command Ring              \
     \                                     \
     |      8, 12-word segments            |
  95.|                                     |
     +-------------------------------------+
  96.|      Command Put Pointer            |
     +-------------------------------------+
  97.|      Response Get Pointer           |
     +-------------------------------------+
  98.|                                     |
     |     Unused                          |
 127.|                                     |
     +-------------------------------------+
 128.|                                     |
     |     MSCP Response Ring              |
     \                                     \
     \     12, 8-word segments             \
     |                                     |
 223.|                                     |
     +-------------------------------------+
 224.|     Command Get Pointer             |
     +-------------------------------------+
 225.|     Response Put Pointer            |
     +-------------------------------------+
 226.|                                     |
     |     Unused                          |
 248.|                                     |
     +-------------------------------------+
 249.|                                     |
     |                                     |
     |     DMA registers                   |
     |                                     |
 254.|                                     |
     +-------------------------------------+
 255.|     Adaptor Identifier              |
     +-------------------------------------+
```

Figure 8-1: TDA I/O Registers

The instruction set of the TDA is a subset of the Mass Storage Control Protocol.[8]

[8]Version 1.2 , 8 April 1982, Edward A Gardner et. al.

Instructions accepted are *Set Controller Characteristics*, *Online*, *Get Unit Status*, *Read*, *Write*, and *Replace* . These are placed by the device driver into the MSCP command ring. Responses to I/O commands (*Read*, *Write*) are 8 words long. Responses from non-I/O commands and error logs or attention packets take up two 8-word segments.

Refer to the MSCP manual for further description of these commands.

A *Put* pointer equal to the *Get* pointer indicates that a ring is empty. Bit 15 of the *Put* pointer being asserted indicates that the ring is full. The bit is set when a packet is placed in the ring which fills the ring, and is cleared by the receiver when it takes a packet from a full ring. When the bit is asserted, it locks out the party putting entries into the ring.

The values in the *Put* and *Get* pointers for both command and response rings are addresses of 16-bit words ([0..511]) due to the 16-bit address/data paths in the TDA.

8.2 I/O Adaptor Identifier

The value of the TDA I/O adaptor identifier register is 00000002.

8.3 Interrupts

The TDA generates an interrupt whenever the response ring is non-empty. Writing any register clears the interrupt.

9. Titan Network Adaptor (TNA)

The Titan Network Adaptor (TNA) provides an Ethernet interface and 8 asynchronous RS-232 serial lines.

The Ethernet interface uses the Seeq 8023/8003 chip set. The interface is full duplex: it can receive its own packets without any special handling. The receiver can handle 15 back-to-back packets without software intervention. The transceiver interface is compatible with Ethernet version 2 and IEEE 802.3.

The serial line interface uses the Dec DC349 Octart chip. The Octart provides 8 independent full-duplex asynchronous serial lines with programmable bit rate, character length, stop bits, and parity. It senses transitions on two input wires and an external register provides one output wire per line for modem control.

9.1 Further Reading

The TNA is essentially two DMA channels fronting the Seeq chip set, and a simple read/write interface to the Octart and Seeq chips. For details on the operation of these chips, consult the Seeq 8003/8023 and Dec DC349 data sheets. The Ethernet is specified in Dec Standard 134.

9.2 Ethernet

To operate the transmitter or the receiver, first set up the corresponding command register in the Seeq chip, then load a packet buffer descriptor (address and length) into the DMA registers, then set the enable bit and wait for an interrupt signalling completion.

The transceiver connects to the printed circuit board via pins in section 3 of the backplane. An internal cable plugs onto these pins and goes to a connector on the Titan’s rear bulkhead. Jumper R1, in the lower left corner of the board selects whether the transceiver interface is compatible with Ethernet version 1 (jumper installed) or IEEE 803.3 (jumper removed).

Three LEDs mounted on the edge of the PC board monitor Ethernet activity. One LED lights when carrier is present, another lights when the transmitter is active, and the third one lights when a collision is detected.

This Ethernet interface was designed for a system clock period of 40 ns. It will work correctly with a clock period as slow as 80 ns; slower clocks will cause DMA under- and overrun errors.

9.2.1 Ethernet Transmitter

To transmit a packet, first load the Seeq transmitter command register, then load the beginning address and length of the packet, then set the transmitter enable bit in the TNA control register. Transmission is complete when the transmitter interrupt bit is set in the TNA status register. To send another packet, clear transmit enable, reload the address and length registers, and then set transmit enable again; it is not be necessary to reload the Seeq transmitter command register. When the transmitter interrupt bit in the TNA status register sets, the TxStatus field contains the ending status for the transmission (ok, collision, underflow). Collision detection, binary exponential backoff, and retransmission are handled automatically (it can be manually overridden).

The transmitter packet buffer address is a word address and the low two bits are ignored. The first two bytes of the first quadword are skipped, aligning the data field of the packet in the second quadword. The transmitter packet buffer length is a 16-bit byte count, making the maximum packet length 64K bytes.

If the result of logically anding the contents of the Seeq transmitter command register with the contents of the Seeq transmitter status register is nonzero, then DMA transfers are stopped and the transmitter interrupt bit in the TNA status register is set. Normally the value 0xD is loaded into the Seeq transmitter command register. This causes the DMA machine to shutdown and generate an interrupt when the Seeq chip signals the end of a successful transmission, or 16 consecutive collisions, or a data underflow; garden-variety collisions cause the Seeq chip to backoff and the DMA machine to restart.

The transmitter may be stopped at any time by clearing the transmitter enable bit in the TNA control register. No further DMA reads will start and no ending interrupt will be generated. If the Seeq chips were transmitting at the time, a malformed (but harmless) Ethernet packet will be generated. The enable bit is cleared by a hardware reset.

9.2.2 Ethernet Receiver

To receive packets, first load the the Seeq Ethernet address and receiver command registers, then load up to 15 packet buffer descriptors (address and length pairs), then set the receiver enable bit in the TNA control register. Each time the interface finishes copying a packet from the Ether into memory, it stores the ending status in the first quadword of the buffer, sets the receiver interrupt bit in the TNA status register, and prepares to copy a subsequent packet into the next buffer.

The receiver packet buffer address is a word address and the low two bits are ignored. The first quadword is skipped initially, and eventually filled with four copies of the ending status. The first two bytes of the second quadword are skipped, aligning the packet data field in the third quadword. The format of the ending status word is shown in figure 9-1.

```
|<4>|< 4  >|<  8   >|<    12    >|<  4   >|
+---+------+--------+-----------+--------+
|   |      |        |           |        |
| 1 |  Rx  |   Rx   |  Ending   | Ending |
|   | Read | Status | QuadWords | Bytes  |
|   |      |        |           |        |
+---+------+--------+-----------+--------+
```

Figure 9-1: TNA Receiver Ending Status

The receiver packet buffer length is the maximum number of quadwords into which the interface may write. The register is 12 bits wide, making the maximum packet length 64K bytes. It is loaded from bits 15-4, skipping the low four bits, to align it with its ending status value. The ending status contains the number of unused quadwords and the number of used bytes in the last used quadword. The ending status written at the front of the packet buffer counts as one of the used quadwords. The length in bytes of a packet is

(beginning quadwords - ending quadwords)*16 - 32 + ending bytes.

This counts the Ethernet destination and source addresses, the type field, and the data field, but not the ending status. The Ethernet preamble and cyclic redundancy checksum are never seen by software. If the ending quadword count is all ones, then the packet overflowed the buffer. A packet of less than 60 bytes is forbidden by the Ethernet spec and flagged by the Seeq chip as a ’short frame’, but the data is correct.

If the result of logically anding the contents of the Seeq receiver command register with the contents of the Seeq receiver status register is nonzero, then DMA transfers are stopped, status is stored in the front of the buffer, and the receiver interrupt bit in the TNA status register is set. Normally the value 0xA0 is loaded into the Seeq receiver command register. This causes the DMA machine to shutdown and generate an interrupt when the Seeq chip signals completion of a good unicast or broadcast packet; the wrong address or any error causes the DMA machine to flush the packet and reuse the memory buffer. Setting the overflow error, crc error, dribble error, short frame, or end of frame bits in the Seeq receiver command register causes a packet with these attributes to consume a buffer and generate an interrupt, rather than being flushed.

Receiver packet buffer descriptors are stored in a ring buffer with 4-bit read and write pointers (called RxRead and RxWrite hereafter). The hardware and software cooperate to manage it as a fifo queue. When RxRead equals RxWrite, the fifo is empty, and when ((RxWrite+1) mod 16) equals RxRead the fifo is full. RxRead increments (mod 16) after the hardware finishes with a packet buffer and before it generates an interrupt signalling completion of a reception. RxWrite increments (mod 16) after the software writes into the receiver packet buffer length register, so load the address part of a descriptor first, then the length.

A hardware reset clears both pointers to zero. The fifo can be initialized to empty by clearing the receiver enable bit in the TNA control register and then repeatedly writing the receiver length register until RxRead equals RxWrite. RxRead and RxWrite appear as fields in the TNA status register. RxRead is also included in the receiver ending status.

To determine how many packet buffers (and which ones) contain completed packets when a receiver interrupt occurs, keep a list of the buffers in the order that they were loaded into the fifo, and remember the previous value of RxRead. The number of completed buffers is

(current RxRead - previous RxRead) mod 16,

and they are at the front of the list. Set previous RxRead to current RxRead before enabling the receiver and recopy it after each interrupt.

Another way to decide if a buffer is full is to zero the first word and wait for it to go non-zero when the ending status is written. The problem with this approach is the high cost of flushing the cache between each read reference while waiting for the ending status to go nonzero.

When the receiver enable bit in the TNA control register is set, each Ethernet packet that passes the address filter in the Seeq chips is copied into a memory buffer specified by the descriptor pairs stored in the fifo. If the descriptor fifo goes empty, the receiver flushes packets as they emerge from the Seeq chips. Software may add more descriptors at any time, up to a maximum of 15. If the receiver is enabled but flushing packets because the fifo is empty, simply adding some descriptors restarts things.

The receiver may be stopped at any time by clearing the receiver enable bit in the TNA control register. No further DMA writes will start and no interrupt will be generated. The enable bit is cleared by a hardware reset.

9.2.3 Ethernet Address Rom

Each TNA has a 32-byte read-only memory containing a guaranteed-unique Ethernet address. This quantity is usually copied into the Seeq receiver and used as the hardware address. Note well that this number will change if the TNA board is changed.

Rom bytes 0 to 5 contain the Ethernet address ready to be loaded into bytes 0 to 5 of the Seeq chip. Bytes 6 and 7 contain a checksum over the preceding six address bytes. Bytes 8 to 15 contain the checksum and address in reverse order. Bytes 16 to 23 are the same as bytes 0 to 7. Bytes 24 to 31 contain the test pattern FF, 00, 55, AA, FF, 00, 55, AA. This rom is Dec part number 23-000A1-09; the address checksum algorithm is specified in Appendix B of Dec Std 134.

9.3 Octart

The serial lines connect to the PC board via 50 pins in section 3 of the backplane. The pinout is compatible with a DZ11 ribbon cable and connector panel. An internal cable plugs onto these pins and goes to a small PC board containing eight telephone modular jacks in the Titan rear bulkhead. Each of the eight lines has six wires: transmit data, receive data, data set ready, data terminal ready, data carrier detect, and ground. Lines 0-3 are wired as data communication devices (DCEs) to which terminals such as VT102s can connect directly. Lines 4-7 are wired as data terminal devices (DTEs) to which VAXs can connect directly.

The Octart contains 64 8-bit registers which are read and written by software to control the serial lines and to move data. The TNA maps these registers into 64 words in its I/O address space. The most significant 24 bits of each word are zero when read and ignored when written. When the Octart asserts its interrupt line, the Octart interrupt bit in the TNA status register is set. The bit is cleared when the TNA status register is read and the Octart is not asserting its interrupt pin.

The Octart monitors two RS-232 control wires (data set ready and carrier detect) and can be programmed to generate interrupts when these lines transition. Software can write an 8-bit register in the TNA which drives one RS-232 control wire (data terminal ready) per line for modem control.

References to Octart registers should not occur closer together than 450 ns.

9.4 I/O Registers

The TNA decodes the least significant nine bits of an address presented to it during an IO read or write operation. Address bits 8-6 select a register or group of registers, and bits 5-0 are decoded by the selected group. The Octart decodes all six low order bits; the boardID ignores them all.

Bits 8-6	I/O Read	I/O Write
7	Board ID	-
6	Ethernet Adr	-
5	Status	Control
4	Octart	Octart
3	-	-
2	Seeq	Seeq
1	DMA Regs	DMA Regs
0	-	Data Term Rdy

Figure 9-2: TNA I/O Registers

9.4.1 Board ID

An IO read with address bits 8-6 equal to 7 returns the value 4, the registered TNA identifier value. IO Writes to this range of addresses are acknowledged but ignored.

9.4.2 Ethernet Address Rom

IO reads with address bits 8-6 equal to 6 select the Ethernet address rom; writes to this range of addresses are acknowledged but ignored. Address bits 4-0 are decoded by the rom to select one of 32 bytes. The rom contains a test pattern and 3 copies of a 48-bit Ethernet address protected by a checksum. The least significant bit of byte 0 of the rom is the multicast bit. Rom bytes 0-5 are usually loaded into Seeq address registers 0-5.

9.4.3 Status and Control

IO reads with address bits 8-6 equal to 5 reference the TNA status register. Writes to the same address reference the TNA control register. The control register is the control field of the status register; all other status register fields are read-only.

```
|<  8   >|<  8   >|< 4  >|<  4  >|< 4  >|<  4 >|
+--------+--------+------+-------+------+------+
|        |        |      |       |      |      |
|   Tx   |   Rx   |  Rx  |  Rx   | TNA  | TNA  |
| Status | Status | Read | Write | Ctrl | Ints |
|        |        |      |       |      |      |
+--------+--------+------+-------+------+------+
```

Figure 9-3: TNA Status Register

TxStatus and RxStatus are the Seeq transmitter and receiver status registers. Each time the Seeq chip asserts its interrupt pin, and whenever the CPU reads a Seeq status register, the internal Seeq status registers are copied into external registers on the TNA board. These are the values returned in the TNA status register and stored as ending status at the front of receiver buffers.

RxRead and RxWrite are the receiver packet buffer descriptor fifo read and write pointers. RxRead is incremented by the hardware after ending status has been stored in the buffer and before the interrupt bit is set. RxWrite is incremented after the software writes the receiver DMA length register. When RxRead equals RxWrite the descriptor fifo is empty, and when ((RxWrite+1) mod 16) equals RxRead the fifo is full.

9.4.4 Control Bits

The format of the TNA control bits is shown in figure 9-4. These bits are read/write and are cleared by a hardware reset.

```
  Bit 7   Bit 6  Bit 5  Bit 4
+-------+------+------+------+
|       |      |      |      |
| Byte  | Xcvr |  Tx  |  Rx  |
| Order | Enbl | Enbl | Enbl |
|       |      |      |      |
+-------+------+------+------+
```

Figure 9-4: TNA Status.ctrl field

ByteOrder controls the order in which bytes are referenced within a 32-bit word. If this bit is zero, the order is left-to-right. This bit affects both the Ethernet transmitter and receiver.

XcvrEnbl enables the Ethernet transceiver. If this bit is zero, the transmitter is looped to the receiver, collisions are impossible, and the only packets that will be received are those sent by the local transmitter. This bit must be set to communicate with other stations on the Ethernet.

TxEnbl enables the transmitter. If this bit is set, the transmitter will attempt to send the packet described by the transmitter DMA address and length, and then it will generate an interrupt and wait for TxEnbl to drop.

RxEnbl enables the receiver. If this bit is set and the descriptor fifo is not empty, then packets passing the address filter in the Seeq chip will be copied to memory, and receiver interrupts will be generated.

9.4.5 Interrupt Bits

The format of the TNA interrupt bits is shown in figure 9-5. There is one bit for each source of interrupts on the board. If any of these bits is set, the TNA asserts its interrupt line. These bits are cleared by reading them, and by a hardware reset. If the status register is read with address bit zero equal to one then the interrupt bits aren't cleared by the read.

```
  Bit 3 Bit 2   Bit 1    Bit 0
+-----+-----+--------+--------+
|     |     |        |        |
| Tx  | Rx  | Parity | Octart |
| Int | Int |  Int   |  Int   |
|     |     |        |        |
+-----+-----+--------+--------+
```

Figure 9-5: TNA Status.ints field

TxInt is set after a packet has been entirely transmitted or aborted by an unrecoverable error. It is good practice for software to start a 3-second watchdog timer each time the transmitter is enabled. If the timer expires, the transmitter is hung up and should be reset by clearing its enable bit.

RxInt is set each time the receiver finishes copying a packet into memory. Several packets may arrive before software clears the interrupt bit by reading the status register. Don't expect one distinct interrupt per packet, rather treat a receiver interrupt as a signal to inspect the receiver buffer queue looking for newly arrived packets, and to top off the descriptor fifo with more buffers.

ParityInt is set if the TNA detects bad parity on the IO bus during a cycle when it is reading the bus. This can happen during a DMA read or an IO read or write operation. The TNA can cause the memory controller to detect an IO bus parity error if it generates bad parity during a DMA read or write, or an IO read operation.

OctartInt is set whenever the Octart asserts its interrupt pin. After servicing an Octart interrupt and causing the Octart to deassert its interrupt pin, the TNA status register must be read one more time to clear the Octart interrupt bit. The other three interrupt bits may set between the first and second read of the TNA status register, and they must be handled by software after the second read or they will be lost.

9.4.6 Octart Chip

IO reads and writes with address bits 8-6 equal to 4 reference the Octart. Address bits 5-0 are decoded by the Octart, and reference one of its 64 byte-wide registers. See the Octart data sheet for details. Octart references should occur no closer together than 450 ns.

9.4.7 Seeq Ethernet Chips

IO reads and writes with address bits 8-6 equal to 2 reference the Seeq 8003. Address bits 2-0 are decoded by the Seeq chip and reference one of its 8 byte-wide registers.

The EtherAdr registers should be loaded with a physical Ethernet address before enabling the receiver. Normally bytes 0-5 of the Ethernet Address Rom are loaded into EtherAdr0-5. The least significant bit of byte 0 is the multicast bit.

The transmitter command register, figure 9-7, is an interrupt bit mask. If an error event occurs and its command register bit is set then a transmitter interrupt is generated and DMA transfers stop. If an error event occurs and its command register bit is clear, then no transmitter interrupt is generated and DMA transfers restart, retransmitting the

Bits(2-0	I/O Read	I/O Write
0	-	EtherAdr0
1	-	EtherAdr1
2	-	EtherAdr2
3	-	EtherAdr3
4	-	EtherAdr4
5	-	EtherAdr5
6	RxStatus	RxCommand
7	TxStatus	TxCommand

Figure 9-6: Seeq chip registers

packet. Error events are 16Coll, Coll, and Underflow. Normally the value 0xD is loaded into this register. Hardware reset clears this register.

The transmitter status register is updated at the conclusion of each transmission attempt. It has the same format as the command register.

```
  7   6   5   4     3      2      1      0
+---+---+---+---+------+------+------+-------+
|   |   |   |   |      |      |      |       |
|   |   |   |   | Good |  16  |      | Under |
| 0 | 0 | 0 | 0 |  Pkt | Coll | Coll |  flow |
|   |   |   |   |      |      |      |       |
+---+---+---+---+------+------+------+-------+
```

Figure 9-7: Seeq transmitter command and status registers

If the 'good pkt' bit of the Seeq transmitter command register is not set and no error occurs, then the transmission will succeed, but no interrupt will be generated. If the 'underflow' bit is not set then data underflows cause the transmission attempt to restart. This is not recommended since data starvation on a Titan is an indication of serious problems which retrying probably won't fix, but retrying probably will pollute the Ether with lots of aborted packet fragments. If the '16Coll' bit is not set then the transmitter retries forever at maximum backoff in the face of continuous collisions. This too is not recommended. Setting the 'coll' bit causes the transmitter to give up if it gets a collision, permitting manual control over the retransmission algorithm.

The most significant bits of the receiver command register (figure 9-8) set the address filter mode and its remaining bits are an interrupt mask. If an error event occurs and its command register bit is set, then a receiver interrupt is generated and DMA transfers stop. If an error event occurs and its command register bit is clear, then no receiver interrupt is generated and DMA transfers restart on arrival of the next packet, overwriting the bad one. Error events are ShortPkt, CRCerr, and Overflow. Normally the value 0xA0 is loaded into this register. Hardware reset clears this register.

The receiver status register is updated after receiving a good packet and when reception is aborted by an error. It has the same format as the receiver command register except that bits 7-6 are undefined.

```
   7 - 6     5      4      3       2     1      0
+--------+------+-----+-------+------+-----+------+
|        |      |     |       |      |     |      |
|  Addr  | Good | Any | Short | Drib | CRC | Over |
| Filter |  Pkt | Pkt |  Pkt  |  Err | Err | flow |
|        |      |     |       |      |     |      |
+--------+------+-----+-------+------+-----+------+
```

Figure 9-8: Seeq receiver command and status registers

Only packets passing the address filter are copied into memory. The normal mode is 2, receiving broadcast packets and packets whose destination address matches that loaded into EtherAdr0-5. Mode 3 also receives packets whose destination address has the multicast bit set. Further filtering of multicast packets must be done by software.

Bits 7-6	Filter
0	no packets
1	all packets
2	EtherAdr+broadcast
3	EtherAdr+multicast+broadcast

Figure 9-9: Seeq receiver address filter modes

A 'good packet' is one that is not short, did not overflow, and had a good cyclic redundancy checksum (CRC). 'any packet' means any kind of packet, good or bad, addressed to us or not, except packets aborted by an overflow. A packet with less than 60 bytes is too short according to the Ethernet spec. A 'dribble error' is said to have happened when carrier does not drop on a byte boundary. 'overflow' means that the DMA machine didn't take bytes from the Seeq chip fast enough.

9.4.8 DMA Registers

IO reads and writes with address bits 8-6 equal to 1 reference the DMA registers. Address bits 1-0 are decoded as shown in figure 9-10. TxAddress and RxAddress load from bits 31-2. TxLength loads from bits 15-0; RxLength loads from bits 15-4.

Bits 1-0	I/O Read	I/O Write
3	-	RxLength
2	RxAddress	RxAddress
1	-	TxLength
0	TxAddress	TxAddress

Figure 9-10: TNA DMA registers

When the transmitter starts, it copies TxAddress and TxLength into counters used by the DMA machinery. If a collision occurs, the counters are reloaded from the saved original values and the DMA transfer is restarted. Writing TxAddress or TxLength loads the saved original values. Reading TxAddress returns the current value of the DMA counter; this is only useful for diagnostic purposes. Reading TxLength returns undefined data.

When the receiver starts, it copies RxAddress and RxLength into counters which are used by the DMA machinery. If the packet is discarded for any reason (address mismatch, bad checksum, etc.), the counters are reloaded from the original values and the next arriving packet reuses the current buffer. Instead of simple registers as in the transmitter, RxAddress and RxLength are a fifo queue written by software and read by hardware. When a packet is successfully received, RxRead is incremented (mod 16), and the next buffer descriptor pair is loaded into the DMA registers. Writing RxAddress writes into the address part of the fifo slot pointed to by RxWrite. Writing RxLength writes into the length part of the fifo slot pointed to by RxWrite and then increments (mod 16) RxWrite. Reading RxAddress returns the current value of the DMA counter; this is only useful for diagnostic purposes. Reading RxLength returns undefined data.

9.4.9 Data Terminal Ready Register

An IO write with address bits 8-6 equal to 0 loads the DTR register. This is an 8-bit write-only register which drives 8 RS-232 wires, one per Octart line. Reading the DTR register returns undefined data. Software must maintain a shadow copy.

10. Titan Fiber Adaptor (TFA)

The Titan Fiber Adaptor (TFA) implements a 100 megabit per second, fiber optic, full-duplex, point-to-point communication link that transfers data in variable-length packets. It contains an independent receiver and transmitter that share a Titan I/O bus interface.

Packet buffers are specified via an *address* and *length* register. The *address* is the base address of the buffer, which must be line-aligned in memory (least two significant bits zero). For the transmitter, the *length* register is the number of lines (quadwords) in the packet; for the receiver, the *length* register is the maximum number of lines that can be written to the buffer. The receiver and transmitter each have one (1) set of packet descriptor registers.

Both the receiver and transmitter have 255-word FIFOs to mask temporary increases in DMA latency. The transmitter fills its FIFO before starting transmission on the fiber optic cable. The receiver performs a DMA write whenever its FIFO has a line available. The FIFOs can be written (pushed) and read (popped) via I/O register operations.

10.1 Packet Format

The basic transmission unit is a *packet* comprised of a sequence of 32-bit words as shown in Figure 10-1. The *preamble* is the binary value 10101010101010101010101010101011. The *CRC* is the 32-bit Ethernet polynomial, shown in Figure 10-2, over the data words. The preamble and CRC are added during transmission, and removed during reception, by the TFA. Packets are Manchester-encoded on the fiber optic cable.

```
+------------+
|  Preamble  |
+------------+
|            |
|    Data    |
|            |
+------------+
|  Data CRC  |
+------------+
```

Figure 10-1: TFA Packet Format

The only limitation upon the number of *data* words present in a packet is the 20-bit *length* register, which supports at most 16M bytes of data.

$$X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^{8} + X^{7} + X^{5} + X^{4} + X^{2} + X + 1$$

Figure 10-2: TFA CRC Polynomial

10.2 I/O Registers

The TFA registers are shown in Figure 10-3. For the transmitter (Tx) and the receiver (Rx), the *address* and *length* registers define the main memory packet buffer, while the *data* register provides access to the TFA's FIFO buffer.

All registers are 32 bits wide, except for the *TxLength* and *RxLength* registers which are 20 bits wide.

Address	I/O Read	I/O Write
XXXXXXX0	Tx Address	Tx Address
XXXXXXX1	Tx Length	Tx Length
XXXXXXX2	Tx Data	Tx Data
XXXXXXX3	Status	Control
XXXXXXX4	Rx Address	Rx Address
XXXXXXX5	Rx Length	Rx Length
XXXXXXX6	Rx Data	Rx Data
XXXXXXX7	I/O Identifier	-

Figure 10-3: TFA I/O Registers

10.3 I/O Adaptor Identifier

The value of the TFA I/O adaptor identifier register is 00000006.

10.4 Status Register

```
|<8>|<1 >|<1 >|<1 >|< 8 >|<1 >|<1 >|<1 >|< 8 >|< 1 >|<1>|
+---+----+----+----+-----+----+----+----+-----+-----+---+
|   |    |    |    |     |    |    |    |     |     |   |
| 0 |TxEn|TxPX|TxFE|TxCnt|RxEn|RxPX|RxFE|RxCnt|RxCRC|BPE|
|   |    |    |    |     |    |    |    |     |     |   |
+---+----+----+----+-----+----+----+----+-----+-----+---+
```

Figure 10-4: TFA Status Register

The format of the TFA status register is shown in Figure 10-4. The fields are:

TxEn — Transmitter enable. This bit indicates that the transmitter is sending the packet defined by the *TxAddress* and *TxLength* registers. When the packet has been sent, the TFA clears the *TxEn* bit. The *TxEn* bit is also cleared when a *TxReset* command is issued.

TxPX — Transmitter packet transferred. When the transmitter has successfully completed transmission of a packet, it sets the *TxPX* bit. The *TxPX* bit is cleared by a *TxReset* command.

TxFE — Transmitter FIFO error. If during packet transmission, the transmitter FIFO is empty when the *TxLength* register is non-zero, the *TxFE* bit is set; an *underflow* has occurred. The *TxFE* bit is cleared by a *TxReset* command.

TxCnt — Transmitter FIFO count. *TxCnt* is the number of words currently in the transmitter FIFO. Issuing a *TxReset* command flushes the transmitter FIFO, which zeros *TxCnt*.

RxEn — Receiver enable. This bit indicates that the receiver will(is) copy(copying) the next incomming packet to the buffer defined by the *RxAddress* and *RxLength* registers. If there is an incomming packet at the time that a *RxStart* command is issued, it is ignored. After a packet has been received (successfully or not), the *RxEn* bit is cleared. The *RxEn* bit may also be cleared by issuing a *RxReset* command.

RxPX — Receiver packet transferred. When the receiver has successfully received a packet, it sets the *RxPX* bit. The *RxPX* bit is cleared when a *RxReset* command is issued.

RxFE — Receiver FIFO error. If during packet reception, the receiver FIFO is full when a word should be written to it, the *RxFE* bit is set; an *overflow* has occurred. The *RxFE* bit is cleared by a *RxReset* command.

RxCnt — Receiver FIFO count. *RxCnt* is the number of words currently in the transmitter FIFO. Issuing a *RxReset* command flushes the receiver FIFO, which zeros *RxCnt*.

RxCRC — Receiver CRC error. If a packet is received, but the transmitted CRC does not match the locally computed CRC, the *RxCRC* bit is set. The *RxCRC* bit is cleared by a *RxReset* command.

BPE — Bus parity error. If during a CPU I/O register access of the TFA, or a TFA originated DMA transaction, the TFA detects a bus parity error, then the *BPE* bit is set. The *BPE* bit is cleared whenever the TFA control register is written.

10.5 Control

```
|<27>|< 1  >|<   1   >|<   1   >|<   1   >|<   1   >|
+----+------+---------+---------+---------+---------+
|    |      |         |         |         |         |
| 0  | Loop | RxStart | RxReset | TxStart | TxReset |
|    |      |         |         |         |         |
+----+------+---------+---------+---------+---------+
```

Figure 10-5: TFA Control Register

The TFA control register format is shown in Figure 10-5. The function of the field is:

Loop — Receiver loopback. The receiver takes is serial input from the transmitter output rather than the fiber optic input (rx) cable whenever the *Loop* bit is set. The transmitter output is still present on the fiber optic output (tx) cable.

RxStart — Receiver start. Setting the *RxStart* bit enables the receiver to copy the next incomming packet into the buffer defined by *RxAddress* and *RxLength*. Clearing the *RxStart* bit has no effect.

RxReset — Receiver reset. Setting the *RxReset* bit clears the *RxEn*, *RxPX*, *RxFE*, and *RxCRC* bits in the status register, and flushes the receiver FIFO. If a reception is in progess, it is aborted. Clearing the *RxReset* bit has no effect.

TxStart — Transmitter start. Setting the *TxStart* bit starts transmission of the packet defined by the *TxAddress* and *TxLength* registers. Clearing the *TxEnable* bit has no effect.

TxReset — Transmitter reset. Setting the *TxReset* bit clears the *TxEn*, *TxPX*, and *TxFE* bits in the status register, and flushes the transmitter FIFO. If a transmission is in progress, it is aborted.

Clearing the *TxReset* bit has no effect.

10.6 Interrupts

The TFA generates an interrupt whenever any of the *TxPX*, *TxFE*, *RxPX*, *RxFE*, or *BPE* bits of the TFA status register are asserted.

10.7 Transmitting A Packet

To transmit a packet, the follow actions should be performed:

1. Issue a *TxReset* command via the *control* register to initialize the transmitter.
2. Write the *TxAddress* and *TxLength* registers.
3. Write any header words into the transmitter FIFO via the *TxData* register.
4. Issue a *TxStart* command via the *control* register.
5. Wait (via interrupt or busy-wait) for the *TxEn* bit to clear in the *status* register.
6. If the *TxPX* bit is set in the *status* register, the packet was successfully transmitter. Otherwise, the *TxFE* bit should be set, indicating that a DMA underrun occurred.
7. Issue a *TxReset* command via the *control* register to clear the *TxPX* or *TxFE* bit, and deassert the transmitter interrupt.

10.8 Receiving A Packet

To receive a packet, the follow actions should be performed:

1. Issue a *RxReset* command via the *control* register to initialize the receiver.
2. Write the *RxAddress* and *RxLength* registers.
3. Issue a *RxStart* command via the *control* register.
4. Wait (via interrupt or busy-wait) for the *RxEn* bit to clear in the *status* register.
5. If the *RxPX* bit is set in the *status* register, a packet was successfully received. Otherwise, the *RxFE* bit should be set, indicating that a DMA overrun occurred.
6. If the *RxPX* bit is set, and the *RxCRC* bit is set, the received packet had an incorrect CRC.
7. If the *RxPX* bit is set, and the *RxCRC* bit is clear, check if the packet was not an integral number of quadwords, indicated by a non-zero *RxCnt* field of the *status* register. If so, the trailer words must be manually read from the receiver FIFO via the *RxData* register and appended to the packet buffer starting at the address indicated by the *RxAddress* register.
8. Determine the length of the received packet by subtracting the original packet base address from the current value of the *RxAddress* register, plus the value of the *RxCnt* field.
9. Issue a *RxReset* command via the *control* register to clear the *RxPX* or *RxFE* bit, and deassert the receiver interrupt.

I. ALU Sign Corrector

For signed comparisons, the sign corrector works as follows. It examines the signs of the two operands, A and B. Then

```
(1)     sign(A) =  sign(B) => true sign := sign(A - B)

(2)     sign(A) <> sign(B) => true sign := carry(A - B)
```

Case (1) follows from the fact that if the signs of the two operands are equal, then their difference has a two's complement representation. For case (2), observe that A - B = A + (-B) so the operation performed is

```
 vs   s                          vs   s
  1 | 1aaaaaa...aa    or      0 | 0aaaaaa...aa
+ 1 | 1yyyyyy...yy          + 0 | 0yyyyyy...yy
```

where 's' denotes the sign bit, and 'vs' is the 'virtual' sign bit carrying the true sign, regardless of overflow (I use 'y' to represent the two's complement of 'b'). Because 1 + 1 = 0, the true sign bit is the carry (out of the sign position).

Unsigned comparisons are simpler, since both operands are by definition positive. Therefore there is only one case to consider,

```
 vs
  0 | aaaaaaa...aa
+ 1 | yyyyyyy...yy
```

and, since 1 + x = x', the true sign is the complement of the carry (out of the sign position).

Since it is possible for the overflow correction to leave an erroneous zero result, the sign corrector recomputes bit 0 (the least significant bit) as well, according to the rule

```
lsb := lsb(A - B) or sign(A - B)
```

This is a problem only when A - B = 2**31, which one cannot represent.

Now suppose that no carry is forced into the ALU. Then the operation it performs is A - B - 1 instead of A - B. This by itself is no problem, but the sign corrector decides what to do by looking at the signs of A and B! We have four cases to consider, based on the signs of A and B. Two of them are easy to dispatch, when A, B >= 0 and when A, B < 0; in these situations overflow is not possible, and the sign corrector will compute true sign := sign(A - B - 1).

When A >= 0 and B < 0, we must be more careful, since the sign corrector will use the carry as the true sign. Since we may write

```
A - B - 1 = A + B'
```

the computation would take the form

```
 vs   s
  0 | 0aaaaaa...aa
+ 0 | 0xxxxxx...xx
```

where the 'x's represent the one's complement of the 'b's. With both sign bits being zero, the carry is always 0, and we get the proper sign. Likewise, when A < 0 and B >= 0 we have

```
 vs   s
  1 | 1aaaaaa...aa
+ 1 | 1xxxxxx...xx
```

in which a carry is guaranteed, matching the ’true sign’.

We should also consider what happens if no carry is supplied when doing unsigned comparisons. In this mode, the ’true sign’ of the result is assumed to be the complement of the carry. The only interesting case is when the minuend is the largest number (it is all ones), since by not forcing a carry in we are computing A - B - 1 = A - (B + 1), and B + 1 is not representable. Its negation is represented correctly with the implicit sign bit (always one), however, so that we once again get the right answer.

Table of Contents

List of Figures