Butterfly II — VMEbus Interface (B2VME) Functional Specification

11

THIS DOCUMENT IS BBN ACI PROPRIETARY.

11.1 Introduction

This chapter is a functional specification of the Butterfly II VMEbus Interface (B2VME). This chapter assumes some familiarity with the VMEbus specification, the T-bus specification, the Butterfly II architecture, and the Motorola 88000 documentation.

The B2VME is the first in a family of function cards that comprise the Butterfly II. The primary function of the B2VME is to provide an I/O path between the Butterfly II and peripherals on a VMEbus system. In addition, it appears as a "normal" processor node on the Butterfly II switch.

Figure 11-1 is a block diagram of the B2VME. The B2VME consists of five modules: the CPU, the memory, the switch interface, the Test and Control System (TCS) interface and the VMEbus interface. These modules communicate with each other across the T-bus. The T-bus is a synchronous, multiplexed address/data bus that supports a variety of transactions. The T-bus transaction protocols are designed to avoid deadlock situations while obtaining good performance in communication among devices on the bus and throughout the system. The operation of the T-bus is documented in the *T-bus Specification*. Dedicated paths between the CPU and memory, and between the TCS and Switch Interface Gate Array (SIGA), have been added to enhance performance and simplify the design. This chapter describes the CPU, the memory system, the TCS module, and the configuration and control registers of the B2VME. The interface between the B2VME and the Butterfly II switch is implemented by the Switch Interface Gate Array (SIGA) that is described in detail in the switch chapter.

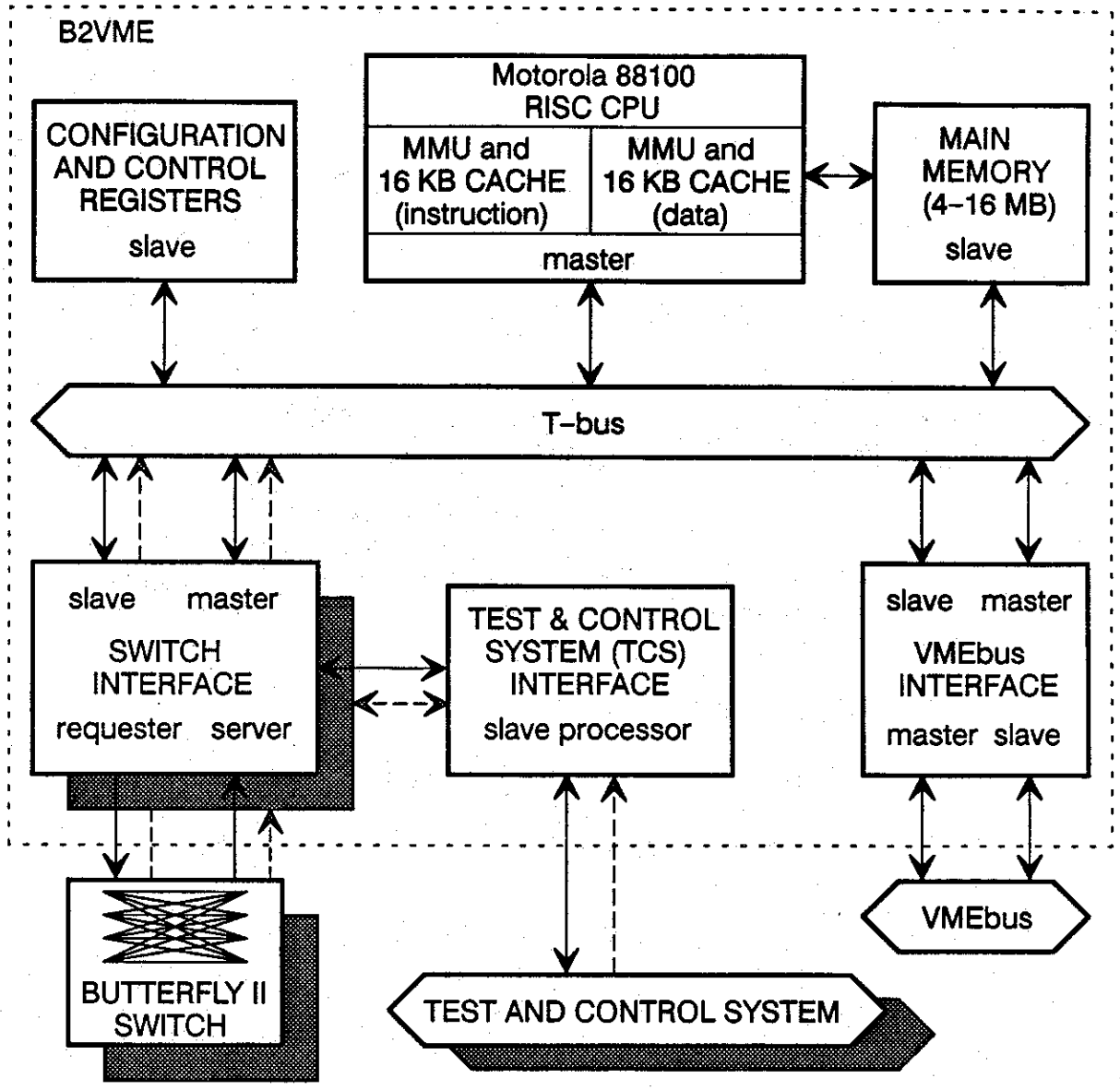


Figure 11–1. B2VME block diagram.

11.2 CPU

The CPU includes a a Motorola 88100 microprocessor, two Motorola 88200 Cache/MMU chips, and an interface to the T-bus.

The CPU interface includes a T-bus master finite state machine that generates single cycle read and write requests and cache burst read and write requests on the T-bus, in addition to the fast path to memory.

Because the operation most frequently performed by the CPU is fetching instructions from local memory, the CPU also has a specialized **fast path** that bypasses the T-bus and allows the CPU to communicate directly with the memory to optimize performance. This path has circuitry to detect when the CPU will be given ownership of the T-bus in advance of the T-bus arbitration circuitry, and a dedicated address path. The response to a memory fetch is returned over the T-bus, whether the access used the fast path or not.

The CPU can execute code stored on a remote node. Fetching each instruction over the switch, however, slows execution appreciably, so code is almost always stored locally. With the instruction cache turned on, the remote access cost may be amortized over many executions of the code, such as iterations of a loop. In this case, or for very short sections of code, execution across the switch may be appropriate.

CPU Memory Access Timing

TO BE DISCUSSED:
1. cache hit
2. cache miss ⇒ fast path (burst / single word — applies to below too?)
3. cache miss if fast path disabled ⇒ T-bus
4. cache miss ⇒ switch

[The following is the most recent (June 27, 1988) calculations from Robert Wells and Tom Downey. THESE ARE STILL UNDER DISCUSSION AND EVALUATION.]

Access Type	Clock Ticks (20MHz)	Time (usec)
cache hit	3	0.15
local, non-cacheable	11	0.55
local, cache miss	17	0.85
remote, non-cacheable	28	1.4
remote, cache miss	34	1.7

To compute peak bandwidths, divide the number of bytes transferred by the time to transfer. For cacheable data, assume that the data unit pipeline allowed the second, third, and fourth words of a 16-byte load to be handled in one clock tick each. Given this, the following bandwidths result:

Access Type	Peak Bandwidth (MB/sec)	Formula
cache hit	80	4/.05 (4 bytes per tick)
local, non-cacheable	7.3	4/.55
local, cache miss	16	16/(.85+3*.05)
remote, non-cacheable	2.9	4/1.4
remote, cache miss	8.6	16/(1.7+3*.05)

Turning the XMEM Instruction Into T-bus Locks

The 88000 instruction set includes the exchange memory (XMEM) instruction to support atomic operations. Refer to the *88000 User's Manual* for a description of this instruction. When the CPU executes the atomic instruction XMEM, the DLOCK output pin is asserted. The B2VME hardware maintains the atomicity presumed by the CPU by opening and maintaining a T-bus lock when DLOCK is asserted. If the reference is to a bypassed page, the lock is not generated. The T-bus FREE_LOCKS protocol is executed when the DLOCK pin goes from asserted to de-asserted.

NOTE

RESTRICTION OF USE
One restriction applies to the XMEM instructions: atomic instructions cannot be included in locked augmented sequences unless they access the memory module that is locked, or access bypassed memory. If this restriction is violated, the instruction will terminate in a bus error. Note that the XMEM "atomic" instruction within augmented sequences that are bypassed is not guaranteed to be atomic, since no lock is generated. The lock augmentation is controlled by the Augmentation register, described in detail later.

Interrupts to the CPU

The B2VME CPU receives interrupts from three kinds of sources: other processors, VMEbus devices, and on-board timers. These are listed below. The 88100 microprocessor has only one interrupt level, so all interrupts are OR'ed together to generate the interrupt signal to the processor chip. When an interrupt is detected, the interrupt handler reads the Interrupt Source register to find out which type of interrupt has occurred. The Interrupt Source register has a field for each interrupt type, and is described in detail in the register summary section.

- Interrupts from processors (typically other processors, but the CPU may generate an interrupt to itself if desired)
  - Maskable interprocessor interrupt
  - Non-maskable interprocessor interrupt
- Interrupts from VMEbus devices — Seven levels are distinguished.
- Interrupts from on-board timers
  - Interrupts Disabled Timer
  - Real Time Clock (RTC) timer, called the Time Of Next Interrupt (TONI) register — Each of two SIGA chips on the B2VME has two

TONI registers, for a total of four programmable timers driven by real time; normally, only one SIGA or the other is in use.

A Butterfly II processor can generate a remote interrupt request by setting a bit in the Interprocessor Interrupt register at the destination node. The Interprocessor Interrupt register for each processor node resides in page 1 in bank 0 of its physical address space, and is therefore accessible to every processor in the machine. The operating system controls access to the interprocessor interrupt by giving this page appropriate protection attributes.

The destination processor further controls remote interrupts with the Interrupt Enable Mask. The Interrupt Enable Mask is applied to the Interprocessor Interrupt register input. If the Interrupt Enable Mask bit for the interprocessor interrupt is zero, then the interrupt request will not be generated. Attempts to set the Interprocessor Interrupt register when it is masked result in a bus error.

A register for generating a non-maskable interrupt resides in a supervisor page of configuration and control registers. The non-maskable interrupt should be used only when fatal errors have occurred. The non-maskable interrupt will interrupts the CPU unless the CPU has disabled its entire interrupt system by manipulating its internal processor status register.

In addition to interprocessor interrupt requests, VMEbus interrupt requests can also interrupt the B2VME processor. VMEbus interrupts to the processor are subject to the Interrupt Enable Mask. If the mask bit for a VMEbus level is zero, then the interrupt to the processor will not be generated, even if the VMEbus interrupt request signal at that level is asserted. When a VMEbus interrupt occurs, the level of the highest priority pending interrupt is indicated by a field in the Interrupt Source register. The B2VME can also generate interrupts onto the VMEbus, as described later.

Finally, interrupts are generated to the B2VME processor, and all Butterfly II processors, by the expiration of the Interrupts Disabled Timer and by the real time clock. These, and other timers, are discussed below.

11.3 Memory Subsystem

["THIS SECTION NEEDS MORE WORDS." IT WILL PROBABLY BE BEEFED UP AS WORK ON THE MEMORY CHAPTER PROGRESSES.]

The B2VME memory subsystem features four megabytes of dynamic RAM arranged as 1M × 38-bit words. Each word has 32 data bits, one tag bit, and five parity bits, as described below. The memory array is implemented using 80-nanosecond, 1M × 1-bit DRAMs. The memory array is expandable to 16 megabytes when 4-megabit DRAMs become available. Read or write cycles to

the memory array complete in 200 nanoseconds. Writes are acknowledged immediately. Memory accesses are never split on the T-bus. Accesses are refused when the memory is refreshing, or if a write has not completed. The memory subsystem supports the T-bus locking protocol.

The memory subsystem supports byte, halfword, and word reads and writes. 33-bit burst reads and writes are also supported. New data is supplied every 100 nanoseconds during burst reads. The T-bus "PAUSE, MORE" protocol is used to throttle data requests during bursts.

One tag bit is associated with each word in the array to support the steal/sneak protocol discussed above. The value of the tag bit is returned on T_AD<32> of the T-bus with every read access. The tag bit is set or cleared with write accesses, subject to the value of T_AD<32>.

Byte-wide parity generation and checking is performed on every memory access. The tag bit has its own parity bit. Parity errors are reported on the M_PARITY signal.

FOR GLOSSARY: T_LATE_ERROR — an old name, still found on some schematics, for the M_PARITY signal, from a time when it used to be a T-bus signal. M_PARITY is now independent of the T-bus, and indicates a memory parity error.

The memory subsystem performs a "hidden" refresh cycle once every 12.8 microseconds. Hidden refresh utilizes internal counters to generate the refresh address, and thus requires minimal external support circuitry. The refresh cycle is included in the memory control finite state machine. A refresh cycle requires five T-bus clock cycles (250 nanoseconds). During a refresh cycle, all T-bus requests to memory are responded to with REFUSED. Refresh requests have the highest priority of any request to the memory.

11.3.1 Loading Interleaver and VMEbus Address Mapper

The interleaver and the VMEbus address mapper perform similar functions for the Butterfly II switch and the VMEbus, respectively. Both of these RAMs look at the system physical address on the T-bus, and change some subset of the address bits into different bits that are then shipped out over their respective I/O ports (the switch and the VMEbus). The address translation must happen in an expedient fashion in both cases, because any time used to calculate the new address can directly impact the transfer speed. Static memories efficiently solve this problem, and are employed in the B2VME. An address is presented on the address lines and the access time later, the new address bits appear on the data lines. The new address bits are simply the contents of the memory.

The mechanism to load the interleaver is described... [NEEDED - HERE OR MEMORY CHAPTER???]

11.4 TCS Slave Interface and Debugging

11.4.1 TCS Slave Interface

Like every other Butterfly II board, the B2VME includes a Test and Control System slave processor to support bootstrapping, power-on servicing, and monitoring of various environmental conditions.

The B2VME TCS slave has access to every addressable location in the B2VME via the T-bus. Thus, it can access memory, VMEbus devices, and configuration registers. It can also make references across the switch. To reduce hardware cost, the TCS slave shares the T-bus interface used by the SIGA. When the TCS master requests a data transfer into or out of the B2VME, the TCS slave uses a dedicated four-wire communication channel to the SIGA to fulfill the request. The four wires are data in, data out, clock and frame. The protocol for this communication channel is defined in the Butterfly II Test and Control System chapter.

The B2VME TCS slave processor has several functions. It monitors the ambient temperature in one location and the three supply voltage levels (+48, +5 and -5 volts) on the board. It supplies this information to the TCS master when requested. It controls the on-board power supply. In addition, the TCS master can reset the B2VME. Finally, the B2VME TCS slave can inform the TCS master of the B2VME's board type.

The B2VME TCS slave interface is implemented using a Motorola 68HC11. This device has 2 kilobytes of on-chip EEPROM, general purpose parallel I/O and serial I/O, A/D channels, a priority interrupt structure, a number of timers, and a 6800-type instruction set.

The TCS slave interface is powered by a supply located centrally in the Butterfly II chassis. The power is distributed to the B2VME through the midplane connector.

11.4.2 Power-up and Resetting

The B2VME is reset by application of power and by TCS action, as described below. There is no reset button on the board. There is no way to explicitly force a reset across the switch. No VMEbus device can explicitly reset the B2VME, and when the B2VME resets the VMEbus that does not reset the B2VME board.

A power-on reset circuit in the TCS interface leaves the B2VME in a reset state after power-on. Before the B2VME can run, the TCS master must load the appropriate bootstrap code, set several configuration registers, then clear the power-on reset.

The TCS clears the power–on reset in two steps. First, the TCS clears the reset for all B2VME components except the CPU. With the CPU still held reset, the TCS can perform appropriate start–up tests, configure the hardware by loading control and configuration registers, and load bootstrap code into memory. The second step is to clear the reset for the CPU, so it starts executing. Processing after a reset is one kind of exception processing, and is described in detail in the Motorola *88100 User's Manual.*

11.4.3 Debug Connector

A 96–pin DIN connector carries all T–bus signals, the clk20 signal, debug_driven and debug_slavepause, for diagnostic purposes. During initial debugging of the B2VME, a small board is plugged into this connector to provide the circuitry necessary to run an EPROM–based debugger. The board includes a UART that supports two serial lines and its interface circuitry, a 32–kilobyte EPROM and a T–bus slave interface.

When the debugging daughter board is plugged into the B2VME, the daughter board disables the low 64 kilobytes of normal memory and replaces it with the debugging board's address space, listed below. The EPROM is read–only, and responds to any transfer size. The DUART is accessed only by byte transfers. See debugging board documentation for the read/write capability and function of DUART locations.

0x00008000 to 0x0000FFFF EPROM (all 32 kilobytes)

0x00004000 to 0x0000403C DUART

0x00008000 to 0x0000FFFF EPROM (all 32 kilobytes)

11.5 VMEbus Interface

This section briefly describes the VMEbus, and then discusses the operation of each VMEbus feature implemented by the B2VME. A detailed description of the VMEbus can be found in *The VMEbus Specification*, a Motorola publication available from Motorola [or from BBN ACI???]. The B2VME is designed to revision C.1 of the specification.

The VMEbus interface on the B2VME is a general purpose, bus coupling mechanism that provides a path between the T–bus and a VMEbus. Through the T–bus connection, VMEbus devices can access local or remote memory, perform locked operations, and interrupt any processor on the machine. Similarly, any Butterfly II processor can access VMEbus memory, perform indivisible operations on VMEbus memory, and interrupt VMEbus devices.

11.5.1 Brief Description of the VMEbus

The VMEbus Specification defines an interface, called the VMEbus, used to connect data processing, data storage and peripheral control devices. The VMEbus is an asynchronous bus defined in broad terms that allow a wide range in performance while still conforming to the specification. This results in some difficulty in characterizing maximum cycle times on the VMEbus, because cycles can be extremely long without violating the specification.

The VMEbus Specification defines a functional module as "a collection of electronic circuitry that works together to accomplish a task". The types of functional modules defined in *The VMEbus Specification* include:

- VMEbus master
- VMEbus slave
- Bus timer
- Interrupt generator
- Interrupt handler
- VMEbus requester
- VMEbus arbiter
- IACK daisy chain driver
- System clock driver
- Power monitor

A VMEbus device implements a subset of this list to achieve the functionality required of the device. For example, a CPU board might consist of a VMEbus master, an interrupter, an interrupt handler, and a requester. A disk controller might include a VMEbus master, slave, interrupter, interrupt handler, etc. A memory board might consist of only a VMEbus slave.

A device called a *system controller* is required in each VMEbus system. It physically resides in slot one of a VMEbus, and performs the functions of arbiter, system clock generator, bus timer, and IACK daisy chain driver. The system controller function is sometimes performed by a device that expects to be the only VMEbus master in a system — for example, a CPU board.

11.5.2 B2VME Functionality

Figure 11-2 is a block diagram of the B2VME VMEbus interface.

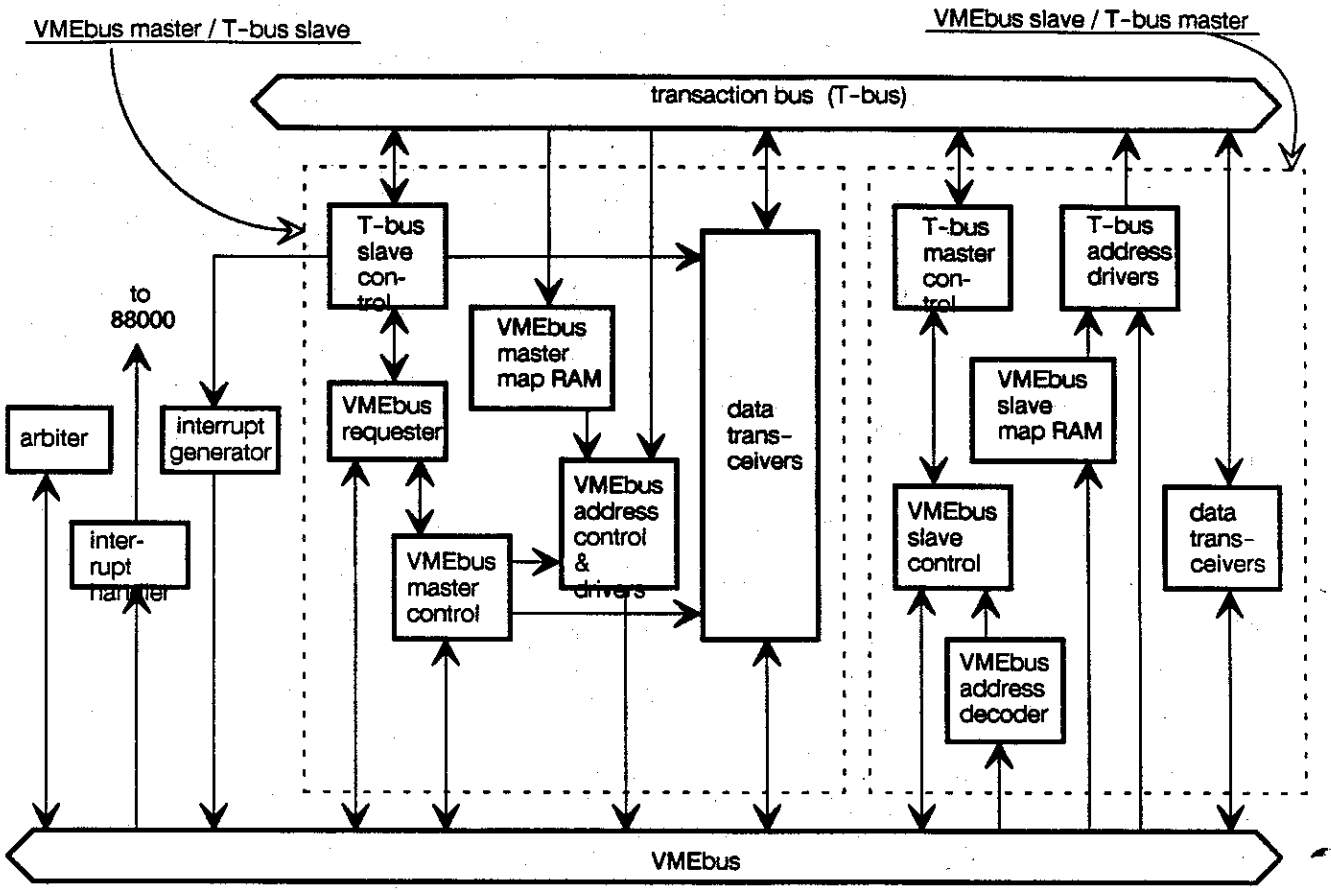


Figure 11-2. VMEbus interface block diagram.

The B2VME can be used in a several different configurations. In some environments, the B2VME may be the only VMEbus master in the system. When this is the case, the B2VME may also be required to perform the VMEbus system controller task. In other environments, the B2VME will provide a path that a VMEbus master will use as a DMA (direct memory access) target, from the VMEbus into Butterfly II memory. In this case, the B2VME will function primarily as a VMEbus slave. Because the B2VME must perform a variety of tasks on the VMEbus, it implements all but one (power monitoring) of the functional modules defined in *The VMEbus Specification*. The system controller functions (arbiter, system clock driver, bus timer, and IACK daisy chain driver) can be enabled or disabled by setting a jumper on the B2VME. Note that when the B2VME is functioning as a VMEbus master, it is also functioning as a T-bus slave. Similarly, if the B2VME is functioning as a VMEbus slave, it is also a T-bus master.

VMEbus Master

The B2VME VMEbus master interface accepts requests from the T-bus, and converts them into requests on the VMEbus. The VMEbus master interface splits cycles on the T-bus, letting other T-bus masters use the T-bus while the VMEbus transaction is completing. This is necessary because the access time on the VMEbus can be arbitrarily long, and therefore an access to the VMEbus is not guaranteed to complete in any reasonable amount of time from the perspective of the Butterfly II design.

The T-bus locking protocol is observed by the VMEbus master interface. A lock request on the T-bus results in disabling the VMEbus arbitration, and holding the VMEbus until the lock is released. This prevents any other VMEbus master from accessing any VMEbus resource, thus preserving the atomicity of the T-bus lock. Note that if the VMEbus resource is dual-ported, then atomicity is no longer guaranteed. An example of a dual-ported VMEbus resource is the Motorola memory board that connects to both a VMEbus and a faster auxiliary bus called VMX.

The VMEbus address space is accessible to the Butterfly II through the VMEbus master address mapping RAM (VMEbus master mapper). The VMEbus master mapper divides four megabytes of Butterfly II address sapce and allocates it into 8-kilobyte blocks of VMEbus address space. The master mapper takes nine address bits (21..13) of the address offset field and maps them into the six VMEbus address modifier bits, and bits 31..13 of the VMEbus address. Refer to *The VMEbus Specification* for more information about the address modifiers. The low order 13 address bits pass directly through to the VMEbus. The VMEbus master mapper is loaded by the interleaver loader described above. Figure 11-3 illustrates the VMEbus master mapper.

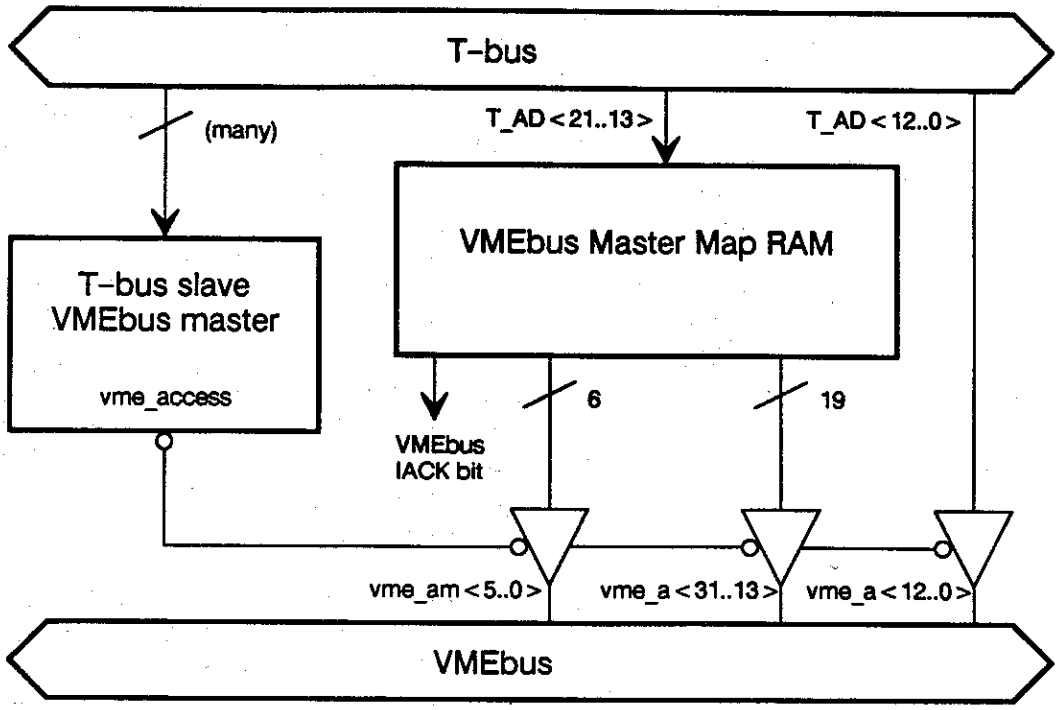


Figure 11-3. VMEbus master mapper.

VMEbus Slave

The VMEbus slave interface responds to single-word read and write requests on the VMEbus. These requests translate into memory references across the T-bus. The T-bus master can handle split cycle and non-split cycle responses from the target address. These may result in extremely long VMEbus cycles for access to memory that does not reside on the local node. The VMEbus slave interface on the B2VME does not respond to sequential transfer requests on the VMEbus (also called "block transfers" in *The VMEbus Specification*).

A master on the VMEbus addresses Butterfly II memory through the Butterfly II address mapping RAM (VMEbus slave mapper). This mapper takes the several-megabyte block of VMEbus address space and allocates it into eight-kilobyte blocks of Butterfly II address space. When configured to respond to standard address format on the VMEbus, the window in VMEbus address space is two megabytes; configured for extended VMEbus addressing, the window is eight megabytes. The VMEbus slave mapper takes bits 21..13 of VMEbus address and translates them into the high 21 bits of the Butterfly II system physical address. In addition, the VMEbus slave mapper sources the T-bus signals T_PATH<1..0> and T_PRIORITY<1..0>, and the SIGA input signal INTERLEAVED. The VMEbus slave mapper allows the process to utilize the Butterfly II addressing scheme to the extent required to obtain the

type of performance required. For example, if the fastest transfer is required and T-bus locks can be ignored, then all accesses can be mapped *bypassed* to the local node. If the VMEbus device transferring data to the Butterfly II wants to spread it out over the machine, this can be accomplished automatically by mapping into *interleaved* memory. These accesses would be subject to the maximum switch lock time, and therefore may be very slow.

The occurrence of read-modify-write cycles on the VMEbus is difficult to predict because there is no indication that a VMEbus read-modify-write cycle is taking place until after the read has completed. Each map location can be configured as read-modify-writable memory in the VMEbus slave mapper. Any accesses to this memory opens a T-bus lock. The lock is closed when the trailing edge of VMEbus address strobe is detected.

Figure 11-4 illustrates the VMEbus slave mapper.

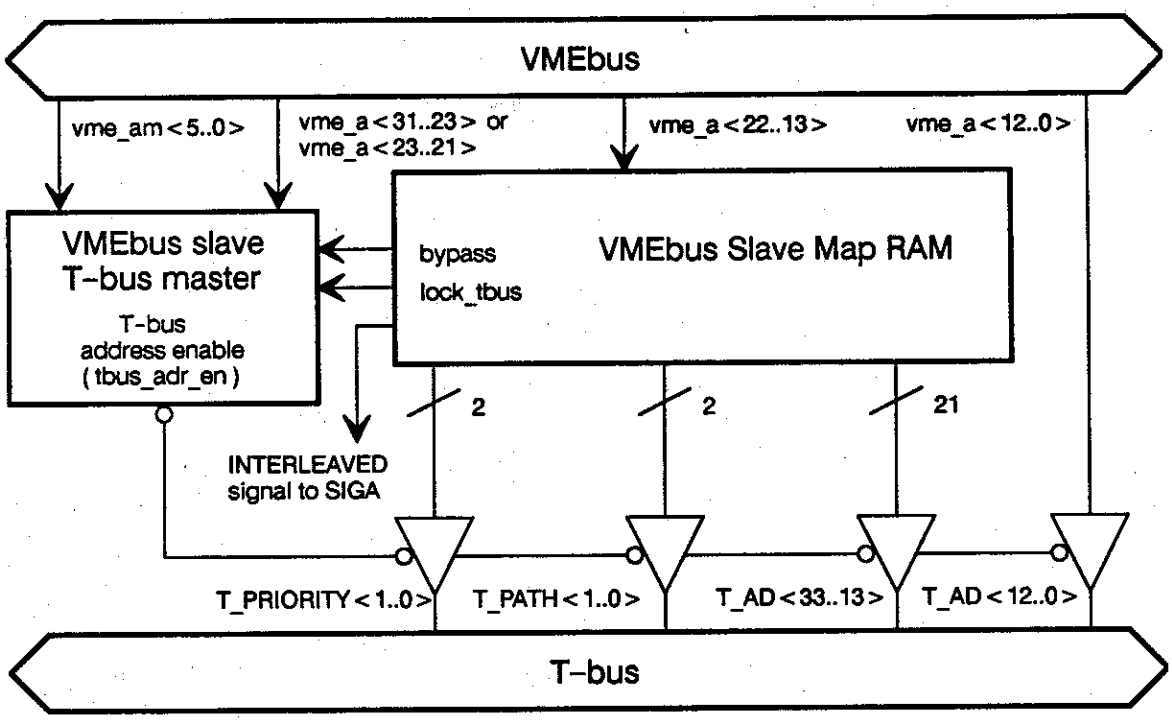


Figure 11-4. VMEbus slave mapper.

NOTE

TECHNICAL DETAIL
The B2VME VMEbus slave mapper receives the six bits of VMEbus address modifier. In the initial implementation, these bits are used only to determine whether the reference is using standard or extended addressing. This processing is performed in a programmable logic array (PLA) chip, however, so future

applications needing more subtle effects may be accommodated by substituting a differently programmed chip.

VMEbus Interrupt Handling and Generation

The B2VME generates interrupts to the VMEbus, and handles interrupts from the VMEbus. A Signetics SCB68154 chip implements the interrupt generator. This device has two internal read/write registers, the VMEbus Interrupt Request register and the VMEbus Interrupt Vector / Control register. The protocol for generating an interrupt on the VMEbus is discussed in detail in the SCB68154 data sheet in the 1986 *Signetics Microprocessor Manual.*

Interrupts received from the VMEbus result in interrupts to the 88000 on the B2VME as described in the section on interrupts to the CPU.

VMEbus Requester

The VMEbus master on a device signals the requester to obtain the bus when a transaction is required. A requester must obtain the bus before any transfers can be initiated. The requesters on the VMEbus communicate with the VMEbus arbiter to determine ownership of the bus. There are two types of requesters: release-on-request and release-when-done.

Release-on-request requesters do not let go of the bus after the transaction is completed. Instead, they wait until another master requests the bus before releasing it. If another transaction is requested by the the release-on-request requester's master, then arbitration does not take place again, because ownership has not been relinquished. Release-on-request requesters are useful when the associated master is the primary, or most frequent, owner of the bus.

Release-when-done requesters release the bus during or after the transaction is completed. Releasing the bus during the last transaction allows arbitration to take place while the transaction completes, and reduces the overhead of changing masters on the bus. Release-when-done requesters are useful when the associated master is not the primary master on the VMEbus.

The B2VME requester can be configured either as release-on-request or as release-when-done to tailor its performance to the application.

VMEbus System Controller

The B2VME can function as the VMEbus system controller when necessary. This option is jumper selectable. As the system controller, the B2VME will

perform the following VMEbus functions: arbiter, bus timer, IACK daisy chain driver and system clock driver.

VMEbus Arbitration

Three types of arbitration can be implemented on the VMEbus: prioritized, round-robin-select and single level. The simplest form of arbitration, single level, only monitors requests and issues grants on bus level 3. Because bus grants are daisy chained within a single level on the VMEbus, priority is assigned only by slot order in the VMEbus card cage. The B2VME VMEbus interface implements single level arbitration. Refer to *The VMEbus Specification* for descriptions of the other arbitration schemes.

The VMEbus arbiter is implemented as a finite state machine clocked at 32 megahertz. When the B2VME is functioning as VMEbus system controller, it also pulls up the bus grant 2, bus grant 1 and bus grant 0 signals to a high state.

IACK Daisy Chain Driver

The VMEbus interrupt acknowledge (IACK) signal is carried on two bits. One bit is common to all devices and announces that an interrupt acknowledgment is present. The other bit is daisy-chained from device to device, and each device must repeat that signal onto the next link of the chain, unless that device is processing the acknowledgment. The VMEbus system controller must take the single-wire signal and drive it onto the daisy chain. In the B2VME, the IACK daisy chain is driven by the SCB68154 (interrupt generator).

[THE SCB68154 CHIP SPECIFICATION AND DATA SHEET SHOULD BE INCLUDED AS AN APPENDIX TO THIS CHAPTER.]

VMEbus System Clock Driver

The VMEbus system controller must drive a 16 megahertz clock signal on the VMEbus. When the B2VME is operating as system controller, this signal will be generated by dividing the arbiter's 32 megahertz finite state machine clock by two and driving the result onto the bus.

VMEbus Bus Timer

See information in timer section.

11.5.3 Resets to and from the VMEbus

The Butterfly II generates a reset on the VMEbus by setting a bit in the VMEbus control register. Resetting the B2VME board does not reset the

VMEbus. Likewise, resetting the VMEbus does not reset the B2VME. VMEbus resets can be monitored by the B2VME. If the VMEbus is turned off, all signals are terminated correctly such that no spurious signals occur and the VMEbus is held reset.

11.5.4 Performance

The VMEbus slave interface sustains data rates of 8 megabytes per second. The VMEbus master interface performs single transfers at a rate of 5 megabytes per second. [THESE DESIGN FIGURES HAVE YET TO BE CHECKED BY MEASUREMENT.]

11.5.5 Summary of VMEbus Features Supported

In summary, the B2VME supports the following VMEbus-related features:

- T-bus initiated read and writes
- T-bus locking
- VMEbus initiated single reads and single writes
- Jumper selectable release-on-request or release-when-done
- Single level arbiter
- Interrupt handler
- Interrupt generator
- System clock driver
- Bus timer

11.6 Augmentation

[THIS SECTION MAY MOVE TO THE FUNCTION BOARD CHAPTER.]

Augmentation is a feature of the Butterfly II architecture that allows each CPU in the system to perform some operations auxiliary to normal functionality. The primary goal of augmentation is to provide a mechanism for performing atomic operations in the multiprocessor environment of the Butterfly II, where many processors can access the same memory. In addition, augmentation permits the implementation of some features, unique to the Butterfly II, that are not supported by the 88000.

An augmented memory reference results in the occurrence of some operation in addition to, or in place of, the reference. For example, augmenting a read or write operation with a memory lock causes the path to the memory to be held open after the read has completed. This gives the CPU exclusive access to the

target memory module until it releases the memory by clearing the lock augmentation. A mechanism of this type would not be required in a uniprocessor environment, where only one CPU initiates transfers to the memory.

11.6.1 Augmentable References

When augmentations are enabled, they affect only a subset of the references generated by the CPU. Augmentations are defined as side effects to data references, and thus instruction fetches are never augmented. In addition, augmenting certain data references, such as references associated with exception processing in the CPU, would result in incorrect system behavior and must be prevented. The architecture therefore supports an addressing facility that prevents augmentation.

Augmentation of data references can be prevented by mapping the data in the *bypassed* address space. A reference to local bypassed memory can be identified by detecting that the bypass bit in its physical address is set. The physical address is the address generated by the memory management unit. Refer to the memory system chapter for a complete explanation of the structure of the Butterfly II address space.

Table 11-1 lists the types of data that should not be augmented and should therefore reside in bypassed address space.

Table 11-1. Types of data references that should be mapped bypassed.

- MMU page table walks
- Exception vectors
- Supervisor stack references
- Augmentation register (AR)
- Configuration and control registers besides AR

11.6.2 The Augmentation Register

The Augmentation Register (AR) controls augmentations. The AR entry in the register summary section shows the assignment of the AR bits.

11.6.3 Disable Interrupts Augmentation

Interrupts to the processor are disabled by setting the disable interrupts bit in the AR. Setting this bit prevents the interrupt request input to the processor from being asserted. Disabling interrupts is a key step toward maintaining atomicity. This is discussed further below.

Interrupts are serviced by the processor when it is between instructions. Unfortunately, disabling the interrupt request input to the CPU does not guarantee that previously pending interrupts will not be serviced, because there is an internal interrupt request pipeline. For example, if a device is requesting an interrupt, and interrupts become disabled, the interrupt request will no longer be asserted at the processor interrupt request pin. However, the processor may still see and service the interrupt because the request was latched internally before it was de-asserted. If interrupts are disabled with a read instruction, and an interrupt has been latched internally, it will be serviced immediately following the read and before any subsequent instructions can be executed.

Interrupts to the processor cannot be disabled indefinitely. The Disable Interrupts augmentation is subject to the Interrupts Disabled Timer, discussed below in the section on configuration and control registers.

11.6.4 Lock Augmentation

Setting the lock bit in the AR causes the path to the resource being accessed, and the resource itself, to be held open after the access has completed.

The first augmentable reference with the lock augmentation enabled turns into an OPEN operation on the T-bus. Subsequent references turn into MAINTAIN T-bus accesses.

A transition from one to zero of the AR lock bit automatically generates a T-bus FREE_LOCKS cycle. This allows clearing the AR lock bit and generating the FREE_LOCKS to be performed in a single CPU instruction.

Refer to the *T-bus Specification*, or to the switch chapter, for a complete description of the Butterfly II lock protocol.

NOTE

TECHNICAL DETAIL
If the lock bit in the AR is "1", and the CPU locks a resource, the CPU interface remembers this. Then, when the lock bit is cleared to "0", the CPU interface automatically generates a FREE_LOCKS cycle on the local T-bus. If a remote resource was locked through a switch connection, the local SIGA knows this and, seeing the FREE_LOCKS on the local T-bus, propagates the drop-lock condition over the switch to the remote node, where the remote SIGA generates a FREE_LOCKS cycle on the remote T-bus, freeing the locked resource. The CPU cannot explicitly generate a FREE_LOCKS cycle, only implicitly as a side effect of clearing the lock bit in the AR after having locked a resource, or as a side effect of the XMEM instruction (described

above). If no resource is locked, setting and clearing the lock bit in the AR does not generate a FREE_LOCKS cycle.

11.6.5 Exception Processing During Locked Sequences

[THIS MAY BELONG BETTER IN A SECTION OR CHAPTER ON "PROGRAMMING THE B2VME" OR SOMETHING LIKE THAT.]

In the demand-paged memory environment of the Butterfly II, page fault and copy-on-write exceptions occur routinely during normal operation. However, processing these exceptions requires too much time to allow a locked path to remain open. Consequently, locked sequences are terminated when these faults occur. In addition, atomicity can not be guaranteed if the CPU can be interrupted during a locked sequence. If a locked sequence were interruptible, the interrupt handler might try to reference remote memory while the switch path out of the board was locked. Either the interrupt handler would have to be prevented from successfully completing the reference, or it would have to be permitted to break the lock.

If the interrupt routine was permitted to break the lock and the locked memory was accessed, then atomicity would no longer be guaranteed. If the interrupt handler was prevented from accessing the memory until the lock was freed, a deadlock situation could result because the interrupt handler would not return from the interrupt until it could access the memory, but the memory would not be freed until the interrupt handler returned and allowed the augmentation to complete. Alternatively, the interrupt handler could return from the interrupt without completing the interrupt service routine, and retry after the lock was released. In this case, the interrupt would not actually be serviced until the augmentation ended, and thus allowing the exception processing during the augmentation would not accomplish anything.

Disabling interrupts to the CPU during locked sequences would handle the problem of maintaining atomicity. However, a conflicting constraint in the Butterfly II is that to support real-time applications, the maximum delay before servicing an interrupt must be guaranteed. Consequently, interrupts cannot be disabled for arbitrary periods.

Page faults, copy-on-write faults and interrupts are all exceptions that can be handled similarly during locked sequences. The Augmentation Register has two bits called the *exception action* bits that can, with software support, handle the problems described.

The exception action bits distinguish three actions: continue, restart and abort, which can be taken at the end of exception processing. Under normal circumstances, the processor resumes operation at the instruction that terminated in

an exception. The "continue" exception action code indicates that the CPU should proceed normally.

If the exception action code is "restart", the exception handler loads the restart address, which has been saved in a software-defined register or memory, into the PC and returns from exception. The instruction sequence is then re-executed starting from the restart address. If the exception action code is "abort", then getting any exception is fatal to the process, and the process is aborted.

[EXAMPLE CODE BELOW IS REALLY THREE-PHASE. DESCRIPTION BELOW WAS FOR 68000 CODE AND MUST BE REVISED FOR THREE PHASE 88000 CODE. EXAMPLE CODE IS ATOMIC ADD, BUT DESCRIPTION WAS FOR READ TWO WORDS AND WRITE THEM BACK SWAPPED.]

Using the exception action bits, a two-phased approach can be used to prevent page-fault and copy-on-write exceptions, and to disable interrupts at the appropriate time during a locked sequence. A locked sequence can be divided into two phases: the information gathering phase and the data modification phase.

During the first phase, all data required to perform the locked operation is referenced and write access is tested. In addition, a label at the end of the locked sequence is referenced to ensure that all of the code is resident in main memory. The "restart" code is loaded into the exception action bits, the restart address is saved, and the lock bit in the AR is set. Page faults and write access violations are processed during phase one. The switch path to remote memory (if one is required) is established and locked. Interrupts should not be disabled during phase one.

If a bus error or an interrupt is taken during phase one, the AR is cleared by the exception handler. The lock bit in the AR changes from "1" to "0". This generates a T-bus FREE_LOCKS sequence (if a resource has been locked) and the switch path is torn down (if it has been opened). The exception processing is performed. At the end of the exception processing, the restart address is loaded into the PC and the CPU returns from the exception. The sequence starts over at the beginning of the phase one instruction sequence (i.e., at the restart address). If bus errors or interrupts continue occurring, execution resumes at the restart address again and again, until a full pass is made through phase one.

During the phase two, new data is written to memory. Note that until a write occurs that changes the destination, the entire process can be restarted without any problems. However, as soon as a modifying write has been performed, the state of the memory may no longer be consistent, and thus restarting the operation may result in incorrect results.

The transition from phase one to phase two takes place after all data and code has been referenced, and the path to the memory is opened and locked. Interrupts are disabled by reading the address in the AR block that sets the interrupt disable bit. Recall that reading the interrupt disable location to set the bit ensures that if an interrupt is pending, it will be serviced immediately after the read, and before any other instruction can be executed. The "abort" code is then loaded into the AR exception action bits. Setting the disable interrupts bit and the exception action bits must occur sequentially, because an interrupt can happen immediately after the disable interrupts bit is set, and it is not fatal. If the exception action bits were set to "abort" simultaneously with the setting of the disable interrupt bit, the interrupt which could occur immediately after this instruction would abort the process rather than restart it.

The transition to phase two completes when the "abort" code is loaded. At this time, all exceptions are fatal to the process. The path to the memory to be modified is already open (i.e., locked) from phase one, and data modification proceeds.

At the end of phase two, the AR is cleared. This causes a transition from "1" to "0" of the lock bit, which causes a FREE_LOCKS. Clearing the AR also re-enables interrupts and resets the exception action to "continue". The locked sequence is thus terminated.

An example of a locked sequence that conforms to the two-phased methodology described is shown below.

```
/* C code equivalent */
int aa_uaug(value,addr)
int *addr;
int value;
{
      *addr += value;     /* body done atomically, via user augmentation */
      return *addr;
}

      ;; ----- Augmentation register definitions -----
      def   AR_BLOCKBOTTOM,    0xE0758000
      def   AR_LOCK,     0x0004       ; lock switch path
      def   AR_DISINT,   0x0100       ; disable interrupts
      def   AR_RESTART,  0x0200       ; exception action -- restart
      def   AR_ABORT,    0x0600       ; exception action -- abort

      text
      global _aa_uaug
_aa_uaug:
      ; Assumes that bus error handler looks for restart address in r10
      or.u  r10,r0,hi16(_aa_uaug)     ; Set restart address into r10
      or    r10,r10,lo16(_aa_uaug)

      ;; Lock the switch path, restart if it faults
      ;; We exploit our knowledge that the flags are in the low half,
      ;;  to load the top half once and use it three times
```

```
        or.u   r12,r0,hi16(AR_BLOCKBOTTOM)
        ld     r13,r12,lo16(AR_BLOCKBOTTOM+AR_LOCK+AR_RESTART)

        br.n   bottom          ; Make sure all code pages are resident
        ld     r11,r3,0        ; Lock down data value, make sure it is resident

        ;; Transition to phase 2, disabling interrupts
        ;; We are reusing the top half loaded above
middle:
        ld     r13,r12,lo16(AR_BLOCKBOTTOM+AR_DISINT+AR_RESTART)
               ; Might take pending interrupt here

        addu r13,r13,0         ; Drain load/store pipe, take traps

        ;; Transition to phase 3, committing the transaction
        ;; We are reusing the top half loaded above
        ld     r13,r12,lo16(AR_BLOCKBOTTOM+AR_LOCK+AR_DISINT+AR_ABORT)

        addu   r2,r2,r11       ; Do the addition
        st     r2,r3,0         ; Store updated value at argument address

return:
        jmp.n r1               ; Return to caller (updated value is in r2)
        ld     r12,r12,lo16(AR_BLOCKBOTTOM)    ; Clear the augmentations
```

11.6.6 Synchronized Access Augmentation

Remote accesses can be attempted less frequently ("backed off") after the switch interface has received a reject. This throttles the retry rate and achieves a better pattern of switch traffic than would be attained if retries were allowed at any time. The SIGA has a very general mechanism for specifying the initial transmission and retry criterion for each message class. This mechanism is described in detail in the switch chapter.

Setting the *synchronized access* augmentation asserts the T-bus signal T_SYNC. This indicates to the SIGA that the current access should use the "Slot0" transmission strategy. By correctly programming the SIGA, this can produce the effect of attempting the current access using the backoff criterion on the initial transmission, rather than waiting until a reject has occurred. A use for this augmentation is described below.

11.6.7 Steal Augmentation

[SOME OF THIS SECTION MAY BELONG IN MEMORY CHAPTER]

A tag bit is associated with the 32 bits of each word in Butterfly II memory. The value of the tag bit is returned with every read access. A memory location is defined to be *stolen* when its tag bit is set. Reads from a location that has been stolen terminate in bus errors. The tag bit is associated with 32-bit quantities,

and attempting to access any portion (byte, half-word, word) of a location after it has been stolen results in a bus error.

There are two methods for stealing a memory location in the Butterfly II. One method uses the AR, and the other uses the Process Configuration Register (PCR). The latter method is described below in the section describing the PCR.

A location can be stolen by setting the steal bit in the AR and performing an augmentable write reference to the location. Note that the write will succeed regardless of the current value of the tag; no error is generated by writing to a stolen location.

To determine that a location is not already stolen before stealing it, the user must set the lock bit as well as the steal bit, and read from the location before performing the write. Success in reading the location indicates that it is not stolen (because reads from stolen locations terminate in bus errors), and locking the switch path ensures atomicity between the read and the write. [IS THIS OLD? ISN'T SETTING SNEAK AND LOOKING AT SNEAK DATA BETTER?]

The synchronized access augmentation can be used by the bus error handler when a read terminates in a "stolen" bus error. The bus error handler can reissue the read with the synchronized access bit of the AR asserted, signaling the SIGA that the read should be tried again, but at a throttled rate. This prevents the switch from being flooded with retries when a location is stolen and multiple processors are waiting for it to be freed.

[NEEDS AN APPROXIMATE ANALYSIS OF WORST CASE.]

11.6.8 Sneak Augmentation

The bus error resulting from reading a stolen location can be overridden by setting the *sneak* bit in the AR. If an operation terminates normally on the T-bus (i.e., with COMPLETED) with the tag bit set, and the sneak augmentation is enabled, the CPU cycle terminates normally instead of with a bus error. The tag information is stored in the sneak data bit in the PCR, discussed below.

The sneak augmentation can be used only on non-cacheable data because only one bit of sneak data is preserved in a bus transaction. Cache-fill transactions, generated automatically by the processor when accessing cacheable data, request four words of data, thus three bits of sneak data are lost in a cache fill. Attempting to sneak cacheable data results in an a bus error.

11.6.9 Absolute Switch Priority Augmentation

The *absolute priority* augmentation overrides the priority scheme mechanism described below in the section on default switch priority and priority schemes. If either of the absolute priority augmentation bits in the AR is cleared to "0", the priority encoded in both of them is the switch priority of all the remote data references until they are both set to "1" again. The absolute priority bits are asserted low to conform with the T-bus specification.

11.6.10 Set Bank <1..0> Augmentation

The Butterfly II addressing structure permits direct access to 8 megabytes of memory at each remote node in a machine with 512 switch ports. In "small" machines (those with 64 or fewer switch ports), the addressing structure permits direct access to 32 megabytes. Refer to *Butterfly II System Architecture* for a complete description of the address structure.

Setting bank bits <1..0> in the AR permits access to the remaining 24 megabytes addressable on each node. Access to these three additional, 8-megabyte banks is controlled by three bits in the User Augmentation Enable Mask register, so access to remote banks can be allowed on a per-bank basis. The bank mask scheme is illustrated in Figure 11-5.

In a Butterfly II machine with less than 64 switch ports, the bank bits override the physical address bits. [CLARIFICATION OF THIS IS PENDING.]

Figure 11-5. Remote bank mask scheme.

11.7 The Process Configuration Register

[THIS SECTION MAY MOVE TO THE FUNCTION BOARD CHAPTER.]

The Process Configuration Register (PCR) is an 8–bit register containing information associated with the software process currently running on the processor.

11.7.1 Default Switch Priority and Priority Schemes

The priority bits on the T–bus are used by the SIGA to determine the switch priority of a remote memory access. Each master on the T–bus must drive the priority bits. The default priority of the messages sent can be set independently in each master interface.

In the CPU interface, the default priority may be different for different types of messages. For example, a burst read may have a different priority than a single read, and a steal may have a different priority than a sneak.

Adjusting the priority as a function of the message type is a feature of the Butterfly II architecture that was not available in the original Butterfly machine. Consequently, a variety of priority schemes may need to be examined before an optimal approach can be determined. In addition, dependent upon the application, varying schemes may be required. To facilitate the testing and use of different priority schemes, a hardware mechanism is provided to let the software select a priority algorithm. The software may select from up to four different schemes programmable in the firmware of the machine.

Two bits in the PCR select the priority scheme to use. In addition, two bits select the default priority. The default priority bits may or may not be used in the priority scheme.

11.7.2 The Path Bits

The two path bits in the PCR select between the primary and secondary switch networks available in a Butterfly II machine for remote references. Two bits are used to also differentiate local accesses from switch accesses; this permits local T–bus slaves to detect accesses without having to decode all of the upper bits of the address. "11" indicates local access, "01" means SIGA A, and "10" means SIGA B.

11.7.3 Sneak Data

The contents of the 33rd data bit resulting from a memory reference that was augmented with a sneak can be found by reading bit 6 of the PCR. This location is updated on every augmentable reference after the sneak bit has been set.

11.8 Timers

A variety of timers are available in the B2VME. The timers are provided to guarantee maximum interrupt latency, to prevent inappropriate use of the Lock and Disable Interrupts augmentations, to be used as a software tool, and to detect faulty hardware. Table 11–2 is a table of the timers available on the B2VME, the range of each timer, a brief description of its function, and the action taken if the timer expires. Each timer is described below.

Table 11-2. Timers in the B2VME.

Name and Range	Purpose	Action on Expiration
	LOCK AND INTERRUPT TIMERS	
CPU Lock Timer 1 - 255 microseconds	Limit how long the CPU may hold a lock.	Generate a FREE_LOCKS cycle. CPU will later get a "maintain present" error.
Interrupts Disabled Timer 1 - 255 microseconds	Help guarantee maximum interrupt service latency.	Interrupt.
Interrupts Pending / Abort Retries Timer 1 - 255 microseconds	Help guarantee maximum interrupt service latency.	Signal SIGA to abort retries in case connection establishment is in progress. CPU gets bus error if retries are aborted.
	SWITCH PROTOCOL TIMERS	
Reject Timer 1 microsecond - 0.49 seconds	Prevent SIGA from trying too long to establish a connection.	Bus error.
Connection Timer 1 - 255 microseconds	Prevent switch connection from being held open too long.	Tear down connection. CPU gets bus error — code and timing depend on when timer expires.
	REAL TIME CLOCK TIMERS	
Time Of Next Interrupt - A (TONI-A) 1 microsecond - 1 hour	Allow software to ask for an interrupt at a specified time.	Interrupt.
Time Of Next Interrupt - B (TONI-B) 1 microsecond - 1 hour	Allow software to ask for an interrupt at a specified time.	Interrupt.
	VMEbus INTERFACE TIMERS	
VMEbus Arbiter Timer 4 - 1020 microseconds	Limit how long VMEbus bus grant may be asserted without bus busy.	Arbiter removes bus grant.
VMEbus B2VME Master Bus Timer 1 - 255 microseconds	Limit how long the B2VME as VMEbus master may await a response from a slave.	Remove VMEbus signal address strobe. Generate bus error on T-bus.
VMEbus System Bus Timer 4 - 1020 microseconds	Limit how long any VMEbus master may await a response from a slave.	Assert VMEbus signal BERR.

11.8.1 Switch Reject Timer

The Switch Interface Gate Array (SIGA) implements a timer, called the switch reject timer, that limits the maximum time spent trying to establish a path through the Butterfly II switch. The switch reject timer is enabled whenever a remote memory access is attempted. The SIGA samples the state of this timer whenever it receives a reject. If the timer has expired, the SIGA stops trying to establish the connection, and a bus error is returned to the CPU. In normal operation, the switch reject timer should not time out. Its expiration may indicate broken hardware, or attempting to access a non-existent or disconnected switch port.

11.8.2 Switch Connection Timers

The SIGA contains two switch protocol timers: the Reject Timer and the Connection Timer. These detect anomalies in switch message transactions. Expiration of the Reject Timer is signaled on the B2VME's T-bus as a bus error with error code Rej_TO. Expiration of the Connection Timer similarly results in a bus error with code Conn_TO or Wait_TO, depending on when during the connection the timer expires. See the switch chapter for more details.

11.8.3 Lock Timer

[THIS SECTION MAY MOVE TO SWITCH CHAPTER.]

Every T-bus master that generates a lock requires a timer to prevent keeping a slave locked too long. The B2VME has two masters that can generate locks: the CPU interface and the VMEbus slave/T-bus master interface. The CPU interface has a timer that is enabled when the slave is locked and disabled when the lock is freed. If the timer expires, a FREE_LOCKS cycle is generated, but the master's lock bit is not cleared. Thus, the CPU still considers itself to hold the slave locked. If the CPU does not clear the lock before making another reference to that slave, it will make a MAINTAIN access, because it will appear to be locked. The slave will return an error because it will no longer be locked because the FREE_LOCKS cycle was generated.

The VMEbus slave does not have a lock timer because the VMEbus lock accesses should always complete in two cycles.

11.8.4 Interrupt Pending

[THIS SECTION MAY MOVE TO SWITCH CHAPTER.]

The time to establish a path through the Butterfly II switch to a remote memory can be large. When a CPU executes an instruction which makes a reference to

remote memory, the instruction does not complete until the path has been established through the switch, and the data has been read or written. Since interrupts are serviced only between CPU instructions, they could be ignored for longer than the maximum permitted delay to service interrupts, if the time to establish the switch path was too long. The switch reject timer could address this problem, but it would impose conflicting constraints on the duration of this timer. For detecting broken or missing hardware, the switch reject timer should be set to a long timeout period. To guarantee maximum interrupt latency, however, the switch reject timer would have to be set to a relatively short (microseconds range) time. The "siga_abort_retries" input to the SIGA is used to address this problem.

When an interrupt is being requested of the processor, the abort retries finite state machine is enabled. This state machine waits for a programmable period of time, and then asserts the "siga_abort_retries" signal. The siga_abort_retries signal is asserted only if the processor is currently waiting for an access to complete. The time period can be specified from 1 to 255 microseconds, or infinite. Setting the time to infinite disables the siga_abort_retries mechanism.

If the siga_abort_retries signal is asserted when the SIGA has received a reject, the attempt to open the switch path (retry) is aborted and a bus error is returned to the CPU. The interrupt can then be serviced, as the CPU is no longer in the middle of an instruction. Using the interrupt request signal to abort switch retries in this manner decouples the maximum switch latency from the maximum latency to service an interrupt. Refer to the switch chapter for more information about the switch reject timer and siga_abort_retries.

11.8.5 Interrupts Disabled Timer

[THIS SECTION MAY MOVE TO FUNCTION BOARD CHAPTER.]

The interrupts disabled timer is enabled when interrupts to the processor are disabled by the disable interrupts augmentation. This timer can be set by software from 1 to 255 microseconds. This timer is used to ensure that interrupts are not disabled so long that the maximum interrupt latency cannot be guaranteed.

When the interrupts disabled timer expires, an interrupt is generated to the CPU. The interrupt handler should clear the AR. Any locks associated with the augmentation are freed by the T-bus FREE_LOCKS cycle that automatically occurs when the lock bit changes from one to zero.

11.8.6 Real Time Clock Interrupt (TONI)

Refer to the switch chapter.

11.9 Bus Errors

There are a variety of conditions in the Butterfly II machine that terminate CPU cycles with a bus error. Because the number of bus error conditions is rather large, all bus error causes are prioritized and encoded. The 88000 can read the encoded information and can use it as an offset into a dispatch table in the bus error handler. This mechanism is included to improve the bus error service latency.

Figure 11-6 illustrates the encoding of the bus error information and also provides a cross reference of the sections in this document where each condition is described. The error code is five bits, and appears in bits 6..2; error code bits 7, 1 and 0 are always "1".

```
e d c b a Switch Requester / CSU Error    [ THIS IS PER 7/7/88 LIST ]
========= ====================
0 0 0 0 0 Maintain_Absent (1a)
0 0 0 0 1 Maintain_Present (1b)
0 0 0 1 0 Stolen Verify
0 0 0 1 1 Lock_Address (2)
0 0 1 0 0 Wait_TO (4a)
0 0 1 0 1 Idle_TO (4b)
0 0 1 1 0 Rej_Abort (5)
0 0 1 1 1 Rej_TO (6)
0 1 0 0 0 Reverse (7)
0 1 0 0 1 Check (8)
0 1 0 1 0 Miscellaneous CSU Error

      Priority is from highest (1) to lowest (8).
      within a given priority, errors are mutually exclusive
      (i.e., 4a,4b...).

e d c b a Switch Server Error
========= ==================
0 1 1 0 0 Downstream_Refused
0 1 1 0 1 Downstream_Write
0 1 1 1 0 Downstream_Late
0 1 1 1 1 Downstream_OTL

e d c b a VMEbus Master/T-bus Slave
========= ========================
1 0 0 0 0 Vme Maintain_absent
1 0 0 0 1 Vme_Maintain_present
1 0 0 1 0 Vme_Burst_attempt
1 0 0 1 1 Vme_Bus_error

e d c b a Local I/O
========= ========================
1 0 1 0 0 Local_lock_attempt
1 0 1 0 1 Local_burst_attempt
1 0 1 1 0 Local_load_ar_error

e d c b a Memory
```

```
========= ========
1 1 1 0 0 Mem_Maintain_absent
1 1 1 0 1 Mem_Maintain_present
1 1 0 0 0 Mem_Parity_err

e d c b a CPU Interface
========= =================
1 1 0 0 1 CPU_sneak_on_cacheable
1 1 0 1 0 CPU_stolen
1 1 0 1 1 CPU_write_error
1 1 1 1 0 CPU_no_response

1 1 1 1 1 RESERVED
0 1 0 1 1 RESERVED???
1 0 1 1 1 RESERVED???
```

Switch Error Definitions:

Maintain_Absent	A NORMAL was issued to a slave during its idle state and the slave was locked.
Maintain_Present	A MAINTAIN was issued to a slave during its idle state and the slave was NOT locked.
Stolen Verify	???
Lock_Address	A function request [EITHER READ OR WRITE???] was made to a locked requester during its idle state with a node address different than that which opened the locked sequence.
Wait_TO	The switch connection timer expired while the requester was waiting for a function response.
Idle_TO	The switch connection timer expired while the requester was in its idle state.
Rej_Abort	The switch reject timer was forced to expire by the the REJ_ABORT input pin.
Rej_TO	The switch reject timer expired while the requester was attempting to open a connection.
Reverse	The requester detected an incorrect polarity of the reverse signal during a function response.
Check	The requester detected an incorrect checksum during a function response.

Downstream_Write	A downstream write error was detected while the downstream server was sending data.
Downstream_OTL	A downstream T-bus slave did not respond to the server's request.
Downstream_Late	A downstream T-bus slave responded with a LATE error.
Downstream_Refused	A downstream T-bus slave responded with REFUSED-LOCKED when the server thought itself locked.
CSU Error	An error was made accessing the CSU. It could be one or both of the of the following: 1) An OPEN lock was requested. 2) A Multi-word transfer was requested.

[ABOVE DESCRIPTIONS ARE SIGA ONLY. NEED OTHERS TOO.]

Figure 11-6. Bus error encoding.

11.10 Registers by Functional Group

The registers described here are accessible by any T-bus master on the board. The T-bus slave interface for these status and configuration registers responds for all of the B2VME registers except the VMEbus mapper RAMs (both master and slave) and the interleaver RAM. These devices are loaded by the SIGA-assisted interleaver loader, discussed below.

The registers are described in five groups, divided roughly by function. The groups are:

- User and configuration registers — control process state and configuration of the machine
- Interrupt system registers — involved with interrupts within the machine
- Bus error registers — indicate the source of a bus error
- Latency control registers — timers that prevent hangups
- VMEbus interface registers — involved with VMEbus operation, including VMEbus interrupts and timers

[THE FOLLOWING SUBSECTIONS NEED SOME POLISHING. IN CASE OF DISCREPANCY WITH THE REGISTER LIST IN THE LAST SECTION, THE REGISTER LIST SHOULD BE BELIEVED.]

11.10.1 User and Configuration Registers

User Registers

The B2VME has three user-accessible registers: the *interprocessor interrupt* register, the *process configuration* register and the *augmentation register block.* [ALSO LIGHTWEIGHT STEAL.] Each of these registers resides on its own page to provide maximum flexibility for the operating system in permitting each individual process to access these registers. The interprocessor interrupt register also resides in a globally accessible page (bank 0), and thus it can be accessed remotely without an augmentation. The PCR and AR Block are intended for use by the local processor, and thus they do not reside on globally accessible pages. Every bit in these registers is covered by a mask bit. Attempts to set a bit that is masked in any of these registers returns a register load error. The PCR and Interprocessor Interrupt registers are read/write. The AR Block is read only, although reads of the AR Block result in the setting and clearing of its contents.

Except for the user registers noted above, all registers are intended to be accessible by privileged processes only. The page where these registers reside should be mapped in supervisor address space.

Process Context

The process context must be saved and restored upon servicing interrupts and process switching. In addition to the normally required 88000 context, the B2VME contains registers that are part of the process context and must be saved, cleared and restored appropriately. These are listed below.

- **Augmentation register** (AR)
- **Process Configuration register** (PCR)
- **Lightweight Steal register**
- **User Augmentation Enable Mask register**, if its contents vary from process to process
- **PCR Disable Mask register**, if its contents vary from process to process

Augmentation Register (read/write)

The Augmentation Register (AR) can be read and written as data at location 0xE0780004. You can also load the AR by reading the AR block with bits 13..2 set to correspond to the AR bits that you want to set.

Loading the Augmentation Register

The AR appears in several locations in the local address space. Accessing the AR at different locations produces different side effects. These side effects allow efficient use of the AR during normal operation and during exception processing.

The AR resides in a 16-kilobyte block of memory called the AR block. Bits 15..2 in the address of an AR block reference represent augmentations. Reading a location in the AR block enables the augmentations whose bits in the address are "1". For example, referring to Figure 11-8, reading location 0xE0758400 enables the lock augmentation. Reading location 0xE075A400 enables lock and steal.

The AR is cleared either by reading from location zero of the AR block, or by reading from the AR read and clear register. Reading the AR read and clear register returns the contents of the AR and clears it. The contents of the AR can be read without clearing it by reading the AR Read Register.

The AR is part of the state of the current process, and thus it must be saved and restored when the CPU is processing exceptions. An efficient way to implement this is to read the AR from the AR read and clear address at the beginning of the exception handler. At the end of the exception handler, the AR value can be loaded back into the AR by writing it to the AR read/write register.

A map of the AR registers and other B2VME registers is illustrated in the section on configuration and control registers.

Augmentation Register Read and Clear (read)

Reading this register returns the contents of the AR and clears it. This register is intended for use by exception handling software to efficiently save the state of the AR, and clear it.

User Augmentation Enable Mask Register (read/write)

The Augmentation Register Mask is applied to reads of the AR block. It is not applied to the Augmentation Register itself because that register resides in a privileged page to which users will not have access, thus it does not need to be protected by the mask.

If an AR block access attempts to set a bit that is masked, a register load error is returned. The bit gets set, and it is the responsibility of the bus error handler to clear it.

In some environments, the operating system may want to give a user processes access to a subset of the augmentations available, but not to all of them. But,

because the entire AR block resides in two pages of memory, the operating system has very little granularity with which to control access to the AR. If the augmentation register is mapped into user space at all, the user has access to many of the augmentations.

The User Augmentation Enable Mask is applied to the AR to control user process access to each bit in the AR. If the mask bit is "1" for a given AR bit, then the user is permitted to enable the augmentation. If the mask bit is "0", then attempts by a user process to set the AR bit result in a bus error.

The User Augmentation Enable Mask is initialized by power-up to all zeros. It resides in a page of registers intended only for supervisor access. The AR read-and-clear and AR read/write registers are also in this page of registers.

Lightweight Steal

Using the augmentation register to steal memory locations requires a large overhead of instructions for turning on the AR bits in the right sequence with turning off interrupts. To address this problem, the Butterfly II architecture includes an alternate method for steal that requires very little overhead.

Setting the *lightweight steal* bit in the Lightweight Steal register causes the next XMEM instruction to turn into a steal. Recall that the XMEM instruction automatically generates a T-bus lock that is freed when the write completes. If the read portion of the XMEM completes, then the location is not stolen, and the write portion steals the location. The lightweight steal register is cleared when a steal occurs; thus, clearing the register is not necessary.

Process Configuration register (PCR) (read/write)

The PCR holds information about the way remote accesses are presented to the switch while the current process is executing. It also contains the *sneak data* bit, which is how the process obtains part of the result of a sneak access, an augmentation that circumvents a lock.

PCR Disable Mask (read/write)

The Process Configuration register also resides in a page that may be mapped user accessible. Each bit in the PCR is covered by a mask bit. Attempts to write to a masked bit result in a register load error.

Machine Configuration (read/partial write)

This register has one bit, the machine size bit, that indicates whether to use small (less than 64 switch ports) or large (64 to 512 switch ports) machine addressing. The small machine contains a 2–column switch, while the large machine has a 3–column switch. This bit can be read back, and also the switch port number where the board is plugged in can be read through this register.

11.10.2 Interrupt System Registers

Interrupt Mask (read/write)

The upper seven bits of the Interrupt Mask are applied to incoming VMEbus interrupts. If an interrupt is masked, it will not generate an interrupt to the 88000. Bit 0 of the Interrupt Mask is applied to the Interprocessor Interrupt register. If a user attempts to set the interrupt bit when it is masked, a register load error is returned.

Interprocessor Interrupt (read/write)

This one–bit register can be set to generate an interrupt to the processor. It is cleared by reading the Interrupt Source register, or by writing a zero to the Interprocessor Interrupt register.

Non–maskable Interprocessor Interrupt (read/write)

This one–bit register can be set to generate a non–maskable interrupt to the processor. It is cleared by reading the Interrupt Source register, or by writing a zero to the Non–maskable Interprocessor Interrupt register.

Interrupt Source (read)

The current sources of interrupts are partially encoded and recorded in this register.

11.10.3 Bus Error Registers

Bus Error Vector (read)

The 5–bit encoded source of the most recent bus error to the CPU is recorded in this register. See the section on bus errors for the encoding of these bits.

Write Error Source (read)

When an error is returned to the CPU on a write, the slave cannot always supply an error code because the CPU may still be driving the T-bus with write data. In this case, the source of the write error, and some of the state, are stored in this register to aid in determining the type of error that occurred. Note that not all error bits apply to each bus slave.

11.10.4 Latency Control Registers

CPU Lock Timer (write)

The CPU Lock Timer register is loaded with the 8-bit number *t*, where 0xff-*t* is the number of microseconds before the CPU lock is timed out.

Interrupts Disabled Timer (write)

The Interrupts Disabled Timer register is loaded with the 8-bit number *t*, where 0xff-*t* is the number of microseconds before an interrupt is generated because interrupts have been disabled for too long.

Interrupts Pending/Abort Retries Timer (write/partial read)

Bits 8..1 of the Interrupts Pending/Abort Retries Timer register represent the number *t*, where 0xff-*t* is the number of microseconds an interrupt must be pending before it causes switch accesses to be aborted. If bit 0 is set, the time is infinite, and the timer is effectively disabled. Only bit 0 can be read back.

11.10.5 VMEbus Interface Registers

Six registers configure and control the VMEbus interface.

VMEbus Configuration (read/partial write)

The VMEbus Configuration register selects the 4-megabyte window in the VMEbus space to which the B2VME VMEbus slave interface will respond. Also, it selects whether the B2VME responds to VMEbus standard or extended address format.

The VMEbus Configuration register also controls whether the B2VME VMEbus master interface uses a release-on-request or release-when-done requester.

Finally, the *reset VMEbus* bit generates a VMEbus reset.

All of the above bits can be read back, and additionally, the status of the VMEbus *system reset* and *system fail* signals are monitored by reading this register.

VMEbus Arbiter Timer

The VMEbus system controller must time out two signals on the VMEbus: *bus grant* and *address strobe*.

If the arbiter grants the bus by asserting bus grant, and no master responds by asserting *bus busy*, the arbiter timer must indicate to the arbiter that no device has responded and therefore the grant should be removed. This situation should never occur if all devices are designed according to *The VMEbus Specification*, but timing it out allows forward progress if it does occur.

The VMEbus Arbiter Timer register is loaded with the 8-bit number *t*, where 0xff–*t* is (the number of microseconds)/4 before an arbitration cycle is timed out. This timer is applied to the arbiter bus grant and is disabled as soon as a VMEbus master acknowledges the grant. Under normal operation, this timer should never expire.

This used only when the B2VME is VMEbus system controller.

VMEbus System Bus Timer

If address strobe is asserted and no slave device responds with the VMEbus signal DTACK, the system controller must time out address strobe by asserting BERR. This situation occurs more frequently than timing out bus grant. For example, on system power-up, a CPU may wish to poll the bus to find available memory. The address strobe timer can be set by software from 4 to 1000 microseconds.

The VMEbus Bus Timer register is loaded with the 8-bit number *t*, where 0xff–*t* is (the number of microseconds)/4 before a VMEbus access is timed out. This timer is applied to the VMEbus address strobe, and thus affects all VMEbus transactions.

VMEbus B2VME Master Bus Timer

The VMEbus B2VME Master Bus Timer register is loaded with the 8-bit number *t*, where 0xff–*t* is the number of microseconds before a VMEbus access initiated by the B2VME VMEbus master interface is timed out. This timer prevents the maximum switch latency of the Butterfly II machine from being affected by long VMEbus timeouts.

The VMEbus System Bus Timer, described above, times out the whole VMEbus. The B2VME Master Bus Timer only times out references initiated

by the B2VME's VMEbus master interface. This one is set short; the system one is set long.

VMEbus Interrupt Generator Registers

The VMEbus interrupt generator chip (SCB68154) used by the B2VME contains a register to set the interrupt level requested, and a register to supply an interrupt vector. Both of these registers are read/write, and are described schematically in the register list here, and in detail in the 1986 *Signetics Microprocessor Manual*.

11.11 Register Address Map

[HERE SHOULD BE A LIST OF ALL REGISTERS, INCLUDING SIGA REGISTERS, SORTED BY ADDRESS.]

0x00000000 to 0x0000FFFF Debug daughter board, if installed (see Debug Connector section); if no daughter board is installed, this is normal memory

0x80002000 Interprocessor Interrupt register

0xE0740000 Process Configuration register (PCR)

0xE0741000 Lightweight Steal register

0xE0758000 to 0xE075BFFC Augmentation register, as AR block

0xE0760000 Machine Configuration Read register (on read)
Machine Configuration Write register (on write)

0xE0760004 Non-maskable Interprocessor Interrupt register

0xE0760008 Interrupts Pending / Abort Retries Timer register

0xE0760200 VMEbus B2VME Master Bus Timer register

0xE0760204 VMEbus Configuration register

0xE0760208 VMEbus System Bus Timer register

0xE076020C VMEbus Arbiter Timer register

0xE0760800 VMEbus Interrupt Vector / Control register

0xE0760804 VMEbus Interrupt Request register

0xE0761000 to 0xE0761FFF VMEbus Slave Map RAM registers (no T-bus access)

0xE0780000 Augmentation register, as AR read-and-clear (on read) migrated to E07800C
CPU Lock Timer register (on write)

0xE0780004 Augmentation register, as AR read/write

0xE0780008	User Augmentation Enable Mask register
0xE078000C	Bus Error Vector register
0xE0780010	PCR Disable Mask register
0xE0780014	Interrupt Enable Mask register
0xE0780018	Interrupt Source register (on read) Interrupts Disabled Timer register (on write)
0xE078001C	Write Error register [~~MAY BE~~ has been COMBINED WITH BUS ERROR VECTOR REGISTER]
0xE07E0000	RTC (Real Time Clock) Control A register
0xE07E0004	TONI (Time Of Next Interrupt) A register
0xE07E8000	RTC (Real Time Clock) register
0xE07E8004	Transmit Time Configuration register
0xE07E8008	Message Classification register
0xE07E800C	Timer Configuration register
0xE07E8010	Frame Timer register
0xE07E8014	Server Connection Timer register
0xE07E8018	Priority Promotion Timer register
0xE07E8020	Switch Message Checksum register
0xE07E8024	Switch Port Control / Interleave Control register
0xE07E8028	Synchronizer Delays register
0xE07E802C	Primary / Secondary Control register
.....	other SIGA registers
0xE07A0000 to 0xE07A1FFF	Interleaver / VMEbus Master Map RAM Read/Write Control registers
.....	SIGA B registers, as SIGA A registers above but using pages x through y

Figure 11-7. B2VME T-bus registers.

11.12 B2VME Register Summary

Certain registers are present in B2VME function board hardware, but are described in other chapters or other documents. These are:

- CPU and CMMU registers. These are described in Motorola literature.
- Test and Control System (TCS) slave registers. These are described in the TCS chapter. [I AM NOT SURE WHAT I MEAN HERE.]

- SIGA control and configuration registers. These are described in the TCS chapter or the switch chapter. [I HAVE TO DECIDE WHICH.]

The B2VME registers described here are accessed via the T-bus, except the three RAMs noted. The registers are:

- Non-VMEbus B2VME control and configuration registers:
  - Augmentation register (AR)
  - Bus Error Vector register (and Write Error register)
  - CPU Lock Timer register
  - Interleaver RAM registers (no T-bus access)
  - Interprocessor Interrupt register
  - Interrupt Enable Mask register
  - Interrupt Source register
  - Interrupts Disabled Timer register
  - Interrupts Pending / Abort Retries Timer register
  - Lightweight Steal register
  - Machine Configuration Read register, and Machine Configuration Write register
  - Non-maskable Interprocessor Interrupt register
  - PCR Disable Mask register
  - Process Configuration register (PCR)
  - User Augmentation Enable Mask register
- B2VME VMEbus interface registers:
  - VMEbus Arbiter Timer register
  - VMEbus B2VME Master Bus Timer register
  - VMEbus Configuration register
  - VMEbus Interrupt Request register
  - VMEbus Interrupt Vector / Control register
  - VMEbus Master Map RAM registers (no T-bus access)
  - VMEbus Slave Map RAM registers (no T-bus access)
  - VMEbus System Bus Timer register
  - (also, the Interrupt Enable Mask register and the Interrupt Source register contain VMEbus-related fields)

Each of these registers, in alphabetical order, is summarized below. Registers in this summary should be accessed as word (32-bit) quantities, aligned on word boundaries, even if the only meaningful bits lie in an upper byte of the word. The values of unused bits are irrelevant (don't care) on a write, and are unspecified on a read.

The B2VME registers described here are intended for write access by privileged processes only, typically supervisor mode only, with three exceptions. The three user-writable registers are the Augmentation register (accessed as the AR block), the Interprocessor Interrupt register and the Process Configuration register. [ALSO LIGHTWEIGHT STEAL???] **The hardware does not enforce any register access restrictions explicitly based on the CPU mode (supervisor, user).** Instead, it is up to the software to map the registers to pages with appropriate access protection. Further, writing to each field in the three user-writable registers is protected by a mask bit in an associated mask register. Thus, to access a register, the register must be mapped to permit access in the current mode; and if the register is protected by a mask and the operation is a write, that mask must be set to permit access also.

The description of each register includes its contents after power-on. The TCS power-up operation modifies the contents of several registers, some of which depend on the configuration of the system. See TCS user or site administration documentation for those details.

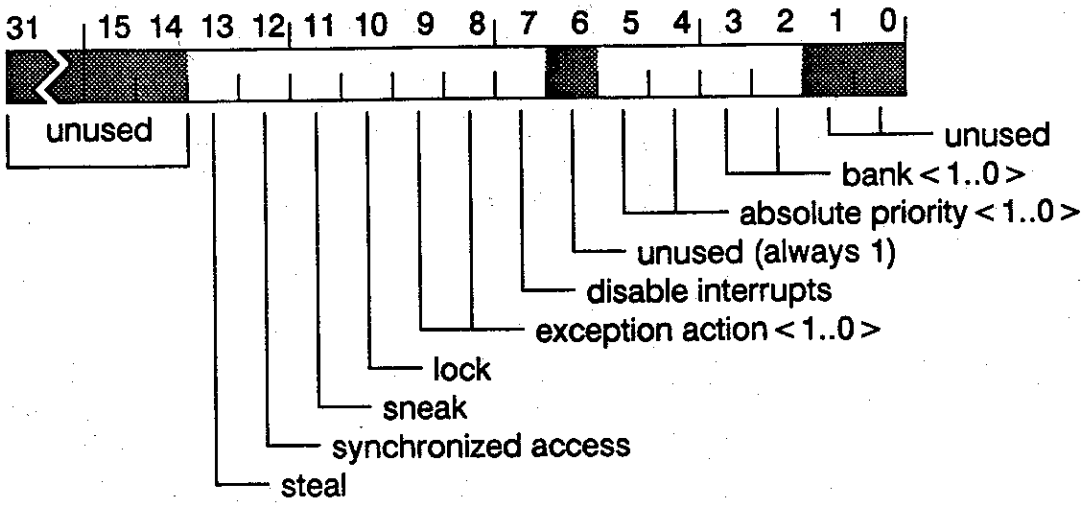


Figure 11-8. Augmentation register (AR).

FUNCTION Enable augmentations to 88000 CPU instructions.

ADDRESS 0xE0758000 to 0xE075BFFC — AR block
0xE0780000 — AR read-and-clear (CPU Lock Timer register on write)
0xE0780004 — AR read/write

ACCESS AR block — read only, with side effects, subject to User Augmentation Enable Mask register
AR read-and-clear — read only, with side effects
AR read/write — read/write

DESCRIPTION There is just one Augmentation register, and it may be referenced in three ways. It is intended that the operating system permit user-mode access to the AR only via the AR block.

- Reading a (word-aligned) location in the AR block causes bits 13..2 of the address to be loaded into bits 13..2 of the AR. The data returned in bits 31..16 is unspecified, and in bits 15..0 is all ones. WARNING: WHAT DATA IS RETURNED MAY BE REVISED. Writing to the AR block has the same effect as reading it; the write data is ignored.
- Reading the AR read-and-clear location returns the contents of the AR and clears the AR to zero.
- Accessed at the AR read/write location, the AR may be read without clearing it, and may be written.

The *bank* bits permit access to the high 24 megabytes of the 32 megabytes of address space on remote nodes of a large machine. On a small machine, the bank bits override certain physical address bits. [NEEDS CLARIFICATION.]

The *absolute priority* bits define a value to override the *priority scheme* bits of the Process Configuration register. These bits determine the priority of switch messages; see the PCR description and the switch chapter for details. If the absolute priority<1..0> bits of the AR are...

0 0 then T_PRIORITY<1..0> = 0 0
0 1 then T_PRIORITY<1..0> = 0 1
1 0 then T_PRIORITY<1..0> = 1 0
1 1 then T_PRIORITY<1..0> = as determined by PCR

The *disable interrupts* bit, when set to one, prevents assertion of the CPU's interrupt request pin. Requests already in the CPU's internal pipeline are not disabled. Interrupts from the VMEbus, from the real time clock timer(s), and from the maskable interprocessor interrupt are disabled by this bit. Two interrupts are **not** disabled by this bit: the interrupts disabled too long timeout, and the non-maskable interprocessor interrupt.

The *exception action* bits describe what action to take at the end of exception processing. The hardware does not use these bits; they are provided for use by software, to tell the exception handler software what to do. Their meaning is only a programming convention, although "00" should mean "continue" because the AR is cleared to zero upon power-up.

0 0 continue (proceed normally)
0 1 restart (resume instruction at restart address)
1 0 abort (getting any exception is fatal to the process)
1 1 undefined

The *lock* bit, when set to one, causes the path to the resource that will be accessed, and the resource itself, to be held open after the access has completed. Changing the lock bit from "1" to "0" generates a T-bus FREE_LOCKS cycle if a resource has been locked.

The *sneak* bit, when set to one, makes read operations immune to whether the location read is stolen; the read operates normally instead of causing a bus error. The sneak bit has no effect on reading non-stolen locations.

The *synchronized access* bit, when set to one, asserts the T-bus bit T_SYNC on switch transactions, which tells the SIGA to use "Slot0" transmission strategy. The intent is that the SIGA will be set up so this causes initial switch transmissions to use a delay similar to the backoff applied to retransmissions. Then, if several CPUs are all accessing the same server, congestion is less than if transmission were not delayed. [NEEDS CLARIFICATION.]

The *steal* bit of the AR, when set to one, causes augmentable write references to steal the location referenced. If the write succeeds, it sets the steal bit on the stolen location; that bit indicates that the location is stolen. Writing to an already stolen location is allowed. See also the Lightweight Steal register.

Power-on clears all defined bits of the Augmentation register to zero.

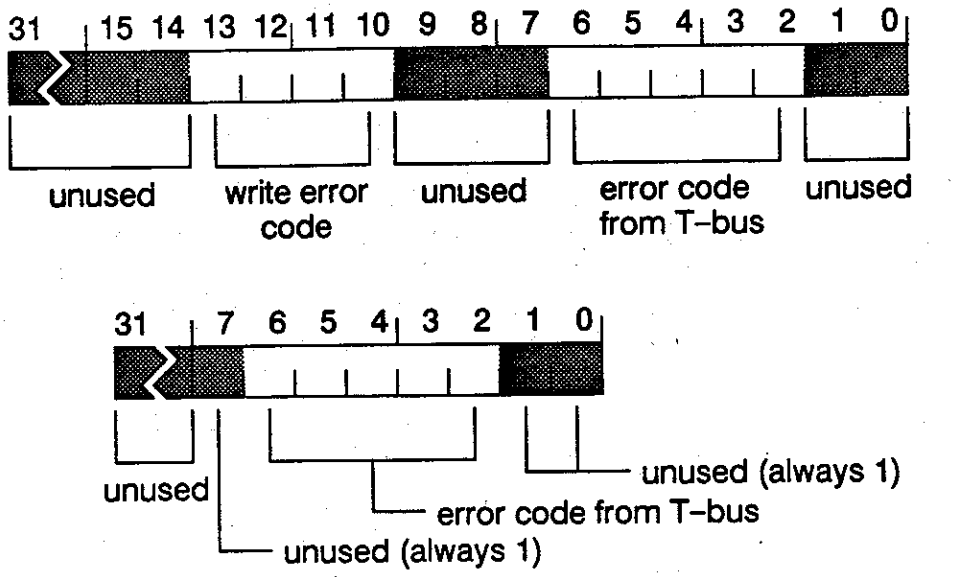


Figure 11-9. Bus Error Vector register.

FUNCTION Indicate the cause of a bus error.

ADDRESS 0xE078000C (~~Write Error register is at 0xE078001C~~)

ACCESS read only

DESCRIPTION An operation initiated by the CPU may result in an error detected by a T-bus slave. In the B2VME, these slaves are the memory, the SIGAs, the VMEbus master interface, and the status and configuration registers. The slave indicates the error by responding with an error reply (T-bus control bits T_RR<2..1> are 000) and, if possible, placing a code describing the error on the T-bus data wires. The CPU interface conveys this to the CMMU as an M-bus (memory bus) error, and the CMMU in turn conveys the error to the CPU, which receives it as an M-bus error exception. The error description code placed on the T-bus is captured in the Bus Error Vector register, for use by software handling the exception. Thus, the Bus Error Vector register contains the error code for the most recent bus error

to the CPU. If there has been no such error, the contents of the register are undefined.

If the error regards a write operation for which the CPU is currently driving data onto the T-bus (that is, a non-split cycle write), then the error description code cannot be placed on the T-bus at the same time. In this case, an error description code is captured in the Write Error register, and the Bus Error Vector register is loaded with a code that indicates the real error code is in the Write Error register. Thus, the Write Error register contains the error code for the most recent bus error to the CPU, *if* the Bus Error Vector register so indicates. Otherwise, the contents of the register are undefined.

NOTE: The Write Error register may be incorporated into the Bus Error Vector register, as bits 15..8 thereof.

The Bus Error Vector register error codes are:

The Write Error register error codes are:

```
 THIS IS OLD!!!
access to VMEbus master/T-bus slave
access to local [MEMORY???]
access to [REMOTE???] memory
VMEbus or memory is locked
attempt to burst read or write (not a memory error)
attempt to open or maintain a lock
attempt to set a masked bit (local only)
attempt to steal a location (not a memory error)
```

After power-on, the contents of the Bus Error Vector register is unspecified.

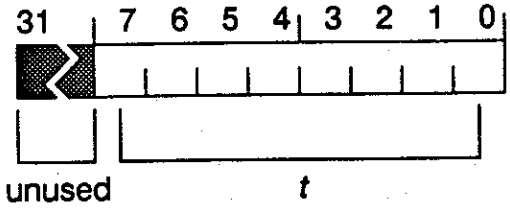


Figure 11–10. CPU Lock Timer register.

FUNCTION Limit how long the CPU may hold a lock.

ADDRESS 0xE0780000

ACCESS write only ~~(AR read-and-clear is at this address on read)~~ E0780010

DESCRIPTION When the CPU interface to the T–bus generates a locked transaction, the CPU Lock Timer is loaded from this register and begins counting. It stops when the CPU interface frees the lock. If the timer expires, the CPU is not explicitly notified, but a FREE_LOCKS cycle is automatically generated. The CPU's T–bus master interface, however, still believes the lock is held. If the CPU references the locked location without first freeing the lock, the reference will be a MAINTAIN access, to which the slave will return a "maintain present" error.

A CPU lock is timed out after 0xFF–*t* microseconds, where *t* is the contents of the CPU Lock Timer register; the contents of the timer itself are not accessible to software.

After power–on, the contents of the CPU Lock Timer register is unspecified.

?

Figure 11–11. Interleaver RAM registers.

FUNCTION Reduce contention for access to memory by distributing addresses among different function boards.

ADDRESS 0xE07A0000 — 0xE07A1FFF (shared with VMEbus Master Map RAM and write control registers???)

ACCESS read/write???

DESCRIPTION When the B2VME generates a remote reference (and therefore a switch request), system physical addresses from the T-bus are translated into [WHAT NAME??? modified??? interleaved???] addresses by the interleaver, using these registers. The ten??? bits T_AD<22..13>??? select one of 1024??? Interleaver RAM registers, whose bits then supply bits [???] of the [???] address as indicated. The [???] pin on the SIGA controls whether the SIGA uses the unmodified address straight from the T-bus or the address as modified by bits from the interleaver.

These registers are located in SIGA register space because the interleaver loader mechanism is used to load them.

After power–on, the contents of the Interleaver RAM registers is unspecified.

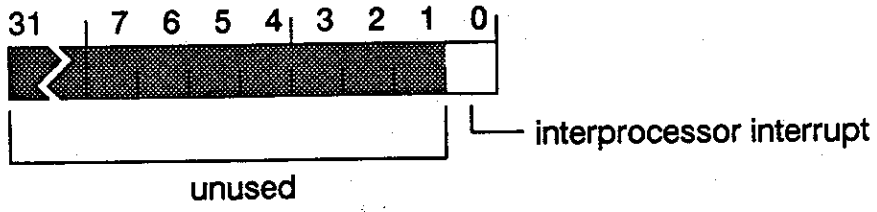


Figure 11–12. Interprocessor Interrupt register.

FUNCTION Interrupt the CPU.

ADDRESS 0x80002000 (in bank 0, so accessible remotely without an augmentation), subject to Interrupt Enable Mask register

ACCESS read/write

DESCRIPTION Setting the *interprocessor interrupt* bit in this register generates an interrupt to the CPU. This register can be written across the switch, so any other processor may interrupt this CPU, subject to the Interrupt Enable Mask register. It is expected that the operating system will impose further constraints on writing to this register. This bit is cleared either by writing a "0" into it, or by reading the Interrupt Source register.

Power–on clears the one defined bit of the Interprocessor Interrupt register to zero.

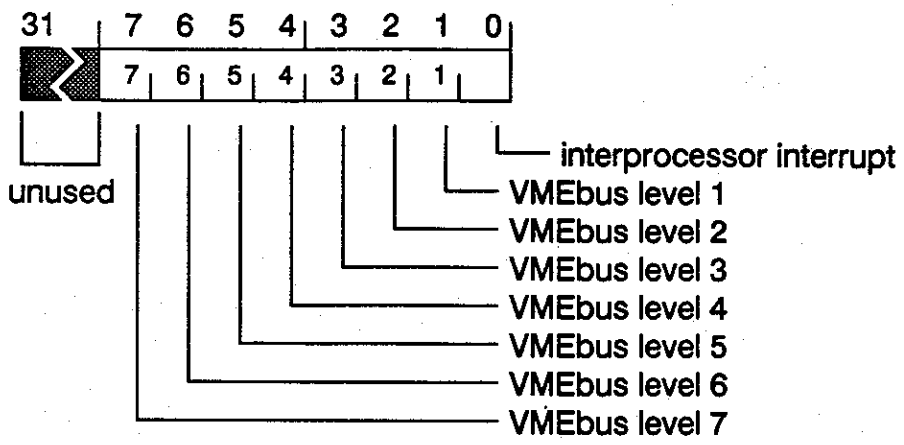


Figure 11–13. Interrupt Enable Mask register.

FUNCTION Enable interprocessor and VMEbus interrupts to the CPU.

ADDRESS 0xE0780014

ACCESS read/write

DESCRIPTION If a *VMEbus* mask bit in this register is "1", then the corresponding interrupt, if asserted, is recognized by the CPU. If a VMEbus mask bit is "0", then the CPU is insensitive to assertion of the VMEbus interrupt request on that level. The VMEbus interrupt request state is not latched in the B2VME; if the enable bit for a level is "1", the state seen by the CPU shows whether any VMEbus device (including the B2VME) is currently asserting a request on that level. VMEbus devices collectively assert and de-assert the VMEbus interrupt request level signals. The mask bits determine whether these signals can interrupt the B2VME CPU.

If the *interprocessor interrupt* mask bit is "1", writing to the Interprocessor Interrupt register is permitted. If the bit is "0", then an attempt to write to the Interprocessor Interrupt register returns a register load error instead. Clearing the mask bit in the Interrupt Enable Mask register to "0" does *not* clear or disable the Interprocessor Interrupt register, so if an interrupt was already pending it remains so.

Power–on clears all defined bits of the Interrupt Enable Mask register to zero, so the interrupts masked by this register are disabled.

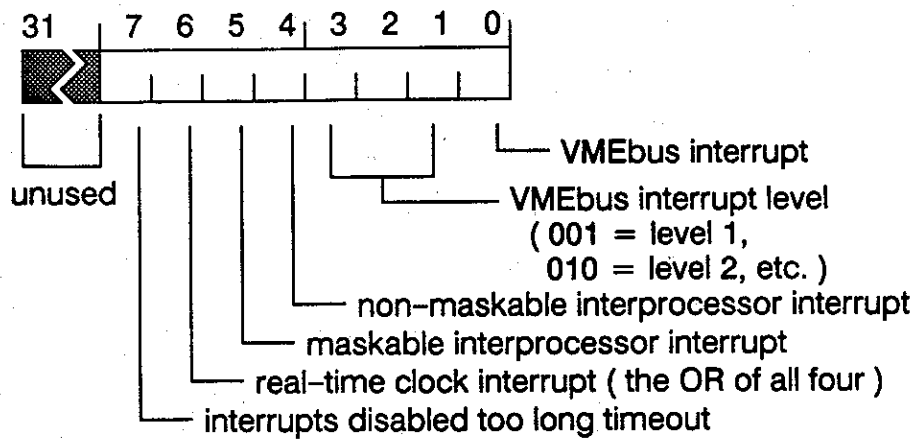


Figure 11-14. Interrupt Source register.

FUNCTION Indicate the cause of a CPU interrupt.

ADDRESS 0xE0780018

ACCESS read only (Interrupts Disabled Timer value is at this address on write)

DESCRIPTION There are fourteen sources of interrupts on the B2VME. The 88100 has only one interrupt level, so all interrupt sources are OR'ed together to generate a CPU interrupt. More than one type of interrupt may be asserted. Only the VMEbus interrupts are prioritized.

The *VMEbus interrupt* bit, if "1", indicates that at least one VMEbus interrupt request level is asserted. These levels are prioritized, and the *VMEbus interrupt level* bits indicate the level of the highest priority request currently asserted. If the VMEbus interrupt bit is "0", the contents of the VMEbus interrupt level bits is unspecified.

The *non-maskable interprocessor interrupt* bit, if "1", indicates that the interrupt bit of the Non-maskable Interprocessor Interrupt register has been set to "1".

The *maskable interprocessor interrupt* bit, if "1", indicates that the interrupt bit of the Interprocessor Interrupt register has been set to "1".

The *real-time clock interrupt* bit, if "1", indicates that one or both of the Time Of Next Interrupt (TONI) registers, in one or both of the SIGAs, has expired. Each SIGA has two real-time timers, but in normal operation, only one SIGA will be producing such interrupts.

The *interrupts disabled too long timeout* bit, if "1", indicates that the Interrupts Disabled Timer has expired.

PROGRAMMING ADVICE: Reading the Interrupt Source register clears both the non-maskable and the maskable interprocessor interrupts. The software should then save the state of the AR and PCR. The AR should then be cleared. The AR can be read and cleared in one access using the AR read-and-clear address. The interrupts disabled too long timeout will be cleared within 3 microseconds of clearing the AR. The VMEbus interrupt(s) get cleared by doing an IACK cycle on the VMEbus. The request at each asserted level must be given a separate IACK response. The real time clock interrupt is cleared by writing a value greater than the current time to the SIGA's TONI register that interrupted, or by changing that SIGA's configuration register to disable TONI interrupts.

Interrupts from the VMEbus and from the Interprocessor Interrupt register are maskable via the Interrupt Enable Mask register. Interrupts from the real-time clock are maskable via the configuration register in the SIGA. However, interrupts from the Non-maskable Interprocessor Interrupt register and from the Interrupts Disabled Timer are *not* maskable by the B2VME interrupt support circuitry; they will interrupt the CPU unless the CPU has turned off interrupts internally.

After power-on, the contents of the Interrupt Source register is unspecified.

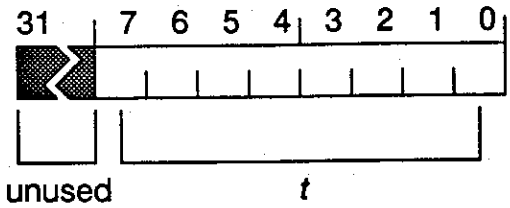


Figure 11–15. Interrupts Disabled Timer register.

FUNCTION Help guarantee maximum latency of interrupt servicing, by detecting protracted use of the *disable interrupts* augmentation.

ADDRESS 0xE0780018

ACCESS write only (Interrupt Source register is at this address on read)

DESCRIPTION When the Augmentation register *disable interrupts* bit is set to one, the Interrupts Disabled Timer is loaded from this register and begins counting. It stops when the disable interrupts bit is cleared to zero. If the timer expires, an "interrupts disabled too long" interrupt is generated. That interrupt is cleared by clearing the disable interrupts bit in the AR.

The interrupts disabled augmentation is timed out after 0xFF–*t* microseconds, where *t* is the contents of the Interrupts Disabled Timer register; the contents of the timer itself are not accessible to software.

After power–on, the contents of the Interrupts Disabled Timer register is unspecified.

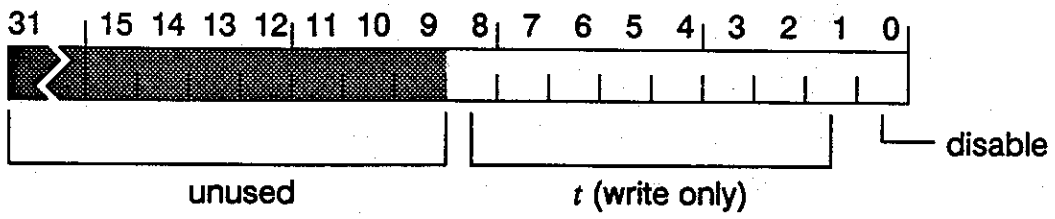


Figure 11–16. Interrupts Pending / Abort Retries register.

FUNCTION Help guarantee maximum latency of interrupt servicing, by detecting delay in switch transactions.

ADDRESS 0xE0760008

ACCESS write (all bits) / partial read (the *disable* bit only)

DESCRIPTION When a CPU interrupt request is asserted, a timer is loaded from the Interrupts Pending / Abort Retries register and begins counting. It stops when **all** interrupt requests are gone. If the timer expires, the signal SIGA_ABORT_RETRIES is asserted. This signal is an input to the SIGA, and, if asserted when the SIGA makes a retry attempt to establish a switch connection, the SIGA stops trying to make the connection and instead returns a bus error. If there is no switch access in progress that is re-trying to establish a connection, then the signal is ignored. The signal is de-asserted when **all** interrupt requests are gone.

A pending interrupt is timed out, if a remote reference is being attempted, after 0xFF–*t* microseconds, where *t* is the contents of the Interrupts Pending / Abort Retries register; the contents of the timer itself are not accessible to software. If bit 0 of the register is set to "1", the timer is disabled and the value of *t* is irrelevant; bit 0 must be "0" for the timer to operate.

After power-on, the contents of the Interrupts Pending / Abort Retries register is unspecified.

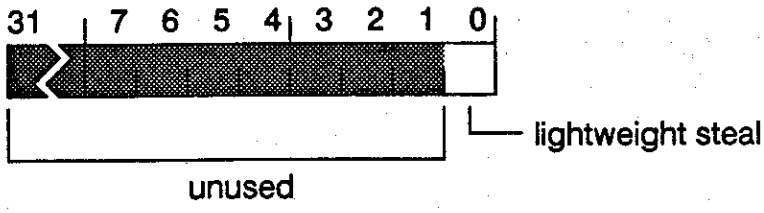


Figure 11-17. Lightweight Steal register.

FUNCTION Turn the next XMEM instruction into a steal.

ADDRESS 0xE0741000

ACCESS read/write

DESCRIPTION Setting this bit to "1" causes the next XMEM instruction to steal the location being exchanged. The lightweight steal cycle clears the Lightweight Steal register to zero. Thus, when you set the bit and then execute an XMEM, you steal the location and clear the Lightweight Steal register bit in one operation. This register is a part of the process state and must be saved along with the AR and PCR registers when switching context. See also the Augmentation register *steal* bit.

Power-on clears the one defined bit of the Lightweight Steal register to zero.

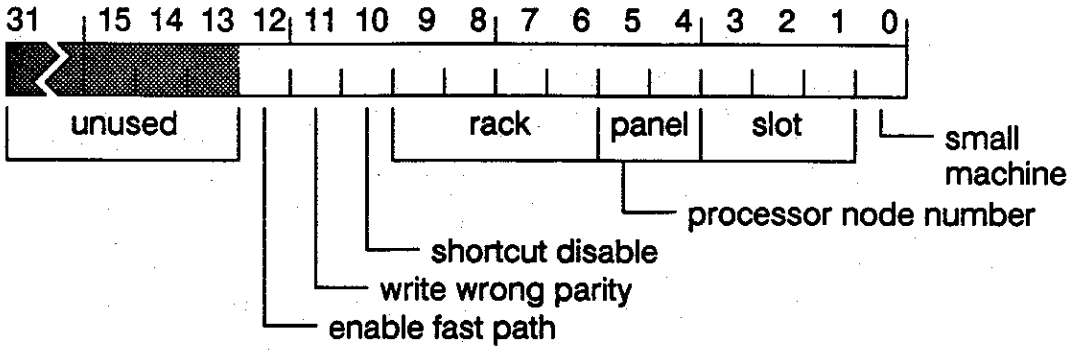


Figure 11–18. Machine Configuration Read register.

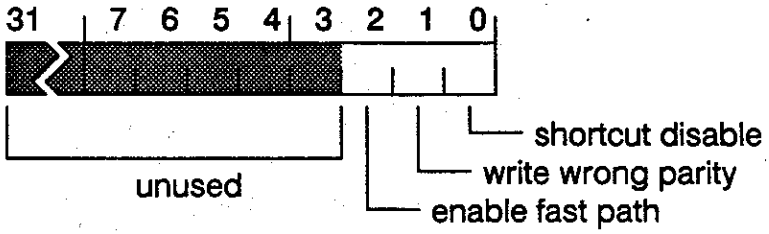


Figure 11–19. Machine Configuration Write register.

FUNCTION Indicate and control certain basic operating characteristics.

ADDRESS 0xE0760000

ACCESS read/write, with different bit assignments for read than for write

DESCRIPTION There is just one Machine Configuration register, and it is given two names to emphasize that its bit assignment is different when it is read than when it is written.

The *small machine* bit indicates whether the machine is "small" (two switch columns, and therefore less than 64 switch ports) or "large" (three switch columns, 64 to 512 switch ports). The bit is "1" in a small machine, "0" in a large machine. This bit is writable only by the TCS; to the CPU, it is read–only.

The *processor node number* field is the number of the switch port that this B2VME is plugged into. This field is read–only; the rack and panel sub–fields are set by DIP switches on the midplane, and the slot sub–field is hard wired in the midplane circuit etch.

The *shortcut disable* bit, when set to "1", causes remote memory references whose address corresponds to the local node to go over the switch. When the shortcut disable bit is cleared to "0", such references "short–cut" the switch and go directly to memory.

The *write wrong parity* bit, when set to "1", causes incorrect parity to be written, into local memory only, on any size of write, by any T–bus master (CPU, switch or VMEbus). This bit is intended for use in testing the memory. When this bit is cleared to "0", correct parity is written.

The *enable fast path* bit, when set to "1", enables the fast path used whenever the CPU makes a read request to local memory. When the enable fast path bit is cleared to "0", the fast path to memory is never used.

Power–on clears all defined bits of the Machine Configuration Write register to zero. In the Machine Configuration Read register, the processor node number field is not affected by power–on; the small machine bit is set appropriately by TCS power–up operations before system software executes on the node.

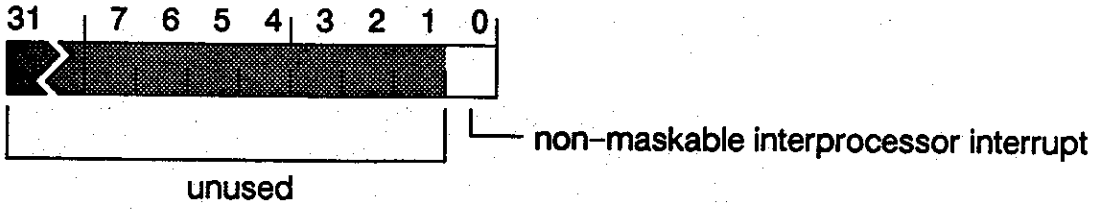


Figure 11-20. Non-maskable Interprocessor Interrupt register.

FUNCTION Interrupt the CPU, unless the CPU itself has interrupts disabled.

ADDRESS 0xE0760004

ACCESS read/write

DESCRIPTION Setting the *non-maskable interprocessor interrupt* bit of this register generates a non-maskable interrupt to the CPU. This interrupts the CPU unless the CPU has all interrupts disabled via its internal Processor Status register. The non-maskable interrupt should be used only when a fatal error has occurred. This bit is cleared either by writing a "0" into it, or by reading the Interrupt Source register.

Power-on clears the one defined bit of the Non-maskable Interprocessor Interrupt register to zero.

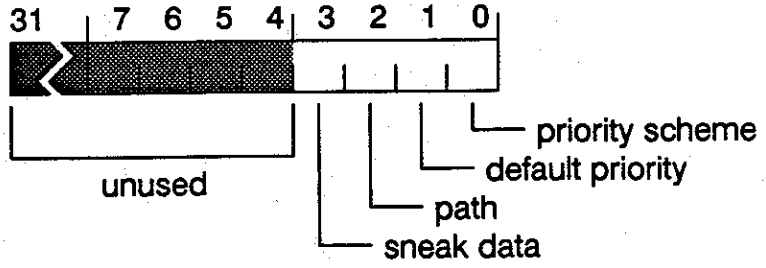


Figure 11–21. PCR Disable Mask register.

FUNCTION	Prohibit setting bits in the Process Configuration register.
ADDRESS	0xE0780010
ACCESS	read/write
DESCRIPTION	If an attempt is made to set a bit in the Process Configuration register that is disabled by its mask bit being set to one in the PCR Disable Mask register, then a register load error is returned. Power–on clears all defined bits of the PCR Disable Mask register to zero, so all bits of the PCR are initially writable.

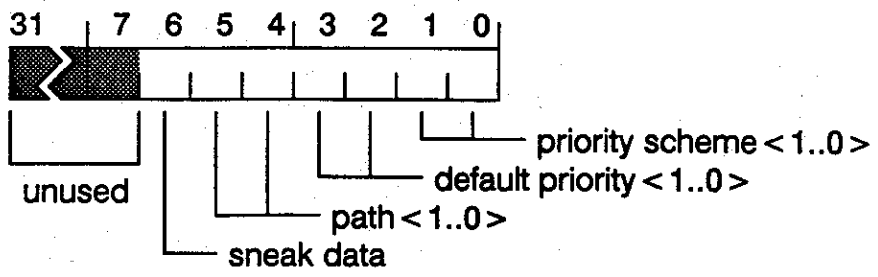


Figure 11–22. Process Configuration register (PCR).

FUNCTION Hold certain information associated with the software process currently running on the processor.

ADDRESS 0xE0740000

ACCESS read/write, subject to PCR Disable Mask register

DESCRIPTION The CPU interface uses the *priority scheme* bits to select one of four priority schemes, one of which is given by the PCR *default priority* bits. This selection determines the value of the T_PRIORITY < 1..0 > bits to place on the T-bus. The T_PRIORITY < 1..0 > bits are used only by the SIGA (and thence by SGAs), so this mechanism is irrelevant for references to local memory. The priority scheme selection is as follows:
if priority scheme < 1..0 > are...

0	0	then T_PRIORITY < 1..0 > = ???
0	1	then T_PRIORITY < 1..0 > = ???
1	0	then T_PRIORITY < 1..0 > = ???
1	1	then T_PRIORITY < 1..0 > = ???

Note: the priority scheme bits may be overridden by the *absolute priority* bits of the Augmentation register.

The *path* bits describe the access path being used, as follows:

0	0	illegal (causes a no-response bus error)
0	1	use SIGA B (or local, depending on Physical Address)
1	0	use SIGA A (or local, depending on Physical Address)
1	1	local access (not via either switch, regardless of Physical Address)

The *sneak data* bit contains the 33rd data bit of the most recent memory reference that was augmented with a sneak (that is, with the sneak bit of the AR set to "1"). This bit is writable by software, but normally is written only when restoring the entire register to return to a previous process context.

After power-on, the contents of the Process Configuration register is unspecified.

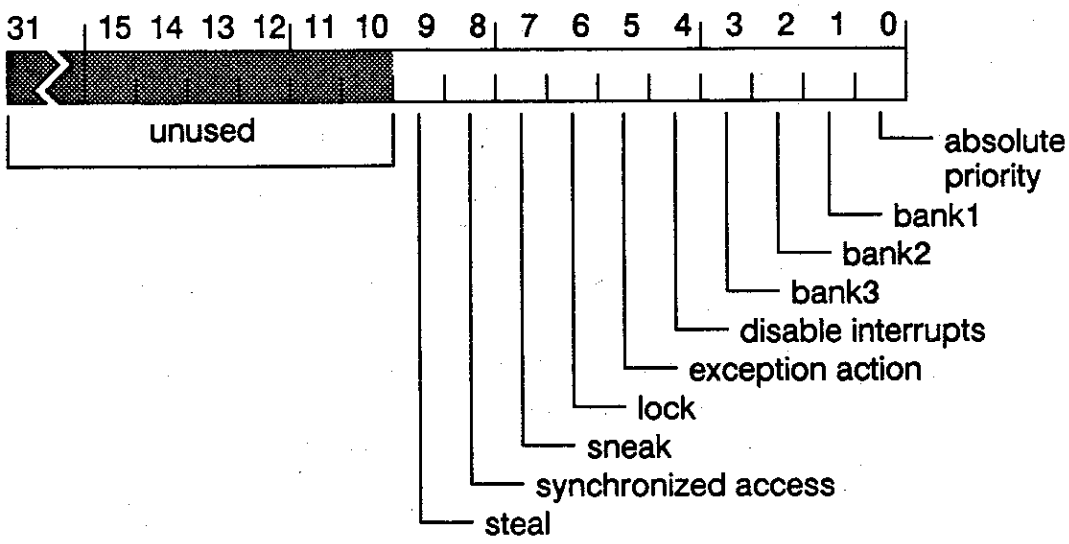


Figure 11-23. User Augmentation Enable Mask register.

FUNCTION Permit enabling certain augmentations via the AR block.

ADDRESS 0xE0780008

ACCESS read/write

DESCRIPTION The bits of the User Augmentation Enable Mask register control write access to the corresponding bits in the AR, but only when the AR is accessed as the AR block.

- If a given mask bit is "1", the controlled bit(s) of the AR will be set or cleared, as determined by bit(s) in the address used.
- If the mask bit is "0", the controlled AR bit(s) are not affected by attempts to change them. If the operation tries to set the masked AR bit to "1", a register load error results. Attempting to clear an AR bit to "0" does not generate an error, even if the controlling mask bit is "0".
- If the operation tries to set some AR bits that are masked and some that are not masked, then the bits that are not masked get set, the masked bits do not get set, and a register load error occurs. In this case, because some but not all of the attempted bits may have been set, the AR may contain an unintended value, and it is the responsibility of the exception handler to ensure that the AR contains an appropriate value.

The *absolute priority* bit masks both of the absolute priority bits of the AR.

The *bank1*, *bank2* and *bank3* bits mask the two bank bits of the AR, but not in a bit-for-bit sense. If bank1 is "1", then AR bits bank<1..0> may be set to 01; if bank1 is "0", attempts to set the AR bits to 01 result in a register load error. Similarly, bank2 being "1" allows setting AR bank<1..0> to

10, and bank3 controls setting the AR bits to 11. The AR bits bank<1..0> may always be set to 00; that setting is not controlled by a mask bit.

The *disable interrupts* bit masks the disable interrupts bit of the AR.

The *exception action* bit masks both of the exception action bits of the AR.

The *lock, sneak, synchronized access* and *steal* bits each mask the bit of the same name in the AR.

For the meaning of the bits that this register masks, see the Augmentation register.

After power–on, the contents of the User Augmentation Enable Mask register is unspecified.

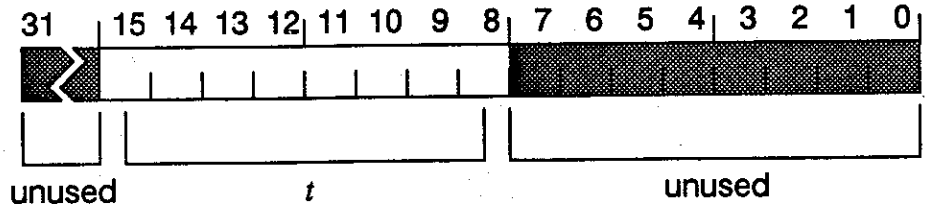


Figure 11-24. VMEbus Arbiter Timer register.

FUNCTION Detect when no VMEbus device responds to bus grant.

ADDRESS 0xE076020C

ACCESS write only

DESCRIPTION If the B2VME is VMEbus system controller, one of its duties as bus arbiter is to time out the *bus grant* signal. When the arbiter asserts bus grant, the VMEbus Arbiter Timer begins counting, and stops when a bus master acknowledges the grant by asserting the *bus busy* signal. If no device responds, expiration of this timer causes the arbiter to remove bus grant. This occurs entirely within the B2VME VMEbus interface, and is not explicitly visible to the B2VME CPU or to any other T-bus device. This timeout never occurs if all devices obey *The VMEbus Specification*; but timing out bus grant allows forward progress if a violation does occur.

The arbitration cycle is timed out after $4\times(0xFF-t)$ microseconds, where t is the contents of the VMEbus Arbiter Timer register. The timer is initialized from the register each time the B2VME asserts bus grant; the contents of the timer itself are not accessible to software.

After power-on, the contents of the VMEbus Arbiter Timer register are unspecified.

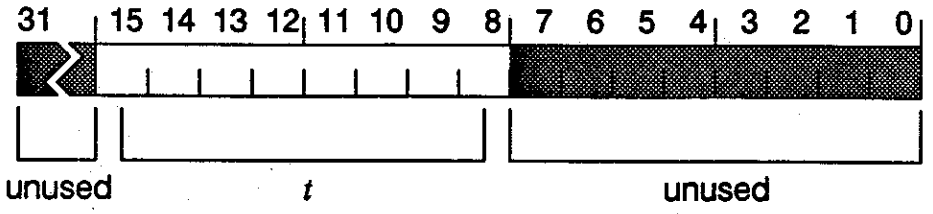


Figure 11–25. VMEbus B2VME Master Bus Timer register.

FUNCTION Detect lack of response from a slave to the B2VME as bus master.

ADDRESS 0xE0760200

ACCESS write only

DESCRIPTION When the B2VME is VMEbus master, it asserts the VMEbus signal *address strobe* and waits for a slave to respond with *DTACK* ("data acknowledge"). The VMEbus B2VME Master Bus Timer starts counting when the B2VME asserts address strobe, and stops when DTACK is received. If no response is received, expiration of this timer causes the B2VME's VMEbus master interface to remove address strobe, and its T-bus slave to assert a bus error on the T-bus. Without this timeout, access to a non-responding VMEbus device would be timed out only by the VMEbus system controller. The system controller's bus timer is typically set much longer than the VMEbus B2VME Master Bus Timer, so relying on the system timer would compromise responsiveness. The longer timeout would be especially serious for VMEbus access across the Butterfly II switch, where maximum switch latency would be affected.

Any VMEbus reference initiated by the B2VME is timed out after 0xFF–*t* microseconds, where *t* is the contents of the VMEbus B2VME Master Bus Timer register. The timer is initialized from the register each time the B2VME asserts address strobe; the contents of the timer itself are not accessible to software.

After power-on, the contents of the VMEbus B2VME Master Bus Timer register are unspecified.

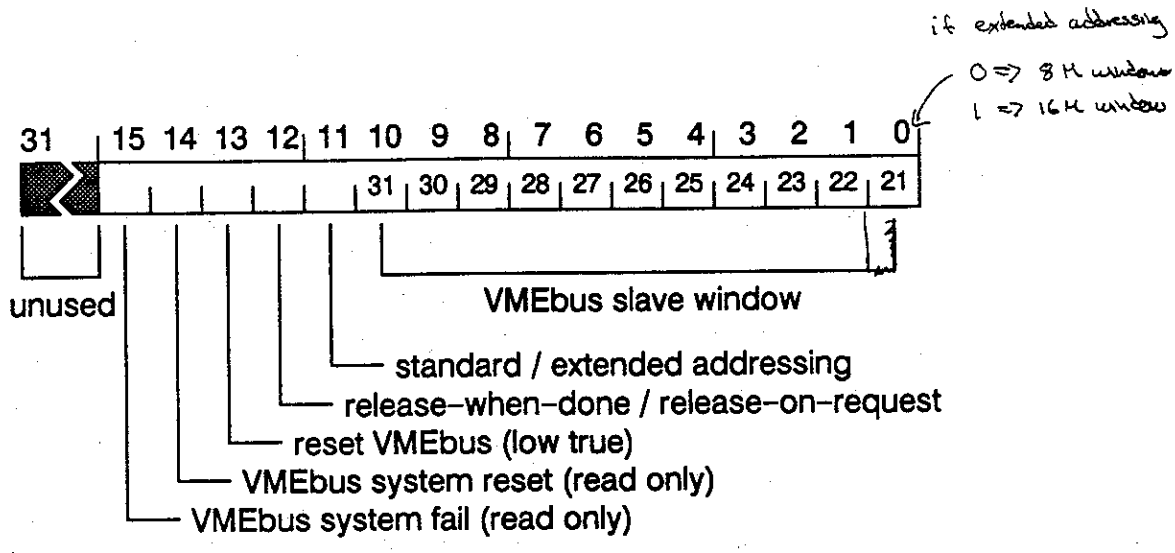


Figure 11-26. VMEbus Configuration register.

FUNCTION Control the configuration of the B2VME as a device on the VMEbus.

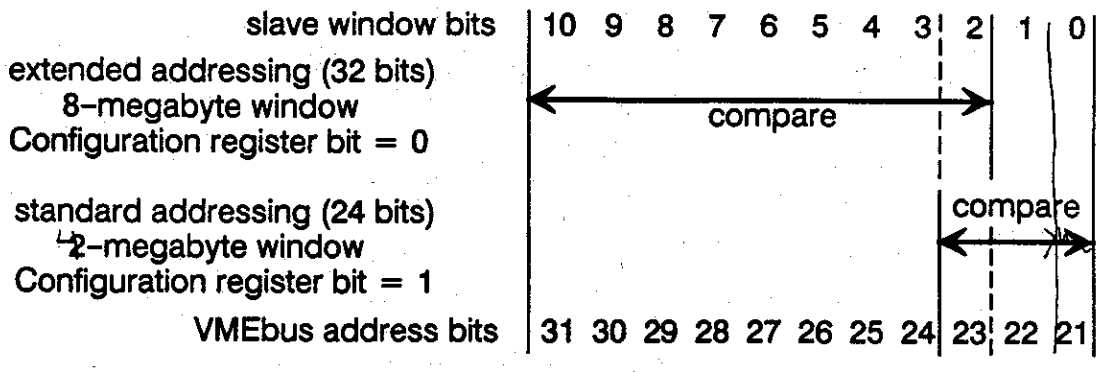
ADDRESS 0xE0760204

ACCESS read (all bits) / partial write (all but the *VMEbus system reset* and *fail* bits)

DESCRIPTION The VMEbus slave mapper compares the *VMEbus slave window* to the high bits of the address from the VMEbus, to determine whether the VMEbus is referencing Butterfly II memory and therefore the B2VME should respond. See below for the number of bits compared.

The *standard / extended addressing* bit determines the type of VMEbus addressing to which the B2VME responds as a VMEbus slave. This also determines how many bits of the VMEbus address are compared to the VMEbus slave window, and therefore the size of the window from VMEbus address space into Butterfly II address space. Note that the addressing used when the B2VME is VMEbus master is not determined by this bit, but rather by the vme_am<5..0> bits of the VMEbus Master Map RAM register.

value	addressing
0	extended (32-bit); compare vme_a<31..23> against VMEbus Configuration register <10..2> (nine bits); window is eight megabytes
1	standard (24-bit); compare vme_a<23..21> against VMEbus Configuration register <2..0> (~~three~~ bits); window is ~~two~~ megabytes



The *release-when-done / release-on-request* bit controls when the B2VME relinquishes VMEbus mastership. A "1" indicates release-when-done, a "0" release-on-request.

The *reset VMEbus* bit, when cleared to "0", generates a reset on the VMEbus as long as it remains "0". The bit must be set to "1" to de-assert the reset signal on the VMEbus. The low-true sense of this bit ensures that upon power-on, which clears the bit to zero, the VMEbus is held reset. Resetting the B2VME does not generate a VMEbus reset, nor does resetting the VMEbus reset the B2VME.

The *VMEbus system reset* and *VMEbus system fail* bits read as "1" if the VMEbus has been reset or suffered a system failure since this register was last read; otherwise, these bits are "0". Reading this register clears these bits to "0". The setting and clearing of these bits is controlled by devices on the VMEbus; see *The VMEbus Specification* for details.

After power-on, the contents of the VMEbus Configuration register are as follows. Bits 9..0 (most of the VMEbus slave window) are unspecified. Bits 13..10 are zero. Bit 14 will be "1" because the B2VME itself is holding the VMEbus reset. Bit 15 depends on the state of other VMEbus devices.

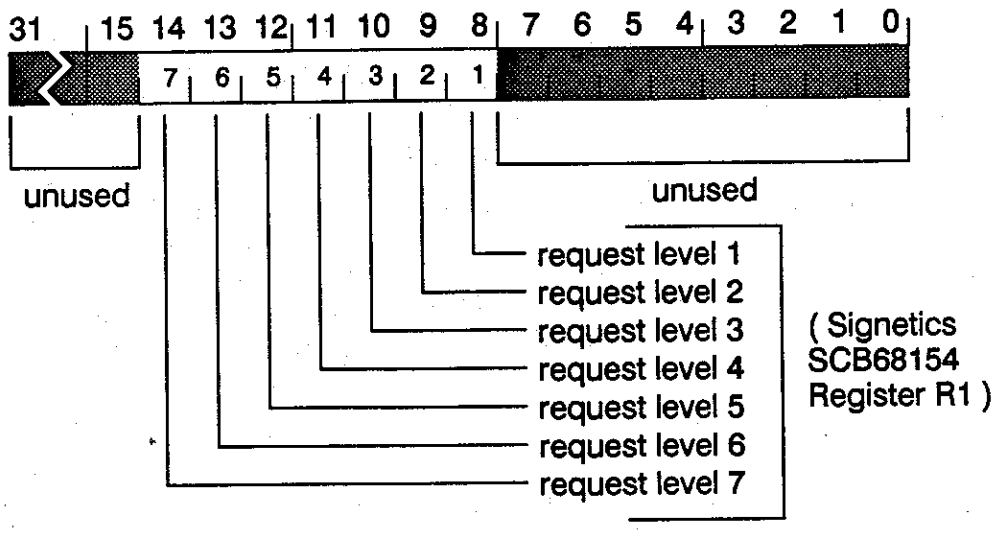


Figure 11-27. VMEbus Interrupt Request register.

FUNCTION Select the level(s) on which the B2VME generates a VMEbus interrupt request.

ADDRESS 0xE0760804

ACCESS read/write

DESCRIPTION Writing to this register generates an interrupt request(s) to the VMEbus, on each level whose corresponding bit in the data written is a "1". However, placing the request onto the VMEbus is under control of the *enable interrupts* bit in the VMEbus Interrupt Vector / Control register. If the B2VME is already requesting VMEbus interrupts on some levels, the levels requested by writing to this register are added to those already present, like an inclusive-OR function. The bits of this register are cleared individually by IACK cycles on the VMEbus, or all together by the *clear interrupts* bit of the VMEbus Interrupt Vector / Control register.

Power-on clears all defined bits of the VMEbus Interrupt Request register to zero. —OR???— After power-on, the contents of the VMEbus Interrupt Request register is unspecified.

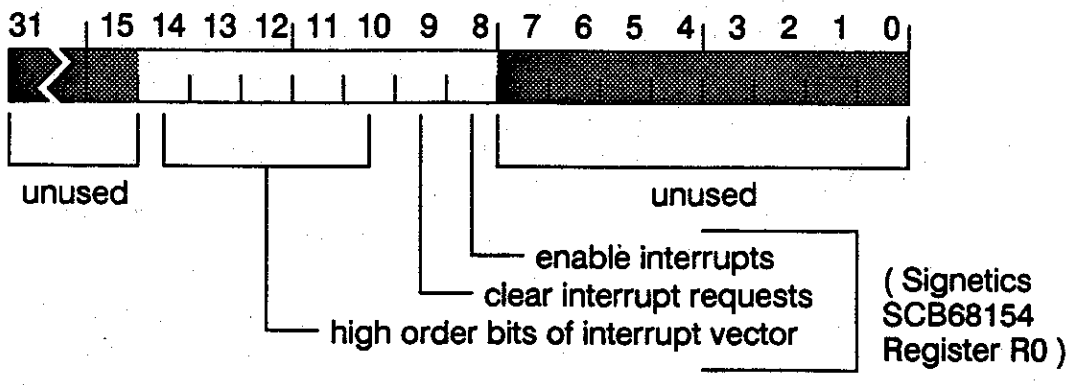


Figure 11-28. VMEbus Interrupt Vector / Control register.

FUNCTION Control interrupts the B2VME generates onto the VMEbus.

ADDRESS 0xE0760800

ACCESS read/write

DESCRIPTION The *enable interrupts* bit, when set to "1", allows the B2VME to generate interrupt requests onto the VMEbus.

The *clear interrupt requests* bit, when set to "1", forces off all VMEbus interrupt requests that the B2VME is asserting. This clears VMEbus Interrupt Request register.

The *high order bits of interrupt vector* field specifies the highest five bits of the interrupt vector that the B2VME will place on the VMEbus. This occurs when the B2VME has requested an interrupt and some VMEbus device has responded with IACK to our requested level.

Refer to the Signetics SCB68154 data sheet for further details on this register, and on the associated VMEbus Interrupt Request register.

Power-on clears all defined bits of the VMEbus Interrupt Vector / Control register to zero. —OR???— After power-on, the contents of the VMEbus Interrupt Vector / Control register is unspecified.

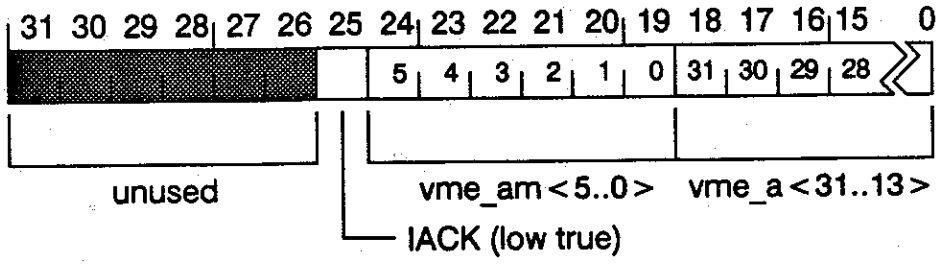


Figure 11-29. VMEbus Master Map RAM registers.

FUNCTION Map Butterfly II addresses into VMEbus addresses.

ADDRESS ~~0xE07A0000 to 0xE07A1FFF~~ (shared with interleaver)

ACCESS read/write

DESCRIPTION When the B2VME is VMEbus master, system physical addresses from the T-bus are translated into VMEbus addresses by the master mapper, using these registers. The nine bits T_AD<21..13> select one of 512 VMEbus Master Map RAM registers, whose bits then supply parts of the VMEbus address as indicated.

The *IACK* bit is used when the B2VME is responding as a VMEbus **interrupt handler** to an interrupt request generated by a VMEbus **interrupter** device. When an **interrupt handler** receives an interrupt, it must generate an interrupt acknowledge (IACK) cycle to get the interrupt vector from the **interrupter**. To generate the IACK cycle, the **interrupt handler** performs a halfword (16-bit) read from a specially-mapped address. The address is mapped by the VMEbus Master Map RAM registers, and the particular map register used has its IACK bit asserted (set to "0"). Bits 3..1 of the address are the interrupt level obtained from bits 3..1 of the Interrupt Source register. This causes the **interrupter** to respond with an interrupt vector, which is returned to the **interrupt handler** as the result of its halfword read.

These registers are located in SIGA register space because the interleaver loader mechanism is used to load them.

After power-on, the contents of the VMEbus Master Map RAM registers is unspecified.

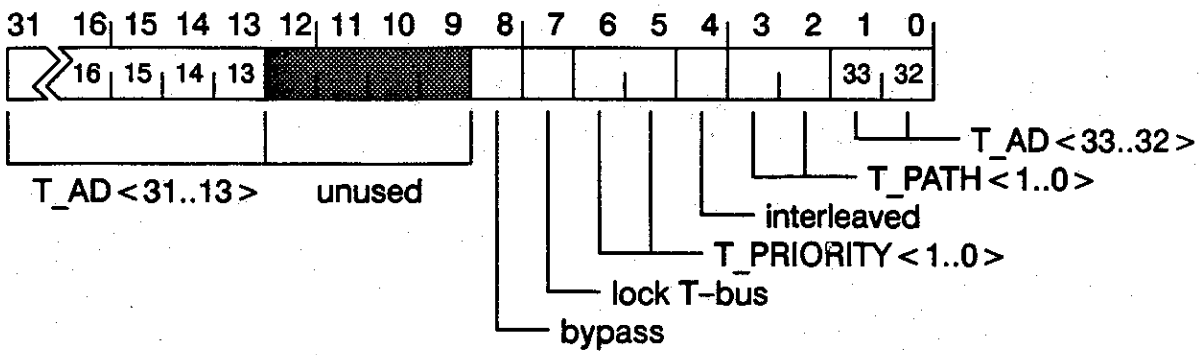


Figure 11–30. VMEbus Slave Map RAM registers.

FUNCTION Map VMEbus addresses into Butterfly II addresses.

ADDRESS 0xE0761000 to 0xE0761FFC
or E0720000 to E0723FFC

ACCESS read/write

DESCRIPTION When the B2VME is a VMEbus slave, responding to some other VMEbus device, system physical addresses from the VMEbus are translated into Butterfly II addresses by the slave mapper, using these registers. The ten bits vme_a<22..13> select one of 1024 VMEbus Slave Map RAM registers, whose bits then supply parts of the address and other data placed on the T-bus as indicated. The three bits named *bypass*, *lock T-bus* and *interleaved* describe the type of reference to make. Bypass tells the VMEbus slave (T-bus master) to make an operation that will bypass a lock; lock T-bus tells it to make a locked operation. The VMEbus slave encodes the type of reference into bits T_LOCKOP<1..0>. The interleaved bit tells the SIGA whether to take the high nine address bits from the T-bus (T_AD<33..25>) or from the interleaver (MOD<8..0>).

After power-on, the contents of the VMEbus Slave Map RAM registers is unspecified.

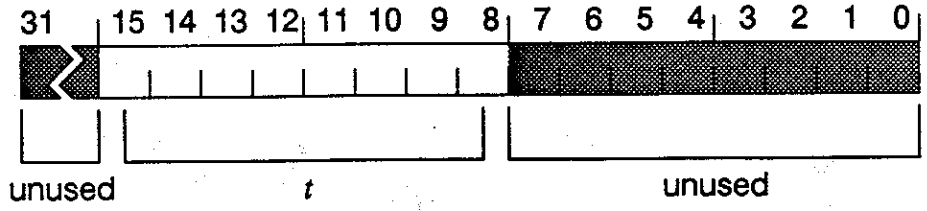


Figure 11-31. VMEbus System Bus Timer register.

FUNCTION Detect lack of response from a slave to any VMEbus master.

ADDRESS 0xE0760208

ACCESS write only

DESCRIPTION If the B2VME is VMEbus system controller, one of its duties is to time out the *address strobe* signal. When any VMEbus master asserts address strobe, the VMEbus System Bus Timer begins counting, and stops when a slave device responds with the signal *DTACK* ("data acknowledge"). If no slave responds, expiration of this timer causes the B2VME system controller hardware to assert the VMEbus signal *BERR* ("bus error"). This occurs entirely within the B2VME VMEbus interface, and is not explicitly visible to the B2VME CPU or to any other T-bus device. This situation may occur in a properly operating system. For example, on power-up, a program may poll the VMEbus to find available memory.

A VMEbus access is timed out after $4\times(0xFF-t)$ microseconds, where t is the contents of the VMEbus System Bus Timer register. If the B2VME is VMEbus system controller, the timer is initialized from the register each time address strobe is asserted by any VMEbus master; the contents of the timer itself are not accessible to software.

After power-on, the contents of the VMEbus System Bus Timer register is unspecified.

Switch Interface Gate Array (SIGA) Specification

A

A.1 Introduction

The SIGA is a gate array device which serves as the bidirectional interface between a Computational Node and the Switch network of the Butterfly II Parallel Processor. As such, the SIGA provides devices on each Computational Node with virtually transparent read and write data access to similar devices on physically remote nodes. The SIGA accomplishes this by accepting/presenting data via the standard interface that these devices support – namely the T-Bus – and then presenting/accepting this same data to the Butterfly Switch interface for transport.

This document will present both a detailed functional and operational description of the SIGA. It is intended to be used as a design guide for both hardware and software system engineers. This specification is necessarily limited in its scope and thus will touch upon other Butterfly II-related subjects only when it is necessary for completeness. Therefore, it is assumed that the reader of this document has a general knowledge of the concepts of the Butterfly II architecture and operation. The reference documents are as follows:

- T-Bus Specification (Ward Harriman)
- Switch Gate Array Design Specification (Ward Harriman)
- Butterfly II Level Converter Array Specification (Mike Sollins)
- Switch Protocol Specification (Ward Harriman)

Figure A-1. Reference documents.

A.2 Terminology

The following terms will be used throughout this document:

Byte Refers to an 8–bit quantity.

Anticipation A feature of the SIGA design that allows the SIGA to take advantage of certain parallel optimizations.

Downstream Node The node which services a switch transaction.

Drop–Lock When the Requestor negates Frame during a locked sequence, causing the Server to issue a FREE–LOCK.

Function Response A generic term for the various incarnations of a response to a function request from some downstream T–Bus slave to an upstream T–Bus slave. This includes the transformations that the response undergoes as it travels from the downstream T–Bus, downstream SIGA, Switch, upstream SIGA, and finally the upstream T–Bus (see Function Request).

Function Request A generic term for the various incarnations of a request from some upstream T–Bus master to a downstream T–Bus slave. This includes the transformations that the request undergoes as it travels through the upstream T–Bus, upstream SIGA, Switch, downstream SIGA, and finally the downstream T–Bus (see Function Response).

Final Locked message The same as a Locked message except that the Switch path is released by letting Frame = 0 for at least two Switch Intervals after the operation has been acknowledged.

Half–Word Refers to a 16–bit quantity (see Word).

Initial Locked message Occurs under the same circumstances as the Unlocked message except that the Switch path is held open once the operation has been acknowledged without errors.

Local Errors Errors which originate in the Requestor.

Logical Route Address A 9–bit Switch node address generated from either the Interleaver or the T–Bus. This address is then transformed, possibly by randomizing some of the bits, into the Physical Route Address.

Locked message A message which occurs when the Switch path was already locked and causes it continue to be locked after the operation has been acknowledged.

Message With the exception of Reject, a Message is the assertion of Frame (downstream message) or Reverse (upstream message) possibly with associated data on the data lines.

Message Acknowledgment Also known as M_ACK. This refers to the assertion of Reverse for at least two Switch Intervals during a function response. It indicates that

	the downstream Server has Acknowledged the receipt of a Function Request.
Message Header	The part of a downstream Switch message that carries routing information. That part is stripped–off by the Switch and thus never reaches the downstream Server. The message header for an upstream Switch message is null.
Message Body	The part of the downstream Switch message that carries the command, address, data, and checksum bytes.
Multi–Word Transfer	Refers to a read or write function request that involves more than one word (32 bit) of data.
Operational State	A SIGA initialization state which allows full operation of the SIGA.
Physical Route Address	The transformation of the Logical Route Address after some of its bits have been randomized. The Physical Route Address is placed into the downstream Message Header.
Quick–Drop	This is an optimization in the Requestor where the R_FRAME signal is negated as soon as possible after an R_REVERSE is received.
Quiescent State	A SIGA initialization state which allows partial operation of the SIGA.
Remote Errors	Errors which originate in the Server.
Reject	An assertion of Reverse for one Switch Interval. Indicates that a message was rejected at either a Server or an SGA.
Sequence	A function request followed by a function response.
Split–Cycle	A T–Bus Read transaction where the Master releases the bus while the Slave is completing the transaction.
Switch Interval	The 25 ns period in which Switch data is propagated.
Switch Modulus	The number of ports that a basic switching element can handle. That number is currently eight.
Transaction	Another word for a Sequence.
Unlocked Message	Occurs when the Switch path had previously been "torn–down". This occurs whenever Frame was "0" for at least two Switch Intervals. Once the operation has been acknowledged, the path is torn–down again.
Upstream Node	The node which initiates a switch transaction.
Valid Message	A downstream message which carries a read or write request and does not violate switch protocol.

Word Refers to a 32-bit quantity (see Half-Word).

A.3 Document Standards

The following describes some of the standard syntax and expressions used in this document.

A.3.1 Register Definition Syntax

A typical register definition is shown in Figure A-2. Referring to Figure A-2, the "-" in any bit indicates that this bit is a "don't care" on a write and indeterminate on a read. If "-" totally fills a field of eight bits, that field should NEVER be written to but of course, can be read from. The entire register may be referred to in any one of the following ways:

(1) Protocol_Timer_Config<15..0>
(2) Protocol_Timer_Config
(3) PTC<15..0>
(4) PTC

The sub-fields, shown in Figure A-2 within "[]", can be referred to in various ways. For instance, the "Cnt" subfield could be referred to as:

(1) Protocol_Timer_Config<15..12>
(2) Protocol_Timer_Config.Cnt<3..0>
(3) Protocol_Timer_Config.Cnt
(4) PTC.Cnt

```
Register: Protocol_Timer_Config<15..0>

15                    0
|                     |
3..0   3..0   7......0
CCCC   PPPP   --NNNNNN
[Cnt]  [Pre]  [Con]
```

Figure A-2. Register syntax definition.

A.3.2 Logical Operators

Figure A-3 shows the standard operators used in this document.

OPERATOR	FUNCTION
&	logical "and"
#	logical "or"
$	logical "exor"
!	logical "not"
!$	logical "exnor"
|	concatenate

Figure A–3. Example – logical operators.

A.3.3 Timing Diagram Symbols

Timing diagrams use ASCII characters to represent signal states. Figure A–4 illustrates some of those symbols and their associated meanings. In addition, if no clock signal is present in a timing diagram, it is assumed that each character column represents an active transition of the appropriate clock.

SYMBOL	MEANING
H	logical "1"
_	logical "0"
.....	continue previous state
?????	state unknown and unimportant

Figure A–4. Example – signal symbols.

A.4 Functional Overview

The following describes the basic functionality of the SIGA at a conceptual level.

A.4.1 Functional Unit Description

The SIGA is composed of four basic elements, the Requestor, Server, Control Net Interface and the Config/Status Unit. Although these are physically colocated and share some common logic and control, they are functionally independent units and will be described separately.

A.4.1.1 Requestor

The Requestor is a T-Bus slave device which transparently couples physically remote T-Bus slave devices to the local T-Bus by interacting with both the Switch and the downstream Server. The Requestor appears to the current T-Bus master as a segment of memory which is out of the range of physical memory at the local node. Signals on

the T-Bus alert the Requestor that the current access is for a remote location and the Requestor then initiates the switch transaction to comply with the master's read or write request.

Since the transaction is not completed immediately (indicated by the Requestor with a PROMISE response), the requesting T-Bus master follows the T-Bus protocol and releases the bus so that other devices may use it. The Requestor eventually regains control of the T-Bus, alerts the requesting master that the read or write operation has been completed, and returns data or an error indication. If the current sequence is locked, as requested by the T-Bus master, and no errors are encountered, the Requestor holds open the Switch path for the next transaction rather than rearbitrating for a new Switch path. Any errors that may have occurred during this operation are signaled to the T-Bus Master through the ERROR response.

A.4.1.2 Server

The Server acts as a master on the local T-Bus of the downstream node and services requests from the upstream node's Requestor. When a new valid message enters the Server from the Switch, the Server obtains the local T-Bus; locks the T-Bus slave, if desired; performs the read or write operation; and then returns the data and/or error byte to the Upstream Node's Requestor. The Server can also initiate other special operations independently of receiving a new Switch message. This operation, known as drop-locks, is described elsewhere in this document.

A.4.1.3 TCS Control Unit

The basic purpose of the TCS Control Unit (TCU) is to give the serial interface of the TCS Control Slave Processor access to the T-Bus interface - in essence, to act as a protocol converter. A secondary function is to allow the TCS Slave Processor DIRECT access to some of the internal functions of the SIGA, rather than forcing it to access via the T-Bus interface. This is useful for fault-tolerance and "out-of-band" functions such as bootstrapping.

A.4.1.4 Configuration/Status Unit

The Config/Status Unit (CSU), acting as a T-Bus slave, allows read/write access to all programmable parameters of the Requestor, Server and TCS Control Unit. The CSU also provides convenient access to the internal state of certain nodes for testability.

A.4.2 System Operation

From a high-level view, the SIGA is one link in the chain of devices that allows a T-Bus device to fulfill a function request with a function response. The role of the SIGA in fulfilling both function requests and responses is now described.

A.4.2.1 Function Requests

A local T-Bus master in the upstream node, usually the CPU, initiates the sequence by placing an address on the T-Bus, which is detected by the SIGA Requestor as a remote access request. During the T-Bus request phase, the SIGA stores the address, produces and stores the bid, and command bytes. It then initiates the downstream message at the Switch interface by asserting Frame and placing the bid symbols on the Switch data lines. Normally, this message transmission is initiated by the SIGA immediately upon receiving the address from the T-Bus, but it can be programmed to start later. On a write, the SIGA loads its data registers during the response phase of the T-Bus cycle. All operations are split-cycle and thus the Server will release the bus while it processes the transaction request.

If there is no Switch contention, the assembled message continues to be transmitted and is ultimately appended with a checksum derived from the message data bytes. If there is Switch contention, a Reject is generated by the Switch and eventually makes its way upstream to the Requestor via the Reverse line. When this happens, the Requestor negates Frame, waits for a predetermined amount of time and then retries the message by asserting Frame and sending the message components stored from the first attempt.

Sometime after the beginning of the message reaches the Server at the downstream node (i.e., it is not Rejected by the Switch), that Server begins arbitration for its local bus to complete the transaction. If the device on the downstream node is locked to a remote bus master other than the Server, the Server issues a Reject which propagates upstream and is eventually detected at the upstream Requestor. This Reject is treated exactly the same by the Requestor as a Reject from the Switch. Note that this is the ONLY instance in which the Server will issue a Switch Reject - an Initial Message.

Assuming that the Requestor receives neither a Switch Reject nor a Server Reject, it deasserts Frame for one switch interval while it sends the checksum byte. Within the checksum byte, the "forward" bit is reset. This event would normally cause the forward drivers of the SGA's to turn off after they send the checksum byte. However, the current implementation of the SGA ignores this bit and turns-on its forward drivers in response to the Frame profile. The Requestor then awaits a response from the Server. Note that the forward bit is not used by the current SGA's.

In the meanwhile, the downstream Server begins processing the request by arbitrating for the local T-Bus. Assuming that the target downstream bus slave was not locked to a downstream master other than the Server, the Server obtains the local bus and possibly opens the local memory lock. The Server will open the lock only if this action was requested in the downstream message. This would occur if the master on the upstream node's local bus requested an OPEN lock when it initiated a transaction through it's associated Requestor.

Once the downstream Server obtains the local bus, it makes a function request to perform the appropriate read or write operation. The only exception to this is when the Server detects a checksum error in the downstream message. If this occurs, instead of making a request, the Server releases control of the T-Bus, creating a "dead" bus cycle

and thereby aborting the transaction. This action on an aborted transaction should eliminate any unwanted side-effects if the switch message is corrupted.

A.4.2.2 Function Responses

Assuming that a read transaction was requested, the downstream Server completes the read as a normal local T-Bus master. As soon as the read data is obtained by the Server, a message is returned to the upstream Requestor. This happens (over the same data wires which the downstream message was sent) by asserting Reverse and applying data to the Switch data lines. The upstream message contains the read data, and possible error data; a checksum; and a message acknowledgment, or M_ACK which is implicit in the assertion of Reverse for at least two Switch intervals. If a write transaction was requested, the Server writes the data to the address specified in the downstream message and sends back an M_ACK with an error byte data and checksum after the data has been accepted by the slave on the local T-Bus. In short, a read returns data/errors and an acknowledgment whereas a write only returns possible errors and an acknowledgment.

In the case of a read transaction, the upstream Requestor detects the M_ACK and alerts the local split-cycle master which initiated the request that the requested data has been returned. That master then completes the operation by retrieving the data. In the case of a write transaction, the Requestor also alerts the initiating local bus master that the write was completed but returns only error information.

In the absence of errors, the Requestor will continue to hold the Switch path open by asserting Frame only if the sequence was initiated with an OPEN. If that master decides to release the lock, the Requestor will tear-down the switch connection by negating Frame and will enter its unlocked idle state. This is the state that it was in at the beginning of this discussion of function requests. If the upstream bus master does not release the lock, it may initiate another read or write transaction. This and subsequent transactions are referred to as locked transactions. Outside of errors, locked transactions end only when the upstream T-Bus master which OPENed, MAINTAINed or BYPASSed the SIGA Requestor lock decides to release that lock with a FREE-LOCKS command.

Subsequent message transactions in a locked sequence differ from the initial transactions described above in three major ways. First, locked messages do not contain any bids because the path has already been established. Second, the Switch will never issue a Reject because the path has already been established and is being reserved for the Requestor. And third, the downstream Server will never issue a Reject because it will already have exclusive use of the local memory lock. Aside from these exceptions, subsequent locked transactions occur in exactly the same manner as unlocked transactions. As mentioned previously, the upstream T-Bus master owning the SIGA Requestor MUST release that lock explicitly with a FREE-LOCKS.

A.4.3 Basic Message Formats

Message formats differ mainly with the type of function request; read or write. Within a read or write message, the downstream and upstream messages corresponding to a function request and response also differ.

A.4.3.1 Read Messages

Read message formats differ mainly depending on whether or not they are downstream or upstream messages.

A.4.3.1.1 Downstream

Downstream Read messages are differentiated partly because of their data format and partly because of the state of Frame at the beginning and end of the message. The formats for three possible SIGA Requestor read operations are considered:

1. An Unlocked Read occurs when the Switch path had previously been "torn-down". This occurs whenever Frame was "0" for at least two Switch Intervals. Once the operation has been acknowledged, the path is torn-down again.
2. An Initial Locked Read occurs under the same circumstances as the Unlocked Read except that the Switch path is held open once the operation has been acknowledged.
3. A Locked Read is a read which occurs when the Switch path was already locked and it continues to be locked after the operation has been acknowledged.

In all cases, the Requestor waits for a Message Acknowledgment (M_ACK) from the downstream Server before completing the message. Figure A-5 illustrates the three read message types for a two column switch. In this figure, the "d" field indicates the direction of the LCON drivers which interface data with the LCON. When d = "P" (Output), the Requestor is sourcing data to the Requestor/LCON interface. When d = "I" (Input), the LCON drivers are sourcing data to the Requestor/LCON interface. The "f" field is the state of the Frame bit. Data is MSB at the left of the field.

```
Unlocked          Initial           Locked
Read              Locked Read       Read
============      ============      ============
d f   data        d f   data        d f   data
- - --------      - - --------      - - --------
P 0 xxxxxxxx      P 0 xxxxxxxx      P 1 xxxxxxxx
P 0 xxxxxxxx      P 0 xxxxxxxx      P 0 xxxxxxxx
P 1 -bid1---      P 1 -bid1---      P 1 -cmd----
P 1 -bid2---      P 1 -bid2---      P 1 -addr1--
P 1 -cmd----      P 1 -cmd----      P 1 -addr2--
P 1 -addr1--      P 1 -addr1--      P 1 -addr3--
P 1 -addr2--      P 1 -addr2--      P 0 -check--
P 1 -addr3--      P 1 -addr3--      I 1 00000000
P 0 -check--      P 0 -check--      I 1 00000000
I 1 xxxxxxxx      I 1 xxxxxxxx            "
I 1 xxxxxxxx            "             M_ACK
      "             M_ACK         and read data
   M_ACK        and read data            "
and read data           "          I 1 xxxxxxxx
      "          I 1 xxxxxxxx      P 1 xxxxxxxx
I 0 xxxxxxxx     P 1 xxxxxxxx
P 0 xxxxxxxx
```

Figure A-5. Read switch message format - downstream.

A.4.3.1.2 Upstream

When a downstream read message has been received and processed by a Server, an upstream message is returned to the initiating Requestor based on the operation requested. Under normal conditions, the Upstream Message is composed of two parts: the returned data (with checksum) and the M_ACK (Message Acknowledge). The returned data is the contents of the remote memory location read, which can be 1, 2 or 4 words in length. With the exception of rare error conditions, the actual message data field is almost always a multiple of four.

Figure A-6 illustrates the upstream message. The "r" field is the Reverse signal. Data is MSB at left of the field.

```
1-word, 4-byte Read
==========
r   data
- --------
0 xxxxxxxx
1 -data a-
1 -data b-
1 -data c-
1 -data d-
1 -check--
0 xxxxxxxx
```

Figure A-6. Read switch message format - upstream.

A.4.3.2 Write Messages

Write message formats differ mainly depending on whether or not they are downstream or upstream messages.

A.4.3.2.1 Downstream

Downstream Write messages are differentiated partly because of their data format and partly because of the state of Frame at the beginning and end of the message. The formats for three possible SIGA Requestor write operations are considered: In all cases, the Requestor waits for a Message Acknowledgment (M_ACK) from the downstream Server before completing the message. Figure A–7 illustrates the three write message types for a two column switch. In the figure, The "d" field is the direction of the LCON drivers which interface data with the SGA. When d = I, the Requestor is sourcing data to the Requestor/LCON interface. When d = P, the LCON drivers are sourcing data to the Requestor/LCON interface. The "f" field is the state of the Frame bit. Data is MSB at left of the field.

1. An Unlocked Write occurs when the Switch path had previously been "torn–down" by the fact that Frame was "0" for at least two Switch Intervals. Once the operation has been acknowledged, the path is torn–down again.
2. An Initial Locked Write occurs under the same circumstances as the Unlocked Write except that the Switch path is held open once the operation has been acknowledged.
3. A Locked Write is a write which occurs when the Switch path was already locked and it continues to be locked after the operation has been acknowledged.

```
Unlocked          Initial           Locked
Write             Locked.Write      Write
============      ============      ============
d f   data        d f   data        d f   data
- - --------      - - --------      - - --------
I 0 xxxxxxxx      I 0 xxxxxxxx      P 1 xxxxxxxx
I 0 xxxxxxxx      I 0 xxxxxxxx      P 0 xxxxxxxx
P 1 -bid1---      P 1 -bid1---      P 1 -cmd----
P 1 -bid2---      P 1 -bid2---      P 1 -addr1--
P 1 -cmd----      P 1 -cmd----      P 1 -addr2--
P 1 -addr1--      P 1 -addr1--      P 1 -addr3--
P 1 -addr2--      P 1 -addr2--      P 1 -data a-
P 1 -addr3--      P 1 -addr3--      P 1 -data b-
P 1 -data a--     P 1 -data a-      P 1 -data c-
P 1 -data b-      P 1 -data b-      P 1 -data d-
P 1 -data c-      P 1 -data c-      P 0 -check--
P 1 -data d-      P 1 -data d-      I 1 xxxxxxxx
P 0 -check--      P 0 -check--      I 1 xxxxxxxx
I 1 xxxxxxxx      I 1 xxxxxxxx            "
I 1 xxxxxxxx            "               M_ACK
      "                 "                 "
    M_ACK             M_ACK             I 1 xxxxxxxx
      "                 "               P 1 xxxxxxxx
I 0 xxxxxxxx      I 1 xxxxxxxx
P 0 xxxxxxxx      P 1 xxxxxxxx
```

Figure A-7. Write switch message format – downstream.

A.4.3.2.2 Upstream

When a downstream write message has been received and processed by a Server, an upstream message is returned to the initiating Requestor based on the operation requested. Under some conditions, the Server will not act on the downstream message and will instead send a Reject back to the Requestor. Under normal conditions however, upstream messages contain an M_ACK, an error byte (normally all 0's) and a checksum.

The following illustrates the only possible return message for a write. The "r" field is the Reverse signal. Data is MSB at left of field.

```
Any Write
==========
r   data
- --------
0 xxxxxxxx
1 -error--
1 -check--
0 xxxxxxxx
0 xxxxxxxx
```

Figure A-8. Write switch message format – upstream.

A.5 Detailed Functional Description

The Requestor, Server, TCU and Configuration/Status Unit are now described in detail.

A.5.1 Requestor

The Requestor is described from the point of view of its overall operation and its two major interfaces: the T-Bus interface and the Switch Interface.

A.5.1.1 Operation

The operation of the Requestor is described by discussing its major functions.

A.5.1.1.1 Overview

The Requestor is a local T-Bus slave which creates a logical coupling to a physically remote T-Bus slave via the Switch. The Requestor acts as the "initiator" of this coupling on the Switch and thus can be thought of as a "slave" on the T-Bus but a "master" to the Switch. Referring to Figure A-9, the Requestor contains three major functional units: Bus Interface Unit (BIU), Switch Tx Unit (STU), and the Switch Rx Unit (SRU). The BIU is clocked by the T-Bus clock and both the STU and SRU are clocked by the Requestor Switch clock (R_CLK). Interfacing of control signals between these units is accomplished with handshake synchronizers, as shown. The BIU handles all of the T-Bus transactions of the Requestor. The STU translates function requests that it receives from the BIU into Switch transactions. The SRU receives reply messages from the Switch and passes their status, in the form of a status code, back to the STU and their data back to the BIU. The STU serves as the single interface for control information between the T-Bus side and Switch side of the Requestor and therefore control information in either direction must pass through the STU. This is done to reduce the number of control interfaces that the BIU must deal with.

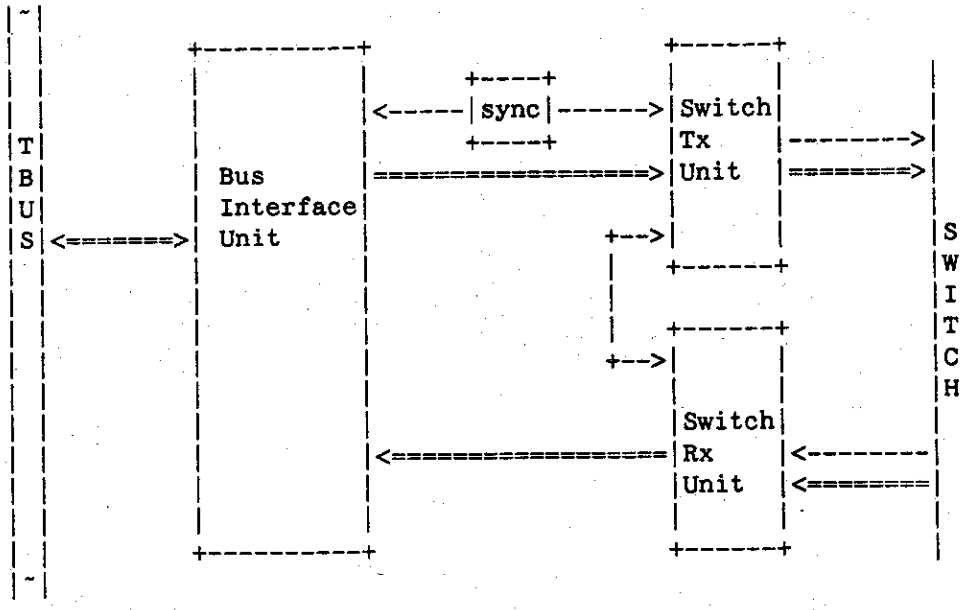


Figure A-9. Requestor block diagram.

The BIU/STU interface is a streamlined request/response type interface where for each BIU request there is an STU response. The BIU presents an encoded function request to the STU and sets an "execute" flag. When the STU is done operating on that request, it sets a "done" flag and returns a status code and data to the BIU. Both the BIU and STU are responsible for handling their own functions independently and they have very little real-time knowledge of each other's state. This approach simplifies the Requestor design and carries the request/response philosophy throughout the system.

The BIU has four major responsibilities: (1) screen T-Bus requests for correctness; (2) transfer screened T-Bus requests to the STU if a Switch transaction is indicated by that T-Bus request; (3) receive replies from the STU; and (4) pass replies, including any errors, as responses to the T-Bus. The BIU acts as a T-Bus slave which is always in split-cycle mode. In other words, it NEVER responds immediately to a function request from a T-Bus master except when a request error is detected. Outside of those exceptions, the BIU always responds with a PROMISE to T-Bus requests.

The BIU screens T-Bus requests for both T-Bus protocol violations and illegal function requests. Without exception, these conditions will prevent the BIU from ever activating the STU to complete an initial function request. The BIU can also initiate certain function requests to the STU independently of T-Bus requests. An example of this is the drop-lock function which may under certain conditions be initiated by the BIU rather than the T-Bus.

The STU acts on a function request from the BIU and initiates the Switch transaction to carry out that request. The STU also is responsible for assembling and transmitting the data in an outgoing message. It also handles things such as the message start/retry and priority promotion algorithms and deals with various protocol timeout violations.

The SRU is fairly simple in function. It detects the return message of a function request initiated by the STU, verifies the checksum and alerts the STU of the incoming message and the checksum status. The SRU also detects Switch Rejects.

A.5.1.1.2 RTC and related functions

The Real Time Clock, besides being useful as a system timekeeper, is central to the operation of much of the Requestor. It is used to directly control the functions of the Time_Of_Next_Interrupt and the Priority_Time_Slot mechanisms. These mechanisms are described in this section. The RTC is also used, in a less direct manner, to control the Protocol Timers. Protocol timers are discussed elsewhere in this document.

A.5.1.1.2.1 Real Time Clock and Prescaler

The RTC is basically a large (32 bits) counter which is updated every one microsecond from a divided-down version of the Switch clock. Since the frequency of the Switch may vary in different applications, the Real Time Clock uses a programmable prescaler to divide the Switch frequency down to a one microsecond time base. A functional diagram of the Real Time Clock is shown in Figure A-10.

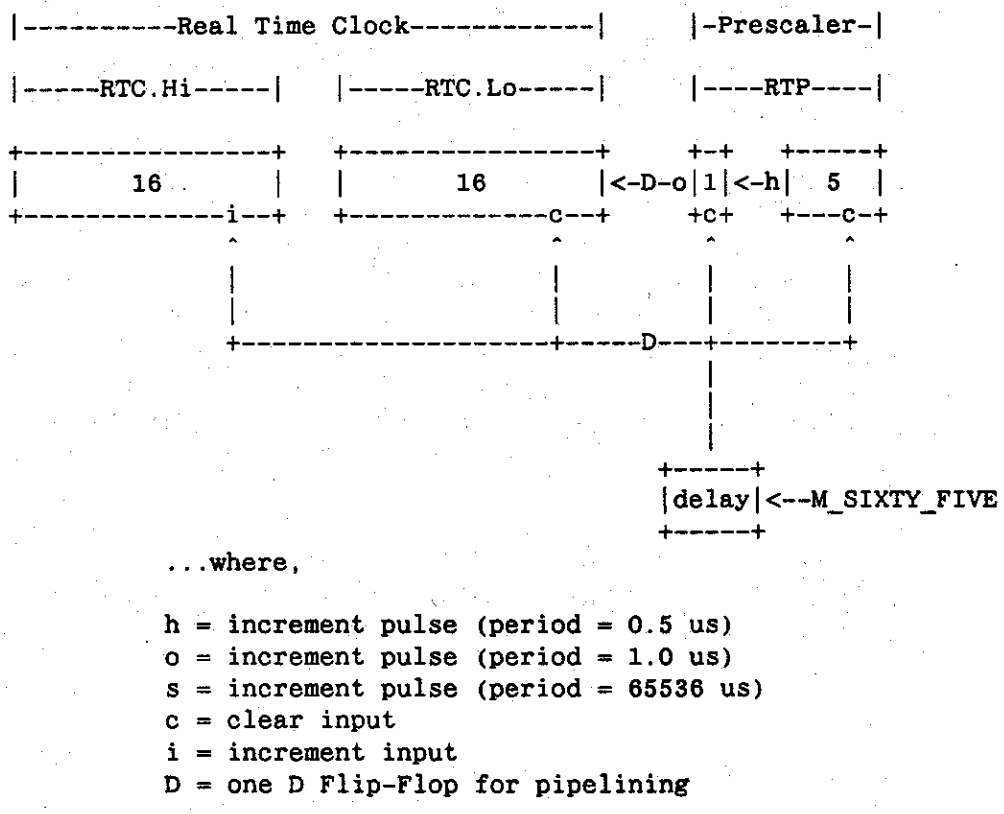


Figure A-10. Functional diagram – Real Time Clock.

Figure A-10 shows that prescaler is actually composed of two parts. The first part is a count-up prescale counter that has a programmable terminal count value. This 5-bit terminal value is supplied by the Real_Time_Prescale subfield of the ConfigA register (REQ_ConfigA.Real_Time_Prescale). The 5-bit counter drives the second part of the prescaler: a divide-by-two. The divide-by-two then generates the one-microsecond time base used by the Real Time Clock. Figure A-10 also shows the presence of the M_SIXTY_FIVE signal. This signal is a system-wide pulse which occurs every 65 milliseconds and lasts for one Switch Interval. It is used to keep all the Real Time Clocks on all nodes in synchronization.

The M_SIXTY_FIVE resets the entire prescaler and the lower-half of the Real Time Clock. In addition, it increments the upper-half of the Real Time Clock. Figure A-10 also shows a "pipeline" delay for the M_SIXTY_FIVE signal. The Configuration bits, REQ_ConfigA.Sixty_Five_Delay<1..0>, allow the adjustment of this delay. The adjustment values and their effects are shown in Figure A-11.

```
DD   Delay
==   =====
00   none
01   1 Switch interval
10   2 Switch intervals
11   3 Switch intervals

...where,

D..D  = ConfigB.Sixty_Five_Delay<1..0>
```

Figure A-11. Sixty_Five_Delay settings.

WARNING

The setting DD = 00 is for test purposes only and must NOT be used in normal operation.

In actual operation, the prescaler RTP<4..0> counts-up at the Switch frequency until it reaches the count stored in REQ_ConfigA.Real_Time_Prescale, where it generates an increment pulse lasting one Switch Interval. In the next Switch clock interval, the prescaler rolls-over to zero. Thus, the ConfigB.Real_Time_Prescale must always be programmed to make RTP<5> have a period of 0.5 microseconds.

WARNING

Because of hardware speed considerations, the OMSP generated by the RTP is actually pipelined by one Switch Interval. Therefore, the RTP appears to be running "ahead"

of the RTC by one Switch interval. This fact only becomes significant for the Slotted Start/Retry criterion. See that section for further details.

The Real Time Clock is basically, as mentioned previously, a large counter. The register definition of the Real Time Clock is shown in Figure A-12.

```
Register: Real_Time_Clock<31..0>

31                                0
|                                 |
15.............0 15.............0
HHHHHHHHHHHHHHHH LLLLLLLLLLLLLLLL (read)
HHHHHHHHHHHHHHHH ---------------- (write)
[Hi]             [Lo]

...where,

H..H  = high-order value (in 65,536 us)
L..L  = low-order value (in 1 us)
```

Figure A-12. Register definition – Real_Time_Clock.

Referring to Figure A-12, both the upper and lower-halves of the Real Time Clock (RTC.Hi) can be read from during actual operation. However, the lower-half should not normally be written to because of unwanted side-effects.

WARNING

Writing to RTC.Lo will cause the write data to override the count function but NOT override the clear function of that register. This means that the signal M_SIXTY_FIVE will clear the counter on its next occurrence. Therefore, there is not much meaning to initializing RTC.Lo.

WARNING

Any reads of the RTC must be taken as needed. This means that if the entire 32 bits must be read, it should be done in a single word-mode operation. Performing this same function with two serial half-word operations will yield incorrect results. In addition, any reads of the Real Time Clock have an uncertainty of approximately one microsecond. For writes, ONLY the half-word mode is acceptable for loading a value into the RTC.Hi register. This operation should only be attempted after reading the

RTC.Lo register and determining that it will not overflow when the write is being performed.

When performing reads of the Real Time Clock, the Configuration/Status Unit must take some special action to ensure that the read data is valid (stable). This is required because the Switch and T-Bus clocks are not always ensured to be synchronous and thus the Real Time Clock may be advancing as it is being read. The CSU accomplishes this goal in the following manner:

> When a read request for the Real Time Clock is detected by the CSU, the CSU immediately asserts the external SIGA pin: T_NSPAUSE_SIGA, and sends a request across a handshake synchronizer to the RTC controller logic. The RTC controller logic then waits for the next occurrence of the one microsecond increment pulse from: Real_Time_Prescaler<5>. When this occurs, the CSU is ensured of having a stable reading from the Real Time Clock for at least one microsecond. The RTC controller logic then sends an acknowledgment back across the handshake synchronizer where the CSU, upon detecting this event, negates T_NSPAUSE_SIGA and allows the data to be read. This is what contributes to the one microsecond uncertainty mentioned above.

WARNING

The CSU relies on the fact that the requesting T-Bus master will ensure that the total time - from the next occurrence of the one-microsecond increment pulse to the reading of data - will take no more than 1 us. This time includes the synchronizer delay from the RTC controller, the response time of the CSU, and time for any pauses that the T-Bus master may assert. Excluding the assertion of those pauses (T_NMPAUSE_xxxx) from the T-Bus master, the delay in the SIGA will be: 2*p(R_CLK) + 6*p(T_CLK) nanoseconds. The "p" represents the period of the indicated clock in nanoseconds. Therefore, the T-Bus master should use EXTREME caution when causing the assertion of T_NMPAUSE_xxxx. Beyond that, the CSU cannot guarantee the accuracy of the read data!

A.5.1.1.2.2 Time Of Next Interrupt

The Time Of Next Interrupt or TONI registers, are two 32-bit registers (A and B) which in combination with the Real Time Clock, are used to schedule an interrupt to occur at some moment in the future. Both registers, and their associated control logic, are completely independent from each other although they both interact with the Real Time Clock.

The TONI control logic performs a 32-bit subtraction between the current TONI_A (TONI_B) register values and the value of the entire Real Time Clock each time the

OMSP is valid. Whenever this subtraction yields a negative (two's-complement form) number, the SIGA sets (= 1) the bit: TONIA_Config.Status (TONIB_Config.Status).

Normally, whenever time the Status bit is asserted, an external pin, M_TONIA_INT (M_TONIB_INT), is also asserted (= 1). This can be enabled/disabled - asynchronously to the OMSP - by setting the TONIA_Config.Enable (TONIB_Config.Enable) bit to a 1/0. Disabling will force ONLY the pin to a "0". The associated status bit will still reflect the result of the current subtraction. Figure A-13 illustrates the TONI register definition. Figure A-14 illustrates the TONIA(B) configuration register definition.

```
Register: Time_Of_Next_Interrupt

31.............................0
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT

...where,

T..T = interrupt value
```

Figure A-13. Register definition - Time_Of_Next_Interrupt.

```
Register: TONIA(B)_Config

31.............................0
------------------------------E (write)
------------------------------SE (read)

...where,

E = asynchronously enable external pin
  = 0 disable M_TONIA(B)_INT external pin
  = 1 enable M_TONIA(B)_INT external pin

S = status
  = 0 TONIA(B) interrupt is not active
  = 1 TONIA(B) interrupt is active
```

Figure A-14. Register definition - TONIA(B)_Config.

The actual subtraction that is performed to initiate the interrupt is shown in Figure A-15. When performing writes to the TONI register, the Configuration/Status Unit must take some special action to ensure that the TONI register is not updated in the middle of the difference operation. The CSU accomplishes this goal in the following manner:

When a write request for the TONI register is detected by the CSU, the CSU immediately asserts the external SIGA pin: T_NSPAUSE_SIGA and sends a request across a fixed-delay handshake synchronizer to the TONIA(B) controller

logic. The TONIA(B) controller logic then waits for the next occurrence of the OMSP before it actually loads the TONIA(B) register. Because of pipelining, the TONIA(B) Subtraction Unit is ensured of having exactly one microsecond in which to complete the subtraction. The TONIA(B) controller then sends an acknowledgment back across the handshake synchronizer where the CSU, upon detecting this, negates T_NSPAUSE_SIGA, thus freeing-up the T-Bus master. This means, of course, that the SIGA will assert T_NSPAUSE_SIGA for approximately one microsecond.

```
TONIA(B)_Config<1> = 1 IFF,

   (TONIA(B)<31..0> - RTC<31..0>) < 0

...where TONIA(B) and RTC are treated as unsigned
   32-bit numbers and the difference is treated
   as a two's-complement number.
```

Figure A–15. Rule – time of next interrupt calculation.

A.5.1.1.2.3 Priority Time Slot

The Switch protocol provides a mechanism by which initial messages may be transmitted at various levels of priority in order to place an upper bound on remote access time. Normally, this priority is set by the T-Bus bits, T_PRIORITY<1..0>, during the request phase of the T-Bus transaction. In this case, the initial message is transmitted/retransmitted with the priority set during the T-Bus transaction which initiated the message. However, the Requestor can also force these bits to their EXPRESS value independently of the T-Bus transaction via the Priority Time Slot mechanism.

This mechanism works by assigning each Requestor a particular active time slot which is based on the value of the Real Time Clock. When that time slot "arrives", any pending Initial Switch message in the Requestor will have its priority raised to the EXPRESS level (=00). The priority is "sticky" in that once raised to EXPRESS, it remains there until the T-Bus initiates a new Initial Switch message. This new Initial message updates the priority with the value of T_PRIORITY<1..0>, as normal.

The equation for determining the active Priority Time Slot is shown in Figure A–16.

```
Priority Time Slot is active IFF the equation,

    (RTC.Lo<15..0> !$ PTC.Slot<15..0>) # PTC.Mask<15..0>

...is all 1's
```

Figure A–16. Rule – priority time slot promotion.

This equation takes a slot value (PTC.Slot), compares it on a bit-by-bit basis with a portion of the Real Time Clock (RTC.Lo) and then logically "or's" the result with the priority slot mask (PTC.Mask). It then detects the result for the presence of all "1's". Essentially, the RTC.Lo and the PTC.Slot are compared for equality on a word basis with some of the bits excluded, or "don't cared", in the comparison. A given bit position is excluded by setting the corresponding bit position in the Mask subfield to a "1". The Mask and Slot subfields, which are defined in Figure A-17, are programmable via the Configuration/Status Unit. The Priority Time Slot function can be disabled so that it NEVER promotes the priority of any message by negating (=0) the ConfigB.Ena_Priority_Promotion bit. The fully programmable capability of the Priority Time Slot allows the slot to be valid at different nodes in almost any order. It also allows the period of occurrence of the slot at a given node to be adjusted from constant up to 65 ms. Of course, the minimum time that a "slot" can be active at a given Requestor is one microsecond. Note that it is possible for the "slot" to arrive while the Requestor is sending out bids. This could result in one Bid being sent at lower priority and the remaining bid(s) sent at EXPRESS priority. However, logic in the Requestor ensures that no updating of priority occurs DURING Bid transmission. In addition, no updating will occur while the Requestor is either "idle" or "waiting". The "waiting" state is where the Requestor STU is waiting for a slotted/random start criterion to become valid.

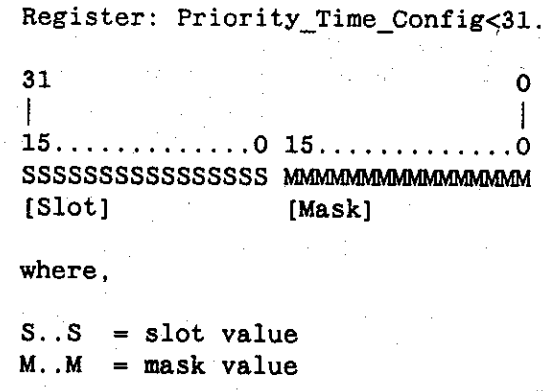


Figure A-17. Register definition - Priority_Time_Config.

Note that the purpose of the Priority Slot Value is NOT to ensure that a single high priority message be present in the Switch at any given time. Rather, the goal is to define the maximum bandwidth of priority messages to make the servicing of these messages as predictable as possible. In addition, the Priority Time Slot mechanism only applies to Initial Switch Messages (locked or not), which are always attempting to make a connection with some downstream node. Subsequent messages do not send Bids and thus are not affected by the Priority Time Slot mechanism.

A.5.1.1.3 Function Request Types

The Requestor handles various types of function requests from a T-Bus master. Those functions include read and writes of either bytes, words, or multiple words. Byte reads/writes may be of one to four bytes but must NOT wrap across word boundaries.

WARNING

It is important not to violate word wrapping because the Requestor does NOT check for this condition. Word reads/writes MUST be word-aligned and multiple read/writes are limited to a maximum of four words.

A.5.1.1.4 T-Bus Request Screening

T-Bus requests to the BIU of the Requestor are screened for both context errors and T-Bus protocol errors before any action is taken on them. Protocol errors include such things as a T-Bus master requesting an illegal (=00) T_PRIORITY field or illegally wrapping across word boundaries. Currently, protocol errors are NOT detected. Context errors, mostly relating to errors in handling locking, are listed in Figure A-18.

1. Requestor was asked to access a node within a locked sequence which is different than the node which opened that sequence. (Lock Address Error)
2. Requestor was asked to MAINTAIN a remote lock when it was never opened. (Maintain Present Error)
3. Requestor was not asked to MAINTAIN, BYPASS or OPEN a lock that was not yet explicitly released with FREE-LOCK. In other words, a NORMAL was issued while the Requestor was locked. (Maintain Absent Error)

Figure A-18. Requestor T-Bus screening errors.

Any of these errors will cause the Requestor to return an ERROR response with the appropriate error code on the T-Bus (See: "Error Detection and Reporting"). In addition, no Switch message will leave the STU. If the Switch path happens to be locked, any of these errors will also cause the BIU to initiate a sequence which will tear-down the Switch path (drop-lock) providing certain conditions are met. See "Locked Sequences" for more details.

NOTE

The Requestor, if unlocked, will treat a BYPASS in the same manner as a NORMAL Function Request; that is, it will NOT open a lock.

A.5.1.1.5 Initial Message Start/Retry Criterion

The Requestor can use one of several different methods to decide when to first begin transmission of an Initial Message and when to retry that transmission if the Switch rejects it. These methods are referred to as: slotted, random and immediate. The start

transmission time can be programmed to correspond to either one of two fixed time slots, one of two random numbers, or immediate transmission. The retry can correspond to either one of two fixed time slots or one of two random numbers. Only some combinations of these start and retry criterion are available for a given initial message.

The operation of random and slotted start and retry are described first. The process of selecting the various random/slotted start and retry criterion for a given message is then explained.

A.5.1.1.5.1 Random Start/Retry

There is a random number generator associated with the start/retry criterion. The generator is 12 bits long and is continuously updated at the Switch frequency. Each time an initial message start/retry occurs and the random backoff is selected, a new random number is transferred from the generator to a 12–bit count–down counter. This counter, known as the backoff counter, also runs at the Switch frequency. When the backoff counter reaches –1, the Requestor is released to start/retry the initial message transmission.

Before the backoff counter is actually loaded with the random number, that number is logically "anded" with a 12–bit backoff mask. When the Requestor first attempts the start/retry of an initial message, the backoff mask is initialized, forcing some number of most significant contiguous bits of the random number to zero as they are loaded into the backoff counter. After a certain number of Switch rejects for the same initial message, the mask is "shifted left" to allow an increase in the maximum allowable value of the next 12–bit random number loaded into the backoff counter. Thus, the random backoff limit, in terms of Switch intervals, is a binary number of length 12, or 4096. Each time a Switch reject is encountered, the Requestor makes a decision about whether or not to shift the backoff mask. That decision is made by adding a constant number to an accumulator after each Switch reject. Each time the accumulator overflows, the mask is shifted. Therefore, the mask may not change for several rejects.

In implementation, a register specifies randomization characteristics for the random start/retry criterion. This register is duplicated to allow for two sets of characteristics to be stored simultaneously. The mechanism for choosing one set or the other is described in a subsequent section. Each register is 8 bits long and specifies the initial mask setting, the constant value for accumulator addition and whether or not immediate start transmission is requested. These registers, and the random specifications which they describe, are subfields of the Transmit_Time_Config Register known as "Random0" and "Random1". Figure A–19 illustrates the structure of the random registers. Referring to Figure A–19, the immediate field, "I", when "1", forces an initial random start to be immediate, ignoring any randomization parameters. For initial retries, the "I" field is ignored and the randomization parameters are always used. The constant value for accumulator addition is specified by the "EE" field. This number is added to a 3–bit accumulator, which is then tested for overflow. The initial backoff mask is specified by the 5–bit identifier, "MMMMM", which is loaded directly into a Johnson Counter. The output of the Johnson Counter is decoded to derive a 12–bit

backoff mask as shown in Figure A–20. Figure A–20 also shows how the counter vances once loaded with an initial value. This advancement, of course, is governec the overflow of the 3–bit accumulator. Also note that the LSB of the backoff mask c never be cleared.

```
Register: Transmit_Time_Config.Random0<7..0>,
          Transmit_Time_Config.Random1<7..0>

7......0
IMMMMMEE

where,

I     = immediate
EE    = accumulator addition constant
MMMMM = initial comparison mask
```

Figure A–19. Register definition – Transmit_Time_Config.Random0,1.

	mask identifier<5..0>	backoff mask<11..0>
increasing count	000000	000000000001
|	000001	000000000011
|	000011	000000000111
|	000111	000000001111
|	001111	000000011111
|	011111	000000111111
|	111111	000001111111
|	111110	000011111111
|	111100	000111111111
|	111000	001111111111
|	110000	011111111111
V	100000	111111111111

Figure A–20. Random start/retry bit mask encoding.

During the INITIAL start/retry, five of the mask identifier bits related to the initial message are specified by the "MMMMM" field in the random register. The sixth, most significant bit is ALWAYS initialized to "0". So, if MMMMM = "11111", the initial backoff identifier would be: "011111". In this case, the maximum possible random backoff is "1111110", or 128 Switch intervals (recalling that the backoff counter overflows at –1). Once the maximum identifier of "100000" has been reached, the counter "wraps around" and thus the next backoff mask will be "000000". The "multiply by two" effect of the left-shifting backoff mask is intended to implement an exponentially increasing random backoff. An equation summarizing the preceding discussion is shown in Figure A–21.

WARNING

The initial mask identifier MUST be a value which would result in a legal Johnson Counter value as shown in Figure A-20. Legal Values would be: "00011" or "01111" for example. Illegal values would be: "00100" or "10110", for example.

```
                                                [M + int(R*E/8)]
Maximum backoff (Switch intervals) = 2

     ...where,

     M = initialized value of MMM bits
     R = number of rejects
     E = value of the EE bits
```

Figure A-21. Equation – maximum exponential random backoff.

A.5.1.1.5.2 Slotted Start/Retry

Slotted start and retry involves holding-off transmission based on the "arrival" of a pre-specified time slot. Once a slot has "arrived", a message assigned to that slot for starting can start transmission, and a message assigned to that slot for retry can retry transmission. The time slots are derived from the comparison of the Real Time Clock and a register used to specify the slot characteristics. This register is duplicated to allow for two sets of characteristics to be stored simultaneously. The mechanism for choosing one set over the other is described in a subsequent section. Each register is 8 bits long and specifies the comparison mask, the comparison value, and whether or not immediate start transmission is requested. These registers, and the slot specifications which they describe, are subfields of the Transmit_Time_Config Register known as "Slot0" and "Slot1". Figure A-22 shows the structure of the slot registers.

```
Register: Transmit_Time_Config.Slot0<7..0>,
          Transmit_Time_Config.Slot1<7..0>

7......0
IMMDDDDD

...where,

I     = immediate
MM    = mask specification
          00  4.0 us slot period
          01  2.0 us slot period
          10  1.0 us slot period
          11  0.5 us slot period
DDDDD = phase specification (restricted, see text)
```

Figure A–22. Register definition – Transmit_Time_Config.Slot0,1.

Referring to Figure A–22, the slot register contains three sub–fields: the compare mask field, specified by the two bit number, "MM"; the compare data field, specified by the five bit number, "DDDDD"; and immediate field, "I". The immediate field, when "1", forces an initial slotted start to be immediate, ignoring any slot parameters. For initial retries, the "I" field is ignored and the slot parameters are always used. The comparison for an active slot is made partially by comparing bits of the "D" sub–field with bits of the of the Real Time Clock and Real Time Prescaler. The "M" sub–field is used to either compare some of those bits with zeros or to ignore them in the comparison. This operation is shown in Figure A–23.

```
given, nnnnnnnn = RTC.Lo<1..0> | RTP<5..0>

mm    compare     with        cycle period
==    ========    ========    ============
00    OOODDDDD    nnnnnnnn       4 us
01    XOODDDDD    nnnnnnnn       2 us
10    XXODDDDD    nnnnnnnn       1 us
11    XXXDDDDD    nnnnnnnn      .5 us
```

Figure A–23. Rule – start/retry valid slot comparison.

Referring to Figure A–23, the D field can only take on values that are less than or equal to the setting of the Real_Time_Prescaler<4..0>.

WARNING

Values outside this range may cause the message to never be transmitted, and are therefore illegal.

Figure A–23 also demonstrates the two properties of the slots: frequency and phase. The D field allows setting a number of phases equal to the setting of REQ_ConfigA.Real_Time_Prescale<4..0> plus one. The M field allows the comparison to occur at varying time intervals.

Because of hardware limitations, the concatenated quantity, (RTC.Lo<1..0> | RTP<5..0>), does not act exactly like an eight bit counter. The RTP portion is actually running one switch interval "ahead" of the RTC.Lo<1..0> portion. This means that the RTC actually increments on the 000000–to–000001 transition of the RTP portion, rather than on the 111111–to–000000 portion. A sample transition would look like that in Figure A–24.

RTC.Lo<1..0>	RTP<5..0>
10	11111100
10	11111101
10	11111110
10	11111111
10	00000000
11	00000001
11	00000010
11	00000011

Figure A–24. Start/retry slot comparison count sequence.

A.5.1.1.5.3 Start/Retry Criterion Selection

A function request from a master on the T–Bus is transformed into a Switch message by the Requestor. Depending on certain parameters of that function request, the Requestor categorizes the message into one of four Message Classes. Each of these classes will have a different start and retry criterion. The correspondence of start/retry criterion based on message classes is shown in Figure A–25.

Class	Start	Retry
00	Slot0/Immediate	Slot0
01	Slot1/Immediate	Slot1
10	Random0/Immediate	Random0
11	Random1/Immediate	Random1

Figure A–25. Start/retry criterion based on message classes.

A class is selected for each Switch message based on the state of three bits of T-Bus function request that initiated the message. Those bits are the T-Bus signals T_LOCKOP<1> and T_RR<1..0>. The Requestor uses the encoded state of those three bits to "look up" the class of the message. The lookup table itself is a 16-bit register known as the Message_Classification Register. This register is defined in Figure A-26.

```
Register: Message_Classification<15..0>

15                                          0
|                                           |
10   10   10   10   10   10   10   10
CC   CC   CC   CC   CC   CC   CC   CC
[A]  [B]  [C]  [D]  [E]  [F]  [G]  [H]

...where given that nnn = T_LOCKOP<1> | T_RR<1..0>,
   the subfields selected and the type of function
   request that selects them are,

nnn    Subfield     Function Request
===    ========     ===========================
000      MC.H       Unlocked Writes
001      MC.G       Unlocked Reads
010      MC.F       Auxiliary Unlocked Writes
011      MC.E       Auxiliary Unlocked Reads
100      MC.D       Locked Writes
101      MC.C       Locked Reads
110      MC.B       Auxiliary Locked Writes
111      MC.A       Auxiliary Locked Reads
```

Figure A-26. Register definition – Message_Classification.

To illustrate the Message Start/Retry Criterion selection with an example, suppose that a function request to the Requestor may have set, (T_LOCKOP<1> | T_RR<1..0>) = 100. From Figure A-26, this would cause the Requestor to look in the Message Classification register "D" subfield (for Locked Writes). In this subfield, the Requestor would find the "class of message" corresponding to the particular function request. If the "D" subfield were a "10", that particular message would have use the parameters in Random0 register for both message start and retry.

Both the Start/Retry Random and Start/Retry Slot registers are actually subfields of the Transmit_Time_Config Register. The bit definition for this register is illustrated in Figure A-27.

```
Register: Transmit_Time_Config<31..0>

31                                          0
|                                           |
7......0   7......0   7......0   7......0
IMMMMMEE   IMMMMMEE   IMMDDDDD   IMMDDDDD
[Random1]  [Random0]  [Slot1]    [Slot0]

...where, Random0, Random1, Slot0 and Slot1
   are previously defined
```

Figure A-27. Register definition – Transmit_Time_Config.

NOTE

Function requests can be forced to completely ignore the Message Classification register on a request-by-request basis. This occurs whenever a request is made and the T-Bus signal: T_SYNC is asserted (=1). In this case, the message is automatically classed as "00" and both initial transmission and retry criterion is taken from the Transmit_Time_Config.Slot0 register.

A.5.1.1.6 Switch Tx Protocol Timers

The Requestor contains timers which monitor the progress of the transmitted message and alert the Requestor if they detect an error condition. Specifically, there are two timers, the Reject Timer and Connection Timer. The Reject Timer determines how long the Requestor will attempt to open a Switch path in the face of Switch rejects. The Connection Timer monitors how long the Requestor will keep a Switch path open once the rejection period is finished. Parameters for both the Reject Timer and the Connection Timer are contained in the Protocol_Timer_Config Register.

A.5.1.1.6.1 Reject Timer

The Reject Timer is enabled at the beginning of the first attempt to transmit an initial message. Each time the Requestor receives a reject, it first examines the Reject Timer. If the timer has underflowed (the underflow is latched), the Requestor halts the transmission attempt and returns the Rej_TO Error code to the T-Bus master. The Requestor also tears-down the Switch path whether or not it was locked. Parameters for the Reject Timer are located in the Protocol_Timer_Config Register.

The Reject Timer is structured as a 4-bit down-counter clocked by a selectable prescaled time base. The reload value for the counter is contained in Protocol_Timer_Config.Cnt<3..0>. A 4-bit prescale parameter, located in

Protocol_Timer_Config.Pre<3..0>, is used to select the desired prescale time base from one of sixteen possible frequencies. Those frequencies are derived from the low-to-high transition of bits of the Real Time Clock, Real_Time_Clock.Lo<15..0>, as illustrated in Figure A-28. The Reject Timer is continually loaded with PTC.Cnt<3..0> until it begins transmitting Bid #1. An equation for the maximum Reject timeout is shown in Figure A-29.

PRE	Q	PRE	Q
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

```
...where,

PRE = Protocol_Timer_Config.Cnt<3..0>
Q   = selection from Real_Time_Clock.Lo, bit Q
```

Figure A-28. Reject timer prescale selection.

```
given,

CNT = Protocol_Timer_Config.Cnt<3..0>
PRE = Protocol_Timer_Config.Pre<3..0>

...then,
                    (PRE + 1)
Timeout = CNT * 2            microseconds

                             (PRE + 1)
...with an uncertainty of 2            microseconds
```

Figure A-29. Equation – reject timeout.

A.5.1.1.6.2 Connection Timer

The Connection Timer is loaded each time the Requestor sends Bid 1. This means that it is reloaded both just before transmitting an initial message and after the Requestor receives each Switch reject. Like the Reject timer, its underflow condition is latched.

The Connection Timer's timeout has two different effects depending on when it occurs. If the timeout occurs while the Requestor is waiting for a message acknowledgment (M_ACK), the Switch path is torn-down (whether locked or not) and a Conn_TO Error is returned to the T-Bus master. If the timeout occurs while a Switch path is locked, but after the M_ACK was received, the Requestor will tear-down the Switch

path but cannot return an error to the T-Bus master immediately. Rather, it waits until the next T-Bus master makes a request to return a Wait_TO Error. In the "race condition" case where the M_ACK and connection timer underflow occur on the same clock edge, a Conn_TO Error is detected.

The Connection Timer is structured as an 8-bit down-counter clocked at 1 MHz by a bit from the Real Time Prescaler, Real_Time_Prescale<5>. The counter underflows at -1. The reload value for the counter is contained in, Protocol_Timer_Config.Con<7..0>. The equation for the maximum connection timeout is shown in Figure A-30.

```
given,

CON = Protocol_Timer_Config.Con<7..0>

then,

Timeout = CON + 1  microseconds

...with an uncertainty of 1 microsecond
```

Figure A-30. Equation - connection timeout.

A.5.1.1.6.3 Protocol Timer Programming

As previously mentioned, the parameters for the Protocol Timers are contained in subfields of the Protocol_Timer_Config Register as shown in Figure A-31.

```
Register: Protocol_Timer_Config<15..0>

15                    0
|                     |
3..0    3..0    7......0
CCCC    PPPP    NNNNNNNN
[Cnt]   [Pre]   [Con]

...where, Cnt, Pre and Con have
   been previously defined.
```

Figure A-31. Register definition - Protocol_Timer_Config.

A.5.1.1.7 Anticipation Support

The operation of the Requestor has two main goals: (1) to pass a T-Bus function request to the Switch as quickly and efficiently as possible, and (2) to return the corresponding function response from the upstream Switch message to the T-Bus master as quickly and efficiently as possible. Certain techniques can be used to take advantage of the expected operation of the logic in the function request and response path. These

techniques are known collectively as "anticipation". The use of anticipation in achieving the two main goals of the Requestor are now discussed.

A.5.1.1.7.1 Function Requests

Maximizing downstream function request efficiency in the Requestor involves balancing the desire for speed with the desire to maintain a streamlined Switch protocol. These tradeoffs become apparent when considering a multi-word write sequence. Here, the Requestor could signal its Switch Transmit Unit to begin transmitting as soon as possible after receiving the T-Bus request. This would always work if the T-Bus were guaranteed to supply all words of a multi-word transfer at a bandwidth equivalent to the bandwidth of the Switch. However, this will not always be the case as the variations between the clock frequency of the T-Bus and the Switch, combined with the ability of the current T-Bus master to assert PAUSE, create the possibility of the STU "running out of data" in some circumstances.

To circumvent this problem, two immediate options are available. First, change the Switch protocol to allow the insertion of "null data word" fields when data is not available. Second, the Requestor could be programmed to signal the STU to start only after a specified number of words have been written during the data portion of the T-Bus transfer. The first alternative is unattractive because it increases Switch bandwidth and unnecessarily introduces complexity into the Switch message protocol. The second option is therefore implemented in the Requestor. The programmed parameter is known as, FQ_Anticipation, and can be set to any of the thresholds listed in Figure A-32.

```
Register: Requestor_ConfigA.FQ_Anticipation<2..0>

210     Anticipation
===     ============
000     after first data word transferred
001     after second data word transferred
010     after third data word transferred
011     after fourth data word transferred
1XX     immediately after T-Bus request
```

Figure A-32. Register definition – Requestor_ConfigA.FQ_Anticipation<2..0>.

Since it is possible for the FQ_Anticipation to be set greater than the last word of a particular write, the Requestor will commit to transmission when either the last word has been written OR the Requestor FQ_Anticipation threshold has been reached – whichever occurs first. For example, if FQ_Anticipation were a "011" and a three word write occurred, anticipation would take place after the third word were written. In addition, an Interleaved request (I_INTERLEAVED = 1) will cause a "1XX" setting to signal the STU in the cycle AFTER the T-Bus request. The threshold should be set based on the T-Bus and Switch clock frequencies, the maximum number of PAUSE assertions expected during a write, and the handshake synchronizer delay setting.

WARNING

Because of current hardware restrictions, FQ_Anticipation MUST be set to 1XX. Therefore, any T-Bus master MUST be able to supply write data fast enough to prevent the Switch message from running out of data.

A.5.1.1.7.2 Function Responses

Anticipation during function responses would allow the Requestor to take advantage of the synchronizer settling time by beginning the T-Bus request BEFORE the message checksum has been verified. Unfortunately, the Requestor is limited in the amount of anticipation that it can provide. Whatever anticipation the Requestor can extract from an upstream message, that anticipation has to be constant over all messages. This is because the Requestor STU-to-BIU handshake synchronizer has to compensate for message anticipation and cannot have its setting varied according to the expected upstream message type. And of course, even if the anticipator could vary its setting, the return message profile is not always known.

In fact, the Requestor SRU must assume a minimum expected upstream message length before starting anticipation. That minimum message length is two bytes. And since the SRU cannot tell if the assertion of Reverse is a Reject until the second byte, the minimum anticipation of the Checksum byte is one Switch Interval (for a function response to a write request). This then limits anticipation of all messages to one byte. By comparison, the Server has a minimum message length of 5 bytes and can thus take greater advantage of anticipation techniques.

As previously mentioned, Switch to T-Bus anticipation usually requires some minimum setting on the receiving T-Bus synchronizer. However, it turns out that no MINIMUM setting of Req_ConfigA.BIU_Synch<3..0> is required to compensate for the small amount of Requestor SRU anticipation. This is because pipeline overhead already accounts for this anticipation. However, a minimum setting IS required to meet the minimum settling time for the synchronizer. For more details on this subject, see: "Special Topics/Synchronization".

A.5.1.1.8 Locked Sequences

Sometimes an upstream T-Bus master wishes to perform several consecutive function requests to a locked remote T-Bus slave without the overhead of opening the Switch connection before each request. A mechanism known as Switch locking allows such multiple accesses by keeping the Switch path open between function requests. All transactions that take place during locking are known as locked sequences. A locked sequence has three distinct events: opening, maintaining and closing. Each of these events has different characteristics and restrictions for the Requestor.

A.5.1.1.8.1 Opening and Maintaining Locks

Opening a Switch lock begins with an otherwise normal function request from a T-Bus master that carries with it a request for "opening a lock" to a remote T-Bus slave. The upstream Requestor transfers the OPEN lock request to the downstream Server via a bit in the message protocol. Since the Switch path has not yet been established, either the Switch or the downstream Server may reject the message. A Switch reject will occur because of normal Switch contention and the Server reject will occur if the downstream target was locked. The Requestor, not knowing the source of the Switch reject, will simply retry the message transmission within the constraints of the Protocol Timers.

Assuming that the message finally does "get through" to the downstream Server, that Server "opens a lock" to the target T-Bus slave in accordance to the T-Bus protocol. Meanwhile the upstream Requestor, recognizing that it has established the beginning of a locked sequence, does not normally tear-down the Switch connection upon receiving an M_ACK unless an error was detected. This is discussed in detail in the "Auto Drop" section.

Once a locked Switch path is established with OPEN lock, it must be explicitly instructed to remain open by the upstream T-Bus master. This is accomplished by following the OPEN function request with either: another OPEN, a MAINTAIN, or BYPASS function request. Essentially, the Requestor takes no special action on either of these requests but does demand their presence. If the OPEN/MAINTAIN/BYPASS protocol is violated by subsequently initiating a NORMAL function request, the Requestor will respond to the offending T_Bus master with an ERROR and tear-down the Switch path. This mechanism is described in the "T-Bus Request Screening" section.

A.5.1.1.8.2 Dropping Locks

The Requestor has a flag, known as the "drop-lock request" flag, which causes the Requestor to negate Frame and return to its unlocked Idle state. Although the flag does not cause this action until the Requestor BIU is in its Locked Idle state, it can be set at any time. Once set, a drop-lock condition is said to be active. There are three distinct scenarios under which a drop-lock condition may occur: (1) A T-Bus master which is locked to the Requestor may issue a FREE-LOCK, (2) The Requestor issues an ERROR response (under certain conditions), and (3) a Connection Timer timeout.

Whatever the cause of the drop-lock condition, the Requestor BIU waits until it returns naturally to its Locked Idle state before taking action. Once there, the Requestor BIU will then enter the "unlock" state in which it will fulfill the drop-lock request flag by commanding the Requestor STU to negate Frame. During this state, the Requestor BIU will issue a REFUSED response to ANY T-Bus Master that accesses it. Once the Requestor BIU has been signaled by the STU that Frame was negated, the BIU returns to its Unlocked Idle state. Of course, the drop-lock request flag is then also negated.

The downstream Server, knowing that it was previously locked, interprets the subsequent loss of its incoming Frame to be a FREE-LOCKS. The Server, sensing an unexpected loss of Frame, then issues a FREE-LOCKS to the local T-Bus.

The first drop-lock scenario - a FREE-LOCKS issued by a T-Bus master - is the most conventional. The FREE-LOCKS request is the only function request that is NOT explicitly transmitted to the downstream Server in the form of a message. Instead, the Requestor responds to a FREE-LOCKS by negating Frame to the Switch interface. Because the drop-lock condition can be entered at any time, a T-Bus master can issue a FREE-LOCKS at any time - whether the Requestor is idle or acting on a current split-cycle. However, the Requestor must be already locked to the T-Bus master which made the request. If not, the BIU will ignore the FREE-LOCK request.

In the ERROR response scenario, the Requestor will NEVER enter the drop-lock condition when the ERROR response is due to a Remote Class Error. However, it MAY enter the drop-lock condition when the ERROR response is due to an FQ or Switch Class Error. This conditional action is described in the "Auto Drop" section. Error classes are discussed in the "Error Detection and Reporting" section. However, if those conditions ARE valid for a drop-lock, the Requestor processes the drop-lock in the same manner as the FREE-LOCKS scenario. Unlike the FREE-LOCKS however, drop-lock processing takes place almost immediately after the event which caused the drop-lock condition (responding with an ERROR). This is because the Requestor BIU always enters its Locked Idle state immediately after issuing an ERROR response.

The Connection Timer timeout scenario is slightly different from the previous two. When the Connection Timer times-out, it indirectly causes the drop-lock condition by eventually causing an ERROR response (Wait_TO or Idle_TO) by the Requestor BIU. This normally would be sufficient because the BIU would then enter the drop-lock condition, which would then signal the Requestor STU to negate Frame. However, one of the reasons that the Connection Timer may have timed-out was because the Requestor BIU had lost its T-Bus clock (T_CLK). In this case, Frame would never get negated. Therefore, the Requestor STU takes the initiative to negate Frame immediately after a Connection Timer timeout. For consistency, the drop-lock mechanism continues as normal. When the Requestor STU finally gets the request from the BIU to negate Frame, the STU simply ignores that request.

A.5.1.1.8.3 Auto Drop

Auto drop is a parameter set by the Req_ConfigA.Ena_Auto_Drop bit. When asserted (= 1) the Requestor will be permitted to enter the drop-lock condition whenever an ERROR response is generated because of an FQ or Switch Class error. Otherwise, the Requestor will NEVER enter the drop-lock condition due to an ERROR response. This is because the only other class of Requestor error - Remote Error - will NEVER cause the drop-lock condition.

A.5.1.1.9 Stolen Bit Support

Because of the structure of the Switch message format, only one bit of Stolen information can be transferred between upstream and downstream nodes during a given message. The Requestor records the state of the Stolen bit during the word transferred in a byte write operation. It is this state that is reflected in the Switch message. Normally, the Requestor expects the Stolen bit to be asserted only during a BYTE write operation. In fact, it is illegal to assert the Stolen bit to the Requestor during a multi-word operation.

NOTE

If the Stolen bit IS asserted during a multi-word write, the state of the first word written is recorded.

The Requestor provides a mechanism to verify that the Stolen bits of all words in a multi-word write are zero, and prevent the message from being transmitted if this is not the case. The Ena_Stolen_Verify bit in the Req_ConfigB register, when asserted, will enable this verification of Stolen bits in a multi-word write. There is however, a small price to pay for this feature: the FQ_Anticipation register must be set to its MAXIMUM value (=011). This is because the Requestor must load all words of a multi-word write and verify the Stolen bits before committing to transmission. The Requestor cannot "call back" the outgoing message. Figure A-33 summarizes the rules for verifying the Stolen bit. If the rules of Figure A-33 are adhered to and a particular multi-word write has some of the Stolen bits asserted, the Requestor will respond with an ERROR ("Stolen_Verify" error code) to the T-Bus master. The Requestor, of course, will NOT transmit the message in this case.

```
To enable the verification of Stolen bits on a multi-word writes,

  1) Set FQ_Anticipation = 011, AND...

  2) Assert (=1) the Req_ConfigB.Ena_Stolen_Verify bit
```

Figure A-33. Rules - Stolen bit verification - multi-word write.

For single-word reads, the Requestor presents to the T-Bus a Stolen bit (T_AD<32>) which is the same state as the Stolen bit in the upstream Checksum byte. For multi-word reads, the Requestor always assumes that the words of the transfer are NOT Stolen until it encounters an asserted Stolen bit in the Checksum byte. When this occurs, only the last word received by the Requestor is assumed to be Stolen. This fact is transmitted to the T-Bus by asserting T_AD<32> during the transfer of the last word on the T-Bus.

WARNING

Due to current hardware restrictions, Ena_Stolen_Verify must ALWAYS be negated (=0). Therefore, multi-word writes CANNOT be screened for stolen bits by the Requestor. This must be handled by the T-Bus Master.

A.5.1.1.10 Quick Drop

The Requestor STU has an option which enables it to negate Frame during an Initial Message as soon as the STU detects an asserted Reverse. This can be done without the STU actually waiting to see if Reverse is going to be a Reject or an actual message. This action is allowed only when the STU is transmitting an Initial Message (NOT an Initial Locked Message) because in this situation, the only possible responses are: Reject or an upstream Switch message. In either case, the Requestor will negate Frame immediately if the bit: Requestor_ConfigB.Ena_Quick_Drop is asserted (=1). Essentially, Quick Drop is an optimization which will free up the Switch earlier – although only by one Switch Interval – than if Quick Drop were not enabled.

A.5.1.1.11 Reverse Profile Monitoring

The Requestor is enabled to monitor the profile of Reverse for errors asserting (=1) the Req_ConfigB.Ena_Rev_Err bit. Once enabled, the Requestor will report a Switch Class Error (Reverse_Error) whenever it observes an incorrect state for Reverse during an upstream message. Since there is more than one possible Reverse profile for a given Function Request, not every Switch Interval of Reverse can be checked for a given state (0/1) because either state may be valid. However, when the Reverse profile is incorrect in ANY place that is checked, a Reverse_Error is reported.

Figure A-34 illustrates how the Requestor checks the Reverse profile. The "x's" represent where either state is valid and is therefore not checked by the Requestor.

```
TYPE      #WORDS              RETURN MSG FORMAT
=====     =========           ================

                              +--- first received
                              |
                              V
write     any                 xxL

read      non-multi           xxHH,HL

          two-words           xxHH,HL
             "                xxHH,HxHH,HL

          three-words         xxHH,HL
             "                xxHH,HxHH,HL
             "                xxHH,HxHH,HxHH,HL

          four-words          xxHH,HL
             "                xxHH,HxHH,HL
             "                xxHH,HxHH,HxHH,HL
             "                xxHH,HxHH,HxHH,HxHH,HL
```

...where,

x = don't care
H = check for Reverse = 1
L = check for Reverse = 0

Figure A-34. Requestor Reverse profile monitoring.

NOTE

The Requestor will NOT specifically check that Reverse was negated (=0) when the Function Request was initiated. However, it DOES begin looking for a 0-to-1 transition of Reverse in order to recognize the beginning of the upstream message. Therefore, if Reverse were to be "hung high" when the Requestor began its Function Request, the Requestor would eventually timeout the Connection Timer.

A.5.1.1.12 Error Detection and Reporting

Errors delivered by the Requestor to an initiating T-Bus master can be divided into three classes depending on which part of the SIGA detects them. The classes are: 1) FQ Errors - which are detected by the BIU from the original Function Request; 2)

Switch Errors - which are detected by the STU and SRU because of Switch interactions and 3) Remote Errors - which are detected by the downstream Server and are "reflected" up to the initiating T-Bus Master.

For a given Function Request/Response sequence, errors from different classes can occur simultaneously. Since only one error can be reported at a time, a sense of "priority" exists between error classes. If there is a FQ Error, it always be reported, regardless of the presence of Switch or Remote Errors. If there is no Local Error, than any Switch Errors will be reported, regardless of the presence of Remote Errors. If there is neither a Local nor a Switch Error, then and only then will any Remote Errors are reported.

Figure A-35 shows the Error Codes for the Requestor which include the FQ and Switch type errors. Note that WITHIN a given Error Class, the errors are again not all mutually exclusive, and are therefore given "within-class" priorities. A more detailed description of the three Error Classes follows.

```
Requestor Error Codes:

7       0
|       |
PPPPdcba

d c b a  Requestor Error          Class
= = = =  ====================    =========
0 0 0 0  Maintain_Absent-(1a)     FQ
0 0 0 1  Maintain_Present-(1b)    FQ
0 0 1 0  Stolen_Verify(2)         FQ
0 0 1 1  Lock_Address-(3)         FQ
0 1 0 0  Wait_TO-(4a)             Switch
0 1 0 1  Idle_TO-(4b)             Switch
0 1 1 0  Rej_Abort(5)             Switch
0 1 1 1  Rej_TO-(6)               Switch
1 0 0 0  Reverse-(7)              Switch
1 0 0 1  Check-(8)                Switch

...where,

P..P = Requestor_ConfigA.Error_Prefix<3..0>.
       Priority is from highest (1) to lowest (8).
       Within a given priority, errors are mutually
       exclusive (i.e., 4a,b...).
```

Figure A-35. Requestor error codes.

A.5.1.1.12.1 FQ Errors

FQ Errors are detected by the BIU during the original Function Request. Their detection, when enabled, will ALWAYS prevent the Function Request from initiating a Switch access. If the Requestor is unlocked, it will NOT assert Frame after detecting

an FQ Error. If the Requestor is locked, it MAY immediately tear-down the lock if certain conditions are met. See "Auto Drop" for more details.

FQ Error types and their definitions are illustrated in Figure A-36.

Lock Address Violation	Requestor was asked to access a node within a locked sequence which is different than the node which opened that sequence. (Only detected if configured to do so.)
Maintain Present	Requestor was asked to MAINTAIN a remote lock when it was never OPENed. (Only detected if configured to do so.)
Maintain Absent	Requestor was not asked to MAINTAIN, BYPASS or OPEN a lock that was not yet explicitly released with FREE-LOCK. (Only detected if configured to do so.)

Figure A-36. FQ error definitions.

A.5.1.1.12.2 Switch Errors

Switch Errors are caused by a variety of conditions that are detected by the logic which monitors the progress of the Switch message as it enters and returns from the Switch interface. Unlike FQ Errors, Switch Errors are detected once the Switch transaction is already underway. They are reported to the T-Bus Master only when the transaction is "finished", either normally or due to some timeout. Therefore, Switch Errors can only have a special effect on Frame during a locked sequence. In this case, the Requestor MAY immediately tear-down the lock if certain conditions are met. See "Auto Drop" for more details.

Switch Error types and their definitions are illustrated in Figure A-37.

Wait_TO	The Switch Transmit Connection Timer overflowed while the Requestor was waiting for a Function Response. (See: "Connection Timer")
Idle_TO	The Switch Transmit Connection Timer overflowed while the Requestor was in its idle state. (See: "Connection Timer")
Rej_Abort	The Switch Transmit Reject Timer was forced into overflow by the REJ_ABORT input pin. (See: "Reject Timer")
Rej_TO	The Switch Transmit Reject Timer overflowed while the Requestor was attempting to open a connection. (See: "Reject Abort")
Reverse	The Requestor detected an incorrect polarity of the Reverse signal during a Function Response. (See: "Reverse Profile Monitoring")

Check	The Requestor detected an incorrect Checksum during a Function Response. (See: "Checksum Support")

Figure A-37. Switch error definitions.

A.5.1.1.12.3 Remote Errors

Remote Errors include: 1) errors which are detected within the Server logic itself, and 2) errors generated as T-Bus errors responses by a downstream T-Bus slave device. Both types errors are simply passed-through "as is" to the upstream Requestor. This Requestor simply "hands" them - without differentiation - to the initiating T-Bus Master. Remote Errors, unlike FQ and Switch Errors, can NEVER cause the Requestor to "drop" a lock.

For a summary of the "Server-sourced" Remote errors, see: "Server/Operation/Error Reporting".

A.5.1.1.13 Disabled Operation

The Requestor can be disabled via a number of bits in the Requestor_ConfigB register. These include: Ena_REQ_BIU, Ena_REQ_STU, Ena_REQ_SRU, and Ena_REQ_CNT. These bits reset the four major blocks of the Requestor.

WARNING

In normal operation, these bits SHOULD ALWAYS BE ASSERTED/NEGATED AT THE SAME TIME. Otherwise, erratic Requestor operation may result.

When these bits are disabled (=0), the Requestor T-Bus interface will respond "REFUSED" to any T-Bus master that tries to access it. The Requestor will also ignore any assertions of REVERSE from the Switch interface.

A.5.1.1.14 Configuration Registers

The Requestor has two general Configuration Registers. They are: Requestor_ConfigA and Requestor_ConfigB. In general, both Configuration Registers are used to set miscellaneous parameters and enable/disable certain functions. Figure A-38 shows the structure of Requestor_ConfigA.

```
Register: Requestor_ConfigA<31..0>

BIT/FIELD    FUNCTION (read/write)
=========    =====================
<31..29>    REQ_Slave_Num[3]
    <28>    Modulo_8
    <27>    Columns_2
    <26>    Ena_Auto_Drop
<25..23>    FQ_Anticipation[3]
<22..19>    STU_Synch[4]
<18..15>    BIU_Synch[4]
<14..11>    Error_Prefix[4]
 <10..9>    Sixty_Five_Delay[2]
  <8..6>    CSU_Slave_Number[3]
  <5..1>    Real_Time_Prescale[5]
     <0>    Columns_1
```

Figure A-38. Register definition – Requestor_ConfigA.

The bit definition of Requestor_ConfigA is shown in Figure A-39. This register contains mostly configuration bits that affect the run-time parameters of the Requestor. All bits are "high-true" and are reset (low) upon system reset. The structure of Requestor_ConfigB is shown in Figure A-40. The bit definition of Requestor_ConfigB is shown in Figure A-41. This register contains mostly configuration bits that enable/disable different functions and error reports of the Requestor. All bits are "high-true" and are reset (low) upon system reset.

REQ_Slave_Num[3] Configures the T-Bus slave number that the Requestor will respond with (on the T_SOURCE<2..0> pins) when making a Function Response.

Modulo_8 Configures the Requestor to expect either a modulo-8 element (=1) or a modulo-16 (=0) Switch element.

Columns_2 Configures the Requestor to expect either a 2-column (=0) or a 3-column Switch.

Ena_Auto_Drop Enables the Requestor to tear-down a connection when a Function_Request or Switch class of error is detected (=1). Otherwise, these types of error will only be reported by the Requestor and no special action will be taken (=0).

FQ_Anticipation[3] Configures the Requestor for the desired Function Request Anticipation. (See: "Anticipation Support")

STU_Sync[4] Configures the settling time of the Switch Transmit Unit's (STU) handshake synchronizer which receives an "execute" signal from the Bus Interface Unit (BIU). This signal is used to initiate a Function Request on the Switch. (See: "Synchronization")

BIU_Sync[4]	Configures the settling time of the Bus Interface Unit's (BIU) handshake synchronizer which receives a "completed" signal from the switch transmit unit (STU). This signal is used to indicate that a function response has been received by the SRU. (See: "Synchronization")
Error_Prefix[4]	Configures the Prefix (T-Bus bits: D7-D4) of the Error code response for Requestor errors. (See: "Error Handling")
Sixty_Five_Delay[2]	Configures the pipeline delay of M_SIXTY_FIVE pulse. Millisecond pulse as seen by the Requestor. WARNING: DO NOT USE THE "00" SETTING. (See: "Real Time Clock" for further details)
CSU_Slave_Number[3]	Configures the Slave number that the CSU will respond with (on the T_SOURCE<2..0> pins) when making a Function Response.
Real_Time_Prescale[5]	Configures the terminal count of the Real Time Prescaler. (See: "Real Time Clock" for further details)
Columns_1	Configures the SIGA for a 1-column switch. (See: "Real Time Clock" for further details)

Figure A-39. Bit definition - Requestor_ConfigA.

```
Register: Requestor_ConfigB<31..0>

BIT/FIELD    FUNCTION (read/write)
=========    =====================
 <31..23>    Route_Address_Mask[9]
     <22>    Ena_Stolen_Verify_Err
     <21>    Ena_Maintain_Absent_Err
     <20>    Ena_Maintain_Present_Err
     <19>    Ena_Lock_Addr_Err
     <18>    Ena_Wait_TO_Err
     <17>    Ena_Idle_TO_Err
     <16>    Ena_Rej_Abort_Err
     <15>    Ena_Rej_TO_Err
     <14>    Ena_Check_Err
     <13>    Ena_Reverse_Err
     <12>    Ena_Remote_Err
     <11>    Ena_Quick_Drop
     <10>    Ena_Priority_Promotion
      <9>    Ena_Interleaver
      <8>    Ena_Reject_Abort
      <7>    Ena_Reject_Timer
      <6>    Ena_Conn_Timer
      <5>    Ena_Switch_Frame
      <4>    Ena_REQ_BIU
      <3>    Ena_REQ_STU
      <2>    Ena_REQ_SRU
      <1>    Ena_REQ_CNT
      <0>    SPARE
```

Figure A-40. Register definition – Requestor_ConfigB.

Route_Address_Mask[9] Configures the randomization mask for the Bus Interface Unit's translation of the Logical Route Address to the Physical Route Address. (See: "Route Address Generation")

The Enable Error bits allow the indicated errors to be reported(=1), or to be unreported (=0). With some noted exceptions they DO NOT prevent the errors from occurring. The error functions that these bits enable/disable are described in the "Error Handling" section. The bits are as follows:

Error Bit
= = = = = = = = =
Ena_Stolen_Verify_Err – must ALWAYS be negated (=0)
Ena_Maintain_Absent_Err – enable/disable detection
Ena_Maintain_Present_Err – enable/disable detection
Ena_Lock_Addr_Err – enable/disable detection
Ena_Wait_TO_Err – enable/disable detection
Ena_Idle_TO_Err – enable/disable detection

Ena_Rej_Abort_Err - enable/disable detection and mechanism
Ena_Rej_TO_Err
Ena_Check_Err
Ena_Reverse_Err
Ena_Remote_Err

Ena_Stolen_Verify_Err Must ALWAYS be negated (=0). (See: Anticipation Support)

Ena_Maintain_Absent_Err Enables (=1) or disables (=0) the detection of a Maintain_Absent_Error. Disabling this bit allows the Requestor to initiate a Switch transaction and pass through the incorrect T_LOCKOP<1..0> field. (See: "Error Detection and Reporting/FQ Errors")

Ena_Maintain_Present_Err Enables (=1) or disables (=0) the detection of a Maintain_Present_Error. Disabling this bit allows the Requestor to initiate a Switch transaction and pass through the incorrect T_LOCKOP<1..0> field. (See: "Error Detection and Reporting/FQ Errors")

Ena_Lock_Address_Err Enables (=1) or disables (=0) the detection of a Lock_Address_Error. Disabling this bit allows the Requestor to initiate a Switch transaction with a potentially incorrect Physical Route Address. (See: "Error Detection and Reporting/FQ Errors")

Ena_Wait_TO_Err Enables (=1) or disables (=0) the detection of a Wait_TO_Error. Disabling this bit will NOT prevent the Requestor from responding to a Connection Timer overflow in it normal manner. (See: "Error Detection and Reporting/Switch Errors")

Ena_Idle_TO_Err Enables (=1) or disables (=0) the detection of an Idle_TO_Error. Disabling this bit will NOT prevent the Requestor from responding to a Connection Timer overflow in it normal manner. (See: "Error Detection and Reporting/Switch Errors")

Ena_Rej_Abort_Err Enables (=1) or disables (=0) the generation AND detection of an Idle_TO_Error. Disabling this bit will prevent the Requestor from recognizing the state of the pin: M_REJ_ABORT. (See: "Error Detection and Reporting/Switch Errors")

Ena_Rej_TO_Err Enables (=1) or disables (=0) the detection of a Rej_TO_Error. Disabling this bit will NOT prevent the Requestor from responding to a Reject Timer overflow in it normal manner. (See: "Error Detection and Reporting/Switch Errors")

Ena_Check_Err Enables (=1) or disables (=0) the detection of a Checksum Error. (See: "Error Detection and Reporting/Remote Errors")

Ena_Reverse_Err Enables (=1) or disables (=0) the detection of a Checksum Error. (See: "Error Detection and Reporting/Remote Errors")

Ena_Remote_Err Enables (=1) or disables (=0) the detection of a Checksum Error. (See: "Error Detection and Reporting/Remote Errors")

Ena_Quick_Drop Enables (=1) or disables (=0) the Requestor Switch Transmitter to negate Frame as early as possible on an Unlocked operation. (See: "Quick Drop")

Ena_Priority_Promotion Enables (=1) or disables (=0) the Priority Promotion mechanism. (See: "Priority Promotion")

Ena_Interleaver Enables (=1) or disables (=0) the Requestor's detection of the INTERLEAVED pin. (See: "Interleaver Support")

Ena_Reject_Abort Enables (=1) or disables (=0) the Requestor's responding to the REJ_ABORT pin. (See: "Reject Timer")

Ena_Reject_Timer Enables (=1) or disables (=0) the operation of the Reject Timer. This bit will override the Ena_Reject_Abort bit.

Ena_Conn_Timer Enables (=1) or disables (=0) the operation of the Connection Timer.

Ena_Switch_Frame Enables (=1) or disables (=0) the assertion of the REQ_SW_FRAME pin. This function overrides any other function which effects the assertion of the REQ_SW_FRAME pin.

Ena_REQ_BIU Enables (=1) or resets (=0) the Requestor Bus Interface Unit. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_STU, Ena_REQ_SRU, Ena_REQ_CNT. (See: "Disabled Operation")

Ena_REQ_STU Enables (=1) or resets (=0) Requestor Switch Transmit Unit. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_BIU, Ena_REQ_SRU, Ena_REQ_CNT. (See: "Disabled Operation")

Ena_REQ_SRU Enables (=1) or resets (=0) Requestor Switch Receive Unit. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_BIU, Ena_REQ_STU, Ena_REQ_CNT. (See: "Disabled Operation")

Ena_REQ_CNT Enables (=1) or resets (=0) Requestor Counter (Timer) Module. WARNING: MUST ALWAYS HAVE THE SAME STATE AS: Ena_REQ_BIU, Ena_REQ_STU, Ena_REQ_SRU. (See: "Disabled Operation")

Columns_1 Configures the Requestor to expect a 1-column Switch (=1). In this case, the Requestor still uses Columns_2 to determine the Bid construction. When negated (=0), the Requestor uses Columns_2 for

both number of bids to be sent AND bid construction. (See: "Downstream Message Components")

Figure A–41. Bit definition – Requestor_ConfigB.

A.5.1.1.15 Test Registers

The Requestor also contains a test register, Requestor_TestA. This register contains bits that are related to production testing of the SIGA, and unlike all other configuration registers, a read of Requestor_TestA does not yield the data last written. The write bits are initialized in their negated state and are related to production testing of the SIGA. The read bits are used to observe the internal state of the Requestor. They will yield no useful information during normal operation.

WARNING

Write bits of Req_TestA SHOULD NEVER BE ASSERTED DURING NORMAL OPERATION.

The write structure of Requestor_TestA is shown in Figure A–42.

```
Register: Requestor_TestA<31..0>

BIT/FIELD    FUNCTION (write)
=========    ================
     <31>    SPARE
     <30>    TST_CNT_RTP_FORCE_OMSP
     <29>    SPARE
     <28>    TST_CNT_RJT_LOAD_COUNTER

     <27>    TST_CNT_RJT_DECREMENT_COUNTER
     <26>    TST_CNT_COT_LOAD_ENABLE
     <25>    TST_CNT_RSR_BOC_COUNT_DISABLE
     <24>    TST_CNT_RSR_MSK_REG_COUNT_ENABLE

     <23>    TST_CNT_RSR_ADDER_LOAD_ENABLE/RANDOM_CLEAR
     <22>    TST_CNT_RSR_IDLE_DISABLE
     <21>    TST_TIO_RND
   <20..0>   SPARE[21]
```

Figure A–42. Register definition – Requestor_TestA (write).

The function of the bits in Figure A–42 are listed below:

TST_CNT_FORCE_OMSP This bit affects only the real time prescaler. When asserted, this bit will cause the One MicroSecond Pulse signal to be forced high continuously. This will cause other portions of the TM to count unusually quickly.

When not asserted, the Real Time Prescaler will generate OMSP normally every n clock cycles.

TST_CNT_RJT_LOAD_COUNTER

When asserted, this bit will force the continuous loading of the four bit counter within the reject_timeout counter. The counter will be loaded with the initial value specified in the configuration register. When not asserted, the counter is loaded normally, at the beginning of every Requestor operation.

TST_CNT_RJT_DECREMENT_COUNTER

The reject_timeout counter will be decremented on every cycle when this bit is asserted. When both the decrement and load test bits are asserted, the load_counter takes precedence. A reject_timeout is issued from the reject_timer the cycle after the count reaches zero. Under normal operation, with this bit negated, the counter is decremented every n microseconds.

TST_CNT_COT_LOAD_ENABLE

This test bit is used to force the eight bit counter within the connection_timeout unit to load continuously from the configuration registers. With this bit negated, the counter is loaded at the beginning of every Requestor bid transmission.

TST_CNT_RSR_BOC_COUNT_DISABLE

When this bit is asserted, the back_off counter within the random restart/retry logic is disabled, preventing random timeouts from occurring when the counter reaches its terminal count.

TST_CNT_RSR_MSK_REG_COUNT_ENABLE

This bit is used to force the six bit mask_register_out Johnson counter to increment on every cycle. This permits the counter to be incremented without starting a Requestor operation. Under normal operation, this counter is incremented only after n switch rejects have occurred during an attempted Requestor transmission.

TST_CNT_RSR_ADDER_LOAD_ENABLE/RANDOM_CLEAR

This bit controls two testing functions. When asserted, this bit will force the backoff added to load a new value. This permits the adder to be tested without the normal requirement that the Requestor be issued a switch reject for each adder load operation. The other function of this bit is to clear the 12 bit random number generator. This will prevent random numbers from being OR'ed into the backoff counter, simplifying the testing of the module.

TST_CNT_RSR_IDLE_DISABLE

When asserted, this bit will disable the idle state of the Requestor from clearing the backoff adder and Johnson counter. With this bit asserted, the functions of these counters can be tested without starting a

Requestor operation. When de-asserted, with the Requestor in the idle state, the backoff adder is cleared, and the Johnson counter will load the initial state specified in the Requestor configuration registers.

TST_TIO_RND The nine bit random number generator, used in the RQ_BI_TI module to fill the logical route address bits in the Requestor bids, is cleared whenever this bit is asserted. This simplifies the predictions of Requestor bids in testing since random numbers are no longer inserted.

The read structure of Requestor_TestA is shown in Figure A–43.

```
Register: Requestor_TestA<31..0>

BIT/FIELD    FUNCTION (read)
=========    ===============
     <31>    TST_TM_SSR_SLOT_VALID
     <30>    TST_TM_RSR_PTS_ACTIVE
     <29>    TST_TM_RSR_RANDOM_TO
     <28>    TST_TM_RSR_RANDOM_GENERATOR

     <27>    TST_TM_RSR_CARRY_OUT
     <26>    TST_TM_COT_CONNECTION_TIMEOUT
     <25>    TST_TM_RJT_REJECT_TIMEOUT
     <24>    TST_TM_RJT_DECREMENT

     <23>    TST_TM_RTP_Q5
     <22>    TST_SR_REJ_DET
  <21..15>   TST_SR_FSM
     <14>    TST_ST_LOCKED
   <13..1>   TST_ST_FSM
      <0>    TST_ST_RAND_ROUTE
```

Figure A–43. Register definition – Requestor_TestA (read).

The function of the bits in Figure A–43 are listed below:

TST_TM_SSR_SLOT_VALID This bit indicates the state of the slotted start/retry module output SLOT_VALID. This signal may be used by the Requestor to retransmit a bid after a switch reject has put it into the backoff state.

TST_TM_RSR_PTS_ACTIVE The state of the Priority Time Slot signal is readable from this bit. The PTS signal will boost the priority of a bid after a switch reject has been issued.

TST_TM_RSR_RANDOM_TO The Random Timeout from the Random Start/Retry module may be read, which has the same function as the slot_valid signal, depending on the message class of the Requestor operation.

TST_TM_RSR_RANDOM_GENERATOR
This bit reads the MSB of the 12-bit pseudo random number generator inside the backoff counter of the random start/retry module.

TST_TM_RSR_CARRY_OUT This bit is asserted when the backoff adder of the random start/retry module has an overflow, indicating that the Johnson counter will be incremented on the next backoff occurrence of the current Requestor operation.

TST_TM_COT_CONNECTION_TIMEOUT
The connection timeout indicates that the allotted Requestor connection time has expired. If the Requestor is in the WAIT state, the operation will be aborted.

TST_TM_RJT_REJECT_TIMEOUT
The reject timeout indicates that the time permitted for the Requestor to make a connection has expired. When the Requestor is in the backoff state, waiting to retransmit, the operation will be aborted.

TST_TM_RJT_DECREMENT This bit indicates when the reject_timeout module 4-bit counter receives a decrement pulse from the OMSP prescaler.

TST_TM_RTP_Q5 The RTP Q5 register may be read with this bit. This is the MSB of the Real Time Prescaler counter, which generates the OMSP signal. OMSP occurs on the falling edge of this signal.

TST_SR_REJ_DET This is the internal signal generated by the Requestor receiver indicating that a switch reject has been received during a Requestor operation.

TST_SR_FSM These seven bits show the current state of the Requestor receiver. The states, from MSB to LSB are: END, CHECK, BYTE2, BYTE1, W1_B1, ARMED, and IDLE.

TST_ST_LOCKED This bit is asserted whenever the Requestor transmitter is in a locked operation.

TST_ST_FSM These thirteen bits indicate the current state of the Requestor transmitter state machine. The state, from MSB to LSB are: DONE, TO_IDLE, LOCKED_IDLE, LOCKED_DONE, BACKOFF2, BACKOFF1, WAIT, CHECK, DATA, CMD, BID1, HOLD, and IDLE.

TST_ST_RAND_ROUTE This bit reads the MSB of the 9-bit psuedo random number generator used in the logical route address of the Requestor bids.

A.5.1.2 Switch Message Protocol

The Requestor fully generates and supports the Butterfly Switch protocol. That support is described below.

A.5.1.2.1 Physical Route Address Generation

The Switch route address from the T-Bus field, T_AD<33..25>, is actually a logical address. This Logical Route Address, which has two possible sources, undergoes a transformation to derive the Physical Route Address. It is the Physical Route Address which is assembled into the bid symbols of the downstream Switch message. The Logical Route address is used in the calculation of the Header Partial Sum (see the Requestor/Checksum Calculation section). During a given function request, the two possible sources of Logical Route Address for the Requestor are the T-Bus (T_AD<33..25>) and the interleaver port (I_MOD<8..0>). The interleaver port is chosen if: (1) the I_INTERLEAVED pin is asserted on the SIGA during the T-Bus request cycle AND (2) the Enable_Interleave bit in the Requestor_ConfigB register is asserted.

NOTE

It is assumed that both the T-Bus Master making the request and the Interleaver will force any unused bits in Logical Route Address to "0" as it is presented to the pins of the SIGA.

Whichever routing address is actually chosen, that 9-bit quantity undergoes a transformation. It is modified to allow the randomization of a selectable number of the routing bits. The random bits that potentially replace routing bits are obtained from a 9-bit random number generator, the Random Route Generator, which runs at the T-Bus clock rate. A bit in the route address can be specified as random by setting a corresponding bit in the Route Address Mask register to a "1". The transformation for the Physical Route Address generation can be expressed by an equation as shown in Figure A-44.

```
temp<8..0> = MOD & INT & INT_EN
             # [T_SNN & (!INT # !INT_EN)]

 PRA<8..0> = (RAND & RAM) # (temp & !RAM)

     ...where,

     T_SNN     = T_AD<33..25>
     MOD       = I_MOD<8..0>
     INT       = T_INTERLEAVED
     INT_EN    = Req_ConfigB.Ena_Interleaver
     RAND      = RAND<8..0>, random # generator
     RAM       = Route_Address_Mask<8..0>
     PRA       = Physical Route Address
```

Figure A–44. Equation – Physical Route Address generation.

The first equation in Figure A–44 represents the selection of either the Interleaver port or the T–Bus port for the Logical Route Address. The second equation randomizes selected bits in the Logical Route Address. The Route Address Mask is located in the Req_ConfigB configuration register.

A.5.1.2.2 Downstream Message Components

Some of the relevant aspects of the downstream Switch message components are now discussed. For a more detailed explanation of Switch message definition and protocol, see the reference documents.

A.5.1.2.2.1 Header

The construction of the message header, which contains the bid symbols, varies depending on the modulus of the Switch, which can be either 8 or 16. The SIGA design will support both options, although the modulo–8 Switch is the most likely to be encountered. In addition, the Requestor can support a one, two or three column Switch. Figure A–45 shows the format of the bid symbols in both modulus configurations. As seen from Figure A–45, certain bid symbols may never be sent if the Switch is small enough. Note that a modulo–8 switch is always expected to have at least two switch columns and a modulo–16 can have as few as one. The random bits mentioned in Figure A–45 are obtained from a separate random number generator known as the Random Route Generator.

```
7                                   0
|                                   |
0  0  P1  P0  Rd  Rc  Rb  Ra  (BID 1)  (first sent)
0  0  P1  P0  Rd  Rc  Rb  Ra  (BID 2)       V
0  0  P1  P0  Rd  Rc  Rb  Ra  (BID 3)  (last sent)

...where,

P1..P0 = priority from T-Bus: PRIORITY<1..0>
Ra..Rd = Physical Route Address (see below...)
```

COL1	COL2	MOD8	BID1 Rd Rc Rb Ra	BID2 Rd Rc Rb Ra	BID3 Rd Rc Rb Ra
0	0	0	n2 n1 n0 R8	R7 R6 R5 R4	R3 R2 R1 R0
0	0	1	0 R8 R7 R6	0 R5 R4 R3	0 R2 R1 R0
0	1	0	R7 R6 R5 R4	R3 R2 R1 R0	----------
0	1	1	0 R5 R4 R3	0 R2 R1 R0	----------
1	0	0	n2 n1 n0 R8	----------	----------
1	0	1	0 R8 R7 R6	----------	----------
1	1	0	R7 R6 R5 R4	----------	----------
1	1	1	0 R5 R4 R3	----------	----------

```
...where,

COL2        = Requestor_ConfigA.Columns_2
COL1        = Requestor_ConfigA.Columns_1
MOD8        = Requestor_ConfigA.Modulo_8
n1,n2,n3    = random bits
---------   = Bid is NOT transmitted
```

Figure A–45. Bit definition – downstream message header.

A.5.1.2.2.2 Body

The message body, which contains the command, address, data and checksum bytes, varies based on the type of message being sent downstream. The general format is shown in Figure A–46. Figure A–46, of course, shows a single word write message. For multi–word write transfers there would be correspondingly more data bytes. For a read message, the difference would be that all data fields would be missing and bit S would be forced to a zero.

```
7                                                   0
|                                                   |
L1   L0   R1   R0   S2   S1   S0   A24  (first sent)
A23  A22  A21  A20  A19  A18  A17  A16        |
A15  A14  A13  A12  A11  A10  A9   A8         |
A7   A6   A5   A4   A3   A2   A1   A0         |
D31  D30  D29  D28  D27  D26  D25  D24        |
D23  D22  D21  D20  D19  D18  D17  D16        |
D15  D14  D13  D12  D11  D10  D9   D8         |
D7   D6   D5   D4   D3   D2   D1   D0         |
                    .                         |
   <possible additional write words>          |
                    .                         |
                                              V
F    0    0    S    CS3  CS2  CS1  CS0  (last sent)

...where,

L1..L0   = lock operation from T-Bus: T_LOCKOP<1..0>
R1..R0   = portion of field from T-Bus: T_RR<1..0>
           R1  R0
           ==  ==
           0   0    write
           0   1    read
           1   0    <unused>
           1   1    <unused>
S2..S0   = size information from T-Bus: T_SIZE<2..0>
A24..A0  = address information from T-Bus: T_AD<24..0>
D31..D0  = data information from T-Bus: T_AD<31..0>
F        = enable forward drivers
           F = 0   disable forward drivers next clock
           F = 1   enable forward drivers next clock
S        = Stolen Bit
CS3..CS0 = message checksum
```

Figure A–46. Bit definition – downstream message body (write).

NOTE

The current SIGA design ALWAYS forces the "F" bit to be a "0".

A.5.1.2.3 Checksum Support

The Requestor and Server each have two separate units of checksum logic. The first, known as the Transmit Checksum Unit, calculates the message checksum during its

transmission. The second, known as the Receive Checksum Unit, calculates and verifies the checksum for the incoming message.

The elements included in the calculation of the checksum of a downstream message vary depending on the type of message being transmitted. For any initial message (locked or unlocked), the Requestor always initializes its Transmit Checksum Unit with the "flash" sum of the Logical Route Address. The Logical Route Address can, of course, come from either the MOD pins (interleaved access) or from the T-Bus (non-interleaved). For any locked messages, the Requestor always initializes its Transmit Checksum Unit to zero.

In the same way, the downstream Server must initialize its Receive Checksum Unit to ITS node checksum whenever it expects an initial message. This initialization value will, of course, match that calculated by a Requestor about to transmit to that Server's node. For locked messages, the Server will initialize its Receive Checksum Unit to zero, just as the Requestor does with its Transmit Checksum Unit.

In an upstream message, there are NEVER any routing bits to contend with. Therefore, the downstream Server always initializes its Transmit Checksum Unit to zero, as does the Requestor's Receive Checksum Unit.

A.5.1.2.4 Checksum Calculation

The checksum for a downstream message is actually calculated in two parts. If the message is an initial (locked or unlocked) one, a partial sum of the message header is calculated (by separate logic) and stored in the Transmit Checksum Unit. Then, the Transmit Checksum Unit adds (exor's) the initial value, if any, to the bytes of the body of the message as it is transmitted.

A.5.1.2.4.1 Header Partial Sum

The header partial sum is derived by considering only the Logical Route Address bits. This means that the priority and random bits are not included in the calculation. This approach eases the design of the checksum logic and makes it independent of the Switch modulus. The equation for this calculation is shown in Figure A-47.

```
HPS<3> = R8 $ R7 $ R3
HPS<2> = R6 $ R2
HPS<1> = R5 $ R1
HPS<0> = R4 $ R0

...where,

HPS<3..0>  = Header Partial Sum
R8..R0     = Logical Route Address
```

Figure A-47. Equation - Requestor header partial sum calculation.

A.5.1.2.4.2 Message Checksum

As previously mentioned, the header partial sum is added (exor'ed) to the body of a downstream message if and only if that message is an initial message. The message checksum calculation is shown in Figure A–48. Figure A–48 shows the calculation for a single word write message. For write messages with more words, those bytes would be included in the same manner as the data bytes in the figure. For read messages, the data field would be missing entirely from the calculation.

```
CS<3> = HPS<3> $ exor(L1,S2,A23,A19,A15,A11,A7,A3,
                      D31,D27,D23,D19,D15,D11,D7,D3,F)

CS<2> = HPS<2> $ exor(L0,S1,A22,A18,A14,A10,A6,A2,
                      D30,D26,D22,D18,D14,D10,D6,D2,0)

CS<1> = HPS<1> $ exor(R1,S0,A21,A17,A13,A9,A5,A1,
                      D29,D25,D21,D17,D13,D9,D5,D1,0)

CS<0> = HPS<0> $ exor(R0,A24,A20,A16,A12,A8,A4,A0
                      D28,D24,D20,D16,D12,D8,D4,D0,S)

...where,

exor'ed components from: "Bit Definition - Message Body"
CS<3..0>  = message checksum
HPS<3..0> = Header Partial Sum
```

Figure A–48. Equation – message checksum (see text).

NOTE

The "F" field is always "0".

A.5.1.2.5 T–Bus Interface

The Requestor supports the standard T–Bus protocol with some small limitations. For one, the Requestor does NOT support unaligned transfers which fall across word (32-bit) boundaries. In addition, when it is locked to a T–Bus Master and in its "WAIT" state, the Requestor will always issue a REFUSED LOCKED to ANY T–Bus query while it is busy processing a split–cycle request. This means that it will even REFUSED LOCKED to its own T–Bus master! This is a hardware optimization which should cause no problems. The locking T–Bus master normally has no reason to query the Requestor until the Requestor finishes its current operation.

Figure A–49 shows the Requestor's state–dependent T–Bus responses while it is in some of its more interesting states.

```
NEXT RESPONSE                CONDITION
=============  ================================================
State = IDLE  (satisfied a function request, waiting for new one):
 PROMISE          !LOCKED & !DROP_LOCK &  read
 PROMISE          !LOCKED & !DROP_LOCK &  write & !multi
 MORE             !LOCKED & !DROP_LOCK &  write &  multi
 REFUSED          !LOCKED &  DROP_LOCK
 REFUSED LOCKED    LOCKED & !DROP_LOCK & !my_master
 PROMISE           LOCKED & !DROP_LOCK &  my_master &  read
 PROMISE           LOCKED & !DROP_LOCK &  my_master &  write & !multi
 MORE              LOCKED & !DROP_LOCK &  my_master &  write &  multi
 REFUSED           LOCKED &  DROP_LOCK

State = WAIT  (waiting for function request to traverse Switch)
    -or-
State = BREQ  (making T-Bus request for T-Bus with split response):
 REFUSED          !LOCKED
 REFUSED LOCKED    LOCKED
```

Figure A-49. Requestor T-Bus responses (partial list).

A.5.1.2.6 LCON Interface

The LCON is a the physical and logical link between the SIGA-Requestor and the "input" port of the Switch Gate Array (SGA). In other words, for the SIGA, the LCON interface is the logical Switch interface. The LCON provides the Requestor with: 1) level conversion to and from the ECL levels of the SGA and 2) reclocking of data, Frame, Reverse and the 65 ms pulse to and from the SGA.

Figure A-50 shows the Requestor's LCON (Switch) Interface Pins.

PIN NAME	TYPE	FUNCTION
R_DATA<7..0>	bidirectional	Requestor-LCON data bus
R_FRAME	output	Frame output to Switch
R_REVERSE	input	Reverse input from Switch
R_NENA_BACK	output	LCON TTL driver enable
M_SIXTY_FIVE	input	65 ms timer input

Figure A-50. Requestor LCON (Switch) interface pins.

A.5.1.2.6.1 Data Bus Enable Control

The Requestor controls the enables of both its own output drivers and the LCON's output drivers to the SIGA-LCON data interface - R_DATA<7..0>. To control its own output drivers, the Requestor generates an internal signal called, nena_out. When asserted (=0), nena_out enables the Requestor's R_DATA<7..0> drivers. To control

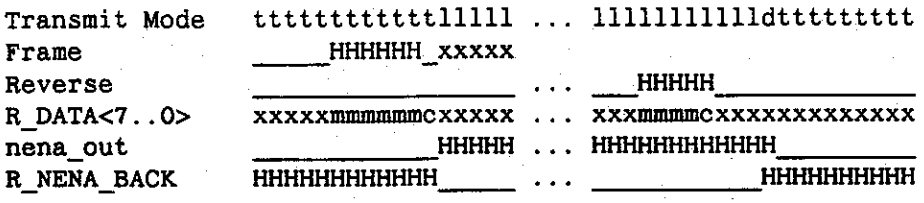
the LCON, the Requestor provides the R_NENA_BACK signal to directly enable(= 0)/disable(= 1) the LCON's output drivers to R_DATA<7..0>. In addition, R_NENA_BACK, after a flip-flop delay, is used to enable/disable the LCON's Switch data ECL interface bus. When the Requestor is driving R_DATA<7..0>, it is in "Talk" Mode. When the LCON is driving that bus, the Requestor is in "Listen" Mode.

There are two major reasons why the Requestor separately provides the R_NENA_BACK signal. First, the Requestor already "knows" which direction the bus should be driving, and therefore this logic need not be repeated in the LCON. Second, this configuration gives the Requestor the ability to prevent bus contention.

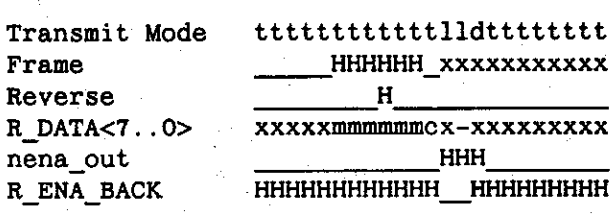
Bus contention can occur when the direction of data changes on the LCON interface. If R_NENA_BACK changed on the same clock edge as nena_out, there would be contention on R_DATA<7..0> each time both of those signals changed. However, because of timing skew and minimum delays, contention is actually only a problem when the Requestor tries to enable its own drivers as it disables the LCON's backward drivers. This occurs during the transition from Listen to Talk Mode. But since the Requestor has separate control of its own output drivers and the LCON's, it can prevent this case of contention. It does this by inserting a "dead" state for one Switch Interval where neither the Requestor nor the LCON is driving R_DATA<7..0>.

The Requestor is considered "quiescent" when it is not transmitting messages and not waiting for any replies. When quiescent, the Requestor is in Talk Mode. The Requestor tries to stay in Talk Mode whenever possible, making the transition to Listen only for the absolute minimum time necessary. This situation is the mirror image to the Server. It is always in Listen Mode when quiescent and tries to stay in Talk Mode for as little time as possible.

When the Requestor finishes transmitting the checksum of an Initial or Locked message, it transitions directly into Listen Mode. Once there, it waits for either a Reject (which could have been detected and latched during the message transmission) or a return message. When either of those two events are complete, the Requestor transitions back to the Talk Mode, via the dead state. Figure A-51 shows this sequence for both a replied and a rejected Switch message. Note from Figure A-51 that there is a dead state only when making a transition from Listen to Talk Mode. Although not shown in the figure, subsequent Locked messages act in the exact same manner.



(a) Message Returned, No Reject



(b) Reject Latched during Tx

...where,

m..m	is a message
c	is the checksum
t	is Talk Mode
l	is Listen Mode
d	is the dead state
-	floating bus

Figure A–51. Timing – Requestor switch data bus enable.

A.5.2 Server

The Server is described from the point of view of its overall operation and its two major interfaces: the T-Bus interface and the Switch Interface.

A.5.2.1 Operation

The operation of the Server is described by discussing its major functions.

A.5.2.1.1 Overview

The Server is a local T-Bus master which creates a logical coupling to a physically remote T-Bus slave via the Switch. The Server acts as the "responder" of this coupling on the Switch and thus can be thought of as a "master" on the T-Bus but a "slave" to the Switch. Referring to Figure A–52, the Server contains three major functional units: Bus Interface Unit (BIU), Switch Tx Unit (STU), and the Switch Rx Unit (SRU). The BIU is clocked by the T-Bus clock and both the STU and SRU are clocked by the Switch clock. Interfacing of control signals between these units is accomplished with handshake synchronizers, as shown. The SRU receives function requests from the

Switch and translates those requests into commands for the BIU. The BIU handles all of the T-Bus transactions of the Server to comply with a given function request. When a T-Bus slave device responds to a function request, the BIU picks-up that response and passes it as a command to the STU. The STU then initiates an upstream Switch message to return the function response.

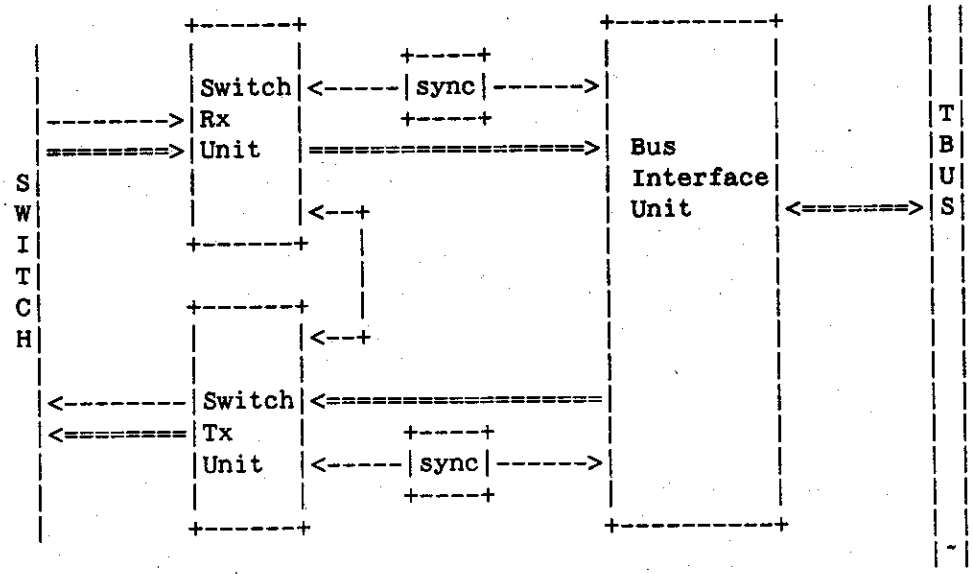


Figure A-52. Server block diagram.

The SRU detects the downstream message of a function request, verifies the checksum and alerts the BIU of the incoming message and the checksum status. The SRU also causes Switch rejects when either the BIU has explicitly commanded this action or when the SRU decides to on its own. The BIU will command a Switch reject when a function request is trying to access a T-Bus device which is locked to a T-Bus device other than the Server. The SRU will NOT initiate a reject without a command from the BIU and thus CANNOT correctly handle a non sequitur downstream message. A non sequitur would occur, for instance, when the SRU receives a function request (in the form of a downstream message) and knows that the STU has not even begun to send an upstream Switch message in response to the last function request.

The SRU has the additional responsibility of initiating a FREE-LOCKS command to the BIU when the Switch path is locked and the incoming Frame signal negates unexpectedly. This situation is known as "dropping a lock" and is the ONLY time when the Server does not create a Function Response as a result of an explicit function request.

The SRU/BIU interface is a streamlined request/response type interface where for each SRU request there is an BIU response. The SRU presents an encoded function request to the BIU and sets an "execute" flag. When the BIU is done operating on that request, it sets a "done" flag and returns a status code and data to the SRU. The SRU also has the ability to "interrupt" the pending BIU operation. This is accomplished with a "terminate" handshake signal from the SRU. The "terminate" handshake receives a "terminate-done" from the BIU when the BIU finishes. This "interrupt" path

is used for situations where the BIU may be indefinitely "hung" because a failed T-Bus slave is continuously asserting Slave pause.

Both the SRU and BIU are responsible for handling their own functions independently and they have very little real-time knowledge of each other's state. This approach simplifies the Server design and carries the request/response philosophy throughout the system.

The BIU has three major responsibilities: (1) initiate T-Bus requests to comply with a command from the SRU; (2) receive responses from the T-Bus; (3) transfer those responses, along with any error indications, to the STU. To accomplish the T-Bus request/response transfer, the BIU supports most of the T-Bus protocol.

The STU is a fairly simple device. It acts on a function response from the BIU and initiates the upstream Switch message to carry out that response. The STU also is responsible for assembling and transmitting the data in an outgoing message.

A.5.2.1.2 Anticipation Support

The operation of the Server has two main goals: (1) to pass a downstream Switch function request to a T-Bus slave as quickly and efficiently as possible, and (2) to return the corresponding function response from that T-Bus slave as quickly and efficiently as possible. Certain techniques can be used to take advantage of the expected operation of the logic in the function request and response path. These techniques are known collectively as "anticipation". The use of anticipation in achieving the two main goals of the Server are now discussed.

A.5.2.1.2.1 Function Requests

Maximizing downstream function request efficiency in the Server involves balancing the desire for speed with the desire for eliminating unwanted side-effects. The speed issue relates to the desire to transfer data from an incoming Switch message to the T-Bus as soon as it is available. Unwanted side-effects involve taking any action on the T-Bus that would cause a change in stored data in a T-Bus slave device given that the downstream message was corrupted. Two extreme approaches could be taken in the design of the Server. First, the Server could wait until the entire downstream message had been received, including the checksum; verify the checksum; and then begin access to the T-Bus. Second, the Server could begin access to the T-Bus immediately upon receiving a downstream message.

The first approach would cause the Server to waste valuable time in accessing the T-Bus, and the second could possibly cause unwanted side effects. Since one of the design goals of the Butterfly II is that data integrity should take precedence over speed, a compromise between the first and second approaches is implemented in the Server.

The Server "anticipates" the verification of the downstream checksum and begins it's request for T-Bus drivership. The timing is set up such that the Server BIU is commanded by the SRU to make a bus request at a specific moment in time. In fact, the

SRU commands the BIU (input to the BIU synchronizer) to begin the T-Bus request EXACTLY five Switch intervals before the "Checksum_is_OK" signal is valid. This is true for both reads and writes. Therefore, the synchronizer setting, Server_ConfigA.BIU_Xfer_Sync<3..0> should be set accordingly. See "Synchronizer Settings" for more details.

A.5.2.1.2.2 Function Responses

The Server uses a similar technique as the Requestor for anticipating T-Bus transactions. Of course, in the case of the Server, the anticipation is for Function Responses rather than Function Requests. The Server_ConfigA.Multi_Head_Start<1..0> register is used to set the anticipation for multi-word writes. Figure A-53 illustrates its settings.

```
Register: Server_ConfigA.Multi_Head_Start<1..0>

10    Wait until...
==    =============
00    all words are transferred
01    three words have been transferred
10    two words have been transferred
11    one word has been transferred
```

Figure A-53. Register definition – Server_ConfigA.Multi_Head_Start<1..0>.

In addition, the Server_ConfigA.Ena_Byte_Head_Start bit, when asserted (=1), begins anticipation whenever the T-Bus Slave responds with EARLY-ACK.

Normally, the Server will anticipate for reads only. However, in some hardware configurations it is possible to anticipate on writes. When Server_ConfigB.Ena_Wr_Head_Start is asserted (=1), the Server treats writes exactly the same way as reads for all purposes.

WARNING

Using anticipation in multi-word writes can cause unusual side-effects if the multi-word write does not complete in time. This is because the Server SRU may mistakenly believe that the write data buffers are actually stable until the upstream Requestor has seen the Function Response and taken some action. As seen by the Server, this response takes quite long, at least 4–6 Switch Intervals. Thus, if the multi-word write takes only this long to complete, there is no problem.

WARNING

Using read anticipation requires that the T-Bus Slave issue an ERROR before transferring any data.

NOTE

The EARLY-ACK response has no meaning for multi-word reads or writes, and this response is ignored by the Server. Also, the Server must examine the T_RR field even though T_SPAUSE may be asserted.

A.5.2.1.3 Locked Sequences

The Server's handling of locked sequences parallels that of the Requestor and is described in the "Requestor/Operation/Locked Sequences" section. Like the Requestor, the Server's locked sequence has three distinct events: opening, maintaining and dropping.

The Server becomes locked if and only if it receives an Initial Locked message (OPEN, by definition is the command). It remains locked as long as it returns any function response except Reject. When a lock is dropped at the upstream Requestor, Frame is negated. As mentioned in the "Requestor/Operation/Locked Sequences" section, a Requestor drop-lock function request can occur as the result of a T-Bus master issuing a FREE-LOCK or possibly a Requestor Switch Class error. The Server NEVER knows the reason for the drop-lock request, it simply issues the perfunctory FREE-LOCK to a T-Bus slave.

A.5.2.1.4 Stolen Bit Support

Because of the structure of the Switch message format, only one bit of Stolen information can be transferred between upstream and downstream nodes during a given message. Therefore, during byte reads, the Stolen bit from the Server's T-Bus is transported to the upstream Requestor exactly as it is read from T_AD<32> during the data transfer cycle of the T-Bus. For multi-word reads, the Server continues the T-Bus transaction, reading and storing all of the intended words even when it encounters a Stolen bit BEFORE the last word of the transfer.

However, when the Server finally transmits that data to the upstream Requestor, it acts differently depending on whether or not the data contains a Stolen bit. If it does not, all of the multi-word data is included in the upstream message and the Stolen bit in the Checksum byte is sent negated. If it does, the Server ends transmission of the data AFTER it sends the Stolen word, and it asserts the Stolen bit in the Checksum byte.

The upstream Requestor always assumes that the words of a multi-word transfer are NOT Stolen until it encounters an asserted Stolen bit in the Checksum byte. When this occurs, the LAST word and only the last word received by the Requestor is assumed to be Stolen.

For byte write transfers, the Server presents the state of the Stolen bit in the downstream Checksum byte to the downstream T-Bus bit, T_AD<32>. For multi-word writes however, the state of ALL Stolen bits transported downstream is assumed by the Server to be "0". In this case, the Server will ignore the state of the Stolen bit in the downstream Checksum byte.

A.5.2.1.5 Error Reporting

Errors delivered by the Server (Requestor "Remote Error" Class) are transported by the Server to the upstream Requestor via the function response Switch message. Those errors may have one of two sources: they could originate from the Server itself, or they could be errors passed to the Server from a downstream Slave. The error codes due to the Server are shown in Figure A-54. Their definitions are shown in Figure A-55. Other remote slave errors are described in other system documents.

```
Server Error Codes:

7       0
|       |
PPPPPPba

b a  Server Error
= =  ==================
0 0  Downstream_Refused
0 1  Downstream_Write
1 0  Downstream_Late
1 1  Downstream_OTL

...where,

   P..P = Server_ConfigA.Error_Prefix<5..0>
```

Figure A-54. Server remote error codes and definitions.

Downstream_Write	A downstream write error was detected from a T-Bus Slave while the downstream Server was sourcing data. Because of the direction of the data bus, the Server cannot return the actual error code.
Downstream_OTL	A downstream T-Bus Slave did not respond to the Server's request. Specifically, the Slave did not assert T_DRIVEN in the T-Bus cycle following the Server's T-Bus request.
Downstream_Late	A downstream T-Bus slave responded with a LATE ERROR.

Downstream_Refused A downstream T-Bus slave responded with REFUSED-LOCKED when the Server thought itself to be locked.

Figure A-55. Server remote error definitions.

A.5.2.1.6 Disabled Operation

The Server can be disabled via a number of bits in the Server_ConfigB register. These include: Ena_BIU and Ena_SRU. These bits reset the two major blocks of the Server.

WARNING

In normal operation, these bits SHOULD ALWAYS BE ASSERTED/NEGATED AT THE SAME TIME. Otherwise, erratic Server operation may result.

A.5.2.1.7 Configuration Registers

The Server has two general Configuration Registers, known as Server_ConfigA and Server_ConfigB, which are used to set miscellaneous parameters and enable/disable certain functions. The structure of Server_ConfigA is shown in Figure A-56. The bit definition of Server_ConfigA is shown in Figure A-57. This register contains mostly configuration bits that affect the run-time parameters of the Server. All bits are "high-true" and are reset (low) upon system reset. The structure of Server_ConfigB is shown in Figure A-58. The bit definition of Server_ConfigB is shown in Figure A-59. This register contains mostly configuration bits that affect the run-time parameters of the Server. All bits are "high-true" and are reset (low) upon system reset.

```
Register: Server_ConfigA<31..0>

BIT/FIELD    FUNCTION (read/write)
=========    ====================
     <31>    Ena_Wr_Head_Start
     <30>    Ena_Byte_Head_Start
   <29..28>  Multi_Head_Start[2]
   <27..24>  RX_Init_CS[4]
   <23..18>  Error_Prefix[6]
     <17>    Ena_BIU
     <16>    Ena_SRU
   <15..12>  STU_Freed_Sync[4]
    <11..8>  STU_Done_Sync[4]
     <7..4>  BIU_Free_Sync[4]
     <3..0>  BIU_Xfer_Sync[4]
```

Figure A-56. Register definition – Server_ConfigA.

Ena_Wr_Head_Start	Enables the Server to anticipate during write-type Function Responses (=1). Otherwise, anticipation will only occur for read-type Function Responses. (See: "Anticipation Support")
Ena_Byte_Head_Start	Enables the Server to anticipate during byte-type Function Responses (=1). Otherwise, anticipation will not occur for byte-type Function Responses (=0). (See: "Anticipation Support")
Multi_Head_Start[2]	Configures the Server for the desired Function Response Anticipation for all multi-word operations. (See: "Anticipation Support")
RX_Init_CS[4]	Configures the initial checksum for Initial Messages. NOTE: This register must contain the logical INVERSE of the initial checksum. (See: "Checksum Calculation")
Error_Prefix[6]	Configures the Prefix (T-Bus bits: D7-D2) of the Error code response for Server error. (See: "Error Handling")
Ena_BIU	Enables the by releasing its reset signal (=1). Otherwise, the BIU will be held in reset (=0). (See: "Disabled Operation")
Ena_SRU	Enables the SRU by releasing its reset signal (=1). Otherwise, the SRU will be held in reset (=0). (See: "Disabled Operation")
STU_Freed_Sync[4]	Configures the settling time of the Switch Transmit Unit's (STU) handshake synchronizer which receives a "freed" signal from the Bus Interface Unit (BIU). This signal indicates that the BIU has acted on a previous "free" command from the SRU. (See: "Synchronization")
STU_Done_Sync[4]	Configures the settling time of the Switch Transmit Unit's (STU) handshake synchronizer which receives a "done" signal from the Bus Interface Unit (BIU). This is used to indicate completion of a Function Request. (See: "Synchronization")
BIU_Free_Sync[4]	Configures the settling time of the Bus Interface Unit's (BIU) handshake synchronizer which receives a "free" signal from the Switch Receive Unit (SRU). This is used to issue a FREE-LOCK. (See: "Synchronization")
BIU_Xfer_Sync[4]	Configures the settling time of the Bus Interface Unit's (BIU) handshake synchronizer which receives a "xfer" from the Switch Receive Unit (SRU). This is used to initiate a Function Request. (See: "Synchronization")

Figure A-57. Bit definition - Server_ConfigA.

```
Register: Server_ConfigB<31..0>

BIT/FIELD   FUNCTION (read/write)
=========   =====================
  <31..8>   not used
   <7..6>   spare
      <5>   Dis_Frame
      <4>   Ena_SOC
      <3>   Dis_Check_Err
   <2..0>   SER_Slave_Num[3]
```

Figure A–58. Register definition – Server_ConfigB.

Dis_Frame	Disables the SRU by forcing it to see the incoming Frame negated, regardless of its actual state (= 1). Otherwise, the SRU will see the actual incoming Frame (= 0). (See: "Disabled Operation")
Ena_SOC	Enables the SRU to recognize the start of a new connection (= 1). Otherwise, the SRU will ignore this event (= 0). (See: "Disabled Operation")
Dis_Check_Err	Disables the detection of checksum errors (= 1). Otherwise, the detection is enabled (= 0). (See: "Checksum Calculation)
SER_Slave_Num[3]	Configures the Slave number that the Server will place on the T_SOURCE<2..0> pins when it is making a T-Bus Function Request.

Figure A–59. Bit definition – Server_ConfigB.

A.5.2.1.8 Test Registers

The Server contains a read–only test register which should NEVER be accessed during normal operation. Figure A–60 shows the structure of that register which is used mostly for observing internal states. Figure A–61 shows the bit definition of SOME of the bits in the Server_TestA register.

```
Register: Server_TestA<31..0>

BIT/FIELD   FUNCTION (read-only)
=========   ====================
     <31>   <unused>
     <30>   SRU believes it is locked
     <29>   SRU refusing new connections
     <28>   Synchronized "Enable New SOC's"
     <27>   SRU "Should be Checksum"
     <26>   SRU Checksum OK signal
     <25>   SRU Anticipation Signal
     <24>   Checksum errors occurred
  <23..20>  <unused>
  <19..16>  Running Version of Rx Checksum

   <15..8>  Internal State of SRU FSM
       <15>    SRU has seen Reverse come and go
                  and has seen Frame go away.
                  Transition to 9, 10, or 13 will occur
       <14>    SRU has seen first Reverse and is waiting
                  for the end of the Reverse transmission
       <13>    SRU is waiting for lock to be FREE-LOCKed
       <12>    SRU is waiting for first Reverse
       <11>    SRU receiving Checksum byte
       <10>    SRU receiving a command
        <9>    SRU is idle
        <8>    Bad SOC seen (low true)
```

Figure A-60. Register definition – Server_TestA.

SRU believes it is locked The BIU will issue a FREE-LOCKS request if Frame is negated for more than one Switch Interval.

SRU refusing new connections Indicates that there is no active connection and that new connections will be refused (with Reject). The SRU IS currently and WILL be idle until re-enabled. (See: "Disabled Operation")

Synchronized Enable New SOC's
The synchronized version of Server_ConfigB.4. The programmer should check this bit before assuming that the SRU will Reject or accept new connections. (See: "Disabled Operation")

SRU "Should be Checksum" Indicates that the Checksum should have arrived. This is used in conjunction with the "SRU Anticipation Signal" to determine if the SRU is properly anticipating the reception of the Checksum byte.

SRU "Checksum OK" Indicates to the BIU that the T-Bus operation should, in fact, take place.

SRU Anticipation Signal Indicates to the BIU that it should begin the T-Bus request. See SRU "Should be Checksum" above.

Checksum errors occurred Indicates that a checksum error did occur sometime in the past. This bit is negated whenever Server_ConfigB.4 is negated.

Figure A-61. Bit definition - Server_TestA.

A.5.2.2 Switch Message Protocol

The Server fully generates and supports the Butterfly Switch protocol. That support is described below.

A.5.2.2.1 Upstream Message Components

Unlike the Requestor, the Server never has to create a message header with routing information because the return path to the upstream Requestor has already been established. The Server need only return a checksum with data and/or error code information. Figure A-62 shows a typical upstream Server message as a response to a word-read function request. The significance of the "E" and "S" bits are described in: "Stolen and Error Messages". The upstream message body for a write is always of the same format whether the function request was multi-word or non-multi word. Figure A-63 shows a typical upstream Server message as a response to a word write Function Request. The significance of the "E" and "S" bits are described in: "Stolen and Error Messages".

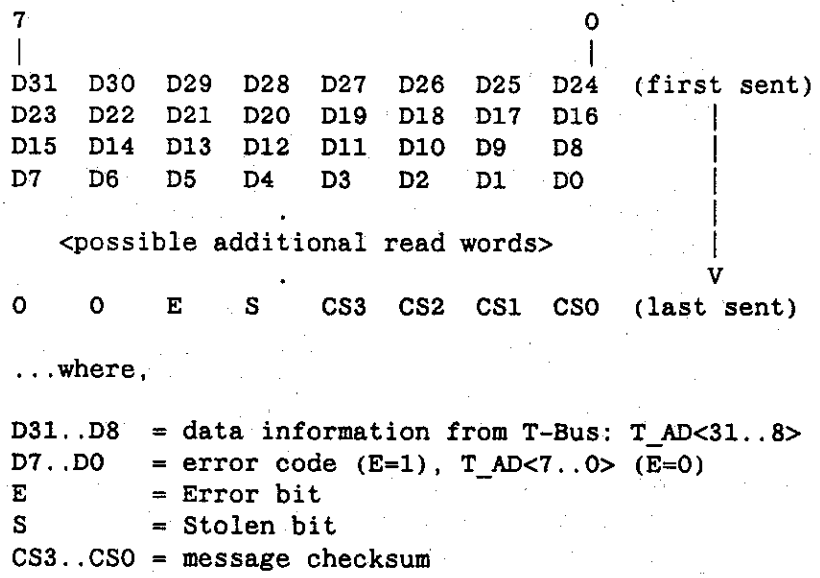


Figure A-62. Bit definition - upstream message body (read).

```
7                                       0
|                                       |
D7   D6   D5   D4   D3   D2   D1   D0   (first sent)
0    0    E    S    CS3  CS2  CS1  CS0  (last sent)

...where,

D7..D0   = error code (E=1), unknown (E=0)
E        = Error bit
CS3..CS0 = message checksum
```

Figure A-63. Bit definition – upstream message body (write).

A.5.2.2.2 Stolen and Error Messages

When the Upstream Read message has Stolen and/or Error bits asserted in the checksum, their presence modify the meaning of the message byte (or bytes) PRECEDING the checksum byte. In the case of an asserted (= 1) Stolen bit, the Server is indicating that ONLY the previous four bytes are stolen. This is consistent with what can happen on the T-Bus side of the Server. There, a T-Bus Slave may happen to return a Stolen data word which is not necessarily the last word of the read operation. The Server's BIU will continue to read any data "past" the Stolen word, but its STU will always END transmission of the Upstream Switch Message on the Stolen word – ignoring the rest. The consequence for the Upstream Requestor is that the "S" bit always modifies the LAST word received. The "S" bit has no meaning for Upstream write messages and is ignored.

When the Error bit is asserted (= 1) during an Upstream Read message, the Server is indicating that the byte immediately PRECEDING the Checksum contains the Error Code and that any other bytes in the message are "garbage" data. The T-Bus protocol demands that all Slaves respond with "ERROR" during the FIRST word transfer and that an "ERROR" response ends the T-Bus transfer. Therefore, an Upstream Read Message with E = 1 will only contain one word of data. Assertion of the "E" bit has higher priority than assertion of the "S" bit, so they will never be asserted simultaneously in a given Upstream message.

Figure A-64 shows a summary of the effect of the "E" and "S" bits on an Upstream Message.

```
E S   previous byte is...
= =   ====================
0 0   Data byte, previous word is NOT stolen (reads only)
0 1   Data byte, previous word is stolen (reads only)
1 0   Error Code (reads or writes)

Note: the value ES = 11 will never occur
```

Figure A–64. Interpretation of checksum E and S bits.

A.5.2.2.3 Upstream Message Types

The previous discussions about message formats can be brought together to produce an enumeration of the possible Upstream Message types. This summary is shown in Figure A–65.

TYPE	#WORDS	STOLEN or ERRORS	RETURN MSG FORMAT
write	any	none	XC
	"	any error	ZC
read	non-multi	none	DDDDC
	"	either on word1	DDDEC
	two-words	none	DDDDDDDDC
	"	either on word1	DDDEC
	"	stolen on word2	DDDDDDDDC
	three-words	none	DDDDDDDDDDDDC
	"	either on word1	DDDEC
	"	stolen on word2	DDDDDDDDC
	"	stolen on word3	DDDDDDDDDDDDC
	four-words	none	DDDDDDDDDDDDDDDDC
	"	either on word1	DDDEC
	"	stolen on word2	DDDDDDDDC
	"	stolen on word3	DDDDDDDDDDDDC
	"	stolen on word4	DDDDDDDDDDDDDDDDC

```
NOTE: Frame is high for entire return message.

X = don't care
Z = always an Error Code
E = Error Code (Checksum bit 5 = 1)
  = Data Byte (Checksum bit 5 = 0)
C = Checksum Byte
```

Figure A–65. Upstream message types.

A.5.2.2.4 Checksum Calculation

Checksum support for the Server is described in the "Requestor/Operation/Checksum Calculation" section. The actual calculation performed by the Server is shown in Figure A-66. Figure A-66 shown the calculation for a single word read message. For read messages with more words, those bytes would be included in the same manner as the data bytes in the figure. For write messages, the data field would be missing entirely from the calculation and only the error byte would be included.

```
CS<3> =  exor(D31,D27,D23,D19,D15,D11,D7,D3,0)

CS<2> =  exor(D30,D26,D22,D18,D14,D10,D6,D2,0)

CS<1> =  exor(D29,D25,D21,D17,D13,D9,D5,D1,E)

CS<0> =  exor(D28,D24,D20,D16,D12,D8,D4,D0,S)

...where,

        CS<3..0>  = message checksum
```

Figure A-66. Equation - message checksum (single-word read, see text).

A.5.2.2.5 Rejects

A Reject is the assertion of Reverse for exactly one Switch Interval. Rejects are not, strictly speaking, messages; because the Switch data pins do not carry any known data. The Server produces a Reject (assertion of Reverse for only one Switch Interval) in either of three conditions: 1) An addressed downstream T-Bus slave is found to be locked during an Initial Switch Message, 2) The Server has been configured to reject all Downstream messages, or 3) The Server's SRU state machine is busy while trying to return to its "idle" state.

During the Initial Switch message, the targeted Downstream device may, in fact, be locked to a device other than the Server. The Server issues a Reject to indicate this fact to the Upstream Requestor. Once the Server has successfully locked some device, it is still possible for a Locked Message to attempt an access to device other than one to which the Server is currently locked. In this situation however, the Server does NOT issue a Reject. Instead, it sends an error response to the upstream Requestor. (See: "Error Reporting")

The Server can also be configured - via the Requestor_ConfigA.Ena_SOC bit - to issue a reject on any new incoming message. This is a synchronized enable such that it can be asserted/negated at any time. The Server will continue to process any pending transactions but will prevent any new ones. Thus, the Server can be "gracefully" removed from the Switch interface.

Whenever the Server is in any state other than its "idle" state (locked or unlocked), it will refuse new attempts at a connection (Frame high preceded by Frame low for for at least two Switch Intervals) by issuing a Reject. There are many instances when a new connection attempt would indicate an Switch protocol violation, and thus a Reject issued by the Server would make little difference. However, there are some situations where the Server would correctly issue a Reject while it is off processing some event. For instance, a drop-lock would cause the Server to begin issuing a FREE-LOCK on the T-Bus. If new downstream Switch message attempted to access the Server before it finished the transaction, the Server would issue a Reject.

A.5.2.3 T-Bus Interface

The Server supports the standard T-Bus protocol with some small limitations. For one, the Server does NOT support unaligned transfers which fall across word (32-bit) boundaries. The Server also expects to see an ERROR response as the FIRST response from a T-Bus Slave if that slave is going to issue any ERROR's. If the Slave cannot issue an ERROR in the cycle immediately following the T-Bus request (i.e., the first response cycle), it must assert T_NSPAUSE_xxx until it decides if the request is an error or not.

A.5.2.4 LCON Interface

The LCON is a the physical and logical link between the SIGA-Server and the "input" port of the Switch Gate Array (SGA). In other words, for the SIGA, the LCON interface is the logical Switch interface. The LCON provides the Server with: 1) level conversion to and from the ECL levels of the SGA and 2) reclocking of data, Frame, Reverse to and from the SGA.

Figure A-67 shows the Server's LCON (Switch) Interface Pins.

PIN NAME	TYPE	FUNCTION
S_DATA<7..0>	bidirectional	Server-LCON data bus
S_FRAME	input	Frame input from Switch
S_REVERSE	output	Reverse output to Switch
S_NENA_BACK	input	LCON TTL driver enable

Figure A-67. Server LCON (Switch) interface pins.

A.5.2.4.1 Data Bus Enable Control

The Server controls the enables of both its own output drivers and the LCON's output drivers to the SIGA-LCON data interface - S_DATA < 7..0 > . It does so in a manner complementary to the Requestor's method (see "Requestor/Operation/LCON Interface/Data Bus Enable Control"). The Server uses the same concept of "Talk" and "Listen" mode as the Requestor.

The Server is considered "quiescent" when it is not transmitting messages and not waiting for any replies. When quiescent, the Server is in Listen Mode. The Server tries to stay in Listen Mode whenever possible, making the transition to Talk only for the absolute minimum time necessary. This situation is the mirror image to the Requestor. It is always in Talk Mode when quiescent and tries to stay in Listen mode for as little time as possible.

When the Server receives the checksum of a downstream message, it transitions to Talk mode – via the "dead" state. It remains in Talk mode until the T–Bus transaction is complete and the upstream return message has been sent. Once the upstream checksum has been sent, the Server transitions immediately into Listen mode (no contention is possible – as with the Requestor).

A.5.3 TCS Control Unit (TCU)

The basic purpose of the TCS Unit (TCU) is to allow the Test and Control System (TCS) Slave Processor access to the T–Bus interface – in essence, to act as a protocol converter. Normally, this involves the TCU acting like a T–Bus Master – performing reads and writes. However, the TCU is flexible enough so that it can also generate or "spoof" responses for any T–Bus Master or Slave. A "spoofed" response essentially involves issuing a response on the T–Bus in the absence of a request. This can used, for instance, to free–up an observing T–Bus Master who's locked Slave has failed. In this case, the TCU can "make believe" that IT is the "failed" slave.

A secondary function of the TCU is to allow the TCS Slave Processor DIRECT access to the CSU Map, rather than forcing it to make an access via the T–Bus interface. This is useful for fault–tolerance and bootstrapping.

A.5.3.1 I/O Description

The TCU interface is composed of four pins on the SIGA. The pins and their basic functions are shown in Figure A–68.

C_CLK	The data shift clock. Data is shifted into the SIGA on each rising edge of C_CLK. Data is shifted out of the SIGA on each falling edge of C_CLK.
C_IN	TCS data into the SIGA.
C_OUT	TCS data out of the SIGA. This is a tri–state signal which is driven when C_NEXECUTE is asserted (=0).
C_NEXECUTE	Asynchronously initiates execution of a command (=0) and enables C_OUT. In addition, negating C_NEXECUTE (=1) resets the TCU interface.

Figure A–68. TCU I/O signal description.

A.5.3.2 Read/Write Operation

The TCU contains 16 addressable registers - each 8 bits wide. The TCS Slave can read any register by clocking-in the required address (4 bits), a Read/Write bit (=1), and assert C_NEXECUTE (=0). A read operation is illustrated in Figure A-69. Some additional details for Read operations - not apparent from Figure A-69 - are now discussed.

1. C_IN data is clocked-in on the positive edge of C_CLK and C_OUT data is clocked-out on the negative edge of C_CLK.
2. Data can be clocked in or out at any desired rate, provided that the AC specifications of the C_CLK pin are not violated. The duty cycle of C_CLK is variable within the AC specifications. There is no MAXIMUM high (=1) or low (=0) time for C_CLK.
3. Reads are non-destructive and can be aborted at any time.
4. C_NEXECUTE is not synchronized with C_CLK and can be asserted at any time after the address and Read/Write bit has been clocked-in.
5. The C_OUT pin may be used to monitor, in real time, the value of a particular bit. This is done by reading the appropriate register, shifting-out the desired bit using C_CLK, and then holding C_CLK steady. C_CLK can be held in either state (1 or 0) as long as it does not make another positive transition.
6. Extra data bits preceding the negative transition of C_NEXECUTE, are ignored.

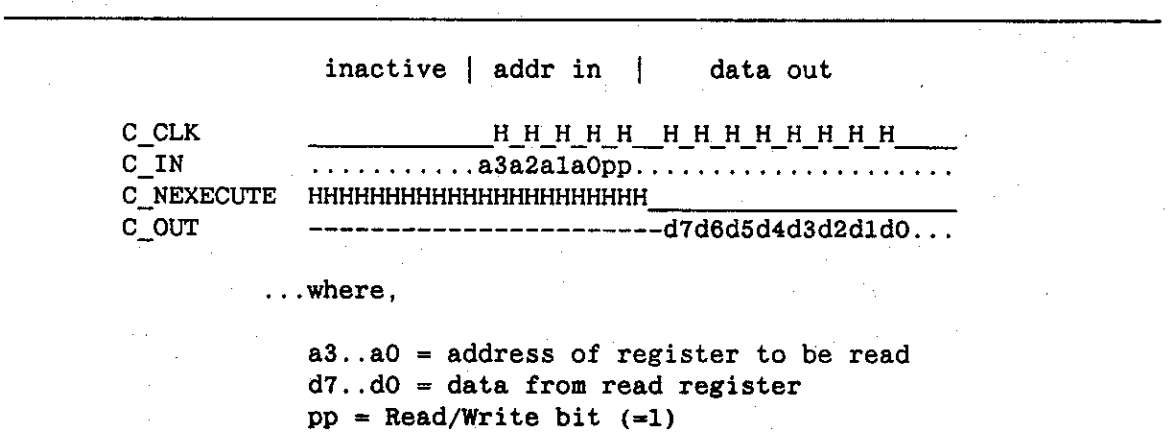


Figure A-69. Timing - TCU read operation.

A write operation is performed by clocking-in four bits of data, 4 bits of address, a Read/Write bit (=0), and then asserting C_NEXECUTE (=0). This is illustrated in Figure A-70. Some additional details for Write operations - not apparent from Figure A-70 - are now discussed.

1. C_IN data is clocked-in on the positive edge of C_CLK and C_OUT data is clocked-out on the negative edge of C_CLK.
2. Data can be clocked in or out at any desired rate, provided that the AC specifications of the C_CLK pin are not violated. The duty cycle of C_CLK is variable

within the AC specifications. There is no MAXIMUM high (=1) or low (=0) time for C_CLK.

3. Reads are non-destructive and can be aborted at any time. Reads of the TBUS_Response register can be aborted as well. However, if this is done AFTER C_NEXECUTE has been asserted, the T-Bus operation may be aborted.

4. C_NEXECUTE is not synchronized with C_CLK and can be asserted at any time after the address and Read/Write bit has been clocked-in.

5. C_NEXECUTE need only be asserted for a short moment to begin execution of the command. The minimum low time is described in "AC Specifications".

6. Extra data bits preceding the negative transition of C_NEXECUTE, are ignored.

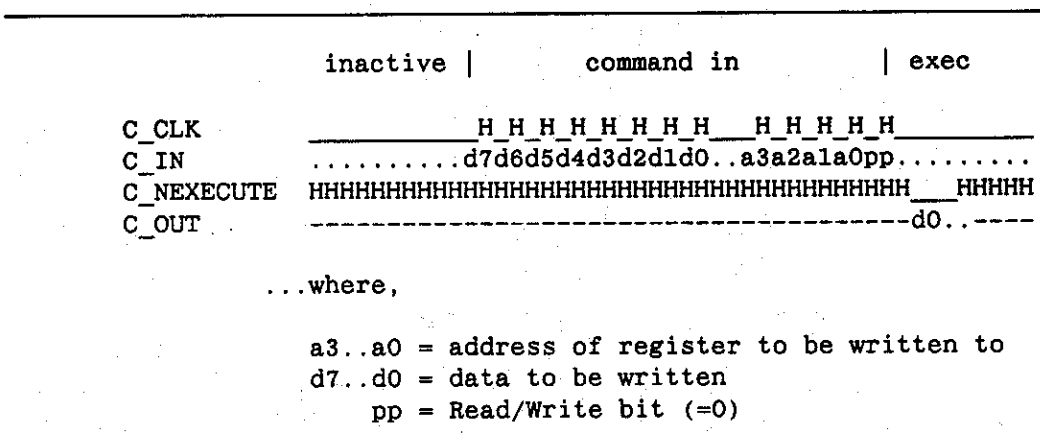


Figure A-70. Timing – TCU write operation.

A.5.3.3 Register Map

The register map for the 16 TCU registers is shown in Figure A-71.

a3..a0	DESCRIPTION
0	T_AD<7..0> (data)
1	T_AD<15..8> (data)
2	T_AD<23..16> (data)
3	T_AD<31..24> (data)
4	T_AD<7..0> (addr)
5	T_AD<15..8> (addr)
6	T_AD<23..16> (addr)
7	T_AD<31..24> (addr)
8	TBUS_Response
9	TBUS_Command
A	TBUS_Command_Modifier_0
B	TBUS_Command_Modifier_1
C	CSU Map<7..0>
D	CSU Map<8>
E	unused
F	unused

Figure A-71. TCU register map.

Referring to Figure A-71, registers 0 through 3 are special registers. For write operations, their contents are loaded, via the TCU interface, with the data to be written TO some T-Bus slave. For read operations, their contents are replaced with the data read FROM some T-Bus slave. Registers 4 through 7 are loaded ONLY by the TCU interface. The contents of these registers are placed on the T-Bus during the address phase of a T-Bus request.

The registers at address "C" and "D" are used to initialize CSU_Map<8..0>. Register "D" - bit "0", corresponds to CSU_Map<8>. Bits 7 through 1 of register "D" are unused. Figure A-72 shows the definition of the TBUS Response and Command Registers. Referring to Figure A-72, the TBUS_Response register is a read-only register which is valid after a T-Bus operation has been executed. The "Done" bit is monitored after a T-Bus command is initiated by the TCU. When asserted (= 1), it indicates that the operation is complete. See the "T-Bus Operations" section for more detail. The "Drive_AD" bit indicates that the T_AD Bus was driven during a T-Bus access (= 1). The remaining bits in the TBUS_Response register are the "responses" received from the T-Bus operation.

```
Register: TBUS_Response<7..0> (read only)

BIT/FIELD  FUNCTION (read only)
=========  ======================
      <7>  Done
      <6>  Drive_AD
      <5>  T_DRIVEN
      <4>  M_PARITY
      <3>  T_AD<32>
   <2..0>  T_RR<2..0>

Register: TBUS_Command<7..0>

BIT/FIELD  FUNCTION
=========  ======================
   <7..6>  output T_AD<33..32> (addr)
   <5..3>  output T_SIZE<2..0>
   <2..0>  output T_RR<2..0>

Register: TBUS_Command_Modifier_0<7..0>

BIT/FIELD  FUNCTION
=========  ======================
   <7..0>  unused
      <3>  Response
      <2>  output T_AD<32> (data)
   <1..0>  output T_PATH<1..0>

Register: TBUS_Command_Modifier_1<7..0>

BIT/FIELD  FUNCTION
=========  ======================
      <7>  output T_SYNC
   <6..5>  output T_PRIORITY<1..0>
   <4..3>  output T_LOCKOP<1..0>
   <2..0>  output T_SOURCE<2..0>
```

Figure A-72. Register definitions – TBUS Response and Command registers.

The TBUS_Command and BUS_Command_Modifier_1 registers contains the indicated fields to be placed on the T-Bus during the address phase of any operation. The TBUS_Command_Modifier_0 register outputs the "T_PATH" field during the address phase of any operation and the T_AD<32> bit during the data phase of a write operation.

The "Response" field of the TBUS_Command_Modifier_0 register, has a special function. When asserted (= 1), the TCU will place a "0" on the T_REQUEST and drive the T-Bus FOR A SINGLE CYCLE with the register settings intended for the address

phase of a T-Bus cycle. This is used for "spoofing" a T-Bus response. When the "Response" field is a "1", the TCU makes a normal T-Bus Request with T_REQUEST asserted (=1).

A.5.3.4 Normal T-Bus Operations

The TCU can be used to read and write, one to four bytes. Multi-word transfers are not allowed. The TCU can also OPEN and FREE locks although this is not recommended because the TCS Slave interface is relatively slow.

A read or write operation is set up by loading the desired data into the registers. The operation is actually initiated by a read of the TBUS_Response register. Since the MSB of this register is the "Done" bit, C_CLK should be disabled just after C_NEXECUTE is asserted (=0). This allows asynchronous monitoring of the "Done" bit. Terminating the read by negating (=1) C_NEXECUTE will abort the T-Bus request.

The TCU will retry after becoming REFUSED but will ignore a REFUSED LOCKED. In other words, the TCU will not become an "observing master".

A.5.3.5 Special T-Bus Operations

The TCU can FREE-LOCKS for any T-Bus master by specifying the correct T_SOURCE field value and performing a write operation. The TCU can also spoof any one-cycle response of a Slave by asserting the "Response" bit in the TBUS_Command_Modifier_0 register. For instance, it can issue a COMPLETED or ERROR for some Slave that is known to be faulty.

A.5.3.6 CSU Map Initialization

The CSU_Map is a 9-bit quantity which maps the SIGA CSU into a desired 8K page. This quantity is initialized by the TCU and is one of the first things that must be done to the SIGA upon power-up. If the CSU_Map is not initialized, it defaults to the setting of all 1's.

A.5.4 Configuration/Status Unit

The Configuration Status Unit (CSU) is the T-Bus Slave interface which allows any T-Bus master read and write access to the SIGA's configuration and status registers.

A.5.4.1 Normal Register Accesses

The CSU is limited in its support of the T-Bus protocol and is NOT optimized for minimum wait states (Slave pause cycles). The CSU will respond to a T-Bus query ONLY when T-Bus bits T_AD<24..16> match CSU_Map<8..0>. The CSU_Map is initialized by the TCU (See: TCS Control Unit/CSU Map Initialization).

In the cycle following a request to the CSU, the CSU will either respond with an ERROR or go on to complete the requested function. Figure A–73 shows the TCU responding with an ERROR.

```
T-Bus cycle #   |  0   |   1  |   2  |
T-Bus cycle     | req  | resp |  end |

T_NSPAUSE_SIGA HHHHHHHHHH_______HHHHH
T_RR<3..0>                xxxxxxxxeeeee

        ...where,

        x..x = invalid response
        e..e = ERROR response
```

Figure A–73. Timing – CSU ERROR access.

Note from Figure A–73, that T_NSPAUSE_SIGA is asserted for only one cycle. The ERROR response is triggered by exactly two conditions: 1) T_SIZE<2> = 1, or 2) T_LOCKOP<1> = 1. This means that the CSU will not support multi–word writes or locking. A normal read and write operation are shown in Figure A–74. Note from Figure A–74 that T_AD<32> is always a "0" on a read and a "don't care" on a write. In addition, during write operations, data is set up to the configuration latches during cycle #1, written to them during cycle #2, and held at the configuration latches during cycle #3.

```
T-Bus cycle #          |  0   |   1  |   2  |   3  |
T-Bus cycle            | req  | resp | resp |  end |

T_NSPAUSE_SIGA         HHHHHHHHHH_______________HHHH
T_RR<3..0>             ??????????xxxxxxxxxxxxxxcccc

T_AD<32>     (read)    ????????????XXXXXXXXXXXXX____
T_AD<31..0>  (read)    ????????????XXXXXXXXXXXXXRRRR

T_AD<32>     (write)   XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
T_AD<31..0>  (write)   ?????????WWWWWWWWWWWWWWWWWWW

            ...where,

            x..x = invalid response
            c..c = COMPLETED response
            X..X = invalid data
            W..W = valid write data
```

Figure A–74. Timing – normal CSU read/write.

A.5.4.2 Synchronized Accesses

Certain accesses to the CSU must be synchronized to the One Microsecond Pulse (OMSP). These include: 1) read/writes of the Real Time Clock, and 2) writes to the TONI_A or TONI_B registers. This mechanism is described in: "Requestor/Operation/RTC and Related Functions". Essentially, all this means to the CSU timing diagram in Figure A-74, is that cycle #2 is repeated until the synchronization pulse is received from the RTC or TONI_A/B controller.

A.5.4.3 Interleaver Loader

The CSU provides support for loading and reading the Interleaver Modulus Ram through the use of two special registers: Interleave_Address and Interleave_Data; and an external pin to the SIGA: I_NACCESS. Reads and writes to both the Interleave_Address and Interleave_Data registers are different than accesses to other configuration/status registers in the SIGA. The structure of the Interleaver_Address register is shown in Figure A-75. The structure of the Interleaver_Data register is shown in Figure A-76. As seen in Figure A-76, read/write access to the I_D register does not involve any data transfer within the SIGA.

```
Register: Interleave_Address

31.............................0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

...where,

A..A = interleaver address
```

Figure A-75. Register definition – Interleave_Address.

```
Register: Interleave_Data

31.............................0
------------------------------  (read)
------------------------------  (write)
```

Figure A-76. Register definition – Interleave_Data.

A.5.4.3.1 Address Register Access

When a T-Bus master reads the Address_Register, the CSU immediately responds with a Slave Pause cycle by asserting (=0) the T_NSPAUSE_SIGA pin, as it does with all other accesses. However, in the following cycle, the CSU also asserts the I_NACCESS pin and places the contents of the Interleave_Address register on the T-Bus.

The CSU then waits for exactly seven (7) T-Bus cycles in this state. The mapping of the I_A register to the T-Bus during this "wait" state is shown in Figure A-77, part (a). In the cycle following the wait period, the CSU then negates (=1) both T_NSPAUSE_SIGA and I_NACCESS, and maps the I_A to the T-Bus as shown in Figure A-77, part (b).

```
T_AD<33>    = Interleave_Address<1>
T_AD<32>    = Interleave_Address<0>
T_AD<31..0> = Interleaver_Address<31..0>

          (a) wait (cycle 3 - 9)

T_AD<33>    = 0
T_AD<32>    = 0
T_AD<31..0> = Interleaver_Address<31..0>

          (b) end (cycle 10)
```

Figure A-77. Interleave_Address register to T-Bus mapping.

The timing for writes to the I_A register is exactly the same as for reads. The actual timing for Interleave_Address register read/write access is shown in Figure A-78.

```
T-Bus cycle # |  0   |   1   |  2   |...|  10  |  11  |
T-Bus cycle   | req  | resp  | wait |...|  end |  ?   |

T_NSPAUSE_SIGA HHHHHHHHH_____________..._____HHHHHHHHHHH
T_RR<3..0>                              ...  cccccc

I_NACCESS   (read) HHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (read)     ------???aaaaa...aaabbbb

I_NACCESS  (write) HHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (write)     ddddddddddddd...dddd???

       ...where,

       c..c = COMPLETED response
       a..a = "wait" type read of I_A (bit swapping)
       b..b = "end" type read of I_A (bit masking)
       d..d = data written TO the I_A register
```

Figure A-78. Timing - CSU Interleave_Address register read/write access.

A.5.4.3.2 Data Register Access

The Interleave_Data access is EXACTLY the same as the Interleave_Address access EXCEPT for two key features: (1) during writes, no data is actually stored in the SIGA, and (2) during reads, the SIGA does NOT drive the T_AD<33..0> field. During this time, logic external to the SIGA will manipulate the Modulus Ram, and the SIGA is

basically being used as an address decoder and T-Bus control signal driver. The actual timing for Interleave_Data register read/write access is shown in Figure A-79. Note from Figure A-79 that the CSU temporarily drives the T-Bus during cycle #1. The data is unknown.

```
T-Bus cycle # |  0   |   1   |  2   |...|  10  |  11  |
T-Bus cycle   | req  | resp  | wait |...|  end |  ?   |

T_NSPAUSE_SIGA HHHHHHHHH_____________...____HHHHHHHHHHH
T_RR<3..0>                           ...   ccccc

I_NACCESS (read) HHHHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (read) xxxxxaaaaaaa---bb...bbbbbbb

I_NACCESS  (write) HHHHHHHHHHHH_____...____HHHHHHHHHHH
T_AD<33..0> (write)   ???????????xxx...xxxxxxxxx

        ...where,

        c..c = COMPLETED response
        a..a = unknown data driven by CSU (only for one cycle)
        b..b = data from/to Interleaver (not driven by SIGA)
```

Figure A-79. Timing - CSU Interleave_Data read/write access.

A.5.4.4 Debug Support

The CSU supports "freezing" a CSU read or write for debugging purposes. This is accomplished by initiating a normal T-Bus access (see Figure A-74, "Timing - normal CSU read/write") and asserting (=0) and holding the pin, M_NDEBUG, during cycle #1 and #2. This will cause the CSU to repeat cycle #2 indefinitely until M_NDEBUG is negated (=1). When this occurs, the CSU will continue with cycle #3 as normal.

For read cycles this means that T_AD<31..0> will have the real-time state of any register being read. By reading a test register, for example, the state machine of the STU can be observed while it sends a message.

For write cycles, the use is somewhat limited. It simply means that T_AD<31..0> can be manipulated in real time from the master (or logic analyzer). Since during cycle #2 the configuration latches are transparent, so that any external manipulation will be seen internally in real time.

A.5.4.5 Restriction Summary

The following restrictions apply to CSU operation: [TO BE SUPPLIED]

A.6 Programming Model

This section provides a memory map of the previously defined SIGA registers, as well as a compilation of all SIGA Error Codes.

1. The CSU will flag as an ERROR any multi-word access or an OPEN or MAINTAIN. Therefore, the CSU does not support these operations. However, byte masking on writes IS supported.
2. The CSU will NOT check for unaligned transfers. It is illegal to request an operation with an unaligned address.
3. Synchronized Accesses rely on the presence of R_CLK to complete. If R_CLK is non-existent, the CSU will pause the T-Bus Master indefinitely. The only way to release the pause would be to assert the M_NRESET pin.
4. The Stolen bit (T_AD<32>) is not supported on either reads or writes.

A.6.1 Memory Map

Figure A-80 shows the memory map of the various registers. Note from Figure A-80 that the "M" field is programmable via the CNU_Config.CSU_Map bits.

```
T_AD<24..0>              REGISTER
====================     ====================================
  15  12         2 10
  |   |         | ||
M 000 XXXXXXXX000 bb  TONIA_Config
                001 bb  Time_Of_Next_InterruptA

M 001 XXXXXXXX000 bb  TONIB_Config
                001 bb  Time_Of_Next_InterruptB

M 100 0XXXXXXX000 bb  Protocol_Timer_Config | Message_Class
                001 bb  Transmit_Time_Config
                010 bb  Priority_Time_Config
                011 bb  Requestor_ConfigA
                100 bb  Requestor_ConfigB
                101 bb  Requestor_TestA
                110 bb  Real_Time_Clock (hi/lo)
                111 bb  <reserved>

      1XXXXXXX000 bb  Server_ConfigA
                001 bb  Server_ConfigB
                010 bb  Server_TestA

M 101 0XXXXXXXXXX xx  Interleave_Address_Reg
      1XXXXXXXXXX xx  Interleave_Data_Reg

        ...where,

           M  = (T_AD<24..16> = CNU_Config.CSU_Map<8..0>)
           bb =  00  byte 0 <31..24>
                 01  byte 1 <23..16>
                 10  byte 2 <15..8>
                 11  byte 3 <7..0>
           xx =  no byte addressing capability
```

Figure A–80. SIGA memory map.

A.6.2 Error Code Summary

Figure A–81 presents an Error Code summary for the SIGA. Figure A–82 summarizes the Error Code definitions.

```
Requestor/CSU Error Codes:

7       0
|       |
PPPPdcba

d c b a  Requestor/ CSU Error
= = = =  ====================
0 0 0 0  Maintain_Absent-(2a)
0 0 0 1  Maintain_Present-(2b)
0 0 1 0  Stolen_Verify-(1)
0 0 1 1  Lock_Address-(2)
0 1 0 0  Wait_TO-(3a)
0 1 0 1  Idle_TO-(3b)
0 1 1 0  Rej_Abort(4)
0 1 1 1  Rej_TO-(5)
1 0 0 0  Reverse-(6)
1 0 0 1  Check-(7)
1 0 1 0  Misc. CSU Error

...where,

P..P = Requestor_ConfigA.Error_Prefix<3..0>
Priority is from highest (1) to lowest (8).
Within a given priority, errors are mutually
exclusive (i.e., 4a,b...).

     Server Error Codes:

     7       0
     |       |
     PPPPPPba

     b a  Server Error
     = =  ==================
     0 0  Downstream_Refused
     0 1  Downstream_Write
     1 0  Downstream_Late
     1 1  Downstream_OTL

...where,

P..P = Server_ConfigA.Error_Prefix<5..0>
```

Figure A-81. Error code summary.

Maintain_Absent An NORMAL was issued to the Requestor during its idle state and it was locked.

Maintain_Present	A MAINTAIN was issued to the Requestor during its idle state and it was NOT locked.
Lock_Address	A Function Request was made to a locked Requestor during its idle state with a node address was different than that which opened the locked sequence.
Wait_TO	The Switch Transmit Connection Timer overflowed while the Requestor was waiting for a Function Response.
Idle_TO	The Switch Transmit Connection Timer overflowed while the Requestor was in its idle state.
Rej_Abort	The Switch Transmit Reject Timer was forced into overflow by the REJ_ABORT input pin.
Rej_TO	The Switch Transmit Reject Timer overflowed while the Requestor was attempting to open a connection.
Reverse	The Requestor detected an incorrect polarity of the Reverse signal during a Function Response.
Check	The Requestor detected an incorrect Checksum during a Function Response.
CSU Error	An error was made accessing the CSU. It could be one or both of the of the following: 1) An OPEN lock was requested or 2) A Multi-word transfer was requested.
Downstream_Write	A downstream write error was detected while the downstream Server was sourcing data.
Downstream_OTL	A downstream T-Bus slave did not respond to the Server's request.
Downstream_Late	A downstream T-Bus slave responded with a LATE ERROR.
Downstream_Refused	A downstream T-Bus slave responded with REFUSED-LOCKED when the Server thought itself locked.

Figure A-82. Error code definition summary.

A.7 Special Topics

This section describes some of the special topics relating to SIGA operation.

A.7.1 Initialization States

The external Reset signal is resynchronized by the SIGA for use by all synchronous logic clocked by all three major clocks (R_CLK, S_CLK and T_CLK). When Reset is applied and then released, all internal storage logic that needs to be initialized, will be so initialized. The SIGA will now be in its first initialization state, known as the Quiescent State.

In this state, the SIGA Switch and T-Bus interfaces are partially disabled. The Server's Switch interface responds to any assertions of downstream Frame with Rejects. The Requestor's Switch interface ignores any assertions of the upstream Reverse. The Server's T-Bus interface makes no T-Bus requests and the Requestor's T-Bus interface responds to any remote function requests with a REFUSED. The Configuration/Status Unit and the TCU, however, are operational. Normally, in the Quiescent state, the TCU will initialize the CSU's mapping logic via the CNU_Config.CSU_Map<8..0> register. Once the Control Net initializes the CSU_Map, any T-Bus master can then initialize the SIGA registers via the CSU.

Once this is accomplished, the SIGA is in the Operational State. The Operational State is the normal operational mode of the SIGA.

A.7.2 Synchronization

Because of the use of multiple clocks, the SIGA design inherently requires the use of synchronizers to implement handshaking across clock boundaries. Some of these synchronizers are in non-critical paths and are thus implemented in the most cost-effective manner. In particular, these synchronizers are of the "large uncertainty, fixed-delay" variety. This means that there delay is not programmable and that "input-to-output" delay is not constant over changes in input. These are used in areas such as: 1) Between the external reset pin, M_NRESET, and the internal reset destinations, 2) Between the TCU negation of C_NEXECUTE and the T_Bus access. These synchronizers are designed to provide a MINIMUM of 100 ns settling time (T_CLK <= 22 MHz, R_CLK,S_CLK <= 45 MHz).

The other variety of synchronizers - used in critical path applications - are the "variable delay, zero uncertainty" synchronizers. These are used between the T-Bus and Switch interfaces along the Function request/response paths. These are the synchronizers which have four bits of configuration to control the settling time. Figure A-83 shows the various settings for ALL variable-delay synchronizers. Figure A-83 should be used in combination with the clock period of the logic RECEIVING the synchronizer data to determine the actual settling time. For instance, if a 100 ns settling time on the positive edge is desired for the STU Synchronizer, the register: Requestor_ConfigA.STU_Sync<3..0>, should be set to a "0110". This is because assuming R_CLK =

40 MHz (25 ns period), the synchronizer will require four clock periods – at 25 ns apiece – to obtain the total of 100 ns.

On the other hand, the BIU Synchronizer control, set by Requestor_ConfigA.BIU_Sync<3..0>, would need a setting of "0010" to obtain the same settling time. Here, of course, the clock period is twice as long as the STU Synchronizer so the number of synchronizer clock delays is half.

3210	# CLOCK DELAYS	TRANSFER EDGE
0000	1	Positive
0001	1	Negative
0010	2	Positive
0011	2	Negative
0100	3	Positive
0101	3	Negative
0110	4	Positive
0111	4	Negative
1000	5	Positive
1001	5	Negative
1010	ILLEGAL	-
1011	ILLEGAL	-
1100	ILLEGAL	-
1101	ILLEGAL	-
1110	ILLEGAL	-
1111	ILLEGAL	-

Figure A-83. Variable-delay synchronizer settings.

NOTE

Currently, it is recommended that only the POSITIVE transfer edge be used for any setting.

NOTE

It has been determined that a settling time of 100 ns is a reasonable goal for the variable-delay synchronizers.

A.8 Pin Description and Pinout

The next page begins a pin description of the SIGA.

PIN NAME	TYPE	DESCRIPTION
C_CLK	IN	TCU input clock
C_IN	IN	TCU data input
C_NEXECUTE	IN	TCU execute handshake input
C_OUT	OUT	TCU data output
F_AD<24..16>	IN	T-Bus input for T_AD<24..16>
F_PATH<1..0>	IN	T-Bus input for T_PATH<1..0>
F_REQUEST	IN	T-Bus input for T_REQUEST
F_RR<2..0>	IN	T-Bus input for T_RR<2..0>
F_SIZE_2	IN	T-Bus input for T_SIZE_2
F_SOURCE<2..0>	IN	T-Bus input for T_SOURCE<2..0>
I_INTERLEAVED	IN	=0: do NOT use I_MOD<8..0> for route address =1: use I_MOD<8..0> for route address
I_MOD<8..0>	IN	Interleaver data input
I_NACCESS	OUT	=0: CSU Interleaver loader is active =1: CSU Interleaver loader is NOT active
M_NDEBUG	IN	=0: Debug mode during CSU access (TEST ONLY) =1: Do NOT enter debug mode (NORMAL MODE)
M_NFLOAT	IN	=0: Tri-state all outputs (TEST ONLY) =1: Normal output operation (NORMAL MODE)
M_NRESET	IN	=0: Hardware reset to SIGA =1: Normal operational mode
M_NSELECT	IN	=0: Select CSU, attach to T_PATH<1/0> =1: Do NOT select CSU
M_PARA	OUT	Parametric nand tree output (TEST ONLY)
M_PARITY	IN	=0: No parity error during T-Bus response =1: Parity error during T-Bus response
M_REJ_ABORT	IN	=0: Do NOT abort Switch retries =1: Abort Switch retries
M_SIXTY_FIVE	IN	=0: 65 ms pulse NOT active =1: 65 ms pulse active (one R_CLK period)
M_TONIA_INT	OUT	=0: TONIA interrupt is active =1: TONIA interrupt is NOT active
M_TONIB_INT	OUT	=0: TONIB interrupt is active =1: TONIB interrupt is NOT active
R_CLK	IN	Requestor clock input
R_DATA<7..0>	BID	Requestor Switch data interface
R_FRAME	OUT	Requestor Switch Frame output
R_NENA_BACK	OUT	=0: Enable LCON to drive R_DATA<7..0> =1: Disable LCON from driving R_DATA<7..0>
R_REVERSE	IN	Requestor Switch Reverse Input
S_CLK	IN	Server clock input
S_DATA<7..0>	BID	Server Switch data interface
S_FRAME	IN	Server Switch Frame input
S_NENA_BACK	OUT	=0: Disable LCON from driving S_DATA<7..0> =1: Enable LCON to drive S_DATA<7..0>
S_REVERSE	OUT	Server Switch Reverse Input
T_AD<33..25>	BID	T-Bus input/output for T_AD<33..25>
T_AD<24..16>	OUT	T-Bus output for T_AD<24..16>
T_AD<15..0>	BID	T-Bus input/output for T_AD<15..0>
T_CLK	IN	T-Bus input clock
T_DRIVEN	OUT	T-Bus output for T_DRIVEN

T_ENA_HOLD	IN	=0: Disable T-Bus input latches
		=1: Enable T-Bus input latches
T_ENA_TDAT.2	OUT	=0: Enable T_AD<33..0> drivers
		=1: Disable T_AD<33..0> drivers
T_ENA_TDAT<1..0>	OUT	=0: Disable T_AD<33..0> drivers
		=1: Enable T_AD<33..0> drivers
T_ENA_TRANS.1	OUT	=0: Enable transaction T-Bus field
		=1: Disable transaction T-Bus field
T_ENA_TRANS.0	OUT	=0: Disable transaction T-Bus field
		=1: Enable transaction T-Bus field
T_LOCKOP<1..0>	BID	T-Bus input/output for T_LOCKOP<1..0>
T_MPAUSE	OUT	T-Bus output for T_MPAUSE
T_NBGRANT_SIGM	IN	=0: SIGA Master granted next T-Bus
		=1: SIGA Master NOT granted next T-Bus
T_NBGRANT_SIGS	IN	=0: SIGA Slave granted next T-Bus
		=1: SIGA Slave NOT granted next T-Bus
T_NBREQ_SIGM	OUT	=0: SIGA Master is requesting T-Bus
		=1: SIGA Master is NOT requesting T-Bus
T_NBREQ_SIGS	OUT	=0: SIGA Slave is requesting T-Bus
		=1: SIGA Slave is NOT requesting T-Bus
T_NDRIVEN_SIGA	OUT	=0: SIGA is driving T-Bus next cycle
		=1: SIGA is NOT driving T-Bus next cycle
T_NSPAUSE_SIGA	OUT	=0: SIGA is pausing T-Bus next cycle
		=1: SIGA is NOT pausing T-Bus next cycle
T_PATH<1..0>	OUT	T-Bus output for T_PATH<1..0>
T_PRIORITY<1..0>	BID	T-Bus input/output for T_PRIORITY<1..0>
T_REQUEST	OUT	T-Bus output for T_REQUEST
T_RR<2..0>	OUT	T-Bus output for T_RR<2..0>
T_SIZE.2	OUT	T-Bus output for T_SIZE.2
T_SIZE<1..0>	BID	T-Bus input/output for T_SIZE<1..0>
T_SOURCE<2..0>	OUT	T-Bus output for T_SOURCE<2..0>
T_SPAUSE	OUT	T-Bus output for T_SPAUSE
T_SYNC	BID	T-Bus input/output for T_SYNC

The following page shows the SIGA pinout sorted by pin function.

SIGA PINOUT SORTED BY PIN FUNCTION

R15	C_CLK	R06	R_DATA.6	B12	T_DRIVEN
T14	C_IN	P06	R_DATA.7	C12	T_ENA_HOLD
R14	C_NEXECUTE	R05	R_FRAME	C03	T_ENA_TDAT.0
P13	C_OUT	T05	R_NENA_BACK	B03	T_ENA_TDAT.1
B09	F_AD.16	T04	R_REVERSE	A03	T_ENA_TDAT.2
C09	F_AD.17	T13	S_CLK	C14	T_ENA_TRANS.0
A10	F_AD.18	T12	S_DATA.0	C15	T_ENA_TRANS.1
B10	F_AD.19	P11	S_DATA.1	D01	T_LOCKOP.0
C10	F_AD.20	R11	S_DATA.2	D02	T_LOCKOP.1
A11	F_AD.21	T11	S_DATA.3	E01	T_MPAUSE
B11	F_AD.22	P10	S_DATA.4	A06	T_NBGRANT_SIGM
C11	F_AD.23	R10	S_DATA.5	C07	T_NBGRANT_SIGS
A12	F_AD.24	T10	S_DATA.6	C05	T_NBREQ_SIGM
A05	F_CLK	P09	S_DATA.7	B05	T_NBREQ_SIGS
A07	F_PATH.0	R13	S_FRAME	C06	T_NDRIVEN_SIGA
C08	F_PATH.1	R12	S_NENA_BACK	B06	T_NSPAUSE_SIGA
B14	F_REQUEST	P12	S_REVERSE	C13	T_PATH.0
C04	F_RR.0	P02	T_AD.0	A14	T_PATH.1
B04	F_RR.1	N03	T_AD.1	E02	T_PRIORITY.0
A04	F_RR.2	F14	T_AD.10	E03	T_PRIORITY.1
F03	F_SIZE_2	F15	T_AD.11	A13	T_REQUEST
G03	F_SOURCE.0	F16	T_AD.12	D14	T_RR.0
F01	F_SOURCE.1	G14	T_AD.13	D15	T_RR.1
F02	F_SOURCE.2	G15	T_AD.14	D16	T_RR.2
B02	I_INTERLEAVED	G16	T_AD.15	E14	T_SIZE.0
M02	I_MOD.0	H14	T_AD.16	E15	T_SIZE.1
M01	I_MOD.1	H15	T_AD.17	E16	T_SIZE.2
L03	I_MOD.2	J15	T_AD.18	D03	T_SOURCE.0
L02	I_MOD.3	J14	T_AD.19	C01	T_SOURCE.1
L01	I_MOD.4	P01	T_AD.2	C02	T_SOURCE.2
K03	I_MOD.5	K16	T_AD.20	B15	T_SPAUSE
K02	I_MOD.6	K15	T_AD.21	B13	T_SYNC
K01	I_MOD.7	K14	T_AD.22	A09	VDD
J03	I_MOD.8	L16	T_AD.23	A15	VDD
P03	I_NACCESS	L15	T_AD.24	B01	VDD
R02	M_NDEBUG	L14	T_AD.25	B16	VDD
P14	M_NFLOAT	M16	T_AD.26	J01	VDD
T15	M_NRESET	M15	T_AD.27	J16	VDD
B07	M_NSELECT	M14	T_AD.28	T01	VDD
R03	M_PARA	N16	T_AD.29	T08	VDD
C16	M_PARITY	N02	T_AD.3	T16	VDD
R04	M_REJ_ABORT	N15	T_AD.30	A02	VSS
J02	M_SIXTY_FIVE	N14	T_AD.31	A08	VSS
P04	M_TONIA_INT	P16	T_AD.32	A16	VSS
T03	M_TONIB_INT	P15	T_AD.33	H01	VSS
P05	R_CLK	N01	T_AD.4	H16	VSS
R09	R_DATA.0	M03	T_AD.5	R01	VSS
R08	R_DATA.1	H02	T_AD.6	R16	VSS
P08	R_DATA.2	H03	T_AD.7	T02	VSS
R07	R_DATA.3	G01	T_AD.8	T07	VSS

P07	R_DATA.4	G02	T_AD.9	T09	VSS
T06	R_DATA.5	B08	T_CLK		

The following page shows the SIGA pinout sorted by pin number.

SIGA PINOUT SORTED BY PIN NUMBER

A02	VSS	D16	T_RR.2	N14	T_AD.31
A03	T_ENA_TDAT.2	E01	T_MPAUSE	N15	T_AD.30
A04	F_RR.2	E02	T_PRIORITY.0	N16	T_AD.29
A05	F_CLK	E03	T_PRIORITY.1	P01	T_AD.2
A06	T_NBGRANT_SIGM	E14	T_SIZE.0	P02	T_AD.0
A07	F_PATH.0	E15	T_SIZE.1	P03	I_NACCESS
A08	VSS	E16	T_SIZE.2	P04	M_TONIA_INT
A09	VDD	F01	F_SOURCE.1	P05	R_CLK
A10	F_AD.18	F02	F_SOURCE.2	P06	R_DATA.7
A11	F_AD.21	F03	F_SIZE_2	P07	R_DATA.4
A12	F_AD.24	F14	T_AD.10	P08	R_DATA.2
A13	T_REQUEST	F15	T_AD.11	P09	S_DATA.7
A14	T_PATH.1	F16	T_AD.12	P10	S_DATA.4
A15	VDD	G01	T_AD.8	P11	S_DATA.1
A16	VSS	G02	T_AD.9	P12	S_REVERSE
B01	VDD	G03	F_SOURCE.0	P13	C_OUT
B02	I_INTERLEAVED	G14	T_AD.13	P14	M_NFLOAT
B03	T_ENA_TDAT.1	G15	T_AD.14	P15	T_AD.33
B04	F_RR.1	G16	T_AD.15	P16	T_AD.32
B05	T_NBREQ_SIGS	H01	VSS	R01	VSS
B06	T_NSPAUSE_SIGA	H02	T_AD.6	R02	M_NDEBUG
B07	M_NSELECT	H03	T_AD.7	R03	M_PARA
B08	T_CLK	H14	T_AD.16	R04	M_REJ_ABORT
B09	F_AD.16	H15	T_AD.17	R05	R_FRAME
B10	F_AD.19	H16	VSS	R06	R_DATA.6
B11	F_AD.22	J01	VDD	R07	R_DATA.3
B12	T_DRIVEN	J02	M_SIXTY_FIVE	R08	R_DATA.1
B13	T_SYNC	J03	I_MOD.8	R09	R_DATA.0
B14	F_REQUEST	J14	T_AD.19	R10	S_DATA.5
B15	T_SPAUSE	J15	T_AD.18	R11	S_DATA.2
B16	VDD	J16	VDD	R12	S_NENA_BACK
C01	T_SOURCE.1	K01	I_MOD.7	R13	S_FRAME
C02	T_SOURCE.2	K02	I_MOD.6	R14	C_NEXECUTE
C03	T_ENA_TDAT.0	K03	I_MOD.5	R15	C_CLK
C04	F_RR.0	K14	T_AD.22	R16	VSS
C05	T_NBREQ_SIGM	K15	T_AD.21	T01	VDD
C06	T_NDRIVEN_SIGA	K16	T_AD.20	T02	VSS
C07	T_NBGRANT_SIGS	L01	I_MOD.4	T03	M_TONIB_INT
C08	F_PATH.1	L02	I_MOD.3	T04	R_REVERSE
C09	F_AD.17	L03	I_MOD.2	T05	R_NENA_BACK
C10	F_AD.20	L14	T_AD.25	T06	R_DATA.5
C11	F_AD.23	L15	T_AD.24	T07	VSS
C12	T_ENA_HOLD	L16	T_AD.23	T08	VDD
C13	T_PATH.0	M01	I_MOD.1	T09	VSS
C14	T_ENA_TRANS.0	M02	I_MOD.0	T10	S_DATA.6
C15	T_ENA_TRANS.1	M03	T_AD.5	T11	S_DATA.3
C16	M_PARITY	M14	T_AD.28	T12	S_DATA.0
D01	T_LOCKOP.0	M15	T_AD.27	T13	S_CLK
D02	T_LOCKOP.1	M16	T_AD.26	T14	C_IN
D03	T_SOURCE.0	N01	T_AD.4	T15	M_NRESET

D14	T_RR.0	N02	T_AD.3	T16	VDD
D15	T_RR.1	N03	T_AD.1		

A.9 A.C./D.C. Parameters

All SIGA input and bidirectional pins have a light pullup resistor, a diode protection network (max = 2000V) and latch-up (max = 200 ma). All inputs and output have standard TTL VIL/VIH and VOL/VOH characteristics. All outputs and bidirectional pins have 4 ma drive capability - except T_ENA_TDAT<2..0> and T_ENA_TRANS<1..0>, which have 8 ma drive capability. The SIGA will dissipate less than 3 watts.

The following page shows the A.C. timing parameters.

NOTE

For the B2VME, the following A.C. parameters override the normal ones:

PIN/CLASS	Tsu	Thld	Tpd (min/max)	LOAD
T_NDRIVEN_SIGA	-	-	2.0/11.0	20.0
F_SOURCE<2..0>	21.0	0.0	-	-

SIGA A.C. CHARACTERISTICS

PIN/CLASS	Tsu	Thld	Tpd (min/max)	LOAD
TBUS:				
T_DRIVEN	25.0	0.0	-	-
T_MPAUSE	25.0	0.0	-	-
T_SPAUSE	25.0	0.0	-	-
T_NBGRANT_SIGM	25.0	0.0	-	-
T_NBGRANT_SIGS	25.0	0.0	-	-
T_REQUEST	(a)	(a)	2.0/18.0	30.0
T_RR<2..0>	(a)	(a)	2.0/18.0	30.0
T_PATH<1..0>	(a)	(a)	2.0/18.0	30.0
T_SOURCE<2..0>	(a)	(a)	2.0/18.0	30.0
T_SIZE.2	(a)	(a)	2.0/18.0	30.0
T_SIZE<1..0>	20.0	0.0	2.0/18.0	30.0
T_SYNC	20.0	0.0	2.0/18.0	30.0
T_LOCKOP<1..0>	20.0	0.0	2.0/18.0	30.0
T_PRIORITY<1..0>	20.0	0.0	2.0/18.0	30.0
T_AD<33..0>	20.0	0.0	2.0/18.0	30.0
T_NBREQ_SIGM	-	-	2.0/13.0	20.0
T_NBREQ_SIGS	-	-	2.0/13.0	20.0
T_NDRIVEN_SIGA	-	-	2.0/13.0	20.0
T_NSPAUSE_SIGA	-	-	2.0/13.0	20.0
T_ENA_TDAT<2..0>	-	-	2.0/15.0	30.0
T_ENA_TRANS<1..0>	-	-	2.0/15.0	30.0
T_ENA_HOLD	(d)			
FAST:				
F_REQUEST	25.0	0.0	-	-
F_RR<2..0>	24.0	0.0	-	-
F_SOURCE<2..0>	25.0	0.0	-	-
F_PATH<1..0>	25.0	0.0	-	-
F_SIZE_2	25.0	0.0	-	-
F_AD<24..16>	25.0	0.0	-	-
SWITCH - REQ:				
R_DATA<7..0>	2.0	6.0	2.0/13.0	20.0
R_REVERSE	2.0	6.0	-	-
R_FRAME	-	-	2.0/13.0	20.0
R_NENA_BACK	-	-	2.0/13.0	20.0
R_CLK	(c)			
SWITCH - SER:				
S_DATA<7..0>	2.0	2.0	2.0/13.0	20.0
S_FRAME	2.0	6.0	-	-

S_REVERSE	-	-	2.0/13.0	20.0
S_NENA_BACK	-	-	2.0/13.0	20.0
R_CLK		(c)		
TCS:				
C_IN	50.0	50.0	-	-
C_OUT	-	-	2.0/50.0	20.0
C_NEXECUTE	50.0	50.0	-	-
INTERLEAVER:				
I_MOD<8..0>	17.0	0.0	-	-
I_INTERLEAVED	24.0	0.0	-	-
I_NACCESS	-	-	2.0/30.0	20.0
MISCELLANEOUS:				
M_TONIA_INT	-	-	2.0/30.0	20.0
M_TONIB_INT	-	-	2.0/30.0	20.0
M_PARITY	21.0	0.0	-	-
M_NSELECT	25.0	0.0	-	-
M_NDEBUG	25.0	24.0	-	-
M_SIXTY_FIVE	7.9	14.9	-	-
M_NRESET	(b)	(b)	-	-
M_REJ_ABORT	(b)	(b)	-	-

NOTES:

specific:

(a) No internal connection to SIGA - timing is unimportant
(b) Synchronized within SIGA - timing is unimportant
(c) MINIMUM HIGH time for [RS]_CLK = 5.3 ns
(d) [TF]_CLK rising to T_ENA_HOLD rising = 15 ns minimum
T_ENA_HOLD minimum HIGH time = 6 ns
T_ENA_HOLD falling to [TF]_CLK rising = 4.5 ns minimum

general:

1. All times in nanoseconds
2. All loads in picofarads
3. TBUS, FAST and INTERLEAVER timing are relative to rising T_CLK
4. SWITCH - REQ timing is relative to rising R_CLK
5. SWITCH - SER timing is relative to rising S_CLK
6. TCS timing is relative to falling C_CLK

XC88100 CPU ERRATA Ver. 1.7
C89B Mask revision D.5
01/16/89

Please refer all questions to the Motorola Applications Hotline at (512) 891-APPS.

ERRATA:

1. There is a time interval in each clock cycle when the assertion of the PLLEN (Phase Lock Loop Enable) signal may not be properly recognized. The interval is a 1-2 ns window centered approximately 8.5 ns after the falling edge of the clock.

   Workaround: If a board is experiencing this problem, PLLEN can be delayed a few ns to cause the signal to miss the window. PLLEN can also be synchronized to the rising edge of the clock to guarantee that the window will be missed.

2. Data memory accesses during an exception handler can modify the contents of the Data Memory Transaction Register 0 (DMT0, cr8). This can fool the exception handler into thinking that a Data Access Exception has occurred.

   Workaround: The exception handler should immediately copy DMT0 to a spare register and to perform subsequent reads from this register instead of DMT0.

3. Integer divide immediate instructions (DIV and DIVU) will return an incorrect result if the immediate operand matches the binary pattern (X0111XXXXXXXXXXX), where an X represents a don't care.

   Workaround1: Do not use immediate values in the range of 0x3800 - 0x3FFF or 0xB800 - 0xBFFF when issuing an integer divide immediate instruction.

   Workaround2: Use idiv in place of the idiv.imm instruction.

4. A Floating point multiply instruction will not work properly if either operand is double precision and the second half of the instruction initiation gets a scoreboard hold.

   Workaround: Insure that the operands are available when using the floating point multiply instruction with double precision operands. One way to accomplish this is to insert a trap not taken before each floating point multiply instruction that uses double precision operands.

5. The cpu may skip instructions following store double (STD) and floating point instructions (involving double precision operands), if it receives an unexpected wait from the code cmmu.

   Workaround 1: Convert all store doubles to two store singles and only perform floating point arithmetic with single precision operands.

   Workaround 2: For single processor systems only. Disable bus snooping on the code cache and insert a trap not taken after every code cmmu control register access.

   Workaround 3: For multiprocessor systems only. Disable bus snooping on the code cache, prevent remote cmmu register accesses and insert a trap not taken after every code cmmu control register access.

6. The CPU will ' lock-up ' if an RTE is executed in Serial Mode and the Shadow Scoreboard is not clear. Most software implementations will encounter this in the interrupt handler only.

   Workaround 1: Do not use the floating point unit (FPU) in Serial Mode.

   Workaround 2: Modify the interrupt handler to flush the FPU (reenable the FPU, making sure that the cpu is not in serial mode, and stall for 60 cycles in an idle loop) and clear the SSBR before executing an RTE.

7. If a floating point instruction with a double precision destination generates an imprecise exception simultaneous with any other exception of higher priority, then the SFU imprecise exception is never recognized.

Workaround 1: Do not run code that causes imprecise exceptions.

Workaround 2: Do not generate double precision results.

Workaround 3: Prevent the simultaneous exceptions by placing a "trap not taken" before and after the floating point instruction and by disabling interrupts during its execution.

8. If an interrupt occurs with interrupts enabled but shadowing frozen, the interrupt exception vector is taken. This is incorrect; any exception recognized with shadowing frozen should go to the Error vector.

   Workaround: Interrupts should never be enabled if shadowing is frozen.

GENERAL INFORMATION:

1. Data in the data shadow registers (DSD0-DSD2) is only valid if the corresponding transaction is a store. If running in master/checker mode, do not store data from these registers to memory when the data is invalid.

2. Due to the current test environment, the 88100 is not guaranteed to work in Checker Mode(PCE pin asserted). Though it is felt that Checker Mode operates properly, until it can be verified on the tester, use of this mode is at the users own risk.

CURRENT DOCUMENTATION:

MC88100 User's Manual, MC88100UM/AD
MC88100 Technical Summary, Document BR588/D Rev. 1

88200 CMMU ERRATA Ver. 1.3
C67C Mask revision C.0
CMMU ID Register Ver. 00100
01/05/88

Please refer all questions to the Motorola Applications Hotline at (512) 891-APPS.

ERRATA:

1. The use of the xmem (exchange register with memory) instruction with the .bu option may cause the cmmu to write corrupted data to RAM.

   Workaround: Do not use the .bu extension with the xmem instruction. Only use xmem on 32 bit values.

2. There is a time interval during each clock cycle when the assertion of the PLLEN (Phase Lock Loop Enable) signal may not be properly recognized. The interval is a 1-2 ns window centered approximately 8.5 ns after the falling edge of the clock.

   Workaround: If a board is experiencing this problem, PLLEN can be delayed a few ns to cause the signal to miss the window. PLLEN can also be synchronized to the rising edge of the clock to guarantee that the window will be missed.

3. A snooping cmmu should invalidate a cache entry if it detects an MBUS master attempting to write (or read with intent to modify) to a "global" memory location which is cached by the snooping cmmu and is marked "shared unmodified". The snooping cmmu fails to perform the invalidation if it is servicing a pending PBUS write which hits an ATC entry marked "global" or "writethrough".

   Workaround: Do not use bus snooping.

CURRENT DOCUMENTATION:

MC88200 User's Manual, MC88200UM/AD
MC88200 Technical Summary, Document BR589/D Rev. 1

XC88200 CMMU ERRATA Ver. 1.1

C67C Mask revision C.1
CMMU ID Register Ver. 00101
01/05/88

Please refer all questions to the Motorola Applications Hotline at (512) 891-APPS.

ERRATA:

1. The use of the xmem (exchange register with memory) instruction with the .bu option may cause the cmmu to write corrupted data to RAM.

   Workaround: Do not use the .bu extension with the xmem instruction. Only use xmem on 32 bit values.

2. There is a time interval during each clock cycle when the assertion of the PLLEN (Phase Lock Loop Enable) signal may not be properly recognized. The interval is a 1-2 ns window centered approximately 8.5 ns after the falling edge of the clock.

   Workaround: If a board is experiencing this problem, PLLEN can be delayed a few ns to cause the signal to miss the window. PLLEN can also be synchronized to the rising edge of the clock to guarantee that the window will be missed.

GENERAL INFORMATION:

1. The cache should be flushed before setting the CI (Cache Inhibit) bit in the Supervisor and User Area Pointer Registers (SAPR, UAPR) , the BATC Write Ports (BWP0 - BWP7), and the PATC entries.

2. While in MBUS checker mode (MCE pin asserted), the Bus Request (BR) and Bus Acknowledge (BA) signals are not tristated. These signals will continue to drive valid states to allow for duplicate arbitration logic.

CURRENT DOCUMENTATION:

MC88200 User's Manual, MC88200UM/AD
MC88200 Technical Summary, Document BR589/D Rev. 1