Design and Implementation of The PLUS Multiprocessor

R. Bisiani, and M. K. Ravishankar

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

27 January 1991

1. Introduction

This document describes the design and implementation of a node in the PLUS shared-memory multiprocessor architecture. PLUS is a general-purpose, medium-scale multiprocessor which can have upto 64 processor nodes. Each node in the system is built around the Motorola 88000 processor and cache chip-set and contains upto 32 Mbytes of main memory, all of which is accessible from any node. The nodes are interconnected by a scalable, mesh-topology network for the purpose of integrating the distributed memory modules into a single, globally shared memory.

The main problem faced by any such architecture is the latency of remote memory operations, which are often an order of magnitude slower than accesses to local memory. Caching, in one form or another, is the obvious answer to this problem. At one extreme, we have purely software-based "caching" and coherency protocols. In these, memory is cached on a page-basis, with the main memory itself serving as a cache, and coherency is usually maintained by manipulating read and write permissions to the page. For example, if a given page is cached at many processors, each of them has read permission, but not write permission to the page. Whenever one writes to the page, it encounters a page fault and the fault handler executes a protocol that invalidates copies on other nodes, giving it sole ownership of the page. This scheme can be implemented virtually on any collection of processors interconnected by a network, but applications must be tuned extremely carefully, in order not to incur the overhead of the protocol too often. At the other extreme, there are hardware-based caches that employ demand-caching and complex coherency protocols. These are designed and built at considerable expense; their complexity increases rapidly with the size of the system. Inevitably, compromises must be made, and since these will be cast in concrete, require substantial investment to ensure that the design decisions are meaningful.

In the PLUS system, we combine the flexibility of software-based caching policy with the efficiency of coherence maintenance in hardware. A logical page, which is 4K-bytes in size, can be "cached", under software control, in the main memory of several nodes. This action is called *replication*, to distinguish it from the usual connotations associated with the word *caching*. System software can choose any one of the physical copies to map into the virtual address space of a given process; usually it is appropriate to pick the geographically nearest copy. In this way, page replication reduces both the cost of remote memory references and contention for that logical page.

The software makes the replication scheme visible to hardware via special *translation tables* maintained on each node, and hardware is responsible for keeping all copies of a page consistent. Unlike most cache and memory consistency protocols which rely on invalidation, PLUS uses an update protocol for this purpose. That is, whenever any processor writes to a logical page, the consistency maintenance mechanism sends messages around the system to update each copy; the granularity of an update can be as small as a byte. The update protocol has the following advantages over an invalidation protocol: It eliminates the need for additional valid bits in memory. Since copies are never invalidated, the nodes that own a copy never have to incur the overhead of remote memory references. Finally, compared to a purely software-based scheme, the entire time-consuming layer of consistency protocol implemented in software is eliminated.

Obviously, write operations to replicated logical pages will usually take a relatively long time to complete. In PLUS, this cost is hidden by adopting a *weak-ordering* memory system. That is, when a processor writes to such a page, the task of updating all copies is carried out by independent, dedicated hardware; the processor can continue immediately after issuing the write operation. As a result, several write operations issued by a single processor can be in progress in the system concurrently, and they may even complete out of order. (In a strong-ordering model, write operations to memory take place and complete in the order in which they are issued.) The PLUS coherency protocol guarantees that in the quiescent state, all memory copies will be consistent. In the dynamic state, however, one would sometimes need to explicitly ensure strong ordering of certain memory operations. For instance, a process may set a semaphore, update shared data structures, and then clear the semaphore, thus exposing the updated variables to other processes. It is necessary to ensure somehow, even in a weakly ordered memory system, that the semaphore is cleared only after all earlier memory operations have completed. The PLUS system provides a *fence* mechanism for just this purpose. That is, after a process executes a fence operation, any subsequent write or semaphore operation (discussed below) is suspended until all earlier memory operations have completed. The fence enforces strict ordering between operations preceding and following it. It is the programmer's responsibility to invoke this operation explicitly whenever necessary.

The last focus of research in the PLUS system concerns synchronization. The main problem here is to minimize the duration that any process has to spend in a critical region, thus maximizing the amount of concurrent activity. However, virtually all recent processor designs have tended towards simple, RISC-style operations, and this applies to semaphore operations, too. To accomplish anything moderately involved such as inserting or extracting an element from a shared queue, one has to execute a certain amount of code within a critical region; the simpler the semaphore operations, the larger the code. Moreover, this increases the interference and additional memory traffic imposed by processors that have failed to acquire a semaphore and keep trying for it repeatedly. The conclusion is that simple operations are acceptable when they can be implemented efficiently, but with semaphores it is better to provide a set of complex operations. In PLUS, we have identified about half-a-dozen such operations that are needed frequently. Semaphore variables can be anywhere in memory, they can even be in replicated pages; the operations are carried out atomically at the semaphore location and all copies are kept consistent. The complex operations often eliminate several round-trip network transits between the invoking processor and the memory location, that would otherwise be necessary with simpler schemes. These semaphore operations are not part of the native instruction set of the Motorola 88000 system, but have been defined by extending the processor *xmem* instruction. Therefore, they are called *extended* operations.

There is one final optimization with respect to the extended operations in PLUS. The synchronization operations (such as *fetch-and-add*) return one 32-bit word result. As we shall see later on, semaphore operations almost always require remote memory access; replication is of no help in this case. Consequently, the result is available only after some delay. To hide this latency, the processor that issued an extended operation is not required to wait for the result. Rather, the operation is carried out atomically and asynchronously by independent hardware, while the processor continues with normal instruction execution. It later issues a special *verify* operation to synchronize with the extended operation and retrieve its result.

Figure 1-1 is an overview of a PLUS node, consisting of a Motorola MC88100 processor, a data Cache/Memory Management Unit (CMMU), a code CMMU, 512 Kbytes of fast SRAM memory, the Mbus subsystem, a network interface (NI), and an external host interface. The Mbus subsystem is the heart of the global shared-memory logic. It has upto 32 Mbytes of DRAM main memory which is part of the global shared-memory space, and it contains all the necessary logic for maintaining consistency of replicated pages, as well as executing the extended operations. This requires a number of special functional units, such as an ALU for carrying out the extended operations, and tables for maintaining replication information. Most of these units can be addressed as I/O-space devices. A substantial portion of the Mbus subsystem control logic is implemented in microcode.

The NI is an interface between the Mbus subsystem and the PLUS interconnection network, which is a

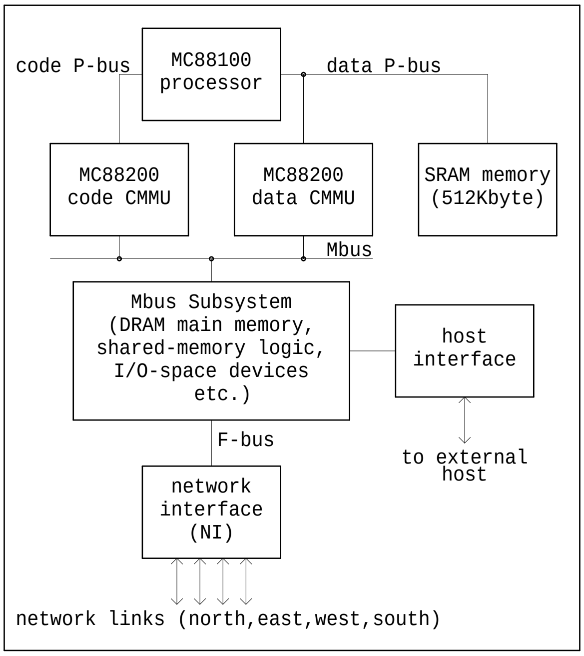


Figure 1-1: Overview of PLUS Node

2-D mesh network[1] for nodes to be able to access remote memory locations. The nodes can be configured in any rectangular mesh shape, as long as the size along either dimension is within 8. Whenever a node makes a remote memory reference, or updates a local memory location that is replicated on another node, the shared-memory control logic sends packets through the network to carry out the operation and keep all copies consistent. At the other end, the receiving node's shared memory control logic carries out the request, and sends back a reply packet if necessary. The NI provides an autonomous interface for buffering, and actually transmitting and receiving these network packets. It is connected to the Mbus subsystem via the F-bus.

The external host interface is a generic 32-bit interface between the node and a general-purpose workstation or *host*. This external host can, in principle, address any location that the node processor can address via the Mbus, including the local as well as remote main memory, and other local functional units. The external host is necessary to configure certain programmable logic array devices on the PLUS nodes, to download initialization code into the system, and to carry out physical I/O operations on behalf of the PLUS system. There is also a FIFO buffer in this interface, in which memory accesses made by the local processor are recorded. The external host can monitor the dynamic memory access patterns via this buffer, in order to tune the page replication structure to improve performance.

The 512 K-bytes of SRAM memory reside on the data P-bus of the processor. Since the code and data caches are relatively small (16K bytes each), one can greatly improve the performance of selected applications with the help of the fast SRAM memory. However, there are a couple of limitations to it: this memory is virtually addressed, and is private to the node. That is, it has to be explicitly managed by the user, and it cannot be addressed from other remote nodes.

[1]The network topology is not part of the focus of this research project. Our implementation employs an off-the-shelf mesh-topology network which has sufficient bandwidth and speed for our purposes.

The rest of this document begins with a detailed overview of the PLUS system in Section 2, including details of the replication structure, coherency requirements, and extended operations. It also contains a first look at the major details of implementation; the emphasis here is on the data path. The individual components of the Mbus subsystem datapath are described in detail in Section 3. The overall operation of a PLUS node is described in Sections 4 and 5, with the former containing most of the hardwired control logic and the latter the microcoded logic, which is responsible for overseeing most of the major functions in the PLUS architecture. The interface to an external host is covered fully in Section 6, and Section 7 finally concludes with brief notes on some software issues. A separate document, entitled *PLUS Prototype Design Documentation*, contains the schematic diagrams and all the PAL device definitions.

2. Overview of the System

2.1 Replication and Coherency Mechanism in PLUS

Given that a logical page can be replicated on multiple nodes, and the assumption of a weakly-ordered memory system, there are three basic requirements of the system:

- The first is that in the quiescent state, all copies of any replicated page should be consistent. That is, if all processors stop writing to memory, eventually (when the pending write operations have finished propagating through the system) all copies of any replicated page should become identical. This property allows a processor to map any physical copy into its virtual address space, and such a system is termed *weakly-coherent*.
- In the dynamic state in a replicated, weakly-ordered memory system, write operations may finish out of order, and there can be transient inconsistencies among replicated copies. The second requirement of the PLUS system is that a single processor still see its own actions in a strongly-ordered way. That is, whenever it reads any memory location, it receives the latest value that it wrote to that location, assuming no other processor overwrote it in the meantime. Without this assumption of *write-read consistency*, it would be extremely tedious, if not impossible, to write meaningful and understandable programs.
- Finally, if a node executes the *fence* operation, all its operations succeeding the fence should be strongly ordered with respect to its operations preceding the fence. That is, the former can proceed only after all of the latter have completed. The fence mechanism allows the user to selectively impose, whenever necessary, a strong ordering on the actions of one processor as seen by other processors.

In addition, one must not forget the need to keep *cached* locations consistent with replicated memory. This is especially tricky as caching policies and consistency mechanisms are usually totally different from the replication scheme. This Section describes the basic ideas behind the solutions to these issues in the PLUS architecture.

The consistency maintenance protocol in PLUS is quite straightforward: First, all physical copies of a given logical page have to be ordered (by software) in a linear list called the *copy-list*, the first of which is designated the *master* copy. Any write or synchronization operation to the logical page by any processor must always begin at the master copy and propagate down the copy-list, in order. This is sufficient to guarantee that the memory system is weakly coherent.

In order to allow the hardware to maintain consistency of replicated pages, the replication structure is exposed to the hardware by means of special *translation tables* distributed among all the nodes. Each node includes a *master-copy* table and a *next-copy* table, each containing one entry per local physical page. The first table contains pointers (i.e., node address and physical page location on that node) to the master copy for each local physical page, while the latter points to the next copy, if any, along the physical page's copy-list. Note that as far as the hardware is concerned, the replication structure is static. It can only be modified by software, and whenever it does so, the tables must be updated.

For any given logical page, a processor is free to map any convenient physical copy, local or remote, into its address space. The physical address obtained after a logical address undergoes address translation identifies both a physical node and the physical address of the location on that node. (The physical address encoding scheme is described in detail in Section 2.3.2.) If the processor executes a read operation, it is performed on the addressed node. If it is a write or an extended operation, however, the coherence maintenance hardware carries out the following algorithm:

```
if (physical address points to local copy) {
    if (master-copy entry for this page points to local page) {
        carry out the operation;
        if (next-copy entry for this page points to a valid next copy) {
            get the physical address of the next copy from next-copy table;
            send an update message to the node with the next copy;
        }
    } else {
        get the physical address of the master copy from master-copy table;
        forward the operation to the node with the master copy;
    }
} else {
    /* physical address points to remote node and location on that node */
    /* NOTE -- the remote node may not have the master copy!! */
    forward the operation to the remote node;
}
```

When a node receives a write or semaphore operation request from a remote node as a result of the above, it executes the following steps. Note that such a request must be to a local physical page.

```
if (master-copy entry for this page points to local page) {
    carry out the operation;
    if (next-copy entry for this page points to a valid next copy) {
        get the physical address of the next copy from next-copy table;
        send an update message to the node with the next copy;
    } else {
        return an acknowledgement to the node originating the operation;
    }
} else {
    get the physical address of the master copy from master-copy table;
    forward the operation to the node with the master copy;
}
```

Finally, when a node receives an update request, it performs the following steps. Note that such a request must be addressed to a local physical page.

```
write (i.e., update) local memory;
if (next-copy entry for this page points to a valid next copy) {
    get the physical address of the next copy from next-copy table;
    send an update message to the node with the next copy;
} else {
    return an acknowledgement to the node originating the operation;
}
```

The reason for the acknowledgement messages will become clear in a moment. With a little examination, it should be obvious that the above mechanism guarantees that the memory system is weakly-coherent (assuming that the network preserves the chronological ordering of messages between any given pair of nodes).

The remaining two requirements of write-read consistency and strong ordering of operations around a fence are met by means of a hardware *write tag buffer* (or WTB) on each node. Whenever a processor writes to a remote location, or a location that is replicated on other nodes, the global physical address presented by the processor is captured in the WTB. (If no free WTB entry is available, the operation is suspended.) The same thing happens if the processor invokes an extended operation. The WTB entry is freed after the operation is carried out on all the copies and an acknowledgement is finally received. The WTB enables write-read consistency to be achieved by suspending the processor whenever it reads a location that matches a valid entry in the WTB. The fence mechanism is implemented by suspending any write or extended operation issued after a fence, until the WTB becomes empty, i.e., all outstanding entries are acknowledged.

The PLUS system can, in fact, also support execution in a completely strong-ordering mode. In this mode (which is enabled by a control bit that can be turned on selectively by software) every write or extended operation to memory by a given node must complete before the next such operation by that node is allowed to start. This ensures that memory always reflects the true ordering of operations as issued by the processor, and is useful for running programs that rely on this memory model.

Finally, note that while the master-copy and next-copy tables capture the static replication structure, it is impossible in the PLUS architecture to track the dynamically cached entries to keep the caches coherent with replicated memory. For this reason, we rely on software-enforced restrictions on caching policies to keep both cached and replicated copies coherent, as follows. If a node has no physical copy of a given logical page, it cannot cache the page at all. The coherency mechanism in PLUS relies on the page's copy-list to keep everything consistent, but the copy-list is not concerned with nodes that have no physical copy of the page. If a node has a physical copy of a replicated page, it should cache it only in write-through mode. This ensures that whenever the processor on the node writes to that page, the operation comes through the cache where it is visible to the coherency maintenance hardware. In addition, whenever some other node writes to this logical page, any cached entry should be either updated or invalidated when the physical memory on the node gets written. In a PLUS node, the caches are part of the Motorola MC88200 CMMU devices, which monitor or *snoop* writes to their local memory caused by remote processors, and invalidate any cached entry for that memory location.

2.2 Extended (Synchronization) Operations

The PLUS architecture includes a variety of synchronization operations, from simple ones such as *fetch-and-set* to sophisticated queue operations. All these operations return one 32-bit word result which is the current value of the memory location of interest, and may atomically update the memory location. As we saw in Section 2.1, extended operations must be executed first at the master copy location, and any changes to memory are propagated to other nodes along the copy-list. Therefore, the result of any extended operation has to come from the master copy location. Obviously, the probability of the local copy being the master copy is pretty slim. For this and other reasons, the extended operations are executed asynchronously and atomically by independent hardware, while the processor is allowed to proceed with instruction execution, thus absorbing their latency. The processor later executes a *verify* operation to synchronize with, and retrieve the result of the extended operation that it issued earlier. As a further optimization, a processor is allowed to have multiple extended operations in progress concurrently; it is not required to verify the result of one extended operation before it can issue the next. For example, a process could insert several items one after another in a shared queue. They would all execute in a pipelined fashion, thus achieving a high degree of concurrency. In PLUS, each processor can have upto 8 extended operations in progress concurrently. Each invocation returns a 3-bit verify tag or *vtag*, which is used along with the verify operation to retrieve the corresponding result.

The extended operations, naturally, are not in the native instruction set of our chosen processor, the Motorola MC88100. Rather, they are invoked by extending its instruction set, by redefining the *xmem* instruction. Normally, this instruction would atomically exchange a register with a memory word which is the semaphore variable. In PLUS, the *xmem* instruction is invoked with the extended op-code in the top 2-6 bits of the register argument (a variable-length encoding is used); the remaining bits in the register may contain a data argument for the extended operation.[2] Instead of returning the value in the addressed semaphore location, the hardware automatically allocates and returns a verify tag (i.e., after completing the *xmem* instruction, the register argument contains the verify tag), and asynchronously and atomically executes the extended operation on the addressed memory location. It is an error if no free tag is available when the *xmem* instruction is executed.

Table 2-1: Extended Operations

Name	Opcode	Description
XCHG	00	Exchange. Return the current word in the addressed location, and replace it with the zero-extended 30-bit argument.
CXCHG	01	Conditional exchange. Return the current word in the addressed location, and, if its MSB is 0, replace it with the zero-extended 30-bit argument and set its MSB.
ENQ	10	Enqueue. The addressed location must be the first word in the queue page (i.e., the tail element offset). Return the current word pointed to by this offset, and if its valid bit is clear enqueue the zero-extended 30-bit argument. That is, write the argument, set its valid bit, and update the offset. (See text.)
FADD	1100	Fetch-and-add. Return the current word in the addressed location, and add the sign-extended 28-bit argument to the location.
MIN	1101	Minimum. Return the current word in the addressed location, and replace it with the sign-extended 28-bit argument if it is less than the current value.
FSET	111100	Fetch-and-set. Return the current word in the addressed location, and set its MSB.
DEQ	111101	Dequeue. The addressed location must be the *second* word in the page (i.e., the head element offset). Return the current word pointed to by this offset, and if its valid bit is set, dequeue the element, i.e., clear the valid bit and update the offset. (See text.)
DRD	111111	Delayed or deferred read. Return the current word in the addressed location.

Table 2-1 describes the set of extended operations in PLUS.[3] Note that the op-code is embedded in the most significant 2-6 bits of the register data argument of the *xmem* instruction. The remaining bits are extended on the high-order side into a full 32-bit data argument, if necessary, as described in the table. The ENQ and DEQ functions operate on a queue structure which occupies one entire 4K-byte page. The first word of the page is a 12-bit byte offset from the top of the page, pointing to the first free element of the queue, i.e., where the next element can be enqueued. The second word, likewise, is an offset to the head of the queue from where the next element can be dequeued. The rest of the page comprises a 1022-element circular buffer (so the offsets wrap back to 8 from 4092). The MSB of each word is a *valid* bit that is set if and only if there is an element queued there. This structure is set up by software when the queue is allocated. It is then maintained entirely by hardware whenever any processor invokes the ENQ or DEQ operation on the queue. The operations fail if the addressed queue is full or empty, respectively; the outcome can be known by examining the valid bit of the returned result. All the other extended operations function on a single memory word, as described in the table.

---

[2]This implementation only recognizes word versions of the *xmem* instruction; byte versions will behave unpredictably. Also, the addressed location must not be in SRAM memory or some I/O-space region (see Section 2.3.2). Otherwise, the *xmem* instruction will be executed normally, not as an extended operation.

[3]At the moment, only four of them have been implemented--FADD, FSET, ENQ, and DEQ. Furthermore, the queue pages (i.e., locations addressed by ENQ and DEQ operations) must not be replicated.

It is beyond the scope of this document to discuss the relative merits of the individual synchronization operations; each has its own unique advantages. The CXCHG operation is intended to be used in conjunction with FADD and XCHG to implement, in software, queue structures that are larger than a single page. The FADD operation can be used to manipulate the offsets, and CXCHG enqueues a word if the valid bit is clear, while XCHG is used to dequeue and clear the valid bit. FADD can also be used in other ways, such as for barrier synchronization. DRD is not a synchronization operation at all. Rather, it is meant for *prefetching* selected words from remote memory, to reduce the access latency.

2.3 The Mbus Subsystem

All of the architectural features described so far, replication, coherence maintenance in hardware, and the extended operations, are implemented in the Mbus subsystem shown in Figure 1-1. This section provides a brief overall picture of the structure of the Mbus subsystem on a PLUS node. The emphasis here is on the data path structure; we do not examine the control logic in much detail at this moment.

2.3.1 The Mbus Subsystem Datapath

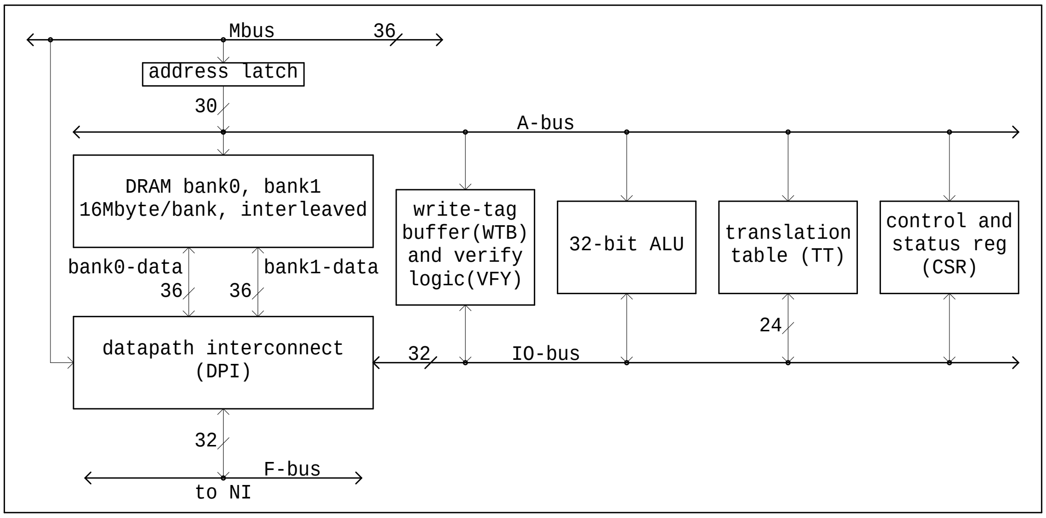


Figure 2-1: The Mbus Subsystem

The Mbus Subsystem of Figure 1-1 is presented in greater detail in Figure 2-1. Only the data path is shown, the control logic is not included in this figure. The Mbus subsystem contains four main buses, as shown. The Mbus and F-bus were briefly introduced in Figure 1-1.

The Mbus is the main bus on each node over which all accesses to the node's memory and other functional units take place, whether they originated locally or remotely. It is a physical address bus, and the Mbus subsystem deals with global physical addresses. That is, a logical address presented by the processor first undergoes virtual-to-physical translation in the CMMUs, and the resulting 32-bit physical address appearing on the Mbus identifies both a node and the physical memory location on that node. The structure of this physical address is defined later in Section 2.3.2. There may be physical copies of the page replicated on other nodes as well, but a given node must always map the logical page to the same physical copy. Whenever the shared-memory control logic needs to know about remote copies of local physical pages, i.e., to carry out the coherence protocols, it has to consult the master-copy and next-copy tables introduced in Section 2.1. For details regarding the Mbus specification, please consult the

Motorola MC88200 CMMU manual.

The A-bus is the physical address bus driven by an address latch on the Mbus. Finally, all non-memory functional units that can be read or written by the processor (except the CMMUs) reside on the IO-bus. The devices on this bus are not physical I/O devices; rather, they comprise various functional units required by the shared memory control logic, as well as for the execution of the extended operations. Although distinct from the Mbus, the IO-bus usually functions as an extension of the former.

The individual blocks shown in Figure 2-1 are briefly described below, and references to the actual design drawing pages are included. Each of them is treated in much greater detail in Section 3.

2.3.1.1 Address Latch

Every Mbus transaction address is latched in the address latch, from where it is broadcast over the A-bus to several destinations within the node, as shown. The A-bus is 30 bits wide, and specifies the address of a 32-bit word. The address latch is shown on page 2 of the schematic drawings, along with a lot of other control logic.

2.3.1.2 DRAM Main Memory

The global, shared memory in the PLUS system is partitioned among all the nodes. Each node has two interleaved memory banks, each of which can be either 4 or 16 Mbytes in size, depending on whether one uses 1 or 4 Mbit DRAM devices. Memory on different nodes can be configured differently. The two banks are interleaved on the low order word address. Each bank also has a parity bit per byte.

The node's main memory and its controller are described in Section 3.1. See pages 5 and 6 of the schematic diagrams for other details.

2.3.1.3 Translation Table (TT)

The translation table is a separate block of SRAM-based hardware that contains both the master-copy and next-copy tables introduced in Section 2.1 above. Recall that each of these subtables contains one entry for each page on the local node. Since the node can have upto 32 Mbytes, i.e. 8K pages, the two tables together require 16K words in the TT. The master-copy and next-copy subtables occupy the top and bottom halves of this SRAM memory, respectively. The TT can be indexed by the page-number field of a local memory physical address to identify the master copy and next copy along the copy-list for that address.

The structure of the TT and the format of each entry are described fully in Section 3.2. Further details can be found on page 10 of the schematic diagrams.

2.3.1.4 Write Tag Buffer (WTB)

The need for the WTB was established in Section 2.1. The shared memory control logic on the node captures the physical address of any write or extended operation executed by the local processor in the WTB, if there is a remote copy of that physical page. The control logic causes the operation to be propagated to all the copies according to the coherency maintenance algorithm, after which an acknowledgement is finally received. At that point the WTB entry is freed. There can be upto 8 entries in the WTB. If no entry is free when the local processor executes a write or extended operation to memory, that operation is suspended.

The WTB is used to implement write-read consistency and the fence operation. If the local processor reads a location that matches an entry in the WTB, it is blocked until all matching entries have been freed. Likewise, after the processor invokes a fence, any write or extended operation to memory is blocked as long as the WTB is not empty. It is also needed in order to run the system in a completely strong-ordering mode (see the STRORD~ CSR bit in Section 3.6.3). In this mode, if the processor issues a write or extended operation to memory and the WTB is not empty, the operation is suspended.

The WTB, consisting of 8 address entries and comparators to detect a match, is implemented in two Xilinx programmable logic cell array devices. It is described in Section 3.3, and shown on page 8 of the design schematics.

2.3.1.5 ALU and Verify Control Logic (VFY)

This 32-bit ALU, implemented using two 16-bit Cypress Semiconductor CY7C9101 bit-slice devices, is mainly used to carry out the extended operations described in Section 2.2. It also has a 16-word register file, 8 of which are used as temporary storage for the results of the extended operations. Recall from Section 2.2 that the processor on a node can issue an extended operation and immediately proceed with instruction execution, while the extended operation is carried out independently by dedicated hardware, finally returning a 32-bit result. This result is, in fact, held in one of the registers in the ALU, until the processor can read it with a *verify* operation.

The verify control logic maintains the status of all pending extended operations executed by the node. It includes status bits for each verify register in the ALU, indicating which ones have been allocated and which contain a valid result. When the processor invokes an extended operation by executing the *xmem* instruction, this logic also allocates a free verify register and returns a 3-bit verify tag, or *vtag*, identifying it. (If no such free register exists it is an error.) Finally, if the processor attempts to verify the result before it becomes available, the status bits flag the situation and the operation is suspended.

The ALU may also be used by the control logic to carry out page copy operations across nodes to alter the replication structure. See Section 3.5, and page 9 of the schematic drawings, for further details about the ALU. Section 3.4 describes the verify control logic, and its details can be found on page 8 of the schematic diagrams. Most of it is implemented in the same two Xilinx devices as the WTB.

2.3.1.6 Control and Status Register (CSR)

Each PLUS node includes an assortment of control and status bits. These include the node's 6-bit ID register, various interrupt request bits, and error status bits. Some of these are controlled by the processor, others by hardwired control logic, and yet others by the microsequencer. Section 3.6 covers these in full detail, and their implementation is shown on page 18 of the schematic drawings.

2.3.1.7 Datapath Interconnect (DPI)

This block interconnects five different data buses or ports of the Mbus subsystem: the Mbus itself, data ports of the two main memory banks, the IO-bus, and the F-bus. The DPI contains several multiplexers and buses to route data between the 5 ports. The F-bus interface in the DPI includes a double-buffered input packet register and an output packet register, which can both be accessed by the NI. In addition, the DPI contains several miscellaneous units, such as a 32-bit free-running counter synchronized among all the nodes for performance evaluation purposes, a timer for detecting Mbus timeout errors as well as for generating a periodic interrupt signal for the MC88100 processor, and counters for measuring the data cache hit-rate. Owing to its central location, the DPI has a vital role in the execution of virtually every activity on the node, whether initiated by the local processor, its external host, or from another node.

The DPI is implemented in four Xilinx programmable logic cell array devices. It is described in detail in Section 3.8. The input and output packet registers are separately covered in Section 3.7. Also see page 7 of the schematic figures.

2.3.2 Addressing

As mentioned at the beginning of Section 2.3, the Mbus subsystem deals with global, physical addresses. The 32-bit physical address appearing on the Mbus is decoded into three regions, depending on address bits 31-20, as shown in the figure below:

	31	30 … 0		
local main memory	0	local main memory address		
	31	30 … 25	24 … 0	
remote main memory	1	xxxyyy	main memory address on remote node	
	31	30 … 20	19 … 16	15 … 0
local I/O-space device	1	11111111111	dddd	address within selected device

xxxyyy = remote node ID, dddd = I/O-space device no.

Bits 30-25 of a remote main memory address, shown as *xxxyyy* above, are actually the 6-bit remote node ID. This field is further divided into two 3-bit subfields: bits 30-28 (*xxx*) and bits 27-25 (*yyy*), forming an <*x*,*y*> co-ordinate address in the 2-D mesh topology of the PLUS multiprocessor.[4]

The above encoding implies that there can be at most 64 nodes in a PLUS system. Upto 32 Mbytes of main memory can be installed on each node, all of which is directly addressable from any node, with the following one exception: The address mapping scheme precludes mapping the top 1 Mbyte of the main memory on node <7,7> into the virtual address space of other nodes. Doing so would require bits 31-20 of the physical address to be all 1's. But this is instead interpreted as local I/O-space. It is, however, possible for nodes to address this region of memory *indirectly*, by having it replicated elsewhere and mapping that copy into their virtual address space.

The I/O-space region does not contain physical I/O devices, really. Rather, it includes all the special functional units of the node's Mbus subsystem, such as the ALU, TT, CSR, etc., so that they can be initialized and maintained by the processor. In addition, many special operations, such as *fence*, *verify*, and interrupts to other nodes or external hosts, are also mapped into read or write operations on specific I/O-space pages. There are 16 subspaces in a node's local I/O-space region, identified by address bits 19-16. The regions currently in use are laid out in table 2-2. Notice that sometimes the same physical device appears in more than one I/O-subspace, depending on the operation. Most of the devices are described in greater detail in Section 3.

In general, it is only possible to read or write an entire word of an I/O-space functional unit, such as the TT or the ALU. That is, it is not possible to address an individual byte within a word in these units. It is not possible to address I/O-space regions on remote nodes.

An external host attached to a given node can access main memory on any node in the system, and the

---

[4]The *x* and *y* dimensions are also known as the east-west and north-south axes of the PLUS interconnection network, respectively.

Table 2-2: I/O-space Regions

Name	Addr 19-16	Usermode Privileges	Description
TTPG	0	None	The 16K-word translation table. (Section 3.2.)
ALUPG	1	None	ALU registers 0-15, selected by address bits 5-2. System software saves and restores the ALU verify registers during context switches. (Sections 3.4 and 3.5.)
VFYPG	2	Read-only	Reading this region starts a *verify* operation. Address bits 4-2 contain the verify tag or *vtag*. Bit 5 must be 0. (Sections 3.4 and 3.5.)
VWSTPG	3	Read-only	Reading this region returns one word of VFY and WTB status bits. This allows a user process to identify extended operations that have completed and whose results can be verified. System software saves and restores the status bits during context switches (Section 3.4). The WTB status bits are included only for hardware debugging.
VWRSTPG	4	None	System software writes to this region to reset VFY and WTB status bits. It may use this operation to recover from error conditions.
FENCEPG	5	Write-only	Writing to this page invokes the *fence* operation.
CSRPG	6	None	A one-word control and status register. (Section 3.6.)
CMMUPG	7	None	Internal registers of the two CMMUs. The code CMMU is at base address *0xfff7e000*, the data CMMU at *0xfff7f000*.
CTRPG	8	Read-only	32-bit free-running counter used for performance measurement. (Section 3.8.6.4.)
CACHEHITPG	9	Read-only	32-bit word containing the values of the data cache hit-rate counters, used for performance measurement. (Section 3.8.6.5.)
IDPG	10	None	System software writes to this region to initialize the node's ID registers. There are two ID registers on each node: in the DPI and the CSR (Sections 3.8.6.3 and 3.6.4); both are initialized. The value written must contain the node's ID in bits 30-25 *and* bits 5-0.
RMTPINTPG	12	None	System software on a node or external host writes to this region to interrupt the processor on another remote node. The value written must contain the remote node's ID in bits 30-25. (Section 3.6.1.)
RMTEXTINTPG	13	None	System software on a node writes to this region to interrupt the external host on a remote node. The 32-bit value written must contain the remote node's ID in bits 30-25. (Section 3.6.1.)

following I/O-space regions on the local node: CSRPG, CMMUPG, CTRPG, CACHEHITPG, and RMTPINTPG. In addition, the host can read TTPG, ALUPG, and VWSTPG regions for hardware and software debugging purposes.

2.3.3 Mbus Subsystem Control

The Mbus subsystem is controlled by a combination of hardwired logic and a PROM-based microsequencer. Only the most frequent and time-critical operations are implemented in hardwired logic. For example, if the processor on a node initiates an Mbus transaction to read a local main memory location, or write such a location that is not replicated on any other node, the transaction is handled entirely by a hardwired memory controller. All other processor transactions involving remote memory locations, local I/O-space regions, or extended operations are handled by the microsequencer. If the addressed location is a remote memory word, the microsequencer sends a request message to the remote node to carry out the operation. Or, if it is a local memory write or extended operation, but the addressed page is replicated, then, too, the microsequencer must send messages out to ensure that all copies of the addressed page are kept consistent. And, finally, if the transaction is to an I/O-space region, the microsequencer is required to identify the specific device that is addressed, and directly control it in order to complete the transaction.

The microsequencer on a node is also responsible for handling requests arriving over the network from other nodes, or from the local external host, and for this purpose it must behave as an active Mbus master. It is quite possible for the microsequencer, as bus master, to address some non-memory location, in which case it has to behave as both master and slave simultaneously.

The above separation between hardwired logic and microcoded logic is intended to minimize the complexity of control hardware, while preserving the efficiency of the most frequent operations, i.e., processor accesses to local memory. Since the majority of Mbus transaction types are implemented in microcode, large portions of the details of implementation can be changed relatively easily. The microcode space consists of 2K words, where each word is 72 bits wide. The current implementation uses 64 of these bits; the rest are spares.

In addition to the local main memory controller, there are a number of other hardwired control units for general housekeeping, such as timing out hung transactions, and others that play a part in enforcing various memory consistency requirements described in Section 2.1. Most of these are contained on pages 2 and 18 of the schematic diagrams. In this document, they are described in Section 4. The design of the main memory and its controller is explained in Section 3.1, and can be found on page 6 of the design drawings. The microsequencer and some control logic closely associated with it are described in Section 5 in detail, and depicted on pages 16, 17, and 18 of the schematic drawings.

2.4 The Network Interface

The PLUS system uses a mesh topology packet switching network in order to integrate the main memory modules on all the nodes into a single shared-memory space. Since packet transmission and reception is a fairly complex and time-consuming process, it is relegated to an independent controller in the network interface, whose design is outlined in this section. It is fairly straightforward, and we will not discuss it beyond this section in this document.

2.4.1 NI Datapath

The Mbus subsystem contains an input packet register (IPR) and an output packet register (OPR). The microsequencer (i.e., the primary Mbus subsystem controller) places packets to be transmitted in the OPR and signals the NI, which then autonomously sends them over the network to the appropriate destination node. Similarly, incoming packets are automatically received by the NI and properly formatted and placed in the IPR one at a time, for processing by the microsequencer. The NI on each node includes FIFO packets buffers, byte-wide full-duplex links to its four neighbouring nodes, and packet routing logic. The IPR and OPR on the node are accessible to it via the F-bus. Figure 2-2 shows a block diagram of the datapath of the NI; the control logic is not shown. The function of each block in the figure is briefly outlined in the following paragraphs.

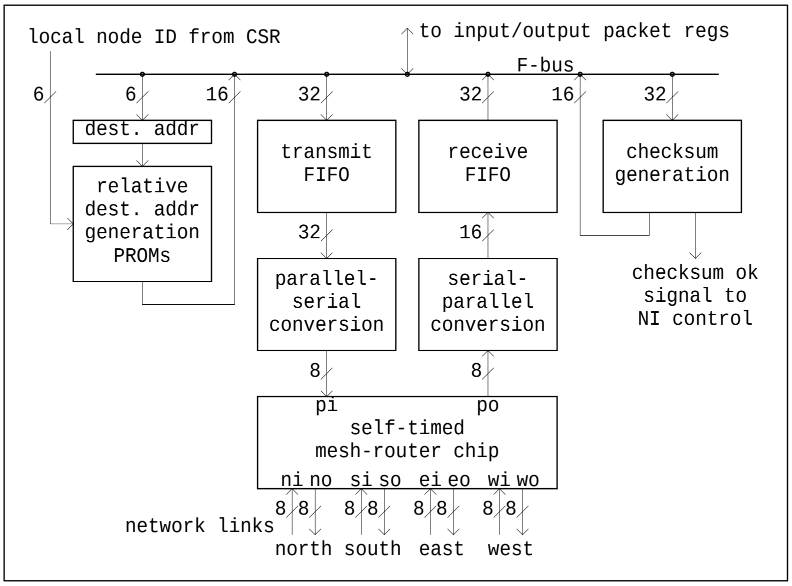


Figure 2-2: The Network Interface

2.4.1.1 Packet Router Chip

The NI uses the self-timed mesh-router chip (MRC), designed at the California Institute of Technology, for routing packets through the PLUS system. This chip has 5 full-duplex, 8-bit wide data ports: one to the local node (*pi* and *po*), and one each to the north, south, east, and west neighbour nodes. It is the MRCs of the PLUS system that are interconnected in a mesh topology. They automatically route packets to their appropriate destinations without any intervention from the intermediate nodes. The MRC and its self-timed protocol are described elsewhere in detail. Also see Mead and Conway's *Introduction to VLSI Systems*, Section 7.7, entitled *Self-timed signaling*.

One significant advantage of using the self-timed MRCs for message routing is that the various nodes can be clocked totally independently. In fact, they can even be clocked at different rates; there is no need to distribute a single, synchronous clock throughout the entire system.

2.4.1.2 Transmit and Receive FIFOs

Outgoing packets placed in the OPR are first buffered in the transmit FIFO by the NI. Likewise, incoming packets are buffered in the receive FIFO before being transferred to the IPR. The FIFOs serve three goals: First, they smooth out any bursty behaviour in the rate at which a node generates or consumes packets. Second, being asynchronous devices, they isolate the unclocked, self-timed logic of the packet router from the rest of the node. Finally, they help avoid deadlocks in the network. The packet routers themselves are deadlock-free. However, nodes could still deadlock due to insufficient buffering within them. For instance, if two nodes flood each others' packet buffers, they can eventually reach a state in which each wants to transmit a packet, but needs the other to consume one to free a packet buffer. In the PLUS system, there is a limit on the number of unconsumed packets a node could have introduced into the network at any time. Given this restriction, the FIFOs are sufficiently deep that the above deadlock situation can never arise.

2.4.1.3 Parallel-Serial and Serial-Parallel Conversion Logic

As one can see from the figure, the FIFOs are 32-bits wide, while the interface to the MRC is only 8-bits wide. Hence, the NI contains parallel-serial circuitry to multiplex the transmit FIFO data onto the 8-bit port of the MRC. The receive FIFO, however, is written 16 bits at a time, although it is 32 bits wide. Incoming bytes from the network are assembled into 16-bit quantities and written into the appropriate half of the FIFO.

2.4.1.4 Relative Address Generation PROMs

The destination node address in the OPR is a 6-bit absolute value (similar to the node ID field of a remote memory address, Section 2.3.2). The MRC, on the other hand, uses a relative <*x,y*> addressing scheme (see Section 2.4.2 below, and the MRC documentation). Therefore, before a packet in the OPR is transferred to the transmit FIFO, these PROMs, in conjunction with the node's ID register, are used to translate the 6-bit absolute address into a 16-bit relative address.

2.4.1.5 Packet Checksum Generation and Verification

The NI computes a simple 16-bit checksum and appends it to every transmitted packet, and verifies it on received packets. The checksum is essentially a bit-wise exclusive-or of individual 16-bit half-words of the packet. If the checksum on an input packet is incorrect, the NI signals an error which can be monitored by an external host.

2.4.2 Packet Formats

The NI can transmit packets of four different formats. The largest type of packet is illustrated below; the number of bits in each individual field is shown in parentheses:

	31					0
Control	destination(16)		type(6)	strb(4)	wtag(3)	vtag(3)
Address	0	origin(6)	address(23)			00
Data	data(32)					
	checksum(16)					

The control word is present in all packets, but both the address word and the data word are optional, hence the four different packet formats. The most significant two bits of the type field indicate the packet format as follows: The address word is present if the type field MSB is 1, and the data word is present if the following bit is 1. The control word is always transmitted first, followed by the address word, if any, and then the data word, if it is present. The checksum is transmitted last. Within a word, the most significant byte is transmitted first.

The individual packet fields are briefly described below:

Destination The relative address or displacement of the destination node with respect to the transmitting node. It actually consists of two 8-bit subfields, the most significant byte being the relative displacement in the *x*, or east-west direction, and the next byte that in the *y*, or north-south direction. This field is present in a packet only before it is transmitted. It is stripped off by the router chips by the time the packet reaches its destination. (See the MRC documentation.)

Type The packet type, defining the operation requested or the type of the result. The top two bits are constrained according to the packet format as described earlier. The different packet types and their contents are defined in Appendix II.

Strb Byte strobes, indicating which bytes of the 4-byte data field, if any, are relevant to the packet. It is obtained from the Mbus byte strobes control field MC<3:6>.

Wtag Tag identifying a WTB entry allocated for a write or extended operation. It is transmitted along with the request packets to carry out the operation, and finally returned in an acknowledgement packet to the originating node to free the WTB entry.

Vtag Tag identifying a VFY entry (register in the ALU) allocated for the result of an extended operation. It is transmitted along with the extended operation request packet, and returned with the result of the operation to the originating node.

Origin Usually present only in request packets. It is the absolute address of the node originating the request, so that any response or acknowledgement can be returned to it.

Address Usually present only in request packets. It is the main memory word address on the destination node on which the requested operation is to take place.

Data In request packets, data to be written or argument of an extended operation. In response packets, the result of a read or extended operation.

Appendix II contains a list of packet types currently defined in the PLUS system.

2.4.3 NI Control

The NI control logic is partitioned among four independent state machines, one at either end of each FIFO. One state machine transfers packets from the OPR, whenever it is full, into the transmit FIFO. In the process, it translates the destination address field of the OPR from an absolute to a relative address form as required by the MRC. It also calculates the checksum and appends it to the end of the packet in the FIFO.

A second state machine handles the reverse operation. It transfers packets from the receive FIFO into the IPR. The IPR is double-buffered, so this state machine can be assembling a packet in the first stage while an earlier one in the second stage is still being processed by the microcoded control logic. The state machine also verifies the received checksum and signals the external host if it is incorrect. Since these two state machines share the F-bus, they cannot be active simultaneously; the NI has an arbitration mechanism to resolve conflicts between the two.

The remaining two state machines actually transmit and receive packets in the two FIFOs. They control the parallel-serial conversion logic and the transmit and receive ports, *pi* and *po*, of the MRC.

The transmitting portion of the NI is depicted on page 11 of the schematic drawings, and the receiving circuitry on page 12. Page 13 shows the router chip and its connections. The control logic for transferring OPR packets into the output FIFO is implemented in PALs *XMTFSM0* and *XMTFSM1*, while that for transferring input FIFO packets into the IPR is in PALs *RCVFSM0* and *RCVFSM1*. The two state machines controlling the MRC are implemented in PALs *MRCPIFSM* and *MRCPOFSM*, and the checksum control and accumulation logic in PALs *CHKSMCTL* and *CHKSM0-3*. For further details see the related documentation *PLUS Prototype Design Documentation*.

2.5 The SRAM Memory

The half-megabyte, single wait-state SRAM memory on the data P-bus can be used by applications as a software-controlled cache in order to improve their performance appreciably. The data CMMU can sustain one request from the processor every P-bus cycle as long as the transaction hits the cache. As soon as there is a cache miss and an access has to go through the Mbus, however, there is an order of magnitude slow-down. Since the data cache is a rather modest 16-Kbytes in size, cache misses can be fairly frequent. The SRAM memory is an architectural feature to overcome this limitation. However, since it resides directly on the processor's data P-bus and is addressed virtually, it must be managed entirely in software and used with care. There is no hardware protection or other support for managing this memory.

The data P-bus address is decoded to select either the data CMMU or the SRAM as follows: The former is addressed if the transaction is to supervisor space (the data P-bus S/U~ bit is high), or address bits 31-26 are *not* 111110. The SRAM memory is addressed if the transaction is to user space and address bits 31-19 are 1111100000000.[5] If neither device is addressed, a P-bus fault occurs. Both read and write accesses to the SRAM memory incur one wait cycle. Since the P-bus protocol allows transactions to be pipelined, a set of address and data pipeline registers separate the SRAM memory module from the data P-bus. The registers also prevent this memory from loading the P-bus too heavily.

The SRAM memory is controlled by three PALs: *SRAMDCD* to decode data P-bus addresses, and *SRAMCTL0* and *SRAMCTL1* to handle the bus protocol. For further design details, refer to pages 3 and 4 of the schematic diagrams, and PAL definitions in the document *PLUS Prototype Design Documentation*.

[5]This address decoding is based on the belief that the SRAM memory is more useful to user processes, rather than to system software.

3. Design of the Mbus Subsystem

In the last Section we presented an overview of the entire system, including a brief, functional description of various components on a node. In this section, we concentrate on the detailed design of individual functional units in the Mbus subsystem. We describe the interface of each unit to the rest of the node, and in particular, to the microcoded control logic. The description also includes references to the actual design drawings. Each block is treated fairly independently; we shall see how the Mbus subsystem works as a whole later in Sections 4 and 5.

3.1 DRAM Memory

This section describes the implementation of the main memory on a PLUS node. Figure 3-1 shows a block diagram of the interleaved memory banks, and the pipeline stage between the memory data ports and the Mbus, implemented in the DPI Xilinx devices.

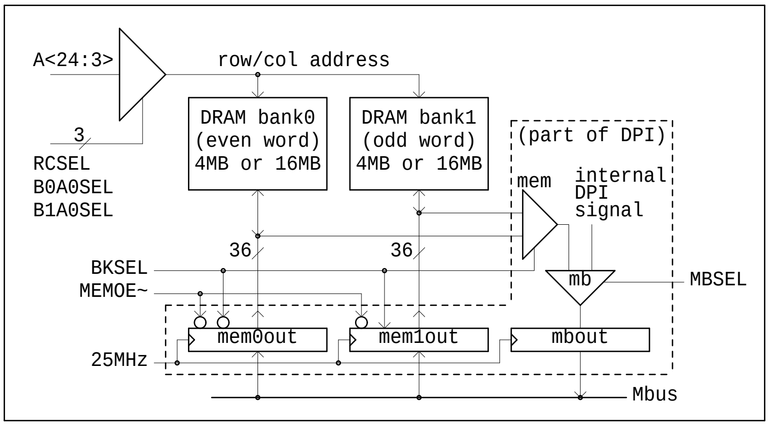


Figure 3-1: Main Memory Banks and Interface to Mbus

Each memory bank is built out of four 4Mx9 fast page-mode DRAM memory modules. The extra bit per byte is meant for parity[6]. There is one pipeline stage between the Mbus and the memory data ports. Data read from the memory banks goes through two muxes *mem* and *mb*, controlled by BKSEL and MBSEL. BKSEL is automatically generated by the DRAM control logic to select the output of the appropriate bank during read operations. During writes, it is the clock-enable input for the pipeline register feeding bank1, and inverted clock-enable for the bank0 input pipeline register. The MBSEL signal is generated by the microsequencer to select either memory output data or another datapath internal to the DPI (see Section 3.8). Unless the microsequencer is actively driving the Mbus from this datapath, MBSEL must by default select the memory data, so that processor accesses to local memory can complete properly.

The memory controller supports single word accesses, and 4-word burst accesses aligned on a 4-word boundary. Only the CMMUs use the burst mode; the microsequencer can only make single word accesses. Figures 3-2 and 3-3 illustrate the best-case timing of memory read and write operations, respec-

[6]Even though it is usually preferable to incorporate error-detection and correction circuitry in the memory modules of large-scale systems, the less conservative, but simpler parity-checking scheme was adopted in PLUS for the following reasons: The Mbus specifications include such parity lines, and the CMMUs already generate and check parity. In our prototype system built out of off-the-shelf components, it was far easier and cheaper to make use of this ready-made facility.

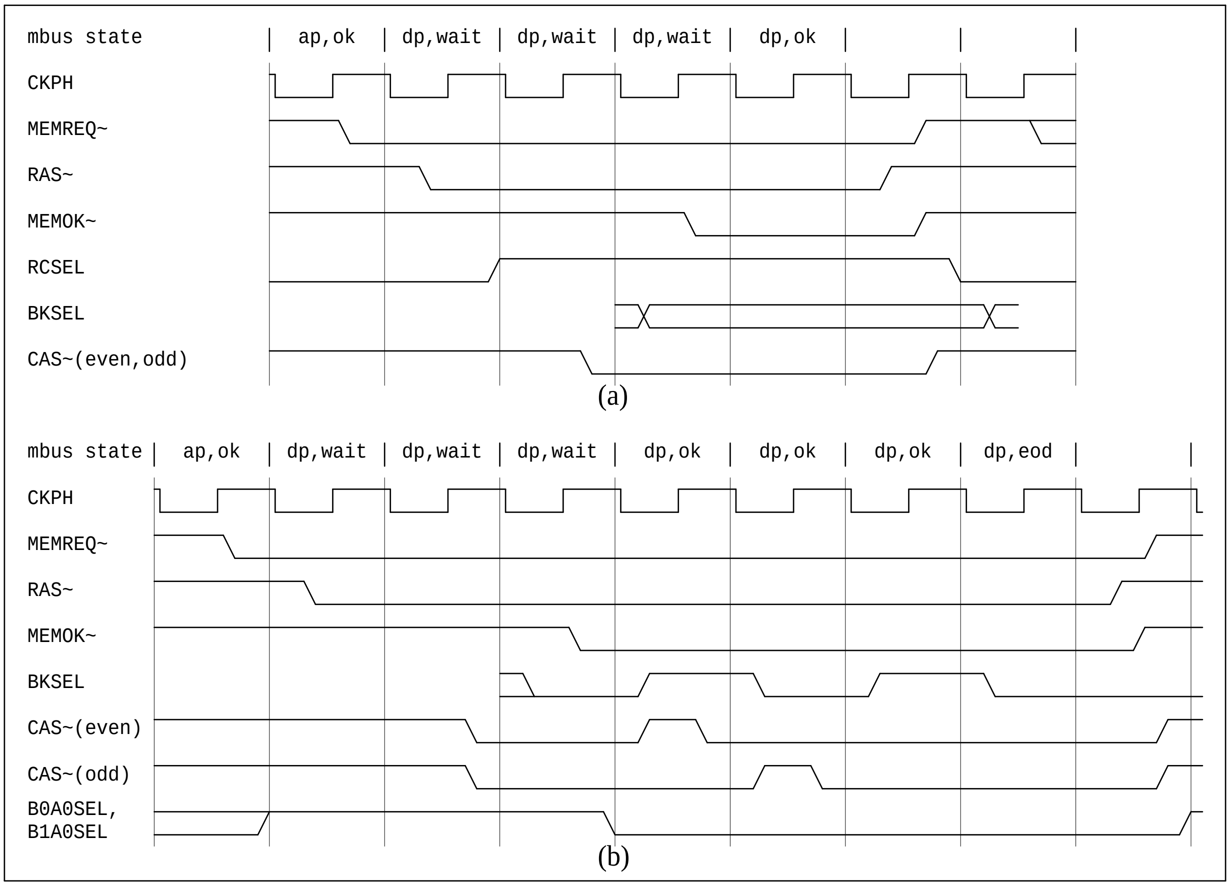


Figure 3-2: Memory Read Timing: (a) Single-word (b) 4-word Burst

tively. The timing diagrams assume DRAM access time to be 80ns, and a 25MHz system clock rate. If the access time is increased to 100ns, the clock rate must be reduced to 20MHz. Some of the signals shown in the figures are described below.

CKPH — Clock phase signal used by the memory CAS generation PALs, which are clocked at twice the clock rate, and must know the phase of the system clock.

MEMOK~ — Active-low signal from the memory controller indicating that it is ready to complete the transaction; the Mbus can be released from wait state in the following cycle.

BKSEL — Bank select mux control, and clock enable for pipeline registers, generated by the memory controller as described above.

RCSEL — Row/column address mux control. This is merely the RAS signal delayed by a cycle.

B0A0SEL — Column address mux control for the least significant bit of the address for bank0, generated by the memory controller. This signal is toggled after the first word of a burst access, thereby changing the column address LSB to a 1.

B1A0SEL — Similar to B0A0SEL, but for bank1.

Memory operations are triggered by the active-low MEMREQ~ signal, which is produced by the address decoding logic described later in Section 4.1.2. A burst access is indicated by the LDT Mbus control bit being a 0 (not shown in the figures) in the first data phase of the Mbus transaction. If it is a read access, there are at least three wait cycles after the address phase before the data is available on the Mbus. During a single-word access, the CAS signal remains asserted until the end of the transaction. This is

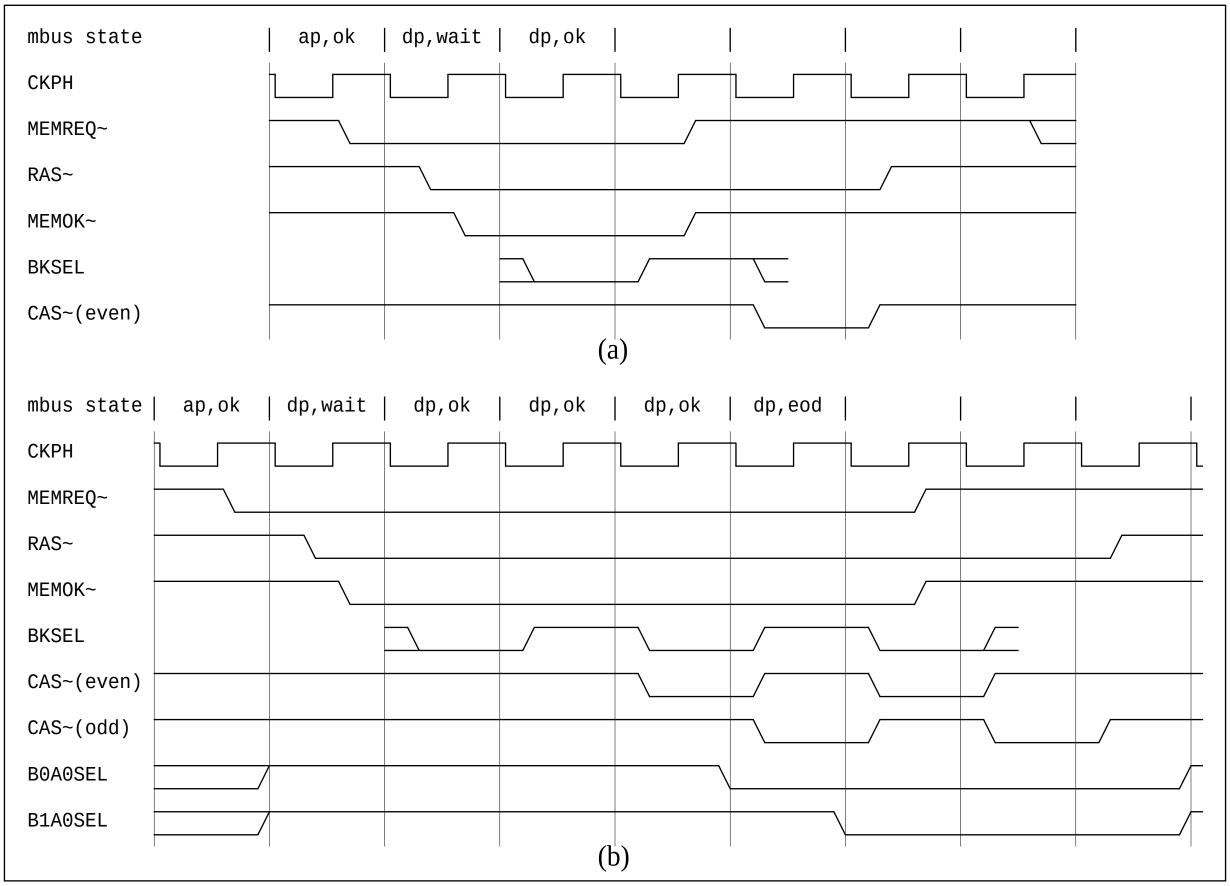


Figure 3-3: Memory Write Timing: (a) Single-word (b) 4-word Burst

useful when the microsequencer accesses memory as Mbus master, as it may need to have the data available for multiple cycles. If a burst access is indicated, the memory system produces one word every cycle after the initial three wait states.

In the case of write operations, the memory controller imposes at least one wait state after the address phase of the Mbus transaction. It will only carry out a write after it sees an *ok* Mbus phase, to make sure that no other state machine wishes to abort or retry the transaction for any reason. The memory requires two cycles beyond the transaction to finish the operation, owing to pipeline delays in the Mbus-to-memory datapath. A new Mbus transaction begun during this interval must wait in address phase until the memory controller is done.

Pages 5 and 6 of the schematic diagrams contain the memory banks, address multiplexers, and the hardwired memory control logic. PAL *DRAMFSM* contains the memory control state machine and produces the RAS signal. *DRAMCTL0* and *DRAMCTL1* control the address muxes, generate the memory write-enable control signal as well as the output-enable signal (MEMOE~) for the pipeline registers (Figure 3-1). They also produce several intermediate signals needed for CAS generation. The CAS signals for the two banks are obtained from PALs *DRAMCAS0* and *DRAMCAS1*, which are clocked at twice the system clock rate in order to meet the setup time requirements of the pipeline registers in the DPI. Finally, the *DRAMREFR* PAL contains the refresh counter.

3.2 The Translation Table

The translation table is the means by which the global page replication scheme becomes visible to the hardware for maintaining consistency among multiple copies of any given page. The structure of the TT was defined abstractly in Section 2.1, and is described in further detail below.

All physical copies of a given logical page are globally ordered by software in a linear *copy-list*, the first of which is designated the *master-copy*. The TT on each node actually consists of two sub-tables: a *master-copy* and a *next-copy* table, maintained by software. For each physical local main memory page, the master-copy table identifies the physical location of the master page, and the next-copy table identifies the physical location of the copy immediately following the local copy in the copy-list, if any. Each entry also has two status bits: the *M* bit is set if the master copy is on a remote node, and the *C* bit is set if there is another copy in the copy-list following the local one. The status bits are duplicated in both sub-tables. Since a node has upto 32Mbytes of memory or 8K pages, the two tables together require 16K entries. The TT is written or updated by system software whenever the replication configuration is changed. The processor can directly address the TT memory via the TTPG I/O-space region. During normal operation, the control logic can observe the TT status bits for a given local physical address by using the page number field of the local address to index into either sub-table. (*Note:* The *M* and *C* status bits are treated as if they were 0 when the TTEN~ bit in the control and status register is negated. That is, there is assumed to be no replication in this case. TTEN~ is automatically negated on power-up.)

The format of each sub-table entry, which occupies one word in the TT, is shown in the following figure. Note that bits 7-0 are non-existent in the TT.

31	30 25	24 12	11 9	8
C	node ID	page address on remote node	unused	M

M = "master remote" bit, C = "next copy valid" bit.

When the *M* bit of an entry is 0, the master copy is local and there is really no need to keep its physical address in the master-copy sub-table; the Mbus address already points to it. Instead, system software should duplicate the next-copy entry in the master-copy table. Organizing the TT in this manner allows any Mbus transaction or input packet to be processed with just one table lookup instead of two. If the microsequencer comes across a local memory write or extended operation (from either the local processor or an input packet), it consults the master-copy sub-table entry for the addressed location. If *M*=1, it forwards the operation to the master copy; the next-copy entry is irrelevant. If *M*=0 and *C*=1, it needs to carry out the operation locally and send an update packet to the next copy, whose physical address can be obtained from the same master-copy entry. Finally, if the microsequencer is handling an update request packet, it refers to the next-copy entry for the addressed page. If *C*=1, it sends another update message to the following copy.

The following table summarizes the four possible TT entries for each local page. The last entry in the table corresponds to the case in which the local copy is the only copy in the system for the logical page.

M	C	Master-copy entry	Next-copy entry
1	1	master copy address	next copy address
1	0	master copy address	not applicable
0	1	next copy address	next copy address
0	0	not applicable	not applicable

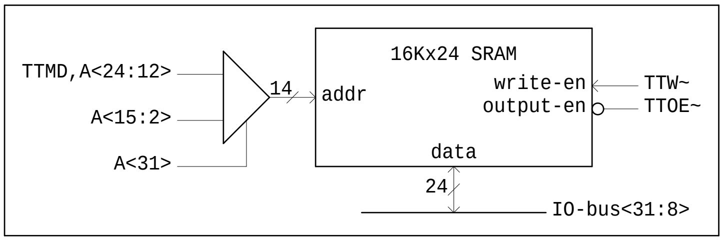


Figure 3-4: The Translation Table

Figure 3-4 shows a block diagram of the translation table and associated logic. The TT address input is selected automatically by the mux shown in the figure. If the MSB of the address latch A<31> is 0, local memory is addressed. So there is a potential need to use the translation table, while it is definitely not addressed as an I/O-space device. Therefore, the mux selects the page no. field of the address latch (A<24:12>) to index into the TT. At the same time, the TTMD microinstruction bit picks the desired sub-table; it must be 0 for the master-copy and 1 for the next-copy sub-table. If A<31> is 1, on the other hand, the Mbus address is not a local memory address, so there is no need to use the TT for address translation. However, it might be addressed as an I/O-space device. Therefore, the mux selects the low-order 14 bits A<15:2> for addressing the TT.

The TT status bits can only be observed by driving them out on the IO-bus. They have to be monitored by hardwired control logic to detect write operations by the processor to local, replicated memory, even when the microsequencer is inactive (see the discussion in Section 4.1.2). Hence, the microsequencer must by default enable the translation table output data onto the IO-bus. The TT output-enable signal TTOE~ is derived from the IOSEL microinstruction field.

When the processor directly writes to the TT as an I/O-space device, the microsequencer is responsible for completing the operation by asserting the TT memory write-enable signal TTW~, which is generated from the UDEVW microinstruction field.

The TT memory and its address mux are contained on page 10 of the schematic drawings.

3.3 Write Tag Buffer

Whenever the processor on a node writes or issues an extended operation on a remote or replicated location, its microcoded control logic starts the process of packets being sent around to all the remote copies to update the location, while the processor continues with instruction execution. In doing so, the microsequencer captures the physical address of that operation in the WTB until it is completed and acknowledged, according to the coherency protocol described in Section 2.1. This has two functions: If the processor reads a location to which an earlier write is still in progress, the WTB detects this and causes the read operation to be suspended. The second function is in the implementation of the *fence* operation. After the processor executes a *fence*, any write or extended operation is suspended until the WTB is empty. The motivation behind these two actions was discussed in Section 2.1. This section contains a detailed description of the design and use of the WTB.

Figure 3-5 shows a block diagram of the WTB. This implementation of the WTB has 8 entries, each containing a 30-bit address register and comparator. Each entry also has a *busy* status bit. The 8 *busy* bits are priority-encoded to produce WTAG, a 3-bit tag denoting the next free entry that can be allocated. If all entries are busy the active-high WTBFULL signal gets asserted, and if all of them are free the active-high

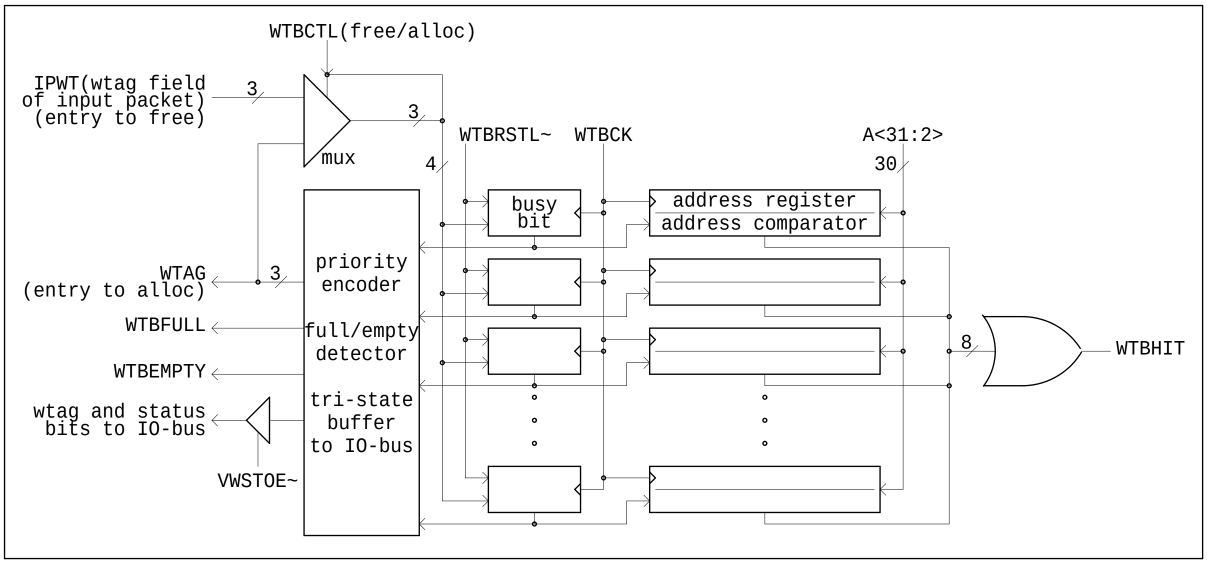


Figure 3-5: The Write Tag Buffer

WTBEMPTY signal is asserted. The status bits are automatically cleared upon power-up.

Whenever the processor executes a write or extended operation to a page that is remote or has a remote copy, the microsequencer enters the Mbus transaction address A<31:2> (the least 2 significant bits are irrelevant as the Mbus addresses are word addresses) in the next free WTB entry indicated by WTAG, and invokes the memory consistency protocol described in Section 2.1. Basically, appropriate request packets are sent to the remote copies, with the allocated WTAG being included in all packets. Eventually, WTAG is returned in an *ack* packet (shown as the IPWT signal in the figure) to the node that originated the operation, at which point the microsequencer frees the WTB entry by clearing the corresponding *busy* bit. Note that if the WTB is full when the processor executes a write or extended operation, it is automatically suspended by hardwired control logic (see Section 4.1.4).

The WTB helps implement write-read consistency as follows. Whenever a CMMU starts an Mbus read operation, the transaction address in the address latch is broadcast to the busy WTB entries, and compared with the contents of the address registers there. If any of them matches, the active-high WTBHIT signal is asserted and hardwired control logic once again suspends the read operation (see Section 4.1.4). Normally, the address comparison is carried out on all 30 bits. However, if the addressed location is in local memory (A<31>=0), the CMMU could potentially be reading an entire cache-line of four words from memory. It is possible that the addressed location is not in the WTB, however one of the remaining three words is. To prevent the CMMU from reading any of them behind the back of the WTB, the address comparison is restricted to the top 28 bits when A<31> is 0.

The fence operation is implemented as follows. When the processor writes to the FENCEPG I/O-space page, the microsequencer causes the FENCE~ bit in the control and status register to be asserted if the WTBEMPTY signal is false. As long as FENCE~ remains asserted any further write or extended operation by the processor is blocked by control logic (see Section 4.1.4). Whenever the node receives an *ack* packet, the microsequencer not only clears the WTB *busy* bit pointed to by the packet, but also negates the FENCE~ bit if the WTB has become empty (i.e., WTBEMPTY is true). At this point the fence operation is complete and the processor can continue. See Section 5.1.7 for a description of how the microsequencer controls the FENCE~ bit.

The WTBEMPTY signal is used a little differently when the STRORD~ or strong-ordering mode bit in the CSR is on (Section 3.6.3). In this case, writes and extended operations to memory are blocked if the WTB is not empty, i.e., if an earlier such operation is still in progress.

The following list describes control signals for the WTB, most of which are derived from microinstruction fields (see Section 5.1.3). See Appendix I for timing requirements pertaining to the WTB.

WTBCK	The WTB is clocked on a low-to-high transition on this input. The UWTBCK microinstruction bit is inverted to generate this clock signal.
WTBCTL	Allocate/free control input: 0 to allocate an entry, and 1 to free it. It also controls the mux shown in Figure 3-5 as follows: 0 selects WTAG and 1 selects IPWT. Clocking the WTB with WTBCTL=0 causes A<31:2> to be loaded into all the free address registers and the *busy* bit of the entry indicated by WTAG to be set. This disables the clock input to the corresponding address register and enables its comparator until the entry is freed (i.e., the *busy* bit is cleared). Clocking the WTB with WTBCTL=1 causes the *busy* bit of the entry pointed to by IPWT to be cleared.
VWRSTL~	Reset input for the WTB, generated from the UDEVW microinstruction field. It clears all *busy* bits in the WTB asynchronously; i.e., the WTB need not be clocked. The status bits are also automatically cleared upon power-up, and whenever the system is reset by an external host.
VWSTOE~	Signal to enable the busy status bits and WTAG onto the IO-bus. It is generated by decoding the IOSEL microinstruction field. Section 3.4 defines the layout of the WTB status bits.

The WTB is implemented in two Xilinx XC3030 devices in 84-pin packages. They are shown as *WTB0* and *WTB1* on page 8 of the schematic diagrams. Each device has approximately half of each address register and comparator. The control logic is also distributed among the two devices. The Xilinx devices have to be configured upon power-up from an external host (see Section 6.2.2).

3.4 Verify Logic

The concept of *verify* operations was introduced in Section 2.2 in association with that of extended operations, and is briefly reviewed below. The extended operations in PLUS return a 32-bit result asynchronously; i.e, some unspecified time after the operation is executed. Each node can have upto 8 such operations pending at any time. The extended operation itself returns a 3-bit verify *tag*; the issuing processor subsequently uses this tag as a word address into the VFYPG I/O-space region to retrieve the result. This is called the *verify* operation. If the result is still not available at this time, the verify operation is suspended by the microcoded control logic. When the result does become available, it is held in one of registers 0-7 of the 16-word ALU register file, until the processor successfully verifies it. The verify tags are simply the addresses of these 8 ALU registers, also called *verify registers* or *verify entries*. This section describes some of the control logic needed to maintain the status of pending extended operations and to implement the verify operation.

Figure 3-6 shows the control logic for maintaining the status of pending extended operations. For each verify register in the ALU, the control logic has corresponding *busy* and *pending* status bits. *Busy*=1 indicates that the register has been allocated for the result of an extended operation, and *pending*=1 indicates that the result has not yet been received. The *pending* bit is irrelevant when *busy*=0. The 8 *busy* bits are priority encoded to produce VTAG, which is the next free entry that can be allocated. If all entries are busy, the active-high VFYFULL signal gets asserted. The status bits are cleared upon power-up.

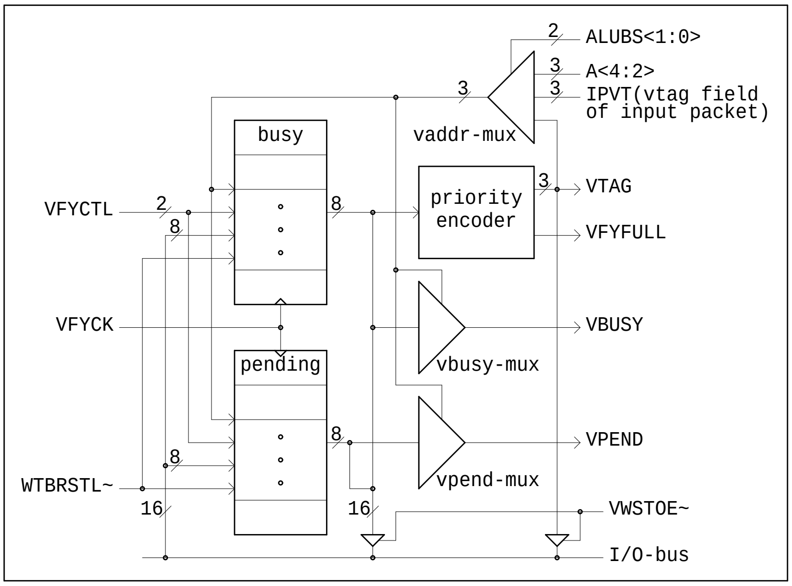


Figure 3-6: The Verify Control Logic

When the processor issues an *xmem* instruction to invoke an extended operation, the microsequencer returns the current VTAG to the processor during the read transaction for that instruction. (Actually, not just the tag, but the entire VFY and WTB status word is returned. This word is described later in this section.) If the master copy for the addressed location is on the same node, the microsequencer takes over the Mbus after the *xmem* instruction is complete, carries out the extended operation, and places the result in the ALU register indicated by VTAG. It also sets the *busy* bit and clears the *pending* bit for that entry. If, on the other hand, the master copy is remote or there is no local copy at all, the microsequencer sends an extended operation request packet, together with VTAG, to the appropriate remote node and sets both *busy* and *pending* bits for the entry indicated by VTAG. Eventually, the operation is executed on the remote node and the tag is returned along with the 32-bit result in a *verify-response* packet. The tag is contained in the *vtag* field of the packet, shown as IPVT in Figure 3-6. The microsequencer places the result in the ALU register indicated by IPVT and clears the *pending* bit for that entry.

Note that there must be a free verify entry when the *xmem* instruction is executed. Otherwise the microsequencer aborts the read Mbus transaction for that instruction. It is not sufficient to simply suspend the transaction until an entry is available, since the only way an entry can become free is for the processor to proceed and verify its result.

When the processor executes a verify operation by reading the VFYPG I/O-space region, the address latch bits A<4:2> contain the tag indicating the verify entry of interest. The microsequencer uses these to select and test the status bits for that entry, shown as VBUSY and VPEND in Figure 3-6. If the result is still pending, it effectively suspends the Mbus transaction by forcing it to *retry*[7]. Otherwise, it retrieves the result from the ALU and returns it to the processor, and frees the verify entry by clearing the concerned *busy* bit.

---

[7] Suspending Mbus transactions by forcing a *retry*, rather than via wait states, allows the Mbus to be freed for use by the microsequencer to process input packets.

The verify registers in the ALU and the VFY status bits are part of the process context that must be saved and restored during a context-switch. For this purpose, the ALU registers can be read or written by system software via the ALUPG I/O-space region. The status bits can be addressed as a word in the VWSTPG I/O-space region; they are read or written from the IO-bus in the format shown in Figure 3-7. The status word can also be read by a user program to determine which of its extended operations have their results available. Notice that in addition to the VFY status bits, the status word also consists of other WTB-related signals. The latter are included purely for debugging purposes, and cannot be overwritten.

<table>
<tr><td>Bit 31</td><td colspan="4">VFY busy status bits 7:0</td><td></td></tr>
<tr><td></td><td colspan="4">VFY pending status bits 7:0</td><td></td></tr>
<tr><td></td><td colspan="4">WTB busy status bits 7:0</td><td></td></tr>
<tr><td></td><td>wtbfull</td><td>wtag<2:0></td><td>vfyfull</td><td>vtag<2:0></td><td>Bit 0</td></tr>
</table>

Figure 3-7: Verify Logic and WTB Status Word

The following paragraphs describe the signals used to control the VFY logic block. These are basically derived from various microinstruction fields (see Section 5.1.3). See Appendix I for timing requirements.

VFYCK — A low-to-high transition on this input clocks the VFY logic. The UVFYCK microinstruction bit is inverted to generate this clock signal.

ALUBS — This 2-bit field controls the *vaddr* mux shown in Figure 3-6. If ALUBS<1>=0, the mux selects VTAG. Otherwise, it chooses A<4:2> or IPVT, depending on whether ALUBS<0> is 0 or 1, respectively. The mux output identifies the *busy* and *pending* bit pair that can be set or cleared through VFYCTL (see below). It also controls two other muxes that produce the values of the chosen status bits, VBUSY and VPEND, which can be tested by the microsequencer.

See Section 3.5 to see how ALUBS also simultaneously controls the ALU B-address mux in the execution of extended operations and verify operations.

VFYCTL — This 2-bit field affects the VFY status bits selected by the output of the *vaddr* mux as follows. The action becomes effective on the next rising edge of the VFYCK clock signal.

- 00: allocates an entry, i.e., sets both *busy* and *pending* bits,
- 01: fills a verify register by setting *busy* and clearing *pending*,
- 10: frees an entry by clearing the *busy* bit, and
- 11: writes or restores all status bits from the IO-bus; the *vaddr* mux output is ignored.

VWRSTL~ — Clears all VFY status bits asynchronously, independent of VFYCK. (This is the same input, generated from the UDEVW microinstruction field, that clears the WTB status bits.) The VFY status bits are also automatically cleared on power-up, and whenever the system is reset by an external host.

VWSTOE~ — Drives the VFY and WTB status word onto the IO-bus. This signal is decoded from the IOSEL microinstruction field. It is used when the processor reads the VWSTPG I/O-space page, or to return the verify tag to the processor when it invokes an extended operation.

The verify status bits are implemented in the same Xilinx devices as the WTB; the *pending* bits in *WTB0* and the *busy* bits in *WTB1*. The priority encoder and *vaddr* mux are implemented in discrete devices outside the Xilinx; the latter in the *VFY1* PAL. See page 8 of the schematic diagrams.

3.5 The ALU

The 32-bit ALU mainly serves the following three functions: First, registers 0-7 of its 16-word register-file are used as temporary storage for the results of extended operations until they can be verified by the processor, as described in Sections 2.2 and 3.4. Second, the microsequencer uses the remaining registers to implement most of PLUS's extended operations. Finally, it can be used by the microsequencer to carry out page-copy operations between nodes.

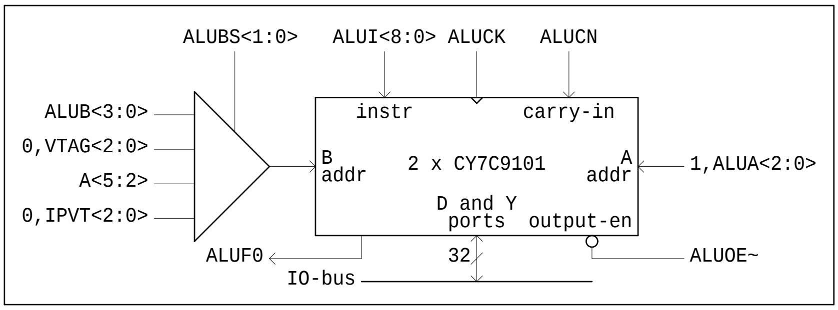


Figure 3-8: The ALU

Figure 3-8 shows a block diagram of the ALU and its peripheral circuitry. The ALU is built out of two 16-bit Cypress Semiconductor CY7C9101 bit-slice devices. The data input and output ports of these devices, D and Y, are both tied to the IO-bus. The ALU is controlled directly by microinstruction bits as described below. For complete details regarding how the device operates, consult the Cypress Semiconductor data book. Also see Appendix I for some timing information.

ALUA, ALUB — ALU register file A and B-port addresses. The microsequencer only specifies 3 of the 4 A-port address bits; its MSB is wired to 1 since the lower half of the register-file is intended for verify results, and is never manipulated by the microsequencer. The B-port address input, ALUB, first goes through a 4:1 mux as shown.

ALUBS — This 2-bit field controls the ALU B-port address mux as follows. See Section 3.4 for some related details.

- 00: Selects the ALUB microinstruction field.
- 01: Selects the 3-bit VTAG from VFY logic; the address MSB is set to 0. This option is to be used in case an extended operation invoked by the processor is executed on the same node (i.e., it has the master copy), and the result has to be placed in the ALU register pointed to by VTAG.
- 10: Selects A<5:2> from the Mbus address latch. This option is used when the processor reads either the VFYPG or the ALUPG I/O-space region. In the former case, A<4:2> contains the verify tag and A<5> must be 0.
- 11: Selects IPVT, the 3-bit *vtag* field of an input packet; the address MSB is set to 0. This option is employed when the node receives a *verify-response* packet from another node in response to an extended operation request, and the result contained in the packet is to be stored in the ALU register pointed to by IPVT.

ALUI, ALUCN — The ALU instruction and carry input bits, obtained from the microsequencer.

ALUOE~ — ALU Y-port output enable signal, decoded from the IOSEL microinstruction field.

ALUCK — Clock input for the ALU from the microsequencer. Its rising edge completes an instruction; it should be held high if the ALU is idle.

The ALU control bits from the microsequencer can also be used for other purposes, as described later in

Section 5. This is usually signalled by the active-low ALUMD~ microinstruction bit being negated. At such times, ALUCK must be held inactive in the high state.

The ALU produces two status bits: the active-high ALUF0 signal which is asserted when the result of an ALU operation is 0, and the sign bit (i.e., MSB) of the ALU output on the IO-bus.

See page 9 of the schematic drawings for complete details of the ALU and its B-port address mux.

3.6 Control and Status Register

The CSR is a single word that primarily consists of interrupt request bits, various error status bits, the node ID, and miscellaneous other control signals. It can be read as a single word through the CSRPG I/O-space region (Section 2.3.2). However, it is not written as a word. Rather, individual bits are controlled by various means, some by writing to the CSRPG region, but many are set by hardwired logic because of some event occurring on the node. In many cases, a bit is asserted by one agent and negated by another. For example, the FENCE~ bit is set when the processor writes to the FENCEPG I/O-space region, but cleared by the microsequencer when the WTB becomes empty. This section contains a description of the function of each bit or field in the CSR. First, we present the format of the CSR word, and then describe individual fields in succeeding paragraphs.

undefined	XID<2>	XID<1>	XID<0>	YID<2>	YID<1>	YID<0>	NODE0~
undefined	*undefined*	FENCE~	TTEN~	PGCPRQ~	TOERRST~	TRERRST~	UERRST~
STRORD~	*undefined*	XFFINT~	RFFINT~	CLKINT~	EXTINT~	UINT~	RMTINT~
undefined	*undefined*	*undefined*	*undefined*	*undefined*	*undefined*	*undefined*	*undefined*

Figure 3-9: The Control and Status Register

The control and status register is read via the CSRPG I/O-space region. For this purpose, the CSR can be enabled onto the IO-bus as a single word when the active-low CSOE~ signal, obtained by decoding the IOSEL microinstruction field, is asserted. The layout of the CSR word when it is read over the IO-bus is shown in Figure 3-9, bit 31 first, and bit 0 last.

The individual CSR bits, and the ways in which they are controlled, are described in the following sections. We should note here that when a CSR bit is controlled by writing to the CSRPG I/O-space region, the microsequencer is responsible for carrying out the operation. However, it does not have to control each such bit individually. Instead, it merely decodes the write operation and asserts the UCSW microinstruction signal, and hardwired control logic does the rest according to the description below.

Finally, it should also be mentioned that all CSR bits, except XID<2:0>, YID<2:0>, and NODE0~, are automatically negated upon power-up, and whenever the node is reset by an external host.

3.6.1 Interrupts

There are six signals on each node that can interrupt the MC88100 processor on that node. These interrupt request bits are logically OR-ed to generate the processor interrupt input signal. The processor should then read the CSR to determine which requests are asserted. It can mask interrupts off all at once, if necessary, but there is no way to mask individual requests selectively. Furthermore, if the same interrupt is raised more than once before the processor recognizes it, the multiple occurrences get merged into one request; i.e., they are not queued in hardware. The six interrupt request bits are described below.

RMTINT~ Active-low interrupt to the processor from a remote node. The microsequencer asserts this bit (using the UBITOP microinstruction field, described in Section 5.1.7) in response to a *processor-processor interrupt* packet received from a remote node (see RMTPINTPG I/O-space region in Section 2.3.2). RMTINT~ is normally negated by the processor, by writing to word 0 in the CSRPG I/O-space region; the value written is irrelevant.

EXTINT~ Active-low interrupt to the processor from an external host attached to the local or remote node. The local host can directly assert this bit by writing a 1 to word 2 in the CSRPG I/O-space region. Alternatively, a remote host can send an *EXT-processor interrupt* packet to the node by writing to the RMTPINTPG I/O-space region (Section 2.3.2). Upon receiving the request, the microsequencer asserts EXTINT~ as described above. EXTINT~ is normally negated by the processor by writing a 0 to word 2 in the CSRPG region.

UINT~ Active-low interrupt to the processor from the microsequencer, generated by the UBITOP microinstruction field. This interrupt is invoked to indicate various events; the microsequencer uses ALU register 11 to indicate the reason for the interrupt, as described below. UINT~ is negated by writing to word 1 in the CSRPG I/O-space region; the value written is immaterial.

Figure 3-10 shows the various bits of ALU register 11 used to convey the reason for the interrupt to the processor. Each bit is active-high. The register must be cleared by software upon reset and after such an interrupt.

CLKINT~ Active-low periodic clock interrupt to the processor, asserted every 2^{18} cycles (about 10msec). The interrupt request is generated with the help of a 14-bit counter in the DPI Xilinx devices (see Section 3.8.6.6). It is negated by writing to word 3 in the CSRPG I/O-space region; the value written is irrelevant.

RFFINT~ Active-low interrupt to the processor, asserted when the NI receive FIFO is full. It is negated by writing to word 4 in the CSRPG region; the value written is ignored.

XFFINT~ Active-low interrupt to the processor, asserted when the NI transmit FIFO is full. It is negated by writing to word 5 in the CSRPG I/O-space region; the value written is ignored.

The last two interrupts, RFFINT~ and XFFINT~ warn the processor of congestion or overload in the network. It is upto software to deal with this situation, if at all.

The IRQEXT~ CSR bit is a single interrupt request signal on each node that is sent to the external host attached to that node. Both the local and remote processors can cause this request to be asserted. The local processor can directly write to the CSR bit, as described below, while remote processors send a *processor-EXT interrupt* packet by writing to the RMTEXTINTPG I/O-space region (Section 2.3.2). The microsequencer, upon receiving the packet, asserts the request as described below:

IRQEXT~ Active-low interrupt request to the local external host. This bit is asserted by writing a 1 to word 15 in the CSRPG I/O-space region, and negated (usually by the host) by writing a 0 to the same address.

The interrupt request signals are implemented in PALs *CSCTL0*, *CSCTL1*, and *CSCTL2*, shown on page 18 of the schematic drawings.

Bit	Description
0-1	Unused.
2	Page copy operation completed.
3	TT does not contain valid next-copy entry for requested page-copy operation.
4	Mbus request for carrying out page-copy operation was unsuccessful.
5	Mbus request for carrying out an extended operation was unsuccessful.
6	Queue operation issued by local processor addresses a replicated page.
7	Queue operation request from a remote node addresses a replicated page.
8	Page-copy or a locally issued extended operation aborted with an error.
9	Page-copy or a locally issued extended operation aborted with a retry.
10	Remote packet transaction aborted with an error.
11	Remote packet transaction aborted with a retry.
12	Verify packet received, but status bits are not <busy,pending>.
13-31	Unused.

Figure 3-10: ALU Register 11 Status Bits Associated with Microsequencer Interrupt UINT~

3.6.2 Mbus Error Status

Whenever an Mbus transaction is aborted with an error by hardwired or microcoded control logic, the event is recorded in status bits of the CSR, which can be read by software to get some idea of the nature of the error. (If a CMMU aborts a transaction in this manner, one of its internal registers records the event.) Note that more than one such bit may be asserted at the same time. The Mbus error status bits in the CSR are the following:

UERRST~
: Active-low status bit asserted when the microsequencer aborts an Mbus transaction with an error. The reason for the interrupt is indicated in ALU register 11, as shown in Figure 3-10. UERRST~ is negated by writing to word 8 in the CSRPG I/O-space region; the value written is immaterial.

TRERRST~
: Active-low status bit asserted when any Mbus transaction is aborted by hardwired logic for the following reasons: an illegal local memory address, a CMMU attempting a burst access to a remote page, or a CMMU attempting a burst write to a local, replicated memory page. The last two are intended to catch errors in system software that enforces caching policies. One should never see a burst write to a replicated page, as the page cannot be cached in write-back mode. Nor should there ever be a burst access to a remote page as such a page cannot be cached at all. (See the discussion at the end of Section 2.1.) TRERRST~ is negated by writing to word 9 in the CSRPG I/O-space region; the value written is irrelevant.

TOERRST~
: Active-low status bit asserted when any Mbus transaction is aborted by the control logic for not completing within the timeout period (about 10-20 msec). It is negated by writing to word 10 in the CSRPG I/O-space region; the value written is immaterial.

See Sections 4.1.2 and 4.1.5 for a description of the control logic that causes the last two events.

The Mbus error status signals are implemented in PAL *CSCTL2*, shown on page 18 of the schematic drawings.

3.6.3 Miscellaneous Control

A number of *mode* bits in the CSR on each node control the operation of that node. These include bits to indicate whether a *fence* operation is in progress, whether the TT is enabled, and whether the node should run in *strong-ordering* mode, as described below:

FENCE~ This active-low bit is used to indicate that a *fence* operation is in progress. It is asserted by the microsequencer, using the UBITOP microinstruction field, when the processor writes to the FENCEPG I/O-space region (Section 2.3.2) and if the WTB is not empty at that time. As long as FENCE~ remains asserted, write and extended operations by the processor are blocked. Whenever the microsequencer receives an *ack* packet and clears a WTB entry, it negates this bit if the WTB has become empty.

Note that only those accesses by the processor that appear on the Mbus are blocked when FENCE~ is asserted. If a write operation hits a cache entry (cached in write-back mode) there is no way to block it, and it will complete even if FENCE~ is true. But this is good enough for our purposes as shared memory locations are not supposed to be cached in write-back mode, anyway.

TTEN~ Active-low translation table enable bit, controlled by writing to word 12 in the CSRPG I/O-space region. Writing a 1 asserts the bit and a 0 negates it. When TTEN~ is negated, the Mbus control logic and microsequencer ignore the TT status bits; i.e., they behave as if no local page is replicated on any other node.

STRORD~ Active-low *strong-ordering* mode bit. When it is asserted, memory operations by the processor that appear on the Mbus are strongly ordered. That is, any write or extended operation by the processor to global memory will only proceed after any previous such operation has completed. This bit is asserted by writing a 1 to word 7 in the CSRPG I/O-space region, and negated by writing a 0 to that address.

STRORD~ should be turned on when running programs that rely on a strongly ordered memory model. Note, however, that the model only applies to processor transactions that appear on the Mbus. If a write operation hits a cache entry (cached in write-back mode) there is no way to block it, and it will complete even if an earlier memory operation is still in progress. Still, this model is sufficient for our purposes since shared memory locations are not supposed to be cached in write-back mode, anyway.

PGCPRQ~ This active-low bit is asserted by the processor to request the microsequencer to create a replicated copy of a local physical page on another remote node. The physical address of the local page to be copied should have been written to ALU register 12 (see Section 7.4). The processor writes a 1 to word 11 in the CSRPG I/O-space region to assert this bit. When the microsequencer is done with the page copy operation, it negates PGCPRQ~ using the UBITOP microinstruction field (see Section 5.1.7).

All the above control bits are implemented in PAL *CSCTL0*, shown on page 18 of the schematic drawings.

3.6.4 ID Register

The 6-bit ID register in the CSR contains the node's absolute *x*- and *y*-coordinates in the mesh structure of the PLUS interconnection network. Each co-ordinate is 3-bits wide. The ID register is used by the NI to translate the absolute destination address field of packets in the OPR into relative destination addresses, as required by the MRC (see Section 2.4). The ID register on node <0,0> should be permanently cleared by installing jumper *J7*. Those on all other nodes have to be initialized by system software as described below.

XID, YID *x*- and *y*-coordinates of the node's ID, the former is the more significant part. This register is initialized by the processor by writing to the IDPG I/O-space region (Section 2.3.2). The microsequencer is responsible for carrying out the write operation by asserting the IDW~ signal, which is generated from the UDEVW microinstruction field.

NODE0~ This bit really signifies whether jumper *J7* has been installed on the node or not. The jumper must be installed on node <0,0> and this permanently clears its CSR ID register. NODE0~ is 0 if the jumper is present, and 1 otherwise.

The ID register is shown as *IDREG* on page 18 of the schematic drawings. Jumper *J7* can also be found on the same page.

3.7 NI Interface

This section defines the interface between the Mbus subsystem and the NI, chiefly consisting of the input and output packet registers, IPR and OPR, and a small number of control and status signals between the two modules. The microsequencer places packets to be transmitted in the OPR, which is then copied into the output FIFO by the NI and finally transmitted. On the receiving side, the NI transfers all incoming packets to the IPR, one at a time, for consumption by the microsequencer.

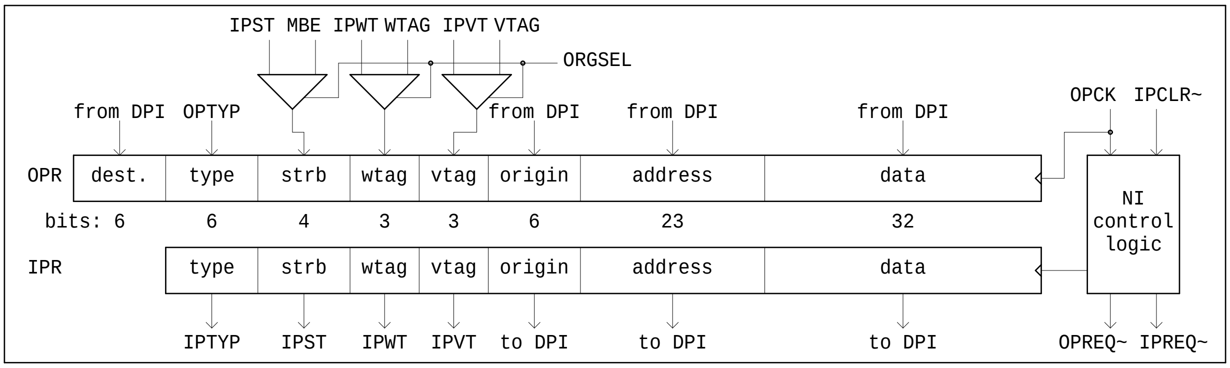


Figure 3-11: Interface Between the Mbus Subsystem and the NI

The IPR and OPR, including the individual packet fields, are shown in Figure 3-11. The figure only shows the interface visible to the microsequencer. What is not shown is that the IPR is actually double-buffered. The NI can assemble an incoming packet in its first stage, while the microsequencer processes an earlier packet in the second stage. When it is done, the microsequencer clears the second stage, and the NI can then transfer the new packet from the first stage to the second in a single clock cycle.

Both OPR and IPR are wide enough to contain the largest packets. Refer to Section 2.4.2 for a description of the individual packet fields. Note that the destination field of the OPR contains a 6-bit absolute address of the destination node. This is eventually converted by the NI into a 16-bit relative address. The control and status signals pertaining to the two registers are described below. See Appendix I for timing-related information.

OPCK OPR clock signal from the microsequencer. Its rising edge clocks the entire register, and when high, it also causes OPREQ~ to be asserted (see below). To avoid inadvertently transmitting the same packet more than once, the microsequencer should hold OPCK high for only one cycle while clocking a packet into the OPR.

OPREQ~ Output packet request. This active-low signal is asserted by the NI when it sees that OPCK is high, and negated after it has copied the packet in OPR into the transmit

FIFO. Before entering a new packet into the OPR, the microsequencer should test this bit to ensure that the register is empty.

IPREQ~ Input packet request. This active-low signal is asserted by the NI when it transfers a packet into the second stage of the IPR for the microsequencer to process it. IPREQ~ is eventually cleared by the microsequencer.

IPCLR~ Active-low signal to clear the IPR, derived from the UIPCLR microinstruction bit. This allows the NI to load the next packet into the IPR.

Many of the IPR and OPR input and output signals shown in the figure are generated and consumed within the DPI, and are explained in Sections 3.8.4 and 3.8.5. The remaining signals are described below:

OPTYP Output packet type field input from the microsequencer.

WTAG, VTAG The tags produced by the WTB and VFY logic blocks.

MBE Mbus byte-enable strobe signals, latched in a temporary register, and fed to the OPR.

ORGSEL This microinstruction bit controls whether the *origin*, *strb*, *wtag*, and *vtag* fields of the OPR are fed from the local node, or forwarded from the corresponding fields of the IPR. Whenever the microsequencer forwards a received packet to another node (for example, when it receives a write request for a page for which the node does not have the master copy), or sends an update message in response to a request packet, or sends a reply to the originating node, these fields are forwarded from the IPR. Only when the node sends an initial request message should they be generated locally. ORGSEL must be 1 to forward the above fields of the IPR to the OPR.

IPTYP Input packet type field. It can be tested by the microsequencer and other control logic.

IPST Byte-enable bits indicating which bytes of the IPR data field contain relevant information. This is how a node can write individual bytes on a remote node. When the microsequencer writes to memory in response to a write or update request packet, it generates the Mbus byte strobe control signals MC<3:6> from the IPR IPST signals. Also see Section 5.3.1.

IPWT If the received packet is an *ack* packet, this field identifies a WTB entry to be freed.

IPVT If the received packet is a *verify-response* packet, this field is used to address the ALU register in which the result is to be stored.

The *destination*, *origin*, *address*, and *data* fields of the IPR and OPR are implemented in the DPI Xilinx devices, shown on page 7 of the schematic drawings. This includes various muxes that produce the appropriate inputs for the OPR fields. The remaining OPR fields, including their input muxes, are implemented in three PAL devices *OPCTL0-OPCTL2*, while the corresponding IPR fields are implemented in two multi-stage pipeline registers *IPCTL0-IPCTL1*, shown on pages 11 and 12 of the schematic drawings.

3.8 Datapath Interconnect

The DPI is the last major functional block in the Mbus subsystem. It interconnects functional units residing on the five major buses: the Mbus, memory bank0 and bank1 data buses, the NI interface F-bus, and the IO-bus. The DPI pins, where the five buses terminate, are called *ports*. Internally, the DPI contains several pipeline registers, multiplexers, tristate drivers and buses to route data between the five ports. In addition, there are functional units to perform various housekeeping functions, such as a timer to time out hung Mbus transactions, and a 32-bit free running counter for performance evaluation purposes.

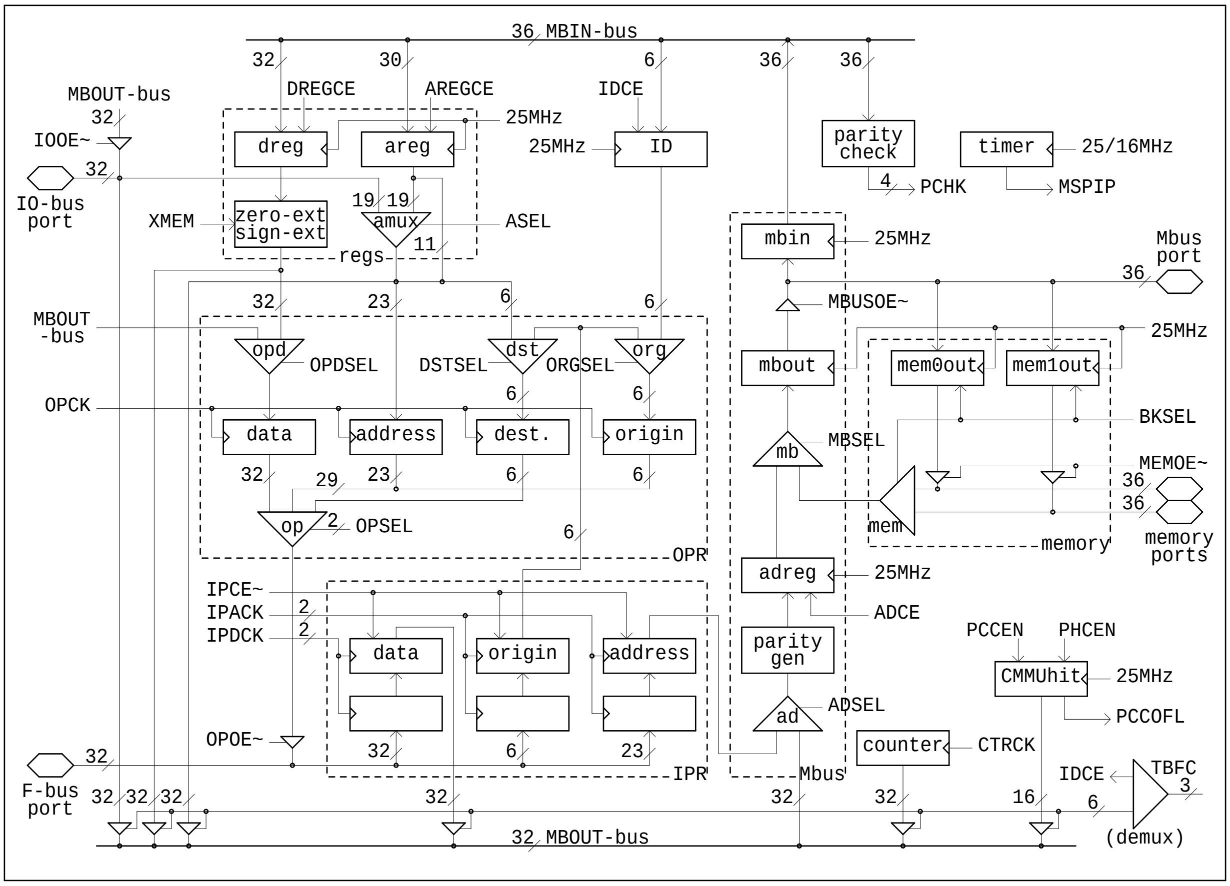


Figure 3-12: The Datapath Interconnect

Figure 3-12 shows a detailed picture of the internal architecture of the DPI. There are two main buses inside the DPI: MBIN-bus is fed from the Mbus port, and distributes Mbus address and data information to various destinations. MBOUT-bus can be driven by a number of sources in the DPI as well as the IO-bus port, and from there eventually reaches the Mbus through a number of pipeline registers and multiplexers. MBOUT-bus can also be driven out onto the external IO-bus; hence, it can be thought of as an extension of the IO-bus. All-in-all, data can be exchanged easily between devices on the Mbus, the DPI MBOUT-bus, and the IO-bus.

In addition to the two internal buses in the DPI, there are several dedicated paths between individual logic blocks. The DPI logic blocks are grouped together according to their function, as shown by the dotted outlines in the figure. For example, two such groups comprise most of the input and output packet registers. The following paragraphs describe each such block in detail. The DPI is controlled by a combination of the microsequencer, the NI, and the memory controller.

The DPI is implemented in four XC3042 Xilinx programmable logic cell arrays, one 8-bit slice per device. They are referred to as *XDP0* (the least significant byte) through *XDP3* (the most significant), and are shown on page 7 of the schematic drawings. They have to be configured upon power-up, together with the WTB and VFY Xilinx devices, from an external host (see Section 6.2.2). For information pertaining to the timing of various datapaths inside the DPI, see Appendix I.

3.8.1 Memory Section

This section, together with part of the Mbus section (see below), comprises a dedicated, fast path between the main memory ports and the Mbus port. It is capable of sustaining burst transfers to or from memory at 25MHz. This part of the DPI has been described in detail in Section 3.1.

3.8.2 Mbus Section

The Mbus section of the DPI essentially transfers address or data information between the Mbus and the rest of the DPI. The pipeline registers shown in this section are required to meet the 25MHz system clock frequency. The pertinent control signals are described below:

ADSEL	Microinstruction bit that controls the *ad* mux. If 0, it selects MBOUT-bus, otherwise the IPR address register is chosen. Although the latter is only 23 bits wide, it is padded with 7 leading and 2 trailing 0's to make it look like a legal local memory word address.
ADCE	Active-high clock enable signal for the pipeline register *adreg*. It is a microinstruction bit.
MBSEL	Microinstruction bit controlling the *mb* mux. When 0, it selects the output of the *mem* mux, otherwise the output of the pipeline register *adreg* is chosen. This bit must default to 0 if the microsequencer is idle, so that data can continue to flow freely between the Mbus and the memory banks.
MBUSOE~	Mbus port output enable signal, generated automatically by control logic. Essentially, the DPI Mbus port output is enabled in any cycle in which the bus has been granted to some master (CMMU or microsequencer), and neither CMMU is driving the bus.

3.8.3 Address and Data Registers

Registers *areg* and *dreg* are normally meant for capturing the address and data, respectively, of an Mbus transaction. They can then be transmitted in an output packet. Or, if they are arguments of an extended operation invoked by the processor, the microsequencer can route them over the MBOUT-bus back to the Mbus or to the IO-bus, for carrying out the operation. The address register is clocked automatically by hardwired logic, but the data register is controlled by the microsequencer (see below).

The microsequencer uses the multiplexer *amux* to select bits 30-12 of either register *areg* or the IO-bus port input, depending on whether the ASEL microinstruction bit is 0 or 1, respectively. The latter is to be used when a local memory address is translated via the TT to produce the physical address of a remote copy of the location, with bits 30-12 of the TT output replacing the corresponding bits from *areg*. The remaining lower-order bits are not affected since they specify the offset within a page, which is the same for all copies of a replicated page.

The *zero-extend/sign-extend* logic block in this section is used for the following purpose. As we saw in Section 2.2, the processor uses the *xmem* instruction to invoke an extended operation, whose variable-length opcode is obtained from the most significant 2-6 bits of the register data argument of the *xmem* instruction. The register argument is written out by the processor over the Mbus, and latched in register *dreg* by the microsequencer, during the write transaction for that instruction. Once the argument is in *dreg*, this logic block can optionally strip off the extended-opcode field and sign-extend or zero-extend the remaining data bits. This function is enabled when the XMEM control signal from the microsequencer is high. Refer to Table 2-1 to see how the arguments are extended.

The control inputs for the address and data registers are summarized below:

AREGCE	Clock enable input for address register *areg*, generated automatically by control logic. Because of the pipeline stage *mbin* between the Mbus and this register, AREGCE is asserted to latch an Mbus address into *areg* one cycle after the address phase is completed (i.e., during the first data phase). Therefore, as long as the Mbus is inactive or in address phase, *areg* holds the address from the previous transaction.
DREGCE	Active-high clock enable input for data register *dreg*. This bit is obtained directly from the microsequencer.
ASEL, XMEM	The functions of these microinstruction bits have been described above.

3.8.4 Output Packet Register

The function and layout of the OPR have been described in Section 3.7. The *destination*, *origin*, *address* and *data* OPR fields are implemented in the DPI. This section describes how the input for each field is generated, and what the relevant control signals are.

The *dst* mux selects a 6-bit destination value from either bit positions 30-25 of the output of multiplexer *amux* or the *origin* field of the IPR. Since the former is itself chosen from two sources, the OPR *destination* field is actually written from one of the three sources listed below:

- Bits 30-25 of register *areg* in the DPI. This choice is appropriate for transmitting a request packet to the node indicated by the node ID field of a remote memory address in *areg*.
- Bits 30-25 of IO-bus port input, for forwarding a request to a node determined by looking up the translation table.
- The IPR *origin* field, when sending any kind of response packet (e.g., *ack* or *read-response* packet) to the node originating a request.

The *dst* mux is controlled by the DSTSEL microinstruction bit. It selects the output of multiplexer *amux* if DSTSEL=0, and the IPR *origin* field otherwise.

The *org* mux chooses the 6-bit OPR *origin* field input from either the node ID register in the DPI or the IPR *origin* field. The former is appropriate when a request packet originates on the local node. The latter should be used while forwarding a request, sending an update request, or returning a response packet. The *org* mux is controlled by the microinstruction ORGSEL bit, and it selects the DPI ID register when ORGSEL=0. Otherwise it selects the IPR *origin* field.

Bit positions 24-12 of the OPR *address* field (i.e., the page no.) are obtained from the output of *amux* multiplexer. This allows either a remote memory page no. from *areg* or the translated page no. of a master-copy or next-copy page from the TT to be written to this field. The remaining address bits 11-2 are copied directly from register *areg*.

The OPR *data* field can be fed from either register *dreg* (after being optionally sign or zero-extended), or the MBOUT-bus. The latter is helpful, for example, in forwarding the IPR data field to an output packet, or in routing data on the IO-bus to the OPR, instead of first having to send it all the way around to *dreg*. The *opd* mux selects register *dreg* if the OPDSEL control signal is 0, and the MBOUT-bus otherwise. This control signal is obtained from the UOPDSEL microinstruction bit (see Section 5.1.5).

The entire OPR is clocked by the OPCK microinstruction bit. Once the OPR has been filled in this way, the NI reads individual packet words via the *op* mux by controlling the OPSEL input, and asserting OPOE~ to enable the DPI F-bus port output drivers. When the MSB of OPSEL=0, the *op* mux selects the 6-bit *destination* field, which appears on bits 30-25 of the DPI F-bus port. (The remaining fields of the control word of the packet are outside the DPI.) When OPSEL=10 the address word is selected, and if OPSEL=11 the data word is read out, according to the packet format shown in Section 2.4.2.

3.8.5 Input Packet Register

The function and layout of the IPR have been described in Section 3.7. The IPR *origin*, *address* and *data* IPR fields are implemented in the DPI as a two-stage register. The NI extracts words of an input packet from the receive FIFO, and assembles them in the first stage of the IPR. When the entire packet has been assembled and the second stage is free, the NI transfers the packet to it for processing by the microcoded control logic. The NI produces the clocks, IPACK and IPDCK, for both stages of the IPR, as well the active-low clock enable signal, IPCE~, for its second stage. The clocks are actually two signals each since the NI has to write the upper and lower halves of each packet word separately. The details can be found in the accompanying document *PLUS Prototype Design Documentation*.

The IPR *origin* field can be forwarded in another request packet or written to the *destination* field of a response packet in the OPR. The IPR *address* and *data* fields are usually routed by the microsequencer to the Mbus port in the execution of Mbus transactions. In the process, the 23-bit address field is zero-extended into a full 32-bit local memory word address. They may also be routed to other locations within the DPI, such as the OPR, or to the IO-bus.

3.8.6 Miscellaneous

Apart from the five major groups of logic blocks described so far, the DPI contains several other functional units and data paths which are briefly outlined below.

3.8.6.1 IO-bus Port

As seen from Figure 3-12, the DPI interconnects the IO-bus and its internal MBOUT-bus via tristate buffers. The active-low output enable signal IOOE~ for the IO-bus port is derived from the IOSEL microinstruction field. Going in the other direction, the IO-bus port input can be driven onto the MBOUT-bus by controlling the TBFC microinstruction field.

3.8.6.2 MBOUT-bus

The MBOUT-bus inside the DPI can be driven by a number of sources, including the IO-bus port input. It is the primary bus for exchanging address and data information between the Mbus port, the IO-bus port, and various registers internal to the DPI. The tristate buffers driving MBOUT-bus are controlled by the TBFC microsequencer field, which is decoded inside the DPI. The bus has internal pullup resistors, so all bus lines go high when they are not driven.

3.8.6.3 ID Register

This 6-bit register contains the local node ID, and its sole purpose is to fill the *origin* field of request packets originating on the node. Outside the DPI, there is another identical ID register which is part of the control and status register. However, there are not enough pins on the DPI to access this external register quickly, hence the need for an internal copy. Both registers should be initialized by the processor after power-up, by writing to the IDPG I/O-space region (see Section 2.3.2); this one in the DPI is written from bit positions 30-25. The microsequencer carries out the write operation by setting TBFC=0, which is decoded within the DPI to generate the ID register clock enable signal, IDCE, in the following cycle.

3.8.6.4 32-bit Globally Synchronous Counter

The DPI on each node contains a 32-bit free-running counter intended for system performance measurement, which can be read via the CTRPG I/O-space region (see Section 2.3.2). It is clocked at 1/16-th the system clock rate, or every 640ns assuming 25MHz system clock. At this rate, the counter wraps around approximately every 45.8 min.

Usually, performance measurement becomes a lot easier, and faster, if the counters on all nodes can be kept synchronized; otherwise we need an entire layer of software protocol to achieve this synchronization. In the PLUS system, the counters on all the nodes are simultaneously cleared to 0 when the DPI Xilinx devices are configured after power-up. Furthermore, even though there is no single, central system clock, it is possible to broadcast the slower-speed counter clock from node <0,0> to the remaining nodes over PLUS's mesh interconnection network, so that the counters can remain synchronized forever afterwards (modulo clock propagation delays of the order of 1us). Appendix III contains a description of how the counter clock is distributed throughout the system.

3.8.6.5 Data Cache Hit-rate Counters

These are two 12-bit counters, implemented in DPI devices *XDP1* and *XDP2*, that monitor the data cache hit-rate. The former counts all data P-bus cycles, while the latter counts only those that result in a data cache hit. The count enable signals for the two counters, PCCEN and PHCEN, are automatically generated by control logic that watches the data CMMU status bits TM<1:0> and TR<0> (see Table 4-6 *Tag Monitor Signal Functions* in the Motorola MC88200 CMMU manual for details). The first counter generates the PCCOFL signal when it overflows.[8] The PAL watches for this event, then disables both counters and asserts the active-low PHCRDY~ signal to the external host to indicate that the counters can be read (see Section 6.3). After the counters are read, they are automatically cleared and counting resumes.

The counters can be read via the CACHEHITPG I/O-space region. They appear as a single word on the MBOUT-bus: the top 8 bits of the P-bus cycles counter on bits 15-8, and the top 8 bits of the cache-hit counter on bits 23-16; the remaining bits in the word are undefined. Both counters are controlled by the *HITRT1* PAL, shown on page 1 of the schematic drawings.

3.8.6.6 Timer

This is a 14-bit counter implemented in the least significant DPI Xilinx device, *XDP0*, and it is used to deliver a periodic interrupt to the processor, as well as to timeout and recover from hung Mbus transactions. It is clocked at 1/16-th the system clock rate, or every 640ns, assuming 25MHz system clock. The 14-bit counter asserts the active-high MSPIP signal every time it overflows. The interval between successive assertions of this signal is, therefore, 2^{14+4} cycles, or about 10msec at the above clock rate. The *TIMEOUT* PAL, shown on page 18 of the schematic drawings, uses MSPIP to generate the periodic interrupt to the processor, as well as to timeout hung Mbus transactions.

---

[8]Owing to speed limitations in the Xilinx device, the counter signals an overflow as soon as its three most significant bits all reach 1, regardless of the lower-order bits. So, they stop counting after 3584 P-bus cycles, rather than 4096 cycles as one might have expected.

3.8.6.7 Parity Generation and Checking

The main memory on each node includes a parity bit per byte for error checking. Whenever the CMMUs read and write memory, they generate and verify parity internally, and signal a fault to the processor if necessary. (The Mbus as well as the DPI datapath between the memory and the Mbus contain parity data lines for this purpose.) The other devices, including data read from remote memory locations, contain no such parity bits. Therefore, to prevent spurious parity errors, one has to generate parity in the DPI if the processor reads such locations. Furthermore, if the microsequencer writes to local memory (e.g, while processing an input packet or an external host request), it should generate and write the proper parity bits as well, in case the location is read by the processor at some later time. For these reasons, the DPI contains parity generation logic in its Mbus section (see Figure 3-12).

If the microsequencer is Mbus master, it can optionally check for parity errors while reading memory. The parity check logic in the DPI produces four PCHK bits, one per byte, which are active-high parity error signals. These are OR-ed externally to generate the active-low parity error signal PRTYERR~, which can be tested by the microsequencer. Note that PRTYERR~ does not itself trigger any kind of exception.

Refer to Appendix I for the timing delays involved in the parity generation and checking logic.

3.8.6.8 IPR Origin Field-ID Register Comparator

This comparator (not shown in the figure) compares the 6-bit IPR *origin* field with the ID register in the Xilinx devices. Whenever the microsequencer receives and processes an update request packet, it uses the output of this comparator to determine whether to invalidate any cached entry in the CMMUs or not. A successful comparison implies that the update packet is for a write or extended operation that originated on this node, and so there is no need to invalidate any cached entry.

4. Mbus Transactions

We have, thus far, looked at a static picture of the various components of the PLUS system, viewing individual functional units of the datapath separately in detail, but without bothering very much about the dynamic operation of a node as a whole. We now concentrate on the latter; specifically, we describe the hardwired and microcoded control logic in the Mbus subsystem. This section deals with the hardwired logic that decodes and responds to all Mbus transactions, while Section 5 describes the design of the microsequencer and its operation.

The Mbus is the center of all activity on the PLUS node. All accesses to the node's main memory or I/O-space regions, whether by the processor or the microsequencer, must take place in the form of Mbus *transactions*. A transaction is a single read or write access. A bus master may execute multiple transactions in a single *tenure*, i.e., after arbitrating for and obtaining the bus, it may make multiple read or write accesses before releasing the Mbus. This is how the microsequencer can, for instance, perform the extended operations atomically. An Mbus tenure may also consist of zero transactions. For example, when a node receives an *ack* packet, the microsequencer obtains bus-mastership, clears the WTB entry indicated by the packet and releases the bus; there is no read or write transaction.

Every Mbus transaction address is latched in the address latch and decoded to identify a particular *slave device*, which is woken up to execute the transaction. Other than the CMMUs, there are in effect only two Mbus slave devices: the local main memory, and everything else including local I/O-space devices as well as remote memory. The local main memory has its own hardwired control logic to respond to Mbus transactions, while the latter class is controlled by the microsequencer. The main memory controller has already been covered in Section 3.1; the microsequencer is described later in 5.

This section outlines the general flow of an Mbus transaction. It introduces several autonomous control units that direct the progress of these transactions or produce control signals for the microsequencer, which is responsible for handling most types of operations. These must be understood before one can completely comprehend the microcoded control logic of the PLUS node. Their detailed design is shown on pages 2 and 18 of the schematic diagrams. For information regarding the CMMUs and the Mbus itself, please consult the Motorola MC88200 CMMU manual.

4.1 Hardwired Mbus Control Logic

4.1.1 Address Latch Control

Every Mbus transaction address is captured in the address latch and held until the addressed slave device is done with the transaction. Normally, all slaves are done as soon as the Mbus transaction is completed. However, write operations to local main memory lag behind by 2 cycles owing to pipeline stages between the Mbus and the memory. Hence, the address latch is held beyond the end of the Mbus transaction as long as the local memory controller is busy. The Mbus is held in wait state during this period to keep any new transaction in address phase. This task is carried out by two PALs, *SLVFSM* and *ALEMEMRQ*. The address latch is controlled by the ALE signal.

4.1.2 Address Decoding and Slave Notification

The address latch output signals A<31:2> and other Mbus control signals are decoded to determine the slave device to be woken up. According to the addressing scheme described in Section 2.3.2, local memory is selected only if the address MSB is 0. In addition, if the MC<3> (*lock*) and MC<4> (*cache inhibit*) Mbus control bits are both 1 during the transaction, then it is part of an *xmem* instruction; i.e., an extended operation is indicated. The decoding logic asserts the MEMREQ~ signal to wake up the local main memory controller if local memory is addressed *and* it is not an extended operation request. Otherwise, it asserts CMMUREQ~ to wake up the microsequencer to process the transaction. At the same time, the RD Mbus control bit is latched, the XOPREQ bit is asserted if it is an extended operation request, and the IOSP~ signal is asserted if the top 12 address bits are 1 (i.e., it is an I/O-space address).

If MEMREQ~ was asserted, the memory controller wakes up and carries out the requested read or write transaction according to the Mbus protocol. If the microsequencer was woken up by CMMUREQ~, on the other hand, it must first decode A<31> (address MSB), IOSP~, RD, and XOPREQ to determine how to proceed. If IOSP~ is asserted, the microsequencer must then decode address bits 19-16 to identify the particular device or operation according to Table 2-2. Note that if a CMMU was addressed, the microsequencer is still woken up. It should recognize the situation and simply let the CMMU complete the transaction. Also, thus far we have tacitly assumed that a CMMU was bus master, and the microsequencer was idling, waiting for CMMUREQ~. If the microsequencer is itself the Mbus master, there is no need to test CMMUREQ~; it can straightaway proceed to decode the other bits after starting the transaction. Further details regarding transaction execution can be inferred from later sections; it is sufficient to understand the significance of the various control signals and the task confronting the microsequencer at this point.

There is one other special case to be considered. If the processor writes to a local memory location that is replicated on other nodes, not only is the memory controller notified of the transaction (by the MEMREQ~ signal), but also the microsequencer has to get involved to update the remote copies, according to the memory consistency protocol described in Section 2.1. If the TT is enabled (indicated by the TTEN~ bit in the control and status register), and the TT status bits indicate that the local page is replicated on other nodes (see Section 3.2), then the REPWREQ~ signal is asserted to alert the microsequencer. It should then start the process of propagating the write operation to the remote memory copies. The local memory controller automatically completes the write operation to the local copy if the Mbus transaction ends successfully.

Finally, if the Mbus transaction addresses a non-existent local memory location, the TRERR~ signal is asserted to abort the transaction, and this event is noted in the TRERRST~ bit of the control and status register. If only 8Mbytes of main memory are present on the node instead of 32Mbytes, jumper *J8* should be installed to let the address decoding logic know of this fact. The transaction is aborted in the same way if there was a burst access to a remote memory page, or there was a burst write to a local, replicated page. In the former case, the addressed page was presumably being cached by a CMMU, and cached in write-back mode in the latter, both of which are considered to be system software errors. (See the discussion at the end of Section 2.1.)

The control signals presented in this section are produced by PALs *SLVFSM*, *ALEMEMRQ*, *ADEC0*, and *REPWFSM*.

4.1.3 Synchronizing the Slave Device with the Mbus

Both the memory controller and the microsequencer can take a variable number of clock cycles to respond to their respective wakeup signals. In the meantime, if the Mbus has entered data phase, the PAL *DEVSYN* holds the Mbus in wait state by asserting the SYNWAIT~ signal until the slave device responds. The memory controller responds with MEMOK~, and the microsequencer with USEQOK to acknowledge that it is now ready to carry the transaction through. In addition, if the *REPWFSM* PAL asserts REPWREQ~ to wake up the microsequencer as described above in Section 4.1.2, it holds the Mbus in wait state until the microsequencer asserts USEQOK.

4.1.4 Enforcement of Coherency Requirements

In Section 2.1, we saw that the replicated, weakly-coherent memory system must meet the requirements of write-read consistency and strong-ordering around a fence operation, which is accomplished in the PLUS system with the help the WTB. As a consequence of these requirements, there are four conditions under which a memory operation by the processor must be suspended until the condition goes away:

- If the processor reads any memory location that matches a WTB entry, the transaction must be suspended until the entry is cleared. This is necessary to preserve the sequential semantics of the local processor, i.e., the write-read consistency defined in Section 2.1.
- If the processor issues a *fence* operation and then executes a write or an extended operation to memory while the fence is still pending (i.e., the WTB is not empty), it must be suspended until the WTB becomes empty.
- If the processor executes a write or extended operation to memory, and the STRORD~ (strong-ordering mode) bit in the CSR is on (Section 3.6), and the WTB is not empty, this operation should be suspended until the WTB becomes empty.
- Finally, if the processor executes a write or extended operation to memory while the WTB is full, it has to be suspended until a free WTB entry becomes available.

The PAL *TRRETRY* recognizes all these conditions and responds by asserting the TRRETRY~ signal. This effectively suspends the processor transaction by forcing an Mbus *retry*. The slave devices, memory and microsequencer, also see the retry signal and abort without taking any action. The transaction is retried later, but the Mbus is freed for use by the microsequencer. Whenever a CMMU is made to retry in this manner, Mbus requests from the other CMMU are blocked until the first has regained the bus.

The reason for suspending the transaction by forcing it to retry, rather than by asserting wait states, is as follows. Once the processor is blocked because of any of the above conditions, the only way it can proceed is after the node receives one or more *ack* packets, thus freeing entries in the WTB. However, in order to process incoming packets, the microsequencer must first obtain the Mbus. Without the provision of a retry, the processor and the microsequencer would be deadlocked, the former holding the bus while being suspended, and the latter waiting for the bus to become free.

4.1.5 Mbus Timeout

The PAL *TIMEOUT* catches transactions that have hung around for too long and aborts them by asserting TOERR~. A transaction can time out in two ways: holding the Mbus for too long in a single tenure, or being retried several times and cumulatively holding the bus for too long. Both situations are caught. The timeout period is at least 2^{18} clock cycles (about 10ms at 25MHz), established with the help of a 14-bit counter in a DPI Xilinx device (see Section 3.8.6.6). The timeout error event is recorded in the TOERRST~ bit of the control and status register.

4.1.6 Mbus Status Generation

PALs *SS3SS2* and *SS1SS0* generate the active-low global Mbus status signals ERROR~, RETRY~, WAIT~, and EOD~ (end-of-data, to terminate a burst transfer), by OR-ing corresponding signals asserted by various devices on the node. In addition, the active-high RTRYERR signal is produced by logically OR-ing ERROR~ and RETRY~, and is used by control logic that need not distinguish between the two events. Both hardwired control logic and the CMMUs respond to generated status immediately, i.e., in the same cycle. However, the microsequencer is too slow to do so. Hence, the status signals must be latched as described below, to be tested by the microsequencer in the following cycle. See Sections 5.2 and 5.3.2 for more information on their use.

EXTWAITL	Active-low signal that is true in the current cycle if any device other than the microsequencer asserted its wait signal in the previous cycle.
RTRYERRL	RTRYERR latched at the end of the previous cycle.
ERRORLL~	The ERROR~ signal latched from two cycles before.

4.1.7 Extended Opcode Latch Control

During the data phase of every Mbus transaction, the top 6 Mbus data bits are latched in the extended opcode register *XOPREG*. The register's output signals XOP<5:0> can be tested by the microsequencer. It normally tests them during the write transaction of an *xmem* operation, or immediately afterwards, to decode the extended operation to be executed. Note that the register is only valid until the first data phase of the following transaction.

4.2 Examples

Figure 4-1 shows several examples of Mbus transaction fragments illustrating the timing relationship between some of the signals described so far. The ALE, or address latch enable, signal is negated immediately after the Mbus address is captured in the address latch. It may remain negated even after the end of the transaction, until the memory RAS signal is negated. If so, the Mbus is held in wait state to hold any new transaction in address phase, as depicted in Figure 4-1(a). The same figure also shows the active-low MEMREQ~ signal asserted to wake up the memory controller. If the memory is ready (e.g., not busy doing a refresh cycle) it immediately starts the memory cycle. MEMREQ~ remains asserted until the end of the transaction.

Figure 4-1(b) shows CMMUREQ~ asserted to awaken the microsequencer. CMMUREQ~ isn't asserted until the address phase is over, and it is negated after the first non-wait data phase. Signals A<31>, IOSP~, RD, and XOPREQ are all valid by this time, although not shown in the figure. In both Figures 4-1(a) and (b), SYNWAIT~ holds the Mbus in wait state starting from the first the data phase until the addressed device responds with MEMOK~ or USEQOK, as the case may be. SYNWAIT~ is always asserted for at least one cycle.

Figure 4-1(c) shows both the memory and the microsequencer being activated by signals MEMREQ~ and REPWREQ~ because of the processor writing to a local, replicated memory page. The latter is asserted from the second data phase. In this situation, SYNWAIT~ goes away as soon as the memory controller asserts MEMOK~, but REPWREQ~ holds the Mbus in wait state until the microsequencer asserts USEQOK. Note that in all these cases, the microsequencer can itself hold the Mbus in wait state through its UMBST<1:0> signals.

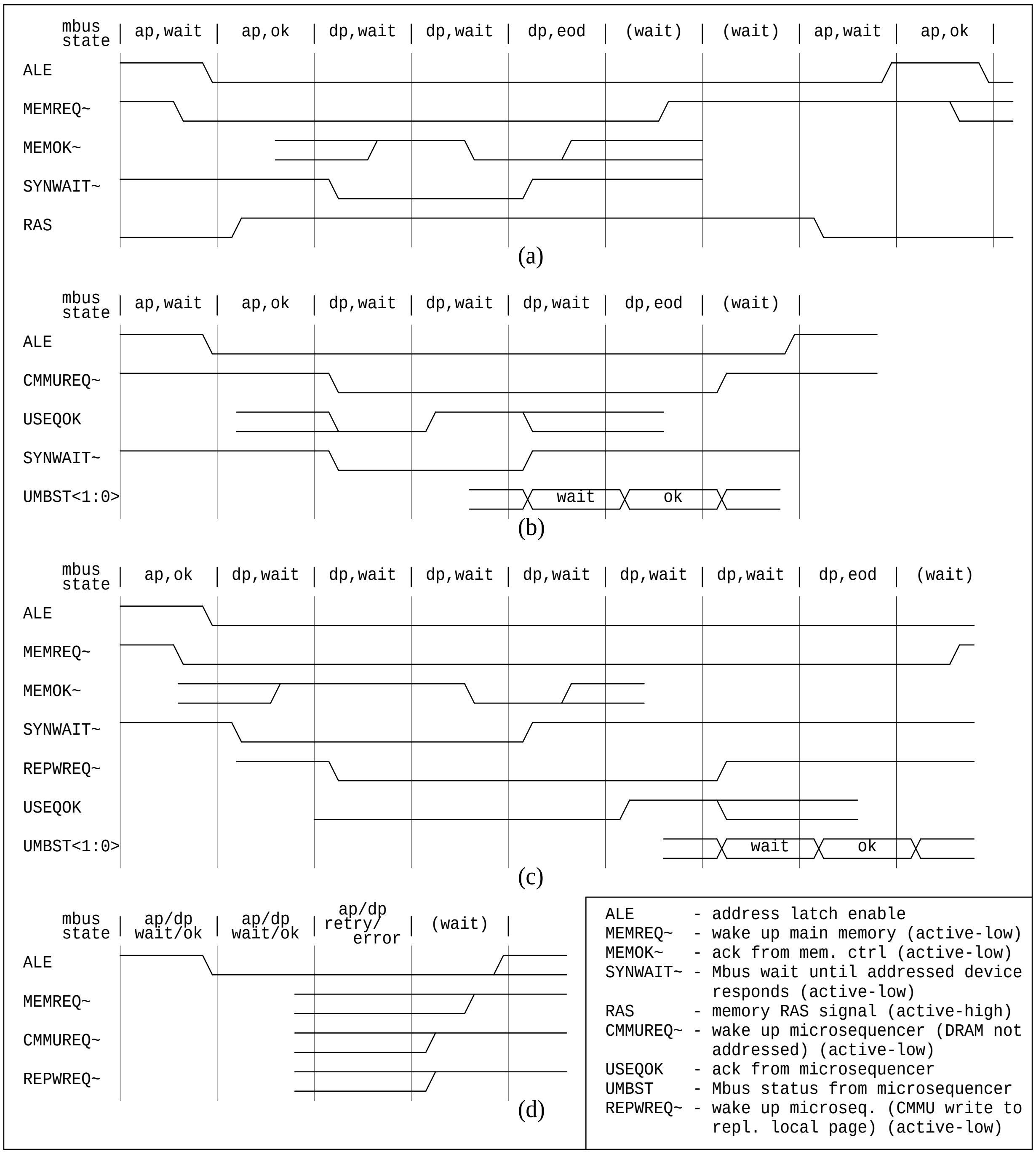


Figure 4-1: Examples of Slave Control Unit Timing. (a) DRAM slave, (b) microsequencer slave, (c) Both slaves, (d) Aborted transaction.

Finally, Figure 4-1(d) shows that all three request signals, MEMREQ~, CMMUREQ~, and REPWREQ~, are negated immediately after an Mbus retry or error cycle. The memory controller's response to Mbus transactions is described in Section 3.1. To understand more about the microsequencer and the way it deals with the Mbus, refer to Section 5.3.

5. The Microsequencer

The microcoded control logic on the PLUS node controls a major portion of its Mbus subsystem. It is responsible for handling all processor operations on I/O-space or remote memory locations, as well as write operations to local memory pages that are replicated on other nodes. It is responsible for carrying out the extended operations invoked by the processor. Finally, it must also process all received packets as well as handle all requests from the external host. The microsequencer consists of a 2Kx72 instruction store and a 5-way-branch sequencing logic. Only 64 of the 72 bits are used in this implementation; the remaining are spares. The microsequencer relies on quite a bit of auxiliary hardwired logic to carry out its functions, for example, to interface to the Mbus, and to decode several of its microinstruction fields. These are shown on pages 16 and 18 of the schematic drawings. The microsequencer itself can be found on page 17.

In Section 5.1, the microinstruction format and the function of each individual microinstruction field is documented. Its 5-way-branch sequencing mechanism is described in Section 5.2. Since the microsequencer must take a very close and active part in Mbus transactions, its interface to the Mbus is described in detail in Section 5.3. Finally, Section 5.4 concludes with an informal summary of the key tasks performed by the microsequencer. For complete details, however, one must consult the microcode itself, as it is too large to be documented here rigorously.

5.1 Microinstruction Definition

The microinstruction format is defined below in separate tables grouped according to their function. Most of the microinstruction fields have only a single function, but the ALU-control fields are dual-purpose. When these do not control the ALU they can be used to control other functional units (indicated by the active-low ALUMD~ bit being high). At such times the microsequencer should not clock the ALU, i.e., it should maintain ALUCK high.

In the following tables, the 64 relevant microinstruction bit positions are numbered 0 through 63, with bit 0 being pin *Q0* of device *UPROM0*, and bit 63 being pin *Q7* of device *UPROM7*. (See page 17 of the schematic diagrams.) The name of any field that is shared with an ALU control field is marked with an asterisk. For information regarding the timing requirements of individual microinstruction fields, see Appendix I.

5.1.1 Mbus Arbitration and Control

This section defines the microinstruction bits used by the microsequencer to gain Mbus mastership, and control the Mbus. The actual process of requesting for and obtaining the Mbus, which is carried out by hardwired logic on behalf of the microsequencer, is described later in Section 5.3.1.

The microsequencer's Mbus request UMBUSRQ has the highest priority of all requests. It is mainly used if the microsequencer needs to grab the Mbus for carrying out an extended operation issued by the processor. The three signals USEQBA, UMCAP, and UMCS are relevant for controlling the Mbus when the microsequencer is bus master. Owing to Mbus timing requirements, there is an external pipeline stage between these microinstruction bits and the Mbus. So they become effective one cycle later. Their usage is fully explained in Section 5.3.1. The Mbus status control field, UMBST, is used in both slave and master modes, and is effective in the current cycle. The following table summarizes these signals.

Name	Bits	Description
UMBUSRQ*	22	If ALUMD~ is high, assert Mbus request. Active-high.
USEQBA	26	Assert Mbus active. Similar to the CMMU BA signal, but effective in the next cycle. Active-high. (See Section 5.3.1 for its usage.)
UMCAP	27	Mbus address/data phase. 1: address, 0: data. Effective in the next cycle.
UMCS<1:0>	29:28	Specifies the remaining Mbus control bits, depending on UMCAP, as follows: If UMCAP=1, UMCS<0> = Mbus RD (MC<2>) bit, and UMCS<1> = G (MC<5>) bit. If UMCAP=0, UMCS<1:0> specifies Mbus byte strobe bits (MC<3:6>) as follows: 00: MC<3:6> = Hold (i.e., the current state is held through the next cycle), 01: MC<3:6> = IPST<3:0> (i.e., the *strb* field of input packet register), 10: MC<3:6> = all 0's (i.e., idle phase). Also forces RD to 1. 11: MC<3:6> = all 1's (i.e., a full-word operation). Effective in the next cycle.
UMBST<1:0>	31:30	Mbus status. 00: ok, 01: wait, 10: retry, 11: error.

5.1.2 ALU Control

The ALU control signals are derived directly from the microsequencer, as shown in the following table. See Cypress Semiconductor's CY7C9101 ALU data sheet for complete details.

Name	Bits	Description
ALUA<2:0>	8:6	Least significant 3 bits of the A-port address. The address MSB is always 1.
ALUB<3:0>	12:9	B-port address.
ALUBS<1:0>	24:23	Selects the source of B-port address, as defined below: 00: ALUB from the microsequencer, 01: VTAG<2:0> from VFY logic; the address MSB is 0, 10: A<5:2> from the address latch, 11: IPVT<2:0> (*vtag* field of the IPR); the address MSB is 0. It also controls the VFY address mux as follows (see Figure 3-6): 0x: VTAG from VFY logic, 10: A<4:2> from the address latch, 11: IPVT<2:0> (*vtag* field of the IPR).
ALUCN	13	Carry input to the ALU.
ALUI<8:0>	22:14	Instruction input to the ALU.
ALUCK	25	ALU clock. Must remain high if the ALU is inactive.

Note that the ALUBS bits control a 4-to-1 mux feeding the B-address port of the ALU, as well as a 3-to-1 mux in the VFY logic. These two seemingly unrelated functional units are simultaneously controlled by the same field since the first 8 ALU registers are really temporary registers for the results of extended operations. The details can be found in Sections 3.4 and 3.5. Also note that one of the B-address inputs to the ALU is A<5:2>. This allows the processor to read and write all 16 registers of the ALU via the ALUPG I/O-space region, and also read the verify registers in the VFYPG region, which correspond to the first 8 ALU registers. In the latter case, software must ensure that bit A<5> is 0.

The ALU control fields can also control other functional units in the node, in which case they are usually not meaningful to the ALU. At such times the ALU clock ALUCK should be held high to prevent inadvertently destroying the internal state of the ALU.

5.1.3 Write Tag Buffer and Verify Logic Control

Name	Bits	Description
WTBCTL	0	WTB control: 0: allocate an entry, 1: free an entry.
UWTBCK	1	WTB clock. High-to-low transition of this bit clocks the WTB.
VFYCTL<1:0>	3:2	Verify status bits control, corresponding to the following operations: 00: Allocate VFY entry, 01: Write result into VFY register, 10: Read VFY register and free the entry, 11: Overwrite all VFY status bits from the IO-bus.
UVFYCK	4	VFY logic clock. High-to-low transition of this bit clocks the VFY logic.

The clock signals UWTBCK and UVFYCK are inverted by external CMOS buffers. See Sections 3.3 and 3.4 for descriptions of the WTB and VFY blocks.

5.1.4 Datapath Interconnect Control

The following microinstruction fields control various registers, multiplexers, and tristate buffers in the DPI, which is described in detail in Section 3.8.

Name	Bits	Description
DREGCE	32	Data register *dreg* clock enable input. Active-high.
XMEM	33	If asserted, strips the extended opcode field from data register *dreg*, and zero- or sign-extends the rest of it as appropriate. Active-high.
ASEL	43	Address mux *amux* control, selects bits 30-12 of register *areg* or IO-bus: 0: *areg*, 1: IO-bus.
ADSEL	34	Controls the *ad* mux to select the following signals: 0: MBOUT-bus, 1: *address* field of input packet register.
ADCE	35	Clock enable input for register *adreg*. Active-high.
MBSEL	36	Controls the *mb* mux to select the following sources: 0: memory data from mux *mem*, 1: register *adreg*.
TBFC<2:0>	39:37	Controls the tristate buffers driving MBOUT-bus, and produces the clock-enable signal for the DPI ID register as follows: 000: Write the MBIN-bus bits 30-25 to the ID register in the next cycle, 001: enable IO-port input to MBOUT-bus, 010: enable register *areg* (after going through mux *amux*) to MBOUT-bus, 011: enable register *dreg* (after zero/sign extension, if any) to MBOUT-bus, 100: enable the *data* field of the input packet register to MBOUT-bus, 101: enable the 32-bit free-running counter onto MBOUT-bus, 110: enable the data cache hit-rate counters onto MBOUT-bus, 111: no-op.

5.1.5 NI Interface Control

The following microinstruction fields control the input and output packet registers, IPR and OPR, which were described in Sections 3.7, 3.8.4, and 3.8.5.

Name	Bits	Description
UOPDSEL*	12	Controls the mux feeding the OPR *data* field. If this bit and ALUMD~ are both 1, the mux selects MBOUT-bus, otherwise it selects *dreg*.
DSTSEL	44	Controls the *dst* mux in the DPI to select one of the following: 0: remote node ID field from bits 30:25 of *amux* multiplexer output, 1: the *origin* field of the input packet register.
ORGSEL	45	Controls the *org* mux in the DPI to select one of the following: 0: the local node ID in the DPI ID register, 1: the *origin* field of the input packet register. Also controls multiplexers feeding the *wtag*, *vtag* and *strb* fields of the OPR, as follows: 0: the WTAG, VTAG, and MBE (Mbus byte enable) signals from the node, 1: their respective counterparts from the IPR, for being forwarded.
OPTYP<5:0>*	11:6	The type field of output packets.
OPCK	46	OPR clock input. Low-to-high transition clocks the OPR. The high state also causes the OPR busy bit OPREQ~ to be asserted. Must remain low while idle.
UIPCLR*	20	If ALUMD~ is high, clears the input packet register. Active-high.

5.1.6 Sequencing Control

The following microinstruction bits control the sequencing logic, described in detail in Section 5.2.

Name	Bits	Description
UJMPEN	54	Jump to 0 if an Mbus *retry* or *error* occurred in the previous cycle. This has priority over normal sequencing. Active-high.
CCSEL<4:0>	53:49	Condition code select field, selects two condition code bits to be tested.
NUA<10:2>	63:55	Next microinstruction address.

If UJMPEN is 1, and if there was an Mbus *retry* or *error* in the previous cycle, the microsequencer does not actually jump to 0, but randomly to one of the four locations 0-3, depending on the outcome of CCSEL. Therefore, these four locations should all contain the same instruction.

5.1.7 Miscellaneous Control

The microinstruction bits defined in this section are mainly used to control various devices on the IO-bus. In particular, many of them are needed to carry out accesses to I/O-space by the processor or the external host.

The RRX~ bit mentioned in the table below is a special state bit maintained by the microsequencer that has the following function. When the processor reads a remote memory location, the microsequencer sends a request packet to the remote node and waits for the response, while holding the Mbus in wait state during this period. If it receives some other packet instead of the response to the read request, it must force the processor to *retry*, thus freeing the Mbus to process the received packet. (Otherwise, the processor and the microsequencer are deadlocked.) When the processor eventually retries the read operation, the microsequencer has to recognize it as such, and refrain from sending another request packet. It uses the RRX~, or *read-request transmitted* bit for this purpose. Before sending a read request, the microsequencer always checks this bit, and sends the request (and sets RRX~) only if it was negated. When the response finally arrives, the microsequencer returns it to the CMMU and negates RRX~[9].

Name	Bits	Description
ALUMD~	5	Active-low ALU-mode bit. If 1, the ALU control fields can be used to control other functional units.
IOSEL<2:0>	42:40	Selects the IO-bus source, as follows: 000: none, 001: ALU Y-output, 010: Translation table data, 011: Control/Status register, 100: WTB and VFY status, 101: DPI IO-port, 110: Data word from the external host interface, 111: Address word from the host interface (see Section 6.1).
TTMD	47	Translation table mode input bit. Selects one of its two sub-tables as follows: 0: master-copy, 1: next-copy.
USEQOK	48	Ack signal asserted by the microsequencer in response to being woken up to service an Mbus transaction (see Section 4.1.3, Figure 4-1). Active-high.
UCSW*	13	If ALUMD~ is high, indicates that the CSRPG I/O-space region is being written. The addressed CSR bit is then automatically controlled by hardwired logic as described in Section 3.6. Active-high.
UBITOP<2:0>*	16:14	When ALUMD~ is high, controls various CSR bits as follows: 000: no-op, 001: Negate the active-low page-copy request bit PGCPRQ~. 010: Assert the active-low remote node interrupt request RMTINT~, 011: Assert the active-low microsequencer interrupt request UINT~, 100: Negate the active-low FENCE~ bit if WTBEMPTY is true, 101: Assert FENCE~ if WTBEMPTY is false, 110: Negate the active-low RRX~ (*read-request transmitted*) bit (see text), 111: Assert RRX~.
UDEVW<2:0>*	19:17	When ALUMD~ is high, this field produces control signals as follows: 00x: no-op, 010: Reset the Mbus request state machine (see Section 5.3.1), 011: Write IO-bus bits 31-8 to the translation table, 100: Write IO-bus bits 5-0 to the CSR ID register, 101: Reset the WTB and VFY status bits, 110: Acknowledge successful completion of external host request, 111: Return error status in response to external host request (see Section 6.1).

5.2 Sequencing Mechanism

The microsequencer has two modes of sequencing depending on whether the UJMPEN bit is 0 or 1. If it is 0, the NUA<10:2> microinstruction field specifies the top 9 bits of the 11-bit next microinstruction address, while CCSEL<4:0> select two condition code bits (or two logical combinations of condition code bits) which serve as the remaining two bits of the next microinstruction address NUA<1:0>. In this way, the microsequencer can make a 4-way branch every cycle when UJMPEN is 0. If UJMPEN is 1, however, the sequencing logic not only selects and tests two condition code bits as described above, but also

[9]Note that another such state bit is required for remote read operations requested by the external host. The microsequencer uses the MSB of the ALU Q-register for this purpose.

Table 5-1: Condition Code Selection

CCSEL	NUA<1>	NUA<0>	Description
0	0	0	NUA<1:0> unconditionally forced to 0.
1	0	1	NUA<1:0> unconditionally forced to 1.
2	1	0	NUA<1:0> unconditionally forced to 2.
3	1	1	NUA<1:0> unconditionally forced to 3.
4	!CMMUREQ~+ !REPWREQ~	!UMCOE~+ !REPWREQ~	Signals tested in the microsequencer's top-level idle loop. See text.
5	A<31>	!IOSP~	Address latch MSB, I/O-space signal decoded from the address latch output.
6	RD	XOPREQ	Read request, xmem request signals.
7	A<19>	A<18>	Address latch bits 19 and 18, to decode IO-space regions.
8	A<17>	A<16>	Address latch bits 17 and 16, to decode IO-space regions.
9	!TTEN~ & IO<31>	!TTEN~ & IO<8>	Translation table *M* and *C* status bits (see Section 3.2).
10	UMRQ<1>	UMRQ<0>	Mbus mastership request, priority-encoded (see Section 5.3): 00: microsequencer request, 01: external host request, 10: input packet request, 11: no request.
11	EXTWAITL	!OPREQ~	Mbus *wait* status from the previous cycle, OPR busy.
12	IO<31>	!OPREQ~	ALU MSB or sign bit, OPR busy.
16	IPTYP<5>	IPTYP<4>	Input packet *type* field bits 5 and 4.
17	IPTYP<3>	IPTYP<2>	Input packet *type* field bits 3 and 2.
18	IPTYP<1>	IPTYP<0>	Input packet *type* field bits 1 and 0.
19	XOP<5>	XOP<4>	Extended operation opcode bits 5 and 4.
20	XOP<3>	XOP<2>	Extended operation opcode bits 3 and 2.
21	XOP<1>	XOP<0>	Extended operation opcode bits 1 and 0.
22	!EXTRDL~	!PGCPRQ~	External host read request (EXTRD) latched (see Section 6.1), page-copy request bit from the CSR.
23	VBUSY	VPEND	Selected VFY status bits (see Figure 3-6, Section 3.4).
24	ALUF0	SCTRCK	ALU F=0 status, clock for the 32-bit free-running counter. The counter should be read only if the clock is high.
25	!RRX~	!IPREQ~	*Read request transmitted* state bit, Input packet request.
26	!UMERR~	!ERRORLL~	State bit indicating that an Mbus transaction was aborted by a *retry* or *error* when the microsequencer was master, Mbus *error* status bit from two cycles ago. See Section 5.3 for the usage of these condition code bits.
27	VFYFULL	!LCLORG~	VFY entries full, IPR origin field = local ID.

simultaneously checks the RTRYERRL bit, i.e., the logical OR of the Mbus *retry* and *error* status bits latched at the end of the previous cycle. (In fact, this is the *only* way the microsequencer can test these Mbus status bits.) If RTRYERRL is false, the microsequencer branches normally as described above. If RTRYERRL is true, however, the top 9 bits of the next microinstruction address, NUA<10:2>, are forced to 0, and the microsequencer jumps to one of locations 0-3 depending on the outcome of the CCSEL test. These locations should all contain the (same) first instruction of the error-handling microcode. As a result, the sequencing logic can effectively make a 5-way branch when UJMPEN is 1.

The condition code combinations selected by CCSEL are defined in table 5-1. Note that active-low inputs are inverted (indicated by a "!") by the condition-code selection logic, in order to present all status signals uniformly in positive-logic. Any signal tested by the microsequencer must become valid within the first 10ns of the clock cycle, and remain stable until the end of that cycle. Most condition code signals always meet this requirement since they are at the output of a pipeline register. However, some of them may not. See Appendix I for some timing specifications.

The microsequencer normally idles in its top-level loop with CCSEL=4, waiting for a request. In this state, the condition code muxes logically OR two status bits each, as shown in Table 5-1. The CMMUREQ~ signal is asserted by Mbus control logic if it sees a non-local-memory transaction or a locked Mbus transaction, as described in Section 4.1.2. If a processor writes to a local memory page that is replicated on other nodes, the control logic asserts REPWREQ~ instead. (Since the microsequencer is in its idle loop, these must be CMMU transactions.) Finally, if an external host request or input packet has to be processed, a hardwired state machine arbitrates for the Mbus and asserts UMCOE~ after acquiring the bus. (This is described in Section 5.3.1 below.) Neither CMMUREQ~ nor REPWREQ~ can be true in this state as the bus has been granted to the microsequencer. So, the condition code test at the top of the idle loop can result in a 4-way branch, as follows:

- *11:* REPWREQ~ asserted. Processor writing to a local page that is replicated on other nodes.
- *10:* CMMUREQ~ asserted. Processor accessing I/O-space or remote-memory location, or issuing an extended operation request. The microsequencer should then test the Mbus operation type and the address space, by using CCSEL=6 and 5 in successive instructions, to precisely pin down the operation to be carried out.
- *01:* UMCOE~ asserted. Either an input packet or an external host request has to be processed. The Mbus has already been acquired by a hardwired state machine. The microsequencer should proceed to test the UMRQ bits to determine the source of the request. See Sections 5.3.1 and 5.3.2 for additional information.
- *00:* No request. The microsequencer stays in its idle loop.

The function of the microcoded control logic is sketched briefly in Section 5.4. To understand it in depth, consult the microcode itself.

The sequencing logic is implemented in several PALs: *UAMUX0*, *UAMUX1*, *CCMUX00*, *CCMUX01*, *CCMUX10*, and *CCMUX11*, shown on page 17 of the schematic diagrams. In addition, there is a *current microinstruction address* register (*CUAREG* on the same page) that is used to implement single-stepping logic for debugging. This is described later in Section 6.2.3.

5.3 Microsequencer Interface to the Mbus

Unlike the CMMUs or the hardwired memory controller, the microsequencer is not fast enough to respond to Mbus status or arbitration signals in a single cycle. While the former devices can test these signals in the current cycle, the microsequencer has to latch them in a pipeline register and test them in the following cycle. This makes it hard to implement the Mbus arbitration protocol directly in the microsequencer. It also imposes some restrictions on how the microsequencer responds to Mbus *wait*, *retry*, and *error* status signals. This section describes how these problems are tackled.

5.3.1 Obtaining Mbus Mastership

The microsequencer responds to Mbus signals after a one cycle delay, whereas the Mbus arbitration protocol requires immediate response. For this reason, as well as for better efficiency, the microsequencer does not actively participate in this protocol for bus mastership. Instead, a hardwired state machine, called the *bus request* state machine, recognizes when the microsequencer needs to gain bus mastership and goes through the request protocol on its behalf. After obtaining the bus, it alerts the microsequencer by asserting the active-low UMCOE~ signal, which also enables the microsequencer's Mbus control signals MC<0:6> onto the bus. The UMCAP and UMCS microinstruction bits, and the *strb* or byte-strobes field of the IPR, if necessary, serve to generate the Mbus control signals when the microsequencer is Mbus master.

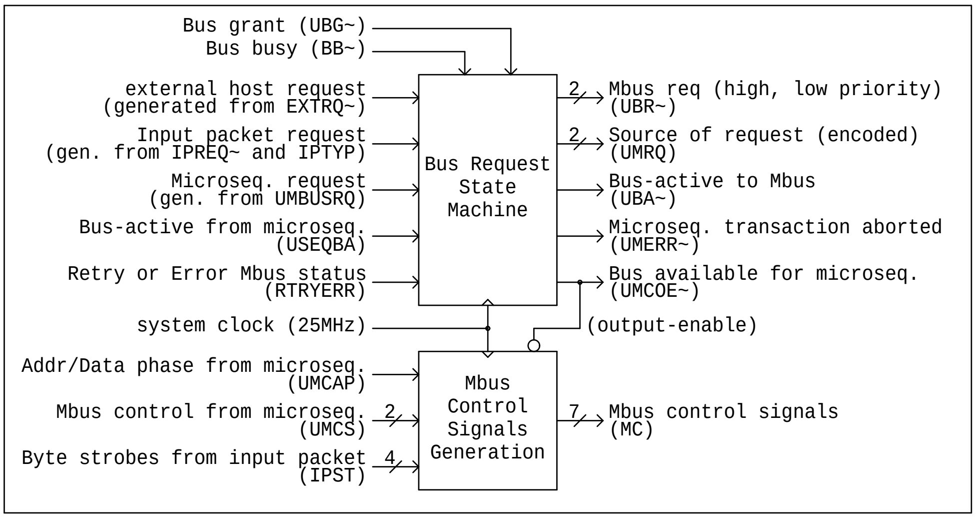


Figure 5-1: Mbus Request State Machine

Figure 5-1 shows a block diagram of the bus request state machine and the pipeline stage that generates Mbus control signals for the microsequencer. The bus request state machine recognizes the following three sources of request for Mbus mastership:

- Input packet request IPREQ~, except when it is a *read-response* or *EXT-read-response* packet. The latter two do not require the microsequencer to become an Mbus master. Rather, either the CMMU or the microsequencer is presumably already a bus master, having sent a remote read request, and is waiting for such a packet to arrive to complete the transaction.
- External host interface request signal EXTMBRQ~ (see Section 6.1), and
- Bus request from the microsequencer itself (i.e., UMBUSRQ), asserted, for instance, when it needs to carry out an extended operation issued by the processor.

The first two result in a low-priority Mbus request, and the last gets the highest priority. When the bus has been granted and is free, the state machine asserts UMCOE~ to wake up and microsequencer and simultaneously enable its Mbus control register onto the bus. The microsequencer wakes up, examines the priority-encoded request bits UMRQ to determine the source of the request, and typically executes some number of Mbus transactions to complete it, before releasing the Mbus. The extended operations usually require more than one transaction, since most of them are read-modify-write operations. In other cases, such as handling an *ack* or a *verify-response* packet, the microsequencer doesn't have to perform any transaction; its entire Mbus tenure can simply consist of idle phases while it processes the packet.

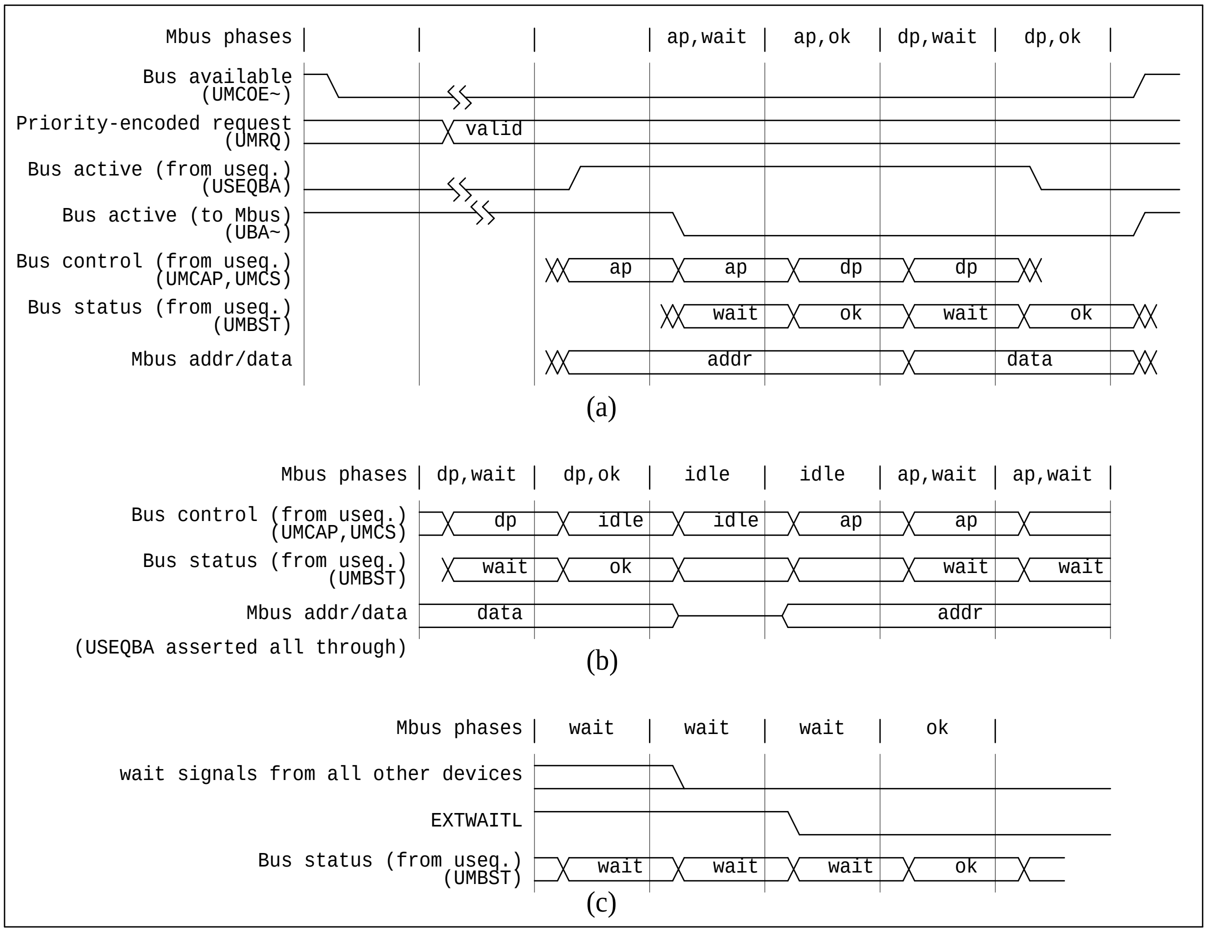


Figure 5-2: Microsequencer as Mbus Master: (a) Complete Single Transaction, (b) In Between Multiple Transactions, (c) Finishing up Address or Data Phase

Figure 5-2 shows the relative timing between various signals involved in arbitration and the microsequencer's tenure as Mbus master. Some time after the microsequencer is woken up by the UMCOE~ signal, it asserts USEQBA to indicate that it will be actively using the Mbus. The bus request state machine asserts the true *bus active* signal[10] UBA~ (i.e., the one that is seen by other Mbus devices) one cycle after the microsequencer asserts USEQBA. This is due to a pipeline stage in the state machine necessitated by Mbus timing requirements. So, the microsequencer's Mbus tenure actually starts one cycle after it asserts USEQBA. The microsequencer then executes whatever Mbus transactions it has to. It gets address and data information to and from the Mbus through the DPI, which itself contains several pipeline stages (see Figure 3-12 in Section 3.8). Therefore, the implementation of Mbus master function in the microsequencer requires careful attention to pipeline stages in both data and control paths.

To meet various timing requirements, the microsequencer must place the address information on the Mbus at least coincident with the assertion of USEQBA, i.e., at least one cycle before UBA~ gets asserted. If the microsequencer performs more than one Mbus transaction in a single tenure, there must be at least one idle phase between transactions, and the microsequencer must start driving the address for the next

[10]A bus master asserts its *bus active* signal to indicate that the other Mbus signals are valid.

transaction onto the Mbus while it is still in idle phase. The microsequencer can retain the Mbus through multiple transactions by keeping USEQBA asserted all the time.

The bus request state machine negates UBA~, thus releasing the Mbus, when it finds USEQBA negated and the Mbus in an *ok* state. Therefore, it is possible for the microsequencer to wind up its part in a transaction and release USEQBA early, and let the Mbus transaction complete as soon as the other devices (memory and CMMUs) are ready to do so. The bus request state machine also releases the Mbus immediately after a transaction is aborted by a *retry* or *error* status. If a microsequencer transaction is aborted in this manner, the state machine also asserts the active-low UMERR~ bit and refrains from making any further bus requests. Furthermore, the UMRQ priority-encoded request bits for the aborted tenure are also retained. This is necessary to let the microsequencer clean up after the aborted transaction in a manner described in Section 5.3.2 below. After cleaning up, the microsequencer has to reset the bus request state machine, by using the UDEVW microinstruction field defined in Section 5.1.7.

The bus request state machine and Mbus control signals register of Figure 5-1 are on page 16 of the schematic drawings. The PAL *UMRQREG* captures and priority-encodes a particular request, while PAL *UBRFSM* contains the actual bus request state machine. It is loosely modelled after the CMMU state machine described in Section 5.3.1.1 *Bus Arbitration State Machines* of the Motorola MC88200 CMMU manual. The Mbus control signals pipeline register is implemented in the *UMCREG* PAL.

5.3.2 Handling Mbus Status Signals

The microsequencer has to simultaneously monitor three Mbus status signals whenever it is involved in an Mbus transaction: *error*, *retry*, and *wait*. The problem is aggravated by the fact that the microsequencer cannot respond immediately to the Mbus status in its current cycle; it can only latch and test them in the following cycle.

Both *retry* and *error* status signals cause an Mbus transaction to be aborted immediately. The only way the microsequencer can respond to such an event is by asserting the UJMPEN microinstruction bit to test the RTRYERRL (Mbus *retry* or *error* in the previous cycle) bit. As described in Section 5.2, if UJMPEN is asserted and RTRYERRL is true, the microsequencer will blindly jump to one of locations 0-3. This creates a problem in that functional units might have been left hanging in some indeterminate state. For example, the microsequencer might have been in the midst of processing an input packet request when the error occurred. If the microsequencer exits to locations 0-3 the packet is still alive in the IPR and will probably cause the problem to recur, unless the microsequencer takes some clean-up action such as dropping the packet. To know what has to be cleaned up, the microsequencer requires some context information from the aborted transaction.

The status bits UMERR~, ERRORLL~, and UMRQ<1:0> provide a minimal context information to the microsequencer after any Mbus transaction is aborted. The bus request state machine for the microsequencer asserts UMERR~ if a microsequencer Mbus transaction is aborted. Furthermore, the state machine stops making any more bus requests, so the type of the request that was being serviced is frozen in the UMRQ bits. After jumping to location 0-3 on an abort, the microsequencer should execute the following error-handling code: It should immediately assert CCSEL=26 to test UMERR~ and ERRORLL~. ERRORLL~ is the Mbus *error* status bit from 2 cycles ago. In this particular case, it lets the microsequencer know if the transaction was aborted due to a *retry* or an *error*. This condition code test has the following four possible consequences:

- *00*: The microsequencer was not the bus master for the aborted transaction, which was aborted by a *retry*. There is really nothing for the microsequencer to clean up. The CMMU that was Mbus master will eventually retry the transaction.

- *01*: The microsequencer was not the bus master for the aborted transaction, which was aborted by an *error*. The CMMU that was Mbus master should recognize the bus error and trigger exception processing in the processor. Once again, there ought to be nothing for the microsequencer to clean up.
- *10*: The microsequencer was bus master for the aborted transaction, which was aborted by a *retry*. This situation should never occur if the microsequencer was in the midst of processing an input packet (as determined from the UMRQ bits). If it does, it is due to either a hardware or system software error. If the microsequencer was processing an external host request, on the other hand, it should simply return to the idle loop. The request remains pending and will assert itself again.
- *11*: The microsequencer was bus master for the aborted transaction, which was aborted by an *error*. The sequencer should then test the UMRQ bits to determine what type of request was being processed, and clean up the aborted request.

After cleaning up in the last two cases, the microsequencer must explicitly reset the bus request state machine.

The following list summarizes the possible places in a transaction where a *retry* or *error* can occur. Generally speaking, the microsequencer should not take any irreversible action as long as there is any possibility of the transaction being forced to *retry*.

- If an Mbus transaction has the G (global) control bit set, a CMMU may assert *retry* in the 4th cycle of the transaction.
- If the processor performs a write or extended operation and the WTB is full, it will be forced to retry in the 3rd cycle.
- If the processor performs a write or extended operation and a fence is pending (i.e., the FENCE~ CSR bit is on), it will be forced to retry in the 3rd cycle.
- If the processor performs a write or extended operation, and the STRORD~ CSR bit is on, and the WTB is not empty, it will be forced to retry in the 3rd cycle.
- If the processor reads a local or remote memory location that matches a valid WTB entry, it will be forced to retry in the 5th cycle.
- An Mbus transaction to non-existent local memory will be aborted with an *error* in the 2nd data phase.
- A microsequencer Mbus transaction can timeout and be aborted with an *error* after about 10-20 msec.
- A CMMU Mbus transaction can timeout and be aborted with an *error* in any cycle after the first. (The timeout period for the transaction is not really this short. However, if the transaction is retried repeatedly over a long duration, one of the retry attempts may eventually be aborted in this manner.)

Mbus *wait* states can be tested normally by the microsequencer, but still only in the following cycle. Figure 5-2(c) represents the conservative way for the microsequencer to respond to *wait* states during either address or data phases of Mbus transactions. Since the microsequencer cannot react instantaneously to wait signals asserted by other devices, it must be conservative in releasing its own *wait* signal (UMBST=1), particularly when it is the Mbus master. The microsequencer can test the active-high EXTWAITL signal, which is asserted if any device other than the microsequencer asserted Mbus *wait* in the previous cycle. The microsequencer must maintain its own Mbus wait signal until EXTWAITL is negated, and only then release it.

If one is fully familiar with the conditions under which wait states can occur, and accounts for them in writing microcode, the above restriction can be relaxed. For example, it is quite acceptable for the microsequencer to maintain address phase for just one cycle, if it is clear that no one could possibly be asserting Mbus wait in that cycle. The following list summarizes the possible cases in which devices other than the microsequencer will assert an Mbus *wait*.

- If an Mbus transaction has the G (global) control bit set, the CMMUs will assert *wait* in the 2nd and 3rd cycles of the transaction.
- If a CMMU device is addressed by an Mbus transaction, it will assert *wait* during the first 3 data phases.
- At the end of an Mbus transaction, hardwired control logic will assert *wait* for upto three cycles beyond the end of the transaction, in order to let the main memory complete the transaction or a refresh cycle.
- The Mbus control logic will always assert *wait* in the first data phase. It will remain asserted until the addressed device (memory or microsequencer) responds. If memory was addressed, it may hold the Mbus in wait state for upto 8 cycles during reads, and upto 6 cycles during writes.

Once again, it is beyond the scope of this document to go into any further detail on this subject. Please refer to the actual microcode for any additional clarification.

5.4 Description of the Microcoded Control Logic

In this section, we very briefly sketch the function of the microsequencer in the PLUS node. It is an abstract, informal, and not necessarily complete specification of the microsequencer's task. In particular, it is far from a cycle-by-cycle description of the actual implementation. The material presented here should help both the casual reader understand the microsequencer's responsibilities, as well as those interested in the precise details of the microcoded control logic implementation.

The top-level idle loop of the microsequencer has already been described in Section 5.2. We begin at a point where the microsequencer in its idle loop has just recognized a request and identified its source. As described in that section, there are essentially four possibilities:

- CMMUREQ~, asserted if the processor read or wrote remote or IO-space, or issued an *xmem* instruction,
- REPWREQ~, asserted if the processor wrote to a local, replicated memory location,
- input packet request, and
- external host request.

The last two are recognized by the microsequencer after the bus request state machine obtains Mbus ownership and asserts UMCOE~. It then tests the UMRQ bits produced by this state machine to distinguish between the two.

Figures 5-3 through 5-6 sketch the microcode required to handle these four cases, respectively. They do not form a complete specification. For example, Figure 5-5 only describes the processing of some of the input packet types. Nor do these definitions mention how the microsequencer handles Mbus transactions aborted by an *error* or *retry*. However, the former ought to be fairly obvious, given the examples in the figure, and the latter subject has been treated in some detail in Section 5.3.2. Finally, the reader should be cautioned that in the actual implementation, the microsequencer's actions are not necessarily in the exact same order as described in this section.

```
/* CMMUREQ~ asserted; processor accessed remote memory or IO-space, or invoked an xmem instruction */
/* on local or remote memory */
if (physical address is in remote memory) {     /* A<31>=1, IOSP~ false */
    if (read transaction but not xmem instr) {  /* RD=1, XOPREQ=0 */
        /* check if this is a retried attempt and request already sent */
        if (! read-request-transmitted) /* RRX~ false */
            send read-req pkt to remote node and set RRX~;
        while (IPR empty) wait;
        if (IPR.type == read-response)
            return IPR.data to Mbus, clear RRX~, clear IPR;
        else                            /* some other packet */
            force Mbus retry;           /* free the Mbus; CMMU retries later */
    } else if (write transaction but not xmem instruction) {
        allocate WTB entry and send write-req pkt to remote node;
    } else if (read transaction of xmem instruction) {
        if (no VFY entry free)          /* VFYFULL=1 */
            abort Mbus transaction;
        else
            return current VTAG;        /* VFY and WTB status word (Fig. 3-7) */
    } else {                            /* write transaction of xmem instr */
        decode extended opcode;         /* XOP<5:0> */
        allocate WTB and VFY entries;
        send appropriate extended operation request pkt to remote node;
    }
} else if (physical address is in IO-space) {
    decode address bits A<19-16>;       /* see Section 2.3.2 for details */
    if (read transaction) {
        read addressed device and return result to Mbus;
        /* if processor read VFYPG and result still pending, have to force Mbus retry */
    } else                              /* write transaction */
        write addressed device or carry out the appropriate operation;
} else {                                /* addr in local memory; must be an xmem */
    if (read transaction) {             /* read transaction of xmem instr */
        if (no VFY entry free)
            abort Mbus transaction;
        else
            return current VTAG;
    } else {                            /* write transaction of xmem instr */
        decode extended opcode;
        if (master copy for addressed page is remote) { /* from TT */
            allocate WTB and VFY entries;
            send appropriate extended operation request pkt to remote node;
        } else {                        /* master copy is local */
            terminate CMMU transaction and request for and obtain Mbus;
            execute extended op, store its result in ALU verify reg VTAG;
            /* see Section 2.2 for details, and FADD example in Figure 5-5 */
            set the VFY busy bit and clear the pending bit for the verify reg;
            if (there is a valid next-copy entry for this page) {
                allocate WTB entry;
                send update-req packet(s) to the next copy if needed;
            }
            release Mbus;
        }
    }
}
```

Figure 5-3: Definition of Microcode for Handling CMMUREQ~

```
/* External host request (UMCOE~ asserted, and UMRQ<1:0> = 01).   Mbus ownership */
/* already obtained for microsequencer by bus request state machine */
if (host requested read operation) {            /* EXTRDL~ asserted */
    get address from host interface via IO-bus and start Mbus read;
    if (physical address is in local memory) {
        wait for memory to return data (until EXTWAITL negated);
        return memory data to IO-bus and ack host request;
    } else if (physical address is in remote memory) {
        /* check if this is a repeated attempt and request already sent */
        if (! EXT-read-request-transmitted)     /* EXT-RRX is in ALU Q-reg MSB */
            send EXT-read-req pkt to remote node and set EXT-RRX;
        while (IPR empty) wait;
        if (IPR.type == EXT-read-response) {
            return IPR.data to IO-bus and ack host request;
            clear EXT-RRX, clear IPR;
        } else {                                /* some other packet */
            /* end transaction without acknowledging host request, thus freeing Mbus */
            /* the host request remains pending and re-asserts itself later */
        }
    } else {                                    /* address in IO-space */
        decode address bits A<19-16> and read the addressed device;
        return result to IO-bus and ack host request;
    }
} else {        /* host requested write operation */
    get address and data from host interface via IO-bus;
    start Mbus write transaction;
    if (physical address is in local memory) {
        wait for memory to be ready to write (until EXTWAITL false);
        ack host request;
        /* memory written when microsequencer ends transaction */
    } else if (physical address is in remote memory) {
        send EXT-write-req pkt to remote node;
        ack host request;
    } else {                                    /* address in IO-space */
        decode address bits A<19-16>;
        write addressed device or carry out the appropriate operation;
        ack host request;
    }
}
release Mbus;
```

Figure 5-4: Definition of Microcode for Handling External Host Requests

```
/* Input packet request (UMCOE~ asserted, and UMRQ<1:0> = 10).  Mbus ownership already */
/* obtained for microsequencer by bus request state machine */
case (IPR.type) {          /* decode input packet type */
VFY-response:
    store IPR.data in ALU[IPR.vtag] and clear its pending VFY status bit;
ack:
    free the WTB entry given by IPR.wtag, and clear FENCE~ CSR bit if WTB empty;
proc-proc-interrupt:
EXT-proc-interrupt:
proc-EXT-interrupt:
    assert RMTINT~, EXTINT~, or IRQEXT~ CSR bit as appropriate;
read-req:
EXT-read-req:
    get address from IPR.addr and start read transaction;
    wait for memory data to be ready (until EXTWAITL negated);
    return data in a read-response or EXT-read-response pkt as appropriate;
write-req:
    get address from IPR.addr, data from IPR.data, and start Mbus write;
    if (master copy for addressed page is remote) {     /* from TT */
        forward the write-req pkt to master node;
        abort write transaction;
    } else {
        if (there is a valid next-copy entry for this page) {
            wait for memory to be ready to write (until EXTWAITL false);
            send an update-req pkt to the next copy;
            /* local memory is written when the transaction ends */
        } else
            return ack pkt to origin of request;        /* to IPR.origin */
    }
EXT-write-req:
    get address from IPR.addr, data from IPR.data and write memory;
FADD-req:
    get address from IPR.addr and start read transaction;
    if (master copy for addressed page is remote)
        forward FADD-req pkt to master node;
    else {
        wait for memory data to become available, but hang on to Mbus;
        return data in a VFY-response pkt to origin of request;
        add FADD argument in IPR.data to memory data (using the ALU);
        write above result to same memory location (i.e., IPR.addr);
        if (there is a valid next-copy entry for this page)
            send an update-req pkt with new memory data to the next copy;
        else
            return ack pkt to origin of FADD request;
    }
...
/* other extended operation request packets also handled similarly */
...
update-req:
    get address from IPR.addr, data from IPR.data and write memory;
    if (there is a valid next-copy entry for this page)
        send an update-req pkt to the next copy;
    else
        return ack pkt to origin of request;
/* read-response and EXT-read-response packets handled elsewhere */
}
release Mbus and clear IPR;
```

Figure 5-5: Definition of Microcode for Handling Input Packet Requests

```
/* REPWREQ~ asserted; processor wrote to local, replicated memory */
if (master copy for addressed page is remote)                /* from TT */
    allocate WTB entry and send write-req pkt to master node;
else {  /* master copy is local */
    wait for memory to be ready to write (until EXTWAITL negated);
    if (there is a valid next-copy entry for this page) {     /* from TT */
        allocate WTB entry;
        send the data being written in an update-req pkt to the next copy;
    }
}
/* local memory is written when the transaction ends */
```

Figure 5-6: Definition of Microcode for Handling REPWREQ~

6. External Host Interface

Every PLUS system requires at least one external host to configure and initialize the nodes after power-up, download code into the system, and to perform physical I/O operations on its behalf. It is also desirable to attach a microprocessor at the host interface of every node for monitoring the sequence of memory accesses made by the processor on that node, which information can be fed back to algorithms that determine the page replication policy.

A host attached to a PLUS node can read and write any memory location, local or remote, by presenting the appropriate physical address according to the usual format described in Section 2.3.2. It can also access many of the I/O-space regions on the local node. The microsequencer on the node is responsible for carrying out the requested operation. However, replication is ignored for all external host accesses. In particular, this implies that write operations to memory take place only at the addressed node, regardless of whether the node has the master copy for the page, and whether the addressed page is replicated on other nodes. One further restriction is that there is no way for a host to invoke an extended operation. Finally, the PLUS system can only be a slave of the external host; the processors on the PLUS nodes can never address memory on the external host.

There are two functionally distinct sets of signals at the host interface. The first is called the EXT interface, which is used by an external host for configuring and controlling the node, as well as reading and writing memory or I/O-space regions. The second, called the ATF interface, is primarily for monitoring the memory access patterns on the node via the *address trace FIFOs*, in order to tune the global page replication structure. Basically, data memory references by the processor are automatically recorded in the FIFO, which can be scanned by the external host or microprocessor. This section contains a functional description of both interfaces.

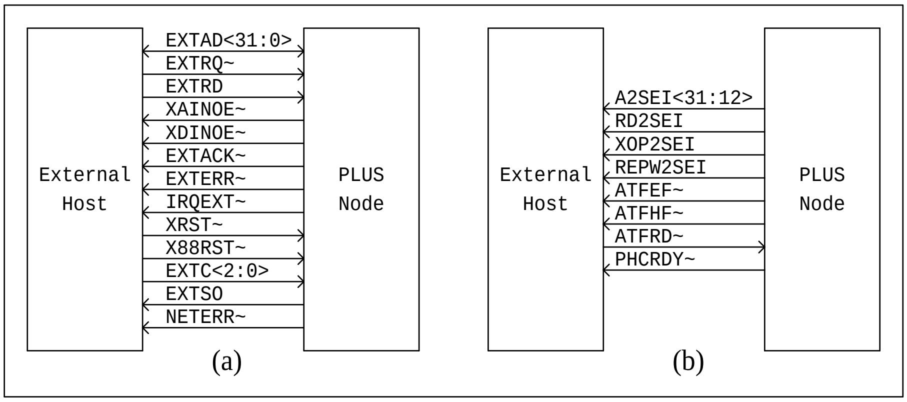


Figure 6-1: External host interface: (a) EXT interface signals, (b) ATF interface signals

The set of signals between a PLUS node and its external host is displayed in Figure 6-1. The EXT interface consists of the following signals:

EXTAD<31:0>	Address/data wires.
EXTRQ~	Active-low host operation request bit.
EXTRD	Read/write function bit (1: read, 0: write)
XAINOE~	Active-low address enable bit; enables address onto EXTAD.
XDINOE~	Active-low data enable bit; enables data onto EXTAD.

EXTACK~ Active-low acknowledgement in response to EXTRQ~.

EXTERR~ Active-low error indication in response to EXTRQ~.

IRQEXT~ Active-low interrupt request. This is actually a CSR bit (see Section 3.6.1).

XRST~ Active-low signal to reset the entire node, including the processor and CMMUs. This signal can be broadcast to the entire system.

X88RST~ Active-low signal to reset only the processor and CMMUs. This signal can be broadcast to the entire system.

EXTC<0> Dual purpose signal: (*a*) Xilinx configuration serial input data, and (*b*) load/shift mode input bit for the shift-register used in single-step debugging (see Section 6.2.3). The serial input data for Xilinx configuration can be broadcast to the entire system.

EXTC<1> Dual purpose signal: (*a*) Xilinx configuration clock control; its falling edge triggers the on-board Xilinx configuration clock generator (see Section 6.2.2), and (*b*) clock input for the debugging shift-register; its rising edge clocks the register. The Xilinx configuration clock signal can be broadcast to the entire system.

EXTC<2> Microsequencer single-step control; its falling edge triggers a single-step. Jumper *J10* must be installed to enable single-stepping.

EXTSO Serial output of the debugging shift-register (see Section 6.2.3).

NETERR~ Active-low network error signal to the external host. Each node produces two error signals IPERR~ and MRCERR~, which indicate a network error. These status bits from all the nodes in the entire system can be logically OR-ed together to produce NETERR~ and fed to an external host as follows: Each node not only sends its NETERR~ signal to its external host interface, but also to its south and west neighbours. Also, each node generates its NETERR~ signal by logically OR-ing its local error bits, as well as the error signals received from its north and east neighbours. In this way, the NETERR~ signal sent to the external host interface on node <0,0> is the logical OR of the error signals from all the nodes.

The ATF interface signals are the following:

A2SEI<31:12> Mbus transaction address bits 31-12 via the ATF.

RD2SEI Mbus transaction RD control bit via the ATF.

XOP2SEI The XOPREQ (extended operation request) bit via the ATF.

REPW2SEI Active-high version of the REPWREQ~ (processor writing to replicated local page) bit via the ATF.

ATFEF~ Active-low ATF *empty* signal.

ATFHF~ Active-low ATF *half-full* signal.

ATFRD~ Active-low ATF *read* signal from the external host. This signal is buffered and used to read the FIFOs.

PHCRDY~ Active-low signal to the external host indicating that the data CMMU cache-hit rate counters (see Section 3.8.6.5) are ready to be read. This signal is not really part of the ATF, but is included in this physical interface.

The functions of both interfaces are described in greater detail in the following sections. The EXT interface is shown on page 14 and the ATF interface on page 15 of the schematic drawings.

6.1 Host Read and Write Protocols

This section contains a description of how an external host can read and write the global main memory or I/O-space locations on the attached PLUS node. The host is constrained to make single 32-bit read or write accesses at a time. We first introduce the signals involved, and then explain the protocol.

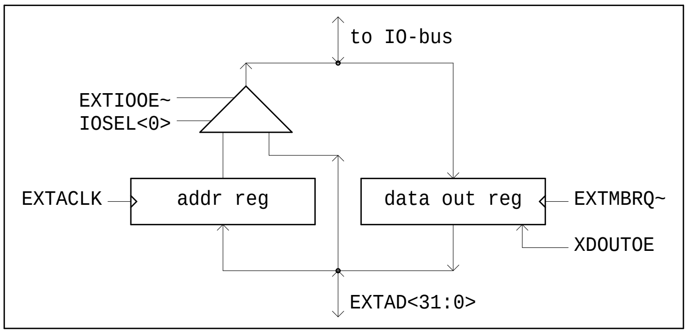


Figure 6-2: EXT Interface Datapath

Figure 6-2 is a functional block diagram of the datapath between the EXT interface signals and the IO-bus, all of which is implemented in four 74F652 devices. The control logic is not shown. Note that the EXT interface address register is distinct from the Mbus address latch. The signals shown in the figure are described below; EXTAD has already been introduced earlier.

IOSEL<0>	This microinstruction bit controls the mux shown in the figure as follows: 1: EXT interface address register, 0: EXTAD (to receive data).
EXTIOOE~	This active-low signal enables the output of the mux in Figure 6-2 onto the IO-bus. It is generated from the IOSEL microinstruction field.
EXTACLK	Address register clock, generated by the EXT interface control logic, described below.
EXTMBRQ~	Active-low Mbus request signal generated by the EXT interface control logic in response to a request, EXTRQ~, from the host. When it is negated at the end of the Mbus transaction, its rising edge also serves to clock the output data register.
XDOUTOE	Active-high output-enable signal that can drive the data register onto EXTAD, for returning the result of a read operation.

Figure 6-3 shows a timing diagram of the EXT interface read and write protocols. Only the interface signals are included; internal signals generated by the controlling state machine are not shown. The external host starts a read access by asserting EXTRD high and driving EXTRQ~ low. The EXT interface control state machine on the node reads the transaction address from the host by asserting XAINOE~, clocks the address into its address register and asserts the active-low EXTMBRQ~ request. This eventually causes the bus request state machine described in Section 5.3.1 to obtain the Mbus and wake up the microsequencer to service the request. If the read transaction is successful, the microsequencer eventually drives the result on the IO-bus and acknowledges the state machine (via its UDEVW field, not shown in the figure). At this point the microsequencer is done with its part in this process. The control state machine negates EXTMBRQ~ (which also clocks the data into the output data register), drives it onto EXTAD, and asserts EXTACK~. The external host can then latch the result present on the EXTAD lines and negate EXTRQ~, which in turn causes the state machine to negate EXTACK~. The host must wait for this last event before starting another access.

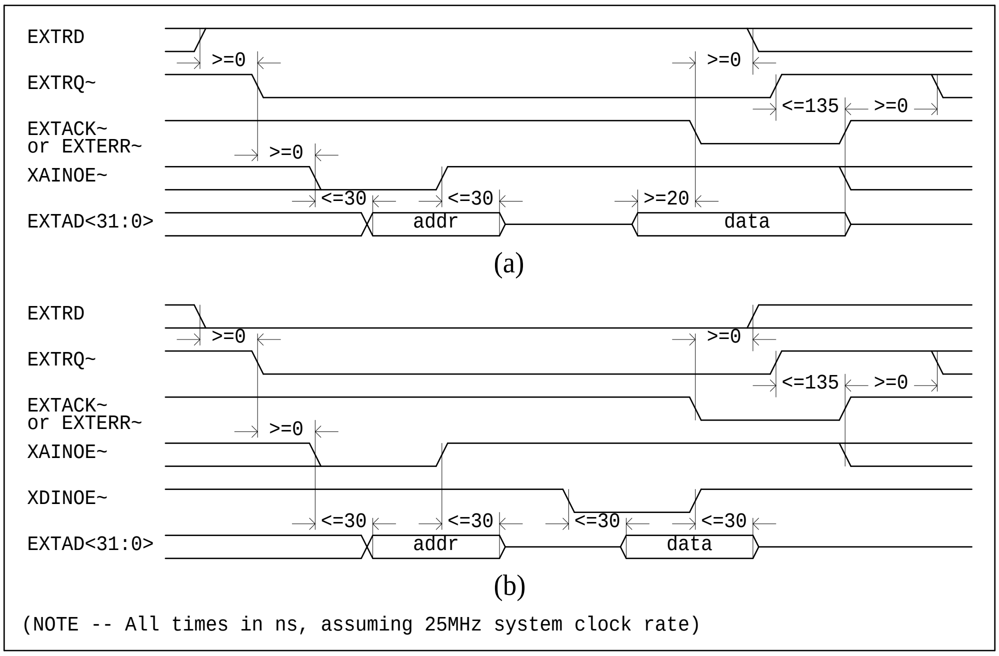


Figure 6-3: EXT Interface Protocol and Timing: (a) read cycle, (b) write cycle

If the microsequencer fails to complete the request successfully, it returns an error status using its UDEVW field (not shown in the figure), and the control state machine negates EXTMBRQ~ as before, but asserts EXTERR~ instead of EXTACK~ to the external host. EXTERR~ is negated when the latter negates EXTRQ~.

A write access is quite similar. The external host drives both EXTRD and EXTRQ~ low to start the access. The EXT interface control state machine on the node reads and latches the address and asserts EXTMBRQ~ to wake up the microsequencer, as before. It also immediately asserts XDINOE~, which should cause the host to drive the EXTAD wires with the data to be written. After the microsequencer carries out the access, the protocol is terminated as in the case of a read access, except that there is no need to return a result.

The EXT interface control state machine is implemented in the *EXTCTL* PAL shown on page 14 of the schematic drawings.

6.2 Configuring and Controlling the System

An external host is required for configuring various programmable logic arrays on the PLUS nodes after power-up, and downloading initial code into the system. It can also be used, in some cases, to debug the microcode in a single-step fashion. There are several control signals included in the EXT interface for these functions. These are described in the following.

6.2.1 System Reset

In this section, we define the reset process, including the automatic power-up reset and direct reset by the external host. The external host can send two reset signals to the node: XRST~ to reset the entire node to a known, clean state, and X88RST~ to reset the processor and CMMUs only. The latter allows the host to download initial code into the node's memory before releasing the processor. Both reset signals from the external host are active-low.

The entire node is automatically reset during the power-up sequence (which lasts several 100 msec), and remains reset until XRST~ is negated. X88RST~ must remain asserted beyond the power-up sequence[11] for at least 20 usec (512 clock cycles) before it is negated to release the processor and CMMU reset inputs. If X88RST~ was negated to begin with, it must first be explicitly asserted after the power-up sequence. This ensures that the processor is not started on an uninitialized node if the external host leaves the reset lines in an indeterminate state during power-up. These reset signals can be used by the host at any time, for example, to recover from fatal errors that leave the system in an indeterminate state.

Both reset signals XRST~ and X88RST~ can be globally broadcast from a single host interface to all the nodes in the PLUS system. Each node not only receives the two reset signals from the host interface, but also from its south and west neighbour nodes. It can choose one of the three sets via jumpers *J4* and *J9*. The reset signals so chosen are then used internally as well as broadcast to the north and east neighbours. It is intended that the south-west corner node (i.e., node <0,0>) be connected to an external host from which the reset signals propagate to the entire system.

The two reset signals from the external host should have a pulse-width of at least 20 usec (512 clock cycles) in both the low and the high states. The reset logic is shown on page 19 of the schematic diagrams. The *RSTCTL* PAL is responsible for generating reset signals for various functional units on the node.

6.2.2 Configuring the Xilinx Devices

After the system is powered-up, the 6 Xilinx devices on each node are configured in a daisy-chained, slave serial mode from the external host. (See the Xilinx XC3000 documentation. All the nodes in the system can be configured in parallel as described below.) The EXTC<0> signal from the external host is the bit serial input to the least significant Xilinx device of the DPI, *XDP0*. Starting there, the 6 devices are daisy chained for configuration in the sequence *XDP0*, *XDP1*, *XDP2*, *XDP3*, *WTB0*, and *WTB1*.

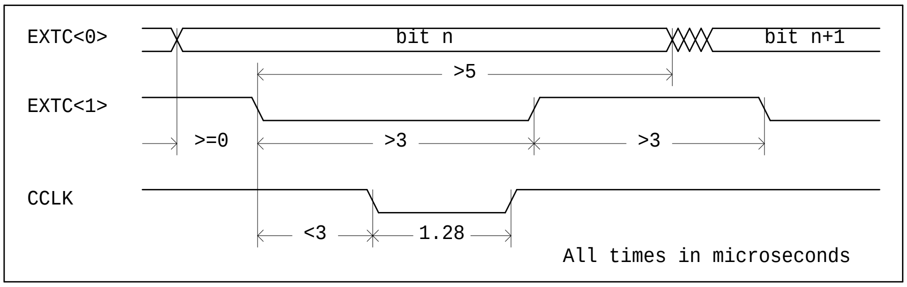


Figure 6-4: Xilinx Configuration Timing

The EXTC<0> and EXTC<1> signals from the external host are the configuration data serial input and clock, respectively. However, the Xilinx configuration clock input, CCLK, has a *maximum* pulse-width requirement of 5usec in the low-state, which not every host may be able to guarantee. Hence, there is on-board logic that generates a low-going CCLK of the right duration when it sees a falling-edge on the

[11]To be on the safe side, one should allow at least 1 sec for the power-up sequence to complete.

EXTC<1> signal. Before starting to configure the Xilinx devices, the external host should negate the system reset signal XRST~, and wait for the XINIT~ signal from the Xilinxes to go high. After downloading configuration data, the host should continue cycling EXTC<1> until the DNPG~ output from the Xilinxes goes high. (The EXTC<0> input is irrelevant at this point.) The external host can read XINIT~ and DNPG~ via the 16-bit shift-register described in Section 6.2.3. Figure 6-4 shows the timing required of the EXTC<0> and EXTC<1> signals from the external host, and the relationship between CCLK and EXTC<1>. For details on the format of configuration input data see the Xilinx XC3000 family Programmable Logic Array data book.

The Xilinx devices on all nodes in the PLUS system can be configured in parallel from a single external host. The EXTC<1:0> signals can be broadcast to all nodes in a manner similar to the reset signals described in Section 6.2.1. Each node receives the two signals from the host interface as well as from its south and west neighbours. It can choose one of these three sets via jumpers *J5* and *J11*, which are then used to configure the local Xilinx devices, as well as broadcast to the north and east neighbours.

The EXTC<1:0> signals from the external host are also used for controlling a shift-register used for debugging the microsequencer, as described in the following section. Note that both activities go on concurrently; there is no way to select one mode and not the other. However, the configuration clock generation logic automatically stops listening to these signals once the Xilinx devices are configured.

The circuitry to control the Xilinx configuration process is shown on page 14 of the schematic diagrams. In particular, the CCLK configuration clock signal is generated by the *EXTCTL* PAL.

6.2.3 Single-Step Debugging

The microsequencer can be run in single-step mode under the control of the external host. A falling edge on the EXTC<2> signal from the host triggers a step. Jumper *J10* must be installed to enable single-stepping. The single-stepping logic is implemented not by controlling the microsequencer clock, but rather by controlling a multiplexer that selects either the normal next microinstruction address or the current microinstruction address, depending on whether the microsequencer should single-step or stay in its current state. There is one exception to this rule: If the UJMPEN microinstruction bit is asserted, and there was an Mbus *retry* or *error* condition in the previous cycle, the sequencer jumps to location 0, regardless of the single-step control input. This is necessary to prevent the sequencer from missing the transient RTRYERRL signal, which lasts for only a single cycle after an aborted Mbus transaction.

The external host can observe the state of the microsequencer through a 16-bit 74F676 shift-register that can be loaded in parallel and serially shifted out. The EXTC<0> signal from the host controls the load/shift mode bit of the shift-register as follows: 1: parallel-load, 0: serial-shift, while the rising edge of the EXTC<1> signal clocks the shift-register. Its serial output is sent to the external host as the EXTSO signal. The EXTC<2> signal must have a pulse width of at least 80ns in both low and high states. See the 74F676 data sheet for the timing relationship between EXTC<0>, EXTC<1> and EXTSO.

The following 13 bits can be loaded into the shift-register and shifted out by the host; the remaining 3 bits of the shift-register are unused. Note that some Xilinx status bits have been included in the shift-register, along with the microsequencer state information.

XINIT~ Active-low INIT signal from the Xilinx devices, wire-AND-ed. The devices can be configured only after this signal is negated. After the shift-register is parallel-loaded, this signal is immediately available at the EXTSO output.

DNPG~	The DONE/PG signal from the Xilinx devices, wire-AND-ed. When this signal goes high, it indicates that the devices have been configured.
CUA<10:0>	The 11-bit address of the current microinstruction being executed.

XINIT~ is the first bit out of the shift-register, followed by DNPG~, and finally CUA<10> through CUA<0>.

The single-stepping logic is implemented in PALs *UAMUX0* and *UAMUX1* shown on page 17 of the schematic drawings. Register *CUAREG* holds the top 9 bits of the current microinstruction address, while PALs *CCMUX00* and *CCMUX10* contain the remaining 2 address bits.

6.3 The ATF Interface

The physical addresses of selected Mbus transactions are recorded in the *address trace FIFO*, a 512-entry buffer. An external host or dedicated microprocessor can read the FIFO via the ATF interface in order to analyze the memory access patterns, and modify the replication structure, if necessary. Only memory accesses by the data CMMU are recorded, I/O-space accesses and transactions by the code CMMU and microsequencer are not. This lets us concentrate on references by the local processor to just those pages that might be or have to be replicated.

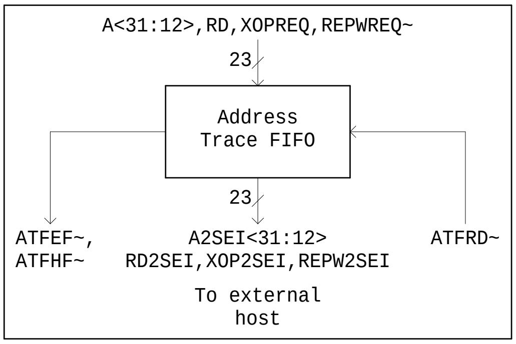


Figure 6-5: The Address Trace FIFO

Figure 6-5 shows a block diagram of the ATF. At the end of every successful memory transaction by the data CMMU, the global, physical page address bits 31-12 from the address latch are entered into the ATF (unless it is full). In addition, the following 3 bits are also recorded:

- the Mbus RD bit, indicating whether it was a read or a write transaction,
- the XOPREQ bit from the *SLVFSM* PAL, identifying extended operations being issued by the processor, and
- the REPWREQ~ bit from the *REPWFSM* PAL to distinguish writes to local, replicated pages. This active-low bit is inverted before being inserted into the ATF. Therefore, the corresponding ATF output, REPW2SEI, is active-high.

These 3 bits, together with A<31>, let the external host classify the physical address into sub-categories according to whether it was a local, remote, or replicated page, and whether the processor issued a read, write, or extended operation. The transaction information is recorded automatically and unobtrusively, without interfering with normal processing. Furthermore, transactions that are aborted with a *retry* or *error* are not recorded, ensuring that even if the CMMU is forced to retry the same transaction several times, it is recorded only once.

The FIFO *empty* and *half-full* status signals ATFEF~ and ATFHF~ can be tested by the external host over the ATF interface, and the FIFO itself can be read by the host by means of the active-low ATFRD~ signal. On the node, the ATF is controlled by the *ATFCTL* PAL shown on page 15 of the schematic drawings.

The ATF interface also sends the PHCRDY~ signal from the cache-hit-rate counter logic (Section 3.8.6.5) to the external host. This active-low signal is asserted when the P-bus cycle counter overflows, indicating that the counters can be read.

7. Software Considerations

This section outlines some of the software procedures needed to initialize and run the PLUS system. Immediately after being powered-up, the PLUS nodes contain absolutely no code whatsoever; the processors are automatically held in the reset state. Furthermore, the six Xilinx devices on each node are also uninitialized. An external host first has to configure the Xilinx devices via the host interface on node <0,0>. (Refer to Section 6.2.2 in this document, the accompanying document *PLUS Prototype Design Documentation*, and the Xilinx XC3000 series manual for further details.) Subsequently, the host can download start-up code into each node before releasing the processors from the reset state.

The following sections describe various initialization as well as normal operating procedures pertaining to the PLUS nodes.

7.1 ID Register Initialization

Before the nodes or an external host on a host interface can make remote memory references, each node should initialize its ID register via the IDPG I/O-space region (see Section 2.3.2). The only exception to this rule is the external host on the host interface of node <0,0>, which can write to remote memory locations before the ID registers have been initialized. This is necessary for downloading code into each node of a bare system after power-up, and is possible only by having jumper *J7* installed on the node, which clears its CSR ID register. Naturally, the initial code downloaded into each node has to contain the node's ID value. The easiest way to accomplish this is to reserve a pre-determined location on each node, in which the external host places their IDs during the downloading process.

7.2 Caching Policies

The hardware that maintains memory consistency among the replicated physical copies of virtual pages relies on the system software enforcing the right caching and coherency maintenance policies within the CMMUs. The caching policy depends both on whether a node has a copy of a given logical page, and whether some other node will ever access this page. System software should observe the following rules with regard to the caching policy:

- If a logical page is completely private to a given node (i.e., the physical page is on the same node, it is not replicated on any other node, nor will any other node or any remote external host ever access this page), then any caching policy is permissible. If a local external host is to touch this page, bus snooping should be enabled in the CMMUs.
- If a given node has the physical page for a given logical page, and the page is not replicated anywhere else, but it is shared (i.e., other nodes or remote external hosts may access this page), then the node cannot cache the page in write-back mode. It may cache the page in write-through mode. If other nodes can write to the page, then the CMMUs on the node should snoop Mbus transactions to invalidate cached entries when such writes take place.
- If a given node has a physical copy of a logical page that is also replicated on other nodes, then the node cannot cache the page in write-back mode. It may cache it in write-through mode. Furthermore, if any node can write to the page, bus snooping should be enabled in the CMMUs.
- When a node does not have a local physical copy of a given virtual page, it cannot cache the page at all. Bus snooping, naturally, is irrelevant in this case.
- If a node has a copy of a queue page (see the ENQ and DEQ extended operations in Section 2.2), it cannot cache the page in write-back mode, even if the page is completely private to the node.

Since snooping cannot be enabled or disabled selectively on a page-by-page basis, it will, in general, always have to be enabled.

7.3 Context Switches

Each node uses registers 0-7 in the ALU as temporary storage for the results of extended operations, until they are retrieved by the processor via the verify operation (see Section 3.4). Since there are only a fixed number of such registers on each node, they have to be saved and restored during process context-switches, as must the VFY *busy* and *pending* status bits. The system software should examine the VFY *busy* status bits (via the VWSTPG I/O-space region defined in Section 2.3.2) to check which entries have to be saved. If the *pending* status bit of any such entry is asserted, it must wait until the result is received and the *pending* bit is negated. When the process context is resumed, the system software should not only restore the verify registers in the ALU, it should also restore the VFY status bits indicating which entries are valid.

7.4 Replicating a Page

Once system software decides to create a replicated copy of a page on a given node, the main problem is to somehow carry out the replication atomically. That is, at the end of the actual copying process, the new copy should be consistent with the existing ones. Furthermore, the copy-list for the page should include the new copy and the translation tables on the relevant nodes updated to reflect the new copy-list. One possible method is to suspend all the processes that can touch the logical page until the copy has been created, and the copy-list, i.e. translation tables, updated. At this point, the new copy can be mapped by processors into their virtual address space. This approach, however, is quite inefficient compared to one that does not require processes to be suspended during the page replication process.

Replicating a page while the system is in operation, however, can be tricky. The problem, specifically, is how to create the copy, and at the same time propagate any writes to the logical page to the new copy, such that the two are merged in the right chronological order. This requirement is not necessarily guaranteed automatically, owing to uncontrollable network delays. Consider the following example: A word is read from an existing copy and transmitted in a *page-copy* packet to the new page as part of the copy operation. A little later, a processor writes to the same word, causing update packets to be sent everywhere, including the new copy. If the page-copy packet has to traverse a longer network distance to the new page than the update packet, the latter may overtake it. Consequently, the new page will be inconsistent with the other copies. We describe below one method for creating a consistent copy. It relies on the microsequencer to carry out the copy operation and requires that the new copy not be the master copy in the new copy-list.

Before the copy operation is started, system software has to update the copy-list by inserting the new page in it at any suitable location other than being the first or the master copy. It then updates the translation table on the node with the new copy (i.e., creates the master-copy and next-copy pointers), and then updates the next-copy table entry on the node with the *previous* copy; no other translation table has to be updated. Any write or extended operation to the logical page will now propagate to all the copies including the new page. At this point, the copy operation is begun between the predecessor to the new page and the new page itself. That is, the microsequencer on the predecessor is set up to read and transmit the entire page to the new copy, one word at a time (described in further detail below). When the copy operation is complete, the new page has been fully initialized and can be mapped into the virtual address space of any processor. Since the new page receives page-copy packets as well as update packets from the same source, i.e., its predecessor in the copy-list, the chronological ordering of copy and update operations is preserved, and the new page is consistent with the other copies.

The microsequencer on the new copy’s predecessor is woken up to perform a page-copy operation by writing to the CSRPG and asserting the PGCPRQ~ bit (Section 3.6.3). The processor should have written the physical address of the local page to be copied into ALU register 12. The microsequencer obtains the physical page address on the destination node, where the new copy is to be created, by looking up the TT. When it is done copying the entire page, it clears the PGCPRQ~ bit and interrupts the processor.

Appendix I. Timing Requirements

Normally, the control signals specified by the microsequencer in one cycle are effective in the same cycle. For example, if the microsequencer asserts the clock enable signal for any of the DPI registers *adreg*, *areg*, or *dreg*, that register is clocked in that cycle. In many other cases, owing to combinatorial logic propagation delays, a microinstruction signal requires more than one cycle to take effect. There are also several combinatorial data paths in the system, particularly in the DPI, that require more than a single cycle to stabilize. This appendix contains the timing specification of several important data and control paths.

The following table shows the timing of several datapaths pertaining to the DPI Xilinx devices. Refer to Figure 3-12 in Section 3.8.

Description	ns
ASEL microinstruction bit setup time before clocking the OPR	> 55
DPI *dreg* register clock setup time before clocking the OPR	> 80
XMEM microinstruction bit setup before clocking the OPR	> 60
UOPDSEL microinstruction bit setup before clocking the OPR	> 55
ADSEL microinstruction bit setup before clocking *adreg* register (no parity generation)	> 25
MBOUT-bus setup before clocking *adreg* register (no parity generation)	> 15
Parity-generation logic input valid to its output valid	< 30
ASEL microinstruction bit valid to MBOUT-bus valid	< 30
DPI *dreg* register clock to MBOUT-bus valid	< 60
XMEM microinstruction bit valid to MBOUT-bus valid	< 35
IO-bus port input valid MBOUT-bus valid	< 20
TBFC microinstruction field valid to MBOUT-bus valid	< 50
MBOUT-bus valid to IO-bus valid	< 20
DPI *mbin* register clock to PRTYERR~ parity check signal valid (assuming data held constant)	< 80

The timing of the WTB and VFY logic is given in the following table. Refer to Figures 3-5 and 3-6 in Sections 3.3 and 3.4.

Description	ns
WTBCTL microinstruction bit setup before clocking the WTB	> 60
VFYCTL microinstruction field setup before clocking the VFY logic	> 35
ALUBS microinstruction field setup before clocking the VFY logic	> 65
UWTBCK microinstruction bit falling edge to WTBFULL, WTBEMPTY, WTAG valid	< 65
ALUBS microinstruction field valid to VBUSY, VPEND valid	< 80
UVFYCK microinstruction bit falling edge to VFYFULL, VTAG valid	< 55
Address latch output valid to the WTBHIT output of the WTB valid[12]	< 100

[12] The address latch output is valid about 25ns into an address phase cycle, provided the latch is enabled (transparent).

The following table pertains to the IO-bus, and miscellaneous devices residing on it. Refer to Figures 3-5 and 3-6 in Sections 3.3 and 3.4.

Description	ns
IO-bus setup time before clocking the ALU	> 60
IO-bus setup time before clocking VFY logic, CSR, or TT	> 20
ALU control microinstruction fields valid to ALUF0 ALU status bit valid	< 70
ALU control microinstruction fields valid to ALU driving valid data on IO-bus	< 70
IOSEL microinstruction field valid to DPI IO-bus port driving IO-bus	< 60
IOSEL microinstruction field valid to VFY/WTB status word driving IO-bus	< 60
IOSEL microinstruction field valid to ALU driving IO-bus	< 35
IOSEL microinstruction field valid to any other device driving IO-bus	< 25

Note that none of the tables takes the clock to output delay of the microinstruction pipeline register into account. In writing microcode, it will often be necessary to add an extra 10ns to the timing figures in these tables to account for this delay. Also, if the microsequencer clocks the WTB, VFY logic, the ALU, the TT, the CSR ID register, or the OPR in a particular cycle, it should ensure that the control and data inputs to that device are held through the end of that cycle.

Appendix II. Packet Types

The following tables list the set of packet types used in the PLUS system, and identify the relevant fields in each case. The destination field, which is present in all packets transmitted, is not shown. The top two bits of the packet type field are constrained to indicate whether the packet contains address and data words, as described in Section 2.4.2. Bits 3 and 2 are encoded so that the most urgent packets--write, update and ack packets--can be decoded immediately without looking at any other bits. The first table below lists the three packets mentioned above. The remaining tables are grouped by the top 2 bits.

Name	Type	Origin	Address	Data	Strb	Wtag	Vtag
Ack	00 0000					x	
Write-req	11 0100	x	x	x	x	x	
Update-req	11 1000	x	x	x	x	x	

The remaining packets are listed below. The first table lists control packets:

Name	Type	Origin	Address	Data	Strb	Wtag	Vtag
Proc-proc int	00 1100						
EXT-proc int	00 1101						
Proc-EXT int	00 1110						
Page-copy ack	00 1111						

Response packets:

Name	Type	Origin	Address	Data	Strb	Wtag	Vtag
VFY-response	01 1100			x			x
VFY-resp-Ack	01 1101			x		x	x
EXT-read-resp	01 1110			x			
Read-response	01 1111			x			

Request packets that contain an address word but no data word:

Name	Type	Origin	Address	Data	Strb	Wtag	Vtag
FSET-req	10 1100	x	x		x	x	x
DEQ-req	10 1101	x	x		x	x	x
EXT-read-req	10 1110	x	x				
Read-req	10 1111	x	x				

Request packets that contain both an address word and a data word:

Name	Type	Origin	Address	Data	Strb	Wtag	Vtag
FADD-req	11 1100	x	x	x	x	x	x
ENQ-req	11 1101	x	x	x	x	x	x
EXT-write-req	11 1110		x	x			
Page-copy-req	11 1111	x	x	x			

Except for *read-response* and *EXT-read-response* packet types, all the other packet types are of interest only to the microsequencer. Hence, they can be redefined, or new types added as needed.

Appendix III. Jumper Configurations

Each PLUS node contains several jumpers and edge connectors that have to be configured properly, depending on the position of the node in the entire system. There are jumpers to indicate the size of installed main memory on the node, to enable microcode single-stepping, to route global signals throughout the system, and to terminate the network at the edges of the mesh interconnection. This section lists and describes all of them.

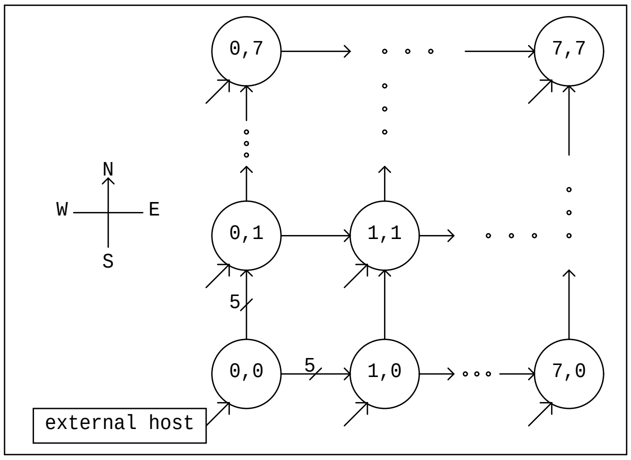
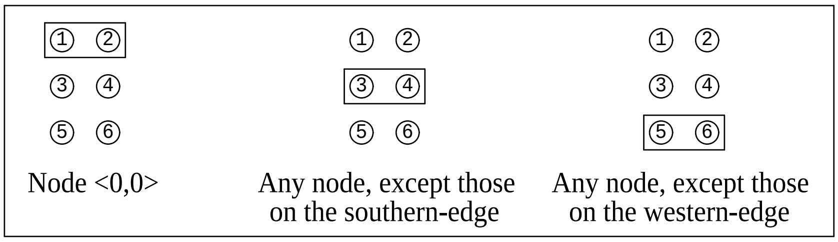


Figure III-1: Propagating Global Signals from Node <0,0> through the System

As shown in Figure III-1[13], a PLUS multi-processor system requires five global signals to be broadcast throughout the system, principally so that it can be controlled or monitored from a single external host. The EXTC<1:0> signals are used to configure the Xilinx devices on all the nodes from the external host at node <0,0>. Similarly, the reset signals from the external host, XRST~ and X88RST~, are also broadcast through the system. Finally, node <0,0> generates and broadcasts the CTRCK signal, which clocks the 32-bit free-running counters in the nodes' DPI (see Section 3.8.6.4), keeping all of them synchronized.

From the above figure, it should be clear that each node receives each global signal from three possible sources: its west and south neighbours, and its own host interface (or the node itself, in the case of the CTRCK signal). Each node picks one of these three sources for its internal use via a jumper, and also buffers the selected signal and propagates it to its north and east neighbours. The five jumpers to select the five global signals are *J4*, *J5*, *J6*, *J9*, and *J11*, shown on pages 14 and 20 of the schematic diagrams. They should be set identically on a given node, according to its position, as shown below:



[13]In the figure, node <0,0> is shown with an external host, while the remaining nodes may or may not have one at their local host interface. In this appendix, we will ignore the presence of any such additional hosts.

The remaining jumpers on each node are listed below. See pages 2, 17, and 18 of the schematic drawings.

- *J7:* Installed on node <0,0> only, to clear its CSR ID register.
- *J8:* Installed on a node if it has 8Mbytes of memory instead of the full 32Mbytes.
- *J10:* Installed to allow single-stepping the microsequencer.

Finally, the MRC (mesh router chip) request signals that go off the edges of the system should be looped back to their corresponding acknowledge inputs, as described below. This ensures that any packet destined for a non-existent node will be "routed off" the edge of the system. All connectors mentioned below are on page 13 of the schematic diagrams.

- *Northern and eastern edges:* Connect pins 4 and 6 of connectors *J13* and *J12*, respectively.
- *Western and southern edges:* Connect pins 28 and 30 of connectors *J15* and *J14*, respectively.

Appendix IV. LED Displays

Each node contains nine LED displays indicating the status of some key hardware components. These are described below:

- *D1*: This LED is on whenever the node is powered up.
- *D2*: If a node encounters an interconnection error (either an unexpected *tail* bit received by the self-timed router chip, or a bad packet checksum), this LED is turned on.
- *D3*: This LED indicates activity on the *po* port of the self-timed router chip (i.e., data being received by the node).
- *D4*: This LED indicates activity on the *pi* port of the self-timed router chip (i.e., data being transmitted out of the node).
- *D5*: When the input packet register is full, this LED is on.
- *D6*: If the output packet register is full, this LED is on.
- *D7*: This display goes on whenever the Mbus is busy.
- *D8*: This LED indicates whether the Xilinx devices have been configured or not. It should be on after power-up, and should go off after configuration is complete.
- *D9*: When the processor and CMMUs are reset (i.e., after power-up and when the X88RST~ or XRST~ input from the host interface is asserted), this display is on. It is off if the processor is running.

Table of Contents

List of Figures

List of Tables