Hardware Architecture Considerations in the WE32100 Chip Set

Michael L. Fuccio
AT&T Bell Laboratories

Benjamin Ng
AT&T Information Systems

The interchip communication protocols for the WE32100 microprocessor chip set had to be designed to support several ambitious architectural goals.

The WE32100 series of VLSI chips is an upgrade and addition to the WE32000 (formerly, Bellmac-32A) 32-bit microprocessor family. The WE32100 family is a CMOS chip set that differs from its predecessor in operating speed as well as in hardware architecture. Some software architecture modifications of the upward-compatible type were also implemented. Here, we discuss the WE32100 chip set's interchip protocols as they relate to system hardware architectures. These protocols were defined at the same time the WE32100 family was designed. We also examine architectural trade-offs as they pertain to the WE32100 chip family.

Goals

A number of goals guided the definition of the hardware architecture. We wanted to

- provide an interchip protocol that would support competitive performance in a 32-bit microprocessor operating at a prescribed frequency,
- provide a true all-VLSI chip set requiring no SSI "glue,"
- allow for multiple chip set configurations, to permit a system designer to customize the chip set to his needs,
- support the software architecture of the entire chip set,
- retain some compatibility with predecessor chips so that Bellmac customers desiring higher performance via the WE32100 chip set would not need to perform wholesale architectural changes when upgrading, and
- allow for easy interfacing to commercial memories and peripherals.

Alternatives investigated

We investigated over a dozen alternatives for an interchip communications protocol. They fell into three categories:

- asynchronous protocols compatible with the WE32000 (Bellmac-32A) microprocessor module,
- asynchronous protocols with pipelined memory transactions, and
- fully synchronous protocols with "split-read" memory transactions.

The first category included about eight variations on the theme. The WE32000 module has a four-cycle (zero-wait-state) memory transaction. We felt that a cycle could be eliminated from this access and, further, that excess over-

0272-1732/86/0400-0029$01.00 © 1986 IEEE

head on the WE32000 module could be eliminated. This alternative and its variations, then, provided an asynchronous, three-cycle memory transaction for the CPU and any other bus master.

We evaluated the possibility of a memory transaction of two (or less) clock cycles, and the preliminary timing information we obtained indicated that, for a two-cycle memory access, very little memory access time would be available for common memory parts. This did not meet our first goal, that is, that the speed be competitive. It was our judgment that a two-cycle memory transaction would not be usable with common memory well into the foreseeable future. Furthermore, certain functionality would have been lost if the transaction had been made two cycles long.

In sampling asynchronous signals, there is a high cost associated with preventing input signal metastability. Avoiding metastability requires a signal input to be doubly latched. This requires additional latching times (clock edges), which a two-cycle transaction could not have provided.

As a result of all these difficulties, we abandoned the two-cycle approach in favor of the three-cycle one.

The second class of alternatives also addressed our desire to reduce the length of a memory transaction to two cycles. In these alternatives, a three-cycle access would overlap, by one cycle, the subsequent access. This memory transaction overlap would yield an effective two-cycle transaction. We felt that this overlap could occur often, given the instruction prefetch policy of the CPU and the early availability of the next instruction. More careful analysis, however, showed that the occurrence of memory accesses that could be overlapped is relatively small for the WE32100 CPU. Furthermore, a write transaction cannot be overlapped. Yet another problem, an internal CPU bus conflict, all but eliminated "overlappable" transactions. Given these facts, we abandoned the second class of alternatives.

At the time of the WE32100 development, our investigations of the third class of alternatives were still very exploratory. Furthermore, the alternatives in this class did not meet the goal of compatibility with Bellmac chips. Since compatibility was very important and since the concepts of this class were not well proven, we abandoned it.

Table 1 summarizes how the three classes of protocol alternatives met our goals. The details of the chip set protocols are described in the WE32100 information manual.[1]

In the remainder of this article, we will describe some of the more important issues relating to the realization of a hardware architecture for a 32-bit VLSI chip set.

Table 1.
How the three chip communication protocol alternatives met the WE32100 architectural goals.

Goal	Alternative 1	Alternative 2	Alternative 3
Competitive performance?	Yes	Yes	Yes
Glue logic?	None	Small amount	None
Multiple chip configurations?	Yes	Yes	Yes
Software architecture support?	Yes	Yes	Yes
WE32000-compatible?	Yes	No	No
Easy commercial interface?	Yes	Yes	No

Chip set protocol issues

Here, we examine the issues that arose when we sought to choose a chip set protocol that would meet our goals. These issues involved

- choosing asynchronous or synchronous operation,
- choosing multiplexed or nonmultiplexed addresses and data,
- implementation of transaction types, including reads, writes, interlocked reads, block fetches, and interrupt-vector number fetches,
- exception handling,
- bus arbitration,
- the implementation of the interface to the memory management unit, or MMU,
- the handling of support processors,
- providing for special needs of the chip set architecture,
- choosing the operating frequency, and
- electrical design.

Before discussing each of the issues listed above, we should describe, briefly, some of the features of the WE32100 chip set. Additional detail can be found in several technical papers.[2-5]

The WE32100 CPU has a rich instruction set that includes operating system support instructions. Two instructions requiring I/O protocol support are implemented. The first is the SWAPI (swap interlock) instruction and the second is the MVTRW instruction. We will discuss support for these instructions later. The WE32100 CPU also supports a rich interrupt structure, including an "autovector" feature.

The WE32100 CPU has an on-board instruction cache that fetches a two-word block on a cache miss. This two-word fetch has a hardware protocol support mechanism, which we will discuss later.

The WE32101 MMU provides typical virtual memory functions—that is, address translation and protection—for the WE32100 chip set. The MMU supports segmentation

and paging and, in the paged translation mode, a special I/O protocol feature called "early RAS."

The WE32106 math accelerator unit, or MAU, complies with Draft 10.0 of the proposed IEEE floating-point arithmetic standard.[6] It can be configured as a support processor (i.e., make use of the WE32100 support processor interface) or as a memory-mapped peripheral.

The WE32103 dynamic RAM controller supports up to four banks of dynamic memory. The DRAM controller also supports dual-ported memory and has an interface for an error detection and correction chip. It is programmable for various speeds of dynamic RAM.

The WE32104 direct memory access controller features a system bus and a peripheral bus. The system bus is a 32-bit-wide bus designed for memory-to-memory DMA. The peripheral bus is an eight-bit-wide bus optimized for byte-wide peripherals such as UARTs and LAN interfaces. The DMA controller can perform system-bus-to-peripheral-bus transactions and also allows the CPU to access peripherals, on the peripheral bus, from the system bus. The DMA controller is programmable for burst-mode or cycle-steal transactions and can do double and quad-word fetches.

Asynchronous vs. synchronous protocols. Our intent here is not to pontificate on all that one should know about synchronous and asynchronous buses but rather to clarify some frequently misused terms and to discuss some of the issues relevant to the architecture of the WE32100 chip set.

Synchronous buses. In a synchronous bus, all events (state changes or signal assertions) occur at specific time instants that are invariant with respect to a clock signal. Synchronous buses are typically specified in terms of clock edges—that is, a bus event occurs relative to a change in the state of a clock signal. Another attribute of synchronous buses is that both the transmitter and the receiver on the bus share a common clock or derivatives of a common clock. The shared clock signal provides the means for each bus transmitter or receiver to have a global knowledge of bus activity. If the architecture is sophisticated enough, each entity on a synchronous bus can be aware of the other entities' activity solely because of the shared clock signal. We will pursue this concept further when we discuss the WE32100 support processor interface.

Asynchronous buses. In asynchronous buses, bus events (state changes or signal assertions) do *not* occur at well-defined times—they can occur at any time. Asynchronous bus transactions require a two-way negotiation between transmitter and receiver; that is, the transmitter must inform the receiver that a bus event has occurred. (It usually does this by means of a *strobe signal.*) The bus receiver must then acknowledge that it has recognized the event. This recognition signal is also asynchronous. Thus, bus events on an asynchronous bus are typically governed by the assertion and negation of strobe signals. Because of the asynchronous nature of bus events, there is no shared information among entities on the asynchronous bus. Such entities can only react to bus events, since they cannot be made aware of the activity of other bus entities in a global sense.

Entities on an asynchronous bus typically do not share a clock signal governing bus transactions. Typical VLSI devices advance their internal state via a clock signal. This causes some difficulty in connecting the asynchronous bus to the synchronous internals of such devices. Metastability is a problem here.[6-9] To avoid having a sampled asynchronous signal become metastable, one should be sure the signal is doubly latched. That is, the signal, once sampled, should have its sampled value latched some time later to resolve a potential metastability condition. The need to doubly latch an asynchronous signal means that more time must be dedicated to guaranteeing the state of that signal. This can mean an overall loss of bandwidth for an asynchronous bus compared to a synchronous bus.

Pseudosynchronous buses. A pseudosynchronous bus has the attributes of both a synchronous and an asynchronous bus. The WE32100 chip set's buses can be characterized as pseudosynchronous. Its I/O protocol, for example, is synchronous (timing is specified in relation to the chips' input clock), whereas its method of indicating bus events is asynchronous (strobe signals are provided to indicate such events). Furthermore, members of the WE32100 family sample and doubly latch many of the signals from the bus receiver; this implies that WE32100 chips are allowed asynchronous behavior. The WE32100 support processor interface, however, relies on synchronous behavior—behavior that in the WE32100 chip set can be guaranteed because of the way that I/O timing is specified in relation to the input clock.

In an optimum configuration, the members of the WE32100 chip set operate synchronously among themselves but present an asynchronous interface to the external system. Indeed, the CPU, MMU, and MAU always share the same clock. In a less than optimum configuration, the DMA controller and the DRAM chips need not run on the same clock as the other chips in the set.

Multiplexed vs. nonmultiplexed addresses and data. In designing a 32-bit VLSI chip set, one must decide whether address and data transmission should be shared or kept separate. A 32-bit system with a full 32-bit address space as well as a 32-bit data transmission capability needs 64 wires (I/Os) to transmit both address and data. For VLSI chips, the number of chip I/Os that one can have is limited; moreover, chip I/Os can be a major contributor to overall chip cost. Thus, in the VLSI domain, it is desirable to keep chip I/Os to a minimum.

To take advantage of the full potential of a 32-bit architecture, one wants to transfer data in chunks of at least 32 bits. A 32-bit address space and a 32-bit data path suggests that 32 chip I/Os can be shared by addresses and data (i.e., can be multiplexed). Such multiplexing has been used to good effect in a few 32-bit VLSI microprocessors. The drawback to multiplexing is that most common static

RAMs and peripheral chips require an address to be available when they present data to the microprocessor. This requirement means that the address and data must be nonmultiplexed, and it dictated that the WE32100 chip set have separate 32-bit address and data buses.

Though this approach was more expensive in terms of the number of chip I/Os, it did avoid the need for SSI glue. A microprocessor with multiplexed address and data lines handles the need of common static RAMs and peripheral chips for nonmultiplexed addresses and data by latching the address when it is present and presenting the latched address to the memory while accepting or transmitting data on the bus. This process typically reduces the amount of time available for memory access; hence, the WE32100 has nonmultiplexed address and data buses.

Read transactions. The goal of a read transaction is to move data from memory (or a peripheral) to the CPU to undergo interpretation or modification. Optimum bandwidth is achieved when a full 32-bit data chunk is transferred. Common microprocessor read transactions consist of two parts. The first is the issuance of the address to be read from. The second is the acknowledgment of the read request by the memory along with the putting of data on the data bus so it can be sampled by the CPU. Let us call the first part the *address portion* and the second part the *data portion*. A typical microprocessor (which has asynchronous protocols) waits, after the issuance of the address, for the data acknowledgment to occur. Such a microprocessor occupies the bus for the duration of the wait. Such a wait is commonly called a wait state, and a wait state is the duration of some clock signal (usually the CPU clock). Ideally the address portion and the data portion of a transaction are separated by zero wait states. This means that the CPU does not need to wait for data. Therefore the CPU is busier and the performance of the CPU is improved.

A very important parameter, zero-wait-state memory access time, is the critical time between the address portion and data portion allowed for memory to present data to the CPU. That is, zero-wait-state memory access time is the time allotted for the primary memory system to complete a read or write operation without incurring wait states. A good read transaction protocol should maximize the zero-wait-state memory access time to ease the computer designer's timing budget requirements as well as to accommodate a broad spectrum of memory types, costs, and technologies. Therefore, to maximize zero-wait-state memory access time (and hence performance), the address portion and data portion of the transaction should be minimized. To be sure, the ratio of memory access time to total transaction time (at zero wait states) ideally should approach 1. Hence it is desirable that $T_m/T_{tot} \rightarrow 1$, where T_m is memory access time (zero wait states) and T_{tot} is total transaction time (zero wait states).

It is clear that if the T_m/T_{tot} ratio is near 1 and the data being transferred is a 32-bit word, the information transfer is maximally efficient for the microprocessor. This does not necessarily mean that the memory system is being used to maximal efficiency. If the memory system access time, T_{mp}, is much less than T_m ($T_{mp} << T_m$), clearly the microprocessor is slower than the memory system and the speed of the memory system cannot be exploited by the microprocessor. This presents an interesting problem to the hardware architect. He must ask if the I/O protocol should anticipate the evolution of memory technology and try to achieve maximum memory efficiency for fast memories. The naive answer to this question is yes. This means that if $T_{mp} >> T_m$, the only way to make the T_m/T_{tot} ratio approach 1 (and hence produce an efficient transfer rate) is to reduce T_{tot}. This may be easier said than done. Reducing T_{tot} may affect the hardware architecture of the microprocessor chip. Typically $T_{tot} = n * T_c$, where T_c is the period of the bus clock and n is a positive integer. One can see that making n smaller is one approach to reducing T_{tot}. However, reducing n may fundamentally change the internal chip architecture, design, and clocking strategy. Designing VLSI chips to reduce T_{tot} is a laudable goal, but practical design considerations may not allow n to be as small as 1 or 2, especially with an asynchronous or pseudosynchronous I/O protocol. Sampling and doubly latching asynchronous signals using a clock to generate sampling edges and then making decisions based on the sampled values can easily require n to be greater than 1.

For the address portion of the access, a virtual memory environment requires that the virtual address be translated into a physical address. It is the physical address that is ultimately decoded by the memory system. Ideally the address translation should not lengthen the memory transactions. Practically, however, a clock cycle is typically added to the duration of the address portion of a transaction. We will discuss methods for eliminating this address translation overhead later. In an asynchronous or pseudosynchronous protocol it is necessary to indicate the presence (validity) of the physical address so that it can be decoded. This is easily done through a physical address strobe signal that is separate from the virtual address strobe.

Write transaction. Implementing the write transaction involves many of the same concerns as implementing the read transaction. The goal of optimizing memory access time for a zero-wait-state transaction still holds. Further, the need for a data size indication is critical because one cannot allow all 32 bits on the data bus to be written to memory when the data size to be transacted is less than 32 bits. That would corrupt memory that should not be written.

Interlocked transactions. Computers intended to support operating systems and higher-level languages should provide for the implementation of semaphore operations. Most modern microprocessors provide this support via a test and set instruction in which an operand is read from memory, tested, modified, and returned to memory. Note that there is a read and a write transaction for this operand. To enforce mutual exclusion, one must make this test and set operation atomic—that is, indivisible.

When a potential bus master is attempting to gain ownership of the bus, other processors such as the DMA controller or another CPU are excluded until both the read transaction and the write transaction of the test and set operation have been completed. Bus ownership should *never* be given to another bus master between the read and write transactions.

Block fetch. As previously described, address translation can increase the duration of memory transactions by one cycle. This reduces the efficiency of information transfer to and from the CPU and increases the CPU's bus occupancy. This increased bus occupancy in turn increases the contention for a shared bus. One method of reducing the address translation overhead is to fetch more than one datum while only issuing one address. That is, if more than one datum is fetched—say four—only one address translation, not four, needs to be performed when these data are fetched. The key to this reduction in translation overhead lies in the memory system being able to respond with the data even though it is given only one address. The scheme for the return of data can be to return data from consecutive addresses beginning with the address issued, or it can be to return an aligned block of data. The fetched data must not cross page or segment boundaries, or access permissions may be violated and a running process may obtain "illegal" access into the space of other processes. Furthermore, the data from another segment or page may be treated as part of the running process's data, and hence the running process will err.

Another requirement of the block fetch mechanism is that it be dynamically selectable so that hardware incapable of providing multiple words for a single address need not be required to provide multiple pieces of data. If the block fetch capability can be accommodated, the memory system can tell the CPU that multiple pieces of data are available for transfer.

The WE32100 microprocessor uses a two-word block fetch mechanism. This fits very well with its internal instruction cache architecture, which has a two-word block size. All instruction fetches begin as a two-word block fetch. If the memory system is capable of returning two words with only one address, it indicates so to the CPU. The CPU will then expect two instruction words from memory. The two instruction words will be returned to the CPU and placed in a block in the instruction cache. The WE32100 CPU expects the two-word instruction block to be double-word-aligned; that is, it expects that if an even-word address (address bit 2 = 0) is issued, the first word returned will be the even word in the block and the second word returned will be from the odd-word address (address bit 2 = 1). If the address issued is is an odd-word address (address bit 2 = 1), the first word returned from memory will be from the odd-word address and the second word will be from the even address in the block (address bit 2 = 0). This scheme is very easy to implement since the memory system need only recognize a block fetch transaction, return the addressed word, complement an address bit, and return the second word. Since the cache block is aligned, the access cannot cross a segment or page boundary.

The WE32100 CPU begins a block fetch transaction only on instruction fetches.

Interrupt-vector number fetch. An interrupt is a mechanism whereby a device can obtain the attention and the service of a microprocessor. A device causes an interrupt by asserting an interrupt signal or multiple, priority-encoded signals. Some microprocessors (including the WE32100), upon receiving an interrupt, respond with an interrupt-vector number fetch. This transaction is used to fetch a vector number that is used as a pointer into a jump table. This vector number determines which interrupt service code is to be executed. The interrupt-vector number fetch is exactly like a read transaction—since the respondent (i.e., the interrupt controller) is not necessarily memory-mapped, a new status must be introduced to indicate a special transaction. Interrupts typically are not acknowledged until the end of the currently executing instruction. This makes saving the internal state of the microprocessor simpler so that the context of the machine can be restored easily when the interrupt service is completed. In the WE32100 chip set, the address accompanying the interrupt-vector number fetch is ignored by the MMU.

The transactions discussed above—read, write, block fetch, interlocked, and interrupt-vector number fetch—are fundamental to the operation of a 32-bit microprocessor system. The size of the datum to be transacted should be indicated by means of a signal, and the transaction type should also be indicated with a signal. The interlocked transaction makes special demands on the bus arbitration scheme.

The following sections describe how the basic transaction types are changed when special system design requirements must be met.

Exceptions. Since it cannot be guaranteed that every bus transaction can be completed successfully, in every bus protocol specification there must exist a method by which to indicate to the initiator of a transaction that the transaction cannot be completed. For example, if the memory subsystem has detected that a datum being read has been corrupted, the requester must be informed that the transaction cannot be completed in normal fashion. We will use the term "exception" to describe such an abnormal condition.

Bus exceptions can be divided into two types. The first includes those that can be resolved only through software intervention; we call these "fault exceptions." Examples of this type of exception are addressing of nonexisting memory, nonresponse from a subsystem because of hardware failure, and data errors detected by a parity-checking unit.

The second type of bus exception includes those that can be resolved by the hardware in the system; we call these "retry exceptions." An example of this type is the detection of an error by an error detection and correction unit in the memory subsystem. If the error can be corrected, the requester can be asked to retry the transaction. Another ex-

ample of the retry exception occurs in systems in which there is a hierarchy of buses. Consider a situation in which there is a multiple-master backplane bus connecting several boards in a system and in which there are local buses on each of those boards. The following scenario can create a bus deadlock: A master has acquired the backplane bus and is trying to gain access to a resource on a board while the local bus master of that board is starting a transaction requiring access to the backplane bus. A deadlock occurs until one of the masters defers its access to the other. A common way of resolving such a deadlock is to have the requester on the local bus relinquish its control of the bus and then retry its access later.

The WE32100 chip set supports both exception types. The retry exception type can be divided further into exceptions that require the relinquishing of bus mastership and exceptions that do not. The signals for exceptions that do not are called FAULT and RETRY; the signal for exceptions that do is called, appropriately, RELINQUISH AND RETRY. In an asynchronous system, these signals can be asserted at any time in a transaction. In addition, a feature called delayed exceptions allows these signals to be asserted even after a DTACK (data transfer acknowledge) signal has been asserted (that is, up to one half of a cycle later). This allows a possible reduction in the number of wait states by allowing more time for the determination of an exception condition. In a synchronous system, these signals may be asserted synchronously with the CPU's SRDY signal. The SRDY signal in the WE32100 CPU is a synchronous version of the DTACK signal. This is, again, an example of the dual asynchronous/synchronous nature of the WE32100 chip set.

Assertion of FAULT causes the current instruction to be halted and control to be transferred to a different piece of software—the fault handler, as it is commonly called. Assertion of RETRY causes the current transaction to end and to be retried when RETRY is negated. Assertion of RELINQUISH AND RETRY has the same effect as RETRY but, in addition, causes the bus requester to give up bus mastership. The bus requester indicates that it has given up bus mastership by asserting the RELINQUISH AND RETRY ACKNOWLEDGE signal. The deferred transaction is retried when the RELINQUISH AND RETRY signal is negated.

Bus arbitration. In systems in which there can be multiple masters of the bus, there needs to be a method of bus arbitration whereby one of the potential masters is identified as being allowed to initiate a transaction on the bus. Bus arbitration implementations can be divided into two categories—distributed and centralized. In distributed bus arbitration, the decision-making process is performed individually by each master that requires use of the bus. Of course, this process must yield a result that is accepted by all masters. In centralized bus arbitration, all requests for the bus are routed to a controller, which then decides which master gets the bus. The distributed method of arbitration has the potential of having very low performance overhead because the decision-making process is performed in parallel by every master. In the centralized scheme, overhead may be higher because the master to which the bus will be given must wait for an acknowledge signal from the central arbiter before obtaining ownership. Distributed arbitration, however, also has the potential to be costly in terms of on-chip logic because arbitration logic must exist in every bus master.

In a shared-bus architecture, it is very important that bus arbitration be performed efficiently. Otherwise, many bus cycles may be wasted if there is frequent activity by more than one bus master, such as in the cycle stealing mode of direct memory access.

We chose the centralized bus arbitration scheme for the WE32100 chip set. The advantages of this scheme included ease of implementation and compatibility with the previous-generation CPU.

In a centralized bus arbitration scheme, the CPU is typically, although not always, the arbiter. For example, a chip set can be designed in which the CPU must request bus mastership from an external bus controller instead of being the default master and granting the bus whenever another device requests it.

The sequence of activity in a centralized arbitration scheme is as follows: A device sends a request to the arbiter requesting bus mastership. The arbiter grants the request, after arbitrating among the requests it has received according to request priority or some other request attribute. The device performs transactions on the bus and then relinquishes control of it. The arbiter then grants the next requesting device bus mastership. There are two ways in which to implement this series of steps—the "two-wire" arbitration scheme and the "three-wire" arbitration scheme.

In the two-wire scheme, there are two signals (hence the name) used to signal the requesting and the granting of bus mastership. These signals are called BUS REQUEST and BUS GRANT. In the two-wire scheme, the activity outlined above proceeds as follows: The device requesting bus mastership asserts the BUS REQUEST signal. Bus mastership is granted after the current master relinquishes the bus, and the mastership is signaled by the assertion of BUS GRANT to the requesting device. The requesting device then performs bus transactions. When it is done, it gives up mastership by negating the BUS REQUEST signal. The bus arbiter then negates the BUS GRANT signal to that device.

In the three-wire arbitration scheme, there are three signals—BUS REQUEST, BUS REQUEST ACKNOWLEDGE, and BUS GRANT. In this scheme, the sequence of steps is as follows: The requester asserts BUS REQUEST to the arbiter. The arbiter then asserts BUS REQUEST ACKNOWLEDGE to acknowledge the receipt of the request. Next, the requester negates BUS REQUEST, and it then waits for the arbiter to assert BUS GRANT, whereupon it takes control of the bus. This scheme makes it possible to hide the arbitration time behind an ongoing access.

We chose the two-wire scheme for the WE32100 chip set, primarily because of its compatibility with the scheme used in the WE32100's predecessor. In the two-wire scheme,

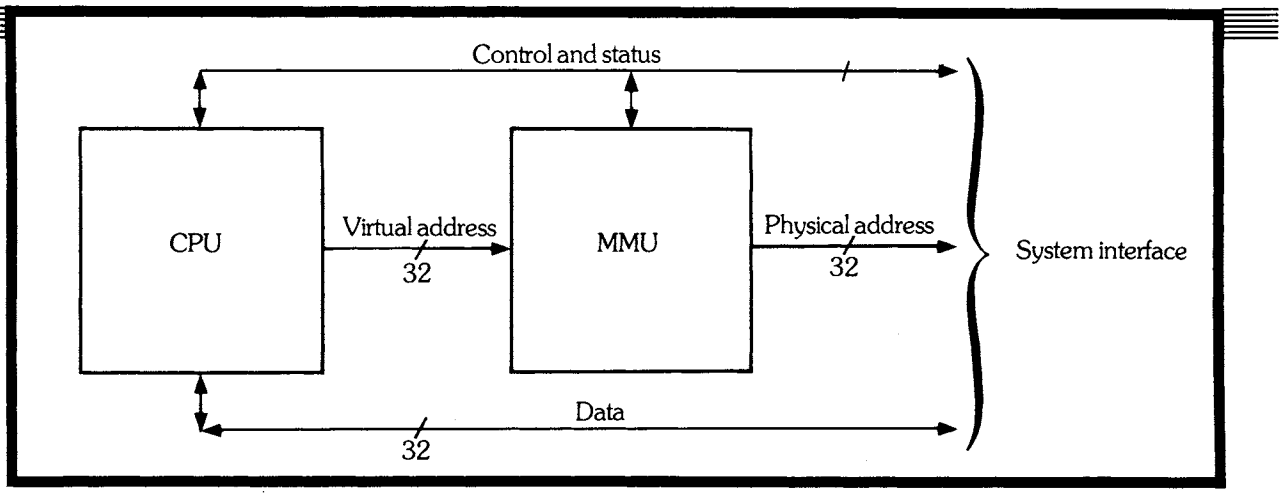

Figure 1. CPU-MMU architecture with a split address bus.

there are two modes of operation for the CPU—it either is or is not the default master. If it is the default master, the CPU is the master of the bus if no other device is requesting it. If a requester asserts BUS REQUEST, the CPU gives up bus mastership and asserts BUS REQUEST ACKNOWLEDGE. The CPU regains bus mastership when the requester negates BUS REQUEST, after it (the CPU) first negates BUS REQUEST ACKNOWLEDGE. If the CPU is not the default master, it must request the bus every time it wants to perform a bus transaction; i.e., it must assert BUS REQUEST, which is now an output, and must wait for a BUS REQUEST ACKNOWLEDGE, which is now an input, before it can start a transaction. It negates BUS REQUEST when it completes its transaction.

Interface to the MMU. Virtual memory is very important to 32-bit systems. Software programs are growing larger than ever. Multiprogramming environments surely need virtual memory and the attendant capability for memory protection it provides. There are many different virtual memory schemes and architectures. Since the focus of this article is the hardware interfaces of VLSI chips, we shall focus on methods of interfacing "generic" memory management hardware to a VLSI microprocessor.

The question to be answered is, "Where should the memory management hardware reside?" That is, "Should the memory management hardware be placed on the CPU or should it reside on separate devices?" As previously mentioned, memory management (i.e., address translation) can extend the duration of the address portion of an access. This assumes that the translation is performed by hardware not on the CPU. If memory management hardware exists on the CPU, there is no need to issue a virtual address. Hence, all accesses (from the external system's point of view) are saved the translation, and memory management overhead disappears.

It is a truth in VLSI architecture that off-chip communications are a performance-limiting constraint. On-chip communications, though they are certainly becoming a VLSI scaling problem, are nevertheless much faster than communications to external devices. Since the translation aspect of memory management is involved with each transaction to memory, it would seem that interchip communication overhead associated with translation could be avoided by placing the memory management hardware on the CPU. The problem with this lies in the method by which translation is performed.

Let us examine the case of minicomputers, since they have memory management hardware. Minicomputers typically maintain tables of descriptors corresponding to fractions of the virtual address space of their CPU. When the associated fraction of virtual memory is accessed, a descriptor is used to map the virtual address to a physical address. The descriptors are typically located in very fast memory near the CPU—a translation buffer—so that the lookup of the relevant descriptor and the accompanying translation can be done very fast. This translation buffer contains enough descriptors to describe the entire virtual address space of the CPU. Usually this implies a large (128-entry or more) translation buffer. However, VLSI architectures are critically constrained by chip area. (The general wisdom is that chip cost is proportional to the cube of chip area.) One can clearly see that large translation buffers will increase chip area and thus chip cost. Consequently, the area constraint makes off-chip memory management attractive.

Off-chip memory management. As mentioned above, interchip communication is a performance-limiting factor in today's VLSI architectures. Indeed, an off-chip memory management unit will add overhead to memory transactions. With luck, this overhead will be only one clock cycle.

An off-chip MMU for a 32-bit architecture is shown in Figure 1. In this arrangement, the address bus is split between the CPU and memory. This suggests that the MMU is dual-ported and hence requires a large number of I/O signals. The number of I/O signals is also an important constraint on VLSI chips. To minimize chip area, one should limit I/Os to a reasonable number. We have previously described the advantages of nonmultiplexed address and data buses, for which we accepted 64 I/Os for perfor-

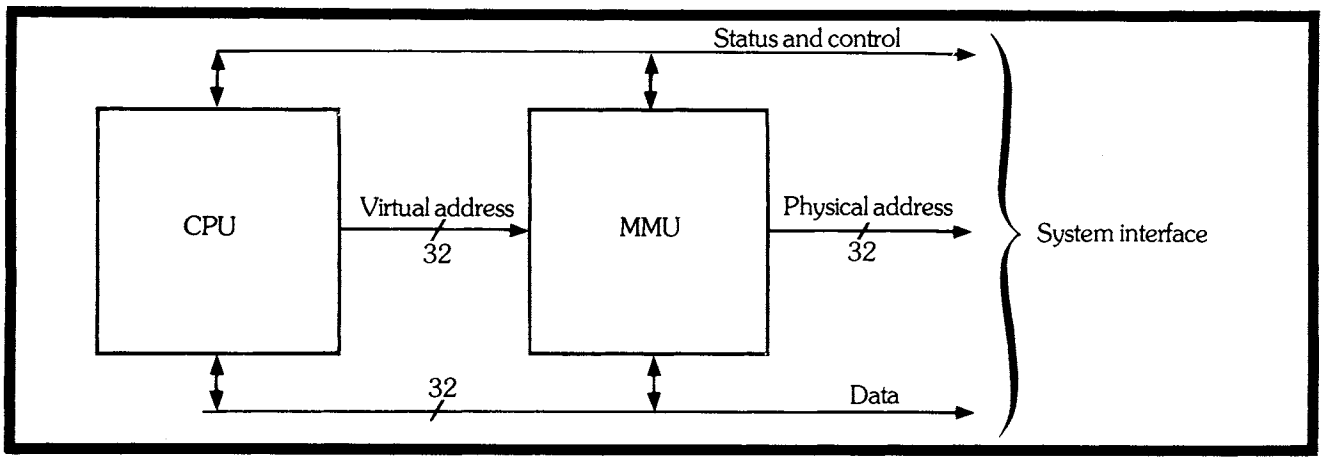

Figure 2. CPU-MMU architecture with a split address bus and a shared data bus.

mance reasons. Hence it is not unreasonable to postulate 64 I/Os for the MMU address bus (32 virtual and 32 physical addresses), though the physical address need not be a full 32 bits.

The splitting of the address bus has the potential for eliminating address translation overhead despite the interchip communication problem. This potential can be realized by pipelining (overlapping) accesses to memory. That is, if the address portion of the $n+1$th transaction can be made to overlap the data portion of the nth transaction, the virtual address for the $n+1$th access can be sent to the MMU while the nth physical address is being used to access memory. This overlap hides the interchip communication on the virtual address bus (see again Figure 1). Of course this overlap must be implemented as part of the CPU's internal architecture as well as part of the I/O.

The $n+1$th address must be available while the nth access is pending. As noted previously, this is not an easy policy to enforce on the CPU, especially a CPU with a complex instruction set containing instructions that take more than one cycle to execute. So now we must ask, "Is the architecture shown in Figure 1 complete?" The answer to this question is no. What is not shown is a mechanism to load descriptors into the MMU. Figure 2 shows an architecture with such a mechanism. In this architecture, a data bus has been added to the MMU for the purpose of loading the descriptors into the MMU. This loading of descriptors can be done by the CPU (by means of write transactions) or by the MMU itself (by reading from the memory system). Note that adding the data bus to the MMU I/O increases the number of I/O signals by 32. (Interchip control signals are also needed; they are not shown in Figure 2.)

Adding 32 I/O signals to the MMU is abhorrent even if one neglects signals needed for control. Adding 32 MMU I/O signals to the architecture shown in Figure 2 will increase the area and cost of the MMU significantly and may even make it unrealizable. And with control signals added, the number of I/O signals may be more than can be put on the chip periphery. However, the architecture of Figure 2 is highly desirable, particularly if address overlap can be obtained. But it is pinout-intensive and not cost-effective given current VLSI technology.

An alternative is shown in Figure 3. In this architecture the physical address bus is multiplexed with the data bus. The physical addresses and the data are demultiplexed via latches and bidirectional tristatable buffers. Control signals manage these buffers and latches. The multiplexing of the MMU's address and data buses greatly decreases the number of I/O signals on the MMU compared to the architecture of Figure 2. This makes implementation of the architecture in VLSI easier but creates a need for SSI glue to interface the VLSI components.

Another architecture using shared address and data buses is shown in Figure 4. This architecture is pin-efficient and requires no SSI glue. However, it does not lend itself to address overlap as do the previously described architectures. It suffers perhaps more from the overhead of off-chip communications than do the other architectures. This overhead occurs in a critical path, namely, the address path.

In Figure 4, the CPU must generate a valid virtual address—which can take a relatively long time—and the MMU must latch it. Once the virtual address is latched, the CPU must remove it from the address bus and put its outputs into a high-impedance state so that the MMU can use the address bus for the physical address. If the interface between the two VLSI chips is asynchronous, the CPU must be told by the MMU to remove the virtual address from the address bus. This two-way communication required by asynchronous operations further increases the address translation overhead. If the interface between the two VLSI chips is synchronous, the CPU can remove the virtual address from the address bus at a prearranged time conveyed by the common clock. Thus, a synchronous interface is preferable to an asynchronous one because it eliminates a communication step. We adopted the architecture shown in Figure 4—with the synchronous interface—for the WE32100 CPU and MMU.

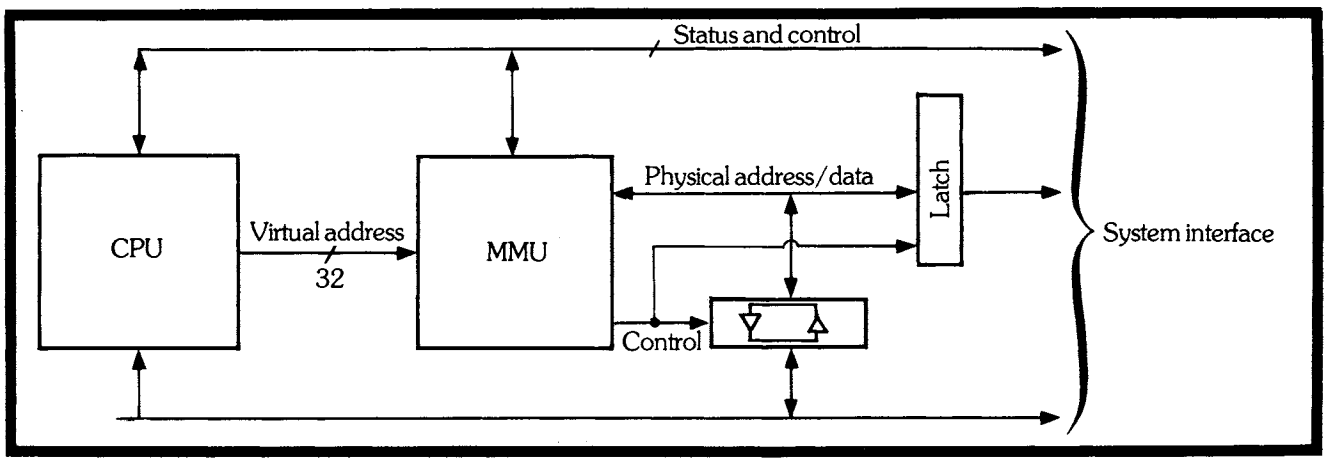

Figure 3. CPU-MMU architecture in which physical addresses and data are multiplexed on the same bus.

The next issue to be considered is the method by which the MMU removes its physical address from the bus when a transaction has been completed so that subsequent transactions can be initiated by the CPU. Since the CPU is the master of the bus, with the MMU providing surrogate address and control information, the CPU can, upon a data transfer acknowledge, inform the MMU that it can remove the physical address. If the interface between the CPU and the memory system is synchronous, the MMU can also look at the data transfer acknowledge to know when to remove the physical address. Thus, again, a synchronous interface saves one communication path. If the interface between the CPU and the memory system is asynchronous, the two devices cannot sample an asynchronously asserted signal and guarantee that they will both latch the same value. They may latch different values if the sampled asynchronous signal is in transition between logic levels at the sampling instant. Because the two devices cannot guarantee that they will both latch the same value, the CPU must relay the asynchronous data transfer acknowledge to the MMU, causing extra interchip communication overhead. A synchronous data transfer acknowledge, on the other hand, can be sampled by both the CPU and MMU to save this overhead.

MMU as a bus master. As previously mentioned, the MMU requires a fairly large number of descriptors to be stored in a fast, nearby memory. Having fast access to the descriptors implies that they should be stored on board the memory management chip so that they will not need to be fetched from off the chip for each memory transaction. The relevant issue here is whether the MMU can store the full complement of descriptors. As we noted earlier, minicomputers typically have large translation buffers. The full complement of descriptors needed by an MMU is unlikely to fit onto one piece of VLSI silicon.

One solution is storing a subset of the descriptors in on-chip caches. On-chip descriptor caches make use of tem-

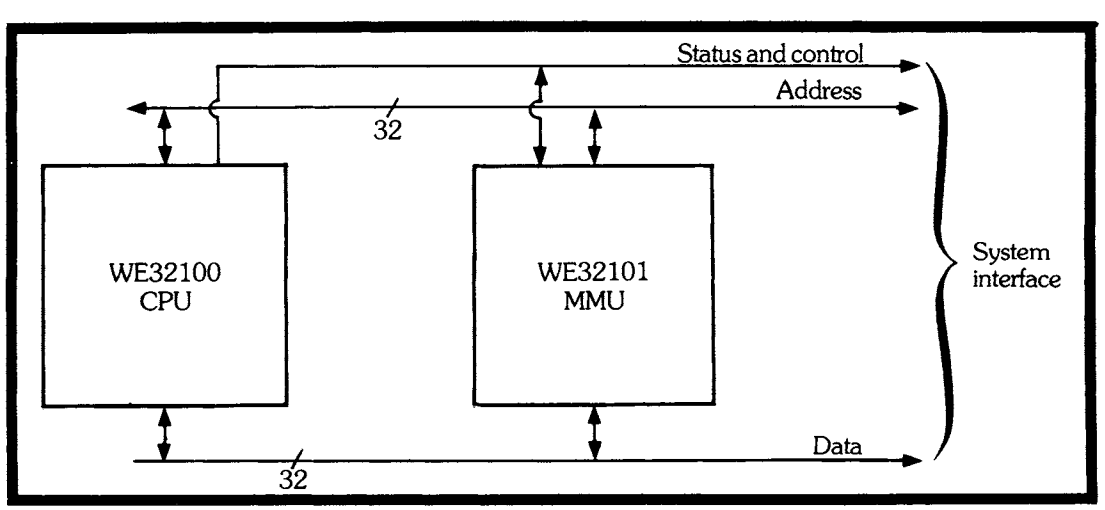

Figure 4. CPU-MMU architecture with shared address and data buses.

poral program locality to achieve high hit rates. If enough descriptors can be cached to map the working set of the running process, a high hit rate and correspondingly high performance will be obtained. For cache misses, the MMU can easily fetch the descriptors from memory and relieve the operating system from that chore. The effectiveness of miss processing is critical to performance even at high hit rates. To perform this processing, the MMU must be able to preempt an ongoing memory transaction, fetch the descriptors, and resume the transaction after address translation. The key requirement here is to have a mechanism in the CPU by which the transaction can be preempted and the control of the bus relinquished—something similar to the RELINQUISH AND RETRY exception. The normal bus arbitration scheme is inappropriate because bus ownership is not surrendered by the CPU until after the transaction has completed. That is, bus arbitration is not preemptive. For uniformity of system design, MMU-initiated memory transactions should be the same as CPU-initiated transactions.

The WE32101 MMU can preempt an ongoing memory transaction so that it can fetch descriptors. While the MMU has control of the buses, exceptions can occur. The WE32101 MMU has a fault code register that captures the exception and then passes the exception to the CPU. By reading the fault code register in the MMU, the exception handling mechanism can diagnose the cause and effect the correction of the exception condition. The full complement of fault conditions captured in the WE32101 makes for easier debugging of system code. [5]

Another function associated with virtual memory is protection. In a typical protection scheme, permissions to access memory are checked by the memory manager before the transaction to the memory system is allowed to complete. Access permissions are typically part of the mapping descriptors. [10] The MMU must be provided with a mechanism to preempt the ongoing transaction by asserting exception signals, usually the FAULT signal. It must also have a mechanism for indicating page- and segment-not-present conditions (page faults). Further, when the MMU is bus master it must be capable of accepting faults and conveying them to the CPU. The memory management approach used in the WE32100 chip set is described by Lu, Dietrich, and Fuccio. [5] The hardware architecture is that shown in Figure 4. The CPU-MMU interaction is synchronous while the memory interface is pseudosynchronous. The CPU's address strobe is used to initiate translation, and its negation is used to remove the MMU (the WE32101) from the bus.

Support processor interface. For our purposes, we will define a support processor as a processor that is adjunct to the CPU, one for which the CPU waits before continuing. Support processors are important because they provide increased system performance by executing operations that the CPU may not be able to handle efficiently. The classic example is floating-point operations. By using specialized hardware—a support processor—to perform these operations, one can obtain a dramatic performance improvement over a software implementation. The protocol used to interface a support processor to a system must be efficient to reduce the overhead arising from employing the support processor.

Many trade-offs are involved in developing an efficient interface. The issues are numerous and complex. We considered three ways of implementing the WE32100 support processor interface. However, before we discuss the advantages and disadvantages of each, let us examine the four steps involved in the operation of a support processor. These steps are common to all support processor protocols.

The first step is to identify which one of several possible support processors will handle the current instruction and to pass that instruction to it. The second step is to send the operands needed to execute the instruction to the support processor. The third step is to identify any errors that may have occurred during the operation. The fourth step is to store the results of the operation in a location outside the support processor, if necessary.

One method of interfacing to a support processor requires virtually no support from the CPU at all. What is required is for the support processor to monitor the CPU instruction stream as it is fetched by the CPU over the bus. When the CPU fetches a support processor instruction, it suspends operation. The support processor only executes those support processor instructions intended for itself. When the support processor is done, the CPU resumes execution. While seemingly attractive, this option is not very feasible with today's microprocessors. Pipelining, instruction prefetching, and on-chip caches make it very difficult, if not impossible, for a support processor to monitor the CPU's instruction stream without duplicating most of the CPU's internals.

This difficulty, then, leads to options in which the CPU must provide information to the support processors when it encounters a support processor instruction. The next two options we will discuss are basically the same in the way they perform steps 1 and 3 but different in the way they perform steps 2 and 4. With these two options, when the CPU reaches a support processor instruction, it must wake up a support processor to execute the required operation. It can do this by means of a broadcast transaction over the bus. The support processor signals the CPU when it is done and passes status information back to the CPU.

What we now must discuss is how operands get to, and results are removed from, the support processor. For operands located in memory, one method is to have the CPU fetch them into itself and then transfer them to the support processor. In this case only the CPU is involved in the access to memory; hence, it is the only device concerned with bus phenomena such as exceptions. The second method is to have the CPU perform the memory access and to have the support processor monitor the bus and "catch" the data when it returns from memory. Here both the support processor and the CPU must be cognizant of bus phenomena. Similarly, in order to store results back into mem-

ory, the CPU can first transfer them from the support processor into itself and then write them to memory. Alternatively, the CPU can perform a write to memory with the support processor, again monitoring the bus and supplying the necessary data.

The first method has the advantage of allowing, but not requiring, the CPU and the support processor to be synchronized. Synchronization is needed when the support processor must reliably track CPU memory transactions. Synchronization also gives the CPU an opportunity to massage the data before it passes it to the support processor. This is useful if, for instance, the data is misaligned and the support processor expects aligned data. The CPU can do the alignment and an alignment capability does not have to be provided in the support processor(s). The second method provides the advantage of fast transfers of operands and results from and to memory. (For operands and results coming from or going to the CPU, no advantage accrues from one method or the other, since there is only one method for doing such transfers.)

In the WE32100 we chose to minimize the overhead associated with the transferring of operands to the support processor even though it meant increasing the complexity of the interface and requiring synchronous operation between the CPU and the support processor. We will now examine the operation of the WE32100 support processor interface in detail.

The first step the CPU performs upon encountering a support processor instruction is to perform a support processor ID broadcast. Only the support processor whose ID matches the ID field of the data on the data bus will acknowledge this transaction. If no device acknowledges, the CPU will take an internally generated exception. Note that a support processor must be able to abort its current operation upon seeing this transaction. The reason for this will become evident below.

Once the support processor has received a command, it is ready to receive memory-based operands, if any are required. These operands are fetched by the CPU via a normal read transaction. The status lines indicate a support processor operand transaction so that the support processor will recognize that the data fetched is an operand for the support processor. The support processor samples data on the same clock edge as does the CPU. The support processor runs in synchrony with the CPU so that it can latch the data reliably. The negation of the data strobe and the assertion of the data ready signal indicate to the support processor that the transaction has completed and that the data is valid, respectively. The data will be ignored if there was an exception during the transaction—that is, if the data ready signal was not asserted. Note that the support processor does not know what type of exception occurred. (Two devices, even if synchronous, cannot be guaranteed to see the same value of a sampled asynchronous signal. Since the CPU can receive asynchronous acknowledges and bus exceptions, the support processor cannot reliably monitor these signals and remain in lockstep with the CPU during the memory transaction.) The support processor therefore assumes that the transaction can be retried and is ready to receive the data during the retried transaction. However, if the transaction is faulted, the CPU restarts the support processor instruction from the beginning. Remember that the support processor will abort any ongoing operation whenever the support processor command broadcast transaction occurs and that therefore the support processor will also restart the instruction.

Once the support processor has received all the necessary operands, it performs the given command. Upon completing it, it asserts the done signal to the CPU. This indicates to the CPU that the support processor has finished the command and is ready to pass status back to the CPU. The CPU initiates a read transaction with a status line indicating a support processor status read, whereupon the support processor either places the status on the data bus or asserts the fault signal if an internal exception was made during the execution of the command.

Finally, if there were no exceptions during command execution, the results may be stored in memory. The CPU initiates the usual write transaction, with the following exceptions: The status lines indicate a support processor data write, and the support processor, not the CPU, provides data on the data bus. Again, the negation of the data strobe and the assertion of the data ready signal indicate to the support processor the successful completion of the transaction. If there were bus exceptions, i.e., if the data ready signal was not asserted, the support processor will be ready to present the same data again on the next occurrence of a support processor data write transaction. Again, as was the case with the operand fetch transaction, the support processor assumes the transaction will be retried. If it was faulted, the CPU will restart the support processor instruction. Note that this implies that if a result is to be stored in memory, the support processor must not permanently change its internal state until the result has been stored successfully.

Memory designs and the WE32103 dynamic RAM controller. Thus far, we have discussed issues relating to the hardware architecture of general-purpose VLSI computer systems. We have dealt with matters mostly under the control of the architect of the VLSI chips. System design aspects that are not always under control of the VLSI architect must be considered and, in fact, presupposed. It is impossible to take into account all possible VLSI system designs. One aspect not discussed thus far but critical to all system designs is the memory system design. We have made the assertion that address, data, data size, status, and appropriate strobe signals are all that are needed to provide a memory interface. This assertion is based primarily on an evaluation of commonly available semiconductor static RAMs. Rather than requiring new memory system designs, the WE32100 chip set provides an I/O protocol compatible with three common memory types:

- static RAM,
- dynamic RAM, and
- static RAM cache memory.

The pseudosynchronous protocol of the WE32100 CPU easily accommodates a design based on static RAM. The assertion concerning which signals are needed for a memory interface is correct for most common static memory designs.

Dynamic memory, however, involves a different set of design concerns. Unlike static memory, dynamic RAM must be refreshed periodically to preserve the contents of its memory cells. Furthermore, today's dynamic memory chips commonly employ a multiplexed addressing scheme in which half of the address selects a row of the semiconductor dynamic memory array and the other half a column of the array. In the WE32100 architecture, we parceled out this functionality to a separate VLSI component in the chip set. The WE32103 dynamic RAM controller provides the refreshing and address multiplexing needed by dynamic memories. Implementing dynamic memory control on a separate VLSI chip was justifiable because we couldn't predict how a system designer would design his memory system. A separate VLSI dynamic RAM controller provides many design options within the area and I/O constraints of VLSI.

One design option that can be considered in a VLSI RAM controller is two-ported memory architecture. A two-ported memory system is one in which there are two separate buses providing access into the memory system. Two-ported memory is different from dual-ported memory, and the two should not be confused. In a two-ported memory system, two bus masters, each on its own bus, vie for access to the memory array. An arbiter must decide which bus master will win access. In a dual-ported memory system, the two bus masters concurrently obtain access to the memory system. Only when both bus masters access the same memory address will arbitration take place. In other words, in a two-ported memory arbitration is required when two bus masters concurrently seek access to the entire memory subsystem, whereas in a dual-ported memory it is required only when two bus masters contend for a single, specific memory location. Logically there is little difference between these two schemes, but they are quite different in implementation.

A dynamic RAM controller is a logical place to have arbitration logic for a two-ported memory. The WE32103 dynamic RAM controller can be configured with a second DRAM controller to support two-ported memory (Figure 5). In this figure, the two dynamic RAM controllers cooperate with each other to determine which of any contending bus masters (buses A and B) will be permitted access to the memory array. One DRAM controller is the master while the other is a slave. This distinction serves to determine which DRAM controller will refresh the memory.

The WE32103 DRAM controller supports the WE32100 block fetch transaction by generating a second address for the DRAM array. (Recall that the CPU expects two words of data but only issues one address.)

The WE32103 DRAM controller makes use of the WE32101 MMU architecture to eliminate a cycle on typical DRAM accesses. This feature, called "early RAS," makes use of the fact that the MMU, when performing paged translations, leaves the lower 11 bits of the virtual address untranslated. Hence, the DRAM controller can use these bits as a row address and can begin a DRAM access before the physical address is available, thus saving a CPU clock cycle. "Early RAS" employs synchronism and therefore the CPU, MMU, MAU, and DRAM controller must share a clock to use it.

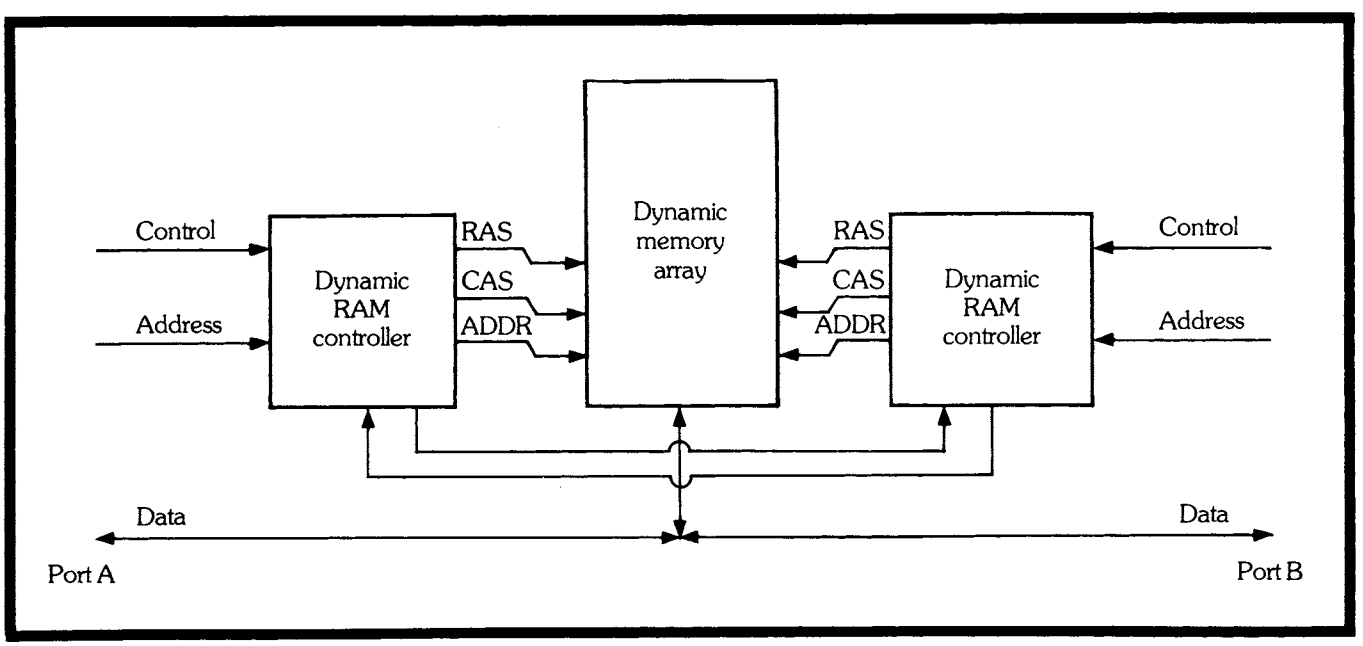

Figure 5. Two-ported memory with WE32103 dynamic RAM controllers.

Because the refresh behavior of the dynamic memory system can critically affect total system performance, the WE32103 provides the system designer with six different refresh timing modes. Thus, he can choose the refresh timing that yields the best performance. The WE32103 also allows the memory access time to be configured to produce the most efficient memory interface. The WE32103 DRAM controller supports high-density dynamic RAMs, including 1M-bit devices. This support naturally leads to architectural support for memory error detection and correction. Today's wisdom is that the higher densities found on dynamic RAMs make them more sensitive to soft errors and other failure modes. Hence, systems that must be reliable and fault tolerant will require memory error detection and correction. The WE32103 DRAM controller provides an interface for an error detection and correction capability.

Static RAM caches are the third type of memory supported by the WE32100 chip set. The goal of cache memory architectures is to match the speed of the memory system to the speed of the central processor.[11-14] By designing a small, high-speed, typically static, RAM cache near the central processor, the system architect can take advantage of program locality to improve overall system performance provided a high cache-hit ratio is obtained.

When we say that memory speed and processor speed are matched, we mean that the central processor has a zero-wait-state access time. To make cache design easier, one should provide facilities to allow a system designer time to access the cache memory and report exceptions. The WE32100 chip set supports a synchronous data transfer acknowledge as well as synchronous bus exceptions to provide more time for the cache memory transaction to complete. Approximately one half of a cycle of extra time is made available to the system designer to generate these synchronous signals, provided he meets their setup time. One such signal is SRDY on the WE32100 CPU. Asynchronous signals require no setup time; however, they must be valid approximately one half of a cycle earlier than the synchronous signals if they are to be recognized without an additional wait cycle having to be inserted in the transaction.

Another feature of the hardware architecture supporting cache memory designs is an indication from the memory manager that the contents of a segment are cacheable or noncacheable. This feature allows an operating system to designate certain segments as noncacheable. A noncacheable segment may be an I/O buffer, which is read or written only once during the lifetime of a process. Rather than throwing program text and data out of the cache in order to cache the contents of an I/O buffer, one may find it performance-improving to leave current, local data in the cache and not overwrite it with less valuable (less local) data. The WE32101 memory management unit provides a signal to represent the state of a segment descriptor's cacheable bit, to assist the cache designer in controlling the contents of the data cache. Using this signal, the system designer can mark segments as noncacheable.

Shared bus architecture and DMA. The benefits of direct memory access channels in computer architectures has been demonstrated repeatedly. The drawback to DMA in a shared bus architecture is that the CPU can be denied access to the bus while the DMA is ongoing. Cycle stealing is the normal mode of DMA transfer in a shared bus architecture. In cycle stealing, the DMA controller "steals" bus time between CPU bus transactions. If the bus occupancy of the central processor is low, such bus cycle stealing generally will not grossly inhibit central processing.

One method of reducing bus conflict between the CPU and DMA controller is to give the DMA controller very high performance. If the DMA is performed rapidly, there is less time for bus conflict. This argues for a full 32-bit DMA capability. Indeed, a DMA transfer over a 32-bit bus will require roughly half the number of bus transactions as a 16-bit-wide DMA transfer. Certainly, then, 32-bit systems should have high-performance, 32-bit-wide DMA channels. Such channels are ideal for memory-to-memory transfers because the memory in a 32-bit system is 32 bits wide. The question then arises, "What about fast I/O transfers when the I/O port is less than 32 bits wide?" Such is the case with common byte-wide peripherals such as UARTs and LAN controllers. It seems reasonable to remove these eight-bit peripherals from the 32-bit bus because they do not require its full bandwidth but yet contend for its use. To relieve the high-performance 32-bit system bus from the overhead associated with low-performance peripherals, the WE32104 DMA controller provides a separate eight-bit-wide peripheral bus.

It performs memory-to-memory transfers on the 32-bit-wide system bus. Peripheral-to-memory and memory-to-peripheral transfers can also be performed. On such transfers, the DMA controller buffers and packs or unpacks bytes, allowing better utilization of the system bus.

The WE32104 DMA controller has four independent, programmable DMA channels. Each channel can have its access to the system and peripheral buses prioritized. The DMA controller allows various arbitration schemes so that the performance of the four channels can be optimized for a particular system's DMA characteristics.

The DMA controller performs transactions in one of two modes, either in burst mode or in cycle-steal mode. As mentioned before, cycle stealing is the DMA operation in which the DMA controller takes possession of the shared bus (the system bus in the WE32100 chip set) to perform a small number of DMA transactions. In the burst mode the DMA controller takes possession of the shared bus for a long time and performs many transactions before relinquishing it. The burst mode of DMA transfer has a direct effect on the system's response to interrupts. Indeed, while the DMA controller has possession of the bus, the CPU cannot service a pending interrupt. This situation can be highly undesirable in some systems. The WE32104 DMA controller therefore possesses a signal that can be used to preempt the burst transfer and cause the remainder of the transfer to be done in the cycle-steal fashion. When this is

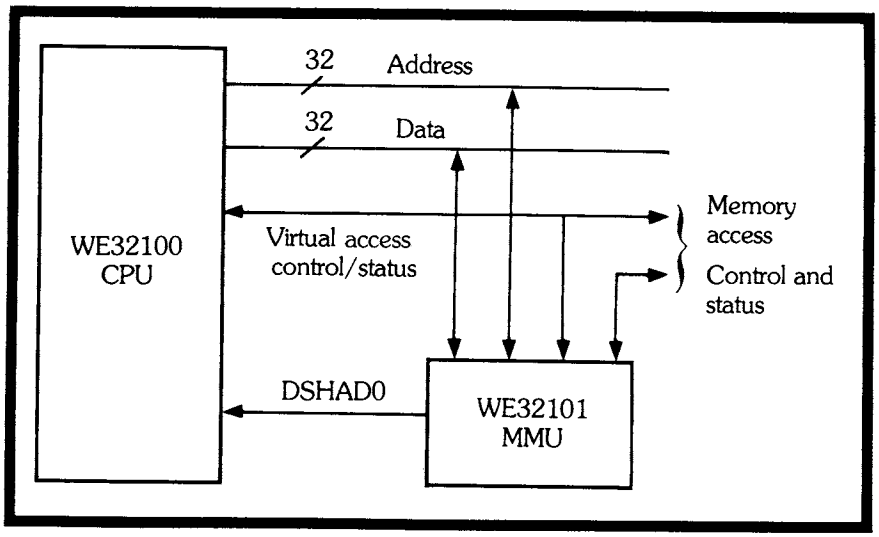

Figure 6. The WE32100 CPU configured with the WE32101 MMU.

done, the DMA controller relinquishes the bus and the pending interrupt can be serviced.

In a shared bus architecture, all bus transactions should look the same to the system. The WE32100 CPU, WE32101 MMU, and WE32104 DMA controller have fundamentally identical bus transactions. Like the CPU, the DMA controller can perform two-word block transactions. In addition, the DMA controller has a quad-word bus transaction that is a logical extension of the two-word block transaction. (Recall that block transactions are selectable.) Maximum DMA performance for memory-to-memory transfers occurs when the DMA controller performs quad-word transactions in burst mode.

The WE32104 DMA controller has a command chaining feature. It can fetch a list of linked DMA commands from memory. The CPU can set up many DMA operations by means of this linked-command list.

The CPU needs access to the configuration registers internal to the DMA controller; hence, the DMA controller can be viewed as a peripheral. The WE32104 DMA controller provides a "no glue" peripheral interface—one that adheres to the shared bus protocol—to the WE32100 CPU.

Special considerations in the design of the WE32100 architecture. When the hardware architecture of a chip set is defined, the most important consideration is consistency. Each member of the chip set must support the hardware and software features of the other members. In the WE32100 chip set, a good example is the block fetch feature mentioned previously. The idea behind this feature is to fill the two-word block in the CPU's instruction cache efficiently. The block fetch also makes it possible to avoid an address translation, provided the two instruction words do not straddle a memory segment boundary. In the MMU, the segment resolution is double-word, so the MMU's software architecture permits a double-word block fetch, saving an address translation. This alone is not enough, however, since most dynamic RAM designs cannot easily support block fetching. Hence, support for it had to be added to the WE32103 DRAM controller.

Another example of how the software architecture affects the hardware architecture is the MOVTRW (move translated word) instruction in the CPU. With this instruction, the CPU issues a virtual address and expects the corresponding physical address to be returned to the destination location. This requires a special arrangement with the MMU. In this operation, the CPU begins a normal read transaction while issuing a special MOVTRW status. The MMU responds by placing the translated address on the data bus rather than on the address bus. The MMU then signals the termination of the transaction.

Another special demand placed on the WE32100 chip set is that it support many chip set configurations. Many of the architectural decisions made in the course of WE32100 development sprang from the desire to attain this goal.

The WE32100 chip set can be configured in many meaningful ways. The CPU can stand alone or it can be the center of a multichip solution. Furthermore, a simple multiprocessor capability on the CPU allows it to share a bus with other CPUs as well as with peripheral devices.

In an architecture requiring memory management, a system designer can use the WE32101 MMU or build his own MMU. To be sure, the WE32101 MMU has a flexible architecture, and for unusual cases more than one WE32101 can be used to enhance the size of the translation buffer. Such flexibility in MMU design is not afforded by on-chip memory management units.

The WE32106 MAU dramatically improves floating-point arithmetic. It is optimally configured as a support processor, although multiple MAUs can be configured as memory-mapped peripherals. The WE32104 DMA controller gives the system designer the capability to move data at very high rates.

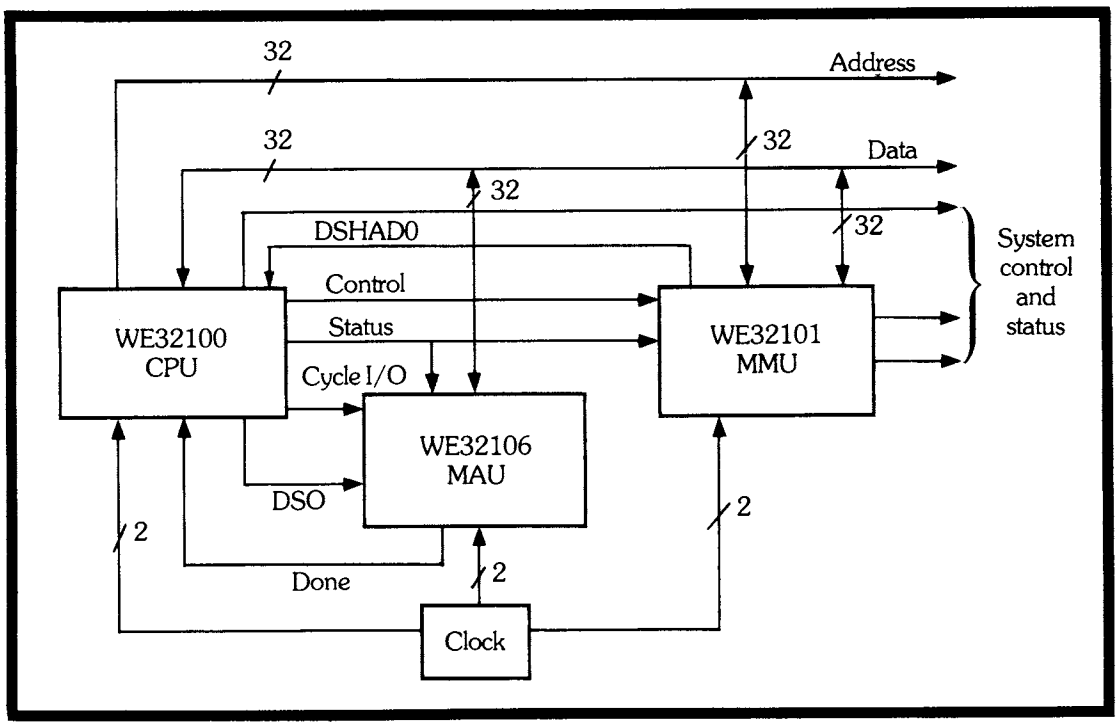

Figure 7. The WE32100 CPU configured with the WE32106 MAU and WE32101 MMU.

We should mention a member of the chip set we have not discussed in detail here—the WE32105 system interface controller, or SIC. This chip is a compatibility bridge between first-generation (WE32000) and second-generation (WE32100) microprocessors. It performs status decoding and byte-strobe and wait-state generation for a system designer. It runs in synchrony with the WE32100 CPU, MMU, and MAU but provides an asynchronous interface to the system designer. This chip had a profound influence on the chip set architecture if for no other reason than it allowed many additional chip set configurations to be supported in a consistent fashion.

One of the most useful attributes of the WE32100 chip set is that no configuration requires SSI glue logic. This "no glue" solution provides high functional density and saves precious board space.

Some WE32100 configurations are shown in Figures 6 through 9. Figure 6 shows a CPU with an MMU. The salient feature of the CPU-MMU configuration is shared address and data buses. The MMU becomes a bus master in its own right when it performs miss processing to fill its internal translation cache. The MMU translation overhead is one cycle, hence a native three-cycle access becomes a four-cycle access with virtual to physical address translation. The MMU uses the DSHAD0 signal to preempt an ongoing CPU access when it is performing miss processing.

Figure 7 shows a CPU with an MMU and MAU. The most notable feature of this configuration is, again, the shared, nonmultiplexed address and data buses. The math accelerator unit, when configured as a support processor, takes data directly off the data bus. Floating-point operand addresses are generated by the CPU.

Figure 8 shows a CPU with an MMU, MAU, DMA controller, and DRAM controller. This configuration provides shared address and data buses but also provides a separate peripheral bus for the DMA controller. This peripheral bus is ideally suited to the character-oriented I/O needed for devices such as UARTs and LAN interfaces. Figure 8 also shows that multiple MMUs can provide a larger translation buffer, if one is required, and that the DRAM controller can control multiple banks of dynamic RAM.

Figure 9 shows a CPU with a system interface controller. The SIC provides many common functions, such as byte-write strobes and wait-state generation circuits, that are usually implemented with SSI logic. The WE32100 chip set can be configured with or without the SIC chip.

Electrical design. An important concern of the VLSI architect is the electrical characteristics of the chips he is designing. VLSI chips must be able to accommodate a variety of system designs. This means they must support the capacitive load of other integrated circuits in a system. However, supporting a large capacitive load requires the chips to have powerful (hence big) I/O buffers. Large capacitance also tends to greatly slow down communications between members of a chip set. A trade-off between the capacitive load to be supported and I/O performance must be effected.

The "no glue" goal of the WE32100 chip set meant that its members had to minimally support a capacitive load of 130 pF. This allowed an operating frequency of 18 MHz to

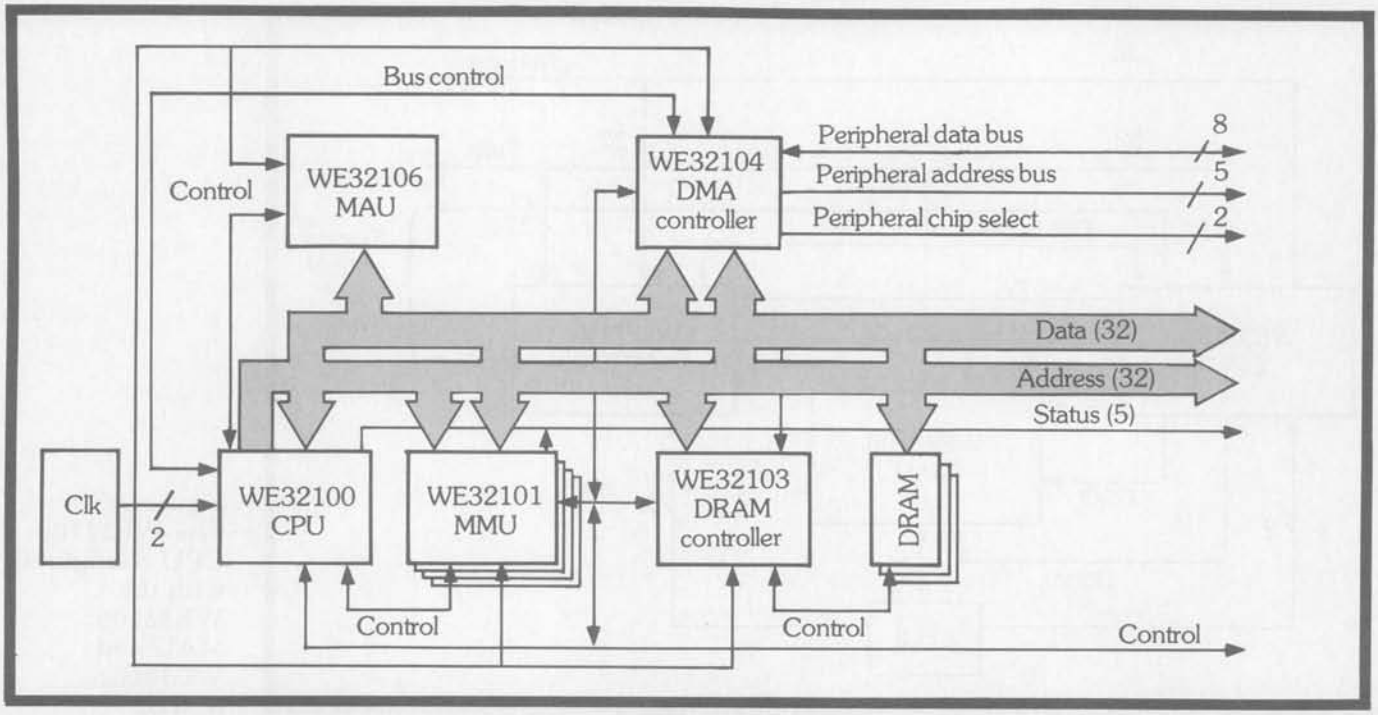

Figure 8. The WE32100 CPU configured with the MAU, MMU, WE32104 DMA controller, and WE32103 dynamic RAM controller.

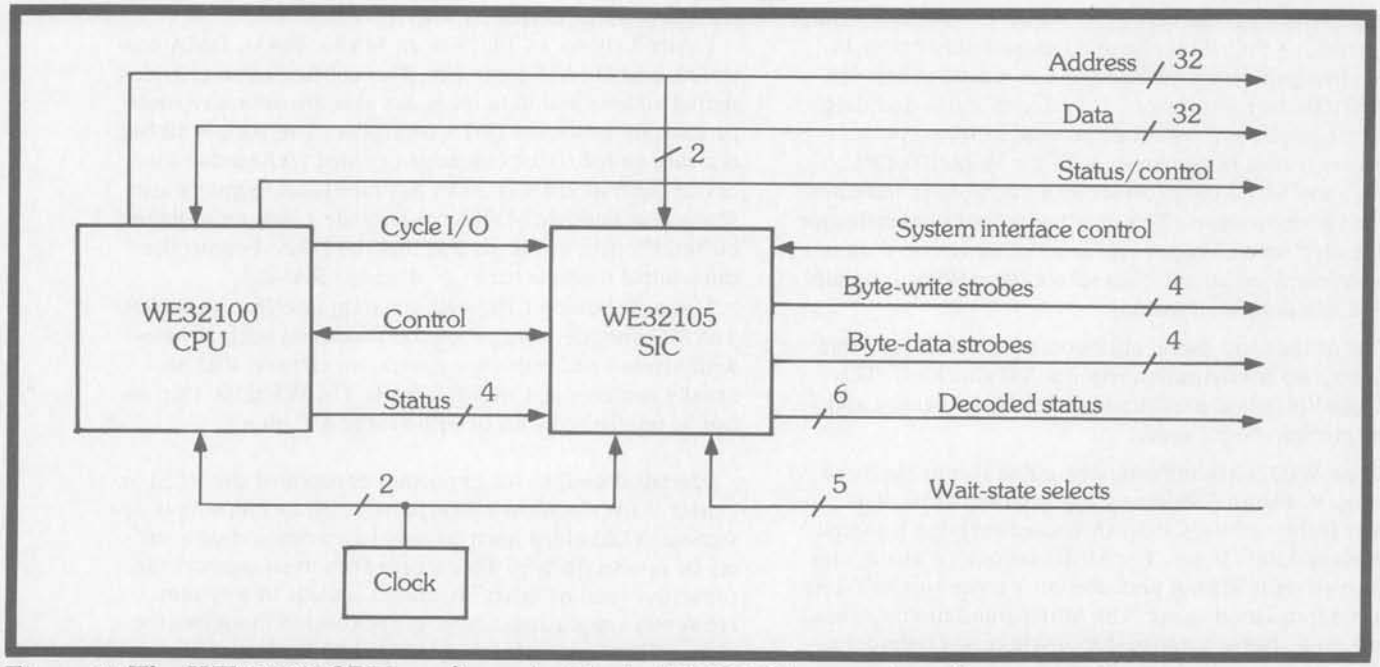

Figure 9. The WE32100 CPU configured with the WE32105 system interface controller.

be realizable. As mentioned before, off-chip communications is a performance limiter for VLSI devices and capacitance is the primary reason for this limitation.

If capacitance is the root of most of the difficulty in VLSI I/O design, then inductance is also a contributor to the problem. Indeed, the package inductance for VLSI

chips can induce on-chip noise voltage that can cause on-chip logic to take incorrect values. A high-frequency chip set architecture must be designed to keep induced voltage noise below the switching logic threshholds. This can be accomplished by reducing the number of concurrently switching signals and also by adding multiple power and ground leads (thereby reducing inductance) to the members of the chip set. In the WE32100 chip set architecture, the assertion of address and data signals on their respective buses occurs one cycle apart. This reduces the number of concurrently switching signals and hence keeps the induced noise within tolerable limits.

One of the critical tasks of the WE32100 chip set architects was specifying the timing budget for interchip communications. This had to be done for each signal between each chip for each supported configuration. Further, the timing budgets had to be considered across the process variations between chips for best- and worst-case operating conditions. This ensured that the chip set would be functional even if worst-case and best-case processing manifested itself in the configured chip set. Such variation in processing is particularly tricky in asynchronous interfaces.

Budgeting timing is one of the most painstaking aspects of architectural design—every nanosecond must be accounted for. The penalty for miscalculation is very high. The chip set architect cannot draw a block diagram and walk away.[15] He must oversee the design and verification process until the architecture is realized.

WE32100 heritage

The WE32100 chip set—a true second-generation 32-bit microprocessor—is descended from the first 32-bit microprocessor, the WE32000 (Bellmac-32A). The WE32100's designers wanted to ensure that its architecture would be compatible with that of its predecessor. This narrowed the range of architectural trade-offs we had to consider.

The WE32100's predecessor was meant to have minicomputer functionality from the beginning. This includes high-level-language support, virtual memory, and floating-point capability. Unlike the WE32100, other 32-bit microprocessors have an 8- or 16-bit ancestry dominated by microcontrollers in which HLL and virtual memory were alien concepts.[16,17]

Not only is the WE32100 CPU a second-generation chip but so is the WE32101 MMU. The Bellmac-32A MMU was the first 32-bit VLSI memory management unit.

An important part of the WE32100 heritage is the manner in which the design of the chip set was handled. All the members of the chip set were designed concurrently—that is, the architectural design proceeded from a chip set perspective. Architectural decisions were based on how the entire chip set was affected; chip-specific decisions were avoided. A further advantage to developing all chips concurrently was the ability it gave the designers to make trade-offs where interchip timing was concerned. The designers of communicating chips could negotiate when they ran into tricky timing situations. Such negotiation could not have occurred if the chips had not been designed concurrently.

We have discussed some of the hardware architecture issues that face a VLSI computer architect. Of course, no architecture is optimal everywhere. Trade-offs must be made, and here we have focused on the alternatives faced by the architects of the WE32100 chip set.

The key to an architecture is that it be implementable and perform well. Invariably this means facing real-world constraints. In a good architecture, however, there should always be room to improve the architecture as the technology of implementation improves.

There is never full freedom in a real-world architecture. Within the goals set forth above, the WE32100 is very successful. The members of the chip set worked, at speed, the first time. The robustness of the interchip protocols has been verified many times over for the various configurations. Furthermore, the members of the WE32100 chip set often were near perfect on their first silicon, requiring only minor rework.

Acknowledgments

The successful realization of the WE32100 chip set was the work of many people. We acknowledge the contributions of the members of the architecture team, the design teams, the verification teams, and the software and development tool groups.

References

1. *UNIX Microsystem-WE32100 Microprocessor Information Manual*, AT&T Technologies Inc., Morristown, N.J., Jan. 1985.
2. A. Berenbaum, M. Condry, and P. Lu, "The Operating System and Language Support Features of the Bellmac-32 Microprocessor," *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, Calif., Mar. 1982, pp. 30-38.
3. M. Fuccio, L. Goyal, and B. Ng, "Hardware Configurations and I/O Protocol of the WE32100 Microprocessor Chip Set," *AFIPS Conf. Proc.*, Vol. 54, 1985 NCC, pp. 241- 246.
4. A. K. Goksel, J. Fields, U. Gumaste, and C. Kung, "An IEEE Standard Floating Point Chip," *Proc. Int'l Solid-State Circuits Conf.*, New York, Feb. 1985.

5. P. M. Lu et al., "Architecture of a VLSI MAP for Bellmac-32 Microprocessor," *Digest of Papers—Compcon Spring 83*, San Francisco, Feb.-Mar. 1983, pp. 213-217.

6. IEEE Task P754, *A Proposed Standard for Binary Floating-point Arithmetic, Draft 10.0*, Dec. 2, 1982.

7. T. J. Chaney, S. M. Ornstein, and W. M. Littlefield, "Beware the Synchronizer," *Digest of Papers—Compcon 72*, San Francisco, Sept. 1972, pp. 317-319.

8. T. J. Chaney and C. E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Trans. Computers*, Vol. C-22, No. 4, Apr. 1973, pp. 421-422.

9. S. T. Flannagan, "Synchronization Reliability in CMOS Technology," *IEEE J. Solid-State Circuits*, Vol. SC-20, No. 4, Aug. 1985, pp. 880-882.

10. P. J. Denning, "Virtual Memory," *Computing Surveys*, Vol. 2, No. 3, Sept. 1970, pp. 153-189.

11. J. Bell, D. Casasent, and C. G. Bell, "An Investigation of Alternative Cache Organizations," *IEEE Trans. Computers*, Vol. C-23, No. 4, Apr. 1974, pp. 346-351.

12. D. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, 1982.

13. H. Stone, *Introduction to Computer Architecture*, SRA, Chicago, 1975.

14. A. Tanenbaum, *Structured Computer Organization*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

15. J. A. Mills, "A Pragmatic View of the System Architect," *Comm. ACM*, Vol. 28, No. 7, July 1985, pp. 708-717.

16. D. R. McGlynn, *Microprocessors: Technology, Architecture and Applications*, John Wiley and Sons, New York, 1976.

17. K. Short, *Microprocessors and Programmed Logic*, Prentice-Hall, Englewood Cliffs, N.J., 1980.

For further reading

I. Catt, "Time Loss Through Gating of Asynchronous Logic Signal Pulses," *IEEE Trans. Electronic Computers*, Vol. EC-15, No. 1, Feb. 1966, pp. 108-111.

P. Denning, J. E. Savage, and J. R. Spirn, "Models for Locality in Program Behavior," tech. report 107, Dept. of Electrical Engineering, Princeton Univ., Apr. 1972.

A. K. Goksel et al., "A VLSI Memory Management Chip: Design Considerations and Experience," *IEEE J. Solid-State Circuits*, Vol. SC-19, No. 3, June 1984, pp. 325-328.

G. J. Lipovski, *Microcomputer Interfacing: Principles and Practice*, Lexington Books, Lexington, Mass., 1980.

A. C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

Michael L. Fuccio is a member of the Digital Signal Processor Design Group at AT&T Bell Laboratories, Holmdel, New Jersey. He was an architect of the first-generation Bellmac memory management unit and a member of the architecture team for the WE32100 microprocessor chip set. A member of Tau Beta Pi, Eta Kappa Nu, and the IEEE, he received the BEEE degree from the State University of New York at Stony Brook in 1980 and the MSEE degree from the Georgia Institute of Technology. He is pursuing graduate study at Rutgers University, where he is concerned with multiprocessor and parallel computing architectures for digital signal processing and computer graphics.

Questions about this article can be directed to Fuccio at AT&T Bell Laboratories, Room 2D-205, Crawfords Corner Rd., Holmdel, NJ 07733.

Benjamin Ng has been a member of the technical staff at AT&T Information Systems since 1981. He was a member of the architecture team for the WE32100 chip set and since has been an architect of a next-generation memory management unit. His areas of interest include computer architecture and computer graphics. A member of Tau Beta Pi, Eta Kappa Nu, the IEEE, and the ACM, he received a BS degree in electrical engineering from Cornell University in 1981 and an MS degree in electrical engineering from Columbia University in 1982.