Architecture of a New Microprocessor

Bernard L. Peuto
Zilog, Inc.

Increased capabilities, architectural compatibility, and clearly defined interfaces were the chief architectural goals of Zilog's new Z8000 microprocessor family. Here is an account of how those goals were met for two members of that family—the Z8000 CPU and the MMU.

The Z8000 family is a new set of microprocessor components (CPU, CPU support chips, peripherals, and memories) which supports the Z8000 architecture. The account of how architectural goals were selected and achieved for two key members of this family—the Z8000 CPU and the memory management unit—illustrates how much of a challenge microprocessor architecture represents to the semiconductor industry. MOS technology shows enormous potential, but it is still difficult to use because of limitations on pin count, power dissipation, speed, and complexity.[1]

Since this discussion is restricted to technical issues, we will not allude to the many additional factors (marketing considerations, human considerations, self-imposed restrictions, etc.) which make architecture such a fascinating and difficult discipline. Furthermore, no attempt has been made to exhaustively describe the Z8000 architecture and components. Interested readers should consult the specific manuals for a more complete description.[2,3]

The goals of the Z8000 architecture: increased capabilities, architectural compatibility, increased clarity

The primary reason for introducing a new system architecture is to significantly improve the control and processing capabilities of microprocessors while maintaining their price/performance advantages. Technical advances have permitted the implementation of substantially increased processor power, but the most significant motivation for a new component family is generality. Only through such a family could we provide for architecturally compatible growth over a wide range of processing power requirements.

Our approach was a staged system architecture which attempts to provide new components, enhanced features, and new functions, while protecting the user's investment in hardware and software. The Z8000 family supports a single unified architecture for all small, medium, and high-end user applications which are implemented using a mix of components within the same family.

The goals of the Z8000 architecture can be grouped into three categories: increased capabilities, architectural compatibility over a wide range of processing powers, and increased clarity. In all these cases the resulting architectural features apply either to the basic architecture (that seen by an applications programmer) or to system architecture (that seen by a system designer or an operating system programmer).

Increased capabilities. All existing 8-bit microprocessors and many 16-bit minicomputers suffer from having a small address space. So, one of our goals was to provide access to a large address space (8M bytes). A second goal was to provide more resources in terms of registers (16 general-purpose 16-bit registers), in terms of data types (from bits to 32 bits), and in terms of additional instructions compared to existing microprocessors (multiply and divide, multiple register saving instructions, specialized instructions for compiler support etc.).

To facilitate complex applications it was important to support multiprogramming with good hardware support of task switching, interrupts, traps, and two execution modes. Operating systems also required a good hardware protection system.

Finally, we wanted to increase overall system performance. This resulted in the choice of an implementation using a 16-bit-wide data path to memory.

Architectural compatibility. One of the important lessons learned from previous computer system designs is that the design of a new family architecture is a rare occurrence. One way to apply this lesson is to design a unified architecture compatible over a wide range of processing powers. If we anticipate user growth from small to large systems within a family architecture, then such an approach can significantly increase its life.

The two versions of the Z8000 (a 40-pin unsegmented and a 48-pin segmented version) are designed to achieve this goal, but many other features contribute indirectly to the family compatibility. For small aplications an unsegmented Z8000 with one or more 64K-byte address spaces can be used. For medium applications, a segmented Z8000 and one memory management unit allows direct access to 4M bytes of address space. For large applications a segmented Z8000 and multiple pairs of MMUs allow the use of several 8M-byte address spaces.

Since the segmented Z8000 can run in an unsegmented mode, both systems are compatible. Finally, to achieve even larger processing power through hardware replication, the architecture provides basic mechanisms for both multiprocessing and distributed processing.

Clarity. Clarity in an architecture is a measure of how well key interfaces are defined and specified. This is an elusive but important goal in a family where new and unforeseen components will be added during the life of its architecture.

We felt bus protocols were so important that we developed an independent specification for the Z-bus along with the individual device manuals.

Clarity in terms of the basic architecture means regularity and extendability of the instruction set, as well as the general and simple handling of the operating system interfaces. Clarity in terms of the system architecture means a well-defined method of communication between the various components. The key link between these components is the Z-bus, which is a shared system bus. In the section on communication with other devices, we describe some of the various types of bus protocols. At Zilog we felt this was so important that we developed an independent specification for the Z-bus along with the individual device manuals.[4]

Comparison with other system architectures

We are convinced that the differences between microprocessor system architecture and large computer system architecture are not sufficient to require a different design approach, although they certainly influence the details of design compromises. The last section of this paper deals with implementation tradeoffs and illustrates some particular compromises. (In a few places we mix implementation considerations with descriptions of architectural tradeoffs. Despite the importance of separating an architecture from its implementation, we found that this separation is often absent during the actual creation of a new architecture.)

Two differences between conventional computer systems and microprocessor systems have the greatest impact: price structure and component boundary differences. For high-end LSI systems, it makes sense to have one unified architecture, but unlike their computer family counterparts (IBM 360/370, PDP-11) different implementations cannot be justified on a price/performance basis. Speed and performance are mainly dependent on the state of technology, and therefore, for a given application, a user will waste the speed willingly since another slower implementation would cost the same. This does not exclude different versions of one implementation, which reflect only different test and production criteria such as package type, functional temperature range, and even speed range.

Most computer systems have both external and internal interfaces. External interfaces which define system boundaries are often standardized (e.g., the IBM channel interface or the DEC unibus). The internal interfaces of most mini or large computer systems are essentially hidden. In contrast, the component boundaries of a microprocessor-based system represent actual interfaces, and most users must be familiar with them as well as with external interfaces. Because the component interfaces are more visible and often must be more general, the microprocessor-oriented system bus emerges as a key standardization link to allow a wider mix of components and designs.

The basic architecture

Address space considerations. It is advantageous to have more than one address space, with each address space as large as possible. In the Z8000, memory references and I/O references are viewed as references to different address spaces. The I/O space is discussed in the section below on communication with other devices. Memory references may be instructions or data and stack accesses, with each type of access possible in either system or normal modes. The Z8000 distinguishes between each of these reference possibilities by using different combinations of its status lines. Separating the various address spaces can be used to increase the total number of addressable bytes and to achieve protection. The size of each address space depends on the versions of the Z8000 used. The 40-pin package version allows each address space to be at most 64K bytes, the 48-pin package version allows each address space to be at most 8000K bytes.

The 40-pin version is intended for systems, often used as dedicated systems, where the program and data spaces are small. In this case, relocation is not usually important. Using the different address spaces, one has a simple way to address in practice up to 4 x 64K bytes (with a maximum of 6 x 64K bytes). Some simple protection is achieved by separating these spaces in hardware.

The 48-pin version with one or more MMUs is intended for the medium to large applications where relocation and better memory protection are important.[3] In these cases, status information can also be used to separate between address spaces by using multiple MMUs. But it is also essential to achieve the detailed memory protection required. (It is possible to use the 48-pin version without an MMU.) For these high-end applications, the address spaces are so large that one is unlikely to exhaust them. Experience with large computers shows that 8M bytes is probably adequate. The current implementation of the Z8000 uses 8M-byte address spaces, but the architecture provides for 31-bit address (2147M bytes).

In both versions, the Z8000 allows direct access to each address space. Direct access means that the addresses used in instructions or registers have as many bits as the address space size requires. In other schemes the effective address is a combination of a shorter field in the instruction and other extension bits often found in an implied register. Despite the shorter address fields, we believe this "indirect access" does not save bytes, because extra instructions must be used to load and save the implied registers, which are typically in short supply.

Registers. The Z8000 is primarily a memory-to-register architecture. This characteristic does not entirely exclude other organizations, and mechanisms exist in the Z8000 to support them. For example, memory-to-memory operations are supported for strings, whereas stack operations are supported for procedure and process changes. This choice provides upward compatibility with the Z80. A register architecture also results in good performance, since register accesses are made at a greater speed than memory accesses in the current implementation.

Experience with register-oriented machines seems to confirm that four general-purpose registers are not enough and that a "proper" number is between eight and 32.[5] The Z8000 supports bytes, words (16-bit), and long words (32-bit), and a few instructions even use quadruple-word (64-bit) data elements. If we choose 16, 16-bit registers allow eight 32-bit registers as well as four 64-bit registers (Figure 1). Since addresses are 32 bits, the necessity of at least eight 32-bit registers was obvious. The impact of the 4-bit register field on the instruction format depends also on the number of address modes and operands. Sixteen registers allowed a reasonable tradeoff, whereas 32 registers would have resulted in too few one-word instructions.

With one minor restriction any register can be used by any instruction as an accumulator, source operand, index, or memory pointer. This regularity of the structure is so important that it is worthwhile to sacrifice any possible encoding improvements in instruction formats which could result from dedicating registers to special functions. Encoding improvements based on instruction frequency, so that frequent instructions use one word, are more effective in saving space without having a negative effect on the architecture.

Why not have specialized registers? The difficulty lies in the fact that the restrictions caused by dedication are inconsistent with one another.

Most applications dedicate the available registers to specific functions. For example, most high-level languages require a stack pointer and a stack frame pointer. Then why not, one might argue, have specialized registers? The difficulty lies in the fact that the restrictions caused by dedication are inconsistent with one another. If the architecture supplies only general-purpose registers, the user is free to dedicate them to specific usages for his application without restrictions. This is important in the context of microprocessors where user applications are not well known and where high-level languages are still used infrequently.

For example, the Z8000 allows software stacks to be implemented with any register. There are also two hardware supported stacks, but the registers used are still general-purpose and can participate in any operation. There is no allocated stack frame pointer, since any register can be used by means of the proper combination of addressing modes. The savings realized by register specialization are unattractive when the given function can still be performed simply. The loss that would result from restricting the applications would be too great. In contrast, significant savings result from excluding R0 from use as an index or memory pointer. This exclusion allows one to distinguish between the indexed and direct addressing modes which use the same combination of the instruction address mode field. The price is small, since R0 still can be an accumulator or source register and 15 others accumulator, index, and/or memory pointers are available. In this case the restriction made sense.

Another decision to be made about registers is their size. Since the architecture handles multiple data types we must have multiple data register sizes, which can hold each data type. The solution of the problem is implemented in the architecture by pairing registers, two 1-byte registers make a word register, two word registers make a long word register, etc.

Data types. Users would like to have as many directly implemented data types as possible. A data type is supported when it has a hardware representa-

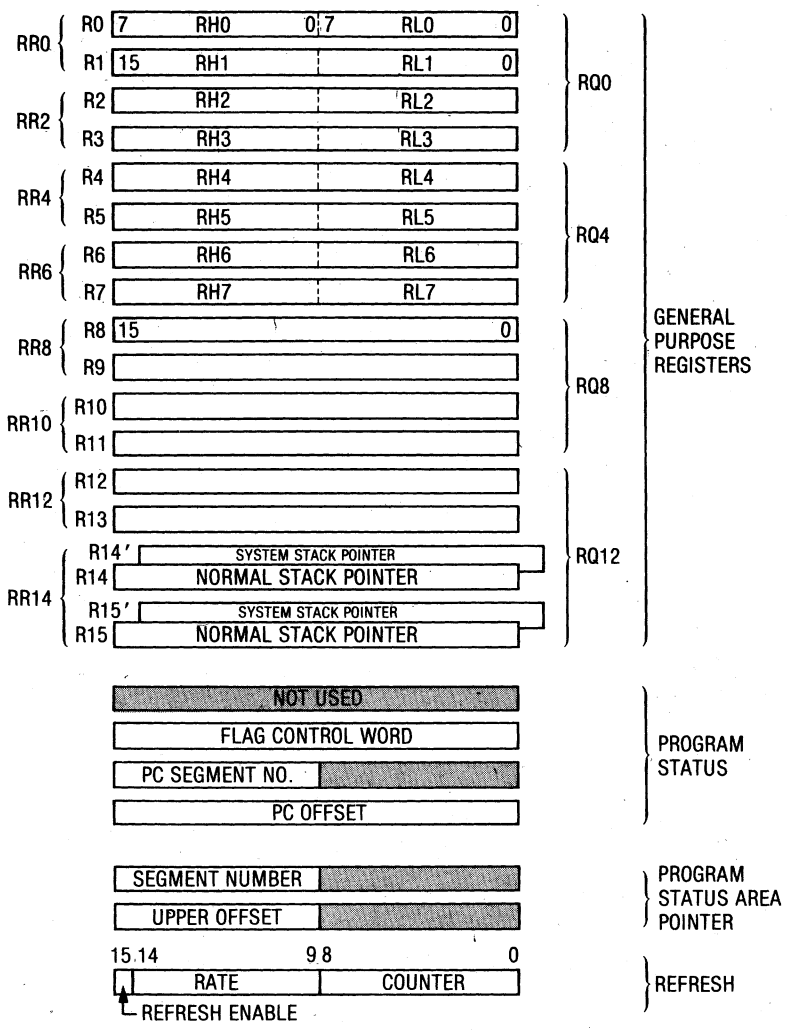


Figure 1. CPU registers (segmented version).

tion and instructions which directly apply to it. New data types can always be simulated in terms of basic data types, but hardware support provides faster and more convenient operations. At the same time, a proliferation of fully supported data types complicates the architecture and the implementations.

The Z8000 supports several primitive types in the architecture and provides expansion mechanisms. The basic data types are obviously the ones expected to be used most frequently. The extended data types are built using existing data types and manipulated using existing instructions.

The basic data type is the byte, which is also the basic addressable element. All other data types are referenced using their first byte address and their length in bytes. The architecture also supports the following data types: bytes (8 bits), words (16 bits), long words (32 bits), bytes, and word strings. In addition, bits are fully supported and addressed by number within a byte or word. BCD digits are supported and represented as two 4-bit digits in 1 byte. One consequence of this data type organization is that byte, word, and long-word registers are needed to support them. The Z8000 even provides quadruple register—another extension—used in long-word manipulation.

Other data types are supported by using one of the preceding data types; for example, addresses are manipulated as long words, and each element (segment number or offset) can be manipulated as a byte or a word. Instructions are one to five-word strings, the program status is four words, etc.

As the family grows, support for new data types will be added. The architecture will need to support them in its registers or in memory if they do not fit in registers (as strings are implemented today). But most important, the architecture will have to support the addition of new instructions to its repertoire.

Instructions. In designing an instruction format the architect must decide how to allocate a limited number of bits to the opcode field, address mode field, and other operand subfields. Instruction usage statistics are the best source of data to influence decisions about instruction set format.[1, 6, 7] Behind their usage lies a strong technical position: we do not

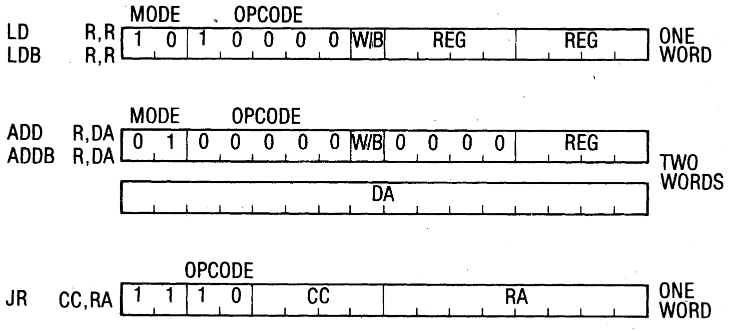


Figure 2. Examples of instruction formats (nonsegmented version).

believe that any one of the various instruction set structures—register oriented, memory oriented, stack oriented, symmetrical, or asymmetrical, etc.—are always better when used exclusively. Thus the task of the architect is to decide what his most important goals are, and for each of them adapt the best features of the various structures so that on the average, and for his set of goals, an optimum solution can be found. We do not believe that the optimum will be very sharp; it will be more like a range of applications for which the resulting composite structure works well. We decided to use a register structure for compatibility, multiple word instructions for speed, memory-to-memory instructions for strings, stack structure for process control and procedure support, "short" instruction for byte density improvement, etc.

Instruction format consideration. The Z8000 has over 110 distinct instruction types; several instruction formats are illustrated in Figure 2. The opcode field specifies the type of instruction (for example, ADD and LD). The mode field indicates the addressing modes (for example, Register (R), Direct Address (DA). The data element type (W/B) and register designator fields complete the basic instruction fields. Long word instructions use a different opcode value from their byte or word counterpart. Frequent instructions are encoded in a single word, and less frequent instructions which use more than two operands use two words. There are often additional fields for special elements such as immediate values or condition code descriptors (CC). Instructions can designate one, two, or three operands explicitly. The instruction TRANSLATE AND TEST is the only one with four operands and is also the only one with an implied register operand.

Several restraints can guide the proper choice of an instruction format. A large number of opcodes (used or reserved) is very important: having a given instruction implemented in hardware saves bytes and improves speed. But one usually needs to concentrate more on the completeness of the operations available on a particular data type rather than on adding more and more esoteric instructions which, if used frequently, will not significantly affect performance. Great care must be given to the problem of expanding the instruction set so, for example, new data types can be added.

Addressing modes. The Z8000 has eight addressing modes: ***register*** (R), ***indirect register*** (IR), ***direct address*** (DA), ***indexed*** (X), ***immediate*** (IM), ***base address*** (BA), ***base indexed*** (BX), and ***relative address*** (RA). Several other addressing modes are implied by specific instructions such as autoincrement or autodecrement.

Although a very large number of addressing modes is beneficial, usage statistics demonstrate that not all combinations of operands, address modes, and operators are meaningful.[6] The five basic addressing modes of R, IR, DA, X, and IM are the most frequently used and apply to most instructions with more than one address mode. For two-operand instructions, statistics show that most of the time the destination is a register. Other cases of addressing mode combinations and less basic addressing modes are associated with special instructions. Thus, the frequent combination of autodecrement for the destination operand with the five basic address modes for the source operand is provided by the PUSH instruction. The combination of autoincrement addressing modes for both source and destination operands is one of the block move instructions. In essence, the address mode field space has been traded for opcode field space. This allows more instructions and combinations while staying within a one-word format.

The price for this tradeoff is the infrequent occurrence of pairs or triples of instructions simulating a missing addressing mode. This situation occurs in most instruction sets in any case.

Code density. Because current microprocessors are restricted to primitive pipeline structures, their speed is largely dependent on the number of executed instruction words. Therefore, code density is not only important because of program size reduction but also because of speed improvement. One would like to encode in the smallest number of bits the most frequent instructions. The basic instruction size increment was chosen to be a word for reasons dealing with alignment, speed penalties, and hardware complexity. Thus the most frequent one and two-operand instructions take one word in their register or register-to-register forms. Less frequent instructions or instructions which use more than two operands use at least two words.

The Z8000 goes even further by selecting several special instructions as "short" instructions which take only one word, when normally they would take two words. These instructions, such as LOAD BYTE REGISTER IMMEDIATE and LOAD WORD REGISTER IMMEDIATE (for small immediate values), CALL RELATIVE, and JUMP RELATIVE, are so frequent statistically that they deserve such special treatment.

A one-word JUMP RELATIVE and DECREMENT AND JUMP ON NON-ZERO also have a very significant impact on speed. The short offset mechanism used by addresses (and described below) is also designed to allow one-word addresses. Compared to previous microprocessors, the largest reduction in size and increase in speed results from the Z8000's consistent and regular structure of the architecture and from its more powerful instruction set—which allows fewer instructions to accomplish a given task.

High-level language support. For microprocessor users, the transition from assembly language to high-level languages will allow greater freedom from architectural dependency and will improve ease of programming.[8] It is easy and tempting to adapt a computer architecture to execute a particular high-level language efficiently.[9] Most programming languages act as a filter and can be supported by a subset of available hardware with greater efficiency.[10] But efficiency for one particular high-level language is likely to lead to inefficiency for unrelated languages. The Z8000 will be used in a wide variety of applications, and we know that a large number of users will still be using assembly languages. Since the Z8000 is a general-purpose microprocessor, language support has been provided only through the inclusion of features designed to minimize typical compilation and code-generation problems. Among these is the regularity of the Z8000 addressing modes and data types. The addressing structure provided by segmentation should support procedures that result from structured programming. Access to parameters and local variables on the procedure stack is supported by index with short offset address mode as well as base address and base indexed address modes. In addition, address arithmetic is aided by the INCREMENT BY 1 TO 16 and DECREMENT BY 1 TO 16 instructions.

Testing of data, logical evaluation, initialization, and comparison of data are made possible by the instructions TEST, TEST CONDITION CODES, LOAD IMMEDIATE INTO MEMORY, and COMPARE IMMEDIATE WITH MEMORY. Compilers and assemblers manipulate character strings frequently, and the instructions TRANSLATE, TRANSLATE AND TEST, BLOCK COMPARE, and COMPARE STRING all result in dramatic speed improvements over software simulations of these important tasks, especially for certain types of languages. In addition, any register can be used as a stack pointer by the PUSH and POP instructions.

Segmentation. In order to provide for convenient code generation and data access, addresses must also be easy to manipulate. Architectures with direct access to memory typically use a linear address space, so that address arithmetic may be used on the entire address. In this case, addresses are manipulated as one of the data types of the same size. This removes the need to distinguish an address as a new data type. In contrast, the Z8000 has a non-linear address space. Addresses are made of two parts: a 7-bit segment number and a 16-bit offset. Only the offset participates in address arithmetic. The segment number is essentially a pointer to a part of the total address space, which can vary in size from 0 to 64K bytes. The hardware representation of a segmented address is a long word or a register pair (Figure 3), which allows the easy manipulation of each part of the address.

The segmented addresses are one of the key mechanisms used to support both large and small memory systems efficiently. The two versions of the Z8000 implementation, the 40-pin unsegmented and the 48-pin segmented, allow the maintenance of the architectural compatibility and ease the growth between these two application groups. The segmented address space guarantees that each 64K-byte address space of the 40-pin version becomes one of the segments of the 48-pin version. Each 40-pin version's 16-bit address becomes an offset within the segment, and a mode exists in the 48-pin package version in which 40-pin version code can be executed. Furthermore, compatibility with any current 8-bit microprocessor such as the Z80 is easy, and a new microcomputer such as the Z8 can address external data in a shared segment with the Z8000.

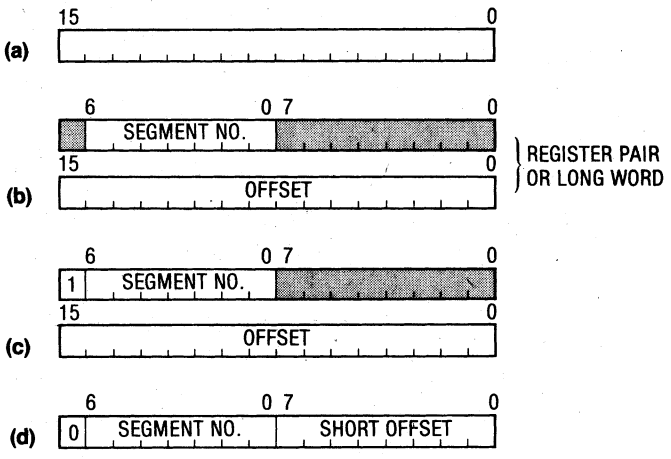


Figure 3. Hardware representation of segmented addresses. Any nonsegmented address is one word, whether it is in a register, memory, or an instruction (Figure 3a). Segmented addresses are always two words in a register or memory (Figure 3b); however, instructions can have one of two forms. The usual case (long offset) requires two words (Figure 3c); however, there is also a short offset form that uses only one word (Figure 3d).

The hardware performance of the Z8000 is also improved by address segmentation. Since a segment number does not participate in arithmetic, it can be put on the bus before the result of an address computation is available. This feature allows the use of MMUs with essentially no impact on memory access time by allowing it to function in parallel with the CPU. Indexing operations are also faster because only a 16-bit addition must be performed. Because of the distinction between the segment number and its offset, one can use shorter addresses without software constraints. Short addresses can use a short offset (fewer than 256 bytes) and thereby reduce program size (Figure 3).

Finally, it is very easy to associate with each of the 128 segments of the address space the protection and dynamic relocation features desirable for larger systems. Relocation allows a user to write his application using logical addresses independent of any physical addresses. Relocation is essential, for example, in a disk-based general data processing system with several users. Relocation is not essential for dedicated applications with code typically residing in

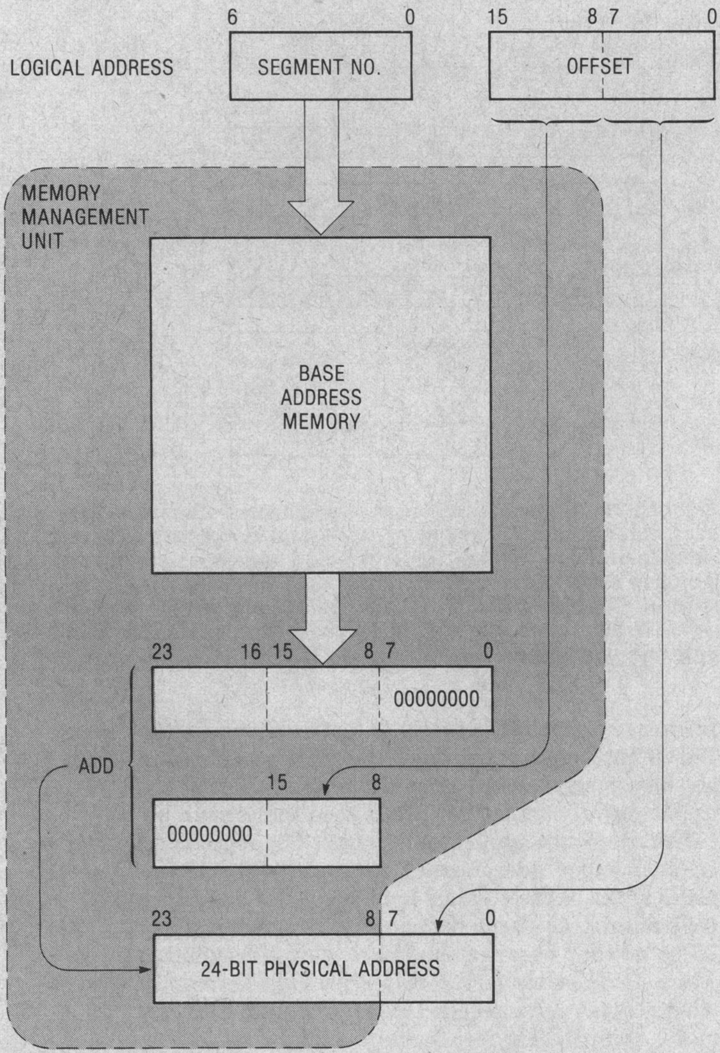


Figure 4. Logical to physical address translation.

ROM. Users whose total memory needs are small are also unlikely to need relocation.

In summary, the choice of a segmented address space has provided—at low cost and with few practical limitations—a powerful solution to the problem of user growth, relocation, and protection as well as virtual memory implementation. We believe that a linear address space could have achieved these results but at a considerably higher price.

The system architecture

Protection facilities. The Z8000 protection facilities can be divided into system protection features and memory protection features. Experience with large computers has demonstrated the advantages of having at least two execution modes with different access rights to hardware facilities. The Z8000 provides the system and normal modes for this purpose. A simple protection system results from the presence of these two modes and their associated stacks. A special class of "privileged" instructions is defined, which deals with I/O, interrupts, traps, and mode changes. Programs in normal mode which attempt to execute a privileged instruction will cause a trap and a change to system mode. The switch from user to system mode can also be caused by the system call instruction. These mechanisms enforce protection and help in designing reliable and efficient operating systems with clean user interfaces. Several other traps are required to achieve a consistent system: segmentation trap, privileged instruction trap, and undefined instruction trap.

A desirable memory protection scheme is one for which protection information (read only, read write, execute only, system only, size of data or code, etc.) is easily associated with the data and code structures of a given application. It is also one for which a large number of different types of protection information can be verified.

The relocation and memory protection mechanisms described above are provided by an external device: the memory management unit.[3] To provide relocation and protection features directly on the Z8000 would have demanded too much simplification. The external MMU has the further advantage of providing for easier growth by the addition of components. The Z8000 40-pin package does not have to carry the burden of the unused advanced relocation and protection features, although some form of protection can be achieved by hardware separation of the different address spaces. With multiple MMUs, the 48-pin package user can control the relocation and protection complexity desired in his application.

The memory management unit. The MMU performs three functions: (1) address translation of logical address to physical address using dynamic relocation, (2) memory protection, and (3) segment management. The addresses manipulated by the programmer, used by the instructions, and output by the Z8000 are called logical addresses. The MMU uses these logical addresses, composed of a 7-bit segment number and 16-bit offset, and transforms them into a 24-bit physical address (Figure 4). A 24-bit origin or base is logically associated with each segment. To form a 24-bit physical address, the 16-bit offset is added to the base for the given segment. In effect, with the help of one memory management device, the Z8000 can address 8M bytes directly within a 16M-byte physical memory space. The reasons for the choice of a large physical address space include an expectation that large systems will want to use extra bits for complex resource management purposes.

Each segment is given a number of attributes when it is initially entered into the MMU. When a memory reference is made, the protection mechanism checks these attributes against the status information from the CPU. If a mismatch occurs, a trap is generated which interrupts the CPU. The CPU can then check the MMU status registers to determine the cause of the trap. Segment attributes include segment size and type (read only, system only, execute only, invalid DMA, invalid CPU, etc.) Other segment protection features include a write warning zone useful for stack operations.

When a memory protection violation is detected, a write inhibit line guarantees that memory will not be incorrectly changed. The invalid DMA and CPU bits indicate that the entry cannot be used by the DMA or CPU respectively, because either the segment number is illegal or the segment entry is not loaded. This fast feature, in conjunction with the segment history information (segment "changed" and segment "referenced" bits) and the segmentation trap mechanism, allows the implementation of a virtual segmented memory system.

The MMU comes in a 48-pin package (Figure 5). The chip inputs are the segment number, the upper 8 bits of the offset, and status information from the CPU. The outputs from the segment chip are the upper 16 bits of the 24-bit physical address and the segmentation trap line. Since the memory management device processes only the upper 8 bits of the offset, the lower 8 bits go directly to memory. This is equivalent to having zeros in the 8 lower bits of the 24-bit origin. Thus, the memory management device only needs to store the upper 16 bits of each base address. Segment limit protection is done in the memory management device, and thus segments can be protected in increments of 256 bytes.

Each MMU stores 64 segment entries that consist of the segment base address, its attributes, size, and status. A pair of MMUs support the 128 segments available in an address space. Additional MMUs can be used to accommodate multiple translation tables. Using the status information provided with each reference, pairs of MMUs can be enabled dynamically.

The memory management device functions constantly while memory references are made, but its translation and protection tables are loaded and unloaded as an I/O peripheral. To achieve this, the memory management device has chip select, address strobe, data strobe, and read/write lines. The Z8000 special byte I/O instructions that use the upper byte of the data bus can load or unload the memory management device.

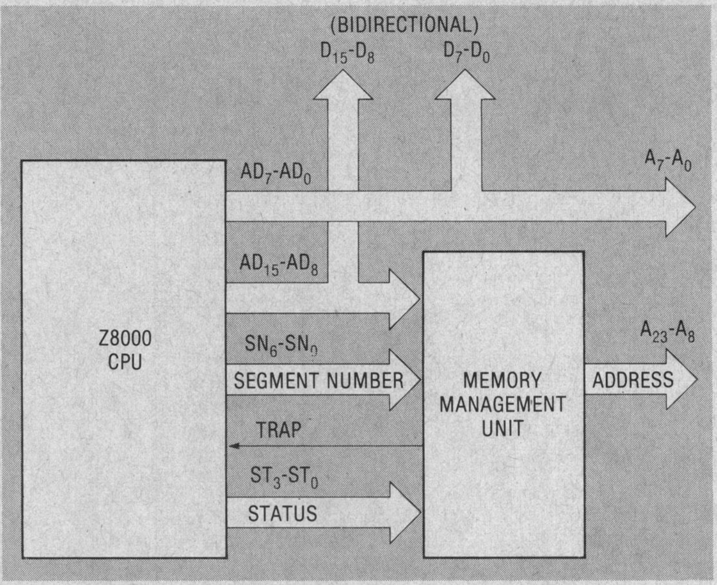


Figure 5. Memory management device with Z8000 CPU.

Mode switching: interrupt and trap handling. From small users in dedicated process control applications to large users in general-purpose data processing applications, asynchronous events such as interrupts and synchronous events like traps must be handled. When these events occur, the state of any currently executing program must be saved during what is generally called a task switch or process switch. The users benefit from the availability of many interrupts and traps. They also benefit from a fast, easy, and uniform handling of process switching.

Peripherals using interrupts have widely varying constraints on interrupt processing time. To solve this problem, peripherals with the same characteristics are often associated with one of several interrupts. A priority enforced among the several interrupts allows the required processing time to be guaranteed. Enabling or disabling the various interrupts is the mechanism used to enforce this processing priority.

In the Z8000, we felt that three levels of interupts were sufficient. A *non-maskable interrupt* represents a catastrophic event which requires special handling to preserve system integrity. In addition there are two maskable interrupts: *non-vectored interrupts* and *vectored interrupts*, which correspond to a fixed mapping of interrupt processing routines and to a variable mapping of interrupt processing routines depending on the vector presented by the peripheral to the Z8000.

Both interrupts and traps result in similar process switches. Information related to the old process (its program status) is saved on a special system stack with a code describing the reason for the switch.. This allows recursive task switches to occur while leaving the normal stack undisturbed by system information. The state of the new process (its new program status) is loaded from a special area in memory—the program status area—designated by a pointer resident in the CPU (see Figure 6).

The use of the stack and of a pointer to the program status area are specific choices made to allow architectural compatibility if new interrupts or traps are added to the architecture. The choice of the two modes of execution has a strong impact on the design of clean user interfaces. Experience has shown that in large systems the normal mode instruction set and the user interfaces together constitute the most important element in achieving architectural compatibility.

Communication with other devices: the Z-bus. The Z-bus is the shared bus which links all the components of the Z8000 family.[4] The variety and performance requirements of the components are so different that in fact the Z-bus is composed of five buses:

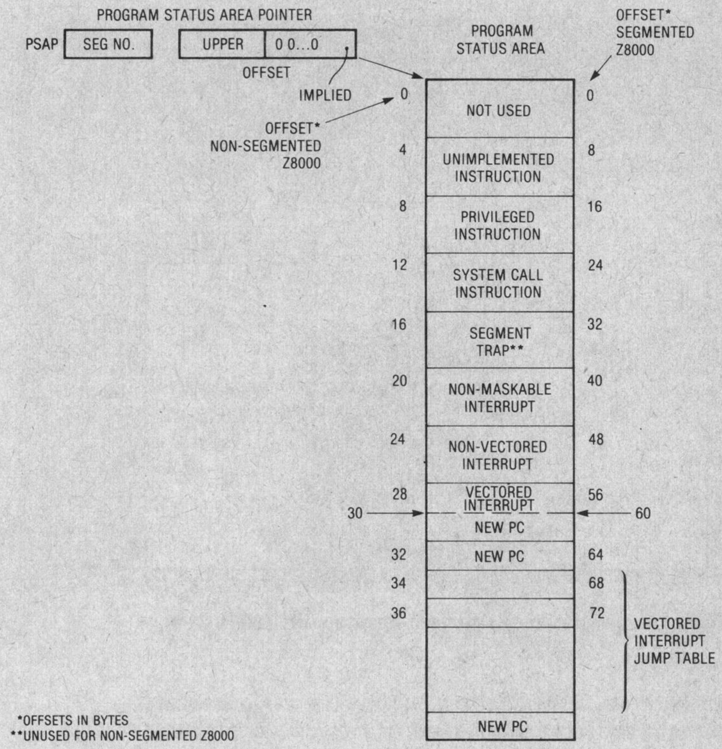


Figure 6. Program status area.

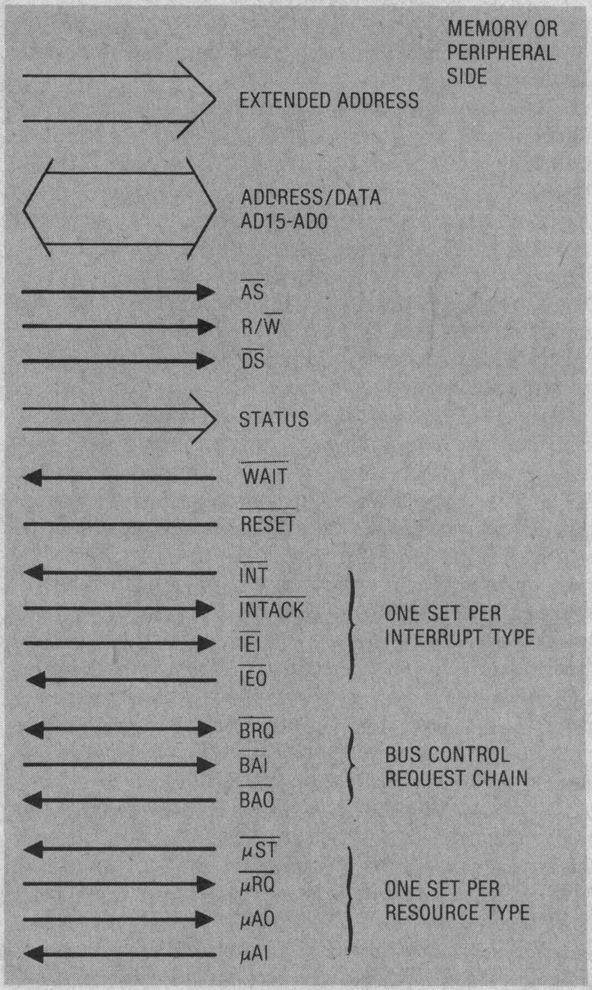


Figure 7. Z-bus signals.

a memory bus, an I/O bus, an interrupt bus, and two resource request buses (Figure 7).

The Z-bus is called a "shared" bus because several components can use it. A bus user is a CPU or a peripheral which can usually generate one or more bus transactions such as memory data request or an I/O request. Identical bus transactions cannot take place at the same time, but serialization mechanisms allow sequential use of the Z-bus. Architecturally, the buses can be grouped into two structures. The I/O structure uses the I/O bus and the interrupt bus. The memory structure uses the memory bus with or without address extensions. Both structures can use the resource request bus and the mastership request bus.

Each bus consists of a set of signals and the protocols which preside over the various types of transactions. Part of each protocol is the timing relationship between relevant signals. The Z8000 CPU provides most of these timing relations. The advantage of such a choice is the significant reduction in the number of components required to build such a system. One consequence is that bus transactions cannot be aborted or delayed freely since some devices, especially memory, have specific timing constraints. The most important consideration for the Z-bus is the need to interface to multiplexed address and data lines of the Z8000 CPU which must fit in 40- and 48-pin packages. The Z-bus maintains these multiplexed address and data lines. Very little speed could be gained by demultiplexing these lines for memory references since memories are themselves multiplexed. The most important advantage of a multiplexed Z-bus is the direct addressability of peripheral internal registers. This feature allows the construction of complex peripherals which maintain a simple program interface.

The Z-bus is known as a transparent or asynchronous bus. Z8000 components do not require that their clocks be synchronized with the CPU clock. The signals used by each transaction provide all the necessary timing. This concept is important: it allows, for example, I/O references to be independent of the speed and clock frequencies required by other Z-bus transactions.

I/O bus versus memory bus. The I/O and memory buses are the most important. The Z8000 family architecture distinguishes between memory and I/O spaces and thus requires specific I/O instructions. This architectural separation allows better protection and has a nicer potential for extension. The I/O and memory buses use a 16-bit address/data bus, which allows 16-bit I/O addresses and 8- or 16-bit data elements. Memory addresses are 16 bits for the 40-pin package or extended to 23 bits using the segmented version. Thus, the memory bus is in fact a logical address bus. The increased speed requirements of future microprocessors is likely to be achieved by tailoring memory and I/O references to their respective characteristic reference patterns and by using simultaneous I/O and memory referencing. These future possibilities require an architectural separation today. Memory-mapped I/O is still possible, but we feel the loss of protection and potential expandability are too severe to justify memory-mapped I/O by itself.

Both the I/O and memory buses need address, data, and control signals. One important implementation decision was to overlap the signals used by the memory and I/O buses on the same Z8000 CPU pins, with the obvious exception of the status signals used to distinguish between the two types of bus requests. For the current Z8000 implementation the resulting reduction in number of pins is significant. In contrast the impossibility of doing concurrent memory and I/O referencing is not very significant since their speeds are essentially the same.

In addition, memories and peripherals both benefit from the availability of early status information defining the bus transaction type (I/O versus memory, read versus write) ahead of the actual transaction so that bidirectional drivers and other hardware elements can be enabled before the reference. The status lines of the Z8000 CPU provide this type of early status.

The I/O structure. Since many peripherals are connected with one CPU, the I/O bus is shared and serialization must be provided. One solution involves using a master/slave protocol. The CPU is a master which can initiate an I/O transation at any time. The peripherals are slaves which participate in a transaction only when requested by the master. In order to find out if a peripheral needs to be serviced the master can poll each in turn. The Z-bus also provides a faster way of getting the attention of a master: an interrupt bus. In contrast, with the I/O transaction data bus, each peripheral sharing the interrupt bus may "try" to use it simultaneously. The interrupt bus uses an interrupt line, interrupt acknowledge line, and two more lines used to form a daisy chain. The daisy chain is an implementation of a distributed arbitration policy between the requests. Priority of processing is determined by the position in the daisy chain, and peripherals can be preempted. Interrupt vectors are used to determine the identity of the peripherals requesting service via an interrupt.

Other buses. The two resource request buses are used to request the control of the Z-bus from the CPU and to request control of any generalized resource.

The Z8000 CPU or any Z-bus compatible CPU does not need to request the bus to access it as a master, and is, therefore, the default master. Other devices can request bus mastership, but they must go through a non-preemptive distributed arbitration using another daisy chain. The CPU always relinquishes the bus at the end of its current bus transaction.

The resource request chain is a generalization of that concept in which each resource requestor has equal importance and can use the resource in a non-preemptive manner. This mechanism in the Z8000 CPU permits one to implement in software the kind of exclusion and serialization mechanisms needed for multiple distributed systems with critical resource sharing.

Multiprocessing. In the context of today's large mainframe systems characterized by multiple processes sharing one processor, one is tempted to design distributed processing systems with many low-cost microprocessors running dedicated processes. Such an approach distributes intelligence towards the peripherals, results in modularization, and permits easier development and growth. Unfortunately, in the past, the problem with such an approach has been software and not hardware. Thus one cannot be expected to provide detailed solutions in hardware to a software problem that has not been solved yet. However, some basic mechanisms have been provided to allow the sharing of address spaces: large segmented address spaces and the external MMU make this possible, and a resource request bus is provided which in conjunction with software provides the exclusion and serialization control of shared critical resources. These mechanisms and new peripherals like the Z-FIO have been designed to allow easy asynchronous communication between different CPUs.

Implementation tradeoffs

The key family decision: producibility. Confronted with the problem of designing a new LSI-based system architecture, we could have ignored package size considerations by accepting packages with 64 or more pins, or we could have ignored mass production technology constraints by using die sizes larger than 260 mils square. Such solutions are often justified in the implementation of an existing computer system. The component boundaries, package limitations, and technological limitations are secondary to achieving the goal of exact membership in the computer family. But if one were to design a new system architecture with the same lack of constraints, the individual component would not be price-competitive—only the total system would be. A new system architecture based on this approach could only be used to design yet another traditional computer.

The Z8000 family provides basic, general-purpose blocks out of which a system solution to most problems can be implemented.

The Z8000 family market is intended to be much broader, and each component of the family must be economically viable. The staged introduction of components which are economically viable by themselves allows us to serve the market from very small configurations to very large configurations by using more components, in any combination. Not only do we believe that this approach does not restrict

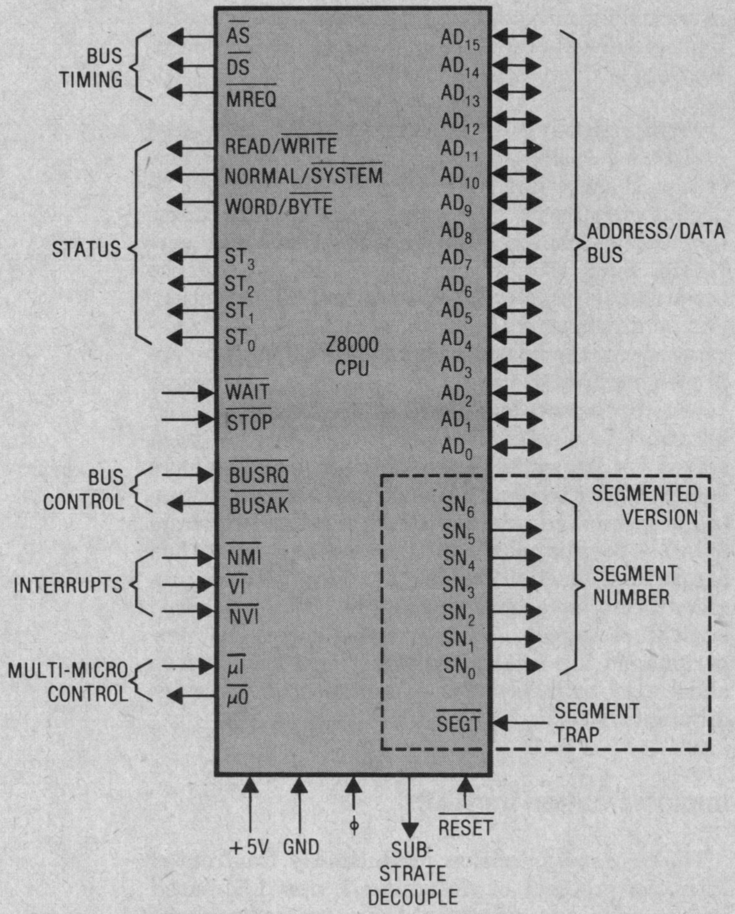


Figure 8. Z8000 pin functions.

system architectural possibilities, but we also believe that the family will be more effective because it will grow with its customer.

The Z8000 family does not always attempt to provide specific architectural solutions, often implemented in hardware, to all system architecture problems. Instead, it provides basic, general-purpose blocks out of which a system solution to most problems can be implemented. The multi-microprocessor and distributed system capabilities of the Z8000 family illustrate the use of open-ended mechanisms to solve a variety of architectural problems, while the memory management of address space illustrates a specific problem supported by a specific solution—the MMU. However, other solutions more appropriate to a particular problem can be used and an advance in the state of the art might be mapped into a new device for the family.

This vision of the family often results in components more powerful and complex than an application may require. The user should not take this as a cause for alarm, but rather as the reason his applications growth will be easier.

Basic CPU implementation decisions. The Z8000 currently uses a 16-bit data bus (Figure 8), an internal register array of 16-bit registers, and a 16-bit parallel ALU. These implementation decisions, which were guided by the technological and practical considerations, have a strong impact on performance.

To achieve good performance with the instruction format and data type envisioned for the Z8000, only a 16-bit bus seens adequate; a 32-bit bus would have necessitated using an unacceptable 56-pin or larger package. Optimal performace is obtained with this chosen bus width if the size of the frequently used register-to-register operations becomes one word. The choice of ALU and internal register widths is a tradeoff between speed of the most frequent operations and the chip area needed to implement a wider ALU or data path inside the CPU.

None of these implementation decisions should limit the architecture. Instructions are from one to five words long, and data types and addresses are not limited to 16 bits. For example, 32-bit words are one of the main data types of the machines, and addresses occupy two words. The *address* mechanism illustrates the strong distinction between an architecture and its implementation. The architectural address representation uses a 32-bit word of which 8 bits are reserved and 1 is a short format/long format descriptor. Thus, the Z8000 architecture provides up to 31-bit addresses, but only 23 are currently implemented and 23 pins of the current package are allocated to addresses.

MMU tradeoffs. The MMU and its relation to the Z8000 CPU illustrate tradeoffs that a microprocessor architect and designer team must make to ensure component manufacturability.

To achieve the goals of good architectural compatibility for high-end systems, it was necessry to include the protection and relocation mechanicms described above. But if all desired features were implemented as a one-chip CPU/MMU combination, it would have been too large and, therefore, uneconomical. And if a reduced set of features were implemented, it would have been architecturally too primitive. Thus, the choice was made to maintain all features and use two chips. This new organization has several significant advantages, such as a capability for multiple MMUs, and allows the access of a DMA device to the MMU.

Given the choice of an external MMU, the next set of decisions concerns package size and circuit speed. Having each relocated segment start on a word boundary would have required a 64-pin package and a very fast 24-bit adder (in fact, a 16-bit adder and 8 bits of carry propagation). In contrast, the decision to start segments on 256-byte boundaries allows the use of a 48-pin package, a fast 8-bit adder, and 8 bits of carry propagation. The latter solution is technically superior and places practically no restriction on the architecture. Segment granularity can be viewed as an implementation restriction and not as an architectural restriction.

Making the 8 low-order bits of the offset go directly to memory also significantly reduces memory access time. Since dynamic memories use these bits first, most of the MMU relocation time is hidden during a normal memory access. The availability of segment numbers earlier than the associated offset bits reinforces this advantage and allows the MMU to result in essentially no memory access speed reduction. Each MMU entry also requires 8 bits less for base and segment size value. This is important: it is desirable to pack as many entries as possible per MMU. With 64 entries a 2K-bit memory is needed, which is technologically difficult in view of the amount of logic surrounding this memory and the complexity of its organization.

The fact than an MMU is only connected to the upper byte of the data bus requires the use of special I/O instructions for its loading and obliges us to replace the possible use of an automatic demand loading of entries by explicit instruction loading. To compensate for the time penalty associated with the loading of potentially unused entries, multiple MMUs are used. They not only allow the implementation of 128 entries, but pairs of MMUs can be automatically enabled by the system and normal mode pins effecting a full environment switch at electronic speed.

We feel this example illustrates one important design approach: to compromise as little as possible on advanced architectural features but to accept compromises which result in implementation ease in order to achieve economical components.

Conclusion

The architectural sophistication of the new 16-bit microprocessors is rapidly approaching the level of the minicomputer and large computer. Problems such as component families, large address spaces, bus standards, I/O structures, software investments, and architectural compatibility are being directly addressed. Some of the solutions to these problems are known, and therefore the transition from 8-bit microprocessors was relatively easy. But the challenges ahead—networks, distributed processing, new applications—are much harder. The impact of microprocessors is already enormous, but we feel they will achieve the often-predicted computer revolution only after these new problems are solved. ■

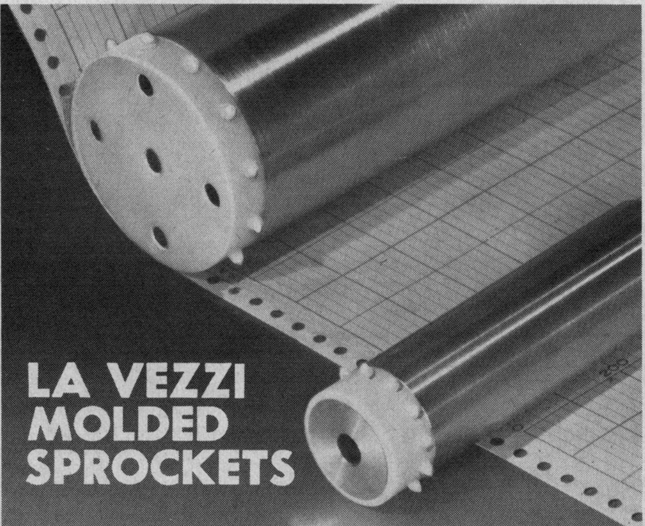
Acknowledgements

The Z8000 family would not exist without the very talented and dedicated designers who contributed to and implemented the ideas described in this paper: Masatoshi Shima for the Z8000, Hiroshi Yonezawa for the MMU, and Ross Freeman for the peripheral devices. Judy Estrin made invaluable contributions to the architecture of the Z8000 and Z8. Many discussions with Charlie Bass, Leonard Shustek, and Forest Baskett have greatly influenced the Z8000. Leonard's instruction set measurements were especially valuable. Dennis Allison, Steve Meyer, Bruce Hunt, and many others must be thanked for their comments on early drafts of this paper.

References

1. B. L. Peuto and L. J. Shustek, "Current Issues in the Architecture of Microprocessors," *Computer*, Vol. 10, No. 2, Feb. 1977, pp. 20-25.
2. Zilog, *Z8000 Technical Manual*, Zilog, Inc., 1979.
3. Zilog, *MMU Technical Manual*, Zilog, Inc., 1979.
4. Zilog, *Z-Bus Specification*, Zilog, Inc., 1979.
5. A. Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *CACM*, Vol. 20, No. 3, Mar. 1977, pp. 143-152.
6. L. J. Shustek, *Analysis and Performance of Computer Instruction Sets*, PhD Dissertation, Dept. of Computer Science, Stanford University, Stanford, Calif., Jan. 1978.
7. B. L. Peuto and L. J. Shustek, "An Instruction Set Timing Model of CPU Performance," *Proc. Fourth Annual Symposium on Computer Architecture*, Mar. 23-25, 1977, pp. 165-178.
8. C. Bass, "PLZ: A Family of System Programming Languages for Microprocessors," *Computer*, Vol. 11, No. 3, Mar. 1978, pp. 34-39.
9. A. S. Tannenbaum, "Implications of Structured Programming for Machine Architecture," *CACM*, Vol. 21, No. 3, Mar. 1978, pp. 237-246.
10. N. G. Alexander and D. B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer*, Vol. 8, No. 11, Nov. 1975, pp. 41-46.

Bernard L. Peuto is one of the guest editors for this special section; his biography appears with the introduction on p. 9.



LA VEZZI MOLDED SPROCKETS

A PRECISE WAY TO DRIVE CHARTS ECONOMICALLY

Drives perforated materials with unvarying accuracy. Maintains chart integrity. 10, 12 and 24-tooth thermoplastic sprockets are formed to exacting specifications. 1/4", 1/10" and 5mm pitch. Immediate delivery.

Our catalog tells all.

LaVezzi machine works, inc.

900 N. Larch Ave. • Elmhurst, Ill. 60126 • (312) 832-8990

Reader Service Number 3