7

Memory Systems

One of the most basic functions of a computer is the retrieval of information stored in a memory element. This action is needed to obtain the instruction to perform; it is also needed to obtain the data on which the instruction operates. One widely used model of memory is shown in Figure 7.1. In this model the memory consists of N consecutive storage locations. The size of a location is dependent on the system architecture, and the width of the data path *(w)* is a function of the implementation mechanisms. But the model remains the same: the address supplies the desired location, and the data is transferred to/from the memory. The number of bits needed in the address is $\lceil \log_2 N \rceil$. We will use this model to represent a memory system, and recognize that for special systems appropriate changes must be made. In many systems, the size of the memory is given in bytes, although that is not the normal width of data transfers. One reason for this is that the systems are byte-addressable, and although the width of the transfer path may not be a single byte wide, the information is obtained by giving the address of the specific byte desired. Then. if more bytes are required, they are obtained as needed by the processing unit.

The design of the memory unit is a series of tradeoffs, since a number of different factors must be considered. These include the size of the memory (N elements), the width of the data path *(w)*, the organization method, and the speed of access. The speed of the memory depends on many factors, including technology of implementation and organizational method. Regardless of the mechanisms involved and the memory technology used, there will be a minimum time to access information, which we will call T_A. This represents the shortest time required to retrieve the information, and includes not only the access time of the memory device, but also any delay caused by additional gates needed to provide sufficient drive capability for the address or the data. Another very important time is the minimum time between accesses. or the memory cycle time, which we will call T_{CY}. For "normal" memory interaction. where information is retrieved

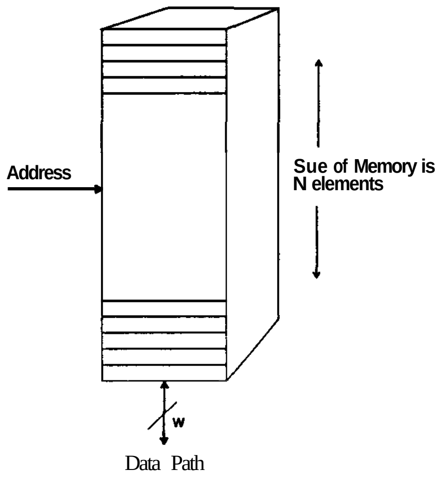


Figure 7.1. Memory Model: Linear **Array** of Locations.

from memory. then work is done on it. and then memory is accessed again. T_{CY} is not a limiting factor. since the action of the system will not result in memory accesses which occur faster than T_{CY}. However, for burst mode access. where several consecutive memory elements are read or written, T_{CY} is a factor that limits the rate of transfer. One simplifying assumption we will make is that the memory times are the **same** for the read and write cycles, which is not always **true.**

The memory itself is configured in such a way that all of the necessary accesses can be made to it. That is, using one or more of the communication protocols described in Chapter 6, the memory is **connected to** elements that need the capability of data transfer **with the** memory. **The simple** representation of Figure 7.2 shows a memory that can be accessed by a **processor** and **I/O** devices. The

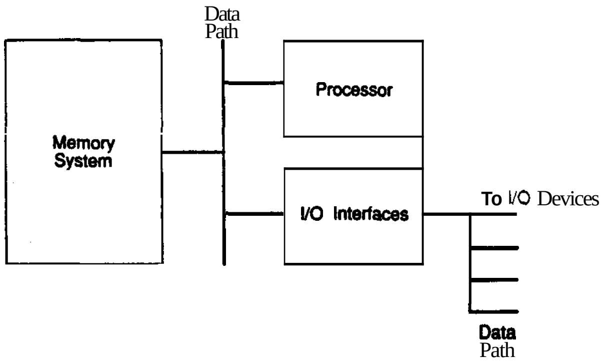


Figure 7.2 Memory System and Connections to Processor and I/O Interfaces.

data path (bus system) is also used to allow the processor to control the action of the I/O interfaces. The configuration shown in Figure 7.2 is a very simplistic representation, and the actual connections to the memory can be as simple or complex as the application requires. In general, however, we would like to create a memory with as large a size (N) as reasonable within the design constraints, with an access time (T_A) and a cycle time (T_{CY}) as short as possible. Let us look at the memory hierarchy mechanisms used to try to accomplish this, then examine details of the memory systems involved.

7.1. Memory Hierarchy: Tradeoffs in Size and Function

In the description of their 1946 IAS machine Burks, Goldstine. and von Neumann recognized that "ideally one would desire an indefinitely large memory capacity such that [infonnation] would be immediately available ..."[BuGo46]. But the realities of the economics and the technology are such that compromises must be made. The IAS machine contained 4.096 words of 40 bits each for the main store, which "exceeds the capacities required for most problems that one deals with at present by a factor of about 10." However, they recognized that the time would come when this would not be sufficient storage for the problems to be solved in the future. and therefore looked forward to the "constructing of a hierarchy of memories. each of which has greater capacity than the preceding, but which is less quickly accessible." The machines of todav indeed match this concept. and can be represented by the block diagram shown in Figure 7.3. The fastest memory elements are those closest to the processor: most systems have a small number of very high speed locations, which we call a register bank. The T_A for the registers is minimal. and in general the infonnation stored in registers is available in the same cycle as it is needed. However, the cost of this type of storage is very high, whether the cost is measured in dollars, silicon real estate, or power dissipation. For this reason, the amount of register storage available in a system is relatively small, from eight to sixteen registers in most general purpose systems, to over a hundred in some special purpose and RISC systems.

The next element in the memory hierarchy is often a cache memory. The purpose of a cache is to enhance the operating speed of the processor by making available the most recently used information by keeping it in a high speed temporary storage. The T_A of a cache is on the order of two CPU cycle times, and we will discuss methods of approaching cache designs in Section 7.4. The amount of

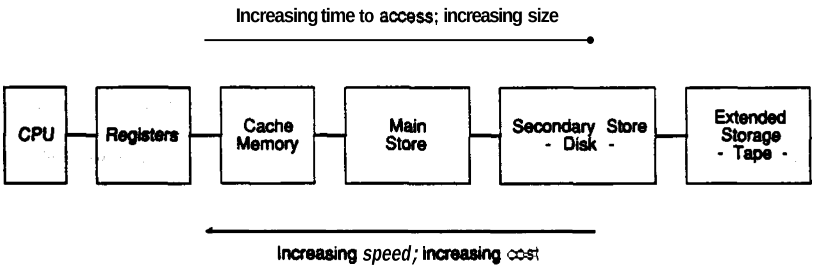

Increasing time to access; increasing size
CPU
Registers
Cache Memory
Main Store
Secondary Store - Disk -
Extended Storage - Tape -
Increasing speed; increasing cost

Figure 7.3. Block Diagram of Hierarchical Memory System.

memory available here is generally small in comparison to the other elements of the system. For example, the VAX 11/780 has a 2-Kbyte cache, and some other processors have even smaller caches. However, as the cost of memory decreases with respect to overall system costs, larger caches are much more common. Many newer systems have caches that contain 16 Kbytes to 64 Kbytes or more incorporated with the processing unit.

The purpose of the cache is to maintain current information for rapid retrieval. This is done in a manner that is transparent to the user. The programmer does not know of the existence of the cache, except that the speed of the system is enhanced over a system with no cache memory. Thus management of the data is done in a fashion determined at design time. in contrast to the virtual memory systems discussed below.

The information in a cache memory is a high speed copy of what is in main store, which is the "standard" memory of the computer. The amount of storage in a main store is system dependent, but it has increased with each passing year. In contrast to the 4.096 locations of the IAS system, many systems require a minimum of 8 Mbytes or more. The technology is now such that it is possible to get 8 Mbytes in eight packages, which is one of the reasons for the increased size of main store. The T_A of main store is about an order of magnitude greater than the T_A for cache. Thus, when a request is made for information, and it is determined not to be in the cache. then the system pauses until the information is retrieved from main store. At that time the processing can continue. Some of the issues involved in the design of the main store are discussed in Section 7.2.

The information resident in main store for a "standard" computer system is a sufficient amount of the operating system to maintain a continuity of action. That is, a portion of the operating system. I/O storage areas, and other basic routines are maintained in the memory of the machine. In addition, the active portions of user programs and data sets are available as well. The portions of the operating system and user programs and data that are not active are kept on the next level of the hierarchy, the secondary store.

The purpose of the secondary store of this hierarchy is to maintain copies of all of the programs and data needed by the computer. Generally this will be a disk, although it could be any block-oriented storage device with a large capacity. Such devices have been built with charge coupled devices, bubble memories, and large RAMs. This device is generally organized into files, and maintaining the files is one of the responsibilities of the operating system. In addition to the files, there is an area which is used to maintain the current copies of user program space; this area is often called "swap space." The swap space is also under the control of the operating system. The procedures and mechanisms established within the computer system to manage the use of the memory system are done so to effectively utilize the available system resources. With a combination of software/system policies and the appropriate hardware, only copies of currently active information need reside within the main store of the machine at any given time. Still, the apparent effect is that user programs execute in "virtual space," which frees up the user from being aware of the exact physical configuration of the system and the orientation of his program.

The T_A of information on the disk is much longer than the T_A for main store. Note that the cache is created from a (relatively) small amount of high speed RAM; and main store is also electrically and randomly accessible, but with lower cost, slower devices than the cache. Secondary store, on the other hand, involves electromechanical devices, and therefore requires relatively long times to find the physical location of the information and effect the **transfer.** The ratio for $T_{A_{\text{MAIN STORE}}} / T_{A_{\text{CACHE}}}$ is on the order of 10, but the ratio for $T_{A_{\text{SECONDARY}}} / T_{A_{\text{MAIN STORE}}}$ is on the order of 100,000. For this reason, when **information** is needed by the processor. and it is not in main store, the operating system will request the **needed** information, place the current **task** on a queue, and get a new **task** to execute while waiting for the information to be retrieved from the disk. This action of "context switching" allows the processor to be **shared** effectively between multiple programs; such systems are often called multipmgrammed or time-shared systems. To be effective, the secondary storage system must be sufficiently large to handle the swapping functions and the necessary file system operations.

The last member of the hierarchy shown in Figure 7.3 is the extended storage. This consists of information stored on magnetic tape, which is slow in comparison to the disk storage. This storage is generally used for permanent storage of programs and data, as well as transfer of information from one computer to another. Some systems have automated tape storage capabilities. so that parts of the extended storage can be considered a random access system with capabilities similar to the disk systems. albeit much slower.

The intended operation of the memory hierarchy is to provide a very large memory capability, with the response time of a cache system and the storage capability of a disk or tape system. The mechanisms used to perform these **tasks** is the subject of the following sections.

7.2. "Standard" Memory Systems: Random Access Storage for Programs and Data

The storage of information in computer systems is accomplished by utilizing collections of individual storage elements, each of which is capable of maintaining a single bit. Thus, for a device to be useful as a memory element it must have two stable states, a reliable mechanism for setting the device to one state or the other, and a mechanism for interrogating the state. Memories have been built of a variety of devices that match this characteristic, including relays, individual vacuum tubes, storage tubes, and delay lines, which form a type of serial memory. In each **case,** information in the form of bits was entered into the memory, and then at some later time extracted for use by the system.

Storage tubes and delay lines allowed for **information** storage in some early machines, but the **central** memory technology next used by most computers utilized the magnetic properties of iron. The mechanism utilized by these memories is depicted in Figure 7.4. **A ferrite** material is fashioned into a circular, doughnut shape, as shown in Figure **7.4(a).** The principle utilized by this **device** is the fact that the magnetic orientation of the fenite material will change to coincide with a forced magnetic field, if the field is **strong** enough. Due to the physical **nature** of the material, once the magnetic orientation has been established, it will remain in that orientation until a different magnetic field is created to change it. This is shown by the **flux-versus-current** diagram of Figure **7.4(b),** which is known as a hysteresis **loop.** When the **current returns** to **zero,** the orientation of the flux remains in the diition that it was established. It will remain in that orientation until a **current** is passed through the drive **line** in the opposite **direction.** The residual magnetic flux within the core is used to store a single bit. If the flux is aligned in one direction, the bit is a zero; alignment in the opposite direction represents a one. The use of the core for the storage of information **requires** at

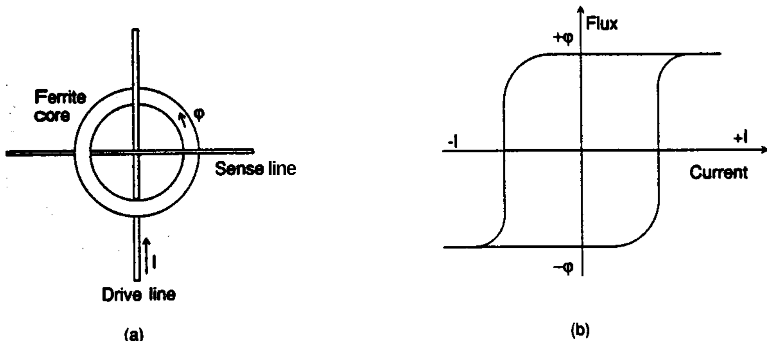


Figure 7.4. Magnetic Memory Mechanisms (a) Magnetic Con for Single Bit. (b) Hysteresis Loop.

least two wires passing through the center of the core. One line is used to write information into the core:

- Forcing current in the drive line as shown in Figure 7.4(a) will create a flux, φ, as shown: let this orientation represent a one.
- Forcing current in the wire in the opposite direction from that shown in the figure will reverse the orientation of φ; let this orientation represent a zero.

The other wire is used to sense the content of the core. Note that establishing the orientation of the magnetic flux can indeed represent binary values; but we not only need to establish the value, we also need to retrieve the value stored in the core. The sense line uses the fact that a wire in a changing magnetic field will pick up a voltage: that voltage is sensed to identify the content of the core. The process works in the following manner:

- A negative current [opposite to the direction shown in Figure 7.4(a)] is established in the drive line; the net result is to leave the core in an orientation representing a zem.
- If the sense line detects a voltage, then the magnetic field is changing. and hence the bit represented was a "one" before the process started.
- If the sense line detects no voltage, then the magnetic field is not changing, and the bit represented by the con was a "zero" before the process started.

To produce information needed by a processor, these characteristics are utilized by core memories in the following fashion:

- **Read:** To read the value stored in a bit. a current is sent through the drive line of the core for that bit Assume that the current direction is that which establishes a "zero" in the core. If the new magnetic orientation agrees with the established orientation. no change is made and the sensed voltage is zero, which corresponds to a "zero" bit. If there is a change in the magnetic orientation, then a nonzero voltage is created on the sense line. which corresponds to a "one" bit. In either case the bit representation of the magnetic flux at the end of the read is a **"zero."**

The overall effect is to **destroy** the data stored in the core, and so core memories are destructive readout devices. This is generally an unacceptable feature, so the value read out is stored in a register and immediately **returned** to the core. For this reason cores generally have $T_{CY} = 2 \times T_A$ since the data must be restored to the accessed location.

- ***Write:*** The first step in the write sequence is to place known data in all of the cores that will be used for the write: this places a constant. known value in the core. The value could be either a "one" or a "zero," but we will assume that it is a zero. This is not absolutely necessary, but is usually combined with the electronics used for the read cycle: the first half of the read **process** above performs this function. Current is then directed to the drive lines on those bits which will have a "one" orientation. while current is inhibited from the cores for those bits that need to maintain a "zero" orientation. In this fashion, the correct orientation is established for the data to be written to the core.

This technology was used for many years to create the main memory for most computers. However, the cost and size of the memories, as well as their speed, became a disadvantage as semiconductor memories were developed. Each bit in the memory required a separate core, with at least two, but usually three, wires through it. The technique of storing information by the magnetic orientation of a ferrous material is now used more prevalently for other types of storage than for the central memory ot a computer. The magnetic orientation of a region of ferrous material on a surface is used to store a bit. and this surface is most often on a rotating magnetic disk, or on a magnetic tape. The reading of the information still **requires** a moving magnetic field, but in a disk or tape unit the movement of the field is caused by physical movement between the surface and a detecting element called a **head.** The head is also used to create the proper fields for writing the information to the magnetic surface. Disk units are utilized to store thousands of bytes, such as floppy disks on a personal computer, to billions of bytes on larger machines. ***Tapes*** have a similar range of storage abilities, and are used on computers of all sizes.

Different types of **electronic** technologies have been used to store information in computers, from tubes to semiconductors. At one level we can examine the storage mechanism by **looking** at the gate level; another level is the device level. Figure 7.5 shows two different gating implementations for storing a single bit These can be cascaded into several bits to store bytes or words. One method of maintaining a bit is to put it into a latch, as shown in Figure **7.5(a).** The simplest gating arrangement to store a bit is cross coupled gates, and these are shown in the figure. The information placed in these gates is established by the input (D) when the enable line (ENB) is asserted. As long as the enable line is **asserted,** whatever information is on the data **line** will be passed to the storage element. When the enable line is **deasserted,** the last value for the data will be retained. This behavior is useful for many computer functions, and can be used to store **information** when needed.

The latch behavior is not the most prevalent mechanism used in storage elements in a processor. The gates shown in Figure 7.5(b) implement an **edge-triggered** function, the behavior generally associated with a register. The **mechanism** shown in the figure is used to capture the value of the data (D) on the rising edge of the clock (CLK). Analysis of the gates implementing the latch is

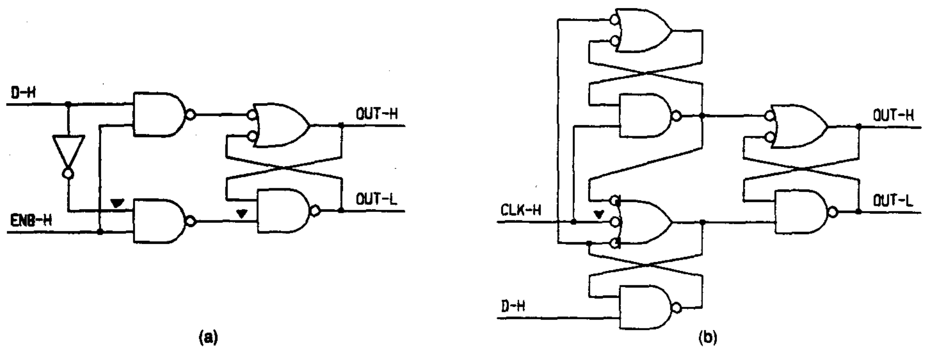


Figure 7.5. Possible Arrangements of Gates for Storing a Single Bit. (a) Latch (Single Bit). (b) Edge-Triggered Flip-Flop.

relatively straightforward. but the register function is very involved. However, in both cases the data must be stable for some window of time around the active edge of the clock (or enable). If this condition is not true, the unit can enter a metastable condition that will cause problems in high speed systems.

The circuitry shown in Figure 7.5 requires many individual transistors or other active devices to create. Therefore. they are used in small numbers in places where the storage requirements are not extensive. Creating enough register or latch type circuits in an integrated circuit ro store a lot of information would not be a good use of silicon real estate. Two types of mechanisms for storing information in semiconduction memories are shown in Figure 7.6. Figure 7.6(a) shows an arrangement of parts that implements a static memory cell. As in the case of the latch, there is cross coupling between the elements, and the device has two stable states. The active action of the system makes sure that the value of the cell remains as set until an external event causes a change. Thus, a value written to this cell will be maintained until the power is lost, or until the contents is changed by the write action. In this it differs from core memories, since it is not a destructive readout mechanism.

Static memories generally have a smaller number of bits per package, and a higher power consumption, than dynamic memories. The static mechanism of Figure 7.6(a) requires six transistors in every cell; other static memory configurations utilize fewer active elements. One of the tasks of memory designers is to reduce the number of components needed in an individual storage cell, since fewer elements means that each individual cell can be smaller and require less power, which in turn leads to larger memories. The memories with the largest capacities use not a static mechanism, but rather a dynamic mechanism, as shown in Figure 7.6(b). Here the value of the bit is not determined by the current flowing through one of two different paths, but rather the bit value is determined by the amount of charge stored on a capacitor. The capacitor is created with semiconductor technology, and is extremely small. The sensing of the charge is also very difficult, and handled by circuitry on the device itself. The information is placed on the capacitor by opening an electronic gate and establishing the proper charge level. Then, the gate is closed, and the charge maintained on the node by electronically isolating it from surrounding influences. However, the time which the charge can be reliably maintained in this manner is not long,

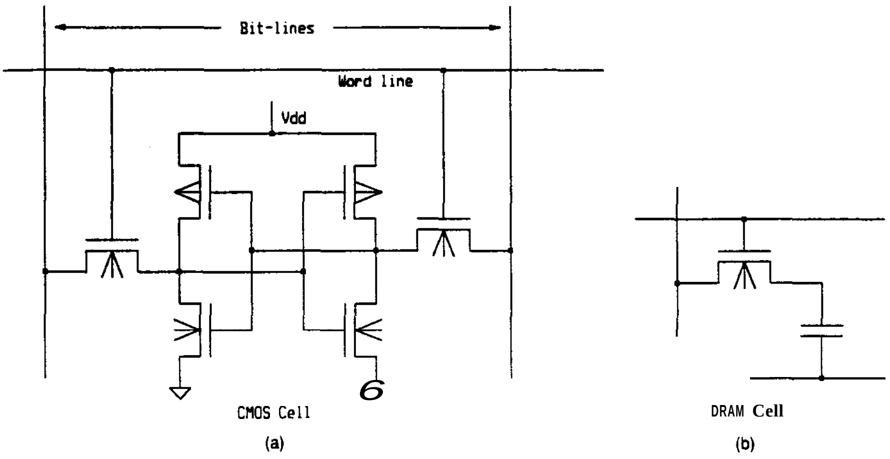


Figure 7.6. Bit Storage Elements for Semiconductor Memories. (a) Static Memory Cell. (b) Dynamic Memory Cell.

and so it must he re-established periodically. This is done by a "refresh cycle. which detects the appropriate bit values and refreshes the hits. The length of time between refresh cycles varies from memory to memory, but a common value is 8 msec: each row must be visited at least once every 8 msec. For this reason dynamic memory controllers are designed to periodically access rows to assure that the data is maintained in the memory cells.

The storage of the information in the cells is only a part of the memory problem. The bits stored must be organized in a reasonable fashion to access the information. The two most prevalent mechanisms are random access and serial access. As the name implies, random access memories are organized such that the information can be accessed in a random fashion. That is, each location has the same access penalty, T_A, and the order of access can be entirely random. The only requirement is a mechanism to decode an address of a specific location, and a data path such that any location accessed can provide the necessary information.

On the other hand, serial access mechanisms are organized such that the data is written and accessed in a serial fashion. Thus, the T_A varies depending on the location of the information in the memory, since the data must pass a mechanism for reading each bit. Examples of serial access devices include magnetic surface systems, such as tape and disk, and serial semiconductor systems, such as shift registers and charge coupled devices.

A simplified block diagram for the random access mechanism is shown in Figure 7.7. The size of the decoding mechanism is dependent upon the size of the array of memory elements; the number of bits in memories increases each year. The mechanism used to decode the address can be designed in a variety of ways. The two most basic mechanisms are the one dimensional (1-D) and two dimensional (2-D) decoding schemes. The I-D scheme accepts an N-bit address. and uses an N to 2^N decoder to identify one of 2^N individual elements. The location identified is then used in the read or write operation. The 2-D scheme accepts the

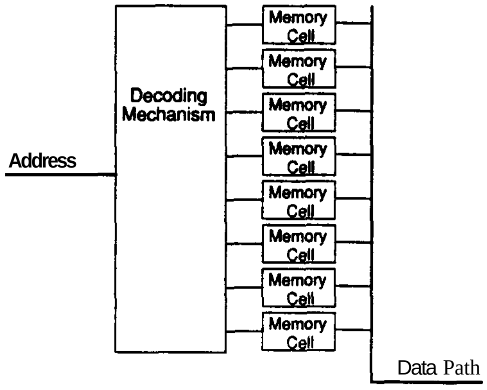


Figure 7.7. Random Access Memory Block Diagram.

N-bit address and divides it into two groups, which we will call X and Y. Thus, $X + Y = N$. These two groups of address lines control X to 2^X and Y to 2^Y decoders, which jointly specify a single element. Note that the memory cells used with the I-D arrangement need only have a single enable line, while the memory cells used with the 2-D arrangement need two enable lines. Thus. the I-D arrangement has a simple cell and a more complicated decoding scheme, while the 2-D arrangement has a slightly more complex cell, with less logic required in the decoding mechanism. These methods are depicted in Figure 7.8, which shows the addressing mechanism for an array of eight cells in a 1-D decoding arrangement, and sixteen cells in a 2-D decoding arrangement. Mechanisms used by manufacturers internal to memories include both the I-D and 2-D methods, as well as other variations of the schemes. Note that there is no reason to stop at two dimensions. and higher mechanisms could be useful in some systems.

The basic ideas of the preceding paragraphs apply not only to individual bits, but also to collections of bits. That is, many memories are not organized as 1-bit entities, but rather some multiple that makes logical as well as manufacturing sense. Memories containing Cbit words are very useful for storing BCD digits, and for use with 4-bit microprocessor systems. Memories organized 8 bits wide are useful for ASCII characters, 8-bit microprocessor systems, and byte-addressable memory systems. Combinations of 4-bit and 8-bit systems can be used as needed to meet other system needs. In dealing with the memories or other storage elements, the principles used in identifying a bit in a memory array can be applied. That is. the individual components can be organized in a one dimensional fashion, a two dimensional fashion, or in some combination of the above schemes.

> *Example 7.1: I-D Design of a register set:* Design a register array that contains eight registers, and that operates with an 8-bit bus. The array should have two control lines, a read line and a write line. Use individual registers in the ALS technology, and a one dimensional addressing scheme. How fast can information be made available on a read? What is the data requirement for a write?

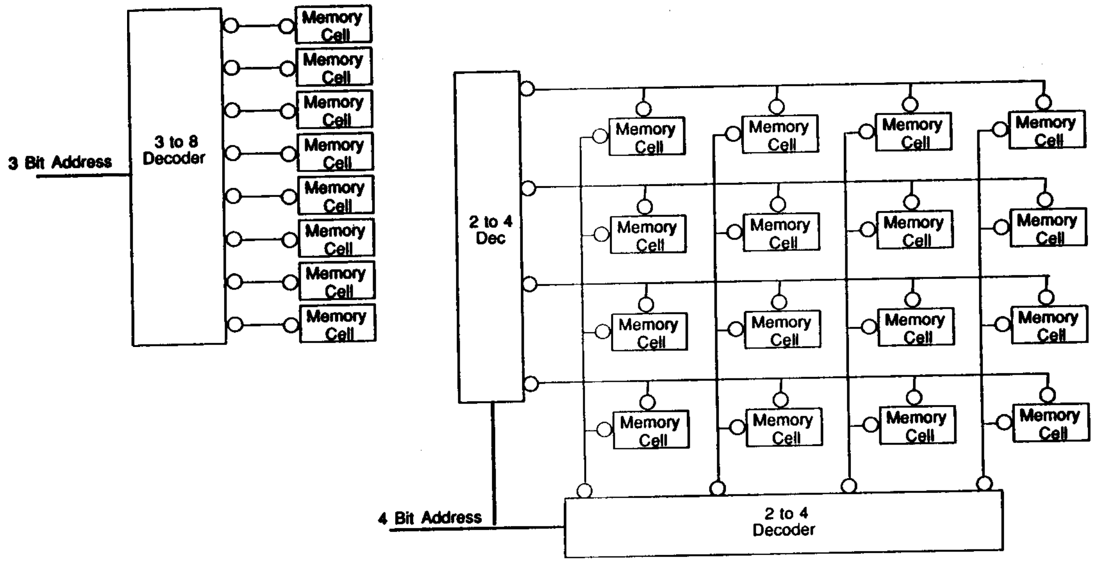


Figure 7.8. Memory Cells Organized in 1-D and 2-D Memory Arrays.

One solution to this is shown in Figure 7.9. The register selected here is the 74ALS299, which has inputs and outputs on the same pin. The inputs not shown in the figure have been appmpriately disabled. With the enable lines (G1, G2) tied low, this device will output its information when the function select lines (S1, SO) are both low. When the function select lies are both high, the outputs are disabled, and a value can be accepted from the bus to the internal register. The address is decoded by two 74ALS138s. When the read line is activated, the function select lines of the appmpriate register are asserted. The delay from assertion of the read line to the output data stable is the sum of the enable-to-output-stable delay of the 74ALS138 and the function-select-to-data-stable delay of the 74ALS299. The sum of the maximum times is 39 nsec; typical times would be shorter. When the write line is asserted, the clock line of the appropriate register is activated. The loading of the register occurs on the low-to-high transition of the clock at the register, which corresponds to the high-to-low transition of the clock line in the figure, since there is a change of assertion level through the 74ALS138. The maximum delay through the 74ALS138 is 17 nsec. and the

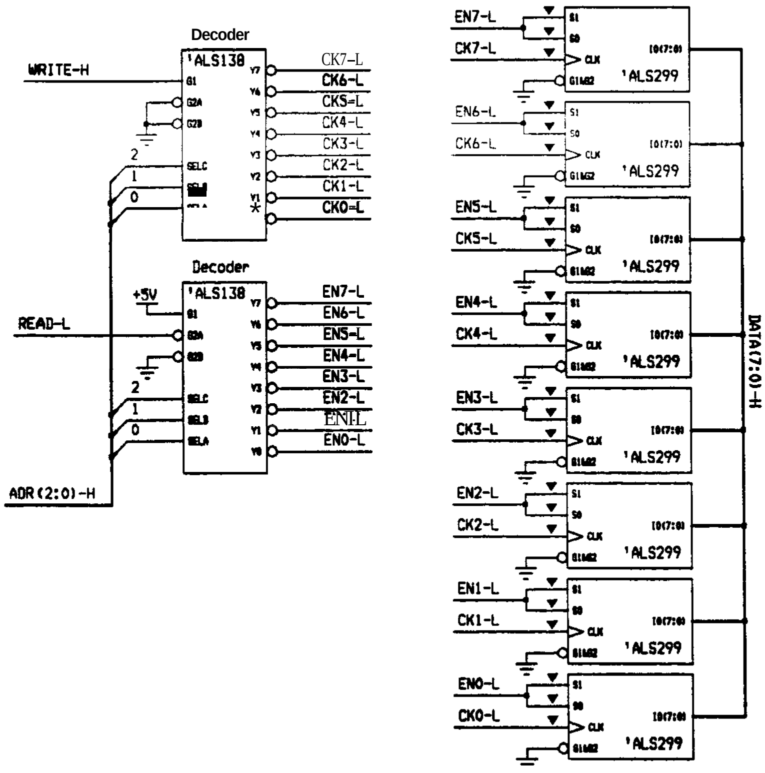


Figure 7.9. Registers with One Dimensional Addressing.

data must be stable on the bus for 16 nsec prior to the rising edge of the clock (at the register).

The arrangement of the pans as shown in the figure gives a register bank with eight registers in ten DIP packages, and a power consumption of about 1.6 watts. This is not a very efficient use of the board space or system power, but the unit can meet some requirements for special systems.

The use of individual registers as shown in Example 7.1 can be used to meet some special requirements, but the normal manner of operation is to use memory elements that contain a larger storage capacity. Nevertheless, the same principles apply, and the memories can be organized in a one dimensional or two dimensional manner.

Example 7.2: 64-Kbyte static RAM system: Design a memory system for an 8-bit microprocessor system. The memory system is to contain 64 Kbytes of static RAM memory, using 8K x 8 RAMs, such as the μPD4464 from NEC Electronics Inc. Do this design in two ways, first as a one dimensional scheme, then as a two dimensional scheme. Communication lines to the memory include the address and data buses, a write line and a read line. Write and read lines are asserted low.

The 4464 is an 8K x 8 RAM with thirteen address lines, two enable lines, an output enable. and a write line. One of the enable lines is asserted high. while the other is asserted low: the write line and the output enable are both asserted low. To attain the 64K space, eight separate memories are required. Figure 7.10 shows one of the possible I-D organizations that can be used. The lower address lines are shared by all memories; the current requirements of each input is only I μa, which does not cause loading problems. The upper three address lines are directed to a 3-to-8 decoder ('138), which enables only one of the memory chips. This allows sharing of all of the read and write lines. as only one memory element will be active at any one time. The burden is on the user of the system to be sure that the address lines do not change while the write line is asserted; such action will cause the data to be corrupted in the memory.

Note that this arrangement can be extended to include more memories by utilizing the unused enable lines of the 3-to-8 decoder. That is, additional decoders combined to make larger decoding systems (4-to-16; 5-to-32; etc.) can be used to make larger 1-D memory systems.

The two dimensional implementation is shown in Figure 7.11. Many of the characteristics are identical: the read and write lines are shared between all of the memories; the 13 least significant address lines are common to all memories, and the data bus is used by all chips. However, two 2-to-4 line decoders ('139s) are used to implement the decoding of the most significant address lines, instead of a single decoder. The use of the '139s allows for doubling the size of the memory (not shown) without the addition of more decoding capability. If the larger capability is not needed, then the function of the second '139 can be filled by an inverter.

The systems shown in Figures 7.10 and 7.11 demonstrate the use of decoders to make one and two dimensional systems. However, certain design criteria have not been considered in the discussion that must be taken into account in designing a specific system. For example. the time from chip enable to output valid for the μPD4464 is twice the time required for

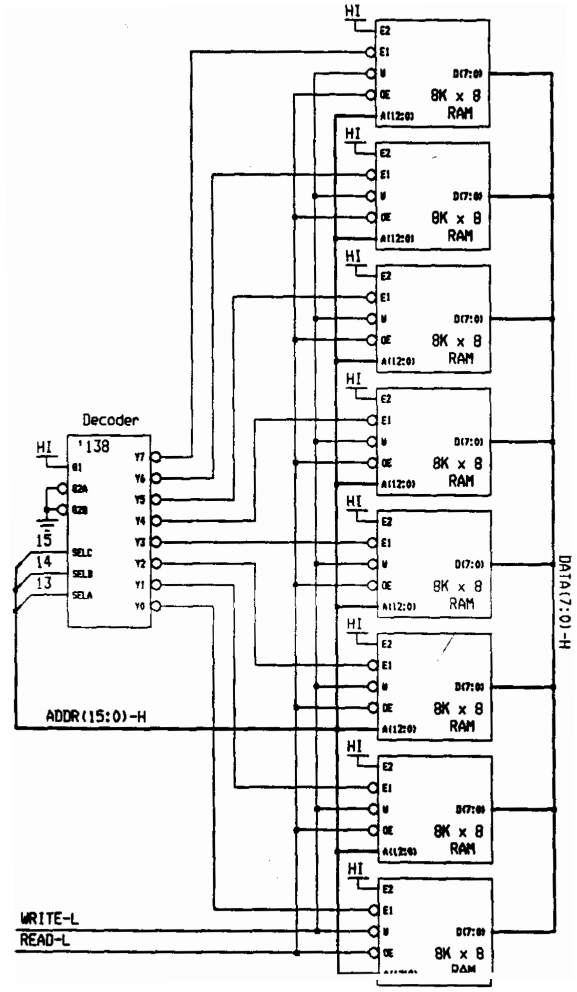


Figure 7.10. 64-Kbyte Memory Array from 8K x 8 Memories; 1-D Organization.

output enable to output valid. If the speed of the system is critical, then a different arrangement may be desired, one in which the chips are enabled all the time, and the output enable lines and write lines activated as needed to perform reads and writes. This changes the configuration of the system, since the read and write lines can no longer be common to all *memory* chips, but the same basic concepts are still applicable to the memory.

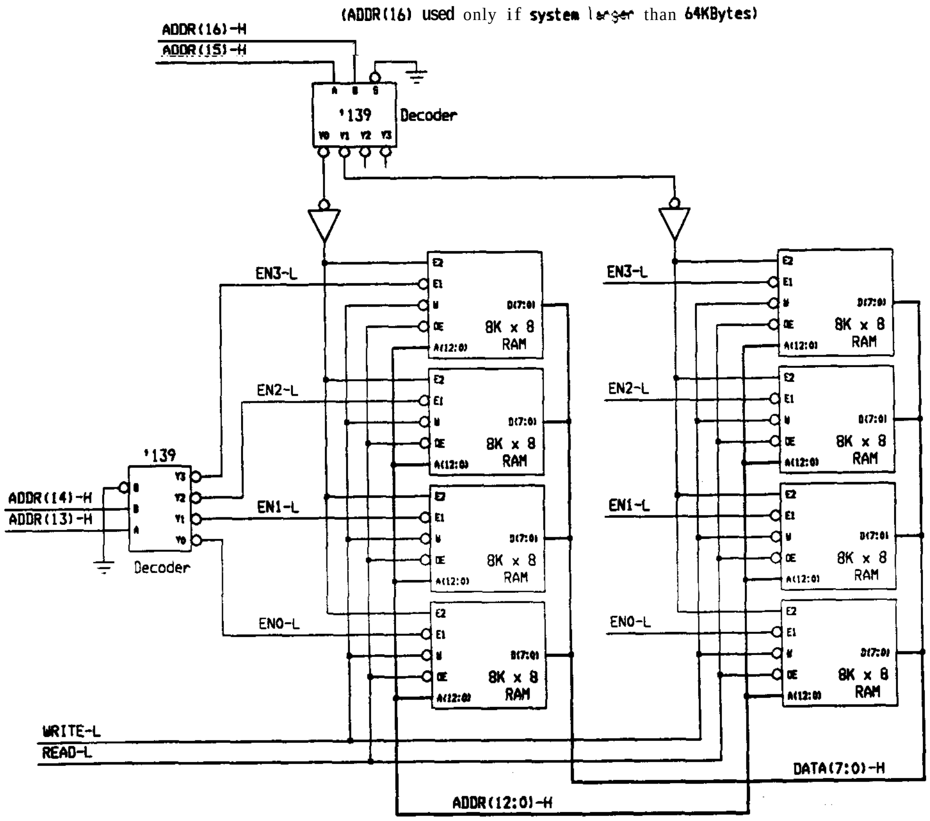
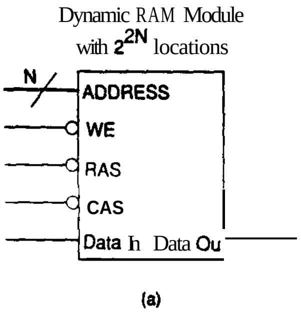


Figure 7.11. **64-Kbyte** Memory Array from **8K** x 8 Memories; **2-D** Organization.

The concepts involved with the address mechanisms are not limited to the examples examined above. Consider the following example, in which **dynamic** memories are used to **create** a **large** random access memory. The addressing mechanism for identifying the appropriate memory module to activate is **1-D** in nature, while **the actual** memories involved are selected in a 2-D fashion.

***Example* 7.3:** *Dynamic memory system:* Design a memory to be used with the time multiplexed **address/data** bus of the NS32332 shown in Figure 6.10. The address will be supplied on the bus during **T1, along with** a data **direction** indication **(DDIN)**. The memory should **respond** to the **assertion** of the address strobe(ADS-L) by initiating a memory request. If it is a write, the data will become **available** during T3; if it is a read, the **data** should be supplied as soon as possible, but no later **than the beginning** of T4. Use dynamic memories to provide as **large** a memory space as possible.

To discuss a system with dynamic memories, first let us examine some of the mechanisms of dealing with the memories. There are a number of device-specific characteristics, but the basic cycle for a read in a dynamic RAM is shown in Figure 7.12(b). Usually, large RAMs such as the dynamic RAMs shown here require so many address lines that the address divided into two parts and time multiplexed on a single set of address lines. These two parts of the address are called the "row address" and the "column address." After the mw address is presented on the address lines for a required period, the mw address strobe (RAS) is asserted. The address is held for a short time, then changed to be the column address. After a required setup time, the column address strobe is asserted (CAS), and the memory access begins. Some time later, which is the *access* time of the memory, the data becomes valid (T_{DATA}). When the RAS is released (T_{REL}), the output data will return to the tri-state condition. A write requires the same operation. but the write enable line is asserted during the operation,



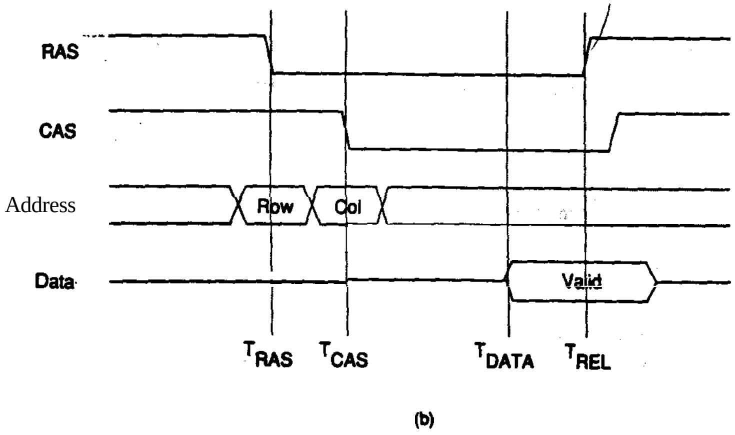


Figure 7.12 Dynamic RAM. (a) Symbol. (b) Timing f a Read Cycle.

and the data is asserted on the data lines by the module requesting the write action. As long as the rows are accessed every $T_{REFRESH}$ the information should be maintained. Thus, one of the design requirements is to access each row within the refresh time, which in many memories is 8 msec.

These individual packages can be combined in reasonable ways to be used in systems. For example. for bused systems the data in and data out pins can be tied together and connected to the system bus with transceivers. One commonly used configuration is to put nine individual memory modules on a single in line package (SIP), which is sufficient for a byte plus parity. This SIP module is used in this example; each SIP contains 256 Kbytes of information. SIP modules that contain 1 Mbyte and 4 Mbytes of information are also available.

Using modules with 256 Kbytes, an 8-Mbyte memory can be constructed with just 32 modules. Drawings of the memory are contained in Figure 7.13(a)–(c). A more complete set of drawings are found in Appendix B. Figure 7.13(a) contains the memory elements themselves and the data buffering transceivers. Note that the data lines are buffered from the bus system with a transceiver. Although the data line of each memory module does not present a large load, there are enough individual memory modules in the system to provide a nontrivial load. The buffer (transceiver) has the effect of isolating the loads from the bus and minimizing the effect of the wires required to carry the signals. Also note that the organization is such that the 4 bytes required for a 32-bit word (assuming that the word is aligned correctly) are all accessed with the same RAS and CAS line. Accesses of information nor aligned on a word boundary must use the proper set of lines, and this is the responsibility of the initiator of the transaction.

The generation of row and address strobes is done by drivers capable of supplying a sufficient amount of current, and these are represented in Figure 7.13(b). The selection of the appropriate megabyte is accomplished by a decoder, which is a 1-D technique. The most significant address lines are used to identify the appropriate megabyte; expanding to a 16-Mbyte address space would require an additional decoder. The decoding is done by a '538, which is chosen for two reasons. The first is that the assertion level of the output is selectable, so that the assertion level is selected to match the gates that follow the decoder. In this case, the gates that follow are '801s which were selected for their drive capability: each RAS and CAS line has 36 individual memory modules attached to it (32 data, 4 parity). The second reason for the selection of the '538 is that, with the proper activation of the control signals. all of the outputs can be asserted simultaneously. This is very useful to allow all 8 Mbytes to perform a refresh cycle at the same time.

The address latches are also included in Figure 7.13(b). The latches accept the address during the first cycle of the transfer, and the address is then broken into three groups: nine bits for the row address, nine bits for the column address, and the most significant bits for identification of the active megabyte. The two least significant bits are not included, since the system is byte-addressable, and these bits merely identify the appropriate byte. Since all 4 bytes are accessed on every cycle, the least significant address lines are not needed here. The 9-bit row/column address bus (RC_BUS) is then presented to four sets of high current drivers. which have the capability

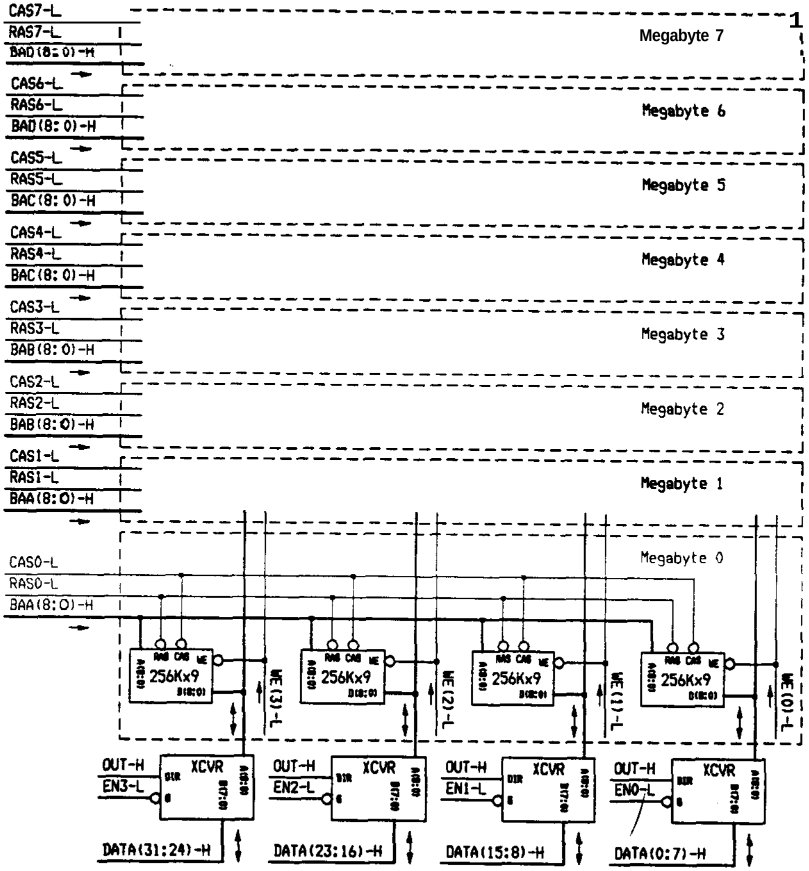


Figure 7.13(a). 8-Mbyte Memory System: Memory Array and Data Buffering Transceivers.

of providing the current needed by the collection of memories. The outputs of the high current drivers are conditioned by damping networks to minimize the undershoot and overshoot, which will occur when switching the lines between high and low logic levels. The address lines of the individual memories are supplied from the damping networks by four separate sets of lines (BAA-BAD). This buffering is required to provide sufficient drive capability, since each address line (BRA(0)-H, for example) supplies the address to 72 separate memory modules. Also included is the mw counter that identifies the appropriate address for refresh. These memories refresh two mws simultaneously; so the 0 line is not involved in the count.

The control logic shown in Figure 7.13(c) coordinates the assertion of all of the signal lines. The coordinator of all of the work is a state machine

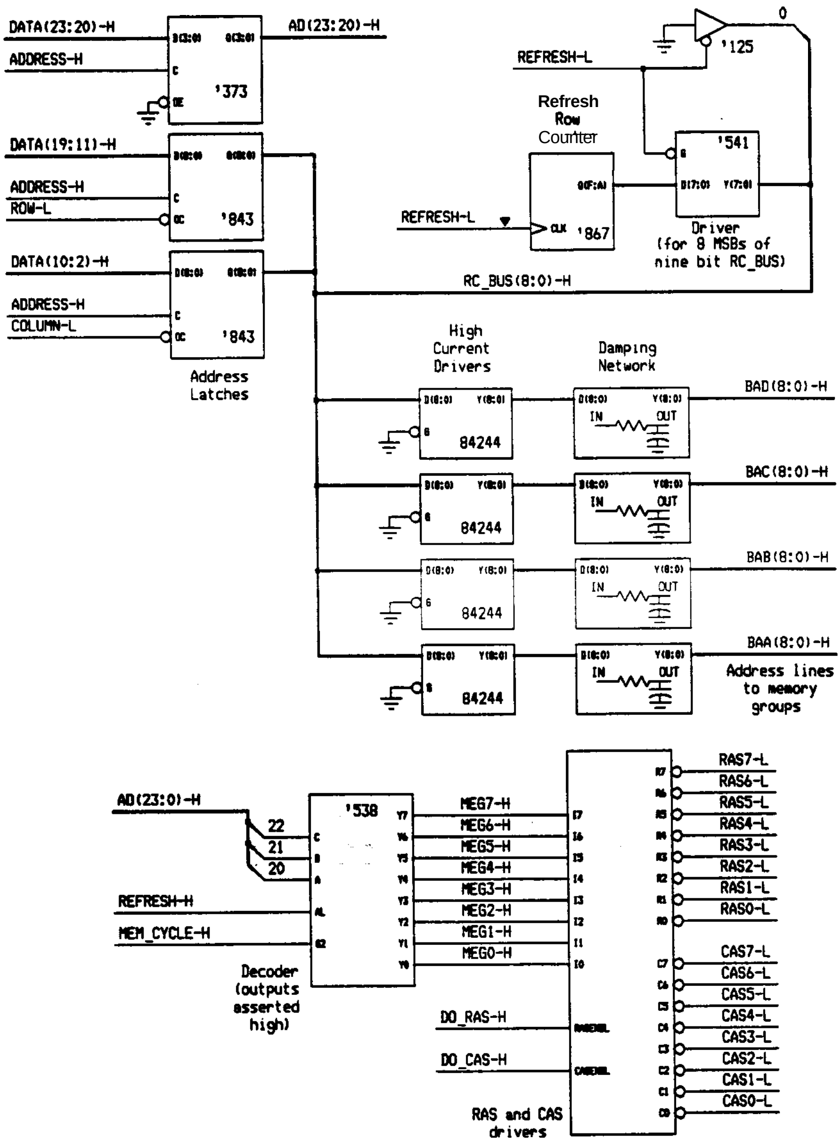


Figure 7.13(b). 8-Mbyte Memory System: RAS and CAS Logic; Address Latches and Drivers.

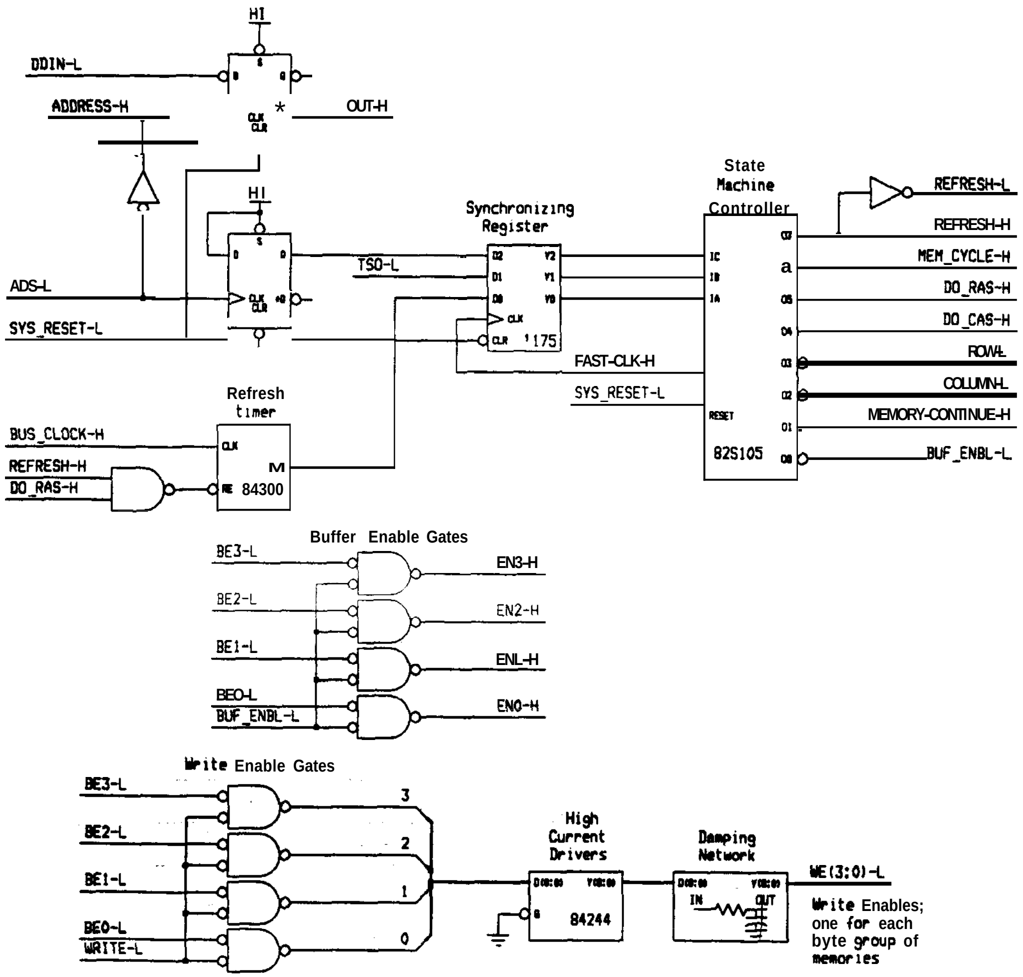


Figure 7.13(c). 8-Mbyte Memory System: Control Logic.

controller (82S105), which has the responsibility of asserting the signals in the order explained above. It is driven by a clock (FAST_CLK), which is four times faster than the bus clock(BUS-CLOCK) and synchronized with it. Thus, the bus clock and the fast clock are generated externally and supplied a the memory system. Using these two clocks in this manner allows the signals to be created in a timely fashion.. The refresh counter is connected to the bus clock, which it counts down to identify the proper time to do a refresh. When the refresh is needed, it sets an internal flag that provides an input to the state machine; when the refresh is recognized the flag is reset.

The other inputs to the state machine are a flag to identify the start of a memory cycle. and the signal TSO. which comes from the timing unit of the microprocessor to identify the end of a cycle. The outputs of the state machine are used to assert RAS. CAS. and the other signals associated with the dynamic RAM. The signal MEMORY-CONTINUE is used to inform the rest of the system that the memory information is ready. This is necessary since the RAM may be in the middle of a refresh cycle when the system makes a memory request.

The two sets of gates in Figure 7.13(c) are to assert the write enable lines and the data transceiver enable lines at the appropriate time. The time is identitied by the state machine controller. hut the appropriate byte is identitied by the byte enable signals generated by the processor. This 7-D mechanism chooses the appropriate bytes. One dimension is provided bv the address: the second dimension is supplied by the processor. Thus. the processor must assume the responsibility of reading and writing information that is not aligned exactly on a 32-bit boundary.

A photograph of such a system is shown in Figure 7.13(d). This sys-tcm contains 8 Mbytes of memory and a 32032 system.

The concepts discussed in this section are applicable to a wtde range of memory organizations and considerations. For moat processing done by general purpose computers. random access is required to the memory. This is true of core memories. semiconductor memories. and other technologies as well. Hence. the individual elements must be individually addressable and accessible through the

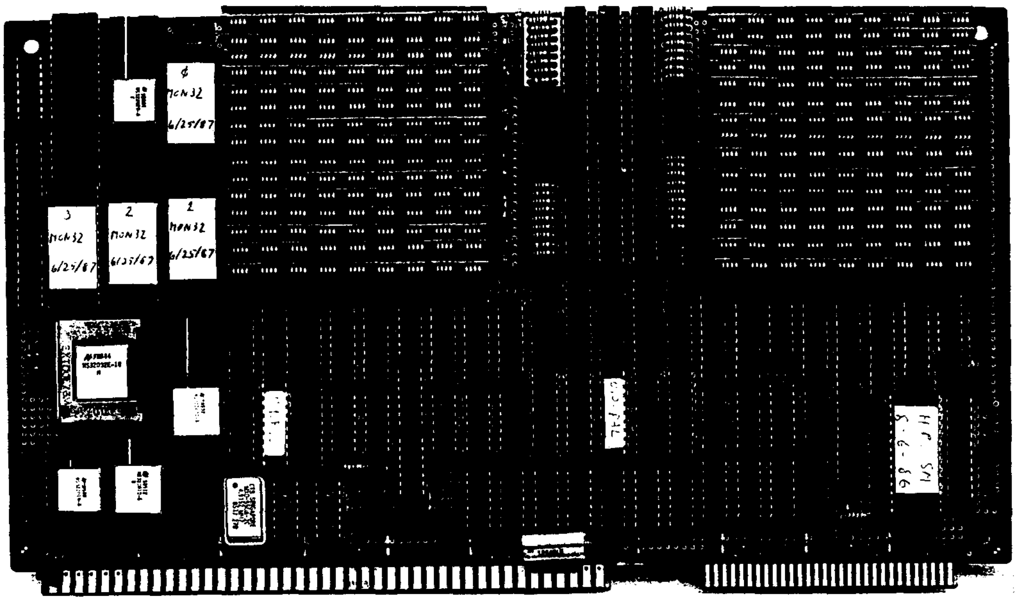


Figure 7.13(d). 8-Mbyte Memory System: 32032 System with 8-Mbytes Dynamic RAM.

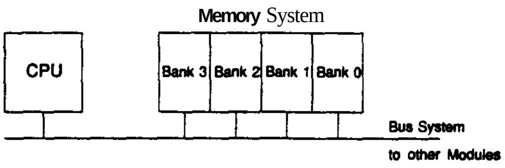


Figure 7.14. CPU with Four Bank Interleaved Memory System.

bus structure. The organization of the memory to access the addressed location can be done in a 1-D, 2-D, or related fashion, as long as only one location is actually enabled.

Creating larger memories. or memories with differing characteristics, can be accomplished by combinations of the mechanisms discussed here. For example, one of the figures of merit for a memory system is bus bandwidth, represented in bytes/sec. Since information can be transferred over a bus structure faster than it can be retrieved from a large memory, one of the ways to increase the bandwidth is to create memory in banks, and interleave the memory banks in time. Consider the system shown in Figure 7.14. The memory requests are sent to all four banks. and the response sent to the processor in different time slots. If the bus width is 4 bytes, and the memory access time is 200 nsec, and if 4 bytes can be sent every 50 nsec, then all four banks can be kept busy (assuming that there are sufficient requests). But each bank is individually organized as a random access system, and interfaces to the bus system in a manner which will allow the transfers to occur in a reasonable fashion. This requires more circuitry, but speeds up the overall data rates.

The choice of a memory organization and the technology in which it is implemented must reflect the constraints of the entire system. The choices will be based on optimizing performance for a given set of resources. If a major requirement is speed, then the designer can afford to put more resources (silicon real estate, board space, power, etc.) in the memory to provide for a minimum response time. If the critical resource is power, such as a battery operated system. then the complexity cannot be increased, and the parts and design mechanisms are optimized for minimal power consumption. Nevertheless, the system architect can choose from a variety of memory and processor organizations to create a system that will fit a particular need.

7.3. Virtual Memory Systems: The Illusion of a Memory Space

One of the principal tenets of stored program computers is that the program resides in a memory space, and the instructions are extracted from memory and executed. If the instruction calls for data manipulation, then the data is identified and utilized to perform whatever calculations are called for by the instruction. In most programs the data also resides in the memory, at least at the beginning of the program, when data is brought into the system, or at the end of the program, when data is prepared for output to the destination device. Thus, the program is loaded into the memory, started, and whatever data manipulations are called for by the program are executed. The program then terminates, and the system moves on to execute the next program.

The statements made in the previous paragraph reflect some assumptions **often made** about the use of the computer. Most machines used today have a collection of system facilities that we have come to call the operating system **(OS).** The operating system has the responsibility of doing many things, among them transferring programs and data to and from memory. When the program has been **loaded,** the operating system **starts** execution of the program at **some** predetermined point. However, most users of computer systems do not consider the effect of what the OS is doing; rather, they have a "model" of what the machine is, and they **are** operating under the assumption that the model is at least functionally **correct.** Such a model may appear as shown in Figure 7.15.

In multiprogrammed systems, we know that other programs will also be utilizing the machine, but generally we think of the machine as "ours," at least for the duration of our program. Knowing that a program will have a program section and a data section. we often think of the machine as shown in Figure 7.16. This simple block diagram shows only the memory and the processing capability. The possible connection between the two is identified by the instruction set. and using that instruction set we are able to perform work. where work is defined as manipulating data. The machine **as** seen in Figure 7.16 is what we think we have: hence, we call it a "vinual machine." In our mental model of the machine, the program starts at location zero and executes through the **instructions** in order. In the physical machine, the program was not loaded at location **zero;** rather, the operating system placed ***the*** information at a location which. for some reason, was available to **be** used. The operating system. then. is responsible for ascertaining what pans of memory are available: if no memory is available. then the OS makes some memory available. In a location known to the operating system there is kept a correspondence between the vinual space, which the program has the illusion of controlling, and the physical space, which contains the actual information **being** manipulated.

The mechanism used to define the relationship between the memory space that the program thinks it is **controlling,** and the actual memory locations being utilized, is called a "virtual memory mapping." The memory mapping mechanism

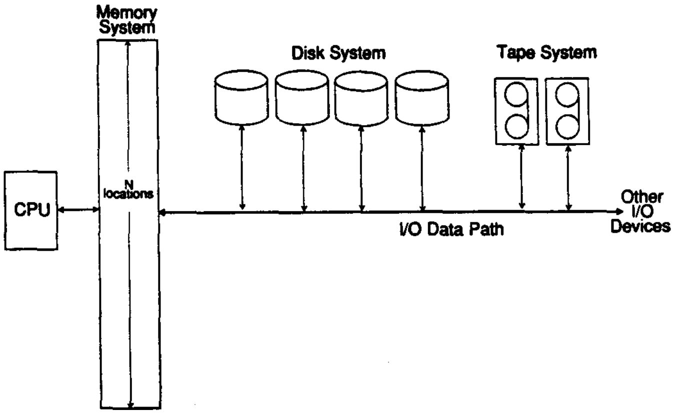


Figure 7.15. Block Diagram of a "Model" Computer.

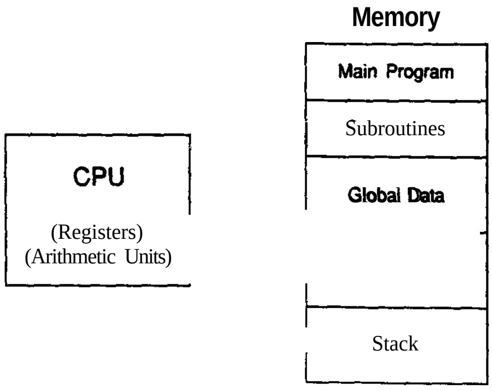


Figure 7.16. User Model for a Computer and Program

is therefore responsible for convening an address issued by the program (the virtual address) to an address that will be used by the memory system to access the information (the real address). Two common mechanisms often considered are segmentation and paging. and some systems utilize both concepts in their implementation. It is not our intention to discuss the pros and cons of one mechanism over another. That can be dealt with more effectively in a discussion of operating systems themselves [BrHa73, PeSi83, Deit84, BiSh88]. Rather. our interest is in the low level operations required to make virtual memory work.

Information is stored in real, physical memory, and, as such, it must be referenced with a valid memory address. However, within the executing program. references are made by the program in any one of a number of different ways. The addressing mechanism, be it a program counter reference, an indirect data reference, or any other method to specify a location in memory, identifies the target location in the virtual space of the program — where the program thinks the information is located. The memory mapping mechanism manipulates this (virtual) address in such a way that the proper location in memory (the real address) is accessed.

One simple mechanism that can be used to allow multiple programs to coexist in a memory, each executing in its own address space, is depicted in Figure 7.17. In this case the operating system has placed the various programs into a large memory, and it will keep a record of the base address for each of the programs. In addition. it will keep a record of the sections of memory not used, in order to accommodate other programs as needed. Then when Program 2 is to be executed, as in the figure, the OS will place the base address of the program into the base address register, so that all references made by the CPU to memory are made relative to the base location of the program. This is an example of register relative addressing. except that all references to memory are made relative to the base register. In this way, the virtual address of the program(the address the program thinks it is using) is translated to a physical address, the actual location of the information, by a simple addition. This mechanism will allow a large memory to be used by a system limited in some other way. For example, the instruction set architecture of the PDP 11 family of computers limits the size of a single

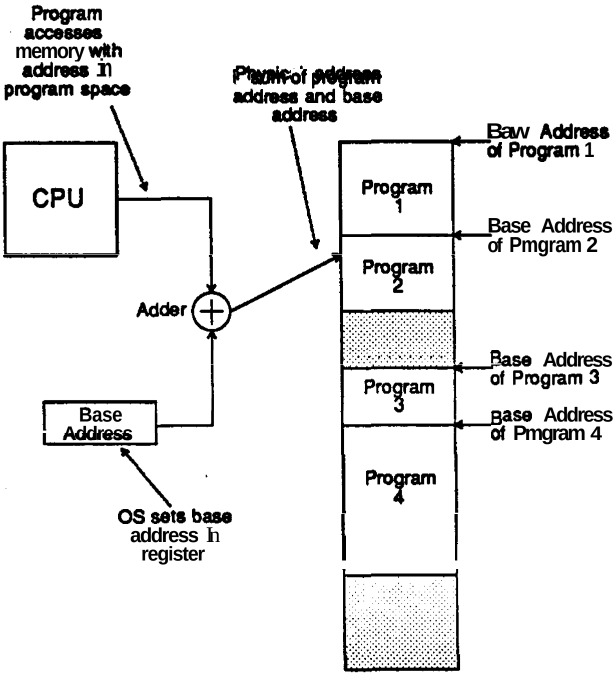
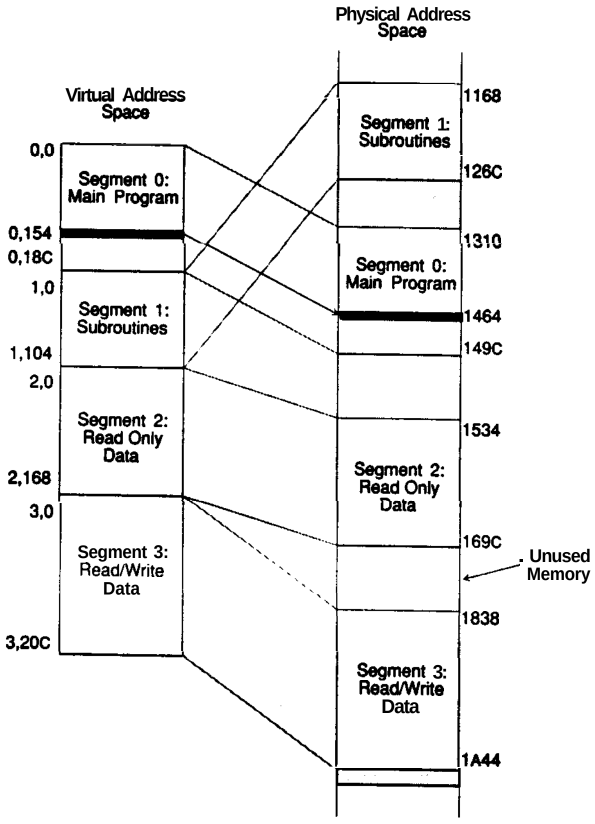

Program accesses memory with address in program space
Sum of program address and base address
Base Address of Program 1
CPU
Program 1
Base Address of Program 2
Program 2
Adder
Base Address of Program 3
Program 3
Base Address
Base Address of Program 4
Program 4
OS sets base address in register

Figure 7.17. Multiple Programs in a Single Memory.

process to 64 Kbytes, the amount of byte-addressable memory that can be reached with a 16-bit address. Yet using the scheme depicted in Figure 7.17, several such programs can exist in a memory that is much larger than 64 Kbytes. The operating system can share the resources of the system between the programs in a reasonable way. In this way, a multiprogrammed environment can be created, allowing programs to share processor and I/O capabilities of the system.

The above scheme considers each program an indivisible block, and must deal with the programs in that manner. However, an extension of the scheme is to divide each program into logical segments, and load the segments into their own sections of memory. This would correspond to the program model shown in Figure 7.16. Then, as the address was created by the system to access a particular piece of information, the address generated by the program would be offset by the value in the appropriate segment register, and the resulting address would be sent to the memory.

The above process can be visualized by considering a program which has been broken into segments, such as shown in Figure 7.18. The program represented in the figure consists of four segments: a main program segment. a subroutine segment. a data segment consisting of read only data, and a data segment with locations that can be both read and modified For this system, the processor will generate addresses consisting of a segment number and an offset within that segment. The memory management mechanism must then translate



Program Segment Table

Segment	Length	Address	Access
0	190	1310	Read. Execute
1	108	1168	Read, Execute
2	16C	1534	Read
3	210	1838	Read. Write

Figure 7.18. **Virtual Address Mechanism** with **Segmentation.**

this into the proper real address. The real address, $ADDR_{REAL}$, can be represented as the sum of the base of the segment and the offset within the segment:

$$ADDR_{REAL} = ADDR_{SEGMENT\ BASE} + ADDR_{SEGMENT\ OFFSET}$$

Thus the creation of the correct address in the system involves identifying the correct segment base and adding to it the segment offset. Part of the addressing mechanism is then to consult the program segment table (PST) for each access: the segment number identifies the element of the PST that refers to the desired segment and using that information, generates the correct physical address. The example shown in Figure 7.18 indicates that address 154 in the main program segment is convened to physical address 1464. The hardware of the system should make this conversion as quickly as possible. and at the same time check the legality of the reference. That is, does the address exceed the length of the segment? Or is the reference a write request into a read only segment? A number of systems include segmentation capability in the processor. the most prolific of which is the 80x86 family of microprocessors. Figure 7.19 gives a register level diagram of the microprocessor. in which the segment registers play a prominent part. Note that the segment descriptor registers work in conjunction with the virtual address generation hardwdare, and that jointly they can generate the required address. Thus, the mechanisms discussed are built into the hardware of the system.

If a virtual memory system is implemented by using segments, then the OS has the responsibility of maintaining the segments, and loading the segment registers with the addresses for the currently executing program. If another program is

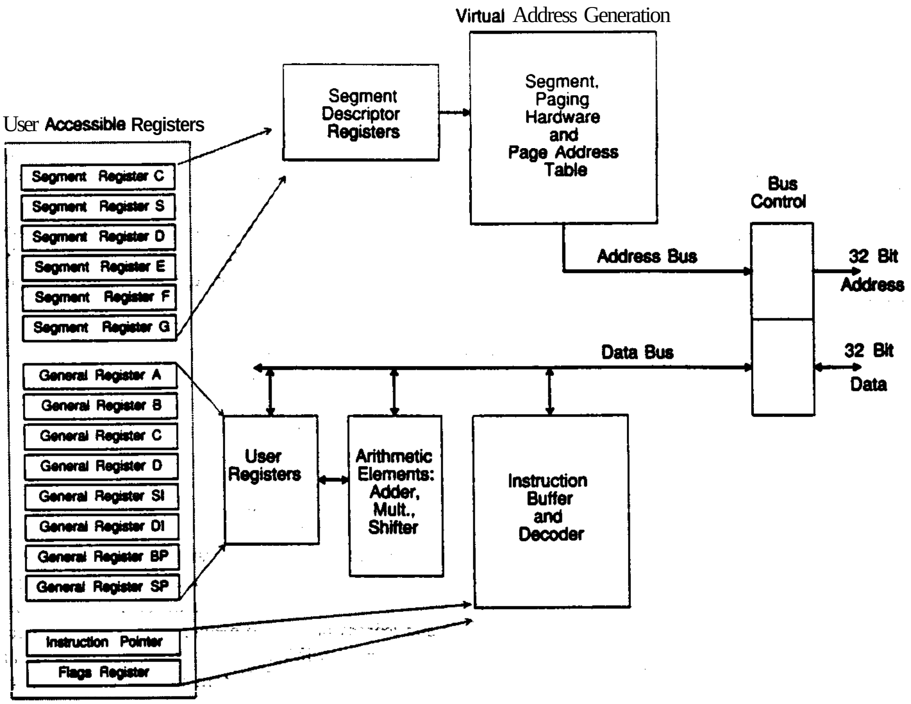


Figure 7.19. Block Diagram of a 80386 Processor.

needed, then all of the segment registers are changed appropriately. Note that this does not exclude the operating system from using the same physical segment for more than one pmgram. For example, the pmgram section of an editor may be needed by several users, and the operating system can be aware of this and set up the segment registers accordingly.

One of the mechanisms alluded to but not actually described has to do with the location of the programs when they are not in the memory executing. The secondary storage medium is used to hold the programs, or portions of the programs, until they are needed. This secondary storage is usually disk, but could be any storage area large enough to hold the entire pmgram, or collection of pmgram segments. One of the tasks of the OS is to control the use of the memory; that is, the programs or program segments that reside in the memory at any given time is determined by the *OS.* If a segment is needed during the execution of a program, and that segment is not in memory, then the *OS* brings in the information from secondary memory. In the process of doing so, it may be necessary for another segment. which is not currently active. to be returned to secondary storage. In this way the OS brings into memory the active programs and data, and those that are not currently active will migrate out of main store as the programs currently running need more memory space.

A segment is a logical entity, such as a pmgram segment or data segment. There is no inherent size of such an entity, so there is no standard size of segments involved in a computer system. Thus, the operating system must keep track of the starting address of the segment, its length. and other information that deals with access privileges. This information is shown as part of the program segment table in Figure 7.18. One ut the protectton issues to be addressed in a system is the containment of programs: a program must not be able to access memory, except as that specific privilege has been granted to the program. As a program requests information in a segment, the *OS* must make sure that the program should have access to that segment. When accessing the information in a segment, the program should be prevented from addressing information beyond the length of the segment. One way to enforce this is to include in the system bounds checking capabilities that compare the requested address against a given maximum. This will allow the system to protect the segments against unauthorized access.

Example 7.4: Memory mapping with segments: Give a block diagram level representation of an address translation mechanism involving segments. The address supplied by the processor consists of two values, the segment number and the offset within the segment. If an out of bounds request is made, the unit should issue an interrupt. The mechanism should be capable of keeping track of 16 programs, each capable of accessing 16 segments.

A block diagram of one solution to this problem is shown in Figure 7.20. The hardware logically sits between the generation of the addresses and the actual memory. The addresses are generated in pairs, consisting of a segment number and an offset within the segment. Before the program can run, the OS loads the appropriate segments into the actual memory, then sets up the addresses and lengths in the two memories shown in the figure. Then the *OS sets the* correct pattern in the Program ID register and initiates the program. The address to be used for the information access in the actual memory is obtained by adding the base address of the segment to the address within the segment. However, the address within the segment is

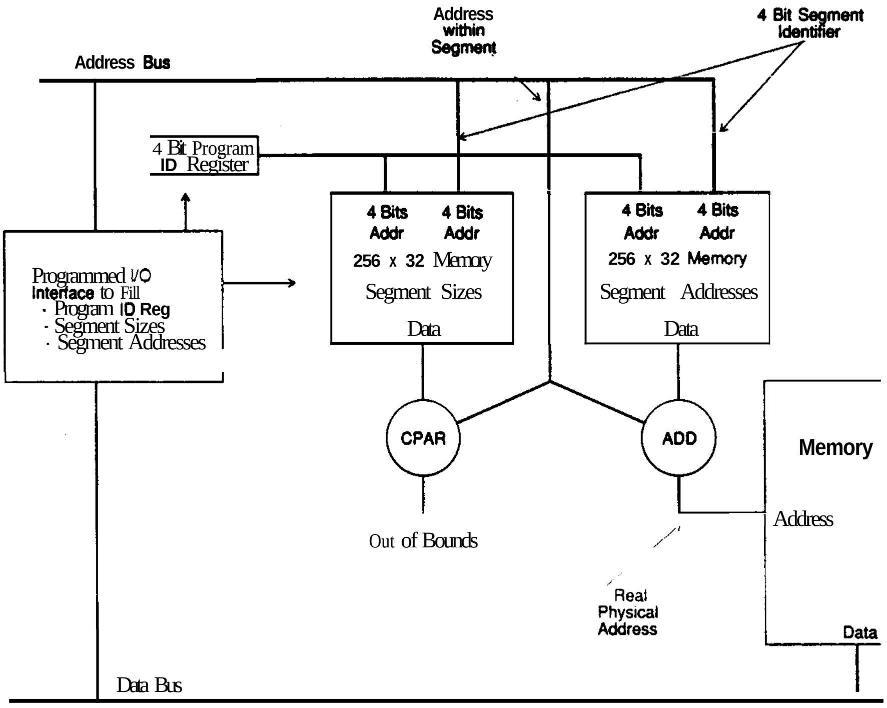


Figure 7.20. Hardware for Address Translation with Segments.

also directed to a comparator that checks the address against a maximum. If the address is too large. then the out of bounds signal will be asserted, and the system will be informed of the problem. The 256 locations in the segment sizes and segment addresses memories allow up to 16 different programs to reside in memory at the same time, and switching between them is accomplished by placing a different pattern in the program ID register. Note that some data paths are not shown, such as the path from the data bus to the segment information memories. Note also that the memories are 32 bits wide, which would allow for alignment on any byte boundary (for a system with 32 address lines). Since most memories are organized (at least) 4 bytes wide, these memories could be 28 bits wide, with the understanding that all segments must be aligned on a doubleword boundary.

As demonstrated by the preceding example, the virtual to real translation is not free, some time is required to generate the real address from the information supplied by the processor. The times involved are the time to access the segment addresses and segment sizes information, and the time to add that information to the address within the segment This overhead is imposed on all references to virtual memory in this scheme. In addition to the overhead on a per reference basis, there is also the overhead of managing the memory space. allocating segments in the available space, collecting the empty space, and so on. All of these operations add to the overhead of the system, and lead to a discussion of different approaches to memory address mechanisms.

Another mechanism for mapping virtual addresses to physical address is to divide the original program and data space into pieces based not on logical boundaries. but rather physical boundaries. Thus the program model shown in figure 7.16 can be modified as shown in Figure 7.21. The pages have the characteristic that they all have exactly the same size, as compared with the segments mentioned above, where the size is not a standard value. This organization allows for the individual elements (in this case pages) to fit in any location in a page frame, since all pages have exactly the same size. The pages all begin on a page boundary. The process of address generation is basically the same as that for segmentation:

$$ADDR_{REAL} = ADDR_{PAGE\ BASE} + ADDR_{PAGE\ OFFSET}$$

The principal difference is that the addition called for in the above equation is a concatenation, not a full addition. That is, since the pages are forced to begin on page boundaries, the least significant address bits (for the first location of a page) are all zero. The address bits that identify the location within the page will not extend into the nonzero bits of the address for the page boundary. Hence. no addition time is required.

The difference between the paging scheme and the segmentation hcheme presented above can be visualized by a different view of the program of Figure 7.18. The addressing scheme is modified to a paging scheme as shown in Figure 7.22. As far as the program is concerned, the only difference is that the accesses are made by specifying the page number and offset within the page, not the segment number and offset within the segment. As before, the correlation between the virtual and physical addresses can be represented in tabular form, shown in the figure as the page table. As shown in the figure, the instruction located at location 344 of page 2 has a physical address of 2F44. Note that the addition is merely a

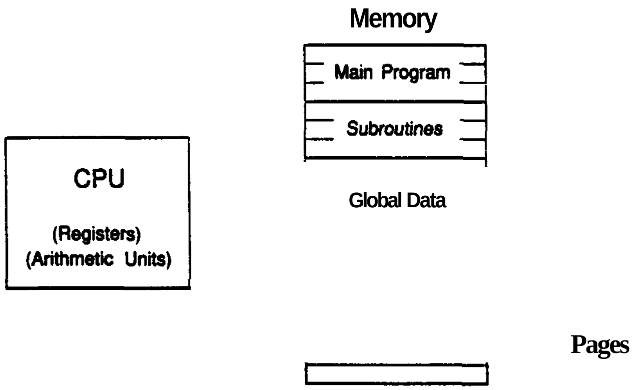
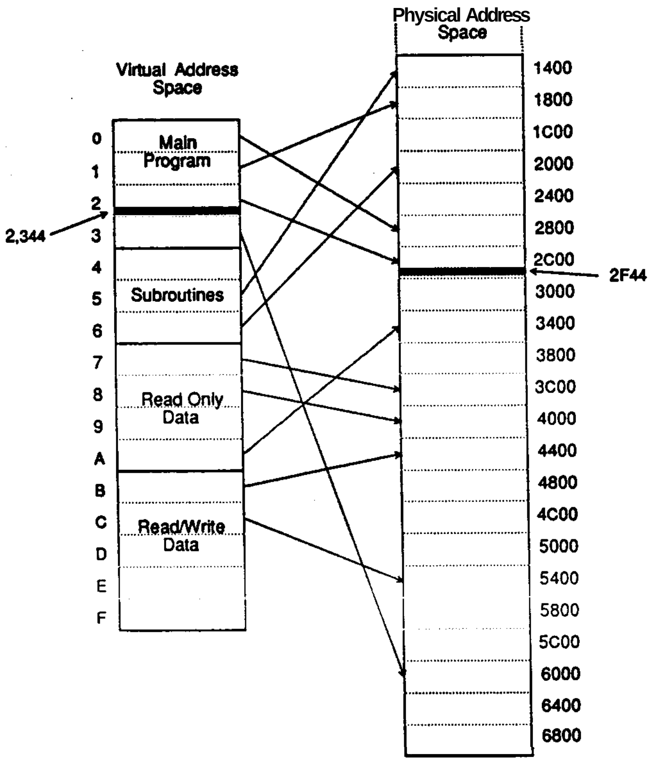


Figure 7.21. User Model for a Computer and Program, Using Pages.



Page Table

Page	Address	Access
0	2800	Read, Execute
1	1800	Read. Execute
2	2C00	Read. Execute
3	6000	Read. Execute
4	Not in Memory	Read. Execute
5	1400	Read, Execute
6	2000	Read. Execute
7	3C00	Read, Execute
8	4000	Read
9	Not in Memory	Read
A	3400	Read
B	4400	Reed
C	5400	Read/Write
D	Not in Memory	Read/Write
E	Not in Memory	Read/Write
F	Not in Memory	Read/Write

Figure 7.22. Virtual Address Mechanism with Paging.

concatenation, since 2C00 + 344 does not have any nonzero overlap (0010 1100 0000 0000 + 0011 0100 0100 = 0010 1111 0100 0100).

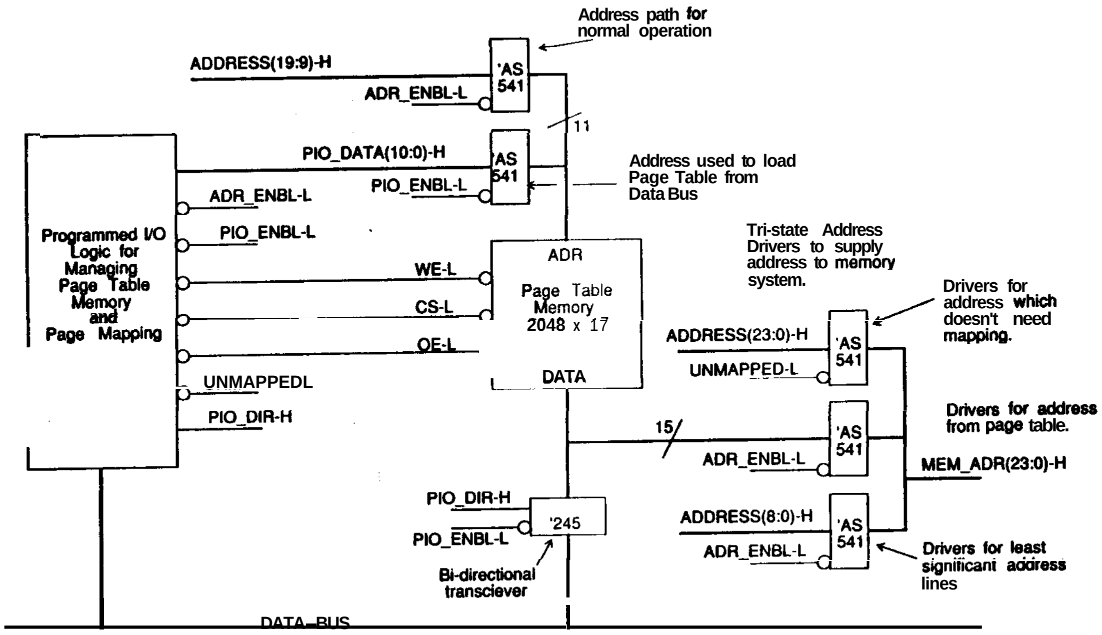
The OS burden changes under this scheme, since the question asked is not if a page will fit, but rather, where should the page be placed. This decision is a function of the method utilized by the operating system to maintain the memory space in the machine, and how much information is dealt with with each page operation. Some systems bring in only those immediately requested by the program. Other systems load into memory not only the requested page, but some surrounding pages as well. For a discussion of the various decisions and their impact on overall system performance see [PeSi83, Deit84, BiSh88].

Pages are generally much smaller than segments, ranging from 256 bytes to 1.024 bytes or more. Since the page size is smaller than a segment size, there will be, in general. many pages in a system. Thus, the table of entries cannot be limited to 16. However, the overall organization of the memory mapping scheme will be very similar. Consider the following example that proposes some hardware to provide a virtual address to real address translation.

Example 7.5: Memory mapping with pages: Design a virtual memory mapper that uses pages of 512 bytes. The page table must be capable of supporting 2,048 pages. The mechanism should function as indicated by Figure 7.22. What is the speed of the address translation mechanism?

One solution to the stated problem is shown in Figure 7.23. The address received from the processor is dealt with in two sections: the page identifier and the offset within the page. Thus, with 512 bytes per page. the 9 least significant bits (ADDRESS(8:0)-H) are used to identify the location within the page. and the remaining bits of the address are used to specify the appropriate page. Also note that with 2,048 pages. each with 512 bytes, the addressable memory is only a megabyte. This is not a large enough memory space for general usage, but will be large enough for some applications. The stated requirement of 2,048 pages necessitates 11 bits of address to identify the appropriate page. The width of the page table memory for this design is 17 bits, which allows 15 bits of address and 2 bits to indicate the status of the addressed page. One bit is used to indicate if the addressed page is in main store or not; the other bit is used to identify whether the page has been modified since it was loaded. If the page has not been modified, then, when the time arrives for it to be removed to make room for a new page, the old page need not be returned to the mass storage device.

Three basic modes of operation of the mechanism are shown. In one mode the page table can be filled with information. In this mode, the address of the page table is provided by the PIO_DATA path, and data can be loaded into the table with the chip select (CS) and write enable (WE) lines from the control logic. The information to place in the page table is loaded from the data bus, via the transceiver. The second mode of operation is normal behavior for the system. In this mode, the 9 least significant lines of the address are obtained directly from the address bus. Since there are 24 address lines total, the remaining 15 lines must come from the page table. The applicable location of the page table is identified by the 11 address lines of higher significance than the lines that identify the location of the address within the page (ADDRESS(19:9)). These lines are fed to the address of the page table, and also the output enable (OE) is asserted. Under these conditions. the page table will output the base address of the



Address path for normal operation
ADDRESS(19:9)-H
'AS 541
ADR_ENBL-L
11
PIO_DATA(10:0)-H
'AS 541
PIO_ENBL-L
Address used to load Page Table from Data Bus
ADR_ENBL-L
PIO_ENBL-L
Programmed I/O Logic for Managing Page Table Memory and Page Mapping
WE-L
CS-L
OE-L
ADR
Page Table Memory 2048 x 17
DATA
UNMAPPEDL
PIO_DIR-H
Tri-state Address Drivers to supply address to memory system.
Drivers for address which doesn't need mapping.
ADDRESS(23:0)-H
UNMAPPED-L
'AS 541
15
'AS 541
ADR_ENBL-L
Drivers for address from page table.
MEM_ADR(23:0)-H
PIO_DIR-H
'245
PIO_ENBL-L
Bi-directional transciever
ADDRESS(8:0)-H
'AS 541
ADR_ENBL-L
Drivers for least significant address lines
DATA_BUS

Figure 7.23. Block Diagram of Logic for Paged Memory Address Translation.

selected page, which will then be concatenated with the 9 least significant lines to form the physical address of the desired virtual location.

The third mode of operation is for addresses that do not need virtual-to-real translation. This could be used, for example, by the OS when it interrogates tables at known physical locations in memory, or activates memory mapped I/O.

Not shown in the logic diagram is any method for keeping track of the order of use of the pages. When a new page needs to be loaded into main store, a number of algorithms may be used to identify the page to be replaced. The algorithm that feels most intuitively correct is to replace the least recently used page. But the hardware required to keep track of the pages in the order of their use is nontrivial and not shown. Other algorithms are also available that optimize the behavior of the system under specific circumstances [PeSi83, Deit84, BiSh88].

As stated earlier, the addition process is one of concatenating the bits in the proper order. and no real addition is required. Thus the time required for the circuit shown in Figure 7.23 is the sum of the delay times in the respective elements:

$$T_{\mathrm{TRANSLATE}} = T_{541} + T_{\mathrm{MEM}} + T_{541}$$

$$= 2 \times T_{541} + T_{\mathrm{MEM}}$$

For 74AS541s and 30 nsec memory. this totals about 42 nsec.

The paging scheme has many advantages that make it very attractive for systems. Since all pages are the same size, any page can be placed in any page frame in memory. The system is able to more effectively utilize all space, and memory does not tend to fragment as it does in systems utilizing segmentation only. The creation of the address utilizes a concatenation process. which saves time over a system that requires an addition. Nevertheless, some problems need to be addressed in real systems. One of the problems is illustrated by the preceding example. Even with a page table memory of 2.048 entries, the maximum size of the memory available to a program in this system is 2.048 x 512 = 1 Mbyte. Since the amount of space used by programs has increased drastically as the relative cost of memory has decreased, this is not large enough for most programs. The 24-bit address space provided by many processors allows for 16 Mbytes of memory, and this is not enough for many programs today. In a simple program used in a university environment for some research problems, the virtual memory space needed by the system exceeded 50 Mbytes. Thus, it is necessary to provide a sufficiently large page table to allow programs to grow to the necessary size.

Any limit on the number of pages that can be accessed by a program will eventually limit the usability of the system. Thus, the approach suggested by Example 7.5 is not sufficient, and the system needs to be modified to allow a larger page capability. This can be accomplished by keeping the page table. not in hardware, but in the memory of the system itself. However, if all accesses needed to obtain page addresses by going to main store, the performance penalty would be very large. One solution is to keep in hardware not all of the page table entries, but rather the most active page entries. In this manner the hardware requirements can be reduced, and still maintain critical page information to speed up the processing.

This is the approach taken in the NS32082, which is the memory management unit (MMU) for the NS32000 processor series. A block diagram of the NS32082 is given in Figure 7.24. This unit has been designed to work in conjunction with the time multiplexed bus of the NS32000 series processors, an example of which is given in Figure 6.10. When the processor generates the ADS signal, the MMU accepts the address and examines a 32 entry page table to see if it is an active page. If the match is successful, then the correct physical address is made available in the next bus cycle, and the bus transactions continue as expected. Thus, the overall effect of the virtual to real translations carried on by the MMU is to add one additional cycle to the four cycles needed for a bus transfer. This increases by 25% the time required to fetch information fmm memory. Studies indicate that for most programs the needed page information will immediately be found in the 32 element page table for around 98% of all accesses. When the address provided by the processor does not match one of the enmes in the hardware page table, this does not mean that the page is not in memory. Rather, the information must be sought from the real page table. which is kept in main store. The NS32082 automatically generates memory requests to fetch this information from main store, whereupon it updates its hardware page table and continues the interrupted processing. This should occur for about 2% of the memory requests. A system block diagram of a NS32032 CPU with a NS32082 MMU and other support chips is shown in Figure 7.25. Note that the ADS signal is directed only to the MMU. which then creates the correct physical address and asserts PAV, physical address valid.

Like segmentation, the paging mechanism allows the system to create a "virtual" memory, which is the appearance of a memory space as it is accessed by a program. This allows the program to access more memory than is resident in the computer, since nonactive portions of the information are maintained on

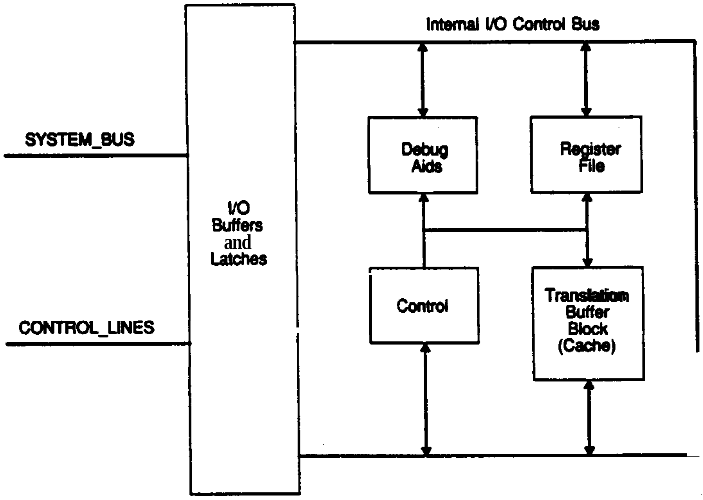


Figure 7.24. Block D i i of NS32082 Memory Management Unit.

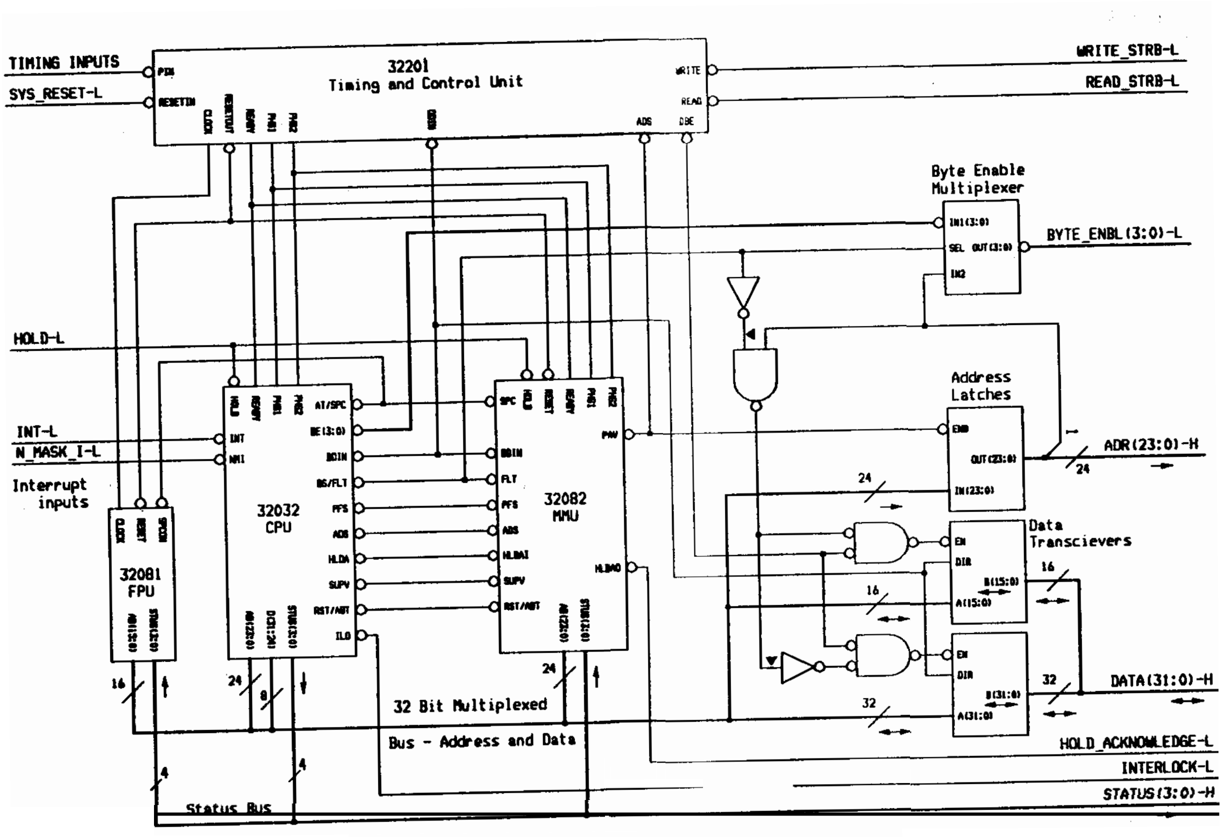


Figure 7.25. NS32032 System with NS32082 Memory Management Unit.

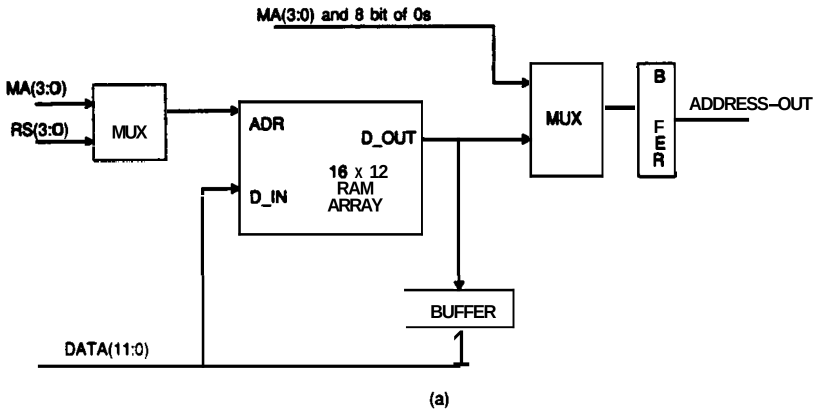
secondary storage. However, this is not the only application of the concept of paging. A number of instruction sets can access a limited range of memory, and paging can be used effectively in these systems as well. For example, the 8080 system has been used for many years, and it has an addressable range of 64 Kbytes. The same can be said for the PDP 11 architecture or the TI9900. However, with newer memories we can get 256 Kbytes to 1 Mbyte or more in a single SIP, which is far more that a single program can address. One way to effectively utilize the memory is a multiprogramming environment. In this fashion, each of the programs can have its own portions of memory, and accesses can be made with the paging mechanisms explained above.

The use of paging to effectively utilize a large memory for systems that limit the memory addressable by a single program allows several programs to reside simultaneously in a memory. Then. as the addresses are created by the program, a translation is performed m access the proper page. Texas Instruments. Inc.. manufactures an LSI device used to perform the mapping for the case a large memory and a small inherent address capability. The block diagram of the device. which is the 74LS612, is shown in Figure 7.26(a). The assumption made by system designers utilizing this device is that the address space is broken into pages of 4,096 bytes. Thus, the 12 LSBs of the address are not touched by the paging mechanism. The 4 MSBs of the address are used to identify one of 16 locations of a page table memory inside the device. Each of these locations contains 12 bits. which identify one of 4.096 pages in the real memory space. With this device mapping can be implemented to a real memory space of 16 Mbytes.

The operating system is responsible for loading the proper page addresses into the page table. which it does by using the programmed I/O instructions and addressing the appropriate register with the RS lines. Once the system has prepared the table, the processor can access up to 16 pages by mapping the addresses into the real memory system. That way, 16 complete programs could reside in a 1 Mbyte memory, and be accessed through the 74LS612. A diagram showing its use with an 6800 system is given in Figure 7.26(b). Logically, the device resides between the processor and the memory. And the operation of the system creates a virtual space that is smaller than the physical space available, and yet uses the concepts presented above.

The use of virmal memory techniques allow effective use of the real memory, whether the available memory is larger or smaller than that needed by a specific program. The program operates under the illusion that it has access to its own memory, independent of other events that may occur in the system. With large processor systems, this results in the use of less real memory than called for by a single program. The use of a large memory with processors that cannot access all of the available storage results in systems that can load several complete programs into the available physical memory. But the basic reason that the systems are effective comes from the observed behavior of programs in execution.

Programs generally exhibit locality when they are running. That is, at any given time, or during a short period, a program will tend to use information in a small number of locales. While a program is executing a loop, the instruction fetches are confined to the memory area where the loop is located. The loop may access an array, and. while the array accesses are going on, the data references are limited to the area where the array is located. But the net result is that the amount of memory needed by a program during any small period will not be the entire addressable space, but rather a portion of i t Thus, a pmgram may require a small number of all of its pages during any particular time slice. This behavior allows



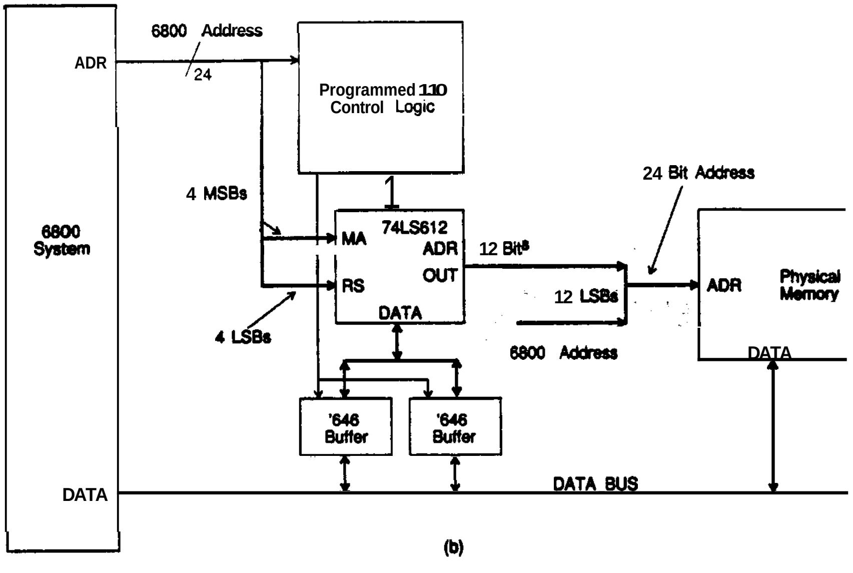


Figure 7.26. Memory Mapper for Small Virtual Space to Large Physical Space. (a) Block **Diagram of 74LS612.** (b) **System Incorporating the 74LS612.**

the virtual memory systems to be effective. The migrating action of pages in and out of main store keeps in memory the parts of the program which are needed.

The element not discussed to this point is the secondary storage. In general, the *secondary* storage mechanisms will be disks, although other mediums are possible. The virtual memory system must work with the secondary storage element and the main store to coordinate the transfer of information between the two. The interface mechanism will be a data transfer protocol as was discussed in Chapter 6: the controllers an activated by the processor and perform whatever tasks they an given, moving information from the secondary storage area to main store. The OS is responsible for requesting these transfers in a reasonable fashion. and maintaining page tables as needed to reflect the contents of system memory. The more pages that exist in main store, the greater the probability that a program will find the information that it needs. Nevertheless, eventually the program will access information on a page not in main store. The virtual address translation mechanism recognizes a request for data that is not in main store and interrupts the processor. The OS then must deal with the program in a reasonable way. Most often, the program is temporarily halted and the system requests that the unavailable information be brought into memory. Meanwhile, the current state of the program is saved, and the information for another program is loaded into the registers of the CPU. The system can then continue execution on another program while awaiting the arrival of data from secondary storage.

Example 7.6: Secondary storage access: Consider the block diagram for a virtual memory system as shown in Figure 7.27. As long as the processor is requesting information from main store, the system will continue executing. When the processor detects a page fault. the OS will need to bring in the appropriate page. On the average. how much time will transpire before the requested information is brought into main store? Assume that the access time to main store is 250 nsec. Also assume that the disk rotates at 3600 RPM. that there are 48 sectors per track, and that a sector and a page have the same size — 512 bytes.

The time required for the transfer will break down into three different times:

- The seek time for the disk, which is the time for the disk to find the desired track.
- The rotational latency, which is the time for the disk to rotate until the desired sector is under the read head.
- The transfer time, which is the actual time to transfer the information to the memory.

Thus the time will be:

$$T_{ACCESS} = T_{SEEK} + T_{ROT\_LAT} + T_{TRANSFER}$$

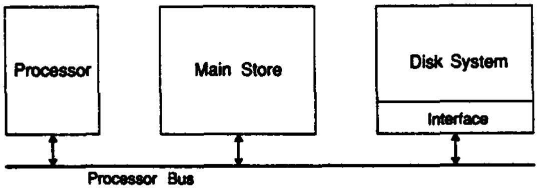


Figure 737. Simplified Block Diagram of Memory System.

The seek time is a characteristic of the individual disk being used. The time to seek to the next track is much smaller than the time to seek to a distant track, but an reasonable time is approximately 20 msec. The rotational latency, on the average, will be half the time for one revolution. At 3,600 RPM, the disk makes 60 revolutions per second, or one every 16.67 msec. and half of that is 8.3 msec. Since there are 48 sectors per track. the minimum transfer time is 1/48 of a revolution time, or about 347 µsec. Thus, the time for the transfer is dwarfed by the other times involved. So,

$$T_{\text{ACCESS}} = T_{\text{SEEK}} + T_{\text{ROT\_LAT}} + T_{\text{TRANSFER}}$$

$$= 20 \text{ msec} + 8.3 \text{ msec} + 0.347 \text{ msec}$$

$$= 28.647 \text{ msec}$$

Note that we have not included in this figure all of the times involved. since some time will be required by the processor to identify the appropriate page and issue the request to have the page transferred to main store. This time is not negligible, and should be accounted for in identifying the detailed costs of the transaction.

Some observations can be made at this point regarding the relative times of the transactions. One interesting piece of information is the ratio of the access time of the disk to the access time of the main store, which we will call R_{VM}, since this is the storage ratio involved with virtual memory. Thus.

$$R_{\text{VM}} = \frac{T_{\text{ACCESS}}}{T_{A_{\text{MAIN STORE}}}}$$

$$= \frac{28.647 \text{ msec}}{250 \text{ nsec}}$$

$$\approx 114{,}500$$

This indicates that over 110,000 transfers could take place while the disk is accessing the information not in memory. Another observation is that, during that period of time a 2 MIP machine (a machine capable of 2 million instructions per second) could execute almost 60,000 instructions. Since a significant amount of work can be done in the time required to obtain the information from secondary storage, an operating system will suspend the process that incurred the page fault, and allow another process to utilize the computational resources of the system.

Another observation is that the principal time involved in T_{ACCESS} is *the* seek time. Thus, systems that strive for high speed can benefit from a device that does not need a physical seek to obtain the information. Two such devices are drums and head-per-track disks. These devices have a much higher cost per bit for storage. but can be used if the circumstances warrant.

As we have seen, virtual memory can be used to more effectively utilize the resources available to a system, and in particular to use memory in an efficacious manner. Segmentation approaches the task by accessing information in logical units, such **as programs,** subroutines, and data **areas.** Note also that a system can organize the information that it **needs** to access **as** segments, **so** that individual data entities can be organized **as** individual segments. Paging accesses information by organizing information into pages, using a standard physical unit **as the** common denominator in all transactions. Both concepts allow the program to divorce itself **from** the placement of the information in physical memory, and allow the instructions to identify the location of information within a virtual memory framework. It is not necessary to use only one or the other method, and some systems combine the two together to form a paged system that also understands segments. This type of an addressing scheme allows the creation of segments, which have unique characteristics, and the further benefits of paging, which allows a regular placement policy.

7.4. Cache Memory: Speed-Up for Main Store

Cache memories are (relatively) small. high-speed memories inserted into the system between the processor and the main store. **The** purpose of cache memory is to speed up the processing rate by allowing the processor to execute at a higher rate than that possible by using main store alone. It utilizes many of the same concepts used with virtual memories. but in a slightly different fashion. One of the first machines to utilize this mechanism was the IBM 360/85 [Lipt68], but the concept has become widely implemented in machines of all sues. Before a cache system is implemented, a **thorough** study of the behavior of the memory system under expected operating conditions must be conducted. In this section we will study some of the mechanisms utilized by cache systems, and determine their effectiveness in different conditions. For a relatively complete discussion of a number of the techniques and their relative merits, **see** [Simt82].

The virtual memory mechanisms **discussed** in the previous section allow programs to execute using a **virtual** memory space, a space that appears **different** to a program than the actual **space** being utilized. **The** program need not have a correct understanding of the amount of memory actually available. Programs can run using very large virtual spaces with a relatively small actual main store. or **the** program can, by its inherent instruction and reference limitations, access only a portion of the actual memory available. In either case, systems that utilize virtual memory tend to have only portions of the pmgram loaded into main store at any one time. These portions are sometimes referred to **as** the active portions of the pmgrams, and during the execution of a program the active portions will change.

One effect of the use of virtual memory techniques is that the apparent processing speed of the system is higher, because the CPU is more effectively utilized by a number of programs, and the amount of time that the CPU is idle is minimized. Of **course,** one exception to this speed enhancement will occur when **the processing** being done is limited not by **the** processor, but rather by the I/O **capa**bilities of the system. That is, a set of I/O bound jobs will not experience **the speedup** improvements that would be seen by a mixture of I/O intensive and **com**pute **intensive** programs. Nevertheless, the system benefits from having only **the active** portions of **the** programs reside in **the** main store at any **one** time.

Cache memories operate on **the** same basic principal: keep in **the** memory (in this case **the** cache) only those portions of **the information needed** and active. In this manner, **the** cache and **the virtual** memory mechanisms **are** similar.

However, there are some major diirences between the virtual memory and cache memory implementations. The two most obvious differences are the visibility of the mechanism and the ratio of access times.

A virtual memory system has high visibility to a program and to the operating system, since the "virtual machine" as seen by a program is accessed by the virtual memory system, which in turn is managed by the operating system. For example, a user has the capability to access more memory than the system actually has through the use of virtual memory, and the OS must maintain page tables and other information in the system to control the various facets of the system operation. The cache, on the other hand. is usually hidden from the user and the system. The decisions as to the operational modes of the cache are made at design time and built into the system. Thus, a program will not know that a cache is being w d , except by the speed of processing.

The ratio of access speeds for the cache. R_{CA}, also differs drastically from R_{VM}. The definition of R_{CA} will be similar to R_{VM}:

$$R_{CA} = \frac{T_{A_{\text{MAIN STORE}}}}{T_{A_{\text{CACHE}}}}$$

Access times for main store and cache memories improve each year, but typical times might be 250 nsec for $T_{A_{\text{MAIN STORE}}}$ and *40* nsec for $T_{A_{\text{CACHE}}}$. Using these times. R_{CA} becomes:

$$R_{CA} = \frac{T_{A_{\text{MAIN STORE}}}}{T_{A_{\text{CACHE}}}}$$

$$= \frac{250 \text{ nsec}}{40 \text{ nsec}}$$

$$= 6.25$$

Instead of a ratio in excess of 110.000 as for R_{VM}, R_{CA} is on the order of 6. This ratio will vary from system to system. but the effect will be the same: there is not enough time to change the task in the processor. Thus, the processing element is halted until the information which was not in cache has been obtained, and then the processing continues.

One of the basic questions is how good the cache memories are, and how to quantify the *effect* The term "good" is a relative measure. and indicates how fast the processor is operating with the cache compared to the operating rate without the benefit of the cache. To identify the effects involved with the cache, let us consider a system organized as shown in **Figure** 7.28. This simple figure indicates the logical organization of the system. but not necessarily the physical organization. The figure of merit that interests us hen is the T_{EFF}, which is the effective access time of the memory system, considering the cache and main store memories together as a single system. A very simplistic formula for this time is given by:

$$T_{EFF} = h \times T_{CA} + (1 - h) \times (T_{CA} + T_{MS})$$

where T_{CA} is the access time of the cache and T_{MS} is the access time of the main store. The other term in the equation. h, is the hit rate, or the fraction of the references found in tbe cache. Thus, $(1 - h)$ is the miss rate, the fraction of the

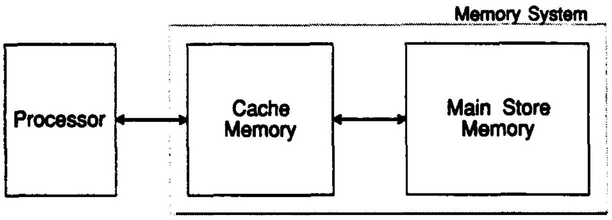


Figure 7.28. System with Cache and Main Store Memories.

references not found in the cache. The first term, h x T_{CA}, indicates that of all the references to memory, the fraction h will incur a penalty of T_{CA}. And the second term, $(1-h) \times (T_{CA}+T_{MS})$, indicates that the remainder of the references, 1 − h. will incur a penalty of $T_{CA} + T_{MS}$. The T_{CA} term appears here to identify the amount of time that is required to ascertain that the desired reference is not in the cache, and the T_{MS} term identifies the amount of time then required to go to main store to obtain the requested information. With a little algebra, the equation can be further reduced:

$$T_{EFF} = T_{CA} + (1 - h) \times T_{MS}$$

This is a very simplistic formula. since it does not include the effects of many of the real problems that occur in caches. But it is sufficient to give some insight into the effectiveness of a cache memory organization.

The formula for T_{EFF} is a linear equation, and will specify straight lines on a linear plot Figure 7.29 gives a plot of T_{EFF} as a function of the hit rate, h. Four different lines appear in the figure. for R_{CA} values of 2.5. 5, 10, and 20. The access time is given in terms of T_{MS}, so that, if T_{EFF} exceeds 1.0, then the response time of the memory system with cache would be worse than the response time of the system without cache. The figure identifies that indeed this situation can occur, but the hit rate must be very poor for T_{EFF} to be greater than 1.0.

A graph that gives a more intuitively pleasing observation of the effect of wing a cache is given in Figure 7.30. Here we plot the speedup of the system, where the speedup. S. is defined as a ratio of the access times without and with cache memory:

$$S = \frac{T_{MS}}{T_{EFF}}$$

The plot indicates that, as the hit rate approaches 1.0, the speedup improves dramatically. This agrees with expectations concerning the use of caches.

This simple formula is not an accurate model of the exact behavior of a cache memory, since it does not account for many details. We will return later to the calculation of T_{EFF}, but let us now consider some of the implementation methods.

Implementation details vary for cache memories, and the following descriptions can be modified to produce results slightly different from those included here. For example, the addressable units in a cache will vary depending on the application, from small units in caches that are inherently small, to large units, for

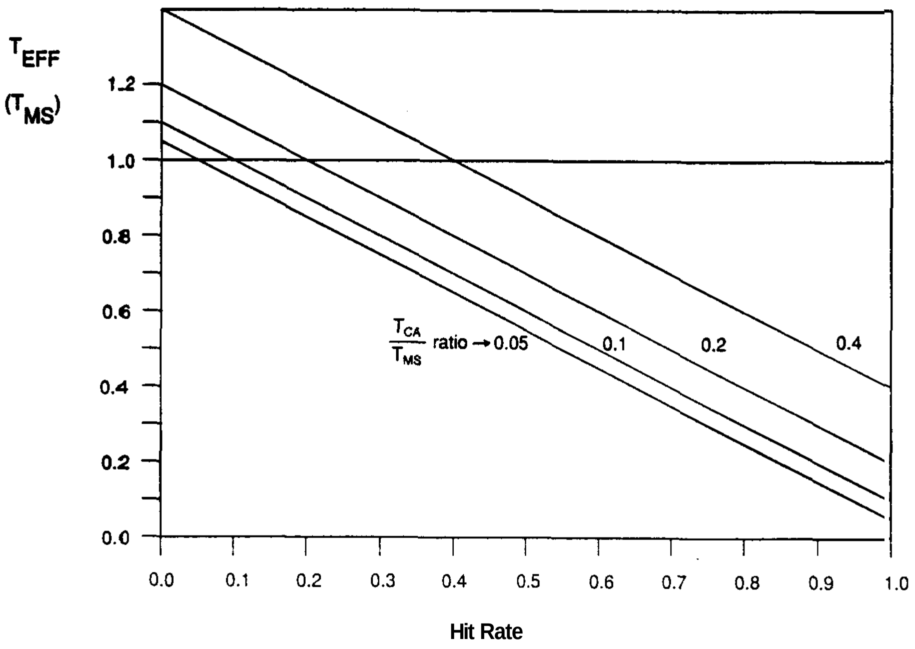


Figure 7.29. Effective Memory Access T i e, T_{EFF} as a Function of Hit Rate, *h*.

caches with an increased memory capacity. But the basic principles of operation will be the same regardless of the implementation.

For our example cache organization we will use a cache size of 32 Kbytes. The system memory may be 16 Mbytes or larger, so the cache can only hold a fraction of the information resident in main store. The cache is organized to access information in some basic unit, which we will call a 1 i . We will let the line size be 32 bytes, so the cache will contain 1,024 1 i . The main,store will also be organized as lines, so that lines can be exchanged between the cache and main store. This basic organization is depicted in Figure 7.31. Whenever then is a hit, the cache provides to the processor the information that was requested. This will occur at the speeds of the processor itself, and only the specified information (byte, word, double word, etc.) is transferred. However. if there is a cache miss. then the data must be brought from the main store into the cache. The information is transferred from the main store to the cache by moving an entire line. Some cache organizations also have the capability of requesting a transfer involving multiple lines.

To this point we have placed no restriction on the location of information in the cache. If the system permits any line from main store to reside in any line in the cache, then we say that the system is a fully associative organization. Thus, when the processor identifies an address, all of the lines in the cache must be interrogated to ascertain if the desired information is in the cache. This leads to very expensive hardware. since, for the example of Figure 7.31, 1,024 locations

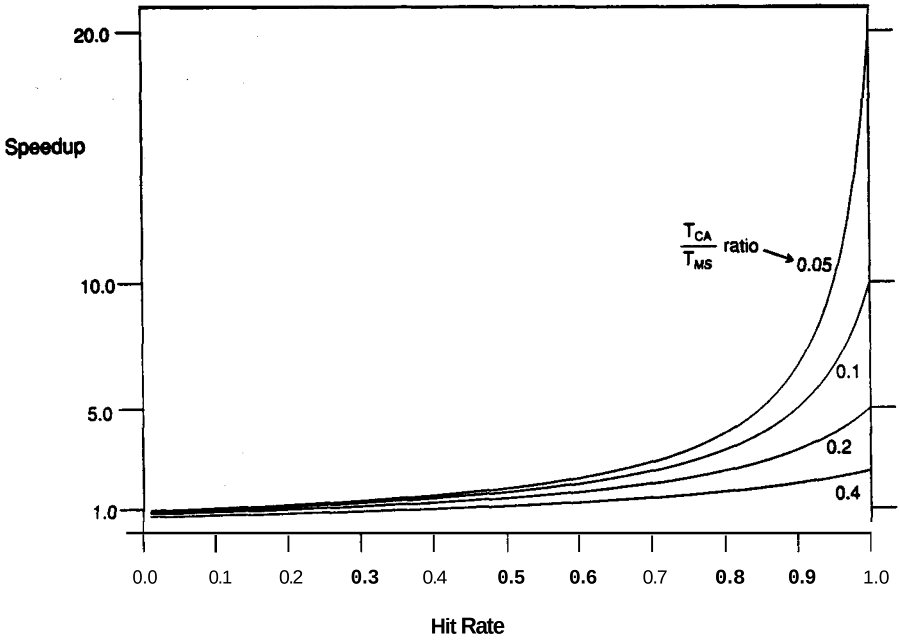
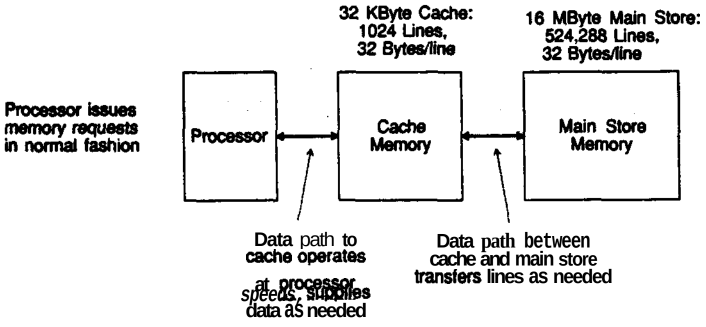


Figure 7.30. *Effective Speedup (T_{MS} / T_{EFF}) of a Cache System.*



32 KByte Cache: 1024 Lines, 32 Bytes/line
16 MByte Main Store: 524,288 Lines, 32 Bytes/line
Processor issues memory requests in normal fashion
Processor
Cache Memory
Main Store Memory
Data path to cache operates at processor speeds, supplies data as needed
Data path between cache and main store transfers lines as needed

Figure 7.31. Organization of Cache and Main Store Memories.

must be searched. The hardware required to search for the address in question will be greatly reduced if restrictions are place on the allowable locations in the cache of the lines in main store. If a line in main store has exactly one location in the cache when it can be found, we say that the cache has direct mapping. With direct mapping, only one location in the cache needs to be queried to find out if the addressed information is available or not. With direct mapping in the organization of Figure 7.31, each line in the cache can hold information from any of 512 lines in the main store. But each line in main store maps to only one line in the cache.

A compromise between the fully associative and direct mapping mechanisms involves a technique known as "set associativity." If, instead of only one location in the cache, a line in main store could be located in om of two locations, we say that the cache is two way set associative. or set associative with a set size of two. Likewise, if the line can be located in one of four locations, then the organization is four way set associative, or set associative with a set size of four. Other set sizes are possible, such as eight and sixteen. The more elements in a set, then the more hardware is required to implement the cache. With set associativity. a line can be found in a limited number of locations, and the hardware needed for the parallel search of those locations is manageable.

If the cache of Figure 7.31 is organized as a four way set associative cache, then then are 256 sets, each with four lines. The organization of the system is

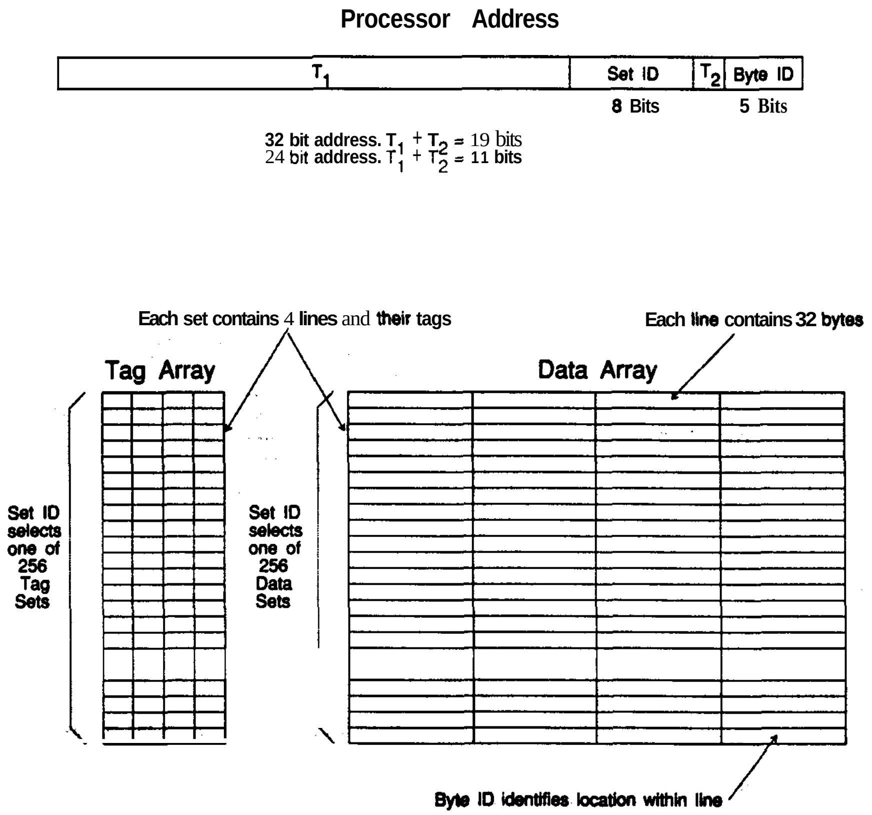

Processor Address
T_1
Set ID
T_2
Byte ID
8 Bits
5 Bits
32 bit address. $T_1 + T_2 = 19$ bits
24 bit address. $T_1 + T_2 = 11$ bits
Each set contains 4 lines and their tags
Each line contains 32 bytes
Tag Array
Data Array
Set ID selects one of 256 Tag Sets
Set ID selects one of 256 Data Sets
Byte ID identifies location within line

Figure 7.32. Organization of a Set Associative Cache with Four Sets.

shown in Figure 7.32. The address, as supplied by the processor is divided into fields, each of which has a specific function. Since there are 32 bytes in a lie. then the 5 least significant bib are required to identify the target location within the line. If the architecture is not byte-addressable, then this requirement changes accordingly. Since most general purpose machines have byte-addressable memories, we will proceed under that assumption. With 256 sets, then 8 bits of the address are needed to specify the appropriate set. The remainder of the bits form pan of a collection of information called a tag, and are stored in a memory that mirrors the organization of the data area of the cache. The tag bits include the remaining bits of the address, a method to identify if there is data in the line (since the cache will be empty upon startup), a dirty bit (to indicate if the line has been changed since it was brought into the cache). and other information that is needed by the system. For example. one of the useful things to know is the order of use of the lines in a set. If this information is available, then when one of the lines needs to be removed to make room for a new line, the least recently used line can provide the location for the new information. Thus, each line in the data section has associated with it a tag in the tag section. The line resident in the tag section is uniquely identified by that tag. The process of ascertaining the presence or absence of information in a cache then consists of examining *M* tags, where there are *M* elements in each set.

One of the characteristics of the cache mechanism is the order in which the lines of a cache are accessed for a given set of physical addresses. Consider, for example. a large set of addresses that is monotonically and uniformly increasing, each address being 4 bytes greater than its predecessor. In Figure 7.32 the processor address is divided into three different groups: set ID bits, byte ID bits, and two groups of tag bits. The byte ID bits are the 5 least significant address bits. The remaining bits in the address comprise the set ID bits and the tag bits. Eight bits are required to specify the set; these 8 bits form the set ID.

The remaining bits are tag bits, and are further divided into two portions. which we call T_1 and T_2. If there are zero bits in T_2 and all of the tag bits are in T_1, then as the address increases successive lines will be placed in successive sets. However, if there is a single bit in T_2 and the set ID bits are relocated in the address accordingly, then a different addressing pattern is formed. As the address increases. then two lines will be located in each set before moving to the next set in sequence. If T_2 has two bits, then four lines are allocated to a set before moving on. The mechanism which is most beneficial may be determined from the expected workload via a simulated address trace.

Example 7.7: Set associative cache system: Give a data path block diagram for a two way set associative cache memory with a capacity of 32 Kbytes. Identify the width of the data paths and the function of each block. Assuming that the memories used in the cache have an access time of 25 nsec, how quickly can the presence of the line in the cache be detected?

With 32 Kbytes in the cache, organized in a two way set associative manner, there are 512 sets, with the previous assumptions. That is, with a line size of 32 bytes, there are 1,024 lines, and with two lines per set, there are 512 sets. Thus. the cache can be made with 512 x 8 memories in a very natural way. A data path block diagram of such an organization is shown in Figure 7.33. Using byte wide memories, three devices would be needed for each tag array, and 32 devices would be needed for each data array. Thus. 70 memory devices would be needed for the cache. The address is divided to that line in the cache. Since the memory changes are not immediately communicated to main store, systems with multiple processors that share a common memory will need additional capabilities if this scheme is used. This is sometimes called the "stale data problem," or the "cache coherency problem," and we will examine it in more detail later. The write back mechanism rewards programs that cause writes to memory to occur in clusters, since several writes can be performed at cache speeds before any main store penalty is incurred. When the write occurs, a mechanism to temporarily hold the line being written out will minimize the overall time penalty.

The transfer mechanism with its associated storage buffer is an example of combining the various techniques discussed. For the purposes of discussion, let us use the system shown in Figure 7.34. The overall block diagram indicates that the CPU receives its information from (and provides information to) the cache. Meanwhile, the cache is connected via a bus system to four memory banks. Each bank of memory is essentially an independent unit, with addressing and timing capabilities needed for random access. The coordination of information transfer on the bus is handled by the bus controller. One of the assumptions made in this organization is that line size is 32 bytes, and that each bank holds 8 bytes of each line. Another assumption is that the transfer of information across the 8-byte bus requires 50 nsec, and that the access (read or write) to information in a memory bank requires 200 nsec. So, for a write back memory, the following sequence of events is one mechanism for performing the data transfers required. There is a cache miss. and a line needs to be brought in. Assume that the line currently in the cache in the location in which the new line is to be placed is not dirty — it has not been changed since being retrieved from memory. The accesses for the needed line are invoked in each bank of the memory. When Bank 0 has the required line, which occurs 200 nsec after initiation of the read access. it will transfer the information to the cache. This will be followed by the transfer for Bank 1, which is ready by the end of the transfer of information by Bank 0. The transfers for Banks 2 and 3 follow. Thus, 250 nsec after initiating the request, the first information is available, and 400 nsec after the request starts, the entire line has been accessed and transferred to the cache.

The second case includes the write back of information, as well as the obtaining of information for the cache. This will occur when a "dirty" line is replaced. In this case, one method of implementing the transfers recognizes that the bus is not used for the first 200 nsec of the above cycle. The information is obtained from the cache and sent to the interface modules of the respective memory banks during the first 200 nsec, and the information is then written back to the banks as shown in Figure 7.34(b). As shown in the figure, this policy leads to the desired information being loaded into the cache within 400 nsec, and the write back portion completed within 600 nsec. However, by staggering the requests in time. the effective time can be made 400 nsec. Thus, the write back scheme can benefit from this one level of storage buffer and increase the apparent speed of operation.

> *Example* 7.8: ***Effective time*** for cache access: Develop a formula for the effective access time for a cache memory that uses a write back scheme. Assume that the system must bring information into the cache to modify it. rather than to have writes that modify information not in the cache go directly to memory. (This assumption is made to create a simpler formula, not to reflect reality.) Assume that the probability that the access is a read

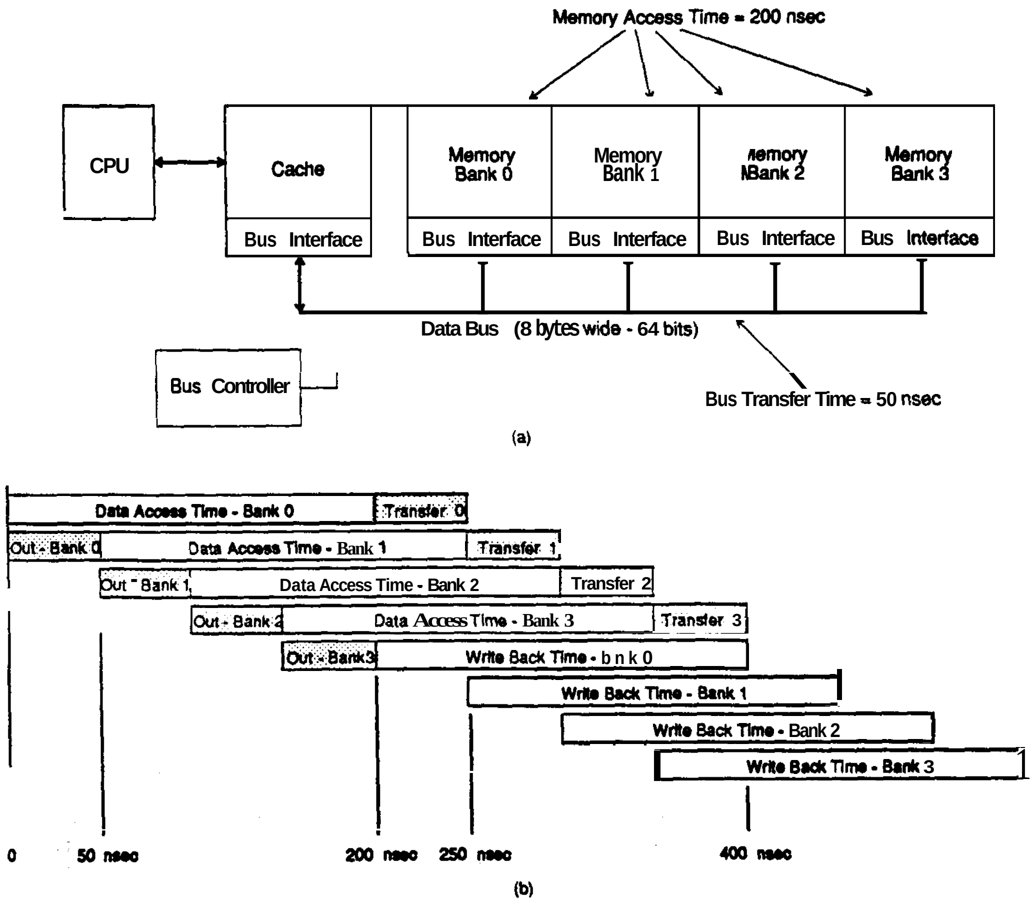


Figure 734. Cache Interaction with Banked Memory. (a) Overall Block Diagram. (b) Timing for Write Back Transfer.

is P_{READ}, and that the probability that a cache miss causes a dirty line to be written to main store is P_{DIRTY}. Also assume that reads and writes to main store incur the same penalty, T_{MS}, and that there is no storage buffer in the system. (Again, this assumption is for a simpler formula rather than to reflect reality.)

The assumptions surrounding this problem have been made in such a manner to simplify the resulting formulas, rather than to reflect how a specific cache has been designed. To identify the costs associated with the various accessing mechanisms, we will examine the costs in each of the four obvious cases: read hit, read miss, write hit, and write miss. The total solution will then be a weighted sum of these cases.

- The read and write hit cases are identical: the desired address is in the cache, and the cost of the access is the cache access time: T_{CA}.
- The read miss will incur a penalty of T_{CA} to ascertain that the information is not in the cache, and then a penalty of T_{MS} to bring in the information. However, this is not sufficient, since there may be a need to write back to main store a dirty line. This incurs a penalty of T_{MS} but occurs only with probability P_{DIRTY}.
- The write miss will also incur a penalty of T_{CA} to ascertain the target address is not in the cache.. In addition, it will require a time T_{MS} to bring in the desired line. It will also require a time T_{MS} with pmbability P_{DIRTY} to write out a line being displaced. However, once the line is in the cache, another T_{CA} is required to write to the spot selected.

Thus, the costs can be summarized by the following table:

	Hit	*Miss*
Read	T_{CA}	$T_{CA} + (1 + P_{DIRTY}) \times T_{MS}$
Write	T_{CA}	$2 \times T_{CA} + (1 + P_{DIRTY}) \times T_{MS}$

The formula for the system is a weighted sum of the above values:

$$T_{EFF} = P_{READ} \times \{ h \times T_{CA} + (1 - h)[T_{CA} + (1 + P_{DIRTY}) \times T_{MS}] \} +$$

$$(1 - P_{READ}) \times$$

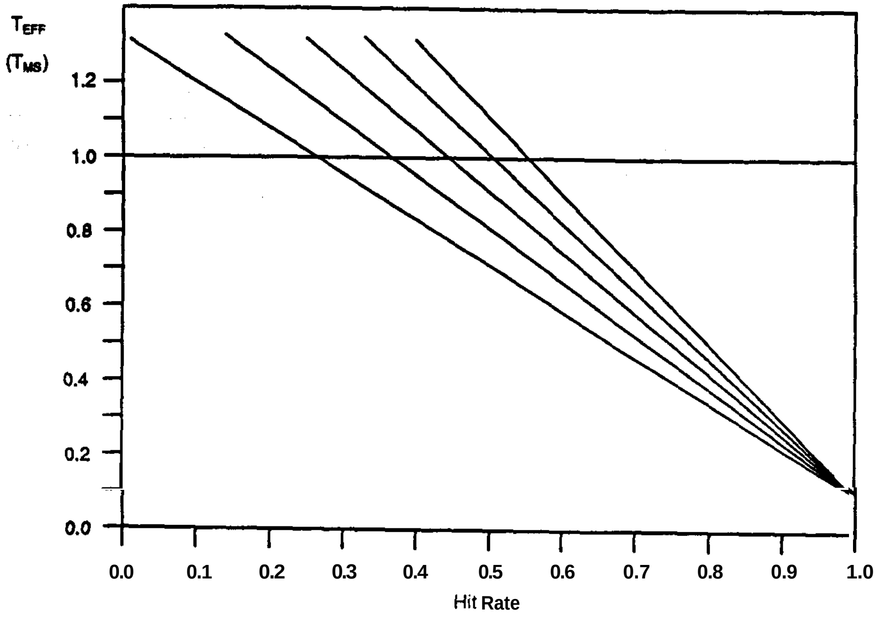
$$\{ h \times T_{CA} + (1 - h) \times [2 \times T_{CA} + (1 + P_{DIRTY}) \times T_{MS}] \}$$

$$= T_{CA} \times [1 + (1 - h) \times (1 - P_{READ})] + T_{MS} \times (1 - h) \times (1 + P_{DIRTY})$$

A family of plots of this equation and its inverse ate shown in Figure 7.35. The various lines are for different values of the probability of a dirty line (P_{DIR}). The assumption here is that the cache. time T_{CA} is one-tenth the main store time T_{MS}. As can be seen from the figure, the pmbability of a dirty line has a large impact on the performance of the system.

Other equations can be derived to more closely reflect reality. The difference will be in the complexity of the analysis, but the approach will be the same.

A number of other issues need to be dealt with in a real system. For example, how does the hardware handle a request for a word aligned across line boundaries? That is, since the system has been assumed to be byte-addressable, what happens when the request is for 4 bytes, the first of which is on one line and the other three ate on another line? A real system must be capable of handling this situation. (Note that one solution is to define the system in such a way that all memory accesses are made on 32-bit boundaries, and obtaining information within must then be done with software rather than hardware. This tradeoff must be made by the system architects at the time of the system definition:) Another real problem concerns the mechanism for physically writing information to the cache. We have not shown in the block diagrams or other examples the data paths nor logic required to write information back to the cache., but this must be done in a



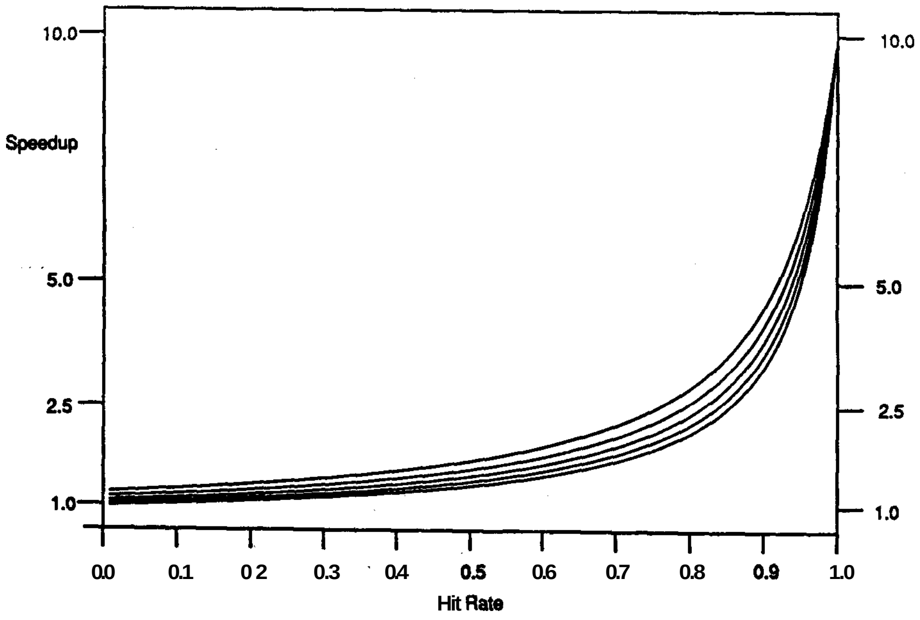


Figure 7.35. Cache Response Characteristics for Cache of Example 7.8. (a) Effective Access Time, T_{EFF} (b) Effective Speedup, T_{MS}/T_{EFF}.

timely fashion. A third troublesome reality is to coordinate the writes that occur because of the processor with writes occurring because of I/O transfers. Some provisions must be made to keep all of the information current and under control.

One of the side effects is the cache coherency problem, which exists for a system with multiple processors, each of which has its own cache. This situation is depicted in Figure 7.36, which shows a multiprocessor with two processing elements, each of which has its own cache. The cache coherency problem is exemplified by the following sequence of events. Processor A accesses a location in the memory, and the line is then loaded into Cache A. Subsequent accesses to the line will be found in the cache, rather than requiring the main store penalty. Processor B now needs the information in the line, so it accesses main store and gets its own copy of the information into Cache B. Further accesses of Processor B for the information are fielded by Cache B. Processor A now changes the information in the line. Processor B no longer has a valid copy of the line. since the information it has in Cache B has been superceded by the action of Processor A. Thus. the information is not coherent. and the situation has the label of the cache coherence problem.

If the write back scheme is used by the cache, then there is indeed a problem, since not only is the information in Cache B incorrect, but Cache B cannot obtain a valid copy until the information has been updated in main store, which will occur at some indeterminate time in the future. A write through scheme will provide a better basis for action. since the information needed by Cache B will be available in main store. Thus, the action of Processor A in updating the line in Cache A should also mark the line in Cache B invalid, so that when Processor B needs the information, it will be required to go to main store to find it. This is not the only solution to the cache coherency problem, but it does indicate why systems capable of multiprocessing organizations often choose a write through scheme as opposed to the write back scheme.

Example 7.9: A cache coherency solution: The Sequent system is a shared memory multiprocessing organization. What mechanism is used to allow each processor to have its own cache?

The Sequent system is a very interesting combination of *the* various mechanisms discussed, both for interface methods and for caching policies. A block diagram of a sample Sequent system is shown in Figure 7.37. The CPU module utilized in this system is the 80386, and included with it are the other devices that allow it to create the information needed by the memory

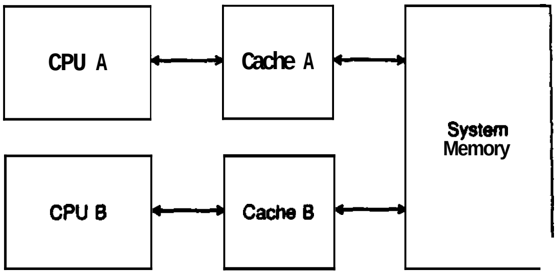


Figure 7.36. Computer System with Two CPUs and Two Caches.

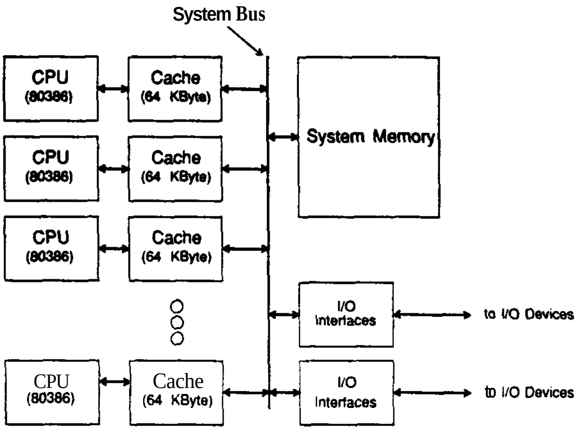


Figure 737. Block Diagram of a Sequent **Multiprocessing System.**

system. That is. there is a virtual-to-real address translation mechanism included with each CPU. Also. high speed floating point units that utilize the IEEE floating point system are available, which will enhance the normal floating point capabilities of the 80386 chip set. The cache system that provides the CPU with the information it needs is a 2 way set associative cache, with a capacity of 64 Kbytes. It is organized with a line size of 16 bytes; this gives 4,096 lines. or 2,048 sets. The mechanism used by the cache to communicate with the main store is a modified write back policy, which violates the intuitive feeling about how cache memory systems for multiple processor/multiple cache systems should work. The reason that the system functions properly can be understood by looking at the mechanisms included with the system bus.

The system bus is a synchronous, time multiplexed bus similar in function to the SBI. However, there ate some important differences. The speed of the bus transactions is 100 nsec, and the mechanism for data transfers is address-datadata. as in the SBI. However, the width of the data path is larger in the Sequent than the 4-byte data path of the SBI. During the address portion of the transfer, the address is asserted on the lines. During the data portion of the transfer, 8 bytes of information are placed on the bus. This allows 16 bytes to be transferred in one addressdatadata exchange. This is designed to be the amount of information in a line, and so transfers from memory to the caches always occur in increments of one line. With 16 bytes per 300 nsec, this leads to a maximum data rate of over 50 Mbytes/sec.

Solving the cache coherence problem requires the work not only of the system memory, but also of all the cache systems as well. The cache modules are organized such that there is an interface to the CPU for the transfer of information to and from the processor, as well as a watch dog interface. which monitors all of the transactions on the system bus. Since the bus is a synchronous protocol, this can be effectively managed. The interface between the cache and the system bus serves two purposes. The first is to exchange information with the system memory to maintain the cache information as a normal cache system should. The second purpose is to watch for memory transactions that take place with the lines it **currently** has in the cache. The second function permits the write back policy to be utilized in the system.

The problem explained above in Figure 7.36 involved two processors accessing the same location in memory. That set of events would **proceed** in a slightly different fashion in the Sequent system:

- Processor A accesses location XYZ in system memory. The system memory responds with the appropriate line. Cache A accepts the line and uses it. However. it keeps track of the fact that it has a private copy of the line.
- Processor B accesses location XYZ in system memory. The system memory responds with the appropriate line. Cache A also has this line: however, Cache A keeps track of the fact that it has not modified the line, so that the copy in system memory **and** the copy in Cache A **are** identical. . Cache B accepts the line and uses it. However, it keeps track of the fact that it has a *shared* copy of the line. In addition, the watch dog interface on Cache A also notes that the line has been obtained by another cache, and marks it as shared.
- Processor A modifies location XYZ; this takes place in Cache A. and does not propagate to system memory. However. Cache A does send out onto the bus a notice to other caches that the line has been modified. The watch dog interface on Cache B sees this and marks the line as invalid in the cache.
- Processor B accesses location XYZ; the line in Cache B has been marked invalid, and the cache then goes to the system bus to get the information. When Cache A notices that someone needs information from location XYZ, and that it has the updated copy of that information, it signals the system memory not to **respond** to the request, and the information comes from Cache A instead Thus, when Cache B requests the information, it does not know from what source the information will come, only that some bus cycles later the information will be provided on the bus.

The use of active interfaces between the cache and the system bus to monitor the data transfers on the bus allows the write back mechanism to function properly. The system will execute the programs as specified by the instruction streams. This organization also allows the creation of the locks needed for system operation. The interlock instructions of the 80386 are executed on a location in memory. The cache first obtains a private copy of the location, and then does not respond itself (nor allows the system memory to respond) until the interlocked transaction is complete.

At this time some comments on cache systems and their **utilization** are in order. First of all, in our discussion on caches, we **assumed** that the available **address directed** the **cache** to the proper spot to find the information requested. However, the question studiously avoided was, which **address** should be **presented** to the cache? Is it a virtual address or a physical address? Machines have been built that utilize **virtual** addresses for the cache access. but the **more** common mechanism is to use physical addresses. The problem remains, then to provide the **virtual-to-real** address translation to diict **the** request to the proper location. To aid in this **process,** an often used mechanism is called the translation **lookaside** buffer **(TLB).** This table, located in fast registers or fast memory, contains the most recent virtual to real translations. and. with this information, the proper address can be presented to **the** cache. This unit **can** be organized in a fashion similar to the cache, or in any manner that will satisfy system requirements for rapid address generation. Basically, this unit provides for the cache memory **the same** function provided by the 32082 for virtual memory — maintain in a rapidly accessible location the translations needed by the system.

As the relative **costs** of system resources change, different approaches to the organization of the cache may be appropriate. In this section we have discussed some of the mechanisms used with set associative caches. One of the mechanisms is the replacement of information within a cache. The decision made concerning which member of a set to replace requires hardware to implement — 'hardware (memory) to remember something about the order in which the members of the set have been accessed. **and** hardware to use that information to ascertain the proper member to replace. As memory costs continue to drop, one of the approaches that becomes more attractive is to use larger cache sizes and a diict mapping policy. With **direct** mapping, the hardware needed to maintain replacement information and to determine replacement priorities is nonexistent, since the target location can be found in only one location in a cache. By using this mechanism. the apparent speed can increase. since no multiplexing is needed between members of a set. At the same time. the hit ratio remains high because the cache is sufficiently large to provide the information needed.

Cache systems allow processors to obtain the data that they need in a timely fashion. So long as the information required by a processor is in the cache, then processing continues without **unneeded** delays for slower memory systems. This mechanism is useful in both "standard" memory systems for uniprocessor systems and for multiprocessing organizations. The added benefit in multiprocessing systems is to localize the **information** quests and to minimize the requests to system **memory.**

7.5. Summary

Memory systems in computers **are** used to maintain the programs and **data** needed by user and operating system alike. In all system components, information is maintained in devices with two stable states; this enables representation of a "one" and a "zero." Collections of these memory mechanisms allow the system to "remember" **information** that it needs. The principal requirement for an effective memory mechanism is **the** ability to store and retrieve the information in an organized fashion.

The speed and retrieval mechanisms **used** by a memory system lead to differing functions. The slower, serially organized elements **are used** to maintain large files and **other information** that can be effectively retrieved in a serial fashion, instead of a random access scheme. **The** faster storage mechanisms **are** used to maintain information **accessed** by the computing **system** in a time critical fashion. **The** storage elements that can supply **information** in **the** shortest time **are** used in register **and** cache systems: **elements** that **are** not quite so fast can be **effectively** used as main **memory** elements. Organization of the random access elements can be done in a 1-D or 2-D fashion or with similar mechanisms, the. requirement being that only one of the various locations is accessed at any one time.

Virtual memory systems allow the user to operate in "virtual space," which is not the same as the actual physical space. The virtual machine is the view of the system as seen by the user, and includes those resources of which the user is aware. The physical machine can be quite different; the physical model is limited by the exact configuration of the system. The virtual machine concept allows users to use more resources than the system actually has, such as larger memory spaces. The same concept allows more memory resources to be shared among many users who are not aware of the entire extent of the system. The virtual memory mechanisms, segmentation and paging, translate requests from the virtual system to the actual physical system. This results in systems with a higher apparent system speed because of the locality observed in programs: during a small portion of the program, only a fraction of the total memory is used by the system.

The locality of programs also allows cache systems to function effectively. The cache allows a small, high speed memory to keep only the most active portions of a program and its data accessible to the CPU. But, since it operates at CPU speeds, the cache mechanism speeds up the overall processing rate of the computer.

7.6. Problems

7.1 Design a 16 element register bank using I-D techniques. For register elements use the '299 as shown in Figure 7.9.

72 Design a 16 element register bank using 2-D techniques. For register elements use the '299 as shown in Figure 7.9.

7.3 The IDT7164 is an 8K x 8 static RAM with 13 address lines and 8 input/output lines, as well as two chip selects (CSI-I, CS2-H), a write enable (WE-L), and an output enable (OE-L). Using this device, design a 64-Kbyte memory using I-D organizational techniques. How many nonmemory devices are required for this memory system?

74 Use the IDT7164 described in Problem 7.3 to create a 128-Kbyte memory system. Use 2-D organizational techniques. How many nonmemory devices are required for the memory system?

7.5 Design the dynamic RAM controller shown in Figure P7.5. The inputs are an 18 line data address, a 9 line refresh address, a refresh request line (REF-H), and a data read line (READ-H) that initiates a read action. The outputs are a 9 line address, which is to go to the dynamic RAM, the row address strobe (RAS-L), the column address strobe (CAS-L), and the ready line (READY-H).

The behavior of the &vice is as follows: A refresh cycle is accomplished by asserting the refresh address on the RAM address lines (and allowing 50 nscc settling time), then asserting the RAS signal for 150 nscc. and releasing both RAS and the address lines. A read cycle is accomplished by asserting 9 bits of the address (and waiting the 50 nsec required for settling), asserting RAS. waiting another 50 nsec, then changing to the other 9

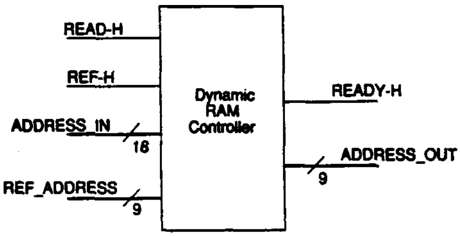


Figure P7.5. Dynamic RAM Controller.

address bits, (and another 50 nsec wait) asserting CAS, waiting 100 nsec, and then asserting the ready line. This condition remains until the READ is released. whereupon the system returns to the quiescent state. Assume a system clock at 20 MHz. To complete the design:

a. Give a data path block diagram. Assume the existance of N-bit 2-1 muxes with hi-state outputs and N-bit tri-state drivers. (These could be constructed from multiple copies of a '257.)

b. Give a state diagram describing the action of the device. Include signal names and assertion levels.

c. Design a circuit to do the work of pen b.

d. Describe what modifications or additional logic would be required to implement the write capability as well as the read capability.

7.6 The block diagram for a general purpose system shown with Problem 4.7 is included here as Figure P7.6. Modify the block diagram to include a segmentation register. That is, provide a way that all addresses to memory can be offset by the value in a segment register. Describe the modifications that

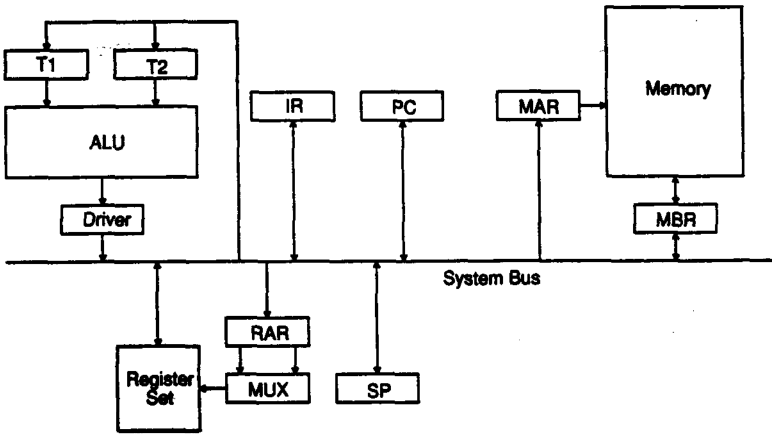


Figure P7.6. Block Diagram for General Purpose Machine.

must be made to an instruction set to control the system with this modification.

7.7 In general, the single segment register of Problem 7.6 does not provide sufficient capabilities to a computer system; a system must be capable of handling several segments. Modify the block diagram of Figure P7.6 to include the capability of several segment addresses. What contributions are made by this enhancement of the arrangement of Problem 7.6?

7.8 Access to a memory system by a number of programs can be enhanced by the use of paging. Modify the block diagram of Figure P7.6 to include a hardware page table, and specify the manner in which the page table is used to generate the effective memory address. Why will this method be faster than the segmentation mechanism of Problem 7.7?

79 A computer system is configured with a disk to provide high speed file storage. The disk system has 32 sectors per track, and stores 512 bytes per sector. The rotational speed of the disk is 3.600 RPM. The average seek time is 30 msec. The average instruction execution time is 1.5 µsec. When a page fault occurs, how much time will be required before execution can continue on that program? Identify each of the contributing times, and describe what is happening during that time. How much would the delay be modified if the seek time of the disk were reduced to 24 msec?

7.10 Consider a cache organization with the following characteristics:

Main memory size:	16 Mbytes
Cache memory size:	32 Kbytes
Cache line size:	64 bytes
Cache cycle time:	50 nanoseconds
Main memory cycle:	500 nanoseconds
Robability of cache hit:	.7
Robability of write:	.25
Rubability line d i :	.2
Cache organization:	4 way set associative

a. Give a representation of the address space. That is, what bits in the address are for what? Assume that then are 2 bits between the byte identifying bits and the set identifying bits.

b. Assume that 128 X 8 memories are used to build the cache. Give a data path block diagram of the cache system. Assume that no parity checking is needed, and that the cache is a write back cache.

c. Find the effective time for a memory access.

7.11 A certain cache memory machine uses a cache that can store 512 blocks of 64 words each. Assume a main memory size of 1,048,576 words.

a. Which bits of the word address should specify the block number?

b. If a *set* associative scheme is used, which bits should specify the *set* number?

c. Describe the worst *case* reference pattern (for maximum cache miss) assuming (i) direct addressing, (ii) *set* associative with two blocks per set, (iii) *set* associative with four blocks per set, (iv) fully associative cache allocation. How likely are these worst *cases?*

7.12 A certain computer has a 16-Mbyte main memory, cycle time of 350 nsec. It also has a 32-Kbyte cache, cycle time of 25 nsec. The cache is set associative, four lines per set. 32 bytes per line.

a. The address space is partially defined below. Complete the specification.

Tag Bits	Set Bits	Tag Bits	Byte in Line
		3	5

b. Assume that an address trace of a program is such that the lines in use are accessed in the following order ... 0 1 2 3 0 1 2 4 0 1 2 3 0 1 2 4 ... If this addressing pattern is continued, what will the effective memory access time be?

7.13 Develop a formula for the effective access time for a cache memory that uses a write through scheme. Assume that writes that are cache misses do not have any effect on the cache at all. except for the time involved in the transaction. Assume also that no buffering is provided for the writes to main store. State all assumptions that you make in the process of problem solution.

7.14 Repeat Problem 7.13 assuming that a buffer is available between the cache and main store. This buffer will allow the operation of the memory system to continue once a write has been initiated. However. if a write is needed and the buffer is in use, the system must wait until the data in the buffer has been transferred to main store.

7.7. References and Readings

[AMD85] Advanced Micro Devices, *Bipolar Microprocessor Logic and Interface Data Book.* Sunnyvale. CA: Advanced M i Devices, 1985.

[AhDe71] Aho, A. V. P. J. Denning, and J. D. Ullman, "Principles of Optimal Page Replacement," *Journal of the ACM.* Vol. 18, No. 1, January 1971, pp. 80–93.

[Baer84] Baer, J. L., "Computer Architecture," *Computer.* Vol 17, No 10, October 1984, pp. 77–87.

[Baer80] Baer, J. L., *Computer System Architecture.* Rockville, MD: Computer Science Press, 1980.

[Bart85] Bartee, T. C., *Digital Computer Fundamentals 6th* edition, New York: McGraw Hill Book Company. 1985.

[BaBr77] Batson, A. A., and R. E. Brundage, "Segment S i and Lifetimes in Algol 60 Programs," *Communications of the ACM.* Vol. 20, No. 1, January 1977, pp. 36–44.

[BaJu70] Batson, A., S. Ju, and D. C. Wood. "Measurements of Segment Size," *Communications of the ACM.* Vol. 13, No. 3, March 1970, pp. 155–159.

[BeNe69] Belady, L. A, R. A. Nelson, and G. S. Shedler, "An Anomaly in the Space-Time Characteristics of Certain Programs Running in Paging Machines," *Communications of the ACM.* Vol. 12, No. 6, June 1969, pp. 349–353.

[BeNe71] Bell, C. G. and A. Newell, *Computer Structures: Readings and Examples.* New York: McGraw Hill Book Company, 1971.

[BiSh88] Bic, L., and A. C. Shnw, *The Logical Design of Operating System.* Englewood Cliffs, NJ: Prentice Hall. 1988.

[BrHa73] Brinch Hansen, P.. *Operating Systems Principles.* Englewood Cliffs. NJ: Prenticc Hall. 1973.

[BuGo46] Burks, A. W.. H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," Institute for Advanced Studies, 1946, reprinted in [Swar76].

[ChOp76] Chu, W. W.. and H. Opderbeck, "Program Behavior and the Page-Fault-Frequency Replacement Algorithm." *Computer.* Vol. 9, No. 11, November 1976, pp. 29–38.

[Deit84] Deitel, H. M., *An Introduction to Operating Systems.* Reading, MA: Addison-Wesley Publishing Company. 1984.

[Denn70] Denning, P. J., "Virtual Memory," ACM *Computing Surveys.* Vol. 2, No. 3, September 1970, pp. 153–189.

[Denn80] Denning, P. J., "Working Sets Past and Present." *IEEE Transactions on Software Engineering.* Vol. SE-6, No. I. January 1980, pp. 64–84.

[Flet80] Fletcher. W. I., *An Engineering Approach to Digital Design.* Englewood Cliffs, NJ: Prentice Hall. 1980.

[FoME85] Fossum, T.. J. B. McElroy, and W. English, "An Overview of the VAX 8600 System." *Digital Technical Journal.* Hudson. MA: Digital Equipment Corporation. 1985. pp. 8–23.

[GeTi83] Gelenbe, E.. P. Tiberio, and J. Boekhorst. "Page Size in Demand Paging Systems." *Acta Informatica.* Vol. 3. No. 1, 1973. pp. 1–23.

[BuFr78] Gupta, R. K., and M. A. Franklin, "Working Set and Page Fault Frequency Replacement Algorithms. A Performance Comparison," *IEEE Transactions on Computers.* Vol. C-27, No. 8, August 1978, pp. 706–712.

[HaVr78] Hamacher, V. C.. Z. G. Vranesic, and S. G. Zaky, *Computer Organization.* New York: McGraw Hill Book Canpany. 1984.

[Haye88] Hayes. I. P., *Computer Architecture and Organization,* 2nd Edition. New York: McGraw Hill Book Company, 1988.

[Kain89] Kain, R Y.. *Computer Architecture, Software and Hardware.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[KaWi73] Kaplan, K. R. and R. O. Winder, "Cache-Based Computer Systems." *Computer* Vol. 6, No. 3, March 1973, pp. 30–36.

[KnRa75] Knuth, D. E., and G. S. Rao, "Activity in *Interleaved* Memory," *IEEE Transactions on Computers.* Vol. C-24, No. 9, September 1975, pp. 943–944.

[Kuck78] Kuck, D. J.. *The Structure of Computers and Computations.* New York: John Wiley & Sons, 1978.

[Lang82] Langdon, G. G.. Jr., *Computer Design.* San Jose, CA: Computeach Press Inc., 1982.

[Lipt68] Liptay, J. S., "Structural Aspects of the System/360 Model 85 — The Cache," *IBM Systems Journal.* Vol. 7, No. 1, 1968., pp. 15–21.

[Mati80] Matick, R. E., "Memory and Storage," in [Ston80], pp. 205–274.

[Mati77] Matick, R. E., *Computer Storage Systems and Technology.* New York: John Wiley & Sons, 1977.

[MaGe70] Mattson, R. L., J. Gecsei, D. L. Slutz, et al., "Evaluation Techniques for Storage Hierarchies." *IBM System Journal.* Vol. 9, No. 2, 1970. pp. 78–117.

[PeSi83] Peterson, J. L.. and A. Silberschatz, *Operating System Concepts.* Reading, *MA:* Addison-Wesley Publishing Company. 1983.

[Pohm84] Pohm, A. V., "High Speed Memory Systems," *Computer.* Vol. 17. No. 10, October 1984. pp. 162–171.

[Rand69] Randell, B., "A Note on Storage Fragmentation and Program Segmentation," *Communications of the ACM.* Vol. 12, No. 7, July 1969. pp. 365–369.

[Schn85] Schneider. G. M. *The Principles of Computer Organization.* New York: John Wiley & Sons, 1985.

[Shiv85] Shiva, S. G., *Computer Design and Architecture.* Boston, *MA:* Little, Brown. 1985.

[SiBe82] Siewiorek. D. P.. C. G. Bell. and A. Newell, *Computer Structures: Principles and Examples.* New York: McGraw Hill Book Company. 1982.

[Smit82] Smith. A. J.. "Cache Memories," ACM *Computing Surveys.* Vol. 14, No. 3, September 1982, pp. 473–530.

[Smit85] Smith. A. J.. "Cache Evaluation and the Impact of Workload Choice," *Proceedings of the 12th Annual International Symposium on Computer Architecture. Silver Springs, MD: IEEE Computer Society Press.* June 1985, pp. 64–73.

[Smit00] Smith, A. I.. "Disk-Cache-Miss Ratio Analysis and Design Considerations," *ACM Transactions on Computer Systems* Vol. 3, No. 3. August 1985. pp. 161–203.

[SmGo83] Smith, J. E.. and J. R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies," *Proceedings of the 10th Annual Symposium on Computer Architecture.* June 1983, pp. 117–123.

[Ston87] Stone. H. S., *High-Performance Computer Architecture.* Reading, *MA:* Addison-Wesley Publishing Company. 1987.

[Stre83] Strecker, W. D., "Transient Behavior of Cache Memories," *ACM Transactions on Computer System.* Vol. 1, No. 4, November 1983, pp. 281–293.

[Swar76] Swartzlander, E. E., Jr. (Ed), *Computer Design Development: Principal Papers.* Rochelle Park, NJ: Hayden Book Company, Inc., 1976.

[Tane84] Tanenbaum, A. S.. *Structured Computer Organization.* Englewood Cliffs, *NJ:* Prentice Hall, 1984.

[TI85] Texas Instruments, The *TTL Data Book.* Volume 2. Dallas, TX: Texas Instruments, 1985.

[ThKn86] Thakkar, S. S., and Knowles, A. E., "A High Performance Memory Management Scheme." *Computer.* Vol. 19, No. 5, May 1986, pp. 8–19.

[Wilk87] Wilkinson, B., *Digital System Design.* Englewood Cliffs, NJ: Prentice Hall International, 1987.