



MEMORY

In the past few decades, CPU processing speed as measured by the number of instructions executed per second has doubled every 18 months, for the same price. Computer memory has experienced a similar increase along a different dimension, quadrupling in size every 36 months, for the same price. Memory speed, however, has only increased at a rate of less than 10% per year. Thus, while processing speed increases at the same rate that memory size increases, the gap between the speed of the processor and the speed of memory also increases.

As the gap between processor and memory speeds grows, architectural solutions help bridge the gap. A typical computer contains several types of memory, ranging from fast, expensive internal registers (see Appendix A), to slow, inexpensive removable disks. The interplay between these different types of memory is exploited so that a computer behaves as if it has a single, large, fast memory, when in fact it contains a range of memory types that operate in a highly coordinated fashion. We begin the chapter with a high-level discussion of how these different memories are organized, in what is referred to as the **memory hierarchy**.

7.1 The Memory Hierarchy

Memory in a conventional digital computer is organized in a hierarchy as illustrated in Figure 7-1. At the top of the hierarchy are registers that are matched in speed to the CPU, but tend to be large and consume a significant amount of power. There are normally only a small number of registers in a processor, on the order of a few hundred or less. At the bottom of the hierarchy are secondary and off-line storage memories such as hard magnetic disks and magnetic tapes, in which the cost per stored bit is small in terms of money and electrical power, but the access time is very long when compared with registers. Between the registers and secondary storage are a number of other forms of memory that bridge the gap between the two.

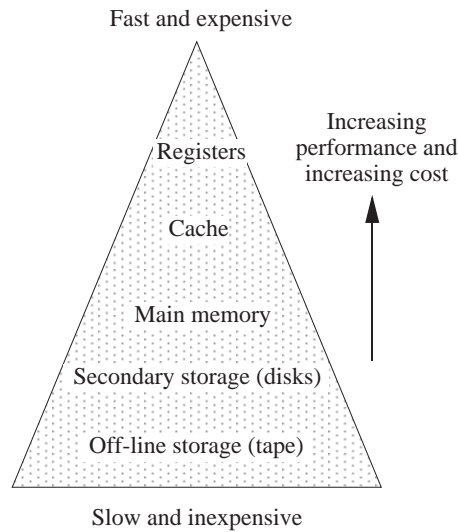


Figure 7-1 The memory hierarchy.

As we move up through the hierarchy, greater performance is realized, at a greater cost. Table 7- 1 shows some of the properties of the components of the memory

Memory Type	Access Time	Cost /MB	Typical Amount Used	Typical Cost
Registers	1ns	High	1KB	–
Cache	5-20 ns	\$100	1MB	\$100
Main memory	60-80ns	\$1.10	64 MB	\$70
Disk memory	10 ms	\$0.05	4 GB	\$200

Table 7- 1 Properties of the memory hierarchy

hierarchy in the late 1990's. Notice that Typical Cost, arrived at by multiplying $\text{Cost/MB} \times \text{Typical Amount Used}$ (in which "MB" is a unit of megabytes), is approximately the same for each member of the hierarchy. Notice also that access times vary by approximately factors of 10 except for disks, which have access times 100,000 times slower than main memory. This large mismatch has an important influence on how the operating system handles the movement of blocks of data between disks and main memory, as we will see later in the chapter.

7.2 Random Access Memory

In this section, we look at the structure and function of **random access memory** (RAM). In this context the term “random” means that any memory location can be accessed in the same amount of time, regardless of its position in the memory.

Figure 7-2 shows the functional behavior of a RAM cell used in a typical com-

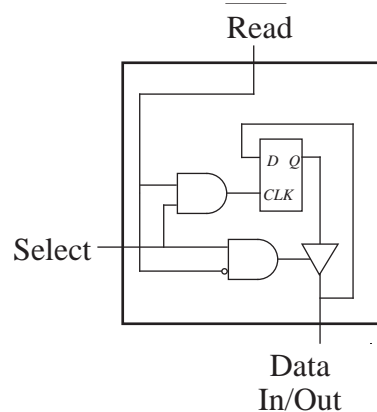


Figure 7-2 Functional behavior of a RAM cell.

puter. The figure represents the memory element as a D flip-flop, with additional controls to allow the cell to be selected, read, and written. There is a (bidirectional) data line for data input and output. We will use cells similar to the one shown in the figure when we discuss RAM chips. Note that this illustration does not necessarily represent the actual physical implementation, but only its functional behavior. There are many ways to implement a memory cell.

RAM chips that are based upon flip-flops, as in Figure 7-2, are referred to as **static** RAM (SRAM), chips, because the contents of each location persist as long as power is applied to the chips. **Dynamic RAM** chips, referred to as **DRAMs**, employ a **capacitor**, which stores a minute amount of electric charge, in which the charge level represents a 1 or a 0. Capacitors are much smaller than flip-flops, and so a capacitor based DRAM can hold much more information in the same area than an SRAM. Since the charges on the capacitors dissipate with time, the charge in the capacitor storage cells in DRAMs must be restored, or **refreshed** frequently.

DRAMs are susceptible to premature discharging as a result of interactions with naturally occurring gamma rays. This is a statistically rare event, and a system

may run for days before an error occurs. For this reason, early personal computers (PCs) did not use error detection circuitry, since PCs would be turned off at the end of the day, and so undetected errors would not accumulate. This helped to keep the prices of PCs competitive. With the drastic reduction in DRAM prices and the increased uptimes of PCs operating as automated teller machines (ATMs) and network file servers (NFSs), error detection circuitry is now commonplace in PCs.

In the next section we explore how RAM cells are organized into chips.

7.3 Chip Organization

A simplified pinout of a RAM chip is shown in Figure 7-3. An m -bit address,

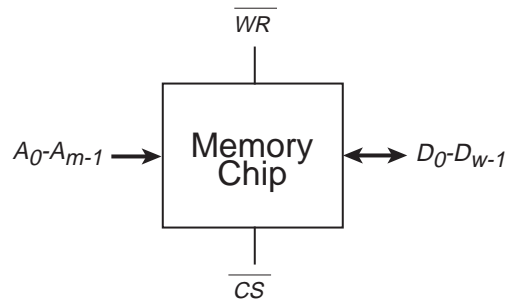


Figure 7-3 Simplified RAM chip pinout

having lines numbered from 0 to $m-1$ is applied to pins A_0 - A_{m-1} , while asserting \overline{CS} (Chip Select), and either \overline{WR} (for writing data to the chip) or WR (for reading data from the chip). The overbars on \overline{CS} and \overline{WR} indicate that the chip is selected when $CS=0$ and that a write operation will occur when $WR=0$. When reading data from the chip, after a time period t_{AA} (the time delay from when the address lines are made valid to the time the data is available at the output), the w -bit data word appears on the data lines D_0 - D_{w-1} . When writing data to a chip, the data lines must also be held valid for a time period t_{AA} . Notice that the data lines are bidirectional in Figure 7-3, which is normally the case.

The address lines A_0 - A_{m-1} in the RAM chip shown in Figure 7-3 contain an address, which is decoded from an m -bit address into one of 2^m locations within the chip, each of which has a w -bit word associated with it. The chip thus contains $2^m \times w$ bits.

Now consider the problem of creating a RAM that stores four four-bit words. A RAM can be thought of as a collection of registers. We can use four-bit registers to store the words, and then introduce an addressing mechanism that allows one of the words to be selected for reading or for writing. Figure 7-4 shows a design

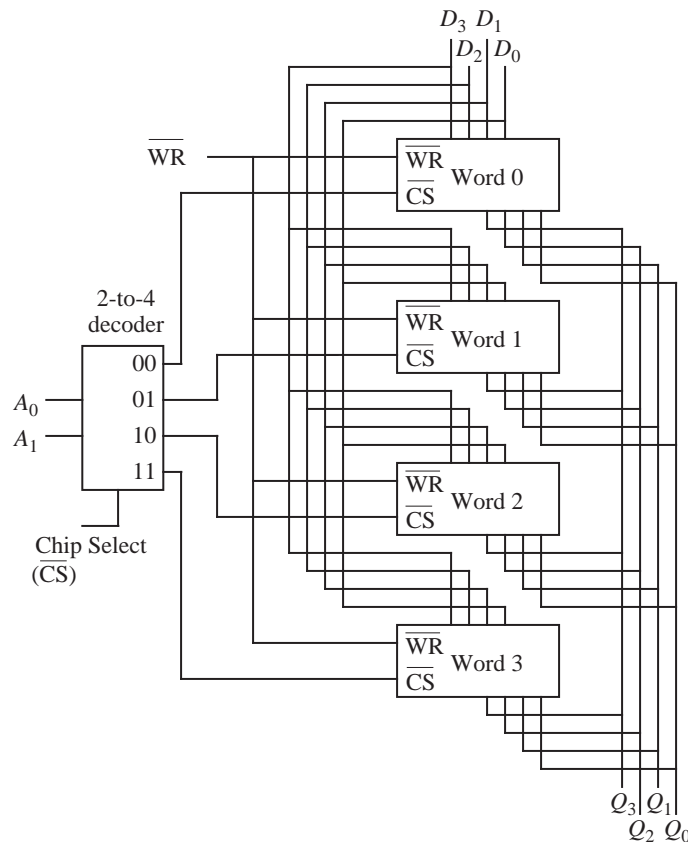


Figure 7-4 A four-word memory with four bits per word in a 2D organization.

for the memory. Two address lines A_0 and A_1 select a word for reading or writing via the 2-to-4 decoder. The outputs of the registers can be safely tied together without risking an electrical short because the 2-to-4 decoder ensures that at most one register is enabled at a time, and the disabled registers are electrically disconnected through the use of tri-state buffers. The Chip Select line in the decoder is not necessary, but will be used later in constructing larger RAMs. A simplified drawing of the RAM is shown in Figure 7-5 .

There are two common ways to organize the generalized RAM shown in Figure 7-3. In the smallest RAM chips it is practical to use a single decoder to select one

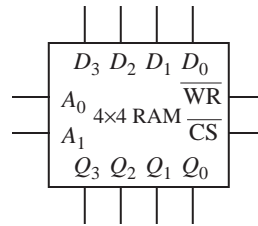


Figure 7-5 A simplified version of the four-word by four-bit RAM.

out of 2^m words, each of which is w bits wide. However, this organization is not economical in ordinary RAM chips. Consider that a $64\text{M} \times 1$ chip has 26 address lines ($64\text{M} = 2^{26}$). This means that a conventional decoder would need 2^{26} 26-input AND gates, which manifests itself as a large cost in terms of chip area – and this is just for the decode.

Since most ICs are roughly square, an alternate decoding structure that significantly reduces the decoder complexity decodes the rows separately from the columns. This is referred to as a 2-1/2D organization. The 2-1/2D organization is by far the most prevalent organization for RAM ICs. Figure 7-6 shows a 2^6 -word

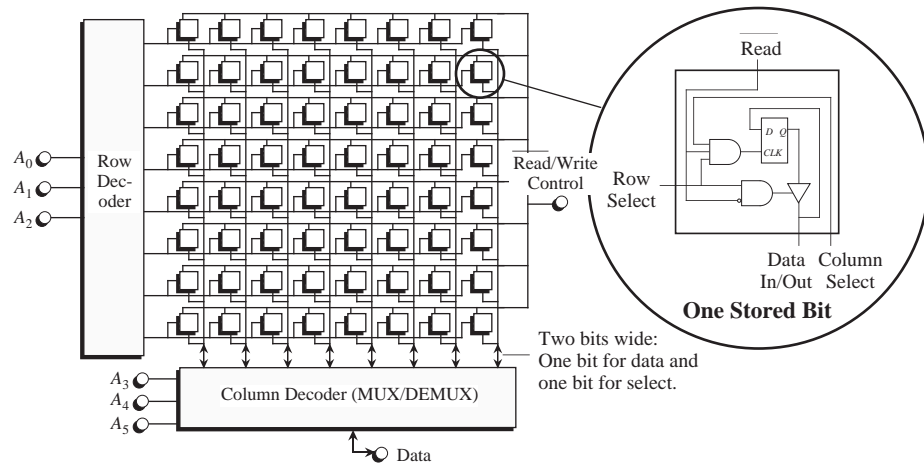


Figure 7-6 2-1/2D organization of a 64-word by one-bit RAM.

$\times 1$ -bit RAM with a 2 1/2D organization. The six address lines are evenly split between a row decoder and a column decoder (the column decoder is actually a MUX/DEMUX combination). A single bidirectional data line is used for input and output.

During a read operation, an entire row is selected and fed into the column

MUX, which selects a single bit for output. During a write operation, the single bit to be written is distributed by the DEMUX to the target column, while the row decoder selects the proper row to be written.

In practice, to reduce pin count, there are generally only $m/2$ address pins on the chip, and the row and column addresses are time-multiplexed on these $m/2$ address lines. First, the $m/2$ -bit row address is applied along with a row address strobe, $\overline{\text{RAS}}$, signal. The row address is latched and decoded by the chip. Then the $m/2$ -bit column address is applied, along with a column address strobe, $\overline{\text{CAS}}$. There may be additional pins to control the chip refresh and other memory functions.

Even with this 2-1/2D organization and splitting the address into row and column components, there is still a great fanin/fanout demand on the decoder logic gates, and the (still) large number of address pins forces memory chips into large footprints on printed circuit boards (PCBs). In order to reduce the fanin/fanout constraints, **tree decoders** may be used, which are discussed in Section 7.8.1. A newer memory architecture that serializes the address lines onto a single input pin is discussed in Section 7.9.

Although DRAMs are very economical, SRAMs offer greater speed. The refresh cycles, error detection circuitry, and the low operating powers of DRAMs create a speed difference that is roughly 1/4 of SRAM speed, but SRAMs also incur a significant cost.

The performance of both types of memory (SRAM and DRAM) can be improved. Normally a number of words constituting a **block** will be accessed in succession. In this situation, memory accesses can be **interleaved** so that while one memory is accessing address A_m , other memories are accessing A_{m+1} , A_{m+2} , A_{m+3} etc. In this way the access time for each word can appear to be many times faster.

7.3.1 CONSTRUCTING LARGE RAMS FROM SMALL RAMS

We can construct larger RAM modules from smaller RAM modules. Both the word size and the number of words per module can be increased. For example, eight $16\text{M} \times 1$ -bit RAM modules can be combined to make a $16\text{M} \times 8$ -bit RAM module, and 32 $16\text{M} \times 1$ -bit RAM modules can be combined to make a $64\text{M} \times 8$ -bit RAM module.

As a simple example, consider using the 4 word \times 4-bit RAM chip shown in Figure 7-5, as a building block to first make a 4-word \times 8-bit module, and then an 8-word \times 4-bit module. We would like to increase the width of the four-bit words and also increase the number of words. Consider first the problem of increasing the word width from four bits to eight. We can accomplish this by simply using two chips, tying their CS (chip select) lines together so they are both selected together, and juxtaposing their data lines, as shown in Figure 7-7.

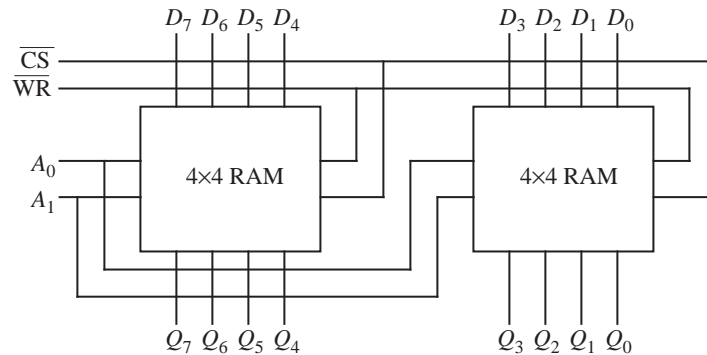


Figure 7-7 Two four-word by four-bit RAMs are used in creating a four-word by eight-bit RAM.

Consider now the problem of increasing the number of words from four to eight. Figure 7-8 shows a configuration that accomplishes this. The eight words are distributed over the two four-word RAMs. Address line A_2 is needed because there are now eight words to be addressed. A decoder for A_2 enables either the upper or lower memory module by using the \overline{CS} lines, and then the remaining address lines (A_0 and A_1) are decoded within the enabled module. A combination of these two approaches can be used to scale both the word size and number of words to arbitrary sizes.

7.4 Commercial Memory Modules

Commercially available memory chips are commonly organized into standard configurations. Figure 7-9 (Texas Instruments, 1991) shows an organization of eight 2^{20} -bit chips on a single-in-line memory module (SIMM) that form a $2^{20} \times 8$ (1MB) module. The electrical contacts (numbered 1 – 30) all lie in a single line. For 2^{20} memory locations we need 20 address lines, but only 10 address lines (A0 – A9) are provided. The 10-bit addresses for the row and column are loaded separately, and the Column Address Strobe and Row Address Strobe signals are applied after the corresponding portion of the address is made available. Although this organization appears to double the time it takes to access any par-

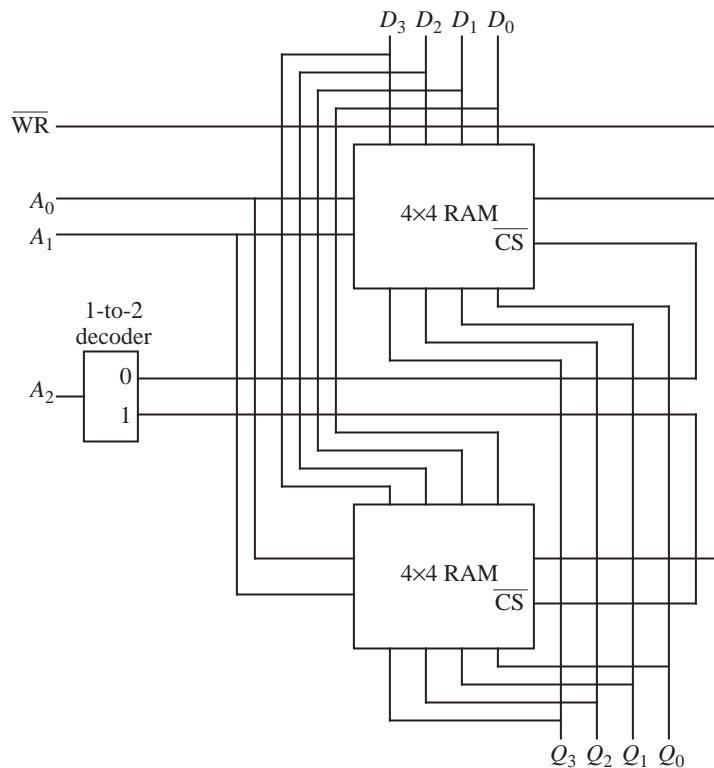


Figure 7-8 Two four-word by four-bit RAMs are used in creating an eight-word by four-bit RAM.

particular memory location, on average, the access time is much better since only the row or column address needs to be updated.

The eight data bits on lines $DQ_1 - DQ_8$ form a byte that is read or written in parallel. In order to form a 32-bit word, four SIMM modules are needed. As with the other “active low” signals, the Write Enable line has a bar over the corresponding symbol (\overline{W}) which means that a write takes place when a 0 is placed on this line. A read takes place otherwise. The RAS line also causes a refresh operation, which must be performed at least every 8 ms to restore the charges on the capacitors.

7.5 Read-Only Memory

When a computer program is loaded into the memory, it remains in the memory until it is overwritten or until the power is turned off. For some applications, the program never changes, and so it is hardwired into a **read-only memory**

shows a four-word ROM that stores four four-bit words (0101, 1011, 1110, and 0000). Each address input (00, 01, 10, or 11) corresponds to a different stored word.

For high-volume applications, ROMs are factory-programmed. As an alternative, for low-volume or prototyping applications, programmable ROMs (PROMs) are often used, which allow their contents to be written by a user with a relatively inexpensive device called a **PROM burner**. Unfortunately for the early videogame industry, these PROM burners are also capable of reading the contents of a PROM, which can then be duplicated onto another PROM, or worse still, the contents can be deciphered through reverse engineering and then modified and written to a new, contraband game cartridge.

Although the PROM allows the designer to delay decisions about what information is stored, it can only be written once, or can be rewritten only if the existing pattern is a subset of the new pattern. Erasable PROMs (EPROMs) can be written several times, after being erased with ultraviolet light (for UV PROMs) through a window that is mounted on the integrated circuit package. Electrically erasable PROMs (EEPROMs) allow their contents to be rewritten electrically. Newer **flash** memories can be electrically rewritten tens of thousands of times, and are used extensively in digital video cameras, and for control programs in set-top cable television decoders, and other devices.

PROMs will be used later in the text for control units and for **arithmetic logic units** (ALUs). As an example of this type of application, consider creating an ALU that performs the four functions: Add, Subtract, Multiply, and Divide on eight-bit operands. We can generate a truth table that enumerates all 2^{16} possible combinations of operands and all 2^2 combinations of functions, and send the truth table to a PROM burner which loads it into the PROM.

This brute force **lookup table** (LUT) approach is not as impractical as it may seem, and is actually used in a number of situations. The PROM does not have to be very big: there are $2^8 \times 2^8$ combinations of the two input operands, and there are 2^2 functions, so we need a total of $2^8 \times 2^8 \times 2^2 = 2^{18}$ words in the PROM, which is small by current standards. The configuration for the PROM ALU is shown in Figure 7-11. The address lines are used for the operands and for the function select inputs, and the outputs are produced by simply recalling the precomputed word stored at the addressed location. This approach is typically faster than using a hardware implementation for the functions, but it is not extensible to large word widths without applying some form of decomposition.

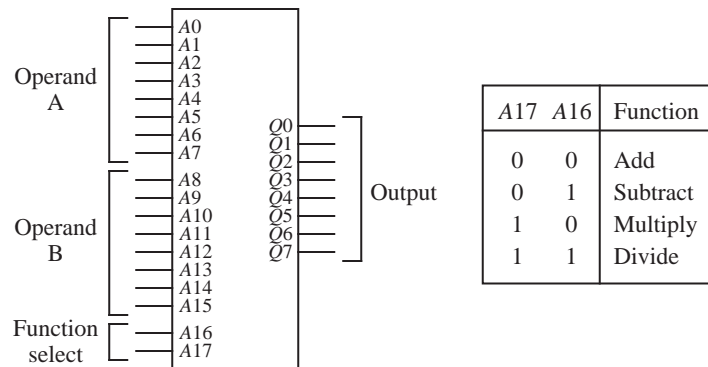


Figure 7-11 A lookup table (LUT) implements an eight-bit ALU.

32-bit operands are standard on computers today, and a corresponding PROM ALU would require $2^{32} \times 2^{32} \times 2^2 = 2^{66}$ words which is prohibitively large.

7.6 Cache Memory

When a program executes on a computer, most of the memory references are made to a small number of locations. Typically, 90% of the execution time of a program is spent in just 10% of the code. This property is known as the **locality principle**. When a program references a memory location, it is likely to reference that same memory location again soon, which is known as **temporal locality**. Similarly, there is **spatial locality**, in which a memory location that is near a recently referenced location is more likely to be referenced than a memory location that is farther away. Temporal locality arises because programs spend much of their time in iteration or in recursion, and thus the same section of code is visited a disproportionately large number of times. Spatial locality arises because data tends to be stored in contiguous locations. Although 10% of the code accounts for the bulk of memory references, accesses within the 10% tend to be clustered. Thus, for a given interval of time, most of memory accesses come from an even smaller set of locations than 10% of a program's size.

Memory access is generally slow when compared with the speed of the central processing unit (CPU), and so the memory poses a significant bottleneck in computer performance. Since most memory references come from a small set of locations, the locality principle can be exploited in order to improve performance. A small but fast **cache memory**, in which the contents of the most commonly accessed locations are maintained, can be placed between the main memory and the CPU. When a program executes, the cache memory is searched first, and the referenced word is accessed in the cache if the word is present. If the

referenced word is not in the cache, then a free location is created in the cache and the referenced word is brought into the cache from the main memory. The word is then accessed in the cache. Although this process takes longer than accessing main memory directly, the overall performance can be improved if a high proportion of memory accesses are satisfied by the cache.

Modern memory systems may have several levels of cache, referred to as Level 1 (L1), Level 2 (L2), and even, in some cases, Level 3 (L3). In most instances the L1 cache is implemented right on the CPU chip. Both the Intel Pentium and the IBM-Motorola PowerPC G3 processors have 32 Kbytes of L1 cache on the CPU chip.

A cache memory is faster than main memory for a number of reasons. Faster electronics can be used, which also results in a greater expense in terms of money, size, and power requirements. Since the cache is small, this increase in cost is relatively small. A cache memory has fewer locations than a main memory, and as a result it has a shallow decoding tree, which reduces the access time. The cache is placed both physically closer and logically closer to the CPU than the main memory, and this placement avoids communication delays over a **shared bus**.

A typical situation is shown in Figure 7-12. A simple computer without a cache

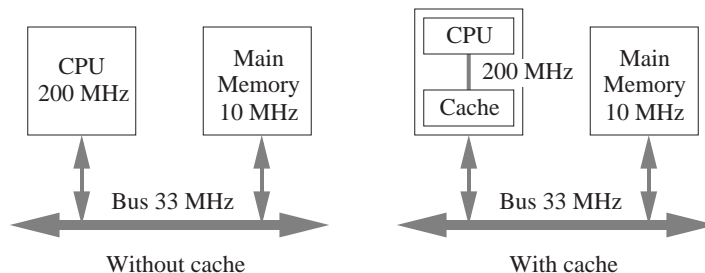


Figure 7-12 Placement of cache in a computer system.

memory is shown in the left side of the figure. This cache-less computer contains a CPU that has a clock speed of 200 MHz, but communicates over a 33 MHz bus to a main memory that supports a lower clock speed of 10 MHz. A few bus cycles are normally needed to synchronize the CPU with the bus, and thus the difference in speed between main memory and the CPU can be as large as a factor of ten or more. A cache memory can be positioned closer to the CPU as shown in the right side of Figure 7-12, so that the CPU sees fast accesses over a 200 MHz direct path to the cache.

7.6.1 ASSOCIATIVE MAPPED CACHE

A number of hardware schemes have been developed for translating main memory addresses to cache memory addresses. The user does not need to know about the address translation, which has the advantage that cache memory enhancements can be introduced into a computer without a corresponding need for modifying application software.

The choice of cache mapping scheme affects cost and performance, and there is no single best method that is appropriate for all situations. In this section, an **associative** mapping scheme is studied. Figure 7-13 shows an associative map-

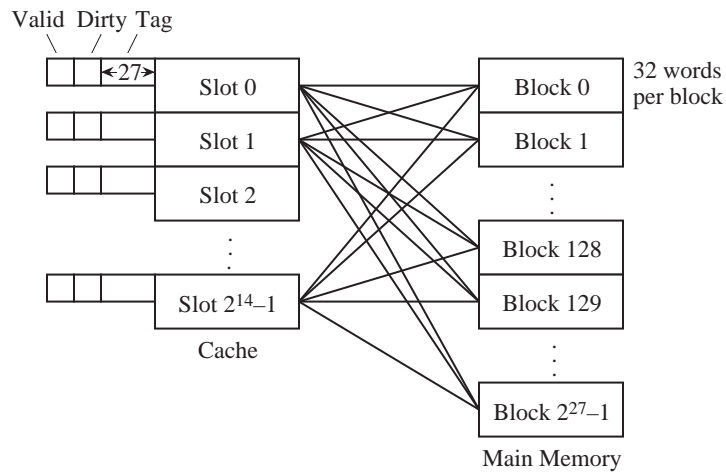


Figure 7-13 An associative mapping scheme for a cache memory.

ping scheme for a 2^{32} word memory space that is divided into 2^{27} **blocks** of $2^5 = 32$ words per block. The main memory is not physically partitioned in this way, but this is the view of main memory that the cache sees. Cache blocks, or **cache lines**, as they are also known, typically range in size from 8 to 64 bytes. Data is moved in and out of the cache a line at a time using memory interleaving (discussed earlier).

The cache for this example consists of 2^{14} **slots** into which main memory blocks are placed. There are more main memory blocks than there are cache slots, and any one of the 2^{27} main memory blocks can be mapped into each cache slot (with only one block placed in a slot at a time). To keep track of which one of the 2^{27} possible blocks is in each slot, a 27-bit **tag** field is added to each slot which holds an identifier in the range from 0 to $2^{27} - 1$. The tag field is the most signif-

icant 27 bits of the 32-bit memory address presented to the cache. All the tags are stored in a special tag memory where they can be searched in parallel. Whenever a new block is stored in the cache, its tag is stored in the corresponding tag memory location.

When a program is first loaded into main memory, the cache is cleared, and so while a program is executing, a **valid** bit is needed to indicate whether or not the slot holds a block that belongs to the program being executed. There is also a **dirty** bit that keeps track of whether or not a block has been modified while it is in the cache. A slot that is modified must be written back to the main memory before the slot is reused for another block.

A referenced location that is found in the cache results in a **hit**, otherwise, the result is a **miss**. When a program is initially loaded into memory, the valid bits are all set to 0. The first instruction that is executed in the program will therefore cause a miss, since none of the program is in the cache at this point. The block that causes the miss is located in the main memory and is loaded into the cache.

In an associative mapped cache, each main memory block can be mapped to any slot. The mapping from main memory blocks to cache slots is performed by partitioning an address into fields for the tag and the word (also known as the “byte” field) as shown below:

Tag	Word
27 bits	5 bits

When a reference is made to a main memory address, the cache hardware intercepts the reference and searches the cache tag memory to see if the requested block is in the cache. For each slot, if the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. All of the tags are searched in parallel, using an **associative memory** (which is something different than an associative mapping scheme. See Section 7.8.3 for more on associative memories.) If any tag in the cache tag memory matches the tag field of the memory reference, then the word is taken from the position in the slot specified by the word field. If the referenced word is not found in the cache, then the main memory block that contains the word is brought into the cache and the referenced word is then taken from the cache. The tag, valid, and dirty fields are updated, and the program resumes execution.

Consider how an access to memory location $(A035F014)_{16}$ is mapped to the

cache. The leftmost 27 bits of the address form the tag field, and the remaining five bits form the word field as shown below:

Tag	Word
1 0 1 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 0 0 0 0 0 0 0	1 0 1 0 0

If the addressed word is in the cache, it will be found in word $(14)_{16}$ of a slot that has a tag of $(501AF80)_{16}$, which is made up of the 27 most significant bits of the address. If the addressed word is not in the cache, then the block corresponding to the tag field $(501AF80)_{16}$ will be brought into an available slot in the cache from the main memory, and the memory reference that caused the “cache miss” will then be satisfied from the cache.

Although this mapping scheme is powerful enough to satisfy a wide range of memory access situations, there are two implementation problems that limit performance. First, the process of deciding which slot should be freed when a new block is brought into the cache can be complex. This process requires a significant amount of hardware and introduces delays in memory accesses. A second problem is that when the cache is searched, the tag field of the referenced address must be compared with all 2^{14} tag fields in the cache. (Alternative methods that limit the number of comparisons are described in Sections 7.6.2 and 7.6.3.)

Replacement Policies in Associative Mapped Caches

When a new block needs to be placed in an associative mapped cache, an available slot must be identified. If there are unused slots, such as when a program begins execution, then the first slot with a valid bit of 0 can simply be used. When all of the valid bits for all cache slots are 1, however, then one of the active slots must be freed for the new block. Four replacement policies that are commonly used are: **least recently used** (LRU), **first-in first-out** (FIFO), **least frequently used** (LFU), and **random**. A fifth policy that is used for analysis purposes only, is **optimal**.

For the LRU policy, a time stamp is added to each slot, which is updated when any slot is accessed. When a slot must be freed for a new block, the contents of the least recently used slot, as identified by the age of the corresponding time stamp, are discarded and the new block is written to that slot. The LFU policy works similarly, except that only one slot is updated at a time by incrementing a frequency counter that is attached to each slot. When a slot is needed for a new block, the least frequently used slot is freed. The FIFO policy replaces slots in

round-robin fashion, one after the next in the order of their physical locations in the cache. The random replacement policy simply chooses a slot at random.

The optimal replacement policy is not practical, but is used for comparison purposes to determine how effective other replacement policies are to the best possible. That is, the optimal replacement policy is determined only after a program has already executed, and so it is of little help to a running program.

Studies have shown that the LFU policy is only slightly better than the random policy. The LRU policy can be implemented efficiently, and is sometimes preferred over the others for that reason. A simple implementation of the LRU policy is covered in Section 7.6.7.

Advantages and Disadvantages of the Associative Mapped Cache

The associative mapped cache has the advantage that any main memory block can be placed into any cache slot. This means that regardless of how irregular the data and program references are, if a slot is available for the block, it can be stored in the cache. This results in considerable hardware overhead needed for cache bookkeeping. Each slot must have a 27-bit tag that identifies its location in main memory, and each tag must be searched in parallel. This means that in the example above the tag memory must be 27×2^{14} bits in size, and as described above, there must be a mechanism for searching the tag memory in parallel. Memories that can be searched for their contents, in parallel, are referred to as **associative**, or **content-addressable** memories. We will discuss this kind of memory later in the chapter.

By restricting where each main memory block can be placed in the cache, we can eliminate the need for an associative memory. This kind of cache is referred to as a **direct mapped cache**, which is discussed in the next section.

7.6.2 DIRECT MAPPED CACHE

Figure 7-14 shows a direct mapping scheme for a 2^{32} word memory. As before, the memory is divided into 2^{27} blocks of $2^5 = 32$ words per block, and the cache consists of 2^{14} slots. There are more main memory blocks than there are cache slots, and a total of $2^{27}/2^{14} = 2^{13}$ main memory blocks can be mapped onto each cache slot. In order to keep track of which of the 2^{13} possible blocks is in each slot, a 13-bit tag field is added to each slot which holds an identifier in the range

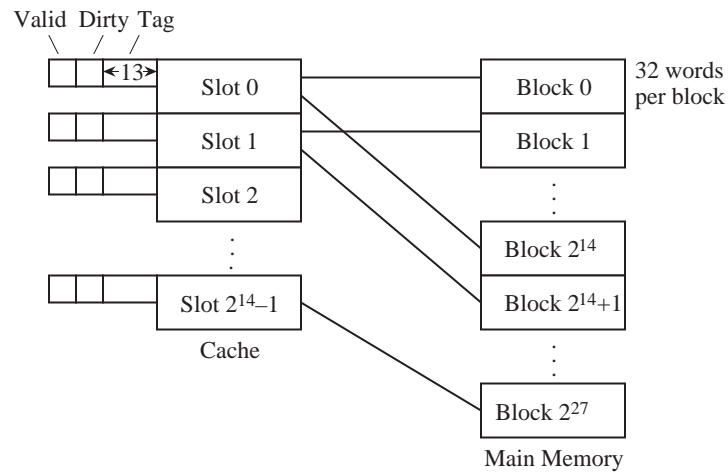


Figure 7-14 A direct mapping scheme for cache memory.

from 0 to $2^{13} - 1$.

This scheme is called “direct mapping” because each cache slot corresponds to an explicit set of main memory blocks. For a direct mapped cache, each main memory block can be mapped to only one slot, but each slot can receive more than one block. The mapping from main memory blocks to cache slots is performed by partitioning an address into fields for the tag, the slot, and the word as shown below:

Tag	Slot	Word
13 bits	14 bits	5 bits

The 32-bit main memory address is partitioned into a 13-bit tag field, followed by a 14-bit slot field, followed by a five-bit word field. When a reference is made to a main memory address, the slot field identifies in which of the 2^{14} slots the block will be found if it is in the cache. If the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. If the tag fields are the same, then the word is taken from the position in the slot specified by the word field. If the valid bit is 1 but the tag fields are not the same, then the slot is written back to main memory if the dirty bit is set, and the corresponding main memory block is then read into the slot. For a program that has just started execution, the valid bit will be 0, and so the block is simply written to the slot. The valid bit for the block is then set to 1, and the program resumes execution.

Consider how an access to memory location $(A035F014)_{16}$ is mapped to the cache. The bit pattern is partitioned according to the word format shown above. The leftmost 13 bits form the tag field, the next 14 bits form the slot field, and the remaining five bits form the word field as shown below:

Tag	Slot	Word
1 0 1 0 0 0 0 0 0 1 1 0	1 0 1 1 1 1 1 0 0 0 0 0 0 0	1 0 1 0 0

If the addressed word is in the cache, it will be found in word $(14)_{16}$ of slot $(2F80)_{16}$, which will have a tag of $(1406)_{16}$.

Advantages and Disadvantages of the Direct Mapped Cache

The direct mapped cache is a relatively simple scheme to implement. The tag memory in the example above is only 13×2^{14} bits in size, less than half of the associative mapped cache. Furthermore, there is no need for an associative search, since the slot field of the main memory address from the CPU is used to “direct” the comparison to the single slot where the block will be if it is indeed in the cache.

This simplicity comes at a cost. Consider what happens when a program references locations that are 2^{19} words apart, which is the size of the cache. This pattern can arise naturally if a matrix is stored in memory by rows and is accessed by columns. Every memory reference will result in a miss, which will cause an entire block to be read into the cache even though only a single word is used. Worse still, only a small fraction of the available cache memory will actually be used.

Now it may seem that any programmer who writes a program this way deserves the resulting poor performance, but in fact, fast matrix calculations use power-of-two dimensions (which allows shift operations to replace costly multiplications and divisions for array indexing), and so the worst-case scenario of accessing memory locations that are 2^{19} addresses apart is not all that unlikely. To avoid this situation without paying the high implementation price of a fully associative cache memory, the **set associative mapping** scheme can be used, which combines aspects of both direct mapping and associative mapping. Set associative mapping, which is also known as **set-direct mapping**, is described in the next section.

7.6.3 SET ASSOCIATIVE MAPPED CACHE

The set associative mapping scheme combines the simplicity of direct mapping with the flexibility of associative mapping. Set associative mapping is more practical than fully associative mapping because the associative portion is limited to just a few slots that make up a set, as illustrated in Figure 7-15. For this example,

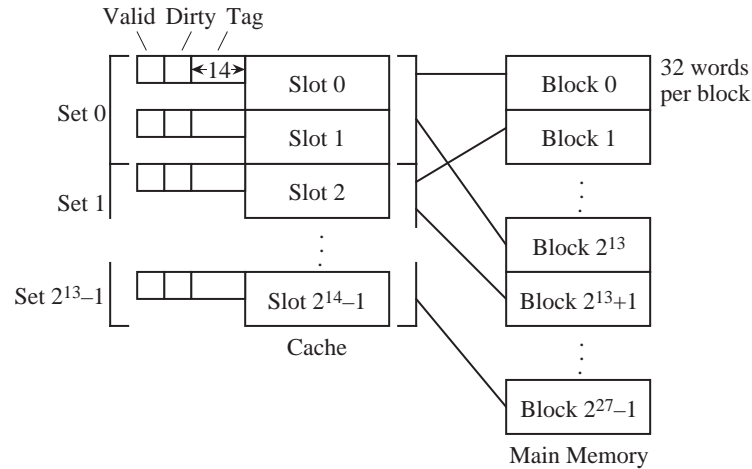


Figure 7-15 A set associative mapping scheme for a cache memory.

two blocks make up a set, and so it is a **two-way** set associative cache. If there are four blocks per set, then it is a four-way set associative cache.

Since there are 2^{14} slots in the cache, there are $2^{14}/2 = 2^{13}$ sets. When an address is mapped to a set, the direct mapping scheme is used, and then associative mapping is used within a set. The format for an address has 13 bits in the set field, which identifies the set in which the addressed word will be found if it is in the cache. There are five bits for the word field as before and there is a 14-bit tag field that together make up the remaining 32 bits of the address as shown below:

Tag	Set	Word
14 bits	13 bits	5 bits

As an example of how the set associative cache views a main memory address, consider again the address $(A035F014)_{16}$. The leftmost 14 bits form the tag field, followed by 13 bits for the set field, followed by five bits for the word field

as shown below:

Tag	Set	Word
1 0 1 0 0 0 0 0 0 0 1 1 0 1	0 1 1 1 1 1 0 0 0 0 0 0 0	1 0 1 0 0

As before, the partitioning of the address field is known only to the cache, and the rest of the computer is oblivious to any address translation.

Advantages and Disadvantages of the Set Associative Mapped Cache

In the example above, the tag memory increases only slightly from the direct mapping example, to 13×2^{14} bits, and only two tags need to be searched for each memory reference. The set associative cache is almost universally used in today's microprocessors.

7.6.4 CACHE PERFORMANCE

Notice that we can readily replace the cache direct mapping hardware with associative or set associative mapping hardware, without making any other changes to the computer or the software. Only the runtime performance will change between methods.

Runtime performance is the purpose behind using a cache memory, and there are a number of issues that need to be addressed as to what triggers a word or block to be moved between the cache and the main memory. Cache read and write policies are summarized in Figure 7-16. The policies depend upon whether or not the requested word is in the cache. If a cache read operation is taking place, and the referenced data is in the cache, then there is a “cache hit” and the referenced data is immediately forwarded to the CPU. When a cache miss occurs, then the entire block that contains the referenced word is read into the cache.

In some cache organizations, the word that causes the miss is immediately forwarded to the CPU as soon as it is read into the cache, rather than waiting for the remainder of the cache slot to be filled, which is known as a **load-through** operation. For a non-interleaved main memory, if the word occurs in the last position of the block, then no performance gain is realized since the entire slot is brought in before load-through can take place. For an interleaved main memory, the order of accesses can be organized so that a load-through operation will always result in a performance gain.

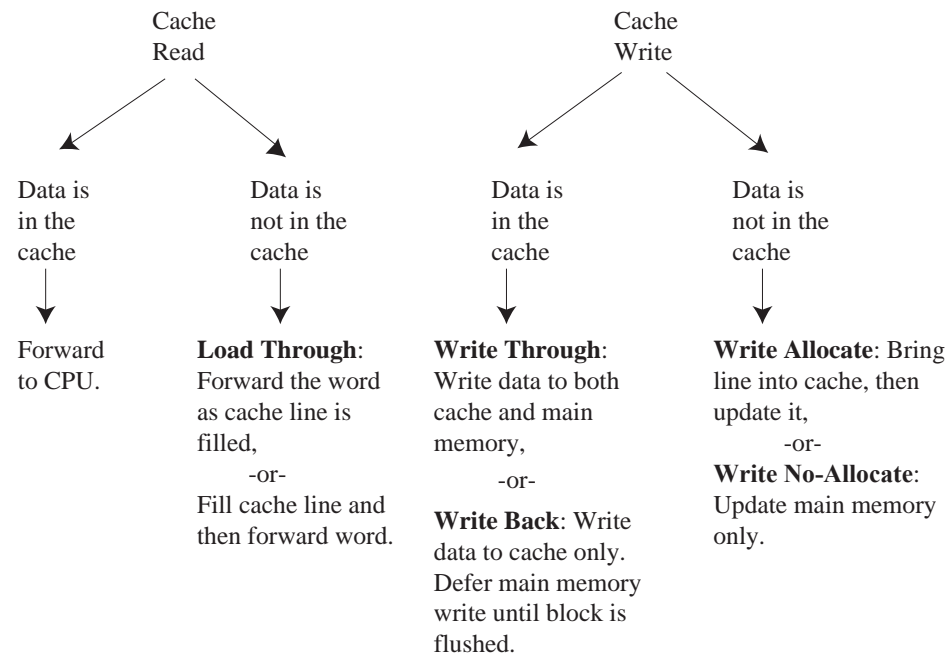


Figure 7-16 Cache read and write policies.

For write operations, if the word is in the cache, then there may be *two* copies of the word, one in the cache, and one in main memory. If both are updated simultaneously, this is referred to as **write-through**. If the write is deferred until the cache line is flushed from the cache, this is referred to as **write back**. Even if the data item is not in the cache when the write occurs, there is the choice of bringing the block containing the word into the cache and then updating it, known as **write-allocate**, or to update it in main memory without involving the cache, known as **write-no-allocate**.

Some computers have separate caches for instructions and data, which is a variation of a configuration known as the **Harvard architecture** (also known as a **split cache**), in which instructions and data are stored in separate sections of memory. Since instruction slots can never be dirty (unless we write self-modifying code, which is rare these days), an instruction cache is simpler than a data cache. In support of this configuration, observations have shown that most of the memory traffic moves away from main memory rather than toward it. Statistically, there is only one write to memory for every four read operations from memory. One reason for this is that instructions in an executing program are only read from the main memory, and are never written to the memory except by

the system loader. Another reason is that operations on data typically involve reading two operands and storing a single result, which means there are two read operations for every write operation. A cache that only handles reads, while sending writes directly to main memory can thus also be effective, although not necessarily as effective as a fully functional cache.

As to which cache read and write policies are best, there is no simple answer. The organization of a cache is optimized for each computer architecture and the mix of programs that the computer executes. Cache organization and cache sizes are normally determined by the results of simulation runs that expose the nature of memory traffic.

7.6.5 HIT RATIOS AND EFFECTIVE ACCESS TIMES

Two measures that characterize the performance of a cache memory are the **hit ratio** and the **effective access time**. The hit ratio is computed by dividing the number of times referenced words are found in the cache by the total number of memory references. The effective access time is computed by dividing the total time spent accessing memory (summing the main memory and cache access times) by the total number of memory references. The corresponding equations are given below:

$$\text{Hit ratio} = \frac{\text{No. times referenced words are in cache}}{\text{Total number of memory accesses}}$$

$$\text{Eff. access time} = \frac{(\# \text{ hits})(\text{Time per hit}) + (\# \text{ misses})(\text{Time per miss})}{\text{Total number of memory access}}$$

Consider computing the hit ratio and the effective access time for a program running on a computer that has a direct mapped cache with four 16-word slots. The layout of the cache and the main memory are shown in Figure 7-17. The cache access time is 80 ns, and the time for transferring a main memory block to the cache is 2500 ns. Assume that load-through is used in this architecture and that the cache is initially empty. A sample program executes from memory locations 48 – 95, and then loops 10 times from 15 – 31 before halting.

We record the events as the program executes as shown in Figure 7-18. Since the memory is initially empty, the first instruction that executes causes a miss. A miss thus occurs at location 48, which causes main memory block #3 to be read into cache slot #3. This first memory access takes 2500 ns to complete. Load-through is used for this example, and so the word that causes the miss at location 48 is passed directly to the CPU while the rest of the block is loaded into the cache

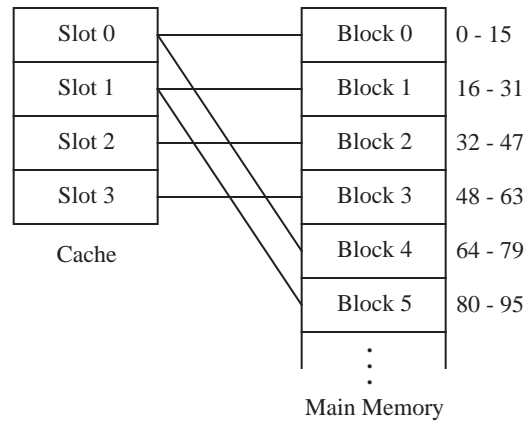


Figure 7-17 An example of a direct mapped cache memory.

Event	Location	Time	Comment
1 miss	48	2500ns	Memory block 3 to cache slot 3
15 hits	49-63	80ns×15=1200ns	
1 miss	64	2500ns	Memory block 4 to cache slot 0
15 hits	65-79	80ns×15=1200ns	
1 miss	80	2500ns	Memory block 5 to cache slot 1
15 hits	81-95	80ns×15=1200ns	
1 miss	15	2500ns	Memory block 0 to cache slot 0
1 miss	16	2500ns	Memory block 1 to cache slot 1
15 hits	17-31	80ns×15=1200ns	
9 hits	15	80ns×9=720ns	Last nine iterations of loop
144 hits	16-31	80ns×144=12,240ns	Last nine iterations of loop
Total hits = 213 Total misses = 5			

Figure 7-18 A table of events for a program executing on an architecture with a small direct mapped cache memory.

slot. The next event consists of 15 hits for locations 49 through 63. The events that follow are recorded in a similar manner, and the result is a total of 213 hits and five misses. The total number of accesses is $213 + 5 = 218$. The hit ratio and effective access time are computed as shown below:

$$\text{Hit ratio} = \frac{213}{218} = 97.7\%$$

$$\text{EffectiveAccessTime} = \frac{(213)(80\text{ns}) + (5)(2500\text{ns})}{218} = 136\text{ns}$$

Although the hit ratio is 97.6%, the effective access time for this example is

almost 75% longer than the cache access time. This is due to the large amount of time spent in accessing a block from main memory.

7.6.6 MULTILEVEL CACHES

As the sizes of silicon ICs have increased, and the packing density of components on ICs has increased, it has become possible to include cache memory on the same IC as the processor. Since the on-chip processing speed is faster than the speed of communication between chips, an on-chip cache can be faster than an off-chip cache. Current technology is not dense enough to allow the entire cache to be placed on the same chip as the processor, however. For this reason, **multi-level caches** have been developed, in which the fastest level of the cache, L1, is on the same chip as the processor, and the remaining cache is placed off of the processor chip. Data and instruction caches are separately maintained in the L1 cache. The L2 cache is **unified**, which means that the same cache holds both data and instructions.

In order to compute the hit ratio and effective access time for a multilevel cache, the hits and misses must be recorded among both caches. Equations that represent the overall hit ratio and the overall effective access time for a two-level cache are shown below. H_1 is the hit ratio for the on-chip cache, H_2 is the hit ratio for the off-chip cache, and T_{EFF} is the overall effective access time. The method can be extended to any number of levels.

$$H_1 = \frac{\text{No. times accessed word is in on-chip cache}}{\text{Total number of memory accesses}}$$

$$H_2 = \frac{\text{No. times accessed word is in off-chip cache}}{\text{No. times accessed word is not in on-chip cache}}$$

$$T_{EFF} = \frac{(\text{No. on-chip cache hits})(\text{On-chip cache hit time}) + (\text{No. off-chip cache hits})(\text{Off-chip cache hit time}) + (\text{No. off-chip cache misses})(\text{Off-chip cache miss time})}{\text{Total number of memory accesses}}$$

7.6.7 CACHE MANAGEMENT

Management of a cache memory presents a complex problem to the system programmer. If a given memory location represents an I/O port, as it may in memory-mapped systems, then it probably should not appear in the cache at all. If it is cached, the value in the I/O port may change, and this change will not be reflected in the value of the data stored in the cache. This is known as “stale”

data: the copy that is in the cache is “stale” compared with the value in main memory. Likewise, in **shared-memory multiprocessor** environments (see Chapter 10), where more than one processor may access the same main memory, either the cached value or the value in main memory may become stale due to the activity of one or more of the CPUs. At a minimum, the cache in a multiprocessor environment should implement a write-through policy for those cache lines which map to shared memory locations.

For these reasons, and others, most modern processor architectures allow the system programmer to have some measure of control over the cache. For example, the Motorola PPC 601 processor’s cache, which normally enforces a write-back policy, can be set to a write-through policy on a per-line basis. Other instructions allow individual lines to be specified as noncacheable, or to be marked as invalid, loaded, or flushed.

Internal to the cache, replacement policies (for associative and set-associative caches) need to be implemented efficiently. An efficient implementation of the LRU replacement policy can be achieved with the **Neat Little LRU Algorithm** (origin unknown). Continuing with the cache example used in Section 7.6.5, we construct a matrix in which there is a row and a column for every slot in the cache, as shown in Figure 7-19. Initially, all of the cells are set to 0. Each time

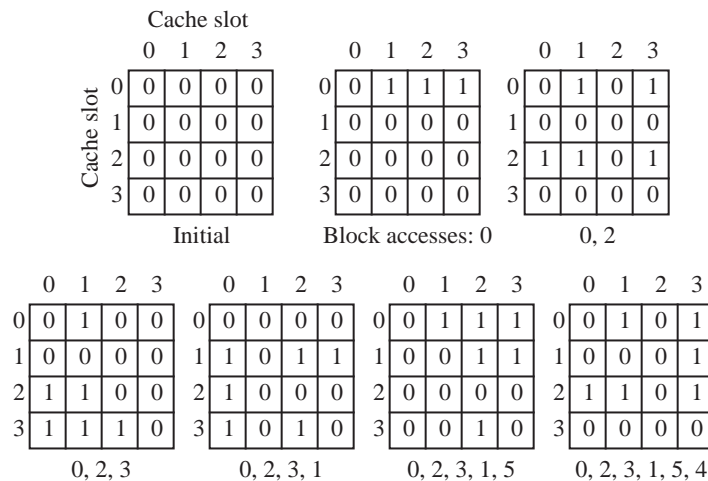


Figure 7-19 A sequence is shown for the Neat Little LRU Algorithm for a cache with four slots. Main memory blocks are accessed in the sequence: 0, 2, 3, 1, 5, 4.

that a slot is accessed, 1's are written into each cell in the row of the table that

corresponds to that slot. 0's are then written into each cell in the column that corresponds to that slot. Whenever a slot is needed, the row that contains all 0's is the oldest and is used next. At the beginning of the process, more than one row will contain all 0's, and so a tie-breaking mechanism is needed. The first row with all 0's is one method that will work, which we use here.

The example shown in Figure 7-19 shows the configuration of the matrix as blocks are accessed in the order: 0, 2, 3, 1, 5, 4. Initially, the matrix is filled with 0's. After a reference is made to block 0, the row corresponding to block 0 is filled with 1's and the column corresponding to block 0 is filled with 0's. For this example, block 0 happens to be placed in slot 0, but for other situations, block 0 can be placed in any slot. The process continues until all cache slots are in use at the end of the sequence: 0, 2, 3, 1. In order to bring the next block (5) into the cache, a slot must be freed. The row for slot 0 contains 0's, and so it is the least recently used slot. Block 5 is then brought into slot 0. Similarly, when block 4 is brought into the cache, slot 2 is overwritten.

7.7 Virtual Memory

Despite the enormous advancements in creating ever larger memories in smaller areas, computer memory is still like closet space, in the sense that we can never have enough of it. An economical method of extending the apparent size of the main memory is to augment it with disk space, which is one aspect of **virtual memory** that we cover in this section. Disk storage appears near the bottom of the memory hierarchy, with a lower cost per bit than main memory, and so it is reasonable to use disk storage to hold the portions of a program or data sets that do not entirely fit into the main memory. In a different aspect of virtual memory, complex address mapping schemes are supported, which give greater flexibility in how the memory is used. We explore these aspects of virtual memory below.

7.7.1 OVERLAYS

An early approach of using disk storage to augment the main memory made use of **overlays**, in which an executing program overwrites its own code with other code as needed. In this scenario, the programmer has the responsibility of managing memory usage. Figure 7-20 shows an example in which a program contains a main routine and three subroutines *A*, *B*, and *C*. The physical memory is smaller than the size of the program, but is larger than any single routine. A strategy for managing memory using overlays is to modify the program so that it keeps track of which subroutines are in memory, and reads in subroutine code as

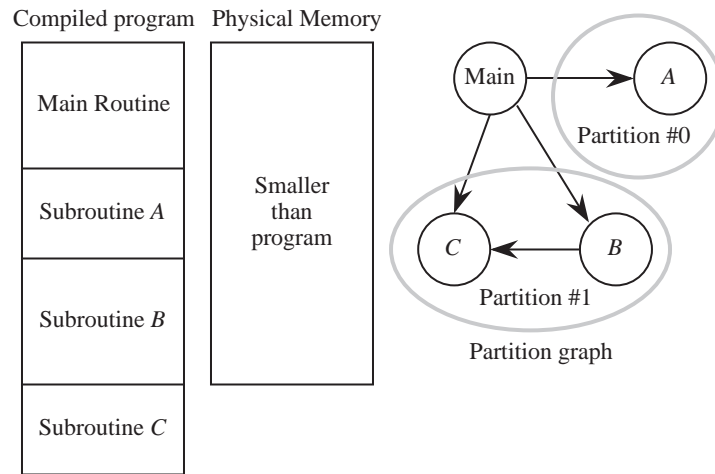


Figure 7-20 A partition graph for a program with a main routine and three subroutines.

needed. Typically, the main routine serves as the **driver** and manages the bulk of the bookkeeping. The driver stays in memory while other routines are brought in and out.

Figure 7-20 shows a **partition graph** that is created for the example program. The partition graph identifies which routines can overlay others based on which subroutines call others. For this example, the main routine is always present, and supervises which subset of subroutines are in memory. Subroutines *B* and *C* are kept in the same partition in this example because *B* calls *C*, but subroutine *A* is in its own partition because only the main routine calls *A*. Partition #0 can thus overlay partition #1, and partition #1 can overlay partition #0.

Although this method will work well in a variety of situations, a cleaner solution might be to let an operating system manage the overlays. When more than one program is loaded into memory, however, then the routines that manage the overlays cannot operate without interacting with the operating system in order to find out which portions of memory are available. This scenario introduces a great deal of complexity into managing the overlay process since there is a heavy interaction between the operating system and each program. An alternative method that can be managed by the operating system alone is called **paging**, which is described in the next section.

7.7.2 PAGING

Paging is a form of automatic overlaying that is managed by the operating system. The address space is partitioned into equal sized blocks, called **pages**. Pages are normally an integral power of two in size such as $2^{10} = 1024$ bytes. Paging makes the physical memory appear larger than it truly is by mapping the physical memory address space to some portion of the virtual memory address space, which is normally stored on a disk. An illustration of a virtual memory mapping scheme is shown in Figure 7-21. Eight virtual pages are mapped to four physical

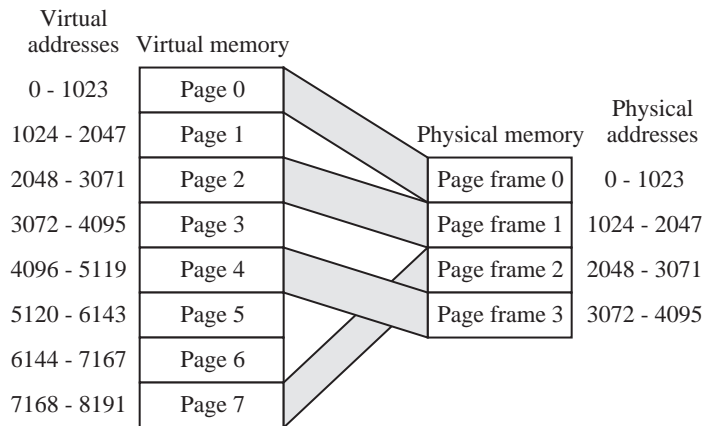


Figure 7-21 A mapping between a virtual and a physical memory.

page frames.

An implementation of virtual memory must handle references that are made outside of the portion of virtual space that is mapped to physical space. The following sequence of events is typical when a referenced virtual location is not in physical memory, which is referred to as a **page fault**:

- 1) A page frame is identified to be overwritten. The contents of the page frame are written to secondary memory if changes were made to it, so that the changes are recorded before the page frame is overwritten.
- 2) The virtual page that we want to access is located in secondary memory and is written into physical memory, in the page frame located in (1) above.
- 3) The page table (see below) is updated to map the new section of virtual memory onto the physical memory.

4) Execution continues.

For the virtual memory shown in Figure 7-21, there are $2^{13} = 8192$ virtual locations and so an executing program must generate 13-bit addresses, which are interpreted as a 3-bit page number and a 10-bit offset within the page. Given the 3-bit page number, we need to find out where the page is: it is either in one of the four page frames, or it is in secondary memory. In order to keep track of which pages are in physical memory, a **page table** is maintained, as illustrated in Figure 7-22, which corresponds to the mapping shown in Figure 7-21.

	Present bit		Page frame
Page #		Disk address	
0	1	01001011100	00
1	0	11101110010	xx
2	1	10110010111	01
3	0	00001001111	xx
4	1	01011100101	11
5	0	10100111001	xx
6	0	00110101100	xx
7	1	01010001011	10

Present bit:
0: Page is not in
physical memory
1: Page is in physical
memory

Figure 7-22 A page table for a virtual memory.

The page table has as many entries as there are virtual pages. The present bit indicates whether or not the corresponding page is in physical memory. The disk address field is a pointer to the location that the corresponding page can be found on a disk unit. The operating system normally manages the disk accesses, and so the page table only needs to maintain the disk addresses that the operating system assigns to blocks when the system starts up. The disk addresses normally do not change during the course of computation. The page frame field indicates which physical page frame holds a virtual page, if the page is in physical memory. For pages that are not in physical memory, the page frame fields are invalid, and so they are marked with “xx” in Figure 7-22.

In order to translate a virtual address to a physical address, we take two page frame bits from the page table and append them to the left of the 10-bit offset, which produces the physical address for the referenced word. Consider the situation shown in Figure 7-23, in which a reference is made to virtual address 1001101000101. The three leftmost bits of the virtual address (100) identify the

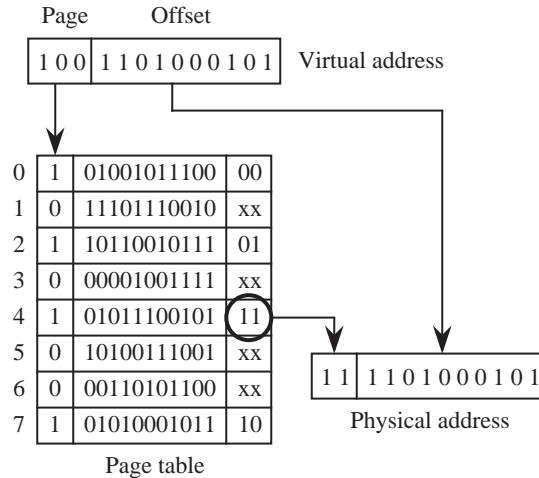


Figure 7-23 A virtual address is translated into a physical address.

page. The bit pattern that appears in the page frame field (11) is appended to the left of the 10-bit offset (1101000101), and the resulting address (111101000101) indicates which physical memory address holds the referenced word.

It may take a relatively long period of time for a program to be loaded into memory. The entire program may never be executed, and so the time required to load the program from a disk into the memory can be reduced by loading only the portion of the program that is needed for a given interval of time. The **demand paging** scheme does not load a page into memory until there is a page fault. After a program has been running for a while, only the pages being used will be in physical memory (this is referred to as the **working set**), so demand paging does not have a significant impact on long running programs.

Consider again the memory mapping shown in Figure 7-21. The size of the virtual address space is 2^{13} words, and the physical address space is 2^{12} words. There are eight pages that each contain 2^{10} words. Assume that the memory is initially empty, and that demand paging is used for a program that executes from memory locations 1030 to 5300. The execution sequence will make accesses to pages 1, 2, 3, 4, and 5, in that order. The page replacement policy is FIFO. Figure 7-24 shows the configuration of the page table as execution proceeds. The first access to memory will cause a page fault on virtual address 1030, which is in page #1. The page is brought into physical memory, and the valid bit and page frame field are updated in the page table. Execution continues, until page #5

0	0	01001011100	xx	0	0	01001011100	xx
1	1	11101110010	00	1	1	11101110010	00
2	0	10110010111	xx	2	1	10110010111	01
3	0	00001001111	xx	3	0	00001001111	xx
4	0	01011100101	xx	4	0	01011100101	xx
5	0	10100111001	xx	5	0	10100111001	xx
6	0	00110101100	xx	6	0	00110101100	xx
7	0	01010001011	xx	7	0	01010001011	xx

After fault on page #1

0	0	01001011100	xx	0	0	01001011100	xx
1	1	11101110010	00	1	0	11101110010	xx
2	1	10110010111	01	2	1	10110010111	01
3	1	00001001111	10	3	1	00001001111	10
4	0	01011100101	xx	4	1	01011100101	11
5	0	10100111001	xx	5	1	10100111001	00
6	0	00110101100	xx	6	0	00110101100	xx
7	0	01010001011	xx	7	0	01010001011	xx

After fault on page #3

0	0	01001011100	xx	0	0	01001011100	xx
1	1	11101110010	00	1	0	11101110010	xx
2	1	10110010111	01	2	1	10110010111	01
3	1	00001001111	10	3	1	00001001111	10
4	0	01011100101	xx	4	1	01011100101	11
5	0	10100111001	xx	5	1	10100111001	00
6	0	00110101100	xx	6	0	00110101100	xx
7	0	01010001011	xx	7	0	01010001011	xx

Final

Figure 7-24 The configuration of a page table changes as a program executes. Initially, the page table is empty. In the final configuration, four pages are in physical memory.

must be brought in, which forces out page #1 due to the FIFO page replacement policy. The final configuration of the page table in Figure 7-24 is shown after location 5300 is accessed.

7.7.3 SEGMENTATION

Virtual memory as we have discussed it up to this point is one-dimensional in the sense that addresses grow either up or down. **Segmentation** divides the address space into **segments**, which may be of arbitrary size. Each segment is its own one-dimensional address space. This allows tables, stacks, and other data structures to be maintained as logical entities that grow without bumping into each other. Segmentation allows for **protection**, so that a segment may be specified as “read only” to prevent changes, or “execute only” to prevent unauthorized copying. This also protects users from trying to write data into instruction areas.

When segmentation is used with virtual memory, the size of each segment’s address space can be very large, and so the physical memory devoted to each segment is not committed until needed.

Figure 7-25 illustrates a segmented memory. The executable code for a word pro-

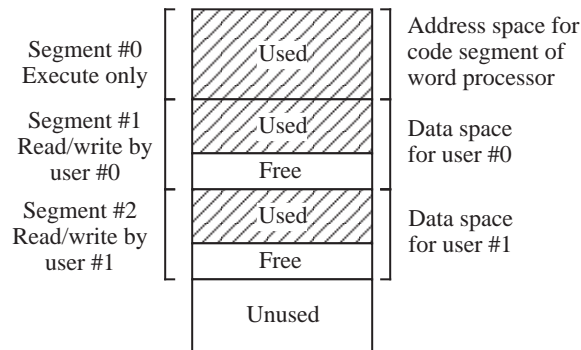


Figure 7-25 A segmented memory allows two users to share the same word processor.

cessing program is loaded into Segment #0. This segment is marked as “execute only” and is thus protected from writing. Segment #1 is used for the data space for user #0, and is marked as “read/write” for user #0, so that no other user can have access to this area. Segment #2 is used for the data space for user #1, and is marked as “read/write” for user #1. The same word processor can be used by both user #0 and user #1, in which case the code in segment #0 is shared, but each user has a separate data segment.

Segmentation is not the same thing as paging. With paging, the user does not see the automatic overlaying. With segmentation, the user is aware of where segment boundaries are. The operating system manages protection and mapping, and so an ordinary user does not normally need to deal with bookkeeping, but a more sophisticated user such as a computer programmer may see the segmentation frequently when array pointers are pushed past segment boundaries in errant programs.

In order to specify an address in a segmented memory, the user’s program must specify a segment number and an address within the segment. The operating system then translates the user’s segmented address to a physical address.

7.7.4 FRAGMENTATION

When a computer is “booted up,” it goes through an initialization sequence that loads the operating system into memory. A portion of the address space may be reserved for I/O devices, and the remainder of the address space is then available for use by the operating system. This remaining portion of the address space may

be only partially filled with physical memory: the rest comprises a “Dead Zone” which must never be accessed since there is no hardware that responds to the Dead Zone addresses.

Figure 7-26a shows the state of a memory just after the initialization sequence.

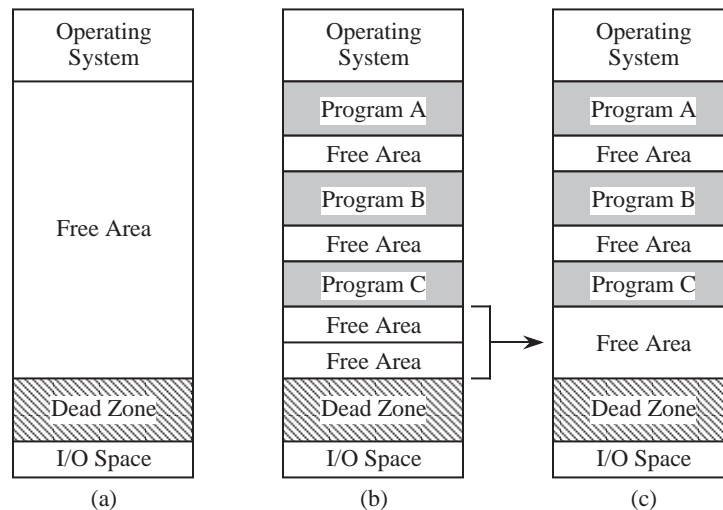


Figure 7-26 (a) Free area of memory after initialization; (b) after fragmentation; (c) after coalescing.

The “Free Area” is a section of memory that is available to the operating system for loading and executing programs. During the course of operation, programs of various sizes will be loaded into memory and executed. When a program finishes execution, the memory space that is assigned to that program is released to the operating system. As programs are loaded and executed, the Free Area becomes subdivided into a collection of small areas, none of which may be large enough to hold a program that would fit unless some or all of the free areas are combined into a single large area. This is a problem known as **fragmentation**, and is encountered with segmentation, because the segments must ultimately be mapped within a single linear address space.

Figure 7-26b illustrates the fragmentation problem. When the operating system needs to find a free area that is large enough to hold a program, it will rarely find an exact match. The free area will generally be larger than the program, which has the effect of subdividing the free areas more finely as programs are mismatched with free areas. One method of assigning programs to free areas is called **first fit**, in which the free areas are scanned until a large enough area is found that will satisfy the program. Another method is called **best fit**, in which the free

area is used that wastes the least amount of space. While best fit makes better use of memory than first fit, it requires more time because all of the free areas must be scanned.

Regardless of which algorithm is used, the process of assigning programs or data to free areas tends to produce smaller free areas (Knuth, 1974). This makes it more difficult to find a single contiguous free area that is large enough to satisfy the needs of the operating system. An approach that helps to solve this problem coalesces adjacent free areas into a single larger free area. In Figure 7-26b, the two adjacent free areas are combined into a single free area, as illustrated in Figure 7-26c.

7.7.5 VIRTUAL MEMORY VS. CACHE MEMORY

Virtual memory is divided into pages, which are relatively large when compared with cache memory blocks, which tend to be only a few words in size. Copies of the most heavily used blocks are kept in cache memory as well as in main memory, and also in the virtual memory image that is stored on a hard disk. When a memory reference is made on a computer that contains both cache and virtual memories, the cache hardware sees the reference first and satisfies the reference if the word is in the cache. If the referenced word is not in the cache, then the block that contains the word is read into the cache from the main memory, and the referenced word is then taken from the cache. If the page that contains the word is not in the main memory, then the page is brought into the main memory from a disk unit, and the block is then loaded into the cache so that the reference can be satisfied.

The use of virtual memory causes some intricate interactions with the cache. For example, since more than one program may be using the cache and the virtual memory, the timing statistics for two runs of a program executing on the same set of data may be different. Also, when a dirty block needs to be written back to main memory, it is possible that the page frame that contains the corresponding virtual page has been overwritten. This causes the page to be loaded back to main memory from secondary memory in order to flush the dirty block from the cache memory to the main memory.

7.7.6 THE TRANSLATION LOOKASIDE BUFFER

The virtual memory mechanism, while being an elegant solution to the problem of accessing large programs and data files, has a significant problem associated

with it. At least two memory references are needed to access a value in memory: One reference is to the page table to find the physical page frame, and another reference is for the actual data value. The **translation lookaside buffer** (TLB) is a solution to this problem.

The TLB is a small associative memory typically placed inside of the CPU that stores the most recent translations from virtual to physical address. The first time a given virtual address is translated into a physical address, this translation is stored in the TLB. Each time the CPU issues a virtual address, the TLB is searched for that virtual address. If the virtual page number exists in the TLB, the TLB returns the physical page number, which can be immediately sent to the main memory (but normally, the cache memory would intercept the reference to main memory and satisfy the reference out of the cache.)

An example TLB is shown in Figure 7-27. The TLB holds 8 entries, for a system

Valid	Virtual page number	Physical page number
1	0 1 0 0 1	1 1 0 0
1	1 0 1 1 1	1 0 0 1
0	- - - - -	- - - -
0	- - - - -	- - - -
1	0 1 1 1 0	0 0 0 0
0	- - - - -	- - - -
1	0 0 1 1 0	0 1 1 1
0	- - - - -	- - - -

Figure 7-27 An example TLB that holds 8 entries for a system with 32 virtual pages and 16 page frames.

that has 32 pages and 16 page frames. The virtual page field is 5 bits wide because there are $2^5 = 32$ pages. Likewise, the physical page field is 4 bits wide because there are $2^4 = 16$ page frames.

TLB misses are handled in much the same way as with other memory misses. Upon a TLB miss the virtual address is applied to the virtual memory system, where it is looked up in the page table in main memory. If it is found in the page table, then the TLB is updated, and the next reference to that page will thus result in a TLB hit.

7.8 Advanced Topics

This section covers two topics that are of practical importance in designing memory systems: tree decoders and content-addressable memories. The former are required in large memories. The latter are required for associative caches, such as a TLB, or in other situations when data must be looked up at high speed based on its value, rather than on the address of where it is stored.

7.8.1 TREE DECODERS

Decoders (see Appendix A) do not scale well to large sizes due to practical limitations on fan-in and fan-out. The decoder circuit shown in Figure 7-28 illustrates

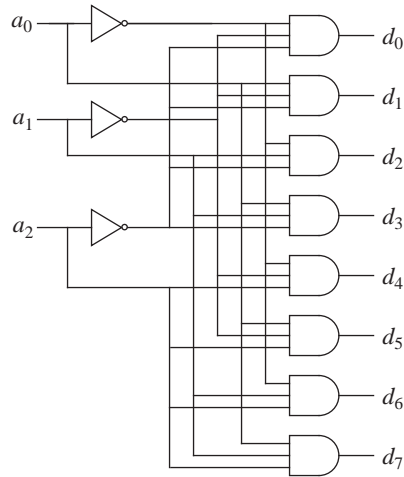


Figure 7-28 A conventional decoder.

the problem. For N address bits, every AND gate has a fan-in of N . Each address line is fanned out to 2^N AND gates. The circuit depth is two gates.

The circuit shown in Figure 7-29a is a **tree decoder**, which reduces the fan-in and fan-out by increasing circuit depth. For this case, each AND gate has a fan-in of F (for this example, $F=2$) and only the address line that is introduced at the deepest level (a_0 here) is fanned out to $2^N/2$ AND gates. The depth has now increased to $\log_F(2^N)$. The large fan-out for the higher order address bits may be a problem, but this can be easily fixed without increasing the circuit depth by adding fan-out buffers in the earlier levels, as shown in Figure 7-29b.

Thus, the depth of a memory decoder tree is $\log_F(2^N)$, the width is 2^N , and the

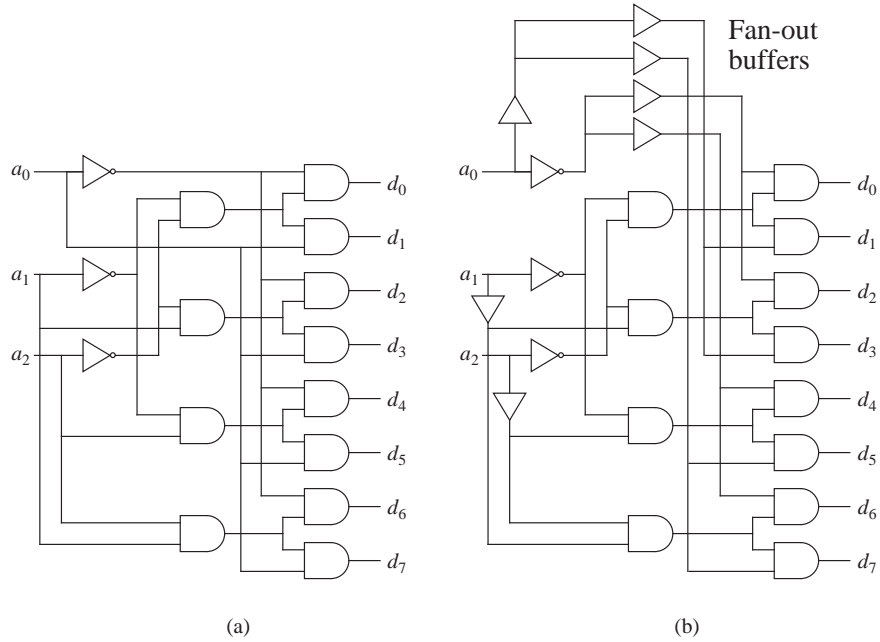


Figure 7-29 (a) A tree decoder; (b) a tree decoder with fan-ins and fan-outs of two.

maximum fan-in and fan-out of the logic gates within the decoder is F

7.8.2 DECODERS FOR LARGE RAMS

For very large RAMs, if the 2-1/2D decoding scheme is not used, tree decoders are employed to keep fanin and fanout to manageable levels. In a conventional RAM an M -bit wide address uniquely identifies one memory location out of a memory space of 2^M locations. In order to access a particular location, an address is presented to the root of a decoder tree containing M levels and 2^M leaves. Starting with the root (the top level of the tree) a decision is made at each i^{th} level of the tree, corresponding to the i^{th} bit of the address. If the i^{th} bit is 0 at the i^{th} level, then the tree is traversed to the left, otherwise the tree is traversed to the right. The target leaf is at level $M - 1$ (counting starts at 0). There is exactly one leaf for each memory address.

The tree structure results in an access time that is logarithmic in the size of the memory. That is, if a RAM contains N words, then the memory can be accessed in $O(\log_F N)$ time, where F is the fan-out of the logic gates in the decoder tree (here, we assume a fan-out of two). For a RAM of size N , $M = \lceil \log_2 N \rceil$ address bits are needed to uniquely identify each word. As the number of words in the

memory grows, the length of the address grows logarithmically, so that one level of depth is added to the decoder tree each time the size of the memory doubles. As a practical example, consider a 128 megaword memory that requires 27 levels of decoding ($2^{27} = 128$ Mwords). If we assume that logic gates in the decoding tree switch in 2 ns, then an address can be decoded in 54 ns.

A four level decoder tree for a 16-word RAM is shown in Figure 7-30. As an

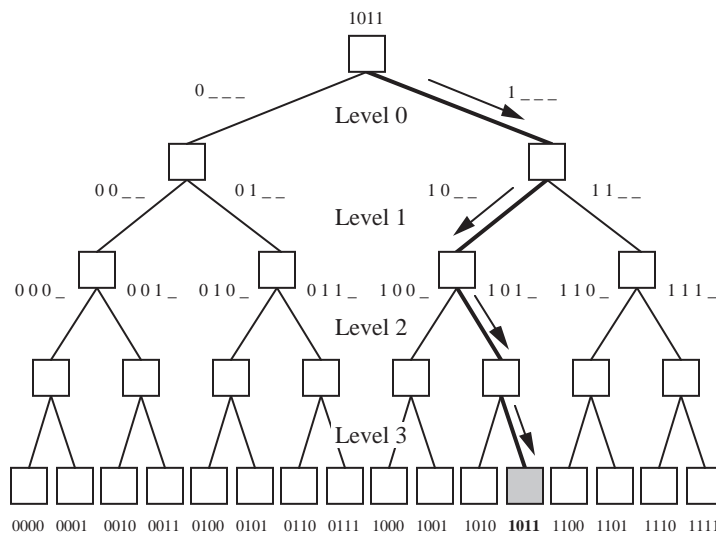


Figure 7-30 A decoding tree for a 16-word random access memory.

example of how the decoder tree works, the address 1011 is presented at the root node. The most significant bit in the address is a 1 so the right path is traversed at Level 0 as indicated by the arrow. The next most significant bit is a 0 so the left path is traversed at Level 1, the next bit is a 1 so the right path is traversed at Level 2, and the least significant bit is a 1 so the rightmost path is traversed next and the addressed leaf is then reached at Level 3.

7.8.3 CONTENT ADDRESSABLE (ASSOCIATIVE) MEMORIES

In an ordinary RAM, an address is applied to the memory, and the contents of the given location are either read or written. In a **content addressable memory** (CAM), also known as an associative memory, a word composed of **fields** is applied to the memory and the resulting address (or index) is returned if the word or field is present in the memory. The physical location of a CAM word is generally not as significant as the values contained in the fields of the word. Rela-

tionships between addresses, values, and fields for RAM and CAM are shown in Figure 7-31.

Address	Value	Field1	Field2	Field3
0000A000	0F0F0000	000	A	9E
0000A004	186734F1	011	0	F0
0000A008	0F000000	149	7	01
0000A00C	FE681022	091	4	00
0000A010	3152467C	000	E	FE
0000A014	C3450917	749	C	6E
0000A018	00392B11	000	0	50
0000A01C	10034561	575	1	84

← 32 bits → ← 32 bits →		← 12 bits →	← 4 bits →	← 8 bits →
Random access memory		Content addressable memory		

Figure 7-31 Relationships between random access memory and content addressable memory.

Values are stored in sequential locations in a RAM, with an address acting as the key to locate a word. Four-byte address increments are used in this example, in which the word size is four bytes. Values are stored in fields in the CAM, and in principle any field of a word can be used to key on the rest of the word. If the CAM words are reordered, then the contents of the CAM are virtually unchanged since physical location has no bearing on the interpretation of the fields. A reordering of the RAM may change the meanings of its values entirely. This comparison suggests that CAM may be a preferred means for storing information when there is a significant cost in maintaining data in sorted order.

When a search is made through a RAM for a particular value, the entire memory may need to be searched, one word at a time, when the memory is not sorted. When the RAM is maintained in sorted order, a number of accesses may still be required to either find the value being searched or to determine the value is not stored in the memory. In a CAM, the value being searched is broadcast to all of the words simultaneously, and logic at each word makes a field comparison for membership, and in just a few steps the answer is known. A few additional steps may be needed to collect the results but in general the time required to search a CAM is less than for a RAM in the same technology, for a number of applications.

Except for maintaining tags in cache memories and in translating among network addresses for routing applications (see Chapter 8), CAMs are not in common use largely due to the difficulty of implementing an efficient design with conventional technology. Consider the block diagram of a CAM shown in Figure

7-32. A Central Control unit sends a comparand to each of 4096 cells, where

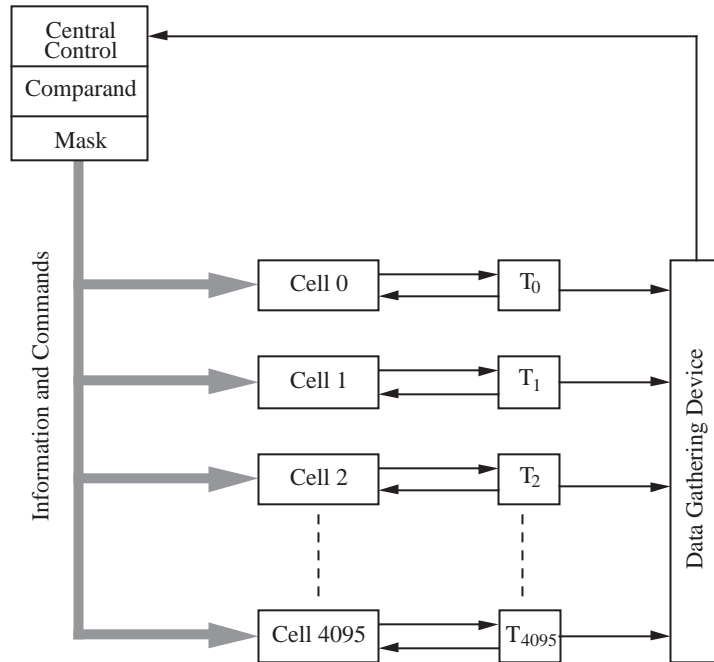


Figure 7-32 Overview of CAM (Foster, 1976).

comparisons are made. The result is put in the Tag bits T_i which are collected by a Data Gathering Device and sent to the Central Control unit (Note that “Tag” is used differently here than in cache memory). When the Central Control unit loads the value to be searched into the comparand register, it sets up a mask to block out fields that are not part of the value. A small local processor in each cell makes a comparison between its local word and the broadcast value and reports the result of the comparison to the Data Gathering Device.

A number of problems arise when an attempt is made to implement this CAM architecture in a conventional technology such as **very large scale integration** (VLSI). The broadcast function that sends the comparand to the cells can be implemented with low latency if a tree structure is used. An H-tree (Mead and Conway, 1980) can be used for the tree layout if it will fit on a single IC. If the tree cannot be contained on a single chip, then connections must be made among a number of chips, which quickly limits chip density. For example, a node of a tree that has a single four-bit input and two four-bit outputs needs 12 input/output (I/O) pins and three control pins if only one node is placed on a chip. A three node subtree needs 25 pins and a seven node subtree needs 45 pins

as illustrated in Figure 7-33. A 63 node subtree requires 325 pins, excluding

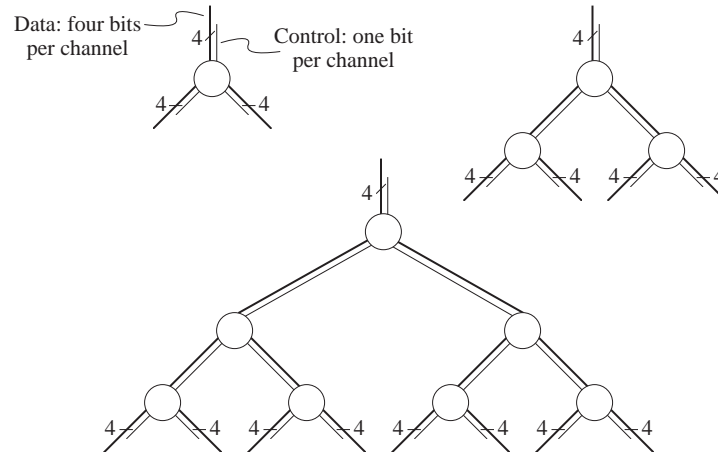


Figure 7-33 Addressing subtrees for a CAM.

power and control pins, which is getting close to the limit of most present day packaging technologies which do not go much higher than 1000 pins per package. A useful CAM would contain thousands of such nodes with wider data paths, so the I/O bandwidth limit is realized early in the design of the CAM. Compromises can be made by multiplexing data onto the limited number of I/O connections but this reduces effective speed, which is a major reason for using a CAM in the first place.

Although implementations of CAMs are difficult, they do find practical uses, such as in TLBs and in computer networks. One application is in a network controller which receives data packets from several processors and then distributes those packets back to the processors or to other network controllers. Each processor has a unique address which the CAM keys on to determine if the target processor for a packet is in its own network or if it must be forwarded to another network.

■ ■ ■ MEMORY DESIGN EXAMPLE: A DUAL-PORT RAM

A **dual-read**, or **dual-port** RAM allows any two words to be simultaneously read from the same memory. As an example, we will design a 2^{20} word by 8-bit dual-read RAM. For our design, any two words can be read at a time, but only one word can be written at a time. Our approach is to create two separate 2^{20} word memories. When writing into the dual-read RAM, the address lines of both

single-read RAMs are set identically and the same data is written to both single-read memories. During a read operation, the address lines of each single-read RAM are set independently, so that two different words can be simultaneously read.

Figure 7-34 shows a block diagram for the dual-read RAM. During a write oper-

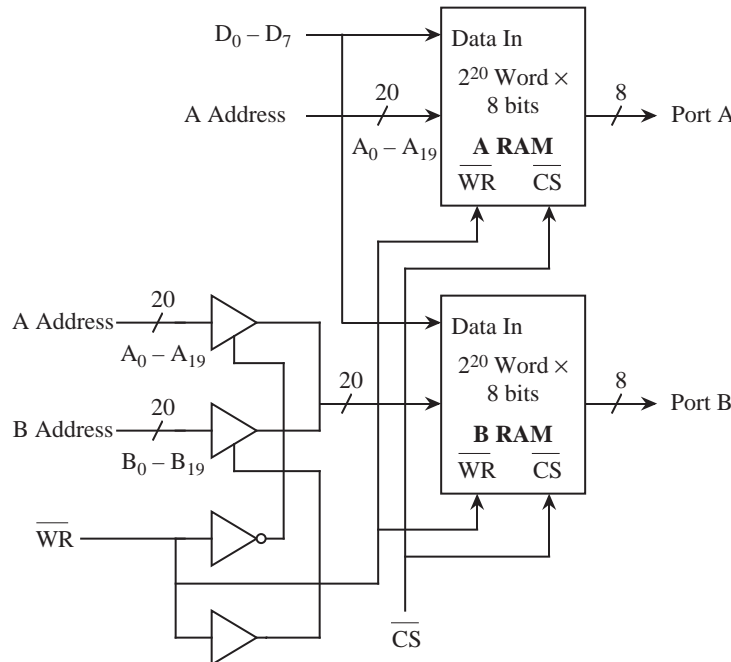


Figure 7-34 Block diagram of dual-read RAM.

ation, the A address is used for both single-read RAMs. Tri-state buffers at the B RAM address inputs are controlled by the $\overline{\text{WR}}$ line. When $\overline{\text{WR}}=0$, the A address is used at the B address input, otherwise, the B address is used at the B address input. The numbers that appear adjacent to the slashes indicate the number of individual lines that are represented by the single line. An 8 next to a slash indicates 8 lines, and a 20 next to a slash indicates 20 lines.

Each tri-state buffer has 20 input lines and 20 output lines, but Figure 7-34 uses a notation in which a single buffer represents 20 separate tri-state buffers that share the same control input. A buffer delay is inserted on the $\overline{\text{WR}}$ line in order to compensate for the delay on the complemented $\overline{\text{WR}}$ line, so that the A and B addresses are not unintentionally simultaneously enabled. ■

7.9 Case Study: Rambus Memory

There was a time when computer technology would be pushed from the laboratory into the marketplace. As the consumer marketplace for computing devices exploded, the “technology push” was replaced by “market pull,” and consumer demand then dominated the preferences of technologists when it came to developing a new memory technology. High performance, expensive memory for high-end processors was displaced by high density, low-cost memory for consumer electronics, such as videogames. It became more profitable for memory manufacturers to address the needs of high volume consumer markets, rather than devote costly chip fabrication facilities to a comparatively small high-end market.

The consumer electronics industry now dominates the memory market, and even high-end, non-consumer processors make heavy use of consumer electronics technology, exploiting architectural enhancements instead, or innovations in supporting technologies (such as high speed interconnects) to compensate for the performance shortcomings of what we might call “videogame memory.”

Videogame memory is not all that low-end, however, and in fact, makes use of extraordinary technology enhancements that squeeze the most performance out of ever denser, low-cost devices. A leading memory technology that is being introduced into Intel-based personal computers in 1999 was developed by Rambus, Inc. The Rambus DRAM (RDRAM) retrieves a block of 8 bytes internal to the DRAM chip on every access, and multiplexes the 8 bytes onto a narrow 8-bit or 16-bit channel, operating at a rate of 800 MHz (or higher).

A typical DRAM core (that is, the storage portion of an ordinary DRAM) can store or retrieve a line of 8 bytes with a 100 MHz cycle. This is internal to the DRAM chip: most DRAMs only deliver one byte per cycle, but the RDRAM technology can multiplex that up to 1 byte per cycle using a higher external clock of 800 MHz. That higher rate is fed to a memory controller (the “chipset” on an Intel machine) which demuxes it to a 32-bit wide data stream at a lower rate, such as 200 MHz, going into a Pentium (or other processor chip).

The Rambus “RIMM” modules (Rambus Inline Memory Modules) look similar to ordinary SIMMs and DIMMs, but they operate differently. The Rambus memory uses **microstrip technology** (also known as **transmission lines**) on the motherboard, which implements a crude shield that reduces **radio frequency** (RF) effects that interfere with data traveling through wires on the motherboard (which are called board **traces**). In designing a printed circuit board (PCB) for

Rambus technology, the critical parameters are (1) dielectric thickness of the PCB, (2) separation of the memory modules, and (3) trace width. There must be a ground plane (an electrical return path) beneath every signal line, with no vias (connections between board layers) along the path. All signals go on the top layer of the PCB. (A PCB can have a number of layers, typically no more than 8). The memory controller and memory modules must all be equally spaced, such as .5 inches from the memory controller to the first RIMM, then .5 in to the next, etc.

The “Rambus Channel” is made up of transmission line traces. The trace widths end up being about twice as wide as ordinary traces, on the order of 12 mils (300 microns). Although 300 microns is relatively small for a board trace, if we want to send 128 signals over a PCB, using a 600 micron **pitch** (center-to-center spacing) with 300 microns between 300 micron traces, this corresponds to a footprint of 128×600 microns = 76 mm. This is a large footprint compared with lower speed solutions that allow a much closer packing density.

In reality, the Rambus Channel only has 13 high speed signals, (the address is serialized onto a single line, there are 8 data lines, 1 parity line, 2 clock lines, and 1 command line) and so the seemingly large footprint is not a near-term problem. With a 16-bit version of the Rambus Channel on the horizon, the bandwidth problem appears to be in hand for a number of years using this technology. Extensibility to large word widths such as 64 bits or 128 bits will pose a significant challenge down the road, however, because the chipset will need to support that same word width – a formidable task with current packaging methods, that already have over 500 pins on the chipset.

Although Rambus memory of this type became available in 1998, the RIMM modules were not widely available until 1999, timed for the availability of a new memory controller (chipset) for the RIMMs. The memory controller is an important aspect of this type of memory because the view of memory that the CPU perceives is different from the physical memory.

Rambus memory is more expensive than conventional DRAM memory, but overall system cost can be reduced, which makes it attractive in low-cost, high performance consumer electronics such as the Nintendo 64 video game console. The Nintendo 64 (see Figure 7-35) has four primary chips: a 64-bit MIPS RS4300i CPU; a Reality coprocessor which integrates all graphics, audio and memory management functions; and two Rambus memory chips.

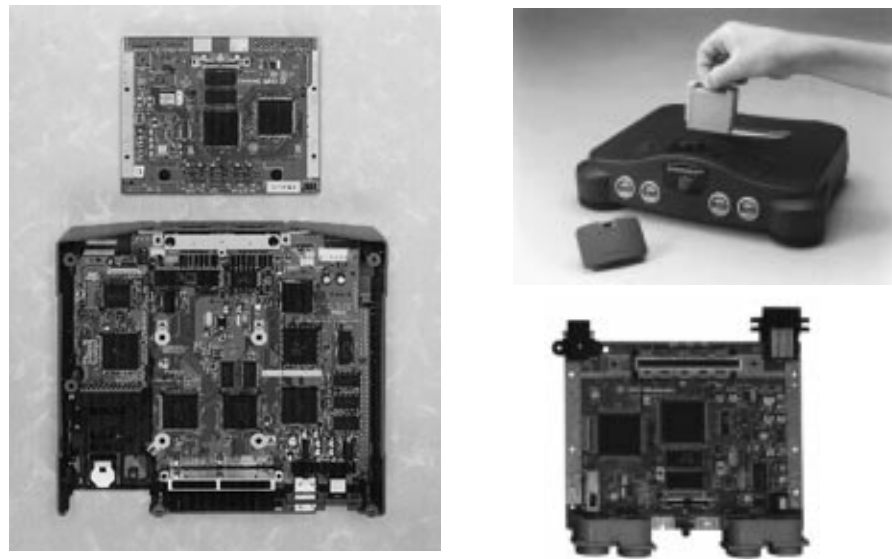


Figure 7-35 Rambus technology on the Nintendo 64 motherboard (top left and bottom right) enables cost savings over the conventional Sega Saturn motherboard design (bottom left). The Nintendo 64 uses costlier plug-in cartridges (top right), as opposed to inexpensive CD-ROMs used by the Sega Saturn. [Photo source: Rambus, Inc.]

The Rambus technology provides the Nintendo 64 with a bandwidth of 562.5 MB/s using a 31-pin interface to the memory controller. By comparison, a system using typical 64-bit-wide synchronous DRAMs (SDRAMs) requires a 110-pin interface to the memory controller. This reduction in pin count allows the memory controller to fit on the same **die** (the silicon chip) as the graphics and sound functions, in a relatively low-cost, 160-pin packaged chip.

The Rambus memory subsystem is made up of two memory chips which occupy 1.5 square inches of board space. An equivalent SDRAM design would require 6 square inches of board space. The space savings of using the Rambus approach enabled Nintendo to fit all of its components on a board measuring five by six inches, which is one quarter the size of the system board used in the competing Sega Saturn. In addition, Nintendo was able to use only a two-layer board instead of the four layers used in the Sega Saturn.

The cost savings Nintendo realized by choosing the Rambus solution over the 64-bit SDRAM approach are considerable, but should be placed in perspective with the overall market. The ability to use a two-layer implementation saved Nintendo \$5 per unit in manufacturing costs. Taken altogether, Nintendo esti-

mates the total bill of materials cost savings over an equivalent SDRAM-based design was about 20 percent.

These cost savings need to be placed in perspective with the marketplace, however. The competing Sega Saturn and Sony Playstation use CD-ROMs for game storage, which cost under \$2 each to mass produce. The Nintendo 64 uses a plug-in ROM cartridge that costs closer to \$10 each to mass produce, and can only store 1% of what can be stored on a CD-ROM. This choice of media may have a great impact on the overall system architecture, and so the Rambus approach may not benefit all systems to the same degree. Details matter a great deal when it comes to evaluating the impact of a new technology on a particular market, or even a segment of a market.

7.10 Case Study: The Intel Pentium Memory System

The Intel Pentium processor is typical of modern processors in its memory configurations. Figure 7-36 shows a simplified diagram of the memory elements and

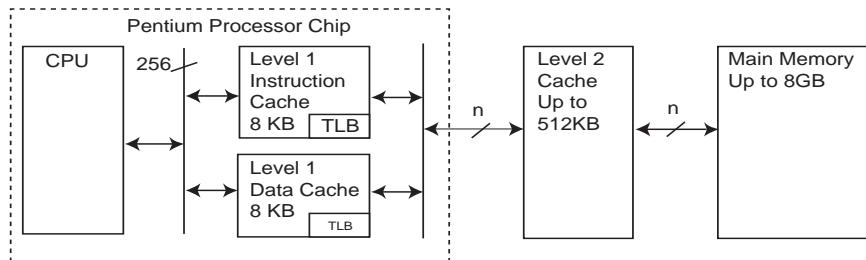


Figure 7-36 The Intel Pentium memory system.

data paths. There are two L1 caches on board the actual Pentium chip, an instruction, or I-cache, and a data, or D-cache. Each cache has a 256 bit (32 byte) line size, with an equally-sized data path to the CPU. The L1 caches are 2-way set associative, and each way has a single LRU bit. The D-cache can be set to write-through or writeback on a per-line basis. The D cache is write no-allocate: misses on writes do not result in a cache fill. Each cache is also equipped with a TLB that translates virtual to physical addresses. The D-cache TLB is 4-way set associative, with 64 entries, and is dual-ported, allowing two simultaneous data reference translations. The I-cache TLB is 4-way set associative with 32 entries.

The L2 cache, if present, is 2-way set associative, and can be 256 KB or 512 KB in size. The data bus, shown as “n” in the figure, can be 32, 64, or 128 bits in size.

THE MESI PROTOCOL

The Pentium D cache, and the L2 cache if present support the MESI cache coherency protocol for managing multiprocessor access to memory. Each D-cache line has two bits associated with it that store the MESI state. Each cache line will be in one of the four states:

- M - Modified. The contents of the cache line have been modified and are different from main memory.
- E - Exclusive. The contents of the cache line have not been modified, and are the same as the line in main memory.
- S - Shared. The line is, or may be shared with another cache line belonging to another processor.
- I - Invalid. The line is not in the cache. Reads to lines in the I state will result in cache misses.

Table 7- 2[†] shows the relationship between the MESI state, the cache line, and

Cache Line State	M Modified	E Exclusive	S Shared	I Invalid
Cache line valid?	Yes	Yes	Yes	No
Copy in memory is...	...out of date.	...valid	...valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line...	...does not go to the bus	...does not go to the bus	...goes to the bus and updates cache	...goes directly to the bus

Table 7- 2 MESI cache line states.

[†]. Taken from *Pentium Processor User's Manual, Volume 3, Architecture and Programming Manual*, © Intel Corporation, 1993.

the equivalent line in main memory. The MESI protocol is becoming a standard way of dealing with cache coherency in multiprocessor systems.

The Pentium processor also supports up to six main memory segments (there can be several thousand segments, but no more than 6 can be referenced through the segment registers.) As discussed in the chapter, each segment is a separate address space. Each segment has a base—a starting location within the 32-bit physical address space, and a limit, the maximum size of the segment. The limit may be either 2^{16} bytes or $2^{16} \times 2^{12}$ bytes in size. That is, the granularity of the limit may be one byte or 2^{12} bytes in size.

Paging and segmentation on the Pentium can be applied in any combination:

Unsegmented, unpaged memory. The virtual address space is the same as the physical address space. No page tables or mapping hardware is needed. This is good for high performance applications that do not have a lot of complexity, that do not need to support growing tables, for example.

Unsegmented, paged memory. Same as for the unsegmented, unpaged memory above, except that the linear address space is larger as a result of using disk storage. A page table is needed to translate between virtual and physical addresses. A translation lookaside buffer is needed on the Pentium core, working in conjunction with the L1 cache, to reduce the number of page table accesses.

Segmented, unpaged memory. Good for high complexity applications that need to support growing data structures. This is also fast: segments are fewer in number than pages in a virtual memory, and all of the segmentation mapping hardware typically fits on the CPU chip. There is no need for disk accesses as there would be for paging, and so access times are more predictable than when paging is used.

Segmented, paged memory. A page table, segment mapping registers, and TLB all work in conjunction to support multiple address spaces.

Segmentation on the Intel Pentium processor is quite powerful but is also quite complex. We only explore the basics here, and the interested reader is referred to (Intel, 1993) for a more complete description.

■ SUMMARY

Memory is organized in a hierarchy in which the densest memory offers the least performance, whereas the greatest performance is realized with the memory that has the least density. In order to bridge the gap between the two, the principle of locality is exploited in cache and virtual memories.

A cache memory maintains the most frequently used blocks in a small, fast memory that is local to the CPU. A paged virtual memory augments a main memory with disk storage. The physical memory serves as a window on the paged virtual memory, which is maintained in its entirety on a hard magnetic disk.

Cache and paged virtual memories are commonly used on the same computer, but for different reasons. A cache memory improves the average access time to the main memory, whereas a paged virtual memory extends the size of the main memory.

In an example memory architecture, the Intel Pentium has a split L1 cache and a TLB that reside on the Pentium core, and a unified L2 cache that resides in the same package as the Pentium although on a different silicon die. When paging is implemented, the page table is located in main memory, with the TLB reducing the number of times that the TLB is referenced. The L1 and L2 cache memories then reduce the number of times main memory is accessed for data. The Pentium also supports segmentation, which has its own set of mapping registers and control hardware that resides on the Pentium.

■ FURTHER READING

(Stallings, 1993) and (Mano, 1991) give readable explanations of RAM. A number of memory databooks (Micron, 1992) and (Texas Instruments, 1991) give practical examples of memory organization. (Foster, 1976) is the seminal reference on CAM. (Mead and Conway, 1980) describe the H-tree structure in the context of VLSI design. (Franklin *et al*, 1982) explores issues in partitioning chips, which arise in splitting an H-tree for a CAM. (Sedra and Smith, 1997) discuss the implementation of several kinds of static and dynamic RAM.

(Hamacher *et al*, 1990) gives a classic treatment of cache memory. (Tanenbaum, 1990) gives a readable explanation of virtual memory. (Hennessy and Patterson, 1995) and (Przybylski, 1990) cover issues relating to cache performance. Segmentation on the Intel Pentium processor is covered in (Intel, 1993). Kingston

Technology gives a broad tutorial on memory technologies at <http://www.kingston.com/king/mg0.htm>.

Foster, C. C., *Content Addressable Parallel Processors*, Van Nostrand Reinhold Company, (1976).

Franklin, M. A., D. F. Wann, and W. J. Thomas, "Pin Limitations and Partitioning of VLSI Interconnection Networks," *IEEE Trans Comp.*, **C-31**, 1109, (Nov. 1982).

Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 3/e, McGraw Hill, (1990).

Hennessy, J. L. and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2/e, Morgan Kaufmann Publishers, San Mateo, California, (1995).

Intel Corporation, *Pentium Processor User's Manual*, Volume 3: *Architecture and Programming Manual*, (1993).

Knuth, D., *The Art of Computer Programming: Fundamental Algorithms*, vol. 1, 2/e, Addison-Wesley, (1974).

Mano, M., *Digital Design*, 2/e, Prentice Hall, (1991).

Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison Wesley, (1980).

Micron, *DRAM Data Book*, Micron Technologies, Inc., 2805 East Columbia Road, Boise, Idaho, (1992).

Przybylski, S. A., *Cache and Memory Hierarchy Design*, Morgan Kaufmann Publishers, (1990).

Sedra, A., and Smith, K., *Microelectronic Circuits*, 4/e, Oxford University Press, New York, (1997).

Stallings, W., *Computer Organization and Architecture*, 3/e, MacMillan Publishing, New York, (1993).

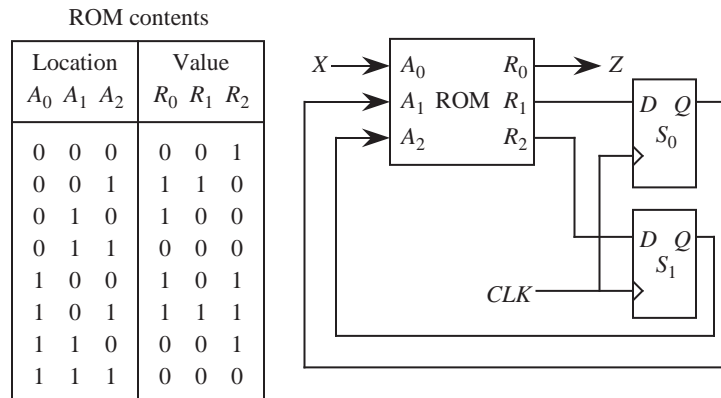
Tanenbaum, A., *Structured Computer Organization*, 3/e, Prentice Hall, Engle-

wood Cliffs, New Jersey, (1990).

Texas Instruments, *MOS Memory: Commercial and Military Specifications Data Book*, Texas Instruments, Literature Response Center, P.O. Box 172228, Denver, Colorado, (1991).

PROBLEMS

7.1 A ROM lookup table and two D flip-flops implement a state machine as shown in the diagram below. Construct a state table that describes the machine.



7.2 Fill in four memory locations for the lookup table shown in Figure 7-11 in which each of the four operations: add, subtract, multiply, and divide are performed on $A=16$ and $B=4$. Show the address and the value for each case.

7.3 Design an eight-word, 32-bit RAM using 8×8 RAMs.

7.4 Design a 16-word, four-bit RAM using 4×4 RAMs and a single external decoder.

7.5 Given a number of n -word by p -bit RAM chips:

- (a) Show how to construct an n -word \times $4p$ -bit RAM using these chips. Use any other logic components that you have seen in the text that you feel are needed.
- (b) Show how to construct a $4n$ -word \times p -bit RAM using these chips.

7.6 Draw the circuit for a 4-to-16 tree decoder, using a maximum fan-in and fan-out of two.

7.7 A direct mapped cache consists of 128 slots. Main memory contains 16K blocks of 16 words each. Access time of the cache is 10 ns, and the time required to fill a cache slot is 200 ns. Load-through is not used; that is, when an accessed word is not found in the cache, the entire block is brought into the cache, and the word is then accessed through the cache. Initially, the cache is empty. Note: When referring to memory, 1K = 1024.

(a) Show the format of the memory address.

(b) Compute the hit ratio for a program that loops 10 times from locations 15 – 200. Note that although the memory is accessed twice during a miss (once for the miss, and once again to satisfy the reference), a hit does not occur for this case. To a running program, only a single memory reference is observed.

(c) Compute the effective access time for this program.

7.8 A fully associative mapped cache has 16 blocks, with eight words per block. The size of main memory is 2^{16} words, and the cache is initially empty. Access time of the cache is 40 ns, and the time required to transfer eight words between main memory and the cache is 1 μ s.

(a) Compute the sizes of the tag and word fields.

(b) Compute the hit ratio for a program that executes from 20–45, then loops four times from 28–45 before halting. Assume that when there is a miss, that the entire cache slot is filled in 1 μ s, and that the first word is not seen by the CPU until the entire slot is filled. That is, assume load-through is not used. Initially, the cache is empty.

(c) Compute the effective access time for the program described in part (b) above.

7.9 Compute the total number of bits of storage needed for the associative mapped cache shown in Figure 7-13 and the direct mapped cache shown in Figure 7-14. Include Valid, Dirty, and Tag bits in your count. Assume that the word size is eight bits.

7.10 (a) How far apart do successive memory references need to be spaced to cause a miss on every cache access using the direct mapping parameters shown in Figure 7-14?

(b) Using your solution for part (a) above, compute the hit ratio and effective access time for that program with $T_{\text{Miss}} = 1000$ ns, and $T_{\text{Hit}} = 10$ ns. Assume that load-through is used.

7.11 A computer has 16 pages of virtual address space but only four physical page frames. Initially the physical memory is empty. A program references the virtual pages in the order: 0 2 4 5 2 4 3 11 2 10.

(a) Which references cause a page fault with the LRU page replacement policy?

(b) Which references cause a page fault with the FIFO page replacement policy?

7.12 On some computers, the page table is stored in memory. What would happen if the page table is swapped out to disk? Since the page table is used for every memory reference, is there a page replacement policy that guarantees that the page table will not get swapped out? Assume that the page table is small enough to fit into a single page (although usually it is not).

7.13 A virtual memory system has a page size of 1024 words, eight virtual pages, four physical page frames, and uses the LRU page replacement policy. The page table is as follows:

Page #	Present bit		Page frame field	
	Disk address			
0	0	01001011100	xx	
1	0	11101110010	xx	
2	1	10110010111	00	
3	0	00001001111	xx	
4	1	01011100101	01	
5	0	10100111001	xx	
6	1	00110101100	11	
7	0	01010001011	xx	

- (a) What is the main memory address for virtual address 4096?
- (b) What is the main memory address for virtual address 1024?
- (c) A fault occurs on page 0. Which page frame will be used for virtual page 0?

7.14 When running a particular program with N memory accesses, a computer with a cache and paged virtual memory generates a total of M cache misses and F page faults. T_1 is the time for a cache hit; T_2 is the time for a main memory hit; and T_3 is the time to load a page into main memory from the disk.

- (a) What is the cache hit ratio?
- (b) What is the main memory hit ratio? That is, what percentage of main memory accesses do not generate a page fault?
- (c) What is the overall effective access time for the system?

7.15 A computer contains both cache and paged virtual memories. The cache can hold either physical or virtual addresses, but not both. What are the issues involved in choosing between caching virtual or physical addresses? How can these problems be solved by using a single unit that manages all memory mapping functions?

7.16 How much storage is needed for the page table for a virtual memory that has 2^{32} bytes, with 2^{12} bytes per page, and 8 bytes per page table entry?

7.17 Compute the gate input count for the decoder(s) of a 64×1 -bit RAM for both the 2D and the 2-1/2D cases. Assume that an unlimited fan-in/fan-out is allowed. For both cases, use ordinary two-level decoders. For the 2 1/2D case, treat the column decoder as an ordinary MUX. That is, ignore its behavior as a DEMUX during a write operation.

7.18 How many levels of decoding are needed for a 2^{20} word 2D memory if a fan-in of four and a fan-out of four are used in the decoder tree?

7.19 A video game cartridge needs to store 2^{20} bytes in a ROM.

(a) If a 2D organization is used, how many leaves will be at the deepest level of the decoder tree?

(b) How many leaves will there be at the deepest level of the decoder tree for a 2-1/2D organization?

7.20 The contents of a CAM are shown below. Which set of words will respond if a key of 00A00020 is used on fields 1 and 3? Fields 1 and 3 of the key must match the corresponding fields of a CAM word in order for that word to respond. The remaining fields are ignored during the matching process but are included in the retrieved words.

Field → 4 3 2 1 0

F	1	A	0	0	0	2	8
0	4	2	9	D	1	F	0
3	2	A	1	1	0	3	E
D	F	A	0	5	0	2	D
0	0	5	3	7	F	2	4

7.21 When the TLB shown in Figure 7-27 has a miss, it accesses the page table to resolve the reference. How many entries are in that page table?