Chapter 16

Memory System Design

Objectives

- To introduce memory design techniques using flip-flops and latches;
- To describe how memory units can be interfaced to the system bus;
- To present a method to design larger memory modules using standard memory chips;
- To discuss various ways of mapping memory modules to the memory address space;
- To explain the reasons for the adverse impact of unaligned data on performance;
- To give details on how interleaved memories are constructed.

In Chapter 4, we have seen how flip-flops and latches can be used to store a bit. This chapter builds on this foundation and explains how we can use these basic devices and build larger memory blocks and modules. We start off with a simple design process that can be used to design memory blocks using flip-flops and latches. In the following section, we present various ways of connecting the memory modules to the system bus. In particular, we describe tristate buffers that are extensively used as interface devices for connection to the system bus. The other techniques use either an open collector device or a multiplexer, both of which are useful only in certain special cases.

We also discuss how larger memories can be built using narrower memory chips. The design process is fairly intuitive. The basic technique involves using a two-dimensional array of memory chips. A characteristic of these designs is the use of chip select. Chip select input can be used to select or deselect a chip or a memory module. Chip select allows us to connect multiple devices to the system bus. Appropriate chip select signal generation facilitates communication among the entities connected to the system bus.

Chip select logic is also useful in mapping memory modules to memory address space. We present details about two ways of mapping a memory module to the address space. In Section 16.7, we explain the reasons why data alignment leads to improved performance. Before ending the chapter, we describe the structure of interleaved memories. We end the chapter with a summary.

16.1 Introduction

Flip-flops and latches provide the basic capability to store a bit of data. In Chapter 4, we have seen several types of flip-flops and latches. These devices can be replicated to build larger memory units. For example, we can place 16 JK flip-flops together to store a 16-bit word. All 16 flip-flops would have their clock inputs tied together to form a single common clock to write a 16-bit word.

There are two types of memories: read/write memory and read-only memory. As the name suggests, a read/write memory allows reading as well as writing. This type of memory is referred to as random access memory (RAM). Read-only memory, on the other hand, allows only reading of its contents. These are referred to as ROMs.

Several types of ROMs are available; each type requires a special kind of writing procedure. Writing into a ROM is called programming the ROM. ROMs can be factory programmed at the time of manufacture. These are suitable when the required quantity is large to amortize the overhead. There are user-programmable ROMs (PROMs). For example, the user can selectively blow electric fuses by sending a high current to the program contents of a PROM. There are also erasable PROMs (EPROMs) that can be programmed several times. Exposing them to an ultraviolet light for a few minutes can erase the contents of the entire EPROM. Electrically erasable PROMs (EEPROMs), on the other hand, allow the user to selectively erase contents. Note that, even though only read/write memories are called random access memories, both read/write memories and read-only memories are random access memories.

ROMs are typically used to store the boot program, as ROMs are nonvolatile memories. That is, ROMs do not require power to retain their contents. That's why they are useful to store the program that your processor can read before loading the operating system. The bulk of the system memory consists of RAM.

RAM can be grouped into two basic types: static and dynamic. Static RAM (SRAM) uses a latch or flip-flop as the basic cell to store a bit. Dynamic RAM (DRAM) uses a different technique. DRAM uses a tiny capacitor to store a bit. Since capacitors leak charge, DRAMs need periodic refreshing to retain their contents. Furthermore, reads are destructive as reading destroys the contents. This characteristic of DRAMs requires write-after-read to restore the contents after a read cycle. The main reason for the popularity of DRAMs is their price advantage over SRAMs. There are also additional advantages of using DRAMs including low power consumption, less heat, and high-density packaging. SRAMs, on the other hand, are expensive but faster than DRAMs. DRAMs are used for main memory, and SRAMs are used for cache memory.

Despite these technical differences, SRAMs and DRAMs as well as ROMs can be treated as simple building blocks to construct larger memory units. To illustrate the concepts, we focus on static RAMs. The techniques we discuss here can be easily extended to other types of memories.

16.2 A Simple Memory Block

In the digital logic chapters, we have discussed several basic building blocks such as D flip-flops, multiplexers, and decoders. We now describe how these digital circuits can be used to build a simple memory block.

16.2.1 Memory Design with D Flip-Flops

We begin our discussion with how one can build memories using D flip-flops. Recall that we use flip-flops for edge-triggered devices and latches for level-sensitive devices. The principle of constructing memory out of D flip-flops is simple. We use a two-dimensional array of D flip-flops with each row storing a word. The number of rows is equal to the number of words the memory should store. We refer to this as "horizontal" expansion to increase the word width and "vertical" expansion to increase the number of words.

In general, the number of columns and the number of rows is a power of two. We use the notation M $\times$ N memory to represent a memory that can store M words, where each word is N-bits long.

Figure 16.1 shows a 4 $\times$ 3 memory built with 12 D flip-flops organized as a 4 $\times$ 3 array. Since all flip-flops in a row are storing a word of data, each row of flip-flops has their clock signals tied together to form a single clock signal for each row. All flip-flops in a column receive input from the same input data line. For example, the rightmost column D inputs are connected to the input data DI0.

This memory requires two address lines to select one of the four words. The two address lines are decoded to select a specific row by using a 2-to-4 decoder. The low-active write signal ($\overline{\mathrm{WR}}$) is gated through an AND gate as shown in Figure 16.1. Depending on the address, only one of the four decoder output lines will be high, permitting the $\overline{\mathrm{WR}}$ signal to clock the selected row to write the 3-bit data present on DI0 to DI2 lines. Note that the decoder along with the four AND gates forms a demultiplexer that routes the $\overline{\mathrm{WR}}$ signal to the row selected by the address lines A1 and A0.

The design we have done so far allows us to write a 3-bit datum into the selected row. To complete the design, we have to find a way to read data from this memory. As each bit of data is supplied by one of the four D flip-flops in a column, we have to find a way to connect these four outputs to a single data out line. A natural choice for the job is a 4-to-1 multiplexer. The MUX selection inputs are connected to the address lines to allow appropriate data on the output lines DO0 through DO2. The final design is shown in Figure 16.1.

16.2.2 Problems with the Design

The memory design example tells us how we can put bit-level devices together to form a higher-level two-dimensional memory. This design, however, has several problems. To understand the kind of problems created by this design, let us look at how the memory unit is connected to the system bus. In Chapter 1, we presented a simplified view of a typical computer system (see Figure 1.5 on page 13). This figure clearly shows how the system bus connects the three main components: processor, memory, and I/O devices.

Focusing on the interactions between the processor and memory unit, we see that the data bus is bidirectional. That is, data input and output lines are not separate. Our memory design shown in Figure 16.1 uses separate data in and out lines. This is the first problem with the design. To be able to connect to the data bus, we should be able to connect the corresponding DI and DO lines (i.e., DI0 and DO0 should be tied together, DI1 and DO1 should be tied together, etc.). However, our design will not allow us to do this.

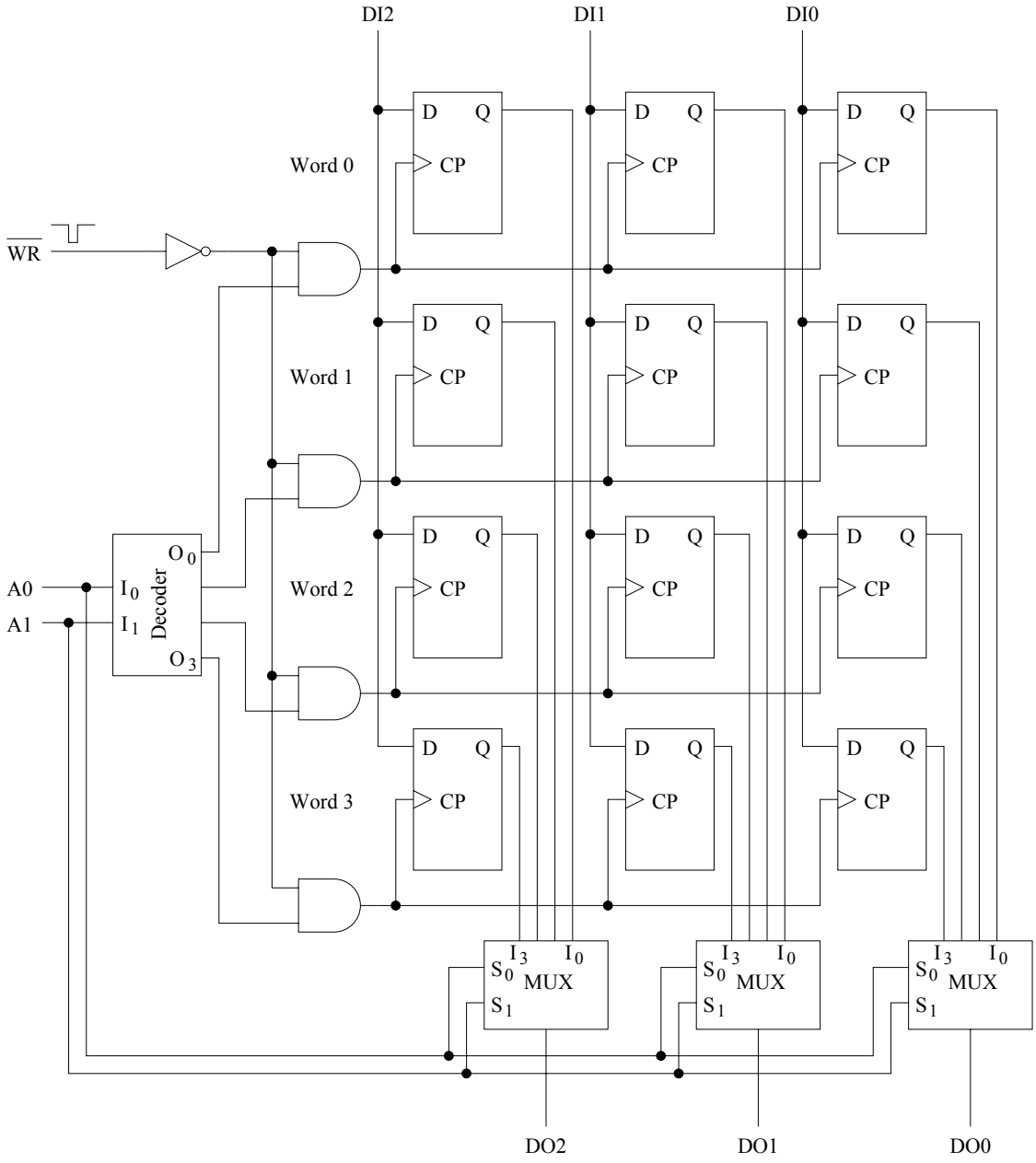


Figure 16.1 A simple 4 × 3 memory design using D flip-flops.

Another problem with this design is that we will not be able to use it as a building block to construct larger memory blocks. For example, we cannot build an 8×3 memory using two of these units. Such a hierarchical design needs a chip select input that either selects or deselects the whole unit.

We return to this design example in Section 16.4 that solves these two problems. Before revising our design, we look at some background information on some of the techniques we can use to connect multiple outputs to a single line as required for our data bus connections.

16.3 Techniques to Connect to a Bus

Data bus connections impose several requirements, some of which we have discussed in the last section. First and foremost, the data bus connections should support bidirectional data transfer. This eliminates those designs that require separate data in and out lines as in our previous design (see Figure 16.1). Furthermore, since several devices can be connected to the data bus, only the selected device should place data on the data bus. All other devices attached to the data bus should disconnect themselves (or at least act as though they were not connected to the data bus). We describe several techniques that satisfy one or both of these requirements.

16.3.1 Using Multiplexers

Based on our discussion of digital logic circuits, we know that we can use multiplexers to send multiple outputs onto a single line. In fact, we have used three 4-to-1 multiplexers in our memory design example in Figure 16.1. Multiplexers, however, do not satisfy the two requirements we have mentioned. You cannot use multiplexers to connect DI and DO inputs. Similarly, they do not provide an efficient way to implement the chip select function.

The next two subsections present two other techniques that are useful in deriving an implementation that satisfies these requirements.

16.3.2 Using Open Collector Outputs

A technique that is commonly used to connect several outputs uses "open collector" outputs. To give you some insight, Figure 16.2*a* shows how a normal gate output is designed (this is a reproduction of the NOT gate implementation shown in Figure 2.6 on page 47). A problem with this design is that the outputs of several gates cannot be connected. To see why, assume that three such normal gate outputs are tied together. In a normal gate, when the transistor is on, it can expect Vcc/R amperes of current through it. If one of the transistors is on and the other two are off, this one transistor would have to carry current that is three times its capacity (i.e., 3Vcc/R amperes). You can generalize this to a larger number of connections and see the corresponding increase in the required current-carrying capacity of each transistor. Remember that passing excessive current causes the transistor to burn out.

The idea of open collector design is to take the resistor out of the chip and provide open collector output to the chip pins (see Figure 16.2*b*). Open collector output can be in state 0 or 1 just as with a normal output. In addition, an open collector output can also be in a high

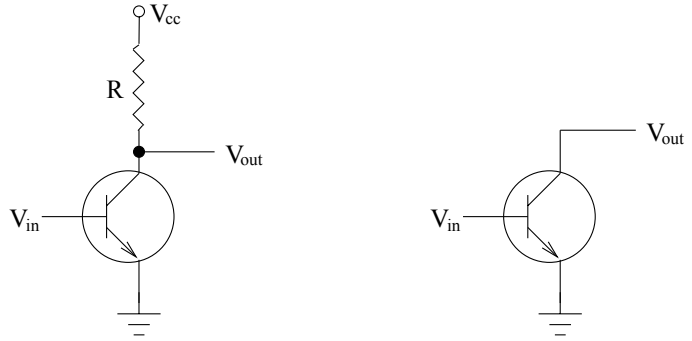
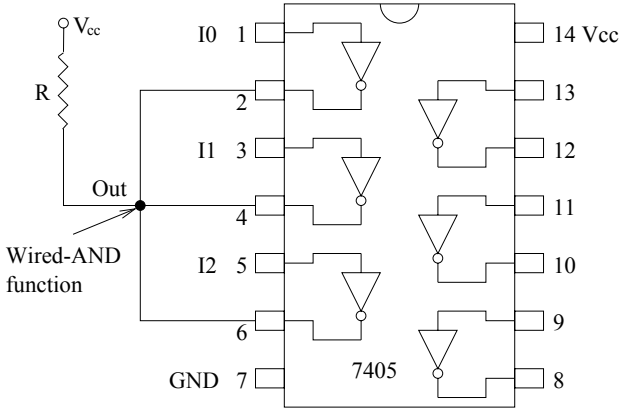


Figure 16.2 A simplified NOT gate implementation with normal and open collector outputs.



I2	I1	I0	Out
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(b) Truth table

Figure 16.3 Open collector inverter chip: (a) 7405 connection diagram and an example circuit; (b) truth table for the example circuit in (a).

impedance (Z) state, in which the transistor is off. With the open collector outputs, we can connect several outputs without worrying about the capacity of these transistors. This is due to the fact that there is a single resistor that is connected to these common outputs as shown in Figure 16.3*a*. Thus we limit the current flow to Vcc/R amperes, independent of the number of connections.

Figure 16.3*a* shows details of the 7405 inverter chip that provides open collector outputs. The chip uses the same connection layout as the 7404 chip we have seen before (see page 50). This figure also shows how we can connect three outputs by "pulling" the **Out** point to Vcc through a resistor R. As noted in the diagram, by connecting these three open collector outputs, we are implementing the logical AND function. This is often referred to as the *wired-AND* function. The truth table in Figure 16.3*b* shows that this circuit implements a three-input NOR function.

Several open collector chips are commercially available. We present details about an example chip in Figure 16.4. This chip is a 4 $\times$ 4 register that uses D latches to store the data. Internally, the design is similar to that shown in Figure 16.1 except that the output goes through an open collector gate. The chip uses two read address lines (RA0 and RA1) and two write address lines (WA0 and WA1) along with two separate read ($\overline{\text{RE}}$) and write enable signals ($\overline{\text{WE}}$) as shown in Figure 16.4*b*. The use of separate address lines and enable signals for read and write operations gives the chip the ability to read and write simultaneously. This figure also shows the function table for read and write operations. We can use this chip to build larger memories by connecting outputs of several chips. For example, we can design an 8 $\times$ 4 memory by using two 74170 chips. The outputs of these two chips can be connected as we did in the open collector inverter chip example.

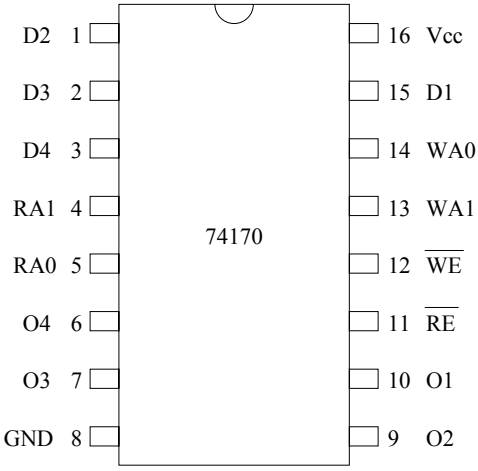
Open collector outputs are also used when signal drivers are needed. BCD to seven-segment decoder/driver chips are prime examples of this type of use of open collector outputs. An example of such a chip is the 74249, which is essentially an open collector version of the 7449 seven-segment decoder chip discussed in Chapter 2 on page 68.

A Problem: Can we use open collector outputs to solve our problem? We can certainly use open collector outputs to tie several outputs together. There is one major problem that makes them unsuitable for our memory design example. That is, the output is in high impedance (Z) state only if the output is supposed to be high. This forces us to apply appropriate inputs to get into this state. For example, if we are using the 7405 chip, we have to make sure that the inputs are low so that the outputs of deselected devices will not interfere with the output of a selected device. Thus, open collector devices are not suitable in the design of memory modules. We present the most commonly used solution next.

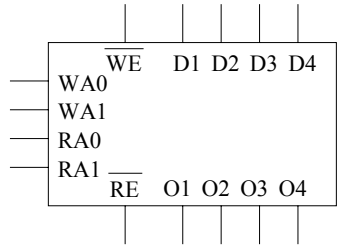
16.3.3 Using Tristate Buffers

With the exception of the open collector devices, all other logic circuits we have discussed have two possible states: 0 or 1. The devices we discuss now are called tristate devices as they can be in three states: 0, 1, or Z state. The states are similar to those associated with the open collector devices. There is one big difference between the two: tristate devices use a separate control signal so that the output can be in a high impedance state, independent of the data input. This particular feature makes them suitable for bus connections.

Figure 16.5*a* shows the logic symbol for a tristate buffer. When the enable input (E) is low, the buffer acts as an open circuit (i.e., output is in the high impedance state Z) as shown in



(a) Connection diagram



(b) Logic symbol

$\overline{WE}$	WA1	WA0	D inputs to
0	0	0	Word 0
0	0	1	Word 1
0	1	0	Word 2
0	1	1	Word 3
1	X	X	None

(c) Write function table

$\overline{RE}$	RA1	RA0	Output from
0	0	0	Word 0
0	0	1	Word 1
0	1	0	Word 2
0	1	1	Word 3
1	X	X	None (Z)

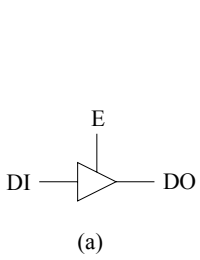
(d) Read function table

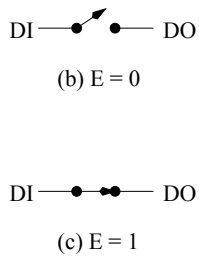
Figure 16.4 An example open collector register chip (X = don't care input, and Z = high impedance state).

Figure 16.5*b*; otherwise, it acts as a short circuit (Figure 16.5*c*). The enable input must be high in order to pass the input data to output, as shown in the truth table (see Figure 16.5*d*).

Two example tristate buffer chips are shown in Figure 16.6. The 74367 chip provides six tristate buffers. These six buffers are grouped into four and two buffers, each with a separate enable input. Both enable inputs are low-active, as represented by $\overline{E1}$ and $\overline{E2}$. The 74368 chip is similar to the 74367 chip except that it provides tristate inverters instead of buffers. In the next section, we use the tristate buffers to modify our memory design example of Figure 16.1.

The 74373 is an example chip that provides tristate outputs. It uses eight D latches internally and feeds each latch $\overline{Q}$ through an inverting tristate buffer. The output enable ($\overline{OE}$) is a low-active input that controls the output of inverting tristate buffers. Thus, when $\overline{OE} = 1$, the outputs O0 through O7 float (i.e., the outputs are in the high impedance state). The data can be written

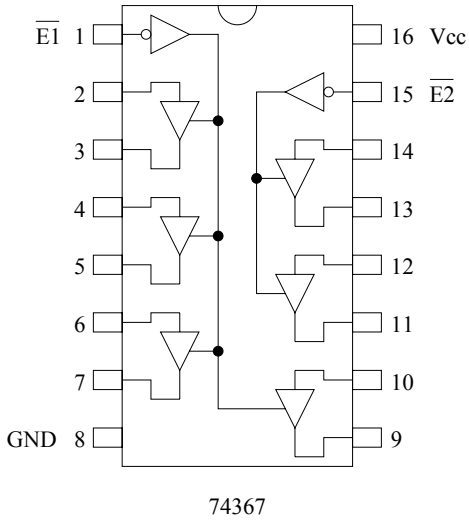




Inputs		Output
E	DI	DO
1	0	0
1	1	1
0	X	Z

(d)

Figure 16.5 Tristate buffer: (a) logic symbol; (b) it acts as an open circuit when the enable input is inactive (E = 0); (c) it acts as a closed circuit when the enable input is active (E = 1); (d) truth table (X = don't care input, and Z = high impedance state).



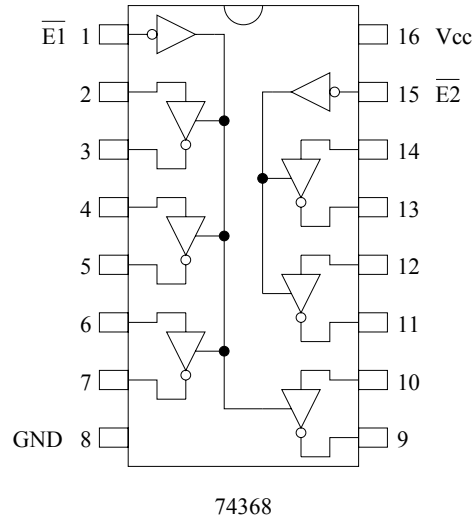
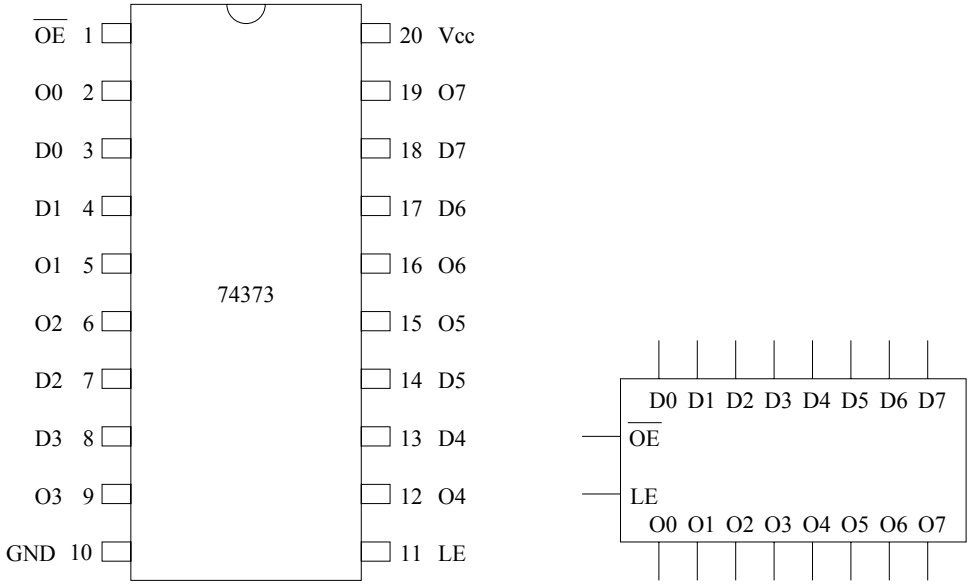


Figure 16.6 Two example tristate buffer chips.

into the register by making the latch enable (LE) input high. Because this chip uses latches, the output follows the input as long as the LE input is high and the $\overline{OE}$ is low. In Section 16.5.1, we discuss how we can use this chip to build larger memories.

16.4 Building a Memory Block

We are now in a position to revise our previous memory design (shown in Figure 16.1) to remedy the two main problems we have identified before. Our revision is a minor one in the sense that we need to pass the outputs of the multiplexers through tristate buffers as shown in



(a) Connection diagram (b) Logic symbol

Figure 16.7 An example 8-bit tristate register.

Figure 16.8. The enable input signal for these output tristate buffers is generated by ANDing the chip select and read signals. The two inverters are used to provide low-active chip select ($\overline{\text{CS}}$) and memory read ($\overline{\text{RD}}$) inputs to the memory block.

With the use of these tristate buffers, we can now tie the corresponding data in and out signal lines together to satisfy the data bus connection requirements. Furthermore, we can completely disconnect the outputs of this memory block by making $\overline{\text{CS}}$ high.

We can represent our design using the logic symbol shown in Figure 16.9. Our design uses separate read and write signals. These two signals are part of the control bus (see Figure 1.5). It is also possible to have a single line to serve as a read and write line. For example, a 0 on this line can be interpreted as write and a 1 as read. Such signals are represented as the $\overline{\text{WR}}$/RD line, indicating low-active write and high-active read (see Exercise 16–5).

16.5 Building Larger Memories

Now that we know how to build memory blocks using devices that can store a single bit, we move on to building larger memory units using these memory blocks. We explain the design process by using some examples. The first example deals with designing an independent memory unit that is not hard-wired to a specific address in the memory address space. In a sense it is a larger version of the design shown in Figure 16.8. The second example shows how these

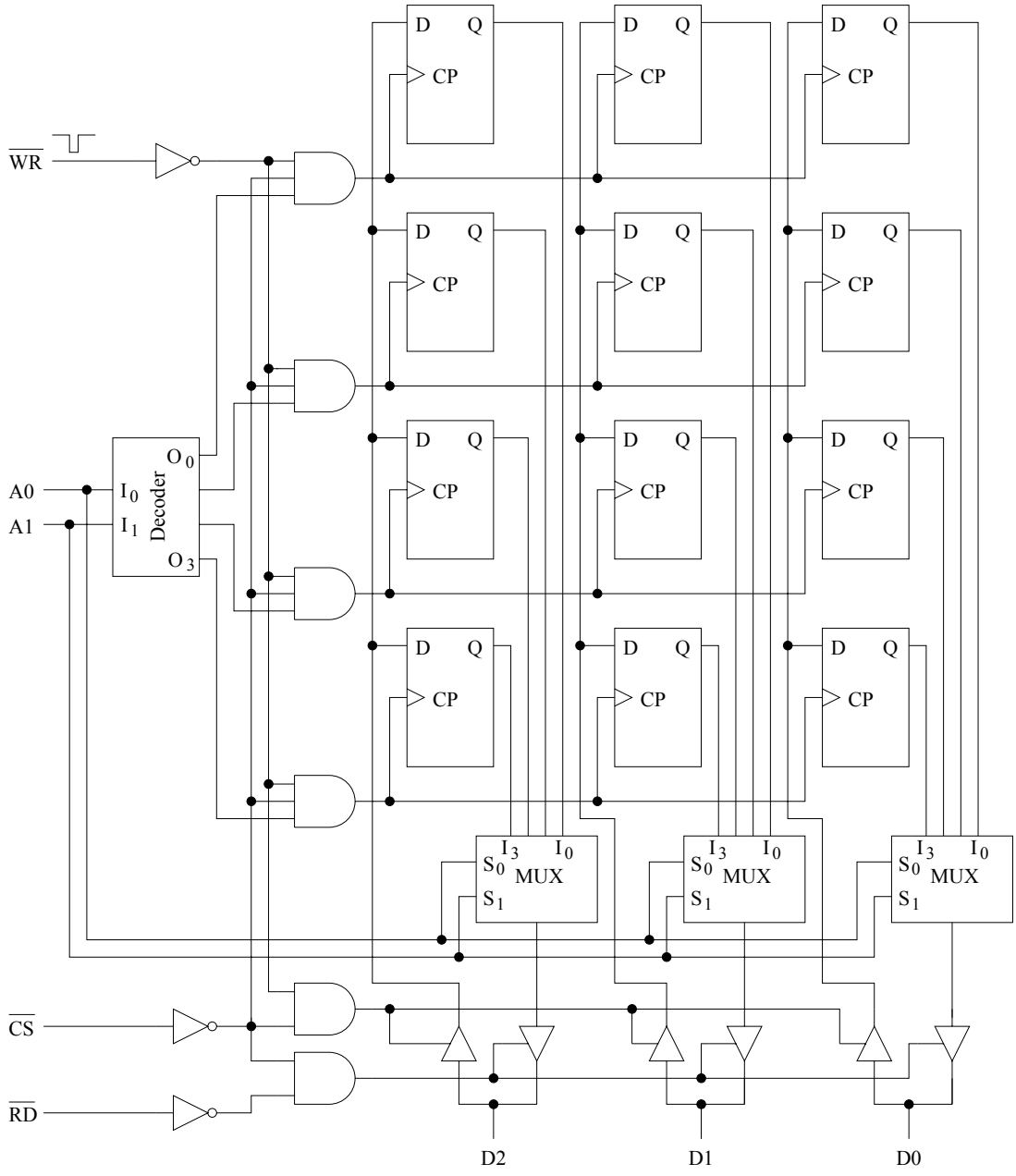


Figure 16.8 A 4 $\times$ 3 memory design using D flip-flops.

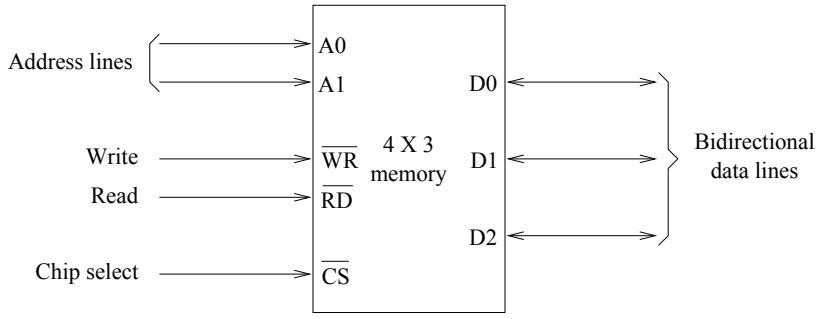


Figure 16.9 Block diagram representation of a 4 × 3 memory.

independent memory units can be mapped into the address space. A more detailed discussion is, however, deferred to the next section that describes full mapping as well as partial mapping of memory units into the address space.

16.5.1 Designing Independent Memory Modules

In this example, we use the 74373 register chip as the basic building block to construct a 2 × 16 memory module. Since each 74373 chip can store eight bits of data, we use four such chips organized as a 2 × 2 array (see Figure 16.10). In each row, the 74373 chip control signals LE and $\overline{\text{OE}}$ are tied together to form a 16-bit word. For example, in the top row, chip#1 receives the upper half of the 16-bit data (D8 to D15) and provides the upper half of the word to the data bus. The other half of the word is taken care of by chip#2.

A word on the notation: To simplify the schematic diagram, it is common practice to use arrows to identify a group of related signals. For example, instead of showing 16 separate lines for the data bus, we use an arrow (a double-headed arrow as the data bus is a bidirectional bus) and identify the group of signals it represents as D0 to D15. Similarly, the input connections of chip#1 are the upper half of the data bus D8 to D15.

By operating two chips in tandem, we double the word length. This technique can be used to build memories with larger word sizes using narrower units or chips. For example, to build a 64-bit word memory, we put eight 74373 chips in each row, with each chip supplying 8 bits of data. This is referred to as "horizontal" expansion, which takes care of the word length.

Next we have to increase the number of words. In our case, we just have to add one more word. Adding words is straightforward as well. To increase the number of words in memory, we simply add more rows, each row being a replication of the first row. This is referred to as "vertical" expansion. In our example, we create a second row exactly like the first one. All we have to do now is to make sure that only one row is selected at any point in time. This is done by the row select logic shown in Figure 16.10. Since our goal is to design an independent memory unit, we have to provide a chip select input. We use the LE input to write data into the selected row. The low-active 1-to-2 decoder does the row selection. In order to predicate both read and

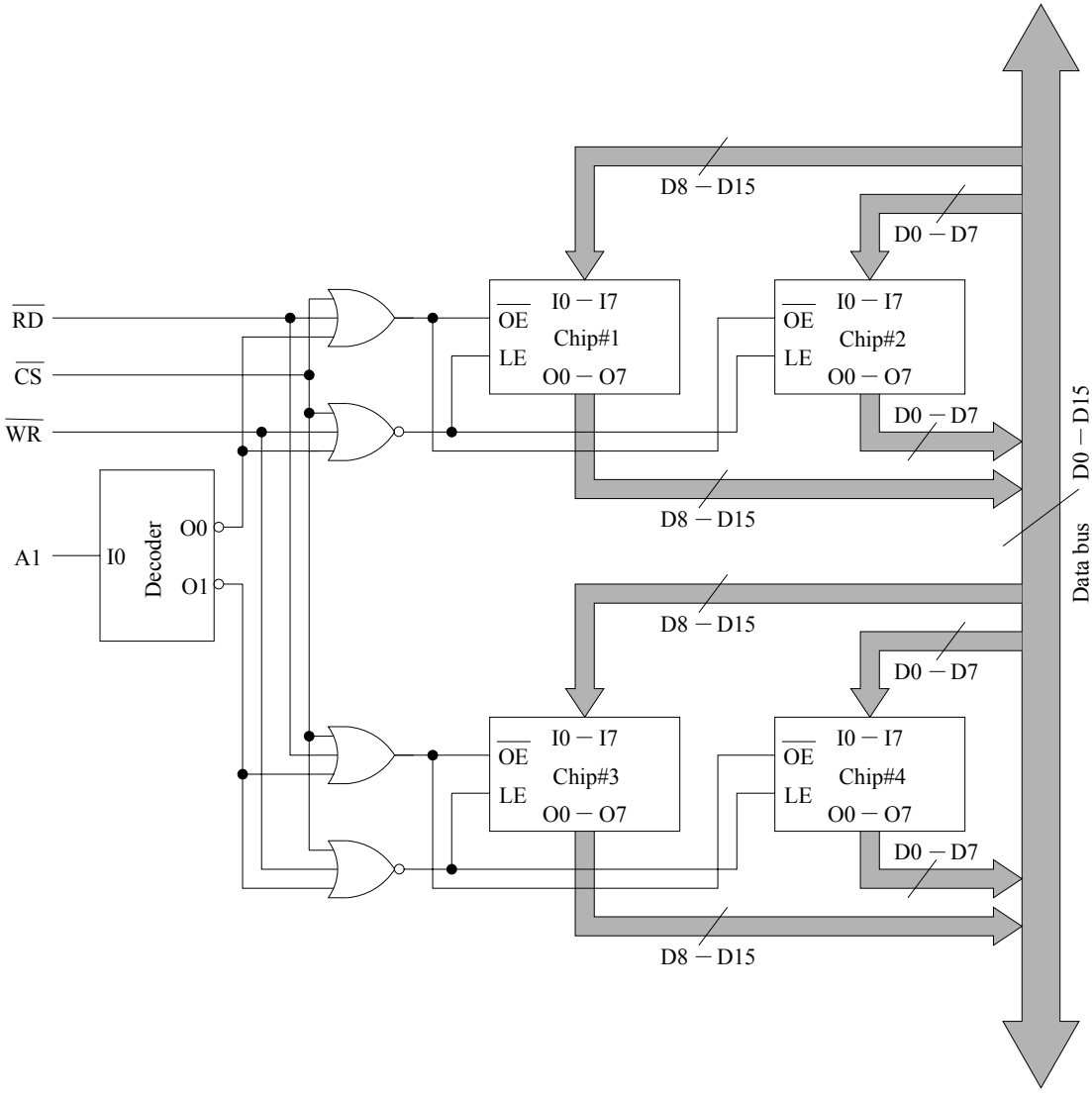


Figure 16.10 A 2 $\times$ 16 memory module using 74373 chips.

write on the active chip select signal, $\overline{\text{CS}}$ is used to gate the read and write signals. The $\overline{\text{OE}}$ line is used to output the data of the selected row.

As you can see from this design, the read operation on a specific row is enabled by the presence of active CS (i.e., $\overline{\text{CS}}$ = 0), selection of the row by the address decoder, and the presence of the $\overline{\text{RD}}$ signal. Similarly, the write operation is also predicated on the presence of chip select and address decoder output.

16.5.2 Designing Larger Memories Using Memory Chips

Before discussing the design procedure, we briefly present details about commercially available memory chips.

Memory Chips

Several commercial memory chips are available to build larger memories. Here we look at two example chips—a SRAM and a DRAM—from Micron Technology.

The SRAM we discuss is an 8-Mb chip that comes in three configurations: 512 K $\times$ 18, 256 K $\times$ 32, or 256 K $\times$ 36. Notice that, in the first and last configurations, word length is not a multiple of 8. These additional bits are useful for error detection/correction. These chips have an access time of 3.5 ns. The 512 K $\times$ 18 chip requires 19 address lines, whereas the 256 K $\times$ 32/36 versions require 18 address lines.

An example DRAM (it is a synchronous DRAM) is the 256-Mb capacity chip that comes in word lengths of 4, 8, or 16 bits. That is, this memory chip comes in three configurations: 64 M $\times$ 4, 32 M $\times$ 8, or 16 M $\times$ 16. The cycle time for this chip is about 7 ns.

In the days when the data bus widths were small (8 or 16), DRAM chips were available in 1-bit widths. Current chips use a word width of more than 1 as it becomes impractical to string 64 1-bit chips to get 64-bit word memories for processors such as the Pentium.

From the details of these 2 example memory chips, we see that the bit capacity of a memory chip can be organized into several configurations. If we focus on the DRAM chip, for example, what are the pros and cons of the various configurations? The advantage of wider memory chips (i.e., chips with larger word size) is that we require fewer of them to build a larger memory. As an example, consider building memory for your Pentium-based PC. Even though the Pentium is a 32-bit processor, it uses a 64-bit wide data bus. Suppose that you want to build a 16 M $\times$ 64 memory. We can build this memory by using four 16 M $\times$ 16 chips, all in a single row. How do we build such a memory using, for example, the 32 M $\times$ 8 version of the chip? Because our word size is 64, we have to use 8 such chips in order to provide 64-bit wide data. That means we get 32 M $\times$ 64 memory as the minimum instead of the required 16 M $\times$ 64. The problem becomes even more serious if we were to use the 64 M $\times$ 4 version chip. We have to use 16 such chips, and we end up with a 64 M $\times$ 64 memory. This example illustrates the tradeoff between using "wider" memories versus "deeper" memories.

Larger Memory Design

Before proceeding with the design of a memory unit, we need to know if the memory address space (MAS) supported by the processor is byte addressable or not. In a byte-addressable space, each address identifies a byte. All popular processors—the Pentium, PowerPC, SPARC, and MIPS—support byte-addressable space. Therefore, in our design examples, we assume byte-addressable space.

We now discuss how one can use memory chips, such as the ones discussed before, to build system memory. The procedure is similar to the intuitive steps followed in the previous design examples.

First we have to decide on the configuration of the memory chip, assuming that we are using the DRAM chip described before. As described in the last section, independent of the configuration, the total bit capacity of a chip remains the same. That means the number of chips required remains the same. For example, if we want build a 64 M $\times$ 32 memory, we need eight chips. We can use eight 64 M $\times$ 4 in a single row, eight 32 M $\times$ 8 in 2 $\times$ 4 array, or 16 M $\times$ 16 in 4 $\times$ 2 array. Although we have several alternatives for this example, there may be situations where the choice is limited. For example, if we are designing a 16 M $\times$ 32 memory, we have no choice but to use the 16 M $\times$ 16 chips.

Once we have decided on the memory chip configuration, it is straightforward to determine the number of chips and the organization of the memory unit. Let us assume that we are using D $\times$ W chips to build an M $\times$ N memory. Of course, we want to make sure that D $\leq$ M and W $\leq$ N.

$$\text{Number of chips required} = \frac{\text{M} \times \text{N}}{\text{D} \times \text{W}},$$

$$\text{Number of rows} = \frac{\text{M}}{\text{D}},$$

$$\text{Number of columns} = \frac{\text{N}}{\text{W}}.$$

The read and write lines of all memory chips should be connected to form a single read and write signal. These signals are connected to the control bus memory read and write lines. For simplicity, we omit these connections in our design diagrams.

Data bus connections are straightforward. Each chip in a row supplies a subset of data bits. In our design, the right chip supplies D0 to D15, and the left chip supplies the remaining 16 data bits (see Figure 16.11).

For each row, connect all chip select inputs as shown in Figure 16.11. Generating appropriate chip select signals is the crucial part of the design process. To complete the design, partition the address lines into three groups as shown in Figure 16.12.

The least significant Z address bits, where Z = $\log_2$(N/8), are not connected to the memory unit. This is because each address going into the memory unit will select an N-bit value. Since we are using byte-addressable memory address space, we can leave the Z least significant bits that identify a byte out of N/8 bytes. In our example, N = 32, which gives us Z = 2. Therefore, the address lines A0 and A1 are not connected to the memory unit.

The next Y address bits, where Y = $\log_2$D, are connected to the address inputs of all the chips. Since we are using 16 M chips, Y = 24. Thus, address lines A2 to A25 are connected to all the chips as shown in Figure 16.11.

The remaining most significant address bits X are used to generate the chip select signals. This group of address bits plays an important role in mapping the memory to a part of the memory address space. We discuss this mapping in detail in the next section. The design shown in Figure 16.11 uses address lines A26 and A27 to generate four chip select signals, one for each row of chips. We are using a low-active 2-to-4 decoder to generate the $\overline{\text{CS}}$ signals.

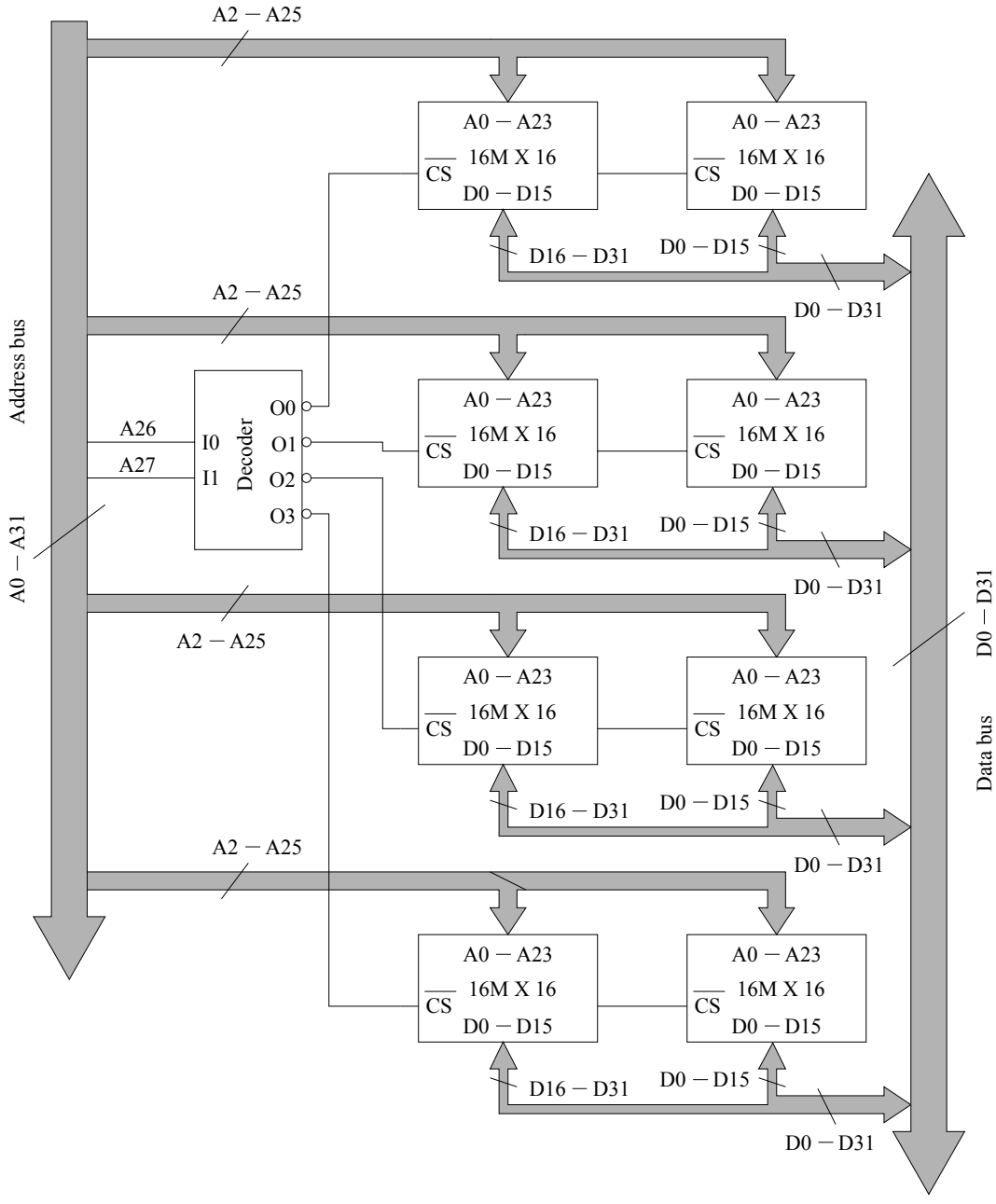


Figure 16.11 Design of a 64 M × 32 memory using 16 M × 16 memory chips.

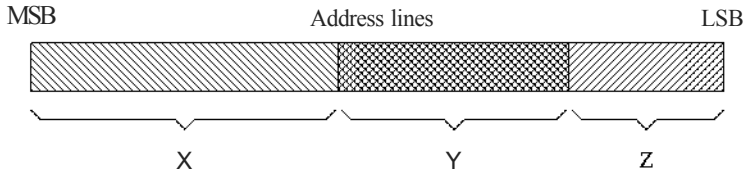


Figure 16.12 Address line partition.

The top row of chips in Figure 16.11 is mapped to the first 64-MB address space (i.e., from addresses 0 to 2^{26} - 1). The second row is mapped to the next 64-MB address space, and so on. After reading the next section, you will realize that this is a partial mapping.

16.6 Mapping Memory

Memory mapping refers to the placement of a memory unit in the memory address space (MAS). For example, the Pentium supports 4 GB of address space (i.e., it uses 32 bits for addressing a byte in memory). If your system has 128 MB of memory, it can be mapped to one of several address subspaces. This section describes how this mapping is done.

16.6.1 Full Mapping

Full mapping refers to a one-to-one mapping function between the memory address and the address in MAS. This means, for each address value in MAS that has a memory location mapped, there is one and only one memory location responding to the address.

Full mapping is done by completely decoding the higher-order X bits of memory (see Figure 16.12) to generate the chip select signals. Two example mappings of 16 M x 32 memory modules are shown in Figure 16.13. Both these mappings are full mappings as all higher-order X bits participate in generating the $\overline{CS}$ signal.

Logically we can divide the 32 address lines into two groups. One group, consisting of address lines Y and Z, locates a byte in the selected 16 M x 32 memory module. The remaining higher-order bits (i.e., the X group) are used to generate the $\overline{CS}$ signal. Given this delineation, it is simple to find the mapping.

We illustrate the technique by using the two examples shown in Figure 16.13. Since the memory modules have a low-active chip select input, a given module is selected if its $\overline{CS}$ input is 0. For Module A, the NAND gate output is low when A26 and A29 are low and the remaining four address lines are high. Thus, this memory module responds to memory read/write activity whenever the higher-order six address bits are 110110. From this, we can get the address locations mapped to this module as D8000000H to DBFFFFFFH. For convenience, we have expressed the addresses in the hexadecimal system (as indicated by the suffix letter H). The address D8000000H is mapped to the first location and the address DBFFFFFFH to the last location of Module A. For addresses that are outside this range, the $\overline{CS}$ input to Module A is high and, therefore, it is deselected.

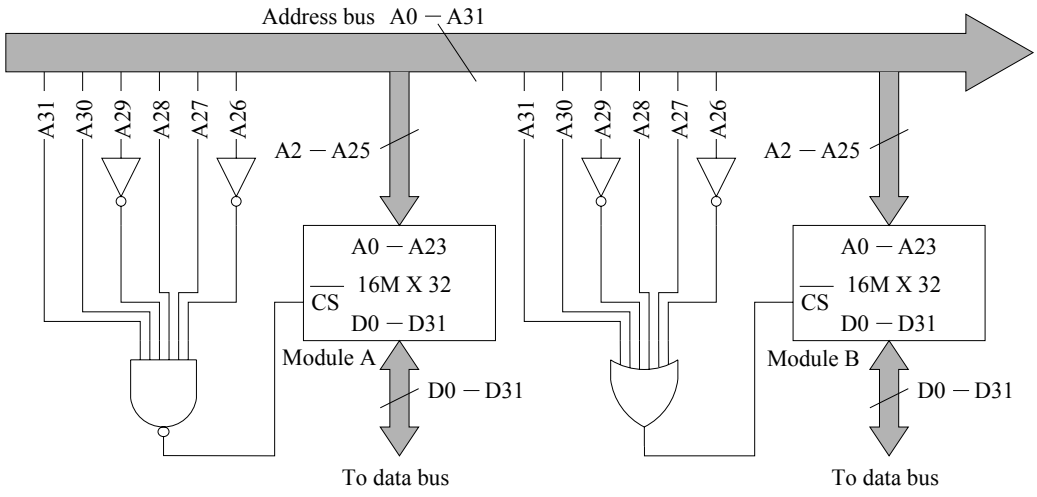


Figure 16.13 Full address mapping.

For Module B, the same inputs are used except that the NAND gate is replaced by an OR gate. Thus, the output of this OR gate is low when the higher-order six address bits are 001001. From this, we can see that mapping for Module B is 24000000H to 27FFFFFH. As these two ranges are mutually exclusive, we can keep both mappings without causing conflict problems.

16.6.2 Partial Mapping

Full mapping is useful in mapping a memory module; however, often the complexity associated with generating the $\overline{\text{CS}}$ signal is not necessary. For example, we needed a 6-input NAND or OR gate to map the two memory modules in Figure 16.13. Partial mapping reduces this complexity by mapping each memory location to more than one address in MAS. We can obtain simplified $\overline{\text{CS}}$ logic if the number of addresses a location is mapped to is a power of 2.

Let us look at the mapping of Module A in Figure 16.14 to clarify some of these points. The $\overline{\text{CS}}$ logic is the same except that we are not connecting the A26 address line to the NAND gate. Because A26 is not participating in generating the signal, it becomes a don't care input. In this mapping, Module A is selected when the higher-order six address bits are 110110 or 110111. Thus, Module A is mapped to the address space D8000000H to DBFFFFFFH and DC000000H to DFFFFFFFH. That is, the first location in Module A responds to addresses D8000000H and DC000000H. Since we have left out one address bit A26, two (i.e., 2^1) addresses are mapped to a memory location. In general, if we leave out k address bits from the chip select logic, we map 2^k addresses to each memory location. For example, in our memory design of Figure 16.11, four address lines (A28 to A31) are not used. Thus, $2^4 = 16$ addresses are mapped to each memory location.

We leave it as an exercise to verify that each location in Module B is mapped to eight addresses as there are three address lines that are not used to generate the $\overline{\text{CS}}$ signal.

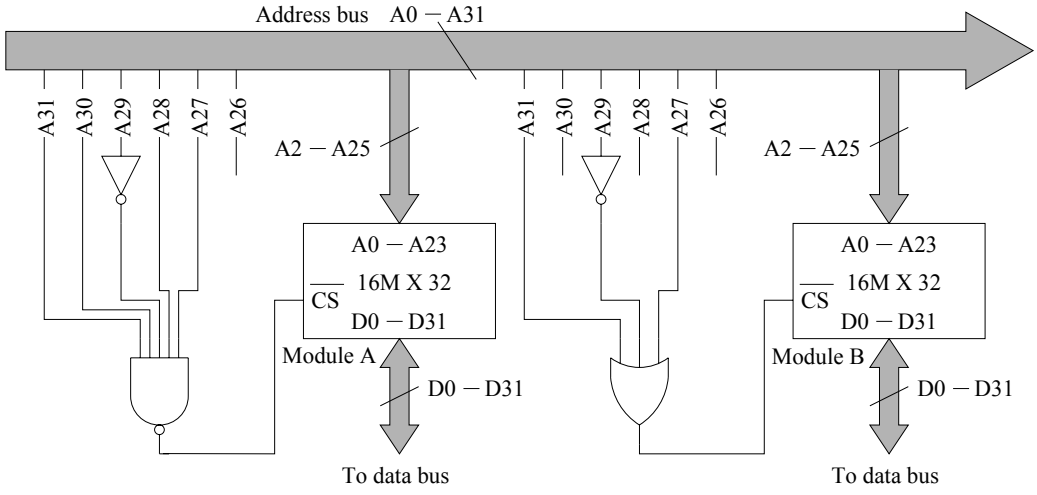


Figure 16.14 Partial address mapping.

16.7 Alignment of Data

We can use our memory example to understand why data alignment improves the performance of computer systems. Suppose we want to read 32-bit data from the memory shown in Figure 16.11. If the address of these 32-bit data is a multiple of four (i.e., address lines A0 and A1 are 0), the 32-bit data are stored in a single row of memory. Thus the processor can get the 32-bit data in one read cycle. If this condition is not satisfied, then the 32-bit data item is spread over two rows. Thus the processor needs to read two 32-bits of data and extract the required 32-bit data. This scenario is clearly demonstrated in Figure 16.15.

In Figure 16.15, the 32-bit data item stored at address 8 (shown by hashed lines) is aligned. Due to this alignment, the processor can read this data item in one read cycle. On the other hand, the data item stored at address 17 (shown shaded) is unaligned. Reading this data item requires two read cycles: one to read the 32 bits at address 16 and the other to read the 32 bits at address 20. The processor can internally assemble the required 32-bit data item from the 64-bit data read from the memory.

You can easily extend this discussion to the Pentium 64-bit data bus. It should be clear to you that aligned data improve system performance.

- *2-Byte Data*: A 16-bit data item is aligned if it is stored at an even address (i.e., addresses that are multiples of two). This means that the least significant bit of the address must be 0.
- *4-Byte Data*: A 32-bit data item is aligned if it is stored at an address that is a multiple of four. This implies that the least significant two bits of the address must be 0 as discussed in the last example.

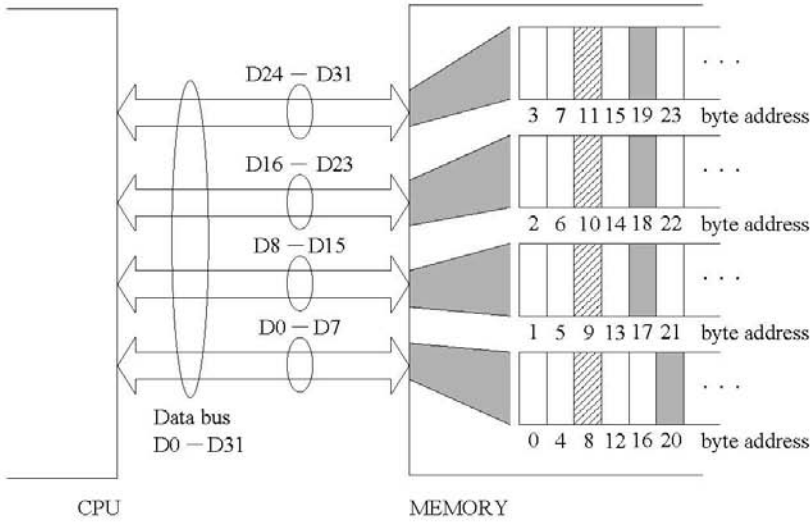


Figure 16.15 Byte-addressable memory interface to the 32-bit data bus.

- *8-Byte Data*: A 64-bit data item is aligned if it is stored at an address that is a multiple of eight. This means that the least significant three bits of the address must be 0. This alignment is important for Pentium processors, as they have a 64-bit wide data bus. On 80486 processors, since their data bus is 32-bits wide, a 64-bit data item is read in two bus cycles and alignment at 4-byte boundaries is sufficient.

The Intel 80X86 family of processors allows aligned and unaligned data items. Of course, unaligned data cause performance degradation. Alignment constraints of this type are referred to as *soft alignment* constraints. Because of the performance penalty associated with unaligned data, some processors, such as the Motorola 68000 and Intel i860, do not allow unaligned data. This alignment constraint is referred to as the *hard alignment* constraint.

16.8 Interleaved Memories

In our memory designs thus far, we have mapped a block of contiguous memory addresses to a memory module. If we want a larger memory, we simply add more memory modules. This modular design is useful in incrementally expanding the memory. This design, however, has one major disadvantage. Since sequential addresses are mapped to the same memory module, we have to wait for the memory access time for each word we read from the memory module. As an example, let's assume that memory access time is four clock cycles. Using our design, we need $8 \times 4 = 32$ cycles to read eight sequential words. Interleaved memories help reduce this access time.

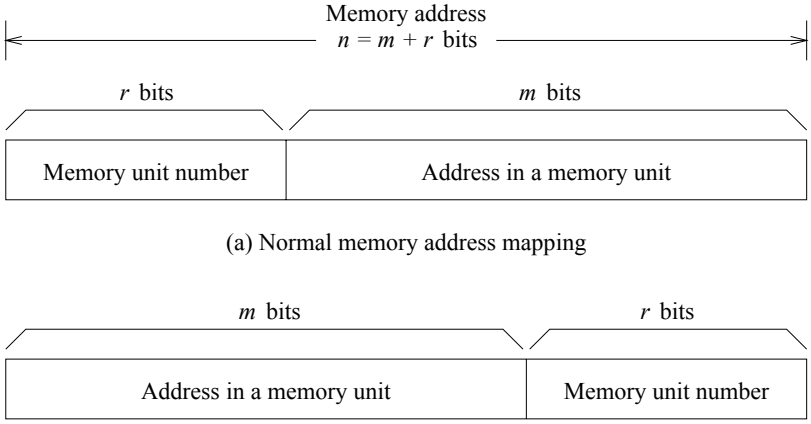


Figure 16.16 Interleaved memories use the lower-order bits to select a memory bank.

16.8.1 The Concept

As we mentioned in Chapter 8, high-performance computers typically use interleaved memories to improve access performance. Essentially, these memories allow overlapped access to hide the memory latency. The key idea is to design the memory system with multiple banks (similar to our memory modules) and access all banks simultaneously so that access time can be overlapped. We explain this in detail next.

In the memory designs we described so far, we divided the n-bit memory address bits into two parts: the higher-order r bits are used to identify a memory module and the lower-order m bits are used to specify a location in the memory module. As shown in Figure 16.16a, $n = r + m$. This technique is sometimes called *high-order interleaving*.

To provide overlapped access to sequential addresses, we have to resort to *low-order interleaving*, as shown in Figure 16.16b. In this design, the lower-order r bits are used to identify a memory module and the higher-order m bits are used to specify a location. We normally use the term interleaved memory to mean low-order interleaving. In these designs, the memory modules are referred to as *memory banks*.

In interleaved memories, memory addresses are mapped on a round-robin basis. Thus, in a B-bank design, address 0 is mapped to bank 0, address 1 to bank 1, and so on. A memory address $addr$ is mapped to memory bank $b = addr$ MOD B. Some example mappings are shown in Table 16.1.

We can implement interleaved memories using two possible designs: synchronized access organization or independent access organization. These two organizations are described next.

Table 16.1 Example mappings

Memory bank number	Memory addresses mapped
Bank 0	$0, B, 2B, \ldots$
Bank 1	$1, B+1, 2B+1, \ldots$
Bank 2	$2, B+2, 2B+2, \ldots$
. . .	. . .
Bank $B-1$	$B-1, 2B-1, 3B-1, \ldots$

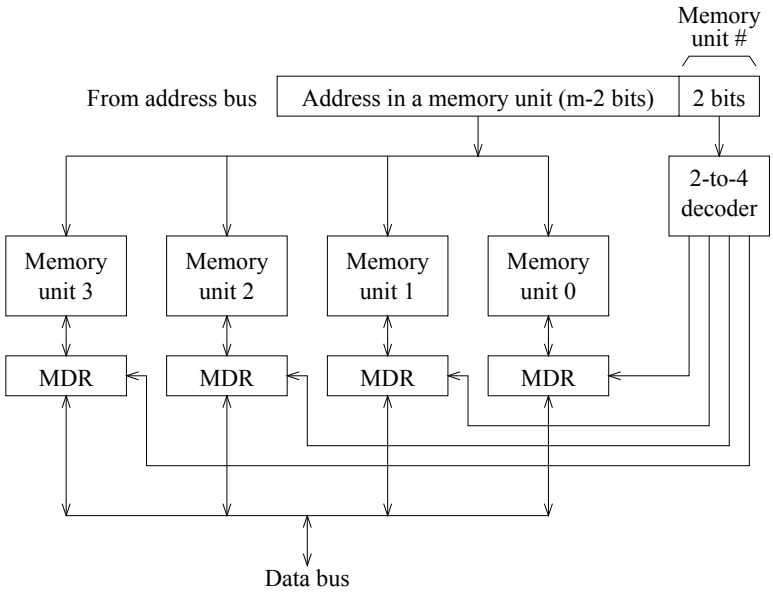
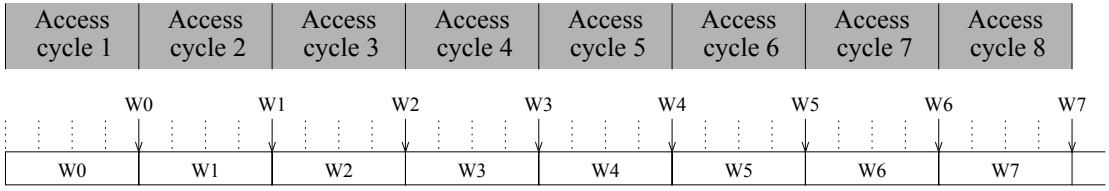


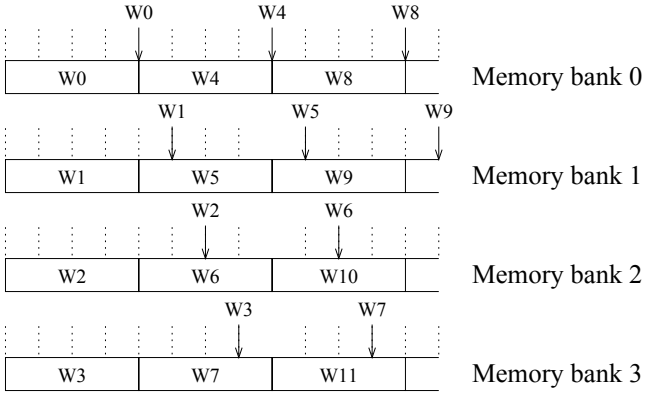
Figure 16.17 Interleaved memory design with synchronized access organization.

16.8.2 Synchronized Access Organization

In this organization, the upper m bits are presented to all memory banks simultaneously. We illustrate the working of this design by means of an example. Figure 16.17 shows this design for a four-bank interleaved memory. As before, it is assumed that the memory access takes four clock cycles. Once the address is presented, all four memory banks initiate their memory access cycles. Thus, in four clock cycles, we will have four data words from the four banks. These data words are latched into four tristate memory data registers (MDRs). We can transfer these four words in four clocks by selecting the appropriate MDR from the lower-order 2-bits



(a) Noninterleaved memory access



(b) Interleaved memory access

Figure 16.18 Interleaved memory allows pipelined access to memory.

using a 2-to-4 decoder (see Figure 16.17). Simultaneously, we can present the next address to the memory banks to initiate the next memory access cycle. Thus, by the time we transfer the four words from the first access cycle, we will have the next four words from the second access cycle. This process is shown in Figure 16.18.

As shown in Figure 16.18*a*, noninterleaved memory access takes four clocks for each access. Interleaved memory, on the other hand, transfers the first word (W0) after four clocks (see Figure 16.18*b*). After that, one word is transferred every clock cycle. Thus, to transfer eight words, we need 12 clock cycles (as opposed to 32 clocks in a noninterleaved design).

16.8.3 Independent Access Organization

A drawback with the synchronized design is that it does not efficiently support access to nonsequential access patterns. The independent access organization allows pipelined access even for arbitrary addresses. In order to support this kind of access, we have to provide each memory bank with a memory address register (MAR) to latch the address that the bank should use (see Figure 16.19). In our example, we can load four different addresses for the four banks. In this design, we do not need the MDR registers to latch the data. Instead, the data are read directly from the memory bank.

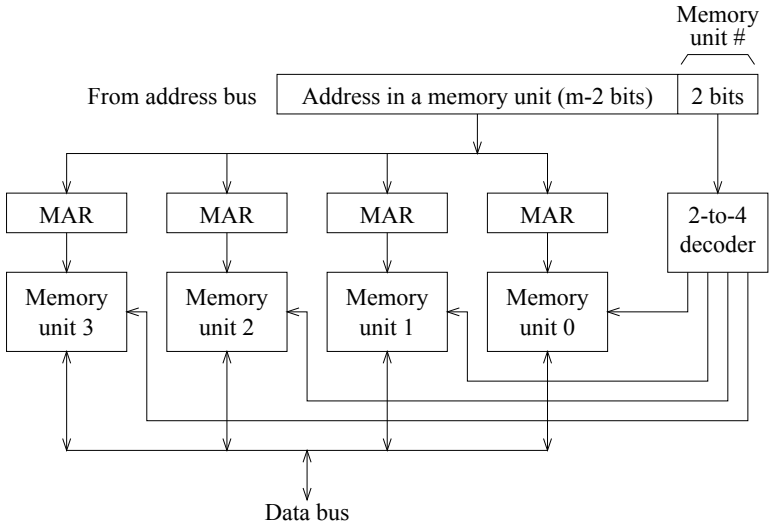


Figure 16.19 Interleaved memory design with independent access organization.

Independent design still provides the same kind of pipelined access as the synchronized access design. To see this, let's trace the first few data transfers from addresses 100, 81, 54, 31, 108, and 121. We can use (address MOD 4) to determine the memory bank number. The first address 100 is sent to bank 0. Once this address is latched in the MAR of bank 0, this bank can initiate the memory access cycle. During the next clock cycle, address 81 is latched into the MAR of bank 1. The next two addresses are sent to banks 2 and 3. After latching the address in the MAR of bank 3, data from bank 0 are available. So the next clock cycle is used to transfer the data from bank 0. We can now load the new address (108) in the MAR register of bank 0. Similarly, during the next clock, data from bank 1 are transferred, and a new address (121) is latched into the MAR of bank 1. This process repeats to provide pipelined access for arbitrary addresses. Of course, this organization also works for the sequential addresses.

16.8.4 Number of Banks

In our examples, we assumed four banks with a memory access time of four cycles. This gives us a data transfer rate of one word per cycle (after the initial startup delay). In general, how many memory banks do we need to provide one word per clock cycle transfer rate if the memory access time is M cycles? From our discussion, it should be clear that the number of banks B should be at least M.

We should caution the reader that interleaved memory does not always provide pipelined access to data. It depends on the memory access pattern. Next we give two examples representing best- and worst-case scenarios.

Example 16.1 *Suppose we have eight memory banks with an access time of six clock cycles. Find the time needed to read 16 words of a vector with stride 1.*

Note that a stride of 1 corresponds to sequential access, assuming that each address supplies a word (see page 308 for information on vector stride). This means we will be accessing the memory sequentially. Thus, whichever design we use, we will get pipelined access. This is the best-case scenario. Since the number of memory banks is at least six, we can get 1 word per cycle data transfer rate. Thus, to complete the transfer of 16 words, we have to wait 6 clocks for the first word and 16 additional clocks to transfer the 16 words. Therefore, we need $6 + 16 = 22$ clock cycles. □

Example 16.2 *Consider the memory system in the previous example. Find the time needed to read 16 words of a vector with stride 8.*

Since the stride is equal to the number of memory banks, we know that all 16 word addresses are mapped to the same bank. In this case, interleaving is not useful at all. We have to access each word sequentially from the same bank. Thus, we need $16 \times 6 = 96$ clock cycles. This is the worst-case scenario. □

16.8.5 Drawbacks

Why do we see interleaved memories only in high-performance computers? Why not in our PCs? One reason is that interleaved memories involve complex design. We have seen some of this complexity in our examples. We need extra registers (MAR or MDR) to latch addresses or data. What we have not shown in these examples is the control circuit needed to transfer data and to load addresses.

Another major reason is that interleaved memories do not have good fault-tolerance properties. That is, if one bank fails, we lose the entire memory. On the other hand, in traditional designs, failure of a module disables only a specific area of the memory address space mapped to the failed module. Furthermore, interleaved memories do not lend themselves well to incremental memory expansion.

16.9 Summary

We have discussed the basic memory design issues. We have shown how flip-flops and latches can be used to build memory blocks. Interfacing a memory unit to the system bus typically requires tristate buffers. Open collector devices and multiplexers are useful only in some special cases.

Building larger memories requires both horizontal and vertical expansion. Horizontal expansion is used to expand the word size, and vertical expansion provides an increased number of words. We have shown how one can design memory modules using standard memory chips. In all these designs, chip select plays an important role in allowing multiple entities to be attached to the system bus.

Chip select logic also plays an important role in mapping memory modules into the address space. Two basic mapping functions are used: full mapping and partial mapping. Full mapping provides a one-to-one mapping between memory locations and addresses. Partial mapping maps each memory location to a number of addresses equal to a power of 2. The main advantage of partial mapping is that it simplifies the chip select logic.

We have discussed the importance of data alignment. Unaligned data can lead to performance degradation. We have discussed the reasons for improvement in performance due to alignment of data.

In the last section, we presented details on interleaved memories, which are typically used in vector processors to facilitate pipelined access to memory.

We have provided only the basic information in designing memory modules. One aspect that we have not looked at is the interface details to handle slow memories. It is important to look at this issue as the speed gap between memories and processors keeps increasing. If a memory unit cannot respond at the rate required by the processor, the memory unit should be able to ask the processor to wait. This is typically done by the memory signaling the processor that it has not completed the operation (memory read or write). For example, the Pentium has a pin labeled READY that any device can pull down to make the processor wait. Thus, slow memories can use the READY input to slow the processor down to the speed of memory. We have discussed these issues in Chapter 5.

Key Terms and Concepts

Here is a list of the key terms and concepts presented in this chapter. This list can be used to test your understanding of the material presented in the chapter. The Index at the back of the book gives the reference page numbers for these terms and concepts:

- Building larger memories
- Chip select
- Data alignment
- Data alignment—hard alignment
- Data alignment—soft alignment
- Interleaved memories
- Interleaved memories—independent access organization
- Interleaved memories—synchronized access organization
- Memory address space
- Memory design with D flip-flops
- Memory mapping—full mapping
- Memory mapping—partial mapping
- Nonvolatile memories
- Open collector outputs
- Tristate buffers

16.10 Exercises

16–1 What is the purpose of the multiplexers in Figure 16.1?

16–2 What is the reason for not being able to connect the data in and out lines in Figure 16.1?

16–3 Suppose that we want to extend the memory block in Figure 16.1 to 4×8. Do you need to change the decoder? How many more multiplexers do you need?

16–4 Suppose that we want to extend the memory block in Figure 16.1 to 16×8. What kind of decoder do you use in your design? What kind of multiplexers do we need?

16–5 Figure 16.8 uses separate read and write lines. Modify this design to use a single line for both read and write operations.

16–6 What are the advantages of tristate buffers over open collector buffers?

16–7 Figure 16.10 shows a 2×16 memory module using four 74373 chips. Suppose that we want to build a 4×16 memory using the same chips. Extend the design in Figure 16.10 to derive the new design.

16–8 We have partitioned the address lines into three groups: X, Y, and Z (see Figure 16.12). Explain the purpose of each group of address bits.

16–9 What are the advantages and disadvantages of full mapping over partial mapping?

16–10 Figure 16.11 shows a 64 M $\times$ 32 memory design. We have stated that this memory block is partially mapped. Give the number of addresses to which each memory location of this block is mapped. What are the addresses of the first location?

16–11 Suppose that we have replaced the 6-input NAND gate in Figure 16.13 by a 6-input AND gate. Give the address mapping of Module A. It is sufficient to give the mapping of the first and last location.

16–12 Suppose that we have replaced the 6-input OR gate in Figure 16.13 by a 6-input NOR gate. Give the address mapping of Module B. It is sufficient to give the mapping of the first and last location.

16–13 We have stated that each address of Module B in Figure 16.14 is mapped to eight addresses. Give the eight addresses to which the first location of Module B is mapped. What are the eight addresses to which the last location is mapped?

16–14 Assume a hypothetical processor with a memory address space of 256 bytes. Design a 16×8 memory system that is mapped to locations 160 to 175 (decimal) using 4×4 memory chips.

16–15 Modify your memory design of the last exercise to partially map to locations 160 to 175 (decimal) and 224 to 239 (decimal). You just need to show the changes to your chip select logic.

16–16 Consider a hypothetical system that supports an address space of 128 bytes. Its data bus width is 8 bits. Design a 16×8 memory block using 4×4 memory chips and map it to address locations 48 to 63. You need to show all the details including the chip select logic needed to map the memory. You can use block diagram notation for memory chips. Note that you can use only 2-input AND gates and inverters to implement your chip select logic.

16–17 How would you change your design in Exercise 16–16 if the mapping were to locations 80 to 95 instead? You need to show only the changes. There is no need to reproduce the part of the circuit that is not changing. Note that you can use only 2-input AND gates and inverters to implement your chip select logic.

16–18 This exercise also refers to the memory block you designed in Exercise 16–16. Show how a single block of 16×8 memory can be mapped to locations 80 to 95 and 112 to 127. In other words, the first location of the 16×8 memory block should respond to addresses 80 and 112, the second location to addresses 81 and 113, and so on. Note that you can use only 2-input AND gates and inverters to implement your chip select logic.

16–19 Consider a hypothetical machine that can address 256 bytes of memory. It uses a 16-bit wide data bus. Suppose that only 32×8 memory chips are available. Using these memory chips, design a 64×16 memory system starting at address 128 (the address is expressed in decimal). You need to show how the address, data, and chip select lines of the chips are connected to the system address and data buses. You don't have to show the read/write connections to the control bus. Be sure to show the necessary chip select logic using only 2-input gates (AND, OR) and inverters. Assume that the chip select is low-active.

16–20 We have mentioned that aligned data improve performance. Discuss the reasons for this improvement.

16–21 Suppose that the data bus width is 32 bits. From the performance viewpoint, does it make sense to talk about 64-bit aligned data? Explain.

16–22 What are the advantages and disadvantages of interleaved memories?

16–23 What are the advantages and disadvantages of synchronized and independent access interleaved memory designs?

16–24 Let B be the number of banks and M be the memory access time in cycles. Explain why we need to have $B \geq M$ to support a one-word-per-cycle data-transfer rate.

16–25 Suppose we have an interleaved memory with eight banks. As in our examples on page 689, the access time is six clock cycles. Find the time needed to access data at the following 10 addresses: 152, 153, 154, 155, 156, 157, 158, 159, 160, and 161.

16–26 Suppose we have an interleaved memory as in the last exercise. Find the time needed to access data at the following 10 addresses: 152, 320, 868, 176, 184, 800, 880, 640, 560, and 160.

16–27 In the last exercise, does it matter whether you use the synchronized access design or the independent access design?