

Chapter 2

Basic Computer Organization

Objectives

- To provide a high-level view of computer organization
- To describe processor organization details
- To discuss memory organization and structure
- To introduce how input/output devices are interfaced
- To illustrate the importance of data alignment

Programming in a high-level language does not require a detailed knowledge of the system hardware. Assembly language programmers, however, should have some basic understanding of the underlying system architecture. A high-level view of computer systems, presented in Section 2.1, consists of three major components: a processor, a memory unit, and input/output devices.

The next three sections discuss these three components in detail. Section 2.2 discusses the basic processor execution cycle. The following two sections look at the number of addresses used in the instructions and how the control flow is altered. Section 2.5 presents some basic concepts about the memory system. Section 2.6 gives a brief overview of how input/output devices are interfaced to the system.

Section 2.7 discusses how data alignment affects execution time of programs. We use the bubble sort example discussed in Chapter 5 to illustrate the impact of data alignment. Section 2.8 concludes the chapter with a summary.

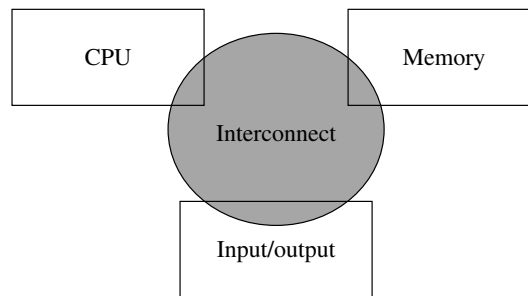


Figure 2.1 High-level view of a computer system.

2.1 Basic Components of a Computer System

A computer system has three main components: a central processing unit (CPU) or processor, a memory unit, and input/output (I/O) devices (see Figure 2.1). These three components are interconnected by a *system bus*. The term “bus” is used to represent a group of electrical signals or the wires that carry these signals. Figure 2.2 shows details of how they are interconnected and what actually constitutes the system bus. As shown in this figure, the three major components of the system bus are the address bus, data bus, and control bus.

The width of the address bus determines the memory addressing capacity of the processor. The width of the data bus indicates the size of the data transferred between the processor and memory or I/O device. For example, the 8086 processor has a 20-bit address bus and a 16-bit data bus. The amount of physical memory that this processor can address is 2^{20} bytes, or 1 MB, and each data transfer involves 16 bits. The Pentium, on the other hand, has 32 address lines and 64 data lines. Thus, the Pentium can address up to 2^{32} bytes, or a 4-GB memory. Furthermore, each data transfer can move 64 bits. In comparison to the Pentium, Intel’s 64-bit processor Itanium uses 64 address lines and 128 data lines.

The control bus consists of a set of control signals. Typical control signals include memory read, memory write, I/O read, I/O write, interrupt, interrupt acknowledge, bus request, and bus grant. These control signals indicate the type of action taking place on the system bus. For example, when the processor is writing data into the memory, the memory write signal is asserted. Similarly, when the processor is reading from an I/O device, the I/O read signal is asserted.

The system memory, also called *main memory* or *primary memory*, is used to store both program instructions and data. I/O devices such as the keyboard and display are used to provide user interface. I/O devices are also used to interface with secondary storage devices such as disks.

The system bus is the communication medium for data transfers. Such data transfers are called the *bus transactions*. Some examples of bus transactions are memory read, memory write, I/O read, I/O write, and interrupt. Depending on the processor and the type of bus used,

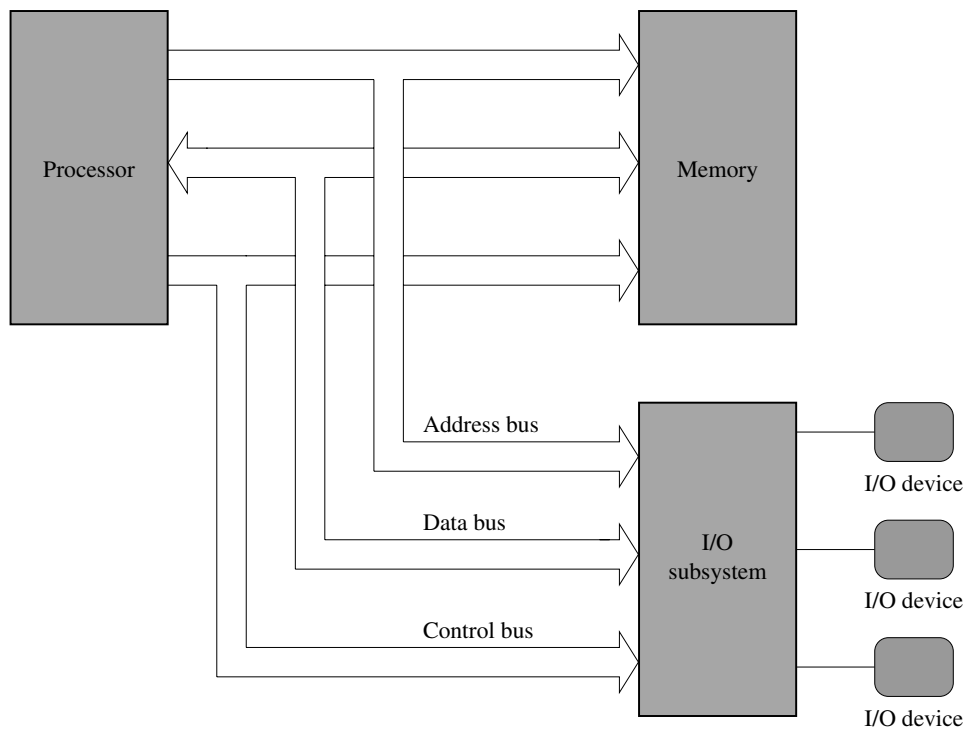


Figure 2.2 Simplified block diagram of a computer system.

there may be other types of transactions. For example, Pentium supports a burst mode of data transfer in which up to four 64 bits of data can be transferred in a burst cycle.

Every bus transaction involves a *master* and a *slave*. The master is the initiator of the transaction and the slave is the target of the transaction. For example, when the processor wants to read data from the memory, it initiates a bus transaction, also called a *bus cycle*, in which the processor is the bus master and memory is the slave. The processor usually acts as the master of the system bus, while components like memory are usually slaves. Some components may act as slaves for some transactions and as masters for other transactions.

When there is more than one master device, which is typically the case, the device requesting the use of the bus sends a *bus request* signal to the bus arbiter using the bus request control line. If the bus arbiter grants the request, it notifies the requesting device by sending a signal on the *bus grant* control line. The granted device, which acts as the master, can then use the bus for data transfer. The bus-request-grant procedure is called the *bus protocol*. Different buses use different bus protocols. In some protocols, permission to use the bus is granted for only one bus cycle; in others, permission is granted until the bus master relinquishes the bus.

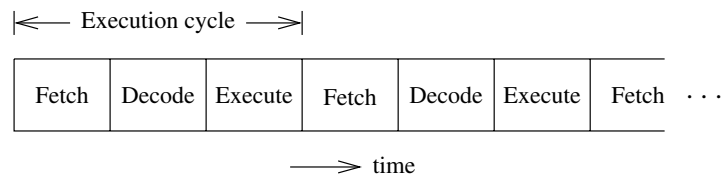


Figure 2.3 Execution cycle of a typical computer system.

2.2 The Processor

The processor acts as the controller of all actions or services provided by the system. It can be thought of as executing the following cycle forever:

1. Fetch an instruction from the memory;
2. Decode the instruction (i.e., identify the instruction);
3. Execute the instruction (i.e., perform the action specified by the instruction).

This process is often referred to as the *fetch-decode-execute* cycle, or simply the *execution* cycle.

This description raises several questions. Who provides the instructions to the processor? Who places these instructions in the main memory? How does the processor know where these instructions are located in the main memory?

When we write programs—whether in a high-level language or in an assembly language—we are providing a sequence of instructions to perform a particular task (i.e., solving a problem). These instructions are translated by a compiler/assembler to an equivalent sequence of machine language instructions that the processor understands.

The operating system, which provides instructions to the processor whenever a user program is not executing, loads the user program into the main memory. The operating system then indicates the location of the user program to the processor and instructs it to execute the program.

2.2.1 The Execution Cycle

The execution cycle of a processor is shown in Figure 2.3. *Fetching* an instruction from the main memory involves placing the appropriate address on the address bus and activating the memory read signal on the control bus to indicate to the memory unit that an instruction should be read from that location. The memory unit requires time to read the instruction at the addressed location. This time is called the *access time*. The memory then places the instruction on the data bus. The processor, after instructing the memory unit to read, waits until the instruction is available on the data bus and then reads the instruction.

Decoding involves identifying the instruction that has been fetched from the memory. To facilitate the decoding process, machine language instructions follow a particular instruction-encoding scheme.

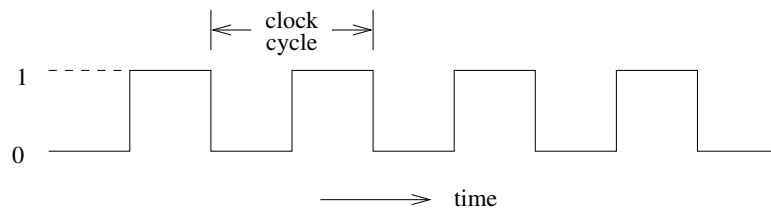


Figure 2.4 Clock signal of a computer system.

To *execute* an instruction, the processor contains hardware consisting of control circuitry and an arithmetic and logic unit (ALU). The control circuitry is needed to provide timing controls as well as to instruct the internal hardware components to perform a specific operation. The ALU is mainly responsible for performing arithmetic operations (such as *add*, *divide*) and logical operations (such as *and*, *or*) on data.

In practice, instructions and data are not fetched, most of the time, from the main memory. There is a high-speed cache memory that provides faster access to instructions and data than the main memory. For example, the Pentium provides a 16 KB on-chip cache. This is divided equally into data cache and instruction cache. The presence of the on-chip cache is transparent to application programs—it helps improve application performance.

2.2.2 The System Clock

The system clock provides a timing signal to synchronize the operations of the system. A clock is a sequence of 1's and 0's, as shown in Figure 2.4. The clock frequency is measured in the number of cycles per second. This number is referred to as Hertz (Hz). We often use the abbreviations MHz and GHz to represent 10^6 and 10^9 cycles per second, respectively.

The system clock defines the *speed* at which the system operates. All processor operations take multiple clock cycles. For example, transfer of data from a memory location to Pentium takes three clock cycles. Thus, the higher the clock rate, the faster the system can work.

The clock period is defined as the length of time taken by one *clock cycle*.

$$\text{Clock period} = \frac{1}{\text{Clock frequency}}$$

For example, a clock frequency of 1 GHz yields a clock period of

$$\frac{1}{1 \times 10^9} = 1 \text{ ns}$$

If it takes three clock cycles to execute an instruction, it takes $3 \times 1 \text{ ns} = 3 \text{ ns}$.

One way to increase the speed of a computer system is to use a higher clock frequency. For example, if we use a clock of 2 GHz, the instruction execution time reduces from 3 ns to 1.5 ns. Clock frequency increases with improvements in technology. The original IBM PC used a clock of 4.77 MHz. Current technology allows clock frequencies higher than 3 GHz.

Table 2.1 Sample Three-Address Machine Instructions

Instruction	Semantics
add dest, src1, src2	Adds the two values at src1 and src2 and stores the result in dest
sub dest, src1, src2	Subtracts the second source operand at src2 from the first at src1 and stores the result in dest
mult dest, src1, src2	Multiplies the two values at src1 and src2 and stores the result in dest

2.3 Number of Addresses

One of the characteristics that shapes the architecture of a processor is the number of addresses used in its instructions. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

Most processors use either two or three addresses. For example, the MIPS processor uses three addresses whereas the Pentium uses two addresses. However, it is possible to design systems with one or even zero address. In the rest of this section, we give details on these machines.

2.3.1 Three-Address Machines

In three-address machines, instructions carry all three addresses explicitly. Most current processors use three addresses. The MIPS processor we discuss in Chapter 12, for example, uses three addresses. Table 2.1 gives some sample instructions of a three-address machine.

On these machines, the C statement

$$A = B + C * D - E + F + A$$

is converted to the following code:

```

mult  T,C,D      ; T = C*D
add   T,T,B      ; T = B + C*D
sub   T,T,E      ; T = B + C*D - E
add   T,T,F      ; T = B + C*D - E + F

```

Table 2.2 Sample Two-Address Machine Instructions

Instruction	Semantics
load dest,src	Copies the value at <code>src</code> to <code>dest</code>
add dest,src	Adds the two values at <code>src</code> and <code>dest</code> and stores the result in <code>dest</code>
sub dest,src	Subtracts the second source operand at <code>src</code> from the first at <code>dest</code> and stores the result in <code>dest</code>
mult dest,src	Multiplies the two values at <code>src</code> and <code>dest</code> and stores the result in <code>dest</code>

```
add    A,A,T        ; A = B + C*D - E + F + A
```

As you can see from this code, there is one instruction for each arithmetic operation. Also notice that all instructions, barring the first one, use an address twice. In the middle three instructions, it is the temporary `T`, and in the last one, it is `A`. This is the motivation for using two addresses, as we show next.

2.3.2 Two-Address Machines

In two-address machines, one address doubles as a source and destination. Usually, we use `dest` to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. We discuss the Pentium processor details in the next few chapters. Table 2.2 gives some sample instructions of a two-address machine.

On these machines, the C statement

```
A = B + C * D - E + F + A
```

is converted to the following code:

```
load  T,C        ; T = C
mult  T,D        ; T = C*D
add   T,B        ; T = B + C*D
sub   T,E        ; T = B + C*D - E
add   T,F        ; T = B + C*D - E + F
add   A,T        ; A = B + C*D - E + F + A
```

Since we use only two addresses, we use a load instruction to first copy the `C` value into a temporary address represented by `T`. If you look at these six instructions, you will notice that the operand `T` is common. If we make this our default, then we don't need even two addresses: we can get away with just one address.

2.3.3 One-Address Machines

In the early machines, when memory was expensive and slow, a special set of registers was used to provide one of the input operands as well as to receive the result of the operation. Because of this, these registers are called the *accumulators*. In most machines, there is just a single accumulator register. This kind of design, called the *accumulator machines*, makes sense if memory is expensive.

In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no need to store the result in memory: this reduces the need for larger memory and speeds up the computation by reducing the number of memory accesses.

2.3.4 Zero-Address Machines

In zero-address machines, the locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in–first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria. We discuss the stack later in this book (see Section 5.1 on page 118).

All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack.

2.3.5 The Load/Store Architecture

In this architecture, instructions operate on values stored in internal processor registers. Only load and store instructions move data between the registers and memory. Table 2.3 gives some sample instructions for the load/store machines. On these machines, the C statement

$$A = B + C * D - E + F + A$$

is converted to the following code:

```

load  R1,B      ; load B
load  R2,C      ; load C
load  R3,D      ; load D
load  R4,E      ; load E
load  R5,F      ; load F
load  R6,A      ; load A
mult  R2,R2,R3  ; R2 = C*D
add   R2,R2,R1  ; R2 = B + C*D
sub   R2,R2,R4  ; R2 = B + C*D - E
add   R2,R2,R5  ; R2 = B + C*D - E + F
add   R2,R2,R6  ; R2 = B + C*D - E + F + A
store A,R2      ; store the result in A
```


Table 2.3 Sample Load/Store Machine Instructions

Instruction	Semantics
load <i>Rd, addr</i>	Loads the <i>Rd</i> register with the value at address <i>addr</i>
store <i>addr, Rs</i>	Stores the value in <i>Rs</i> register at address <i>addr</i>
add <i>Rd, Rs1, Rs2</i>	Adds the two values in <i>Rs1</i> and <i>Rs2</i> registers and places the result in <i>Rd</i> register
sub <i>Rd, Rs1, Rs2</i>	Subtracts the value in <i>Rs2</i> from that in <i>Rs1</i> and places the result in <i>Rd</i> register
mult <i>Rd, Rs1, Rs2</i>	Multiplies the two values in <i>Rs1</i> and <i>Rs2</i> and places the result in <i>Rd</i> register

In this code, we assume that we have six registers to load the values. However, you don't need this many registers. For example, once the value in *R3* is used, we can reuse this register. Typically, RISC processors tend to have many more registers than CISC processors. For example, the MIPS processor has 32 registers and the Intel Itanium processor has 128 registers. Compared to this, the Pentium has only 10 registers.

RISC machines as well as vector processors use this architecture, which reduces the instruction size substantially. If we assume that memory addresses are 32 bits long, an instruction with all three operands in memory requires 104 bits whereas the register-based operands require instructions to be 23 bits, as shown in Figure 2.5. In this figure, we use 5 bits to specify a register as we assume that there are 32 registers as in the MIPS processors. MIPS processor details are given in Chapters 12 and 13.

2.3.6 Processor Registers

Processors have a number of registers to hold data, instructions, and state information. We can classify the processors based on the structure of these registers and how the processor uses them. Typically, we can divide the registers into general-purpose or special-purpose registers. Special-purpose registers can be further divided into those that are accessible to the user programs and those reserved for the system use. The available technology largely determines the structure and function of the register set.

The number of addresses used in instructions partly influences the number of data registers and their use. For example, in three- and two-address machines, there is no need for the internal data registers. However, having a few internal registers improves performance by cutting down the number of memory accesses required to execute a program. RISC processors typically have a large number of registers.

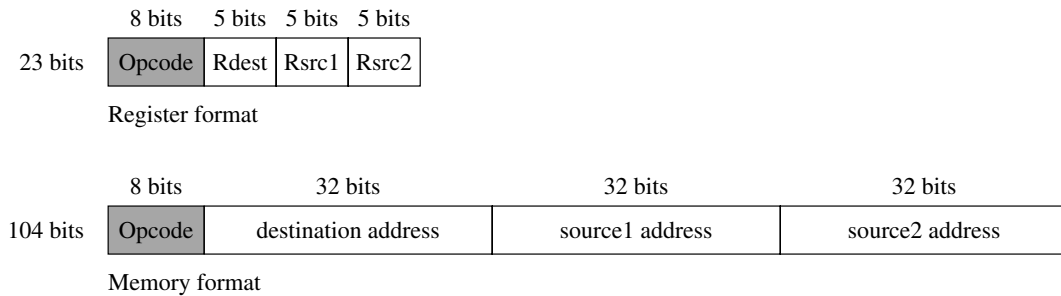


Figure 2.5 A comparison of the instruction size when the operands are in registers versus memory.

Some processors maintain a few special-purpose registers. For example, the Pentium uses a couple of registers to implement the processor stack. Processors also have several registers reserved for the instruction execution unit. Typically, there is an instruction register that holds the current instruction and a program counter that points to the next instruction to be executed.

Registers available in the Pentium processor are described in the next chapter. MIPS processor registers are discussed in Chapter 12.

2.4 Flow of Control

Program execution, by default, proceeds sequentially. This default behavior is due to the semantics associated with the execution cycle described in Section 2.2.1. The program counter (PC) register plays an important role in managing the control flow. At a simple level, the PC can be thought of as pointing to the next instruction. The processor fetches the instruction at the address pointed to by the PC. When an instruction is fetched, the PC is incremented to point to the next instruction. If we assume that each instruction takes exactly four bytes as in the MIPS processors, the PC is automatically incremented by four after each instruction fetch. This leads to the default sequential execution pattern.

However, sometimes we want to alter this default execution flow. In high-level languages, we use control structures such as `if-then-else` and `while` statements to alter the execution behavior based on some run-time conditions. Similarly, we can use procedure calls to alter the sequential execution. In this section, we describe how processors support flow control. We look at both branch and procedure calls next. Interrupt is another mechanism to alter flow control, which is discussed in Chapter 14.

2.4.1 Branching

Branching is implemented by means of a branch instruction. This instruction carries the address of the target instruction explicitly. Branch instructions in processors such as the Pentium are also called the jump instructions. Processors support two types of branches: uncondi-

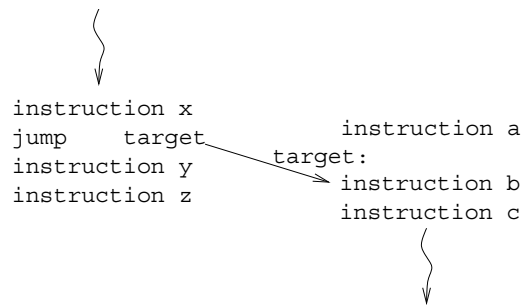


Figure 2.6 Control flow in branching.

tional and conditional. In both cases, the transfer control mechanism remains the same (see Figure 2.6).

Unconditional Branch

The simplest of the branch instructions is the *unconditional branch*, which transfers control to the specified target. Here is an example branch instruction:

```
branch    target
```

Specification of the target address can be done in one of two ways: absolute address or PC-relative address. In the former, the actual address of the target instruction is given. In the PC-relative method, the target address is specified relative to the PC contents. Most processors support absolute address for unconditional branches. Others support both formats. For example, MIPS processors support absolute address-based branch by

```
j        target
```

and PC-relative unconditional branch by

```
b        target
```

In fact, the last instruction is an assembly language instruction. The processor only supports the *j* instruction.

If the absolute address is used, the processor transfers control by simply loading the specified target address into the PC register. If PC-relative addressing is used, the specified target address is added to the PC contents, and the result is placed in the PC. In either case, since the PC indicates the next instruction address, the processor will fetch the instruction at the intended target address.

The main advantage of using the PC-relative address is that we can move the code from one block of memory to another without changing the target addresses. This type of code is called *relocatable code*. Relocatable code is not possible with absolute addresses.

Conditional Branch

In conditional branches, the jump is taken only if a specified condition is satisfied. For example, we may want to take a branch only if two values are equal. Such conditional branches are handled in one of two basic ways:

- *Set-Then-Jump*: In this design, testing for the condition and branching are separated. To achieve communication between these two instructions, a condition code register is used. The Pentium follows this design, which uses a flags register to record the result of the test condition. It uses a compare (`cmp`) instruction to test the condition. This instruction sets the various flag bits to indicate the relationship between the two compared values. Then we can use a conditional jump instruction to jump to the target location if the specified condition bit is set. The Pentium jump instructions are discussed in detail in Part II.
- *Test-and-Jump*: Most processors combine the testing and branching into a single instruction. We use the MIPS processor to illustrate the principle involved in this strategy. The MIPS processor provides several branch instructions that test and branch (for a quick peek, see Table 13.7 on page 374). The one that we are interested in here is the branch on equal instruction shown below:

```
beq    Rsrc1, Rsrc2, target
```

This conditional branch instruction tests the contents of the two registers `Rsrc1` and `Rsrc2` for equality and transfers control to `target` if equal. More details on the MIPS processor branch instructions are given in Chapter 13.

Some processors maintain registers to record the condition of the arithmetic and logical operations. These are called *condition code registers*. These registers keep a record of the status of the last arithmetic/logical operation. For example, when we add two 32-bit integers, it is possible that the sum might require more than 32 bits. This is the overflow condition that the system should record. Normally, a bit in the condition code register is set to indicate this overflow condition. The MIPS processors, for example, do not use condition registers. Instead, it uses exceptions to flag the overflow condition. On the other hand, the Pentium uses condition registers, which are called the flags register.

Some instruction sets provide branches based on comparisons to zero. Some example processors that provide this type of branch instructions include the MIPS processors.

2.4.2 Procedure Calls

The use of procedures facilitates modular programming. Procedure calls are slightly different from the branches. Branches are one-way jumps: once the control has been transferred to the target location, computation proceeds from that location, as shown in Figure 2.6. In procedure calls, we have to return control to the calling program after executing the procedure. Control is returned to the instruction following the call instruction, as shown in Figure 2.7.

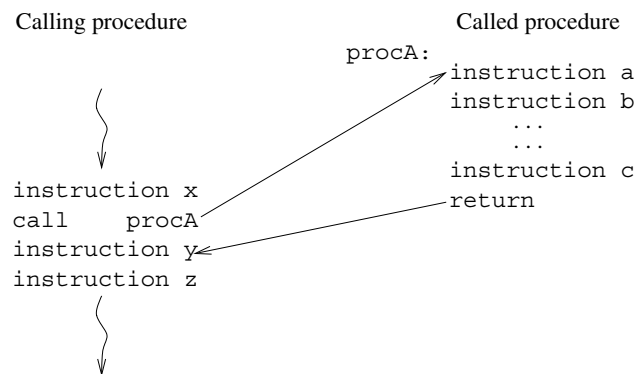


Figure 2.7 Control flow in procedure calls.

From Figures 2.6 and 2.7, you will notice that the branches and procedure calls are similar in their initial control transfer. For procedure calls, we need to return to the instruction following the procedure call. This return requires two pieces of information:

- *End of Procedure:* We have to indicate the end of the procedure so that the control can be returned. This is normally done by a special return instruction. For example, the Pentium uses `ret` and the MIPS uses the `j r` instruction to return from a procedure. We do the same in high-level languages as well. For example, in C, we use the `return` statement to indicate an end of procedure execution. High-level languages allow a default fall-through mechanism. That is, if we don't explicitly specify the end of a procedure, control is returned at the end of the block. In the assembly language, we must specify the end of a procedure by using the return instruction.
- *Return Address:* How does the processor know where to return after completing a procedure? This piece of information is normally stored when the procedure is called. Thus, when a procedure is called, it not only modifies the PC as in the branch instruction, but also stores the return address. Where does it store the return address? Two main places are used: a special register or the stack. Both MIPS and Pentium processors store the address of the instruction *following* the `call` instruction.

The Pentium uses the stack to store the return address. Thus, each procedure call involves pushing the return address onto the stack before control is transferred to the procedure code. The return instruction retrieves this value from the stack to send control back to the instruction following the procedure call. A more detailed description of the procedure call mechanism is found in Chapter 5.

MIPS processors allow any general-purpose register to store the return address. The return statement specifies this register. The format of the return statement is

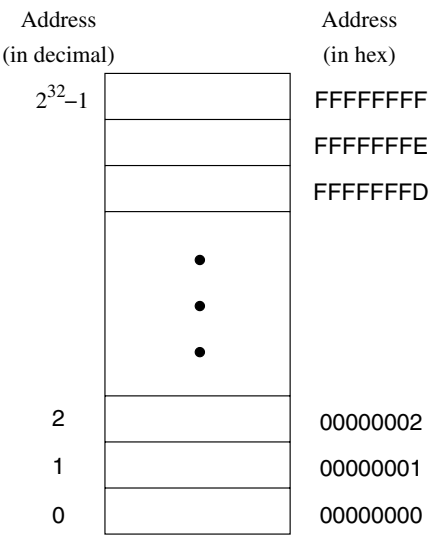


Figure 2.8 Logical view of the system memory.

```
jr    $ra
```

where *ra* is the register that contains the return address. Chapter 13 gives more details on this instruction.

Parameter Passing

The general architecture dictates how parameters are passed on to the procedures. There are two basic techniques: register-based or stack-based. In the first method, parameters are placed in the processor’s internal registers and the called procedure will read the parameter values from these registers. In the stack-based method, parameters are pushed onto the stack and the called procedure would have to read them off the stack.

The advantage of the register method is that it is faster than the stack method. However, because of the limited number of registers, it imposes a limit on the number of parameters. Furthermore, recursive procedures cannot use the register-based mechanism. Because RISC processors tend to have more registers, register-based parameter passing is used in the MIPS processors. The Pentium, due to the small number of registers, tends to use the stack for parameter passing. We describe these two parameter passing mechanisms in detail in Chapter 5.

2.5 Memory

The memory of a computer system consists of tiny electronic switches, with each switch set in one of two states: *open* or *closed*. It is, however, more convenient to think of these states

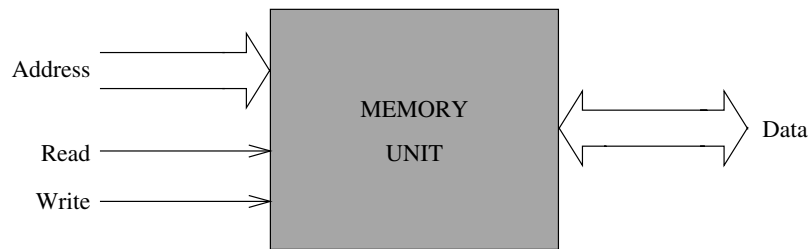


Figure 2.9 Block diagram of the system memory.

as 0 and 1 rather than open and closed. A single such switch can be used to represent two (i.e., binary) numbers: a zero and a one. Thus, each switch can represent a *binary digit* or *bit*, as it is known. The memory unit consists of millions of such bits. In order to make memory more manageable, bits are organized into groups of eight bits called *bytes*. Memory can then be viewed as consisting of an ordered sequence of bytes. Each byte in this memory can be identified by its sequence number starting with 0, as shown in Figure 2.8. This is referred to as the *memory address* of the byte. Such memory is called *byte addressable* memory.

The Pentium can address up to 4 GB (2^{32} bytes) of main memory (see Figure 2.8). This magic number comes from the fact that the address bus of the Pentium has 32 address lines. This number is referred to as the *memory address space* (MAS). The memory address space of a system is determined by the address bus width of the processor used in the system. Typically, 32-bit processors support 32-bit addresses. For example, the MIPS processor we discuss in Chapter 12 also supports 4-GB memory address space.

The actual memory in a system, however, is always less than or equal to the memory address space. The amount of memory in a system is determined by how much of this memory address space is *populated* with memory chips.

2.5.1 Two Basic Memory Operations

The memory unit supports two fundamental operations: *read* and *write*. The *read operation* reads a previously stored data and the *write operation* stores a value in memory. Both of these operations require an address in memory from which to read a value or to which to write a value. In addition, the write operation requires specification of the data to be written. The block diagram of the memory unit is shown in Figure 2.9. The address and data of the memory unit are connected to the address and data buses, respectively. The read and write signals come from the control bus.

Two metrics are used to characterize memory. *Access time* refers to the amount of time required by the memory to retrieve the data at the addressed location. The other metric is the *memory cycle time*, which refers to the minimum time between successive memory operations.

Memory transfer rates can be measured by the bandwidth metric. It specifies the number of bytes transferred per second. For example, a Pentium system with the PC133 memory can transfer 8 bytes at a frequency of 133 times per second. This gives us a bandwidth of $8 * 133 = 1064$ MB/s.

The read operation is nondestructive in the sense that one can read a location of the memory as many times as one wishes without destroying the contents of that location. The write operation, on the other hand, is destructive, as writing a value into a location destroys the old contents of that memory location.

Steps in a typical read cycle

1. Place the address of the location to be read on the address bus,
2. Activate the memory read control signal on the control bus,
3. Wait for the memory to retrieve the data from the addressed memory location and place it on the data bus,
4. Read the data from the data bus,
5. Drop the memory read control signal to terminate the read cycle.

A simple Pentium read cycle takes three clock cycles. During the first clock cycle, steps 1 and 2 are performed. The Pentium waits until the end of the second clock and reads the data and drops the read control signal. If the memory is slower (and therefore cannot supply data within the specified time), the memory unit indicates its inability to the processor and the processor waits longer for the memory to supply data by inserting *wait cycles*. Note that each wait cycle introduces a waiting period equal to one system clock period and thus slows down the system operation.

Steps in a typical write cycle

1. Place the address of the location to be written on the address bus,
2. Place the data to be written on the data bus,
3. Activate the memory write control signal on the control bus,
4. Wait for the memory to store the data at the addressed location,
5. Drop the memory write signal to terminate the write cycle.

As with the read cycle, the Pentium requires three clock cycles to perform a simple write operation. During the first clock cycle, steps 1 and 3 are done. Step 2 is performed during the second clock cycle. Pentium gives memory time until the end of the second clock and drops the memory write signal. If the memory cannot write data at the maximum processor rate, wait cycles can be introduced to extend the write cycle.

2.5.2 Types of Memory

The memory unit can be implemented using a variety of memory chips—different speeds, different manufacturing technologies, and different sizes. The two basic types of memory are the *read-only memory* and *read/write memory*.

A basic property of memory systems is that they are random access memories in that accessing any memory location (for reading or writing) takes the same time. Contrast this with data stored on a magnetic tape. Access time on the tape depends on the location of the data.

Volatility is another important property of a memory unit. A *volatile* memory requires power to retain its contents. A *nonvolatile* memory can retain its values even in the absence of power.

Read-Only Memories

Read-only memory (ROM) allows only read operations to be performed. As the name suggests, we cannot write into this memory. The main advantage of ROM is that it is nonvolatile. Most ROM is factory-programmed and cannot be altered. The term *programming* in this context refers to writing values into a ROM. This type of ROM is cheaper to manufacture in large quantities than other types of ROM. The program that controls the standard input and output functions (called BIOS), for instance, is kept in ROM. Current systems use the flash memory rather than a ROM (see our discussion later).

Other types of ROM include *programmable ROM* (PROM) and *erasable PROM* (EPROM). PROM is useful in situations where the contents of ROM are not yet fixed. For instance, when the program is still in the development stage, it is convenient for the designer to be able to program the ROM locally rather than at the time of manufacture.

In PROM, a fuse is associated with each bit cell. If the fuse is on, the bit cell supplies a 1 when read. The fuse has to be burned to read a 0 from that bit cell. When PROM is manufactured, its contents are all set to 1. To program PROM, selective fuses are burned (to introduce 0's) by sending high current. This is the writing process and is not reversible (i.e., a burned fuse cannot be restored). EPROM offers further flexibility during system prototyping. Contents of an EPROM can be erased by exposing it to ultraviolet light for a few minutes. Once erased, the EPROM can be reprogrammed.

Electrically erasable PROMs (EEPROMs) allow further flexibility. By exposing to ultraviolet light, we erase *all* the contents of an EPROM. EEPROMs, on the other hand, allow the user to selectively erase contents. Furthermore, erasing can be done in place; there is no need to place it in a special ultraviolet chamber.

Flash memory is a special kind of EEPROM. One main difference between the EEPROM and flash memory lies in how the memory contents are erased. The EEPROM is byte-erasable whereas flash memory is block-erasable. Thus, writing in the flash memory involves erasing a block and rewriting it.

Current systems use flash memory for BIOS so that changing BIOS versions is fairly straightforward (you just have to “flash” the new version). Flash memory is also becoming

very popular as a removable media. The SmartMedia, CompactFlash, and Sony's Memory Stick are all examples of various forms of removable flash media.

Flash memory, however, is slower than the RAMs we discuss next. For example, flash memory cycle time is about 80 ns whereas the corresponding value for RAMs is about 10 ns. Nevertheless, since flash memories are nonvolatile, they are used in applications where this property is important. Apart from BIOS, we see them in devices like digital cameras and video game systems.

Read/Write Memory

Read/write memory is commonly referred to as *random access memory* (RAM), even though ROM is also random access memory. This terminology is so entrenched in the literature that we follow it here with a cautionary note that RAM actually refers to RWM.

Read/write memory can be divided into *static* and *dynamic* categories. Static random access memory (SRAM) retains the data, once written, without further manipulation so long as the source of power holds its value. SRAM is typically used for implementing the processor registers and cache memories.

The bulk of main memory in a typical computer system, however, consists of dynamic random access memory (DRAM). DRAM is a complex memory device that uses a tiny capacitor to store a bit. A charged capacitor represents 1 bit. Since capacitors slowly lose their charge due to leakage, they must be *refreshed* periodically to replace the charges representing 1 bit. A typical refresh period is about 64 ms. Reading from DRAM involves testing to see if the corresponding bit cells are charged. Unfortunately, this test destroys the charges on the bit cells. Thus, DRAM is a destructive read memory.

For proper operation, a read cycle is followed by a restore cycle. As a result, the DRAM cycle time, the actual time necessary between accesses, is typically about twice the read access time, which is the time necessary to retrieve a datum from the memory.

Several types of DRAM chips are available. We briefly describe some of the most popular types next.

FPM DRAMs Fast page-mode (FPM) DRAMs are an improvement over the previous generation DRAMs. FPM DRAMs exploit the fact that we access memory sequentially, most of the time. To know how this access pattern characteristic is exploited, we have to look at how the memory is organized. Internally, the memory is organized as a matrix of bits. For example, a 32-Mb memory could be organized as 8 K rows (i.e., 8192 since $K = 1024$) and 4-K columns. To access a bit, we have to supply a row address and a column address. In the FPM DRAM, a page represents part of the memory with the same row address. To access a page, we specify the row address only once; we can read the bits in the specified page by changing the column addresses. Since the row address is not changing, we save on the memory cycle time.

EDO DRAMs Extended Data Output (EDO) DRAM is another type of FPM DRAM. It also exploits the fact that we access memory sequentially. However, it uses pipelining to speed up memory access. That is, it initiates the next request before the previous memory access is completed. A characteristic of pipelining inherited by EDO DRAMs is that single memory reference requests are not sped up. However, by overlapping multiple memory access requests, it improves the memory bandwidth.

SDRAMs Both FPM DRAMs and EDO DRAMs are asynchronous in the sense that their data output is not synchronized to a clock. The synchronous DRAM (SDRAM) uses an external clock to synchronize the data output. This synchronization reduces delays and thereby improves the memory performance. The SDRAM memories are used in systems that require memory satisfying the PC100/PC133 specification. SDRAMs are dominant in low-end PC market and are cheap.

DDR SDRAMs The SDRAM memories are also called single data rate (SDR) SDRAMs as they supply data once per memory cycle. However, with increasing processor speeds, the processor bus (also called front-side bus or FSB) frequency is also going up. For example, Pentium systems now have 533-MHz FSB that supports a transfer rate of about 4.2 GB/s. To satisfy this transfer rate, SDRAMs have been improved to provide data at both rising and falling edges of the clock. This effectively doubles the memory bandwidth and satisfies the high data transfer rates of faster processors.

RDRAMs Rambus DRAM (RDRAM) takes a completely different approach to increase the memory bandwidth. A technology developed and licensed by Rambus, it is a memory subsystem that consists of the RAM, RAM controller, and a high-speed bus called the Rambus channel. Like the DDR DRAM, it also performs two transfers per cycle. In contrast to the 8-byte-wide data bus of DRAMs, Rambus channel is a 2-byte data bus. However, by using multiple channels, we can increase the bandwidth of RDRAMs. For example, a dual-channel RDRAM operating at 533 MHz provides a bandwidth of $533 * 2 * 4 = 4.2$ GB/s, sufficient for the 533-MHz FSB systems.

From this brief discussion it should be clear that DDR SDRAMs and RDRAMs compete with each other in the high-end market. The race between these two DRAM technologies continues as Intel boosts its FSB to 800 MHz.

2.5.3 Storing Multibyte Data

Storing data often requires more than a byte. For example, we need four bytes of memory to store an integer variable that can take a value between 0 and $2^{32} - 1$. Let us assume that the value to be stored is the one in Figure 2.10a.

Suppose we want to store these four bytes of data in memory at locations 100 through 103. How do we store them? Figure 2.10 shows two possibilities: least significant byte

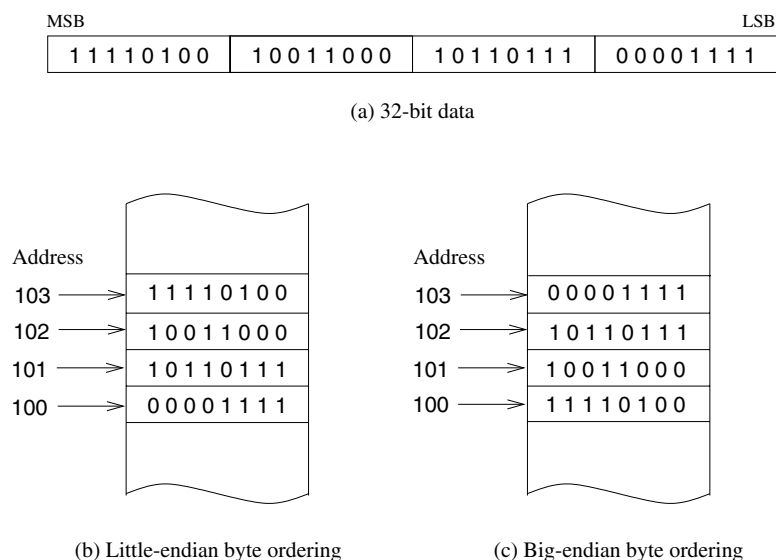


Figure 2.10 Two byte-ordering schemes.

(Figure 2.10b) or most significant byte (Figure 2.10c) is stored at location 100. These two byte-ordering schemes are referred to as the *little endian* and *big endian*. In either case, we always refer to such multibyte data by specifying the lowest memory address (100 in this example).

Is one byte-ordering scheme better than the other? Not really! It is largely a matter of choice for the designers. For example, Pentium processors use the little-endian byte ordering. However, most processors leave it up to the system designer to configure the processor. For example, the MIPS and PowerPC processors use the big-endian byte ordering by default, but these processors can be configured to use the little-endian scheme.

The particular byte-ordering scheme used does not pose any problems as long as you are working with machines that use the same byte-ordering scheme. However, difficulties arise when you want to transfer data between two machines that use different schemes. In this case, conversion from one scheme to the other is required. For example, the Pentium provides two instructions to facilitate such conversion: one to perform 16-bit data conversions and the other for 32-bit data. Later chapters give details on these instructions.

2.6 Input/Output

Input/output (I/O) devices provide the means by which a computer system can interact with the outside world. An I/O device can be purely an input device (e.g., keyboard, mouse), purely

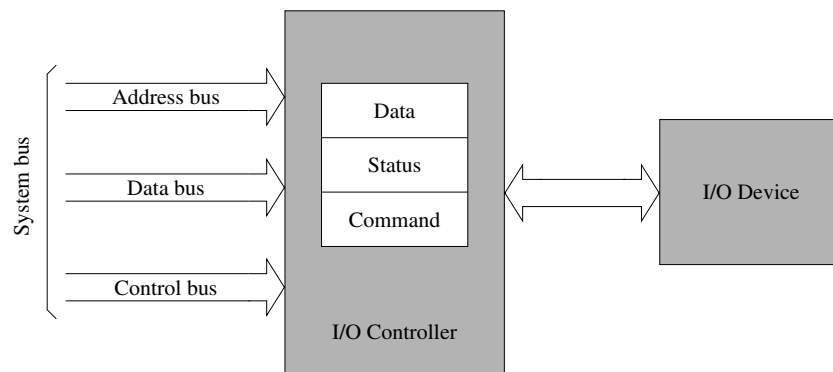


Figure 2.11 Block diagram of a generic I/O device interface.

an output device (e.g., printer, display screen), or both an input and output device (e.g., disks). Here we present a brief overview of the I/O device interface. Chapters 14 and 15 provide more details on I/O interfaces.

Computers use I/O devices (also called *peripheral devices*) for two major purposes—to communicate with the outside world, and to store data. I/O devices such as printers, keyboards, and modems are used for communication purposes, and devices like disk drives are used for data storage. Regardless of the intended purpose of the I/O device, all communications with these devices must involve the systems bus. However, I/O devices are not directly connected to the system bus. Instead, there is usually an *I/O controller* that acts as an interface between the system and the I/O device.

There are two main reasons for using an I/O controller. First, different I/O devices exhibit different characteristics and, if these devices were connected directly, the processor would have to understand and respond appropriately to each I/O device. This would cause the processor to spend a lot of time interacting with I/O devices and spend less time executing user programs. If we use an I/O controller, this controller could provide the necessary low-level commands and data for proper operation of the associated I/O device. Often, for complex I/O devices such as disk drives, special I/O controller chips are available.

The second reason for using an I/O controller is that the amount of electrical power used to send signals on the system bus is very low. This means that the cable connecting the I/O device has to be very short (a few centimeters at most). I/O controllers typically contain driver hardware to send current over long cables that connect the I/O devices.

I/O controllers typically have three types of internal registers—a data register, a command register, and a status register—as shown in Figure 2.11. When the processor wants to interact with an I/O device, it communicates only with the associated I/O controller.

To focus our discussion, let us consider printing a character on the printer. Before the processor sends a character to be printed, it has to first check the status register of the associated

I/O controller to see whether the printer is online/offline, busy or idle, out of paper, and so on. In the status register, three bits can be used to provide this information. For example, bit 4 can be used to indicate whether the printer is online (1) or offline (0), bit 7 can be used for busy (1) or not busy (0) status indication, and bit 5 can be used for out of paper (1) or not (0).

The data register holds the character to be printed, and the command register tells the controller the operation requested by the processor (for example, send the character in the data register to the printer). The following summarizes the sequence of actions involved in sending a character to the printer:

- Wait for the controller to finish the last command;
- Place a character to be printed in the data register;
- Set the command register to initiate the transfer.

The processor accesses the internal registers of an I/O controller through what are known as *I/O ports*. An I/O port is simply the address of a register associated with an I/O controller.

There are two ways of mapping I/O ports. Some processors such as the MIPS map I/O ports to memory addresses. This is called *memory-mapped I/O*. In these systems, writing to an I/O port is similar to writing to a memory address. Other processors like the Pentium have an *I/O address space* that is separate from the memory address space. This technique is called *isolated I/O*. In these systems, to access the I/O address space, special I/O instructions are needed. Pentium provides two instructions—*in* and *out*—to access I/O ports. The *in* instruction can be used to read from an I/O port and the *out* for writing to an I/O port. Chapter 15 gives more details on these instructions.

Pentium provides 64 KB of I/O address space. This address space can be used for 8-bit, 16-bit, and 32-bit I/O ports. However, the combination cannot be more than the I/O address space. For example, we can have 64-K 8-bit ports, 32-K 16-bit ports, 16-K 32-bit ports, or a combination of these that fits the 64-K address space.

Systems designed with processors supporting the isolated I/O have the flexibility of using either the memory-mapped I/O or the isolated I/O. Typically, both strategies are used. For instance, devices like the printer or keyboard could be mapped to the I/O address using the isolated I/O strategy; the display screen could be mapped to a set of memory addresses using the memory-mapped I/O.

Accessing I/O Devices As a programmer, you can have direct control on any of the I/O devices (through their associated I/O controllers) when you program in the assembly language. However, it is often a difficult task to access an I/O device without any help. Furthermore, it is a waste of time and effort if everyone has to develop his or her own routine to access I/O devices (called *device drivers*). In addition, system resources could be abused, either unintentionally or maliciously. For instance, an improper disk driver could erase the contents of a disk due to a bug in the driver routine.

To avoid these problems and to provide a standard way of accessing I/O devices, operating systems provide routines to conveniently access I/O devices. For example, Linux provides a

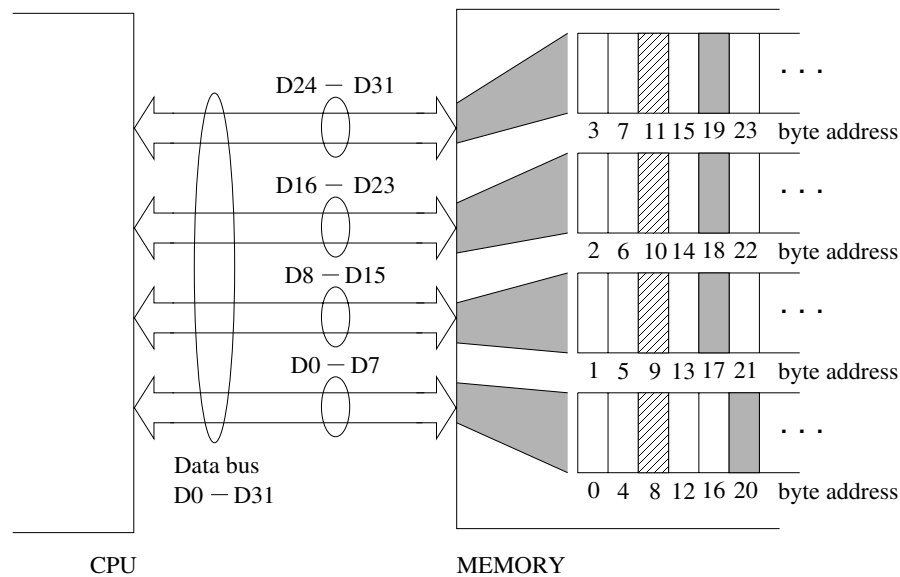


Figure 2.12 Byte-addressable memory interface to the 32-bit data bus.

set of system calls to access system I/O devices. In Windows, access to I/O devices can be obtained from two layers of system software: the basic I/O system (BIOS), and the operating system. BIOS is ROM resident and is a collection of routines that control the basic I/O devices. Both provide access to routines that control the I/O devices through a mechanism called *interrupts*. Interrupts are discussed in detail in Chapter 14.

2.7 Performance: Effect of Data Alignment

Execution time of a program is influenced by several factors—some of which are under the control of the programmer. Other factors that influence the running time of a program include the clock rate of the system, efficiency of the compiler if the program is written in a high-level language, presence of a cache memory, and so on.

Here we look at the influence of data alignment on the performance of the bubble sort example discussed in Chapter 5 (see Example 5.5 on page 142). One of the factors influencing the sort time is the time required to access the array.

Suppose we want to read a 32-bit variable from the memory. Assume that the data bus is 32 bits wide. If the address of this variable is a multiple of four, the 32-bit data are stored in a single row of memory. Thus the processor can get the data in one read cycle. If this is not true, then the 32-bit data item is spread over two rows. Thus the processor reads two 32-bits of data and assembles the required 32-bit data. This scenario is clearly demonstrated in Figure 2.12.

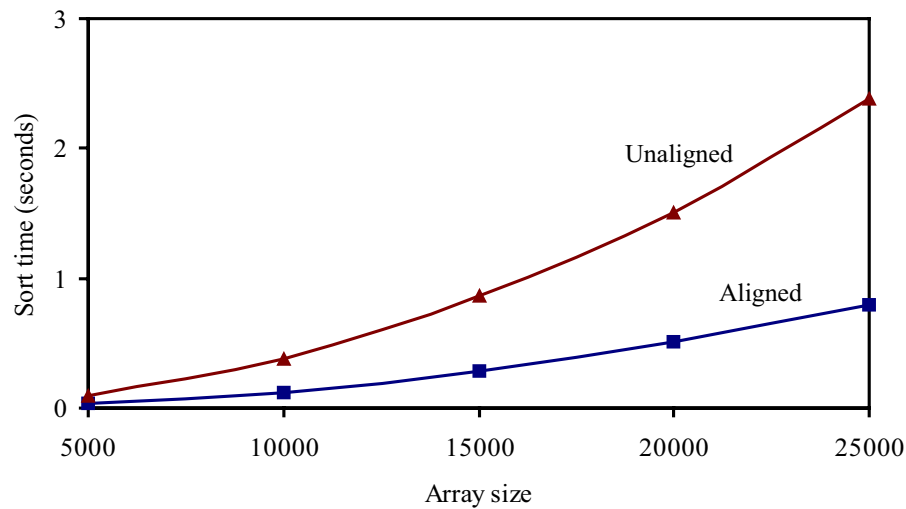


Figure 2.13 Impact of data alignment on the performance of the bubble sort algorithm.

In Figure 2.12, 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.

Figure 2.13 shows the impact of data alignment on the sort time of the bubble sort. These results were obtained on a 2.4-GHz Pentium 4 processor system. The unaligned sort time is approximately three times more than the aligned sort time.

Except for the performance penalty, data alignment is totally transparent to the application. However, to avoid this performance penalty, data should be aligned.

- **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.
- **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 processors such as the Pentium that have a 64-bit-wide data bus. On processors (e.g., 80486) that have 32-bit-wide data bus, a 64-bit data item is read in two bus cycles and alignment at four-byte boundaries is sufficient.

The Intel Pentium family of processors allows aligned and unaligned data items. Of course, unaligned data cause performance degradation. An alignment constraint of this type is referred to as the *soft alignment* constraint. Because of the performance penalty associated with unaligned data, some processors do not allow unaligned data. This alignment constraint is referred to as the *hard alignment* constraint.

2.8 Summary

Programmers should have some basic knowledge about the processor and the system architecture in order to effectively program in the assembly language. This chapter has presented the basics of computer organization.

We started with a high-level view of the system. At this level, a computer system can be thought of as consisting of three main components: a processor, a memory unit, and I/O devices. The remainder of the chapter briefly described these three components.

We also considered the impact of data alignment on the execution time of application programs. By using the bubble sort example discussed in Chapter 5, we demonstrated the influence of data alignment on the sort time.

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:

- 0-address machines
- 1-address machines
- 2-address machines
- 3-address machines
- Absolute address
- Accumulator machines
- Conditional branch
- End of procedure
- Isolated I/O
- Load/store architecture
- Memory-mapped I/O
- Number of addresses
- Parameter passing
- PC-relative
- Procedure call
- Processor registers
- Return address
- Stack machines
- Unconditional branch
- Wait cycles

2.9 Exercises

- 2-1 Describe the execution cycle. What is the default mode of execution?
- 2-2 What are the main components of the system bus? Describe the functionality of each component.
- 2-3 If a system uses a 1.5-GHz clock, what is the clock period?

- 2-4 If a processor has 64 address lines, what is the physical memory address space of this processor? Give the address of the first and last addressable memory locations in hex.
- 2-5 What are the characteristics of the load/store architecture that sets it apart from the other architectures discussed in this chapter?
- 2-6 We stated that the RISC processors use the load/store architecture. What are its advantages over the CISC architectures as exemplified by the Pentium processor?
- 2-7 Give some of the reasons why instruction execution rate is higher in RISC processors than in CISC processors.
- 2-8 From the processor point of view, what are the differences between branches and procedures?
- 2-9 Explain the differences between branches that use absolute address and PC-relative address.
- 2-10 What are the differences between ROM and RAM?
- 2-11 Compare and contrast DRAM and SRAM.
- 2-12 Why do DRAMs need to be refreshed?
- 2-13 What is the difference between volatile and nonvolatile memories?
- 2-14 Can you think of a reason why ROMs tend to be nonvolatile and RAMs volatile?
- 2-15 Consider the Pentium processor with an 800-MHz front-side bus. What is the bandwidth of this bus?
- 2-16 We stated that DDR SDRAMs and RDRAMs compete for the high-end systems that require higher bandwidth to support 533-MHz FSB. Suppose we use a four-channel RDRAM memory subsystem. What is the clock frequency that this subsystem should operate in order to meet the bandwidth requirement of the last question?
- 2-17 Discuss why I/O controllers are used to interface I/O devices to the system.
- 2-18 For each address below, state whether a 32-bit value stored at that address is aligned or not (all numbers are in hex):
 - (a) 12345678 (c) 9128ADCC
 - (b) ABCD755A (d) 38B0F050
- 2-19 Repeat the above exercise for 64-bit values.



<http://www.springer.com/978-0-387-20636-3>

Introduction to Assembly Language Programming
For Pentium and RISC Processors

Dandamudi, S.P.

2005, XXIV, 692 p., Hardcover

ISBN: 978-0-387-20636-3