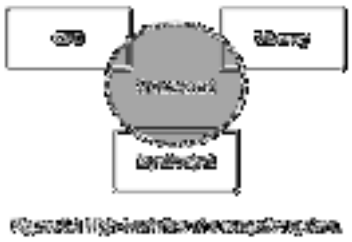
LECTURE 2
Basic Computer Organization

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

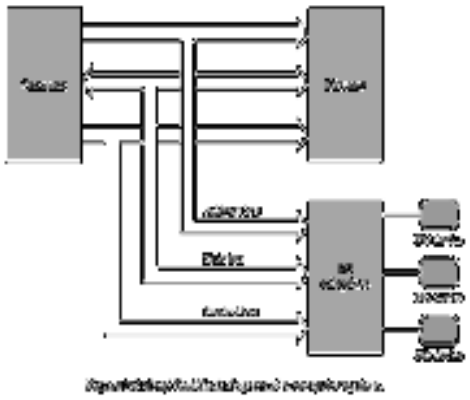
Programming in a high-level language does not require a detailed knowledge of the system hardware. Assembly language programmers, however, should have some basic understanding of the underlying system architecture. A high-level view of computer systems, presented in Section 2.1, consists of three major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



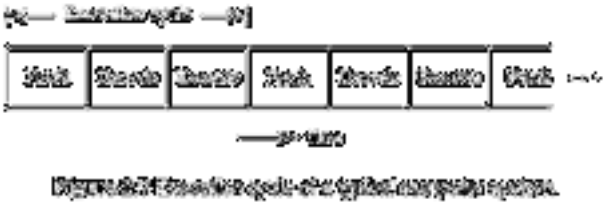
2.1 Basic Components of a Computer System

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 220 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 232 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, 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.



2.2 The Processor

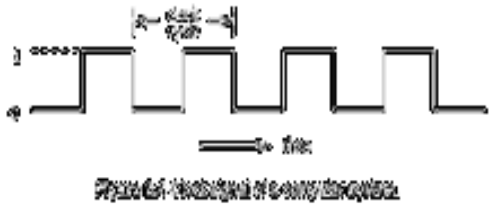
The processor acts as the controller of all actions or services provided by the system. It can be thought of as executing the following cycle forever:

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. 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.

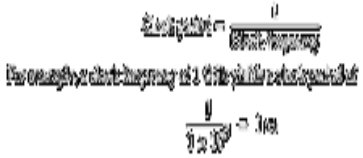
2.2.1 The Execution Cycle

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.



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 106 and 109 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.



If it takes three clock cycles to execute an instruction, it takes 3×1 ns = 3 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.

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.

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 2.1 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`

Table 2.2 Sample Two-Address Machine Instructions

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

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

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

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.

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 Some Load/Store Machine Instructions

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