COMPUTER ORGANIZATION AND ARCHITECTURE

Designing for Performance

SIXTH EDITION

Two and Three. Now, we turn to the question of how these functions are performed or, more specifically, how the various elements of the processor are controlled to provide these functions. Thus, we turn to a discussion of the control unit, which controls the operation of the processor.

16.1 MICRO-OPERATIONS

We have seen that the operation of a computer, in executing a program, consists of a sequence of instruction cycles, with one machine instruction per cycle. Of course, we must remember that this sequence of instruction cycles is not necessarily the same as the *written sequence* of instructions that make up the program, because of the existence of branching instructions. What we are referring to here is the execution *time sequence* of instructions.

We have further seen that each instruction cycle is made up of a number of smaller units. One subdivision that we found convenient is fetch, indirect, execute, and interrupt, with only fetch and execute cycles always occurring.

To design a control unit, however, we need to break down the description further. In our discussion of pipelining in Chapter 12, we began to see that a further decomposition is possible. In fact, we will see that each of the smaller cycles involves a series of steps, each of which involves the processor registers. We will refer to these steps as *micro-operations.* The prefix *micro* refers to the fact that each step is very simple and accomplishes very little. Figure 16.1 depicts the relationship among the various concepts we have been discussing. To summarize, the execution of a program consists of the sequential execution of instructions. Each instruction is executed during an instruction cycle made up of shorter subcycles (e.g., fetch, indirect,

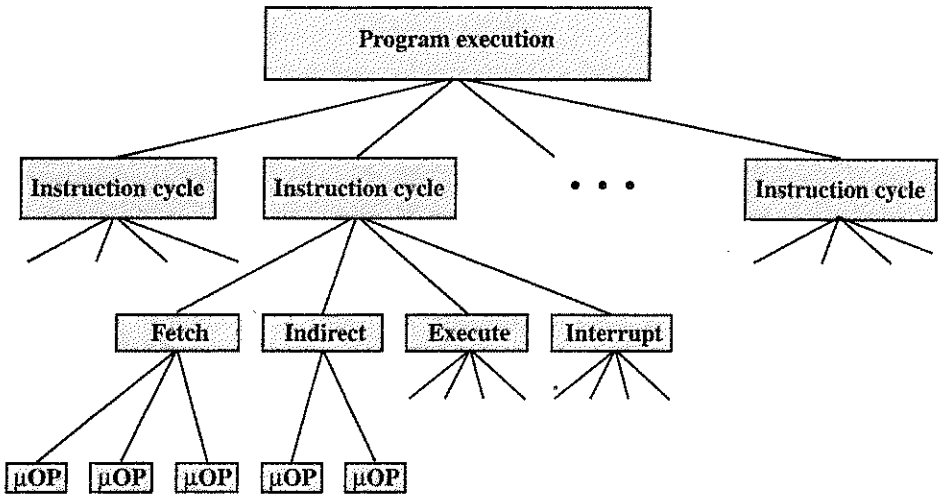

Figure 16.1 Constituent Elements of a Program Execution

execute, interrupt). The performance of each subcycle involves one or mo operations, that is, micro-operations.

Micro-operations are the functional, or atomic, operations of a pro this section, we will examine micro-operations to gain an understandin the events of any instruction cycle can be described as a sequence of su operations. A simple example will be used. In the remainder of this chapte show how the concept of micro-operations serves as a guide to the desi control unit.

The Fetch Cycle

We begin by looking at the fetch cycle, which occurs at the beginnin instruction cycle and causes an instruction to be fetched from memory. poses of discussion, we assume the organization depicted in Figure 12.6. isters are involved:

- **Memory address register (MAR):** Is connected to the address lines c tem bus. It specifies the address in memory for a read or write oper
- **Memory buffer register (MBR):** Is connected to the data lines of the sy It contains the value to be stored in memory or the last value read from
- **Program counter (PC):** Holds the address of the next instruction to b
- **Instruction register (IR):** Holds the last instruction fetched.

Let us look at the sequence of events for the fetch cycle from the poi of its effect on the processor registers. An example appears in Figure 16. beginning of the fetch cycle, the address of the next instruction to be exec the program counter (PC); in this case, the address is 1100100. The first move that address to the memory address register (MAR) because this is register connected to the address lines of the system bus. The second step i in the instruction. The desired address (in the MAR) is placed on the ad the control unit issues a READ command on the control bus, and the resul on the data bus and is copied into the memory buffer register (MBR). We to increment the PC by 1 to get ready for the next instruction. Because t actions (read word from memory, add 1 to PC) do not interfere with each can do them simultaneously to save time. The third step is to move the co the MBR to the instruction register (IR). This frees up the MBR for use possible indirect cycle.

Thus, the simple fetch cycle actually consists of three steps and fo operations. Each micro-operation involves the movement of data into o register. So long as these movements do not interfere with one another, s them can take place during one step, saving time. Symbolically, we can sequence of events as follows:

t_1: MAR $\leftarrow$ (PC)
t_2: MBR $\leftarrow$ Memory
PC $\leftarrow$ (PC) + I
t_3: IR $\leftarrow$ (MBR)

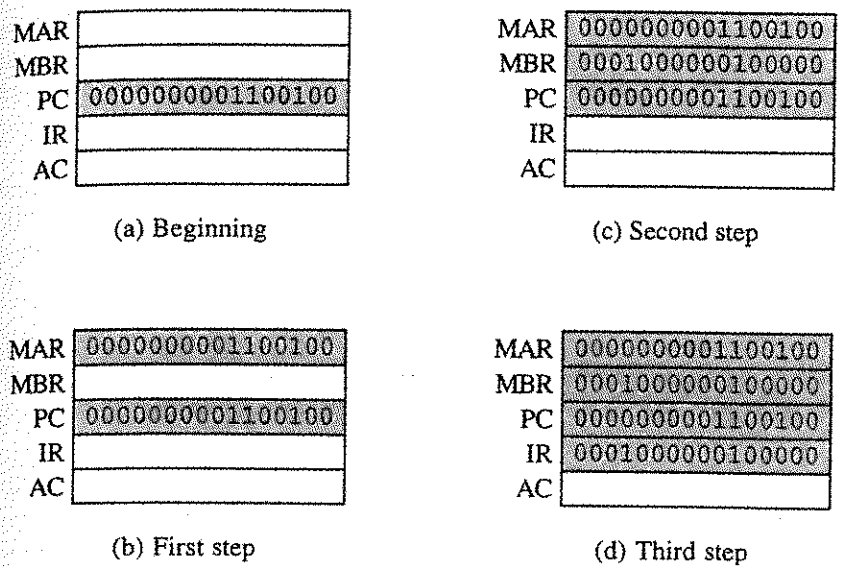

Figure 16.2 Sequence of Events, Fetch Cycle

where I is the instruction length. We need to make several comments about this sequence. We assume that a clock is available for timing purposes and that it emits regularly spaced clock pulses. Each clock pulse defines a time unit. Thus, all time units are of equal duration. Each micro-operation can be performed within the time of a single time unit. The notation (t_1, t_2, t_3) represents successive time units. In words, we have

- **First time unit:** Move contents of PC to MAR.
- **Second time unit:** Move contents of memory location specified by MAR to MBR. Increment by I the contents of the PC.
- **Third time unit:** Move contents of MBR to IR.

Note that the second and third micro-operations both take place during the second time unit. The third micro-operation could have been grouped with the fourth without affecting the fetch operation:

```
t1: MAR ← (PC)
t2: MBR ← Memory
t3: PC ← (PC) + I
    IR ← (MBR)
```

The groupings of micro-operations must follow two simple rules:

1. The proper sequence of events must be followed. Thus (MAR ← (PC)) must precede (MBR ← Memory) because the memory read operation makes use of the address in the MAR.

2. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR ← Memory) and (IR ← MBR) should not occur during the same time unit.

A final point worth noting is that one of the micro-operations involves an addition. To avoid duplication of circuitry, this addition could be performed by the ALU. The use of the ALU may involve additional micro-operations, depending on the functionality of the ALU and the organization of the processor. We defer a discussion of this point until later in this chapter.

It is useful to compare events described in this and the following subsections to Figure 3.5. Whereas micro-operations are ignored in that figure, this discussion shows the micro-operations needed to perform the subcycles of the instruction cycle.

The Indirect Cycle

Once an instruction is fetched, the next step is to fetch source operands. Continuing our simple example, let us assume a one-address instruction format, with direct and indirect addressing allowed. If the instruction specifies an indirect address, then an indirect cycle must precede the execute cycle. The data flow differs somewhat from that indicated in Figure 12.7 and includes the following micro-operations:

t_1: MAR ← (IR(Address))
t_2: MBR ← Memory
t_3: IR(Address) ← (MBR(Address))

The address field of the instruction is transferred to the MAR. This is then used to fetch the address of the operand. Finally, the address field of the IR is updated from the MBR, so that it now contains a direct rather than an indirect address.

The IR is now in the same state as if indirect addressing had not been used, and it is ready for the execute cycle. We skip that cycle for a moment, to consider the interrupt cycle.

The Interrupt Cycle

At the completion of the execute cycle, a test is made to determine whether any enabled interrupts have occurred. If so, the interrupt cycle occurs. The nature of this cycle varies greatly from one machine to another. We present a very simple sequence of events, as illustrated in Figure 12.8. We have

t_1: MBR ← (PC)
t_2: MAR ← Save_Address
PC ← Routine_Address
t_3: Memory ← (MBR)

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may

each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may take one or more additional micro-operations to obtain the save_address and the routine_address before they can be transferred to the MAR and PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

The Execute Cycle

The fetch, indirect, and interrupt cycles are simple and predictable. Each involves a small, fixed sequence of micro-operations and, in each case, the same micro-operations are repeated each time around.

This is not true of the execute cycle. For a machine with N different opcodes, there are N different sequences of micro-operations that can occur. Let us consider several hypothetical examples.

First, consider an add instruction:

```
ADD R1, X
```

which adds the contents of the location X to register R1. The following sequence of micro-operations might occur:

```
t1: MAR ← (IR(address))
t2: MBR ← Memory
t3: R1 ← (R1) + (MBR)
```

We begin with the IR containing the ADD instruction. In the first step, the address portion of the IR is loaded into the MAR. Then the referenced memory location is read. Finally, the contents of R1 and MBR are added by the ALU. Again, this is a simplified example. Additional micro-operations may be required to extract the register reference from the IR and perhaps to stage the ALU inputs or outputs in some intermediate registers.

Let us look at two more complex examples. A common instruction is increment and skip if zero:

```
ISZ X
```

The content of location X is incremented by 1. If the result is 0, the next instruction is skipped. A possible sequence of micro-operations is

```
t1: MAR ← (IR(address))
t2: MBR ← Memory
t3: MBR ← (MBR) + 1
t4: Memory ← (MBR)
    If ((MBR) = 0) then (PC ← (PC) + I)
```

The new feature introduced here is the conditional action. The PC is incremented if (MBR) = 0. This test and action can be implemented as one micro-

operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

Finally, consider a subroutine call instruction. As an example, consider a branch-and-save-address instruction:

```
BSA X
```

The address of the instruction that follows the BSA instruction is saved in location X, and execution continues at location X + I. The saved address will later be used for return. This is a straightforward technique for providing subroutine calls. The following micro-operations suffice:

```
t1: MAR ← (IR(address))
    MBR ← (PC)
t2: PC ← (IR(address))
    Memory ← (MBR)
t3: PC ← (PC) + I
```

The address in the PC at the start of the instruction is the address of the next instruction in sequence. This is saved at the address designated in the IR. The latter address is also incremented to provide the address of the instruction for the next instruction cycle.

The Instruction Cycle

We have seen that each phase of the instruction cycle can be decomposed into a sequence of elementary micro-operations. In our example, there is one sequence each for the fetch, indirect, and interrupt cycles, and, for the execute cycle, there is one sequence of micro-operations for each opcode.

To complete the picture, we need to tie sequences of micro-operations together, and this is done in Figure 16.3. We assume a new 2-bit register called the *instruction cycle code* (ICC). The ICC designates the state of the processor in terms of which portion of the cycle it is in:

00: Fetch
01: Indirect
10: Execute
11: Interrupt

At the end of each of the four cycles, the ICC is set appropriately. The indirect cycle is always followed by the execute cycle. The interrupt cycle is always followed by the fetch cycle (see Figure 12.4). For both the execute and fetch cycles, the next cycle depends on the state of the system.

Thus, the flowchart of Figure 16.3 defines the complete sequence of micro-operations, depending only on the instruction sequence and the interrupt pattern. Of course, this is a simplified example. The flowchart for an actual processor would be more complex. In any case, we have reached the point in our discussion in which the operation of the processor is defined as the performance of a sequence of micro-operations. We can now consider how the control unit causes this sequence to occur.

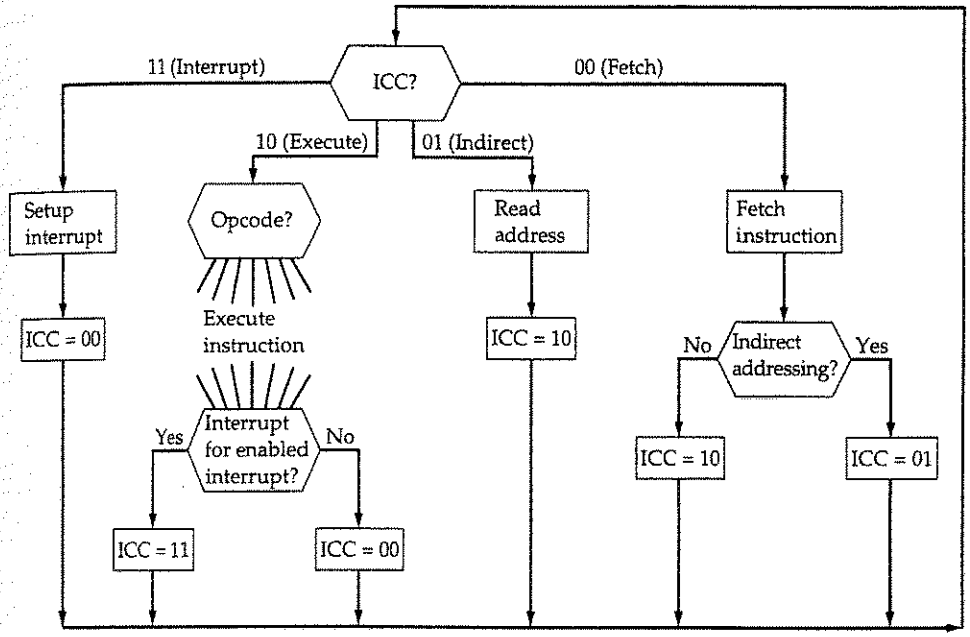

Figure 16.3 Flowchart for Instruction Cycle

16.2 CONTROL OF THE PROCESSOR

Functional Requirements

As a result of our analysis in the preceding section, we have decomposed the behavior or functioning of the processor into elementary operations, called micro-operations. By reducing the operation of the processor to its most fundamental level, we are able to define exactly what it is that the control unit must cause to happen. Thus, we can define the *functional requirements* for the control unit: those functions that the control unit must perform. A definition of these functional requirements is the basis for the design and implementation of the control unit.

With the information at hand, the following three-step process leads to a characterization of the control unit:

1. Define the basic elements of the processor.
2. Describe the micro-operations that the processor performs.
3. Determine the functions that the control unit must perform to cause the micro-operations to be performed.

We have already performed steps 1 and 2. Let us summarize the results. First, the basic functional elements of the processor are the following:

- ALU
- Registers
- Internal data paths
- External data paths
- Control unit

Some thought should convince you that this is a complete list. The ALU is the functional essence of the computer. Registers are used to store data internal to the processor. Some registers contain status information needed to manage instruction sequencing (e.g., a program status word). Others contain data that go to or come from the ALU, memory, and I/O modules. Internal data paths are used to move data between registers and between register and ALU. External data paths link registers to memory and I/O modules, often by means of a system bus. The control unit causes operations to happen within the processor.

The execution of a program consists of operations involving these processor elements. As we have seen, these operations consist of a sequence of micro-operations. Upon review of Section 16.1, the reader should see that all micro-operations fall into one of the following categories:

- Transfer data from one register to another.
- Transfer data from a register to an external interface (e.g., system bus).
- Transfer data from an external interface to a register.
- Perform an arithmetic or logic operation, using registers for input and output.

All of the micro-operations needed to perform one instruction cycle, including all of the micro-operations to execute every instruction in the instruction set, fall into one of these categories.

We can now be somewhat more explicit about the way in which the control unit functions. The control unit performs two basic tasks:

- **Sequencing:** The control unit causes the processor to step through a series of micro-operations in the proper sequence, based on the program being executed.
- **Execution:** The control unit causes each micro-operation to be performed.

The preceding is a functional description of what the control unit does. The key to how the control unit operates is the use of control signals.

Control Signals

We have defined the elements that make up the processor (ALU, registers, data paths) and the micro-operations that are performed. For the control unit to perform its function, it must have inputs that allow it to determine the state of the system and outputs that allow it to control the behavior of the system. These are the external specifications of the control unit. Internally, the control unit must have the logic required to perform its sequencing and execution functions. We defer a discussion of the internal operation of the control unit to Section 16.3 and Chapter 17. The remainder of this section is concerned with the interaction between the control unit and the other elements of the processor.

Figure 16.4 is a general model of the control unit, showing all of its inputs and outputs. The inputs are as follows:

- **Clock:** This is how the control unit "keeps time." The control unit causes one micro-operation (or a set of simultaneous micro-operations) to be performed for each clock pulse. This is sometimes referred to as the processor cycle time, or the clock cycle time.
- **Instruction register:** The opcode of the current instruction is used to determine which micro-operations to perform during the execute cycle.
- **Flags:** These are needed by the control unit to determine the status of the processor and the outcome of previous ALU operations. For example, for the increment-and-skip-if-zero (ISZ) instruction, the control unit will increment the PC if the zero flag is set.
- **Control signals from control bus:** The control bus portion of the system bus provides signals to the control unit, such as interrupt signals and acknowledgments.

The outputs are as follows:

- **Control signals within the processor:** These are two types: those that cause data to be moved from one register to another, and those that activate specific ALU functions.
- **Control signals to control bus:** These are also of two types: control signals to memory, and control signals to the I/O modules.

The new element that has been introduced in this figure is the control signal. Three types of control signals are used: those that activate an ALU function, those that activate a data path, and those that are signals on the external system bus or other external interface. All of these signals are ultimately applied directly as binary inputs to individual logic gates.

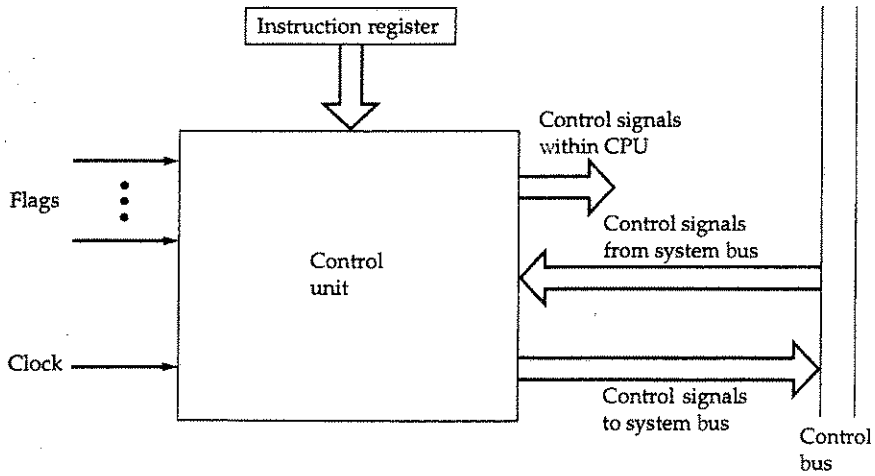

Figure 16.4 Model of the Control Unit

Let us consider again the fetch cycle to see how the control unit maintains control. The control unit keeps track of where it is in the instruction cycle. At a given point, it knows that the fetch cycle is to be performed next. The first step is to transfer the contents of the PC to the MAR. The control unit does this by activating the control signal that opens the gates between the bits of the PC and the bits of the MAR. The next step is to read a word from memory into the MBR and increment the PC. The control unit does this by sending the following control signals simultaneously:

- A control signal that opens gates, allowing the contents of the MAR onto the address bus
- A memory read control signal on the control bus
- A control signal that opens the gates, allowing the contents of the data bus to be stored in the MBR
- Control signals to logic that add 1 to the contents of the PC and store the result back to the PC

Following this, the control unit sends a control signal that opens gates between the MBR and the IR.

This completes the fetch cycle except for one thing: The control unit must decide whether to perform an indirect cycle or an execute cycle next. To decide this, it examines the IR to see if an indirect memory reference is made.

The indirect and interrupt cycles work similarly. For the execute cycle, the control unit begins by examining the opcode and, on the basis of that, decides which sequence of micro-operations to perform for the execute cycle.

A Control Signals Example

To illustrate the functioning of the control unit, let us examine a simple example. Figure 16.5 illustrates the example. This is a simple processor with a single accumulator. The data paths between elements are indicated. The control paths for signals emanating from the control unit are not shown, but the terminations of control signals are labeled C_i and indicated by a circle. The control unit receives inputs from the clock, the instruction register, and flags. With each clock cycle, the control unit reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

- **Data paths:** The control unit controls the internal flow of data. For example, on instruction fetch, the contents of the memory buffer register are transferred to the instruction register. For each path to be controlled, there is a gate (indicated by a circle in the figure). A control signal from the control unit temporarily opens the gate to let data pass.
- **ALU:** The control unit controls the operation of the ALU by a set of control signals. These signals activate various logic devices and gates within the ALU.
- **System bus:** The control unit sends control signals out onto the control lines of the system bus (e.g., memory READ).

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits

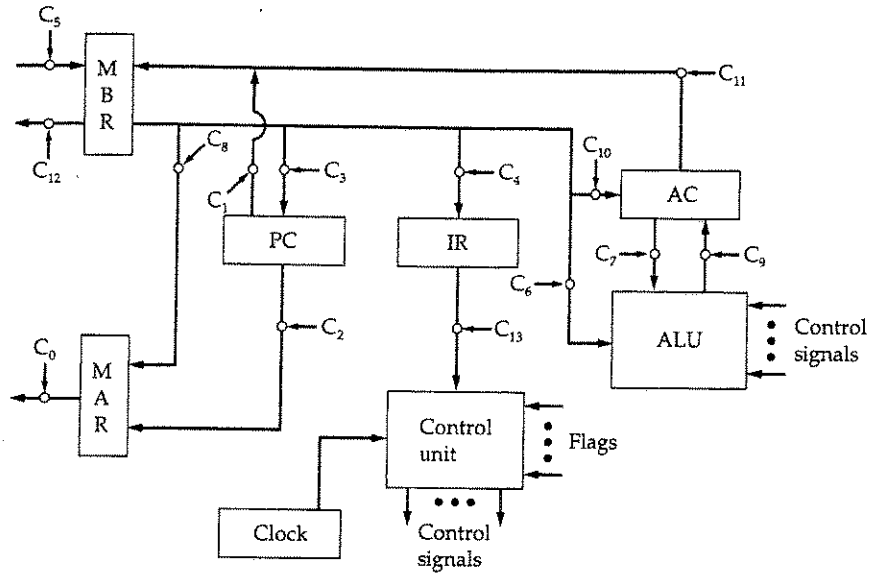

Figure 16.5 Data Paths and Control Signals

a sequence of control signals that causes micro-operations to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize. Table 16.1 indicates the control signals that are needed for some of the micro-operation sequences described earlier. For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown.

It is worth pondering the minimal nature of the control unit. The control unit is the engine that runs the entire computer. It does this based only on knowing the instructions to be executed and the nature of the results of arithmetic and logical

Table 16.1 Micro-Operations and Control Signals

Micro-Operations	Timing	Active Control Signals
Fetch:	t1: MAR ← (PC)	C_2
	t2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t3: IR ← (MBR)	C_4
Indirect:	t1: MAR ← (IR(Address))	C_8
	t2: MBR ← Memory	C_5, C_R
	t3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t1: MBR ← (PC)	C_1
	t2: MAR ← Save-address PC ← Routine-address	
	t3: Memory ← (MBR)	C_{12}, C_W

C_R = Read control signal to system bus.
C_W = Write control signal to system bus.

operations (e.g., positive, overflow, etc.). It never gets to see the data being processed or the actual results produced. And it controls everything with a few control signals to points within the processor and a few control signals to the system bus.

Internal Processor Organization

Figure 16.5 indicates the use of a variety of data paths. The complexity of this type of organization should be clear. More typically, some sort of internal bus arrangement, as was suggested in Figure 12.2, will be used.

Using an internal processor bus, Figure 16.5 can be rearranged as shown in Figure 16.6. A single internal bus connects the ALU and all processor registers. Gates and control signals are provided for movement of data onto and off the bus

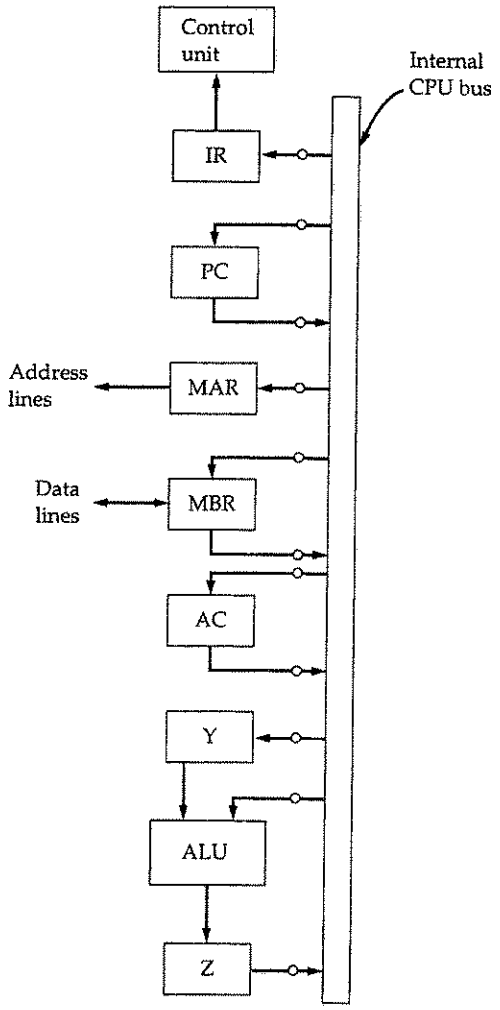

Figure 16.6 CPU with Internal Bus

from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

Two new registers, labeled Y and Z, have been added to the organization. These are needed for the proper operation of the ALU. When an operation involving two operands is performed, one can be obtained from the internal bus, but the other must be obtained from another source. The AC could be used for this purpose, but this limits the flexibility of the system and would not work with a processor with multiple general-purpose registers. Register Y provides temporary storage for the other input. The ALU is a combinatorial circuit (see Appendix A) with no internal storage. Thus, when control signals activate an ALU function, the input to the ALU is transformed to the output. Thus, the output of the ALU cannot be directly connected to the bus, because this output would feed back to the input. Register Z provides temporary output storage. With this arrangement, an operation to add a value from memory to the AC would have the following steps:

t_1: MAR ← (IR(address))
t_2: MBR ← Memory
t_3: Y ← (MBR)
t_4: Z ← (AC) + (Y)
t_5: AC ← (Z)

Other organizations are possible, but, in general, some sort of internal bus or set of internal buses is used. The use of common data paths simplifies the interconnection layout and the control of the processor. Another practical reason for the use of an internal bus is to save space. Especially for microprocessors, which may occupy only a 1/4-inch square piece of silicon, space occupied by interregister connections must be minimized.

The Intel 8085

To illustrate some of the concepts introduced thus far in this chapter, let us consider the Intel 8085. Its organization is shown in Figure 16.7. Several key components that may not be self-explanatory are as follows:

- **Incrementer/decrementer address latch:** Logic that can add 1 to or subtract 1 from the contents of the stack pointer or program counter. This saves time by avoiding the use of the ALU for this purpose.
- **Interrupt control:** This module handles multiple levels of interrupt signals.
- **Serial I/O control:** This module interfaces to devices that communicate 1 bit at a time.

Table 16.2 describes the external signals into and out of the 8085. These are linked to the external system bus. These signals are the interface between the 8085 processor and the rest of the system (Figure 16.8).

The control unit is identified as having two components labeled (1) instruction decoder and machine cycle encoding and (2) timing and control. A discussion of the first component is deferred until the next section. The essence of the control unit is the timing and control module. This module includes a clock and accepts as inputs

KEY POINTS

- An alternative to a hardwired control unit is a microprogrammed control unit, in which the logic of the control unit is specified by a microprogram. A microprogram consists of a sequence of instructions in a microprogramming language. These are very simple instructions that specify micro-operations.
- A microprogrammed control unit is a relatively simple logic circuit that is capable of (1) sequencing through microinstructions and (2) generating control signals to execute each microinstruction.
- As in a hardwired control unit, the control signals generated by a microinstruction are used to cause register transfers and ALU operations.

The term *microprogram* was first coined by M. V. Wilkes in the early 1950s [WILK51]. Wilkes proposed an approach to control unit design that was organized and systematic and avoided the complexities of a hardwired implementation. The idea intrigued many researchers but appeared unworkable because it would require a fast, relatively inexpensive control memory.

The state of the microprogramming art was reviewed by *Datamation* in its February 1964 issue. No microprogrammed system was in wide use at that time, and one of the papers [HILL64] summarized the then-popular view that the future of microprogramming "is somewhat cloudy. None of the major manufacturers has evidenced interest in the technique, although presumably all have examined it."

This situation changed dramatically within a very few months. IBM's System/360 was announced in April, and all but the largest models were microprogrammed. Although the 360 series predated the availability of semiconducter ROM, the advantages of microprogramming were compelling enough for IBM to make this move. Since then, microprogramming has become an increasingly popular vehicle for a variety of applications, one of which is the use of microprogramming to implement the control unit of a processor. That application is examined in this chapter.

17.1 BASIC CONCEPTS

Microinstructions

The control unit seems a reasonably simple device. Nevertheless, to implement a control unit as an interconnection of basic logic elements is no easy task. The design must include logic for sequencing through micro-operations, for executing micro-operations, for interpreting opcodes, and for making decisions based on ALU flags. It is difficult to design and test such a piece of hardware. Furthermore, the design is relatively inflexible. For example, it is difficult to change the design if one wishes to add a new machine instruction.

An alternative, which is quite common in contemporary CISC processors, is to implement a microprogrammed control unit.

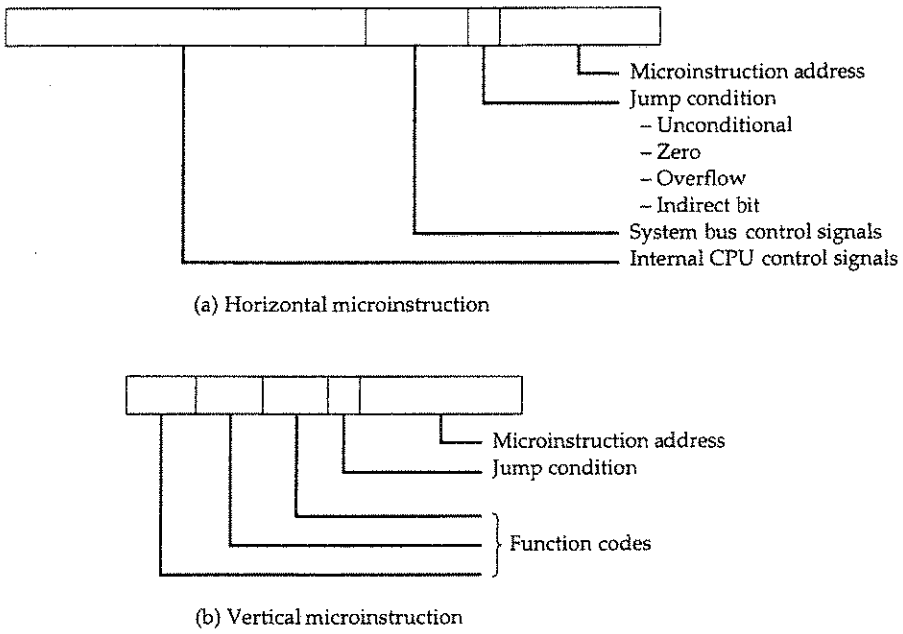

Figure 17.1 Typical Microinstruction Formats

Consider again Table 16.1. In addition to the use of control signals, each micro-operation is described in symbolic notation. This notation looks suspiciously like a programming language. In fact it is a language, known as a *microprogramming language.* Each line describes a set of micro-operations occurring at one time and is known as a *microinstruction.* A sequence of instructions is known as a *microprogram,* or *firmware.* This latter term reflects the fact that a microprogram is midway between hardware and software. It is easier to design in firmware than hardware, but it is more difficult to write a firmware program than a software program.

How can we use the concept of microprogramming to implement a control unit? Consider that for each micro-operation, all that the control unit is allowed to do is generate a set of control signals. Thus, for any micro-operation, each control line emanating from the control unit is either on or off. This condition can, of course, be represented by a binary digit for each control line. So we could construct a *control word* in which each bit represents one control line. Then each micro-operation would be represented by a different pattern of 1s and 0s in the control word.

Suppose we string together a sequence of control words to represent the sequence of micro-operations performed by the control unit. Next, we must recognize that the sequence of micro-operations is not fixed. Sometimes we have an indirect cycle; sometimes we do not. So let us put our control words in a memory, with each word having a unique address. Now add an address field to each control word, indicating the location of the next control word to be executed if a certain condition is true (e.g., the indirect bit in a memory-reference instruction is 1). Also, add a few bits to specify the condition.

The result is known as a *horizontal microinstruction*, an example of which is shown in Figure 17.1a. The format of the microinstruction or control word is as

follows. There is one bit for each internal processor control line and one bit for each system bus control line. There is a condition field indicating the condition under which there should be a branch, and there is a field with the address of the microinstruction to be executed next when a branch is taken. Such a microinstruction is interpreted as follows:

1. To execute this microinstruction, turn on all the control lines indicated by a 1 bit; leave off all control lines indicated by a 0 bit. The resulting control signals will cause one or more micro-operations to be performed.
2. If the condition indicated by the condition bits is false, execute the next microinstruction in sequence.
3. If the condition indicated by the condition bits is true, the next microinstruction to be executed is indicated in the address field.

Figure 17.2 shows how these control words or microinstructions could be arranged in a *control memory*. The microinstructions in each routine are to be executed sequentially. Each routine ends with a branch or jump instruction indicating where to go next. There is a special execute cycle routine whose only purpose is to signify that one of the machine instruction routines (AND, ADD, and so on) is to be executed next, depending on the current opcode.

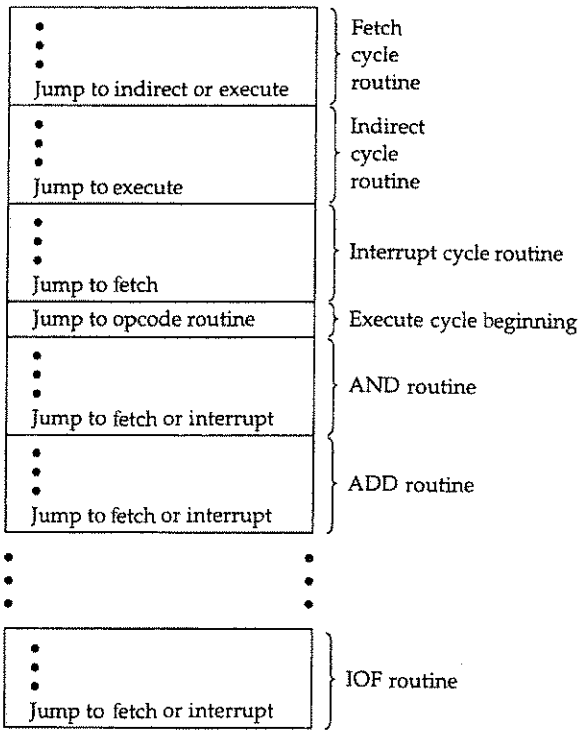

Figure 17.2 Organization of Control Memory

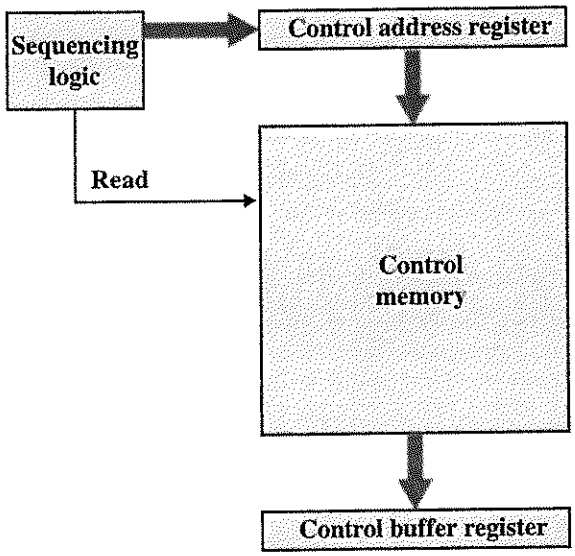

Figure 17.3 Control Unit Microarchitecture

The control memory of Figure 17.2 is a concise description of the complete operation of the control unit. It defines the sequence of micro-operations to be performed during each cycle (fetch, indirect, execute, interrupt), and it specifies the sequencing of these cycles. If nothing else, this notation would be a useful device for documenting the functioning of a control unit for a particular computer. But it is more than that. It is also a way of implementing the control unit.

Microprogrammed Control Unit

The control memory of Figure 17.2 contains a program that describes the behavior of the control unit. It follows that we could implement the control unit by simply executing that program.

Figure 17.3 shows the key elements of such an implementation. The set of microinstructions is stored in the *control memory*. The *control address register* contains the address of the next microinstruction to be read. When a microinstruction is read from the control memory, it is transferred to a *control buffer register*. The left-hand portion of that register (see Figure 17.1a) connects to the control lines emanating from the control unit. Thus, *reading* a microinstruction from the control memory is the same as *executing* that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command.

Let us examine this structure in greater detail, as depicted in Figure 17.4. Comparing this with Figure 16.4, we see that the control unit still has the same inputs (IR, ALU flags, clock) and outputs (control signals). The control unit functions as follows:

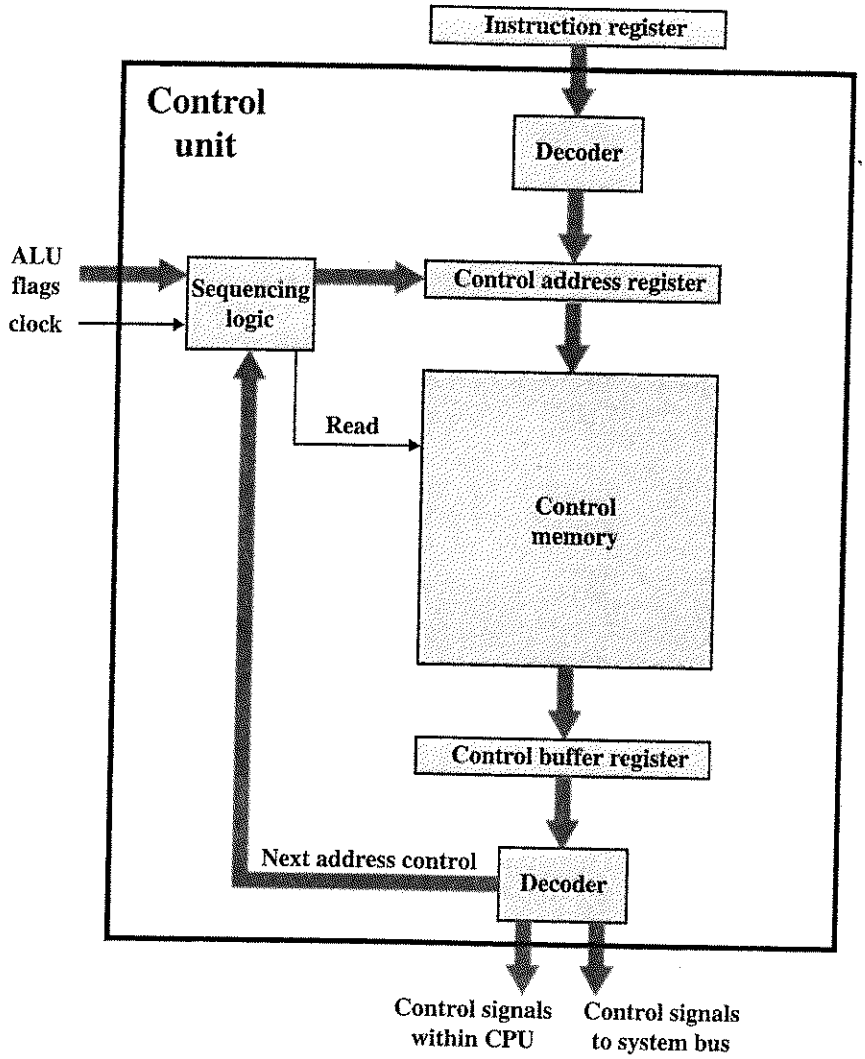

Figure 17.4 Functioning of Microprogrammed Control Unit

1. To execute an instruction, the sequencing logic unit issues a READ command to the control memory.
2. The word whose address is specified in the control address register is read into the control buffer register.
3. The content of the control buffer register generates control signals and next-address information for the sequencing logic unit.
4. The sequencing logic unit loads a new address into the control address register based on the next-address information from the control buffer register and the ALU flags.

All this happens during one clock pulse.

The last step just listed needs elaboration. At the conclusion of each microinstruction, the sequencing logic unit loads a new address into the control address register. Depending on the value of the ALU flags and the control buffer register, one of three decisions is made:

- **Get the next instruction:** Add 1 to the control address register.
- **Jump to a new routine based on a jump microinstruction:** Load the address field of the control buffer register into the control address register.
- **Jump to a machine instruction routine:** Load the control address register based on the opcode in the IR.

Figure 17.4 shows two modules labeled *decoder*. The upper decoder translates the opcode of the IR into a control memory address. The lower decoder is not used for horizontal microinstructions but is used for *vertical microinstructions* (Figure 17.1b). As was mentioned, in a horizontal microinstruction every bit in the control field attaches to a control line. In a vertical microinstruction, a code is used for each action to be performed [e.g., MAR $\leftarrow$ (PC)], and the decoder translates this code into individual control signals. The advantage of vertical microinstructions is that they are more compact (fewer bits) than horizontal microinstructions, at the expense of a small additional amount of logic and time delay.

Wilkes Control

As was mentioned, Wilkes first proposed the use of a microprogrammed control unit in 1951 [WILK51]. This proposal was subsequently elaborated into a more detailed design [WILK53]. It is instructive to examine this seminal proposal.

The configuration proposed by Wilkes is depicted in Figure 17.5. The heart of the system is a matrix partially filled with diodes. During a machine cycle, one row of the matrix is activated with a pulse. This generates signals at those points where a diode is present (indicated by a dot in the diagram). The first part of the row generates the control signals that control the operation of the processor. The second part generates the address of the row to be pulsed in the next machine cycle. Thus, each row of the matrix is one microinstruction, and the layout of the matrix is the control memory.

At the beginning of the cycle, the address of the row to be pulsed is contained in Register I. This address is the input to the decoder, which, when activated by a clock pulse, activates one row of the matrix. Depending on the control signals, either the opcode in the instruction register or the second part of the pulsed row is passed into Register II during the cycle. Register II is then gated to Register I by a clock pulse. Alternating clock pulses are used to activate a row of the matrix and to transfer from Register II to Register I. The two-register arrangement is needed because the decoder is simply a combinatorial circuit; with only one register, the output would become the input during a cycle, causing an unstable condition.

This scheme is very similar to the horizontal microprogramming approach described earlier (Figure 17.1a). The main difference is this: In the previous description, the control address register could be incremented by one to get the next address. In the Wilkes scheme, the next address is contained in the microinstruction.

Table 17.2 *(continued)*

	Arithmetical Unit	Control Register Unit	Conditional Flip-Flop		Next Micro-instruction	
			Set	Use	0	1
28	B to D	E to G	(1)B_1		29	
29	D to B (R)	(G – '1') to E			30	
30	C to D (R)		(2)E_5	1	31	32
31	D to C			2	28	33
32	(D + A) to C			2	28	33
33	B to D		(1)B_1		34	
34	D to B (R)				35	
35	C to D (R)			1	36	37
36	D to C				0	
37	(D – A) to C				0	

*Right shift. The switching circuits in the arithmetic unit are arranged so that the least significant digit of the register C is placed in the most significant place of register B during right shift micro-operations, and the most significant digit of register C (sign digit) is repeated (thus making the correction for negative numbers).

† Left shift. The switching circuits are similarly arranged to pass the most significant digit of register B to the least significant place of register C during left shift micro-operations.

The principal disadvantage of a microprogrammed unit is that it will be somewhat slower than a hardwired unit of comparable technology. Despite this, microprogramming is the dominant technique for implementing control units in contemporary CISC, due to its ease of implementation. RISC processors, with their simpler instruction format, typically use hardwired control units. We now examine the microprogrammed approach in greater detail.

17.2 MICROINSTRUCTION SEQUENCING

The two basic tasks performed by a microprogrammed control unit are as follows:

- **Microinstruction sequencing:** Get the next microinstruction from the control memory.
- **Microinstruction execution:** Generate the control signals needed to execute the microinstruction.

In designing a control unit, these tasks must be considered together, because both affect the format of the microinstruction and the timing of the control unit. In this section, we will focus on sequencing and say as little as possible about format and timing issues. These issues are examined in more detail in the next section.

Design Considerations

Two concerns are involved in the design of a microinstruction sequencing technique: the size of the microinstruction and the address-generation time. The first concern is obvious; minimizing the size of the control memory reduces the cost of that component. The second concern is simply a desire to execute microinstructions as fast as possible.

In executing a microprogram, the address of the next microinstruction to be executed is in one of these categories:

- Determined by instruction register
- Next sequential address
- Branch

The first category occurs only once per instruction cycle, just after an instruction is fetched. The second category is the most common in most designs. However, the design cannot be optimized just for sequential access. Branches, both conditional and unconditional, are a necessary part of a microprogram. Furthermore, microinstruction sequences tend to be short; one out of every three or four microinstructions could be a branch [SIEW82]. Thus, it is important to design compact, time-efficient techniques for microinstruction branching.

Sequencing Techniques

Based on the current microinstruction, condition flags, and the contents of the instruction register, a control memory address must be generated for the next microinstruction. A wide variety of techniques have been used. We can group them into three general categories, as illustrated in Figures 17.6 to 17.8. These categories are based on the format of the address information in the microinstruction:

- Two address fields
- Single address field
- Variable format

The simplest approach is to provide two address fields in each microinstruction. Figure 17.6 suggests how this information is to be used. A multiplexer is provided that serves as a destination for both address fields plus the instruction register. Based on an address-selection input, the multiplexer transmits either the opcode or one of the two addresses to the control address register (CAR). The CAR is subsequently decoded to produce the next microinstruction address. The address-selection signals are provided by a branch logic module whose input consists of control unit flags plus bits from the control portion of the microinstruction.

Although the two-address approach is simple, it requires more bits in the microinstruction than other approaches. With some additional logic, savings can be achieved. A common approach is to have a single address field (Figure 17.7). With this approach, the options for next address are as follows:

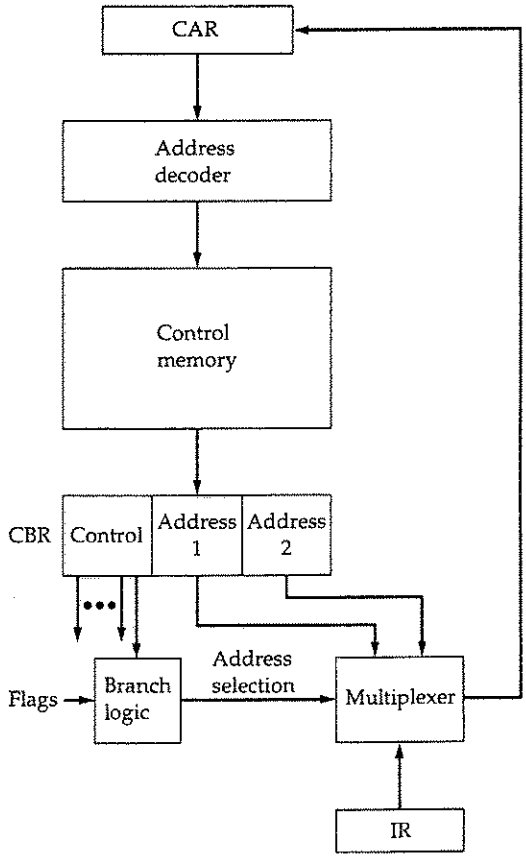

Figure 17.6 Branch Control Logic, Two Address Fields

- Address field
- Instruction register code
- Next sequential address

The address-selection signals determine which option is selected. This approach reduces the number of address fields to one. Note, however, that the address field often will not be used. Thus, there is some inefficiency in the microinstruction coding scheme.

Another approach is to provide for two entirely different microinstruction formats (Figure 17.8). One bit designates which format is being used. In one format, the remaining bits are used to activate control signals. In the other format, some bits drive the branch logic module, and the remaining bits provide the address. With the first format, the next address is either the next sequential address or an address derived from the instruction register. With the second format, either a conditional or unconditional branch is being specified. One disadvantage of this approach is that one entire cycle is consumed with each branch microinstruction. With the other

approaches, address generation occurs as part of the same cycle as control signal generation, minimizing control memory accesses.

The approaches just described are general. Specific implementations will often involve a variation or combination of these techniques.

Address Generation

We have looked at the sequencing problem from the point of view of format considerations and general logic requirements. Another viewpoint is to consider the various ways in which the next address can be derived or computed.

Table 17.3 lists the various address generation techniques. These can be divided into explicit techniques, in which the address is explicitly available in the microinstruction, and implicit techniques, which require additional logic to generate the address.

We have essentially dealt with the explicit techniques. With a two-field approach, two alternative addresses are available with each microinstruction. Using either a single address field or a variable format, various branch instructions can be implemented. A conditional branch instruction depends on the following types of information:

- ALU flags
- Part of the opcode or address mode fields of the machine instruction

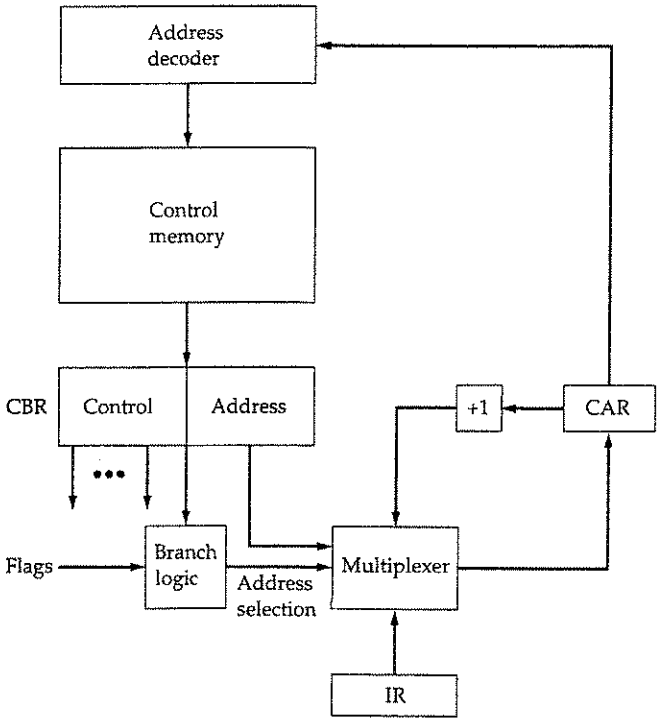

Figure 17.7 Branch Control Logic, Single Address Field

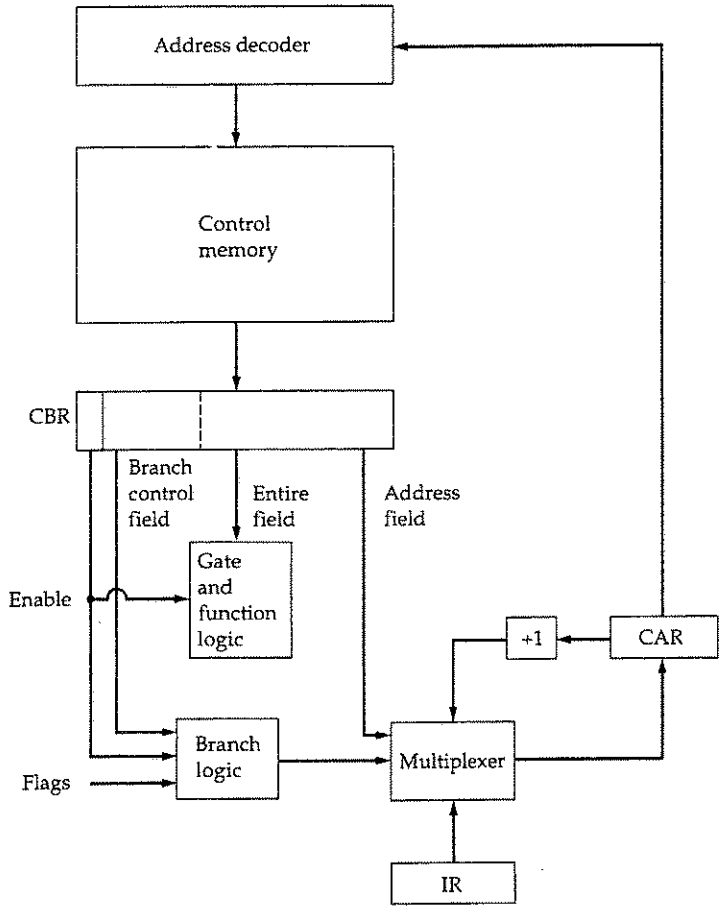

Figure 17.8 Branch Control Logic, Variable Format

- Parts of a selected register, such as the sign bit
- Status bits within the control unit

Several implicit techniques are also commonly used. One of these, mapping, is required with virtually all designs. The opcode portion of a machine instruction must be mapped into a microinstruction address. This occurs only once per instruction cycle.

Table 17.3 Microinstruction Address Generation Techniques

Explicit	Implicit
Two-field	Mapping
Unconditional branch	Addition
Conditional branch	Residual control

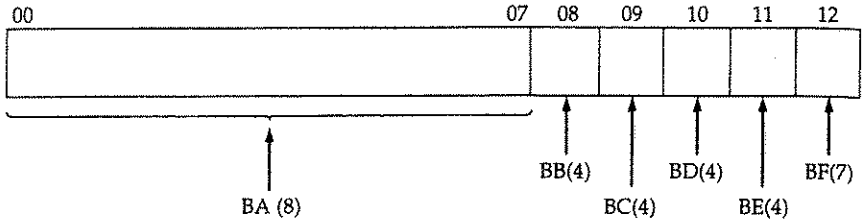

Figure 17.9 IBM 3033 Control Address Register

A common implicit technique is one that involves combining or adding two portions of an address to form the complete address. This approach was taken for the IBM S/360 family [TUCK67] and used on many of the S/370 models. We will use the IBM 3033 as an example.

The control address register on the IBM 3033 is 13 bits long and is illustrated in Figure 17.9. Two parts of the address can be distinguished. The highest-order 8 bits (00–07) normally do not change from one microinstruction cycle to the next. During the execution of a microinstruction, these 8 bits are copied directly from an 8-bit field of the microinstruction (the BA field) into the highest-order 8 bits of the control address register. This defines a block of 32 microinstructions in control memory. The remaining 5 bits of the control address register are set to specify the specific address of the microinstruction to be fetched next. Each of these bits is determined by a 4-bit field (except one is a 7-bit field) in the current microinstruction; the field specifies the condition for setting the corresponding bit. For example, a bit in the control address register might be set to 1 or 0 depending on whether a carry occurred on the last ALU operation.

The final approach listed in Table 17.3 is termed *residual control.* This approach involves the use of a microinstruction address that has previously been saved in temporary storage within the control unit. For example, some microinstruction sets come equipped with a subroutine facility. An internal register or stack of registers is used to hold return addresses. An example of this approach is taken on the LSI-11, which we now examine.

LSI-11 Microinstruction Sequencing

The LSI-11 is a microcomputer version of a PDP-11, with the main components of the system residing on a single board. The LSI-11 is implemented using a microprogrammed control unit [SEBE76].

The LSI-11 makes use of a 22-bit microinstruction and a control memory of 2K 22-bit words. The next microinstruction address is determined in one of five ways:

- **Next sequential address:** In the absence of other instructions, the control unit's control address register is incremented by 1.
- **Opcode mapping:** At the beginning of each instruction cycle, the next microinstruction address is determined by the opcode.
- **Subroutine facility:** Explained presently.

- **Interrupt testing:** Certain microinstructions specify a test for interrupts. If an interrupt has occurred, this determines the next microinstruction address.
- **Branch:** Conditional and unconditional branch microinstructions are used.

A one-level subroutine facility is provided. One bit in every microinstruction is dedicated to this task. When the bit is set, an 11-bit return register is loaded with the updated contents of the control address register. A subsequent microinstruction that specifies a return will cause the control address register to be loaded from the return register.

The return is one form of unconditional branch instruction. Another form of unconditional branch causes the bits of the control address register to be loaded from 11 bits of the microinstruction. The conditional branch instruction makes use of a 4-bit test code within the microinstruction. This code specifies testing of various ALU condition codes to determine the branch decision. If the condition is not true, the next sequential address is selected. If it is true, the 8 lowest-order bits of the control address register are loaded from 8 bits of the microinstruction. This allows branching within a 256-word page of memory.

As can be seen, the LSI-11 includes a powerful address sequencing facility within the control unit. This allows the microprogrammer considerable flexibility and can ease the microprogramming task. On the other hand, this approach requires more control unit logic than do simpler capabilities.

17.3 MICROINSTRUCTION EXECUTION

The microinstruction cycle is the basic event on a microprogrammed processor. Each cycle is made up of two parts: fetch and execute. The fetch portion is determined by the generation of a microinstruction address, and this was dealt with in the preceding section. This section deals with the execution of a microinstruction.

Recall that the effect of the execution of a microinstruction is to generate control signals. Some of these signals control points internal to the processor. The remaining signals go to the external control bus or other external interface. As an incidental function, the address of the next microinstruction is determined.

The preceding description suggests the organization of a control unit shown in Figure 17.10. This slightly revised version of Figure 17.4 emphasizes the focus of this section. The major modules in this diagram should by now be clear. The sequencing logic module contains the logic to perform the functions discussed in the preceding section. It generates the address of the next microinstruction, using as inputs the instruction register, ALU flags, the control address register (for incrementing), and the control buffer register. The last may provide an actual address, control bits, or both. The module is driven by a clock that determines the timing of the microinstruction cycle.

The control logic module generates control signals as a function of some of the bits in the microinstruction. It should be clear that the format and content of the microinstruction will determine the complexity of the control logic module.

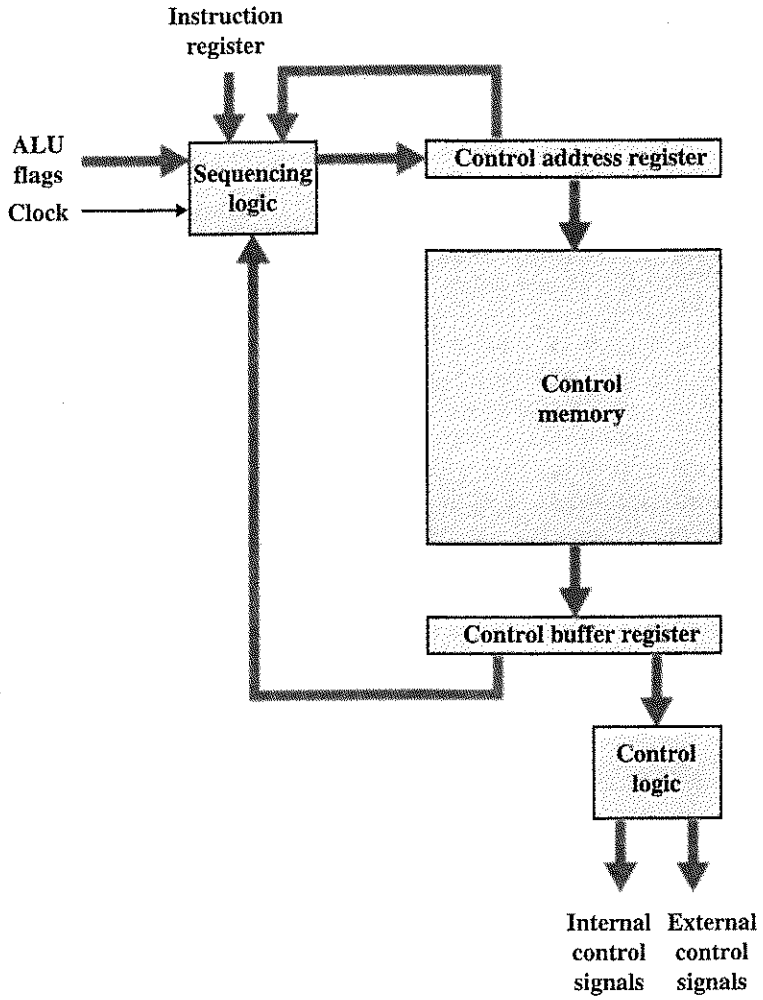

Figure 17.10 Control Unit Organization

A Taxonomy of Microinstructions

Microinstructions can be classified in a variety of ways. Distinctions that are commonly made in the literature include the following:

- Vertical/horizontal
- Packed/unpacked
- Hard/soft microprogramming
- Direct/indirect encoding

All of these bear on the format of the microinstruction. None of these terms has been used in a consistent, precise way in the literature. However, an examination of these pairs of qualities serves to illuminate microinstruction design alternatives. In