Chapter 19 Control Unit Operation and Microprogrammed Control

Learning Objectives

After studying this chapter, you should be able to:

- Explain the concept of micro-operations and define the principal instruction cycle phases in terms of micro-operations.
- Discuss how micro-operations are organized to control a processor.
- Understand hardwired control unit organization.
- Present an overview of the basic concepts of microprogrammed control.
- Understand the difference between hardwired control and microprogrammed control.

In ***Chapter 13****, weWe pointed out that a machine instruction set goes a long way toward defining the processor. If we know the machine instruction set, including an* *understanding of the effect of each opcode and an understanding of the addressing modes, and if we know the set of user-visible registers, then we know the functions that the processor must perform. This is not the complete picture. We must know the external interfaces, usually through a bus, and how interrupts are handled. With this line of reasoning, the following list of those things needed to specify the function of a processor emerges:*

1. *Operations (opcodes)*
2. *Addressing modes*
3. *Registers*
4. *I/O module interface*
5. *Memory module interface*
6. *Interrupts*

This list, though general, is rather complete. Items 1 through 3 are defined by the instruction set. Items 4 and 5 are typically defined by specifying the system bus. Item 6 is defined partially by the system bus and partially by the type of support the processor offers to the operating system.

This list of six items might be termed the functional requirements for a processor. They determine what a processor must do. This is what occupied us in previous chapters. 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.

19.1 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 16**, 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 19.1** 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, execute, interrupt). The execution of each subcycle involves one or more shorter operations, that is, micro-operations.

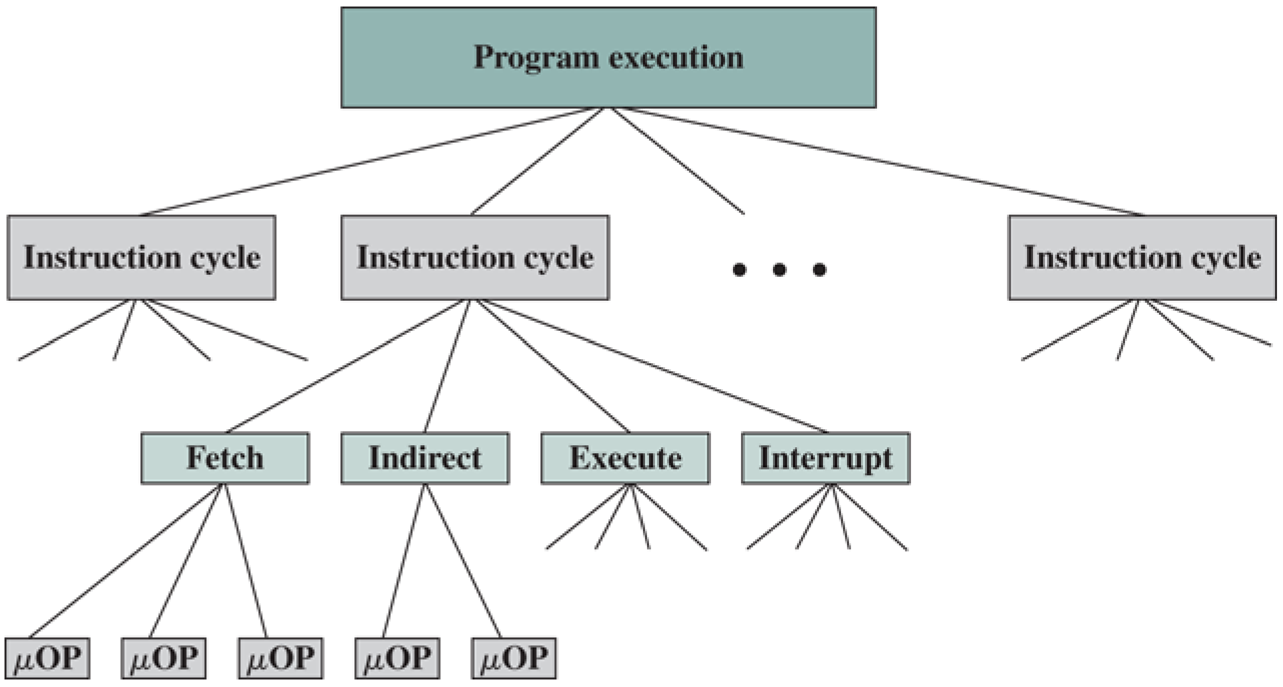


Figure 19.1 Constituent Elements of a Program Execution

Micro-operations are the functional, or atomic, operations of a processor. In this section, we will examine micro-operations to gain an understanding of how the events of any instruction cycle can be described as a sequence of such micro-operations. A simple example will be used. In the remainder of this chapter, we then show how the concept of micro-operations serves as a guide to the design of the control unit.

The Fetch Cycle

We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. For purposes of discussion, we assume the organization depicted in Figure 16.5 (*Data Flow, Fetch Cycle*). We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory (*Data Flow, Fetch Cycle*). Four registers are involved:

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

Let us look at the sequence of events for the fetch cycle from the point of view of its effect on the processor registers. An example appears in Figure 19.2. At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100. The first step is to move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus. The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by the instruction length to get ready for the next instruction. Because these two actions (read word from memory, increment PC) do not interfere with each other, we can do them simultaneously to save time. The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

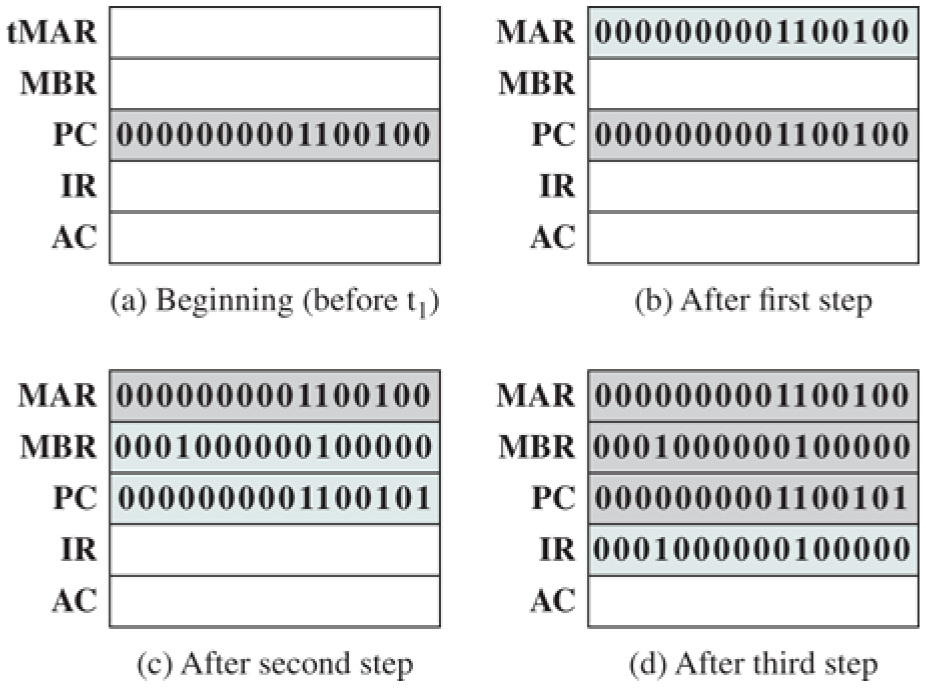


Figure 19.2 Sequence of Events, Fetch Cycle

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

```
t1: MAR  ←   (PC)
t2: MBR  ←  Memory
```

```
    PC   ←   (PC) + I
t3: IR   ←   (MBR)
```

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 $(\text{MAR} \leftarrow (\text{PC}))$ must precede $(\text{MBR} \leftarrow \text{Memory})$ 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 $(\text{MBR} \leftarrow \text{Memory})$ and $(\text{IR} \leftarrow \text{MBR})$ 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** (*Example of Program Execution*). Whereas micro-operations are ignored in that figure, this 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 16.6** (*Data Flow*, *Indirect Cycle*) and includes the following micro-operations:

```
t1: MAR ←  (IR(Address))
```

```
t2: MBR ←  Memory
t3: 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 16.7** (*Data Flow, Interrupt Cycle*). We have

```
t1: MBR ←  (PC)
t2: MAR ←  Save_Address
    PC  ← Routine_Address
t3: 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. Because of the variety of opcodes, there are a number of different sequences of micro-operations that can occur. The control unit examines the opcode and generates a sequence of micro-operations based on the value of the opcode. This is referred to as instruction decoding.

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 $(\text{MBR}) = 0$. 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 $\text{X} + \text{I}$. The saved address will later be used for return. This is a straightforward technique for supporting 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 19.3**. 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:

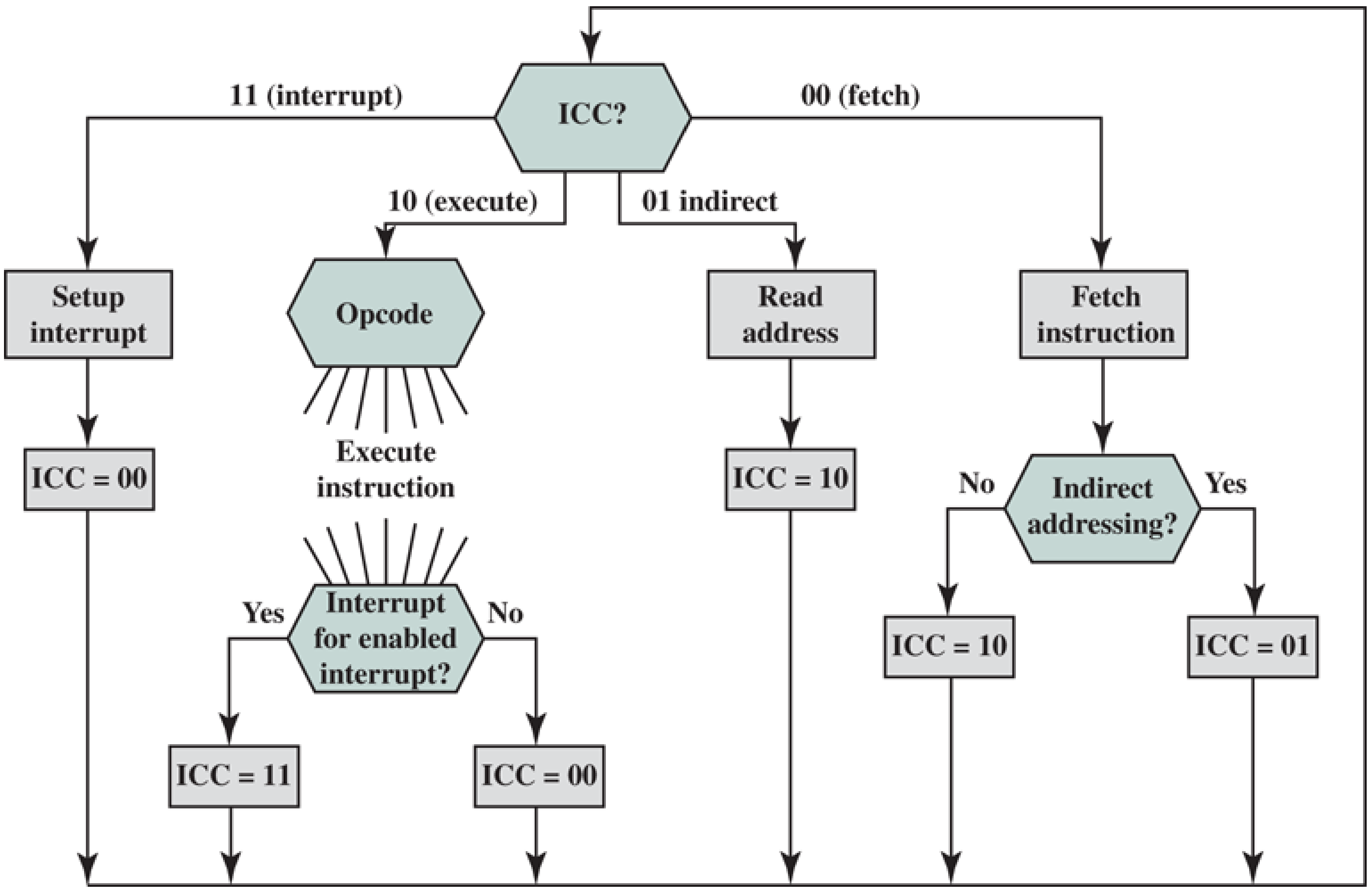


Figure 19.3 Flowchart for Instruction Cycle

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 16.3**, *The Instruction Cycle*). For both the fetch and execute cycles, the next cycle depends on the state of the system.

Thus, the flowchart of **Figure 19.3** 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.

19.2 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 19.1**, 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 19.3** and **19.4**. The remainder of this section is concerned with the interaction between the control unit and the other elements of the processor.

Figure 19.4 is a general model of the control unit, showing all of its inputs and outputs. The inputs are:

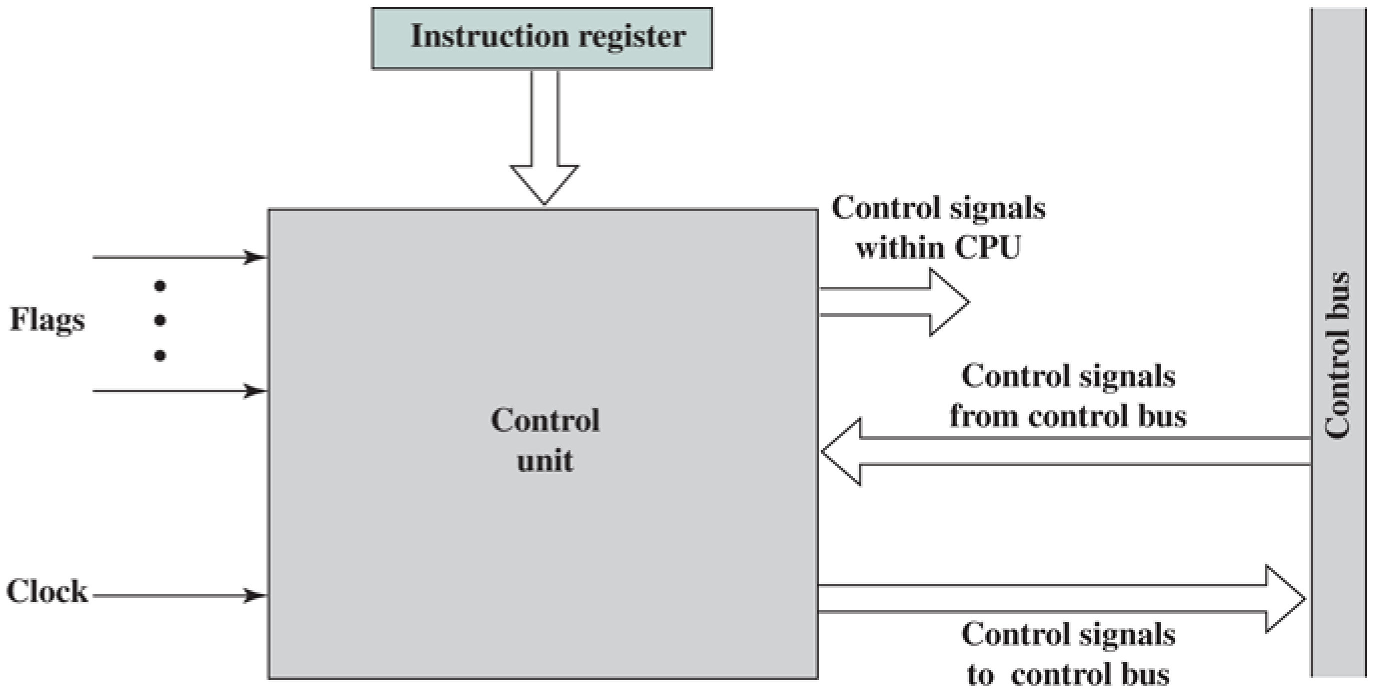

Instruction register
Control signals within CPU
Flags
Control signals from control bus
Control bus
Control unit
Clock
Control signals to control bus

Figure 19.4 Block Diagram of the Control Unit

- **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 and addressing mode of the current instruction are 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.

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.

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.

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 19.5** illustrates the example. This is a simple processor with a single accumulator (AC). 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 IR, 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:

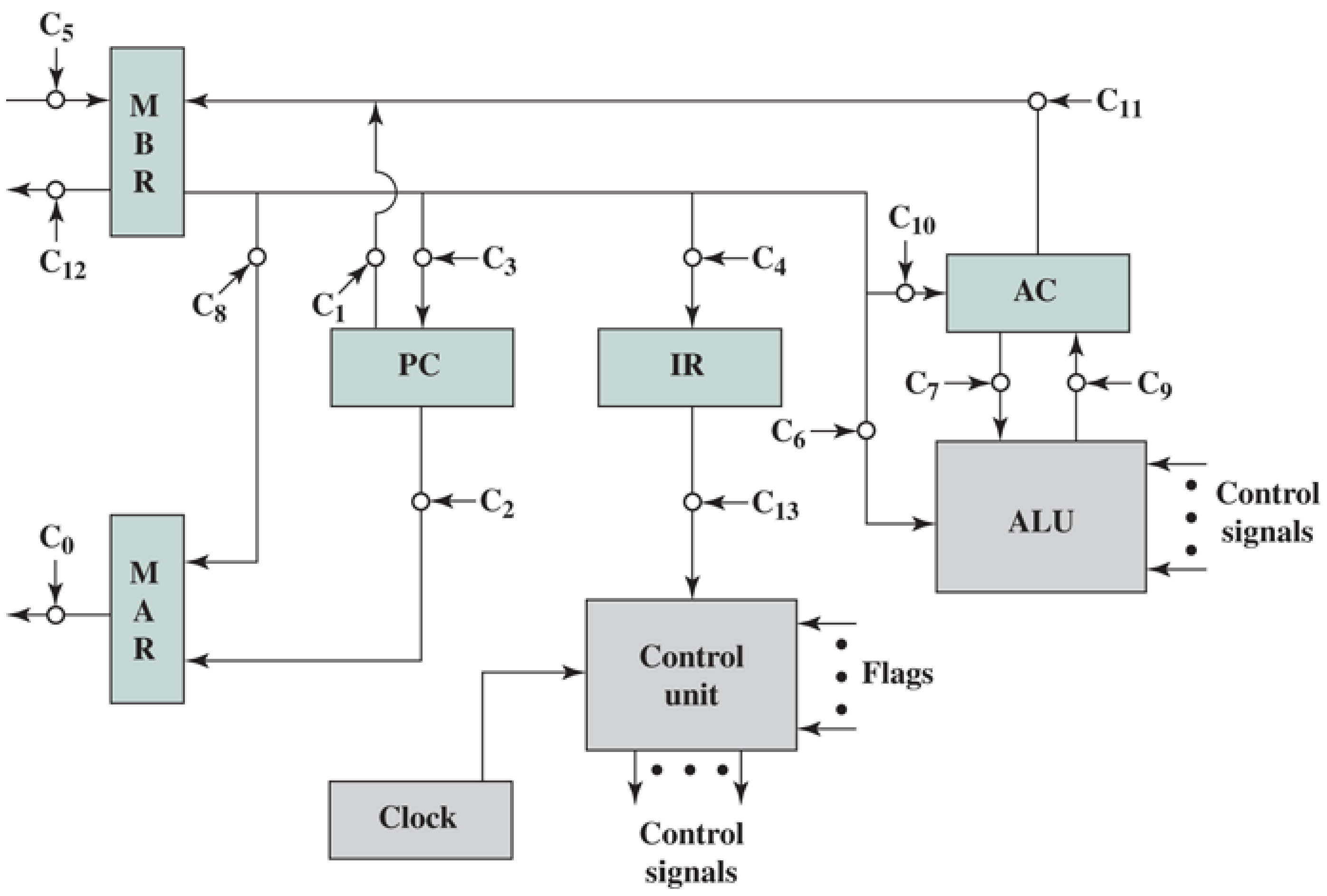


Figure 19.5 Data Paths and Control Signals

- **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 IR. For each path to be controlled, there is a switch (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 circuits 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 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 19.1 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.

Table 19.1 Micro-operations and Control Signals

C_R = Read control signal to system bus.

C_W = Write control signal to system bus.

	Micro-operations	Active Control Signals
Fetch:	$t_1: \text{MAR} \leftarrow (\text{PC})$	C_2
	t_2: $\text{MBR} \leftarrow \text{Memory}$ $\text{PC} \leftarrow (\text{PC}) + 1$	C_5, C_R

	$t_3 : IR \leftarrow (MBR)$	C_4
Indirect:	$t_1 : MAR \leftarrow (IR(Address))$	C_8
	$t_2 : MBR \leftarrow Memory$	C_5, C_R
	$t_3 : IR(Address) \leftarrow (MBR(Address))$	C_4
Interrupt:	$t_1 : MBR \leftarrow (PC)$	C_1
	$t_2 :$ MAR $\leftarrow$ Save-address PC $\leftarrow$ Routine-address	
	$t_3 : Memory \leftarrow (MBR)$	C_{12}, C_W

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 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 19.5 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 16.1** (*Internal Structure of the CPU*), , some sort of internal bus arrangement, as was suggested in Internal Structure of the CPU will be used.

Using an internal processor bus, **Figure 19.5** can be rearranged as shown in **Figure 19.6**. 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 from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

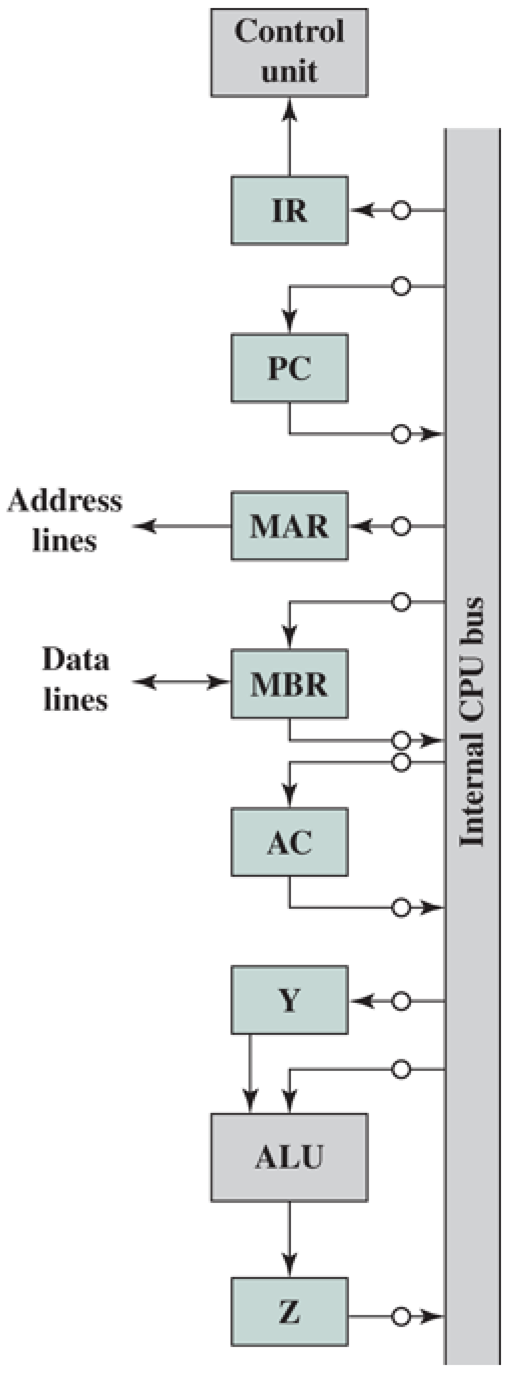


Figure 19.6 CPU with Internal Bus

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 **Chapter 12**) with no internal storage. Thus, when control signals activate an ALU function, the input to the ALU is transformed to the output. Therefore, 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:

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

```
t4: Z ← (AC) + (Y)
t5: 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.

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 19.7**. Several key components that may not be self-explanatory are:

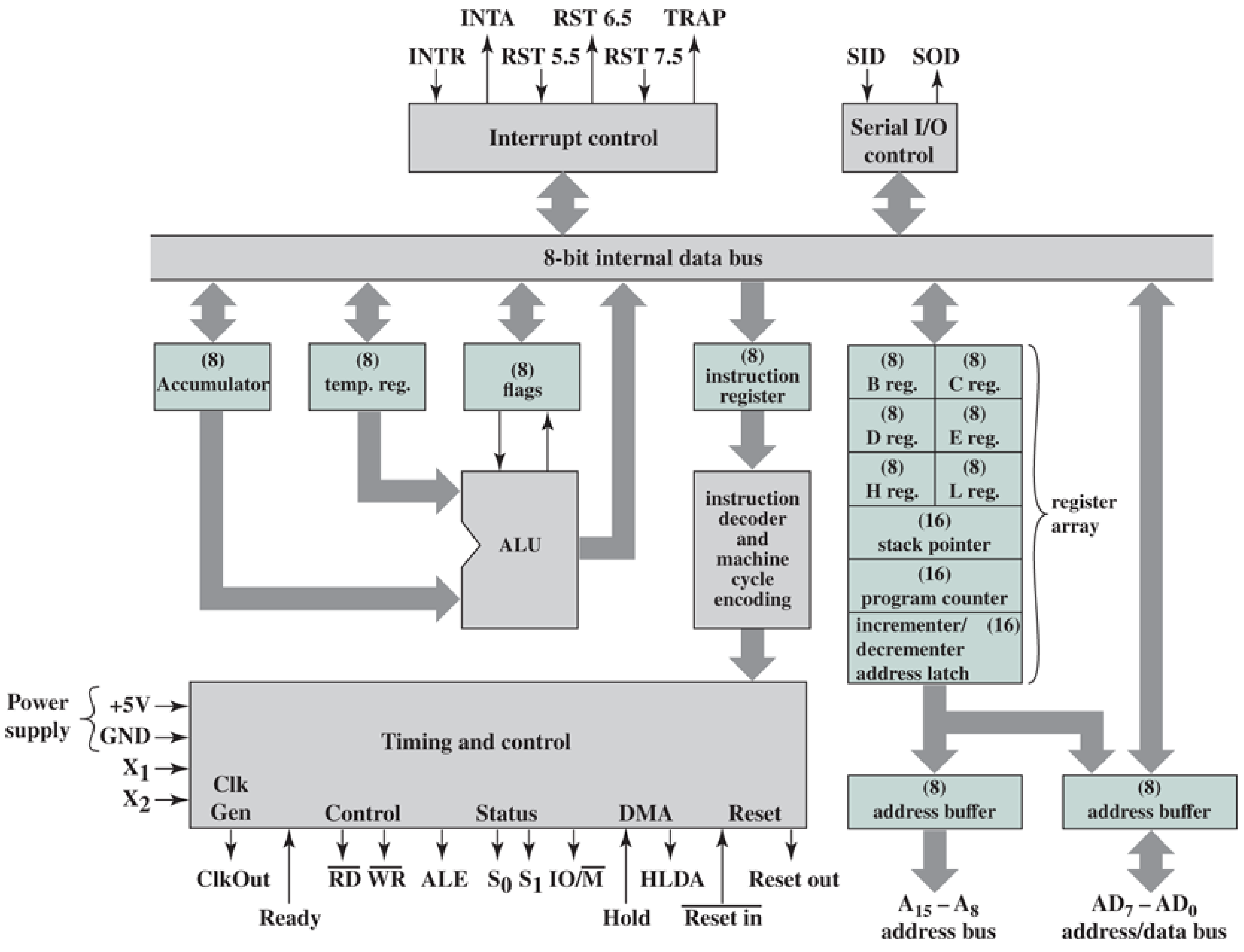


Figure 19.7 Intel 8085 CPU Block Diagram

- **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 19.2 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 19.8**).

Table 19.2 Intel 8085 External Signals

Address and Data Signals
High Address (A15–A8) The high-order 8 bits of a 16-bit address.
Address/Data (AD7–AD0) The lower-order 8 bits of a 16-bit address or 8 bits of data. This multiplexing saves on pins.
Serial Input Data (SID) A single-bit input to accommodate devices that transmit serially (one bit at a time).
Serial Output Data (SOD) A single-bit output to accommodate devices that receive serially.
Timing and Control Signals
CLK (OUT) The system clock. The CLK signal goes to peripheral chips and synchronizes their timing.
X1, X2 These signals come from an external crystal or other device to drive the internal clock generator.
Address Latch Enabled (ALE) Occurs during the first clock state of a machine cycle and causes peripheral chips to store the address lines. This allows the address module (e.g., memory, I/O) to recognize that it is being addressed.
Status (S0, S1) Control signals used to indicate whether a read or write operation is taking place.
IO/M Used to enable either I/O or memory modules for read and write operations.

Read Control (RD) Indicates that the selected memory or I/O module is to be read and that the data bus is available for data transfer.
Write Control (WR) Indicates that data on the data bus is to be written into the selected memory or I/O location.
Memory and I/O Initiated Symbols
Hold Requests the CPU to relinquish control and use of the external system bus. The CPU will complete execution of the instruction presently in the IR and then enter a hold state, during which no signals are inserted by the CPU to the control, address, or data buses. During the hold state, the bus may be used for DMA operations.
Hold Acknowledge (HOLDA) This control unit output signal acknowledges the HOLD signal and indicates that the bus is now available.
READY Used to synchronize the CPU with slower memory or I/O devices. When an addressed device asserts READY, the CPU may proceed with an input (DBIN) or output (WR) operation. Otherwise, the CPU enters a wait state until the device is ready.
Interrupt-Related Signals
TRAP Restart Interrupts (RST 7.5, 6.5, 5.5)
Interrupt Request (INTR) These five lines are used by an external device to interrupt the CPU. The CPU will not honor the request if it is in the hold state or if the interrupt is disabled. An interrupt is honored only at the completion of an instruction. The interrupts are in descending order of priority.
Interrupt Acknowledge Acknowledges an interrupt.
CPU Initialization

RESET IN Causes the contents of the PC to be set to zero. The CPU resumes execution at location zero.
RESET OUT Acknowledges that the CPU has been reset. The signal can be used to reset the rest of the system.
Voltage and Ground
VCC $+5$-volt power supply
VSS Electrical ground

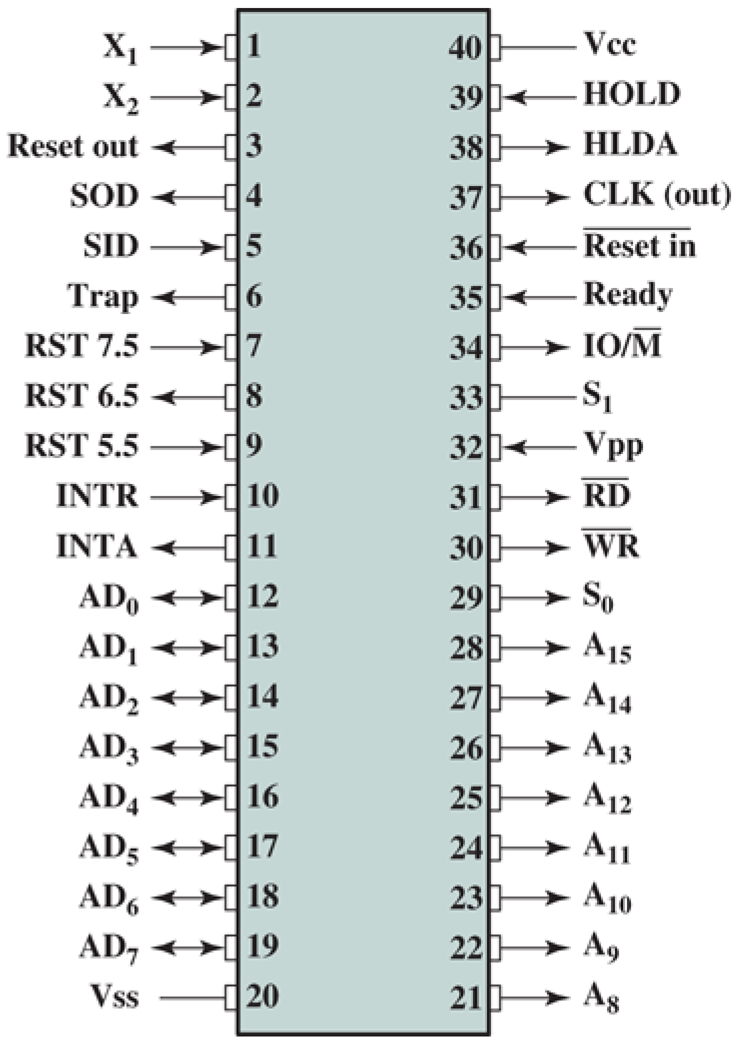


Figure 19.8 Intel 8085 Pin Configuration

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 the current instruction and some external control signals. Its output consists of control signals to the other components of the processor plus control signals to the external system bus.

The timing of processor operations is synchronized by the clock and controlled by the control unit with control signals. Each instruction cycle is divided into from one to five *machine cycles*; each machine cycle is in turn divided into from three to five *states.* Each state lasts one clock cycle. During a state, the processor performs one or a set of simultaneous micro-operations as determined by the control signals.

The number of machine cycles is fixed for a given instruction but varies from one instruction to another. Machine cycles are defined to be equivalent to bus accesses. Thus, the number of machine cycles for an instruction depends on the number of times the processor must communicate with external devices. For example, if an instruction consists of two 8-bit portions, then two machine cycles are required to fetch the instruction. If that instruction involves a 1-byte memory or I/O operation, then a third machine cycle is required for execution.

Figure 19.9 gives an example of 8085 timing, showing the value of external control signals. Of course, at the same time, the control unit generates internal control signals that control internal data transfers. The diagram shows the instruction cycle for an OUT instruction. Three machine cycles (M_1, M_2, M_3) are needed. During the first, the OUT instruction is fetched. The second machine cycle fetches the second half of the instruction, which contains the number of the I/O device selected for output. During the third cycle, the contents of the AC are written out to the selected device over the data bus.

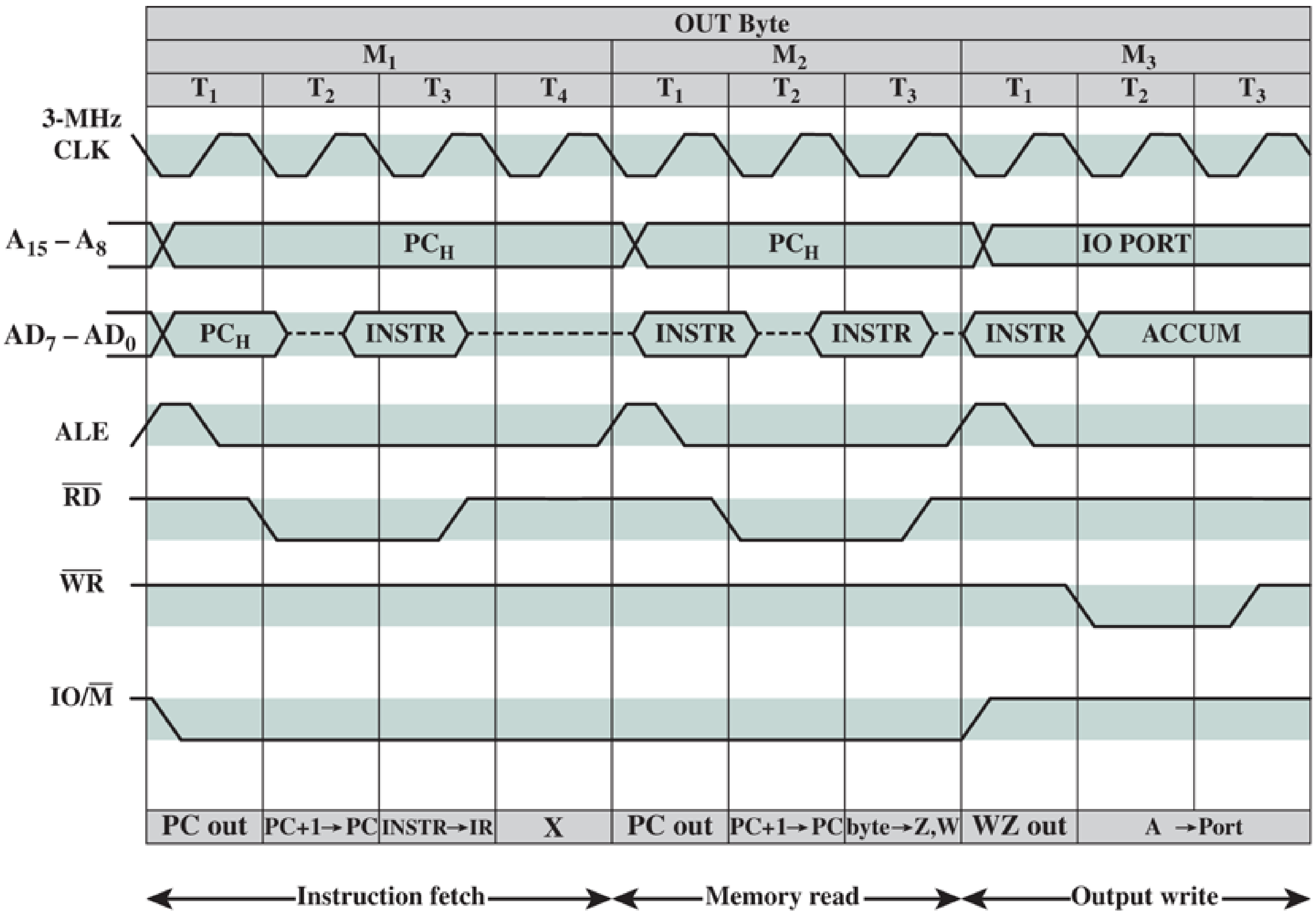


Figure 19.9 Timing Diagram for Intel 8085 OUT Instruction

The Address Latch Enabled (ALE) pulse signals the start of each machine cycle from the control unit. The ALE pulse alerts external circuits. During timing state T_1 of machine cycle M_1, the control unit sets the IO/M signal to indicate that this is a memory operation. Also, the control unit causes the contents of the PC to be placed on the address bus (A_{15} through A_8) and the address/data bus (AD_7 through AD_0). With the falling edge of the Ale pulse, the other modules on the bus store the address.

During timing state T_2, the addressed memory module places the contents of the addressed memory location on the address/data bus. The control unit sets the Read Control (RD) signal to indicate a read, but it waits until T_3 to copy the data from the bus. This gives the memory module time to put the data on the bus and for the signal levels to stabilize. The final state, T_4, is a *bus idle* state during which the processor decodes the instruction. The remaining machine cycles proceed in a similar fashion.

19.3 Hardwired Implementation

We have discussed the control unit in terms of its inputs, output, and functions. We now turn to the topic of control unit implementation. A wide variety of techniques have been used. Most of these fall into one of two categories:

- Hardwired implementation
- Microprogrammed implementation

In a **hardwired implementation**, the control unit is essentially a state machine circuit. Its input logic signals are transformed into a set of output logic signals, which are the control signals. This approach is examined in this section. Microprogrammed implementation is the subject of **Section 19.4**.

Control Unit Inputs

Figure 19.4 depicts the control unit as we have so far discussed it. The key inputs are the IR, the clock, flags, and control bus signals. In the case of the flags and control bus signals, each individual bit typically has some meaning (e.g., overflow). The other two inputs, however, are not directly useful to the control unit.

First consider the IR. The control unit makes use of the opcode and will perform different actions (issue a different combination of control signals) for different instructions. To simplify the control unit logic, there should be a unique logic input for each opcode. This function can be performed by a *decoder*, which takes an encoded input and produces a single output. In general, a decoder will have *n* binary inputs and 2^n binary outputs. Each of the 2^n different input patterns will activate a single unique output. **Table 19.3** is an example for $n = 4$. The decoder for a control unit will typically have to be more complex than that, to account for variable-length opcodes. An example of the digital logic used to implement a decoder is presented in **Chapter 12**.

Table 19.3 A Decoder with 4 Inputs and 16 Outputs

I1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
I2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
I3	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
I4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
O1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
O2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
O3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
O4	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
O5	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

O6	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
O7	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
O8	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
O9	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
O10	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
O11	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
O12	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
O13	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
O14	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
O15	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
O16	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The clock portion of the control unit issues a repetitive sequence of pulses. This is useful for measuring the duration of micro-operations. Essentially, the period of the clock pulses must be long enough to allow the propagation of signals along data paths and through processor circuitry. However, as we have seen, the control unit emits different control signals at different time units within a single instruction cycle. Thus, we would like a counter as input to the control unit, with a different control signal being used for T_1, T_2, and so forth. At the end of an instruction cycle, the control unit must feed back to the counter to reinitialize it at T_1.

With these two refinements, the control unit can be depicted as in **Figure 19.10**.

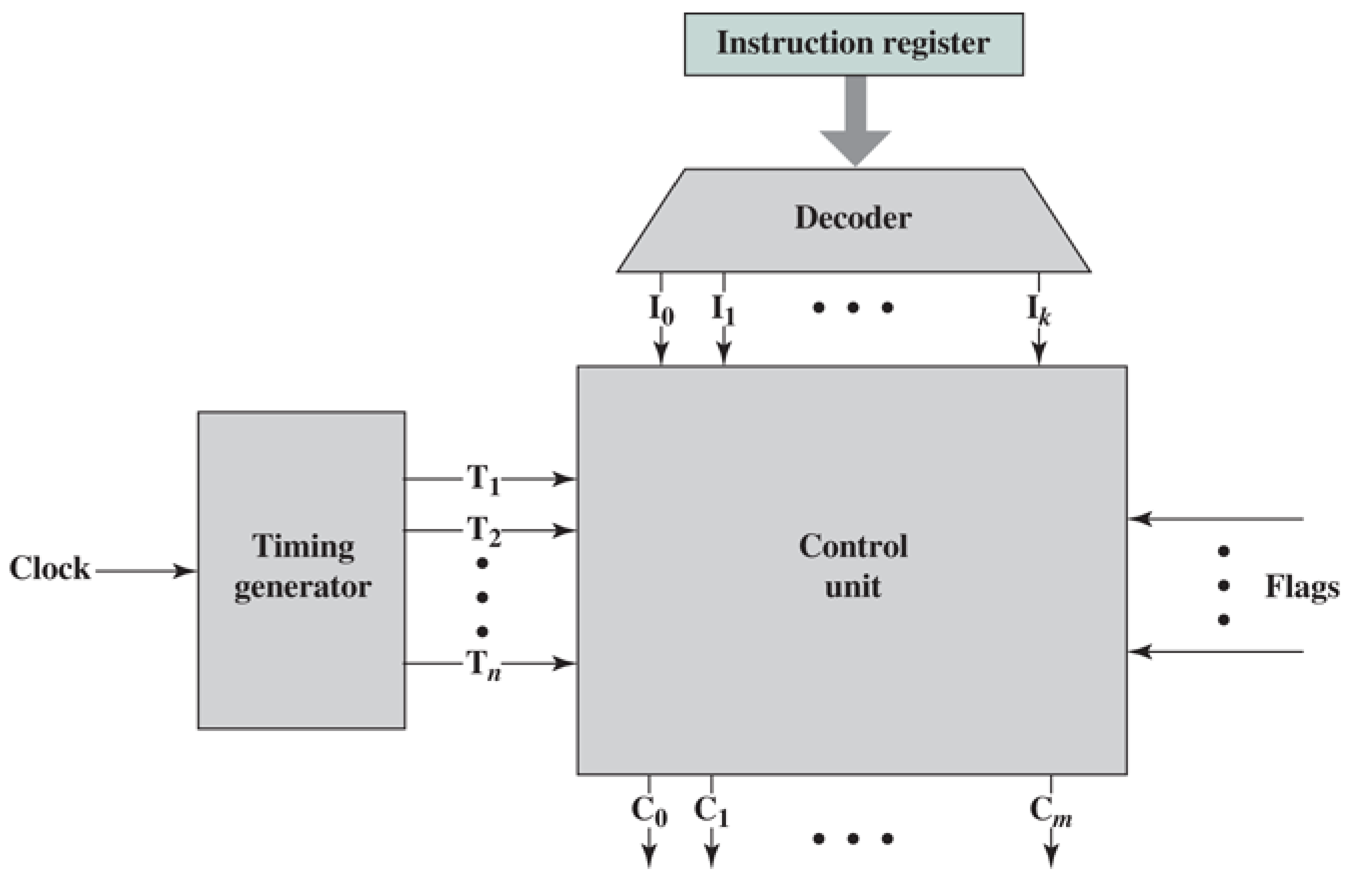


Figure 19.10 Control Unit with Decoded Inputs

Control Unit Logic

To define the hardwired implementation of a control unit, all that remains is to discuss the internal logic of the control unit that produces output control signals as a function of its input signals.

Essentially, what must be done is, for each control signal, to derive a Boolean expression of that signal as a function of the inputs. This is best explained by example. Let us consider again our simple example illustrated in Figure 19.5. We saw in Table 19.1 the micro-operation sequences and control signals needed to control three of the four phases of the instruction cycle.

Let us consider a single control signal, C_5. This signal causes data to be read from the external data bus into the MBR. We can see that it is used twice in Table 19.1. Let us define two new control signals, P and Q, that have the following interpretation:

$$\begin{aligned} &PQ = 00 \text{ Fetch Cycle} \\ &PQ = 01 \text{ Indirect Cycle} \\ &PQ = 10 \text{ Execute Cycle} \\ &PQ = 11 \text{ Interrupt Cycle} \end{aligned}$$

Then the following Boolean expression defines $C_5 : \mathrm{v}$

$$C_5 = P \bullet Q \bullet T_2 + P \bullet Q \bullet T_2$$

That is, the control signal C_5 will be asserted during the second time unit of both the fetch and indirect cycles.

This expression is not complete. C_5 is also needed during the execute cycle. For our simple example, let us assume that there are only three instructions that read from memory: LDA, ADD, and AND. Now we can define C_5 as

$$C_5 = P \bullet Q \bullet T_2 + P \bullet Q \bullet T_2 + P \bullet Q \bullet (\text{LDA} + \text{ADD} + \text{AND}) \bullet T_2$$

This same process could be repeated for every control signal generated by the processor. The result would be a set of Boolean equations that define the behavior of the control unit and hence of the processor.

To tie everything together, the control unit must control the state of the instruction cycle. As was mentioned, at the end of each subcycle (fetch, indirect, execute, interrupt), the control unit issues a signal that causes the timing generator to reinitialize and issue T_1. The control unit must also set the appropriate values of P and Q to define the next subcycle to be performed.

The reader should be able to appreciate that in a modern complex processor, the number of Boolean equations needed to define the control unit is very large. The task of implementing a combinatorial circuit that satisfies all of these equations becomes extremely difficult. The result is that a far simpler approach, known as *microprogramming*, is usually used. This is the subject of the next section.

19.4 Microprogrammed Control

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 semiconductor ROM, the advantages of microprogramming were compelling enough for IBM to make this move. Microprogramming became a popular technique for implementing the control unit of CISC processors. In recent years, microprogramming has become less used but remains a tool available to computer designers. For example, as we have seen on the Pentium 4, machine instructions are converted into a RISC-like format, most of which are executed without the use of microprogramming. However, some of the instructions are executed using microprogramming.

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 has been used in many CISC processors, is to implement a **microprogrammed control unit.**

Consider **Table 19.4**. 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.

Table 19.4 Machine Instruction Set for Wilkes Example

Notation:
$Acc =$ accumulator
$Acc_1 =$ most significant half of accumulator
$Acc_2 =$ least significant half of accumulator
$n =$ storage location n
$C(X) =$ contents of X ($X =$ register or storage location)

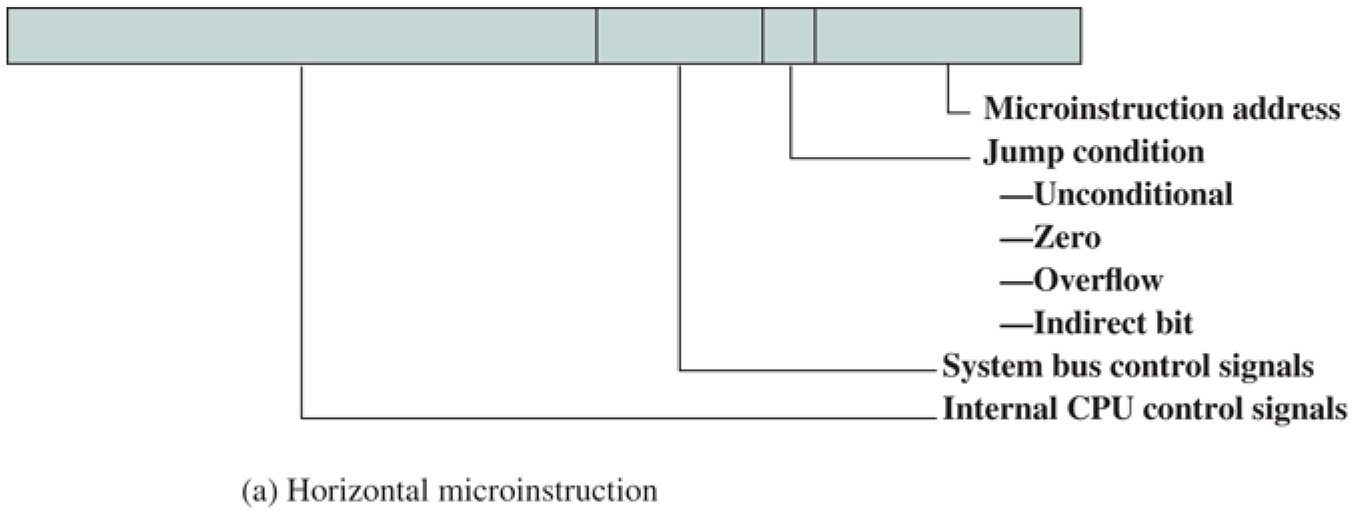
Order	Effect of Order
A n	$C(Acc) + C(n)$ to Acc_1

S n	$C(Acc) - C(n)$ to Acc_1
H n	$C(n)$ to Acc_2
V n	$C(Acc2) \times C(n)$ to Acc, where $C(n) \geq 0$
T n	$C(Acc_1)$ to n, 0to Acc
U n	$C(Acc_1)$ to n
R n	$C(Acc) \times 2^{(n+1)}$ to Acc
L n	$C(Acc) \times 2^{n+1}$ to Acc
G n	IF $C(Acc) < 0$, transfer control to *n*; if $C(Acc) \geq 0$, ignore (i.e., proceed serially)
I n	Read next character on input mechanism into *n*
O n	Send *C*(*n*) to output mechanism

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 19.12a**. 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:


Microinstruction address
Jump condition
—Unconditional
—Zero
—Overflow
—Indirect bit
System bus control signals
Internal CPU control signals
(a) Horizontal microinstruction

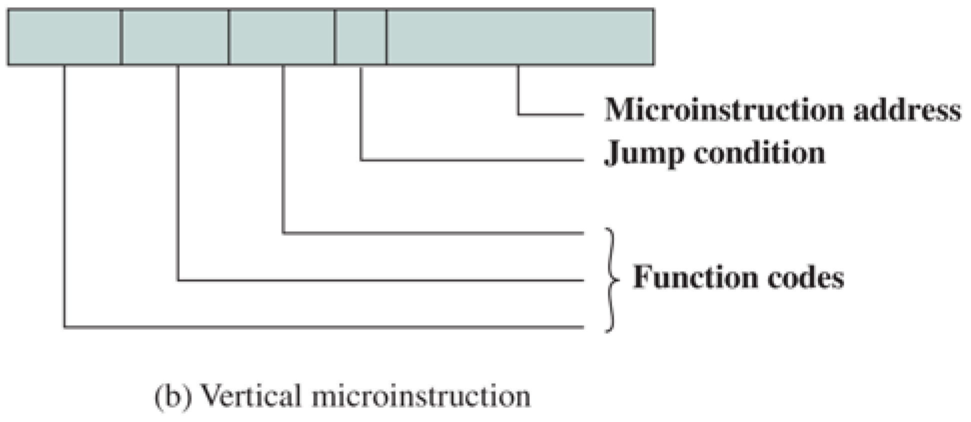


Figure 19.12 Typical Microinstruction Formats

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 19.13 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.

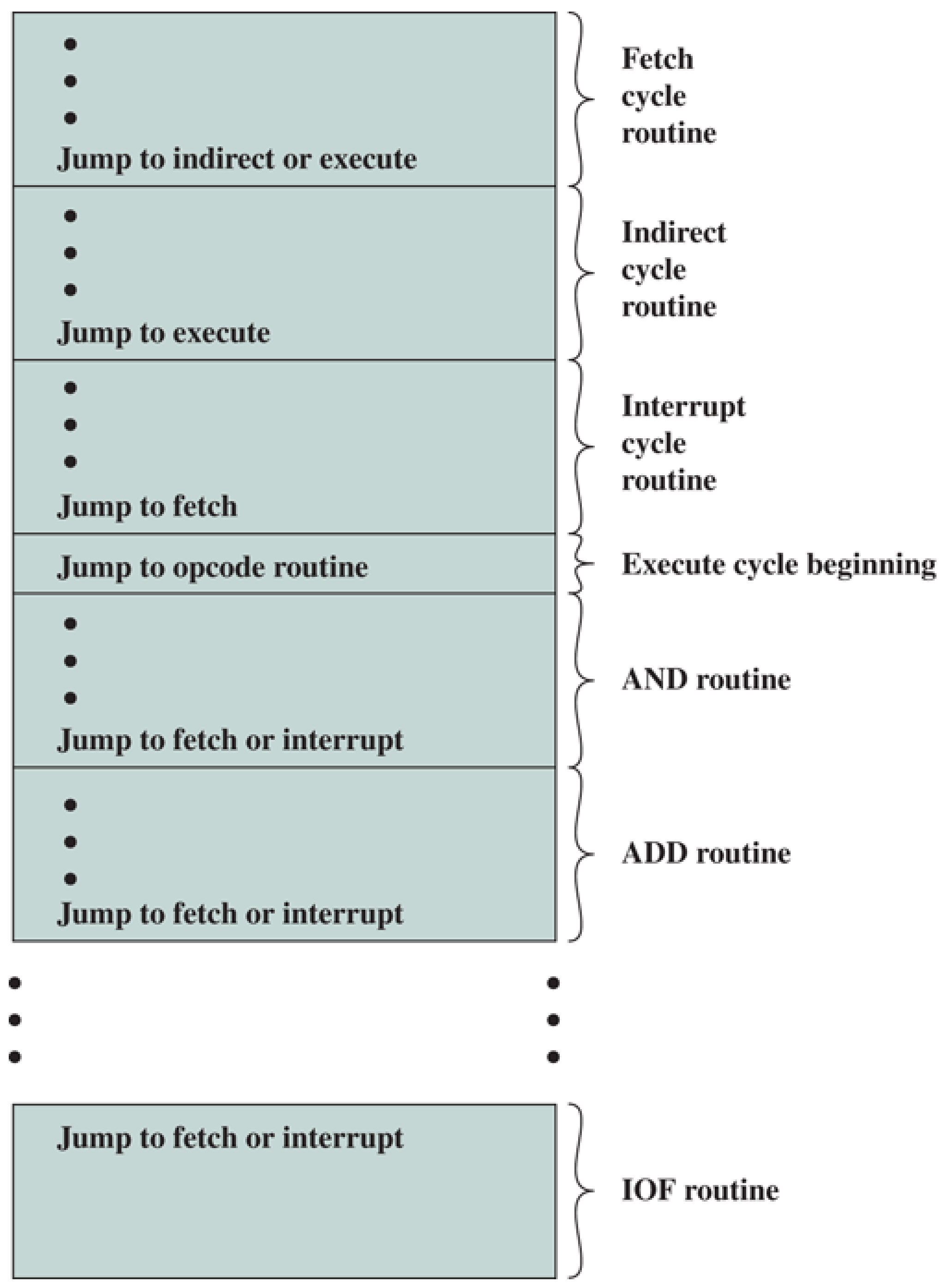


Figure 19.13 Organization of Control Memory

The control memory of **Figure 19.13** 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 19.13** 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 19.14 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 19.12a**) 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.

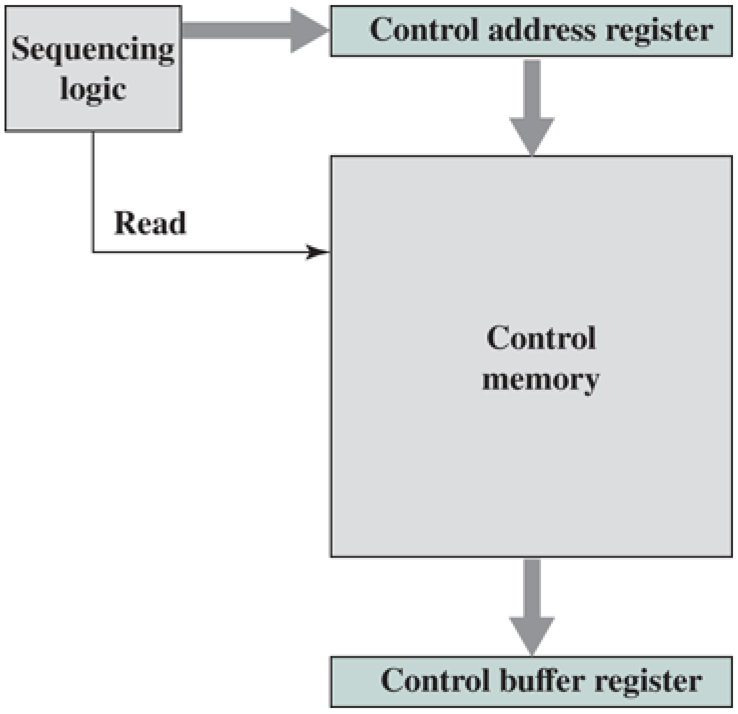


Figure 19.14 Control Unit Microarchitecture

Let us examine this structure in greater detail, as depicted in **Figure 19.15**. Comparing this with **Figure 19.14**, 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:

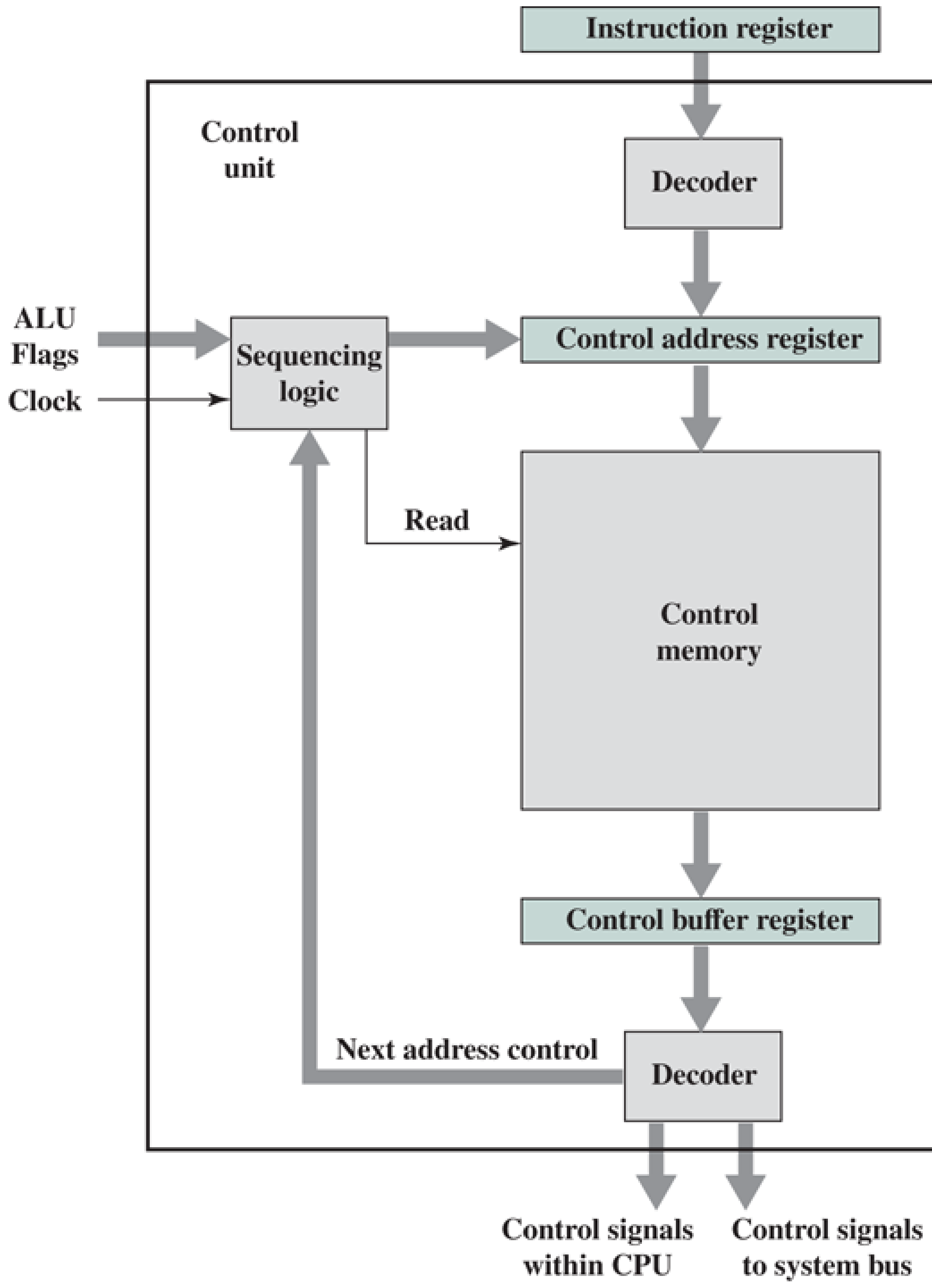


Figure 19.15 Functioning of Microprogrammed Control Unit

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 19.15 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 19.12b). 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., $\text{MAR} \leftarrow (\text{PC})$], 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 19.16. 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.

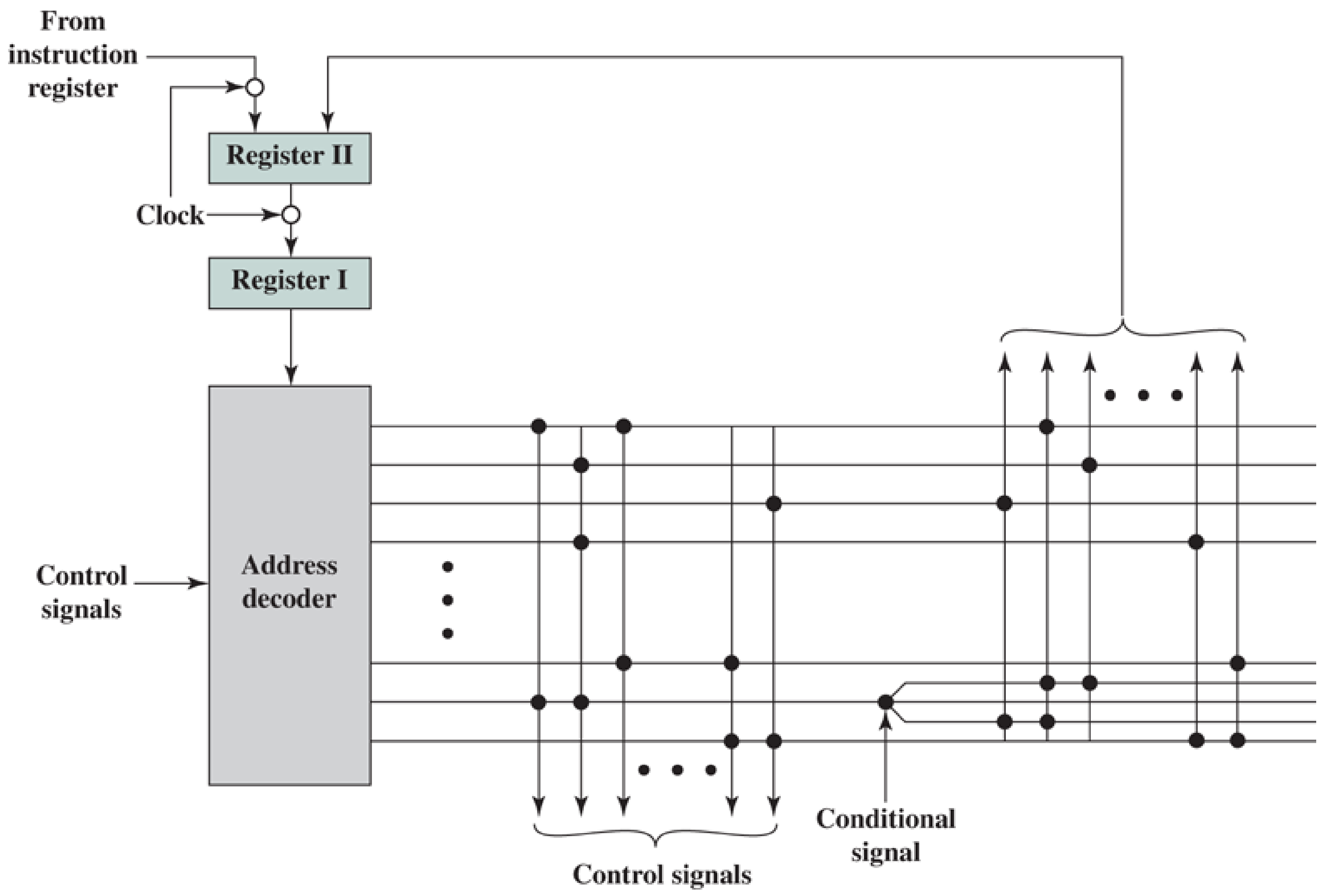


Figure 19.16 Wilkes's Microprogrammed Control Unit

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 19.12a**). 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. To permit branching, a row must contain two address parts, controlled by a conditional signal (e.g., flag), as shown in the figure.

Having proposed this scheme, Wilkes provides an example of its use to implement the control unit of a simple machine. This example, the first known design of a microprogrammed processor, is worth repeating here because it illustrates many of the contemporary principles of microprogramming.

The processor of the hypothetical machine (the example machine by Wilkes) includes the following registers:

A	Multiplicand

B	Accumulator (least significant half)
C	Accumulator (most significant half)
D	Shift register

In addition, there are three registers and two 1-bit flags accessible only to the control unit. The registers are as follows:

E	Serves as both a memory address register (MAR) and temporary storage
F	Program counter
G	Another temporary register; used for counting

Table 19.4 lists the machine instruction set for this example. **Table 19.5** is the complete set of microinstructions, expressed in symbolic form, that implements the control unit. Thus, a total of 38 microinstructions is all that is required to define the system completely.

Table 19.5 Microinstructions for Wilkes Example

Notations: *A*, *B*, *C*, … stand for the various registers in the arithmetical and control register units. *C* to *D* indicates that the switching circuits connect the output of register *C* to the input register *D*; $(D + A)$ to *C* indicates that the output register of *A* is connected to the one input of the adding unit (the output of *D* is permanently connected to the other input), and the output of the adder to register *C*. A numerical symbol *n* in quotes (e.g., "*n*") stands for the source whose output is the number *n* in units of the least significant digit.

* 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.

		Arithmetical Unit	Control Register Unit	Conditional Flip-Flop		Next Microinstruction	
				Set	Use	0	1
	0		*F* to *G* and *E*			1	
	1		(*G* to "1") to *F*			2	
	2		Store to *G*			3	
	3		*G* to *E*			4	
	4		*E* to decoder			—	

A	5	*C* to *D*				16	
S	6	*C* to *D*				17	
H	7	Store to *B*				0	
V	8	Store to *A*				27	
T	9	*C* to Store				25	
U	10	*C* to Store				0	
R	11	*B* to *D*	*E* to *G*			19	
L	12	*C* to *D*	*E* to *G*			22	
G	13		*E* to *G*	(1)C_5		18	
I	14	Input to Store				0	
O	15	Store to Output				0	
	16	$(D + \text{Store})$ to C				0	
	17	$(D - \text{Store})$ to C				0	
	18				1	0	1
	19	*D* to *B* (*R*)*	$(G - 1)$ to E			20	
	20	*C* to *D*		(1)E_5		21	
	21	*D* to *C* (*R*)			1	11	0
	22	*D* to *C* (*L*)†	$(G - 1)$ to E			23	
	23	*B* to *D*		(1)E_5		24	
	24	*D* to *B* (*L*)			1	12	0
	25	“0” to *B*				26	
	26	*B* to *C*				0	
	27	“0” to *C*	“18” to *E*			28	

	28	*B* to *D*	*E* to *G*	(1)B_1		29	
	29	*D* to *B* (*R*)	$(G - \text{“}1\text{”})$ 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	

The first full column gives the address (row number) of each microinstruction. Those addresses corresponding to opcodes are labeled. Thus, when the opcode for the add instruction (A) is encountered, the microinstruction at location 5 is executed. Columns 2 and 3 express the actions to be taken by the ALU and control unit, respectively. Each symbolic expression must be translated into a set of control signals (microinstruction bits). Columns 4 and 5 have to do with the setting and use of the two flags (flip-flops). Column 4 specifies the signal that sets the flag. For example, (1)C_s means that flag number 1 is set by the sign bit of the number in register C. If column 5 contains a flag identifier, then columns 6 and 7 contain the two alternative microinstruction addresses to be used. Otherwise, column 6 specifies the address of the next microinstruction to be fetched.

Instructions 0 through 4 constitute the fetch cycle. Microinstruction 4 presents the opcode to a decoder, which generates the address of a microinstruction corresponding to the machine instruction to be fetched. The reader should be able to deduce the complete functioning of the control unit from a careful study of **Table 19.5**.

Advantages and Disadvantages

The principal advantage of the use of microprogramming to implement a control unit is that it simplifies the design of the control unit. Thus, it is both cheaper and less error prone to implement. A *hardwired* control unit must contain complex logic for sequencing through the many micro-operations of the instruction cycle. On the other hand, the decoders and sequencing logic unit of a microprogrammed control unit are very simple pieces of logic.

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 pure CISC architectures, due to its ease of implementation. RISC processors, with their simpler instruction format, typically use hardwired control units.

19.5 Key Terms, Review Questions, and Problems

Key Terms

control bus

control path

control signal

control unit

hardwired implementation

micro-operations

Review Questions

19.1 Explain the distinction between the written sequence and the time sequence of an instruction.
19.2 What is the relationship between instructions and micro-operations?
19.3 What is the overall function of a processor's control unit?
19.4 Outline a three-step process that leads to a characterization of the control unit.
19.5 What basic tasks does a control unit perform?
19.6 Provide a typical list of the inputs and outputs of a control unit.
19.7 List three types of control signals.
19.8 Briefly explain what is meant by a hardwired implementation of a control unit.
19.9 What is the difference between a hardwired implementation and a microprogrammed implementation of a control unit?
19.10 How is a horizontal microinstruction interpreted?
19.11 What is the purpose of a control memory?

Problems

19.1 Your ALU can add its two input registers, and it can logically complement the bits of either input register, but it cannot subtract. Numbers are to be stored in twos complement representation. List the micro-operations your control unit must perform to cause a subtraction.
19.2 Show the micro-operations and control signals in the same fashion as **Table 19.1** for the processor in **Figure 19.5** for the following instructions:

- Load Accumulator
- Store Accumulator
- Add to Accumulator
- AND to Accumulator
- Jump
- Jump if $\mathrm{AC} = 0$
- Complement Accumulator

19.3 Assume that propagation delay along the bus and through the ALU of **Figure 19.6** are 20 and 100 ns, respectively. The time required for a register to copy data from the bus is 10 ns. What is the time that must be allowed for

a. data from one register to another?
b. the program counter?

19.4 Write the sequence of micro-operations required for the bus structure of **Figure 19.6** to add a number to the AC when the number is

a. immediate operand;
b. direct-address operand;
c. indirect-address operand.

19.5 A stack is implemented as shown in **Figure 19.11** (see **Appendix G** for a discussion of stacks). Show the sequence of micro-operations for

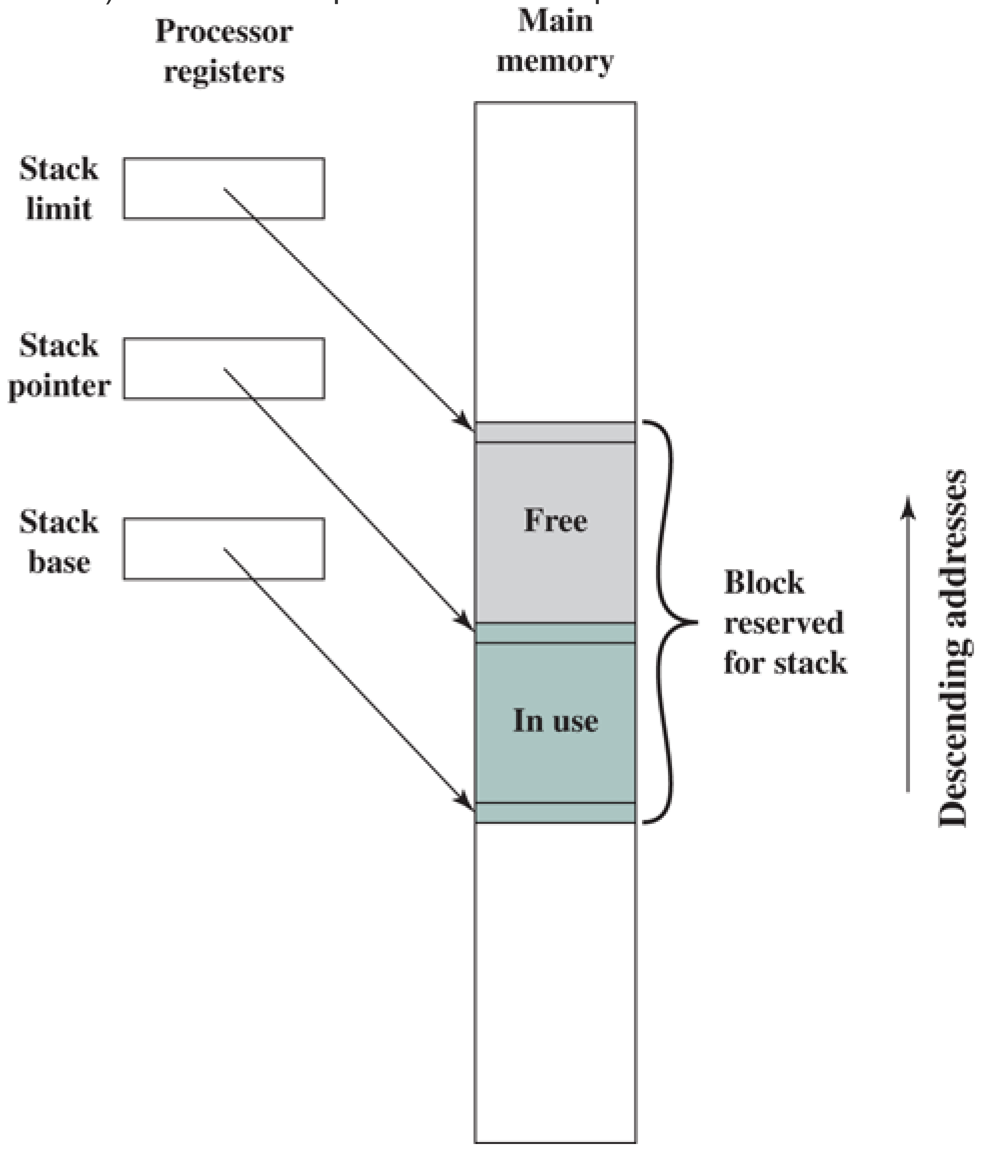


Figure 19.11 Typical Stack Organization (full/descending)

a. popping;
b. the stack

19.6 Describe the implementation of the multiply instruction in the hypothetical machine designed by Wilkes. Use narrative and a flowchart.

19.7 Assume a microinstruction set that includes a microinstruction with the following symbolic form:

$$\mathrm{IF}\,(\mathrm{AC}_0 = 1)\ \mathrm{THEN\ CAR} \leftarrow (\mathrm{C}_{0-6})\ \mathrm{ELSE\ CAR} \leftarrow (\mathrm{CAR}) + 1$$

where AC is the sign bit of the accumulator and C are the first seven bits of the

0 0 – 6

microinstruction. Using this microinstruction, write a microprogram that implements a Branch Register Minus (BRM) machine instruction, which branches if the AC is negative. Assume that bits C_1 through C_n of the microinstruction specify a parallel set of micro-operations. Express the program symbolically.

19.8 A simple processor has four major phases to its instruction cycle: fetch, indirect, execute, and interrupt. Two 1-bit flags designate the current phase in a hardwired implementation.

a. Why are these flags needed?
b. Why are they not needed in a microprogrammed control unit?.