CHAPTER

7

BASIC PROCESSING UNIT

CHAPTER OBJECTIVES

In this chapter you will learn about:

- How a processor executes instructions
- The internal functional units of a processor and how they are interconnected
- Hardware for generating internal control signals
- The microprogramming approach
- Microprogram organization

In this and the next chapter we focus on the processing unit, which executes machine instructions and coordinates the activities of other units. This unit is often called the *Instruction Set Processor* (ISP), or simply the *processor*. We examine its internal structure and how it performs the tasks of fetching, decoding, and executing instructions of a program. The processing unit used to be called the *central processing unit* (CPU). The term "central" is less appropriate today because many modern computer systems include several processing units.

The organization of processors has evolved over the years, driven by developments in technology and the need to provide high performance. A common strategy in the development of high-performance processors is to make various functional units operate in parallel as much as possible. High-performance processors have a pipelined organization where the execution of one instruction is started before the execution of the preceding instruction is completed. In another approach, known as superscalar operation, several instructions are fetched and executed at the same time. Pipelining and superscalar architectures are discussed in Chapter 8. In this chapter, we concentrate on the basic ideas that are common to all processors.

A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program. An instruction is executed by carrying out a sequence of more rudimentary operations. These operations and the means by which they are controlled are the main topic of this chapter.

7.1 SOME FUNDAMENTAL CONCEPTS

To execute a program, the processor fetches one instruction at a time and performs the operations specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered. The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter, PC. After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. A branch instruction may load a different value into the PC.

Another key register in the processor is the instruction register, IR. Suppose that each instruction comprises 4 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are interpreted as an instruction to be executed. Hence, they are loaded into the IR. Symbolically, this can be written as

$$IR \leftarrow [[PC]]$$

2. Assuming that the memory is byte addressable, increment the contents of the PC by 4, that is,

$$PC \leftarrow [PC] + 4$$

3. Carry out the actions specified by the instruction in the IR.

In cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction. These two steps are usually referred to as the *fetch phase;* step 3 constitutes the *execution phase.*

To study these operations in detail, we first need to examine the internal organization of the processor. The main building blocks of a processor were introduced in Figure 1.2. They can be organized and interconnected in a variety of ways. We will start with a very simple organization. Later in this chapter and in Chapter 8 we will present more complex structures that provide high performance. Figure 7.1 shows an organization

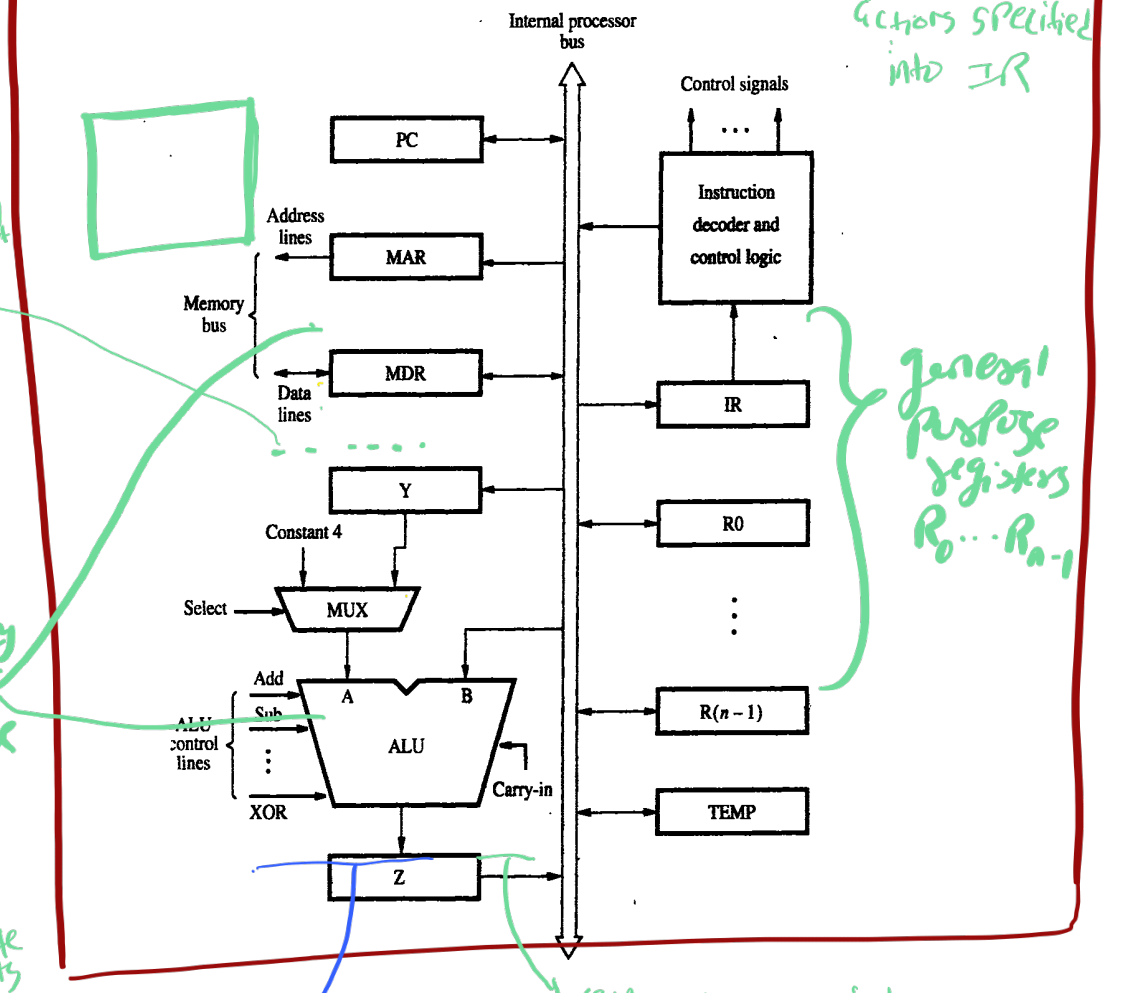


Figure 7.1 Single-bus organization of the datapath inside a processor.

in which the arithmetic and logic unit (ALU) and all the registers are interconnected via a single common bus. This bus is internal to the processor and should not be confused with the external bus that connects the processor to the memory and I/O devices.

The data and address lines of the external memory bus are shown in Figure 7.1 connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. The control lines of the memory bus are connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.

The number and use of the processor registers R0 through R($n - 1$) vary considerably from one processor to another. Registers may be provided for general-purpose use by the programmer. Some may be dedicated as special-purpose registers, such as index registers or stack pointers. Three registers, Y, Z, and TEMP in Figure 7.1, have not been mentioned before. These registers are transparent to the programmer, that is, the programmer need not be concerned with them because they are never referenced explicitly by any instruction. They are used by the processor for temporary storage during execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction.

The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter. We will refer to the two possible values of the MUX control input Select as Select4 and SelectY for selecting the constant 4 or register Y, respectively.

As instruction execution progresses, data are transferred from one register to another, often passing through the ALU to perform some arithmetic or logic operation. The instruction decoder and control logic unit is responsible for implementing the actions specified by the instruction loaded in the IR register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data. The registers, the ALU, and the interconnecting bus are collectively referred to as the *datapath*.

With few exceptions, an instruction can be executed by performing one or more of the following operations in some specified sequence:

- Transfer a word of data from one processor register to another or to the ALU
- Perform an arithmetic or a logic operation and store the result in a processor register
- Fetch the contents of a given memory location and load them into a processor register
- Store a word of data from a processor register into a given memory location

We now consider in detail how each of these operations is implemented, using the simple processor model in Figure 7.1.

7.1.1 REGISTER TRANSFERS

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register. This is represented symbolically in Figure 7.2. The input and output of register R*i* are connected to the bus via switches controlled by the signals Ri_{in} and Ri_{out}, respectively. When Ri_{in} is set to 1, the data on the bus are loaded into R*i*. Similarly, when Ri_{out} is set to 1, the contents of register R*i* are placed on the bus. While Ri_{out} is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows:

- Enable the output of register R1 by setting $R1_{out}$ to 1. This places the contents of R1 on the processor bus.
- Enable the input of register R4 by setting $R4_{in}$ to 1. This loads data from the processor bus into register R4.

All operations and data transfers within the processor take place within time periods defined by the *processor clock*. The control signals that govern a particular transfer are asserted at the start of the clock cycle. In our example, $R1_{out}$ and $R4_{in}$ are set to 1. The registers consist of edge-triggered flip-flops. Hence, at the next active edge of the clock, the flip-flops that constitute R4 will load the data present at their inputs. At the same time, the control signals $R1_{out}$ and $R4_{in}$ will return to 0. We will use this simple model of the timing of data transfers for the rest of this chapter. However, we should point out that other schemes are possible. For example, data transfers may use both the rising and falling edges of the clock. Also, when edge-triggered flip-flops are not used, two or more clock signals may be needed to guarantee proper transfer of data. This is known as *multiphase clocking*.

An implementation for one bit of register R*i* is shown in Figure 7.3 as an example. A two-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. When the control input Ri_{in} is equal to 1, the multiplexer selects the data on the bus. This data will be loaded into the flip-flop at the rising edge of the clock. When Ri_{in} is equal to 0, the multiplexer feeds back the value currently stored in the flip-flop.

The Q output of the flip-flop is connected to the bus via a tri-state gate. When Ri_{out} is equal to 0, the gate's output is in the high-impedance (electrically disconnected) state. This corresponds to the open-circuit state of a switch. When $Ri_{out} = 1$, the gate drives the bus to 0 or 1, depending on the value of Q.

7.1.2 PERFORMING AN ARITHMETIC OR LOGIC OPERATION

The ALU is a combinational circuit that has no internal storage. It performs arithmetic and logic operations on the two operands applied to its A and B inputs. In Figures 7.1 and 7.2, one of the operands is the output of the multiplexer MUX and the other operand

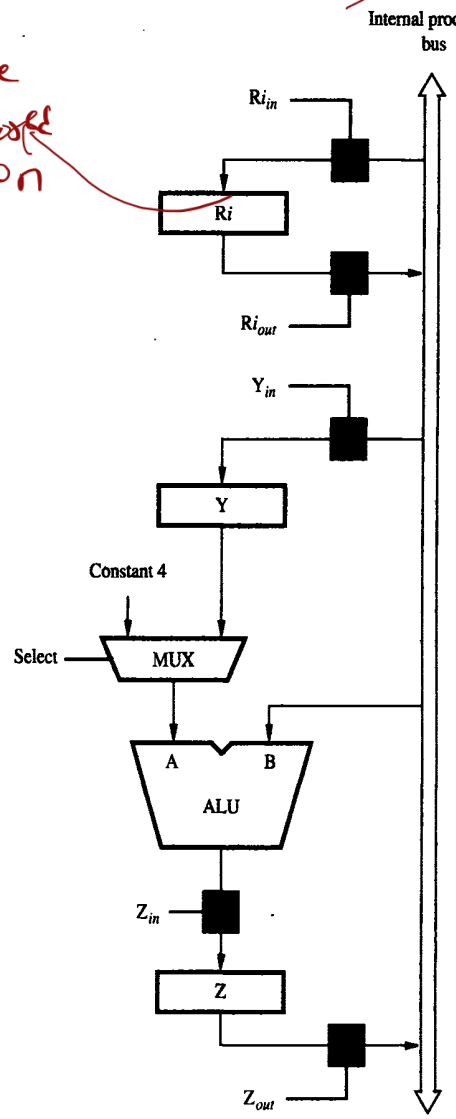


Figure 7.2 Input and output gating for the registers in Figure 7.1.

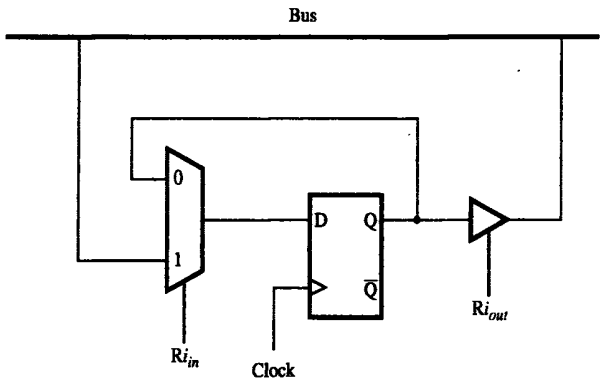


Figure 7.3 Input and output gating for one register bit.

is obtained directly from the bus. The result produced by the ALU is stored temporarily in register Z. Therefore, a sequence of operations to add the contents of register R1 to those of register R2 and store the result in register R3 is

1. $R1_{out}$, Y_{in}
2. $R2_{out}$, SelectY, Add, Z_{in}
3. Z_{out}, $R3_{in}$

The signals whose names are given in any step are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive. Hence, in step 1, the output of register R1 and the input of register Y are enabled, causing the contents of R1 to be transferred over the bus to Y. In step 2, the multiplexer's Select signal is set to SelectY, causing the multiplexer to gate the contents of register Y to input A of the ALU. At the same time, the contents of register R2 are gated onto the bus and, hence, to input B. The function performed by the ALU depends on the signals applied to its control lines. In this case, the Add line is set to 1, causing the output of the ALU to be the sum of the two numbers at inputs A and B. This sum is loaded into register Z because its input control signal is activated. In step 3, the contents of register Z are transferred to the destination register, R3. This last transfer cannot be carried out during step 2, because only one register output can be connected to the bus during any clock cycle.

In this introductory discussion, we assume that there is a dedicated signal for each function to be performed. For example, we assume that there are separate control signals to specify individual ALU operations, such as Add, Subtract, XOR, and so on. In reality, some degree of encoding is likely to be used. For example, if the ALU can perform eight different operations, three control signals would suffice to specify the required operation. We will discuss the limitations and pros and cons of control signal encoding in Section 7.5.1.

7.1.3 FETCHING A WORD FROM MEMORY

To fetch a word of information from memory, the processor has to specify the address of the memory location where this information is stored and request a Read operation. This applies whether the information to be fetched represents an instruction in a program or an operand specified by an instruction. The processor transfers the required address to the MAR, whose output is connected to the address lines of the memory bus. At the same time, the processor uses the control lines of the memory bus to indicate that a Read operation is needed. When the requested data are received from the memory they are stored in register MDR, from where they can be transferred to other registers in the processor.

The connections for register MDR are illustrated in Figure 7.4. It has four control signals: MDR_{in} and MDR_{out} control the connection to the internal bus, and MDR_{inE} and MDR_{outE} control the connection to the external bus. The circuit in Figure 7.3 is easily modified to provide the additional connections. A three-input multiplexer can be used, with the memory bus data line connected to the third input. This input is selected when $MDR_{inE} = 1$. A second tri-state gate, controlled by MDR_{outE} can be used to connect the output of the flip-flop to the memory bus.

During memory Read and Write operations, the timing of internal processor operations must be coordinated with the response of the addressed device on the memory bus. The processor completes one internal data transfer in one clock cycle. The speed of operation of the addressed device, on the other hand, varies with the device. We saw in Chapter 5 that modern processors include a cache memory on the same chip as the processor. Typically, a cache will respond to a memory read request in one clock cycle. However, when a cache miss occurs, the request is forwarded to the main memory, which introduces a delay of several clock cycles. A read or write request may also be intended for a register in a memory-mapped I/O device.

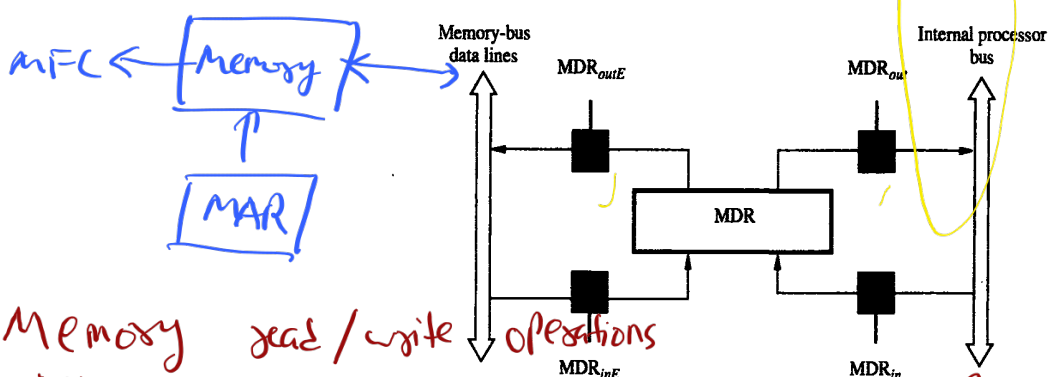


Figure 7.4 Connection and control signals for register MDR.

Such I/O registers are not cached, so their accesses always take a number of clock cycles.

To accommodate the variability in response time, the processor waits until it receives an indication that the requested Read operation has been completed. We will assume that a control signal called Memory-Function-Completed (MFC) is used for this purpose. The addressed device sets this signal to 1 to indicate that the contents of the specified location have been read and are available on the data lines of the memory bus. (We encountered several examples of such a signal in conjunction with the buses discussed in Chapter 4, such as Slave-ready in Figure 4.25 and TRDY# in Figure 4.41.)

As an example of a read operation, consider the instruction Move (R1),R2. The actions needed to execute this instruction are:

1. MAR ← [R1]
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory bus
5. R2 ← [MDR]

These actions may be carried out as separate steps, but some can be combined into a single step. Each action can be completed in one clock cycle, except action 3 which requires one or more clock cycles, depending on the speed of the addressed device.

For simplicity, let us assume that the output of MAR is enabled all the time. Thus, the contents of MAR are always available on the address lines of the memory bus. This is the case when the processor is the bus master. When a new address is loaded into MAR, it will appear on the memory bus at the beginning of the next clock cycle, as shown in Figure 7.5. A Read control signal is activated at the same time MAR is loaded. This signal will cause the bus interface circuit to send a read command, MR, on the bus. With this arrangement, we have combined actions 1 and 2 above into a single control step. Actions 3 and 4 can also be combined by activating control signal MDR_{inE} while waiting for a response from the memory. Thus, the data received from the memory are loaded into MDR at the end of the clock cycle in which the MFC signal is received. In the next clock cycle, MDR_{out} is activated to transfer the data to register R2. This means that the memory read operation requires three steps, which can be described by the signals being activated as follows:

1. $R1_{out}$, MAR_{in}, Read
2. MDR_{inE}, WMFC
3. MDR_{out}, $R2_{in}$

where WMFC is the control signal that causes the processor's control circuitry to wait for the arrival of the MFC signal.

Figure 7.5 shows that MDR_{inE} is set to 1 for exactly the same period as the read command, MR. Hence, in subsequent discussion, we will not specify the value of MDR_{inE} explicitly, with the understanding that it is always equal to MR.

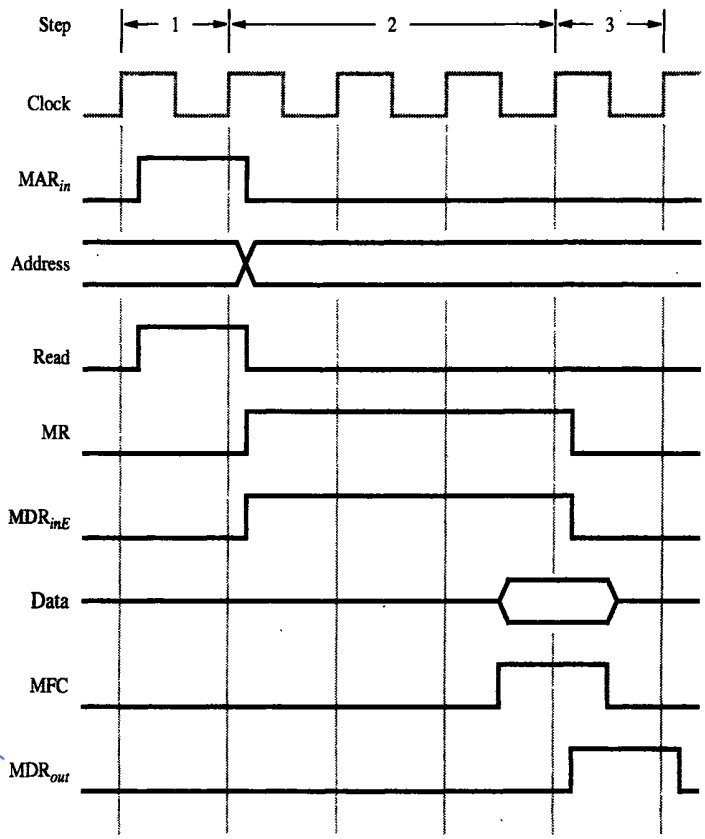


Figure 7.5 Timing of a memory Read operation.

7.1.4 STORING A WORD IN MEMORY

Writing a word into a memory location follows a similar procedure. The desired address is loaded into MAR. Then, the data to be written are loaded into MDR, and a Write command is issued. Hence, executing the instruction Move R2,(R1) requires the following sequence:

1. R1$_{out}$, MAR$_{in}$
2. R2$_{out}$, MDR$_{in}$, Write
3. MDR$_{outE}$, WMFC

As in the case of the read operation, the Write control signal causes the memory bus interface hardware to issue a Write command on the memory bus. The processor remains in step 3 until the memory operation is completed and an MFC response is received.

Steps –
i) Process specifies the address of memory location where the data is to be written
ii) Data to be written is loaded in MDR
iii) Processor request write command
iv) content of MDR will be written to the specified memory location

1. MAR ← R1
2. MDR ← R2
3. start write operation on memory bus.
4. WMFC

Move R2,(R1) → R1's Data should be stored → hence MDR address is required where R2 will be stored

7.2 EXECUTION OF A COMPLETE INSTRUCTION

Let us now put together the sequence of elementary operations required to execute one instruction. Consider the instruction

Add (R3),R1

which adds the contents of a memory location pointed to by R3 to register R1. Executing this instruction requires the following actions:

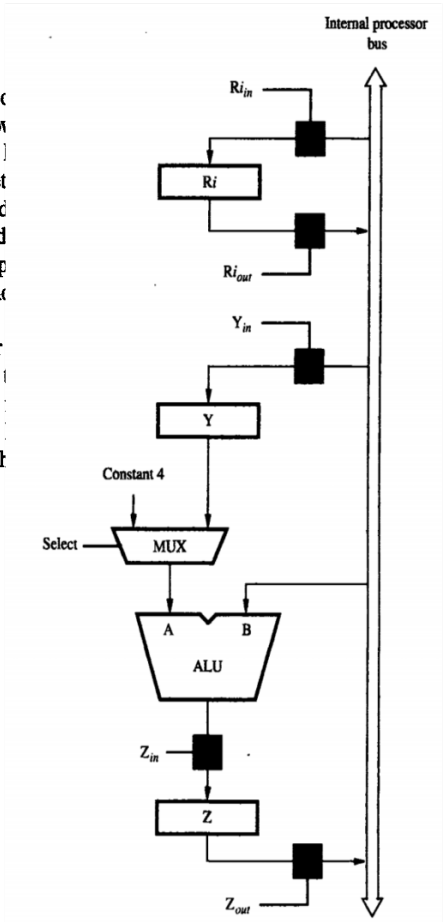
1. Fetch the instruction.
2. Fetch the first operand (the contents of the memory location pointed to by R3).
3. Perform the addition.
4. Load the result into R1.

Figure 7.6 gives the sequence of control steps required to perform these operatic for the single-bus architecture of Figure 7.1. Instruction execution proceeds as follo In step 1, the instruction fetch operation is initiated by loading the contents of the into the MAR and sending a Read request to the memory. The Select signal is set Select4, which causes the multiplexer MUX to select the constant 4. This value is adc to the operand at input B, which is the contents of the PC, and the result is stored register Z. The updated value is moved from register Z back into the PC during step while waiting for the memory to respond. In step 3, the word fetched from the mem is loaded into the IR.

Steps 1 through 3 constitute the instruction fetch phase, which is the same for instructions. The instruction decoding circuit interprets the contents of the IR at beginning of step 4. This enables the control circuitry to activate the control signals steps 4 through 7, which constitute the execution phase. The contents of register are transferred to the MAR in step 4, and a memory read operation is initiated. Tl

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	$R3_{out}$, MAR_{in}, Read
5	$R1_{out}$, Y_{in}, WMFC
6	MDR_{out}, SelectY, Add, Z_{in}
7	Z_{out}, $R1_{in}$, End

Figure 7.6 Control sequence for execution of the instruction Add (R3),R1.



the contents of R1 are transferred to register Y in step 5, to prepare for the addition operation. When the Read operation is completed, the memory operand is available in register MDR, and the addition operation is performed in step 6. The contents of MDR are gated to the bus, and thus also to the B input of the ALU, and register Y is selected as the second input to the ALU by choosing SelectY. The sum is stored in register Z, then transferred to R1 in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

This discussion accounts for all control signals in Figure 7.6 except Y_{in} in step 2. There is no need to copy the updated contents of PC into register Y when executing the Add instruction. But, in Branch instructions the updated value of the PC is needed to compute the Branch target address. To speed up the execution of Branch instructions, this value is copied into register Y in step 2. Since step 2 is part of the fetch phase, the same action will be performed for all instructions. This does not cause any harm because register Y is not used for any other purpose at that time.

7.2.1 BRANCH INSTRUCTIONS

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. Figure 7.7 gives a control sequence that implements an unconditional branch instruction. Processing starts, as usual, with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3. The offset value is extracted from the IR by the instruction decoding circuit, which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4, and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

The offset X used in a branch instruction is usually the difference between the branch target address and the address immediately following the branch instruction.

Step	Action
1	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field-of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, End

Figure 7.7 Control sequence for an unconditional Branch instruction.

For example, if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 46. The reason for this can be readily appreciated from the control sequence in Figure 7.7. The PC is incremented during the fetch phase, before knowing the type of instruction being executed. Thus, when the branch address is computed in step 4, the PC value used is the updated value, which points to the instruction following the branch instruction in the memory.

Consider now a conditional branch. In this case, we need to check the status of the condition codes before loading a new value into the PC. For example, for a Branch-on-negative (Branch<0) instruction, step 4 in Figure 7.7 is replaced with

$$\text{Offset-field-of-IR}_{out}, \text{Add}, Z_{in}, \text{If N} = 0 \text{ then End}$$

Thus, if N=0 the processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

7.3 MULTIPLE-BUS ORGANIZATION

We used the simple single-bus structure of Figure 7.1 to illustrate the basic ideas. The resulting control sequences in Figures 7.6 and 7.7 are quite long because only one data item can be transferred over the bus in a clock cycle. To reduce the number of steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel.

Figure 7.8 depicts a three-bus structure used to connect the registers and the ALU of a processor. All general-purpose registers are combined into a single block called the *register file.* In VLSI technology, the most efficient way to implement a number of registers is in the form of an array of memory cells similar to those used in the implementation of random-access memories (RAMs) described in Chapter 5. The register file in Figure 7.8 is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

Buses A and B are used to transfer the source operands to the A and B inputs of the ALU, where an arithmetic or logic operation may be performed. The result is transferred to the destination over bus C. If needed, the ALU may simply pass one of its two input operands unmodified to bus C. We will call the ALU control signals for such an operation R=A or R=B. The three-bus arrangement obviates the need for registers Y and Z in Figure 7.1.

A second feature in Figure 7.8 is the introduction of the Incrementer unit, which is used to increment the PC by 4. Using the Incrementer eliminates the need to add 4 to the PC using the main ALU, as was done in Figures 7.6 and 7.7. The source for the constant 4 at the ALU input multiplexer is still useful. It can be used to increment other addresses, such as the memory addresses in LoadMultiple and StoreMultiple instructions.

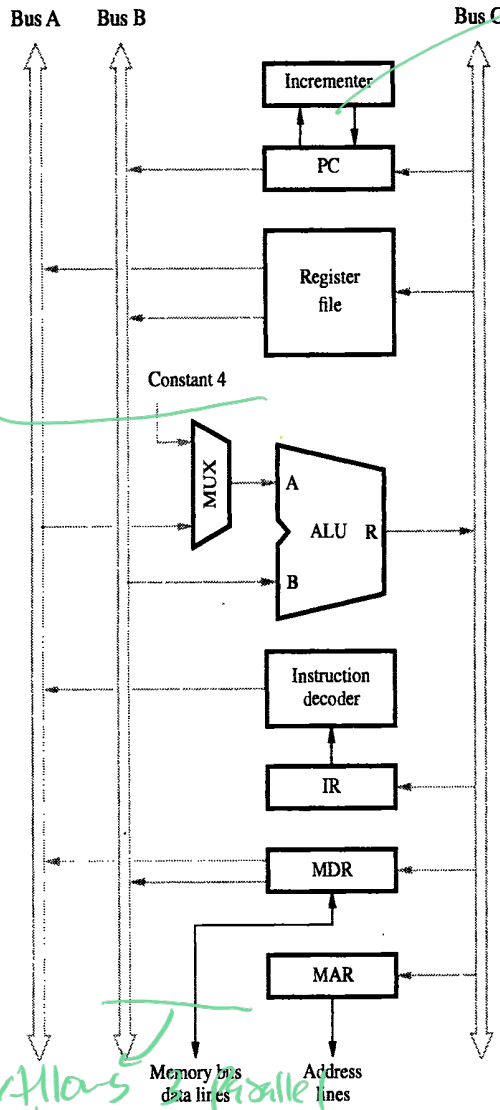


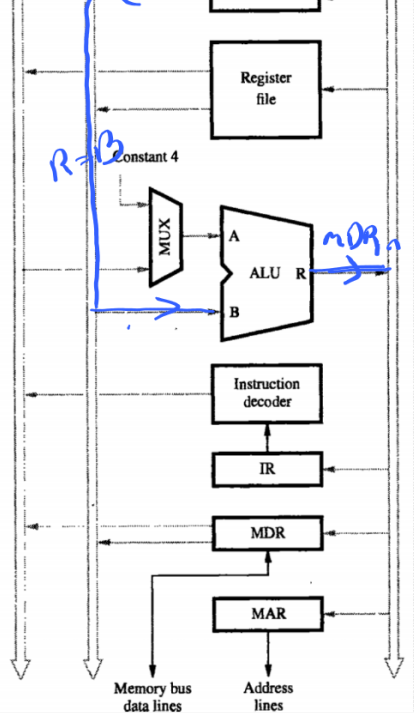
Figure 7.8 Three-bus organization of the datapath.

helps in reading multiple data → but can only write one data

ALU have two inputs hence R_1 & R_2 can be present at the same time

allows parallel data transfers operations

less no of cycles required to execute instructions



Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End

Figure 7.9 Control sequence for the instruction Add R4,R5,R6 for the three-bus organization in Figure 7.8.

Consider the three-operand instruction

Add R4,R5,R6

The control sequence for executing this instruction is given in Figure 7.9. In step 1, the contents of the PC are passed through the ALU, using the R=B control signal, and loaded into the MAR to start a memory read operation. At the same time the PC is incremented by 4. Note that the value loaded into MAR is the original contents of the PC. The incremented value is loaded into the PC at the end of the clock cycle and will not affect the contents of MAR. In step 2, the processor waits for MFC and loads the data received into MDR, then transfers them to IR in step 3. Finally, the execution phase of the instruction requires only one control step to complete, step 4.

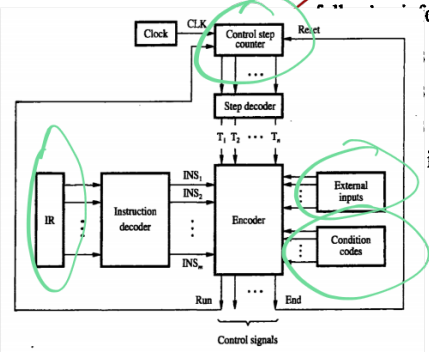
By providing more paths for data transfer a significant reduction in the number of clock cycles needed to execute an instruction is achieved.

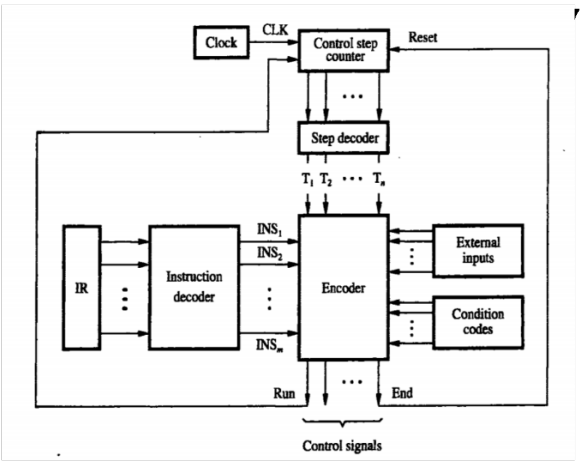
7.4 HARDWIRED CONTROL

To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence. Computer designers use a wide variety of techniques to solve this problem. The approaches used fall into one of two categories: hardwired control and microprogrammed control. We discuss each of these techniques in detail, starting with hardwired control in this section.

Consider the sequence of control signals given in Figure 7.6. Each step in this sequence is completed in one clock period. A counter may be used to keep track of the control steps, as shown in Figure 7.10. Each state, or count, of this counter corresponds to one control step. The required control signals are determined by the following information:

- Contents of the control step counter
- Contents of the instruction register
- Contents of the condition code flags
- External input signals, such as MFC and interrupt requests





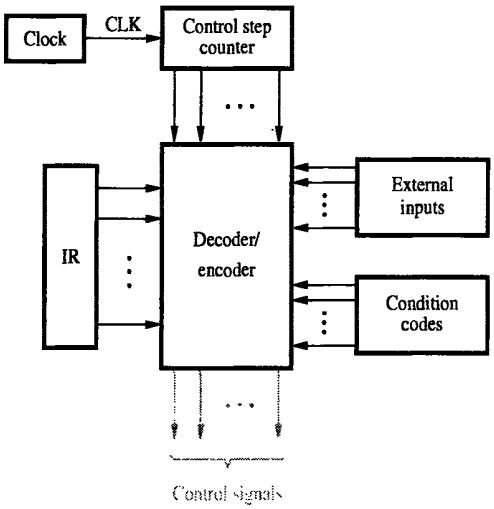


Figure 7.10 Control unit organization.

To gain insight into the structure of the control unit, we start with a simplified view of the hardware involved. The decoder/encoder block in Figure 7.10 is a combinational circuit that generates the required control outputs, depending on the state of all its inputs. By separating the decoding and encoding functions, we obtain the more detailed block diagram in Figure 7.11. The step decoder provides a separate signal line for each step, or time slot, in the control sequence. Similarly, the output of the instruction decoder consists of a separate line for each machine instruction. For any instruction loaded in the IR, one of the output lines INS_1 through INS_m is set to 1, and all other lines are set to 0. (For design details of decoders, refer to Appendix A.) The input signals to the encoder block in Figure 7.11 are combined to generate the individual control signals Y_{in}, PC_{out}, Add, End, and so on. An example of how the encoder generates the Z_{in} control signal for the processor organization in Figure 7.1 is given in Figure 7.12. This circuit implements the logic function

$$Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \cdots \qquad [7.1]$$

This signal is asserted during time slot T_1 for all instructions, during T_6 for an Add instruction, during T_4 for an unconditional branch instruction, and so on. The logic function for Z_{in} is derived from the control sequences in Figures 7.6 and 7.7. As another example, Figure 7.13 gives a circuit that generates the End control signal from the logic function

$$End = T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{N}) \cdot BRN + \cdots \qquad [7.2]$$

The End signal starts a new instruction fetch cycle by resetting the control step counter to its starting value. Figure 7.11 contains another control signal called RUN. When

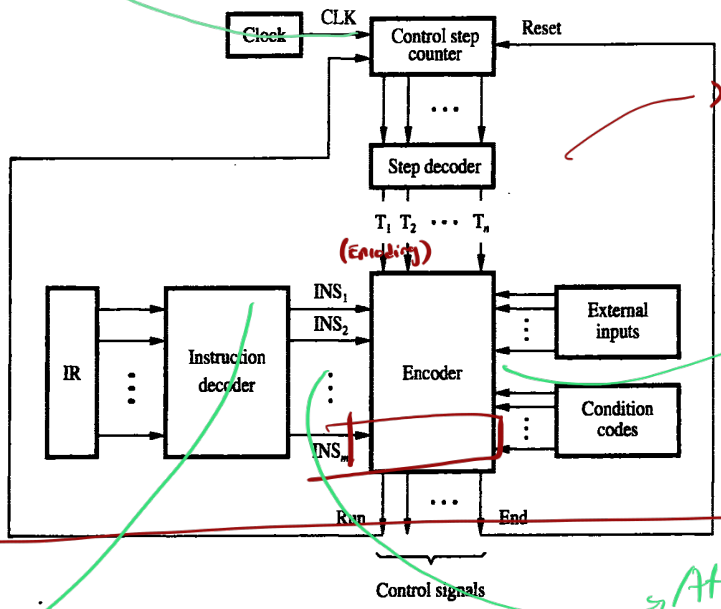


Figure 7.11 Separation of the decoding and encoding functions.

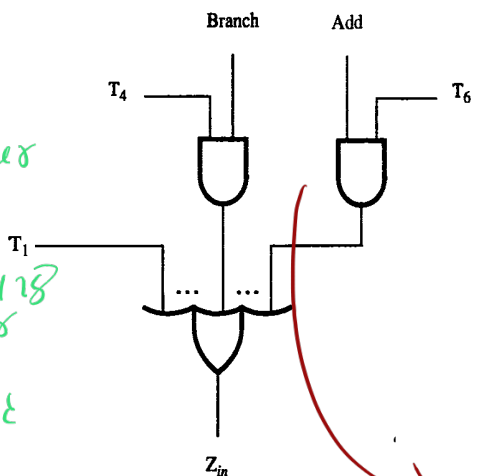


Figure 7.12 Generation of the Z_{in} control signal for the processor in Figure 7.1.

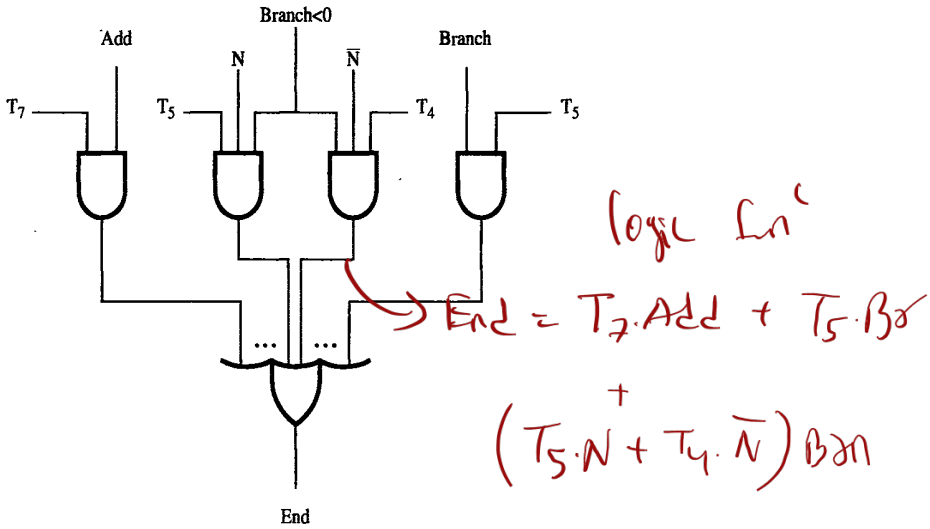


Figure 7.13 Generation of the End control signal.

set to 1, RUN causes the counter to be incremented by one at the end of every clock cycle. When RUN is equal to 0, the counter stops counting. This is needed whenever the WMFC signal is issued, to cause the processor to wait for the reply from the memory.

The control hardware shown in Figure 7.10 or 7.11 can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes, and the external inputs. The outputs of the state machine are the control signals. The sequence of operations carried out by this machine is determined by the wiring of the logic elements, hence the name "hardwired." A controller that uses this approach can operate at high speed. However, it has little flexibility, and the complexity of the instruction set it can implement is limited.

7.4.1 A COMPLETE PROCESSOR

A complete processor can be designed using the structure shown in Figure 7.14. This structure has an instruction unit that fetches instructions from an instruction cache or from the main memory when the desired instructions are not already in the cache. It has separate processing units to deal with integer data and floating-point data. Each of these units can be organized as shown in Figure 7.8. A data cache is inserted between these units and the main memory. Using separate caches for instructions and data is common practice in many processors today. Other processors use a single cache that

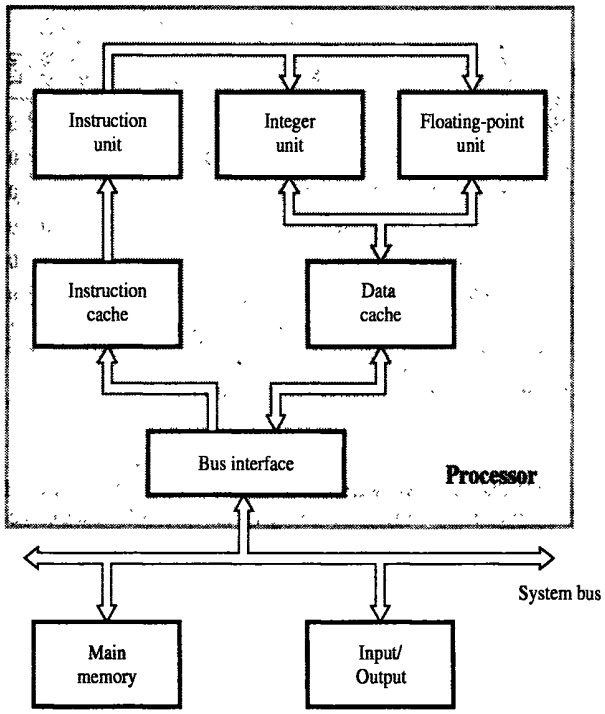


Figure 7.14 Block diagram of a complete processor.

stores both instructions and data. The processor is connected to the system bus and, hence, to the rest of the computer, by means of a bus interface.

Although we have shown just one integer and one floating-point unit in Figure 7.14, a processor may include several units of each type to increase the potential for concurrent operations. The way in which multiple units can be organized to increase the rate of instruction execution will be discussed in Chapter 8.

7.5 MICROPROGRAMMED CONTROL

In Section 7.4, we saw how the control signals required inside the processor can be generated using a control step counter and a decoder/encoder circuit. Now we discuss an alternative scheme, called *microprogrammed control*, in which control signals are generated by a program similar to machine language programs.

all control signals for instruction

control store (CS) - stores microroutines for all instructions of an ISA.

Each sequence of steps is a control word (CW) whose individual bits represent various control signals

control signals

Fetch Operand

T_2: 00 1 1 1

T_3: 10 1 00 0

C.S

1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

Figure 7.15 An example of microinstructions for Figure 7.6.

First, we introduce some common terms. A *control word* (CW) is a word whose individual bits represent the various control signals in Figure 7.11. Each of the control steps in the control sequence of an instruction defines a unique combination of 1s and 0s in the CW. The CWs corresponding to the 7 steps of Figure 7.6 are shown in Figure 7.15. We have assumed that SelectY is represented by Select = 0 and Select4 by Select = 1. A sequence of CWs corresponding to the control sequence of a machine instruction constitutes the *microroutine* for that instruction, and the individual control words in this microroutine are referred to as *microinstructions*.

The microroutines for all instructions in the instruction set of a computer are stored in a special memory called the *control store*. The control unit can generate the control signals for any instruction by sequentially reading the CWs of the corresponding microroutine from the control store. This suggests organizing the control unit as shown in Figure 7.16. To read the control words sequentially from the control store, a *microprogram counter* (μPC) is used. Every time a new instruction is loaded into the IR, the output of the block labeled "starting address generator" is loaded into the μPC. The μPC is then automatically incremented by the clock, causing successive microinstructions to be read from the control store. Hence, the control signals are delivered to various parts of the processor in the correct sequence.

One important function of the control unit cannot be implemented by the simple organization in Figure 7.16. This is the situation that arises when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action. In the case of hardwired control, this situation is handled by including an appropriate logic function, as in Equation 7.2, in the encoder circuitry. In microprogrammed control, an alternative approach is to use conditional branch microinstructions. In addition to the branch address, these microinstructions specify which of the external inputs, condition codes, or, possibly, bits of the instruction register, should be checked as a condition for branching to take place.

The instruction Branch<0 may now be implemented by a microroutine such as that shown in Figure 7.17. After loading this instruction into IR, a branch microinstruction

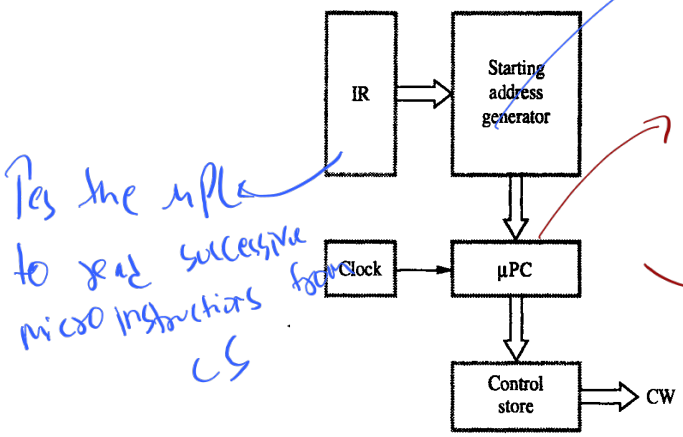


Figure 7.16 Basic organization of a microprogrammed control unit.

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
...	...
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17 Microroutine for the instruction Branch < 0.

transfers control to the corresponding microroutine, which is assumed to start at location 25 in the control store. This address is the output of the starting address generator block in Figure 7.16. The microinstruction at location 25 tests the N bit of the condition codes. If this bit is equal to 0, a branch takes place to location 0 to fetch a new machine instruction. Otherwise, the microinstruction at location 26 is executed to put the branch target address into register Z, as in step 4 in Figure 7.7. The microinstruction in location 27 loads this address into the PC.

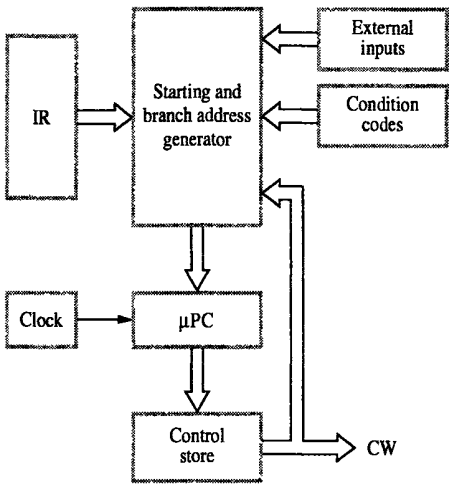


Figure 7.18 Organization of the control unit to allow conditional branching in the microprogram.

To support microprogram branching, the organization of the control unit should be modified as shown in Figure 7.18. The starting address generator block of Figure 7.16 becomes the starting and branch address generator. This block loads a new address into the μPC when a microinstruction instructs it to do so. To allow implementation of a conditional branch, inputs to this block consist of the external inputs and condition codes as well as the contents of the instruction register. In this control unit, the μPC is incremented every time a new microinstruction is fetched from the microprogram memory, except in the following situations:

1. When a new instruction is loaded into the IR, the μPC is loaded with the starting address of the microroutine for that instruction.
2. When a Branch microinstruction is encountered and the branch condition is satisfied, the μPC is loaded with the branch address.
3. When an End microinstruction is encountered, the μPC is loaded with the address of the first CW in the microroutine for the instruction fetch cycle (this address is 0 in Figure 7.17).

7.5.1 MICROINSTRUCTIONS

Having described a scheme for sequencing microinstructions, we now take a closer look at the format of individual microinstructions. A straightforward way to structure microinstructions is to assign one bit position to each control signal, as in Figure 7.15.

However, this scheme has one serious drawback — assigning individual bits to each control signal results in long microinstructions because the number of required signals is usually large. Moreover, only a few bits are set to 1 (to be used for active gating) in any given microinstruction, which means the available bit space is poorly used. Consider again the simple processor of Figure 7.1, and assume that it contains only four general-purpose registers, R0, R1, R2, and R3. Some of the connections in this processor are permanently enabled, such as the output of the IR to the decoding circuits and both inputs to the ALU. The remaining connections to various registers require a total of 20 gating signals. Additional control signals not shown in the figure are also needed, including the Read, Write, Select, WMFC, and End signals. Finally, we must specify the function to be performed by the ALU. Let us assume that 16 functions are provided, including Add, Subtract, AND, and XOR. These functions depend on the particular ALU used and do not necessarily have a one-to-one correspondence with the machine instruction OP codes. In total, 42 control signals are needed.

If we use the simple encoding scheme described earlier, 42 bits would be needed in each microinstruction. Fortunately, the length of the microinstructions can be reduced easily. Most signals are not needed simultaneously, and many signals are mutually exclusive. For example, only one function of the ALU can be activated at a time. The source for a data transfer must be unique because it is not possible to gate the contents of two different registers onto the bus at the same time. Read and Write signals to the memory cannot be active simultaneously. This suggests that signals can be grouped so that all mutually exclusive signals are placed in the same group. Thus, at most one *microoperation* per group is specified in any microinstruction. Then it is possible to use a binary coding scheme to represent the signals within a group. For example, four bits suffice to represent the 16 available functions in the ALU. Register output control signals can be placed in a group consisting of PC_{out}, MDR_{out}, Z_{out}, $Offset_{out}$, $R0_{out}$, $R1_{out}$, $R2_{out}$, $R3_{out}$, and $TEMP_{out}$. Any one of these can be selected by a unique 4-bit code.

Further natural groupings can be made for the remaining signals. Figure 7.19 shows an example of a partial format for the microinstructions, in which each group occupies a field large enough to contain the required codes. Most fields must include one inactive code for the case in which no action is required. For example, the all-zero pattern in F1 indicates that none of the registers that may be specified in this field should have its contents placed on the bus. An inactive code is not needed in all fields. For example, F4 contains 4 bits that specify one of the 16 operations performed in the ALU. Since no spare code is included, the ALU is active during the execution of every microinstruction. However, its activity is monitored by the rest of the machine through register Z, which is loaded only when the Z_{in} signal is activated.

Grouping control signals into fields requires a little more hardware because decoding circuits must be used to decode the bit patterns of each field into individual control signals. The cost of this additional hardware is more than offset by the reduced number of bits in each microinstruction, which results in a smaller control store. In Figure 7.19, only 20 bits are needed to store the patterns for the 42 signals.

So far we have considered grouping and encoding only mutually exclusive control signals. We can extend this idea by enumerating the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can

Microinstruction

F1	F2	F3	F4	F5
F1 (4 bits)	F2 (3 bits)	F3 (3 bits)	F4 (4 bits)	F5 (2 bits)
0000: No transfer 0001: PC_{out} 0010: MDR_{out} 0011: Z_{out} 0100: $R0_{out}$ 0101: $R1_{out}$ 0110: $R2_{out}$ 0111: $R3_{out}$ 1010: $TEMP_{out}$ 1011: $Offset_{out}$	000: No transfer 001: PC_{in} 010: IR_{in} 011: Z_{in} 100: $R0_{in}$ 101: $R1_{in}$ 110: $R2_{in}$ 111: $R3_{in}$	000: No transfer 001: MAR_{in} 010: MDR_{in} 011: $TEMP_{in}$ 100: Y_{in}	0000: Add 0001: Sub ⋮ 1111: XOR 16 ALU functions	00: No action 01: Read 10: Write

F6	F7	F8	...
F6 (1 bit)	F7 (1 bit)	F8 (1 bit)	
0: SelectY 1: Select4	0: No action 1: WMFC	0: Continue 1: End	

Figure 7.19 An example of a partial format for field-encoded microinstructions.

then be assigned a distinct code that represents the microinstruction. Such full encoding is likely to further reduce the length of microwords but also to increase the complexity of the required decoder circuits.

Highly encoded schemes that use compact codes to specify only a small number of control functions in each microinstruction are referred to as a *vertical organization*. On the other hand, the minimally encoded scheme of Figure 7.15, in which many resources can be controlled with a single microinstruction, is called a *horizontal organization*. The horizontal approach is useful when a higher operating speed is desired and when the machine structure allows parallel use of resources. The vertical approach results in considerably slower operating speeds because more microinstructions are needed to perform the desired control functions. Although fewer bits are required for each microinstruction, this does not imply that the total number of bits in the control store is smaller. The significant factor is that less hardware is needed to handle the execution of microinstructions.

Q. 100 control signals
→ For Horizontal Control size = 100
For Vertical — — = $\log_2 100$ = 7 bits
For Diagonal K = 25, 15, 40, 5, 15
2^5 2^4 26 2^3 2^4
5+4+6+3+4
11
22 bits

Horizontal and vertical organizations represent the two organizational extremes in microprogrammed control. Many intermediate schemes are also possible, in which the degree of encoding is a design parameter. The layout in Figure 7.19 is a horizontal organization because it groups only mutually exclusive microoperations in the same fields. As a result, it does not limit in any way the processor's ability to perform various microoperations in parallel.

Although we have considered only a subset of all the possible control signals, this subset is representative of actual requirements. We have omitted some details that are not essential for understanding the principles of operation.

7.5.2 MICROPROGRAM SEQUENCING

The simple microprogram example in Figure 7.15 requires only straightforward sequential execution of microinstructions, except for the branch at the end of the fetch phase. If each machine instruction is implemented by a microroutine of this kind, the microcontrol structure suggested in Figure 7.18, in which a μPC governs the sequencing, would be sufficient. A microroutine is entered by decoding the machine instruction into a starting address that is loaded into the μPC. Some branching capability within the microprogram can be introduced through special branch microinstructions that specify the branch address, similar to the way branching is done in machine-level instructions.

With this approach, writing microprograms is fairly simple because standard software techniques can be used. However, this advantage is countered by two major disadvantages. Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store. If most machine instructions involve several addressing modes, there can be many instruction and addressing mode combinations. A separate microroutine for each of these combinations would produce considerable duplication of common parts. We want to organize the microprogram so that the microroutines share as many common parts as possible. This requires many branch microinstructions to transfer control among the various parts. Hence, a second disadvantage arises — execution time is longer because it takes more time to carry out the required branches.

Consider a more complicated example of a complete machine instruction. In Chapter 2, we used instructions of the type

Add src,Rdst

which adds the source operand to the contents of register Rdst and places the sum in Rdst, the destination register. Let us assume that the source operand can be specified in the following addressing modes: register, autoincrement, autodecrement, and indexed, as well as the indirect forms of these four modes. We now use this instruction in conjunction with the processor structure in Figure 7.1 to demonstrate a possible microprogrammed implementation.

A suitable microprogram is presented in flowchart form, for easier understanding, in Figure 7.20. Each box in the chart corresponds to a microinstruction that controls the transfers and operations indicated within the box. The microinstruction is located