CHAPTER 5

Processing Unit Design

In previous chapters, we studied the history of computer systems and the fundamental issues related to memory locations, addressing modes, assembly language, and computer arithmetic. In this chapter, we focus our attention on the main component of any computer system, the central processing unit (CPU). The primary function of the CPU is to execute a set of instructions stored in the computer's memory. A simple CPU consists of a set of registers, an arithmetic logic unit (ALU), and a control unit (CU). In what follows, the reader will be introduced to the organization and main operations of the CPU.

5.1. CPU BASICS

A typical CPU has three major components: (1) register set, (2) arithmetic logic unit (ALU), and (3) control unit (CU). The register set differs from one computer architecture to another. It is usually a combination of general-purpose and special-purpose registers. General-purpose registers are used for any purpose, hence the name general purpose. Special-purpose registers have specific functions within the CPU. For example, the program counter (PC) is a special-purpose register that is used to hold the address of the instruction to be executed next. Another example of special-purpose registers is the instruction register (IR), which is used to hold the instruction that is currently executed. The ALU provides the circuitry needed to perform the arithmetic, logical and shift operations demanded of the instruction set. In Chapter 4, we have covered a number of arithmetic operations and circuits used to support computation in an ALU. The control unit is the entity responsible for fetching the instruction to be executed from the main memory and decoding and then executing it. Figure 5.1 shows the main components of the CPU and its interactions with the memory system and the input/output devices.

The CPU fetches instructions from memory, reads and writes data from and to memory, and transfers data from and to input/output devices. A typical and

Fundamentals of Computer Organization and Architecture, by M. Abd-El-Barr and H. El-Rewini
ISBN 0-471-46741-3

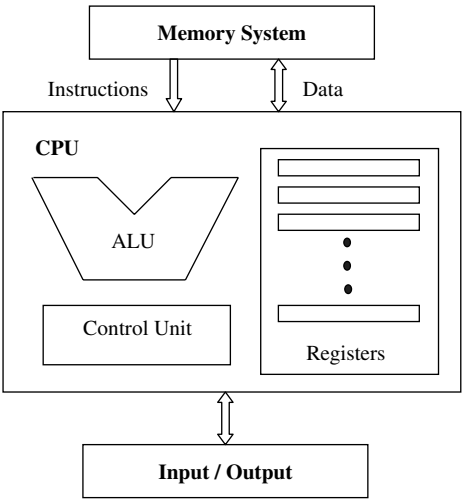

Figure 5.1 Central processing unit main components and interactions with the memory and I/O

simple execution cycle can be summarized as follows:

1. The next instruction to be executed, whose address is obtained from the PC, is fetched from the memory and stored in the IR.
2. The instruction is decoded.
3. Operands are fetched from the memory and stored in CPU registers, if needed.
4. The instruction is executed.
5. Results are transferred from CPU registers to the memory, if needed.

The execution cycle is repeated as long as there are more instructions to execute. A check for pending interrupts is usually included in the cycle. Examples of interrupts include I/O device request, arithmetic overflow, or a page fault (see Chapter 7). When an interrupt request is encountered, a transfer to an interrupt handling routine takes place. Interrupt handling routines are programs that are invoked to collect the state of the currently executing program, correct the cause of the interrupt, and restore the state of the program.

The actions of the CPU during an execution cycle are defined by micro-orders issued by the control unit. These micro-orders are individual control signals sent over dedicated control lines. For example, let us assume that we want to execute an instruction that moves the contents of register *X* to register *Y*. Let us also assume that both registers are connected to the data bus, *D*. The control unit will issue a control signal to tell register *X* to place its contents on the data bus *D*. After some delay, another control signal will be sent to tell register *Y* to read from data bus *D*. The activation of the control signals is determined using either hardwired control or microprogramming. These concepts are explained later in this chapter.

The remainder of this chapter is organized as follows. Section 5.2 presents the register set and explains the different types of registers and their functions. In Section 5.3, we will understand what is meant by datapath and control. CPU instruction cycle and the control unit will be covered in Sections 5.4 and 5.5, respectively.

5.2. REGISTER SET

Registers are essentially extremely fast memory locations within the CPU that are used to create and store the results of CPU operations and other calculations. Different computers have different register sets. They differ in the number of registers, register types, and the length of each register. They also differ in the usage of each register. General-purpose registers can be used for multiple purposes and assigned to a variety of functions by the programmer. Special-purpose registers are restricted to only specific functions. In some cases, some registers are used only to hold data and cannot be used in the calculations of operand addresses. The length of a data register must be long enough to hold values of most data types. Some machines allow two contiguous registers to hold double-length values. Address registers may be dedicated to a particular addressing mode or may be used as address general purpose. Address registers must be long enough to hold the largest address. The number of registers in a particular architecture affects the instruction set design. A very small number of registers may result in an increase in memory references. Another type of registers is used to hold processor status bits, or flags. These bits are set by the CPU as the result of the execution of an operation. The status bits can be tested at a later time as part of another operation.

5.2.1. Memory Access Registers

Two registers are essential in memory write and read operations: the *memory data register* (MDR) and *memory address register* (MAR). The MDR and MAR are used exclusively by the CPU and are not directly accessible to programmers.

In order to perform a write operation into a specified memory location, the MDR and MAR are used as follows:

1. The word to be stored into the memory location is first loaded by the CPU into MDR.
2. The address of the location into which the word is to be stored is loaded by the CPU into a MAR.
3. A write signal is issued by the CPU.

Similarly, to perform a memory read operation, the MDR and MAR are used as follows:

1. The address of the location from which the word is to be read is loaded into the MAR.

2. A read signal is issued by the CPU.
3. The required word will be loaded by the memory into the MDR ready for use by the CPU.

5.2.2. Instruction Fetching Registers

Two main registers are involved in fetching an instruction for execution: the *program counter* (PC) and the *instruction register* (IR). The PC is the register that contains the address of the next instruction to be fetched. The fetched instruction is loaded in the IR for execution. After a successful instruction fetch, the PC is updated to point to the next instruction to be executed. In the case of a branch operation, the PC is updated to point to the branch target instruction after the branch is resolved, that is, the target address is known.

5.2.3. Condition Registers

Condition registers, or flags, are used to maintain status information. Some architectures contain a special program status word (PSW) register. The PSW contains bits that are set by the CPU to indicate the current status of an executing program. These indicators are typically for arithmetic operations, interrupts, memory protection information, or processor status.

5.2.4. Special-Purpose Address Registers

Index Register As covered in Chapter 2, in index addressing, the address of the operand is obtained by adding a constant to the content of a register, called the *index register.* The index register holds an address displacement. Index addressing is indicated in the instruction by including the name of the index register in parentheses and using the symbol *X* to indicate the constant to be added.

Segment Pointers As we will discuss in Chapter 6, in order to support segmentation, the address issued by the processor should consist of a segment number (base) and a displacement (or an offset) within the segment. A segment register holds the address of the base of the segment.

Stack Pointer As shown in Chapter 2, a stack is a data organization mechanism in which the last data item stored is the first data item retrieved. Two specific operations can be performed on a stack. These are the *Push* and the *Pop* operations. A specific register, called the *stack pointer* (SP), is used to indicate the stack location that can be addressed. In the stack push operation, the SP value is used to indicate the location (called the top of the stack). After storing (pushing) this value, the SP is incremented (in some architectures, e.g. *X86*, the SP is decremented as the stack grows low in memory).

5.2.5. 80×86 Registers

As discussed in Chapter 3, the Intel basic programming model of the 386, 486, and the *Pentium* consists of three register groups. These are the general-purpose registers, the segment registers, and the instruction pointer (program counter) and the flag register.

Figure 5.2 (which repeats Fig. 3.6) shows the three sets of registers. The first set consists of general purpose registers A, B, C, D, SI (source index), DI (destination index), SP (stack pointer), and BP (base pointer). The second set of registers consists of CS (code segment), SS (stack segment), and four data segment registers DS, ES, FS, and GS. The third set of registers consists of the instruction pointer (program counter) and the flags (status) register. Among the status bits, the first five are identical to those bits introduced as early as in the 8085 8-bit microprocessor. The next 6–11 bits are identical to those introduced in the 8086. The flags in the bits 12–14 were introduced in the 80286 while the 16–17 bits were introduced in the 80386. The flag in bit 18 was introduced in the 80486.

5.2.6. MIPS Registers

The MIPS CPU contains 32 general-purpose registers that are numbered 0–31. Register *x* is designated by $x. Register $zero always contains the hardwired value 0. Table 5.1 lists the registers and describes their intended use. Registers $at (1), $k0 (26), and $k1 (27) are reserved for use by the assembler and operating system. Registers $a0–$a3 (4–7) are used to pass the first four arguments to routines

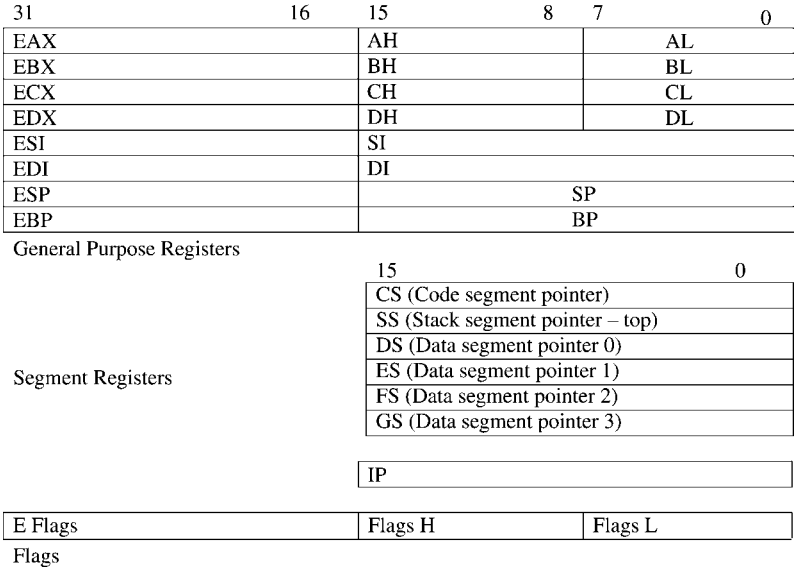

Figure 5.2 The main register sets in 80×86 (80386 and above extended all 16 bit registers except segment registers)

TABLE 5.1 MIPS General-Purpose Registers

Name	Number	Usage	Name	Number	Usage
zero	0	Constant 0	s0	16	Saved temporary (preserved across call)
at	1	Reserved for assembler	s1	17	Saved temporary (preserved across call)
v0	2	Expression evaluation and	s2	18	Saved temporary (preserved across call)
v1	3	results of a function	s3	19	Saved temporary (preserved across call)
a0	4	Argument 1	s4	20	Saved temporary (preserved across call)
a1	5	Argument 2	s5	21	Saved temporary (preserved across call)
a2	6	Argument 3	s6	22	Saved temporary (preserved across call)
a3	7	Argument 4	s7	23	Saved temporary (preserved across call)
t0	8	Temporary (not preserved across call)	t8	24	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)	t9	25	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)	k0	26	Reserved for OS kernel
t3	11	Temporary (not preserved across call)	k1	27	Reserved for OS kernel
t4	12	Temporary (not preserved across call)	gp	28	Pointer to global area
t5	13	Temporary (not preserved across call)	sp	29	Stack pointer
t6	14	Temporary (not preserved across call)	fp	30	Frame pointer
t7	15	Temporary (not preserved across call)	ra	31	Return address (used by function call)

(remaining arguments are passed on the stack). Registers $v0 and $v1 (2, 3) are used to return values from functions. Registers $t0–$t9 (8–15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls. Registers $s0–$s7 (16–23) are calle-saved registers that hold long-lived values that should be preserved across calls.

Register $sp(29) is the stack pointer, which points to the last location in use on the stack. Register $fp(30) is the frame pointer. Register $ra(31) is written with the return address for a function call. Register $gp(28) is a global pointer that points into the middle of a 64 K block of memory in the heap that holds constants and global variables. The objects in this heap can be quickly accessed with a single load or store instruction.

5.3. DATAPATH

The CPU can be divided into a data section and a control section. The data section, which is also called the datapath, contains the registers and the ALU. The datapath is capable of performing certain operations on data items. The control section is basically the control unit, which issues control signals to the datapath. Internal to the CPU, data move from one register to another and between ALU and registers. Internal data movements are performed via local buses, which may carry data, instructions, and addresses. Externally, data move from registers to memory and I/O devices, often by means of a system bus. Internal data movement among registers and between the ALU and registers may be carried out using different organizations including one-bus, two-bus, or three-bus organizations. Dedicated datapaths may also be used between components that transfer data between themselves more frequently. For example, the contents of the PC are transferred to the MAR to fetch a new instruction at the beginning of each instruction cycle. Hence, a dedicated datapath from the PC to the MAR could be useful in speeding up this part of instruction execution.

5.3.1. One-Bus Organization

Using one bus, the CPU registers and the ALU use a single bus to move outgoing and incoming data. Since a bus can handle only a single data movement within one clock cycle, two-operand operations will need two cycles to fetch the operands for the ALU. Additional registers may also be needed to buffer data for the ALU. This bus organization is the simplest and least expensive, but it limits the amount of data transfer that can be done in the same clock cycle, which will slow down the overall performance. Figure 5.3 shows a one-bus datapath consisting of a set of general-purpose registers, a memory address register (MAR), a memory data register (MDR), an instruction register (IR), a program counter (PC), and an ALU.

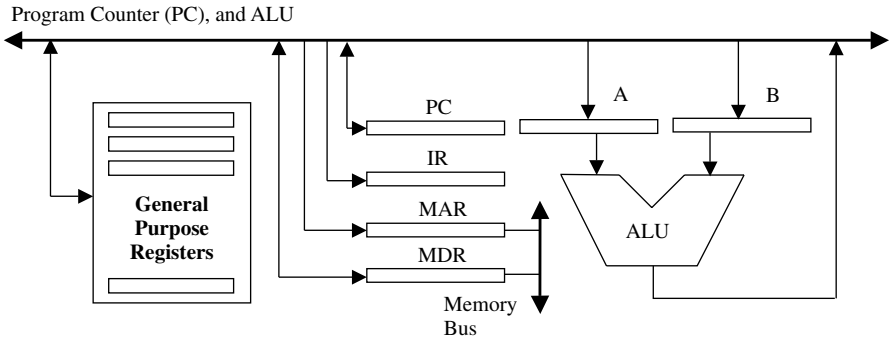

Figure 5.3 One-bus datapath

5.3.2. Two-Bus Organization

Using two buses is a faster solution than the one-bus organization. In this case, general-purpose registers are connected to both buses. Data can be transferred from two different registers to the input point of the ALU at the same time. Therefore, a two-operand operation can fetch both operands in the same clock cycle. An additional buffer register may be needed to hold the output of the ALU when the two buses are busy carrying the two operands. Figure 5.4*a* shows a two-bus organization. In some cases, one of the buses may be dedicated for moving data into registers (*in-bus*), while the other is dedicated for transferring data out of the registers (*out-bus*). In this case, the additional buffer register may be used, as one of the ALU inputs, to hold one of the operands. The ALU output can be connected directly to the in-bus, which will transfer the result into one of the registers. Figure 5.4*b* shows a two-bus organization with in-bus and out-bus.

5.3.3. Three-Bus Organization

In a three-bus organization, two buses may be used as source buses while the third is used as destination. The source buses move data out of registers (*out-bus*), and

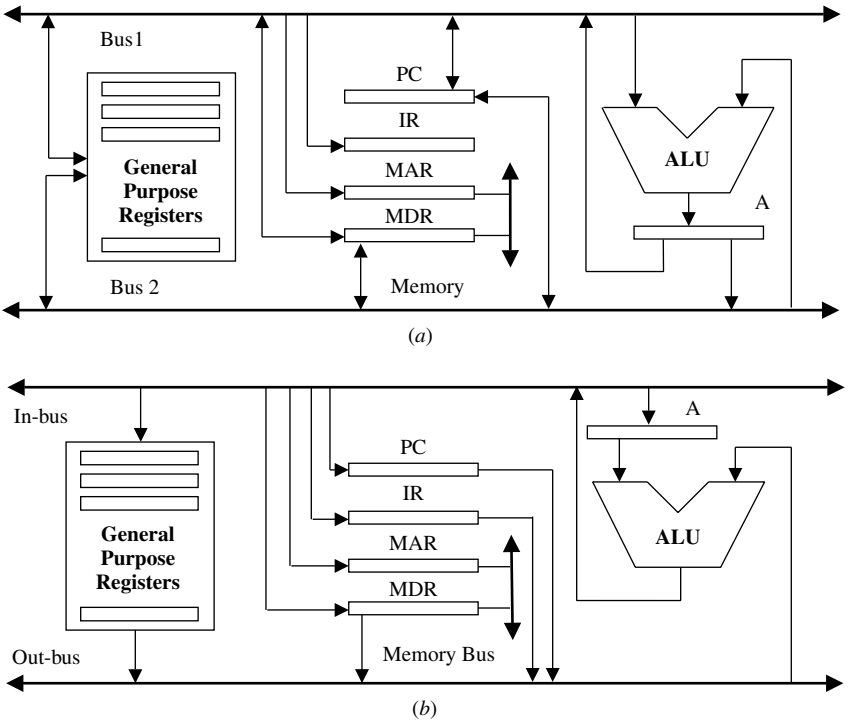

Figure 5.4 Two-bus organizations. (*a*) An Example of Two-Bus Datapath. (*b*) Another Example of Two-Bus Datapath with in-bus and out-bus

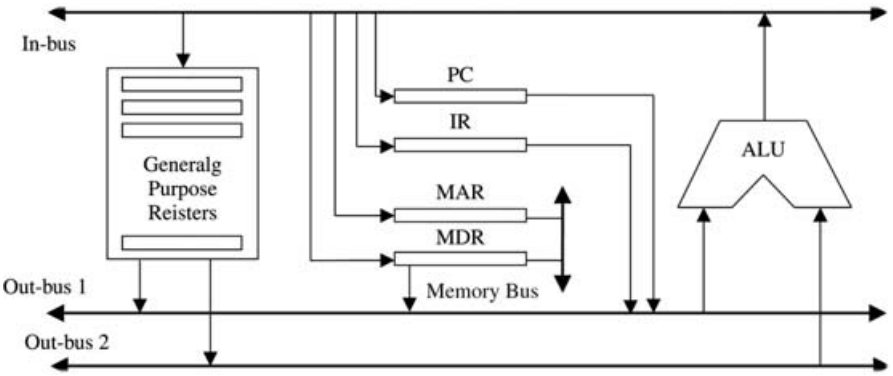

Figure 5.5 Three-bus datapath

the destination bus may move data into a register (*in-bus*). Each of the two out-buses is connected to an ALU input point. The output of the ALU is connected directly to the in-bus. As can be expected, the more buses we have, the more data we can move within a single clock cycle. However, increasing the number of buses will also increase the complexity of the hardware. Figure 5.5 shows an example of a three-bus datapath.

5.4. CPU INSTRUCTION CYCLE

The sequence of operations performed by the CPU during its execution of instructions is presented in Fig. 5.6. As long as there are instructions to execute, the next instruction is fetched from main memory. The instruction is executed based on the operation specified in the opcode field of the instruction. At the completion of the instruction execution, a test is made to determine whether an interrupt has occurred. An interrupt handling routine needs to be invoked in case of an interrupt.

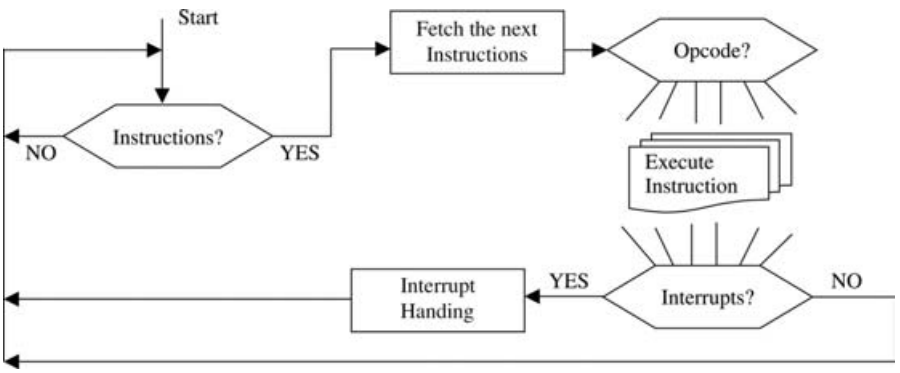

Figure 5.6 CPU functions

The basic actions during fetching an instruction, executing an instruction, or handling an interrupt are defined by a sequence of micro-operations. A group of control signals must be enabled in a prescribed sequence to trigger the execution of a micro-operation. In this section, we show the micro-operations that implement instruction fetch, execution of simple arithmetic instructions, and interrupt handling.

5.4.1. Fetch Instructions

The sequence of events in fetching an instruction can be summarized as follows:

1. The contents of the PC are loaded into the MAR.
2. The value in the PC is incremented. (This operation can be done in parallel with a memory access.)
3. As a result of a memory read operation, the instruction is loaded into the MDR.
4. The contents of the MDR are loaded into the IR.

Let us consider the one-bus datapath organization shown in Fig. 5.3. We will see that the fetch operation can be accomplished in three steps as shown in the table below, where $t_0 < t_1 < t_2$. Note that multiple operations separated by ";" imply that they are accomplished in parallel.

Step	Micro-operation
t_0	MAR $\leftarrow$ (PC); A $\leftarrow$ (PC)
t_1	MDR $\leftarrow$ Mem[MAR]; PC $\leftarrow$ (A) + 4
t_2	IR $\leftarrow$ (MDR)

Using the three-bus datapath shown in Figure 5.5, the following table shows the steps needed.

Step	Micro-operation
t_0	MAR $\leftarrow$ (PC); PC $\leftarrow$ (PC) + 4
t_1	MDR $\leftarrow$ Mem[MAR]
t_2	IR $\leftarrow$ (MDR)

5.4.2. Execute Simple Arithmetic Operation

Add R_1, R_2, R_0 This instruction adds the contents of source registers R_1 and R_2, and stores the results in destination register R_0. This addition can be executed as follows:

1. The registers R_0, R_1, R_2, are extracted from the IR.
2. The contents of R_1 and R_2 are passed to the ALU for addition.
3. The output of the ALU is transferred to R_0.

Using the one-bus datapath shown in Figure 5.3, this addition will take three steps as shown in the following table, where $t_0 < t_1 < t_2$.

Step	Micro-operation
t_0	$A \leftarrow (R_1)$
t_1	$B \leftarrow (R_2)$
t_2	$R_0 \leftarrow (A) + (B)$

Using the two-bus datapath shown in Figure 5.4*a*, this addition will take two steps as shown in the following table, where $t_0 < t_1$.

Step	Micro-operation
t_0	$A \leftarrow (R_1) + (R_2)$
t_1	$R_0 \leftarrow (A)$

Using the two-bus datapath with in-bus and out-bus shown in Figure 5.4*b*, this addition will take two steps as shown below, where $t_0 < t_1$.

Step	Micro-operation
t_0	$A \leftarrow (R_1)$
t_1	$R_0 \leftarrow (A) + (R_2)$

Using the three-bus datapath shown in Figure 5.5, this addition will take only one step as shown in the following table.

Step	Micro-operation
t_0	$R_0 \leftarrow (R_1) + (R_2)$

Add X, R_0 This instruction adds the contents of memory location X to register R_0 and stores the result in R_0. This addition can be executed as follows:

1. The memory location X is extracted from IR and loaded into MAR.
2. As a result of memory read operation, the contents of X are loaded into MDR.
3. The contents of MDR are added to the contents of R_0.

Using the one-bus datapath shown in Figure 5.3, this addition will take five steps as shown below, where $t_0 < t_1 < t_2 < t_3 < t_4$.

Step	Micro-operation
t_0	MAR $\leftarrow$ X
t_1	MDR $\leftarrow$ Mem[MAR]
t_2	$A \leftarrow (R_0)$
t_3	$B \leftarrow (MDR)$
t_4	$R_0 \leftarrow (A) + (B)$

Using the two-bus datapath shown in Figure 5.4*a*, this addition will take four steps as shown below, where $t_0 < t_1 < t_2 < t_3$.

Step	Micro-operation
t_0	MAR $\leftarrow$ X
t_1	MDR $\leftarrow$ Mem[MAR]
t_2	$A \leftarrow (R_0) + (MDR)$
t_3	$R_0 \leftarrow (A)$

Using the two-bus datapath with in-bus and out-bus shown in Figure 5.4*b*, this addition will take four steps as shown below, where $t_0 < t_1 < t_2 < t_3$.

Step	Micro-operation
t_0	MAR $\leftarrow$ X
t_1	MDR $\leftarrow$ Mem[MAR]
t_2	$A \leftarrow (R_0)$
t_3	$R_0 \leftarrow (A) + (MDR)$

Using the three-bus datapath shown in Figure 5.5, this addition will take three steps as shown below, where $t_0 < t_1 < t_2$.

Step	Micro-operation
t_0	MAR $\leftarrow$ X
t_1	MDR $\leftarrow$ Mem[MAR]
t_2	$R_0 \leftarrow R_0 + (MDR)$

5.4.3. Interrupt Handling

After the execution of an instruction, a test is performed to check for pending interrupts. If there is an interrupt request waiting, the following steps take place:

1. The contents of PC are loaded into MDR (to be saved).
2. The MAR is loaded with the address at which the PC contents are to be saved.
3. The PC is loaded with the address of the first instruction of the interrupt handling routine.

4. The contents of MDR (old value of the PC) are stored in memory.

The following table shows the sequence of events, where $t_1 < t_2 < t_3$.

Step	Micro-operation
t_1	MDR ← (PC)
t_2	MAR ← address1 (where to save old PC);
	PC ← address2 (interrupt handling routine)
t_3	Mem[MAR] ← (MDR)

5.5. CONTROL UNIT

The control unit is the main component that directs the system operations by sending control signals to the datapath. These signals control the flow of data within the CPU and between the CPU and external units such as memory and I/O. Control buses generally carry signals between the control unit and other computer components in a clock-driven manner. The system clock produces a continuous sequence of pulses in a specified duration and frequency. A sequence of steps $t_0, t_1, t_2, \ldots,$

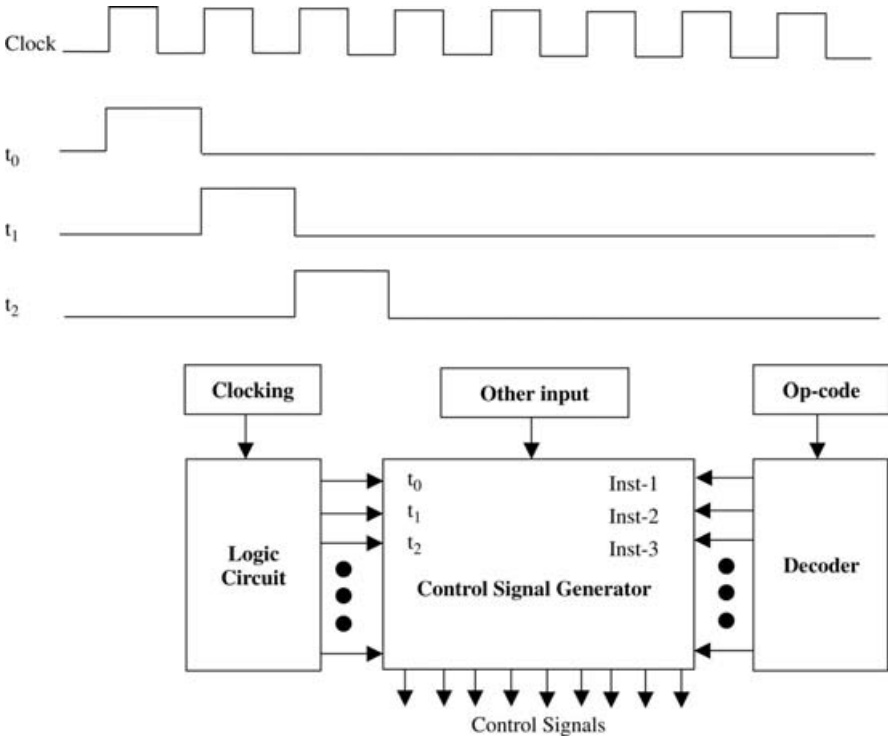

Figure 5.7 Timing of control signals

($t_0 < t_1 < t_2 < \cdots$) are used to execute a certain instruction. The op-code field of a fetched instruction is decoded to provide the control signal generator with information about the instruction to be executed. Step information generated by a logic circuit module is used with other inputs to generate control signals. The signal generator can be specified simply by a set of Boolean equations for its output in terms of its inputs. Figure 5.7 shows a block diagram that describes how timing is used in generating control signals.

There are mainly two different types of control units: *microprogrammed* and *hardwired.* In microprogrammed control, the control signals associated with operations are stored in special memory units inaccessible by the programmer as control words. A control word is a microinstruction that specifies one or more microoperations. A sequence of microinstructions is called a microprogram, which is stored in a ROM or RAM called a control memory CM.

In hardwired control, fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals. Clearly hardwired control is faster than microprogrammed control. However, hardwired control could be very expensive and complicated for complex systems. Hardwired control is more economical for small control units. It should also be noted that microprogrammed control could adapt easily to changes in the system design. We can easily add new instructions without changing hardware. Hardwired control will require a redesign of the entire systems in the case of any change.

Example 1 Let us revisit the add operation in which we add the contents of source registers R_1, R_2, and store the results in destination register R_0. We have shown earlier that this operation can be done in one step using the three-bus datapath shown in Figure 5.5.

Let us try to examine the control sequence needed to accomplish this addition at step t_0. Suppose that the op-code field of the current instruction was decoded to Inst-x type. First we need to select the source registers and the destination register, then we select Add as the ALU function to be performed. The following table shows the needed step and the control sequence.

Step	Instruction type	Micro-operation	Control
t_0	Inst-x	$R_0 \leftarrow (R_1) + (R_2)$	Select R_1 as source 1 on out-bus1 (R_1 out-bus1) Select R_2 as source 2 on out-bus2 (R_2 out-bus2) Select R_0 as destination on in-bus (R_0 in-bus) Select the ALU function Add (Add)

Figure 5.8 shows the signals generated to execute Inst-x during time period t_0. The AND gate ensures that these signals will be issued when the op-code is decoded into Inst-x and during time period t_0. The signals (R_1 out-bus 1), (R_2 out-bus2),

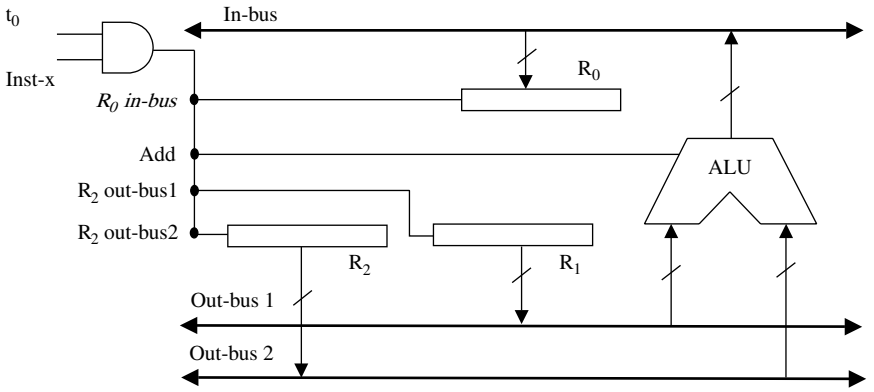

Figure 5.8 Signals generated to execute Inst-x on three-bus datapath during time period t_0

(R_0 in-bus), and (Add) will select R_1 as a source on out-bus1, R_2 as a source on out-bus2, R_0 as destination on in-bus, and select the ALUs add function, respectively.

Example 2 Let us repeat the operation in the previous example using the one-bus datapath shown in Fig. 5.3. We have shown earlier that this operation can be carried out in three steps using the one-bus datapath. Suppose that the op-code field of the current instruction was decoded to Inst-x type. The following table shows the needed steps and the control sequence.

Step	Instruction type	Micro-operation	
t_0	Inst-x	$A \leftarrow (R_1)$	Select R_1 as source (R_1 out) Select A as destination (A in)
t_1	Inst-x	$B \leftarrow (R_2)$	Select R_2 as source (R_2 out) Select B as destination (B in)
t_2	Inst-x	$R_0 \leftarrow (A) + (B)$	Select the ALU function Add (Add) Select R_0 as destination (R_0 in)

Figure 5.9 shows the signals generated to execute Inst-x during time periods t_0, t_1, and t_2. The AND gates ensure that the appropriate signals will be issued when the op-code is decoded into Inst-x and during the appropriate time period. During t_0, the signals (R_1 out) and (A in) will be issued to move the contents of R_1 into A. Similarly during t_1, the signals (R_2 out) and (B in) will be issued to move the contents of R_2 into B. Finally, the signals (R_0 in) and (Add) will be issued during t_2 to add the contents of A and B and move the results into R_0.

5.5.1. Hardwired Implementation

In hardwired control, a direct implementation is accomplished using logic circuits. For each control line, one must find the Boolean expression in terms of the input to the control signal generator as shown in Figure 5.7. Let us explain the

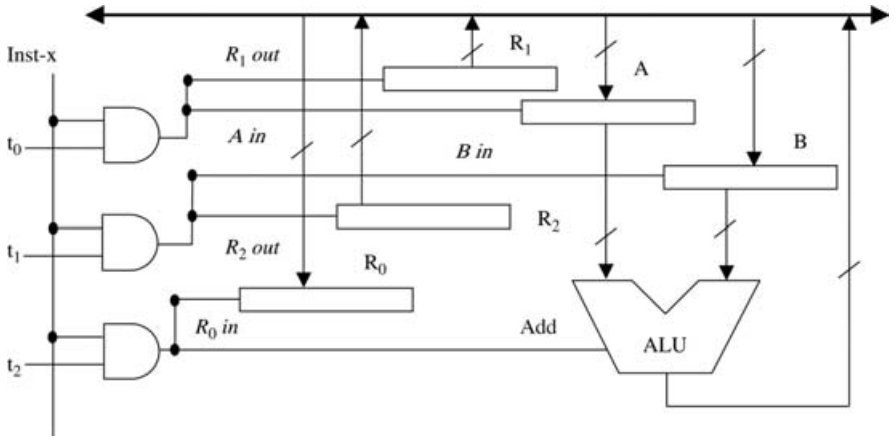

Figure 5.9 Signals generated to execute Inst-x on one-bus datapath during time period t_0, t_1, t_2

implementation using a simple example. Assume that the instruction set of a machine has the three instructions: Inst-x, Inst-y, and Inst-z; and A, B, C, D, E, F, G, and H are control lines. The following table shows the control lines that should be activated for the three instructions at the three steps t_0, t_1, and t_2.

Step	Inst-x	Inst-y	Inst-z
t_0	D, B, E	F, H, G	E, H
t_1	C, A, H	G	D, A, C
t_2	G, C	B, C	

The Boolean expressions for control lines A, B, and C can be obtained as follows:

$$\begin{aligned}
\text{A} &= \text{Inst-x} \cdot t_1 + \text{Inst-z} \cdot t_1 = (\text{Inst-x} + \text{Inst-z}) \cdot t_1 \\
\text{B} &= \text{Inst-x} \cdot t_0 + \text{Inst-y} \cdot t_2 \\
\text{C} &= \text{Inst-x} \cdot t_1 + \text{Inst-x} \cdot t_2 + \text{Inst-y} \cdot t_2 + \text{Inst-z} \cdot t_1 \\
&= (\text{Inst-x} + \text{Inst-z}) \cdot t_1 + (\text{Inst-x} + \text{Inst-y}) \cdot t_2
\end{aligned}$$

Figure 5.10 shows the logic circuits for these control lines. Boolean expressions for the rest of the control lines can be obtained in a similar way. Figure 5.11 shows the state diagram in the execution cycle of these instructions.

5.5.2. Microprogrammed Control Unit

The idea of microprogrammed control units was introduced by M. V. Wilkes in the early 1950s. Microprogramming was motivated by the desire to reduce the complexities involved with hardwired control. As we studied earlier, an instruction is

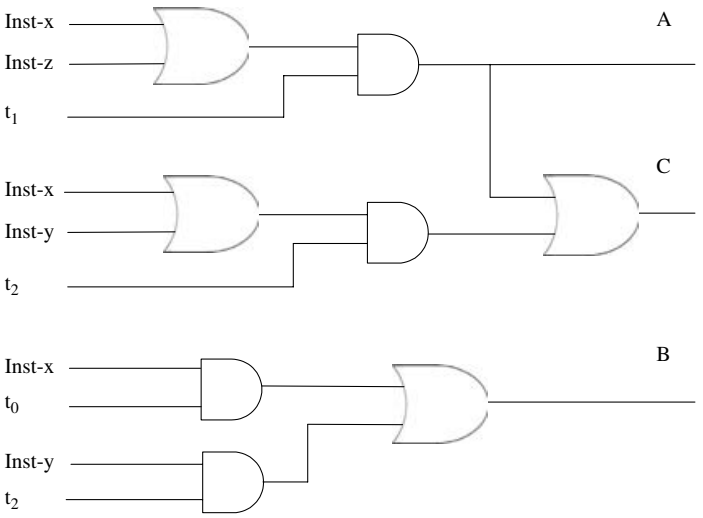

Figure 5.10 Logic circuits for control lines A, B, and C

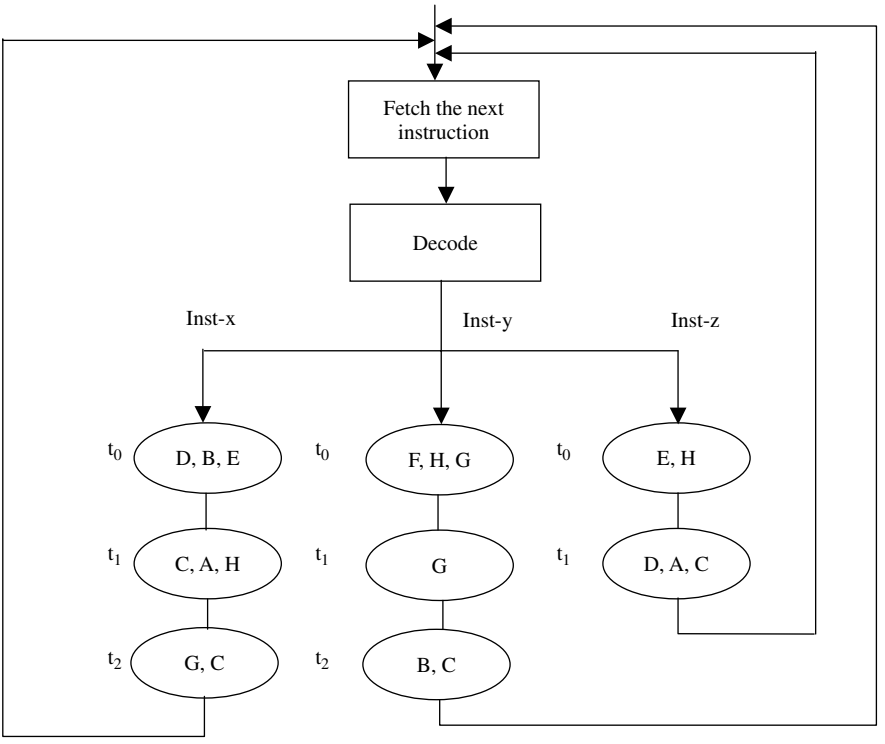

Figure 5.11 Instruction execution state diagram

implemented using a set of micro-operations. Associated with each micro-operation is a set of control lines that must be activated to carry out the corresponding micro-operation. The idea of microprogrammed control is to store the control signals associated with the implementation of a certain instruction as a microprogram in a special memory called a control memory (CM). A microprogram consists of a sequence of microinstructions. A microinstruction is a vector of bits, where each bit is a control signal, condition code, or the address of the next microinstruction. Microinstructions are fetched from CM the same way program instructions are fetched from main memory (Fig. 5.12).

When an instruction is fetched from memory, the op-code field of the instruction will determine which microprogram is to be executed. In other words, the op-code is mapped to a microinstruction address in the control memory. The microinstruction processor uses that address to fetch the first microinstruction in the microprogram. After fetching each microinstruction, the appropriate control lines will be enabled. Every control line that corresponds to a "1" bit should be turned *on.* Every control line that corresponds to a "0" bit should be left *off.* After completing the execution of one microinstruction, a new microinstruction will be fetched and executed. If the condition code bits indicate that a branch must be taken, the next microinstruction is specified in the address bits of the current microinstruction. Otherwise, the next microinstruction in the sequence will be fetched and executed.

The length of a microinstruction is determined based on the number of micro-operations specified in the microinstructions, the way the control bits will be interpreted, and the way the address of the next microinstruction is obtained. A microinstruction may specify one or more micro-operations that will be activated simultaneously. The length of the microinstruction will increase as the number of parallel micro-operations per microinstruction increases. Furthermore, when each control bit in the microinstruction corresponds to exactly one control line, the length of microinstruction could get bigger. The length of a microinstruction could be reduced if control lines are coded in specific fields in the microinstruction. Decoders will be needed to map each field into the individual control lines. Clearly, using the decoders will reduce the number of control lines that can be activated simultaneously. There is a tradeoff between the length of the microinstructions and the amount of parallelism. It is important that we reduce the length of microinstructions to reduce the cost and access time of the control memory. It may also be desirable that more micro-operations be performed in parallel and more control lines can be activated simultaneously.

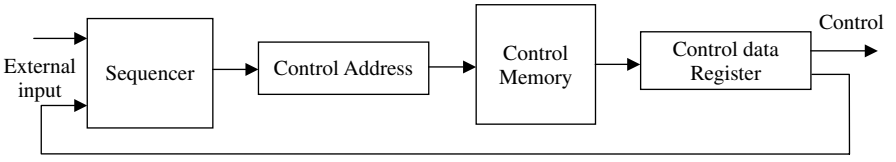

Figure 5.12 Fetching microinstructions (control words)

Horizontal Versus Vertical Microinstructions Microinstructions can be classified as *horizontal* or *vertical.* Individual bits in horizontal microinstructions correspond to individual control lines. Horizontal microinstructions are long and allow maximum parallelism since each bit controls a single control line. In vertical microinstructions, control lines are coded into specific fields within a microinstruction. Decoders are needed to map a field of k bits to 2^k possible combinations of control lines. For example, a 3-bit field in a microinstruction could be used to specify any one of eight possible lines. Because of the encoding, vertical microinstructions are much shorter than horizontal ones. Control lines encoded in the same field cannot be activated simultaneously. Therefore, vertical microinstructions allow only limited parallelism. It should be noted that no decoding is needed in horizontal microinstructions while decoding is necessary in the vertical case.

Example 3 Consider the three-bus datapath shown in Figure 5.5. In addition to the PC, IR, MAR, and MDR, assume that there are 16 general-purpose registers numbered R_0–R_{15}. Also, assume that the ALU supports eight functions (add, subtract, multiply, divide, AND, OR, shift left, and shift right). Consider the add operation Add R_1, R_2, R_0, which adds the contents of source registers R_1, R_2, and store the results in destination register R_0. In this example, we will study the format of the microinstruction under horizontal organization.

We will use horizontal microinstructions, in which there is a control bit for each control line. The format of the microinstruction should have control bits for the following:

- ALU operations
- Registers that output to out-bus1 (source 1)
- Registers that output to out-bus2 (source 2)
- Registers that input from in-bus (destination)
- Other operations that are not shown here

The following table shows the number of bits needed for ALU, Source 1, Source 2, and destination:

Purpose	Number of bits	Explanations
ALU	8 bits	8 functions
Source 1	20 bits	16 general-purpose registers + 4 special-purpose registers
Source 2	16 bits	16 general-purpose registers
Destination	20 bits	16 general-purpose registers + 4 special-purpose registers

Figure 5.13 is the microinstruction for Add R_1, R_2, R_0 on the three-bus datapath.

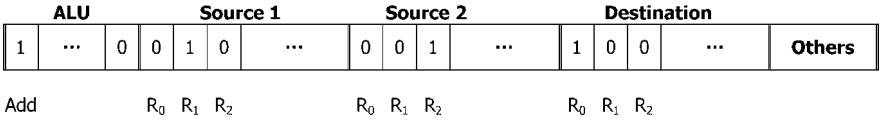

Figure 5.13 Microinstruction for Add R_1, R_2, R_0

Example 4 In this example, we will use vertical microinstructions, in which decoders will be needed. We will use a three-bus datapath as shown in Figure 5.5. Assume that there are 16 general-purpose registers and that the ALU supports eight functions. The following tables show the encoding for ALU functions, registers connected to out-bus 1 (Source 1), registers connected to out-bus 2 (Source 2), and registers connected to in-bus (Destination).

Purpose	Number of bits	Explanations
ALU	4 bits	8 functions + none
Source 1	5 bits	16 general-purpose registers + 4 special-purpose registers + none
Source 2	5 bits	16 general-purpose registers + none
Destination	5 bits	16 general-purpose registers + 4 special-purpose registers + none

Encoding	ALU function
0000	None (ALU will connect out-bus1 to in-bus)
0001	Add
0010	Subtract
0011	Multiple
0100	Divide
0101	AND
0110	OR
0111	Shift left
1000	Shift right

Encoding	Source 1	Destination	Encoding	Source 2
00000	R_0	R_0	00000	R_0
00001	R_1	R_1	00001	R_1
00010	R_2	R_2	00010	R_2
00011	R_3	R_3	00011	R_3
00100	R_4	R_4	00100	R_4
00101	R_5	R_5	00101	R_5
00110	R_6	R_6	00110	R_6
00111	R_7	R_7	00111	R_7
01000	R_8	R_8	01000	R_8
01001	R_9	R_9	01001	R_9
01010	R_{10}	R_{10}	01010	R_{10}

Encoding	Source 1	Destination	Encoding	Source 2
01011	R_{11}	R_{11}	01011	R_{11}
01100	R_{12}	R_{12}	01100	R_{12}
01101	R_{13}	R_{13}	01101	R_{13}
01110	R_{14}	R_{14}	01110	R_{14}
01111	R_{15}	R_{15}	01111	R_{15}
10000	PC	PC	10000	None
10001	IR	IR		
10010	MAR	MAR		
10011	MDR	MDR		
10100	NONE	NONE		

Figure 5.14 is the microinstruction for Add R_1, R_2, R_0 using the three-bus datapath under vertical organization:

Example 5 Using the same encoding of Example 4, let us find vertical microinstructions used in fetching an instruction.

MAR ← PC First, we need to select PC as source 1 by using "10000" for source 1 field. Similarly, we select MAR as our destination by using "10010" in the destination field. We also need to use "0000" for the ALU field, which will be decoded to "NONE". As shown in the ALU encoding table (Example 4), "NONE" means that out-bus1 will be connected to in-bus. The field source 2 will be set to "10000", which means none of the registers will be selected. The microinstruction is shown in Figure 5.15.

Memory Read and Write Memory operations can easily be accommodated by adding 1 bit for read and another for write. The two microinstructions in Figure 5.16 perform memory read and write, respectively.

Fetch Fetching an instruction can be done using the three microinstructions of Figure 5.17.

The first and second microinstructions have been shown above. The third microinstruction moves the contents of the MDR to IR (IR ← MDR). MDR is selected as source 1 by using "10011" for source 1 field. Similarly, IR is selected as the destination by using "10001" in the destination field. We also need to use "0000" ("NONE")

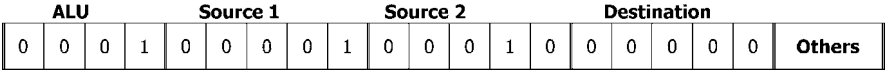

Figure 5.14 Microinstruction for Add R_1, R_2, R_0

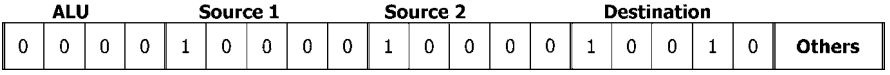

Figure 5.15 Microinstruction for *MAR ← PC*

				R	W	
ALU	**Source 1**	**Source 2**	**Destination**	1	0	**Others**

MDR ← Mem[MAR]

				R	W	
ALU	**Source 1**	**Source 2**	**Destination**	0	1	**Others**

Mem[MAR] ← MDR

Figure 5.16 Microinstructions for memory read and write

	ALU				Source 1					Source 2					Destination					R	W	
MAR ← (PC)	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	0	**Others**
MDR ← Mem[MAR]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**Others**
IR ← MDR	0	0	0	0	1	0	0	1	1	1	0	0	0	0	1	0	0	0	1	0	0	**Others**

Figure 5.17 Microinstructions for fetching an instruction

in the ALU field, which means that out-bus1 will be connected to in-bus. The field source 2 will be set to "10000", which means none of the registers will be selected.

5.6. SUMMARY

The CPU is the part of a computer that interprets and carries out the instructions contained in the programs we write. The CPU's main components are the register file, ALU, and the control unit. The register file contains general-purpose and special registers. General-purpose registers may be used to hold operands and intermediate results. The special registers may be used for memory access, sequencing, status information, or to hold the fetched instruction during decoding and execution. Arithmetic and logical operations are performed in the ALU. Internal to the CPU, data may move from one register to another or between registers and ALU. Data may also move between the CPU and external components such as memory and I/O. The control unit is the component that controls the state of the instruction cycle. As long as there are instructions to execute, the next instruction is fetched from main memory. The instruction is executed based on the operation specified in the op-code field of the instruction. The control unit generates signals that control the flow of data within the CPU and between the CPU and external units such as memory and I/O. The control unit can be implemented using hardwired or microprogramming techniques.

EXERCISES

1. How many instruction bits are required to specify the following:
   (a) Two operand registers and one result register in a machine that has 64 general-purpose registers?

(b) Three memory addresses in a machine with 64 KB of main memory?

2. Show the micro-operations of the *load*, *store*, and *jump* instructions using:
   (a) One-bus system
   (b) Two-bus system
   (c) Three-bus system
3. Add control signals to all the tables in Section 5.4.
4. Data movement within the CPU can be performed in several different ways. Contrast the following methods in terms of their advantages and disadvantages:
   (a) Dedicated connections
   (b) One-bus datapath
   (c) Two-bus datapath
   (d) Three-bus datapath
5. Find a method of encoding the microinstructions described by the following table so that the minimum number of control bits is used and all inherent parallelism among the microoperations is preserved.

Microinstruction	Control signals activated
I_1	*a*, *b*, *c*, *d*, *e*
I_2	*a*, *d*, *f*, *g*
I_3	*b*, *h*
I_4	*c*
I_5	*c*, *e*, *g*, *i*
I_6	*a*, *h*, *j*

6. Suppose that the instruction set of a machine has three instructions: Inst-1, Inst-2, and Inst-3; and A, B, C, D, E, F, G, and H are the control lines. The following table shows the control lines that should be activated for the three instructions at the three steps T0, T1, and T2.

Step	Inst-1	Inst-2	Inst-3
T0	D, B, E	F, H, G	E, H
T1	C, A, H	G	D, A, C
T2	G, C	B, C	

   (a) Hardwired approach:
      (i) Write Boolean expressions for all the control lines A–G.
      (ii) Draw the logic circuit for each control line.
   (b) Microprogramming approach:
      (i) Assuming a horizontal representation, write down the microprogram for instructions Inst-1. Indicate the microinstruction size.

(ii) If we allow both horizontal and vertical representation, what would be the best grouping? What is the microinstruction size? Write the microprogram of Inst-1.

7. A certain processor has a microinstruction format containing 10 separate control fields C_0: C_9. Each C_i can activate any one of n_i distinct control lines, where n_i is specified as follows:

i:	0	1	2	3	4	5	6	7	8	9
n_i:	4	4	3	11	9	16	7	1	8	22

   (a) What is the minimum number of control bits needed to represent the 10 control fields?
   (b) What is the maximum number of control bits needed if a purely horizontal format is used for all control information?

8. What are the main differences between the following pairs?
   (a) Vertical and horizontal microinstructions
   (b) Microprogramming and hardwired control

9. Using the single-bus architecture, generate the necessary control signals, in the proper order (with minimum number of micro-instructions), for *conditional branch* instruction.

10. Write a micro-program for the fetch instruction using the one-bus datapath and the two-bus datapath.

REFERENCES AND FURTHER READING

R. J. Baron and L. Higbie, *Computer Architecture*, Addison Wesley, Canada, 1992.

M. J. Flynn, *Computer Architecture*, Jones and Barlett, MA, USA, 1995.

J. P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, New York, 1998.

J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, *CA*, 2003.

V. P. Heuring and H. F. Jordan, *Computer Systems Design and Architecture*, Addison Wesley, NJ, 1997.

M. Murdocca and V. Heuring, *Principles of Computer Architecture*, Prentice Hall, NJ, USA, 2000.

D. Patterson and J. Hennessy, *Computer Organization and Design*, Morgan Kaufmann, San Mateo, CA, 1998.

W. Stallings, *Computer Organization and Architecture: Designing for Performance*, NJ, 1996.

A. S. Tanenbaum, *Structured Computer Organization*, Prentice Hall, NJ, USA, 1999.