Table 15.5 IA-64 Application Registers

Kernel registers (KR0-7)	Convey information from the operating system to the application.
Register stack configuration (RSC)	Controls the operation of the register stack engine (RSE).
RSE Backing store pointer (BSP)	Holds the address in memory that is the save location for r32 in the current stack frame.
RSE Backing store pointer to memory stores (BSPSTORE)	Holds the address in memory to which the RSE will spill the next value.
RSE NaT collection register (RNAT)	Used by the RSE to temporarily hold NaT bits when it is spilling general registers.
Compare and exchange value (CCV)	Contains the compare value used as the third source operand in the cmpxchg instruction.
User NaT collection register (UNAT)	Used to temporarily hold NaT bits when saving and restoring general registers with the ld8.fill and st8.spill instructions.
Floating-point status register (FPSR)	Controls traps, rounding mode, precision control, flags, and other control bits for floating-point instructions.
Interval time counter (ITC)	Counts up at a fixed relationship to the processor clock frequency.
Previous function state (PFS)	Saves value in CFM register and related information.
Loop count (LC)	Used in counted loops and is decremented by counted-loop-type branches.
Epilog count (EC)	Used for counting the final (epilog) state in modulo-scheduled loops.

- **Processor identifiers:** Describe processor implementation-dependent features.
- **Application registers:** A collection of special-purpose registers. Table 15.5 provides a brief definition of each.

Register Stack

The register stack mechanism in IA-64 avoids unnecessary movement of data into and out of registers at procedure call and return. The mechanism automatically provides a called procedure with a new **frame** of up to 96 registers (r32 through r127) upon procedure entry. The compiler specifies the number of registers required by a procedure with the alloc instruction, which specifies how many of these are local (used only within the procedure) and how many are output (used to pass parameters to a procedure called by this procedure). When a procedure call occurs, the IA-64 hardware renames registers so that the local registers from the previous frame are hidden and what were the output registers of the calling procedure now have register numbers starting at r32 in the called procedure. Physical registers in the range r32 through r127 are allocated in a circular-buffer fashion to virtual registers associated with procedures. That is, the next register allocated after r127 is r32. When necessary, the hardware moves register contents between registers and

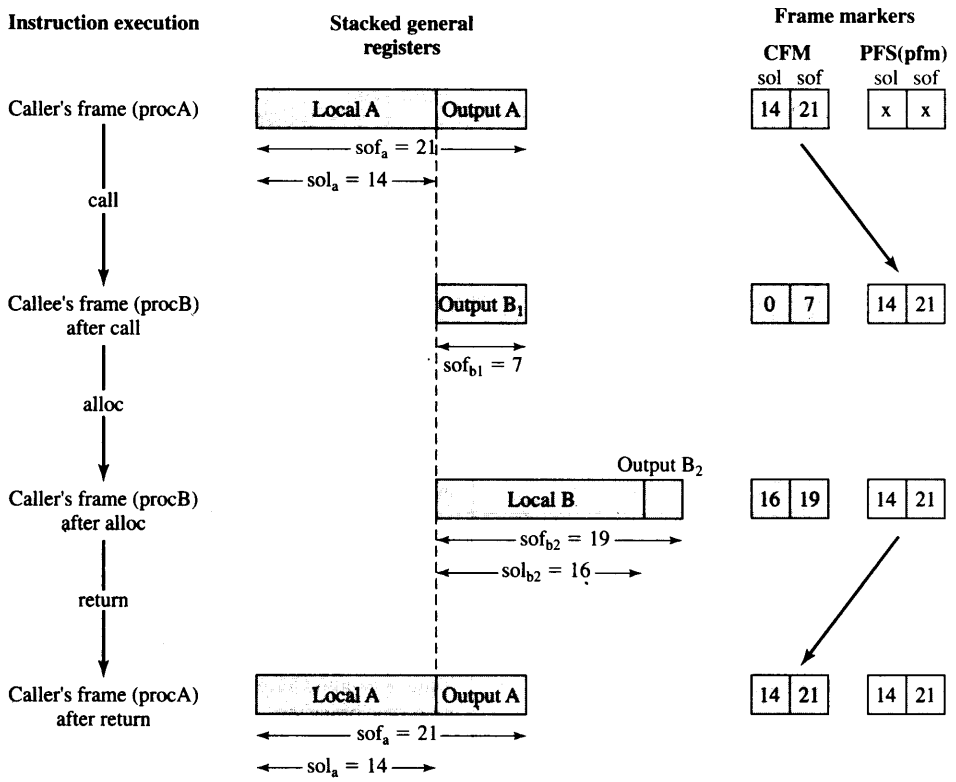


Figure 15.8 Register Stack Behavior on Procedure Call and Return

memory to free up additional registers when procedure calls occur, and restores contents from memory to registers as procedure returns occur.

Figure 15.8 illustrates register stack behavior. The alloc instruction includes sof (size of frame) and sol (size of locals) operands to specify the required number of registers. These values are stored in the CFM register. When a call occurs, the sol and sof values from the CFM are stored in the sol and sof fields of the previous function state (PFS) application register (Figure 15.9). Upon return these sol and sof values must be restored from the PFS to the CFM. To allow nested calls and returns, previous values of the PFS fields must be saved through successive calls so that they can be restored through successive returns. This is a function of the alloc instruction, which designates a general register to save the current value of the PFS fields before they are overwritten from the CFM fields.

Current Frame Marker and Previous Function State

The CFM register describes the state of the current general register stack frame, associated with the currently active procedure. It includes the following fields:

- **sof:** size of stack frame
- **sol:** size of locals portion of stack frame

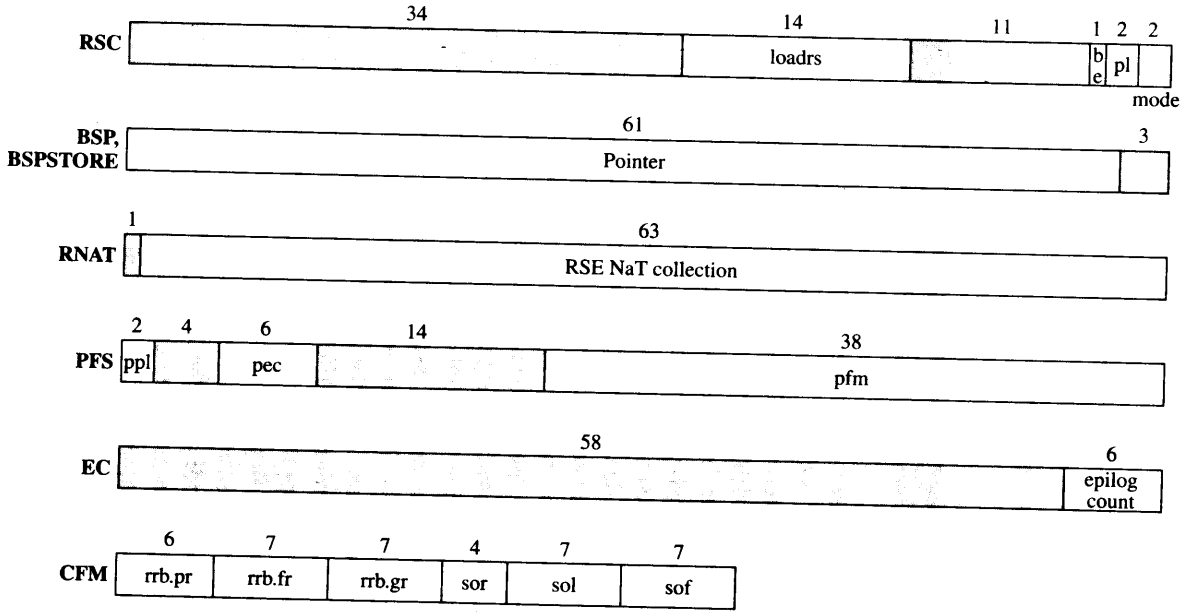


Figure 15.9 Formats of Some IA-64 Registers

- **sor:** size of rotating portion of stack frame; this is a subset of the local portion that is dedicated to software pipelining
- **register rename base values:** Values used in performing register rotation general, floating-point and predicate registers

The PFS application register contains the following fields:

- **pfm:** Previous frame marker; contains all of the fields of the CFM
- **pec:** Previous epilog count
- **ppl:** Previous privilege level

15.5 ITANIUM ORGANIZATION

Intel's Itanium processor is the first implementation of the IA-64 instruction set architecture. The first version of this implementation, known as Itanium, was released in 2001, followed in 2002 by the Itanium 2. The Itanium organization blends superscalar features with support for the unique EPIC-related IA-64 features. Among the superscalar features are a six-wide, ten-stage-deep hardware pipeline, dynamic prefetch, branch prediction, and a register scoreboard to optimize for compile time nondeterminism. EPIC related hardware includes support for predicated execution, control and data speculation, and software pipelining.

Figure 15.10 is a general block diagram of the Itanium organization. The Itanium includes nine execution units: two integer, two floating-point, four memory, and three branch execution units. Instructions are fetched through an L1 instruction cache and fed into a buffer that holds up to eight bundles of instructions. When deciding on functional units for instruction dispersal, the processor views at most two instruction bundles at a time. The processor can issue a maximum of six instructions per clock cycle.

The organization is in some ways simpler than a conventional contemporary superscalar organization. The Itanium does not use reservation stations, reorder buffers, and memory ordering buffers, all replaced by simpler hardware for speculation. The register remapping hardware is simpler than the register aliasing typical of superscalar machines. Register dependency-detection logic is absent, replaced by explicit parallelism directives precomputed by the software.

Using branch prediction, the fetch/prefetch engine can speculatively load an L1 instruction cache to minimize cache misses on instruction fetches. The fetched code is fed into a decoupling buffer that can hold up to eight bundles of code.

Three levels of cache are used. The L1 cache is split into a 16-kbyte instruction cache and a 16-kbyte data cache, each 4-way set associative with a 32-byte line size. The 256-kbyte L2 cache is 6-way set associative with a 64-byte line size. The 3-Mbyte L3 cache is 4-way set associative with a 64-byte line size. All three levels of cache are on the same chip as the processor for the Itanium 2. For the original Itanium, the L3 cache is off-chip but on the same package as the processor.

The Itanium 2 uses an 8-stage pipeline for all but floating-point instructions. Figure 15.11 illustrates the relationship between the pipeline stages and the Itanium 2 organization. The pipeline stages are as follows:

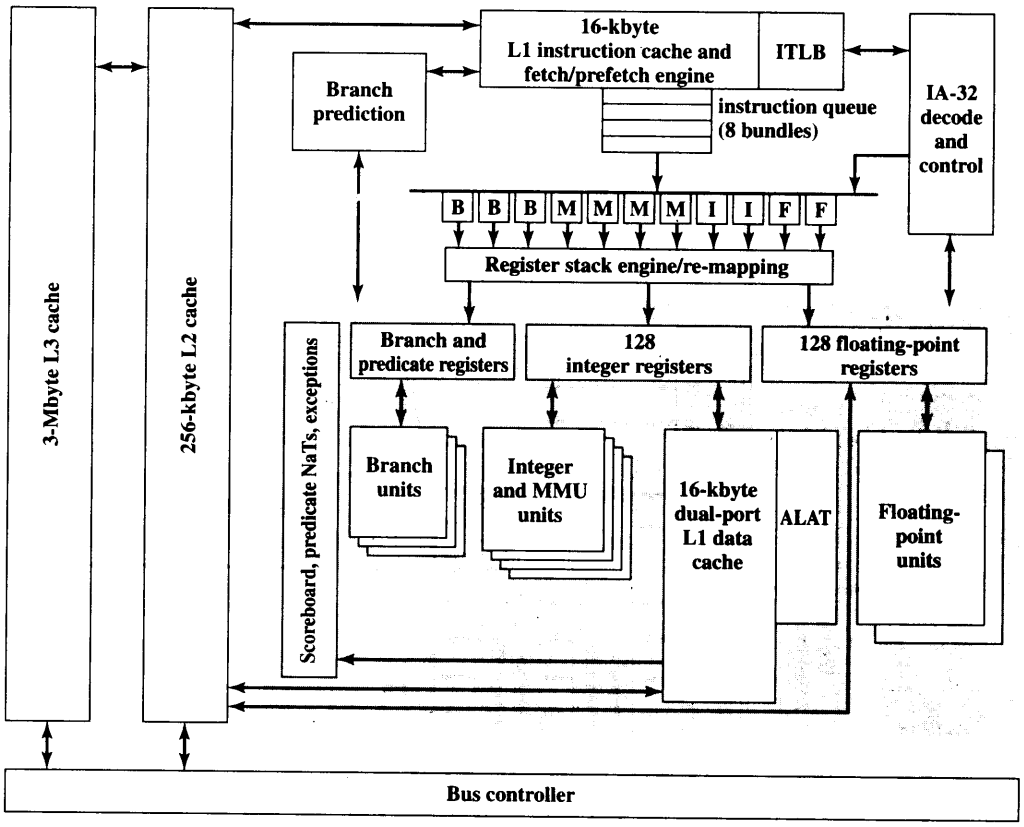


Figure 15.10 Itanium 2 Processor Organization

- **Instruction pointer generation (IPG):** Delivers and instruction pointer to the L1I cache.
- **Instruction rotation (ROT):** Fetch instructions and rotate instructions into position so that bundle 0 contains the first instruction that should be executed.
- **Instruction template decode, expand and disperse (EXP):** Decode instruction templates, and disperse up to 6 instructions through 11 ports in conjunction with opcode information for the execution units.
- **Rename and decode (REN):** Rename (remap) registers for the register stack engine; decode instructions.
- **Register file read (REG):** Delivers operands to execution units.
- **ALU execution (EXE):** Execute operations.
- **Last stage for exception detection (DET):** Detect exceptions; abandon result of execution if instruction predicate was not true; resteer mispredicted branches.
- **Write back (WRB):** Write results back to register file.

For floating-point instructions, the first five pipeline stages are the same as just listed, followed by four floating-point pipeline stages, followed by a write-back stage.

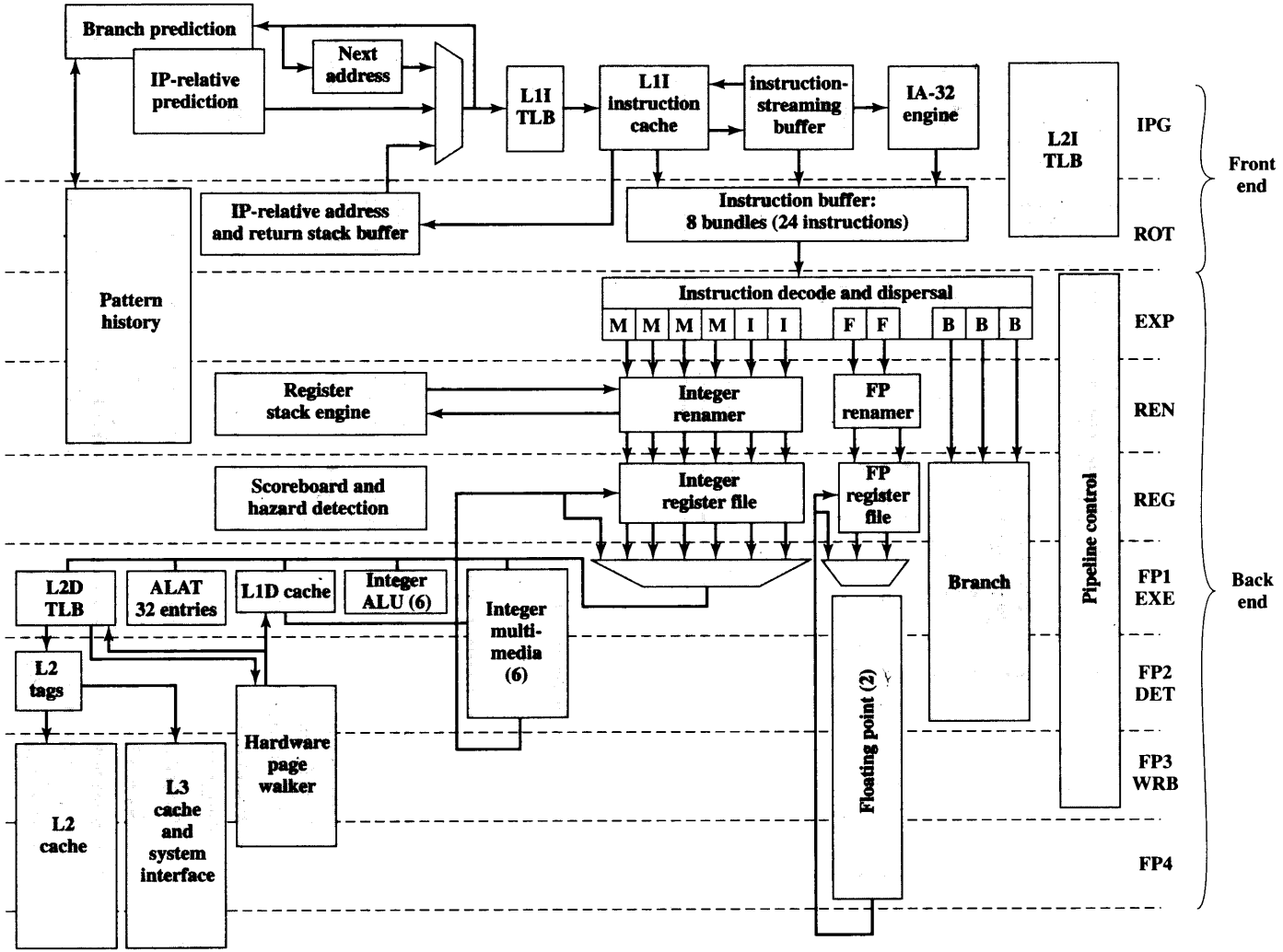


Figure 15.11 Itanium 2 Processor Pipeline [MCNA03]

15.6 RECOMMENDED READING AND WEB SITES

[HUCK00] provides an overview of IA-64; another overview is [DULO98]. [SCHL00a] provides a general discussion of EPIC; a more thorough treatment is provided in [SCHL00b]. Two other good treatments are [HWU01] and [KATH01]. [CHAS00] and [HWU98] provide introductions to predicated execution. Volume 1 of [INTE00a] contains a detailed treatment of software pipelining; two articles that provide a good explanation of the topic, with examples, are [JARP01] and [BHAR00].

For an overview of the Itanium processor architecture, see [SHAR00]; [INTE00b] provides a more detailed treatment. [MCNA03] and [NAFF02] describe the Itanium 2 in some detail.

[EVAN03], [TRIE01], and [MARK00] contain more detailed treatments of the topics of this chapter. Finally, for an exhaustive look at the IA-64 architecture and instruction set, see [INTE00a].

BHAR00 Bharandwaj, J., et al. "The Intel IA-64 Compiler Code Generator." *IEEE Micro*, September/October 2000.

CHAS00 Chasin, A. "Predication, Speculation, and Modern CPUs." *Dr. Dobb's Journal*, May 2000.

DULO98 Dulong, C. "The IA-64 Architecture at Work." *Computer*, July 1998.

EVAN03 Evans, J., and Trimper, G. *Itanium Architecture for Programmers*. Upper Saddle River, NJ: Prentice Hall, 2003.

HUCK00 Huck, J., et al. "Introducing the IA-64 Architecture." *IEEE Micro*, September/October 2000.

HWU98 Hwu, W. "Introduction to Predicated Execution." *Computer*, January 1998.

HWU01 Hwu, W.; August, D.; and Sias, J. "Program Decision Logic Optimization Using Predication and Control Speculation." *Proceedings of the IEEE*, November 2001.

INTE00a Intel Corp. *Intel IA-64 Architecture Software Developer's Manual (4 volumes)*. Document 245317 through 245320. Aurora, CO, 2000.

INTE00b Intel Corp. *Itanium Processor Microarchitecture Reference for Software Optimization*. Aurora, CO, Document 245473. August 2000.

JARP01 Jarp, S. "Optimizing IA-64 Performance." *Dr. Dobb's Journal*, July 2001.

KATH01 Kathail, B.; Schlansker, M.; and Rau, B. "Compiling for EPIC Architectures." *Proceedings of the IEEE*, November 2001.

MARK00 Markstein, P. *IA-64 and Elementary Functions*. Upper Saddle River, NJ: Prentice Hall PTR, 2000.

MCNA03 McNairy, C., and Soltis, D. "Itanium 2 Processor Microarchitecture." *IEEE Micro*, March–April 2003.

NAFF02 Naffziger, S., et al. "The Implementation of the Itanium 2 Microprocessor." *IEEE Journal of Solid-State Circuits*, November 2002.

SCHL00a Schlansker, M.; and Rau, B. "EPIC: Explicitly Parallel Instruction Computing." *Computer*, February 2000.

SCHL00b Schlansker, M.; and Rau, B. *EPIC: An Architecture for Instruction-Level Parallel Processors*. HPL Technical Report HPL-1999-111, Hewlett-Packard Laboratories (www.hpl.hp.com), February 2000.

SHAR00 Sharangpani, H., and Arona, K. "Itanium Processor Microarchitecture." *IEEE Micro*, September/October 2000.

TRIE01 Triebel, W. *Itanium Architecture for Software Developers*. Intel Press, 2001.

Recommended Web Sites:

- **Itanium:** Intel's site for the latest information on IA-64 and Itanium.
- **HP Itanium Technology site:** Good source of information.
- **IMPACT:** This is a site at the University of Illinois, where much of the research on predicated execution has been done. A number of papers on the subject are available.

15.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

advanced load	IA-64 architecture	software pipeline
branch predication	instruction completer	speculative loading
bundle	instruction group	stack frame
control speculation	Itanium	stop
data speculation	major opcode	syllable
execution unit	NaT bit	template field
explicitly parallel instruction computing (EPIC)	predicate register	very long instruction word (VLIW)
hoist	predication	
	register stack	

Review Questions

15.1 What are the different types of execution units for IA-64?
15.2 Explain the use of the template field in an IA-64 bundle.
15.3 What is the significance of a stop in the instruction stream?
15.4 Define predication and predicated execution.
15.5 How can predicates replace a conditional branch instruction?
15.6 Define control speculation.
15.7 What is the purpose of the NaT bit?
15.8 Define data speculation.
15.9 What is the difference between a hardware pipeline and a software pipeline?
15.10 What is the difference between stacked and rotating registers?

Problems

15.1 Suppose that an IA-64 opcode accepts three registers as operands and produces one register as a result. What is the maximum number of different operations that can be defined in one major opcode family?
15.2 What is the maximum effective number of major opcodes?
15.3 At a certain point in an IA-64 program, there are 10 A-type instructions and six floating-point instructions that can be issued concurrently. How many syllables may appear without any stops between them?

15.4 In Problem 15.3,
- **a.** How many cycles are required for a small IA-64 implementation having one floating-point unit, two integer units, and two memory units?
- **b.** How many cycles are required for the Itanium organization of Figure 15.10?

15.5 The initial Itanium implementation had two M-units and two I-units. Which of the templates in Table 15.3 cannot be paired as two bundles of instructions that could be executed completely in parallel?

15.6 An algorithm that can utilize four floating-point instructions per cycle is coded for IA-64. Should instruction groups contain four floating-point operations? What are the consequences if the machine on which the program runs has fewer than four floating-point units?

15.7 In Section 15.3, we introduced the following constructs for predicated execution:

```
        cmp.crel p2, p3 = a, b
(p1)    cmp.crel p2, p3 = a, b
```

where crel is a relation, such as eq, ne, etc.; p1, p2, and p3 are predicate registers; a is either a register or an immediate operand; and b is a register operand.

Fill in the following truth table:

p1	comparison	p2	p3
not present	0		
not present	1		
0	0		
0	1		
1	0		
1	1		

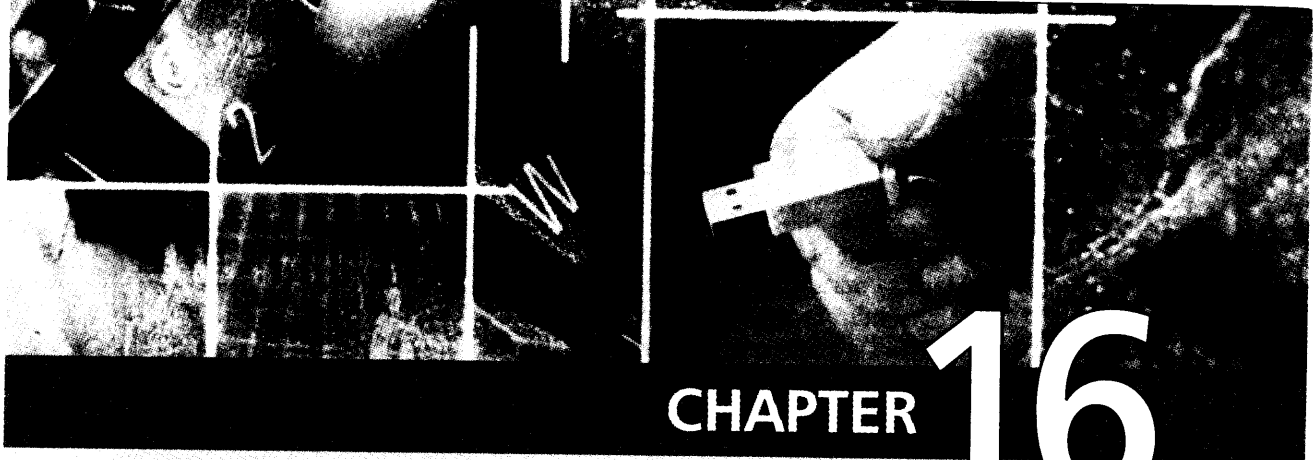
15.8 For the predicated program in Section 15.3, which implements the flowchart of Figure 15.4, indicate
- **a.** Those instructions that can be executed in parallel
- **b.** Those instructions that can be bundled into the same IA-64 instruction bundle

15.9 The IA-64 architecture includes a set of multimedia instructions comparable to those in the IA-32 Pentium architecture (Table 10.11). One such instruction type is the parallel compare instruction of the form pcmp1, pcmp2, or pcmp4, which does a parallel compare 1, 2, or 4 bytes at a time. The instruction pcmp1.gt ri = rj, rk compares the two source operands (rj, rk) byte by byte. For each byte, if the byte in rj is greater than the byte in rk, then the corresponding byte in ri is set to all ones; otherwise the destination byte is set to all zeros. Both operands are interpreted as signed.

Suppose the registers r14 and r15 contain the ASCII strings (see Table 7.1) "00000000" and "99999999" respectively and the register r16 contains an arbitrary string of eight characters. Determine whether the comments in the following code fragment are appropriate.

```
        pcmp1.gt    r8 = r14,r16      // if some char < "0" or
        pcmp1.gt    r9 = r16,r15 ;;   // if some char > "9"
        cmp.ne      p6,p0 = r8,r0 ;;  // p6 = true or
        cmp.ne      p7,p0 = r9,r0 ;;  // p7 = true so that
(p6)    br error                      // this branch executes or
(p7)    br error ;;                   // this branch executes
```

developed. Each instruction in the machine language of the processor is translated into a sequence of lower-level control unit instructions. These lower-level instructions are referred to as microinstructions, and the process of translation is referred to as microprogramming. The chapter describes the layout of a control memory containing a microprogram for each machine instruction is described. The structure and function of the microprogrammed control unit can then be explained.



CHAPTER 16

CONTROL UNIT OPERATION

KEY POINTS

- The execution of an instruction involves the execution of a sequence of substeps, generally called cycles. For example, an execution may consist of fetch, indirect, execute, and interrupt cycles. Each cycle is in turn made up of a sequence of more fundamental operations, called micro-operations. A single micro-operation generally involves a transfer between registers, a transfer between a register and an external bus, or a simple ALU operation.
- The control unit of a processor performs two tasks: (1) It causes the processor to execute micro-operations in the proper sequence, determined by the program being executed, and (2) it generates the control signals that cause each micro-operation to be executed.
- The control signals generated by the control unit cause the opening and closing of logic gates, resulting in the transfer of data to and from registers and the operation of the ALU.
- One technique for implementing a control unit is referred to as hard-wired implementation, in which the control unit is a combinatorial circuit. Its input logic signals, governed by the current machine instruction, are transferred into a set of output control signals.

In Chapter 10, we pointed out that a machine instruction set goes a long way toward defining the processor. If we know the machine instruction set, including an understanding of the effect of each opcode and an understanding of the addressing modes, and if we know the set of user-visible registers, then we know the functions that the processor must perform. This is not the complete picture. We must know the external interfaces, usually through a bus, and how interrupts are handled. With this line of reasoning, the following list of those things needed to specify the function of a processor emerges:

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

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

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

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

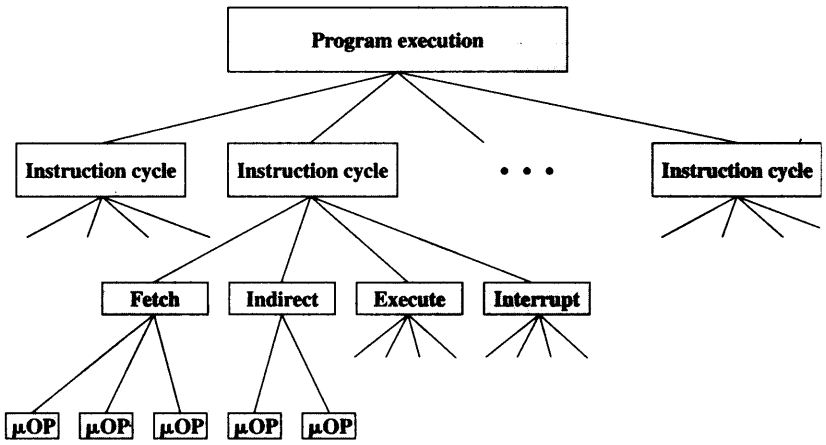
16.1 MICRO-OPERATIONS

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

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

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

Micro-operations are the functional, or atomic, operations of a processor. In this section, we will examine micro-operations to gain an understanding of how the



re 16.1 Constituent Elements of a Program Execution

events of any instruction cycle can be described as a sequence of such micro-operations. A simple example will be used. In the remainder of this chapter, we then show how the concept of micro-operations serves as a guide to the design of the control unit.

The Fetch Cycle

We begin by looking at the fetch cycle, which occurs at the beginning of each instruction cycle and causes an instruction to be fetched from memory. For purposes of discussion, we assume the organization depicted in Figure 12.6. Four registers are involved:

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

Let us look at the sequence of events for the fetch cycle from the point of view of its effect on the processor registers. An example appears in Figure 16.2. At the beginning of the fetch cycle, the address of the next instruction to be executed is in the program counter (PC); in this case, the address is 1100100. The first step is to move that address to the memory address register (MAR) because this is the only register connected to the address lines of the system bus. The second step is to bring in the instruction. The desired address (in the MAR) is placed on the address bus, the

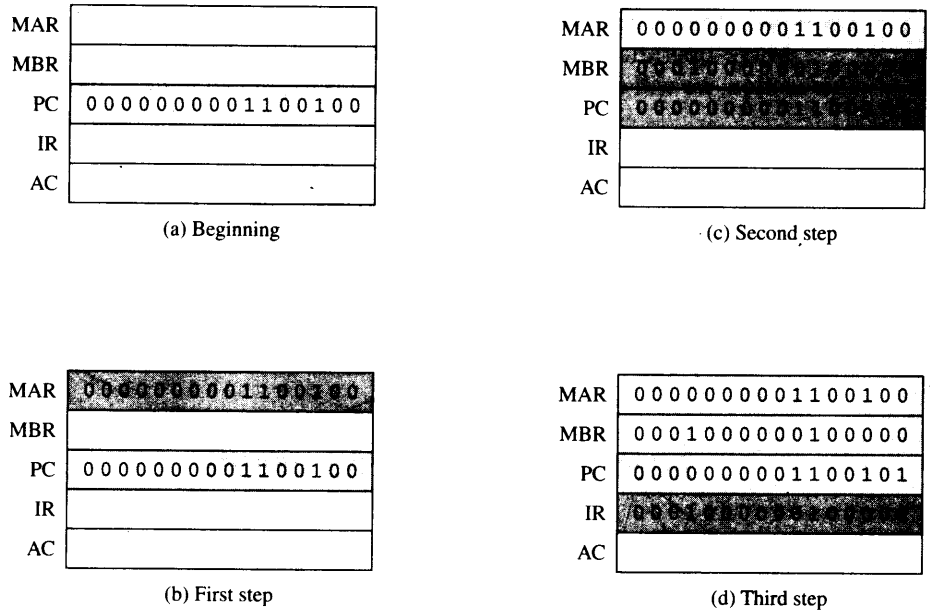


Figure 16.2 Sequence of Events, Fetch Cycle

control unit issues a READ command on the control bus, and the result appears on the data bus and is copied into the memory buffer register (MBR). We also need to increment the PC by 1 to get ready for the next instruction. Because these two actions (read word from memory, add 1 to PC) do not interfere with each other, we can do them simultaneously to save time. The third step is to move the contents of the MBR to the instruction register (IR). This frees up the MBR for use during a possible indirect cycle.

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

t_1: MAR ← (PC)
t_2: MBR ← Memory
PC ← (PC) + I
t_3: IR ← (MBR)

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

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

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

t_1: MAR ← (PC)
t_2: MBR ← Memory
t_3: PC ← (PC) + I
IR ← (MBR)

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

1. The proper sequence of events must be followed. Thus (MAR ← (PC)) must precede (MBR ← Memory) because the memory read operation makes use of the address in the MAR.
2. Conflicts must be avoided. One should not attempt to read to and write from the same register in one time unit, because the results would be unpredictable. For example, the micro-operations (MBR ← Memory) and (IR ← MBR) should not occur during the same time unit.

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

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

The Indirect Cycle

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

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

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

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

The Interrupt Cycle

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

```
t1: MBR ← (PC)
t2: MAR ← Save_Address
    PC ← Routine_Address
t3: Memory ← (MBR)
```

In the first step, the contents of the PC are transferred to the MBR, so that they can be saved for return from the interrupt. Then the MAR is loaded with the address at which the contents of the PC are to be saved, and the PC is loaded with the address of the start of the interrupt-processing routine. These two actions may each be a single micro-operation. However, because most processors provide multiple types and/or levels of interrupts, it may take one or more additional micro-operations to obtain the save_address and the routine_address before they can be transferred to the MAR and

PC, respectively. In any case, once this is done, the final step is to store the MBR, which contains the old value of the PC, into memory. The processor is now ready to begin the next instruction cycle.

The Execute Cycle

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

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

First, consider an add instruction:

```
ADD R1, X
```

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

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

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

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

```
ISZ X
```

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

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

The new feature introduced here is the conditional action. The PC is incremented if (MBR) = 0. This test and action can be implemented as one micro-operation. Note also that this micro-operation can be performed during the same time unit during which the updated value in MBR is stored back to memory.

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

```
BSA X
```

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

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

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

The Instruction Cycle

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

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

00: Fetch

01: Indirect

10: Execute

11: Interrupt

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

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

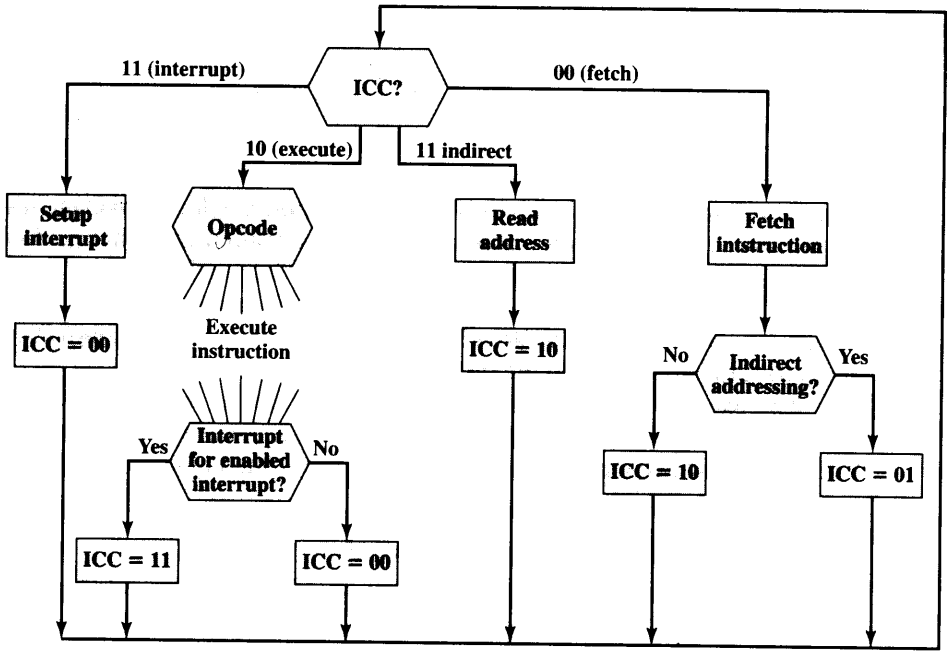


Figure 16.3 Flowchart for Instruction Cycle

16.2 CONTROL OF THE PROCESSOR

Functional Requirements

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

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

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

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

- ALU
- Registers

- Internal data paths
- External data paths
- Control unit

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

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

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

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

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

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

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

Control Signals

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

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

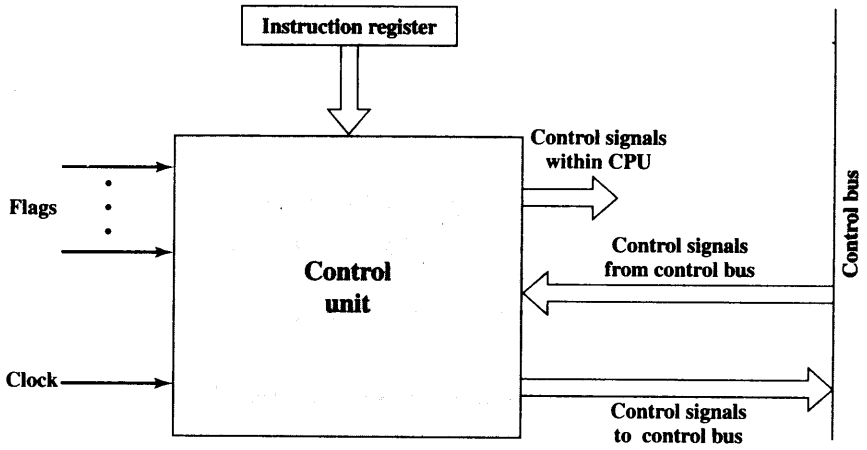


Figure 16.4 Block Diagram of the Control Unit

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

The outputs are

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

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

Let us consider again the fetch cycle to see how the control unit maintains control. The control unit keeps track of where it is in the instruction cycle. At a given point, it knows that the fetch cycle is to be performed next. The first step is to transfer the contents of the PC to the MAR. The control unit does this by activating the control

signal that opens the gates between the bits of the PC and the bits of the MAR. The next step is to read a word from memory into the MBR and increment the PC. The control unit does this by sending the following control signals simultaneously:

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

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

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

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

A Control Signals Example

To illustrate the functioning of the control unit, let us examine a simple example. Figure 16.5 illustrates the example. This is a simple processor with a single accumulator. The data paths between elements are indicated. The control paths for signals

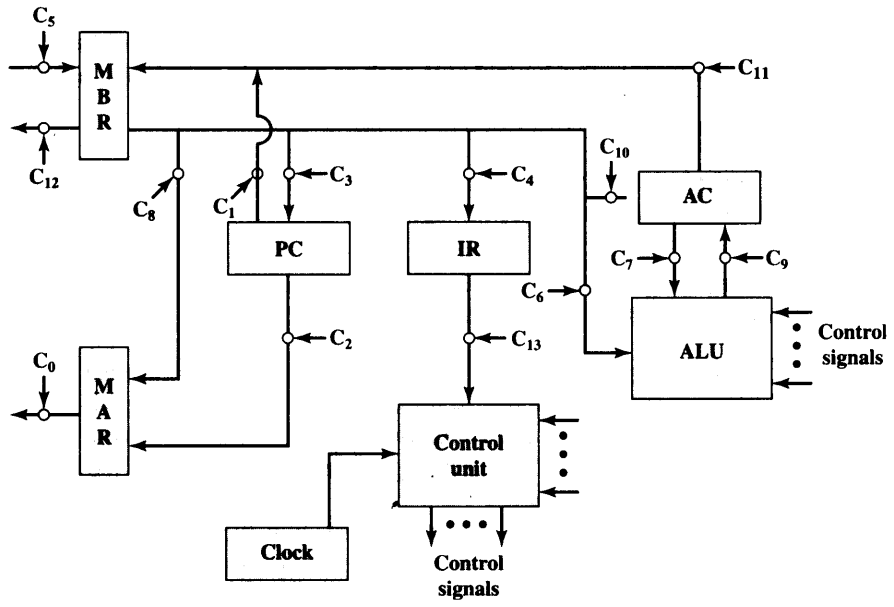


Figure 16.5 Data Paths and Control Signals

emanating from the control unit are not shown, but the terminations of control signals are labeled C_i and indicated by a circle. The control unit receives inputs from the clock, the instruction register, and flags. With each clock cycle, the control unit reads all of its inputs and emits a set of control signals. Control signals go to three separate destinations:

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

The control unit must maintain knowledge of where it is in the instruction cycle. Using this knowledge, and by reading all of its inputs, the control unit emits a sequence of control signals that causes micro-operations to occur. It uses the clock pulses to time the sequence of events, allowing time between events for signal levels to stabilize. Table 16.1 indicates the control signals that are needed for some of the micro-operation sequences described earlier. For simplicity, the data and control paths for incrementing the PC and for loading the fixed addresses into the PC and MAR are not shown.

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

Table 16.1 Micro-Operations and Control Signals

Micro-operations	Timing	Active Control Signals
Fetch:	t_1: MAR ← (PC)	C_2
	t_2: MBR ← Memory PC ← (PC) + 1	C_5, C_R
	t_3: IR ← (MBR)	C_4
Indirect:	t_1: MAR ← (IR(Address))	C_8
	t_2: MBR ← Memory	C_5, C_R
	t_3: IR(Address) ← (MBR(Address))	C_4
Interrupt:	t_1: MBR ← (PC)	C_1
	t_2: MAR ← Save-address PC ← Routine-address	
	t_3: Memory ← (MBR)	C_{12}, C_W

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

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

Internal Processor Organization

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

Using an internal processor bus, Figure 16.5 can be rearranged as shown in Figure 16.6. A single internal bus connects the ALU and all processor registers.

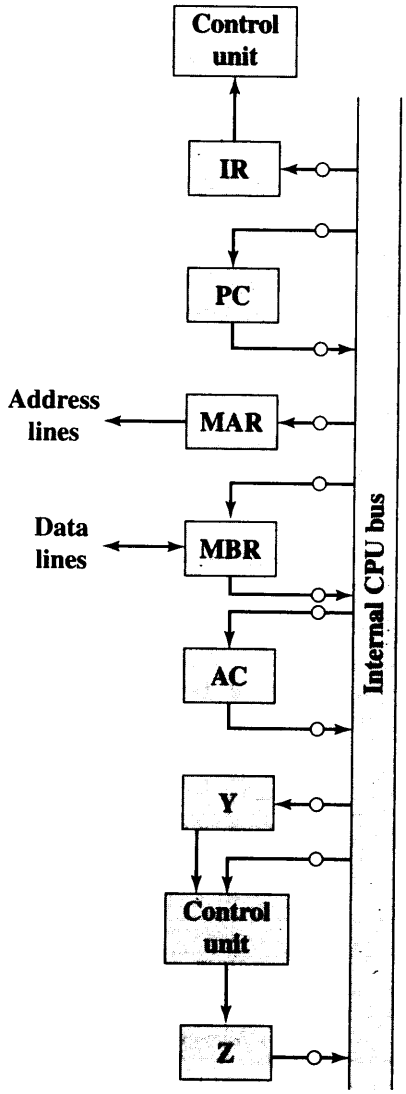


Figure 16.6 CPU with Internal Bus

Gates and control signals are provided for movement of data onto and off the bus from each register. Additional control signals control data transfer to and from the system (external) bus and the operation of the ALU.

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

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

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

The Intel 8085

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

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

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

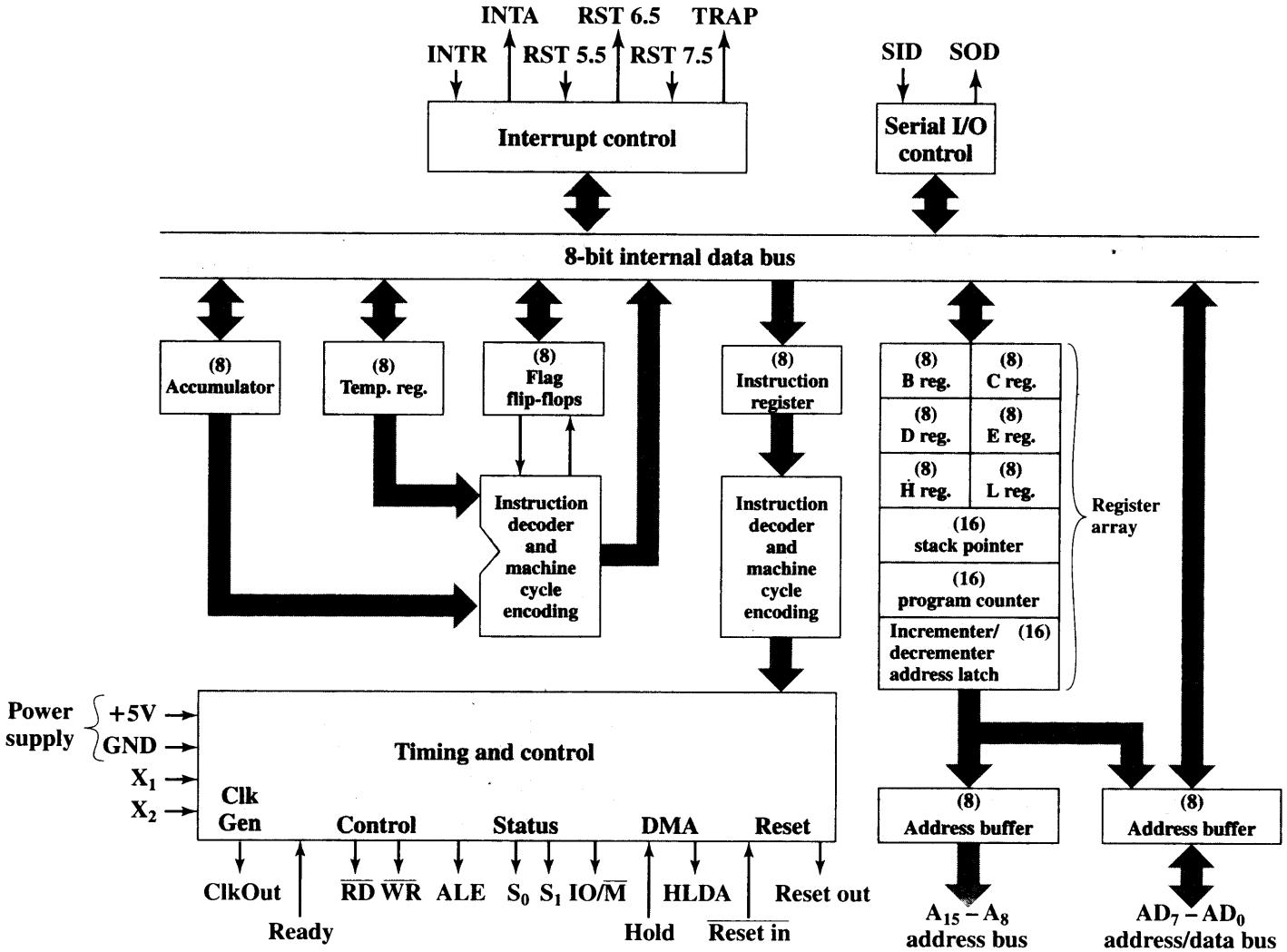


Figure 16.7 Intel 8085 CPU Block Diagram

Table 16.2 Intel 8085 External Signals

Address and Data Signals

High Address (A15–A8)
The high-order 8 bits of a 16-bit address.

Address/Data (AD7–AD0)
The lower-order 8 bits of a 16-bit address or 8 bits of data. This multiplexing saves on pins.

Serial Input Data (SID)
A single-bit input to accommodate devices that transmit serially (one bit at a time).

Serial Output Data (SOD)
A single-bit output to accommodate devices that receive serially.

Timing and Control Signals

CLK (OUT)
The system clock. Each cycle represents one T state. The CLK signal goes to peripheral chips and synchronizes their timing.

X1, X2
These signals come from an external crystal or other device to drive the internal clock generator.

Address Latch Enabled (ALE)
Occurs during the first clock state of a machine cycle and causes peripheral chips to store the address lines. This allows the address module (e.g., memory, I/O) to recognize that it is being addressed.

Status (S0, S1)
Control signals used to indicate whether a read or write operation is taking place.

IO/M
Used to enable either I/O or memory modules for read and write operations.

Read Control (RD)
Indicates that the selected memory or I/O module is to be read and that the data bus is available for data transfer.

Write Control (WR)
Indicates that data on the data bus is to be written into the selected memory or I/O location.

Memory and I/O Initiated Symbols

Hold
Requests the CPU to relinquish control and use of the external system bus. The CPU will complete execution of the instruction presently in the IR and then enter a hold state, during which no signals are inserted by the CPU to the control, address, or data buses. During the hold state, the bus may be used for DMA operations.

Hold Acknowledge (HOLDA)
This control unit output signal acknowledges the HOLD signal and indicates that the bus is now available.

READY
Used to synchronize the CPU with slower memory or I/O devices. When an addressed device asserts READY, the CPU may proceed with an input (DBIN) or output (WR) operation. Otherwise, the CPU enters a wait state until the device is ready.

(*Continued*)

Table 16.2 Continued

Interrupt-Related Signals

TRAP
Restart Interrupts (RST 7.5, 6.5, 5.5)

Interrupt Request (INTR)
These five lines are used by an external device to interrupt the CPU. The CPU will not honor the request if it is in the hold state or if the interrupt is disabled. An interrupt is honored only at the completion of an instruction. The interrupts are in descending order of priority.

Interrupt Acknowledge
Acknowledges an interrupt.

CPU Initialization

RESET IN
Causes the contents of the PC to be set to zero. The CPU resumes execution at location zero.

RESET OUT
Acknowledges that the CPU has been reset. The signal can be used to reset the rest of the system.

Voltage and Ground

VCC
+5 volt power supply

VSS
Electrical ground.

The control unit is identified as having two components labeled (1) instruction decoder and machine cycle encoding and (2) timing and control. A discussion of the first component is deferred until the next section. The essence of the control unit is the timing and control module. This module includes a clock and accepts as inputs the current instruction and some external control signals. Its output consists of control signals to the other components of the processor plus control signals to the external system bus.

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

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

Figure 16.9 gives an example of 8085 timing, showing the value of external control signals. Of course, at the same time, the control unit generates internal

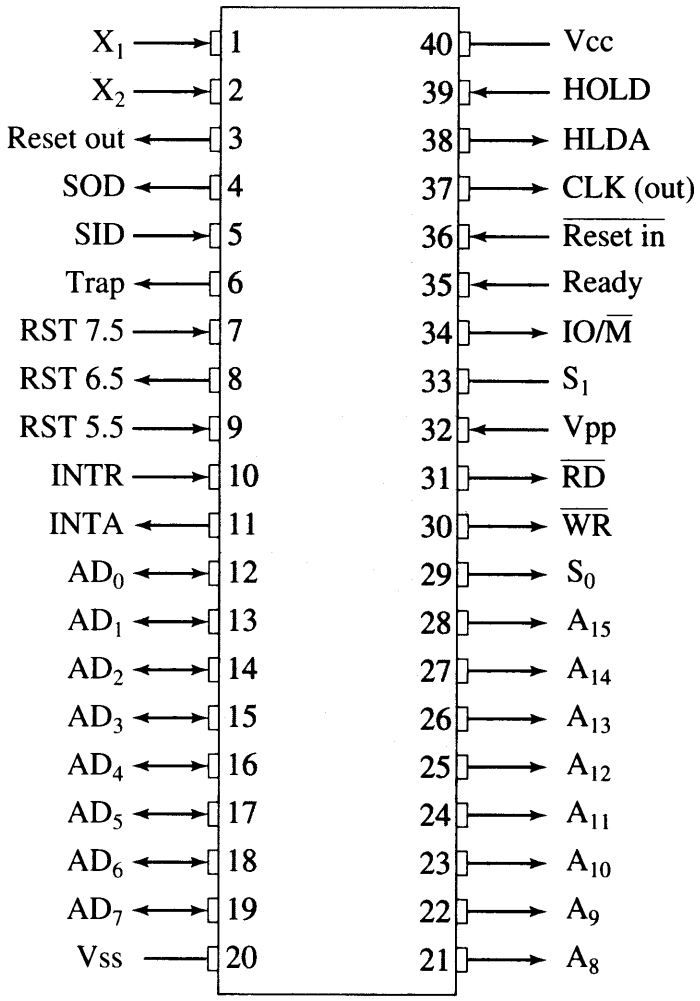


Figure 16.8 Intel 8085 Pin Configuration

control signals that control internal data transfers. The diagram shows the instruction cycle for an OUT instruction. Three machine cycles (M_1, M_2, M_3) are needed. During the first, the OUT instruction is fetched. The second machine cycle fetches the second half of the instruction, which contains the number of the I/O device selected for output. During the third cycle, the contents of the AC are written out to the selected device over the data bus.

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

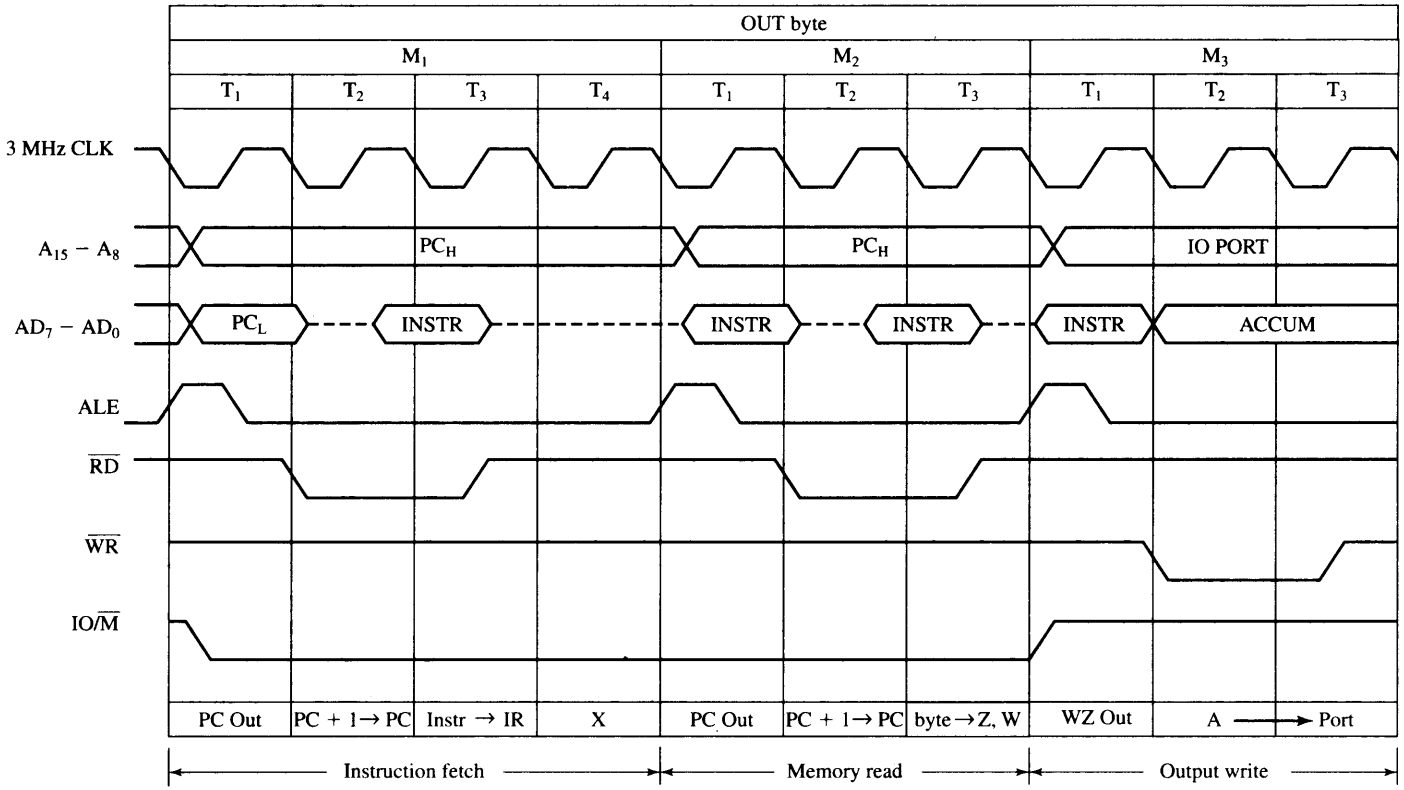


Figure 16.9 Timing Diagram for Intel 8085 OUT Instruction

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

16.3 HARDWIRED IMPLEMENTATION

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

- Hardwired implementation
- Microprogrammed implementation

In a hardwired implementation, the control unit is essentially a combinatorial circuit. Its input logic signals are transformed into a set of output logic signals, which are the control signals. This approach is examined in this section. Microprogrammed implementation is the subject of Chapter 17.

Control Unit Inputs

Figure 16.4 depicts the control unit as we have so far discussed it. The key inputs are the instruction register, the clock, flags, and control bus signals. In the case of the flags and control bus signals, each individual bit typically has some meaning (e.g., overflow). The other two inputs, however, are not directly useful to the control unit.

First consider the instruction register. The control unit makes use of the opcode and will perform different actions (issue a different combination of control signals) for different instructions. To simplify the control unit logic, there should be a unique logic input for each opcode. This function can be performed by a *decoder*, which takes an encoded input and produces a single output. In general, a decoder will have n binary inputs and 2^n binary outputs. Each of the 2^n different input patterns will activate a single unique output. Table 16.3 is an example. The decoder for a control unit will typically have to be more complex than that, to account for variable-length opcodes. An example of the digital logic used to implement a decoder is presented in Appendix A.

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

Table 16.3 A Decoder with Four Inputs and Sixteen Outputs

I1	I2	I3	I4	O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12	O13	O14	O15	O16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

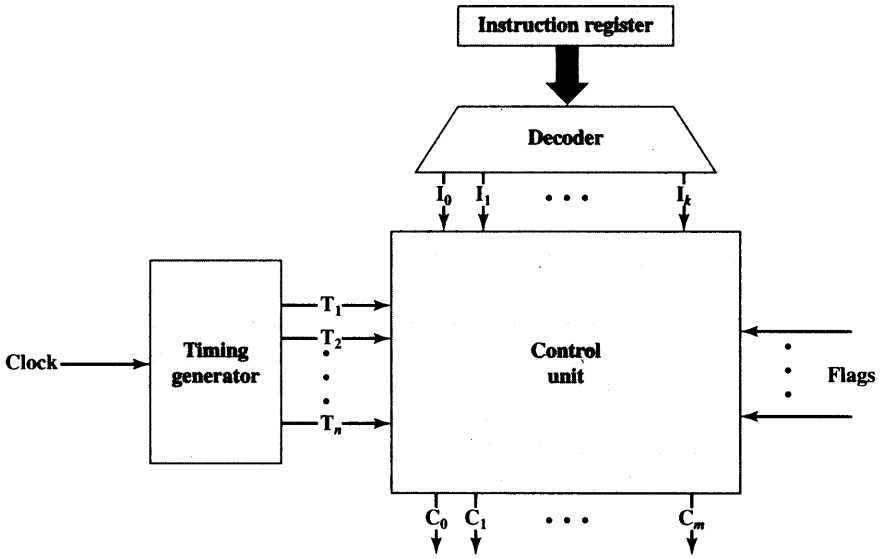


Figure 16.10 Control Unit with Decoded Inputs

With these two refinements, the control unit can be depicted as in Figure 16.10.

Control Unit Logic

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

Essentially, what must be done is, for each control signal, to derive a Boolean expression of that signal as a function of the inputs. This is best explained by example. Let us consider again our simple example illustrated in Figure 16.5. We saw in Table 16.1 the micro-operation sequences and control signals needed to control three of the four phases of the instruction cycle.

Let us consider a single control signal, C_5. This signal causes data to be read from the external data bus into the MBR. We can see that it is used twice in Table 16.1. Let us define two new control signals, P and Q, that have the following interpretation:

PQ = 00 Fetch Cycle
PQ = 01 Indirect Cycle
PQ = 10 Execute Cycle
PQ = 11 Interrupt Cycle

Then the following Boolean expression defines C_5:

$$C_5 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2$$

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

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

$$C_5 = \overline{P} \cdot \overline{Q} \cdot T_2 + \overline{P} \cdot Q \cdot T_2 + P \cdot \overline{Q} \cdot (LDA + ADD + AND) \cdot T_2$$

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

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

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

16.4 RECOMMENDED READING

A number of textbooks treat the basic principles of control unit function; two particularly clear treatments are in [FARH04] and [MANO04].

FARH04 Farhat, H. *Digital Design and Computer Organization.* Boca Raton, FL: CRC Press, 2004.

MANO04 Mano, M. *Logic and Computer Design Fundamentals.* Upper Saddle River, NJ: Prentice Hall, 2004.

16.5 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

control bus	control signal	hardwired implementation
control path	control unit	microoperations

Review Questions

16.1 Explain the distinction between the written sequence and the time sequence of an instruction.

16.2 What is the relationship between instructions and micro-operations?

16.3 What is the overall function of a processor's control unit?

16.4 Outline a three-step process that leads to a characterization of the control unit.

16.5 What basic tasks does a control unit perform?

16.6 Provide a typical list of the inputs and outputs of a control unit.

16.7 List three types of control signals.

16.8 Briefly explain what is meant by a hardwired implementation of a control unit.

Problems

16.1 Your ALU can add its two input registers, and it can logically complement the bits of either input register, but it cannot subtract. Numbers are to be stored in two's complement representation. List the micro-operations your control unit must perform to cause a subtraction.

16.2 Show the micro-operations and control signals in the same fashion as Table 16.1 for the processor in Figure 16.5 for the following instructions:

- Load Accumulator
- Store Accumulator
- Add to Accumulator
- AND to Accumulator
- Jump
- Jump if AC = 0
- Complement Accumulator

16.3 Assume that propagation delay along the bus and through the ALU of Figure 16.6 are 20 and 100 ns, respectively. The time required for a register to copy data from the bus is 10 ns. What is the time that must be allowed for

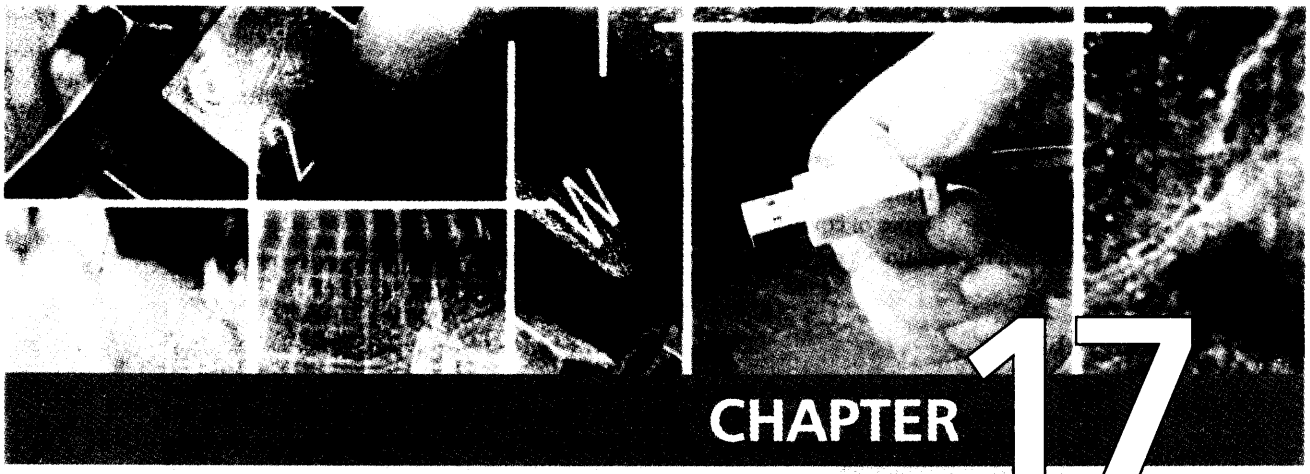
a. transferring data from one register to another?
b. incrementing the program counter?

16.4 Write the sequence of micro-operations required for the bus structure of Figure 16.6 to add a number to the AC when the number is

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

16.5 A stack is implemented as shown in Figure 10.14. Show the sequence of micro-operations for

a. popping
b. pushing the stack



CHAPTER 17

MICROPROGRAMMED CONTROL

KEY POINTS

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

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

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

This situation changed dramatically within a very few months. IBM's System/360 was announced in April, and all but the largest models were microprogrammed. Although the 360 series predated the availability of semiconducter ROM, the advantages of microprogramming were compelling enough for IBM to make this move. Microprogramming became a popular technique for implementing the control unit of CISC processors. In recent years, microprogramming has become less used but remains a tool available to computer designers. For example, as we have seen, on the Pentium 4, machine instructions are converted into a RISC-like format most of which are executed without the use of microprogramming. However, some of the instructions are executed using microprogramming.

17.1 BASIC CONCEPTS

Microinstructions

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

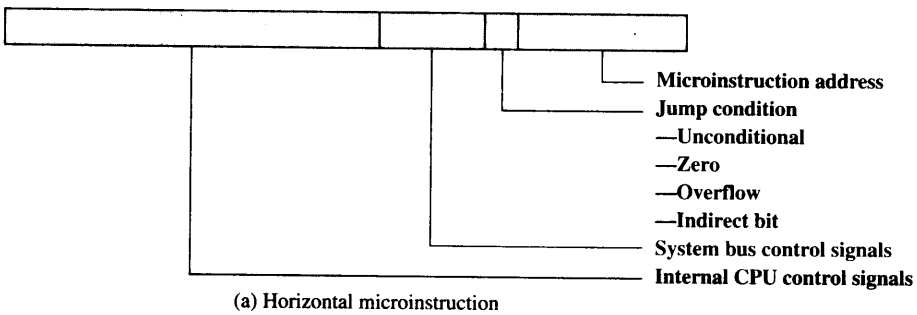
An alternative, which has been used in many CISC processors, is to implement a microprogrammed control unit.

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

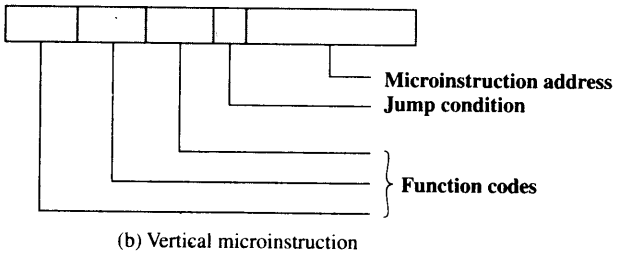
How can we use the concept of microprogramming to implement a control unit? Consider that for each micro-operation, all that the control unit is allowed to do is generate a set of control signals. Thus, for any micro-operation, each control line emanating from the control unit is either on or off. This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

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

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



(a) Horizontal microinstruction



(b) Vertical microinstruction

Figure 17.1 Typical Microinstruction Formats

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

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

Figure 17.2 shows how these control words or microinstructions could be arranged in a **control memory**. The microinstructions in each routine are to be executed sequentially. Each routine ends with a branch or jump instruction indicating

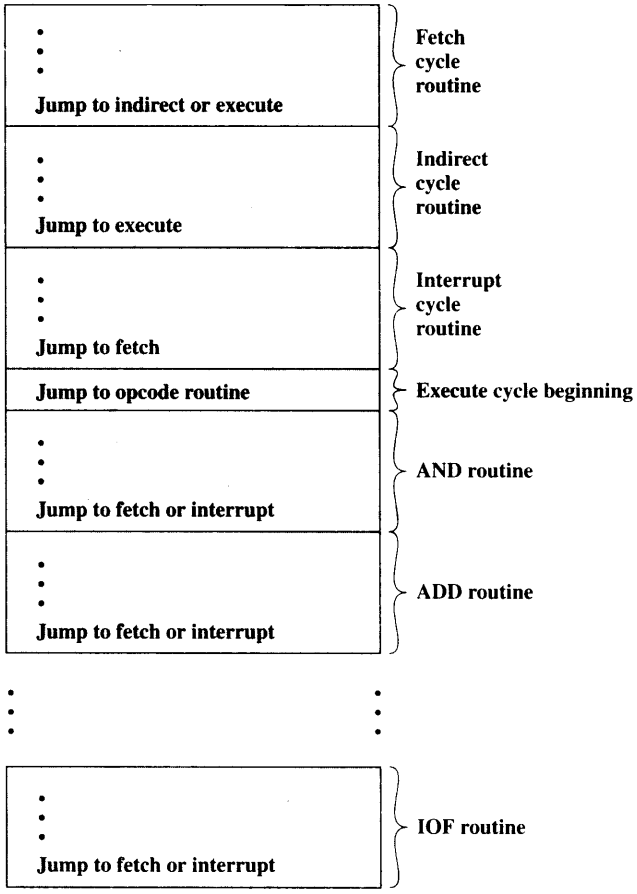


Figure 17.2 Organization of Control Memory

where to go next. There is a special execute cycle routine whose only purpose is to signify that one of the machine instruction routines (AND, ADD, and so on) is to be executed next, depending on the current opcode.

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

Microprogrammed Control Unit

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

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

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

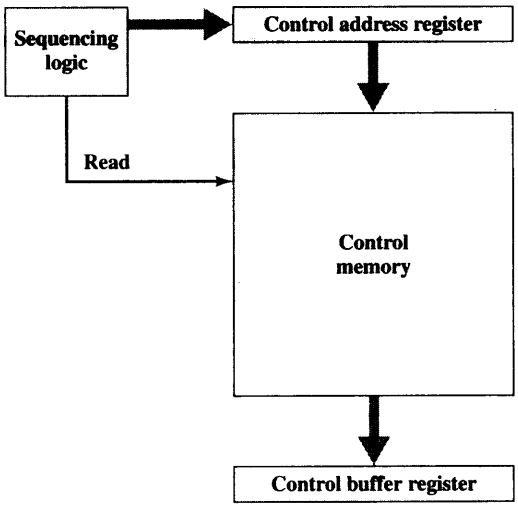


Figure 17.3 Control Unit Microarchitecture