CENTRAL PROCESSING UNIT DESIGNS

The CPU is the key component of a digital computer. Its purpose is to decode instructions received from memory and perform transfer, arithmetic, logic, and control operations with data stored in internal registers, memory, or I/O interface units. Externally, the CPU provides one or more buses for transferring instructions, data, and control information to and from components connected to it. In the generic computer at the beginning of chapter 1, the CPU is a part of the processor and is heavily shaded. CPUs, however, may also appear in computers. Small, relatively simple computers, called microcontrollers, are used in machines and in other digital systems to perform limited or specialized tasks. For example, a microcontroller is present in the keyboard and in the monitor in the generic computer; thus these components are also shaded. In such microcontrollers, the CPU may be quite different from those discussed in this chapter. The word lengths may be short (say, four or eight bits), the number of registers small, and the instruction sets limited. Performance, relatively speaking, is poor but adequate for the task. Most important, the cost of these microcontrollers is very low, making their use cost effective.

In the following pages, we consider two computer CPUs, one for a complex instruction set computer (CISC) and the other for a reduced instruction set computer (RISC). After a detailed examination of the designs, we compare the performance of the two CPUs and present a brief overview of some of the methods used to enhance that performance. Finally, we relate the design ideas discussed to general digital system design.

10-1 TWO CPU DESIGNS

As mentioned in previous chapters, a typical CPU is usually divided into two parts: the datapath and the control unit. The datapath consists of a function unit, registers,

and internal buses that provide pathways for the transfer of information between the registers, the function unit, and other computer components. The datapath may or may not be pipelined. The control unit consists of a program counter, an instruction register, and control logic, and may be either hardwired or microprogrammed. If the datapath is pipelined, the control unit may also be a pipeline. The computer of which the CPU is a part is either a CISC or a RISC, with its own instruction set architecture.

The purpose of this chapter is to present two CPU designs that illustrate combinations of architectural characteristics of the instruction set, the datapath, and the control unit. The designs will be top down, but with reuse of prior component designs, illustrating the influence of the instruction set architecture on the datapath and control units, and the influence of the datapath on the control unit. The material makes extensive use of tables and diagrams. Although we reuse and modify component designs from Chapters 7, 8, and 9, background information from these chapters is not repeated here. References, however, are given to earlier sections of the book, where detailed information can be found.

The two CPUs presented are for a CISC using a non-pipelined datapath with a microprogrammed control unit and a RISC using a pipelined datapath with a hardwired pipelined control unit. These represent two quite distinct combinations of instruction set architecture, datapath, and control unit.

10-2 The Complex Instruction Set Computer

The first design we present is for a complex instruction set computer with a non-pipelined datapath and microprogrammed control unit. We begin by describing the instruction set architecture, including the CPU register set, instruction formats, and addressing modes. The CISC nature of the instruction set architecture is demonstrated by its memory-to-memory access for data manipulation instructions, eight addressing modes, two instruction format lengths, and instructions that require significant sequences of operations for their execution.

We design a datapath for implementing the CISC architecture. This datapath is based on the one initially described in Section 7-9 and incorporated into a CPU in Section 8-10. Modifications are made to the register file, the function unit, and the buses to support the present instruction set architecture.

Once the datapath has been specified, a control unit is designed to complete the implementation of the instruction set architecture. The design of the control unit must involve a coordinated definition of both the hardware organization and the microprogram organization. In particular, dividing the microprogram into microroutines, while at the same time designing the sequencer with which they interact, is a key part of the design. Even the instruction fields and opcodes are tied to this coordinated effort. Following the definition of the hardware and microcode organizations, we detail essential parts of the microcode and the microroutines for representative operations.

Instruction Set Architecture

Figure 10-1 shows the CISC register set accessible to the programmer. All registers have 16 bits. The register file has eight registers, *R*0 through *R*7. *R*0 is a special reg-

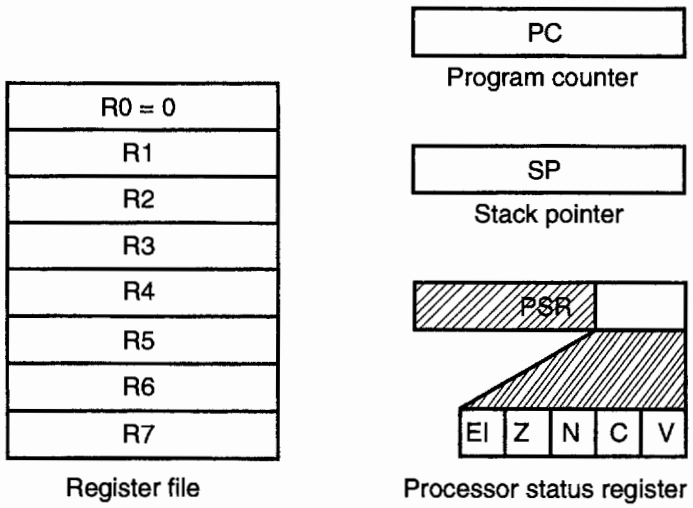


FIGURE 10-1
CPU Register Set Diagram for CISC

ister that always supplies the value zero when it is used as a source and discards the result when it is used as a destination.

In addition to the register file, there is a program counter *PC* and stack pointer *SP*. The presence of a stack pointer indicates that a memory stack is a part of the architecture. The final register is the processor status register *PSR*, which contains information only in its rightmost five bits; the remainder of the register is assumed to contain zero. The *PSR* contains the four stored status bit values *Z*, *N*, *C*, and *V* in positions 3 through 0, respectively. In addition, a stored interrupt enable bit *EI* appears in position 4.

Table 10-1 contains the 42 operations performed by the instructions. Each operation has a mnemonic and a carefully selected opcode. The operations are divided into four groups based on the number of explicit operands and whether the operation is a branch. In addition, the status bits affected by the operation are listed.

Figure 10-2 gives the instruction formats for the CPU. The generic instruction format has five fields. The first, OPCODE, specifies the operation. The next two, MODE and S, are used to determine the addresses of the operands. The last two fields, SRC and DST, are the 3-bit source register and destination register address fields, respectively. In addition, there is an optional second word W that appears with some instructions as an operand or an address, but not with others.

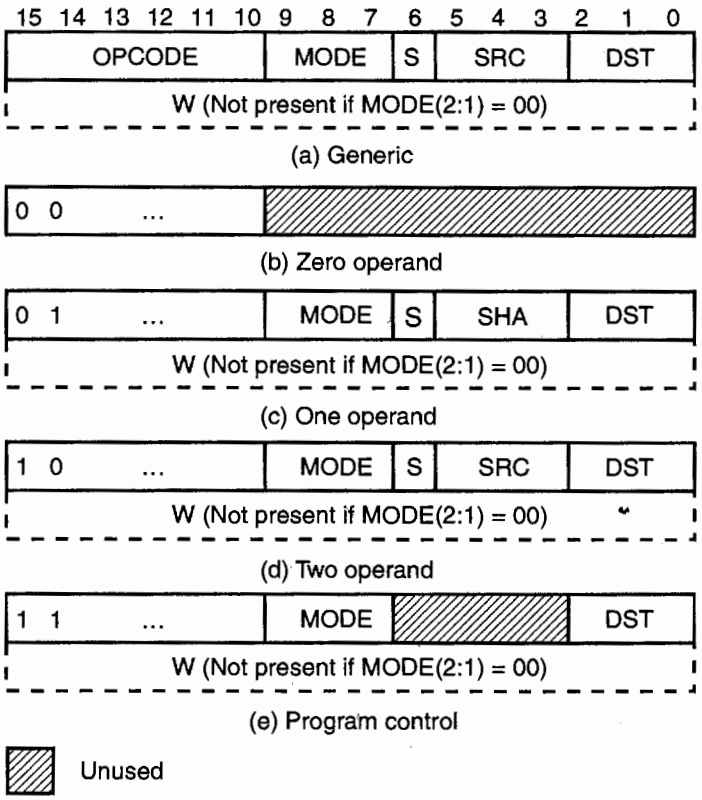
The first two bits of OPCODE, *IR*(15:14), determine the number of explicit operands and how the fields of the format are used. When these bits are 00, either no operand is required or the location of the operand is implied by OPCODE. Only the OPCODE field is needed, as shown in Figure 10-2(b). The four rightmost OPCODE bits can specify up to 16 operations without operands or with implied operand addresses.

If *IR*(15:14) is 01, the instruction has one operand and is a data transfer or data manipulation instruction. Since there is an operand, the MODE field specifies

□ **TABLE 10-1**
CISC Instruction Operations

Instruction	Mnemonic	Opcode	Status Effect
Zero-operand Instructions			
No operation	NOP	000000	None
Push registers	PSHR	000001	None
Pop registers	POPR	000010	None
Move string	MVS	000011	None
Return from procedure	RET	000100	None
Return from interrupt	RTI	000101	From stack
Invalid	000110 through 001111		
One-operand Instructions			
Push	PUSH	010000	None
Pop	POP	010001	None
Increment	INC	010010	ZCNV
Decrement	DEC	010011	ZCNV
Negate	NEG	010100	ZCNV
Complement	COM	010101	ZN
Logical shift right	SHR	011000	ZC
Logical shift left	SHL	011001	ZC
Arithmetic shift right	SHRA	011010	ZCNV
Arithmetic shift left	SHLA	011011	ZCNV
Rotate right	ROR	011100	ZC
Rotate left	ROL	011101	ZC
Rotate right with carry	RORC	011110	ZC
Rotate left with carry	ROLC	011111	ZC
Invalid	010110 through 010111		

Instruction	Mnemonic	Opcode	Status Effect
Two-operand Instructions			
Move	MOVE	100000	None
Exchange	XCH	100001	None
Add	ADD	100010	ZCNV
Add with carry	ADDC	100011	ZCNV
Subtract	SUB	100100	ZCNV
Subtract with borrow	SUBB	100101	ZCNV
Multiply	MUL	100110	ZCNV
Divide	DIV	100111	ZCNV
Compare	CMP	101000	ZCNV
AND	AND	101001	ZN
OR	OR	101010	ZN
Exclusive-OR	XOR	101011	ZN
Invalid	101100 through 101111		
Branch Instructions			
Jump	JMP	110000	None
Call procedure	CALL	110001	None
Branch if zero	BZ	111000	None
Branch if no zero	BNZ	111001	None
Branch if carry	BC	111010	None
Branch if no carry	BNC	111011	None
Branch if negative	BN	111100	None
Branch if no negative	BNN	111101	None
Branch if overflow	BV	111110	None
Branch if no overflow	BNV	111111	None
Invalid	110010 through 110111		



□ FIGURE 10-2
Instruction Formats

the addressing mode for obtaining it. The single address may involve the DST register address in its formation, so the DST field is also present. The S field and SRC field relate to the presence of two operands and so are not used for the typical single operand instructions. But, the shift instructions require a shift amount to indicate how many bits to shift. For maximum flexibility, this shift amount is treated just like a source operand. As a consequence, the SRC field for shifts becomes the SHA field. The shift amount determined using the SHA and S fields is a full 16-bit operand, but only values 0 through 15 are meaningful. There are sufficient OPCODE bits for 16 instructions with a single operand.

Table 10-2 gives the addressing modes specified by the MODE field. The first two bits of MODE specify four different types of addressing: register, immediate, indexed, and relative to the *PC*. The third bit of MODE specifies whether the address generated by these modes is used as an indirect address. The one exception to this is direct addressing, which is obtained by applying indirection to the immediate type. Otherwise, if the third bit equals 0, indirect addressing does not apply whereas, if it equals 1, indirect addressing does apply. For the register type of instruction, MODE(2:1) = 00 and the W word is not needed, since the operand or address comes from a register. For all other modes, the W word appears as the second word of the instruction. The third column of the table provides register transfer statements for each of the addressing modes for the one–operand instructions.

□ **TABLE 10-2**
Addressing Modes

		Register Transfer Description of Operands		
MODE	**Address Mode**	**IR(15:14) = 01 or 11**	**IR(15:14) = 10, S = 0**	**IR(15:14) = 10, S = 1**
000	Register	$R[DST]$	$R[SRC], R[DST]$	$R[DST], R[SRC]$
001	Register Indirect	$M[R[DST]]$	$M[R[SRC]], R[DST]$	$M[R[DST]], R[SRC]$
010	Immediate	W	$W, R[DST]$	$W, R[SRC]$
011	Direct	$M[W]$	$M[W], R[DST]$	$M[W], R[SRC]$
100	Indexed	$M[R[DST] + W]$	$M[R[SRC] + W], R[DST]$	$M[R[DST] + W], R[SRC]$
101	Indexed Indirect	$M[M[R[DST] + W]]$	$M[M[R[SRC] + W]], R[DST]$	$M[M[R[DST] + W]], R[SRC]$
110	Relative	$M[PC + W]$	$M[PC + W], R[DST]$	$M[PC + W], R[SRC]$
111	Relative Indirect	$M[M[PC + W]]$	$M[M[PC + W]], R[DST]$	$M[M[PC + W]], R[SRC]$

If $IR(15:14)$ is equal to 10, then the instruction has two addresses used for true operands. All fields of the generic instruction, including S and SRC, are used for this case for all instructions. One of the addresses, either the source or the destination, uses the addressing modes. If S = 0, then the source uses the addressing mode specified by MODE, and the destination is a register. If S = 1, then the destination uses the addressing mode, and the source is a register. Register transfer descriptions of the resulting addresses are given in the fourth and fifth columns of Table 10-2. Again, depending on the contents of the MODE field, the second instruction word W, which is an address or an immediate operand, may or may not be present.

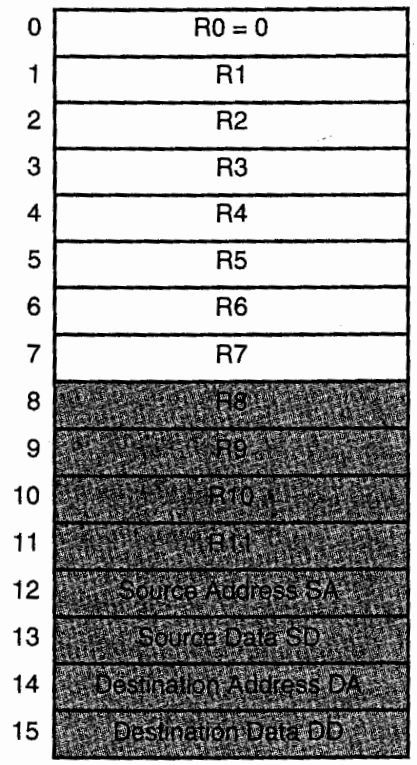
Instructions with $IR(15:14) = 11$ are branches. Aside from the S field and the SHA field for shifts, the format is the same as for $IR(15:14) = 01$. For all instructions of this type, the destination address (not the operand) becomes the new address placed in the program counter *PC*. As a consequence, the register mode is invalid for branch instructions.

Before proceeding to the next step, which defines the datapath to support the instruction set architecture, we will briefly note the characteristics of the architecture that define it as CISC or RISC. Most of the operations given in Chapter 9 are included in the instruction set. A number of operations that do not appear are redundant. The same actions can be achieved by using proper addressing modes with instructions that do appear. For example, LD, ST, IN, and OUT can all be achieved by using the MOVE instruction in a memory-mapped structure. By looking at the formats for the instructions, we find that most of the instructions can operate directly on operands from memory. There are eight addressing modes and two different lengths of instruction formats. In addition, some of the instructions perform complex operations which can be viewed as operations that are likely to take more than one clock cycle for the execution step. These characteristics clearly identify this as a CISC architecture.

Datapath Organization

Rather than beginning from scratch, we will reuse the non-pipelined datapath employed with the microprogrammed control in Section 8-10, with modifications. That datapath was shown in Figure 8-26, and the new, modified datapath based on it is given in Figure 10-6. We treat each modification in turn, beginning with the register file.

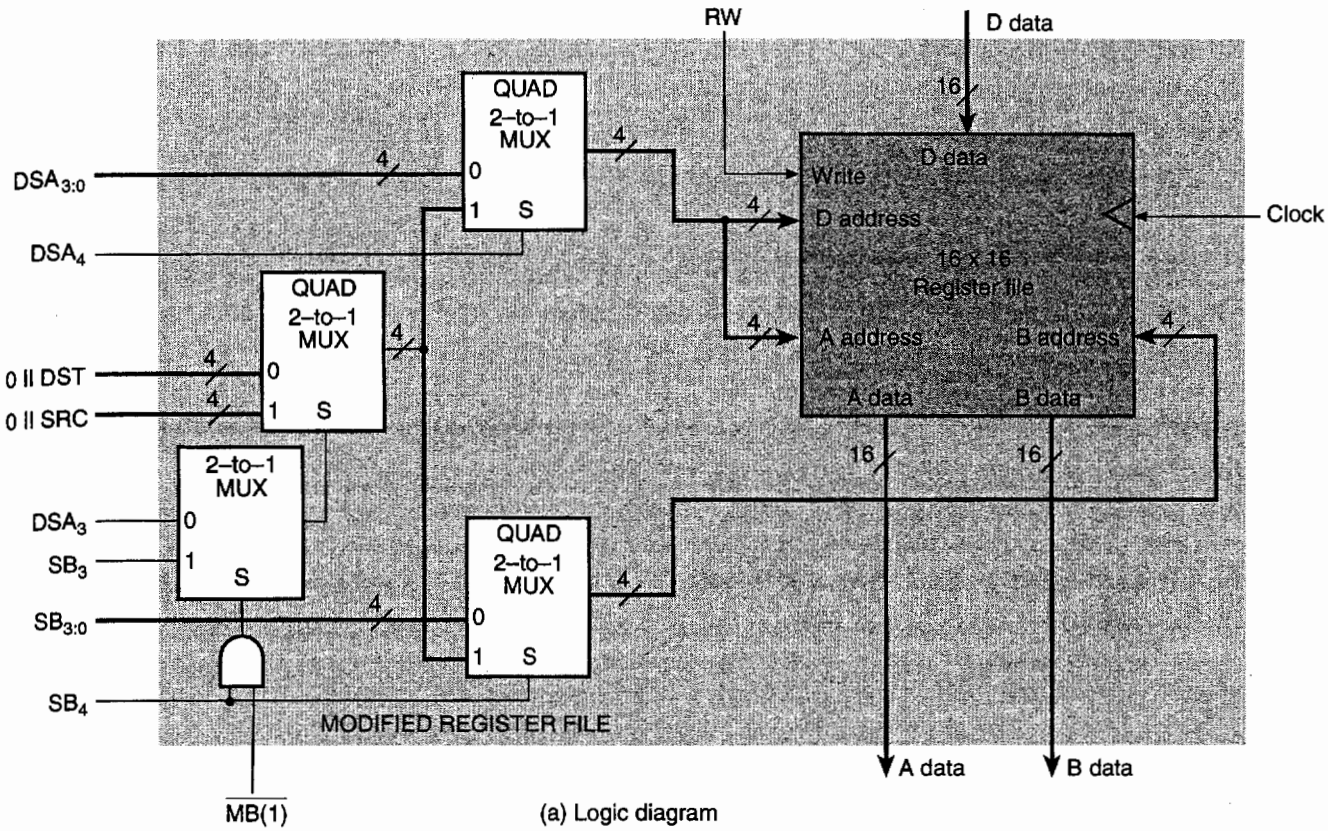
In Figure 8-26, register *R*8 was used as a temporary storage location. In the new microprogrammed architecture, there are complex instructions spanning many clock cycles and performing complicated operations. Thus, more temporary storage is needed for use by the microprograms. To meet this need, we expand the register file from 9 registers to 16. The first 8 registers, *R*0 through *R*7, are visible to the computer programmer. The second eight registers, *R*8 through *R*15, are used as temporary storage for microprogram operands and are hidden from the programmer. Figure 10-3 provides a map of the expanded register file with the temporary registers shaded. As indicated previously, register *R*0 supplies the constant 0, registers *R*1 through *R*7 are available to the programmer for use, and registers *R*8 through *R*15 provide general temporary storage for use by microprograms. The last



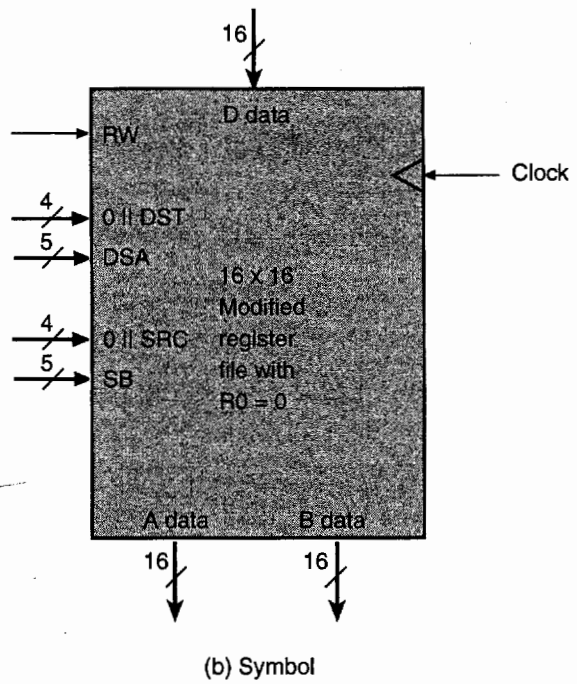
□ **FIGURE 10-3**
Register File Map

four registers, *R*12 through *R*15, have special uses: To keep the microcode simple, standard locations are essential for storing the operands and addresses used by execution microcode for most instructions. Thus, *R*12 is the location for the source address (*SA*), *R*13 for the source data (*SD*), *R*14 for the destination address (*DA*), and *R*15 for the destination data (*DD*).

We cannot access the eight temporary registers based on the 3-bit register addresses available in the instruction. To deal with this problem, we provide, first, 4-bit register addresses from the microinstruction, and second, a microinstruction bit to choose between these addresses and those from the instruction. In addition, the flexibility to allow the register addressed by DST to be a source and by SRC to be a destination is needed to permit results of operations to be placed directly in memory. To accomplish these goals, we modify the register file by adding the logic shown in Figure 10-4(a). The instruction set architecture uses two addresses, one for a source operand and the other for the other source as well as the destination. The register file uses the *B* address for a source, and the *A* and *D* addresses on the file are connected together, giving the same address for the other source and the destination. Although this reduction from three to two addresses is not essential at the microinstruction level, it decreases the number of bits needed for register addresses in the microinstruction and matches the use of the register fields in the instruction formats.



(a) Logic diagram



(b) Symbol

FIGURE 10-4
Modifications to Register File

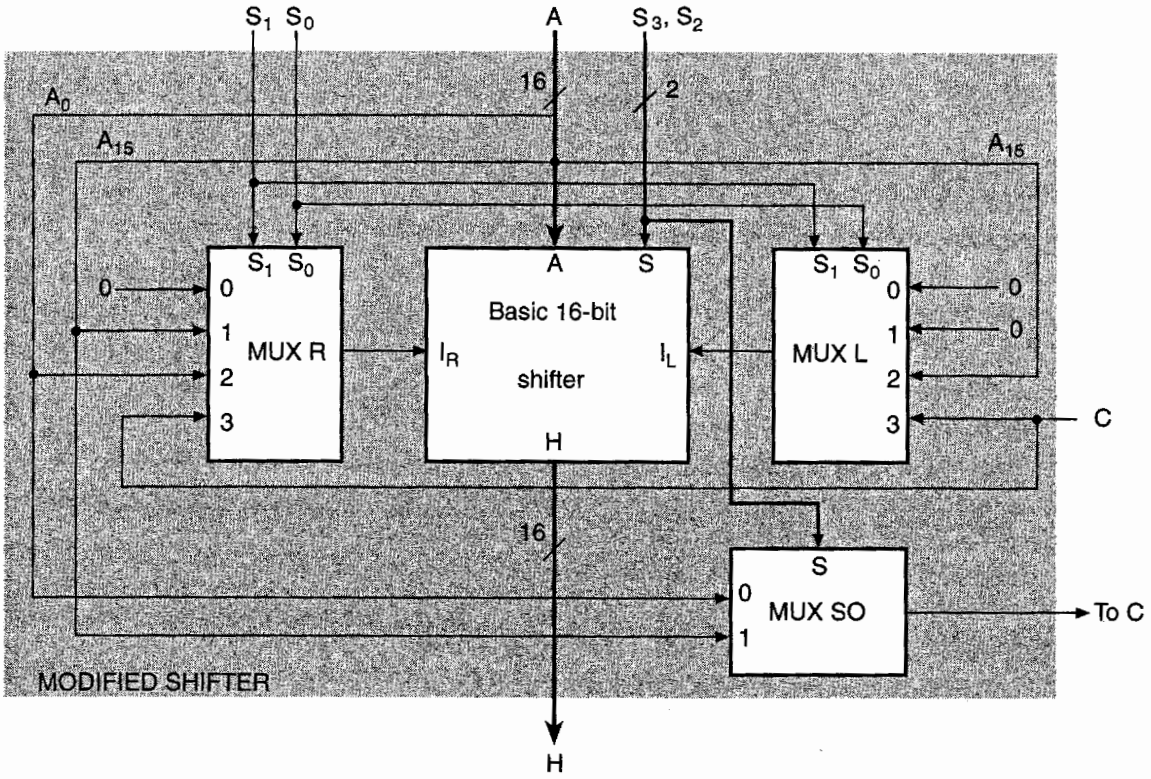
A quad 2–to–1 multiplexer is attached to each of the two address inputs to the register file, to select between an address from the microinstruction and an address from the instruction. There is a 5-bit field in the microinstruction for the combined destination and source address DSA, in addition to a 5-bit field for the *B* address SB. The first bit of each of these fields selects between the register file address in the microinstruction (0) and the register file address in the instruction (1). If an instruction address is selected, whether it is DST or SRC is determined by an additional quad 2–to–1 multiplexer. This multiplexer is controlled by the second bit of the DSA or SB field, depending on which of them has 1 as the first bit. Only one of the two fields DSA and SB is allowed to have a 1 in the first bit in any microinstruction, thereby ensuring that the proper second bit is used to determine the register address. A 0 is appended to the left of the 3-bit fields DST and SRC to cause them to address *R*0 through *R*7. In addition to the first bit, which selects the address source, the addresses from the microinstruction contain four bits so that all 16 registers can be reached. The final change to the register file is to replace the storage elements for *R*0 in the file with open circuits on the lines that were their inputs and with constant zero values on the lines that were their outputs. A symbol for the resulting register file is shown in Figure 10-4(b).

We find that, based on the eight shift instructions provided, the shifter from Figure 7-16 needs to be modified. The modifications involve the end bits of the shift logic. For logical shifts, a 0 is inserted, as before. For the right arithmetic shift, the sign bit is the incoming bit, and for the left arithmetic shift, 0 is the incoming bit. Rotates require that the bit from the opposite end of the shifter be fed around. Finally, rotates with carry require that the carry flip-flop output be provided as an input on both ends of the shifter.

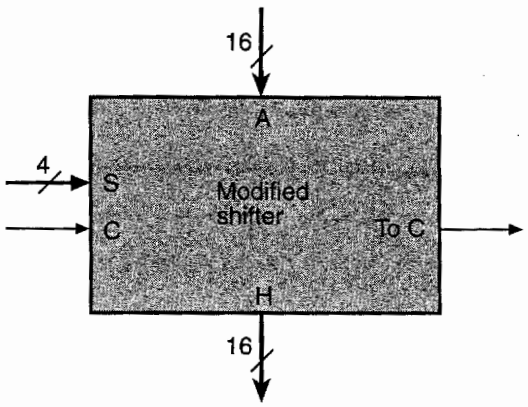
The inputs are furnished by two 4–to–1 multiplexers, MUX *R* and MUX *L*, added to a basic 16-bit shifter, all shown in Figure 10-5(a). Also, the appropriate end bits from the input operand must be sent to the carry flip-flop. A 2–to–1 multiplexer MUX *SO* selects the end bit to pass to the carry flip-flop *C*. The symbol for the new shifter, which replaces the basic shifter from Figure 7-16, appears in Figure 10-5 (b). FS_3, FS_2, FS_1, and FS_0 from the FS field drive the control inputs S_3, S_2, S_1 and S_0, respectively.

The instruction set requires the ability to store the contents of the *PC* in memory and load it back into the *PC*. Likewise, the contents of the *PSR* must be stored to memory and loaded back. Also, it is necessary to be able to store and load the *SP*. So, for these three registers, paths are established to and from the datapath buses. The paths into the datapath for the *PC* and *SP* are provided by inserting a 4–to–1 multiplexer MUX *A* into Bus *A*, as shown in Figure 10-6. The path from *PSR* into the datapath is handled by changing MUX *B* to a 3–to–1 multiplexer. The *PSR* is placed on MUX *B*'s second input, and a zero-filled 5-bit constant from the microprogrammed control is placed on its third input. Paths are also added from Bus *D* to the *PSR*, *PC*, and *SP*.

The final area for modification of the datapath is the status bit hardware that lies at the boundary between the datapath and the control unit. Added there are five flip-flops storing program status bits *EI*, *Z*, *N*, *C*, and *V*, which did not appear in prior



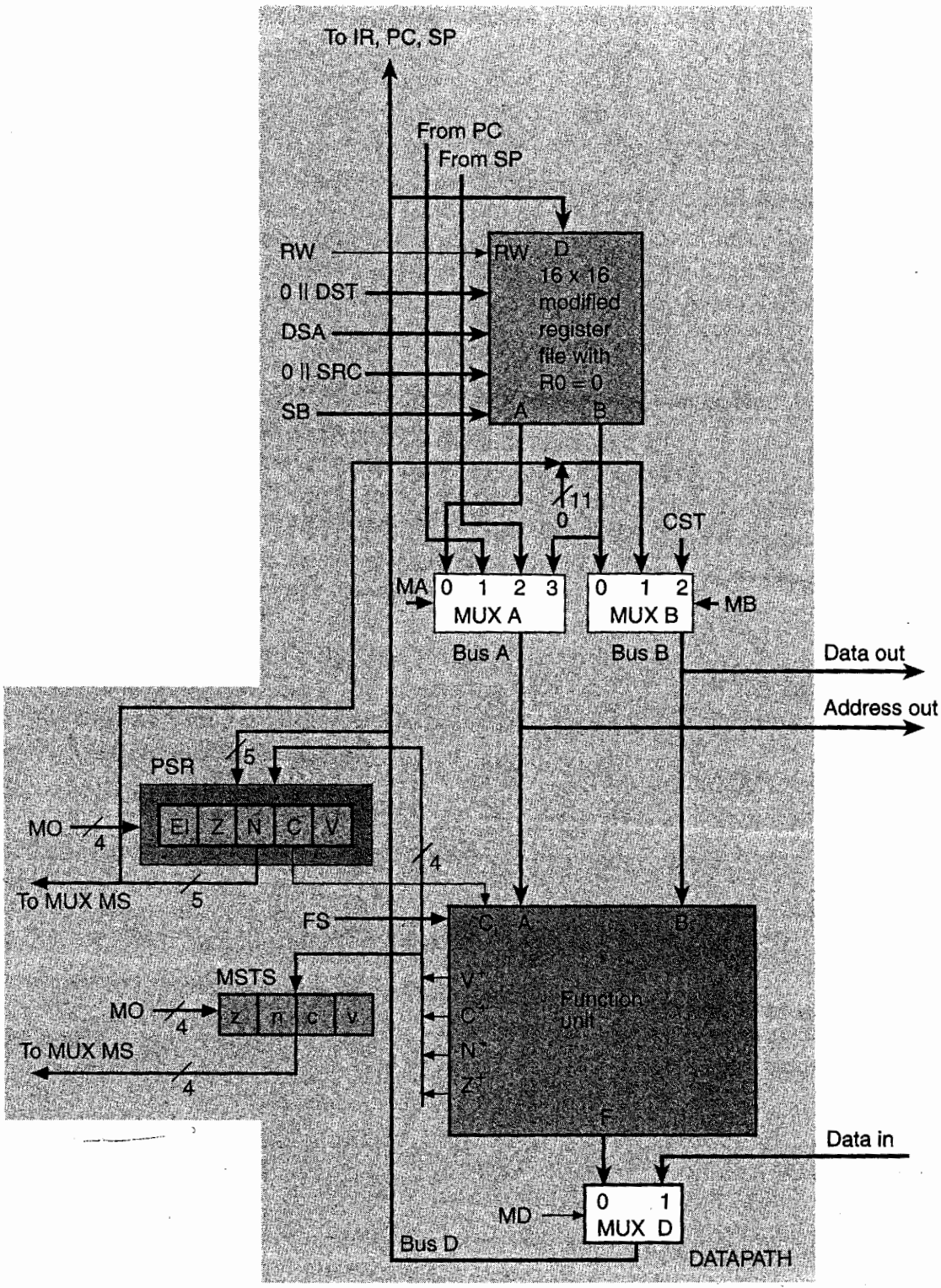
(a) Logic Diagram



(b) Symbol

FIGURE 10-5
16-Bit Shifter and Symbol

datapath designs and which make up the *PSR*. The values of these status bits are used to make decisions at the program level, as illustrated in Section 9-8. To distinguish the status values coming from the function unit from these stored values, the function unit outputs are labeled with a superscript plus sign (+). The logic to selectively control the loading of the status bits in accordance with Table 10-1 is repre-



□ FIGURE 10-6
CISC Datapath

sented by the box surrounding the bits. Also, we find it necessary to make stored status values available for use by the microprogram routines without disturbing the stored *PSR* values for the program level. Thus, a second flip-flop register is provided for temporary storage of the values Z^+, N^+, C^+, and V^+. The bits of this register are labeled *z*, *n*, *c*, and *v*, and they constitute the *microstatus register MSTS*. These additions, plus attachments to the buses, are shown in Figure 10-6. The four control bits labeled MO (miscellaneous operations) are decoded to control the sets of status bits to be enabled for loading. $\mathbf{C}_i$ provides the stored carry bit *C* to the ALU input C_{in} and the Shifter input *C* within the Function unit. C_{in} on the ALU is driven by the least significant bit of FS unless MO = 1011, in which case, it is driven by stored carry bit *C*.

All modifications to the original datapath are represented in Figure 10-6. As a part of the design process, the new datapath needs to be checked to make sure that it has all of the capabilities necessary for implementing the instruction set and addressing modes. Certainly, some decisions have been made that have not been discussed. For example, there is no dedicated multiplication or division hardware, so these operations must be implemented by microprograms controlling the datapath.

Microprogrammed Control Organization

The microprogrammed control unit accompanies the datapath of Figure 10-6 in Figure 10-7. The control consists of four principal parts. One is the control ROM, which has 256 words of 31 bits each. There are three control unit registers: the instruction register *IR*, the program counter *PC*, and the stack pointer *SP*. In some designs the *PC* and *SP* are logically included in the register file and thus are a part of the datapath. Here, since they are separate from the register file and are used primarily for program control, we have included them with the control. Sequencing within the control unit is provided by the microsequencer, which contains two registers: the *control address register CAR* and the *subroutine branch register SBR*. The program counter for the microprogram, the *CAR* simply counts up to the next address in sequence or loads in parallel. With a parallel load, the address can be set to any value and the next-address comes from three sources including the next-address field in the current microinstruction.

Microroutines have subroutines, just as programs do. To distinguish them, we call subroutines for microprograms *microsubroutines*. The *SBR* is used to store the next address for the *CAR* at the time a microsubroutine is entered. This return address is then retrieved at the end of the microsubroutine in order to return microprogram execution to the next microinstruction in the calling microroutine. The final part of the control unit is the instruction decoder, which consists of combinational logic and is also a next address source for the CAR.

The microinstructions stored in the control ROM use the two different formats shown in Figure 10-7 and detailed in Figure 10-8. The first microinstruction field MC selects the format used. If MC is 00, 01, or 10, format A applies. The microinstructions in this format perform data transfers and manipulation and decode instructions. They also handle returns from a microsubroutine. If MC is 11,

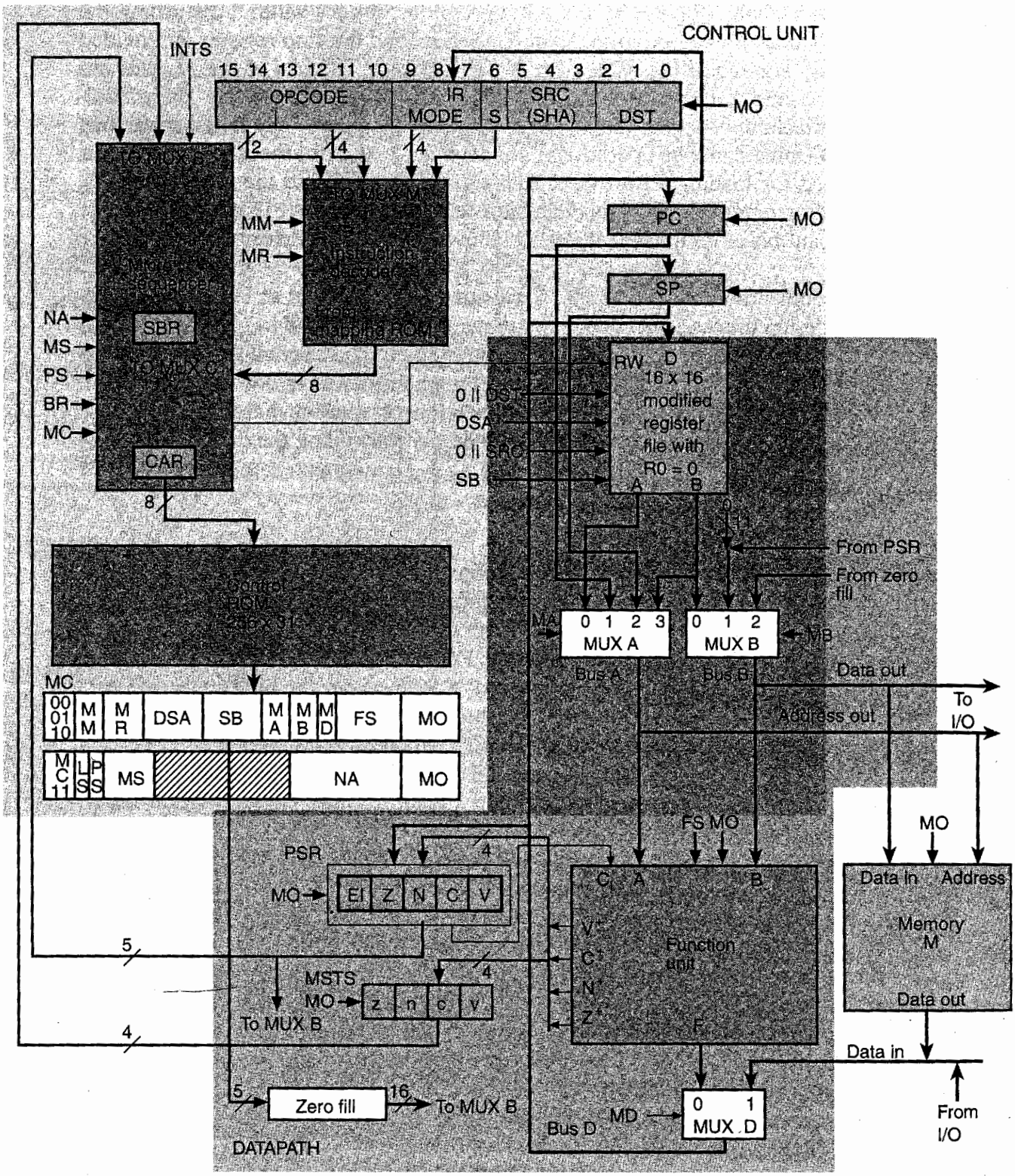
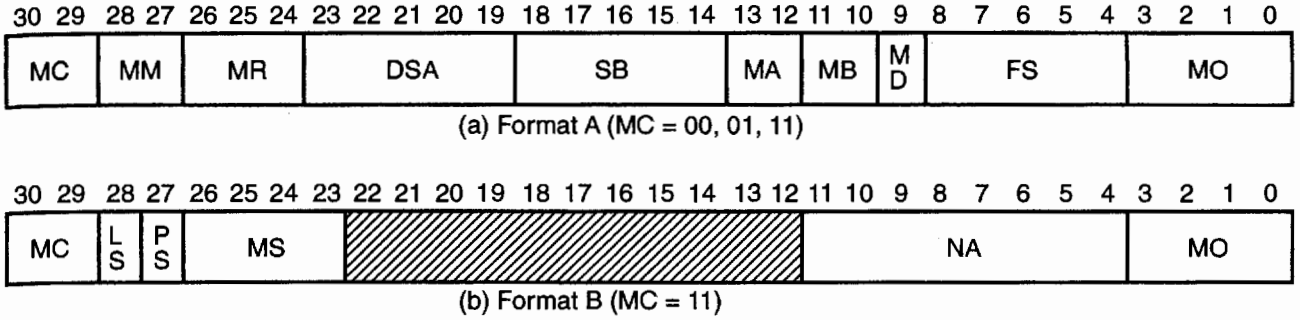


FIGURE 10-7
CISC CPU



□ FIGURE 10-8
Microinstruction Formats

format B applies. Microinstructions in this format change the flow of the microprogram by implementing branches and a microsubroutine call.

In format A, the fields in bits 23 through 4, DSA through FS, control the datapath. The codes for these fields appear in Table 10-3. The actions of the DSA and SB fields have already been described in conjunction with the modified register file. The MA and MB fields control MUX *A* and MUX *B*, respectively, whose selections were described in the discussion of the datapath. Here notation is added: MCST for a zero-filled constant coming from the SB field of a microinstruction. The MD field controls MUX *D*, as in previous designs. The two codes for shifts originally in the FS field have been replaced by the eight codes now required for the modified shifter. Shift operation notations beginning with "l" are logical shifts, with "a" are arithmetic shifts, and with "r" are rotates. An added "c" at the end of the shift notation indicates that the carry is included in the rotate.

All of the remaining fields have some relationship to the operation of the control unit. In order to discuss these fields, we need to examine the parts of the control unit. The heart of the control in Figure 10-7 is the microsequencer, which includes the *SBR*, the *CAR*, and the address determination logic. The microsequencer control fields are given in Table 10-4 and Table 10-5. Initially, we will discuss the microsequencer in terms of the operations performed: we then briefly look at the detailed logic.

Table 10-4 gives key information on the microsequencer operation, since MC specifies the source of the next address to appear in the *CAR*. For code 00, the contents of the *CAR* are incremented to point to the next microinstruction in sequence. For code 01, the next address comes from *SBR*, which holds the return address for a microsubroutine. For code 10, the next address is determined by the instruction decoder. The decoder is capable of executing an unconditional branch or a 4-way, 8-way, or 16-way conditional branch based on the values of the OPCODE or MODE bits. These multiple-way branches, to be detailed later, are faster than using a sequence of 2-way branches. For the first three MC codes, format A is used. Thus, these branches for decoding instructions can be performed simultaneously with data transfer and manipulation.

TABLE 10-3
Control Word Encoding for Microinstruction Format A: Datapath Part

DSA, SB		MA	MB		MD		FS	
$R0 = 0$	00000	Register A	Register B	00	Function	0	$F = A$	00000
$R1$	00001	PC	PSR	01	Data in	1	$F = A + 1$	00001
$R2$	00010	SP	MCST	10			$F = A + B$	00010
$R3$	00011			11			$F = A + B + 1$	00011
$R4$	00100						$F = A + \overline{B}$	00100
$R5$	00101						$F = A + \overline{B} + 1$	00101
$R6$	00110						$F = A - 1$	00110
$R7$	00111						$F = A$	00111
$R8$	01000						$F = A \wedge B$	01000
$R9$	01001						$F = A \vee B$	01010
$R10$	01010						$F = A \oplus B$	01100
$R11$	01011						$F = \overline{A}$	01110
$R12$ (SA)	01100						$F = B$	10000
$R13$ (SD)	01101						$F =$ lsl B	11000
$R14$ (DA)	01110						$F =$ lsr B	10100
$R15$ (DD)	01111						$F =$ asl B	11001
$R[DST]$	10XXX*						$F =$ asr B	10101
$R[SRC]$	11XXX*						$F =$ rol B	11010
							$F =$ ror B	10110
							$F =$ rolc B	11011
							$F =$ rorc B	10111

*Only one of DSA and SB may contain either of these patterns in any microinstruction. The other must contain a pattern beginning with a 0.

TABLE 10-4
Control Word Information for MC and LS

Action	Format	Symbolic Notation	Codes	
			MC	LS
Increment CAR	A	NXT	00	—
Return from subroutine	A	RET	01	—
Map instruction into CAR	A	MAP	10	—
Jump to NA if ST bit is satisfied; else increment CAR	B	BR	11	0
Call subroutine at NA if ST bit is satisfied; else increment CAR	B	CALL	11	1

For MC equal to 11, format B is used, and an unconditional or conditional branch is specified. If the branch is unconditional or if the condition is satisfied, then a jump to the address specified in the 8-bit next-address field NA occurs. If the condition is not satisfied, then the CAR is simply incremented. In addition, if the jump

□ TABLE 10-5
Control Word Information for Polarity Bit and Multiplexer S in Format B Microinstructions

PS			MS		
Action	**Symbolic Notation**	**Code**	**Condition Status Signal**	**Symbolic Notation**	**Code**
Pass status bit unchanged	TS	0	Constant 1 for unconditional transfer	BU	0000
Complement status bit	CS	1	Zero *PSR* bit	BZ	0001
			Negative *PSR* bit	BN	0010
			Carry *PSR* bit	BC	0011
			Overflow *PSR* bit	BV	0100
			Enable-interrupt *PSR* bit	EI	0101
			Zero *MSTS* bit	Bz	0110
			Negative *MSTS* bit	Bn	0111
			Carry *MSTS* bit	Bc	1000
			Overflow *MSTS* bit	Bv	1001
			Interrupt signal *INTS*	BI	1010

occurs and LS equals 1, the incremented version of the *CAR* is stored in *SBR*, the microsubroutine return register. Since there is only a single such register and no way to save its contents elsewhere, only a single level of microsubroutine calls is permitted. Also, for any format B microinstruction, the register file is not to be written.

The remaining fields for the microsequencer appear in Table 10-5. MS is used to select either an unconditional branch or the conditions upon which to branch, just as is done in Chapter 8. Here, however, there are many more conditions on which to branch. For MS equal to 0000, the branch or subroutine call is unconditional. The next five values of MS use the five bits of the *PSR*—*EI*, *Z*, *N*, *C*, and *V*—as conditions. The field PS determines whether the jump occurs when the value of the condition is 1 or 0. If PS is 0, then the jump occurs when the value of the condition is 1; if PS is 1, then the jump occurs when the value of the condition is 0. The next four values of MS cause branches on the microstatus bits z, n, c, and v of *MSTS*. The final value of MS branches on *INTS*, the interrupt status bit.

We now complete our discussion of the microsequencer by examining briefly the implementing hardware in Figure 10-9. The selection of the next microaddress to place in the *CAR* is performed by MUX *C*, which is controlled by MC. The inputs to MC correspond to the desired sources of the next addresses specified in Table 10-4. The selection of branch conditions given in Table 10-5 is accomplished by MUX *S*. Its output enters an exclusive-OR gate that complements the value of the condition based on the value of PS. The resulting exclusive-OR output signal *ST* drives the select input of MUX *R*, which selects between next address NA and the incremented *CAR* value, thereby implementing the conditional branch. To stop the register file from being written for format B (MC = 11) microinstructions, a two-input NAND gate produces *RWE* (register write enable) = 0 for MC = 11. By ANDing *RWE* with the signal otherwise driving *RW* on the register file, we cause

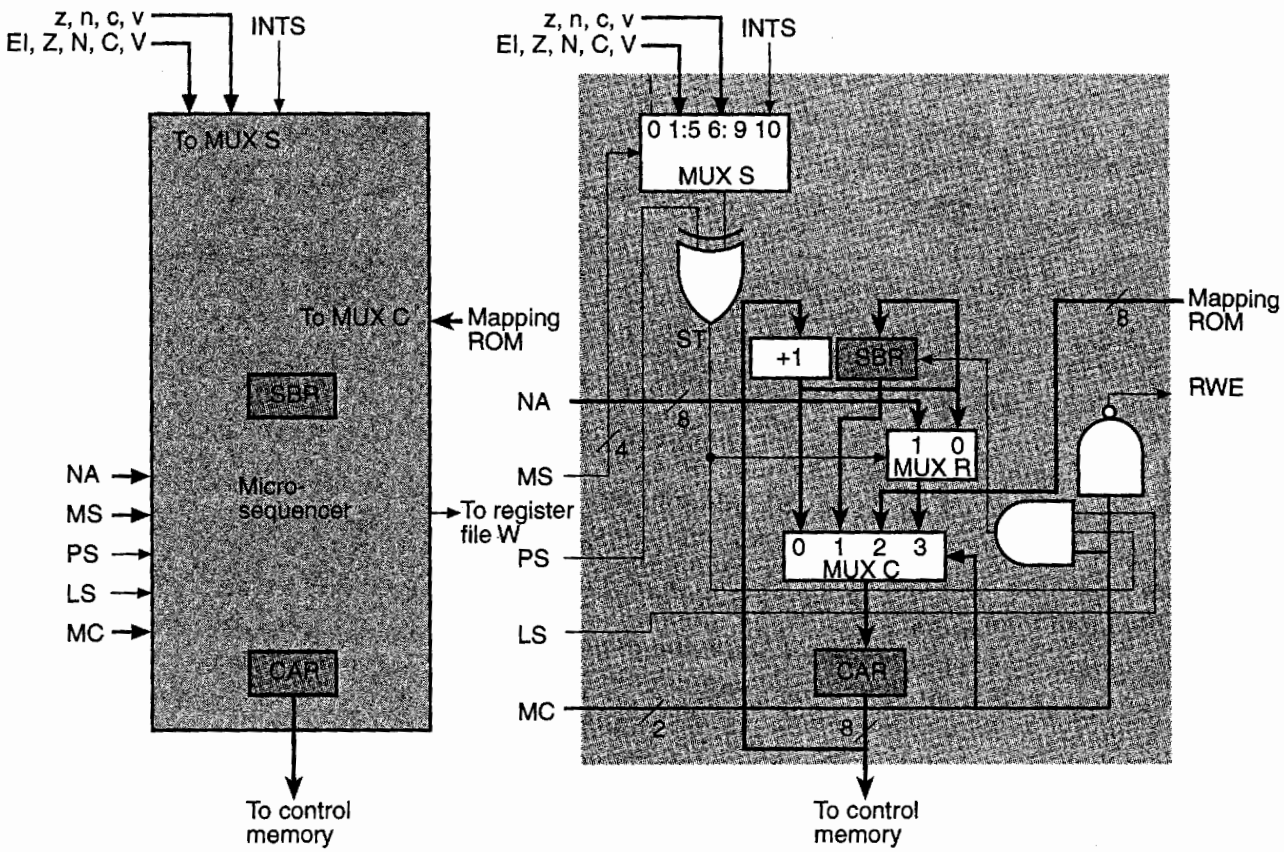


FIGURE 10-9
The Microsequencer

MC = 11 to prevent the writing of the file. The loading of *SBR* for subroutine calls is accomplished by logic consisting of one additional AND gate. The inputs to the AND consist of the two MC bits, LS and ST. The AND output is 1 for MC = 11, LS = 1, and ST = 1. This causes *SBR* to be loaded with the incremented *CAR* value for a microsubroutine call instruction with the branch taken.

The instruction decoder produces control ROM addresses based on its control fields and fields of the instruction in the *IR*. By using control fields, different control ROM addresses can be produced for the same values of the instruction fields. This allows distinct microroutines to be executed in distinct parts of the execution of a given instruction. The control fields for the instruction decoder are given in Table 10-6. MM defines which field of the instruction is involved in determining the address provided. If MM equals 00, then the left two bits of OPCODE are used to determine the address; if MM equals 01, the right four bits of OPCODE are used to determine the address. Since there are two bits of OPCODE that can take on arbitrary values for MM equal to 00, four different addresses can result. Likewise, for MM equal to 01, there are four bits of OPCODE that can assume arbitrary values, so 16 different addresses can result. Thus, using just five

TABLE 10-6
Control Word Encoding for Instruction Decoder

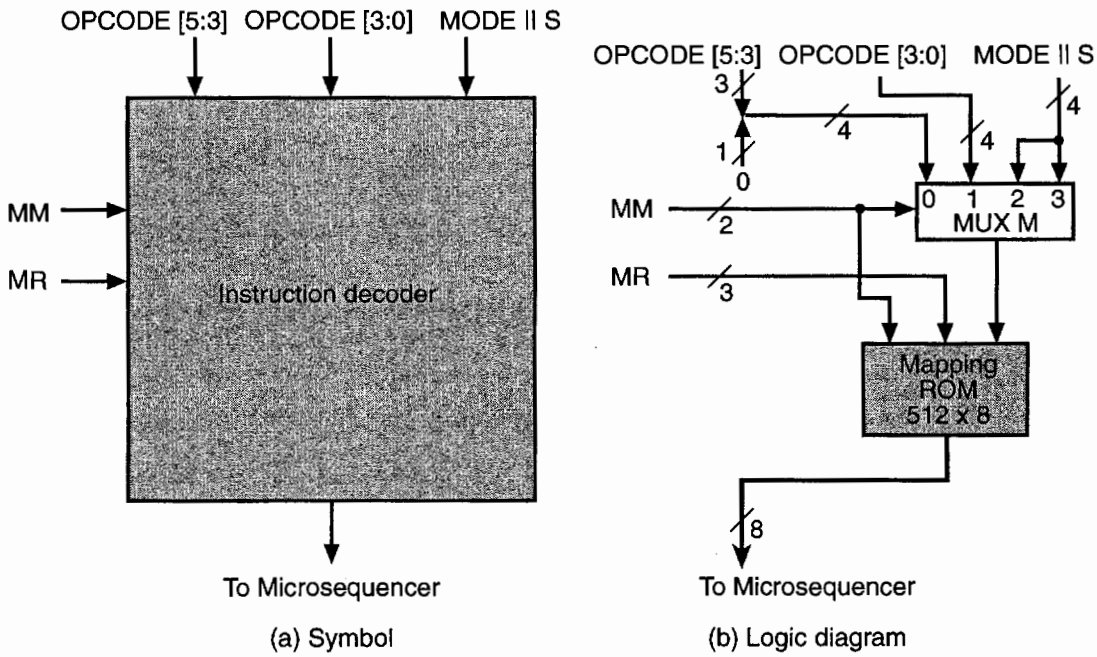
MM		MR	
Select	**Code**	**Region**	**Code**
OPCODE(5:3)	00	0	000
OPCODE(3:0)	01	1	001
MODE || S	10	2	010
MODE || S	11	3	011
		4	100
		5	101
		6	110
		7	111

microinstructions, with execution of only two of them in sequence, it is possible to produce a 64-way branch giving a different address for each of the 64 possible OPCODE values. In contrast, if 2-way branches were used, 63 instructions with six microinstructions in sequence would be required to perform the same decoding to unique addresses.

To handle the MODE and S fields, MM equal to 10 or 11 uses the 4-bit field pair MODE || S to provide up to 16 different addresses. The microprogram region field MR provides distinct sets of addresses for the same *IR* fields. For example, it may be necessary to have several distinct sets of addresses, with each set resulting from values given in the rightmost four bits of OPCODE. The MR field provides a means for the microinstruction to select these various sets. Since this field has three bits, up to eight distinct sets of addresses can be selected for each of the four binary values of MM.

The internal implementation of the instruction decoder, whose symbol is given in Figure 10-10(a) is made up of a quad 4–to–1 multiplexer MUX *M* and a ROM, as shown in Figure 10-10(b). The ROM has nine inputs and eight outputs. MUX *M* is controlled by MM, and the ROM has MM, MR, and the MUX *M* outputs as its inputs. The ROM maps its input values to outputs that are the addresses to which the microsequencer is to jump. Thus, the ROM is referred to as a *mapping ROM*. Because the addresses can be assigned arbitrarily, the location of the various microprogram routines can be determined by the designer and then implemented by the mapping ROM. The content of the ROM is tightly dependent, however, on the relationship between *IR* bit values and the microroutines, so its specification will be deferred until we consider the latter. Although in this case we have chosen a ROM for the decoding process, gate logic or a PLA could also be used.

The final field, MO, for miscellaneous operations is present in both formats A and B. Table 10-7 gives the functions performed for the codes in this field. These codes are carefully defined to provide the control necessary for implementing the instruction set architecture. The codes control the loading of memory, *PC*, *IR*, *SP*, *PSR*, and *MSTS*. In addition, code 1011 replaces the C_{in} value from the FS field



□ FIGURE 10-10
Instruction Decoder

□ TABLE 10-7
Control Information for Miscellaneous Operations for Format A and B Microinstructions

	MO	
Operations	**Symbolic Notation**	**Code**
No operation	—	0000
INACK = 1	INCK	0001
Memory write*	WRITE	0010
Load *PC**	LPC	0011
Load *IR* and increment *PC**	DPC	0100
Increment *PC*	IPC	0101
Load *PSR**	LST	0110
Load *SP**	LSP	0111
Decrement *SP*	DSP	1000
Decrement *SP* and memory write*	DSM	1001
Increment *SP*	ISP	1010
Select *C* as C_{in} for arithmetic and action EST as follows	CIN	1011
Enable update of status bits *Z*, *N*, *C*, and *V*	EST	1100
Enable update of status bits *Z* and *C*	EZC	1101
Enable update of status bits *N* and *Z*	ENZ	1110
Enable update of microstatus bits *z*, *n*, *c*, and *v*	EMS	1111

* Prevents write to register file

with the value stored in the *C* status bit and enables update of the *PSR*. The C_{in} replacement is needed for the ADDC and SUBB instructions. So that information intended for memory or other registers is not written into the register file, writing to the file is blocked for memory write and loads into the *PC*, *IR PSR*, and *SP*. An RW field in the microinstruction has greater flexibility, but requires an extra bit. Note that only one MO code can be used at a time, so there must be a code for each combination of the various operations required.

Microprogram Structure

We approach the microprogram design top down. The top level consists of an ASM- like chart giving a flow of microroutines. These routines have labels similar to the stages in the pipelined CPU in Section 8-11. In this case, however, rather than being performed in a single clock cycle with combinational logic, the routines require the use of the same hardware over multiple cycles. The flow between and, to some extent, within the routines is intimately tied to the instructions and their decoding. Since the mapping ROM can be used for branching simultaneously with a format A data transfer or manipulation operation, it is convenient to control the flow between microroutines entirely by using the mapping ROM. This flow is shown in Figure 10-11; the chart is not strictly an ASM chart, since each rectangular box corresponds to microroutines representing multiple states rather than a single state and to multiple clock cycles rather than a single one.

The execution of each instruction begins with the Instruction Fetch microroutine. The *PC* provides the address and is updated to the next address. The instruction fetched is placed in the *IR*. Then the instruction-decoding process begins, using MUX *M* and the mapping ROM. For MM equal to 00, only the first three bits of OPCODE are used, with the remaining bit set to 0. In addition, the third bit from the left is ignored except when the first two bits equal 01. A five-way branch results. This branch is represented by the five binary decision boxes in the figure. Since the bits of OPCODE that are used denote the number of operands for the instruction being decoded, the destinations of the branches are, in three cases, microroutines to fetch the operands. In another case, program branch addresses are fetched. In the final case, the branch in the chart goes directly to execution. There are three paths to execution blocks that are dependent upon the decision made by the decision boxes. These paths preserve information from the decoding of the three bits of OPCODE in the Instruction Fetch. Thus, there is no need to examine these bits again later in the microprogram in order to determine the operation that is to be executed. But because of the fact that the shift operations require a single operand, but use the two-operand fetch to obtain the shift amount parameter, there is an additional decision required at the end of the two-operand fetch.

In four of the five decisions, an operand fetch routine is performed. Depending upon the first three bits of OPCODE, either a single operand, two operands (or one operand plus a parameter), or a branch address is fetched. The operand address, and parameter values are placed in locations reserved for them in registers *R*12 through *R*15 (*SA*, *SD*, *DA*, and *DD*). The four execution routines find the operands and addresses in these standard register locations and, in most cases, use

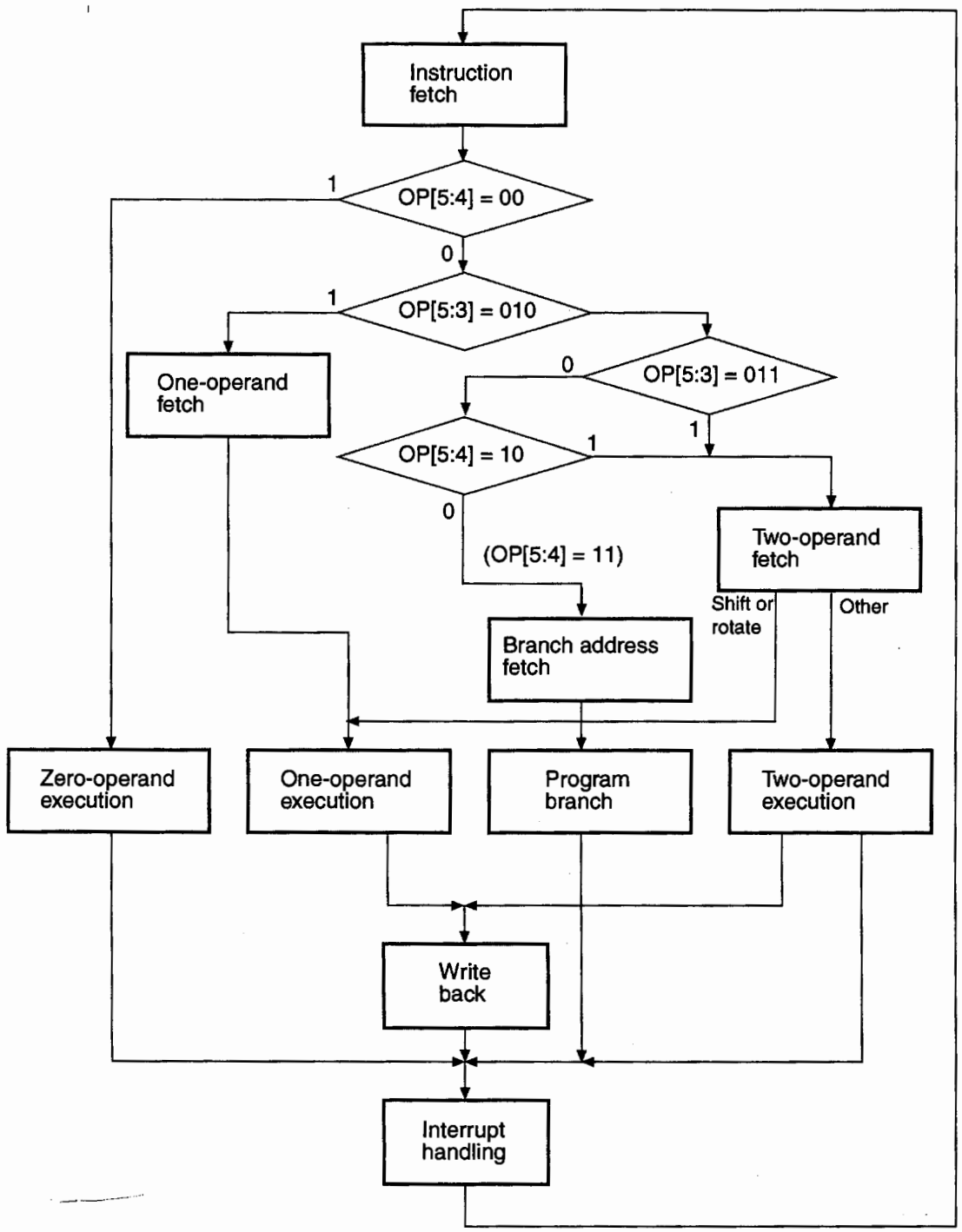


FIGURE 10-11
ASM-like Chart of Microroutines

them to produce a result that is left in standard location *DD*. The Write Back routine also uses the standard register locations to find the result and its address.

Following their execution, it is necessary for most operations to place the result in its destination. This is accomplished by the Write-Back microroutine. Some of the operations, however, do not have a result to be written in that routine. The existence

of these operations is apparent from the paths leading directly from Zero-operand Execution, Program Branch and Two-operand Execution to Interrupt Handling. After each execution microroutine, the program enters the Interrupt Handling microroutine to check for an interrupt before fetching the next instruction.

The flow just described demonstrates the use of the mapping ROM in the Instruction Fetch microroutine. All mappings performed in Instruction Fetch and other microroutines are represented in Table 10-8, the programming table for the mapping ROM. This ROM is used in the execution of particular microinstructions that have the value MAP (10) for MC. The symbolic addresses of these microinstructions are designated in the leftmost column of the table. If the mapping ROM is used in executing a microinstruction, the microinstruction specifies the values of the MM and MR fields in order to define the pattern to be matched to the MUX *M* input to the ROM. In addition to the patterns, the origin in the *IR* of the bits is given for clarity. Where *X*'s appear in bits of the *IR* matching patterns, any possible combination of 1's and 0's on these bits gives the single corresponding symbolic address as the ROM output. If there are four X_s', then 16 ROM rows are represented by a single table row. Where *O*'s or *M*'s with subscripts appear, or where *S* appears, the mapping ROM is being used for an unconditional multiway branch, so these rows really correspond to 8 and 16 different rows giving 8 or 16 symbolic output addresses, respectively. Each of these addresses causes the *CAR* to execute the next microinstruction from a different location. In the case where *O*'s are used, the OPCODE is being decoded into *CAR* addresses for the various operations. Where *M*'s and *S* appear, the MODE and S values are respectively generating the addresses for the microroutines that find the effective addresses specified in the instruction being executed.

Microroutines

We are now prepared to look at the microroutines that implement the CISC CPU. We will use only symbolic addresses for the microinstructions. Assigning binary addresses is quite straightforward because of the addressing flexibility provided by the mapping ROM. Register transfer descriptions are given for each microinstruction. In fact, the microprogram is initially written in terms of these descriptions, and the binary code is added afterward. We use the same position in the tables for fields in both the A and B formats. The field name in the A format is followed by a slash (/), with the name in the B format after the slash. The B format names apply to entries in the tables only for MC equal to 3. Binary values in the fields are given base 16.

The Instruction Fetch microroutine is shown in Table 10-9. The instruction fetch occurs in microaddress IF0, where the instruction is fetched from memory and placed in the *IR*. The *PC* is also simultaneously updated to point to the next instruction. In IF1, instruction decoding begins, with the first two bits of the OPCODE used by the mapping ROM to determine the number of operands in the instruction. According to rows labeled IF1 in Table 10-8, the MM and MR fields must contain 00 and 000. The next microinstruction to be executed, based on the first two bits of OPCODE, is the first microinstruction in one of the following microroutines in Figure 10-11: Zero-operand Execution, One-operand Fetch, Two-operand Fetch, and Branch Address Fetch.

☐ TABLE 10-8
Programming Table for Mapping ROM

Location of Microinstructions Performing a Mapping	ROM Inputs			ROM Outputs—Location of Next Microinstruction
Symbolic Microaddress	**MM**	**MR**	**IR Bits and Match Field**	**Symbolic Microaddress**
IF1	00	000	OPCODE(5:3)‖0 = 00X0	0EX
IF1	00	000	OPCODE(5:3)‖0 = 0100	1OF
IF1	00	000	OPCODE(5:3)‖0 = 0110	2OF
IF1	00	000	OPCODE(5:3)‖0 = 10X0	2OF
IF1	00	000	OPCODE(5:3)‖0 = 1011	BAF
In 1OF Microroutine	00	001	OPCODE(5:3)‖0 = XXXX	1EX
In 2OF Microroutine	00	010	OPCODE(5:3)‖0 = 0110	1EX
In 2OF Microroutine	00	010	OPCODE(5:3)‖0 = 10X0	2EX
In BAF Microroutine	00	011		BEX
0EX	01	000	OPCODE(3:0) = $O_3O_2O_1O_0$	16 0-Op EX Microinstruction Addresses
1EX	01	001	OPCODE(3:0) = $O_3O_2O_1O_0$	16 1-Op EX Microinstruction Addresses
2EX	01	010	OPCODE(3:0) = $O_3O_2O_1O_0$	16 2-Op Ex Microinstruction Addresses
BEX	01	011	OPCODE(3:0) = $O_3O_2O_1O_0$	16 Br-Op EX Microinstruction Addresses
In EX Microroutines	01	100	OPCODE(3:0) = XXXX	WB0
In EX Microroutines	01	101	OPCODE(3:0) = XXXX	INT0
1OF	10	001	MODE ‖ S = $M_2M_1M_0X$	8 1-Op OF Microinstruction Addresses
2OF	10	010	MODE ‖ S = $M_2M_1M_0$ S	16 2-Op OF Microinstruction Addresses
BAF	10	011	MODE ‖ S = $M_2M_1M_0$ X	8 Br-Op Microinstruction Addresses
XCH2	10	100	MODE ‖ S = XXX1	XCH3
XCH2	10	100	MODE ‖ S = XXX1	XCH3
XCH2	10	100	MODE ‖ S = XXX1	XCH4
XCH2	10	100	MODE ‖ S = XXX1	XCH4
XCH2	10	100	MODE ‖ S = XXX1	XCH4
WB0	11	000	MODE ‖ S = XXX0	WB1
WB0	11	000	MODE ‖ S = 0001	WB1
WB0	11	000	MODE ‖ S = 1XX1	WB2
WB0	11	000	MODE ‖ S = X1X1	WB2
WB0	11	000	MODE ‖ S = XX11	WB2
In WB microroutine	11	001	MODE ‖ S = XXXX	INT0
In INT microroutine	11	010	MODE ‖ S = XXXX	IF0

Suppose that OPCODE is 010000 and that MODE is 011. Then, from the second row of Table 10-8, the next microinstruction is in microroutine One-operand Fetch, which begins with microinstruction 1OF. This microroutine is given in

□ TABLE 10-9
Instruction Fetch Microroutine

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
IF0	$IR \leftarrow M[PC], PC \leftarrow PC+1$	0	0	0	00	00	1	0	1	00	4
IF1	$CAR \leftarrow ROM[00000_2 \parallel OPCODE(5{:}4) \parallel 00_2]$	2	0	0	00	00	0	0	0	00	0

□ TABLE 10-10
One-operand Fetch Microroutine

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
1OF	$CAR \leftarrow ROM[10001_2 \parallel MODE \parallel S]$	2	2	1	00	00	0	0	0	00	0
1RG	$DD \leftarrow R[DST], CAR \leftarrow$ 1EX(ROM)	2	0	1	0F	10	0	0	0	00	0
1RGI0	$DA \leftarrow R[DST]$	0	0	0	0E	10	0	0	0	00	0
1RGI1	$DD \leftarrow M[DA], CAR \leftarrow$ 1EX(ROM)	2	0	1	0F	0E	3	0	1	00	0
1IM	$DD \leftarrow M[PC], PC \leftarrow PC+1,$ $CAR \leftarrow$ EX1(ROM)	2	0	1	0F	00	1	0	1	00	5
1DR0	$DA \leftarrow M[PC], PC \leftarrow PC+1$	0	0	0	0E	00	1	0	1	00	5
1DR1	$DD \leftarrow M[DA], CAR \leftarrow$ 1EX(ROM)	2	0	1	0F	0E	3	0	1	00	0
1ID0	$DA \leftarrow M[PC], PC \leftarrow PC+1$	0	0	0	0E	00	1	0	1	00	5
1ID1	$DA \leftarrow DA + R[DST]$	0	0	0	0E	10	0	0	0	02	0
1ID2	$DD \leftarrow M[DA], CAR \leftarrow$ 1EX(ROM)	2	0	1	0F	0E	3	0	1	00	0
1IDI0	$DA \leftarrow M[PC], PC \leftarrow PC+1$	0	0	0	0E	00	1	0	1	00	5
1IDI1	$DA \leftarrow DA + R[DST]$	0	0	0	0E	10	0	0	0	02	0
1IDI2	$DA \leftarrow M[DA]$	0	0	0	0E	00	0	0	1	00	0
1IDI3	$DD \leftarrow M[DA], CAR \leftarrow$ 1EX(ROM)	2	0	1	0F	0E	3	0	1	00	0
1RL0	$DA \leftarrow M[PC], PC \leftarrow PC+1$	0	0	0	0E	00	1	0	1	00	5
1RL1	$DA \leftarrow DA + PC$	0	0	0	0E	0E	1	0	0	02	0
1RL2	$DD \leftarrow M[DA], CAR \leftarrow$ 1EX(ROM)	2	0	1	0F	0E	0	3	1	00	0
1RLI0	$DA \leftarrow M[PC], PC \leftarrow PC+1$	0	0	0	0E	00	1	0	1	00	5
1RLI1	$DA \leftarrow DA + PC$	0	0	0	0E	0E	1	0	0	02	0
1RLI2	$DA \leftarrow M[DA]$	0	0	0	0E	00	0	0	1	00	0
1RLI3	$DD \leftarrow M[DA], CAR \leftarrow$ 1EX(ROM)	2	0	1	0F	0E	3	0	1	00	0

Table 10-10. The first instruction involves the use of the mapping ROM to decode the combined MODE and S fields of the instruction in order to determine the addressing mode. Since there is only a single operand, the S field has no effect on the microroutines. Consequently, we will find the same microroutines used for both S equal to 0 and S equal to 1. This means that there are only 8 addresses to which the MODE values are mapped, instead of 16. Since MODE has the value 011, the mode is direct addressing. The microcode for this mode begins in 1DR0, the address the mapping ROM will provide from 1OF. In 1DR0, the *PC* points to the word W of the instruction. This word is fetched from memory *M* and placed

in the destination address register *DA*. Simultaneously, the *PC* is updated by 1. In 1DR1, the address in *DA* is then used as a direct address to fetch the operand for register *DD* from memory *M*. Simultaneously, the ROM maps its inputs to address 1EX, to begin the executing instruction. This mapping uses MM equal to 00 and MR equal to 001, as shown in Table 10-8. Since the value of the OPCODE(5:4) || 00_2 field to be matched in the ROM is all X's, the contents of OPCODE(5:4) have no effect on the mapping, making it an unconditional jump. In Table 10-10, rather than include this detail, we simply show the next address for the *CAR* as 1EX and use "(ROM)" to indicate that this address was produced by the mapping ROM. The values of MM and MR needed for the mapping ROM appear in the microcode portion of the table in 1DR1.

Otherwise, the Table 10-10 contains eight routines accessed from the 8-way branch in address 1OF, one for each of the eight addressing modes. The first two letters in the symbolic address denote the addressing mode: RG for register, IM for immediate, ID for indexed, and RL for relative. The presence of an I as the third letter denotes the use of indirect addressing in that mode. The one exception to this notation is the use of DR instead of IMI (immediate indirect) for direct addressing..

□ TABLE 10-11
Two-operand Fetch Microroutine Examples

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
2OF	$CAR \leftarrow ROM[10010_2 \parallel MODE \parallel S]$	2	2	2	00	00	0	0	0	00	0
2DR0	$SA \leftarrow M[PC], PC \leftarrow PC + 1$	0	0	0	0C	00	1	0	1	00	5
2DR1	$SD \leftarrow M[SA]$	0	0	0	0D	0C	3	0	1	00	0
2DR2	$DD \leftarrow R[DST], CAR \leftarrow$ 2EX(ROM)	2	0	2	0F	10	0	0	0	10	0
2DRS0	$DA \leftarrow M[PC], PC \leftarrow PC + 1$	0	0	0	0E	00	1	0	1	00	5
2DRS1	$DD \leftarrow M[DA]$	0	0	0	0F	0E	3	0	1	00	0
2DRS2	$SD \leftarrow R[SRC], CAR \leftarrow$ 2EX(ROM)	2	0	2	0D	11	0	0	0	10	0

The microroutine Two-operand Fetch is similar to One-operand Fetch, with two exceptions: A second operand that is always located in a register must be fetched for execution. The S bit determines to which of the two operands the addressing modes are applied and which lies in a register. If S is 0, the addressing mode is applied to the source operand, and the destination operand and result are in a register. If S is 1, the addressing mode is applied to the destination operand and result, and the source is a register. Two-operand Fetch is illustrated for the direct addressing mode in Table 10-11. The mapping ROM is used in 2OF to perform a 16-way branch. In the table, however, the microcode is shown only for the case of direct addressing, MODE = 011. Microinstructions in 2DR0 through 2DR2 handle direct addressing for S = 0 and, in addresses 2DRS0 through 2DRS2, handle direct addressing for S = 1. For the case S = 0, the contents of the W word in *M*[*PC*] are transferred to source address register *SA*, and the *PC* is updated. Then, *SA* is used as the direct address to fetch the source operand in *M*[*SA*]. Finally, the

contents of the destination register $R[DST]$ are transferred to destination data register DD for execution. If there is a result, it will be written into $R[DST]$ in the Write-Back routine. For the case S = 1, direct addressing using word W is applied to obtain destination address DA and destination data DD. The contents of the source register $R[SRC]$ are transferred as the source data to register SD for execution. In this case, in the Write-Back microroutine, a result will be written into memory M at the location addressed by DA.

Branch address fetch is illustrated for direct addressing in Table 10-12. The microinstruction in location BAF performs an 8-way branch based on the MODE field, so that the instruction-specified mode is used to obtain the branch address. In general, the microcode resembles that for the routine One-operand Fetch. However, since we are interested in the destination address rather than the destination operand, the routine jumps to branch execution beginning at location BEX as soon as the destination address has been placed in DA. In the case of direct addressing that is illustrated, the word W is transferred into DA from the address given by the PC, and the PC is updated. The contents of DA will then be transferred into the PC in the branch execution routine. Note that since DA is used, register mode 000 is invalid for branch instructions, because in this case the address is not a memory address, but a CPU register. If the contents of a register are to be used as the branch address, then register indirect, 001, is the proper mode to use.

□ TABLE 10-12
Example of Branch Address Fetch

Sym Add	Register Transfer Description	MC	MM/ LS	MR/ PS	DSA/ MS	SB	MA	MB	MD	FS /NA	MO
BAF	$CAR \leftarrow ROM[10011_2 \parallel MODE \parallel S]$	2	2	3	00	00	0	0	0	00	0
BDR0	$DA \leftarrow M[PC], PC \leftarrow PC + 1,$ $CAR \leftarrow$ BEX(ROM)	2	0	3	0E	00	0	1	1	00	5

The next set of routines perform the actual execution of instructions. In Table 10-13, three instructions not having explicit operands are detailed. In location 0EX, a 16-way branch based on the last four bits of OPCODE jumps to the microroutine for the instruction to be executed. Note that address 0EX is entered from the Instruction Fetch microroutine implying that the first two bits of OPCODE are 00. Thus, the microcode executing the instruction is determined on the basis of all six bits of OPCODE.

The first instruction implemented in the table is push registers, PSHR. .This instruction pushes the seven programmer-accessible registers onto the stack, with $R1$ first. Recall that the stack grows from higher addresses toward lower addresses; thus, the decrement is used when pushing items onto the stack. First, the stack pointer SP is decremented to point to a vacant location. Then, each in a series of six microinstructions saves one register in turn on the stack and decrements the stack pointer SP to provide a location on the stack for the contents of the next reg-

TABLE 10-13
Zero-operand Execution Microroutine Examples

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
0EX	$CAR \leftarrow ROM[01000_2 \parallel OPCODE[3:0]]$	2	1	0	00	00	0	0	0	00	0
PSHR0	$SP \leftarrow SP - 1$	0	0	0	00	00	0	0	0	00	8
PSHR1	$M[SP] \leftarrow R1, SP \leftarrow SP - 1$	0	0	0	00	01	2	0	0	00	9
PSHR2	$M[SP] \leftarrow R2, SP \leftarrow SP - 1$	0	0	0	00	02	2	0	0	00	9
PSHR3	$M[SP] \leftarrow R3, SP \leftarrow SP - 1$	0	0	0	00	03	2	0	0	00	9
PSHR4	$M[SP] \leftarrow R4, SP \leftarrow SP - 1$	0	0	0	00	04	2	0	0	00	9
PSHR5	$M[SP] \leftarrow R5, SP \leftarrow SP - 1$	0	0	0	00	05	2	0	0	00	9
PSHR6	$M[SP] \leftarrow R6, SP \leftarrow SP - 1$	0	0	0	00	06	2	0	0	00	9
PSHR7	$M[SP] \leftarrow R7, CAR \leftarrow$ INT0(ROM)	2	1	5	00	07	2	0	0	00	2
RET0	$PC \leftarrow M[SP]$	0	0	0	00	00	2	0	1	00	3
RET1	$SP \leftarrow SP + 1, CAR \leftarrow$ INT0(ROM)	2	1	5	00	00	0	0	0	00	A
RTI0	$PSR \leftarrow M[SP]$	0	0	0	00	00	2	0	1	00	6
RTI1	$SP \leftarrow SP + 1$	0	0	0	00	00	0	0	0	00	A
RTI3	$PC \leftarrow M[SP],$	0	0	0	00	00	2	0	1	00	3
RTI4	$SP \leftarrow SP + 1, CAR \leftarrow$ INT0(ROM)	2	1	5	00	00	0	0	0	00	A

ister. The final microinstruction saves *R7* and includes a jump to the interrupt microroutine to check for interrupts. Because of the many writes required and the fact that writes are to the stack, the Write Back microroutine is bypassed, and the writes are performed instead in the execution microroutine.

The second instruction implemented in the table is return from procedure, RET. In this instruction, the stored value of the *PC,* which points to the instruction after a call procedure, is returned to the *PC* from the stack. The top item is transferred from the stack to the *PC*, in RET0. In RET1, the stack pointer is incremented and control is transferred to INT0.

The final instruction in the table is return from interrupt, RTI. This instruction is similar to RET, except that when the interrupt occurred, two words—the contents of the *PC* and *PSR*—were placed on the stack. Thus, the value of the *PSR* must be retrieved from the stack before that of the *PC* is retrieved. Note that when the value of the *PSR* is retrieved, the enable interrupt *EI* is restored to the value it had before the interrupt occurred. If *EI* is 1, the interrupt is enabled. As for all zero-operand instructions, the Write-Back microroutine is bypassed, and the microprogram flow goes to INT0, the beginning of the Interrupt Handling microroutine.

Next, we examine a sample of microroutines for the execution of one-operand instructions in Table 10-14. The microroutines begin in 1EX with a 16-way branch based on the last four bits of OPCODE. This completes instruction decoding and selects one of 16 microcode segments that executes the instruction. Note that in the case of the one-operand instructions, the operand is placed in register *DD*, and the result is to be left in that register for the Write Back microroutine. Also, as *DD* is loaded, the codes in MO cause status bits to be set. The three instructions increment (INC), decrement (DEC), and complement (COM) are each executed by a single microinstruction using register

□ TABLE 10-14
One-operand Execution Microroutine Examples

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
1EX	$CAR \leftarrow ROM[01001_2 \parallel OPCODE(3{:}0)]$	2	1	1	00	00	0	0	0	00	0
INC	$DD \leftarrow DD+1, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	00	0	0	0	01	C
DEC	$DD \leftarrow \overline{DD}-1, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	00	0	0	0	06	C
NEG0	$DD \leftarrow \overline{DD}$	0	0	0	0F	00	0	0	0	0E	0
NEG	$DD \leftarrow \overline{DD}+1, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	00	0	0	0	01	C
COM	$DD \leftarrow \overline{DD}, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	00	0	0	0	0E	E
SHR0	$R9 \leftarrow SD$	0	0	0	09	0D	0	0	0	10	0
SHR1	$R9 \leftarrow R9$ (Set MSTS)	0	0	0	09	0	0	0	0	00	F
SHR2	$z{:}CAR \leftarrow SHR6$	3	0	0	6	00	0	0	0	SHR6	0
SHR3	$DD \leftarrow 0 \parallel DD(15{:}1)$	0	0	0	0F	0F	0	0	0	11	0
SHR4	$R9 \leftarrow R9-1$	0	0	0	09	00	0	0	0	06	F
SHR5	$\bar{z}{:}\ CAR \leftarrow$ SHR3	3	0	1	6	00	0	0	0	SHR3	0
SHR6	$DD \leftarrow DD, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	00	0	0	0	00	D

DD. The last instruction, logical shift right (SHR), in contrast, requires six microinstructions and from 2 to 48 clock cycles for its execution. In SHR0 and SHR1, the shift amount value that is stored in *R*13 (SD) is transferred to *R*9, which will be used in a loop to count the number of shifts remaining to be performed. MSTS load is enabled during this operation so that if the shift amount in *R*13 is 0, there is no shifting to be done. In this case, in SHR1, the microroutine jumps to the interrupt microroutine with *DD* unchanged. In SHR2, *DD* is shifted to the right by one bit position, giving the incoming bit the value 0. The composite register (||) notation is used here because it easily describes the incoming bit value. Next, *R*9 is decremented to indicate that one less shift remains to be done. *MSTS* load is enabled during this operation, and in SHR4, the *z* bit is examined. If it is 0, the number of shifts remaining is nonzero, resulting in a jump to SHR3 to perform another shift. If *z* is 1, then there are no remaining shifts. In SHR5, the status values *Z* and *C* are determined and a jump to WB0 occurs, so that the resulting value in *DD* can be stored in the destination location.

The implementation of the execution of two-operand instructions is illustrated in Table 10-15. As with other execution microroutines, 2EX contains a 16-way branch based on the last four bits of OPCODE. Two-operand instructions expect the source address to be in register *SA*, the source operand in *SD*, the destination address in *DA*, and the destination operand in *DD*. Thus, the first instruction illustrated, MOVE, is executed by simply transferring the contents of *SD* to *DD*. The Two-operand Fetch microroutine has done the hard job of obtaining the source operand and of producing the destination address. The Write-Back routine will do the job of placing the new destination data in the destination address. This same notion applies to the add (ADD) and add with carry (ADDC) microcode appearing in the table. The compare (CMP) instruction sets status bits in *PSR*, but is not to change the destination data. Thus, the microcode bypasses Write Back, going directly to Interrupt Handling.

TABLE 10-15
Two-operand Execution Microroutine Examples

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
2EX	$CAR \leftarrow ROM[01010_2 \parallel OPCODE(0:3)]$	2	1	2	00	00	0	0	0	00	0
MOVE	$DD \leftarrow SD, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	0D	0	0	0	10	0
XCH0	$R9 \leftarrow SD$	0	0	0	09	0F	0	0	0	10	0
XCH1	$SD \leftarrow DD$	0	0	0	0E	0F	0	0	0	10	0
XCH2	$DD \leftarrow R9, CAR \leftarrow ROM[10100_2 \parallel MODE \parallel S]$	2	2	4	0F	09	0	0	0	10	0
XCH3	$R[SRC] \leftarrow SD, CAR \leftarrow$ WB0(ROM)	2	1	4	11	0D	0	0	0	10	0
XCH4	$M[SA] \leftarrow SD, CAR \leftarrow$ WB0(ROM)	2	1	4	0C	0D	0	0	0	00	2
ADD	$DD \leftarrow DD + SD, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	0D	0	0	0	02	C
ADDC	$DD \leftarrow DD + SD + C, CAR \leftarrow$ WB0(ROM)	2	1	4	0F	0D	0	0	0	03	B
CMP	$DD \leftarrow DD - SD, CAR \leftarrow$ INT0(ROM)	2	1	5	0F	0D	0	0	0	05	C

The most interesting of the two-operand instructions is exchange (XCH), which exchanges the source and destination data. This microcode segment is a bit unusual, since it has to write results to two locations, yet Write Back can handle only one. As a consequence, the writing of the result into the source is done in the execution microroutine, with the writing of the destination left, as usual, to Write Back. The exchange of the data occurs in XCH0 through XCH2. In addition, in XCH2, the mapping ROM is used to determine whether the addressing modes apply to the source or the destination by examining the value of S in the *IR*. If S is 0, it must be determined whether MODE is 000. If S is 1 or MODE is 000, then the contents of *SD* are returned to register *R*[*SRC*]. Otherwise, for all other modes with S = 0, the contents of *SD* are transferred to *M*[*SA*]. This reasoning corresponds to the five rows labeled with XCH2 (MM = 10 and MR = 100) in the mapping ROM contents in Table 10-8. For the first row, if S = 1, the addressing modes do not apply to the source, so microinstruction XCH3 is executed, to transfer the contents of *SD* to *R*[*SRC*]. For the second row, if S = 0 and MODE = 000, the transfer from *SD* to *R*[*SRC*] in XCH3 is executed, since this is register mode. Otherwise, if S = 0 with at least one of the bits in MODE equal to 1, the contents of *SD* are placed in memory location *SA* using XCH4. This situation corresponds to the last three rows for XCH2 in Table 10-8.

The microinstructions for three branch instructions are presented in Table 10-16. In BEX, there is again a 16-way branch based on OPCODE(3:0). The first instruction, an unconditional jump (JMP), simply takes the effective address from *DA* and places it in the *PC* as the next address. Since, with all branches, there is no Write Back, a jump occurs in the microcode to the Interrupt Handling microroutine. The second instruction, call procedure (CALL), first saves the updated contents of the *PC* onto the stack. It then transfers the destination address to the *PC* to execute the jump. The final two instructions illustrate conditional branches in which the contents of the *PC* are changed only if the condition is satisfied. The first jump occurs for *Z* equal to 1, the second for *Z* equal to 0.

TABLE 10-16
Program Branch Microroutine Examples

ym dd	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
EX	$CAR \leftarrow ROM[01011_2 \parallel OPCODE(3:0)]$	2	1	3	00	00	0	0	0	00	0
MP	$PC \leftarrow DA, CAR \leftarrow$ INT0(ROM)	2	1	5	0E	00	0	0	0	00	3
ALL0	$R8 \leftarrow PC$	0	0	0	08	00	1	0	0	00	0
ALL1	$SP \leftarrow SP - 1$	0	0	0	0	00	0	0	0	00	8
ALL2	$M[SP] \leftarrow R8$	0	0	0	0	08	2	0	0	00	2
ALL3	$PC \leftarrow DA, CAR \leftarrow$ INT0(ROM)	2	1	5	0E	00	0	0	0	00	3
Z0	Z: $CAR \leftarrow$ BRA	3	0	0	1	—	—	—	—	BRA	0
Z1	$CAR \leftarrow$ INT0(ROM)	2	1	5	00	00	0	0	0	00	0
RA	$PC \leftarrow DA, CAR \leftarrow$ INT0(ROM)	2	1	5	0E	00	0	0	0	00	3
NZ0	$\overline{Z}$: $CAR \leftarrow$ BRA	3	0	1	1	—	—	—	—	BRA	0
NZ1	$CAR \leftarrow$ INT0(ROM)	2	1	5	00	00	0	0	0	00	0

For those instructions with results to be stored, the Write-Back microroutine in Table 10-17 is executed. To determine where to put the contents of *DD*, it is necessary to examine the value of S and MODE. If S = 0, then the destination address is register *R*[*DST*]. If S = 1, then the destination is register *R*[*DST*] if MODE = 000. Otherwise, the destination of the result is memory location *SA*. This is the same as the situation in the exchange instruction XCH, except that the specified value of S is opposite. The mapping information is in those rows of Table 10-8 labeled with WB0. Regardless of the Write-Back operation performed, the Interrupt-Handling microroutine is executed next.

□ **TABLE 10-17**
Write Back Microroutine

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	M O
WB0	$CAR \leftarrow ROM[11000_2 \parallel MODE \parallel S]$	2	3	0	00	00	0	0	0	00	0
WB1	$R[DST] \leftarrow DD, CAR \leftarrow$ INT0(ROM)	2	1	5	10	0F	0	0	0	10	0
WB2	$M[DA] \leftarrow DD, CAR \leftarrow$ INT0(ROM)	2	1	5	0E	0F	0	0	0	00	2

The final microroutine, for Interrupt-Handling, is given in Table 10-18. In INT0, if *INTS* is 0, indicating that no interrupt is pending, a jump occurs to IF0, since the processing of instructions is complete. If *INTS* is 1, then the next seven microinstructions are executed to save the values of the *PC* and *PSR* on the stack, disable the interrupts, send an interrupt acknowledge, and load the interrupt vector that results into the *PC*, as described in Chapter 9. This last action starts processing at the beginning of the routine that will service the interrupt. Note that the Interrupt-Handling microroutine is based on the assumption that the interrupt is from an external source: if internal sources are to be considered, they require additional microcode.

□ TABLE 10-18
Interrupt-Handling Microroutine

Sym Add	Register Transfer Description	MC	MM /LS	MR /PS	DSA /MS	SB	MA	MB	MD	FS /NA	MO
INT0	$\overline{INTS}$: $CAR \leftarrow$ IF0	3	0	1	A	00	00	00	00	IF0	0
INT1	$R8 \leftarrow PC$	0	0	0	08	00	1	0	0	00	0
INT2	$SP \leftarrow SP - 1$	0	0	0	00	00	0	0	0	00	8
INT3	$M[SP] \leftarrow R8, SP \leftarrow SP - 1$	0	0	0	00	08	2	0	0	00	9
INT4	$M[SP] \leftarrow PSR$	0	0	0	00	00	2	1	0	00	9
INT5	$PSR \leftarrow 0$	0	0	0	00	00	0	0	0	00	7
INT6	$INACK \leftarrow 1$	0	0	0	00	00	0	0	0	00	1
INT7	$PC \leftarrow IVAD, CAR \leftarrow$ IF0(ROM)	2	3	2	00	00	0	0	1	00	3

10-3 THE REDUCED INSTRUCTION SET COMPUTER

The second design we examine is for a reduced instruction set computer with a pipelined datapath and control unit. We begin by describing the instruction set architecture for this particular RISC design, which is characterized by load/store memory access, four addressing modes, a single instruction format length, and instructions that require only elementary operations. We then design a datapath and control to implement the architecture. The datapath is based on the pipelined datapath initially described in Figure 7-23. In order to implement the instruction set architecture, modifications are made to the register file and the function unit in that figure. The control unit is based on the pipelined control unit added to the datapath in Section 8-11. Due to data and control hazards associated with pipelined designs, not only modifications to the control unit, but further modifications to the datapath, are required.

Instruction Set Architecture

Figure 10-12 shows the CPU registers accessible to the programmer in this RISC. All registers are 32 bits. The register file has 32 registers, *R*0 through *R*31. *R*0 is a special register that always supplies the value zero when it is used as a source and discards the result when it is used as a destination. The size of the programmer-accessible register file is larger in the RISC than in the CISC because of the load/store instruction set architecture. Since the data manipulation operations can use only register operands, many currently active operands should be in the register file. Otherwise, numerous stores and loads would be needed to temporarily save operands in the data memory between data manipulation operations. In addition, in many real pipelines these stores and loads require more than one clock cycle for their execution. To prevent these factors from degrading the performance of the RISC, at least 32 registers are required in the register file.

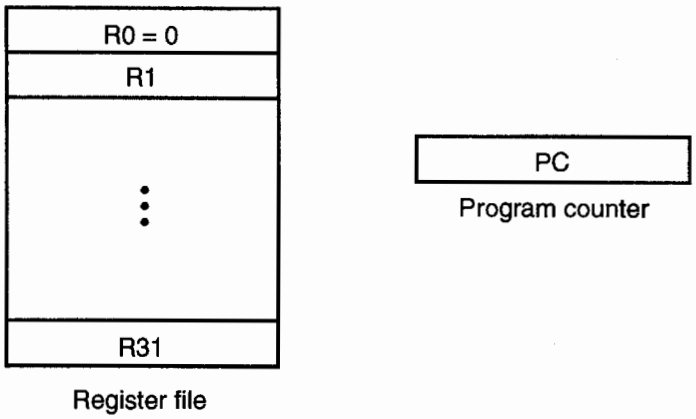


FIGURE 10-12
CPU Register Set Diagram for RISC

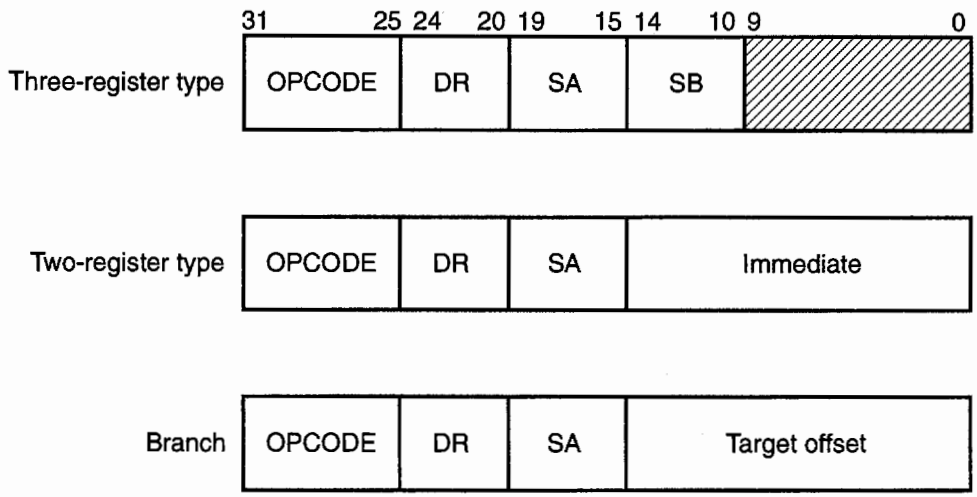


FIGURE 10-13
RISC CPU Instruction Formats

In addition to the register file, only a program counter *PC* is provided. If stack pointer-based or processor status register-based operations are required, they are implemented by sequences of instructions using the register file.

Figure 10-13 gives the three instruction formats for the RISC CPU. The formats use a single word of 32 bits. This longer word length is essential in order to provide realistic address values, since the additional address word used in the CISC CPU is not available in the RISC CPU. The first format specifies three registers. The two registers addressed by the 5-bit source register fields SA and SB contain two operands. The third register, addressed by a 5-bit destination register field DR, specifies the register location for the result. A 7-bit OPCODE provides for a maximum of 128 operations.

The remaining two formats replace the second register with a 15-bit constant. In the two-register format, the constant acts as an immediate operand; in the

branch format, the constant is a *target offset. Target address* is another name for the effective address, particularly if the address is used in a branch instruction. The target address is formed by the addition of the target offset to the contents of the *PC*. Thus, branching uses relative addressing based on the updated value of the *PC*. The branch instructions specify source register SA. Whether the branch or jump is taken is based on whether the contents of the source register are zero. The DR field is used to specify the register in which to store the return address for the procedure call. The rightmost five bits of the 15 bit constant are also used as the shift amount SH.

TABLE 10-19
RISC Instruction Operations

Operation	Symbolic Notation	Opcode	Action
No Operation	NOP	0000000	None
Add	ADD	0000010	$R[DR] \leftarrow R[SA] + R[SB]$
Subtract	SUB	0000101	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$
Set if Less Than	SLT	1100101	If $R[SA] < R[SB]$ then $R[DR] = 1$
AND	AND	0001000	$R[DR] \leftarrow R[SA] \wedge R[SB]$
OR	OR	0001010	$R[DR] \leftarrow R[SA] \vee R[SB]$
Exclusive-OR	XOR	0001100	$R[DR] \leftarrow R[SA] \oplus R[SB]$
Store	ST	0000001	$M[R[SA]] \leftarrow R[SB]$
Load	LD	0100001	$R[DR] \leftarrow M[R[SA]]$
Add Immediate	ADI	0100010	$R[DR] \leftarrow R[SA] + \text{se } IM$
Subtract Immediate	SBI	0100101	$R[DR] \leftarrow R[SA] + \overline{(\text{se } IM)} + 1$
Complement	NOT	0101110	$R[DR] \leftarrow \overline{R[SA]}$
AND Immediate	ANI	0101000	$R[DR] \leftarrow R[SA] \wedge (0 \parallel IM)$
OR Immediate	ORI	0101010	$R[DR] \leftarrow R[SA] \vee (0 \parallel IM)$
Exclusive-OR Immediate	XRI	0101100	$R[DR] \leftarrow R[SA] \oplus (0 \parallel IM)$
Add Immediate Unsigned	AIU	1100010	$R[DR] \leftarrow R[SA] + (0 \parallel IM)$
Subtract Immediate Unsigned	SIU	1100101	$R[DR] \leftarrow R[SA] + \overline{(0 \parallel IM)} + 1$
Move	MOV	1000010	$R[DR] \leftarrow R[SA]$
Logical Left Shift by SH Bits	LSL	0110000	$R[DR] \leftarrow$ lsl $R[SA]$ by SH
Logical Right Shift by SH Bits	LSR	0110001	$R[DR] \leftarrow$ lsr $R[SA]$ by SH
Jump Register	JMR	1100001	$PC \leftarrow R[SA]$
Branch on Zero	BZ	0100000	If $R[SA] = 0$, then $PC \leftarrow PC + 1 + \text{se } IM$
Branch on Nonzero	BNZ	1100000	If $R[SA] \neq 0$, then $PC \leftarrow PC + 1 + \text{se } IM$
Jump	JMP	1000100	$PC \leftarrow PC + 1 + \text{se } IM$
Jump and Link	JML	1000000	$PC \leftarrow PC + 1 + \text{se } IM, R[DR] \leftarrow PC + 1$

Table 10-19 contains the 27 operations to be performed by the instructions. A mnemonic, an opcode, and a register transfer description are given for each operation. All of the operations are elementary and can be described by a single register transfer statement. The only operations that can access memory are Load and Store. A significant number of immediate instructions help to reduce data memory

accesses and speed up execution when constants are employed. Since the immediate field of the instruction is only 15 bits, the leftmost 17 bits must be filled to form a 32-bit operand. In addition to using zero fill, a second method is used called *sign extension*. The most significant bit of the immediate operand, bit 14 of the instruction, is viewed as a sign bit. To form a 32-bit 2's-complement operand, this bit is copied into the 17 bits. In Table 10-19, the sign extension of the immediate field is denoted by se *IM*. The same notation, se *IM*, also represents the sign extension of the target offset field.

The absence of stored versions of status bits is handled by the use of three instructions: Branch if Zero (BZ), Branch if Nonzero (BNZ), and Set if Less Than (SLT). BZ and BNZ are single instructions that determine whether a register operand is zero or nonzero and branch accordingly. SLT stores a value in register $R[DR]$ that acts like a negative status bit. If $R[SA]$ is less than $R[SB]$, a 1 is placed in register $R[DR]$; if $R[SA]$ is greater than or equal to $R[SB]$, a 0 is placed in $R[DR]$. The register $R[DR]$ can then be examined by a subsequent instruction to see whether it is zero (0) or nonzero (1). Thus, using two instructions, the relative values of two operands or the sign of one operand (by letting $R[SB]$ equal to $R0$) can be determined.

The Jump and Link (JML) instruction provides a mechanism for implementing procedures. The value in the *PC* after updating is stored in register $R[DR]$, and then the sum of the *PC* and the sign-extended target offset from the instruction is placed in the *PC*. The return from a called procedure can use the Jump Register instruction with SA equal to DR for the calling procedure. If a procedure is to be called from within a called procedure, then each successive procedure that is called will need its own register for storing the return value. A software stack that moves return addresses from $R[DR]$ to memory at the beginning of a called procedure and restores them to $R[SA]$ before the return can also be used.

Addressing Modes

The four addressing modes in the RISC are register, register indirect, immediate, and relative. The mode is specified by the operation code, rather than by a separate mode field. As a consequence, the mode for a given operation is fixed and cannot be varied. The three-operand data manipulation instructions use register mode addressing. Register indirect, however, applies only to the load and store instructions, the only instructions that access data memory. Instructions using the two-register format have an immediate value that replaces register address SB. Relative addressing applies exclusively to branch and jump instructions and so produces addresses only for the instruction memory.

When programmers want to use an addressing mode, such as indexed addressing, not provided by the instruction set architecture, they must use a sequence of RISC instructions. For example, for an indexed address for a load operation, the desired transfer is

$$R15 \leftarrow M[R5 + 0 \parallel I]$$

This transfer can be accomplished by executing two instructions:

```
AIU R9, R5, I
LD  R15, R9
```

The first instruction, Add Immediate Unsigned, forms the address by appending 17 0's to the left of I and adding the result to *R*5. The resulting effective address is then temporarily stored in *R*9. Next, the Load instruction uses the contents of *R*9 as the address at which to fetch the operand and places the operand in the destination register *R*15. Since, for indexed addressing, I is regarded as a positive offset in memory, the use of unsigned addition is appropriate. This sequence of two operations for indexed addressing is one justification for having unsigned immediate addition available.

Datapath Organization

We use the pipelined datapath in Section 7-11 and Section 8-11 as the basis for the datapath here and deal only with modifications. These modifications affect the register file, the function unit, and the bus structure. The reader should also refer to the datapath in Figure 8-34 and the new datapath shown later in Figure 10-15 in order to understand fully the discussion that follows. We treat each modification in turn, beginning with the register file.

In Figure 8-34, there are 16 16-bit registers, and all registers are identical in function. In the new datapath, there are 32 32-bit registers. Also, the contents of *R*0 are to be read as the constant zero. If a write is attempted into *R*0, the data will be lost. These changes are implemented in the new register file in Figure 10-15. All

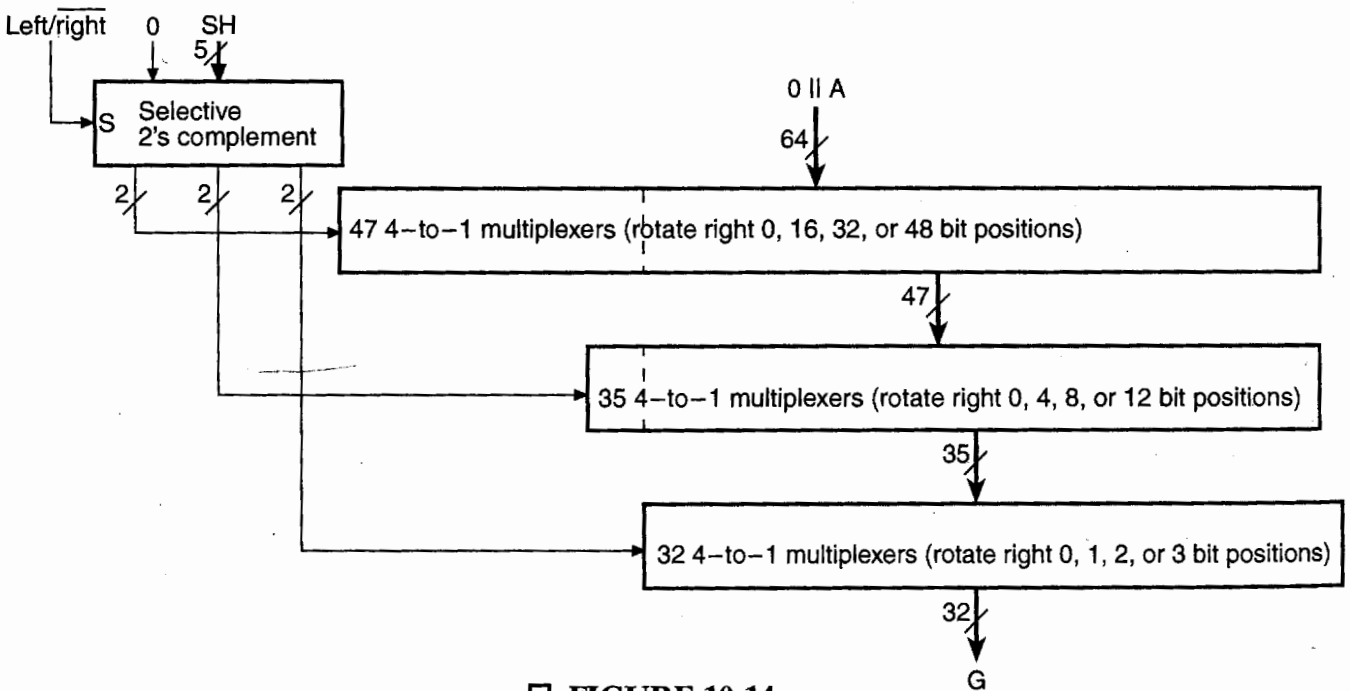


FIGURE 10-14
32-bit Barrel Shifter

data inputs and the data output are 32 bits. To correspond to the 32 registers, the address inputs are all 5 bits. The fixed value of 0 in $R0$ is implemented by replacing the storage elements for $R0$ with open circuits on the lines that were their inputs, and with constant zero values on the lines that were their outputs.

A second major modification to the datapath is the replacement of the single-bit position shifter with a barrel shifter to speed up the execution of multiple-position shifting. This barrel shifter can perform a logical right or logical left shift of from 0 to 31 positions. A block diagram for the barrel shifter appears in Figure 10-14. The data input is 32-bit operand A, and the output is 32-bit result G. Left/right, a control signal decoded from OPCODE, selects a left or right shift. The shift amount field $SH = IR(4{:}0)$ specifies the number of bit positions to shift the data input and takes on values from 0 through 31. A logical shift of p bit positions involves inserting p zeros into the result. In order to provide these zeros and simplify the design of the shifter, we will perform both the left and right shift by using a right rotate. The input to this rotate will be the input data A with 32 zeros concatenated to its left. A right shift is performed by rotating the input p positions to the right; a left shift is performed by rotating $64 - p$ positions to the right. This number of positions can be obtained by taking the 2's complement of the 6-bit value of $0 \,||\, SH$.

The 63 different rotates can be obtained by using three levels of 4–to–1 multiplexers, as shown in Figure 10-14. The first level shifts by 0, 16, 32, or 48 positions, the second level by 0, 4, 8, or 12 positions, and the third level by 0, 1, 2, or 3 positions. The number of positions for A to be shifted, 0 through 63, can be implemented by representing A as a three-digit base-4 integer. From left to right, the digits have weights $4^2 = 16$, $4^1 = 4$, and $4^0 = 1$. The digit values in each of the positions are 0, 1, 2, and 3. Each digit controls a level of the 4–to–1 multiplexers, the most significant digit controlling the first level, the least significant the third level. Due to the presence of 32 zeros in the 64-bit input, fewer than 64 multiplexers can be used in each level. A level requires the number of multiplexers to be 32 plus the total number of positions its output can be shifted by subsequent levels. The output of the first level can be shifted at most $12 + 3 = 15$ positions to the right. Thus, this level requires $32 + 15 = 47$ multiplexers. The output of the second level can be shifted at most 3 positions, giving $32 + 3 = 35$ multiplexers. The final level cannot be shifted further and so needs just 32 multiplexers.

In the function unit, the ALU is expanded to 32 bits, and the barrel shifter replaces the single position shifter. The resulting modified function unit uses the same function codes as in Chapters 7 and 8, except that the two codes for shifts are now labeled as logical shifts. The shift amount SH is a new 5-bit input to the modified function unit in Figure 10-15.

The remaining datapath changes are shown in Figure 10-15. Beginning at the top of the datapath, zero fill has been replaced by the constant unit. The constant unit performs zero fill for $CS = 0$ and sign extension for $CS = 1$. MUX A is added to provide a path for the updated PC, PC_{-1}, to the register file for implementation of the Jump and Link (JML) instruction.

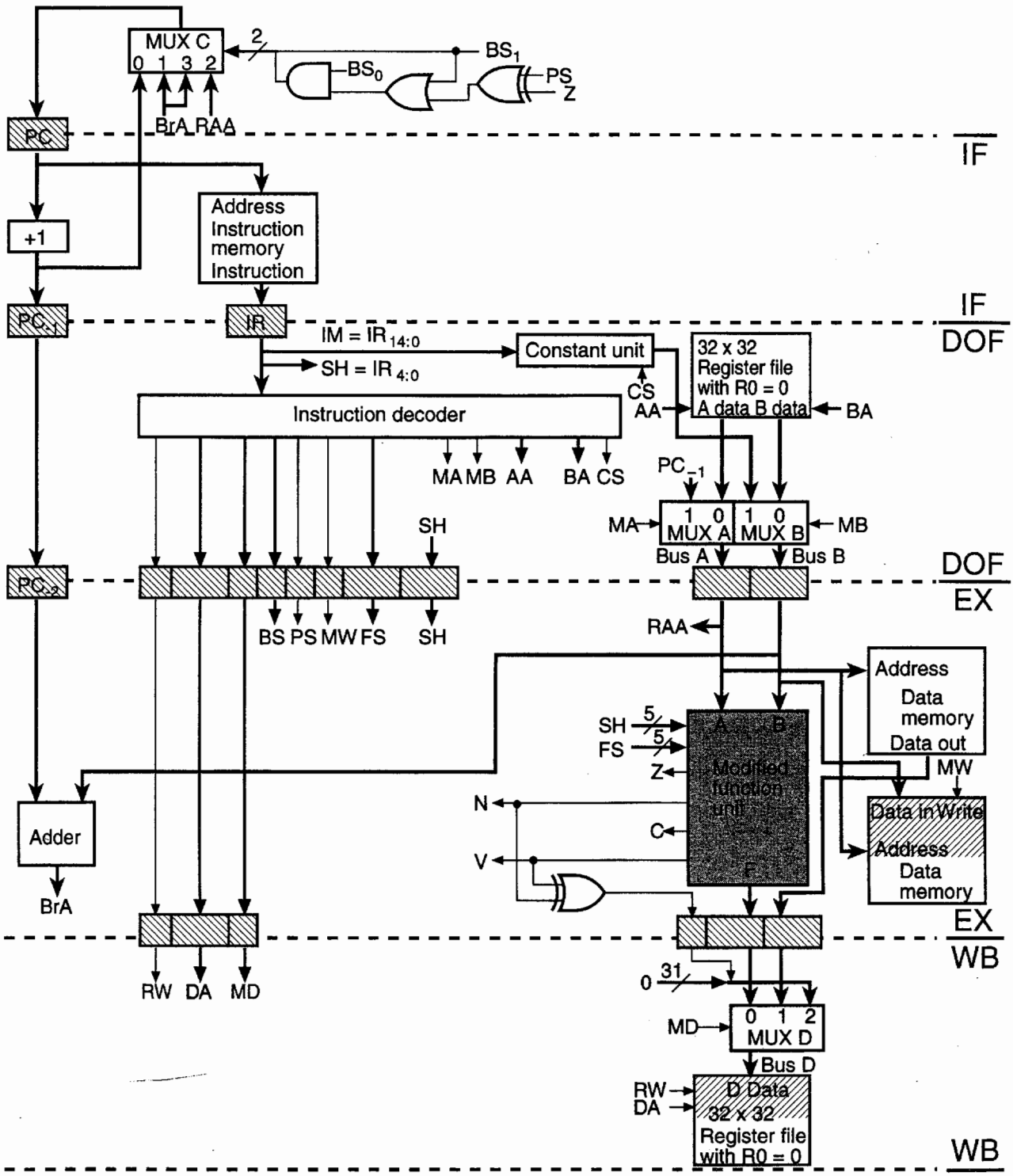


FIGURE 10-15
Pipelined RISC CPU

One other change in the figure helps implement the Set if Less Than (SLT) instruction. This logic provides a 1 to be loaded into $R[DA]$ if $R[AA] - R[BA] < 0$ and a 0 to be loaded into $R[DA]$ if $R[AA] - R[BA] \geq 0$. It is implemented by adding an additional input to MUX *D*. The leftmost 31 bits of the input are 0; the rightmost bit is 1 if *N* is 1 and *V* is 0, i.e., if the result of the subtraction is negative and there is no overflow. It is also 1 if *N* is 0 and *V* is 1, i.e., if the result of the subtraction is positive and there is an overflow. These represent all cases in which $R[AA]$ is greater than $R[BA]$ and which can be implemented using an exclusive-OR of *N* and *V*.

A final difference in the datapath is that the register file is no longer edge triggered and no longer a part of a pipeline platform at the end of the write back (WB) stage. Instead, the register file uses latches and is written much earlier than the positive clock edge. Special timing signals are provided that permit the register file to be written in the first half and to be read in the last half of the cycle. In particular, in the second half of the cycle, it is possible to read data written into the register file during the first half of the same clock cycle. This is called a *read-after-write* register file, and it both avoids added complexity in the logic used for handling hazards and reduces the cost of the register file.

Control Organization

The control organization in the RISC is modified from that in Chapter 8. The modified instruction decoder is essential to deal with the new instruction set. In Figure 10-15, *CS* and *SH* are added from the *IR*, one bit is added to *MD*, and there is a new pipeline platform for *SH* and 2-bit platforms for *MD*.

The remaining control signals are included to handle the new control logic related to the *PC*. All of this logic relates to loading addresses into the *PC* in order to implement branches and jumps. MUX *C* selects from three different sources for the next value of *PC*. The updated *PC* is used to move sequentially through a program. A branch target address *BrA* that is formed from the sum of the updated *PC* value for the branch instruction and the sign-extended target offset is used for branches and jumps. The value in $R[AA]$ is used for the register jump. The selection of these values is controlled by the field *BS*. If $BS0 = 0$, then the updated *PC* is selected by $BS1 = 0$, and $R[AA]$ is selected by $BS1 = 1$. If $BS0 = 1$ and $BS1 = 1$, then *BrA* is selected unconditionally. If $BS0 = 1$ and $BS1 = 0$, then, for $PS = 0$, a branch to *BrA* occurs for $Z = 1$, and for $PS = 1$, a branch to *BrA* occurs for $Z = 0$. This implements the two conditional branch instructions *BZ* and *BNZ*.

In order to have the value of the updated *PC* for the branch and jump instructions when they reach the execution stage, two pipeline registers, PC_{-1} and PC_{-2}, are added. PC_{-2} and the value from the constant unit are inputs to the dedicated adder that forms *BrA* in the execution stage. Note that MUX *C* and the attached control logic are in the EX stage, although shown above the *PC*. The related clock cycle difference causes problems with instructions following branches that we will deal with in later subsections.

The heart of the control unit is the instruction decoder. This is combinational circuitry that converts the operation code in the *IR* into the control signals neces-

sary for the datapath and control unit. In Table 10-20, each instruction is identified by its mnemonic. A register transfer statement and the opcode are given for the instruction. The opcodes are selected such that the least significant five of the seven bits match the bits in the control field FS. This leads to simpler decoding. The register file addresses AA, BA, and DA come directly from SA, SB, and DR, respectively, in the *IR*.

□ TABLE 10-20
Control Words for Instructions

Symbolic Notation	Action	Op Code	Control Word Values								
			RW	MD	BS	PS	MW	FS	MB	MA	CS
NOP	None	0000000	0	—	00	—	0	—	—	—	—
ADD	$R[DR] \leftarrow R[SA] + R[SB]$	0000010	1	00	00	—	0	00010	0	0	—
SUB	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$	0000101	1	00	00	—	0	00101	0	0	—
SLT	If $R[SA] < R[SB]$ then $R[DR] = 1$	1100101	1	10	00	—	0	00101	0	0	—
AND	$R[DR] \leftarrow R[SA] \wedge R[SB]$	0001000	1	00	00	—	0	01000	0	0	—
OR	$R[DR] \leftarrow R[SA] \vee R[SB]$	0001010	1	00	00	—	0	01010	0	0	—
XOR	$R[DR] \leftarrow R[SA] \oplus R[SB]$	0001100	1	00	00	—	0	01100	0	0	—
ST	$M[R[SA]] \leftarrow R[SB]$	0000001	0	00	00	—	1	—	0	0	—
LD	$R[DR] \leftarrow M[R[SA]]$	0100001	1	01	00	—	0	—	—	0	—
ADI	$R[DR] \leftarrow R[SA] + \text{se}\,IM$	0100010	1	00	00	—	0	00010	1	0	1
SBI	$R[DR] \leftarrow R[SA] + \overline{(\text{se}\,IM)} + 1$	0100101	1	00	00	—	0	00101	1	0	1
NOT	$R[DR] \leftarrow \overline{R[SA]}$	0101110	1	00	00	—	0	01110	—	0	—
ANI	$R[DR] \leftarrow R[SA] \wedge (0 \| IM)$	0101000	1	00	00	—	0	01000	1	0	0
ORI	$R[DR] \leftarrow R[SA] \vee (0 \| IM)$	0101010	1	00	00	—	0	01010	1	0	0
XRI	$R[DR] \leftarrow R[SA] \oplus (0 \| IM)$	0101100	1	00	00	—	0	01100	1	0	0
AIU	$R[DR] \leftarrow R[SA] + (0 \| IM)$	1100010	1	00	00	—	0	00010	1	0	0
SIU	$R[DR] \leftarrow R[SA] + \overline{(0 \| IM)} + 1$	1100101	1	00	00	—	0	00101	1	0	0
MOV	$R[DR] \leftarrow R[SA]$	1000000	1	00	00	—	0	00000	—	0	—
LSL	$R[DR] \leftarrow \text{lsl}\,R[SA]\,\text{by}\,SH$	0110000	1	00	00	—	0	10100	—	0	—
LSR	$R[DR] \leftarrow \text{lsr}\,R[SA]\,\text{by}\,SH$	0110001	1	00	00	—	0	11000	—	0	—
JMR	$PC \leftarrow R[SA]$	1100001	0	—	10	—	0	00000	—	—	—
BZ	If $R[SA] = 0$, then $PC \leftarrow PC + 1 + \text{se}\,IM$	0100000	0	—	01	0	0	00000	1	0	1
BNZ	If $R[SA] \neq 0$, then $PC \leftarrow PC + 1 + \text{se}\,IM$	1100000	0	—	01	1	0	00000	1	0	1
JMP	$PC \leftarrow PC + 1 + \text{se}\,IM$	1000100	0	—	11	—	0	—	1	—	1
JML	$PC \leftarrow PC + 1 + \text{se}\,IM, R[DR] \leftarrow PC + 1$	0000111	1	00	11	—	0	00111	1	1	1

Otherwise, to determine the control codes, the CPU is viewed much as is the single-cycle CPU in Figure 8-23. The pipeline platforms can be ignored in this determination; however, it is important to examine the timing carefully to be sure that various parts of the register transfer statement for the operation are taking place in the right stage of the pipeline. For example, note that the adder for the *PC* is in stage EX. This adder is connected to MUX *C* and its attached control logic and to the

incrementer +1 for the *PC*. Thus, all of this logic is in the EX stage, and the loading of the *PC* that begins the IF stage is controlled from the EX stage. Likewise, the input *R*[*AA*] is in the same combinational block of logic and comes not from the *A* Data output of the register file, but from Bus *A* in the EX stage, as shown.

Table 10-20 can serve as the basis for the design of the instruction decoder. It contains the values for all control signals, except the register addresses from *IR*. In contrast to the decoder in Section 8-9, the logic is complex and is most easily designed by using a computer-based logic synthesis program.

Data Hazards

In Section 8-11, we examined a pipeline execution diagram and found that filling and flushing of the pipeline reduced the throughput below the maximum level achievable. Unfortunately, there are other problems with pipeline operation that reduce throughput. In this and the next subsection, we will examine two such problems: data hazards and control hazards. Hazards are timing problems that arise because the execution of an operation in a pipeline is delayed by one or more clock cycles from the time at which the instruction containing the operation was fetched. If a subsequent instruction tries to use the result of the operation as an operand before the result is available, it uses the old or stale value, which is very likely to give a wrong result. To deal with the two types of hazards, we will illustrate at least two solutions, one that uses software and another that uses hardware.

Two data hazards are illustrated by examining the execution of the following program:

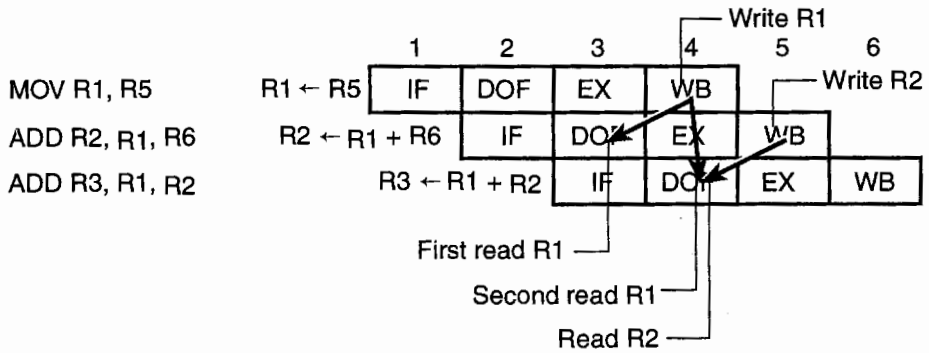
```
1  MOV   R1, R5
2  ADD   R2, R1, R6
3  ADD   R3, R1, R2
```

The execution diagram of this program appears in Figure 10-16(a). The MOV instruction places the contents of *R*5 into *R*1 in the first half of WB in cycle 4. But, as shown by the blue arrow, the first ADD instruction reads *R*1 in the last half of DOF in cycle 3, one cycle before it is written. Thus, the ADD instruction uses the stale value in *R*1. The result of this operation is placed in *R*2 in the first half of WB in cycle 5. The second ADD instruction, however, reads both *R*1 and *R*2 in the second half of DOF in cycle 4. In the case of *R*1, the value read was written in the first half of WB in cycle 4. So the value read in the second half of cycle 4 is the new value. The write-back of *R*2, however, occurs in the first half of cycle 5, after it is read by the next instruction during cycle 4. So *R*2 has not been updated to the new value at the time it is read. This gives two data hazards, as indicated by the large blue arrows in the figure. The registers that are not properly updated to new values are highlighted in blue in the program and in the register transfer statements in the figure. In each of these cases, the read of the involved register occurs one clock cycle too soon with respect to the write of that register.

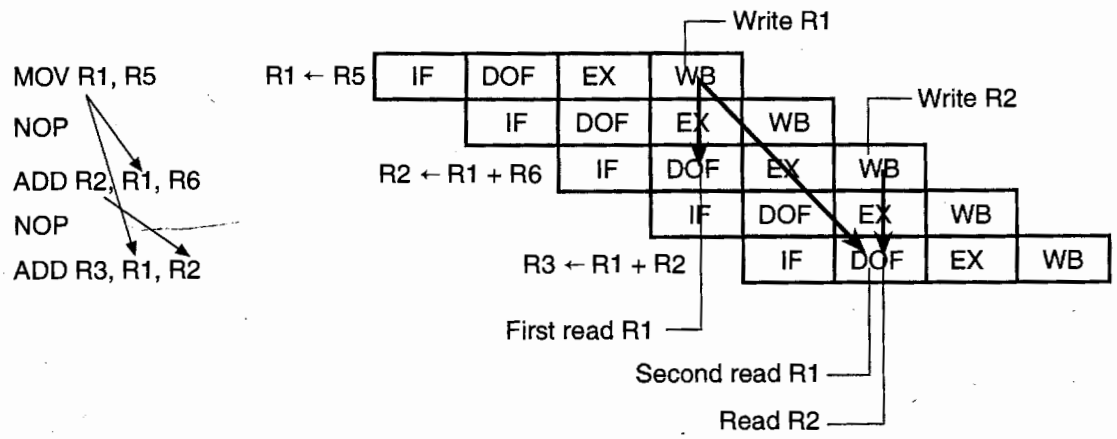
One possible remedy for data hazards is to have the compiler or programmer generate the machine code to delay instructions so that new values are available.

The program is written so that any pending write to a register occurs in the same or an earlier clock cycle than a subsequent read from the register. To accomplish this, the programmer or compiler needs to have detailed information on how the pipeline operates. Figure 10-16(b) illustrates a modification of the simple three-line program that solves the problem. No-operation (NOP) instructions are inserted between the first and second instructions, and between the second and third instructions to delay the respective reads relative to the writes by one clock cycle. The execution diagram shows that, at worst, this approach has writes and subsequent reads in the same clock cycle. This is indicated by the pairs consisting of a register write and a subsequent register read connected by a black arrow in the diagram. Because of the read-after-write assumption for the register file, the timing shown permits the program to be executed on correct operands.

This approach solves the problem, but what is the cost? First of all, the program is obviously longer, although it may be possible to place other, unrelated instructions in the NOP positions instead of just wasting them. Also, the program takes two clock cycles longer and reduces the throughput from 0.5 instruction per cycle to 0.375 instruction per cycle with the NOPs in place.



(a) The data hazard problem



(b) A program-based solution

FIGURE 10-16
Example of Data Hazard

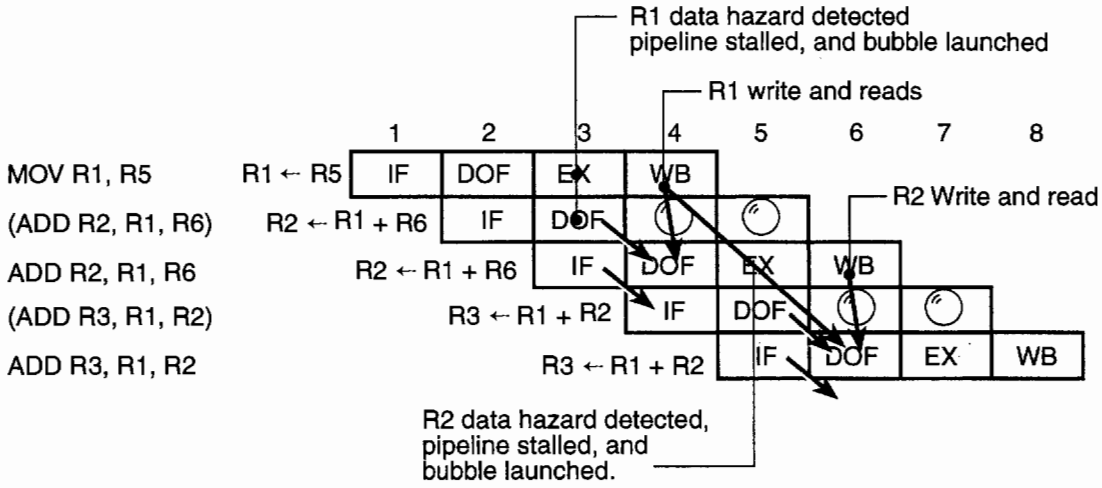


FIGURE 10-17
Example of Data Hazard Stall

Figure 10-17 illustrates an alternative solution involving added hardware. Instead of the programmer or compiler putting NOPs in the program, the hardware inserts the NOPs automatically. When an operand is found at the DOF stage that has not been written back yet, the associated execution and write-back are prevented, and the pipeline flow in IF and DOF is stalled for one clock cycle. Then the flow resumes with completion of the instruction, and a new instruction is fetched as usual. The delay of one cycle is enough to permit a result to be written before it is read as an operand.

When the actions associated with an instruction flowing through the pipe are prevented from happening at a given point, the pipeline is said to contain a *bubble* in subsequent clock cycles and stages for that instruction. In Figure 10-17, when the flow for the first ADD instruction is prevented beyond the DOF stage, in the next two clock cycles a bubble passes through the EX and the WB stages, respectively. The holding of the pipeline flow in the IF and DOF stages delays the microoperations taking place in these stages for one clock cycle. In the figure, this delay is represented by two diagonal blue arrows from the initial location in which the completion of the microoperation is prevented to the location one clock cycle later in which the microoperation is performed. When the pipeline flow is held in IF and DOF for an extra clock cycle, the pipeline is said to be *stalled*, and if the cause of the stall is a data hazard, then the stall is referred to as a *data hazard stall.*

An implementation of data hazard handling for the pipelined RISC that uses data hazard stalls is presented in Figure 10-18. The added or modified hardware is shown in the areas shaded in light blue. For this particular pipeline stage arrangement, a data hazard will occur for a register file read if there is a destination register at the execution stage that is to be written back in the next clock cycle and that is to be read at the current DOF stage as either of the two operands. So we have to determine whether such a register exists. This is done by evaluating the Boolean equations

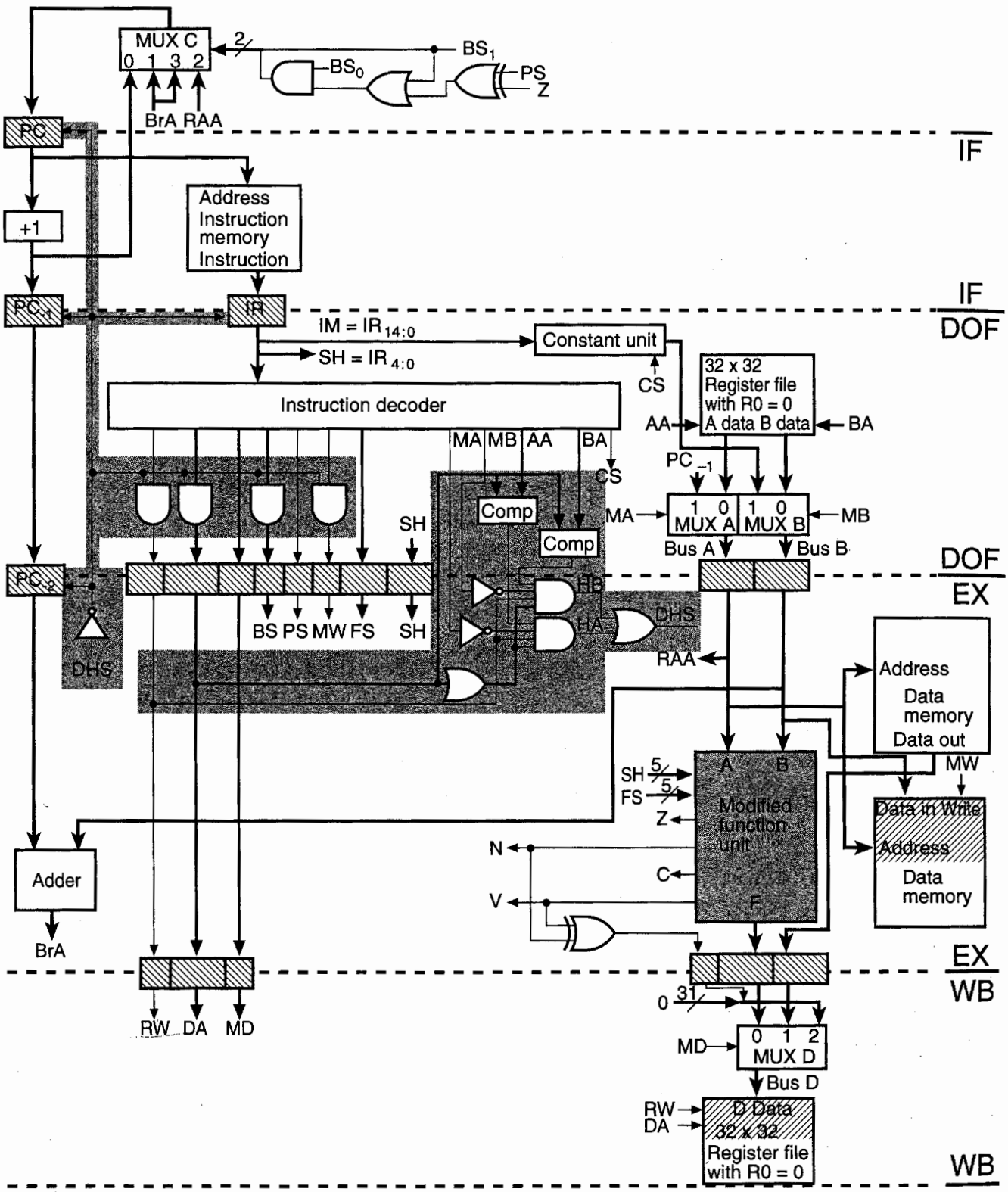


FIGURE 10-18
Pipelined RISC: Data Hazard Stall

$$HA = \overline{MA_{\text{DOF}}} \cdot (DA_{\text{EX}} = AA_{\text{DOF}}) \cdot RW_{\text{EX}} \cdot \sum_{i=0}^{4} (DA_{\text{EX}})_i$$

$$HB = \overline{MB_{\text{DOF}}} \cdot (DA_{\text{EX}} = BA_{\text{DOF}}) \cdot RW_{\text{EX}} \cdot \sum_{i=0}^{4} (DA_{\text{EX}})_i$$

and

$$DHS = HA + HB$$

The following events must all occur for *HA*, which represents a hazard for the A data, to equal 1:

1. *MA* in the DOF stage must be 0, meaning that the A operand is coming from the register file.
2. *AA* in the DOF stage equals *DA* in the EX stage, meaning that there is potentially a register being read in the DOF stage that is to be written in the next clock cycle.
3. *RW* in the EX stage is 1, meaning that register *DA* in the EX stage will definitely be written in WB during the next clock cycle.
4. The OR (Σ) of all bits of *DA* is 1, meaning that the register to be written is not *R*0 and so is a register that must be written before being read. (*R*0 has the same value 0 regardless of any writes to it.)

If all these conditions hold, there is a write pending for the next clock cycle to a register that is the same as one being read and used on Bus *A*. Thus, a data hazard exists for the *A* operand from the register file. *HB* represents the same combination of events for the *B* data. If either of the *HA* or *HB* terms equals 1, there is a data hazard and *DHS* is 1, meaning that a data hazard stall is required.

The logic implementing the above equations is shown in the shaded area in the center of Figure 10-18. The blocks marked "Comp" are equality comparators that have output 1 if and only if the two 5-bit inputs are equal. The OR gate with *DA* entering it ORs together the five bits of *DA* and has output 1 as long as *DA* is not 00000 (*R*0).

DHS is inverted and the inverted signal is used to initiate a bubble in the pipeline for the instruction currently in the *IR* as well as to stop the *PC* and *IR* from changing. The bubble, which prevents actions from occurring as the instruction passes through the EX and WB stages, is produced by using AND gates to force *RW* and *MW* to 0. These 0s prevent the instruction from writing the register file and the memory. AND gates also force *BS* to 0 causing the *PC* to be incremented instead of loaded during the EX stage for a jump register or branch instruction affected by a data hazard. Finally, to prevent the data stall from continuing for the next and subsequent clock cycles, AND gates force *DA* to 0 so that it appears that *R*0 is being written, giving a condition which does not cause a stall. The registers to remain unchanged in the stall are the *PC*, the PC_{-1}, PC_{-2}, and the *IR*. These registers are replaced with registers with load control signals driven by $\overline{DHS}$. When $\overline{DHS}$ goes

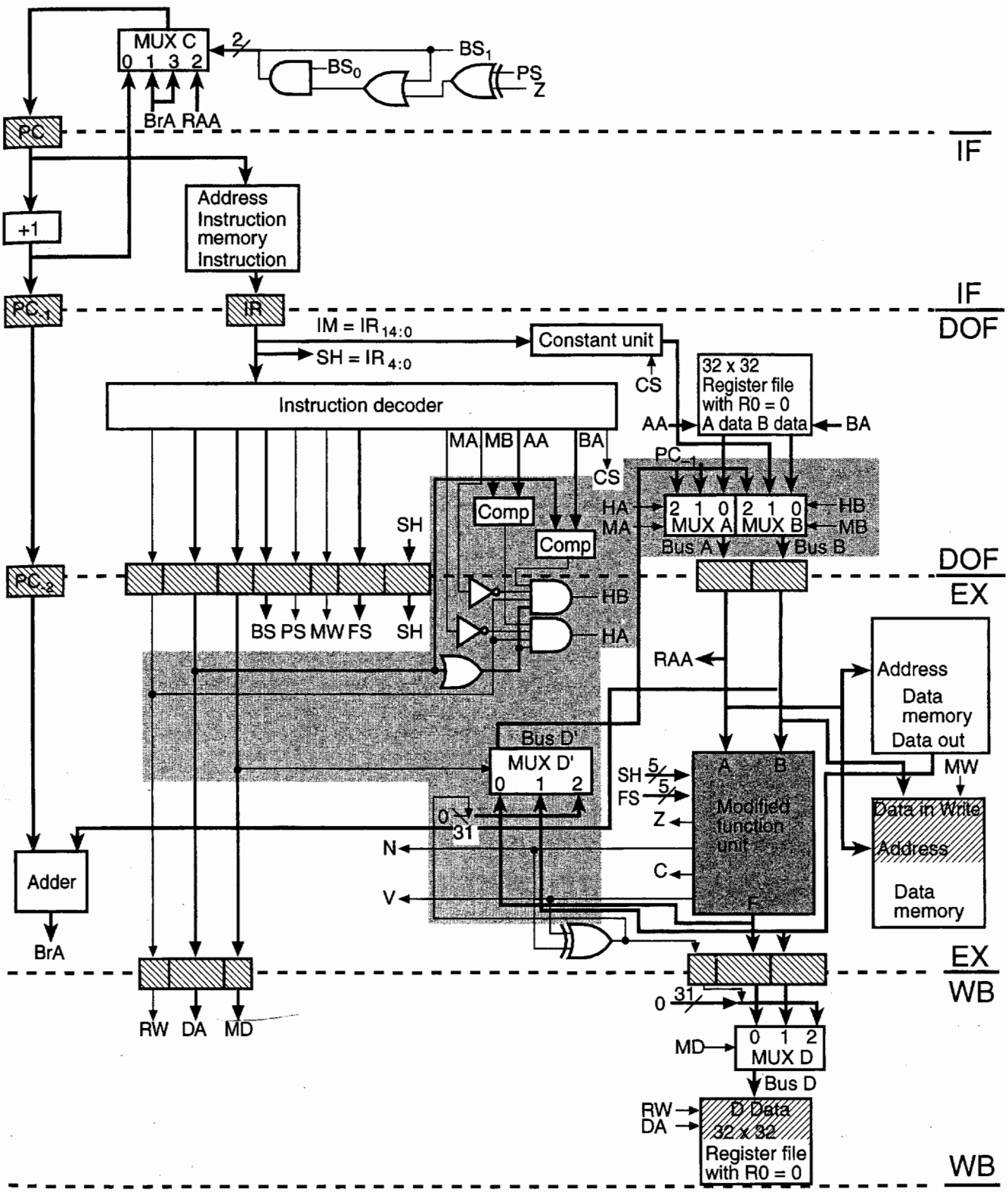
to 0, requesting a stall, the load signals become 0 and these pipeline platform registers hold their contents unchanged for the next clock cycle.

Returning to Figure 10-17, we see that in cycle 3 the data hazard for $R1$ is detected, so that $\overline{DHS}$ goes to 0 before the next clock edge. *RW*, *MW*, *BS*, and *DA* are set to 0, and at the clock edge, a bubble is launched into the EX stage for the ADD. At the same clock edge, the IF and DOF stages are stalled, so the information in them now is associated with clock cycle 4 instead of 3. In clock cycle 4, since DA_{EX} is 0, there is no stall, so the execution of the stalled ADD instruction proceeds. The same sequence of events occurs for the next ADD. Note that the execution diagram is identical to that in Figure 10-16(b), except that the NOPs are replaced by stalled instructions, shown in parentheses. Thus, although it removes the need for programming NOPs into the software, the data hazard stall solution has the same throughput penalty as the program with the NOPs.

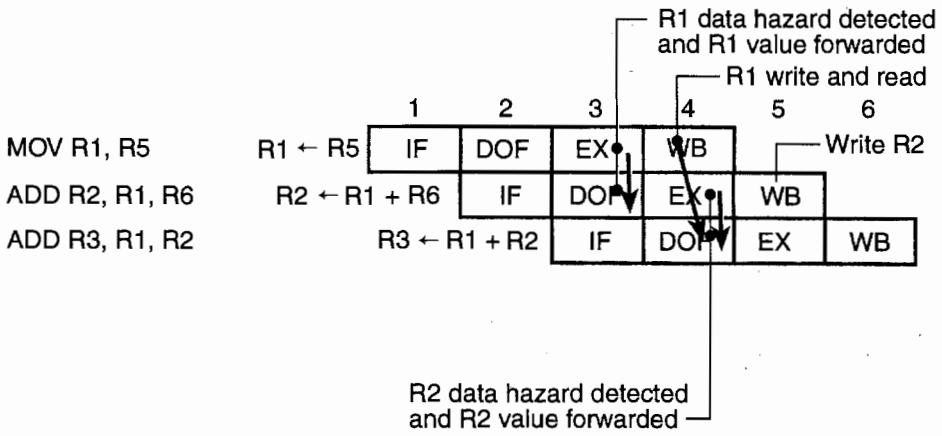
A second hardware solution, *data forwarding*, does not have this penalty. Data forwarding is based on the answer to the following question: When a data hazard is detected, is the result available somewhere else in the pipeline, so that it can be used immediately in the operation having the data hazard? The answer is "almost." The result will be on Bus D, but it is not available until the next clock cycle. The result is to be written into the destination register during that clock cycle. The information needed to form the result, however, is available on the inputs to the pipeline platform that provides the inputs to MUX D. All that is needed to form the result during the current clock cycle is a multiplexer to select from the three values, just as MUX D does. MUX D' is accordingly added to produce the result on Bus D'. In Figure 10-19, instead of reading the operand from the register file, we use data forwarding to replace the operand with the value on Bus D'. This replacement is implemented with an additional input to MUX A and to MUX B from Bus D' as shown. Essentially the same logic as before is used to detect the data hazard, except that the separate detection signals *HA* and *HB* are used directly for A data and B data, respectively, so that the replacement occurs for the operand that has the data hazard.

The data-forwarding execution diagram for the three-instruction example appears in Figure 10-20. The data hazard for $R1$ is detected in cycle 3. This causes the value to go into $R1$ in the next cycle, to be forwarded from the EX stage of the first instruction in cycle 3. The correct value of $R1$ enters the DOF/EX platform at the next clock edge so that execution of the first ADD can proceed normally. The data hazard for $R2$ is detected in cycle 4, and the correct value is forwarded from the EX stage of the second instruction in that cycle. This gives the correct value in the DOF/EX platform needed for the second ADD to proceed normally. In contrast to the data hazard stall method, data forwarding does not increase the number of clock cycles required to execute the program and hence does not affect the throughput in terms of the number of clock cycles required. It may, however, add combinational delay, causing the clock period to be somewhat longer.

Data hazards can also occur with memory access, as well as with register access. For the ST and LD instructions, it is not likely to be able to do a data memory read after a write in a single clock cycle. Further, some memory reads may take



□ FIGURE 10-19
Pipeline RISC: Data Forwarding



□ **FIGURE 10-20**
Example of Data Forwarding

more than one clock cycle, in contrast to what we have assumed here. Thus, the reduction in throughput for a data hazard may be increased due to a longer delay before the data is available. (See Chapter 12.)

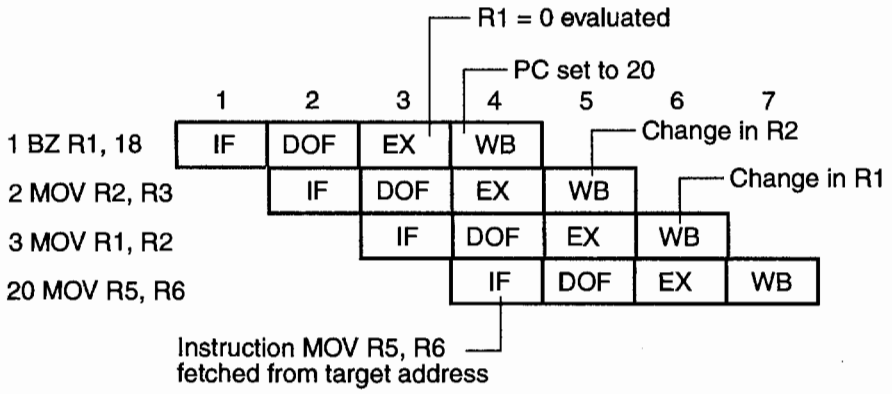
Control Hazards

Control hazards are associated with branches in the control flow of the program. The following program containing a conditional branch illustrates a control hazard:

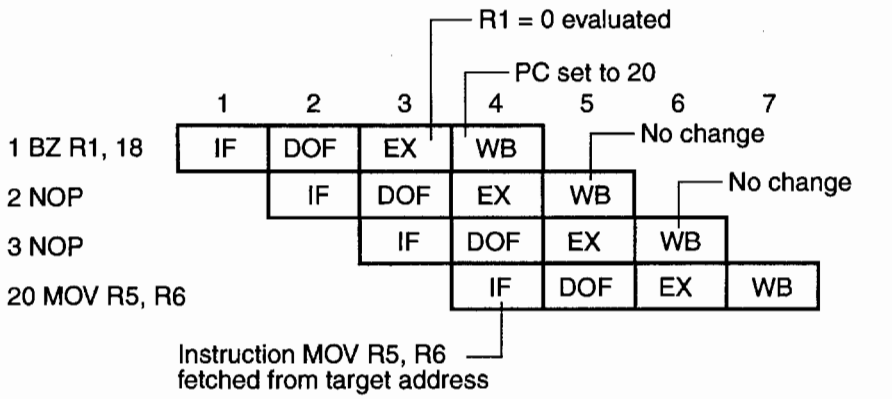
```
1    BZ     R1, 18
2    MOV    R2, R3
3    MOV    R1, R2
4    MOV    R4, R2
20   MOV    R5, R6
```

The execution diagram for this program is given in Figure 10-21(a). If *R*1 is zero, then a branch to the instruction in location 20 (recall that addressing is PC relative) is to occur, skipping the instructions in locations 2 and 3. If *R*1 is non-zero, then the instructions in locations 2 and 3 are to be executed in sequence. Assume that the branch is taken to location 20 because *R*1 is equal to zero. The fact that *R*1 equals 0 is not detected until EX in cycle 3 of the first instruction in Figure 10-21(a). So the *PC* is set to 20 on the clock edge at the end of cycle 3. But the MOV instructions in locations 2 and 3 are into the EX and DOF stages, respectively, after the clock edge. Thus, unless corrective action is taken, these instructions will complete execution, even though the programmer's intention was for them to be skipped. This situation is one form of a *control hazard.*

NOP instructions can be used to deal with control hazards just as they were used with data hazards. The insertion of NOPs is performed by the programmer or compiler generating the machine language program. The program must be written so that only operations intended to be performed, regardless of whether the branch is taken, are introduced into the pipeline before the branch execution actu-



(a) Branch Hazard Problem



(b) Program-based Solution

□ FIGURE 10-21
Example of Control Hazard

ally occurs. Figure 10-21(b) illustrates a modification of the simple three-line program that satisfies this condition. Two NOPs are inserted after the branch instruction BZ. These two NOPs can be performed regardless of whether the branch is taken in the EX stage of BZ in cycle 3 with no adverse effects on the correctness of the program. When control hazards in the CPU are handled in this manner by programming, the branch hazard dealt with by the NOPs is referred to as a *delayed branch*. Branch execution is delayed by two clock cycles in this CPU.

The NOP solution in Figure 10-21(b) increases the time required to process the simple program by two clock cycles, regardless of whether the branch is taken. Note, however, that these wasted cycles can sometimes be avoided by rearranging the order of instructions. Suppose that those instructions to be executed regardless of whether the branch is taken can be placed in the two locations following the branch instruction. In this situation, the lost throughput is completely recovered.

Just as in the case of the data hazard, a stall can be used to deal with the control hazard. But, also as in the case of the data hazard, the reduction in throughput

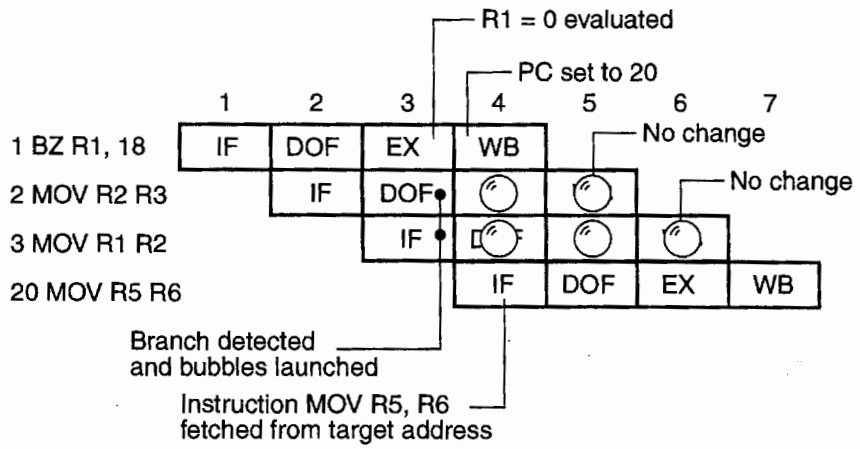


FIGURE 10-22
Example of Branch Prediction with Branch Taken

will be the same as with the insertion of NOPs. This solution is referred to as a *branch hazard stall* and will not be presented here.

A second hardware solution is to use *branch prediction*. In its simplest form, this method predicts that branches will never be taken. Thus, instructions will be fetched and decoded and operands fetched on the basis of the addition of 1 to the value of the *PC*. These actions occur until it is known during the execution cycle whether the branch in question will be taken. If the branch is not taken, the instructions already in the pipeline due to the prediction will be allowed to proceed. If the branch is taken, the instructions following the branch instruction need to be cancelled. Usually, the cancellation is done by inserting bubbles into the execution and write-back stages for these instructions. This is illustrated for the four-instruction program in Figure 10-22. Based on the prediction that the branch will not be taken, the two MOV instructions after BZ are fetched, and the first one is decoded and its operands fetched. These actions take place in cycles 2 and 3. In cycle 3, the condition upon which the branch is based has been evaluated, and it is found that $R1 = 0$. Thus, the branch is to be taken. At the end of cycle 3, the *PC* is set to 20, and the instruction fetch in cycle 4 is performed using the new value of the *PC*. In cycle 3, the fact that the branch is taken has been detected, and bubbles are inserted into the pipeline for instructions 2 and 3. Proceeding through the pipeline, these bubbles have the same effect as two NOP instructions. However, because the NOPs are not present in the program, there is no delay or performance penalty when the branch is not taken.

The branch prediction hardware is shown in Figure 10-23. Whether a branch is taken is determined by looking at the selection values on the inputs to MUX *C*. If the pair of inputs is 01, then a conditional branch is being taken. If the pair is 10, then an unconditional JMR is occurring. If the pair is 11, then an unconditional JMP or JML is taking place. On the other hand, if the pair of inputs is 00, then no branch is occurring. Thus, a branch occurs for all combinations other than 00—i.e., for at least one 1—on the pair of lines. Logically, this corresponds to the OR of the lines, as shown in the figure. The output of the OR is inverted and then ANDed

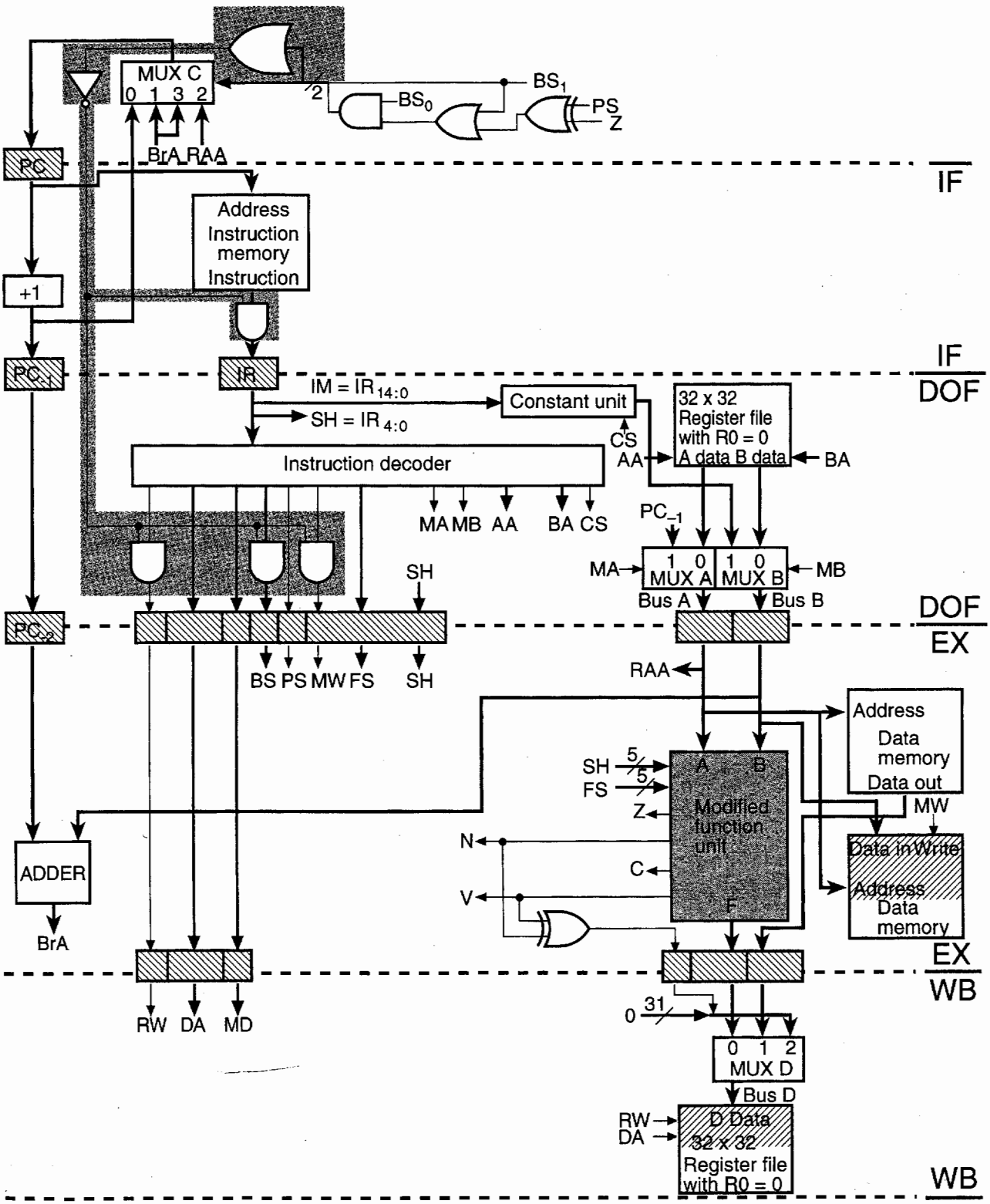


FIGURE 10-23
Pipelined RISC: Branch Prediction

with the *RW* and *MW* fields, so that the register file and the data memory cannot be written for the instruction following the branch instruction if the branch is taken. The inverted output is also ANDed with the *BS* field, so that a branch in the next instruction is not executed. In order to cancel the second instruction following the branch, the inverted OR output is ANDed with the *IR* output. This gives an instruction of all 0's, for which the OPCODE field is defined as NOP. If the branch is not taken, however, the inverted OR output is 1, and the *IR* and the three control fields remain unchanged, given normal execution of the two instructions following the branch.

Branch prediction can also be done on the assumption that the branch is taken. In this case, the instructions must be fetched and operands fetched down the path of the branch target. Thus, the branch target address must be computed and used for fetching the instruction in the branch target location. In case the branch does not take place, however, the updated value of the *PC* must also be saved. As a consequence, this solution will require additional hardware to compute and store the branch target address. Nevertheless, if branches are more likely to be taken than not, the "branch taken" prediction may yield a more favorable cost-performance trade-off than the "branch not taken" prediction.

For simplicity of presentation, we have treated the hardware solutions for dealing with hazards one at a time. In an actual CPU, these solutions need to be combined. In addition, other hazards, such as those associated with writing and reading memory locations, need to be handled.

10-4 More on Design

The two designs considered in this chapter represent two different instruction set architectures and two different supporting CPU organizations. The CISC architecture matches well with the microprogrammed control organization, and the RISC architecture matches well with the pipelined control organization. In this section, we will deal with some of the issues surrounding the two architectures and the two organizations. We begin by doing a very simple comparison of the performance of the two architectures and organizations. Then we cover a few advanced concepts that build upon the foundations established to achieve very high performance. Finally, we relate the two organizations to more general digital systems design.

CISC-RISC Comparison

We compare the CISC and RISC CPUs on the basis of the execution of a simple series of instructions, all of which perform simple register-to-register operations on a pair of registers. Our primary focus will be throughput, i.e., the number of instructions executed per interval of time. Initially, we perform calculations based on clock cycles: then we add a crude estimate of the length of a clock cycle to introduce time into the calculations.

In the CISC processor, a simple register-to-register operation requires two cycles for instruction fetch, three cycles for operand fetch, two cycles for execution,

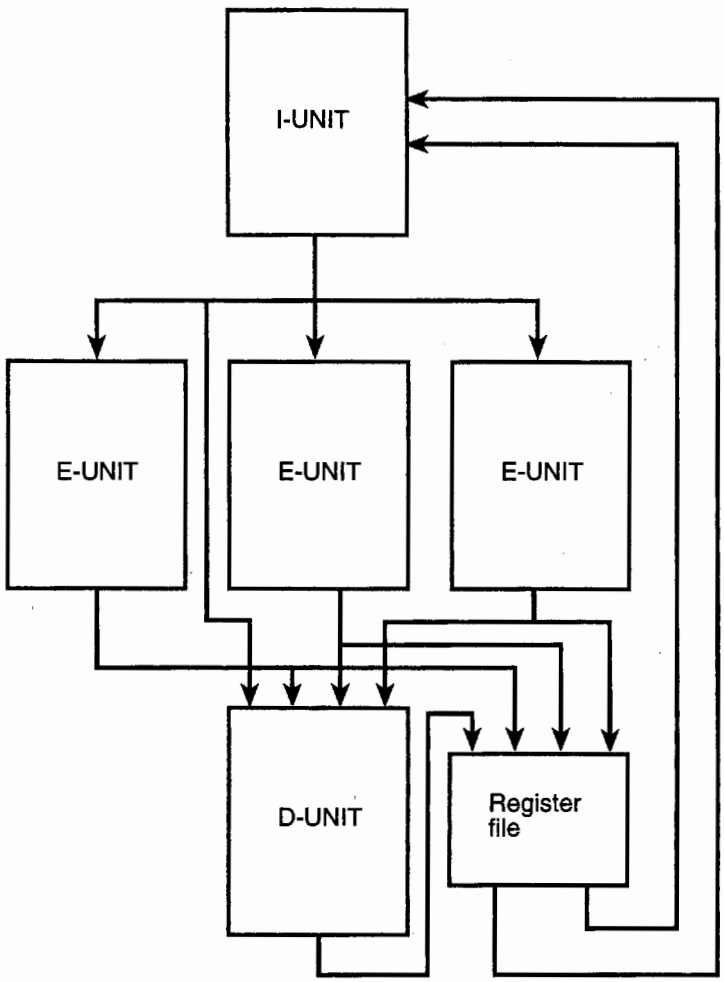
two cycles for write-back, and one cycle for interrupt handing, for a total of 10 clock cycles. Since there is only one instruction being processed at a time, the throughput is 1/10 = 0.1 instruction per clock cycle. Based on the component delay values used in Section 8-7 plus added control and multiplexer delays the clock cycle length is 20 ns. This gives an instruction execution time of 200 ns and a throughput of 5.0 million instructions per second (MIPS).

In the RISC processor, assuming no data hazard or control hazard stalls, a simple register-to-register operation takes four clock cycles. But four instructions are being processed at once, so the throughput is 4/4 = 1 instruction per clock cycle. Based on the values used in Section 8-7, the clock cycle is 5 ns. Thus, resulting throughput is 200 MIPS.

On the basis of this analysis, it would appear that the RISC implementation is 40 times faster than the CISC implementation. But a number of factors have been neglected in the analysis. For one thing, a typical CISC instruction is able to do considerably more than a RISC instruction. For example, try adding the contents of a register to an operand from memory accessed by using indirect indexed addressing with the result returned to memory with the two architectures. The CISC architecture does this with a single instruction: the RISC architecture requires at least six instructions. For these types of instructions, the MIPS-based throughput ratio is cut to from 40 to about 9.

Also, our microprogrammed implementation was not designed specifically with performance in mind. For instance, it is possible to move the multiway branches currently located at the first address of the some microroutines to the address of the unconditional ROM-based jump at the end of some microroutines that precede the execution of those branches. By eliminating the ROM-based jumps, the number of cycles required for the simple register-to-register operation is reduced from 10 to 7. It is also possible to make the microprogrammed datapath and control into a 2-stage pipelined structure. This is done routinely in microprogrammed control designs and would reduce the clock cycle to 14 ns, reducing the original throughput ratio from 40 to 20. If six RISC instructions are required to perform the equivalent of a CISC instruction as previously illustrated, the throughput ratio is further reduced to about 4.7, ignoring the effects of branching that occur in instruction decoding.

Still, although the performance of the two designs now appears to be closer together, we must realize that the simplistic methods we just used for comparison are, in the final analysis, invalid. The most optimistic CISC performance is based on an instruction that makes full use of the power of the CISC architecture. Recall, however, that in doing a simple register-to-register instruction, the performance of the CISC was inferior to that of the RISC. Also, hazards interfere with the performance of the RISC. Truly, the performance of both architectures depends on the sequence of instructions executed in typical programs. Thus, we need to know how the two architectures compare in executing significant, typical, real programs. Such programs are referred to as *benchmarks*. The use of benchmarks, while far from perfect as an evaluation method, gives a better picture of the comparative performance of the two architectures. Even then, note that the comparisons depend on the methods used to assemble or compile the programs,



□ **FIGURE 10-24**
Multiple Execution Unit Organization

since, unless the instruction set architectures are identical, the machine language code will be different.

In terms of benchmarking for general computation, pipeline-based implementations of RISC architectures often outperform comparable microprogram-controlled CISC implementations. Nevertheless, CISC architectures are very viable commercially because they avoid the cost of rewriting massive amounts of software developed for such architectures in the past. The inferiority in the performance of the CISC is partially remedied by using RISC-like pipelines within contemporary CISC implementations, as we illustrate later in this section.

High-Performance CPU Concepts

Among the various methods used to design high-speed CPUs are multiple units organized as a pipeline-parallel structure, microprogramming with pipelines, super-pipelines, and superscalar architectures.

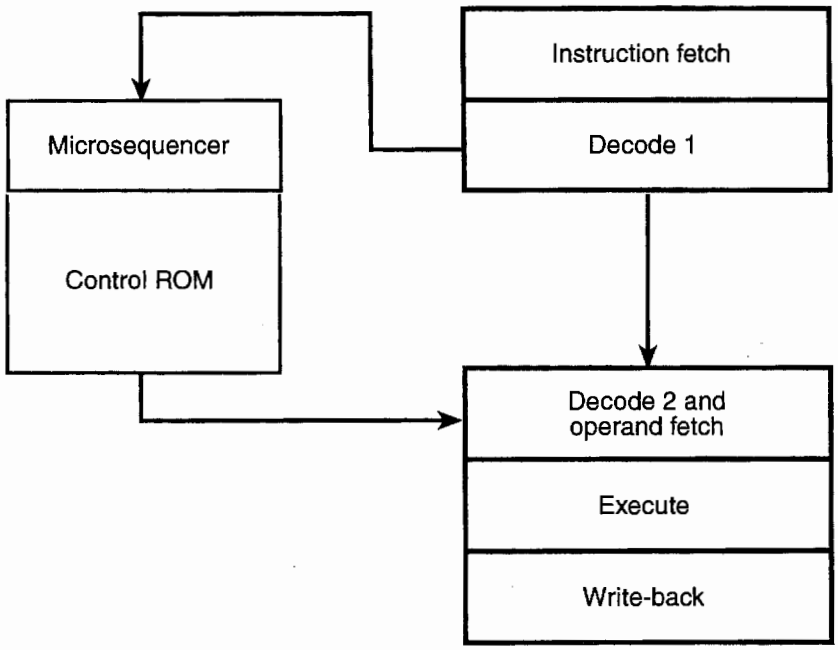
Consider the case in which an operation takes multiple clock cycles to execute, but the instruction fetch and write-back operations can be handled in a single cycle. Then it is possible to initiate an instruction every clock cycle, but not possible to complete the execution of an instruction every cycle. In such a situation, the performance of the CPU can be substantially improved by having multiple execution units in parallel. A high-level block diagram for this kind of system is shown in Figure 10-24. The instruction fetch, decoding, and operand fetch are carried out in the I-unit pipeline. In addition, the I-unit handles branches. When decoding of a non-branch instruction has been completed, the instruction and operands are *issued* to the appropriate E-unit. When execution of the instruction is completed by the E-unit, the write-back to the register file occurs. If a memory access is required, then the D-unit is used to execute the memory write. If the operation is a store, it goes immediately to the D-unit. Note that the actual execution units may be microprogrammed and may also involve internal pipelines.

Suppose that a sequence of three instructions—say, a multiplication, a 16-bit shift, and an addition—has no data hazards. Suppose further that there is a single pipelined E-unit that performs all of these operations, which take 17, 8, and 2 clock cycles, respectively, and that both the multiplication and the shift require multiple passes through portions of the E-unit pipeline. This situation allows only one clock cycle of overlap between pairs of the three instructions. Thus, the fastest that the sequence of operations executes in the E-unit is $17 + 8 + 2 - 2 = 25$ clock cycles. But with an E-unit for each operation, these operations can be executed in max $(17, 1 + 8, 2 + 2)$, which equals 17 clock cycles. The additional 1 and 2 are due to the issuing of one instruction per clock cycle to the E-unit set. The resulting execution throughput is improved by a factor of $25/17 = 1.5$.

Suppose that we must implement a CISC architecture, but we are interested in initiating and completing close to one instruction per short RISC clock cycle for simple, frequently used instructions. We will use a pipelined data path and a combination of pipelined and microprogrammed control, as shown in Figure 10-25. An instruction is fetched and enters the Decode 1 stage. If it is a simple RISC-like instruction that executes completely in the normal Execute stage of the pipeline, it is partially decoded and passed on to the Decode 2 and operand fetch stage. There, decoding is completed, and the instruction is sent on down the pipe. On the other hand, the instruction requires multiple operations, multiple memory accesses, or sequences thereof, the Decode 1 stage produces a microcode address for the ROM. Also, it causes the Decode 2 stage to take microinstructions from the ROM, rather than from the partially decoded instruction from the Decode 1 stage. Execution of microinstructions from the ROM continues until execution of the instruction is completed.

Recall that to execute microinstructions, it is often necessary to have temporary registers in which to store information. An organization of this type will frequently supply temporary registers with a convenient mechanism for switching between temporary registers and the usual programmer-accessible register file.

The preceding organization supports an architecture that has combined CISC-RISC properties. It illustrate that pipelines and microprograms can be compatible and need not be viewed as mutually exclusive. The combined architecture



□ **FIGURE 10-25**
Combined CISC-RISC Organization

allows new software to take advantage of a RISC architecture while preserving compatibility with older software dependent upon a CISC architecture.

In all of the methods considered thus far, the peak throughput possible is one instruction per clock cycle. With this limitation, it is desirable to maximize the clock rate by minimizing the maximum pipeline stage delay. If, as a consequence, a large number of pipeline stages is used, the CPU is said to be *superpipelined*. A superpipelined CPU will generally have a very high clock frequency, in the range of a few hundred or more MHz. In such an organization, however, handling hazards effectively is critical, since any stalling or reinitialization of the pipeline will degrade the performance of the CPU significantly. Also, as more pipeline platforms are added, further dividing up the combinational logic, the setup and propagation delay times of the flip-flops begin to dominate the platform-to-platform delay and the speed of the clock. The improvement achieved is less, and when hazards are taken into account, the performance may actually become worse rather than better.

For fast execution, an alternative to superpipelining is the use of a *superscalar* organization. The goal of this kind of organization is to have a peak rate of initiating instructions in excess of one instruction per clock cycle. A superscalar CPU that fetches a pair of instructions simultaneously by using a double-word wide path from instruction memory is illustrated in Figure 10-26. The processor checks for hazards among the instructions, as well as available execution units in the instruction issue stage of the pipeline. If there are hazards or busy execution units corresponding to the first instruction, then both instructions are held for later issuing. If the first instruction has no hazard and its E-unit is available, but there is a hazard or no available E-unit for the second instruction, then only the first instruction is issued. Other-

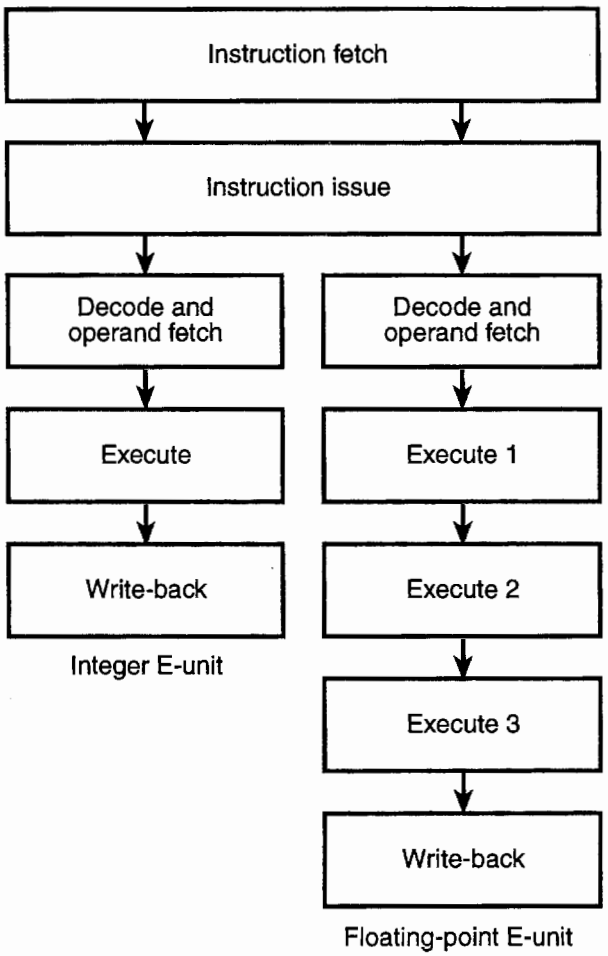


FIGURE 10-26
Superscalar Organization

wise, both instructions are issued in parallel. If a given superscalar architecture has the ability to issue up to four instructions simultaneously, then its peak execution rate is four instructions per clock cycle. If the clock cycle is 5 ns, then such a CPU has a peak execution rate of 800 MIPS. Note that the checks made for hazards between instructions in the execution stages and those in the issue stage become very complex as the maximum number of instructions issued simultaneously is increased. The resulting hardware complexity has the potential for increasing the clock cycle length, so the trade-offs in such a design need to be examined very carefully.

We close this section with two observations. First, as the quest for better performance causes us to design increasingly complex organizations, hazards cause the order of the instructions to play a more important role in the throughput that is achievable. Also, improved performance can be achieved by reducing the number of hazard-producing instructions, such as branches. As a consequence, to fully exploit the performance capabilities of the hardware, the assembly language programmer and the compiler writer need to be very knowledgeable about the behavior of not only the instruction set architecture, but also the underlying organization of the hardware of the CPU.

When multiple execution units are involved, very often the CPU design we have been considering here actually becomes the design for the entire processor, as is shown for the generic computer. This is apparent in the superscalar organization in Figure 10-26, which contains the floating-point unit (FPU). The FPU, the MMU, and the portion of the internal cache that handles data are effectively types of E-units. The portion of the internal cache that handles the instructions can be viewed as a part of the I-Unit that fetches instructions. Thus, in the quest for higher and higher throughput, the realm of the CPU becomes that of the processor, as in the generic computer. With the ever-growing complexity of integrated circuits, the processor is encompassing more and more of the electronic components of a computer.

Recent Architectural Innovations

Beyond the concepts presented in the previous section, more recently, two general trends have become apparent in the most recent high-performance architectures. The first trend is the development of compilers and hardware architectures that permit the compiler to explicitly identify to the hardware instructions that can be executed in parallel. In this approach, the identification of parallelism typically done in hardware in the superscalar architecture has now been moved to a fair degree into the compiler. This releases hardware for other uses, notable more execution units and larger register files. The second trend is the use of techniques that allow the processor to avoid waiting for branches to be taken and for data values to become available. Three techniques that support this trend will be discussed in the remainder of this section.

Instead of waiting for a branch to be taken, the processor will execute both sides of the branch and produce results for both sides. When the results of the branch becomes available, the right result is selected and the computation proceeds. Thus there is no delay waiting for a branch, significantly improving performance for long pipelines. This approach is referred to as *predication* and uses special 1-bit registers referred to as predicate registers that determine which result is used when the branch outcome is known.

Instead of waiting to load data from memory until it is known that it is needed, *speculative loading* of data from memory is performed before it is known for sure whether or not the data is needed. The reason for use of this technique is to avoid the relatively long delay required to fetch an operand from memory. If the data which is speculatively fetched turns out to be the data needed, then the data will be available and the computation can proceed immediately without having to wait for a memory access to get the data.

Instead of waiting for data to become available, *data speculation* uses methods to predict data values and proceeds to compute using these values. When the actual value becomes known and matches the predicted value, then the result produced from the predicted value can be used to carry forward the computation. If the actual value and the predicted value differ, then the result based on the predicted value is discarded and the actual value is used to continue computation. An example of data speculation is permitting a value to be loaded from memory before a store into the same memory location occurring earlier in the program has

been executed. In this case, it is predicted that the store will not change the value of the data in memory so that the value loaded before the store will be valid. If, at the time the store occurs, the loaded value is not valid, the result of computation using it is discarded.

All of these techniques perform operations or sequences of operations for which results are discarded with some frequency. Thus, there is "wasted" computation. To be able to do large amounts of useful computation as well as the wasted computation, more parallel resources as well as specialized hardware for implementing the techniques are required. The payoff in return for the cost of these resources is higher performance.

Digital Systems

The two sizable digital system designs we have examined in this chapter are general-purpose CPUs. How does their design relate to that of other digital systems? First of all, each digital system has an architecture. Although that architecture may not in any way deal with instructions to be executed, it is likely that it still can be described by using register transfer descriptions and, possibly, one or more algorithmic state machines. On the other hand, it might have instructions, but they may be quite different from those for a CPU. The system may have no datapath at all or may have several datapaths. There is likely to be some form of control unit, and there may be multiple control units that interact. The system may or may not include memories. Thus, the total spectrum of digital systems has a very wide range of architectural possibilities.

So what is the connection of the general digital system to the content of this chapter? Simply stated, the connection is design techniques. To illustrate, consider that we have shown in detail how a system with instructions can be implemented using a datapath and a control unit. From here, it is relatively easy to implement a simpler, instructionless system. We have shown how an instruction can be easily decoded using multiplexers and a ROM. This technique can be applied to other digital systems having instructions. We have also shown how microprogramming has been used to implement controls for complex functions. If a system has one or more very complex functions, whether programmable or not, then a microprogrammed control is a possibility. Finally, we have shown how high speeds can be achieved by using pipelines or parallel execution units. Thus, if the goal of a system is high speed, then pipelining or parallel units are techniques to consider.

10-5 Chapter Summary

In this chapter, we examined two CPU designs: a CISC with a conventional datapath and microprogrammed control unit and a RISC with a pipelined datapath and control unit. The instruction set of the CISC uses many operations, with memory access supported by eight addressing modes. The CISC also has many operations that are complex in the sense that they require many clock cycles for their execution. The CISC uses multiple instruction lengths. The RISC, in contrast, has its memory access restricted to load and store operations

and has only four addressing modes, most of which are restricted to specific operations. RISC operations are all simple in the sense that only *one* clock cycle is required for their execution. The RISC employs instructions of a single length.

The non-pipelined datapath for the CISC CPU employs a versatile shifter and status bits and uses modified register file addressing to provide temporary storage registers. In addition, it requires an added multiplexer capability in datapath buses. The RISC datapath is widened to 32 bits and has an enlarged register file. A barrel shifter provides multiple position shifts in a single clock cycle, and miscellaneous added features supply instruction set–specific capabilities.

The CISC control unit includes a stack pointer in addition to the program counter. Control microprograms reside in ROM, and a combination of a multiplexer and a ROM provides fast instruction decoding. The control unit also has extensive jump and conditional branching capabilities, including one level of microsubroutines. The microprogram for the control is modularized to permit many microsubroutines to be shared in implementing the microprogram for the instructions.

The RISC control unit is pipelined and has special hardware added to deal with branches. Pipelined CPUs have both data and control hazard problems. We examined one of each type of hazard, as well as software and hardware solutions for each.

After discussing CISC and RISC performance, we touched on some advanced concepts, including parallel execution units, a combination of microprogrammed control with a pipeline, superpipelined CPUs, superscalar CPUs, and predictive and speculative techniques for high-performance. Finally, we related the design techniques in this chapter to more general digital system design.

REFERENCES

1. DIETMEYER, D. L., *Logic Design of Digital Systems,* 3rd ed. Boston, MA: Allyn-Bacon, 1988.
2. MANO, M. M. *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice Hall, 1988.
3. HAMACHER, V. C., VRANESIC, Z. G., AND ZAKY, S. G. *Computer Organization,* 3rd ed. New York, NY: McGraw-Hill, 1990.
4. HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach,* 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.
5. KANE, G., AND HEINRICH, J. *MIPS RISC Architecture.* Englewood Cliffs, NJ: Prentice Hall, 1992.
6. SPARC INTERNATIONAL, INC. *The SPARC Architecture Manual: Version 8.* Englewood Cliffs, NJ: Prentice Hall, 1992.
7. MANO, M. M. *Computer System Architecture,* 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.

8. PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface.* San Mateo, CA: Morgan Kaufmann, 1994.
9. WEISS, S., AND SMITH, J. E. *POWER and PowerPC.* San Mateo, CA: Morgan Kaufmann, 1994.
10. WYANT, G., AND HAMMERSTROM, T. *How Microprocessors Work.* Emeryville, CA: Ziff-Davis Press, 1994.

PROBLEMS

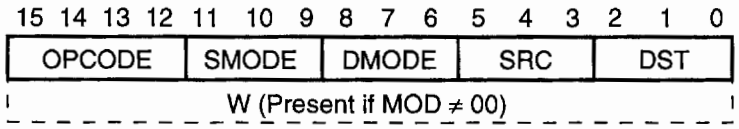
The plus (+) indicates a more advanced problem and the asterisk (*) indicates a solution is available in the Prentice Hall Companion Website Gallery.

10–1. *Write a register transfer statement that describes the operation performed by each of the following instructions for the CISC design by using information from Table 10-1. In those cases in which a second word W is part of the instruction, it is given. Use hexadecimal integers for representing addresses and operands.

(a) 0001 0100 0000 0000
(b) 0100 1001 0000 0110 0000 1111 0000 1111
(c) 1000 0110 0101 0011 0000 0000 1111 1111
(d) 1100 0111 1000 0000 0000 0000 1111 0000

10–2. In the CISC formats in Figure 10-2, the format with $IR(15{:}14) = 11$ has bits $IR(3)$ through $IR(6)$ unused. Currently, this format is able to handle 16 operations. How many operations could be supported if the unused bits were added to the OPCODE? What modifications would be required to the instruction decoder in the CISC design to support this change?

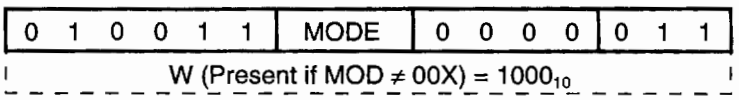
10–3. +Suppose that the CISC instruction format in Figure 10-2 is changed to the following:



SMODE is a mode field for the source operand, and DMODE is a mode field for the destination. Due to the presence of these fields, the S field is no longer necessary.

(a) Based on this new format, how many operations can be specified using OPCODE alone?
(b) How many operations can be specified by: (1) using 000X as the OPCODE specifier for zero-address instruction, with vacant bits $IR(11)$ through $IR(0)$ used as additional OPCODE bits, (2) using 01XX as the OPCODE specifier for one-address instructions, with vacant bits $IR(11)$ through $IR(9)$ as additional OPCODE bits, (3) using 1XXX as the OPCODE specifier for two-address operations, and (4) using 001X as the OPCODE specifier for branch operations, with vacant bits $IR(11)$ through $IR(9)$ as additional OPCODE bits?

10–4. *The following instruction in location 2001_{10} is executed for each of the eight addressing modes in Table 10-2.



Assuming R_i contains i_{10} and M_i contains i_{10}, give the eight effective addresses that result.

10–5. Using the CISC instruction set, write an efficient assembly language program to add the contents of:
(a) registers $R1$ through $R6$, with the result placed in $R7$.
(b) memory locations 100_{10} through 120_{10}, with the result placed in 125_{10}.
In both cases, all register contents are to remain unchanged, except for those of $R7$.

10–6. *For each of the CISC shift operations given, shifter input A equals $F0C6_{16}$, and C equals 1. Using the shifter in Figure 10-5, find shifter output H in hexadecimal for each of the following shift operations
(a) Logical shift left (lsl)
(b) Rotate right with carry (rorc)
(c) Arithmetic shift right (asr)
(d) Rotate left (rol)

10–7. +Design the *PSR* and *MSTS* hardware for the CISC design based on Figure 10-7, Table 10-1, and Table 10-7. Use D flip-flops, gates, and a ROM or PLA. Note carefully all sources that may modify the carry bit.

10–8. Draw an ASM chart for each of the following:
(a) the logical shift right microcode given in Table 10-14.
(b) the microcode for a rotate right with carry operation.

10–9. Write microcode for each of the following two-operand addressing modes with S = 0. Give both a register transfer description and a hexadecimal representation for binary code for each microinstruction as in Table 10-14.
(a) Immediate
(b) Relative
(c) Register indirect
(d) Relative indirect

10–10. *Write microcode for the execution part of each of the following instructions. Give both a register transfer description and a hexadecimal representation for binary code for each microinstruction as in Table 10-14.
(a) Arithmetic shift right
(b) Rotate left with carry
(c) Branch if overflow

10–11. Write microcode for the execution part of each of the following instructions. Give both a register transfer description and the hexadecimal representation for the binary code for each microinstruction as in Table 10-14.
(a) Negate
(b) Exclusive-OR

(c) Branch if no carry
(d) Subtract with borrow

10–12. *Write a microprogram for the execution part of the multiply operation that uses the add and shift right algorithm. Assume that all operands are unsigned integers and that you are to provide both register transfer statements and hexadecimal microcode as in Table 10-14. The multiplicand is in the destination location, and the multiplier is in the source location. After the multiplication is complete, the least significant half of the result is in the source location, and the most significant half is in the destination location. (*Hint:* You will find the rotate right with carry microoperation particularly useful.) You need not provide a write-back routine.

10–13. +Write a microprogram for the divide operation. Assume that all operands are unsigned integers and that you are to provide both register transfer statements and hexadecimal microcode as in Table 10-14. The 16-bit dividend is in the destination location, and the divisor is in the source location. After the division is complete, the quotient is in the source location, and the remainder is in the destination location. You need not provide a write-back routine.

10–14. Write a microprogram for the move string operation. This operation is to copy a string of words of length L in memory locations A through $A + L - 1$ into memory locations B through $B + L - 1$. The three integers A, B, and L are stored on the top of the stack, with A the topmost element. Provide register transfer statements and hexadecimal microcode as in Table 10-14.

10–15. For each of the RISC operations in Table 10-20, list the addressing mode or modes used.

10–16. Simulate the operation of the barrel shifter in Figure 10-14 for each of the following shifts and $A = 7E93C2A1_{16}$. List the hexadecimal values on the 47 lines, 35 lines, and 32 lines out of the three levels of the shifter.
(a) Left, $SH = 12$
(b) Right, $SH = 15$
(c) Left, $SH = 29$

10–17. *For the RISC CPU in Figure 10-15, simulate, in hexadecimal, the processing of the instruction ADI R1 R16 2F01 located in $PC = 10F$. Assume that $R16$ contains 0000001F. Show the contents of each of the pipeline platforms and of the register file (the latter only when a change occurs) for each of the clock cycles.

10–18. Repeat Problem 10–17 for the instruction SLT R31 R10 R16 with $R10$ containing 0000100F and R16 containing 00001022.

10–19. Repeat Problem 10–17 for the instruction LSL R1 R16 000F.

10–20. +Use a computer-based logic minimization program to design the instruction decoder for a RISC from Table 10-20. The field FS need not be done, since it can be wired directly from OPCODE.

10–21. *For the RISC design, draw the execution diagram for the following RISC program, and indicate any data hazards that are present:

```
1 MOV     R7, R6
2 SUB     R8, R8, R6
3 AND     R8, R8, R7
```

10–22. For the RISC design, draw the execution diagram for the following RISC program (with the contents of *R7* nonzero after the subtraction), and indicate any data or control hazards that are present:

```
1 SUB     R7, R7, R6
2 BNZ     R7, 000F
3 AND     R8, R7
4 OR      R5, R7
```

10–23. *Rewrite the RISC programs in Problem 10–21 and Problem 10–22 using NOPs to avoid all data and control hazards.

10–24. Draw the execution diagrams for the program in Problem 10–21, assuming
(a) the RISC CPU with data stall given in Figure 10-18.
(b) the RISC CPU with data forwarding in Figure 10-19.

10–25. Simulate the processing of the program in Problem 10–21 using the RISC CPU with data hazard stall in Figure 10-18. Give the contents of each pipeline platform and the register file (the latter only whenever a change occurs) for each clock cycle. Initially, *R6* contains 00000020_{16}, *R7* contains 00000030_{16}, *R8* contains 00000040_{16}, and the *PC* contains 00000001_{16}. Is the data hazard avoided?

10–26. Repeat Problem 10–25 using the RISC CPU with data forwarding in Figure 10-19.

10–27. Draw the execution diagram for the program in Problem 10–22, assuming the combination of the RISC CPU with branch prediction in Figure 10-23 and the RISC CPU with data forwarding in Figure 10-19.

10–28. *Assuming that the CISC CPU clock cycle is three times that of the RISC CPU clock cycle,
(a) Write a RISC program for the RISC CPU with data forwarding that will execute a CISC DEC operation with the destination addressed using index register-indirect mode.
(b) Compare the processing time for the RISC program with the processing time for the CISC instruction. Include instruction fetch in both calculations.

10–29. +Assuming that the CISC CPU clock cycle is three times that of the RISC CPU clock cycle,
(a) Write the microcode for a two-operand fetch for the relative indirect mode.
(b) Write a RISC program for the RISC CPU with data forwarding that will execute a CISC XOR operation with the destination operand in a register and the source operand addressed using relative indirect mode.
(c) Compare the processing time for the RISC program with the processing time for the CISC ADD instruction. Include instruction fetch in both calculations.