

Supplement to

Logic and Computer  
Design Fundamentals  
4th Edition<sup>1</sup>

# TRADITIONAL CISC DESIGN

**S** elected topics not covered in the fourth edition of *Logic and Computer Design Fundamentals* are provided here for optional coverage and for self-study. This material fits well with the desired coverage in some programs but not may not fit within others due to time constraints or local preferences. This supplement consists of the CISC processor material from Chapter 10 of the 2nd edition of *Logic and Computer Design Fundamentals*. The use of this material is not recommended except as an example of microprogramming applied to a non-pipelined system. Note that the processor described is incomplete, has some architectural inconsistencies, and does not represent current processor microarchitectures.

## Instruction Set Architecture

Figure 1 shows the CISC register set accessible to the programmer. All registers have 16 bits. The register file has eight registers, *R0* through *R7*. *R0* 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.

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 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 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*,

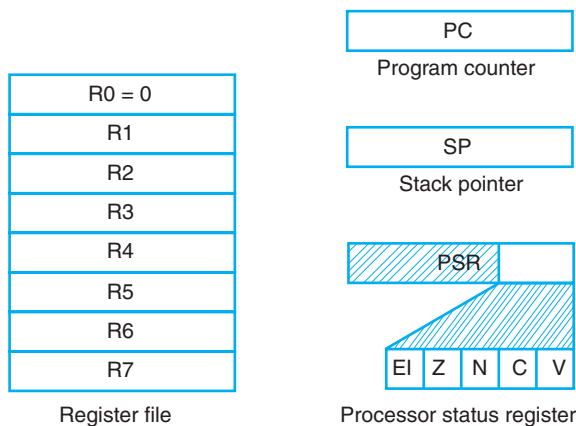
<sup>1</sup>© Pearson Education 1997, 2000, 2004,2008. All rights reserved.

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 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 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 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

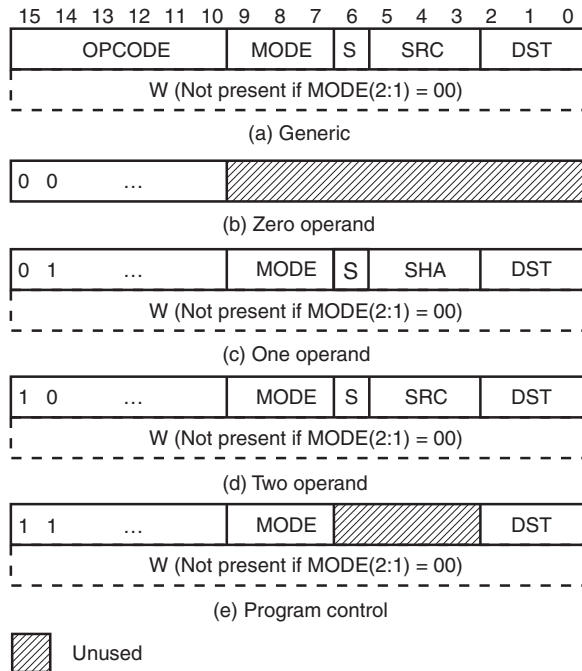


**FIGURE 1**  
CPU Register Set Diagram for CISC

**TABLE 1**  
**CISC Instruction Operations**

**TABLE 1**

Instruction	Mnemonic	Opcode	Status Effect	Instruction	Mnemonic	Opcode	Status Effect
Zero-operand Instructions							
No operation	NOP	000000	None	Move	MOVE	100000	None
Push registers	PSHR	000001	None	Exchange	XCH	100001	None
Pop registers	POPR	000010	None	Add	ADD	100010	ZCNV
Move string	MVS	000011	None	Add with carry	ADDC	100011	ZCNV
Return from procedure	RET	000100	None	Subtract	SUB	100100	ZCNV
Return from interrupt	RTI	000101	From stack	Subtract with borrow	SUBB	100101	ZCNV
Invalid	000110 through 001111			Multiply	MUL	100110	ZCNV
One-operand Instructions							
Push	PUSH	010000	None	Divide	DIV	100111	ZCNV
Pop	POP	010001	None	Compare	CMP	101000	ZCNV
Increment	INC	010010	ZCNV	AND	AND	101001	ZN
Decrement	DEC	010011	ZCNV	OR	OR	101010	ZN
Negate	NEG	010100	ZCNV	Exclusive-OR	XOR	101011	ZN
Complement	COM	010101	ZN	Invalid	101100 through 101111		
Logical shift right	SHR	011000	ZC	Branch Instructions			
Logical shift left	SHL	011001	ZC	Jump	JMP	110000	None
Arithmetic shift right	SHRA	011010	ZCNV	Call procedure	CALL	110001	None
Arithmetic shift left	SHLA	011011	ZCNV	Branch if zero	BZ	111000	None
Rotate right	ROR	011100	ZC	Branch if no zero	BNZ	111001	None
Rotate left	ROL	011101	ZC	Branch if carry	BC	111010	None
Rotate right with carry	RORC	011110	ZC	Branch if no carry	BNC	111011	None
Rotate left with carry	ROLC	011111	ZC	Branch if negative	BN	111100	None
Invalid	010110 through 010111			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 2**  
Instruction Formats

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.

If  $IR(15:14) = 10$ , 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 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.

**TABLE 2**  
**Addressing Modes**

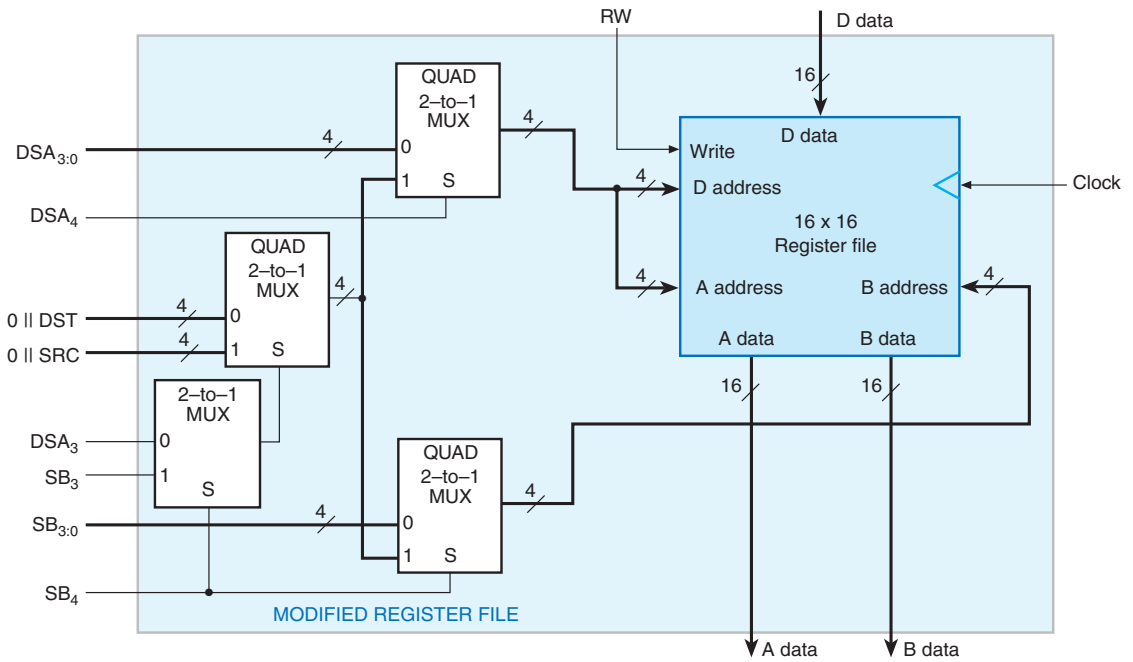
MODE	Address Mode	Register Transfer Description of Operands		
		IR(15:14) 5 01 or 11	IR(15:14) 5 10, S 5 0	IR(15:14) 5 10, S 5 1
000	Register	$R[DST]$	$R[SRC], R[DST]$	$R[DST], R[SRC]$
001	Register Indirect	$M[R[DST]]$	$M[R[SRC]], R[DST]$	$M[R[DST]], R[SRC]$
010	Immediate	$W$	$W, R[DST]$	$W, R[SRC]$
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	$M[M[PC + W]]$	$M[M[PC + W], R[DST]$	$M[M[PC + W], R[SRC]$

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 11 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.

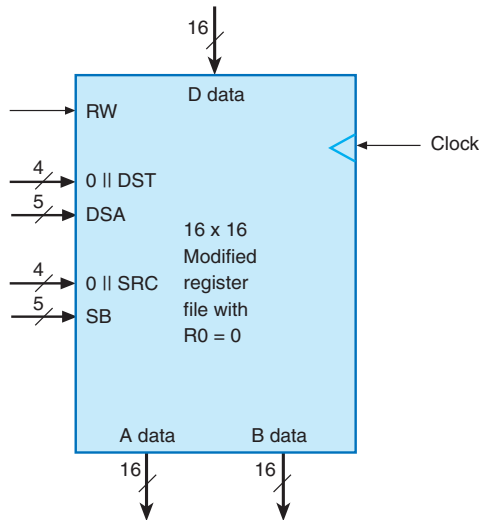
### Datapath Organization

The datapath to be designed is given in Figure 6. In the microprogrammed architecture, there are complex instructions spanning many clock cycles and performing complicated operations. Thus, temporary storage is needed for use by the microprograms. To meet this need, we expand the register file to 16 registers. The first 8 registers,  $R0$  through  $R7$ , are visible to the computer programmer. The second eight registers,  $R8$  through  $R15$ , are used as temporary storage for microprogram operands and are hidden from the programmer. Figure 3 provides a map of the expanded register file with the temporary registers shaded. As indicated previously, register  $R0$  supplies the constant 0, registers  $R1$  through  $R7$  are available to the programmer for use, and registers  $R8$  through  $R15$  provide general temporary storage for use by microprograms. The last four registers,  $R12$  through  $R15$ , 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,  $R12$  is the location for the source address ( $SA$ ),  $R13$  for the source data ( $SD$ ),  $R14$  for the destination address ( $DA$ ), and  $R15$  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 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 3**  
Modifications to Register File

0	R0 = 0
1	R1
2	R2
3	R3
4	R4
5	R5
6	R6
7	R7
8	R8
9	R9
10	R10
11	R11
12	Source Address SA
13	Source Data SD
14	Destination Address DA
15	Destination Data DD

□ **FIGURE 4**  
Register File Map

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 R0 through R7. 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 R0 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 4(b).

We find that, based on the eight shift instructions provided, the shifter from Figure 10-8 of the text needs to be modified. The modifications involve the end bits

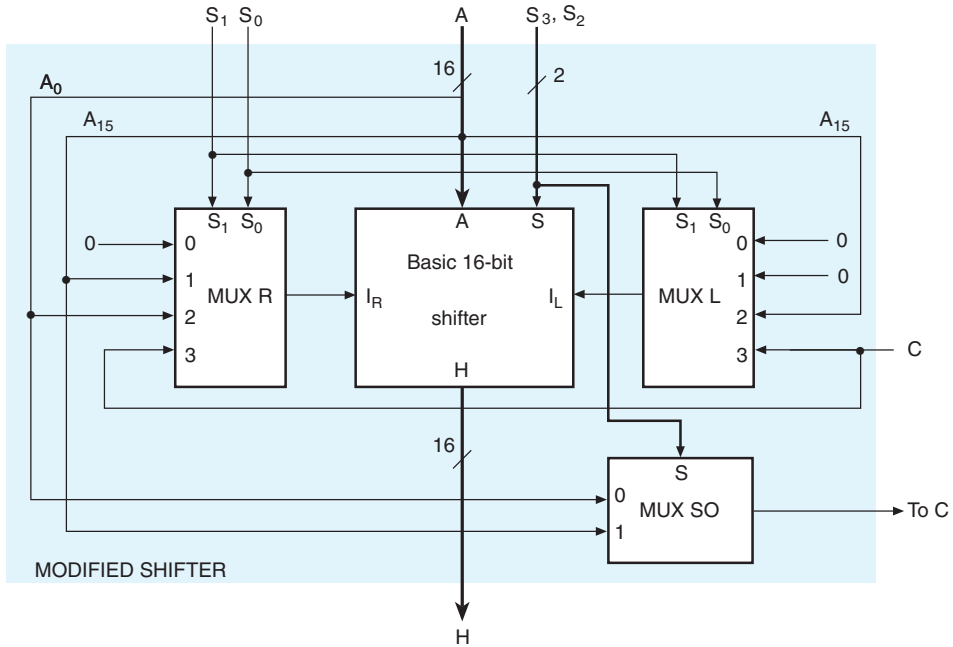
of the shift logic. For logical shifts, a 0 is inserted, as before. For the right arithmetic shift, the sign bit is the incoming bit, and for the left arithmetic shift, 0 is the incoming bit. Rotates require that the bit from the opposite end of the shifter be fed around. Finally, rotates with carry 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 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 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.

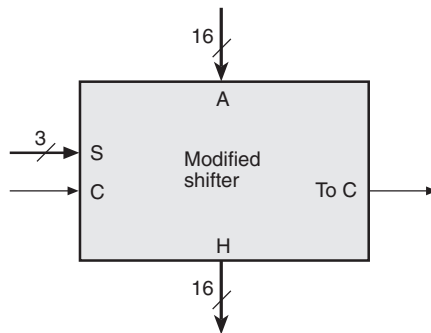
The instruction set requires the ability to store the contents of the *PC* in memory and load it back into the *PC*. Likewise, the contents of the *PSR* must be stored to memory and loaded back. Also, it is necessary to be able to store and load the *SP*. So, for these three registers, paths are established 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 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 design 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 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 11-8 of the text. 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 (1). The logic to selectively control the loading of the status bits in accordance with Table 1 is represented 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  $Z^1$ ,  $N^1$ ,  $C^1$ , and  $V^1$ . 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 6. The four control bits labeled MO (miscellaneous operations) are decoded to control the sets of status bits to be enabled for loading.  $C_i$  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*.

The new datapath is shown in Figure 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



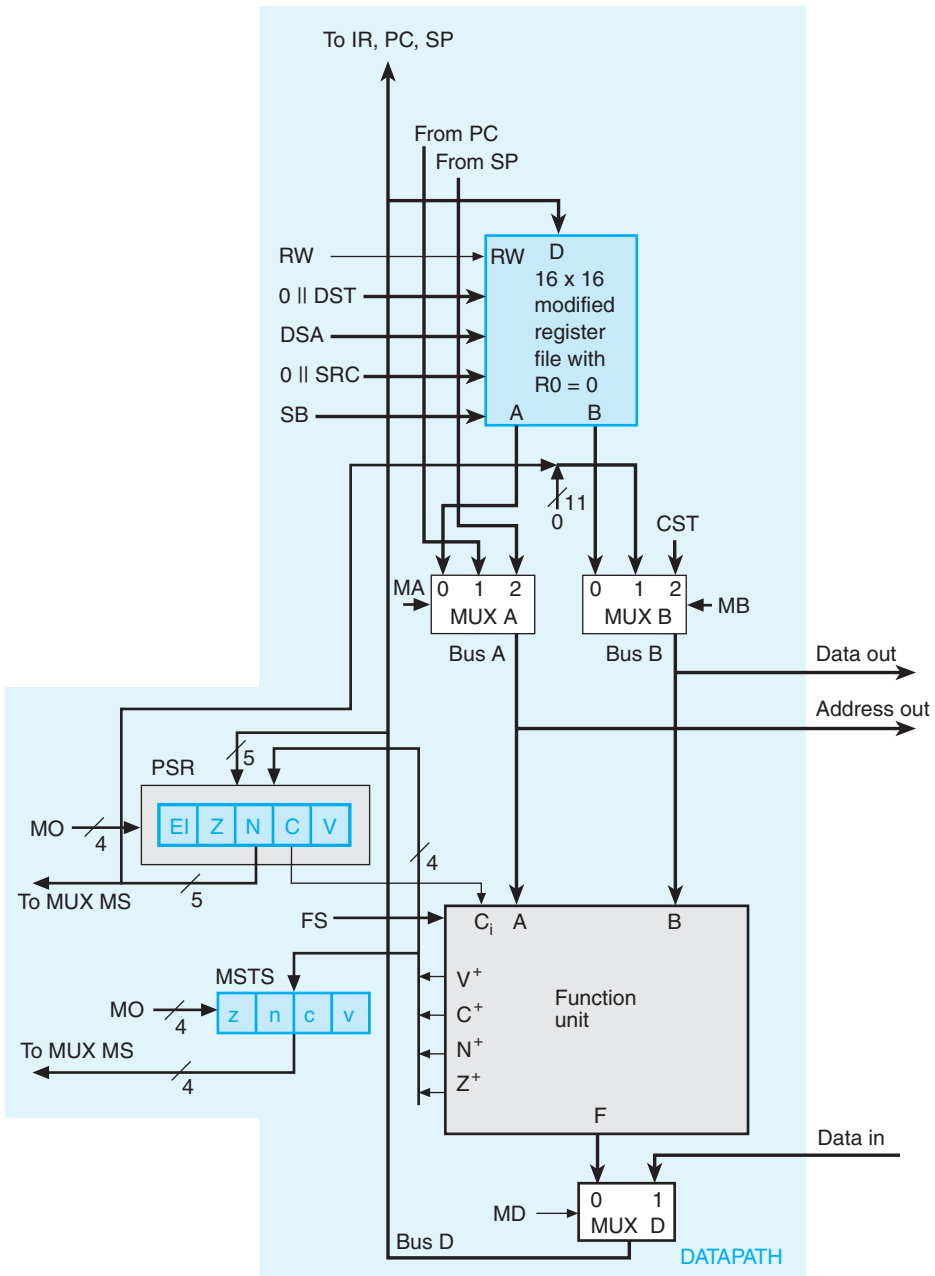
(a) Logic Diagram



(b) Symbol

**FIGURE 5**  
16-Bit Shifter and Symbol

multiplication or division hardware, so these operations must be implemented by microprograms controlling the datapath.



□ **FIGURE 6**  
CISC Datapath

## Microprogrammed Control Organization

The microprogrammed control unit accompanies the datapath of Figure 6 in Figure 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 7 and detailed in Figure 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, 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 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 7 is the microsequencer, which includes the *SBR*, the *CAR*, and the address determination logic. The microsequencer control fields are given in Table 4 and Table 5. Initially, we will discuss the

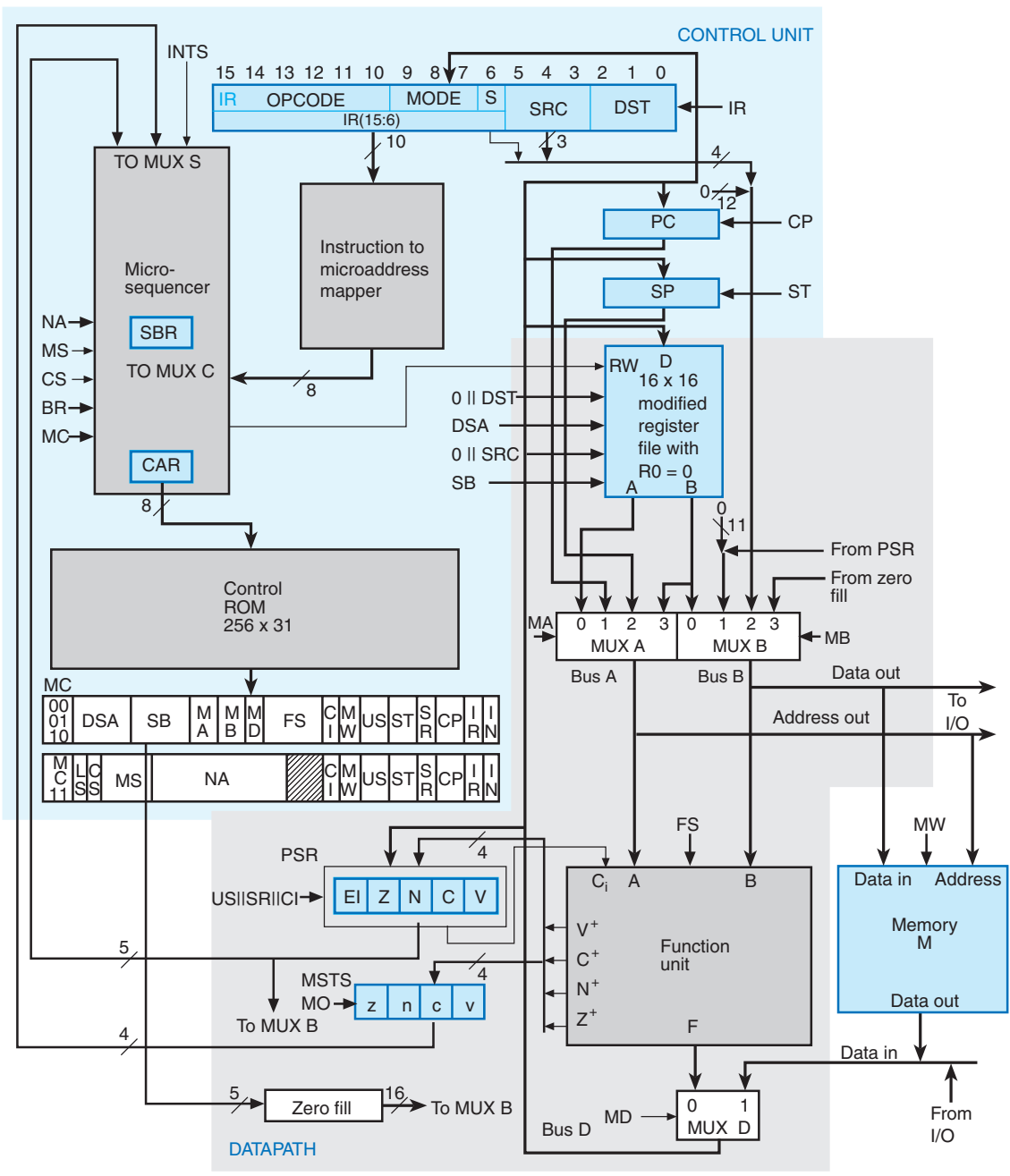
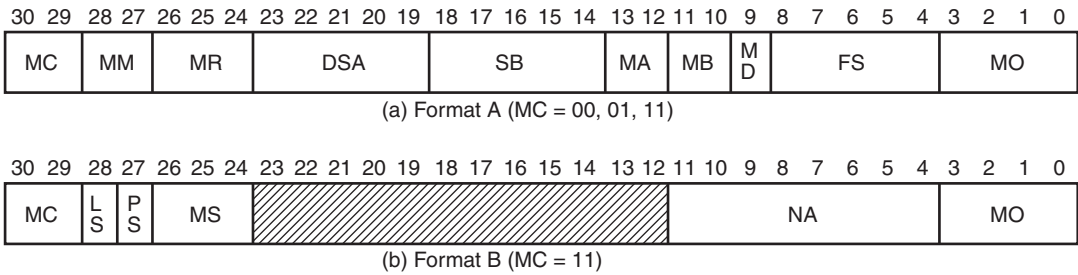


FIGURE 7  
CISC CPU



**FIGURE 8**  
Microinstruction Formats

microsequencer in terms of the operations performed: we then briefly look at the detailed logic.

Table 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

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

	DSA, SB	MA	MB	MD	FS
<i>R0</i> 5 0	00000	Register A	Register B	00	Function
<i>R1</i>	00001	<i>PC</i>	<i>PSR</i>	01	Data in
<i>R2</i>	00010	<i>SP</i>	MCST	10	
<i>R3</i>	00011				$F = A$
<i>R4</i>	00100				$F = A + 1$
<i>R5</i>	00101				$F = A + B$
<i>R6</i>	00110				$F = A + B + 1$
<i>R7</i>	00111				$F = A + \overline{B}$
<i>R8</i>	01000				$F = A + \overline{B} + 1$
<i>R9</i>	01001				$F = A - 1$
<i>R10</i>	01010				$F = A$
<i>R11</i>	01011				$F = A \wedge B$
<i>R12</i> ( <i>SA</i> )	01100				$F = A \vee B$
<i>R13</i> ( <i>SD</i> )	01101				$F = A \oplus B$
<i>R14</i> ( <i>DA</i> )	01110				$F = \overline{A}$
<i>R15</i> ( <i>DD</i> )	01111				$F = B$
<i>R</i> [ <i>DST</i> ]	10XXX*				$F = \text{lsl } B$
<i>R</i> [ <i>SRC</i> ]	11XXX*				$F = \text{lsr } B$
					$F = \text{asl } B$
					$F = \text{asr } B$
					$F = \text{rol } B$
					$F = \text{ror } B$
					$F = \text{rolc } B$
					$F = \text{rorc } B$

\*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 4**  
**Control Word Information for MC and LS**

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

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

□ **TABLE 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 <i>PSR</i> bit	BZ	0001
			Negative <i>PSR</i> bit	BN	0010
			Carry <i>PSR</i> bit	BC	0011
			Overflow <i>PSR</i> bit	BV	0100
			Enable-interrupt <i>PSR</i> bit	EI	0101
			Zero <i>MSTS</i> bit	Bz	0110
			Negative <i>MSTS</i> bit	Bn	0111
			Carry <i>MSTS</i> bit	Bc	1000
			Overflow <i>MSTS</i> bit	Bv	1001
			Interrupt signal <i>INTS</i>	BI	1010

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, for-

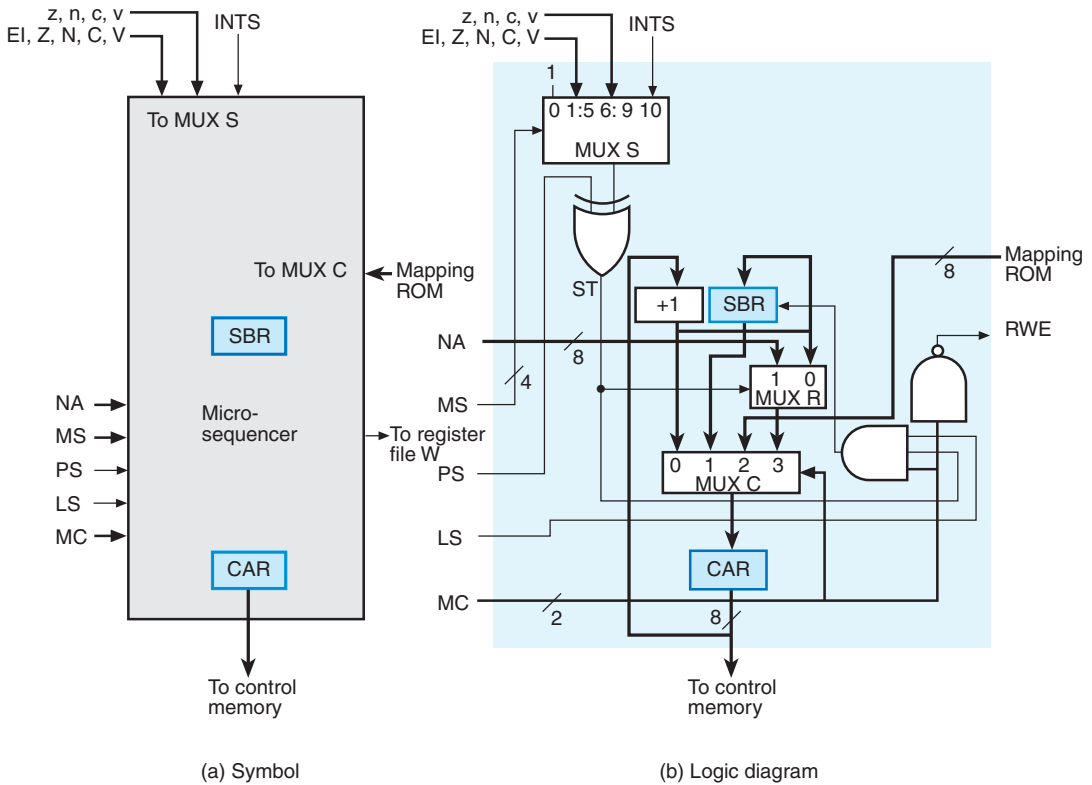
mat A is used. Thus, these branches for decoding instructions can be performed simultaneously with data transfer and manipulation.

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 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 5. MS is used to select either an unconditional branch or the conditions upon which to branch. Here, 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 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 4. The selection of branch conditions given in Table 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 5 11) microinstructions, a two-input NAND gate produces *RWE* (register write enable) 5 0 for MC 5 11. By ANDing *RWE* with the signal otherwise driving *RW* on the register file, we cause MC 5 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 5 11, LS 5 1, and ST 5 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 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



**FIGURE 9**  
The Microsequencer

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 microin-

**TABLE 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

□ **TABLE 6**  
**Control Word Encoding for Instruction Decoder**

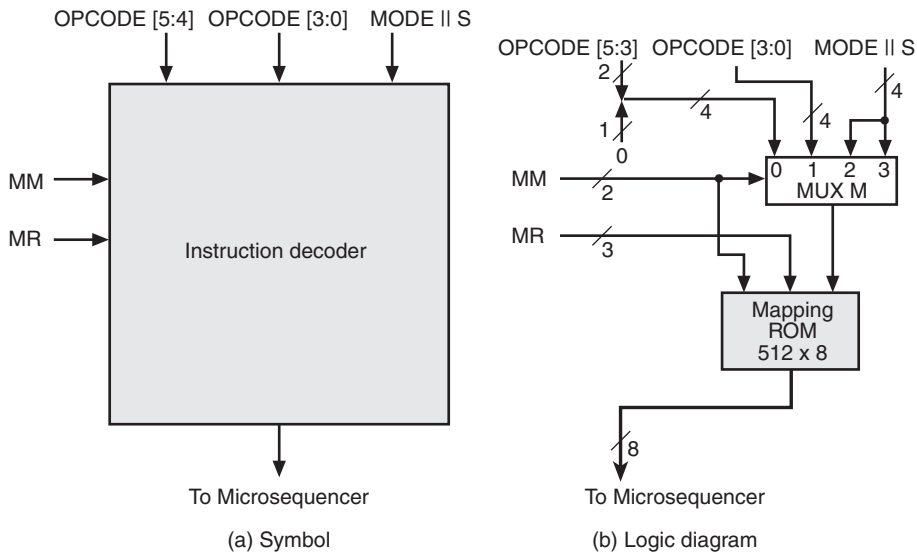
MM		MR	
Select	Code	Region	Code
		6	110
		7	111

structions, 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(a) is made up of a quad 4-to-1 multiplexer MUX *M* and a ROM, as shown in Figure 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 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 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



□ **FIGURE 10**  
Instruction Decoder

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

MO		
Operations	Symbolic Notation	Code
No operation	—	0000
<i>INACK</i> 5 1	INCK	0001
Memory write <sup>p</sup>	WRITE	0010
Load <i>PC</i> <sup>p</sup>	LPC	0011
Load <i>IR</i> and increment <i>PC</i> <sup>p</sup>	DPC	0100
Increment <i>PC</i>	IPC	0101
Load <i>PSR</i> <sup>p</sup>	LST	0110
Load <i>SP</i> <sup>p</sup>	LSP	0111
Decrement <i>SP</i>	DSP	1000
Decrement <i>SP</i> and memory write <sup>p</sup>	DSM	1001
Increment <i>SP</i>	ISP	1010
Select <i>C</i> as <i>C</i> <sub>in</sub> for arithmetic and action EST as follows	CIN	1011
Enable update of status bits <i>Z</i> , <i>N</i> , <i>C</i> , and <i>V</i>	EST	1100
Enable update of status bits <i>Z</i> and <i>C</i>	EZC	1101
Enable update of status bits <i>N</i> and <i>Z</i>	ENZ	1110
Enable update of microstatus bits <i>z</i> , <i>n</i> , <i>c</i> , and <i>v</i>	EMS	1111

---

\* Prevents write to register file

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.

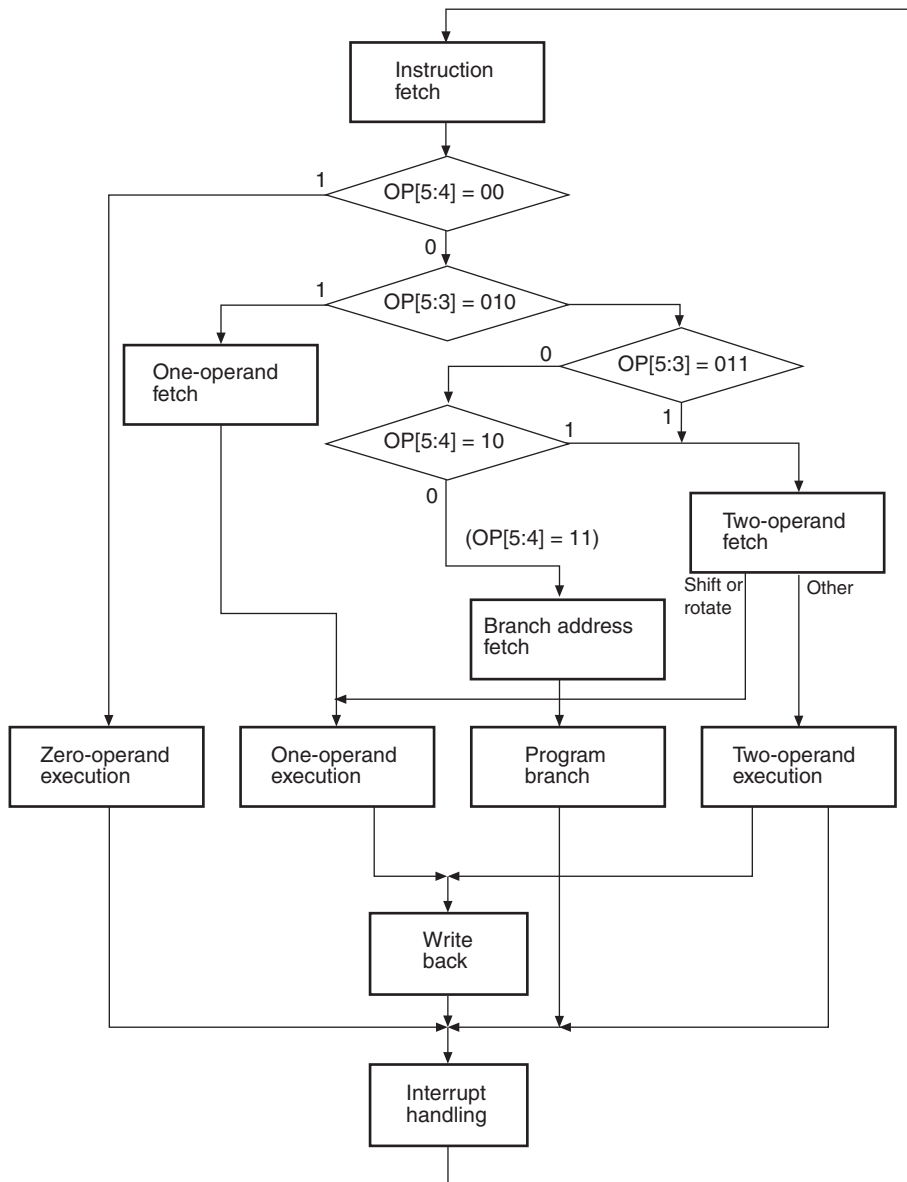
## Microprogram Structure

We approach the microprogram design top down. The top level consists of an ASM-like chart giving a flow of microroutines. 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 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 *R12* through *R15* (*SA*, *SD*, *DA*, and *DD*). The four execution routines find the operands and addresses in these standard register locations and, in most cases, use 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



□ **FIGURE 11**  
ASM-like Chart of Microroutines

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 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  $X_s$ '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 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 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 11: Zero-operand Execution, One-operand Fetch, Two-operand Fetch, and Branch Address Fetch.

Suppose that OPCODE is 010000 and that MODE is 011. Then, from the second row of Table 8, the next microinstruction is in microroutine One-operand Fetch, which begins with microinstruction 1OF. This microroutine is given in Table 10. The first instruction involves the use of the mapping ROM to decode

**TABLE 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 <sub>3</sub> O <sub>2</sub> O <sub>1</sub> O <sub>0</sub>	16 0-Op EX Microinstruction Addresses
1EX	01	001	OPCODE(3:0) = O <sub>3</sub> O <sub>2</sub> O <sub>1</sub> O <sub>0</sub>	16 1-Op EX Microinstruction Addresses
2EX	01	010	OPCODE(3:0) = O <sub>3</sub> O <sub>2</sub> O <sub>1</sub> O <sub>0</sub>	16 2-Op Ex Microinstruction Addresses
BEX	01	011	OPCODE(3:0) = O <sub>3</sub> O <sub>2</sub> O <sub>1</sub> O <sub>0</sub>	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 <sub>2</sub> M <sub>1</sub> M <sub>0</sub> X	8 1-Op OF Microinstruction Addresses
2OF	10	010	MODE ∥ S = M <sub>2</sub> M <sub>1</sub> M <sub>0</sub> S	16 2-Op OF Microinstruction Addresses
BAF	10	011	MODE ∥ S = M <sub>2</sub> M <sub>1</sub> M <sub>0</sub> X	8 Br-Op Microinstruction Addresses
XCH2	10	100	MODE ∥ S = XXX1	XCH3
XCH2	10	100	MODE ∥ S = 0000	XCH3
XCH2	10	100	MODE ∥ S = XX10	XCH4
XCH2	10	100	MODE ∥ S = X1X0	XCH4
XCH2	10	100	MODE ∥ S = 1XX0	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

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

□ **TABLE 9**  
**Instruction Fetch Microroutine**

Sym Add	Register Transfer Description	MM MR DSA								FS	
		MC	/LS	/PS	/MS	SB	MA	MB	MD	/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**  
**One-operand Fetch Microroutine**

Sym Add	Register Transfer Description	MM MR DSA								FS	
		MC	/LS	/PS	/MS	SB	MA	MB	MD	/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
1RG10	$DA \leftarrow R[DST]$	0	0	0	0E	10	0	0	0	00	0
1RG11	$DD \leftarrow M[DA], CAR \leftarrow 1EX(ROM)$	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	3	0	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

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 8. Since the value of the  $OPCODE(5:4) \parallel 00_2$  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, 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 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 11**  
**Two-operand Fetch Microroutine Examples**

Sym Add	Register Transfer Description	MM MR DSA								FS	
		MC	/LS	/PS	/MS	SB	MA	MB	MD	/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 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,  $MODE \ 5 \ 011$ . Microinstructions in 2DR0 through 2DR2 handle direct addressing for  $S \ 5 \ 0$  and, in addresses 2DRS0 through 2DRS2, handle direct addressing for  $S \ 5 \ 1$ . For the case  $S \ 5 \ 0$ , 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  $S \ 5 \ 1$ , 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 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

□ **TABLE 12**  
**Example of Branch Address Fetch**

Sym Add	Register Transfer Description	MM/ MR/ DSA/								FS	
		MC	LS	PS	MS	SB	MA	MB	MD	/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

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.

The next set of routines perform the actual execution of instructions. In Table 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 register. 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 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

□ **TABLE 13**  
**Zero-operand Execution Microroutine Examples**

Sym Add	Register Transfer Description	MM MR DSA								FS	
		MC	/LS	/PS	/MS	SB	MA	MB	MD	/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

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 *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  $R13$  ( $SD$ ) is transferred to  $R9$ , 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  $R13$  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 ( $\parallel$ ) notation is used here because it easily describes the incoming bit value. Next,  $R9$  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 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

□ **TABLE 14**  
**One-operand Execution Microroutine Examples**

Sym	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	$DD \leftarrow DD - 1, CAR \leftarrow WB0(ROM)$	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	$DD \leftarrow DD + 1, CAR \leftarrow WB0(ROM)$	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

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.

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

□ **TABLE 15**  
**Two-operand Execution Microroutine Examples**

Sym Add	Register Transfer Description	MM MR DSA								FS	
		MC	/LS	/PS	/MS	SB	MA	MB	MD	/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

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 5 0, the contents of SD are transferred to M[SA]. This reasoning corresponds to the five rows labeled with XCH2 (MM 5 10 and MR 5 100) in the mapping ROM contents in Table 8. For the first row, if S 5 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 5 0 and MODE 5 000, the transfer from SD to R[Src] in XCH3 is executed, since this is register mode. Otherwise, if

□ **TABLE 16**  
**Program Branch Microroutine Examples**

Sym Add	Register Transfer Description	MM MR DSA								FS	
		MC	/LS	/PS	/MS	SB	MA	MB	MD	/NA	MO
BEX	$CAR \leftarrow ROM[01011_2 \parallel OPCODE(3:0)]$	2	1	3	00	00	0	0	0	00	0
JMP	$PC \leftarrow DA, CAR \leftarrow INT0(ROM)$	2	1	5	0E	00	0	0	0	00	3
CALL0	$R8 \leftarrow PC$	0	0	0	08	00	1	0	0	00	0
CALL1	$SP \leftarrow SP - 1$	0	0	0	0	00	0	0	0	00	8
CALL2	$M[SP] \leftarrow R8$	0	0	0	0	08	2	0	0	00	2
CALL3	$PC \leftarrow DA, CAR \leftarrow INT0(ROM)$	2	1	5	0E	00	0	0	0	00	3
BZ0	Z: $CAR \leftarrow BRA$	3	0	0	1	—	—	—	—	BRA	0
BZ1	$CAR \leftarrow INT0(ROM)$	2	1	5	00	00	0	0	0	00	0
BRA	$PC \leftarrow DA, CAR \leftarrow INT0(ROM)$	2	1	5	0E	00	0	0	0	00	3

□ **TABLE 16**  
**Program Branch Microroutine Examples**

Sym Add	Register Transfer Description	MC	MM	MR	DSA	SB	MA	MB	MD	FS	MO
			/LS	/PS	/MS					/NA	
BNZ0	$\bar{Z}: CAR \leftarrow BRA$	3	0	1	1	—	—	—	—	BRA	0
BNZ1	$CAR \leftarrow INT0(ROM)$	2	1	5	00	00	0	0	0	00	0

S 5 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 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.

For those instructions with results to be stored, the Write-Back microroutine in Table 17 is executed. To determine where to put the contents of *DD*, it is necessary to

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

Sym Add	Register Transfer Description	MC	MM	MR	DSA	SB	MA	MB	MD	FS	M
			/LS	/PS	/MS					/NA	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

examine the value of *S* and *MODE*. If *S* 5 0, then the destination address is register *R[DST]*. If *S* 5 1, then the destination is register *R[DST]* if *MODE* 5 000. 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 8 labeled with WB0. Regardless of the Write-Back operation performed, the Interrupt-Handling microroutine is executed next.

The final microroutine, for Interrupt-Handling, is given in Table 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

□ **TABLE 10-18**  
**Interrupt-Handling Microroutine**

Sym Add	Register Transfer Description	DSA								FS	
		MC	/LS	/PS	/MS	SB	MA	MB	MD	/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

the *PC*, as described in Section 11-9. 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.

## REFERENCES

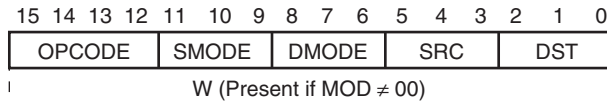
- DIETMEYER, D. L., *Logic Design of Digital Systems*, 3rd ed. Boston, MA: Allyn-Bacon, 1988.
- MANO, M. M. *Computer Engineering: Hardware Design*. Englewood Cliffs, NJ: Prentice Hall, 1988.
- HAMACHER, V. C., VRANESIC, Z. G., AND ZAKY, S. G. *Computer Organization*, 3rd ed. New York, NY: McGraw-Hill, 1990.
- HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 2nd ed. San Francisco, CA: Morgan Kaufmann, 1996.
- KANE, G., AND HEINRICH, J. *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- SPARC INTERNATIONAL, INC. *The SPARC Architecture Manual: Version 8*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- MANO, M. M. *Computer System Architecture*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1993.
- PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. San Mateo, CA: Morgan Kaufmann, 1994.
- WEISS, S., AND SMITH, J. E. *POWER and PowerPC*. San Mateo, CA: Morgan Kaufmann, 1994.
- WYANT, G., AND HAMMERSTROM, T. *How Microprocessors Work*. Emeryville, CA: Ziff-Davis Press, 1994.

## PROBLEMS

he plus (+) indicates a more advanced problem.

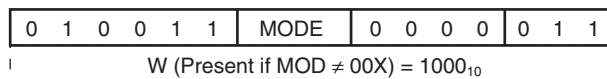
- \*Write a register transfer statement that describes the operation performed by each of the following instructions for the CISC design by using information from Table 1. In those cases in which a second word  $W$  is part of the instruction, it is given. Use hexadecimal integers for representing addresses 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
- In the CISC formats in Figure 2, the format with  $IR(15:14)$  5 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?
- +Suppose that the CISC instruction format in Figure 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.

- Based on this new format, how many operations can be specified using  $OPCODE$  alone?
  - 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?
- \*The following instruction in location  $2001_{10}$  is executed for each of the eight addressing modes in Table 2.



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

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  $R7$ .
6. \*For each of the CISC shift operations given, shifter input  $A$  equals  $F0C6_{16}$ , and  $C$  equals 1. Using the shifter in Figure 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)
7. +Design the *PSR* and *MSTS* hardware for the CISC design based on Figure 7, Table 1, and Table 7. Use D flip-flops, gates, and a ROM or PLA. Note carefully all sources that may modify the carry bit.
8. Draw an ASM chart for each of the following:
  - (a) the logical shift right microcode given in Table 14.
  - (b) the microcode for a rotate right with carry operation.
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 14.
  - (a) Immediate
  - (b) Relative
  - (c) Register indirect
  - (d) Relative indirect
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 14.
  - (a) Arithmetic shift right
  - (b) Rotate left with carry
  - (c) Branch if overflow
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 14.
  - (a) Negate
  - (b) Exclusive-OR
  - (c) Branch if no carry
  - (d) Subtract with borrow
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 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.

- 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 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.
- 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 14.