A Datapath en microprogrammed control

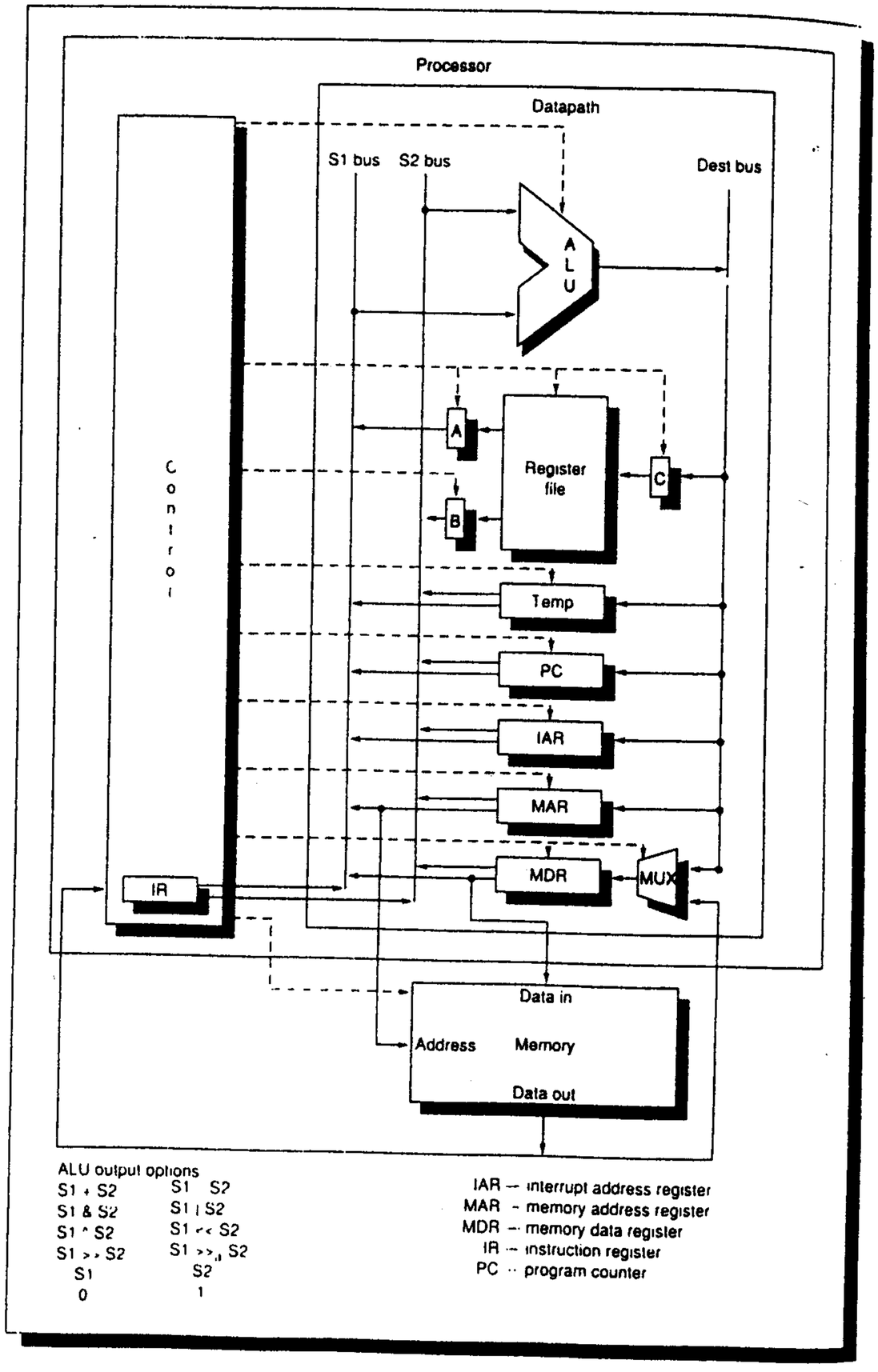

FIGURE 5.1 (See adjoining page.) A typical processor, divided into control and datapath, plus memory. The paths for control are in dashed lines and the paths for data transfer are in solid lines. The processor uses three buses: S1, S2, and Dest. The fundamental operation of the datapath is reading operands from the register file, operating on them in the ALU, and then storing the result back. Since the register file does not need to be read and written every clock cycle, most designers follow the advice of making the frequent case fast by breaking this sequence into multiple clock cycles and making the clock cycle shorter. Thus, in this datapath there are latches on the two outputs of the register file (called A and B) and a latch on the input (C). The register file contains the 32 general-purpose registers of DLX. (Register 0 of the register file always has the value 0, matching the definition of register 0 in the DLX instruction set.) The program counter (PC) and interrupt address register (IAR) are also part of the state of the machine. There are also registers, not part of the state, used in the execution of instructions: memory address register (MAR), memory data register (MDR), instruction register (IR), and temporary register (Temp). The Temp register is a scratch register that is available for temporary storage for control to perform some DLX instructions. Note that the only path from the S1 and S2 buses to the Dest bus is through the ALU.

5.5 Microprogrammed Control

After constructing the first full-scale, operational, stored-program computer in 1949, Maurice Wilkes reflected on the process. I/O was easy—teletypewriters could just be purchased directly from the telegraph company. Memory and the datapath were highly repetitive, and that made things simpler. But control was neither easy nor repetitive, so Wilkes set out to discover a better way to design control. His solution was to turn the control unit into a miniature computer by having a table to specify control of the datapath and a second table to determine control flow at the micro level. Wilkes called his invention *microprogramming*, and attached the prefix "micro" to traditional terms used at the control level: microinstruction, microcode, microprogram, and so on. (To avoid confusion the prefix "macro" is sometimes used to describe the higher level, e.g., macroinstruction and macroprogram.) Microinstructions specify all the control signals for the datapath, plus the ability to conditionally decide which micro

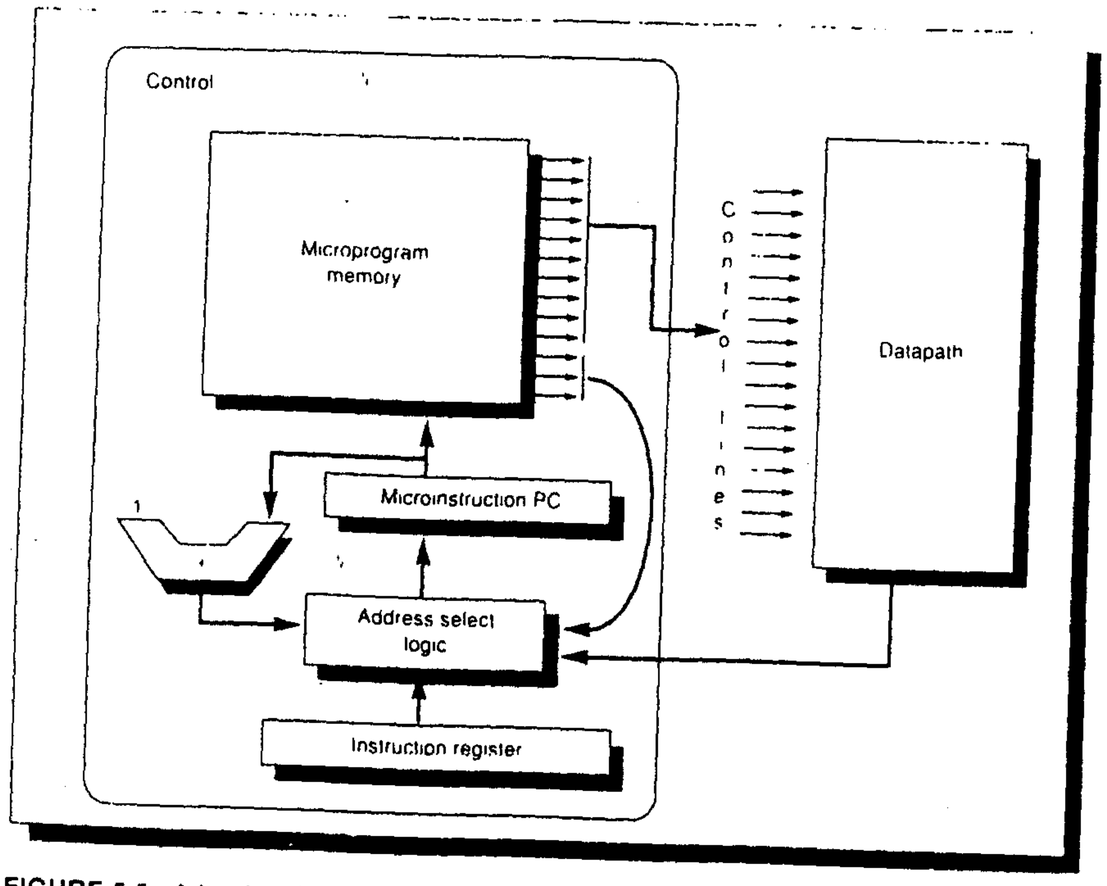

FIGURE 5.5 A basic microcoded engine. Unlike Figure 5.3 (page 206), there is an incrementer and special logic to select the next microinstruction. There are two approaches to specifying the next microinstruction: use a microinstruction program counter, as shown above, or include a next microinstruction address in every microinstruction. Microprogram memory is sometimes called ROM because most early machines use ROM for control

instruction should be executed next. As the name "microprogramming" suggests, once the datapath and memory for the microinstructions are designed, control becomes essentially a programming task: that is, the task of writing an interpreter for the instruction set. The invention of microprogramming enabled the instruction set to be changed by altering the contents of control store without touching the hardware. As we will see in Section 5.10, this ability played an important role in the IBM 360 family—one that was a surprise to its designers.

Figure 5.5 shows an organization for a simple microprogrammed control. The structure of a microprogram is very similar to the state diagram, with a microinstruction for each state in the diagram.

ABCs of Microprogramming

While it doesn't matter to the hardware how the control lines are grouped within a microinstruction, control lines performing related functions are traditionally placed next to each other for ease of understanding. Groups of related control lines are called *fields* and are given names in a microinstruction format. Figure 5.6 shows a microinstruction format with eight fields, each named to reflect its function. Microprogramming can be thought of as supplying the proper bit pattern in each field, much like assembly language programming of "macroinstructions."

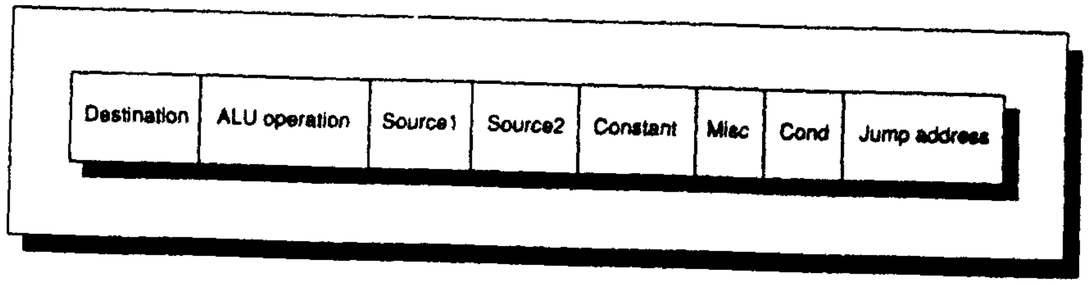

FIGURE 5.6 Example microinstruction with eight fields (used for DLX in Section 5.7).

A program counter can be used to supply the next microinstruction, as shown in Figure 5.5, but some computers dedicate a field in every microinstruction to the address of the next instruction. Some even provide multiple next-address fields to handle conditional branches.

While conditional branches could be used to decode an instruction by testing the opcode one bit at a time, this tedious approach is too slow in practice. The simplest fast instruction decoding scheme is to jam the macroinstruction opcode into the middle of the address of the next microinstruction, similar to an indexed jump instruction in assembly language. A more refined approach is to use the opcode to index a table containing microinstruction addresses that supply the next address, similar to a jump table in assembly code.

The microprogram memory, or *control store*, is the most visible and easily measured hardware in microprogrammed control; hence, it is the focus of techniques to reduce hardware costs. Techniques to trim control-store size include reducing the number of microinstructions, reducing the width of each microinstruction, or both. Just as cost is traditionally measured by control-store size, performance is traditionally measured by CPI. The wise microprogrammer knows the frequency of macroinstructions by using statistics like those in Chapter 4, and hence knows where and how time is best spent—instructions demanding the largest part of execution time are optimized for speed, and the others are optimized for space.

In four decades of microprogramming history there have been a wide variety of terms and techniques for microprogramming. In fact, a workshop has met annually on this subject since 1968. Before looking at a few examples, let us remember that control techniques—whether hardwired or microcoded—are judged by their impact on hardware cost, clock cycle time, CPI, and development time. In the next two sections we will examine how hardware costs can be lowered by reducing control-store size. First we look at two techniques to reduce the width of microinstructions, then one technique to reduce the number of microinstructions.

Reducing Hardware Costs by Encoding Control Lines

The ideal approach to reducing control store is to first write the complete microprogram in a symbolic notation and then measure how control lines are set in each microinstruction. By taking measurements we are able to recognize control bits that can be encoded into a smaller field. If no more than one of, say, 8 lines is set simultaneously in the same microinstruction, then they can be encoded into a 3-bit field ($\log_2 8 = 3$). This change saves 5 bits in every microinstruction and does not hurt CPI, though it does mean the extra hardware cost of a 3-to-8 decoder is needed to generate the original 8 control lines. Nevertheless, shaving 5 bits off control-store width will usually overcome the cost of the decoder.

This technique of reducing field width is called *encoding*. To further save space, control lines may be encoded together if they are only occasionally set in the same microinstruction; two microinstructions instead of one are then required when both must be set. As long as this doesn't happen in critical routines, the narrower microinstruction may justify a few extra words of control store.

There are dangers to encoding. For example, if an encoded control line is on the critical timing path, or if the hardware it controls is on the critical path, then the clock cycle time will suffer. A more subtle danger is that a later revision of the microcode might encounter situations where control lines would be set in the same microinstruction, either hurting performance or requiring changes to the hardware that could lengthen the development cycle.

Example

Assume we want to encode the three fields that specify a register on a bus—Destination, Source1, and Source2—in the DLX microinstruction format in Figure 5.6. How many bits of control store can be saved versus unencoded fields?

Answer

Figure 5.7 lists the registers for each source and destination of the datapath in Figure 5.1 (page 200). Note that the destination field must be able to specify that nothing is modified. Without encoding, the 3 fields require 7 + 9 + 9, or 25 bits. Since $\log_2 7 \approx 2.8$ and $\log_2 9 \approx 3.2$, the encoded fields require 3 + 4 + 4, or 11 bits. Thus, encoding these 3 fields saves 14 bits per microinstruction.

Number	Destination	Source1/Source2
0	(None)	A/B
1	C	Temp
2	Temp	PC
3	PC	IAR
4	IAR	MAR
5	MAR	MDR
6	MDR	IR (16-bit imm)
7	---	IR (26-bit imm)
8	---	Constant

FIGURE 5.7 The sources and destinations specified in the three fields of Figure 5.6 from the datapath description in Figure 5.1. A and B are not separate entries because A can only transfer on the S1 bus and B can only transfer on the S2 bus (see Figure 5.1 on pages 200–201). The last entry in the third column, Constant, is used by control to specify a constant needed in an ALU operation (e.g., 4). See Section 5.7 for its use.

Reducing Hardware Costs with Multiple Microinstruction Formats

Microinstructions can be made narrower still if they are broken into different formats and given an opcode or *format field* to distinguish them. The format field gives all the unspecified control lines their default values, so as not to change anything else in the machine, and is similar to the opcode of a macroinstruction.

Reducing hardware costs by using format fields has its own performance cost—namely, executing more microinstructions. Generally, a microprogram using a single microinstruction format can specify any combination of operations in a datapath and will take fewer clock cycles than a microprogram made up of restricted microinstructions. Narrower machines are cheaper because memory chips are also narrow and tall: It takes many fewer chips for a 16K word by 24-bit memory than for a 4K word by 96-bit memory. (When control memory is on the processor chip, this hardware advantage is no longer true.)

This narrow but tall approach is often called *vertical microcode*, while a wide but short approach is called *horizontal microcode*. It should be noted that the terms "vertical miocrocode" and "horizontal microcode" have no universal definition—the designers of the 8086 considered its 21-bit microinstruction to be more horizontal than other single-chip computers of the time. The related terms *maximally encoded* and *minimally encoded* lead to less confusion.

Figure 5.8 plots control-store size against microinstruction width for three families of computers. Notice that for each family the total size is similar, even though the width varies by a factor of 6. As a rule, minimally encoded control stores use more bits, and the narrow but tall aspect of memory chips means that maximally encoded control stores naturally have more entries. Sometimes designers of minimally encoded machines don't have the option of shorter RAM chips, causing wide microinstruction machines to end up with many words of control store. Since the hardware costs are not lower if microcode doesn't use all the space in control store, machines in this class can end up with much larger control stores than expected from other implementations. The ECL RAM available to build the VAX 8800, for example, led to 2000 K bits of control store.

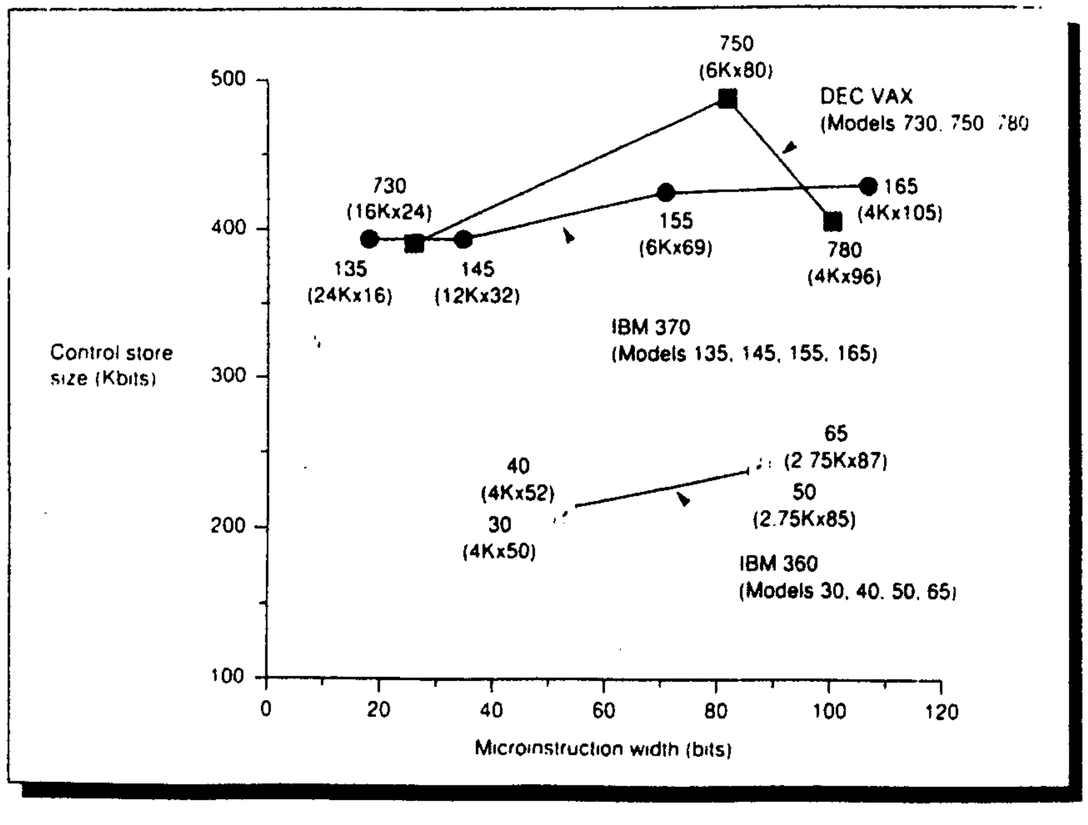

FIGURE 5.8 Size of control store versus width of microinstructions for 11 computer models. Each point is identified by the length and width of control store (not including parity). Models selected from each family are ones that shipped about the same time: IBM 360 models 30, 40, 50, and 65 all shipped in 1965; IBM 370 models 145, 155, and 165 shipped in 1971, with the 135 following in the next year; and the VAX model 780 was shipped in 1978, followed by the 750 in 1980 and the 730 in 1982. The development of the VAX designs all overlapped one another inside DEC.

Reducing Hardware Costs by Adding Hardwired Control to Share Microcode

The other approach to reducing control store is to reduce the number of microinstructions rather than their width. Microsubroutines provide one approach, as well as routines with common "tail" sequences sharing code by jumps.

More sharing can be done with hardwired control assistance. For example, many microarchitectures allow bits of the instruction register to specify the correct register. Another common assist is to use portions of the instruction register to specify the ALU operation. Each of these assists is under microprogrammed control and is invoked with a special value in the appropriate field. The 8086 uses both techniques, giving one 4-line routine responsibility for 32 opcodes. The drawback of adding hardwired control is it may stretch the development cycle because it no longer involves programming, but requires hardware layout for designing and debugging.

This section and the previous two give techniques for reducing cost. The following sections present three techniques for improving performance.

Reducing CPI with Special Case Microcode

As we have noted, the wise microprogrammer knows when to save space and when to spend it. An instance of this is dedicating extra microcode for frequent instructions, thereby reducing CPI. For example, the VAX 8800 uses its large control store for many versions of the CALLS instruction, optimized for register saving depending upon the value in the register-save mask. Candidates for special case microcode can be uncovered by instruction mix measurements, such as those found in Chapter 4 or in Appendix B, or by counting the frequency of use of each microinstruction in an existing implementation (see Emer and Clark [1984]).

Reducing CPI by Adding Hardwired Control

Adding hardwired control can reduce costs as well as improve performance. For example, VAX operands can be in memory or registers, but later machines reduce CPI by having special code for register–register or register–memory moves and adds: ADDL2 Rn, 10(Rm) takes five or more cycles on the 780, but as few as one on the 8600. Another example is in the memory interface, where the straightforward solution is for microcode to continuously test and branch until memory is ready. Because of the delay between the time a condition becomes true and the time the next microinstruction is read, this approach can add one extra clock to each memory access. The importance of the memory interface is underlined by the 780 and 8800 statistics—20% of the 780 clock cycles and 23% of the 8800 are waiting for memory to be ready, these are called

stalls. A *stall* is where an instruction must pause one or more clock cycle waiting for some resource to be available. In this chapter stalls occur only when waiting for memory; in the next chapter we'll see other reasons for stalls.

Many machines approach this problem by having the hardware stall microinstruction that tries to access the memory-data register before the memory operation is completed. (This can be accomplished by freezing the microinstruction address so that the same microinstruction is executed until the condition is met.) The instant the memory reference is ready, the microinstruction that needs the data is allowed to complete, avoiding the extra clock delay to access control memory.

Reducing CPI by Parallelism

Sometimes CPI can be reduced with more operations per microinstruction. This technique, which usually requires a wider microinstruction, increases parallelism with more datapath operations. It is another characteristic of machines labeled horizontal. Examples of this performance gain can be seen in the fact that the fastest models of each family in Figure 5.8 also have the widest microinstructions. Making the microinstruction wider does not guarantee increased performance, however. An example where the potential gain was not realized is found in a microprocessor very similar to the 8086, except that another bus was added to the datapath, requiring six more bits in its microinstruction. This could have reduced the execution phase from three clock cycles to two for many popular 8086 instructions. Unfortunately, these popular macroinstructions were grouped with macroinstructions that couldn't take advantage of this optimization, so they all had to run at the slower rate.

5.7 Putting It All Together: Control for DLX

The control for DLX is presented here to tie together the ideas from the previous three sections. We begin with a finite-state diagram to represent hardwired control and end with microprogrammed control. Both versions of DLX control are used to demonstrate tradeoffs to reduce cost or to improve performance. Because the figures are already too large, the checking for data page faults or arithmetic overflow shown in Figure 5.12 (page 218) is not included in this section. (Exercise 5.12 adds them.)

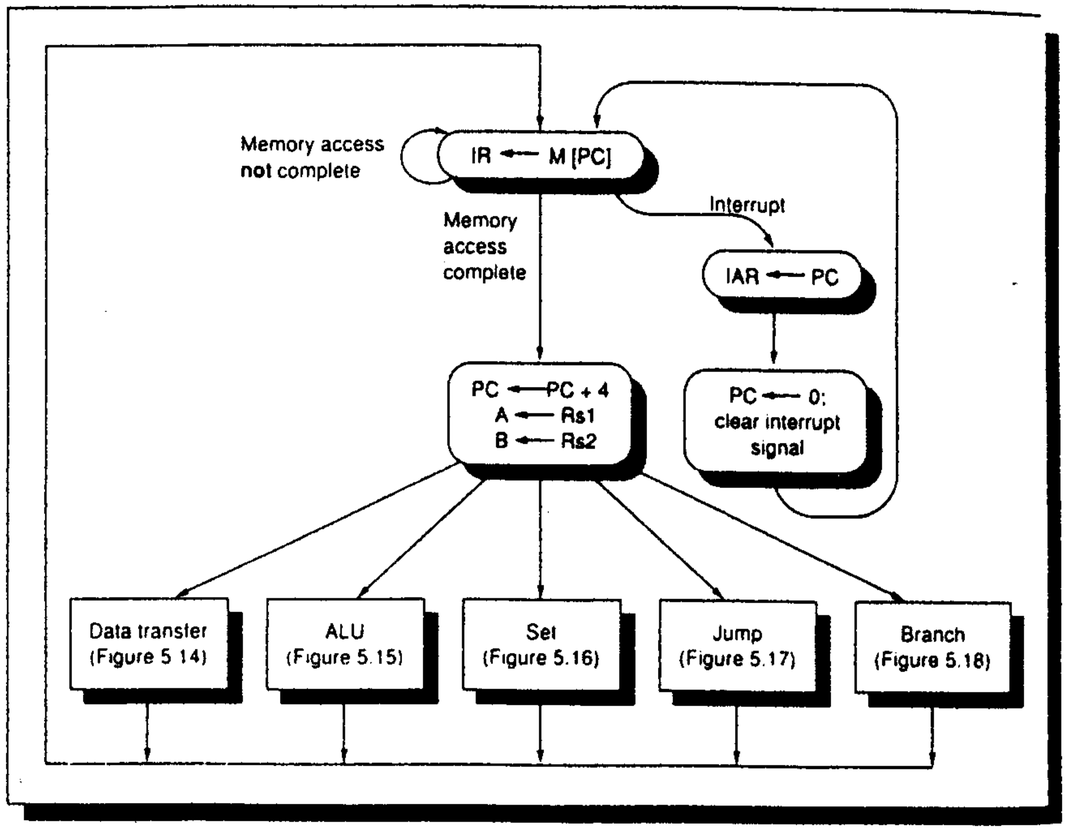

FIGURE 5.13 The top-level view of the DLX finite-state diagram for the non-floating-point instructions. The first two steps of instruction execution—instruction fetch and instruction decode/register fetch—are shown. The first state repeats until the instruction is fetched from memory or an interrupt is detected. If an interrupt is detected, the PC is saved in IAR and PC is set to the address of the interrupt routine. The last three steps of instruction execution—execution/effective address, memory access, and write back—are shown in Figures 5.14 to 5.18 on pages 221–224.

Rather than trying to draw the DLX finite-state machine in a single figure showing all 52 states, Figure 5.13 (see page 220) shows just the top level, containing 4 states plus references to the rest of the states detailed in Figures 5.14 (below) through 5.18 (page 224). Unlike Figure 5.2 (page 205), Figure 5.13 takes advantage of the change to the datapath allowing PC to address memory

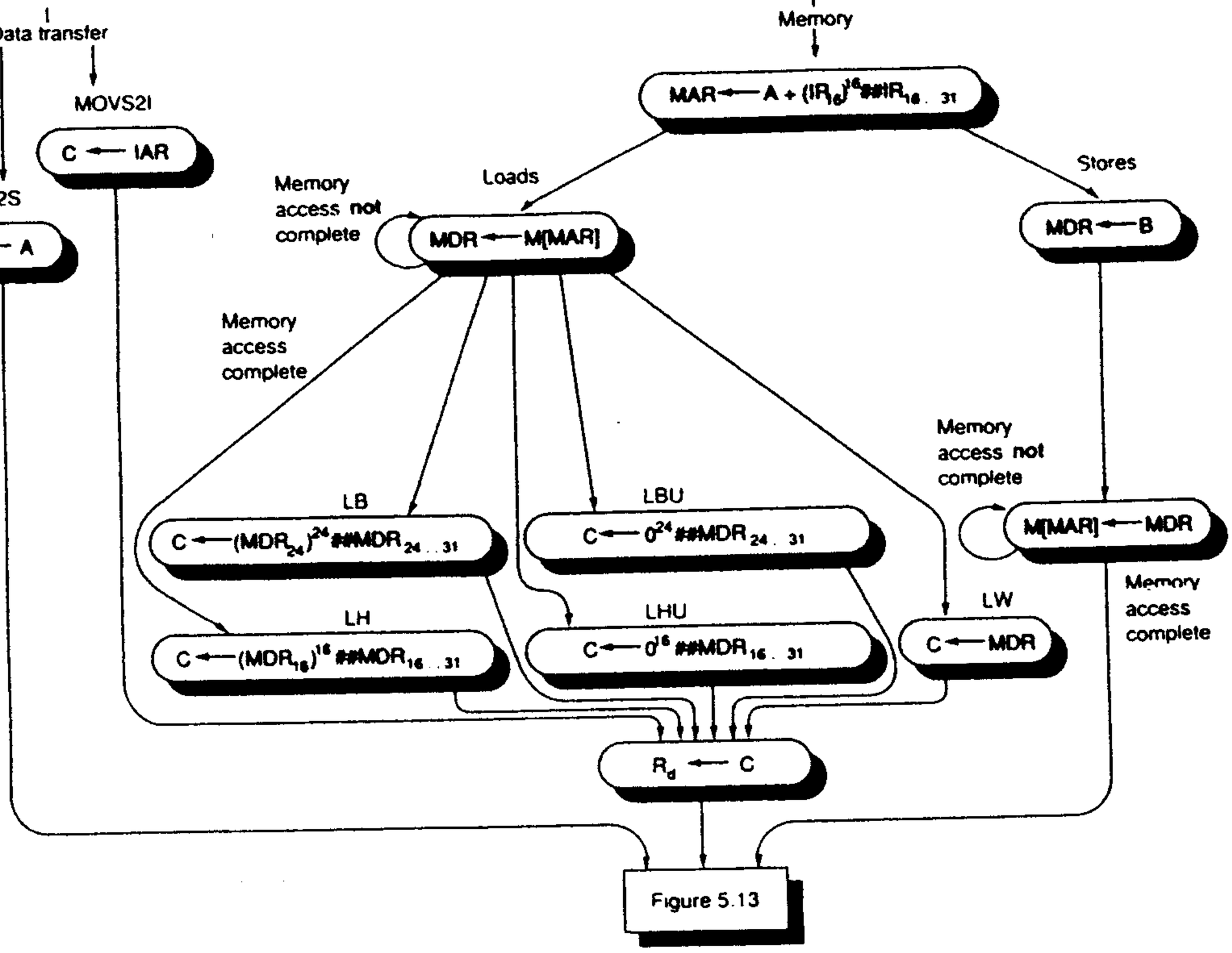

FIGURE 5.14 **The effective address calculation, memory-access, and write-back states for the memory-access and data-transfer instructions of DLX.** For loads, the second state repeats until the data is fetched from memory. The final state of stores repeats until the write is complete. While the operation of all five loads is shown in the states of this figure, the proper operation of writes depends on the memory system writing bytes and halfwords, without disturbing the rest of the word in memory, and correctly aligning the bytes and halfwords (see Figure 3.10, page 97) over the proper bytes of memory. On completion of execution control transfers to Figure 5.13, found on page 220.

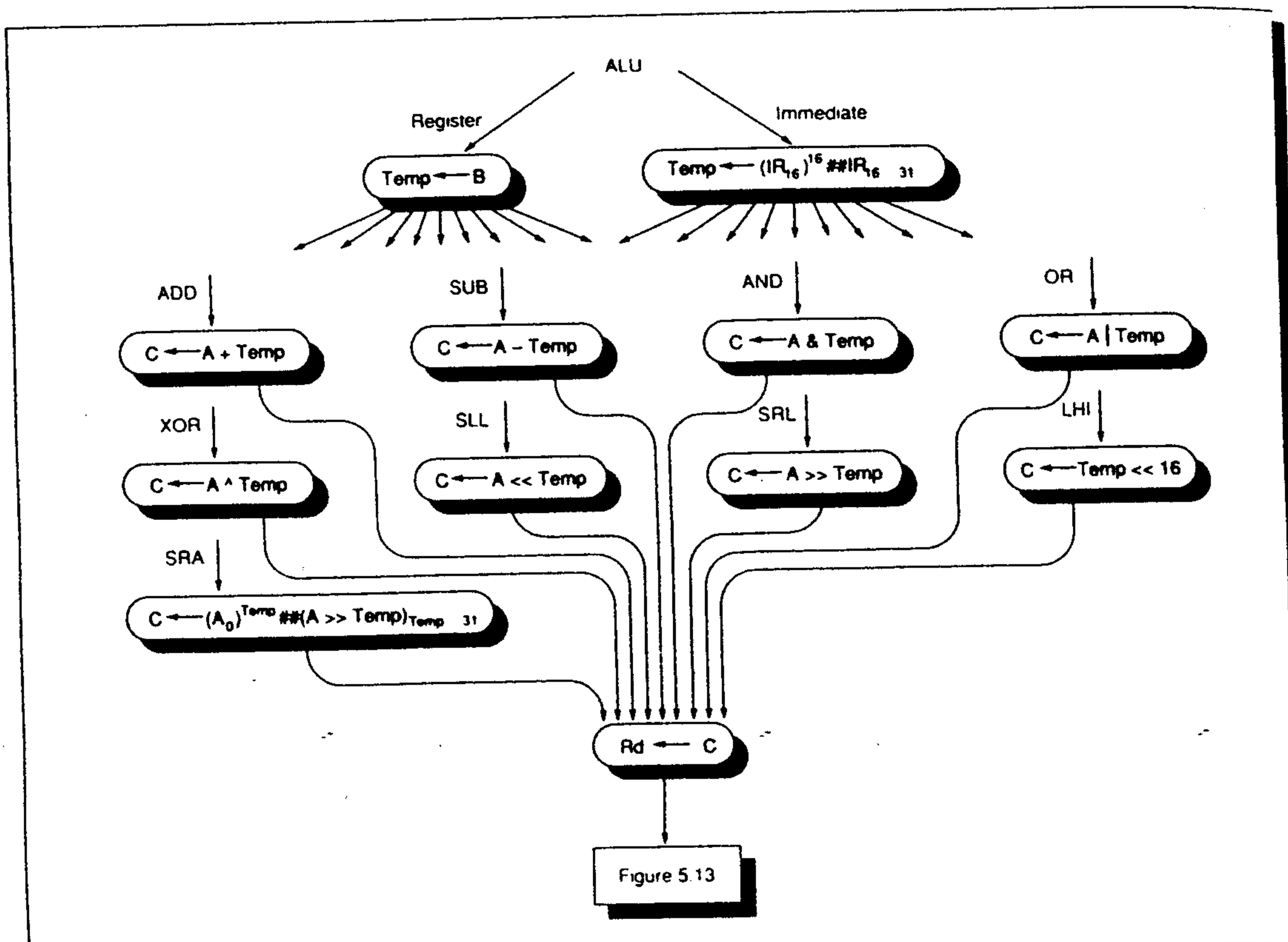

FIGURE 5.15 The execution and write-back states for the ALU instructions of DLX. After putting a register or the sign-extended 16-bit immediate into Temp, 1 of the 9 instructions is executed, and the result (C) is written back into the register file. Only SRA and LHI may not be self-explanatory: The SRA instruction shifts right while it sign extends the operand and LHI loads the upper 16 bits of the register while zeroing the lower 16 bits. (The C operators << and >> shift left and right, respectively; they fill with zeros unless bits are concatenated explicitly using ##, e.g., sign extension). As mentioned above, the check for overflow in ADD and SUB is not included to simplify the figure. On completion of execution control transfers to Figure 5.13 (page 220).

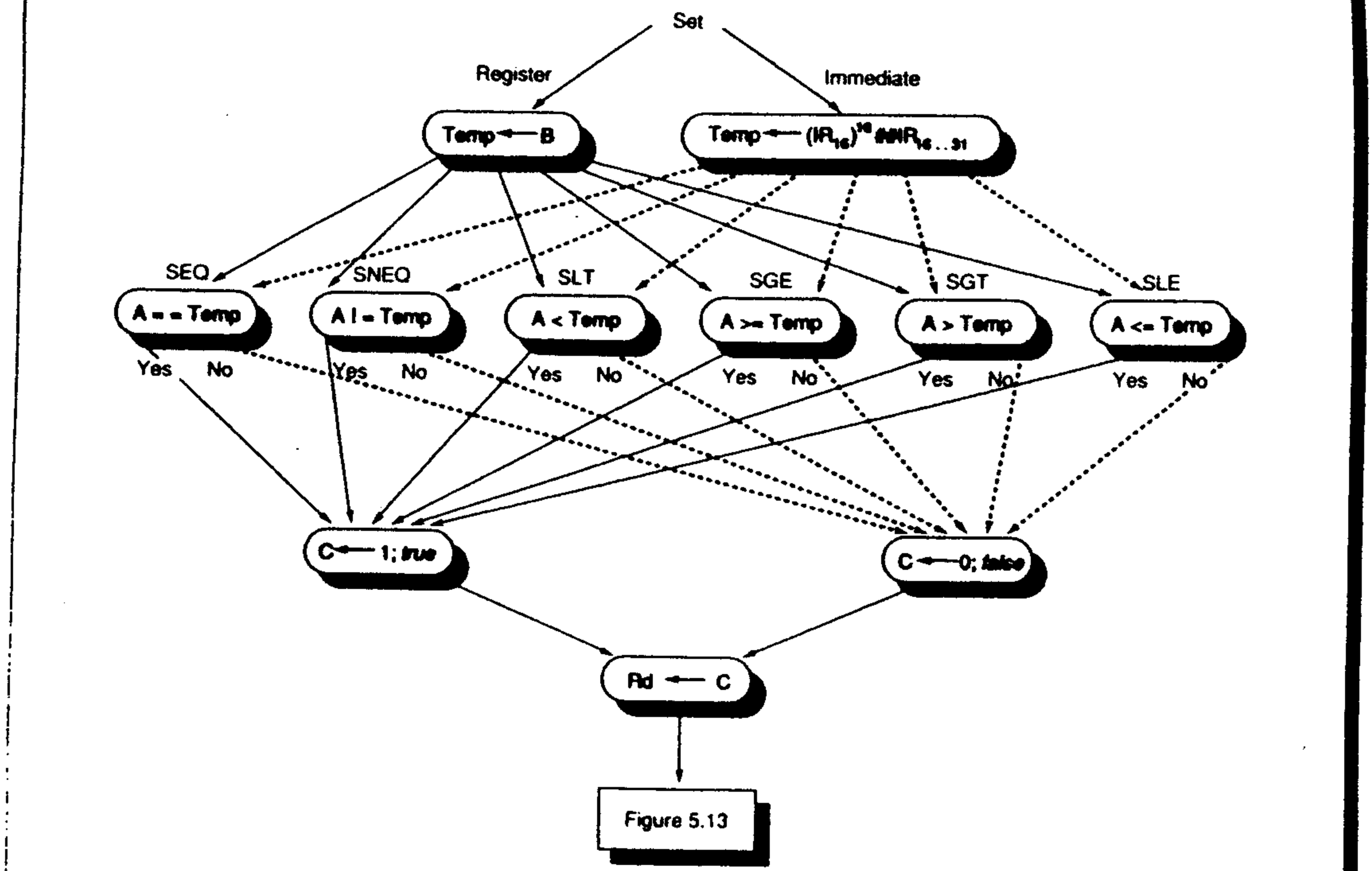

Figure 5.13

FIGURE 5.16 (See adjoining page.) The execution and write-back states for the Set instructions of DLX. After putting a register or the sign-extended 16-bit immediate into Temp, 1 of the 6 instructions compares A to Temp and then sets C to 1 or 0, depending on whether the condition is true or false. C is then written back into the register file, and then execution control transfers to Figure 5.13 (page 220). The dashed lines in this figure and Figure 5.18 are used to make it easier to follow intersecting lines.

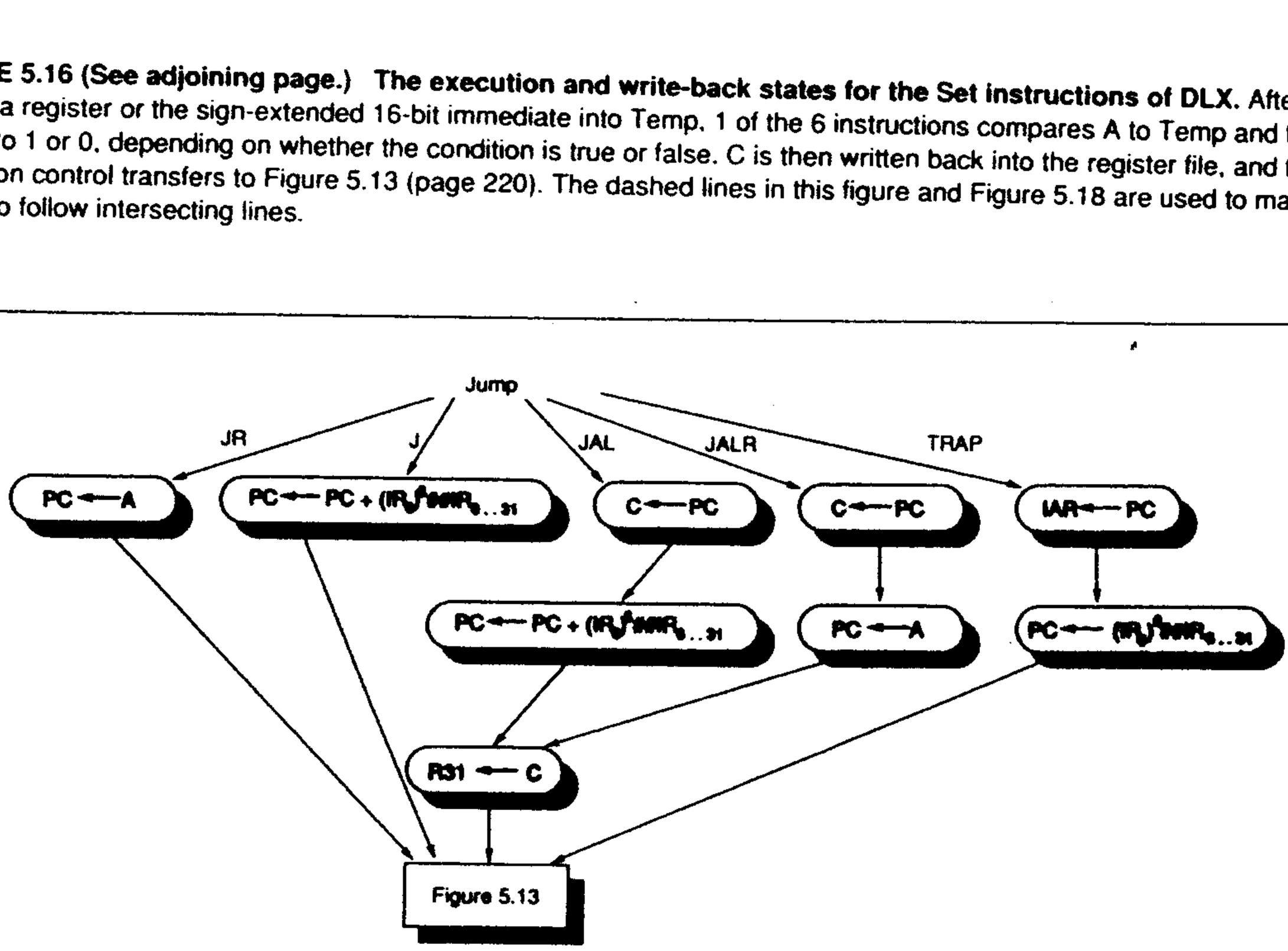

Figure 5.13

FIGURE 5.17 (See adjoining page.) The execution and write-back states for the jump instructions of DLX. With jump and link instructions, the return address is first placed in C before the new value is loaded into PC. Trap saves it in IAR. Note that the immediate in these instructions is 10 bits longer than the 16-bit immediate in all other instructions. Jump and link instructions conclude by writing the return address back into R31. On completion of execution, control transfers to Figure 5.13 (page 220).

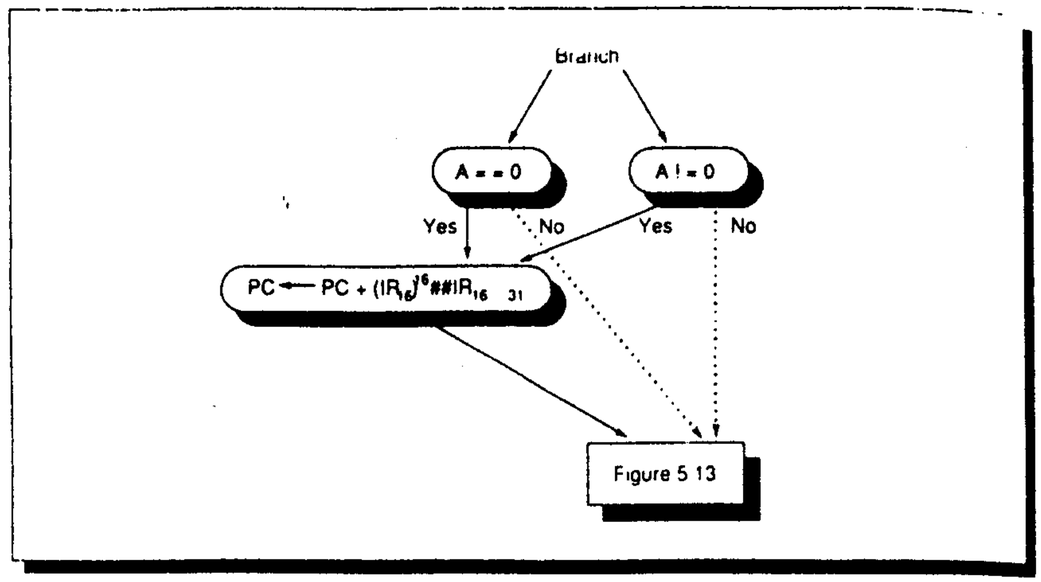

FIGURE 5.18 The execution states for the branch instructions of DLX. The PC is loaded with the sum of the PC and the immediate only if the condition is true. On completion of execution, control transfers to Figure 5.13, found on page 220.

Performance of Hardwired Control for DLX

As stated in Section 5.4, the goal for control designs is to minimize CPI, clock cycle time, amount of control hardware, and development time. CPI is just the average number of states along the execution path of an instruction.

Example

Let's assume that hardwired control directly implements the finite-state diagram in Figures 5.13 to 5.18. What is the CPI for DLX running GCC?

Answer

The number of clock cycles to execute each DLX instruction is determined by simply counting the states of an instruction. Starting at the top, every instruction spends at least two clock cycles in the states in Figure 5.13 (ignoring interrupts). The actual number depends on the average number of times the state accessing memory must repeat because memory is not ready. (These wasted clock cycles are usually called *memory stall cycles* or *wait states*.) In cache-based machines this value is typically 0 (i.e., no repetitions since cache access is 1 cycle) when the data is found in the cache, and 10 or higher when it is not.

The time for the remaining portion of instruction execution comes from the additional figures. Besides two cycles for fetch and decode, loads take four more cycles plus clock cycles waiting for the data access, while stores take just three more clock cycles plus wait states. Three extra clock cycles are also needed by ALU instructions, and set instructions take four. Figure 5.17 shows that jumps take just one extra clock cycle with jump and links taking three. Branches depend on the result: taken branches take 2 more clock cycles, while

DLX instructions	Minimum clock cycles	Memory accesses	Total clock cycles
Loads	6	2	8
Stores	5	2	7
ALU	5	1	6
Set	6	1	7
Jumps	3	1	4
Jump and links	5	1	6
Branch (taken)	4	1	5
Branch (not taken)	3	1	4

FIGURE 5.19 Clock cycles per instruction for DLX categories using the state diagram in Figures 5.13 through 5.18. Determining the total clock cycles per category requires multiplying the number of memory accesses—including instruction fetches—times the average number of wait states, and adding this product to the minimum number of clock cycles. We assume an average of 1 clock cycle per memory access. For example, loads take eight clock cycles if the average number of wait states is one.

untaken branches need just one. Adding these times to the first portion of instruction execution yields the clock cycles per DLX instruction class shown in Figure 5.19.

From Chapter 2, one way to calculate CPI is

$$\text{CPI} = \sum_{i=1}^{n} \left(\text{CPI}_i * \frac{\text{I}_i}{\text{Instruction count}} \right)$$

Using the DLX instruction mix from Figure C.4 in Appendix C for GCC (normalized to 100%), the percentage of taken branches from Figure 3.22 (page 107), and one for the average number of wait states per memory access, the DLX CPI for this datapath and state diagram is calculated:

Loads	8	* 21%	=	1.68
Stores	7	* 12%	=	0.84
ALU	6	* 37%	=	2.22
Set	7	* 6%	=	0.42
Jumps	4	* 2%	=	0.08
Jump and links	6	* 0%	=	0.00
Branch (taken)	5	* 12%	=	0.60
Branch (not taken)	4	* 11%	=	0.44
	Total CPI:			6.28

Thus, the DLX CPI for GCC is about 6.3.

Improving DLX Performance When Control Is Hardwired

As mentioned above, performance is improved by reducing the number of states an instruction must pass through during execution. Sometimes, performance can be improved by removing intermediate calculations that select one of several options, either by adding hardware that uses information in the opcode to later select the appropriate option, or by simply increasing the number of states.

Example

Let's look at improving the performance of ALU instructions by removing the top two states in Figure 5.15 on page 222, which load either a register or an immediate into Temp. One approach uses a new hardware option. Let's call it "X" (see Figure 5.20). The X option selects either the B register or the 16-bit immediate, depending on the opcode in IR. A second approach is simply to increase the number of execution states so that there are separate states for ALU instructions using immediate versus ALU instructions using registers.

For each option, what would be the change in performance, and how should the state diagram be changed? Also, how many states are needed in each option?

Answer

Either change reduces ALU execution time from five to four clock cycles plus wait states. From Figure C.4, ALU operations are about 37% of the instructions for GCC, lowering CPI from 6.3 to 5.9, and making the machine about 7% faster. Figure 5.20 shows Figure 5.15 modified to use the X option instead of the two states that load Temp, while Figure 5.21 simply has many more states to achieve the same result. The total number of states are 50 and 58, respectively.

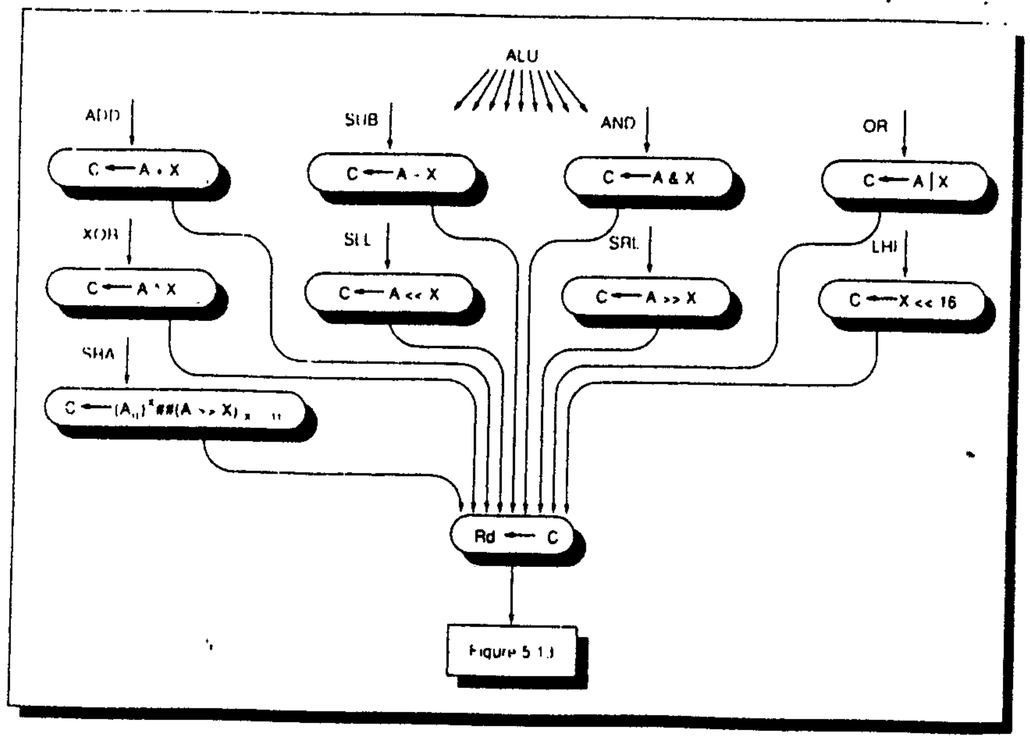

FIGURE 5.20 Figure 5.15 modified to remove the two states loading Temp. The states use the new X option to mean that either B or $(IR_{16})^{16}\#\#IR_{16..31}$ is the operand, depending on the DLX opcode.

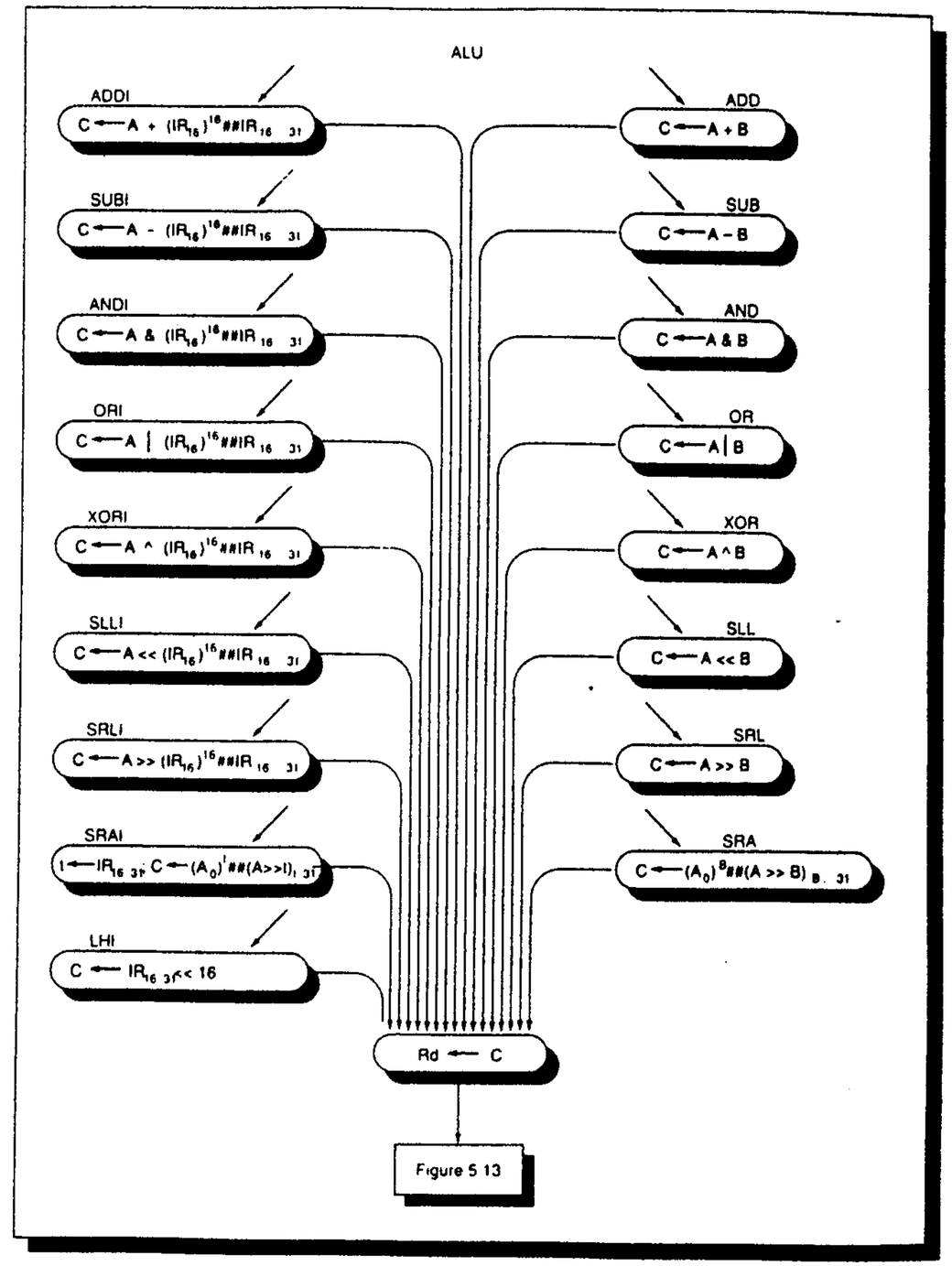

FIGURE 5.21 Figure 5.15 modified to remove the two states loading Temp. Unlike Figure 5.20, this requires no new hardware options in the datapath, but simply more control states.

Control can affect the clock cycle time, either because control itself takes longer than the corresponding operations in the datapath, or because the datapath operations selected by control lengthens the worst-case clock cycle time.

Example

Assume a machine with a 10-ns clock cycle (100-MHz clock rate). Suppose that on closer inspection the designer discovered that all states could be executed in 9 ns, except states that use the shifter. Would it be wise to split those states, taking two 9-ns clock cycles for shift states and one 9-ns clock for everything else?

Answer

Assuming the improvement in the previous example, the average instruction execution time for the 100-MHz machine is 5.9*10 ns or 59 ns. The shifter is only used in the states of four instructions: SLL, SRL, SRA, and LHI (see Figure 5.20). In fact, each of these instructions takes 5 clock cycles (including one wait state for memory access), and only one of the five original clock cycles need be split into two new clock cycles. Thus, the average execution time of these instructions changes from 5*10 ns, or 50 ns, to 6*9 ns, or 54 ns. From Figure C.1 these 4 instructions are about 11% of the instructions executed for GCC (after normalization), making the average instruction execution time 89% * (5.9*9 ns + 11%*54 ns or 53 ns. Thus, splitting the shift state results in a machine that is about 10% faster—a wise decision. (See Exercise 5.8 for a more sophisticated version of this tradeoff.)

Hardwired control is completed by listing the control signals activated in each state, assigning numbers to the states, and finally generating the PLA. Now let's implement control using microcode in a ROM.

Microcoded Control for DLX

A custom format such as this is a slave to the architecture of the hardware and instruction set which it serves. The format must strike a proper compromise between ROM size, ROM-output decoding circuitry size, and machine execution rate.

Jim McKevit et al. [1977]

Before microprogramming can commence, the microinstruction set must be determined. The first step is to list the possible entries for each field of the DLX microinstruction format from Figure 5.6 on page 209. Figure 5.7 on page 211 lists them for the Destination, Source1, and Source2 fields. Figure 5.22 below shows the values for the remaining fields.

Sequencing of microinstructions requires further explanation. The microprogrammed control includes a microprogram counter to specify the address of the next microinstruction if a branch is not taken, as in Figure 5.5 on page 208. In addition to the branches using the Jump address field, three tables are used to decode the DLX macroinstructions. These tables are indexed with the opcodes of the DLX instruction, and supply a microprogram address depending on the value in the opcode. Their use will become clear as we examine the DLX microprogram.

Following the lead of the state diagram, the DLX microprogram is divided into Figures 5.23, 5.25, 5.27, 5.28, and 5.29, with each section of microcode corresponding to one of Figures 5.13 to 5.18 (pages 220–224). The first state in Figure 5.13 becomes the first two microinstructions in Figure 5.23. The first microinstruction (address 0) branches to microinstruction 3 if there is an interrupt pending. Microinstruction 1 fetches an instruction from memory, branching back to itself as long as the memory access is not complete. Microinstruction 2 increments the PC by 4, loads A and B, and then does the first-level decoding. The address of the next microinstruction then depends on which macroinstruction is in the instruction register. The microinstruction addresses for this first-level macroinstruction decode are specified in Figure 5.24. (In reality, the table shown in this figure is specified after the microprogram is written, as both the number of entries and the corresponding locations aren't known until then.)

Opcodes (symbolically specified)	Absolute address	Label	Figure
Memory	5	Mem:	5.25
Move to special	20	Movl2S:	5.25
Move from special	21	MovS2I:	5.25
S2 = B	23	Reg:	5.27
S2 = Immediate	24	Imm:	5.27
Branch equal zero	50	Beq:	5.29
Branch not equal zero	52	Bne:	5.29
Jump	54	Jump:	5.29
Jump register	55	JReg:	5.29
Jump and link	56	JAL:	5.29
Jump and link register	58	JALR:	5.29
Trap	60	Trap:	5.29

FIGURE 5.24 Opcodes and corresponding addresses for decode table 1. The opcodes are shown symbolically on the left, followed by the addresses with the absolute microinstruction address, a label, and the figure where the microcode can be found. If this table were implemented with a ROM it would contain 64 entries corresponding to the 6-bit opcode of DLX. As this would clearly result in many redundant or unspecified entries, a PLA could be used to minimize hardware.

Figure 5.25 contains the DLX load and store instructions. Microinstruction 5 calculates the effective address, and branches to microinstruction 9 if the

macroinstruction in the IR is a load. If not, microinstruction 6 loads MDR with the value to be stored, and microinstruction 7 jumps to itself until the memory is finished writing the data. Microinstruction 8 then jumps back to microinstruction 0 (Figure 5.23) to begin the execution cycle all over again. If the macroinstruction was a load, microinstruction 9 loops until the data has been read. Microinstruction 10 then uses decode table 2 (specified in Figure 5.26) to specify the address of the next microinstruction. Unlike the first decode table, this table is used by other microinstructions. (There is no conflict in multiple uses since the opcodes for each instance are different.)

Suppose the instruction were load halfword. Figure 5.26 shows that the result of decode 2 would be to jump to microinstruction 15. This microinstruction shifts the contents of MDR to the left 16 bits and stores the result in Temp. The following microinstruction shifts Temp right arithmetically 16 bits and puts the result in C. C now contains the 16 rightmost bits of MDR, with the upper 16 bits containing the extended sign. This microinstruction jumps to location 22, which writes C back into the destination register specifier in IR, and then jumps to fetch the next macroinstruction starting at location 0 (Figure 5.23).

Loc	Label	Dest	ALU	S1	S2	C	Misc	Cond	Jump label	Comment
5	Mem:	MAR	ADD	A	imm16			Load?	Load	*Memory instruct.*
6	Store:	MDR	Pass S2		B					*Store*
7	Dloop:						Data write	Mem?	Dloop	
8								Uncond	Ifetch	*Fetch next instr.*
9	Load:						Data read	Mem?	Load	*Load MDR*
10								Decode2		
11	LB:	Temp	SLL	MDR	Constant	24				*Load byte; shift left to remove upper 24 bits*
12		C	SRA	Temp	Constant	24		Uncond	Write1	*Shift right arithmetic to sign extend*
13	LBU:	Temp	SLL	MDR	Constant	24				*LB unsigned*
14		C	SRL	Temp	Constant	24		Uncond	Write1	*Shift right logical*
15	LH:	Temp	SLL	MDR	Constant	16				*Load half*
16		C	SRA	Temp	Constant	16		Uncond	Write1	*Shift right arithmetic*
17	LHU:	Temp	SLL	MDR	Constant	16				*LH Unsigned*
18		C	SRL	Temp	Constant	16		Uncond	Write1	*Shift right logical*
19	LW:	C	Pass S1	MDR				Uncond	Write1	*Load word*
20	Mov2S:	IAR	Pass S1	A				Uncond	Ifetch	*Move to special*
21	MovS2I:	C	Pass S1	IAR						*Move from spec.*
22	Write1:						Rd←C	Uncond	Ifetch	*Write back & go fetch next instruction*

FIGURE 5.25 The section of the DLX microprogram for loads and stores, corresponding to the states in Figure 5.14 (page 221). The microcode for bytes and halfwords takes an extra microinstruction to align the data (see Figure 5.10 page 97). Note that microinstruction 5 loads A from Rd, just in case the instruction is a store. The label Ifetch is for

Opcode	Absolute address	Label	Figure
Load byte	11	LB:	5.25
Load byte unsigned	13	LBU:	5.25
Load half	15	LH:	5.25
Load half unsigned	17	LHU:	5.25
Load word	19	LW:	5.25
ADD	25	ADD/I:	5.27
SUB	26	SUB/I:	5.27
AND	27	AND/I:	5.27
OR	28	OR/I:	5.27
XOR	29	XOR/I:	5.27
SLL	30	SLL/I:	5.27
SRL	31	SRL/I:	5.27
SRA	32	SRA/I:	5.27
LHI	33	LHI:	5.27
Set equal	35	SEQ/I:	5.28
Set not equal	37	SNE/I:	5.28
Set less than	39	SLT/I:	5.28
Set greater than or equal	41	SGE/I:	5.28
Set greater than	43	SGT/I:	5.28
Set less than or equal	45	SLE/I:	5.28

FIGURE 5.26 Opcodes and corresponding addresses for decode tables 2 and 3. The opcodes are shown symbolically on the left, followed by the absolute microinstruction address, the corresponding label, and the figure where the microcode can be found. Since the opcodes are shown symbolically, and they go to the same place in both tables, the same information can be used for specifying decode tables 2 and 3. This similarity is attributable to the immediate version and register version of the DLX instructions sharing the same microcode. If a table were implemented with a ROM, it would contain 64 entries corresponding to the 6-bit opcode of DLX. Again, the many redundant or unspecified entries suggest the use of a PLA to minimize hardware cost.

The ALU instructions are found in Figure 5.27. The first two microinstructions correspond to the states at the top of Figure 5.15 (page 222). After loading Temp with either the register or the immediate, each uses a decode table to vector to the microinstruction that executes the ALU instruction. To save microcode space, the same microinstruction is used whether the operand is a register or an immediate. One of the microinstructions between 25 and 33 is executed, storing its result in C. It then jumps to microinstruction 34, which stores C into the register specified in the IR, and in turn jumps to fetch the next macroinstruction.

Loc	Label	Dest	ALU	S1	S2	C	Misc	Cond	Jump label	Comment
23	Reg:	Temp	Pass S2		B			Decode2		*source2 = reg*
24	Imm:	Temp	Pass S2		Imm			Decode3		*source2 = imm.*
25	ADD/I:	C	ADD	A	Temp			Uncond	Write2	*ADD*
26	SUB/I:	C	SUB	A	Temp			Uncond	Write2	*SUB*
27	AND/I:	C	AND	A	Temp			Uncond	Write2	*AND*
28	OR/I:	C	OR	A	Temp			Uncond	Write2	*OR*
29	XOR/I:	C	XOR	A	Temp			Uncond	Write2	*XOR*
30	SLL/I:	C	SLL	A	Temp			Uncond	Write2	*SLL*
31	SRL/I:	C	SRL	A	Temp			Uncond	Write2	*SRL*
32	SRA/I:	C	SRA	A	Temp			Uncond	Write2	*SRA*
33	LHI:	C	SLL	Temp	Constant	16		Uncond	Write2	*LHI*
34	Write2:						Rd←C	Uncond	Ifetch	*Write back & go fetch next instruction*

FIGURE 5.27 Like the first two states in Figure 5.15 (page 222), microinstructions 23 and 24 load Temp with an operand and then vector to the appropriate microinstruction, depending on the opcode in IR. One of the nine following microinstructions is executed, leaving its result in C. C is written back into the register specified in the register destination field of DLX macroinstruction in IR in microinstruction 34.

Loc	Label	Dest	ALU	S1	S2	C	Misc	Cond	Jump label	Comment
35	SEQ/I:		SUB	A	Temp			Zero?	Set1	*Set equal*
36		C	Pass S2		Constant	0		Uncond	Write4	*A≠T (set to false)*
37	SNE/I:		SUB	A	Temp			Zero?	Set0	*Set not equal*
38		C	Pass S2		Constant	1		Uncond	Write4	*A≠T (set to true)*
39	SLT/I:		SUB	A	Temp			Negative?	Set1	*Set less than*
40		C	Pass S2		Constant	0		Uncond	Write4	*A≥T (set to false)*
41	SGE/I:		SUB	A	Temp			Negative?	Set0	*Set GT or equal*
42		C	Pass S2		Constant	1		Uncond	Write4	*A≥T (set to true)*
43	SGT/I:		RSUB	A	Temp			Negative?	Set1	*Set greater than*
44		C	Pass S2		Constant	0		Uncond	Write4	*T≥A (set to false)*
45	SLE/I:		RSUB	A	Temp			Negative?	Set0	*Set LT or equal*
46		C	Pass S2		Constant	1		Uncond	Write4	*T≥A (set to true)*
47	Set0:	C	Pass S2		Constant	0		Uncond	Write4	*Set to 0 = false*
48	Set1:	C	Pass S2		Constant	1				*Set to 1 = true*
49	Write4:						Rd←C	Uncond	Ifetch	*Write back & fetch next instruction*

FIGURE 5.28 Corresponding to Figure 5.16 (pages 222–223), this microcode performs the DLX Set instructions. As in the previous figure, to save space these same microinstructions execute either the version of set using registers or the version using immediates. The tricky microcode is found in microinstructions 43 and 45, where the subtraction Temp – A is unlike the earlier microcode. Remember that A–$_r$ Temp = Temp – A (see Figure 5.22 on page 229).

Loc	Label	Dest	ALU	S1	S2	C	Misc	Cond	Jump label	Comment
50	Beq:		SUB	A	Constant	0		0?	Branch	*Instr is branch =0*
51								Uncond	Ifetch	*≠0: not taken*
52	Bne:		SUB	A	Constant	0		0?	Ifetch	*Instr is branch ≠0*
53	Branch:	PC	ADD	PC	imm16			Uncond	Ifetch	*≠0: taken*
54	Jump:	PC	ADD	PC	imm26			Uncond	Ifetch	*Jump*
55	JReg:	PC	Pass S1	A				Uncond	Ifetch	*Jump register*
56	JAL:	C	Pass S1	PC						*Jump and link*
57		PC	ADD	PC	imm26		R31←C	Uncond	Ifetch	*Jump & save PC*
58	JALR:	C	Pass S1	PC						*Jump & link reg*
59		PC	Pass S1	A			R31←C	Uncond	Ifetch	*Jump & save PC*
60	Trap:	IAR	Pass S1	PC						*Trap*
61		PC	Pass S2		imm26			Uncond	Ifetch	

FIGURE 5.29 The microcode for branch and jump DLX instructions, corresponding to the states in Figures 5.17 and 5.18 on pages 222–224.

Figure 5.28 corresponds to the states in Figure 5.16 (pages 222–223), except that the top two states that load Temp are microinstructions 23 and 24 of the previous figure; the decode tables will either jump to locations 25 to 34 in Figure 5.27, or 35 to 45 in Figure 5.28, depending on the opcode. The microinstructions for Set perform relative tests by having the ALU subtract Temp from A and then test the ALU output to see if the result is zero or negative. Depending on the test result, C is set to 1 or 0 and written back in the register file before going to fetch the next macroinstruction. Tests for A = Temp, A ≠ Temp, A < Temp, and A ≥ Temp are straightforward using these conditions on the ALU output A – Temp. A > Temp and A ≤ Temp, on the other hand, are not simple, but can be done using the negative condition with the subtraction reversed:

$$(\text{Temp} - \text{A} < 0) = (\text{Temp} < \text{A}) = (\text{A} > \text{Temp})$$

If the result is negative, then A > Temp, otherwise A ≤ Temp. Voila!

Figure 5.29 contains the last of the DLX microcode and corresponds to the states found in Figures 5.17 and 5.18 (pages 222–224). Microinstruction 50, corresponding to the macroinstruction branch on equal zero, tests if A equals zero. If it does, the macroinstruction branch succeeds, and the microinstruction jumps to the microinstruction 53. This microinstruction loads the PC with the PC-relative address and then jumps to the microcode that fetches the new macroinstruction (location 0). If A does not equal zero, the macroinstruction branch fails, so that the next sequential microinstruction (51) executes, jumping to location 0 without changing the PC.

A state usually corresponds to a single microinstruction, although in a few cases above two microinstructions were needed. The jump and link instructions have the reverse case, with two states collapsing into one microinstruction. The actions in the last two states of jump and link in Figure 5.17 are found in microinstruction 57, and similarly for the jump and link register with microinstruction 59. These microinstructions load the PC with the PC-relative branch address and save C into R31.

Performance of Microcoded Control for DLX

Before trying to improve performance or reduce costs of control, the existing performance must be assessed. Again, the process is to count the clock cycles for each instruction, but this time there is a larger variety in performance.

All instructions execute microinstructions 0, 1, and 2 in Figure 5.23 (page 230), giving a base of 3 clocks plus wait states, depending on the repetition of microinstruction 1. The clock cycles for the rest of the categories are:

4 for stores, plus wait states

5 for load word, plus wait states

6 for load byte or load half (signed or unsigned), plus wait states

3 for ALU

4 for set

2 for branch equal zero (taken or untaken)

2 for branch not equal zero (taken)

1 for branch not equal zero (untaken)

1 for jumps

2 for jump and links

Using the instruction mix for GCC in Figure C.4, and assuming an average of 1 wait state per memory access, the CPI is 7.68. This is higher than the hardwired control CPI, because the test for interrupt takes another clock cycle at the beginning, loads and stores are slower, and branch equal zero is slower for the untaken case.

Reducing Cost and Improving Performance of DLX When Control Is Microcoded

The size of a completely unencoded version of the DLX microinstruction is calculated from the number of entries in Figures 5.7 (page 211) and 5.22 (page 229) plus the size of the Constant and Jump address fields. The largest constant in the fields is 24, which requires 5 bits, and the largest address is 61, which requires 6. Figure 5.30 shows the microinstruction fields, the unencoded widths, and the encoded widths. Encoding almost halves the size of control store.

	Dest	ALU operation	Source1	Source2	Constant	Misc	Cond	Jump address	Total
Unencoded	7	11	9	9	5	6	10	6	= 63 bits
Encoded	3	4	4	4	5	3	4	6	= 33 bits

FIGURE 5.30 Width of field in bits of unencoded and encoded microinstruction formats. Note that the Constant and Jump address fields are not encoded in this example, placing fewer restrictions on the microprogram using the encoded format.

The microinstruction can be further shrunk by introducing multiple microinstruction formats and by combining independent fields.

Example

Figure 5.31 shows an encoded version of the original DLX microinstruction format and the version with two formats: one for ALU operations and one for miscellaneous and branch operations. A bit is added to distinguish the two formats. The ALU/Jump (A/J) microinstruction performs the ALU operation specified in the microinstruction; the address of the next microinstruction is specified in the Jump address. For the Transfer/Misc/Branch (T/M/B) microinstruction, the ALU performs Pass S1, while the Misc and Cond fields specify the rest of the operations. The primary change in interpretation of the fields in the new formats is that the ALU condition being tested in the T/M/B format refers to the ALU output from the *prior* A/J microinstruction since there is no ALU operation in T/M/B format. In both formats the Constant and Jump field are combined into a single field under the assumption they are not used at the same time. (For the A/J format, the appearance of a constant in a source field results in fetching the following microinstruction.) The new formats shrink width from the original 33 bits to 22 bits, but the actual size savings depends on the number of extra microinstructions needed because of the reduced options.

What is the increase in number of microinstructions, compared to the single format, for the microcode in Figure 5.23 (page 230)?

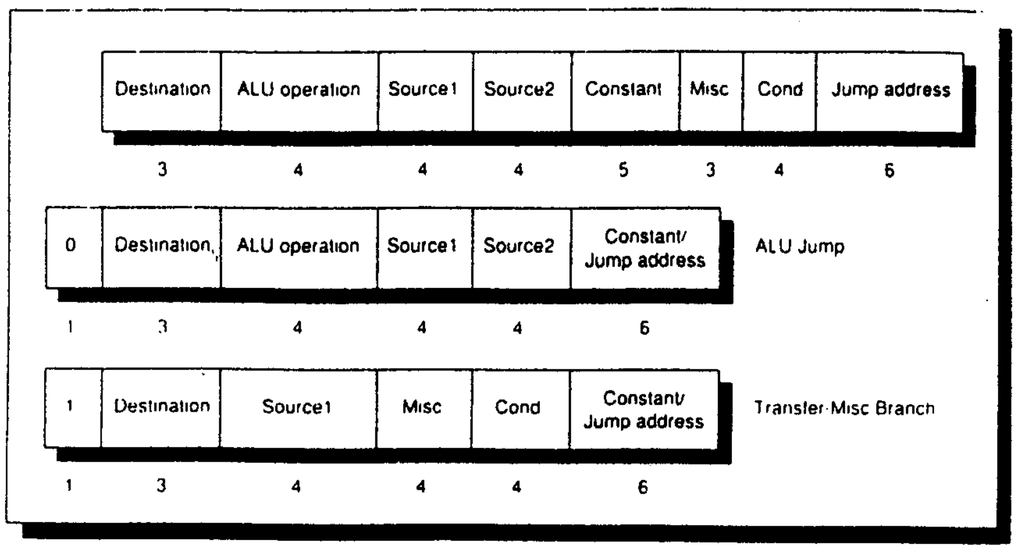

FIGURE 5.31 The original DLX microinstruction format at the top and the dual-format version below. Note that the Misc field is expanded from 3 to 4 bits in the T/M/B to make the two formats the same length.

Answer

Figure 5.32 shows the increase in the number of microinstructions over Figure 5.23 (page 230) because of the restrictions of each format. The five microinstructions in the original format expand to six in the new format. Microinstruction 2 is the only one that expands to two microinstructions for this example.

Sometimes performance can be improved by finding faster sequences of microcode, but normally it requires changes to the hardware. The branch equal zero instruction takes one extra clock cycle when the branch is not taken with hardwired control, but two with microcoded control; while branch not equal zero has the same performance for hardwired and microcoded control. Why would the former differ in performance? Figure 5.29 shows that microinstruction 52 branches on zero to fetch the next microinstruction, which is correct for the branch on not equal zero macroinstruction. Microinstruction 50 also tests for zero for the branch on zero macroinstruction and branches to the microinstruction that loads the new PC. The not zero case is handled by the following microinstruction (51), which jumps to fetch the next instruction—hence, one clock cycle for untaken branch on not equal zero and two for untaken branch on equal zero. One solution is simply to add "not zero" to the microcode branch conditions in Figure 5.22 (page 229) and change the branch on equal microcode to the version in Figure 5.33. Since there are only ten branch conditions, adding the eleventh would not require more than the four bits needed for an encoded version of that field.

This change drops the CPI from 7.68 to 7.63 for microcoded control, yet this is still higher than the CPI for hardwired control.

Example

Let's improve microcoded control so that the CPI for GCC is closer to the original CPI under hardwired control.

Answer

The main performance culprit is the separate test for interrupts in Figure 5.23. By modifying the hardware, decode1 can kill two birds with one stone: In addition to jumping to the appropriate microinstructions corresponding to the opcode, it also jumps to the interrupt microcode if an interrupt is pending. Figure 5.34 shows the revised microcode. This modification saves one clock cycle from each instruction, reducing the CPI to 6.63.

Value	ALU		Misc			Cond	
0	ADD	+	Instr Read	*IR←M[PC]*	- - -		*Go to next sequential microinstruction*
1	SUB	-	Data Read	*MDR←M[MAR]*	Uncond		*Always jump*
2	RSUB (reverse sub)	$-_r$	Write	*M[MAR]←MDR*	Int?		*Pending (between instruction) interrupt?*
3	AND	&	AB← RF	*Load A&B from Reg. File*	Mem?		*Memory access not complete?*
4	OR	|	Rd←C	*Write Rd*	Zero?		*Is the ALU output zero?*
5	XOR	^	R31←C	*Write R31 (for call)*	Negative?		*Is the ALU output less than zero?*
6	SLL	<<			Load?		*Is the macroinstruction a DLX load?*
7	SRL	>>			Decode1 (Fig. 5.24)		*Address table 1 determines next micro-instruction (uses main opcode)*
8	SRA	$>>_a$			Decode2 (Fig. 5.26)		*Address table 2 determines next micro-instruction (uses "func" opcode)*
9	Pass S1	*S1*			Decode3 (Fig. 5.26)		*Address table 3 determines next micro-instruction (uses main opcode)*
10	Pass S2	*S2*					

FIGURE 5.22 The options for three fields of the DLX microinstruction format in Figure 5.6 on page 209. The possible names are shown on the left of the field name, with an explanation of each field to the right. The real microinstruction would contain a bit pattern corresponding to the number in the first column. Combined with Figure 5.7 (page 211), all the fields are defined except the Constant and Jump address fields, which contain numbers supplied by the microprogrammer. $>>_a$ is an abbreviation for shift right arithmetic and $-_r$ means reverse subtract ($B -_r A = A - B$).

Loc	Label	Dest	ALU	S1	S2	C	Misc	Cond	Jump label	Comment
0	Ifetch:							Interrupt?	Intrpt	*Check interrupt*
1	Iloop:						Instr Read	Mem?	Iloop	*IR ←M[PC], wait for memory*
2		PC	ADD	PC	Constant	4	AB←RF	Decode1		
3	Intrpt:	IAR	Pass S1	PC						*Interrupt*
4		PC	Pass S2		Constant	0		Uncond	Ifetch	*PC←0 & go fetch next instruction*

FIGURE 5.23 The first section of the DLX microprogram, corresponding to the states in Figure 5.13 (page 220). The first column contains the absolute address of the microinstruction, followed by a label. The rest of the fields contain values from Figures 5.7 (page 211) and 5.22 for the microinstruction format in Figure 5.6 (page 209). As an example, microinstruction 2 corresponds to the second state of Figure 5.13. It sends the output from the ALU into PC, tells the ALU to add, puts PC onto the Source1 bus, and a constant from the microinstruction (whose value is 4) onto the Source2 bus. In addition, A and B are loaded from the register file according to the specifiers in IR. Finally, the address of the next microinstruction to be executed comes from decode table 1 (Figure 5.24), which depends on the opcode in the instruction register (IR).

Loc	Label	Type	Dest	ALU	S1	S2	Misc	Cond	Const/ Jump	Comment
0	Ifetch:	M/T/B		---		---		Interrupt?	Intrpt	*Check interrupt*
1	Iloop:	M/T/B		---		---	Instr Read	Mem?	Iloop	*IR ←M[PC]: wait for memory*
2		A/J	PC	ADD	PC	Constant	---	---	4	*Increment PC*
3		M/T/B		---		---	AB← RF	Decode1		
4	Intrpt	A/J	IAR	Pass S1	PC		---	---	5	*Interrupt*
5		A/J	PC	SUB	Temp	Temp	---	---	Ifetch	*PC←0 (1 minus 1=0) & go fetch next instruction*

FIGURE 5.32 Version of Figure 5.23 (page 230) using the dual-format microinstruction in Figure 5.31. Note that ALU Jump microinstructions check the S1 and S2 fields for a constant specifier to see if the next address is sequential (as in microinstruction 2), otherwise they go to the Jump address (as in microinstructions 4 and 5). The microprogrammer changed the last microinstruction to generate a zero by subtracting a register from itself rather than through straightforward use of constant 0. Using the constant would have required an additional microinstruction since this format goes to the next sequential instruction if a constant is used. (See Figure 5.31.)

Loc	Label	Dest	ALU	S1	S2	C	Misc	Cond	Jump label	Comment
50	Beq:		SUB	A	Constant	0		not 0?	Ifetch	*Branch =0*
51		PC	ADD	PC	imm16			Uncond	Ifetch	*=0: taken*

FIGURE 5.33 Branch not equal microcode from Figure 5.29 (page 234) rewritten by using a not zero condition in microinstruction 44.

Loc	Label	Dest	ALU	S1	S2	C	Misc	Cond	Jump label	Comment
0	Ifetch:						Instr Read	Mem?	Ifetch	*IR ←M[PC]: wait for memory*
1		PC	ADD	PC	Constant	4	AB←RF	Decode1		*Also go to interrupt if pending interrupt*
2	Intrpt:	IAR	SUB	PC	Constant	4				*Interrupt: undo PC increment*
3		PC	Pass S2		Constant	0		Uncond	Ifetch	*PC←0 & go fetch next instruction*

FIGURE 5.34 Revised microcode that takes advantage of a change of the hardware to have decode1 go to microinstruction 2 if there is a pending interrupt. This microinstruction must reverse the increment of PC in the prior microinstruction so that the correct value is saved.