byte of the instruction is not in main memory—a situation that requires the saved PC to point 50 bytes earlier. Imagine the difficulties of restarting an instruction with six operands, each of which could be misaligned and thus be partially in memory and partially on disk!

The instructions that are hardest to restart are those that modify some of the machine state before it is known whether interrupts can occur. The VAX autoincrement and autodecrement addressing modes would naturally modify registers during the addressing phase of execution rather than at the writeback phase, and so would be vulnerable to this difficulty. To avoid this problem, recent VAXes keep a history queue of the register specifiers and the operations on the registers, so that the operations can be reversed on an interrupt. Another approach, used on the earlier VAXes, is to record the specifiers and the original values of the registers, restoring the original values on interrupt. (The primary difference is that it only takes a few bits to record how the address was changed due to autoincrement or autodecrement versus the full 32-bit register value.)

It is not just addressing modes that make the VAX difficult to restart; long-running instructions mean that interrupts must be checked in the middle of execution to prevent long interrupt latency. `MOVC3`, for example, copies up to 2^{16} bytes and can take tens of milliseconds to finish—far too long to wait for an urgent event. On the other hand, even if there were a way to undo copying in the middle of execution so that `MOVC3` could be restarted, interrupts would occur so frequently, relative to this long-running instruction (see Figure 5.10 on page 216), that `MOVC3` would be restarted repeatedly under those conditions. Such wasted effort from incomplete copies would render `MOVC3` worse than useless.

DEC divided the problem to conquer it. First, the operands—source address, length, and destination address—are fetched from memory and placed into general-purpose registers R1, R2, and R3. If an interrupt occurs during this first phase, these registers are restored, and the `MOVC3` is restarted from scratch. After this first phase, every time a byte is copied, the length (R2) is decremented and addresses (R1 and R3) are incremented. If an interrupt occurs during this second phase, `MOVC3` sets the *first part done* (FPD) bit in the program status word. When the interrupt is serviced and the instruction is reexecuted, it first checks the FPD bit to see if the operands have already been placed in registers. If so, the VAX doesn't fetch the address and length operands, but just continues with the current values in the registers, since that is all that remains to be copied. This permits more rapid response to interrupts while allowing long-running instructions to make progress between interrupts.

IBM had a similar problem. The 360 included the `MVC` instruction, which copies up to 256 bytes of data. For the early machines without virtual memory, the machine simply waited until the instruction was completed before servicing interrupts. With the inclusion of virtual memory in the 370, the problem could no longer be ignored. Control first tries to access all possible pages, forcing all possible virtual memory miss interrupts to occur before moving any data. If any interrupts occur in this phase, the instruction is restarted. Control then ignores interrupts until the instruction is complete. To allow longer copies, the 370

includes MVCL, which can move up to 2^{24} bytes. The operands are in registers and are updated as a part of execution—like the VAX, except that there is no need for FPD since the operands are always in registers. (Or, to speak historically, the VAX solution is like the IBM 370, which came first.)

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.)

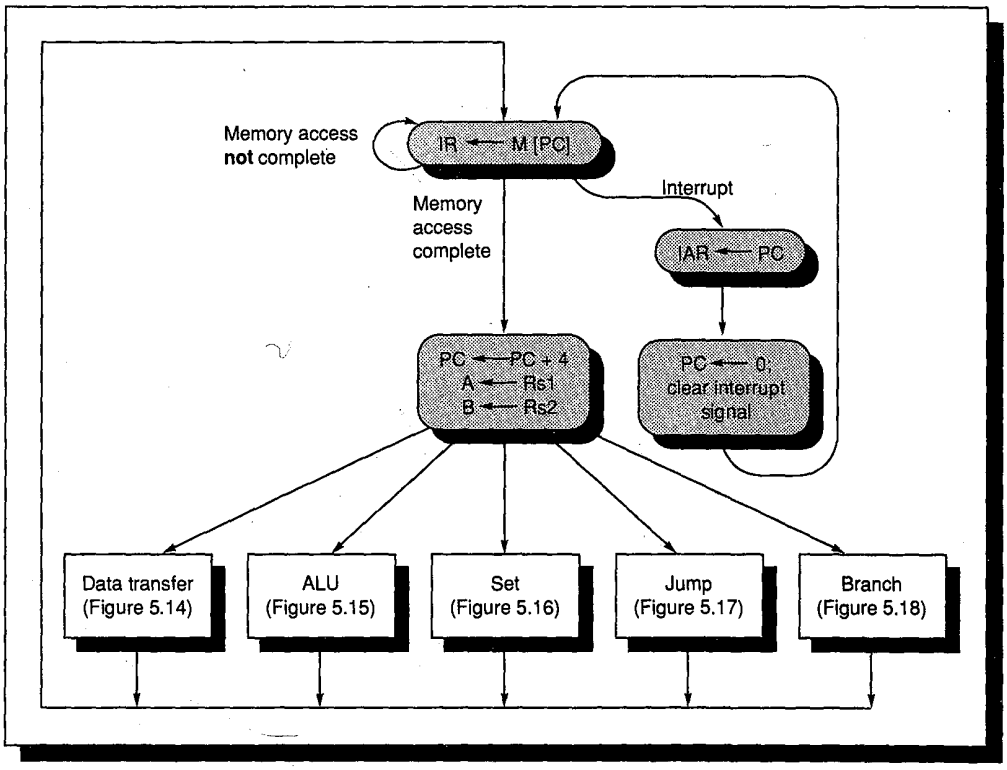

FIGURE 5.13 The top-level view of the DLX finite-state diagram for the non-floating-point instructions. The first two steps of instruction execution—instruction fetch and instruction decode/register fetch—are shown. The first state repeats until the instruction is fetched from memory or an interrupt 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 directly without going through MAR (Figure 5.4 on page 207).

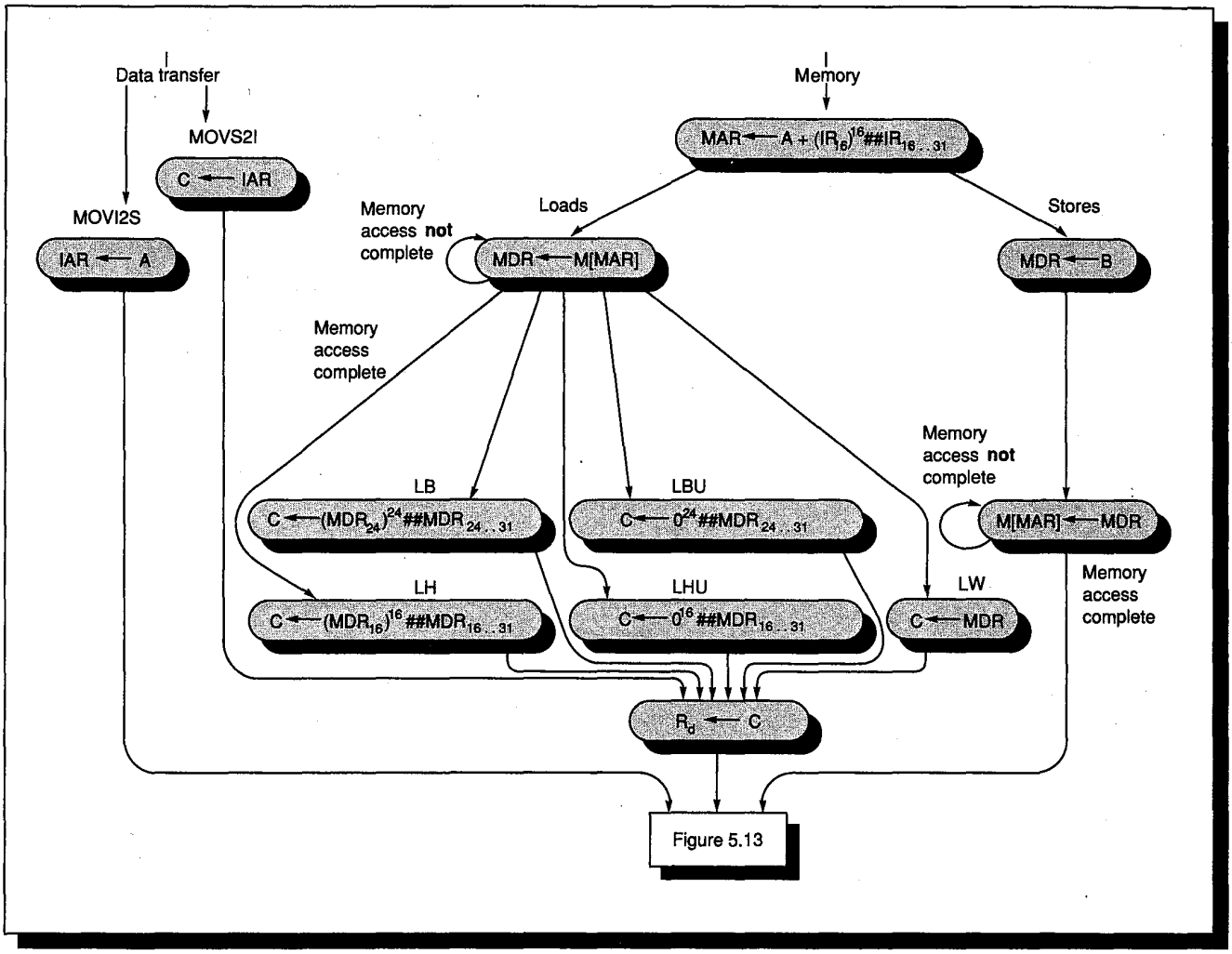

FIGURE 5.14 The effective address calculation, memory-access, and write-back states for the memory-access and data-transfer instructions of DLX. For loads, the second state repeats until the data is fetched from memory. The final state of stores repeats until the write is complete. While the operation 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.

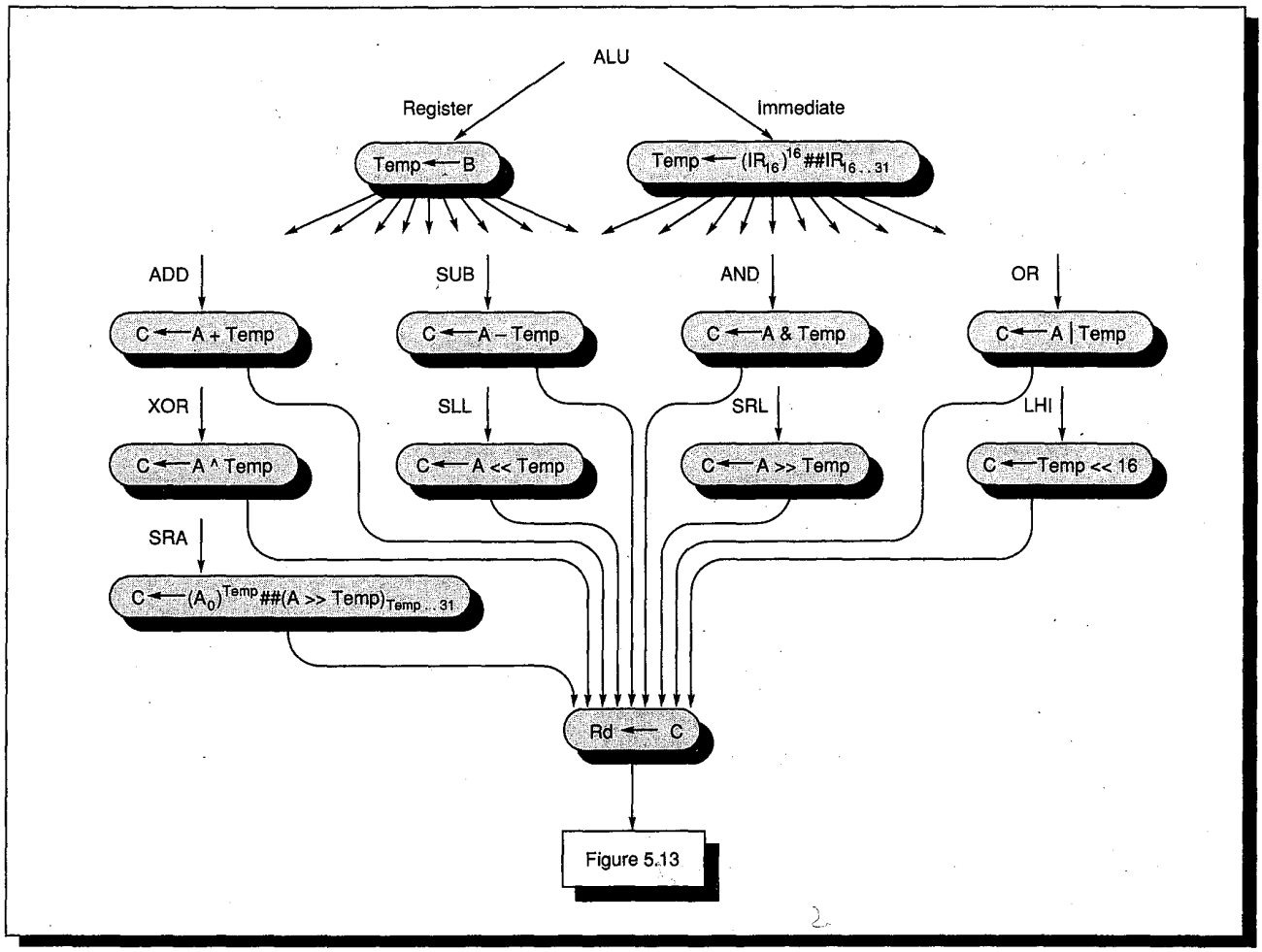

FIGURE 5.15 The execution and write-back states for the ALU instructions of DLX. After putting a register or the sign-extended 16-bit immediate into Temp, 1 of the 9 instructions is executed, and the result (C) is written back into the register file. Only `SRA` and `LHI` may not be self-explanatory: 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 5.16 (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 5.17 (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).

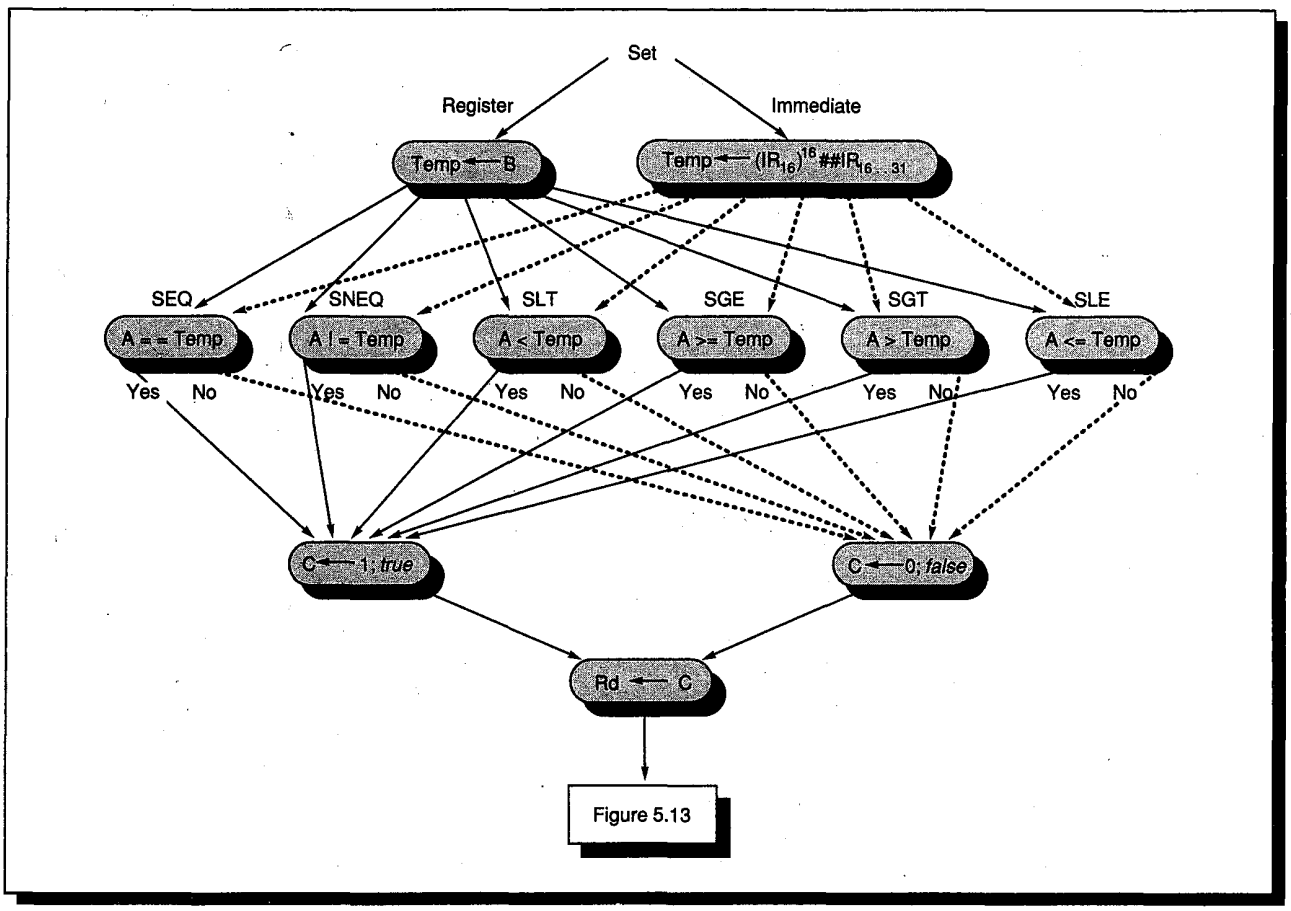

FIGURE 5.16

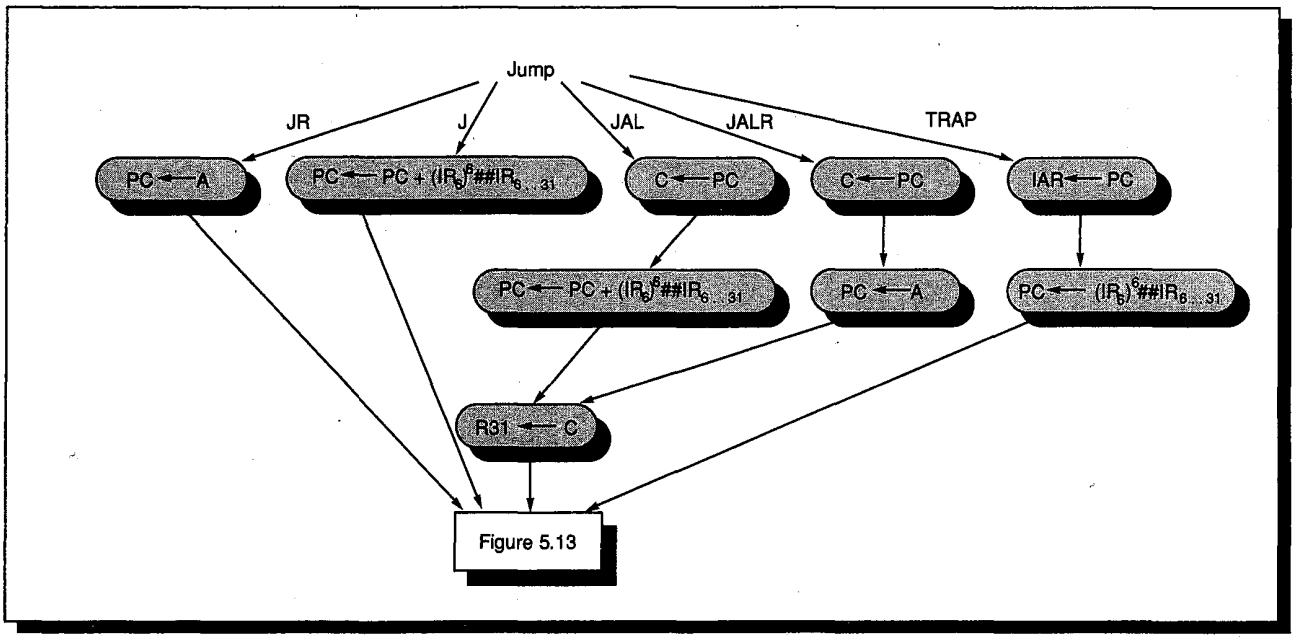

FIGURE 5.17

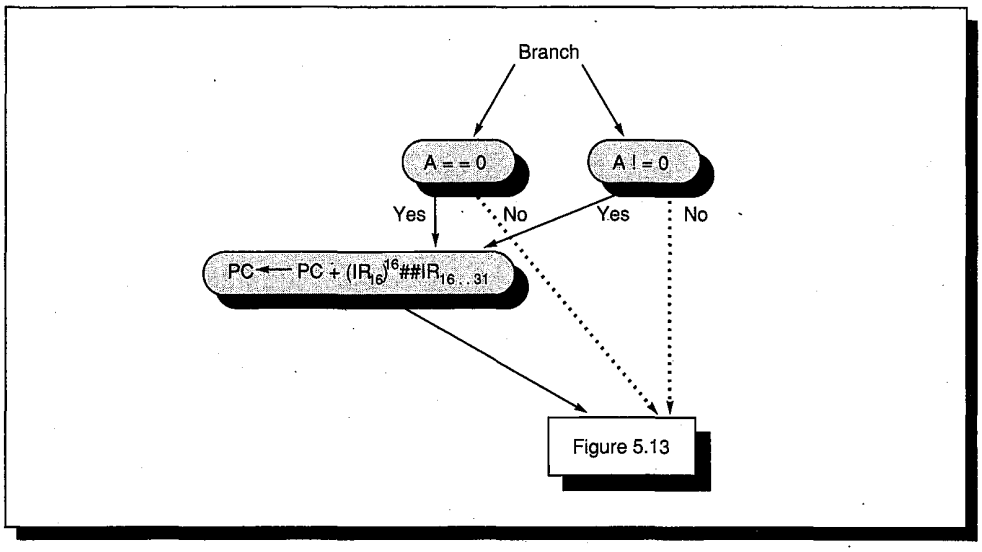

FIGURE 5.18 The execution states for the branch instructions of DLX. The PC is loaded with the sum of the PC and the immediate only if the condition is true. On completion of execution, control transfers to Figure 5.13, found on page 220.

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 use two 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.

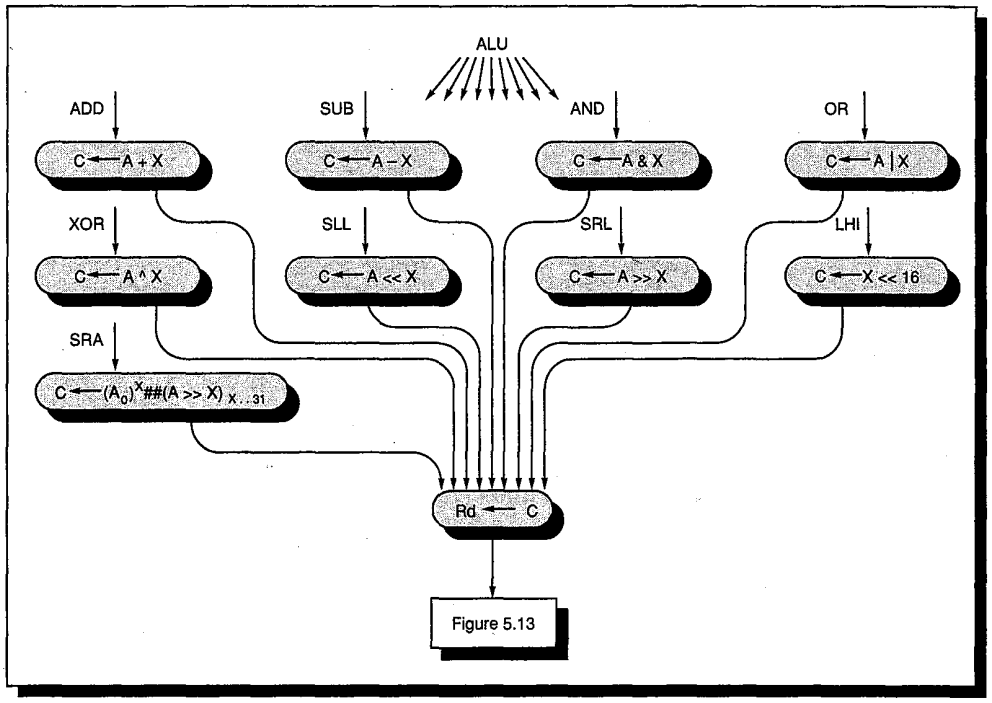

FIGURE 5.20 Figure 5.15 modified to remove the two states loading Temp. The states use the new X option to mean that either B or $(IR_{16})^{16}\#\#IR_{16..31}$ is the operand, depending on the DLX opcode.

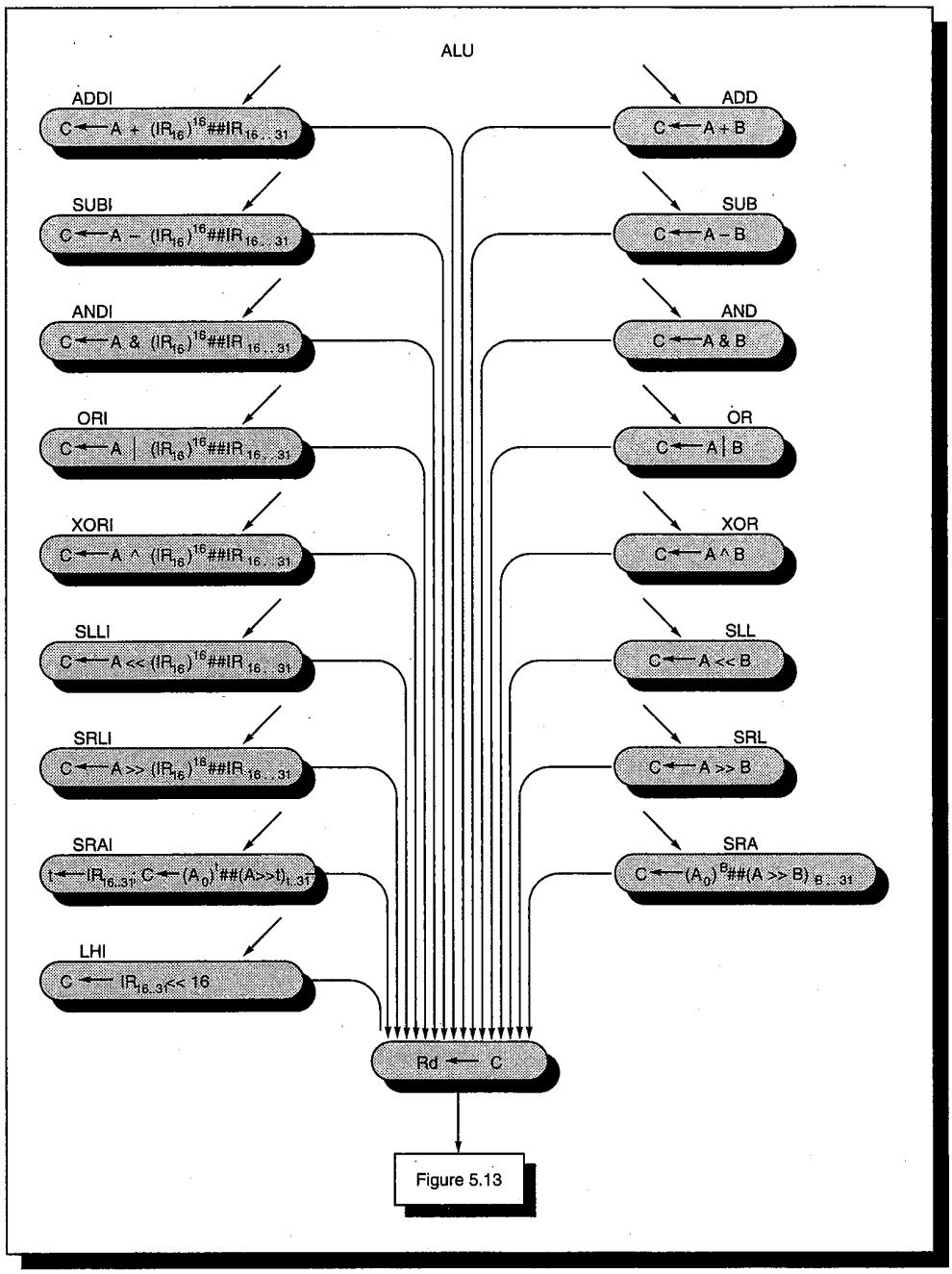

FIGURE 5.21 Figure 5.15 modified to remove the two states loading Temp. Unlike Figure 5.20, this requires no new hardware options in the datapath, but simply more control states.

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.4 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.

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 ($B -_r A = A - B$).

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.)

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).

Opcodes (symbolically specified)	Absolute address	Label	Figure
Memory	5	Mem:	5.25
Move to special	20	MovI2S:	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	MovI2S:	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 3.10, page 97). Note that microinstruction 5 loads A from Rd, just in case the instruction is a store. The label Ifetch is for microinstruction 0 in Figure 5.23 on page 230.

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).

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.

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.

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 operations 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 fields 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)?

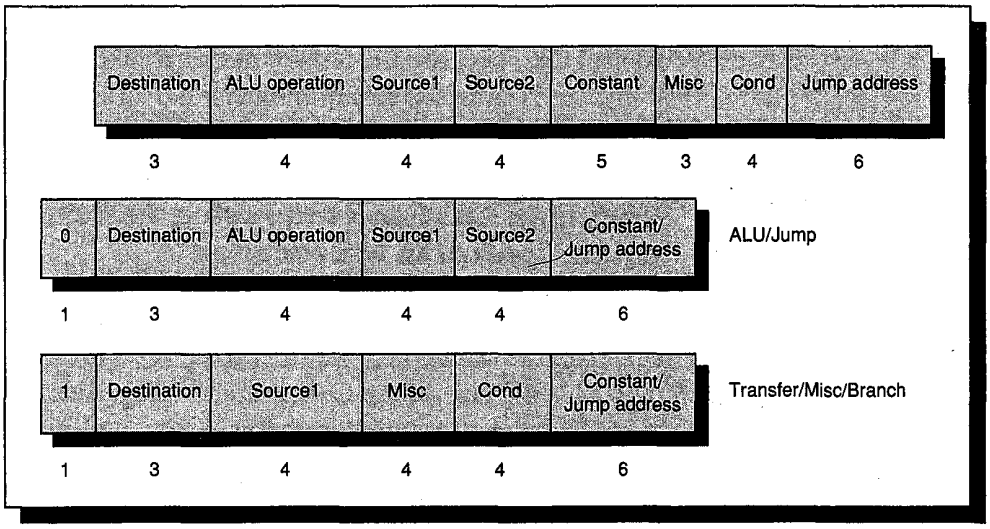

FIGURE 5.31 The original DLX microinstruction format at the top and the dual-format version below. Note that the Misc field is expanded from 3 to 4 bits in the T/M/B to make the two formats the same length.

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.

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 (t minus t=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.)

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.

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.

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.

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.

5.8 Fallacies and Pitfalls

Pitfall: Microcode implementing a complex instruction may not be faster than macrocode.

At one time, microcode had the advantage of being fetched from a much faster memory than macrocode. Since caches came into use in 1968, microcode no longer has such a consistent edge in fetch time. Microcode does, however, still have the advantage of using internal temporary registers in the computation, which can be helpful on machines with few general-purpose registers. The disadvantage of microcode is that the algorithms must be selected before the machine is announced and can't be changed until the next model of the archi-

tecture; macrocode, on the other hand, can utilize improvements in its algorithms at any time during the life of the machine.

The VAX Index instruction provides an example: The instruction checks to see if the index is between two bounds, one of which is usually zero. The VAX-11/780 microcode uses two compares and two branches to do this, while macrocode can perform the same check in one compare and one branch. The macrocode checks the index against the upper limit using **unsigned** comparisons, rather than two's complement comparisons. This treats a negative index (less than zero and so failing the comparison) as if it were a very large number, thus exceeding the upper limit. (The algorithm can be used with nonzero lower bounds by first subtracting the lower bound from the index.) Replacing the index instruction by this VAX macrocode always improves performance on the VAX-11/780.

Fallacy: If there is space in control store, new instructions are free of cost.

Since the length of control store is usually a power of two, at times there may be unused control store available to expand the instruction set. The analogy here is that of building a house and discovering, near completion, that you have enough land and materials left to add a room. This room wouldn't be free, however, since there would be the costs of labor and maintenance for the life of the home. The temptation to add "free" instructions can only occur when the instruction set is not fixed, as is likely to be the case in the first model of a computer. Because instruction set compatibility is a long-term requirement, all future models of this machine will be forced to include these "free" instructions, even if space is later at a premium. This expansion also ignores the cost of a longer development time to test the added instructions, as well as the possibility of costs of repairing bugs in them after the hardware is shipped.

Fallacy: Users find writable control store helpful.

Bugs in microcode persuaded designers of minicomputers and mainframes that it would be wiser to use RAM than ROM for control store. Doing so would enable microcode bugs to be repaired by shipping customers floppy disks rather than by having the field engineer pull boards and replace chips. Some customers and some manufacturers also decided that users should be allowed to write microcode; this opportunity became known as *writable control store* (WCS). Yet by the time WCS was offered, the world had changed to make WCS less attractive than originally envisioned:

- The tools for writing microcode were much poorer than those for writing macrocode. (The authors and many others stepped into that breach to provide better microprogramming tools.)
- At a time when main memory was expanding, WCS was limited to 1–4KB microinstructions. (Few programming tasks are harder than forcing code into too small a memory.)

- Microcoded control became increasingly tailored to the native macroinstruction set, making microprogramming less useful for tasks other than that for which it was intended.
- With the advent of timesharing, programs might run for only milliseconds before switching to other tasks. This meant that WCS would have to be swapped if more than one program needed it, and reloading WCS could easily take longer than a few milliseconds.
- Timesharing also meant that programs had to be protected from each other. Because, at such a low level, microprograms can circumvent all protection barriers, microprograms written by users were notoriously untrustworthy.
- The increasing demand for virtual memory meant that microprograms had to be restartable—any memory access could force the computation to be shelved.
- Finally, companies like DEC that offered WCS provided no customer support for those who wanted to write microcode.

Many customers ordered WCS, but few benefited from it. The death of WCS has been by a thousand small cuts, and WCS is not an option on current computers.

5.9 Concluding Remarks

In his first paper [1953] Wilkes identified advantages of microprogramming that still hold true today. One of these advantages is that microprogramming helps accommodate change. This can happen late in the development cycle, where simply changing some 0s to 1s in the control store can sometimes save redesigning hardware. A related advantage is that by emulating other instruction sets in microcode, software compatibility is simplified. Microprogramming also reduces the cost of adding more complex instructions to a standard microarchitecture to just the cost of a few more words of control store (although there is the pitfall that once an instruction set is created assuming microprogrammed control, it is difficult to ever build a machine without using it). This flexibility allows hardware construction to begin before the instruction set and microcode have been completely written, because specifying control is just a matter of programming. Finally, microprogramming now has the further advantage of having a large set of tools that have been developed to help write, edit, assemble, and debug microcode.

The drawback of microcode has always been performance. This is because microprogramming is a slave to memory technology: The clock cycle time is limited by the time to read microinstructions from control store. In the 1950s, microprogramming was impractical since virtually the only technology available for control store was the same one used for main memory. In the late 1960s and

early 1970s, semiconductor memory was available for control store, while main memory was constructed from core. The factor of ten in cycle time that differentiated the two technologies opened the door for microcode. The popularity of cache memory in the 1970s once again closed this gap, and machines were again built with the same technology for control store and memory.

For these reasons instruction sets invented since 1985 have not relied on microcode. Though no one likes to predict the future—least of all in writing—it is the authors' opinion that microprogramming is bound to memory technology. If in some future technology ROM becomes much faster than RAM, or if caches are no longer effective, microcode may regain its popularity.

5.10 Historical Perspective and References

Interrupts go back to computer industry pioneers Eckert and Mauchly. Interrupts were first used to signal arithmetic overflow on the UNIVAC I and later to alert a UNIVAC 1103 to start online data collection for a wind tunnel (see Codd [1962]). After the success of the first commercial computer, the UNIVAC 1101 in 1953, the first commercial computer to have interrupts, the 1103, was brought out. Interrupts were first used for I/O by A.L. Leiner in the National Bureau of Standards DYSEAC [Smotherman 1989].

Maurice Wilkes learned computer design in a summer workshop from Eckert and Mauchly and then went on to build the first full-scale, operational, stored-program computer—the EDSAC. From that experience he realized the difficulty of control. He thought of a more centralized control using a diode matrix and, after visiting the Whirlwind computer in the U.S., wrote:

I found that it did indeed have a centralized control based on the use of a matrix of diodes. It was, however, only capable of producing a fixed sequence of 8 pulses—a different sequence for each instruction, but nevertheless fixed as far as a particular instruction was concerned. It was not, I think, until I got back to Cambridge that I realized that the solution was to turn the control unit into a computer in miniature by adding a second matrix to determine the flow of control at the microlevel and by providing for conditional micro-instructions. [Wilkes 1985, 178]

Wilkes [1953] was ahead of his time in recognizing that problem. Unfortunately, the solution was also ahead of its time: To provide control, microprogramming relies on fast memory that was not available in the 1950s. Thus, Wilkes's ideas remained primarily academic conjecture for a decade, although he did construct the EDSAC 2 using microprogrammed control in 1958 with ROM made from magnetic cores.

IBM brought microprogramming into the spotlight in 1964 with the IBM 360 family. Before this event, IBM saw itself as many small businesses selling different machines with their own price and performance levels, but also with their own instruction sets. (Recall that little programming was done in high-level languages, so that programs written for one IBM machine would not run on another.) Gene Amdahl, one of the chief architects of the IBM 360, said that managers of each subsidiary agreed to the 360 family of computers only because they were convinced that microprogramming made it feasible—if you could take the same hardware and microprogram it with several different instruction sets, they reasoned, then you must also be able to take different hardware and microprogram them to run the same instruction set. To be sure of the viability of microprogramming, the IBM vice president of engineering even visited Wilkes surreptitiously and had a "theoretical" discussion of the pros and cons of microcode. IBM believed the idea was so important to their plans that they pushed the memory technology inside the company to make microprogramming feasible.

Stewart Tucker of IBM was saddled with the responsibility of porting software from the IBM 7090 to the new IBM 360. Thinking about the possibilities of microcode, he suggested expanding the control store to include simulators, or interpreters, for older machines. Tucker [1967] coined the term *emulation* for this, meaning full simulation at the microprogrammed level. Occasionally, emulation on the 360 was actually faster than the original hardware. Emulation became so popular with customers in the early years of the 360 that it was sometimes hard to tell which instruction set ran more programs.

Once the giant of the industry began using microcode, the rest soon followed. A difficulty in adopting microcode was that the necessary memory technology was not widely available, but that was soon solved by semiconductor ROM and later RAM. The microprocessor industry followed the same history, with limited resources of the earliest chips forcing hardwired control. But as the resources increased, the advantages of simpler design and ease of change persuaded many to use microprogramming.

With the increasing popularity of microprogramming came more sophisticated instruction sets, including virtual memory. Microprogramming may well have aided the spread of virtual memory, since microcode made it easier to cope with the difficulties that arose from mapping addresses and restarting instructions. The IBM 370 model 138, for example, implemented virtual memory entirely in microcode without any hardware support.

Over the years, most microarchitectures became more and more dedicated to support the intended instruction set, so that reprogramming for a different instruction set failed to offer satisfactory performance. With the passage of time came much larger control stores, and it became possible to consider a machine as elaborate as the VAX. To offer a single chip VAX in 1984 DEC reduced the instructions interpreted by microcode by trapping some instructions and performing them in software: 20% of VAX instructions are responsible for 60% of the microcode, yet are only executed 0.2% of the time. Figure 5.35 shows the

reduction in control store by subsetting the instruction set. (The VAX is so tied to microcode that we venture to predict it will be impossible to build a full-instruction-set VAX without microcode.) The microarchitecture of one of the simpler subsetted VAXes, the MicroVAX-I, is described in Levy and Eckhouse [1989].

	Full instruction set (VLSI VAX)	Subset instruction set (MicroVAX 32)
% instructions implemented	100%	80%
Size of control store (bits)	480 K	64 K
Number of chips in processor	9	2
% performance of VAX-11/780	100%	90%

FIGURE 5.35 By trapping some VAX instructions and addressing modes, control store was reduced almost eight-fold. The second chip of the subset VAX is for floating point.

While this book was being written, a landmark legal precedent concerning microcode was set. The question under litigation in *NEC v. Intel* was whether microcode is like writing, and thereby deserves copyright protection (Intel), or whether it is like hardware, which can be patented but not copyrighted (NEC). The importance of this matter lies in the fact that while it is trivial to get a copyright, getting a patent can take as long as a college education. A program can be copyrighted, so the question then follows: What is and isn't a program? Here is the legislated definition:

A 'computer program' is a set of statements or instructions to be used directly or indirectly in a computer in order to bring about a certain result.

After years of preparation and trial, a judge did declare that a microprogram was a program. The lawyers for the losing side then asked him to rescind his decision on grounds of partiality. They had discovered that through an investment club, the judge owned $80 of stock belonging to the client he ruled for. (The tempting sum really was only $80, highly frustrating to one of the authors who acted as an expert witness on the case!) The case was retried, and the new judge ruled that "microcode ... comes squarely within the definition of a 'computer program'..." [Gray 1989, 4]. Of course, the fact that two judges in two different trials made the same decision doesn't mean that the matter is closed—there are still higher levels of appeal available.

References

CLARK, D. W., P. J. BANNON, AND J. B. KELLER [1988]. "Measuring VAX 8800 performance with a histogram hardware monitor," *Proc. 15th Annual Symposium on Computer Architecture* (May–June), Honolulu, Hawaii, 176–185.

CODD, E. F. [1962]. "Multiprogramming," in F.L. Alt and M. Rubinoff, *Advances in Computers*, vol. 3, Academic Press, New York, 82.

EMER, J. S. AND D. W. CLARK [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310.

GRAY, W. P. [1989]. Memorandum of Decision, No. C-84-20799-WPG, U.S. District Court for the Northern District of California (February 7, 1989).

LEVY, H. M. AND R. H. ECKHOUSE, JR. [1989]. *Computer Programming and Architecture: The VAX*, 2nd ed., Digital Press, Bedford, Mass. 358–372

MCKEVITT, J., ET AL. [1977]. *8086 Design Report*, internal memorandum.

PATTERSON, D. A. [1983]. "Microprogramming," *Scientific American* 248:3 (March), 36–43.

REIGEL, E. W., U. FABER, AND D. A. FISCHER, [1972]. "The Interpreter—a microprogrammable building block system," *Proc. AFIPS 1972 Spring Joint Computer Conf.* 40, 705–723.

SMOTHERMAN, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September), 5–15.

TUCKER, S. G. [1967]. "Microprogram control for the System/360," *IBM Systems Journal* 6:4, 222–241.

WILKES, M. V. [1953]. "The best way to design an automatic calculating machine," in *Manchester University Computer Inaugural Conf.*, 1951, Ferranti, Ltd., London. (Not published until 1953.) Reprinted in "The Genesis of Microprogramming" in *Annals of the History of Computing* 8:116.

WILKES, M. V. [1985]. *Memoirs of a Computer Pioneer*, The MIT Press, Cambridge, Mass.

WILKES, M. V. AND J. B. STRINGER [1953]. "Microprogramming and the design of the control circuits in an electronic digital computer," *Proc. Cambridge Philosophical Society* 49:230–238. Also reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 158–163, and in "The Genesis of Microprogramming" in *Annals of the History of Computing* 8:116.

EXERCISES

If finite-state diagrams and microprogramming are review topics, you may want to skip over questions 5.5 through 5.14.

5.1 [15/10/15/15] <5.5> One technique that tries to get the best of both the worlds of vertical and horizontal microarchitectures is a *two-level* control store, as illustrated by Figure 5.36. It tries to combine small control-store size with wide instructions. To avoid confusion the bottom level uses the prefix *nano-*, yielding the terms "nanoinstruction," "nanocode," and so forth. This technique was used in the Motorola 68000, 68010, and 68020, but it was originated in the Burroughs D-machine [Reigel, Faber, and Fischer 1972]. The idea is that the first level has many vertical instructions that point to the few unique horizontal instructions in the second level. The Burroughs D-machine was a general-purpose computer offering writable control store. Its microinstructions were 16 bits wide, with 12 of those bits specifying a nanoaddress, and the nanoinstructions were 56 bits wide. One instruction set interpreter used 1124 microinstructions and 123 nanoinstructions.

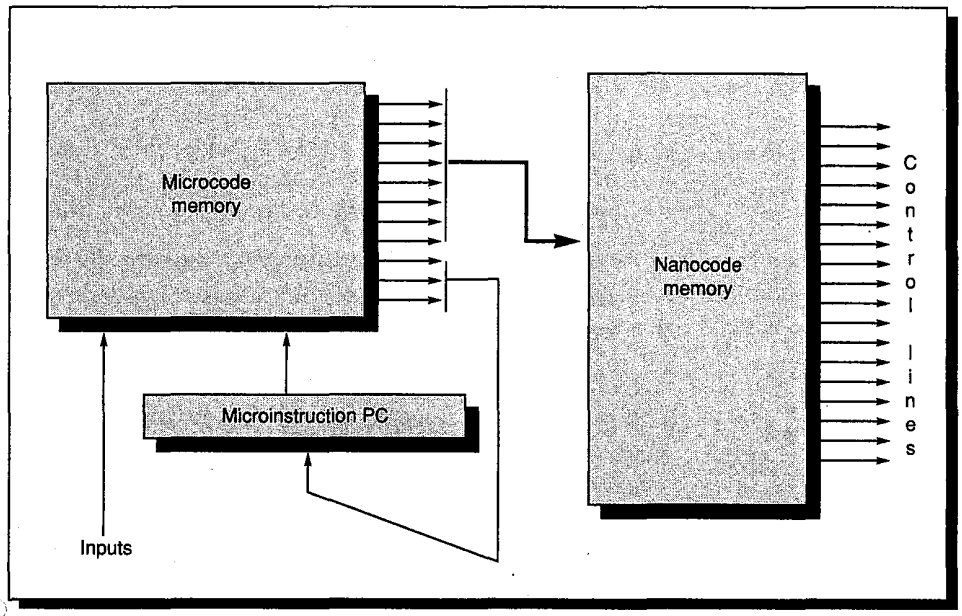

FIGURE 5.36 Two-level microprogrammed implementation showing relationship of microcode and nanocode.

a. [15] <5.5> What is the general formula showing when a two-level control store scheme like Burroughs D-machine uses fewer bits than a single-level control store? Assume there are M microinstructions each a bits wide and N nanoinstructions each b bits wide.

b. [10] Was the two-level control store of the D-machine successful in reducing control-store size versus a single-level control store for the interpreter?

c. [15] After the code was optimized to improve CPI by 10%, the resulting code had 940 microinstructions and 161 nanoinstructions. Was the two-level control store of the D-machine successful in reducing control-store size versus a single-level control store for the **optimized** interpreter?

d. [15] Did optimization increase or decrease the total number of bits needed to specify control? Why would the number of microinstructions decrease and the number of nanoinstructions increase?

5.2 [15] <5.5,5.6> One advantage of microcode is that it can handle rare cases without having the overhead of invoking the operating system before executing the trap routine. Suppose a machine with a CPI of 1.5 has an operating system that takes 100 clock cycles on a trap before it can execute the appropriate code. Suppose the trap code takes 10 clock cycles whether it is microcode or macrocode. For an instruction occurring 5% of the time, what percentage of the time must it trap before a microcode implementation is 1% faster overall than a macrocode implementation?

5.3 [20/20/30] <4.2,5.5,5.6> Let's explore the impact of subsetting an architecture as described in Figure 5.35. Suppose the `MOVC3` instruction were left out of a VAX.

a. [20] Write the VAX macrocode to replace `MOVC3`.

b. [20] Assume the operands are placed in registers R0, R1, and R2 after a trap. Using the data for COBOLX in Figure C.1 in Appendix C on instruction usage (assuming all `MOVC_` are `MOVC3`) and assuming the average `MOVC3` moves 15 bytes, what would be the percentage change in instruction count if `MOVC3` were not interpreted by microcode? (Ignore the cost of traps for this instruction.)

c. [30] If you have access to a VAX, time the speed of `MOVC3` versus a macrocode version of the routine from part a. Assuming that the trap overhead is 20 clock cycles, what is the impact on performance of trapping to software for `MOVC3`?

5.4 [15] <5.6> Assume we have a machine with a clock cycle time of 10 ns and a base CPI of 5. Because of the possibilities of interrupts we must have extra registers containing copies of the values of the registers at the beginning of the instruction. These registers are usually called *shadow registers*. Assume that the average instruction has two register operands that must be restored on an interrupt. The interrupt rate is 100 interrupts per second, and the interrupt cost is 30 cycles plus the time to restore the shadowed registers, each of which takes 10 cycles. What is the effective CPI after accounting for interrupts? What is the performance lost from interrupts?

5.5-5.7 Given the processor design and finite-state diagram for DLX as modified in the end of the hardwired-control portion of Section 5.7, explore the impact of performance of the following changes. In each case show the modified portion of the finite-state machine, describe the changes to the processor (if necessary), the change in the number of states, and calculate the change in CPI using the DLX instruction mix statistics in Figure C.4 for GCC. Show the reasons for the change.

5.5 [12] <5.7> Like the change to the ALU instructions in the second example in Section 5.7 and shown in Figures 5.20 and 5.21, remove the states that load Temp for the Set instructions in Figure 5.16 first by adding the "X" option and then by increasing the number of states.

5.6 [15] <5.7> Suppose that the memory interface was optimized so that it was not necessary to load MAR before a memory access, nor did the data have to be transferred in MDR for a read or write. Instead, any register on the S1 bus could specify the address, any register on the S2 bus could supply the data on a write, and any register on the Dest bus could receive data on a read.

5.7 [22] <5.7> Most computers overlap the fetching of the next instruction with the execution of the current instruction. Propose a scheme that overlaps all instruction fetches except jumps, branches, and stores. You must reorganize the finite-state machine so that the instruction is already fetched, possibly even partially decoded.

5.8 [15] <5.7> The example in Section 5.7 on page 228 assumes everything but the shifter can scale to 9 ns. Alas, the memory system can rarely scale as easily as the CPU. Reperform the analysis in this example, but this time assume that average number of memory wait states is 2 at the 9-ns clock cycle versus 1 at 10 ns in addition to the slowdown for shifts.

5.9-5.14 These questions address use of the microcoded control of DLX as shown in Figures 5.23, 5.25, and 5.27–5.29. In each case show the modified portion of the microcode; describe the changes to the processor (if necessary), the microinstruction fields (if necessary), and the change in the number of microinstructions; and calculate the change in CPI using the DLX instruction-mix statistics in Appendix C for GCC. Show the reasons for the change.

5.9 [15] <5.7> Like the change to the ALU instructions in the second example in Section 5.7, remove the microinstructions that load Temp for the Set instructions in Figure 5.28 (page 233) first by adding the "X" option and then by increasing the number of microinstructions.

5.10 [25] <5.7> Continuing the example in Figure 5.32 (page 237), rewrite the microcode found in Figure 5.29 (page 234) using the dual-format microinstructions of Figure 5.31 (page 236). What is the relative frequency of each type of microinstruction? What is the savings in control-store size versus the original DLX format? What is the change in CPI?

5.11 [20] <3.4, 5.7> Load byte and Load half take a clock cycle longer than Load word because of the alignment of data (see Figure 3.10 on page 97 and Figure 5.25 on page 231). Propose a change that eliminates the extra clock for these instructions. How does this change affect the CPI of GCC? How does it affect the CPI of TeX?

5.12 [20] <5.6, 5.7> Change the microcode to perform the following interrupt tests: page fault, arithmetic overflow or underflow, misaligned memory accesses, and using undefined instructions. Make whatever changes are needed to the microarchitecture and microinstruction format. What is the change in size and performance to perform these tests?

5.13 [20] <5.7> The computer designer must be careful not to tailor her design too closely to a particular, single program. Reevaluate the performance impact of all the example performance improvements in Exercises 5.9 to 5.12 this time using the average instruction mix data in Figure C.4. How do the programs affect the evaluations?

5.14 [20] <5.6, 5.7> Starting with the microcode in Figures 5.27 (page 233) and 5.34 (page 238), revise the microcode so that the next macroinstruction is fetched as early as possible during the ALU instructions. Assume a "perfect" memory system, taking one clock cycle per memory reference. Although technically this improvement speeds up instructions that **follow** ALU instructions, the easiest way to account for higher performance is as faster ALU instructions. How much faster are the ALU instructions? How does it affect overall performance according to GCC statistics?

5.15 [30] <4,5.6> If you have access to a machine that uses one of the instruction sets in Chapter 4, determine the worst-case interrupt latency for that implementation of the architecture. Be sure you are measuring the raw machine latency and **not** the operating system overhead.

5.16 [30] <5.6> Computer architects have sometimes been forced to support instructions that were never published in the original instruction set manual. This situation arises

because some programs are created that inadvertently set unused instruction fields to values other than the architect expected, which raises havoc when the architect tries to use those values to extend the instruction set. IBM solved that problem in the System 370 by trapping on every possible undefined field. Try executing instructions with undefined fields on a computer to see what happens. Do your new instructions compute anything useful? If so, would you use these new instructions in programs?

5.17 [35] <5.4, 5.5, 5.7> Take the datapath in Figure 5.1 and build a simulator that can perform any of the operations needed to implement the DLX instruction set. Now implement the DLX instruction set using:

Microprogrammed control, and

Hardwired control.

For hardwired control see if you can find PLA minimization and state-assignment programs to reduce the cost of control. From these two designs, determine the performance of each implementation and the cost in terms of gates or in terms of silicon area.

5.18 [35] <2.2, 5.5, 5.7> The similarities between the microinstructions and the macroinstructions of DLX suggest that performance can be gained by writing a program that translates from DLX macrocode to DLX microcode. (This is the insight that inspired WCS.) Write such a program and benchmark it. What is the resulting expansion of code size?

5.19 [50] <2.2, 4.4, 5.10> Recent attempts have been made to run existing software on hardwired control machines by building hand-tuned simulators for popular machines. Write such a simulator for the 8086 instruction set. Run some existing IBM PC programs, and see how fast your simulator is relative to an 8-MHz 8086.

5.20 [Discussion] <4,5.5,5.10> Hypothesis: If the first implementation of an architecture uses microprogramming, it affects the instruction set architecture. Why might this be true? Looking at examples in Chapter 4 or elsewhere, give supporting or contradicting evidence from real machines. Which machines will always use microcode? Why? Which machines will never use microcode? Why? What control implementation do you think the architect had in mind when designing the instruction set architecture?

5.21 [Discussion] <5.5,5.10> Wilkes invented microprogramming in large to simplify construction of control. Since 1980 there has been an explosion of computer-aided design software whose goal is also to simplify construction of control. Hypothesis: The advances in computer-aided design software have rendered microprogramming unnecessary. Find evidence to support and refute the hypothesis.

5.22 [Discussion] <5.10> The DLX instructions and the DLX microinstructions have many similarities. What would make it difficult for a compiler to produce DLX microcode rather than macrocode? What changes to the microarchitecture would make the DLX microcode more useful for this application?

It is quite a three-pipe problem.

Sir Arthur Conan Doyle, *The Adventures of Sherlock Holmes*

6 Pipelining

6.1 What Is Pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. Today, pipelining is the key implementation technique used to make fast CPUs.

A pipeline is like an assembly line: Each step in the pipeline completes a part of the instruction. As in a car assembly line, the work to be done in an instruction is broken into smaller pieces, each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is called a *pipe stage* or a *pipe segment*. The stages are connected one to the next to form a pipe—instructions enter at one end, are processed through the stages, and exit at the other end.

The throughput of the pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time. The time required between moving an instruction one step down the pipeline is a machine cycle. The length of a machine cycle is determined by the time required for the slowest pipe stage (because all stages proceed at the same time). Often the machine cycle is one clock cycle (sometimes it is two, or rarely more), though the clock may have multiple phases.

The pipeline designer's goal is to balance the length of the pipeline stages. If the stages are perfectly balanced, then the time per instruction on the pipelined machine—assuming ideal conditions (i.e., no stalls)—is equal to

$$\frac{\text{Time per instruction on nonpipelined machine}}{\text{Number of pipe stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipe stages. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined machine will not have its minimum possible value, though it can be close (say within 10%).

Pipelining yields a reduction in the average execution time per instruction. This reduction can be obtained by decreasing the clock cycle time of the pipelined machine or by decreasing the number of clock cycles per instruction, or by both. Typically, the biggest impact is in the number of clock cycles per instruction, though the clock cycle is often shorter in a pipelined machine (especially in pipelined supercomputers). In the advanced pipelining sections of this chapter we will see how deep pipelines can be used to both decrease the clock cycle and maintain a low CPI.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see Chapters 7 and 10), it is not visible to the programmer. In this chapter we will first cover the concept of pipelining using DLX and a simplified version of its pipeline. We will then look at the problems pipelining introduces and the performance attainable under typical situations. Later in the chapter we will examine advanced techniques that can be used to overcome the difficulties that are encountered in pipelined machines and that may lower the performance attainable from pipelining.

We use DLX largely because its simplicity makes it easy to demonstrate the principles of pipelining. The same principles apply to more complex instruction sets, though the corresponding pipelines are more complex. We will see an example of such a pipeline in the Putting It All Together section.

6.2 The Basic Pipeline for DLX

Remember that in Chapter 5 (Section 5.3) we discussed how DLX could be implemented with five basic execution steps:

1. IF—instruction fetch
2. ID—instruction decode and register fetch
3. EX—execution and effective address calculation
4. MEM—memory access
5. WB—write back

Instruction number	Clock number 1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i*+1		IF	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				IF	ID	EX	MEM	WB	
Instruction *i*+4					IF	ID	EX	MEM	WB

FIGURE 6.1 Simple DLX pipeline. On each clock cycle another instruction is fetched and begins its five-step execution. If an instruction is started every clock cycle, the performance will be five times that of a machine that is not pipelined.

We can pipeline DLX by simply fetching a new instruction **on each clock cycle.** Each of the steps above becomes a *pipe stage*—a step in the pipeline—resulting in the execution pattern shown in Figure 6.1. While each instruction still takes five clock cycles, during each clock cycle the hardware is executing some part of five different instructions.

Pipelining increases the CPU instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline. The increase in instruction throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster!

The fact that the execution time of each instruction remains unchanged puts limits on the practical depth of a pipeline, as we will see in the next section. Other design considerations limit the clock rate that can be attained by deeper pipelining. The most important consideration is the combined effect of latch delay and clock skew. Latches are required between pipe stages, adding setup time plus the delay through those latches to each clock period. Clock skew also contributes to the lower limit on the clock cycle. Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful.

Example

Consider a nonpipelined machine with five execution steps of lengths 50 ns, 50 ns, 60 ns, 50 ns, and 50 ns. Suppose that due to clock skew and setup, pipelining the machine adds 5 ns of overhead to each execution stage. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Answer

Figure 6.2 shows the execution pattern on the nonpipelined machine and on the pipelined machine.

The average instruction execution time on the nonpipelined machine is

Average instruction execution time = 50+50+60+50+50 ns = 260 ns

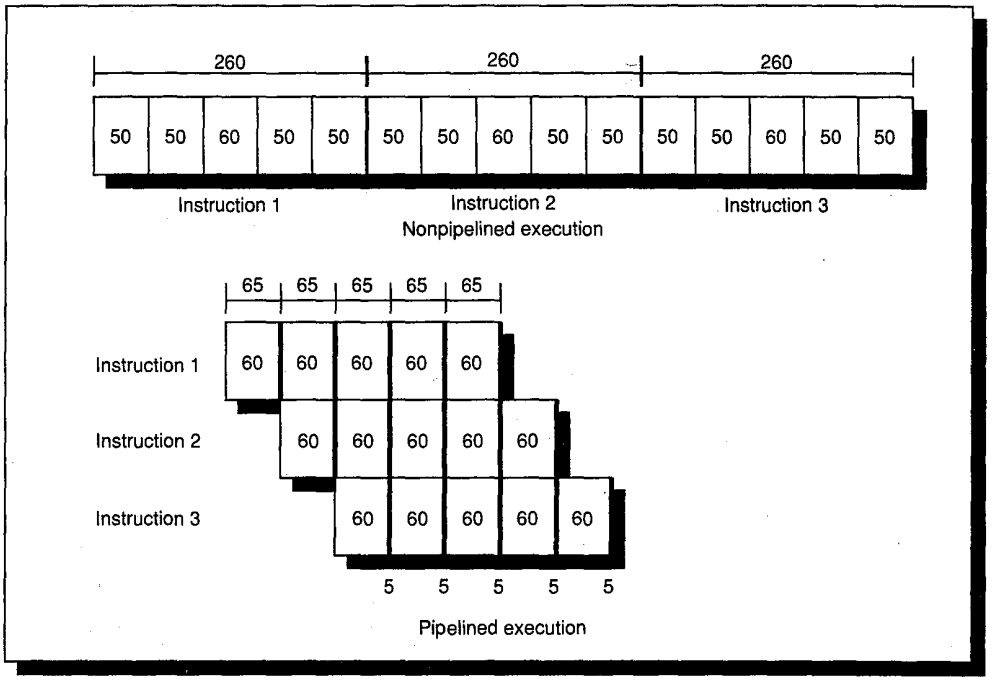

FIGURE 6.2 The execution pattern for three instructions shown for both the nonpipelined and pipelined versions. In the nonpipelined version, the three instructions are executed sequentially. In the pipelined version, the shaded areas represent the overhead of 5 ns per pipestage. The length of the pipestages must all be the same: 60 ns plus the 5-ns overhead. The latency of an instruction increases from 260 ns in the nonpipelined machine to 325 ns in the pipelined machine.

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 60 + 5 or 65 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\text{Speedup} = \frac{\text{Average instruction time without pipeline}}{\text{Average instruction time with pipeline}}$$

$$= \frac{260}{65} = 4 \text{ times}$$

The 5-ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's Law tells us that the overhead limits the speedup.

Because the latches in a pipelined design can have a significant impact on the clock speed, designers have looked for latches that permit the highest possible clock rate. The Earle latch (invented by J. G. Earle [1965]) has three properties that make it especially useful in pipelined machines. First, it is relatively insensitive to clock skew. Second, the delay through the latch is always a constant two-gate delay, avoiding the introduction of skew in the data passing through the latch. Finally, two levels of logic can be done in the latch without increasing the latch delay time. This means that two levels of logic in the pipeline can be overlapped with the latch, so the majority of the overhead from the latch can be

hidden. We will not be analyzing the pipeline designs in this chapter at this level of detail. The interested reader should see Kunkel and Smith [1986].

The next two sections will add refinements and address some problems that can occur in this pipeline. In this discussion (up to the last segment of Section 6.5) we will focus on the pipeline for the integer portion of DLX. The complications that arise in the floating-point pipeline will be treated in Section 6.6.

6.3 Making the Pipeline Work

Your instinct is right if you find it hard to believe that pipelining is as simple as this, because it's not. In this and the following three sections, we will make our DLX pipeline "real" by dealing with problems that pipelining introduces.

To begin with, we have to determine what happens on every clock cycle of the machine and make sure that overlapping instructions doesn't overcommit resources. For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. As we will see, the simplicity of the DLX instruction set makes resource evaluation relatively easy.

The operations that occur during instruction execution, which were discussed in Section 5.3 of Chapter 5, are modified to execute in a pipeline as shown in Figure 6.3. The figure lists the major functional units in our DLX implementation, the pipe stages, and what has to happen in each stage of the pipeline. The vertical axis is labeled with the pipeline stages, while the horizontal axis shows major resources. Each intersection shows what happens for that resource in that stage. In Figure 6.4 we will show similar information using the instruction type as the horizontal axis. The combination of instructions that may be in the pipeline at any one time is arbitrary. Thus, the combined needs of all instruction types at any pipe stage determine what resources are needed at that stage.

Every pipe stage is active on every clock cycle. This requires all operations in a pipe stage to complete in one clock and any combination of operations to be able to occur at once. Here are the most important implications for the data path, as specified in Chapter 5:

1. The PC must be incremented on each clock. This must be done in IF rather than ID. This will require an additional incrementer, since the ALU is already busy on every cycle and cannot be used to increment the PC.
2. A new instruction must be fetched on every clock—this is also done in IF.
3. A new data word is needed on every clock cycle—this is done in MEM.
4. There must be a separate MDR for loads (LMDR) and stores (SMDR), since when they are back-to-back, they overlap in time.
5. Three additional latches are needed to hold values that are needed later in the pipeline, but may be modified by a subsequent instruction. The values latched are the instruction, the ALU output, and the next PC.

Stage	PC unit	Memory	Data path
IF	PC← PC+4;	IR←Mem[PC];	
ID	PC1← PC	IR1←IR	A← Rs1; B← Rs2;
EX			DMAR← A + $(IR1_{16})^{16}$##$IR1_{16..31}$; SMDR←B; or ALUoutput← A *op* (B or $(IR1_{16})^{16}$##$IR1_{16..31}$); or ALUoutput←PC1 + $(IR1_{16})^{16}$##$IR1_{16..31}$; cond← (A *op* 0);
MEM	if (cond) PC←ALUoutput	LMDR← Mem[DMAR] or Mem[DMAR]←SMDR	ALUoutput1← ALUoutput
WB			Rd← ALUoutput1 or LMDR

FIGURE 6.3 The table shows the major functional units and what may happen in every pipe stage in each unit. In several of the stages not all of the actions listed can occur, because they apply under different assumptions about the instruction. For example, there are three operations within the ALU during the EX stage. The first occurs only on a load or store; the second on ALU operations (with the input being B or the lower 16 bits of the IR, according to whether the instruction is register–register or register–immediate); the third operation occurs only on branches. For simplicity, we have shown the branch case only—jumps will add a 26-bit offset to the PC. The variables ALUouput1, PC1, and IR1 save values for use in later stages of the pipeline. Designing the memory system to support a data load or store on every clock cycle is challenging; see Chapter 8 for an in-depth discussion. This type of table and that in Figure 6.4 are loosely based on Davidson's [1971] pipeline reservation tables.

Stage	ALU instruction	Load or store instruction	Branch instruction
IF	IR←Mem[PC]; PC←PC+4;	IR←Mem[PC]; PC←PC+4;	IR←Mem[PC]; PC←PC+4;
ID	A←Rs1; B←Rs2; PC1←PC IR1←IR	A←Rs1; B←Rs2; PC1←PC IR1←IR	A←Rs1; B←Rs2; PC1←PC IR1←IR
EX	ALUoutput←A *op* B; or ALUoutput←A *op* $((IR1_{16})^{16}$##$IR1_{16..31})$;	DMAR← A+ $((IR1_{16})^{16}$##$IR1_{16..31})$; SMDR← B;	ALUoutput←PC1 + $((IR1_{16})^{16}$##$IR1_{16..31})$; cond←(A *op* 0);
MEM	ALUoutput1← ALUoutput	LMDR←Mem[DMAR]; or Mem[DMAR]←SMDR;	if (cond) PC←ALUoutput;
WB	Rd←ALUoutput1;	Rd←LMDR;	

FIGURE 6.4 Events on every pipe stage of the DLX pipeline. Because the instruction is not yet decoded, the first two pipe stages are always identical. Note that it was critical to be able to fetch the registers before decoding the instruction; otherwise another pipeline stage would be required. Due to the fixed instruction format, both register fields are always decoded and the registers accessed (though they are sometimes not needed); the PC and immediate fields can be sent to the ALU as well. At the beginning of the ALU operation the correct inputs are multiplexed in, based on the opcode. With this organization all instruction-dependent operations occur in the EX stage or later. As in Figure 6.3, we include the case for branches, but not jumps, which will have a 26-bit offset rather than a 16-bit offset. Jumps are essentially like branches.

Probably the biggest impact of pipelining on the machine resources is in the memory system. Although the memory-access time has not changed, the peak memory bandwidth must be increased by five times over the nonpipelined machine because two memory accesses are required on every clock in the pipelined machine versus two accesses every five clock cycles in a nonpipelined machine with the same number of steps per instruction. To provide two memory accesses every clock, most machines will use separate instruction and data caches (see Chapter 8, Section 8.3).

During the EX stage, the ALU can be used for three different functions: an effective data-address calculation, a branch-address calculation, or an ALU instruction. Fortunately, the DLX instructions are simple; an instruction in EX does at most one of these, so no conflict arises.

The pipeline we now have for DLX would function just fine if every instruction were independent of every other instruction in the pipeline. In reality, instructions in the pipeline can be dependent on one another; this is the topic of the next section.

6.4 The Major Hurdle of Pipelining—Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to stall the pipeline. The major difference between stalls in a pipelined machine and stalls in a nonpipelined machine (such as those we saw in DLX in Chapter 5) occurs because there are multiple instructions under execution at once. A stall in a pipelined machine often requires that some instructions be allowed to proceed, while others are delayed. Typically, when an instruction is stalled, all instructions later in the pipeline than the stalled instruction are also stalled. Instructions earlier than the stalled instruction can continue, but no new instructions are fetched during the stall. We will see several examples of how stalls operate in this section—don't worry, they aren't as complex as they might sound!

A stall causes the pipeline performance to degrade from the ideal performance. Let's look at a simple equation for finding the actual speedup from pipelining, starting with the formula from the previous section.

$$\text{Pipeline speedup} = \frac{\text{Average instruction time without pipeline}}{\text{Average instruction time with pipeline}}$$

$$= \frac{\text{CPI without pipelining} * \text{Clock cycle without pipelining}}{\text{CPI with pipelining} * \text{Clock cycle with pipelining}}$$

$$= \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{CPI without pipelining}}{\text{CPI with pipelining}}$$

Remember that pipelining can be thought of as decreasing the CPI or the clock cycle time; let's treat it as decreasing the CPI. The ideal CPI on a pipelined machine is usually

$$\text{Ideal CPI} = \frac{\text{CPI without pipelining}}{\text{Pipeline depth}}$$

Rearranging this and substituting into the speedup equation yields:

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{CPI with pipelining}}$$

If we confine ourselves to pipeline stalls,

CPI with pipelining = Ideal CPI + Pipeline stall clock cycles per instruction

We can substitute and obtain:

$$\text{Speedup} = \frac{\text{Clock cycle without pipelining}}{\text{Clock cycle with pipelining}} * \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles}}$$

While this gives a general formula for pipeline speedup (ignoring stalls other than from the pipeline), in most instances a simpler equation can be used. Often, we choose to ignore the potential increase in the clock cycle due to pipelining overhead. This makes the clock rates equal and allows us to drop the first term. A simpler formula can now be used:

$$\text{Pipeline speedup} = \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles}}$$

While we will use this simpler form for evaluating the DLX pipeline, a designer must be careful not to discount the potential impact on clock rate in evaluating pipelining strategies.

Structural Hazards

When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions

cannot be accommodated due to resource conflicts, the machine is said to have a *structural hazard.* The most common instances of structural hazards arise when some functional unit is not fully pipelined. Then a sequence of instructions that all use that functional unit cannot be sequentially initiated in the pipeline. Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute. For example, a machine may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle. This will generate a structural hazard. When a sequence of instructions encounters this hazard, the pipeline will stall one of the instructions until the required unit is available.

Many pipelined machines share a single memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference, the pipeline must stall for one clock cycle; the machine cannot fetch the next instruction because the data reference is using the memory port. Figure 6.5 shows what a one-memory-port pipeline looks like when it stalls during a load. We will see another type of stall when we talk about data hazards.

Instruction	Clock cycle number								
	1	2	3	4	5	6	7	8	9
Load instruction	IF	ID	EX	MEM	WB				
Instruction *i*+1		IF	ID	EX	MEM	WB			
Instruction *i*+2			IF	ID	EX	MEM	WB		
Instruction *i*+3				stall	IF	ID	EX	MEM	WB
Instruction *i*+4						IF	ID	EX	MEM

FIGURE 6.5 A pipeline stalled for a structural hazard—a load with one memory port. With only one memory port, the pipeline cannot initiate a data fetch and instruction fetch in the same cycle. A load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would be instruction *i*+3). Because the instruction being fetched is stalled, all other instructions in the pipeline can proceed normally. The stall cycle will continue to pass through the pipeline.

Example

Suppose that data references constitute 30% of the mix and that the ideal CPI of the pipelined machine, ignoring the structural hazard, is 1.2. Disregarding any other performance losses, how much faster is the ideal machine without the memory structural hazard, versus the machine with the hazard?

Answer

The ideal machine will be faster by the ratio of the speedup of the ideal machine over the real machine. Since the clock rates are unaffected, we can use the following for speedup:

$$\text{Pipeline speedup} = \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycles}}$$

Since the ideal machine has no stalls, its speedup is simply $\frac{1.2*\text{Pipeline depth}}{1.2}$.

The speedup of the real machine is $\frac{1.2*\text{Pipeline depth}}{1.2 + 0.3*1} = \frac{1.2*\text{Pipeline depth}}{1.5}$.

$$\frac{\text{Speedup}_{\text{ideal}}}{\text{Speedup}_{\text{real}}} = \frac{\left(\frac{1.2*\text{Pipeline depth}}{1.2}\right)}{\left(\frac{1.2*\text{Pipeline depth}}{1.5}\right)} = \frac{1.5}{1.2} = 1.25$$

Thus, the machine without the structural hazard is 25% faster.

If all other factors are equal, a machine without structural hazards will always have a lower CPI. Why, then, would a designer allow structural hazards? There are two reasons: to reduce cost and to reduce the latency of the unit. Pipelining all the functional units may be too costly. Machines that support one-clock-cycle memory references require twice as much total memory bandwidth and often have higher bandwidth at the pins. Likewise, fully pipelining a floating-point multiplier consumes lots of gates. If the structural hazard would not occur often, it may not be worth the cost to avoid it. It is also usually possible to design a nonpipelined unit, or one that isn't fully pipelined, with a shorter total delay than a fully pipelined unit. For example, both the CDC 7600 and the MIPS R2010 floating-point unit choose shorter latency (fewer clocks per operation) versus full pipelining. As we will see shortly, reducing latency has other performance benefits and can frequently overcome the disadvantage of the structural hazard.

Example

Many recent machines do not have fully pipelined floating-point units. For example, suppose we had an implementation of DLX with a 5-clock-cycle latency for floating-point multiply, but no pipelining. Will this structural hazard have a large or small performance impact on Spice running on DLX? For simplicity, assume that the floating-point multiplies are uniformly distributed.

Answer

The data in Figure C.4 show that floating-point multiply has a frequency of 6% in Spice. Our proposed pipeline can handle up to a 20% frequency of floating-point multiplies—one every five clock cycles. This means that the performance benefit of fully pipelining the floating-point multiply is likely to be low, as long as the floating-point multiplies are not clustered but are distributed uniformly. If they were clustered, the impact could be much larger.

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. Data

hazards occur when the order of access to operands is changed by the pipeline versus the normal order encountered by sequentially executing instructions. Consider the pipelined execution of these instructions:

```
ADD   R1,R2,R3
SUB   R4,R1,R5
```

The SUB instruction has a source, R1, that is the destination of the ADD instruction. As shown in Figure 6.6, the ADD instruction writes the value of R1 in the WB pipe stage, but the SUB instruction reads the value during its ID stage. This problem is called a *data hazard.* Unless precautions are taken to prevent it, the SUB instruction will read the wrong value and try to use it. In fact, the value used by the SUB instruction is not even deterministic: Though we might think it logical to assume that SUB would always use the value of R1 that was assigned by an instruction prior to ADD, this is not always the case. If an interrupt should occur between the ADD and SUB instructions, the WB stage of the ADD will complete, and the value of R1 at that point **will** be the result of the ADD. This unpredictable behavior is obviously unacceptable.

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
ADD instruction	IF	ID	EX	MEM	WB–data written here	
SUB instruction		IF	ID–data read here	EX	MEM	WB

FIGURE 6.6 The ADD instruction writes a register that is a source operand for the SUB instruction. But the ADD doesn't **finish** writing the data into the register file until three clock cycles after SUB **begins** reading it!

The problem posed in this example can be solved with a simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*). This technique works as follows: The ALU result is always fed back to the ALU input latches. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file. Notice that with forwarding, if the SUB is stalled, the ADD will be completed, and the bypass will not be activated, causing the value from the register to be used. This is also true for the case of an interrupt between the two instructions.

In our DLX pipeline, we must pass results to not only the instruction that immediately follows, but also to the instruction after that. By the third instruction down the line, the ID and WB stages overlap; however, as the write is not finished until the end of WB, we must continue to forward the result. Figure 6.7 shows a set of instructions in the pipeline and the forwarding operations that can occur.

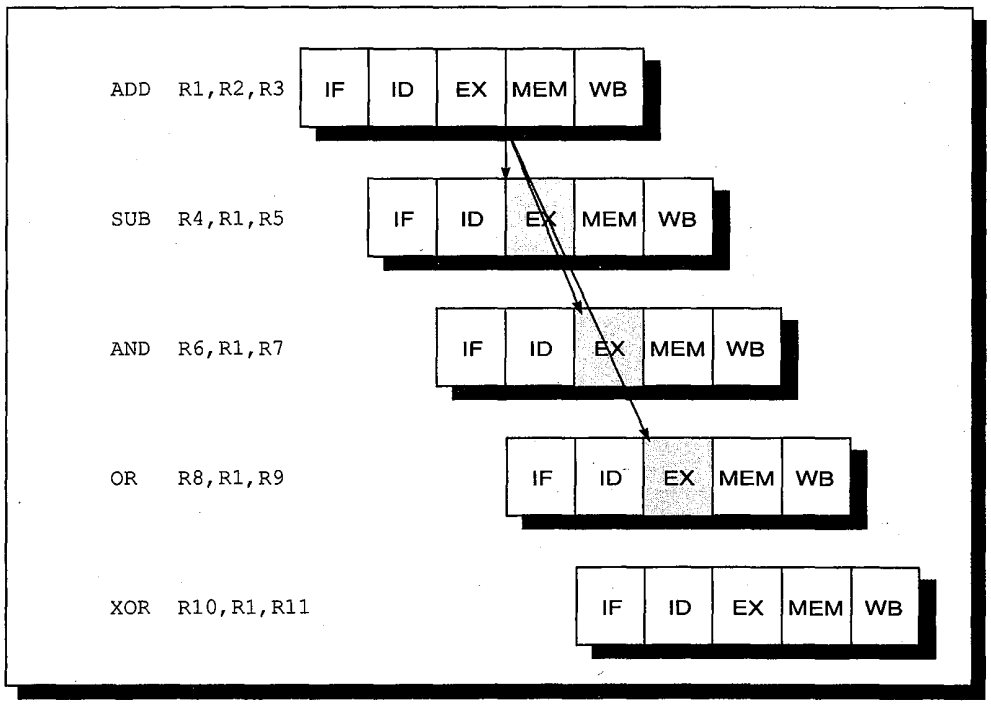

FIGURE 6.7 A set of instructions in the pipeline that need to forward results. The ADD instruction sets R1, and the next four instructions use it. The value of R1 must be bypassed to the SUB, AND, and OR instructions. By the time the XOR instruction goes to read R1 in the ID phase, the ADD instruction has completed WB, and the value is available.

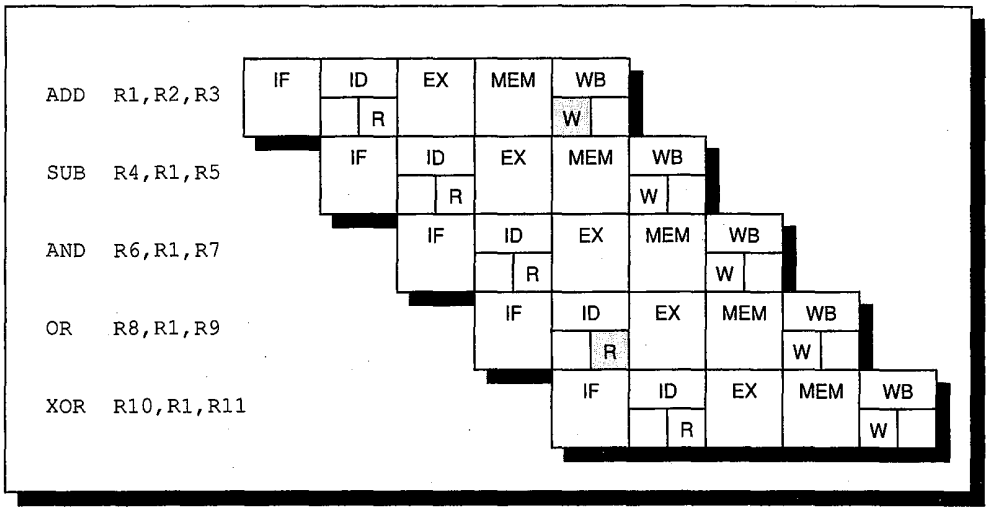

FIGURE 6.8 The same instruction sequence as shown in Figure 6.7, with register reads and writes occurring in opposite halves of the ID and WB stages. The SUB and AND instructions will still require the value of R1 to be bypassed to them, and this will happen as they enter their EX stage. However, by the time of the OR instruction, which also uses R1, the write of R1 has completed, and no forwarding is required. The XOR depends on the ADD, but the value of R1 from the ADD is always written back the cycle before XOR reaches its ID stage and reads it.

It is desirable to cut down the number of instructions that must be bypassed, since each level requires special hardware. Remembering that the register file is accessed twice in a clock cycle, it is possible to do the register writes in the first half of WB and the reads in the second half of ID. This eliminates the need to bypass to a third instruction, as shown in Figure 6.8.

Each level of bypass requires a latch and a pair of comparators to examine whether the adjacent instructions share a destination and a source. Figure 6.9 shows the structure of the ALU and its bypass unit as well as what values are in the bypass registers for the instruction sequence in Figure 6.7. Two ALU result buffers are needed to hold ALU results to be stored into the destination register in the next two WB stages. For ALU operations, the result is always forwarded when the instruction using the result as a source enters its EX stage. (The instruction that computed the value to be forwarded may be in its MEM or WB stages.) The results in the buffers can be inputs into either port on the ALU, via a pair of multiplexers. Multiplexer control can be done by either the control unit (which must then track the destinations and sources of all operations in the pipeline) or locally by logic associated with the bypass (in which case the bypass buffers will contain tags giving the register numbers the values are destined for). In either event, the logic must test if either of the two previous instructions wrote a register that is the input to the current instruction. If so, then the multiplexer select is set to choose from the appropriate result register rather than from the bus. Because the ALU operates in a single pipeline stage, there is no need for a pipeline stall with any combination of ALU instructions once the bypasses have been implemented.

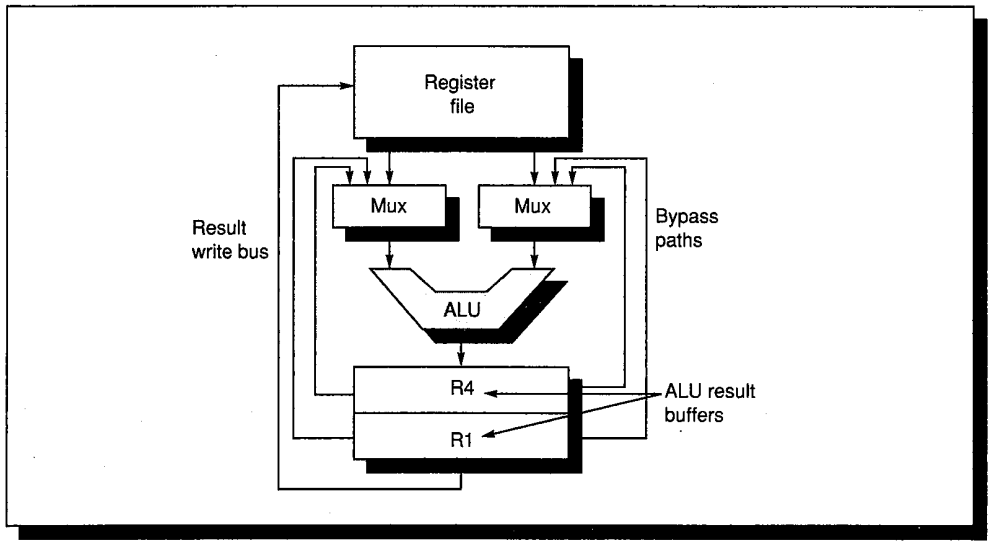

FIGURE 6.9 The ALU with its bypass unit. The contents of the buffer are shown at the point where the AND instruction of the code sequence in Figure 6.8 is about to begin the EX stage. The ADD instruction that computed R1 (in the second buffer) is in its WB stage, and the left input multiplexer is set to pass the just-computed value of R1 (not the value read from the register file) as the first operand to the AND instruction. The result of the subtract, R4, is in the first buffer. These buffers correspond to the variables `ALUoutput` and `ALUoutput1` in Figures 6.3 and 6.4.

A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining would change the order of access to an operand. Our example hazards have all been with register operands, but it is also possible for a pair of instructions to create a dependence by writing and reading the same memory location. In our DLX pipeline, however, memory references are always kept in order, preventing this type of hazard from arising. Cache misses could cause the memory references to get out of order if we allowed the processor to continue working on later instructions while an earlier instruction that missed the cache was accessing memory. For DLX's pipeline we just stall the entire pipeline, effectively making the instruction that contained the miss run for multiple clock cycles. In an advanced section of this chapter, Section 6.7, we will discuss machines that allow loads and stores to be executed in an order different from that in the program. All the data hazards discussed in this section, however, involve registers within the CPU.

Forwarding can be generalized to include passing a result directly to the functional unit that requires it: A result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit. Take, for example, the following sequence:

```
ADD   R1,R2,R3
SW    25(R1),R1
```

To prevent a stall in this sequence, we would need to forward the value of R1 from the ALU both to the ALU, so that it can be used in the effective address calculation, and to the MDR (memory data register), so that it can be stored without any stall cycles.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions i and j, with i occurring before j. The possible data hazards are:

- RAW *(read after write)* — j tries to read a source before i writes it, so j incorrectly gets the old value. This is the most common type of hazard and the one that appears in Figures 6.6 and 6.7.
- WAR *(write after read)* — j tries to write a destination before it is read by i, so i incorrectly gets the new value. This cannot happen in our example pipeline because all reads are early (in ID) and all writes are late (in WB). This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source after a write of an instruction later in the pipeline. For example, autoincrement addressing can create a WAR hazard.
- WAW *(write after write)* — j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard is present only in pipelines that write in more than one pipe stage (or allow an

instruction to proceed even when a previous instruction is stalled). The DLX pipeline writes a register only in WB and avoids this class of hazards.

Note that the RAR *(read after read)* case is not a hazard.

Not all data hazards can be handled without a performance effect. Consider the following sequence of instructions:

```
LW   R1,32(R6)
ADD  R4,R1,R7
SUB  R5,R1,R8
AND  R6,R1,R7
```

This case is different from the situation with back-to-back ALU operations. The LW instruction does not have the data until the end of the MEM cycle, while the ADD instruction needs to have the data by the beginning of that clock cycle. Thus, the data hazard from using the result of a load instruction cannot be completely eliminated with simple hardware. We **can** forward the result immediately to the ALU from the MDR, and for the SUB instruction— which begins two clock cycles after the load—the result arrives in time, as shown in Figure 6.10. However, for the ADD instruction, the forwarded result arrives too late—at the end of a clock cycle, though it is needed at the beginning.

LW R1,32(R6)	IF	ID	EX	**MEM**	WB
ADD R4,R1,R7		IF	ID	**EX**	MEM
SUB R5,R1,R8			IF	ID	**EX**
AND R6,R1,R7				IF	**ID**

FIGURE 6.10 Pipeline hazard occurring when the result of a load instruction is used by the next instruction as a source operand and is forwarded. The value is available when it returns from memory at the end of the load instruction's MEM cycle. However, it is needed at the beginning of that clock cycle for the ADD (the EX stage of the add). The load value can be forwarded to the SUB instruction and will arrive in time for that instruction (EX). The AND can simply read the value during ID since it reads the registers in the second half of the cycle and the value is written in the first half.

The load instruction has a delay or latency that cannot be eliminated by forwarding alone—to do so would require the data-access time to be zero. The most common solution to this problem is a hardware addition called a pipeline interlock. In general, a *pipeline interlock* detects a hazard and stalls the pipeline until the hazard is cleared. In this case, the interlock stalls the pipeline beginning with the instruction that wants to use the data until the sourcing instruction produces it. This delay cycle, called a *pipeline stall* or *bubble*, allows the load data to arrive from memory; it can now be forwarded by the hardware. The CPI for the stalled instruction increases by the length of the stall (one clock cycle in this case). The stalled pipeline is shown in Figure 6.11.

Any instruction	IF	ID	EX	MEM	WB					
`LW R1,32(R6)`		IF	ID	EX	**MEM**	WB				
`ADD R4,R1,R7`			IF	ID	*stall*	**EX**	MEM	WB		
`SUB R5,R1,R8`				IF	*stall*	**ID**	EX	MEM	WB	
`AND R6,R1,R7`					*stall*	IF	ID	EX	MEM	WB

FIGURE 6.11 The effect of the stall on the pipeline. All instructions starting with the instruction that has the dependence are delayed. With the delay, the value of the load that returns in MEM can now be forwarded to the EX cycle of the `ADD` instruction. Because of the stall, the `SUB` instruction will now read the value from the registers during its ID cycle rather than having it forwarded from the MDR.

The process of letting an instruction move from the instruction decode stage (ID) into the execution stage (EX) of this pipeline is usually called *instruction issue*; and an instruction that has made this step is said to have *issued*. For the DLX integer pipeline, all the data hazards can be checked during the ID phase of the pipeline. If a data hazard exists, the instruction is stalled before it is issued. Later in this chapter, we will look at situations where instruction issue is much more complex. Detecting interlocks early in the pipeline reduces the hardware complexity because the hardware never has to suspend an instruction that has updated the state of the machine, unless the entire machine is stalled.

Example

Suppose that 20% of the instructions are loads, and half the time the instruction following a load instruction depends on the result of the load. If this hazard creates a single-cycle delay, how much faster is the ideal pipelined machine (with a CPI of 1) that does not delay the pipeline, compared to a more realistic pipeline? Ignore any stalls other than pipeline stalls.

Answer

The ideal machine will be faster by the ratio of the CPIs. The CPI for an instruction following a load is 1.5, since they stall half the time. Since loads are 20% of the mix, the effective CPI is (0.8*1 + 0.2*1.5) = 1.1. This yields a performance ratio of $\frac{1.1}{1}$. Hence, the ideal machine is 10% faster.

Many types of stalls are quite frequent. The typical code-generation pattern for a statement such as A=B+C produces a stall for a load of the second data value. Figure 6.12 shows that the store need not result in another stall, since the result of the addition can be forwarded to the MDR. Machines where the operands may come from memory for arithmetic operations will need to stall the pipeline in the middle of the instruction to wait for memory to complete its access.

LW R1,B	IF	ID	EX	MEM	WB			
LW R2,C		IF	ID	EX	MEM	WB		
ADD R3,R1,R2			IF	ID	stall	EX	MEM	WB
SW A,R3				IF	stall	ID	EX	MEM	WB

FIGURE 6.12 The DLX code sequence for A=B+C. The ADD instruction must be stalled to allow the load of C to complete. The SW need not be delayed further because the forwarding hardware passes the result from the ALU directly to the MDR for storing.

Rather than just allow the pipeline to stall, the compiler could try to schedule the pipeline to avoid these stalls, by rearranging the code sequence to eliminate the hazard. For example, the compiler would try to avoid generating code with a load followed by an immediate use of the load destination register. This technique, called *pipeline scheduling* or *instruction scheduling,* was first used in the 1960s, and became an area of major interest in the 1980s as pipelined machines became more widespread.

Example

Generate DLX code that avoids pipeline stalls for the following sequence:

```
a = b + c;
d = e - f;
```

Assume loads have a latency of one clock cycle.

Answer

Here is the scheduled code:

```
LW  Rb,b
LW  Rc,c
LW  Re,e        ; swapped with next instruction to avoid stall
ADD Ra,Rb,Rc
LW  Rf,f
SW  a,Ra        ; store/load interchanged to avoid stall in SUB
SUB Rd,Re,Rf
SW  d,Rd
```

Both load interlocks (`LW Rc,c/ADD Ra,Rb,Rc` and `LW Rf,f/SUB Rd,Re,Rf`) have been eliminated. There is a dependence between the ALU instruction and the store, but the pipeline structure allows the result to be forwarded. Notice that the use of different registers for the first and second statements was critical for this schedule to be legal. In particular, if the variable `e` were loaded into the same register as `b` or `c`, this schedule would not be legal. In

general, pipeline scheduling can increase the register count required. In Section 6.8, we will see that this increase can be substantial for machines that can issue multiple instructions in one clock.

This technique works sufficiently well that some machines rely on software to avoid this type of hazard. A load requiring that the following instruction not use its result is called a *delayed load.* The pipeline slot after a load is often called the *load delay* or *delay slot.* When the compiler cannot schedule the interlock, a no-op instruction may be inserted. This does not affect running time, but only increases the code space versus a machine with the interlock. Whether or not the hardware detects this interlock and stalls the pipeline, performance will be enhanced if the compiler schedules instructions. If the stall occurs, the performance impact will be the same, whether the machine executes an idle cycle or executes a no-op. Figure 6.13 shows that scheduling can eliminate the majority of these delays. It is clear from this figure that load delays in GCC are significantly harder to schedule than in Spice or TeX.

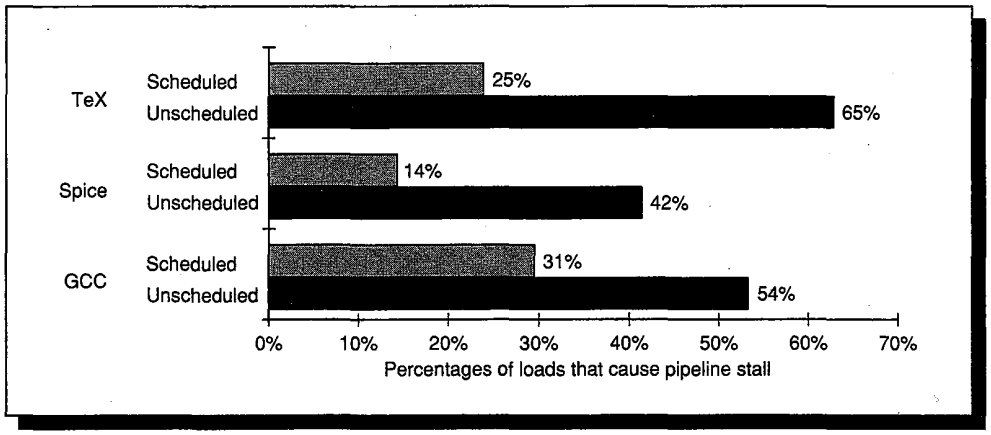

FIGURE 6.13 Percentage of the loads that result in a stall with the DLX pipeline. The black bars show the amount without compiler scheduling; the gray bars show the effect of a good, but simple, scheduling algorithm. These data show scheduling effectiveness after global optimization (see Chapter 3, Section 3.7). Global optimization actually makes scheduling relatively harder because there are fewer candidates available for scheduling into delay slots. For example, on GCC and TeX, when the programs are scheduled but not globally optimized, the percentage of load delays that result in a stall drops to 22% and 19%, respectively.

Implementing Data Hazard Detection in Simple Pipelines

How pipeline interlocks are implemented depends quite heavily on the length and complexity of the pipeline. For a complex machine with long-running instructions and multicycle interdependences, a central table that keeps track of the availability of operands and the outstanding writes may be needed (see Sec-

tion 6.7). For the DLX integer pipeline, the only interlock we need to enforce is load followed by immediate use. This can be done with a simple comparator that looks for this pattern of load destination and source. The hardware required to detect and control the load data hazard and to forward the load result is as follows:

- Additional multiplexers on the inputs to the ALU (just as was required for the bypass hardware for register–register instructions)
- Extra paths from the MDR to both multiplexer inputs to the ALU
- A buffer to save the destination-register numbers from the prior two instructions (the same as for register–register forwarding)
- Four comparators to compare the two possible source register fields with the destination fields of the prior instructions and look for a match

The comparators check for a load interlock at the beginning of the EX cycle. The four possibilities and the required actions are shown in Figure 6.14.

For DLX, the hazard detection and forwarding hardware is reasonably simple; we will see that things become much more complicated when the pipelines are very deep (Section 6.6). But before we do that, let's see what happens with branches in our DLX pipeline.

Situation	Example code sequence	Action

No dependence	`LW **R1**,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	`LW **R1**,45(R2)`
	`ADD R5,**R1**,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX.

Dependence overcome by forwarding	`LW **R1**,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,**R1**,R7`
	`OR R9,R6,R7`
		Comparators detect use of R1 in SUB and forward result of load to ALU in time for SUB to begin EX.

Dependence with accesses in order	`LW **R1**,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,**R1**,R7`
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half. See Figure 6.8 (page 262).

FIGURE 6.14 Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only compare needed is between the destination and the sources on the two instructions following the instruction that wrote the destination. In the case of a stall, the pipeline dependences will look like the third case, once execution continues.

Control Hazards

Control hazards can cause a greater performance loss for our DLX pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. (Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it falls through, it is *not taken,* or *untaken.)* If instruction *i* is a taken branch, then the PC is normally not changed until the end of MEM, after the completion of the address calculation and comparison, as shown in Figure 6.4 (page 256). This means stalling for three clock cycles, at the end of which the new PC is known and the proper instruction can be fetched. This effect is called a *control* or *branch hazard.* Figure 6.15 shows a three-cycle stall for a control hazard.

Branch instruction	IF	ID	EX	MEM	WB					
Instruction *i*+1		*stall*	*stall*	*stall*	IF	ID	EX	MEM	WB	
Instruction *i*+2			*stall*	*stall*	*stall*	IF	ID	EX	MEM	WB
Instruction *i*+3				*stall*	*stall*	*stall*	IF	ID	EX	MEM
Instruction *i*+4					*stall*	*stall*	*stall*	IF	ID	EX
Instruction *i*+5						*stall*	*stall*	*stall*	IF	ID
Instruction *i*+6							*stall*	*stall*	*stall*	IF

FIGURE 6.15 Ideal DLX pipeline stalling after a control hazard. The instruction labeled instruction *i+k* represents the *k*th instruction executed after the branch. There is a difficulty in that the branch instruction is not decoded until after instruction *i* +1 has been fetched. This figure shows the conceptual difficulty, while Figure 6.16 shows what really happens.

Branch instruction	IF	ID	EX	MEM	WB					
Instruction *i*+1		IF	*stall*	*stall*	IF	ID	EX	MEM	WB	
Instruction *i*+2			*stall*	*stall*	*stall*	IF	ID	EX	MEM	WB
Instruction *i*+3				*stall*	*stall*	*stall*	IF	ID	EX	MEM
Instruction *i*+4					*stall*	*stall*	*stall*	IF	ID	EX
Instruction *i*+5						*stall*	*stall*	*stall*	IF	ID
Instruction *i*+6							*stall*	*stall*	*stall*	IF

FIGURE 6.16 What might really happen in the DLX pipeline. Instruction *i* +1 is fetched, but the instruction is ignored and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for instruction *i* +1 is redundant. This will be addressed shortly.

The pipeline in Figure 6.15 is not possible because we don't know that the instruction is a branch until after the fetch of the next instruction. Figure 6.16 fixes this by simply redoing the fetch once the target is known.

Three clock cycles wasted for every branch is a significant loss. With a 30% branch frequency and an ideal CPI of 1, the machine with branch stalls achieves

only about half the ideal speedup from pipelining. Thus, reducing the branch penalty becomes critical. The number of clock cycles in a branch stall can be reduced in two steps:

1. Find out whether the branch is taken or not earlier in the pipeline.
2. Compute the taken PC (address of the branch target) earlier.

To optimize the branch behavior, **both** of these must be done—it doesn't help to know the target of the branch without knowing whether the next instruction to execute is the target or the instruction at PC+4. Both steps should be taken as early in the pipeline as possible.

In DLX, the branches (`BEQZ` and `BNEZ`) require testing only equality to zero. Thus, it is possible to complete this decision by the end of the ID cycle using special logic devoted to this test. To take advantage of an early decision on whether the branch is taken, both PCs (taken and not taken) must be computed early. Computing the branch target address requires a separate adder, which can add during ID. With the separate adder and a branch decision made during ID, there is only a one-clock-cycle stall on branches. Figure 6.17 shows the branch portion of the revised resource allocation table from Figure 6.4 (page 256).

In some machines, branch hazards are even more expensive in clock cycles than in our example, since the time to evaluate the branch condition and compute the destination can be even longer. For example, a machine with separate

Pipe stage	Branch instruction
IF	IR←Mem[PC]; PC←PC+4;
ID	A←Rs1; B←Rs2; PC1←PC; IR1←IR; **BTA←PC+($(IR_{16})^{16}$## $IR_{16..31}$)** **if (Rs1 *op* 0) PC←BTA**
EX	
MEM	
WB	

FIGURE 6.17 Revised pipeline structure (see Figure 6.4, page 256) showing the use of a separate adder to compute the branch target address. The operations that are new or have changed are in bold. Because the branch target address (BTA) addition happens during ID, it will happen for all instructions; the branch condition (Rs1 *op* 0) will also be done for all instructions. The last operation in ID is to replace the PC. We must know that the instruction is a branch before we perform this step. This requires decoding the instruction before the end of ID, or doing this operation at the very beginning of EX when the PC is sent out. Because the branch is done by the end of ID, the EX, MEM, and WB stages are unused for branches. An additional complication arises for jumps that have a longer offset than branches. We can resolve this by using an additional adder that sums the PC and lower 26 bits of the IR. Alternatively, we could attempt a clever scheme that does a 16-bit add in the first half of the cycle and determines whether to add in 10 bits from IR in the second half of the cycle, by decoding the jump opcodes early.

decode and register fetch stages will probably have a *branch delay*—the length of the control hazard—that is at least one clock cycle longer. The branch delay, unless it is dealt with, turns into a branch penalty. Many VAXes have branch delays of four clock cycles or more, and large, deeply pipelined machines often have branch penalties of six or seven. In general, the deeper the pipeline, the worse the branch penalty in clock cycles. Of course, the relative performance effect of a longer branch penalty depends on the overall CPI of the machine. A high CPI machine can afford to have more expensive branches because the percentage of the machine's performance that will be lost from branches is less.

Before talking about methods for reducing the pipeline penalties that can arise from branches, let's take a brief look at the dynamic behavior of branches.

Branch Behavior in Programs

Since branches can dramatically affect pipeline performance, we should look at their behavior so as to get some ideas about how the penalties of branches and jumps might be reduced. We already know the branch frequencies for our programs from Chapter 4. Figure 6.18 reviews the overall frequency of control-flow operations for three of the machines and gives the breakdown between branches and jumps.

All of the machines show a conditional branch frequency of 11%–17%, while the frequency of unconditional branches varies between 2% and 8%. An obvious

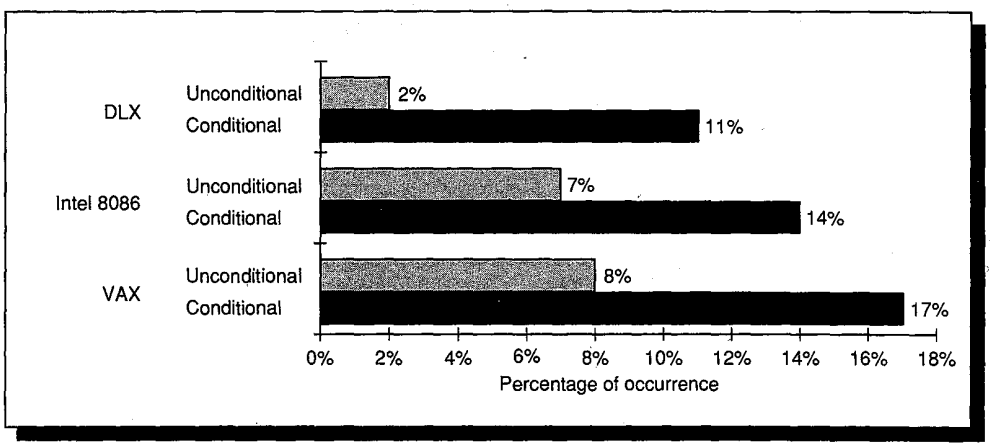

FIGURE 6.18 The frequency of instructions (branches, jumps, calls, and returns) that may change the PC. These data represent the average over the programs measured in Chapter 4. Instructions are divided into two classes: branches, which are conditional (including loop branches), and those that are unconditional (jumps, calls, and returns). The 360 is omitted because the ordinary unconditional branches are not separated from the conditional branches. Emer and Clark [1984] reported that 38% of the instructions executed in their measurements of the VAX were instructions that could change the PC. They measured that 67% of these instructions actually cause a branch in control flow. Their data were taken on a timesharing workload and reflect many uses; their measurement of branch frequency is much higher than the one in this chart.

question is, how many of the branches are taken? Knowing the breakdown between taken and untaken branches is important because this will affect strategies for reducing the branch penalties. For the VAX, Clark and Levy [1984] measured simple conditional branches to be taken with a frequency of just about 50%. Other branches, which occur much less often, have different ratios. Most bit-testing branches are not taken, and loop branches are taken with about 90% probability.

For DLX, we measured the branch behavior in Chapter 3 and summarized it in Figure 3.22 (page 107). That data showed 53% of the conditional branches are taken. Finally, 75% of the branches executed are forward-going branches. With this data in mind, let's look at ways to reduce branch penalties.

Reducing Pipeline Branch Penalties

There are several methods for dealing with the pipeline stalls due to branch delay, and four simple compile-time schemes are discussed in this section. In these schemes the predictions are static—they are fixed for each branch during the entire execution, and the predictions are compile-time guesses. More ambitious schemes using hardware to predict branches dynamically are discussed in Section 6.7.

The easiest scheme is to freeze the pipeline, holding any instructions after the branch until the branch destination is known. The attractiveness of this solution lies primarily in its simplicity. It is the solution used earlier in the pipeline shown in Figures 6.15 and 6.16.

A better and only slightly more complex scheme is to predict the branch as not taken, simply allowing the hardware to continue as if the branch were not

Untaken branch instruction	IF	**ID**	EX	MEM	WB				
Instruction i+1		**IF**	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	**ID**	EX	MEM	WB				
Instruction i+1		**IF**	**IF**	ID	EX	MEM	WB		
Instruction i+2			*stall*	IF	ID	EX	MEM	WB	
Instruction i+3				*stall*	IF	ID	EX	MEM	WB
Instruction i+4					*stall*	IF	ID	EX	MEM

FIGURE 6.19 The predict-not-taken scheme and the pipeline sequence when the branch is untaken (on the top) and taken (on the bottom). When the branch is untaken, determined during ID, we have fetched the fall through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall one clock cycle.

executed. Here, care must be taken not to change the machine state until the branch outcome is definitely known. The complexity that arises from this—that is, knowing when the state might be changed by an instruction and how to "back out" a change—might cause us to reconsider the simpler solution of flushing the pipeline. In the DLX pipeline, this *predict-not-taken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. The pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, however, we need to stop the pipeline and restart the fetch. Figure 6.19 shows both situations.

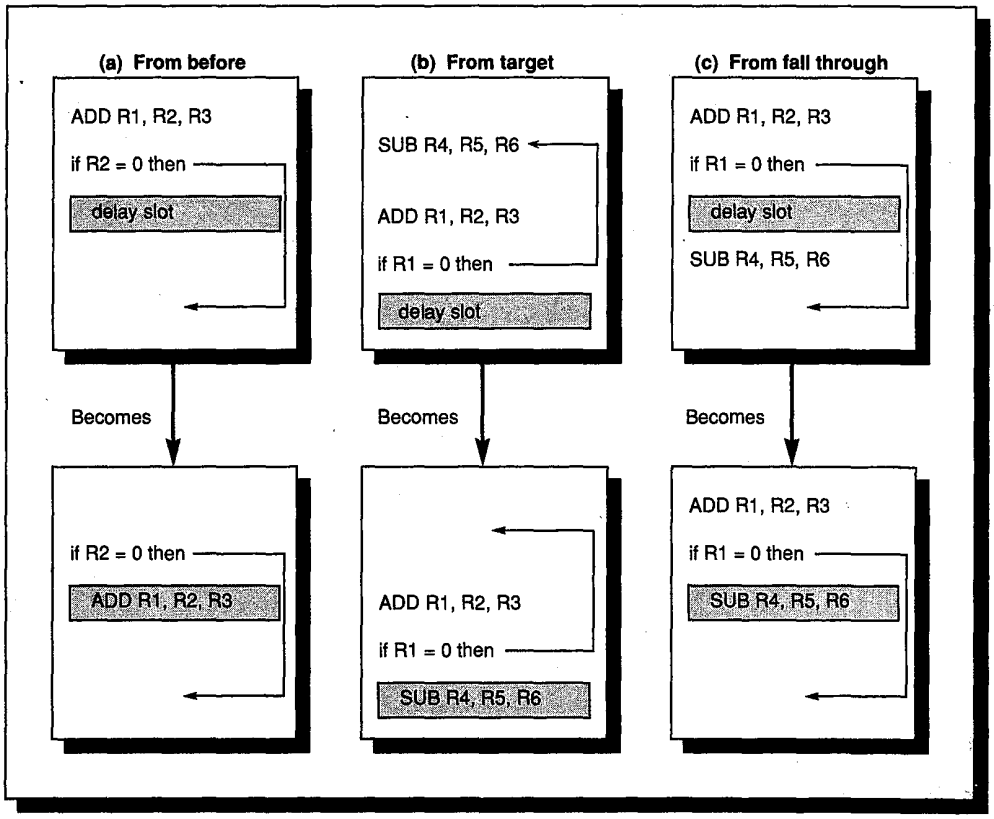

FIGURE 6.20 Scheduling the branch-delay slot. The top picture in each pair shows the code before scheduling, and the bottom picture shows the scheduled code. In (a) the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of `R1` in the branch condition prevents the `ADD` instruction (whose destination is `R1`) from being moved after the branch. In (b) the branch-delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall through, as in (c). To make this optimization legal for (b) or (c), it must be "OK" to execute the `SUB` instruction when the branch goes in the unexpected direction. By "OK" we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if `R4` were a temporary register unused when the branch goes in the unexpected direction.

An alternative scheme is to predict the branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target. Since in our DLX pipeline we don't know the target address any earlier than we know the branch outcome, there is no advantage in this approach. However, in some machines—especially those with condition codes or more powerful (and hence slower) branch conditions—the branch target is known before the branch outcome, and this scheme makes sense.

Some machines have used another technique called delayed branch, which has been used in many microprogrammed control units. In a *delayed branch*, the execution cycle with a branch delay of length n is:

```
branch instruction
sequential successor1
sequential successor2
........
sequential successorn
branch target if taken
```

The sequential successors are in the *branch-delay slots*. As with load-delay slots, the job of the software is to make the successor instructions valid and useful. A number of optimizations are used. Figure 6.20 shows the three ways in which the branch delay can be scheduled. Figure 6.21 shows the different constraints for each of these branch-scheduling schemes, as well as situations in which they win.

The primary limitations on delayed-branch scheduling arise from the restrictions on the instructions that are scheduled into the delay slots and from our ability to predict at compile time whether a branch is likely to be taken or not. Figure 6.22 shows the effectiveness of the branch scheduling in DLX with a single branch-delay slot using a simple branch-scheduling algorithm. It shows that

Scheduling strategy	Requirements	Improves performance when?
(a) From before branch	Branch must not depend on the rescheduled instructions.	Always.
(b) From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.	When branch is taken. May enlarge program if instructions are duplicated.
(c) From fall through	Must be OK to execute instructions if branch is taken.	When branch is not taken.

FIGURE 6.21 Delayed-branch–scheduling schemes and their requirements. The origin of the instruction being scheduled into the delay slot determines the scheduling strategy. The compiler must enforce the requirements when looking for instructions to schedule the delay slot. When the slots cannot be scheduled, they are filled with no-op instructions. In strategy (b), if the branch target is also accessible from another point in the program—as it would be if it were the head of a loop—the target instructions must be copied and not just moved.

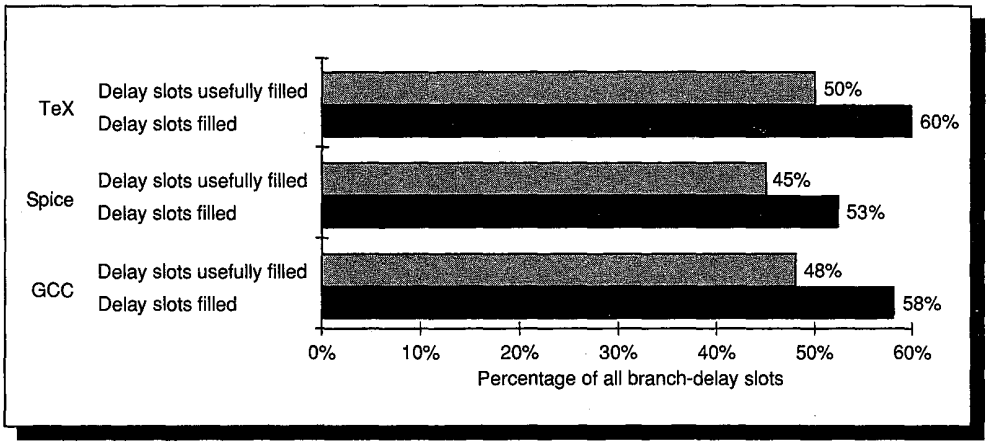

FIGURE 6.22 Frequency with which a single branch-delay slot is filled and how often the instruction is useful to the computation. The solid bar shows the percentage of the branch-delay slots occupied by some instruction other than a no-op. The difference between 100% and the dark column represents those branches that are followed by a no-op. The shaded bar shows how often those instructions do useful work. The difference between the shaded and solid bars is the percentage of instructions executed in a branch delay but not contributing to the computation. These instructions occur because optimization (b) is only useful when the branch is taken. If optimization (c) were used it would also contribute to this difference, since it is only useful when the branch is not taken.

slightly more than half the branch-delay slots are filled, and most of the filled slots do useful work. On average about 80% of the filled delay slots contribute to the computation. This number seems surprising, since branches are only taken about 53% of the time. The success rate is high because about one-half of the branch delays are being filled with an instruction from before the branch (strategy (a)), which is useful independent of whether the branch is taken.

When the scheduler in Figure 6.22 cannot use strategy (a)—moving an instruction from before the branch to fill the branch-delay slot—it uses only strategy (b)—moving it from the target. (For simplicity reasons, the schedule does not use strategy (c).) In total, nearly half the branch-delay slots are dynamically useful, eliminating one-half the branch stalls. Looking at Figure 6.22 we see that the primary limitation is the number of empty slots—those filled with no-ops. It is unlikely that the ratio of useful slots to filled slots, about 80%, can be improved, since this would require much better accuracy in predicting branches. In the Exercises we consider an extension of the delayed-branch idea that tries to fill more slots.

There is a small additional hardware cost for delayed branches. Because of the delayed effect of branches, multiple PCs (one plus the length of the delay) are needed to correctly restore the state when an interrupt occurs. Consider when the interrupt occurs after a taken-branch instruction is completed, but before all the instructions in the delay slots and the branch target are completed. In this case, the PC's of the delay slots and the PC of the branch target must be saved, since they are not sequential.

What is the effective performance of each of these schemes? The effective pipeline speedup with branch penalties is

$$\text{Pipeline speedup} = \frac{\text{Ideal CPI} * \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall cycle}}$$

If we assume that the ideal CPI is 1, then we can simplify this:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Since

Pipeline stall cycles from branches = Branch frequency * Branch penalty

we obtain:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{(1 + \text{Branch frequency} * \text{Branch penalty})}$$

Using the DLX measurements in this section, Figure 6.23 shows several hardware options for dealing with branches, along with their performances (assuming a base CPI of 1).

Scheduling scheme	Branch penalty	Effective CPI	Pipeline speedup over nonpipelined machine	Pipeline speedup over stall pipeline on branch
Stall pipeline	3	1.42	3.52	1.00
Predict taken	1	1.14	4.39	1.25
Predict not taken	1	1.09	4.59	1.30
Delayed branch	0.5	1.07	4.67	1.33

FIGURE 6.23 Overall costs of a variety of branch schemes with the DLX pipeline. These data are for our DLX pipeline using the measured control-instruction frequency of 14% and the measurements of delay-slot filling from Figure 6.22. In addition, we know that 65% of the control instructions actually change the PC (taken branches plus unconditional changes). Shown are both the resultant CPI and the speedup over a nonpipelined machine, which we assume would have a CPI of 5 without any branch penalties. The last column of the table gives the speedup over a scheme that always stalls on branches.

Remember that the numbers in this section are **dramatically** affected by the length of the pipeline delay and the base CPI. A longer pipeline delay will cause an increase in the penalty and a larger percentage of wasted time. A delay of only one clock cycle is small—many machines have minimum delays of five or more. With a low CPI, the delay must be kept small, while a higher base CPI would reduce the relative penalty from branches.

Summary: Performance of the DLX Integer Pipeline

We close this section on hazard detection and elimination by showing the total distribution of idle clock cycles for our benchmarks when run on the DLX integer pipeline with software for pipeline scheduling. Figure 6.24 shows the distribution of clock cycles lost to load delays and branch delays in our three programs, by combining the separate measurements shown in Figures 6.13 (page 268) and 6.22.

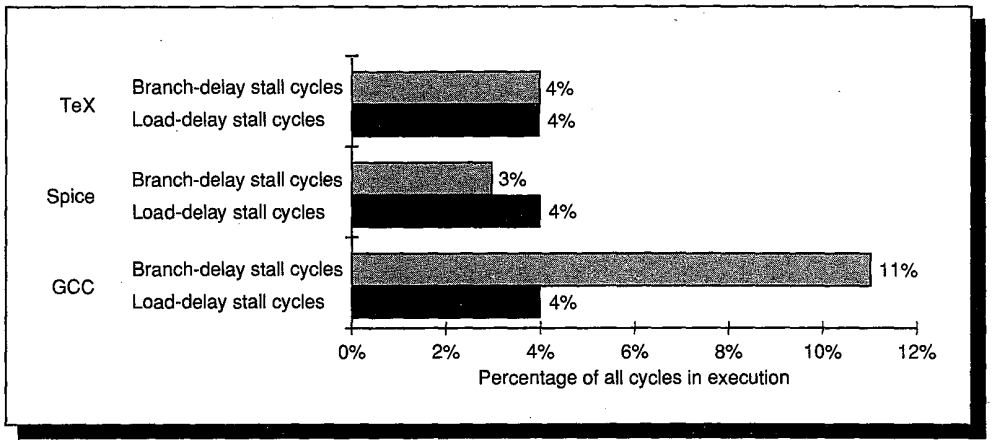

FIGURE 6.24 Percentage of the clock cycles spent on delays versus executing instructions. This assumes a perfect memory system; the clock-cycle count and instruction count would be identical if there were no integer pipeline stalls. This graph says that from 7% to 15% of the clock cycles are stalls; the remaining 85% to 93% are clock cycles that issue instructions. The Spice clock cycles do not include stalls in the FP pipeline, which will be shown at the end of Section 6.6. The pipeline scheduler fills load delays before branch delays and this affects the distribution of delay cycles.

For the GCC and TeX programs, the effective CPI (ignoring any stalls except those from pipeline hazards) on this pipelined version of DLX is 1.1. Compare this to the CPI for the complete nonpipelined, hardwired version of DLX described in Chapter 5 (Section 5.7), which is 5.8. Ignoring all other sources of stalls and assuming that the clock rates will be the same, the performance improvement from pipelining is 5.3 times.

6.5 What Makes Pipelining Hard to Implement

Now that we understand how to detect and resolve hazards, we can deal with some complications that we have avoided so far. In Chapter 5 we saw that interrupts are among the most difficult aspects of implementing a machine; pipelining increases that difficulty. In the second part of this section, we discuss some of the challenges raised by different instruction sets.

Dealing with Interrupts

Interrupts are harder to handle in a pipelined machine because the overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the machine. In a pipelined machine, an instruction is executed piece by piece and is not completed for several clock cycles. Yet in the process of executing it may need to update the machine state. Meanwhile, an interrupt can force the machine to abort the instruction's execution before it is completed.

As in nonpipelined implementations, the most difficult interrupts have two properties: (1) they occur within instructions, and (2) they must be restartable. In our DLX pipeline, for example, a virtual memory page fault resulting from a data fetch cannot occur until sometime in the MEM cycle of the instruction. By the time that fault is seen, several other instructions will be in execution. Since a page fault must be restartable and requires the intervention of another process, such as the operating system, the pipeline must be safely shut down and the state saved so that the instruction can be restarted in the correct state. This is usually implemented by saving the PC of the instruction (during IF) to restart it. If the restarted instruction is not a branch then we will continue to fetch the sequential successors and begin their execution in the normal fashion. If the restarted instruction is a branch, then we will evaluate the branch condition and begin fetching from either the target or the fall through. When an interrupt occurs, we can take the following steps to save the pipeline state safely:

1. Force a trap instruction into the pipeline on the next IF.

2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline. This prevents any state changes for instructions that will not be completed before the interrupt is handled.

3. After the interrupt-handling routine in the operating system receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the interrupt later.

When we use delayed branches it is no longer possible to re-create the state of the machine with the single PC of the interrupted instruction, because the instructions in the pipeline may not be sequentially related. In particular, when the instruction that causes the interrupt is a branch-delay slot, and the branch was taken, then the instructions to restart are those in the slot plus the instruction at the branch target. The branch itself has completed execution and is not restarted. The addresses of the instructions in the branch-delay slot and the target are not sequential. So we need to save and restore a number of PCs that is one more than the length of the branch delay. This is done in the third step above.

After the interrupt has been handled, special instructions return the machine from the interrupt by reloading the PCs and restarting the instruction stream (using RFE in DLX). If the pipeline can be stopped so that the instructions just before the faulting instruction are completed and those after it can be restarted

from scratch, the pipeline is said to have *precise interrupts*. Ideally, the faulting instruction would not have changed the state, and correctly handling some interrupts requires that the faulting instruction have no effects. For other interrupts, such as floating-point exceptions, the faulting instruction on some machines writes its result before the interrupt can be handled. In such cases, the hardware must be prepared to retrieve the source operands, even if the destination is identical to one of the source operands.

Supporting precise interrupts is a requirement in many systems, while in others it is valuable because it simplifies the operating system interface. At a minimum, any machine with demand paging or IEEE arithmetic trap handlers must make its interrupts precise, either in the hardware or with some software support.

Precise interrupts are challenging because of the same problems that make instructions difficult to restart. As we saw in the last chapter, restarting is complicated by the fact that instructions can change the state of the machine before they are **guaranteed** to complete (sometimes called *committed* instructions). Because instructions in the pipeline may have dependences, not updating the machine state is impractical if the pipeline is to keep going. Thus, as a machine is more heavily pipelined, it becomes necessary to be able to back out of any state changes made before the instruction is committed (as discussed in Chapter 5). Fortunately, DLX has no such instructions, given the pipeline we have used.

Figure 6.25 (page 281) shows the DLX pipeline stages and which "problem" interrupts might occur in each stage. Because in pipelining there are multiple instructions in execution, multiple interrupts may occur on the same clock cycle. For example, consider this instruction sequence:

LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

This pair of instructions can cause a data page fault and an arithmetic interrupt at the same time, since the `LW` is in MEM while the `ADD` is in EX. This case can be handled by dealing with only the data page fault and then restarting the execution. The second interrupt will reoccur (but not the first, if the software is correct), and when it does it can be handled independently.

In reality, the situation is not all this straightforward. Interrupts may occur out of order; that is, an instruction may cause an interrupt before an earlier instruction causes one. Consider again the above sequence of instructions `LW; ADD`. The `LW` can get a data page fault, seen when the instruction is in MEM, and the `ADD` can get an instruction page fault, seen when the `ADD` instruction is in IF. The instruction page fault will actually occur first, even though it is caused by a later instruction! This situation can be resolved in two ways. To explain them, let's call the instruction in the position of the `LW` "instruction i" and the instruction in the position of the `ADD` "instruction i+1."

Pipeline stage	Problem interrupts occurring
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation
WB	None

FIGURE 6.25 Interrupts from Chapter 5 that cause stop and restart of the DLX pipeline in a transparent fashion. The pipeline stage where these interrupts occur is also shown. Interrupts raised from instruction or data-memory access account for six out of seven cases. These interrupts and their corresponding names in other processors are in Figures 5.9 and 5.11.

The first approach is completely precise and is the simplest to understand for the user of the architecture. The hardware posts each interrupt in a status vector carried along with each instruction as it goes down the pipeline. When an instruction enters WB (or is about to leave MEM), the interrupt status vector is checked. If any interrupts are posted, they are handled in the order in which they would occur in time—the interrupt corresponding to the earliest instruction is handled first. This guarantees that all interrupts will be seen on instruction i before any are seen on i+1. Of course, any action taken on behalf of instruction i may be invalid, but because no state is changed until WB, this is not a problem in the DLX pipeline. Nevertheless, pipeline control may want to disable any actions on behalf of an instruction i (and its successors) as soon as the interrupt is recognized. For pipelines that could update state earlier than WB, this disabling is required.

The second approach is to handle an interrupt as soon as it appears. This could be regarded as slightly less precise because interrupts occur in an order different from the order they would occur in if there were no pipelining. Figure 6.26 shows two interrupts occurring in the DLX pipeline. Because the interrupt at instruction i+1 is handled when it appears, the pipeline must be stopped immediately without completing any instructions that have yet to change state. For the DLX pipeline, this will be i–2, i–1, i, and i+1, assuming the interrupt is recognized at the end of the IF stage of the `ADD` instruction. The pipeline is then restarted with instruction i–2. Since the instruction causing the interrupt can be any of i–2, ..., i+1, the operating system must determine which instruction faulted. This is easy to figure out if the type of interrupt and its corresponding pipe stage are known. For example, only i+1 (the `ADD` instruction) could get an instruction page fault at this point, and only i–2 could get a data page fault. After handling the fault for i+1 and restarting at i–2, the data page fault will be encountered on instruction i, which will cause i, ..., i+3 to be interrupted. The data page fault can then be handled.

Instruction *i*–3	IF	ID	EX	MEM	WB					
Instruction *i*–2		IF	ID	EX	MEM	*WB*				
Instruction *i*–1			IF	ID	EX	*MEM*	*WB*			
Instruction *i* (LW)				IF	ID	*EX*	*MEM*	*WB*		
Instruction *i*+1 (ADD)					**IF**	*ID*	*EX*	*MEM*	*WB*	
Instruction *i*+2						*IF*	*ID*	*EX*	*MEM*	*WB*

Instruction *i*–3	IF	ID	EX	MEM	WB					
Instruction *i*–2		IF	ID	EX	MEM	WB				
Instruction *i*–1			IF	ID	EX	MEM	WB			
Instruction *i* (LW)				IF	ID	EX	**MEM**	*WB*		
Instruction *i*+1 (ADD)					IF	ID	*EX*	*MEM*	*WB*	
Instruction *i*+2						IF	*ID*	*EX*	*MEM*	*WB*
Instruction *i*+3							*IF*	*ID*	*EX*	*MEM*
Instruction *i*+4								IF	ID	EX

FIGURE 6.26 The actions taken for interrupts occurring at different points in the pipeline and handled immediately. This shows the instructions interrupted when an instruction page fault occurs in instruction *i*+1 (in the top diagram), and a data page fault in instruction *i* in the bottom diagram. The pipe stages in bold are the cycles during which the interrupt is recognized. The pipe stages in italics are the instructions that will not be completed due to the interrupt, and will need to be restarted. Because the earliest effect of the interrupt is on the pipe stage after it occurs, instructions that are in the WB stage when the interrupt occurs will complete, while those that have not yet reached WB will be stopped and restarted.

Instruction Set Complications

Another set of difficulties arises from odd bits of state that may create additional pipeline hazards or may require extra hardware to save and restore. Condition codes are a good example of this. Many machines set the condition codes implicitly as part of the instruction. At first glance, this looks like a good idea, since condition codes decouple the evaluation of the condition from the actual branch. However, implicitly set condition codes can cause difficulties in making branches fast. They limit the effectiveness of branch scheduling because most operations will modify the condition code, making it hard to schedule instructions between the setting of the condition code and the branch. Furthermore, in machines with condition codes, the processor must decide when the branch condition is fixed. This involves finding out when the condition code has been set for the last time prior to the branch. On the VAX, most instructions set the condition code, so that an implementation will have to stall if it tries to determine the branch condition early. Alternatively, the branch condition can be evaluated by the branch late in the pipeline, but this still leads to a long branch delay. On the 360/370 many, but not all, instructions set the condition codes. Figure 6.27 shows how the situation differs on the DLX, the VAX, and the 360 for the fol-

lowing C code sequence, assuming that b and d are initially in registers R2 and R3 (and should not be destroyed):

```
a = b + d;
if (b==0) ...
```

DLX	VAX	IBM 360
ADD R1,R2,R3	ADDL3 a,R2,R3	LR R1,R2
...	...	AR R1,R3
SW a,R1	CL R2,0	ST a,R1
...	BEQL label	...
BEQZ R2,label		LTR R2,R2
		BZ label

FIGURE 6.27 Code sequence for the above two statements. Because the ADD computes the sum of b and d, and the branch condition depends only on b, an explicit compare (on R2) is needed on the VAX and 360. On DLX, the branch depends only on R2 and can be arbitrarily far away from it. (In addition the SW could be moved into the branch-delay slot.) On the VAX all ALU operations and moves set the condition codes, so that a compare must be right before the branch. On the 360, for this example the instruction load and test register (LTR) is used to set the condition code. However, most loads on the 360 do not set the condition codes; thus, a load (or a store) could be moved between the LTR and the branch.

Provided there is lots of hardware to spare, **all** instructions before the branch in the pipeline can be examined to decide when the branch is determined. Of course, architectures with explicitly set condition codes avoid this difficulty. However, pipeline control must still track the last instruction that sets the condition code to know when the branch condition is decided. In effect, the condition code must be treated as an operand requiring hazard detection for RAW hazards on branches, just as DLX must do on the registers.

A final thorny area in pipelining is multicycle operations. Imagine trying to pipeline a sequence of VAX instructions such as this:

```
MOVL R1,R2
ADDL3 42(R1),56(R1)+,@(R1)
SUBL2 R2,R3
MOVC3 @(R1)[R2],74(R2),R3
```

These instructions differ radically in the number of clock cycles they will require, from as low as one up to hundreds of clock cycles. They also require different numbers of data memory accesses, from zero to possibly hundreds. Data hazards are very complex and occur both between and within instructions.

The simple solution of making all instructions execute for the same number of clock cycles is unacceptable because it introduces an enormous number of hazards and bypass conditions, and makes an immensely long pipeline. Pipelining the VAX at the instruction level is difficult (as we will see in Section 6.9), but a clever solution was found by the VAX 8800 designers. They pipeline the microinstruction execution; because the microinstructions are simple (they look a lot like DLX), the pipeline control is much easier. While it is not clear that this approach can achieve quite as low a CPI as an instruction-level pipeline for the VAX, it is much simpler, possibly leading to a shorter clock cycle time.

Load/store machines that have simple operations with similar amounts of work pipeline more easily. If architects realize the relationship between instruction set design and pipelining, they can design architectures for more efficient pipelining. In the next section we will see how the DLX pipeline deals with long-running instructions.

6.6 Extending the DLX Pipeline to Handle Multicycle Operations

We now want to explore how our DLX pipeline can be extended to handle floating-point operations. This section concentrates on the basic approach and the design alternatives, and closes with some performance measurements of a DLX floating-point pipeline.

It is impractical to require that all DLX floating-point operations complete in one clock cycle, or even in two. Doing so would mean either accepting a slow clock or using enormous amounts of logic in the floating-point units, or both. Instead, the floating-point pipeline will allow for a longer latency for operations. This is easier to grasp if we imagine the floating-point instructions as having the same pipeline as the integer instructions, with two important changes. First, the EX cycle may be repeated as many times as needed to complete the operation; the number of repetitions can vary for different operations. Second, there may be multiple floating-point functional units. A stall will occur if the instruction to be issued will either cause a structural hazard for the functional unit it uses or cause a data hazard.

For this section let's assume that there are four separate functional units in our DLX implementation:

1. The main integer unit
2. FP and integer multiplier
3. FP adder
4. FP and integer divider

The integer unit handles all loads and stores to either register set, all the integer operations (except multiply and divide), and branches. For now we will also

assume that the execution stages of the other functional units are not pipelined, so that no other instruction using the functional unit may issue until the previous instruction leaves EX. Moreover, if an instruction cannot proceed to the EX stage, the entire pipeline behind that instruction will be stalled. Figure 6.28 shows the resulting pipeline structure. In the next section we will deal with schemes that allow the pipeline to progress when there are more functional units or when the functional units are pipelined.

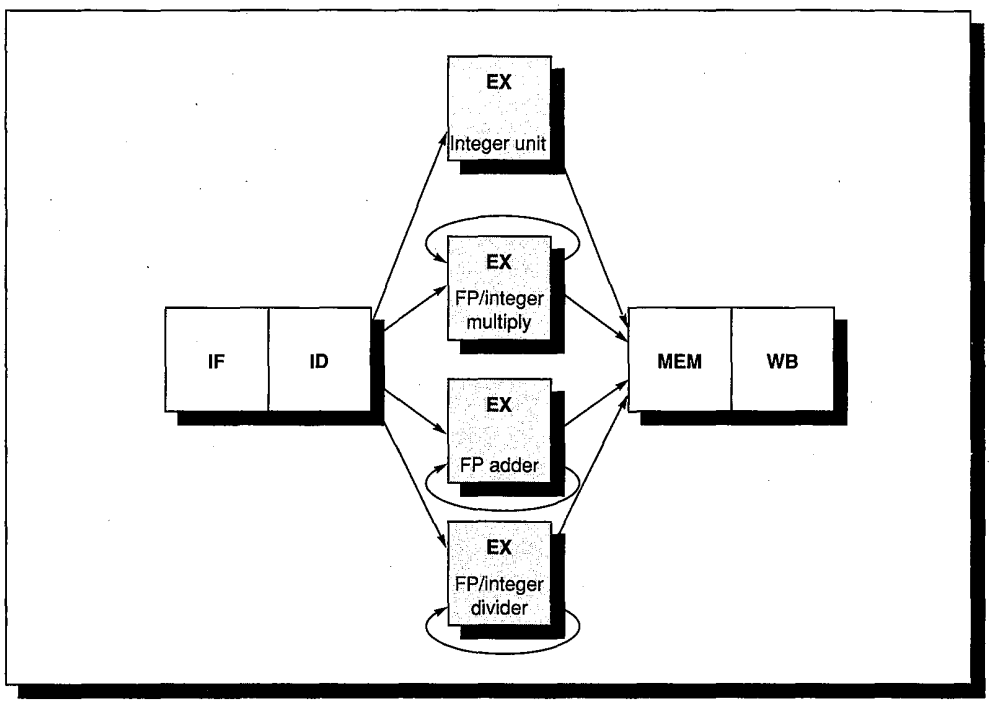

FIGURE 6.28 The DLX pipeline with three additional nonpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The floating-point operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

Since the EX stage may be repeated many times—30 to 50 repetitions for a floating-point divide would not be unreasonable—we must find a way to track long potential dependences and resolve hazards that last over tens of clock cycles, rather than just one or two. There is also the overlap between integer and floating-point instructions to deal with. However, overlapped integer and FP instructions do not complicate hazard detection, except on floating-point memory references and moves between the register sets. This is because, except for these memory references and moves, the FP and integer registers are distinct, and all integer instructions operate on the integer registers while the floating-point operations operate only on their own registers. This simplification of pipeline control is a major advantage of having separate register files for integer and floating-point data.

For now, let's assume that all floating-point operations take the same number of clock cycles—say 20 in the EX stage. What kind of hazard-detection circuitry will we need? Because all operations take the same amount of time, and register reads and writes always occur in the same stage, only RAW hazards are possible; no WAR or WAW hazards can occur. Thus, all we need to track is the destination register of each active functional unit. When we want to issue a new floating-point instruction, we take the following steps:

1. *Check for structural hazard*—Wait until the required functional unit is not busy.
2. *Check for a RAW data hazard*—Wait until the source registers are not listed as destinations by any of the EX stages in the functional units.
3. *Check for forwarding*—Test if the destination register of an instruction in MEM or WB is one of the source registers of the floating-point instruction; if so, enable the input multiplexer to use that result, rather than the register contents.

There is a small complication arising from conflicts between floating-point loads and floating-point operations when they both reach the WB stage simultaneously. We will deal presently with this situation in a more general fashion.

The above discussion assumes that the FP-functional-unit execution times were all the same. However, this does not hold up under practical scrutiny: Floating-point adds can typically be done in less than 5 clock cycles, multiplies in less than 10, and divides in about 20 or more. What we want is to allow the execution times of the functional units to differ, while still allowing the functional units to overlap execution. This would not change the basic structure of the pipeline in Figure 6.28, though it may cause the number of iterations around the loops to vary. Overlapping the execution of instructions whose running times differ, however, creates three complications: contention for register access at the end of the pipeline, the possibility of WAR and WAW hazards, and greater difficulty in providing precise interrupts.

We have already seen that FP loads and FP operations can contend for the floating-point register file on writes. When floating-point operations vary in execution time, they can also collide when trying to write results. This problem can be resolved by establishing a static priority for use of the WB stage. If multiple instructions wish to enter the MEM stage simultaneously, all instructions except the one with the highest priority are stalled in their EX stage. A simple, though sometimes suboptimal, heuristic is to give priority to the unit with the longest latency, since that is the one most likely to be the cause of the bottleneck. Although this scheme is reasonably simple to implement, this change to the DLX pipeline is quite significant. In the integer pipeline, all hazards were checked before the instruction issued to the EX stage. With this scheme for determining access to the result write port, instructions can stall after they issue.

Overlapping instructions with different execution times could introduce WAR and WAW hazards into our DLX pipeline, because the time at which

instructions write is no longer fixed. If all instructions still read their registers at the same time, no WAR hazards will be introduced.

WAW hazards are introduced because instructions can write their results in a different order than they appear. For example, consider the following code sequence:

```
DIVF    F0,F2,F4
SUBF    F0,F8,F10
```

A WAW hazard occurs between the divide and the subtract operations: The subtract will complete first, writing its result before the divide writes its result. Note that this hazard only occurs when the result of the divide will be overwritten **without** any instruction ever using it! If there were a use of F0 between the DIVF and the SUBF, the pipeline would stall because of a data dependence, and the SUBF would not issue until the DIVF was completed. We could argue that, for our pipeline, WAW hazards only occur when a useless instruction is executed, but we must still detect them and make sure that the result of the SUBF appears in F0 when we are done. (As we will see in Section 6.10, such sequences sometimes do occur in reasonable code.)

There are two possible ways to handle this WAW hazard. The first approach is to delay the issue of the subtract instruction until the DIVF enters MEM. The second approach is to stamp out the result of the divide by detecting the hazard and telling the divide unit not to write its result. Then, the SUBF can issue right away. Because this hazard is rare, either scheme will work fine—you can pick whatever is simpler to implement. As a pipeline gets more complex, however, we will need to devote increasing resources to determining when an instruction can issue.

Another problem caused by these long-running instructions can be illustrated with a very similar sequence of code:

```
DIVF    F0,F2,F4
ADDF    F10,F10,F8
SUBF    F12,F12,F14
```

This code sequence looks straightforward; there are no dependences. The problem with which we are concerned arises because an instruction issued early may complete after an instruction issued later. In this example, we can expect ADDF and SUBF to complete **before** the DIVF completes. This is called *out-of-order completion* and is common in pipelines with long-running operations. Since hazard detection will prevent any dependence among instructions from being violated, why is out-of-order completion a problem? Suppose that the SUBF causes a floating-point–arithmetic interrupt at a point where the ADDF has completed but the DIVF has not. The result will be an imprecise interrupt, something we are trying to avoid. It may appear that this could be handled by letting the floating-point pipeline drain, as we do for the integer pipeline. But the interrupt may be in a position where this is not possible. For example, if the

DIVF decided to take a floating-point–arithmetic interrupt after the add completed, we could not have a precise interrupt at the hardware level. In fact, since the ADDF destroys one of its operands, we could not restore the state to what it was before the DIVF, even with software help.

This problem is being created because instructions are completing in a different order from the order in which they were issued. There are four possible approaches to dealing with out-of-order completion. The first is to ignore the problem and settle for imprecise interrupts. This approach was used in the 1960s and early 1970s. It is still used in some supercomputers, where certain classes of interrupts are not allowed or are handled by the hardware without stopping the pipeline. But it is difficult to use this approach in most machines built today, due to features such as virtual memory and the IEEE floating-point standard, which essentially require precise interrupts, through a combination of hardware and software.

A second approach is to queue the results of an operation until all the operations that were issued earlier are complete. Some machines actually use this solution, but it becomes expensive when the difference in running times among operations is long, since the number of results to queue can become large. Furthermore, results from the queue must be bypassed so as to continue issuing instructions while waiting for the longer instruction. This requires a large number of comparators and a very large multiplexer. There are two viable variations on this basic approach. The first is a *history file*, used in the CYBER 180/990. The history file keeps track of the original values of registers. When an interrupt occurs and the state must be rolled back earlier than some instruction that completed out of order, the original value of the register can be restored from the history file. A similar technique is used for autoincrement and autodecrement addressing on machines like VAXes. Another approach, the *future file*, proposed by J. Smith and Plezkun [1988], keeps the newer value of a register; when all earlier instructions have completed, the main register file is updated from the future file. On an interrupt, the main register file has the precise values for the interrupted state.

A third technique in use is to allow the interrupts to become somewhat imprecise, but keep enough information so that the trap-handling routines can create a precise sequence for the interrupt. This means knowing what operations were in the pipeline and their PCs. Then, after handling a trap, the software finishes any instructions that precede the latest instruction completed, and the sequence can restart. Consider the following worst-case code sequence:

Instruction$_1$—a long-running instruction that eventually interrupts execution

Instruction$_2$, ... , instruction$_{n-1}$—a series of instructions that are not completed

Instruction$_n$—an instruction that is finished

Given the PCs of all the instructions in the pipeline and the interrupt return PC, the software can find the state of instruction$_1$ and instruction$_n$. Since instruction$_n$ has completed, we will want to restart execution at instruction$_{n+1}$. After

handling the interrupt, the software must simulate the execution of instruction$_1$, ... , instruction$_{n-1}$. Then we can return from the interrupt and restart at instruction$_{n+1}$. The complexity of executing these instructions properly by the handler is the major difficulty of this scheme. There is an important simplification: If instruction$_2$, ... , instruction$_n$ are all integer instructions, then we know that if instruction$_n$ has completed, all of instruction$_2$, ... , instruction$_{n-1}$ have also completed. Thus, only floating-point operations need to be handled. To make this scheme tractable the number of floating-point instructions that can be overlapped in execution can be limited. For example, if we only overlap two instructions, then only the interrupting instruction need be completed by software. This restriction may reduce the potential throughput if the FP pipelines are deep or if there is a significant number of FP functional units. This approach is used in the SPARC architecture to allow overlap of floating-point and integer operations.

The final technique is a hybrid scheme that allows the instruction issue to continue only if it is certain that all the instructions before the issuing instruction will complete without causing an interrupt. This guarantees that when an interrupt occurs, no instructions after the interrupting one will be completed, and all of the instructions before the interrupting one can be completed. This sometimes means stalling the machine to maintain precise interrupts. To make this scheme work, the floating-point functional units must determine if an interrupt is possible early in the EX stage (in the first three clock cycles in the DLX pipeline), so as to prevent further instructions from completing. This scheme is used in the MIPS R2000/3000 architecture and is discussed further in Appendix A, Section A.7.

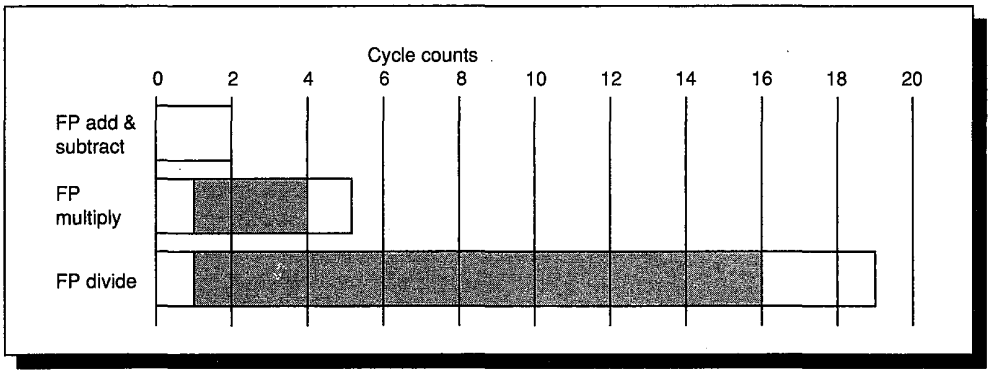

FIGURE 6.29 Total clock cycle count and permissible overlap among double-precision, floating-point operations on the MIPS R2010/3010 FP unit. The overall length of the bar shows the total number of EX cycles required to complete the operation. For example, after five clock cycles a multiply result is available. The shaded regions are times during which FP operations can be overlapped. As is common in most FP units, some of the FP logic is shared—the rounding logic, for example, is often shared. This means that FP operations with different running times cannot overlap arbitrarily. Also note that multiply and divide are not pipelined in this FP unit, so only one multiply or divide can be outstanding. The motivation for this pipeline design is discussed further in Appendix A (page A-31).

Performance of a DLX FP Pipeline

To look at the FP pipeline performance of DLX, we need to specify the latency and issue restrictions for the FP operations. We have chosen to use the pipeline structure of the MIPS R2010/3010 FP unit. While this unit has some structural hazards, it tends to have low-latency FP operations compared to most other FP units. The latencies and issue restrictions for DP floating-point operations are depicted in Figure 6.29 (page 289).

Figure 6.30 gives the breakdown of integer and floating-point stalls for Spice. There are four classes of stalls: load delays, branch delays, floating-point structural delays, and floating-point data hazards. The compiler tries to schedule both load and FP delays before it schedules branch delays. Interestingly, about 27% of the time in Spice is spent waiting for a floating-point result. Since the structural hazards are small, further pipelining of the floating-point unit would not gain much. In fact, the impact might easily be negative if the floating-point pipeline latency became longer.

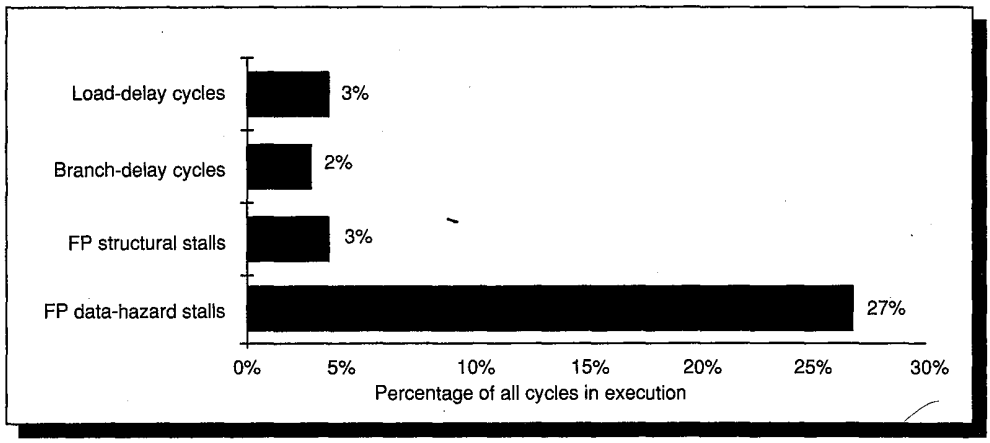

FIGURE 6.30 Percentage of clock cycles in Spice that are pipeline stalls. This again assumes a perfect memory system with no memory-system stalls. In total, 35% of the clock cycles in Spice are stalls, and without any stalls Spice would run about 50% faster. The percentage of stalls differs from Figure 6.24 (page 278) because this cycle count includes all the FP stalls, while the previous graph includes only the integer stalls.

6.7 Advanced Pipelining— Dynamic Scheduling in Pipelines

So far we have assumed that our pipeline fetches an instruction and issues it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction. If there is a data dependence, then we stall the instruction and cease fetching and issuing until the dependence is cleared. Software is responsible for scheduling the instructions to minimize these stalls. This

approach, which is called *static scheduling*, while first used in the 1960s, has become popular more recently. Many of the earlier, heavily pipelined machines used *dynamic scheduling*, whereby the hardware rearranges the instruction execution to reduce the stalls.

Dynamic scheduling offers a couple of advantages: It enables handling some cases when dependences are unknown at compile time, and it simplifies the compiler. It also allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. As we will see, these advantages are gained at a significant increase in hardware complexity. The first two parts of this section deal with reducing the cost of data dependences, especially in deeply pipelined machines. Corresponding to the dynamic hardware techniques for scheduling around data dependences are dynamic techniques for handling branches. These techniques are used for two purposes: to predict whether a branch will be taken, and to find the target more quickly. *Hardware branch prediction*, the name for these techniques, is the topic of the third part of this advanced section.

Dynamic Scheduling Around Hazards with a Scoreboard

The major limitation of the pipelining techniques we have used so far is that they all use in-order instruction issue. If an instruction is stalled in the pipeline, no later instructions can proceed. If there are multiple functional units, these units could lie idle. So, if instruction *j* depends on a long-running instruction *i*, currently in execution in the pipeline, then all instructions after *j* must be stalled until *i* is finished and *j* can execute. For example, consider this code:

```
DIVF   F0,F2,F4
ADDF   F10,F0,F8
SUBF   F6,F6,F14
```

The `SUBF` instruction cannot execute because the dependence of `ADDF` on `DIVF` causes the pipeline to stall; yet `SUBF` does not depend on anything in the pipeline. This is a performance limitation that can be eliminated by not requiring instructions to execute in order.

In the DLX pipeline, both structural and data hazards were checked at ID: When an instruction could execute properly, it was issued from ID. To allow us to begin executing the `SUBF` in the above example, we must separate the issue process into two parts: checking the structural hazards, and waiting for the absence of a data hazard. We can still check for structural hazards when we issue the instruction; thus, we still use in-order instruction issue. However, we want the instructions to begin execution as soon as their data operands are available. Thus, the pipeline will do *out-of-order execution*, which obviously implies *out-of-order completion.*

In introducing out-of-order execution, we have essentially split two pipe stages of DLX into three pipe stages. The two stages in DLX were:

1. ID—decode instruction, check for all hazards, and fetch operands
2. EX—execute instruction

In the DLX pipeline all instructions passed through issue stage in order, and a stalled instruction in ID caused a stall for all instructions behind it. The three stages we will need to allow out-of-order execution are:

1. Issue—decode instructions, check for structural hazards
2. Read operands—wait until no data hazards, then read operands
3. Execute

These three stages replace the ID and EX stages in the simple DLX pipeline.

While all instructions pass through the issue stage in order (in-order issue), they can be stalled or bypass each other in the second stage (read operands), and thus enter execution out of order. *Scoreboarding* is a technique for allowing instructions to execute out of order when there are sufficient resources and no data

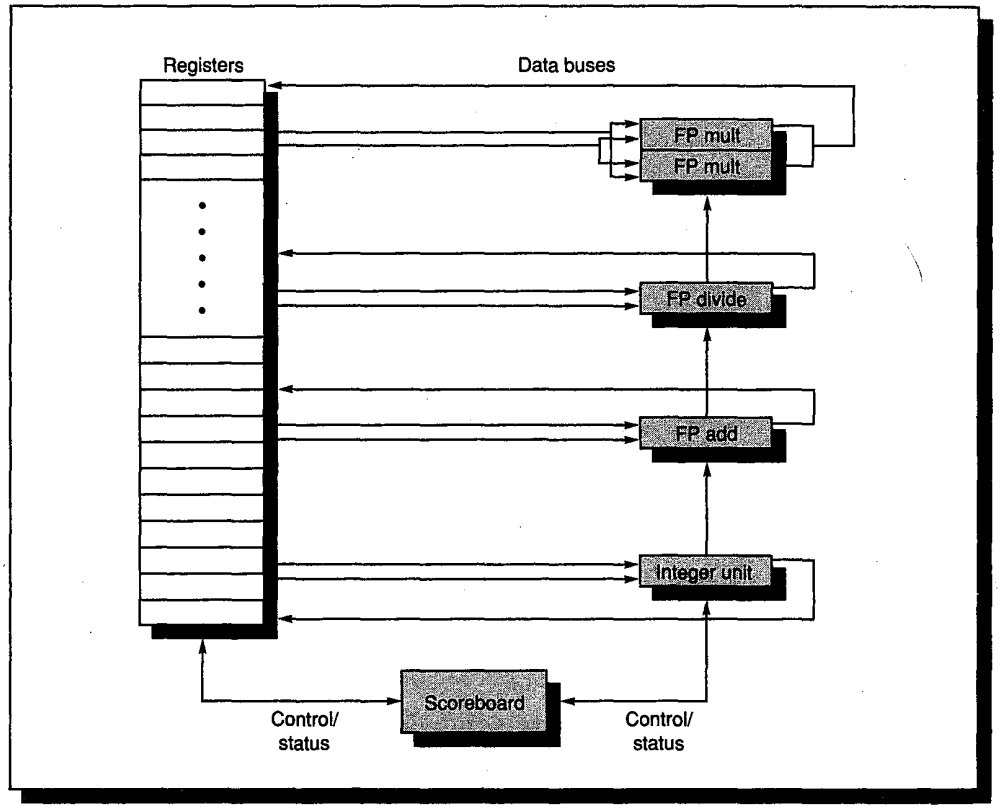

FIGURE 6.31 This shows the basic structure of a DLX machine with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All data flows between the register file and the functional units over the buses (the horizontal lines, called trunks in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units. The details of the scoreboard are shown in Figures 6.32–6.35.

dependences; it is named after the CDC 6600 scoreboard, which developed this capability.

Before we see how scoreboarding could be used in the DLX pipeline, it is important to observe that WAR hazards, which did not exist in the DLX floating-point or integer pipelines, may exist when instructions are executed out of order. Assume our earlier example has changed so that the SUBF destination is F8. If ADDF and SUBF use two different functional units, then it is possible to execute the SUBF before the ADDF, but it will yield an incorrect result if ADDF has not read F8 before SUBF writes its result. The hazard for this case can be avoided by two rules: (1) read registers only during Read Operands, and (2) queue both the ADDF operation **and** copies of its operands. Of course, WAW hazards must still be detected, such as would occur if the destination of the SUBF were F10. This WAW hazard can be eliminated by stalling the issue of the SUBF instruction.

The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the instruction at the front of the queue is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction. The scoreboard takes full responsibility for instruction issue and execution, including all hazard detection. Taking advantage of out-of-order execution requires multiple instructions to be in their EX stage simultaneously. This can be achieved with either multiple functional units or with pipelined functional units. Since these two capabilities—pipelined functional units and multiple functional units—are essentially equivalent for the purposes of pipeline control, we will assume the machine has multiple functional units.

The CDC 6600 had 16 separate functional units, including 4 floating-point units, 5 units for memory references, and 7 units for integer operations. On DLX, scoreboards make sense only on the floating-point unit. Let's assume that there are two multipliers, one adder, one divide unit, and a single integer unit for all memory references, branches, and integer operations. Although this example is much smaller than the CDC 6600, it is sufficiently powerful to demonstrate the principles. Because both DLX and the CDC 6600 are load/store, the techniques are nearly identical for the two machines. Figure 6.31 shows what the machine looks like.

Every instruction goes through the scoreboard, where a picture of the data dependences is constructed; this step corresponds to instruction issue and replaces part of the ID step in the DLX pipeline. This picture then determines when the instruction can read its operands and begin execution. If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction can execute. The scoreboard also controls when an instruction can write its result into the destination register. Thus, all hazard detection and resolution is centralized in the scoreboard. We will see a picture of the scoreboard later (Figure 6.32 on page 296), but first we need to understand the steps in the issue and execution segment of the pipeline.

Each instruction undergoes four steps in executing. (Since we are concentrating on the FP operations, we will not consider a step for memory access.) Let's first examine the steps informally and then look in detail at how the scoreboard keeps the necessary information that determines when to progress from one step to the next. The four steps, which replace the ID, EX, and WB steps in the standard DLX pipeline, are as follows:

1. Issue—If a functional unit for the instruction is free and no other active instruction has the same destination register, the scoreboard issues the instruction to the functional unit and updates its internal data structure. By ensuring that no other active functional unit wants to write its result into the destination register, we guarantee that WAW hazards cannot be present. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared. This step replaces a portion of the ID step in the DLX pipeline.

2. Read operands—The scoreboard monitors the availability of the source operands. A source operand is available if no active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order. This step, together with Issue, completes the function of the ID step in the simple DLX pipeline.

3. Execution—The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution. This step replaces the EX step in the DLX pipeline and takes multiple cycles in the DLX FP pipeline.

4. Write result—Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards. A WAR hazard exists if there is a code sequence like the following:

```
DIVF   F0,F2,F4
ADDF   F10,F0,F8
SUBF   F8,F8,F14
```

`ADDF` has a source operand F8, which is the same register as the destination of `SUBF`. But `ADDF` actually depends on an earlier instruction. The scoreboard will still stall the `SUBF` until `ADDF` reads its operands. In general, then, a completing instruction cannot be allowed to write its results when

- there is an instruction that has not read its operands,
- one of the operands is the same register as the result of the completing instruction, and
- the other operand was the result of an earlier instruction.

If this WAR hazard does not exist, or when it clears, the scoreboard tells the functional unit to store its result to the destination register. This step replaces the WB step in the simple DLX pipeline.

Based on its own data structure, the scoreboard controls the instruction progression from one step to the next by communicating with the functional units. But there is a small complication: There is only a limited number of source operands and result buses to the register file. The scoreboard must guarantee that the number of functional units allowed to proceed into steps 2 and 4 do not exceed the number of buses available. We will not go into further detail on this, other than to mention that the CDC 6600 solved this problem by grouping the 16 functional units together into four groups and supplying a set of buses, called *data trunks*, for each group. Only one unit in a group could read its operands or write its result during a clock.

Now let's look at the detailed data structure maintained by a DLX scoreboard with five functional units. Figure 6.32 (page 296) shows what the scoreboard's information looks like for a simple sequence of instructions:

```
LF      F6,34(R2)
LF      F2,45(R3)
MULTF   F0,F2,F4
SUBF    F8,F6,F2
DIVF    F10,F0,F6
ADDF    F6,F8,F2
```

There are three parts to the scoreboard:

1. Instruction status—Indicates which of the four steps the instruction is in.
2. Functional unit status—Indicates the state of the functional unit (FU). There are nine fields for each functional unit:

   Busy—Indicates whether the unit is busy or not

   Op—Operation to perform in the unit (e.g., add or subtract)

   Fi—Destination register

   Fj,Fk—Source-register numbers

   Qj,Qk—Number of the units producing source registers Fj, Fk

   Rj,Rk—Flags indicating when Fj, Fk are ready; fields are reset when new values are read so that the scoreboard knows that the source operand has been read (this is required to handle WAR hazards)
3. Register result status—Indicates which functional unit will write a register, if an active instruction has the register as its destination.

Instruction status					
Instruction		**Issue**	**Read operands**	**Execution complete**	**Write result**
LF	F6,34(R2)	√	√	√	√
LF	F2,45(R3)	√	√	√	
MULTF	F0,F2,F4	√			
SUBF	F8,F6,F2	√			
DIVF	F10,F0,F6	√			
ADDF	F6,F8,F2				

Functional unit status										
FU no.	**Name**	**Busy**	**Op**	**Fi**	**Fj**	**Fk**	**Qj**	**Qk**	**Rj**	**Rk**
1	Integer	Yes	Load	F2	R3				No	No
2	Mult1	Yes	Mult	F0	F2	F4	1		No	Yes
3	Mult2	No								
4	Add	Yes	Sub	F8	F6	F2		1	Yes	No
5	Divide	Yes	Div	F10	F0	F6	2		No	Yes

Register result status									
	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
FU no.	2	1			4	5			

FIGURE 6.32 Components of the scoreboard. Each instruction that has issued or is pending issue has an entry in the instruction-status table. There is one entry in the functional-unit–status table for each functional unit. Once an instruction issues, the record of its operands is kept in the functional-unit–status table. Finally, the register-result table indicates which unit will produce each pending result; the number of entries is equal to the number of registers. The instruction-status register says that (1) the first LF has completed and written its result, and (2) the second LF has completed execution but has not yet written its result. The MULTF, SUBF, and DIVF have all issued but are stalled, waiting for their operands. The functional-unit status says that the first multiply unit is waiting for the integer unit, the add unit is waiting for the integer unit, and the divide unit is waiting for the first multiply unit. The ADDF instruction is stalled due to a structural hazard; it will clear when the SUBF completes. If an entry in one of these scoreboard tables is not being used, it is left blank. For example, the Rk field is not used on a load, and the Mult2 unit is unused, hence its fields have no meaning. Also, once an operand has been read, the Rj and Rk fields are set to No. These are left blank to minimize the complexity of the tables.

Now let's look at how the code sequence begun in Figure 6.32 continues execution. After that, we will be able to examine in detail the conditions that the scoreboard uses to control execution.

Example

Assume the following EX cycle latencies for the floating-point functional units: Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Using the code segment in Figure 6.32, and beginning with the point indicated by the instruction status in Figure 6.32, show what the status tables look like when MULTF and DIVF are each ready to go to the write-result state.

Answer

There are RAW data hazards from the second LF to MULTF and SUBF, from MULTF to DIVF, and from SUBF to ADDF. There is a WAR data hazard between DIVF and ADDF. Finally, there is a structural hazard on the add functional unit for ADDF. What the tables look like when MULTF and DIVF are ready to go to write result are shown in Figures 6.33 and 6.34, respectively.

Instruction status				
Instruction	**Issue**	**Read operands**	**Execution complete**	**Write result**
LF F6,34(R2)	√	√	√	√
LF F2,45(R3)	√	√	√	√
MULTF F0,F2,F4	√	√	√	
SUBF F8,F6,F2	√	√	√	√
DIVF F10,F0,F6	√			
ADDF F6,F8,F2	√	√	√	

Functional unit status										
FU no.	**Name**	**Busy**	**Op**	**Fi**	**Fj**	**Fk**	**Qj**	**Qk**	**Rj**	**Rk**
1	Integer	No								
2	Mult1	Yes	Mult	F0	F2	F4			No	No
3	Mult2	No								
4	Add	Yes	Add	F6	F8	F2			No	No
5	Divide	Yes	Div	F10	F0	F6	2		No	Yes

Register result status									
	F0	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
FU no.	2			4		5			

FIGURE 6.33 Scoreboard tables just before the MULTF goes to write result. The DIVF has not yet read its operands, since it has a dependence on the result of the multiply. The ADDF has read its operands and is in execution, although it was forced to wait until the SUBF finished to get the functional unit. ADDF cannot proceed to write result because of the WAR hazard on F6, which is used by the DIVF.

Instruction status					
Instruction		Issue	Read operands	Execution complete	Write result
LF	F6,34(R2)	√	√	√	√
LF	F2,45(R3)	√	√	√	√
MULTF	F0,F2,F4	√	√	√	√
SUBF	F8,F6,F2	√	√	√	√
DIVF	F10,F0,F6	√	√	√	
ADDF	F6,F8,F2	√	√	√	√

Functional unit status										
FU no.	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
1	Integer	No								
2	Mult1	No								
3	Mult2	No								
4	Add	No								
5	Divide	Yes	Div	F10	F0	F6			No	No

Register Result status									
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU no.						5			

FIGURE 6.34 Scoreboard tables just before the DIVF goes to write result. ADDF was able to complete as soon as DIVF passed through read operands and got a copy of F6. Only the DIVF remains to finish.

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not result(D)	Busy(FU)← yes; Result(D)←FU; Op(FU)←*op*; Fi(FU)←D; Fj(FU)←S1; Fk(FU)←S2; Qj←Result (S1); Qk←Result(S2); Rj← not Qj; Rk← not Qk
Read operands	Rj and Rk	Rj←No; Rk←No
Execution complete	Functional unit done	
Write result	∀*f*((Fj(*f*)≠Fi(FU) or Rj(*f*)=No) & (Fk(*f*) ≠Fi(FU) or Rk(*f*)=No))	∀*f*(if Qj(*f*)=FU then Rj(*f*)←Yes); ∀*f*(if Qk(*f*)=FU then Rk(*f*)←Yes); Result(Fi(FU))←Clear; Busy(FU)←No

FIGURE 6.35 Required checks and bookkeeping actions for each step in instruction execution. FU stands for the functional unit used by the instruction, D is the destination register, S1 and S2 are the source registers, and *op* is the operation to be done. To access the scoreboard entry named F_j for functional unit FU we use the notation F_j(FU). Result(D) is the value of the result register field for register D. The test on the write-result case prevents the write when there is a WAR hazard. For simplicity we assume that all of the bookkeeping operations are done in one clock cycle.

Now we can see how the scoreboard works in detail by looking at what has to happen for the scoreboard to allow each instruction to proceed. Figure 6.35 shows what the scoreboard requires for each instruction to advance and the bookkeeping action necessary when the instruction does advance.

The costs and benefits of scoreboarding are an interesting question. The CDC 6600 designers measured a performance improvement of 1.7 for FORTRAN programs and 2.5 for hand-coded assembly language. However, this was measured in the days before software pipeline scheduling, semiconductor main memory, and caches (which lower memory-access time). The scoreboard on the CDC 6600 had about as much logic as one of the functional units, which is surprisingly low. The main cost was in the large number of buses—about four times as many as would be required if the machine only executed instructions in order (or if it only initiated one instruction per Execute cycle).

The scoreboard does not handle a few situations as well as it might. For example, when an instruction writes its result, a dependent instruction in the pipeline must wait for access to the register file because all results are written through the register file and never forwarded. This increases the latency and limits the ability of multiple instructions waiting for a result to initiate. WAW hazards would be very infrequent, so the stalls they cause are probably not a significant concern in the CDC 6600. However, in the next section we will see that dynamic scheduling offers the possibility of overlapping the execution of multiple iterations of a loop. To do this effectively requires a scheme for handling WAW hazards, which are likely to increase in frequency when multiple iterations are overlapped.

Another Dynamic Scheduling Approach—The Tomasulo Algorithm

Another approach to parallel execution around hazards was used by the IBM 360/91 floating-point unit. This scheme was credited to R. Tomasulo and is named after him. The IBM 360/91 was completed about three years after the CDC 6600, before caches appeared in commercial machines. IBM's goal was to achieve high floating-point performance from an instruction set and from compilers designed for the entire 360 computer family, rather than for only floating-point–intensive applications. Remember that the 360 architecture has only four double-precision floating-point registers, which limits the effectiveness of compiler scheduling; this fact was another motivation for the Tomasulo approach. Lastly, the IBM 360/91 had long memory accesses and long floating-point delays, which the Tomasulo algorithm was designed to overcome. At the end of the section, we will see that Tomasulo's algorithm can also support the overlapped execution of multiple iterations of a loop.

We will explain the algorithm, which focuses on the floating-point unit, in the context of a pipelined, floating-point unit for DLX. The primary difference between DLX and the 360 is the presence of register–memory instructions in the latter machine. Because Tomasulo's algorithm uses a load functional unit, no

significant changes are needed to add register–memory addressing modes; the primary addition is another bus. The IBM 360/91 also had pipelined functional units, rather than multiple functional units. The only difference between these is that a pipelined unit can start at most one operation per clock cycle. Since there are really no fundamental differences, we describe the algorithm as if there were multiple functional units. The IBM 360/91 could accommodate three operations for the floating-point adder and two for the floating-point multiplier. In addition, up to six floating-point loads, or memory references, and up to three floating-point stores could be outstanding. Load data buffers and store data buffers are used for this function. Although we will not discuss the load and store units, we do need to include the buffers for operands.

Tomasulo's scheme shares many ideas with the CDC 6600 scoreboard, so we assume the reader has understood the scoreboard thoroughly. There are, however, two significant differences. First, hazard detection and execution control are distributed—*reservation stations* at each functional unit control when an instruction can begin execution at that unit. This function is centralized in the scoreboard on the CDC 6600. Second, results are passed directly to functional units rather than going through the registers. The IBM 360/91 has a common result bus (called the *common data bus,* or CDB) that allows all units waiting for an operand to be loaded simultaneously. The CDC 6600 writes results into registers, where waiting functional units may have to contend for them. Also, the CDC 6600 has multiple completion buses (two in the floating-point unit), while the IBM 360/91 has only one.

Figure 6.36 shows the basic structure of a Tomasulo-based floating-point unit for DLX; none of the execution control tables are shown. The reservation stations hold instructions that have been issued and are awaiting execution at a functional unit, as well as the information needed to control the instruction once it has begun execution to the unit. The load buffers and store buffers hold data coming from and going to memory. The floating-point registers are connected by a pair of buses to the functional units and by a single bus to the store buffers. All results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer. All the buffers and reservation stations have tag fields, employed by hazard control.

Before we describe the details of the reservation stations and the algorithm, let's look at the steps an instruction goes through—just as we did for the scoreboard. Since operands are transmitted differently than in a scoreboard, there are only three steps:

1. Issue—Get an instruction from the floating-point operation queue. If the operation is a floating-point operation, issue it if there is an empty reservation station, and send the operands to the reservation station if they are in the registers. If the operation is a load or store, it can issue if there is an available buffer. If there is not an empty reservation station or an empty buffer, then there is a structural hazard and the instruction stalls until a station or buffer is freed.

2. Execute—If one or more of the operands is not yet available, monitor the CDB while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available, execute the operation.

3. Write result—When the result is available, write it on the CDB and from there into the registers and any functional units waiting for this result.

Although these steps are fundamentally similar to those in the scoreboard, there are three important differences. First, there is no checking for WAW and WAR hazards—these are eliminated as a byproduct of the algorithm, as we will see shortly. Second, the CDB is used to broadcast results rather than waiting on the registers. Third, the loads and stores are treated as basic functional units.

The data structures used to detect and eliminate hazards are attached to the reservation stations, the register file, and the load and store buffers. Although different information is attached to different objects, everything except the load

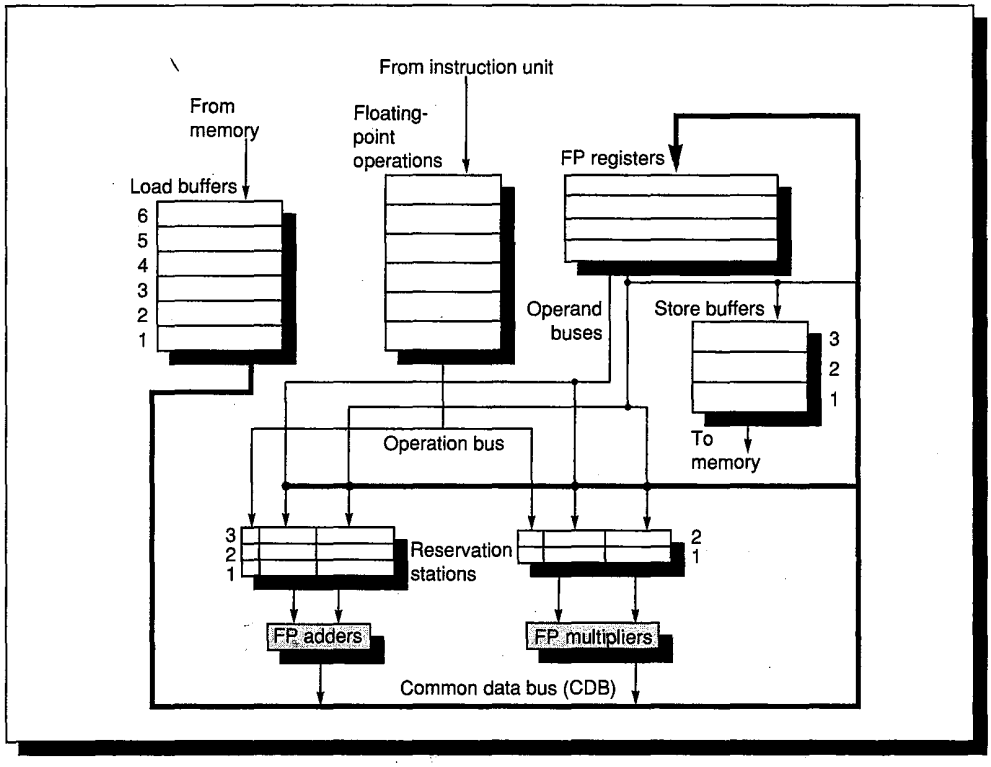

FIGURE 6.36 The basic structure of a DLX FP unit using Tomasulo's algorithm. Floating-point operations are sent from the instruction unit into a queue (called the FLOS, or floating-point operation stack, in the IBM 360/91) when they are issued. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. There are load buffers to hold the results of outstanding loads and store buffers to hold the addresses of outstanding stores waiting for their operands. All results from either the FP units or the load unit are put on the common data bus (CDB), which goes to the FP register file as well as the reservation stations and store buffers. The FP adders implement addition and subtraction, while the FP multipliers do multiplication and division.

buffers contains a tag field per entry. The tag field is a four-bit quantity that denotes one of the five reservation stations or one of the six load buffers. The tag field is used to describe which functional unit will produce a result needed as a source operand. Unused values, such as zero, indicate that the operand is already available. In describing the information, the scoreboard names are used wherever this will not lead to confusion. The names used by the IBM 360/91 are also shown. It is important to remember that the tags in the Tomasulo scheme refer to the buffer or unit that will produce a result; the register number is discarded when an instruction issues to a reservation station.

Each reservation station has six fields:

Op—The operation to perform on source operands S1 and S2.

Qj,Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vi or Vj, or is unnecessary. The IBM 360/91 calls these SINKunit and SOURCEunit.

Vj,Vk—The value of the source operands. These are called SINK and SOURCE on the IBM 360/91. Note that only one of the V field or the Q field is valid for each operand.

Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

The register file and store buffer each have a field, Qi:

Qi—The number of the functional unit that will produce a value to be stored into this register or into memory. If the value of Qi is zero, no currently active instruction is computing a result destined for this register or buffer. For a register, this means the value is given by the register contents.

The load and store buffers each require a busy field, indicating when a buffer is available due to completion of a load or store assigned there. The store buffer also has a field V, the value to be stored.

Before we examine the algorithm in detail, let's see what the system of tables looks like for the following code sequence:

```
1. LF     F6,34(R2)
2. LF     F2,45(R3)
3. MULTF  F0,F2,F4
4. SUBF   F8,F6,F2
5. DIVF   F10,F0,F6
6. ADDF   F6,F8,F2
```

We saw what the scoreboard looked like for this program when only the first load had written its result. Figure 6.37 depicts the reservation stations, load and

store buffers, and the register tags. The numbers appended to the names add, mult, and load stand for the tag for that reservation station—Add1 is the tag for the result from the first add unit. In addition we have included a central table called "Instruction status." This table is included only to help the reader understand the algorithm; it is **not** actually a part of the hardware. Instead, the state of each operation that has issued is kept in a reservation station.

There are two important differences from scoreboards that are observable in these tables. First, the value of an operand is stored in the reservation station in one of the V fields as soon as it is available; it is not read from the register file once the instruction has issued. Second, the ADDF instruction has issued. This was blocked in the scoreboard by a structural hazard.

Instruction status

Instruction		Issue	Execute	Write result
LF	F6,34(R2)	√	√	√
LF	F2,45(R3)	√	√	
MULTF	F0,F2,F4	√		
SUBF	F8,F6,F2	√		
DIVF	F10,F0,F6	√		
ADDF	F6,F8,F2	√		

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk
Add1	Yes	SUB	(Load1)			Load2
Add2	Yes	ADD			Add1	Load2
Add3	No					
Mult1	Yes	MULT		(F4)	Load2	
Mult2	Yes	DIV		(Load1)	Mult1	

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			
Busy	Yes	Yes	No	Yes	Yes	Yes	No	...	No

FIGURE 6.37 Reservation stations and register tags. All of the instructions have issued, but only the first load instruction has completed and written its result to the CDB. The instruction-status table is not actually present, but the equivalent information is distributed throughout the hardware. The notation (X), where X is either a register number or a functional unit, indicates that this field contains the result of the functional unit X or the contents of register X at the time of issue. The other instructions are all at reservation stations or, as in the case of instruction 2, completing a memory reference. The load and store buffers are not shown. Load buffer 2 is the only busy load buffer and it is performing on behalf of instruction 2 in the sequence—loading from memory address R3 + 45. There are no stores, so the store buffer is not shown. Remember that an operand is specified by either the Q field or the V field at any time.

The big advantages of the Tomasulo scheme are (1) the distribution of the hazard detection logic, and (2) the elimination of stalls for WAW and WAR hazards. The first advantage arises from the distributed reservation stations and the use of the CDB. If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast on the CDB. In the scoreboard the waiting instructions must all read their results from the registers when register buses are available.

WAW and WAR hazards are eliminated by renaming registers using the reservation stations. For example, in our code sequence in Figure 6.37 we have issued both the DIVF and the ADDF, even though there is a WAR hazard involving F6. The hazard is eliminated in one of two ways. If the instruction providing the value for the DIVF has completed, then Vk will store the result, allowing DIVF to execute independent of the ADDF (this is the case shown). On the other hand, if the LF had not completed, then Qk would point to the Load1 and the DIVF instruction would be independent of the ADDF. Thus, in either case, the ADDF can issue and begin executing. Any uses of the result of the MULTF would point to the reservation station, allowing the ADDF to complete and store its value into the registers without affecting the DIVF. We'll see an example of the elimination of a WAW hazard shortly. But let's first look at how our earlier example continues execution.

Example

Assume the same latencies for the floating-point functional units as we did for Figure 6.34: Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. With the same code segment, show what the status tables look like when the MULTF is ready to go to write result.

Answer

The result is shown in the three tables in Figure 6.38. Unlike the example with the scoreboard, ADDF has completed since the operands of DIVF are copied, thereby overcoming the WAR hazard.

Instruction status				
Instruction		**Issue**	**Execute**	**Write result**
LF	F6,34(R2)	√	√	√
LF	F2,45(R3)	√	√	√
MULTF	F0,F2,F4	√	√	
SUBF	F8,F6,F2	√	√	√
DIVF	F10,F0,F6	√		
ADDF	F6,F8,F2	√	√	√

Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULT	(Load2)	(F4)		
Mult2	Yes	DIV		(Load1)	Mult1	

Register status									
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Mult1					Mult2			
Busy	Yes	No	No	No	No	Yes	No	...	No

FIGURE 6.38 Multiply and divide are the only instructions not finished. This is different from the scoreboard case, because the elimination of WAR hazards allowed the ADDF to finish right after the SUBF on which it depended.

Figure 6.39 gives the steps for each instruction to go through. Load and stores are only slightly special. A load can be executed as soon as it is available. When execution is completed and the CDB is available, a load puts its result on the CDB like any functional unit. Stores receive their values from the CDB or from the register file and execute autonomously; when they are done they turn the busy field off to indicate availability, just like a load buffer or reservation station.

Instruction status	Wait until	Action or bookkeeping

Issue	Station or buffer empty	`if (Register[S1].Qi ≠0)`
		`  {RS[r].Qj← Register[S1].Qi}`
		`else {RS[r].Vj← S1; RS[r].Qj← 0};`
		`if (Register[S2].Qi≠0)`
		`{RS[r].Qk← Register[S2].Qi};`
		`else {RS[r].Vk← S2; RS[r].Qk← 0}`
		`RS[r].Busy←yes;`
		`Register[D].Qi=r;`

Execute	(RS[r].Qj=0) and (RS[r].Qk=0)	None—operands are in Vj and Vk

Write result	Execution completed at *r* and CDB available	`∀x(if (Register[x].Qi=r)  {Fx← result;`
		`  Register[x].Qi← 0});`
		`∀x(if (RS[x].Qj=r) {RS[x].Vj← result;`
		`  RS[x].Qj ← 0});`
		`∀x(if (RS[x].Qk=r) {RS[x].Vk← result;`
		`  RS[x].Qk ← 0});`
		`∀x(if (Store[x].Qi=r) {Store[x].V← result;`
		`  Store[x].Qi ← 0});`
		`RS[r].Busy←No`

FIGURE 6.39 Steps in the algorithm and what is required for each step. For the issuing instruction, D is the destination, S1 and S2 are the sources, and *r* is the reservation station or buffer that D is assigned to. RS is the reservation-station data structure. The value returned by a reservation station or by the load unit is called the "result." Register is the register data structure, while Store is the store-buffer data structure. When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose value of Qj or Qk is the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus, the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. For simplicity we assume that all bookkeeping actions are done in a single cycle.

To understand the full power of eliminating WAW and WAR hazards through dynamic renaming of registers, we must look at a loop. Consider the following simple sequence for multiplying the elements of a vector by a scalar in F2:

```
Loop:  LD     F0,0(R1)
       MULTD  F4,F0,F2
       SD     0(R1),F4
       SUB    R1,R1,#8
       BNEZ   R1,Loop   ; branches if R1≠0
```

With a branch-taken strategy, using reservation stations will allow multiple executions of this loop to proceed at once. This advantage is gained without unrolling the loop—in effect, the loop is unrolled dynamically by the hardware. In

the 360 architecture, the presence of only 4 FP registers would severely limit the use of unrolling. (We will see shortly, when we unroll a loop and schedule it to avoid interlocks, many more registers are required.) Tomasulo's algorithm supports the overlapped execution of multiple copies of the same loop with only a small number of registers used by the program.

Let's assume we have issued all the instructions in two successive iterations of the loop, but none of the floating-point loads/stores or operations has completed. The reservation stations, register-status tables, and load and store buffers at this point are shown in Figure 6.40. (The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.) Once the system reaches this state, two copies of the loop could be sustained with a CPI close to one provided the multiplies could complete in four clock cycles. We will see how compiler techniques can achieve a similar result in Section 6.8.

An additional element that is critical to making Tomasulo's algorithm work is shown in this example. The load instruction from the second loop iteration could easily complete before the store from the first iteration, although the normal sequential order is different. The load and store can safely be done in a different order, provided the load and store access different addresses. This is checked by examining the addresses in the store buffer whenever a load is issued. If the load address matches the store-buffer address, we must stop and wait until the store buffer gets a value; we can then access it or get the value from memory.

This scheme can yield very high performance, provided the cost of branches can be kept small—this is a problem we will look at later in this section. There are also limitations imposed by the complexity of the Tomasulo scheme, which requires a large amount of hardware. In particular, there are many associative stores that must run at high speed, as well as complex control logic. Lastly, the performance gain is limited by the single completion bus (CDB). While additional CDBs can be added, each CDB must interact with all the pipeline hardware, including the reservation stations. In particular, the associative tag-matching hardware would need to be duplicated at all stations for each CDB.

While Tomasulo's scheme may be appealing if the designer is forced to pipeline an architecture that is difficult to schedule code for or has a shortage of registers, the authors believe that the advantages of the Tomasulo approach are limited for architectures that can be efficiently pipelined and statically scheduled with software. However, as available gate counts grow and the limits of software scheduling are reached, we may see dynamic scheduling employed. One possible direction is a hybrid organization that uses dynamic scheduling for loads and stores, while statically scheduling register–register operations.

Reducing Branch Penalties with Dynamic Hardware Prediction

The previous section describes techniques for overcoming data hazards. If control hazards are not addressed, Amdahl's Law predicts, they will limit pipelined-execution performance. Earlier, we looked at simple hardware schemes for

Instruction status					
Instruction		**From iteration**	**Issue**	**Execute**	**Write result**
LD	F0,0(R1)	1	√	√	
MULTD	F4,F0,F2	1	√		
SD	0(R1),F4	1	√		
LD	F0,0(R1)	2	√	√	
MULTD	F4,F0,F2	2	√		
SD	0(R1),F4	2	√		

Reservation stations						
Name	**Busy**	**Fm**	**Vj**	**Vk**	**Qj**	**Qk**
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULT		(F2)	Load1	
Mult2	Yes	MULT		(F2)	Load2	

Register status									
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	**...**	**F30**
Qi	Load2		Mult2						
Busy	yes	no	yes	no	no	no			

Store buffers			
Field	**Store 1**	**Store 2**	**Store 3**
Qi	Mult1	Mult2	
Busy	Yes	Yes	No
Address	(R1)	(R1)–8	

Load buffers			
Field	**Load 1**	**Load 2**	**Load 3**
Address	(R1)	(R1)–8	
Busy	Yes	Yes	No

FIGURE 6.40 Two active iterations of the loop with no instruction having yet completed. Load and store buffers are included, with addresses to be loaded from and stored to. The loads are in the load buffer; entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store buffers indicate that the multiply destination is their value to store.

dealing with branches (assume taken or not taken) and software-oriented approaches (delayed branches). This section focuses on using hardware to dynamically predict the outcome of a branch—the prediction will change if the branch changes its behavior while the program is running.

The simplest dynamic branch-prediction scheme is a *branch-prediction buffer*. A branch-prediction buffer is a small memory indexed by the lower por-

tion of the branch instruction address. The memory contains a bit that says whether the branch was recently taken or not. This is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs. We don't know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. But this doesn't matter. It is assumed to be correct, and fetching begins in the predicted direction. If the branch prediction turns out to be wrong, the prediction bit is inverted.

This simple one-bit prediction scheme has a performance shortcoming: If a branch is almost always taken, then when it is not taken, we will predict incorrectly twice, rather than once. Consider a loop branch whose behavior is taken nine times sequentially, then not taken once. If the next time around it is predicted not taken, the prediction will be wrong. Thus, the prediction accuracy will only be 80%, even on branches that are 90% taken. To remedy this, two-bit prediction schemes are often used. In a two-bit scheme, a prediction must miss twice in a row before it is changed. Figure 6.41 shows the finite-state machine for the two-bit prediction scheme.

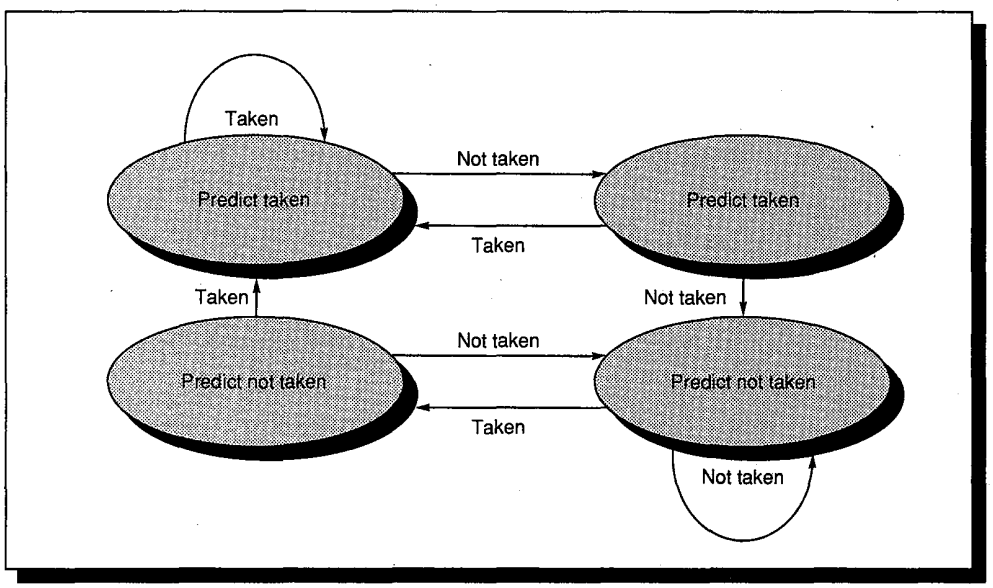

FIGURE 6.41 This shows the states in a two-bit prediction scheme. By using two bits rather than one, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The two bits are used to encode the four states in the system.

The branch-prediction buffer can be implemented as a small, special cache accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction (see Section 8.3 in Chapter 8). If the instruction is predicted as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the

PC is known. Otherwise, fetching and sequential executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 6.41. While this scheme is useful for most pipelines, the DLX pipeline finds out both whether the branch is taken and what the target of the branch is at the same time. Thus, this scheme does not help for the simple DLX pipeline; we will explore a scheme that can work for DLX a little later. First, let's see how well a prediction buffer works with a longer pipeline.

The accuracy of a two-bit prediction scheme is affected by how often the prediction for each branch is correct and by how often the entry in the prediction buffer matches the branch being executed. When the entry does not match, the prediction bit is used anyway because no better information is available. Even if the entry was for another branch, the guess could be a lucky one. In fact, there is about a 50% probability of being correct, even if the prediction is for some other branch. Studies of branch-prediction schemes have found that two-bit prediction has an accuracy of about 90% when the entry in the buffer is the branch entry. A buffer of between 500 and 1000 entries has a hit rate of 90%. The overall prediction accuracy is given by

Accuracy = (% predicted correctly * % that prediction is for this instruction) +

(% lucky guess) * (1–% that prediction is for this instruction)

Accuracy = (90% * 90%) + (50% * 10%) = 86%

This number is higher than our success rate for filling delayed branches and would be useful in a pipeline with a longer branch delay. Now let's look at a dynamic prediction scheme that is useable for DLX and see how it compares to our branch-delay scheme.

To reduce the branch penalty on DLX, we need to know from what address to fetch by the end of IF. This means we must know whether the as yet undecoded instruction is a branch and, if it is a branch, what the next PC should be. If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero. A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a *branch-target buffer*. Because we are predicting the next instruction address and will send it out **before** decoding the instruction, we **must** know whether the fetched instruction is predicted as a taken branch. We also want to know whether the address in the target buffer is for a taken or not-taken prediction, so that we can reduce the time to find a mispredicted branch. Figure 6.42 shows what the branch-target buffer looks like. If the PC of the fetched instruction matches a PC in the buffer, then the corresponding predicted PC is used as the next PC. In Chapter 8 we will discuss caches in much more detail; we will see that the hardware for this branch-target buffer is similar to the hardware for a cache.

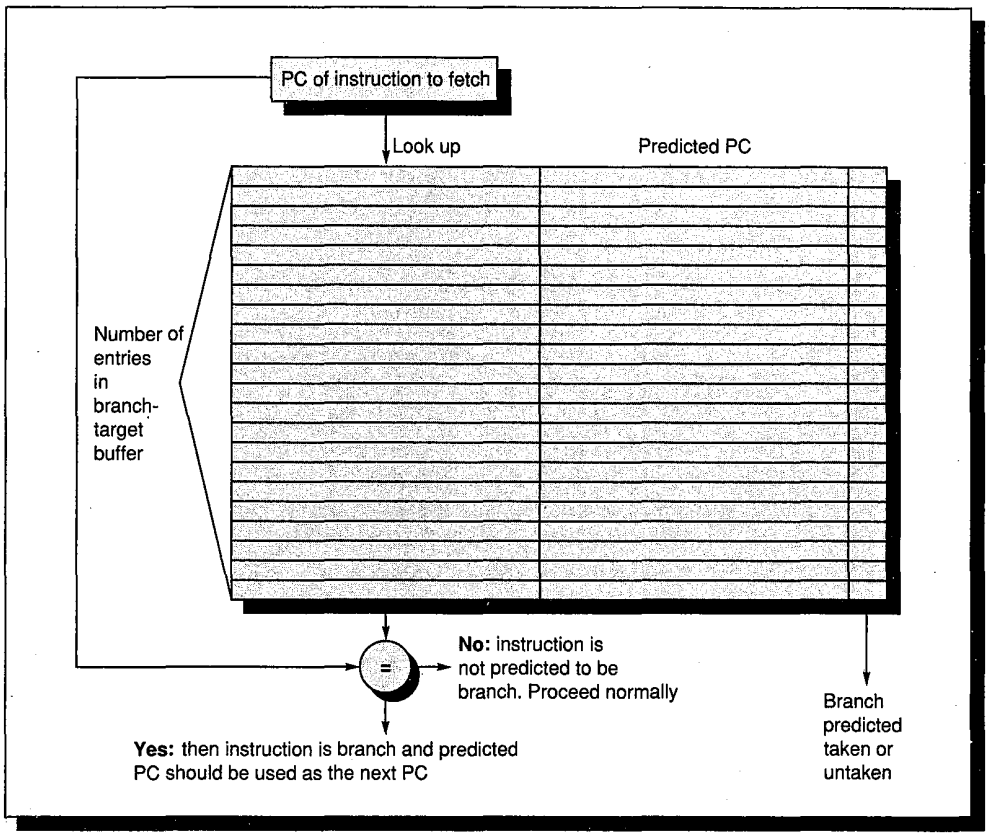

FIGURE 6.42 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a branch. If it is a branch, then the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field just tracks whether the branch was predicted taken or untaken and helps keep the misprediction penalty small.

If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that (unlike a branch-prediction buffer) the entry must be for this instruction, because the predicted PC will be sent out before it is known whether this instruction is even a branch. If we did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower machine. Figure 6.43 shows the steps followed when using a branch-target buffer and when these steps occur in the pipeline. From this we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and is correct. Otherwise, there will be a penalty of at least one clock cycle. In practice, there could be a penalty of two clock cycles because the branch-target buffer must be updated. We could assume that the instruction following a branch or at the branch target is not a branch, and do the update during that instruction time. However, this does complicate the control. Instead, we will take a two-clock-cycle penalty when the branch is not correctly predicted.

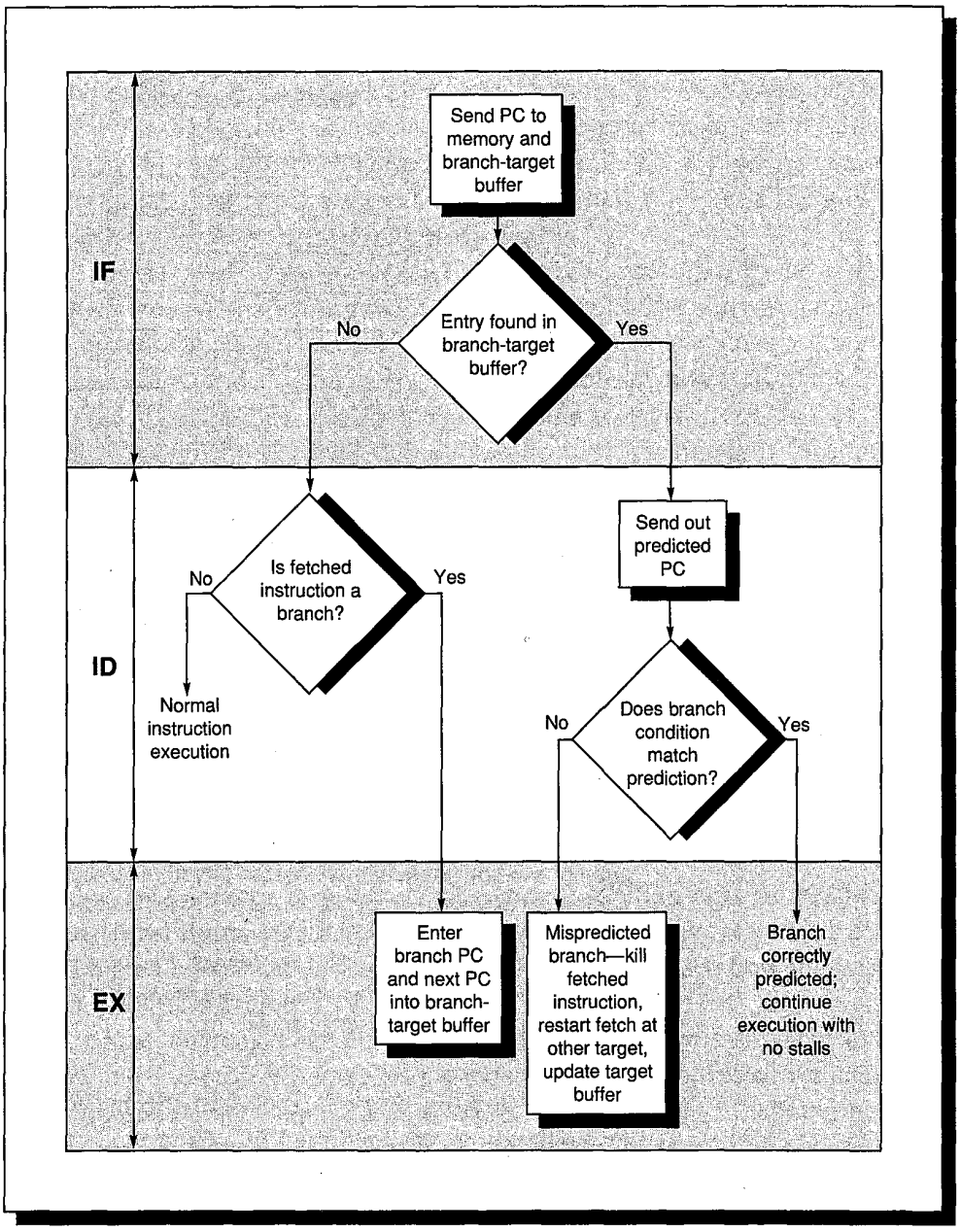

FIGURE 6.43 The steps involved in handling an instruction with a branch-target buffer. If the PC of an instruction is found in the buffer, then the instruction must be a branch, and fetching immediately begins from the predicted PC in ID. If the entry is not found and it subsequently turns out to be a branch, it is entered in the buffer along with the target, which is known at the end of ID. If the instruction is a branch, is found, and is correctly predicted, then execution proceeds with no delays. If the prediction is incorrect, we suffer a one-clock-cycle delay fetching the wrong instruction and restart the fetch one clock cycle later. If the branch is not found in the buffer and the instruction turns out to be a branch, we will have proceeded as if the instruction were a branch and can turn this into an assume-not-taken strategy; the penalty will differ depending on whether the branch is actually taken or not.

To evaluate how well a branch-target buffer works, we first must determine what the penalties are in all possible cases. Figure 6.44 contains this information.

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
Yes	Not taken	Not taken	0
Yes	Not taken	Taken	2
No		Taken	2
No		Not taken	1

FIGURE 6.44 Penalties for all possible combinations of whether the branch is in the buffer, how it is predicted, and what it actually does. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to one clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and one clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and not taken, the penalty is only one clock cycle because the pipeline assumes not taken when it is not aware that the instruction is a branch. Other mismatches cost two clock cycles, since we must restart the fetch and update the buffer.

Using the same probabilities as for a branch-prediction buffer—90% probability of finding the entry and 90% probability of correct prediction—and the taken/not taken percentage taken from earlier in this chapter, we can find the total branch penalty:

$$\begin{aligned}\text{Branch penalty} = {} & \%\text{ branches found in buffer} * \%\text{ incorrect predictions} * 2 + \\ & (1-\%\text{ branches found in buffer}) * \%\text{ taken branches} * 2 + \\ & (1-\%\text{ branches found in buffer}) * \%\text{ untaken branches} * 1\end{aligned}$$

$$\text{Branch penalty} = 90\% * 10\% * 2 + 10\% * 60\% * 2 + 10\% * 40\% * 1$$

$$\text{Branch penalty} = 0.34 \text{ clock cycles}$$

This compares with a branch penalty for delayed branches of about 0.5 clock cycles per branch. Remember, though, that the improvement from dynamic branch prediction will grow as the branch delay grows.

Branch-prediction schemes are limited both by prediction accuracy and by the penalty for misprediction. It is unlikely that we can improve the effective branch-prediction success much above 80% to 90%. Instead, we can try to reduce the penalty for misprediction. This is done by fetching from both the predicted and unpredicted direction. This requires that the memory system be dual ported or have an interleaved cache. While this adds cost to the system, it may be the only way to reduce branch penalties below a certain point.

We have seen a variety of software-based static schemes and hardware-based dynamic schemes for trying to boost the performance of our pipelined machine. Pipelining tries to exploit the potential for parallelism among sequential instructions. In the ideal case all the instructions would be independent, and our DLX pipeline would exploit parallelism among the five instructions simultaneously in the pipeline. Both the static scheduling techniques of the last section and the dynamic techniques of this section focus on maintaining the throughput of the pipeline at one instruction per clock. In the next section we will look at techniques that attempt to exploit overlap more than by the factor of 5, to which we are restricted with the simple DLX pipeline.

6.8 Advanced Pipelining—Taking Advantage of More Instruction-Level Parallelism

To improve performance further we would like to decrease the CPI to less than one. But the CPI cannot be reduced below one if we issue only one instruction every clock cycle. The goal of the techniques discussed in this section is to allow multiple instructions to issue in a clock cycle.

As we know from earlier sections, to keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. Two related instructions must be separated by a distance equal to the pipeline latency of the first of the instructions. Throughout this section we will assume the latencies shown in Figure 6.45. Branches still have a one-clock-cycle delay. We assume that the functional units are fully pipelined or replicated, and that an operation can be issued on every clock cycle.

As we try to execute more instructions on every clock cycle and try to overlap more instructions, we will need to find and exploit more instruction-level parallelism. Thus, before looking at pipeline organizations that require more parallelism among instructions, let's look at a simple compiler technique that will help create additional parallelism.

Instruction producing result	Destination instruction	Latency in clocks
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

FIGURE 6.45 Latencies of operations used in this section. The first column shows the originating instruction type. The second column is the type of the consuming instruction. The last column is the separation in clock cycles to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit, like the one we described for DLX in Figure 6.29 (page 289).

Increasing Instruction-Level Parallelism with Loop Unrolling

To compare the approaches discussed in this section, we will use a simple loop that adds a scalar value to a vector in memory. The DLX code, not accounting for the pipeline, looks like this:

```
Loop: LD     F0,0(R1)     ; load the vector element
      ADDD   F4,F0,F2     ; add the scalar in F2
      SD     0(R1),F4     ; store the vector element
      SUB    R1,R1,#8     ; decrement the pointer by
                          ; 8 bytes (per DW)
      BNEZ   R1,LOOP      ; branch when it's zero
```

For simplicity, we assume the array starts at location 0. If it were located elsewhere, the loop would require one additional integer instruction.

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for DLX with the latencies discussed above.

Example

Show how the vector add loop would look on DLX, both scheduled and unscheduled, including any stalls or idle clock cycles.

Answer

Without any scheduling the loop will execute as follows:

```
                                  Clock cycle issued
Loop: LD     F0,0(R1)                    1
      stall                              2
      ADDD   F4,F0,F2                    3
      stall                              4
      stall                              5
      SD     0(R1),F4                    6
      SUB    R1,R1,#8                    7
      BNEZ   R1,LOOP                     8
      stall                              9
```

This requires 9 clock cycles per iteration. We can schedule the loop to obtain

```
Loop: LD     F0,0(R1)
      stall
      ADDD   F4,F0,F2
      SUB    R1,R1,#8
      BNEZ   R1,LOOP      ; delayed branch
      SD     8(R1),F4     ; changed because interchanged with SUB
```

Execution time has been reduced from 9 clock cycles to 6.

Notice that to create this schedule, the compiler had to determine that it could swap the SUB and SD by changing the address the SD stored to: The address was 0(R1) and is now 8(R1). This is not trivial, since most compilers would see that the SD instruction depends on the SUB and would refuse to interchange them. A smarter compiler could figure out the relationship and perform the interchange. The dependence among the LD, ADDD, and SD determines the clock cycle count for this loop.

In the above example, we complete one loop iteration and finish one vector element every 6 clock cycles, but the actual work of operating on the vector element takes just 3 of those 6 clock cycles. The remaining 3 clock cycles consist of loop overhead—the SUB and BNEZ—and a stall. To eliminate these 3 clock cycles we need to get more operations within the loop. A simple scheme for increasing the number of instructions between executions of the loop branch is *loop unrolling*. This is done by simply replicating the loop body multiple times, adjusting the loop termination code, and then scheduling the unrolled loop. To allow effective scheduling of the loop, we will want to use different registers for each iteration, thus increasing the register count.

Example

Show what our loop looks like unrolled three times (yielding four copies of the loop body), assuming R1 is initially a multiple of 4. Eliminate any obviously redundant computations, and do not reuse any of the registers.

Answer

Here is the result after dropping the unnecessary SUB and BNEZ operations duplicated during unrolling.

```
Loop: LD     F0,0(R1)
      ADDD   F4,F0,F2
      SD     0(R1),F4 ;drop SUB & BNEZ
      LD     F6,-8(R1)
      ADDD   F8,F6,F2
      SD     -8(R1),F8 ;drop SUB & BNEZ
      LD     F10,-16(R1)
      ADDD   F12,F10,F2
      SD     -16(R1),F12 ;drop SUB & BNEZ
      LD     F14,-24(R1)
      ADDD   F16,F14,F2
      SD     -24(R1),F16
      SUB    R1,R1,#32
      BNEZ   R1,LOOP
```

We have eliminated three branches and three decrements of R1. The addresses on the loads and stores have been compensated for. Without scheduling, every operation is followed by a dependent operation, and thus will cause a stall. This loop will run in 27 clock cycles—each LD takes 2 clock cycles, each ADDD 3, the branch 2, and all other instructions 1—or 6.8 clock cycles for each of the four elements.

Although this unrolled version is currently slower than the scheduled version of the original loop, this will change when we schedule the unrolled loop. Loop unrolling is normally done early in the compilation process, so that redundant computations can be exposed and eliminated by the optimizer.

In real programs we do not normally know the upper bound on the loop. Suppose it is n, and we would like to unroll the loop k times. Instead of a single unrolled loop, we generate a pair of loops. The first executes (n mod k) times and has a body that is the original loop. The unrolled version of the loop is surrounded by an outer loop that iterates (n div k) times. In the above example, unrolling improves the performance of this loop by eliminating overhead instructions, though it increases code size substantially. What will happen to the performance increase when the loop is scheduled on DLX?

Example

Show the unrolled loop in the previous example after it has been scheduled on DLX.

Answer

```
Loop: LD     F0,0(R1)
      LD     F6,-8(R1)
      LD     F10,-16(R1)
      LD     F14,-24(R1)
      ADDD   F4,F0,F2
      ADDD   F8,F6,F2
      ADDD   F12,F10,F2
      ADDD   F16,F14,F2
      SD     0(R1),F4
      SD     -8(R1),F8
      SD     -16(R1),F12
      SUB    R1,R1,#32   ;branch dependence
      BNEZ   R1,LOOP
      SD     -24(R1),F16 ; 8-32 = -24
```

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared to 6.8 per element before scheduling.

The gain from scheduling on the unrolled loop is even larger than on the original loop. This is because unrolling the loop exposes more computation that can be scheduled. Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Loop unrolling is a simple but useful method for increasing the size of straightline code fragments that can be scheduled effectively. This compile-time transformation is similar to what Tomasulo's algorithm does with register renaming and out-of-order execution. As we will see, this is very important in attempts to lower the CPI by issuing instructions at a high rate.

A Superscalar Version of DLX

One method of decreasing the CPI of DLX is to **issue** more than one instruction per clock cycle. This would allow the instruction-execution rate to exceed the clock rate. Machines that issue multiple independent instructions per clock cycle when they are properly scheduled by the compiler have been called *superscalar machines*. In a superscalar machine, the hardware can issue a small number (say 2 to 4) of independent instructions in a single clock. However, if the instructions in the instruction stream are dependent or don't meet certain criteria, only the first instruction in sequence will be issued. A machine where the compiler has complete responsibility for creating a package of instructions that can be simultaneously issued, and the hardware does not dynamically make any decisions about multiple issue, should probably be regarded as a type of VLIW (very long instruction word), which we discuss in the next section.

What would the DLX machine look like as a superscalar? Let's assume two instructions issued per clock cycle. One of the instructions could be a load, store, branch, or integer ALU operation, and the other could be any floating-point operation. As we will see, issue of an integer operation in parallel with a floating-point operation is much simpler and less demanding than arbitrary dual issue.

Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. To keep the decoding simple, we could require that the instructions be paired and aligned on a 64-bit boundary, with the integer portion appearing first. Figure 6.46 shows how the instructions look as they go into the pipeline in pairs. This table does not address how the floating-point operations extend the EX cycle, but it is no different in the superscalar case than it was for the ordinary DLX pipeline; the concepts of Section 6.6 apply directly. With this pipeline, we have substantially boosted the rate at which we can issue floating-point instructions. To make this worthwhile, however, we need either pipelined floating-point units or multiple independent units. Otherwise, floating-point instructions can only be fetched, and not issued, since all the floating units will be busy.