Appendix C
Pipelining: Basic and Intermediate Concepts

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 C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

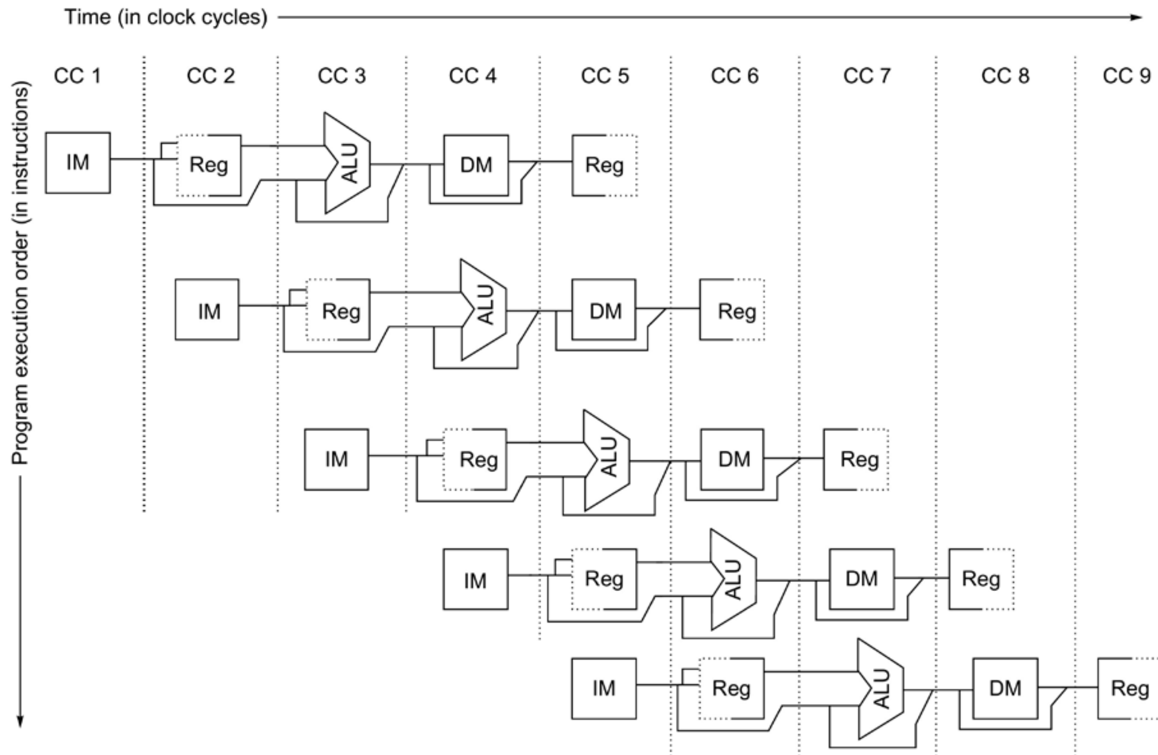

Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This figure shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

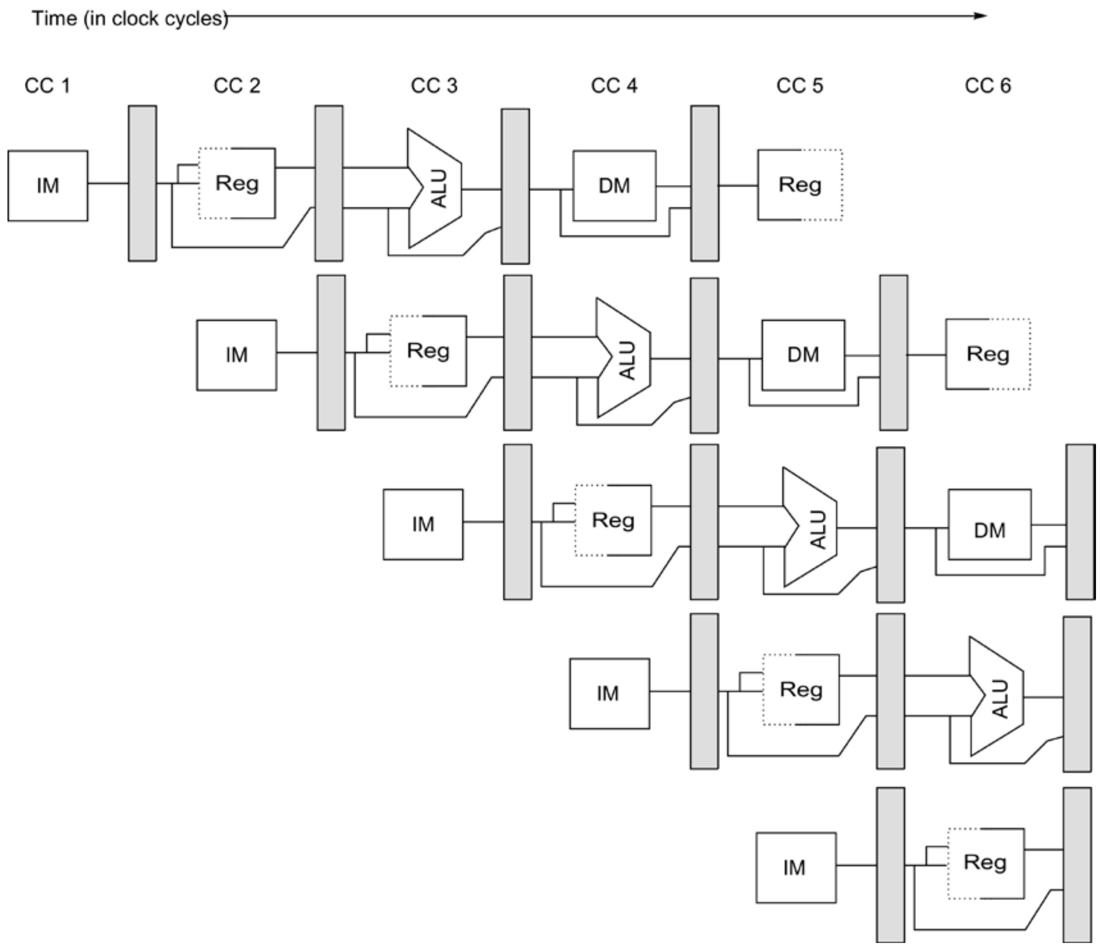

Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

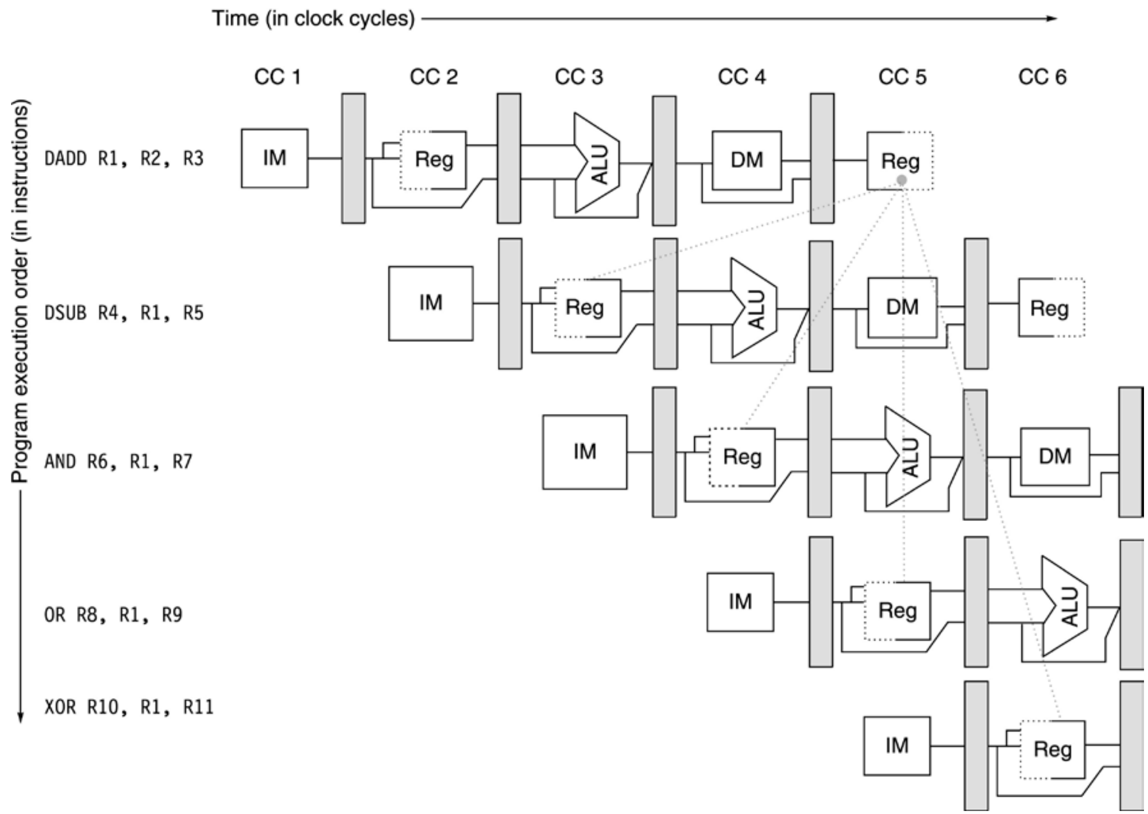

Figure C.4 The use of the result of the add instruction in the next three instructions causes a hazard, because the register is not written until after those instructions read it.

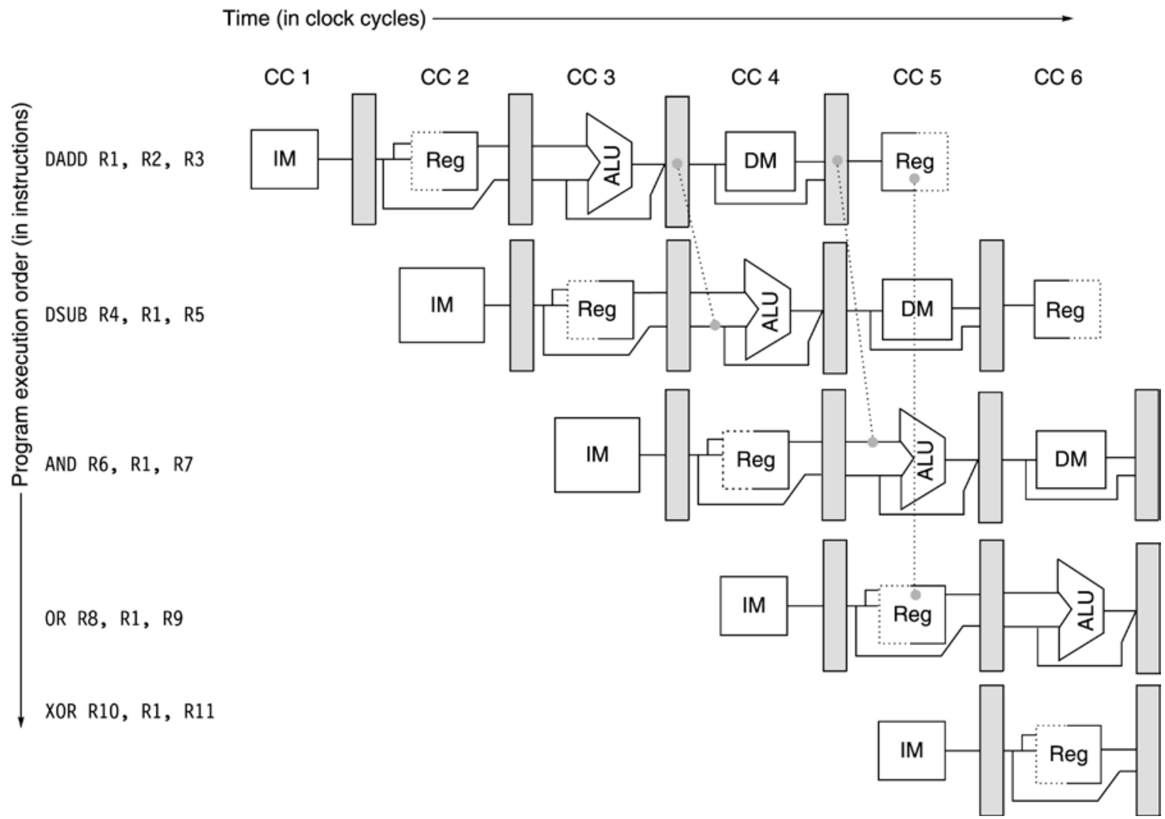

Figure C.5 A set of instructions that depends on the add result uses forwarding paths to avoid the data hazard. The inputs for the `sub` and `and` instructions forward from the pipeline registers to the first ALU input. The `or` receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the `and` instruction was `and x6,x1,x4`.

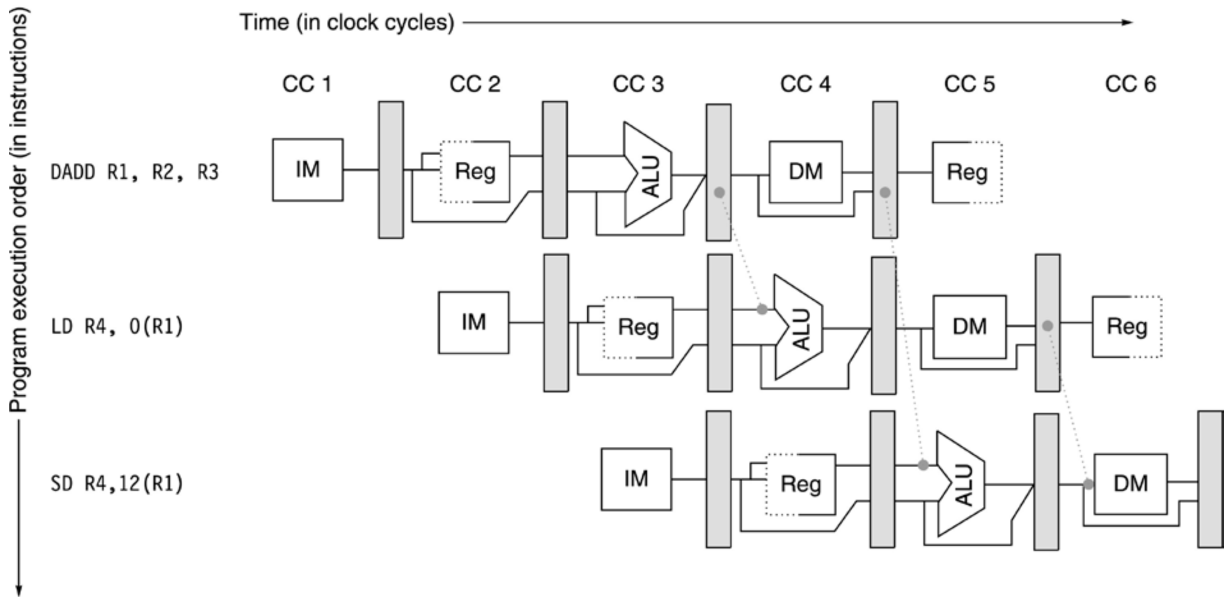

Figure C.6 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown herein), the result would need to be forwarded to prevent a stall.

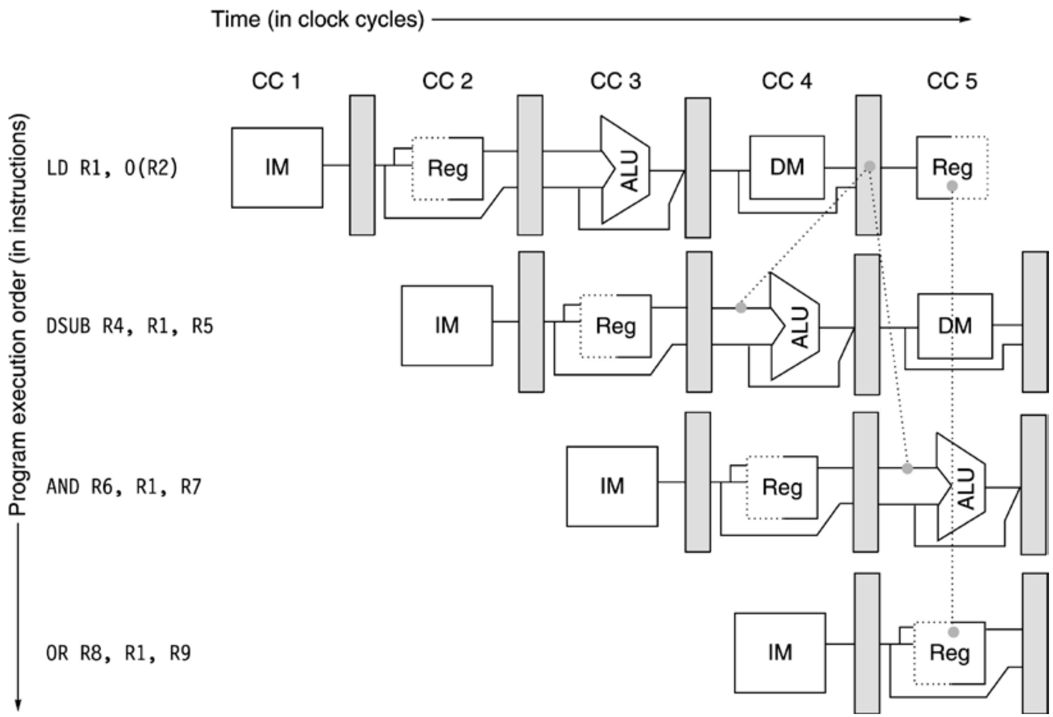

Figure C.7 The load instruction can bypass its results to the and and or instructions, but not to the sub, because that would mean forwarding the result in "negative time."

`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	EX	MEM	WB			
`and x6,x1,x7`			IF	ID	EX	MEM	WB		
`or x8,x1,x9`				IF	ID	EX	MEM	WB	
`ld x1,0(x2)`	IF	ID	EX	MEM	WB				
`sub x4,x1,x5`		IF	ID	Stall	EX	MEM	WB		
`and x6,x1,x7`			IF	Stall	ID	EX	MEM	WB	
`or x8,x1,x9`				Stall	IF	ID	EX	MEM	WB

Figure C.8 In the top half, we can see why a stall is needed: the MEM cycle of the load produces a value that is needed in the EX cycle of the sub, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Figure C.9 A branch causes a one-cycle stall in the five-stage pipeline. The instruction after the branch 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 branch successor is redundant. This will be addressed shortly.

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	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.10 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch 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 1 clock cycle.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay 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				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.11 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there was only one delay slot for most RISC architectures that incorporated them) are executed. If the branch is untaken, execution continues with the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: if the branch is not taken, what should happen to the branch in the branch delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Figure C.12 Branch penalties for the three simplest prediction schemes for a deeper pipeline.

Branch scheme	Additions to the CPI from branch costs			
	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Figure C.13 CPI penalties for three branch-prediction schemes and a deeper pipeline.

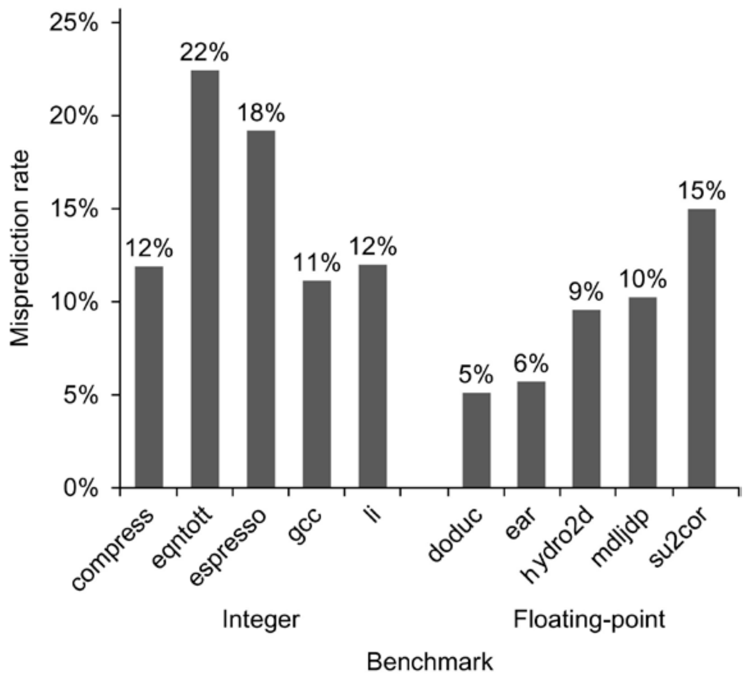

Figure C.14 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

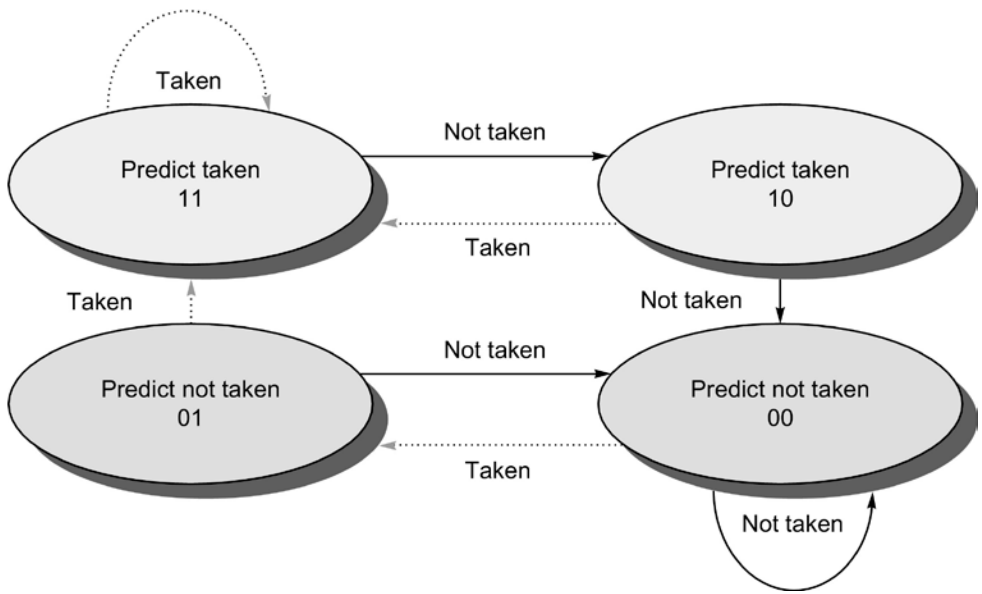

Figure C.15 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an n-bit saturating counter for each entry in the prediction buffer. With an n-bit counter, the counter can take on values between 0 and $2^n - 1$: when the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of n-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general n-bit predictors.

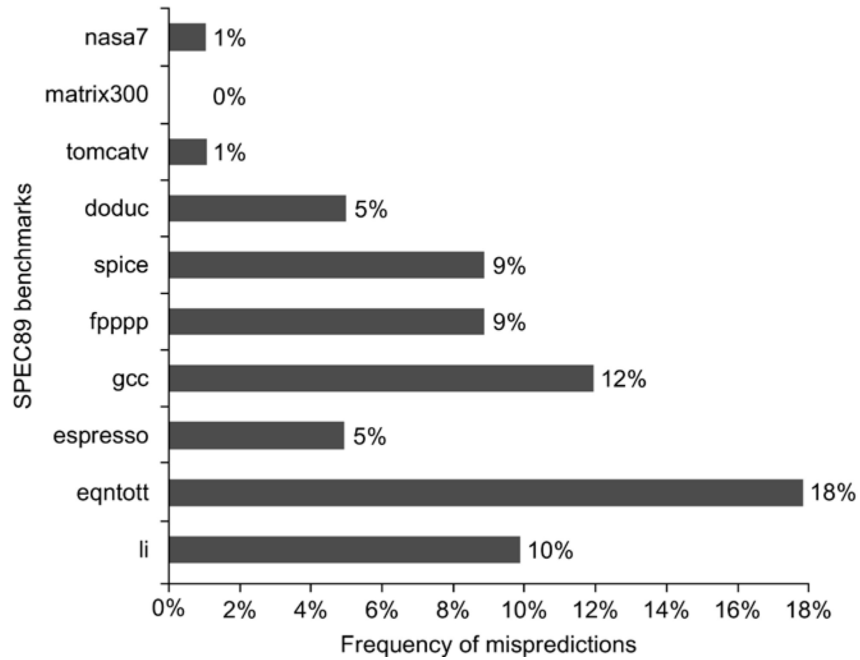

Figure C.16 Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the floating-point programs (average of 4%). Omitting the floating-point kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan et al. (1992). Although these data are for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

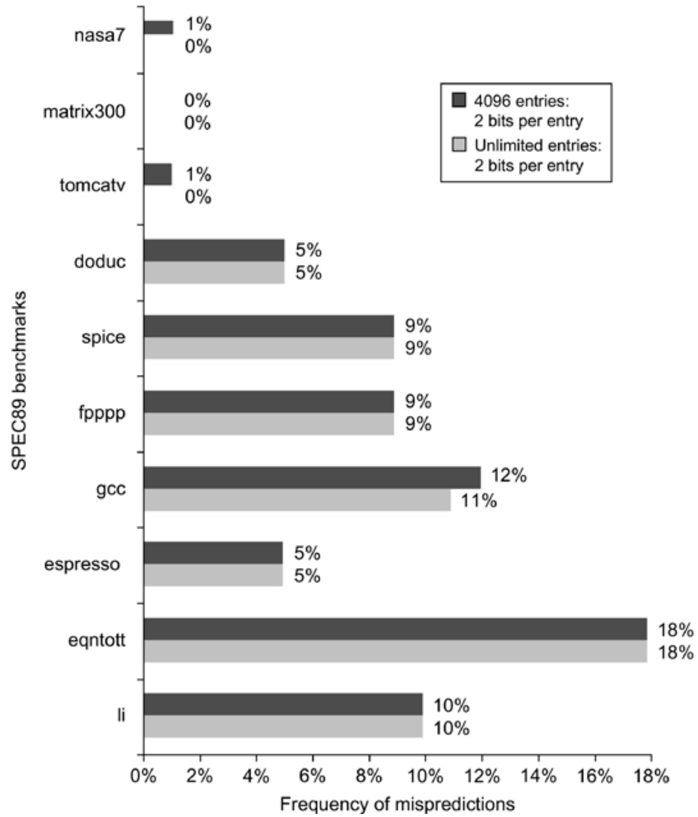

Figure C.17 Prediction accuracy of a 4096-entry 2-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks. Although these data are for an older version of a subset of the SPEC benchmarks, the results would be comparable for newer versions with perhaps as many as 8K entries needed to match an infinite 2-bit predictor.

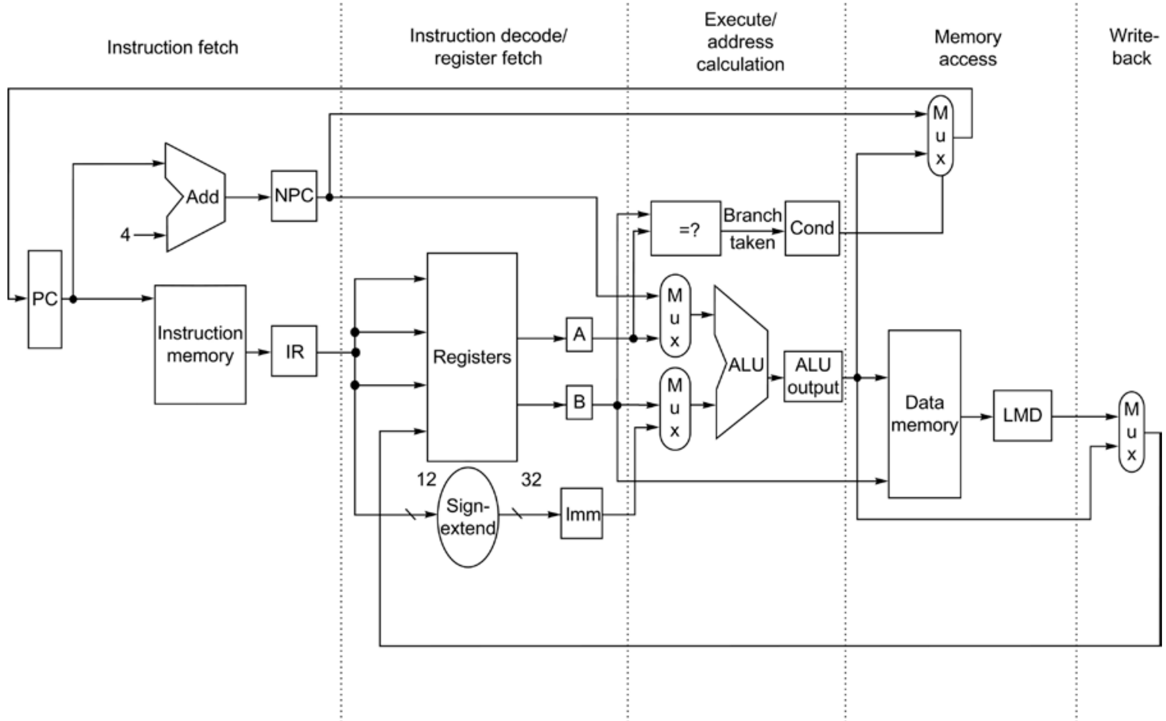

Figure C.18 The implementation of the RISC V data path allows every instruction to be executed in 4 or 5 clock cycles. Although the PC is shown in the portion of the data path that is used in instruction fetch and the registers are shown in the portion of the data path that is used in instruction decode/register fetch, both of these functional units are read as well as written by an instruction. Although we show these functional units in the cycle corresponding to where they are read, the PC is written during the memory access clock cycle and the registers are written during the write-back clock cycle. In both cases, the writes in later pipe stages are indicated by the multiplexer output (in memory access or write-back), which carries a value back to the PC or registers. These backward-flowing signals introduce much of the complexity of pipelining, because they indicate the possibility of hazards.

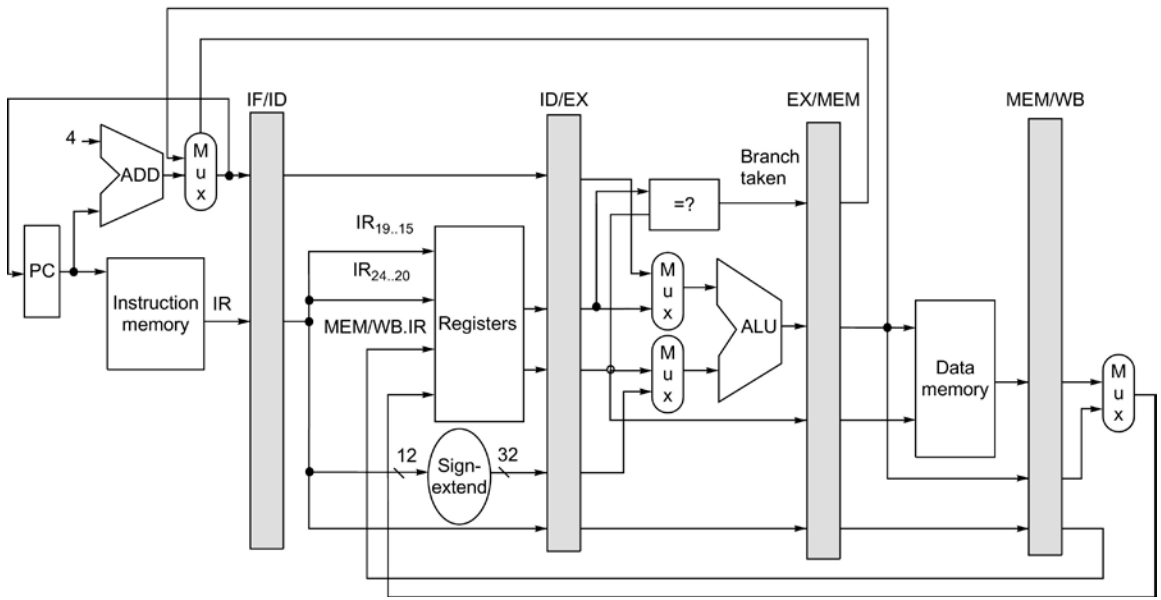

Figure C.19 The data path is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading to one pipeline register for each pipe stage. Recall that the PC is an edge-triggered register written at the end of the clock cycle; hence, there is no race condition in writing the PC. The selection multiplexer for the PC has been moved so that the PC is written in exactly one stage (IF). If we didn't move it, there would be a conflict when a branch occurred, because two instructions would try to write different values into the PC. Most of the data paths flow from left to right, which is from earlier in time to later. The paths flowing from right to left (which carry the register write-back information and PC information on a branch) introduce complications into our pipeline.

Stage	Any instruction		
IF	IF/ID.IR←Mem[PC] IF/ID.NPC,PC←(if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM. ALUOutput} else {PC+4});		
ID	ID/EX.A←Regs[IF/ID.IR[rs1]]; ID/EX.B←Regs[IF/ID.IR[rs2]]; ID/EX.NPC←IF/ID.NPC; ID/EX.IR←IF/ID.IR; ID/EX.Imm←sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load instruction**	**Branch instruction**
EX	EX/MEM.IR←ID/EX.IR; EX/MEM.ALUOutput← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput← ID/EX.A+ID/EX.Imm; EX/MEM.B←ID/EX.B;	EX/MEM.ALUOutput← ID/EX.NPC + (ID/EX.Imm<< 2); EX/MEM.cond← (ID/EX.A == ID/EX.B);
MEM	MEM/WB.IR←EX/MEM.IR; MEM/WB.ALUOutput← EX/MEM.ALUOutput;	MEM/WB.IR←EX/MEM.IR; MEM/WB.LMD← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput]← EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]]← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rd]]← MEM/WB.LMD;	

Figure C.20 Events on every pipe stage of the RISC V pipeline. Let's review the actions in the stages that are specific to the pipeline organization. In IF, in addition to fetching the instruction and computing the new PC, we store the incremented PC both into the PC and into a pipeline register (NPC) for later use in computing the branch-target address. This structure is the same as the organization in Figure C.19, where the PC is updated in IF from one of two sources. In ID, we fetch the registers, extend the sign of the 12 bits of the IR (the immediate field), and pass along the IR and NPC. During EX, we perform an ALU operation or an address calculation; we pass along the IR and the B register (if the instruction is a store). We also set the value of cond to 1 if the instruction is a taken branch. During the MEM phase, we cycle the memory, write the PC if needed, and pass along values needed in the final pipe stage. Finally, during WB, we update the register field from either the ALU output or the loaded value. For simplicity we always pass the entire IR from one stage to the next, although as an instruction proceeds down the pipeline, less and less of the IR is needed.

Situation	Example code sequence	Action

No dependence	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		No hazard possible because no dependence exists on x1 in the immediately following three instructions

Dependence requiring stall	`ld x1,45(x2)`
	`add x5,x1,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		Comparators detect the use of x1 in the `add` and stall the `add` (and `sub` and `or`) before the `add` begins EX

Dependence overcome by forwarding	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x1,x7`
	`or x9,x6,x7`
		Comparators detect use of x1 in `sub` and forward result of load to ALU in time for `sub` to begin EX

Dependence with accesses in order	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x1,x7`
		No action required because the read of x1 by `or` occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half

Figure C.21 Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only comparison 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 (dependence overcome by forwarding). Of course, hazards that involve x0 can be ignored because the register always contains 0, and the preceding test could be extended to do this.

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..6}$)	Matching operand fields
Load	Register-register ALU, load, store, ALU immediate, or branch	`ID/EX.IR[rd] == IF/ID.IR[rs1]`
Load	Register-register ALU, or branch	`ID/EX.IR[rd] == IF/ID.IR[rs2]`

Figure C.22 The logic to detect the need for load interlocks during the ID stage of an instruction requires two comparisons, one for each possible source. Remember that the IF/ID register holds the state of the instruction in ID, which potentially uses the load result, while ID/EX holds the state of the instruction in EX, which is the load instruction.

Pipeline register of source instruction	Opcode of source instruction	Pipeline register of destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs1]`
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs2]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs1]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU	Bottom ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs2]`

Figure C.23 Forwarding of data to the two ALU inputs (for the instruction in EX) can occur from the ALU result (in EX/MEM or in MEM/WB) or from the load result in MEM/WB. There are 10 separate comparisons needed to tell whether a forwarding operation should occur. The top and bottom ALU inputs refer to the inputs corresponding to the first and second ALU source operands, respectively, and are shown explicitly in Figure C.18 on page C.30 and in Figure C.24 on page C.36. Remember that the pipeline latch for destination instruction in EX is ID/EX, while the source values come from the ALUOutput portion of EX/MEM or MEM/WB or the LMD portion of MEM/WB. There is one complication not addressed by this logic: dealing with multiple instructions that write the same register. For example, during the code sequence `add x1, x2, x3; addi x1, x1, 2; sub x4, x3, x1,` the logic must ensure that the `sub` instruction uses the result of the `addi` instruction rather than the result of the `add` instruction. The logic shown here can be extended to handle this case by simply testing that forwarding from MEM/WB is enabled only when forwarding from EX/MEM is not enabled for the same input. Because the `addi` result will be in EX/MEM, it will be forwarded, rather than the `add` result in MEM/WB.

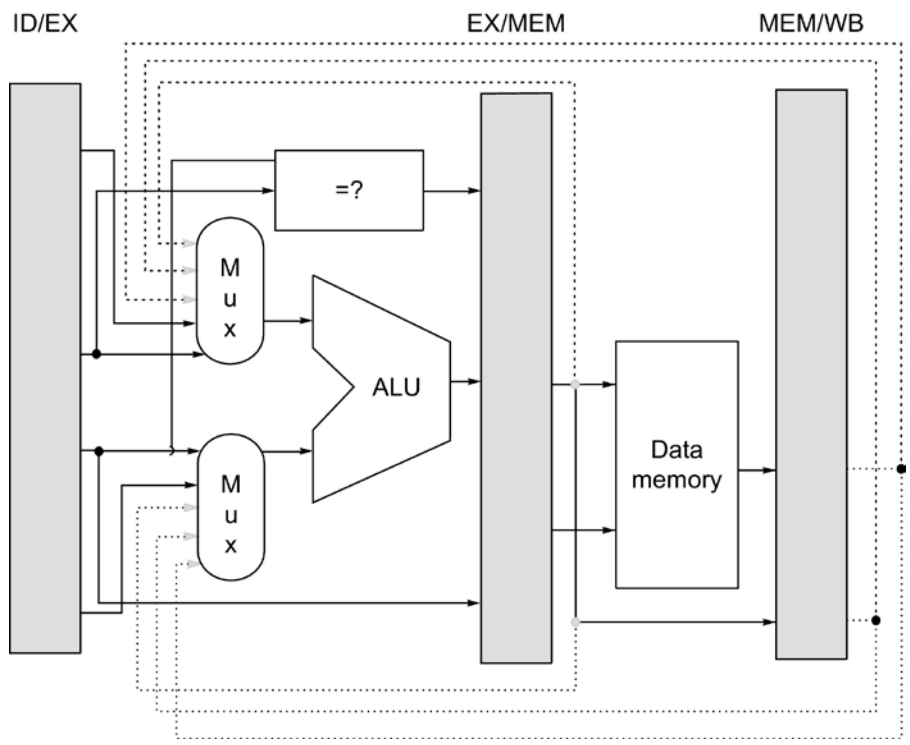

Figure C.24 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of: (1) the ALU output at the end of the EX, (2) the ALU output at the end of the MEM stage, and (3) the memory output at the end of the MEM stage.

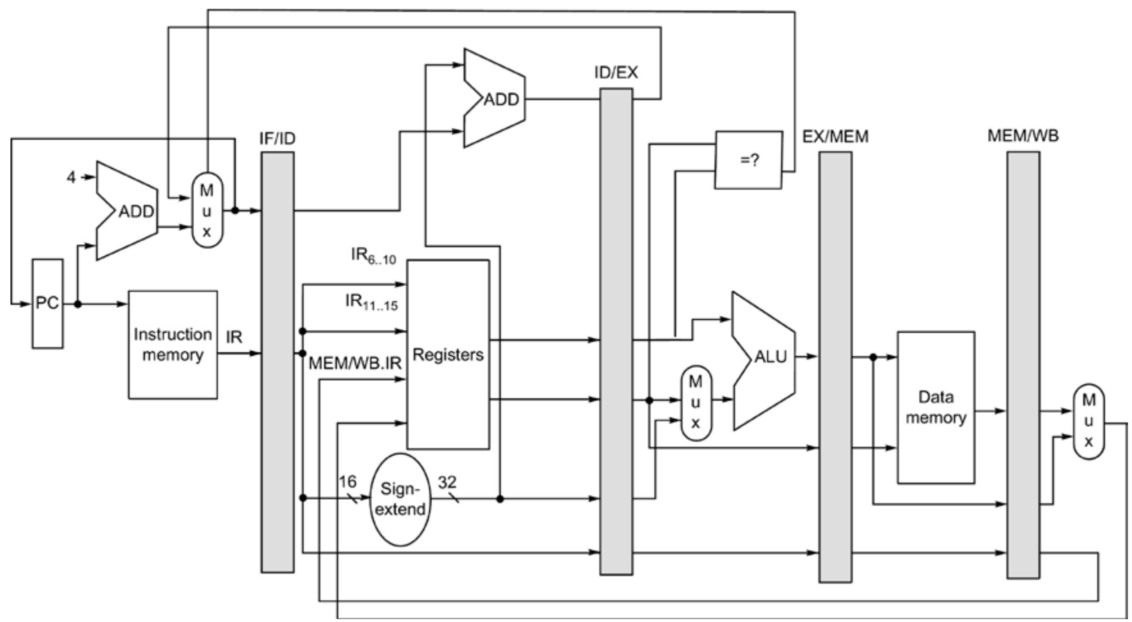

Figure C.25 To minimize the impact of deciding whether a conditional branch is taken, we compute the branch target address in ID while doing the conditional test and final selection of next PC in EX. As mentioned in Figure C.19, the PC can be thought of as a pipeline register (e.g., as part of ID/IF), which is written with the address of the next instruction at the end of each IF cycle.

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure C.26 Five categories are used to define what actions are needed for the different exception types. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, processors should be able to resume after such exceptions.

`ld`	IF	ID	EX	MEM	WB	
`add`		IF	ID	EX	MEM	WB

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

Figure C.27 Exceptions that may occur in the RISC V pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.

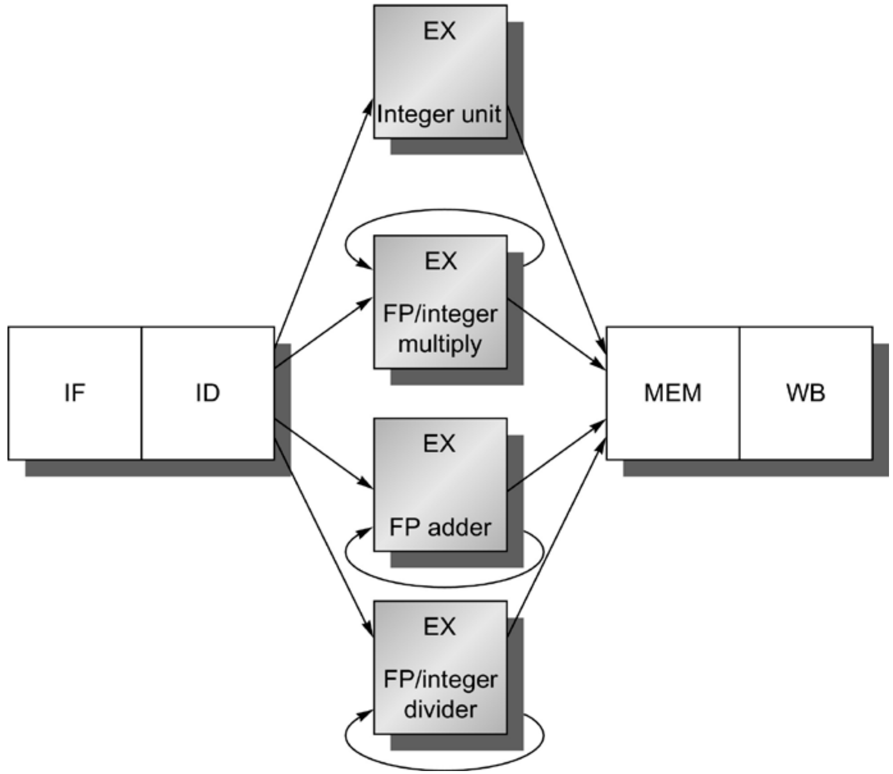

Figure C.28 The RISC V pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure C.29 Latencies and initiation intervals for functional units.

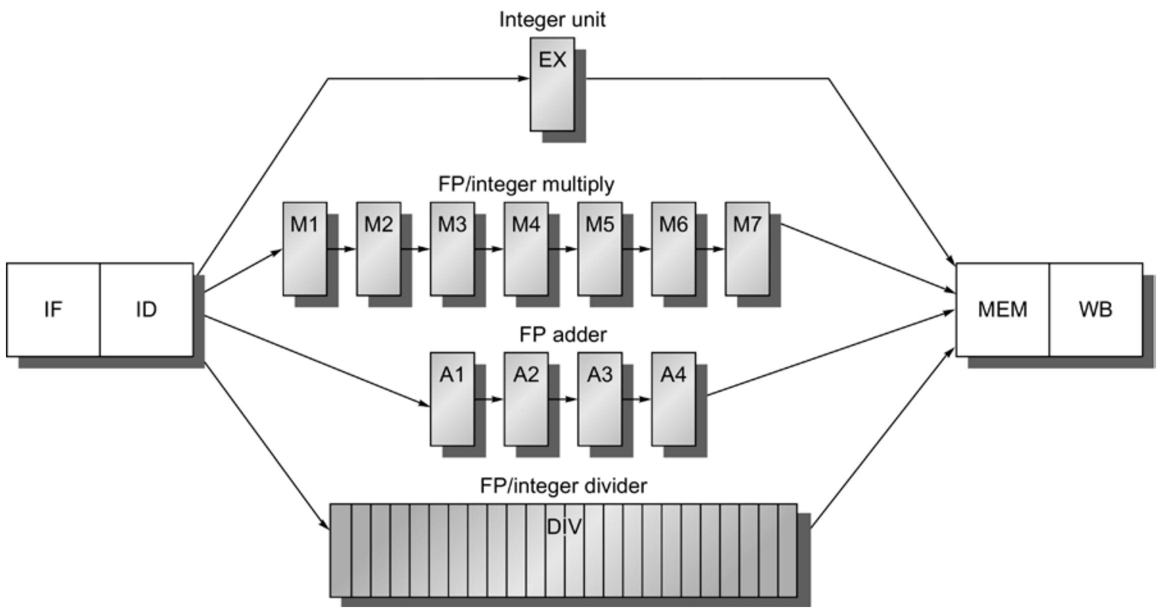

Figure C.30 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.

`fmul.d`	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
`fadd.d`		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
`fadd.d`			IF	ID	*EX*	**MEM**	WB				
`fsd`				IF	ID	*EX*	**MEM**	WB			

Figure C.31 The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

Instruction		Clock cycle number																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`fld`	`f4,0(x2)`	IF	ID	EX	MEM	WB												
`fmul.d`	`f0,f4,f6`		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`fadd.d`	`f2,f0,f8`			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
`fsd`	`f2,0(x2)`					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Figure C.32 A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The `fsd` must be stalled an extra cycle so that its MEM does not conflict with the `fadd.d`. Extra hardware could easily handle this case.

.

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`							IF	ID	EX	MEM	WB

Figure C.33 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, because an earlier divide in the FP unit could also finish on the same clock. Note that although the `fmul.d`, `fadd.d`, and `fld` are in the MEM stage in clock cycle 10, only the `fld` actually uses the memory, so no structural hazard exists for MEM.

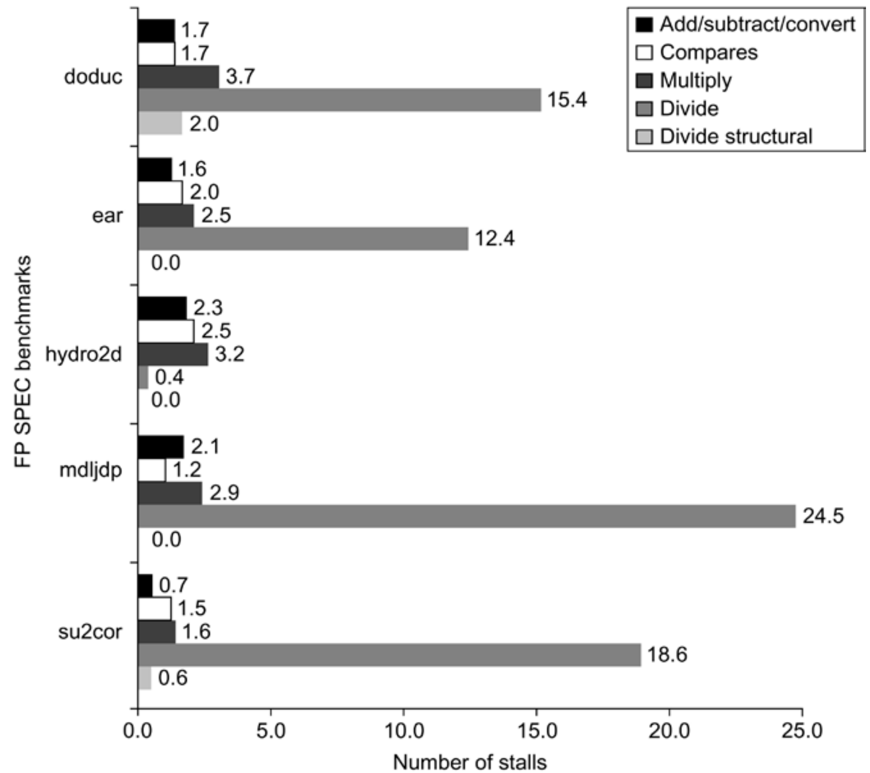

Figure C.34 Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks. Except for the divide structural hazards, these data do not depend on the frequency of an operation, only on its latency and the number of cycles before the result is used. The number of stalls from RAW hazards roughly tracks the latency of the FP unit. For example, the average number of stalls per FP add, subtract, or convert is 1.7 cycles, or 56% of the latency (three cycles). Likewise, the average number of stalls for multiplies and divides are 2.8 and 14.2, respectively, or 46% and 59% of the corresponding latency. Structural hazards for divides are rare, because the divide frequency is low.

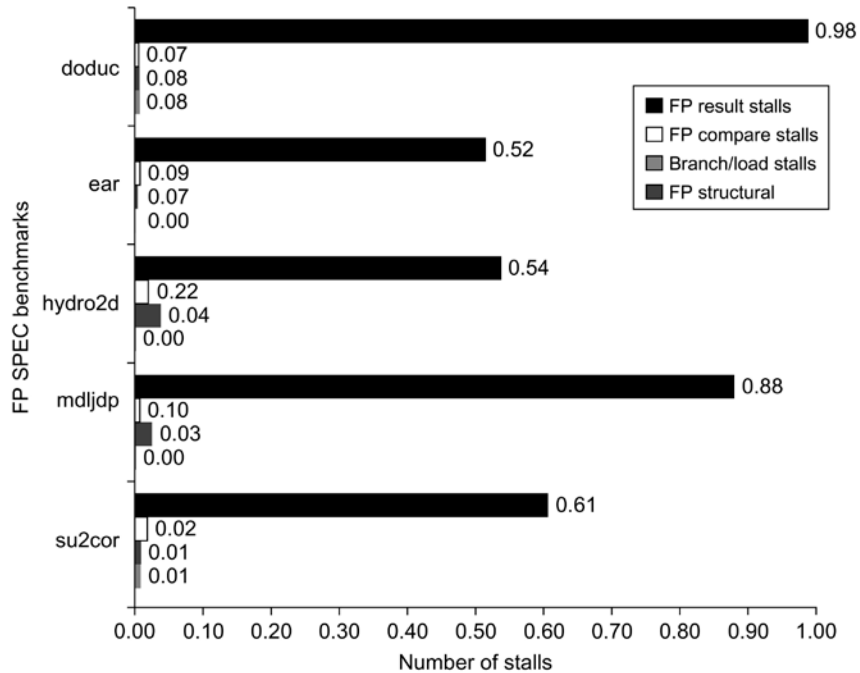

Figure C.35 The stalls occurring for the a simple RISC V FP pipeline for five of the SPEC89 FP benchmarks. The total number of stalls per instruction ranges from 0.65 for su2cor to 1.21 for doduc, with an average of 0.87. FP result stalls dominate in all cases, with an average of 0.71 stalls per instruction, or 82% of the stalled cycles. Compares generate an average of 0.1 stalls per instruction and are the second largest source. The divide structural hazard is only significant for doduc. Branch stalls are not accounted for, but would be small.

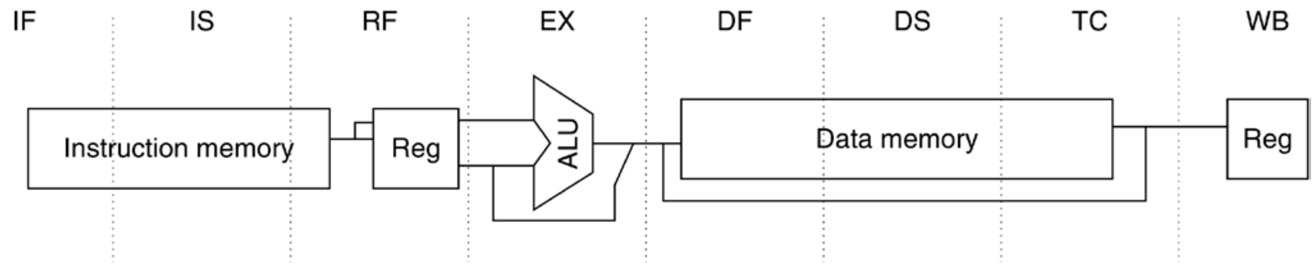

Figure C.36 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, because we cannot write the data into the register until we know whether the cache access was a hit or not.

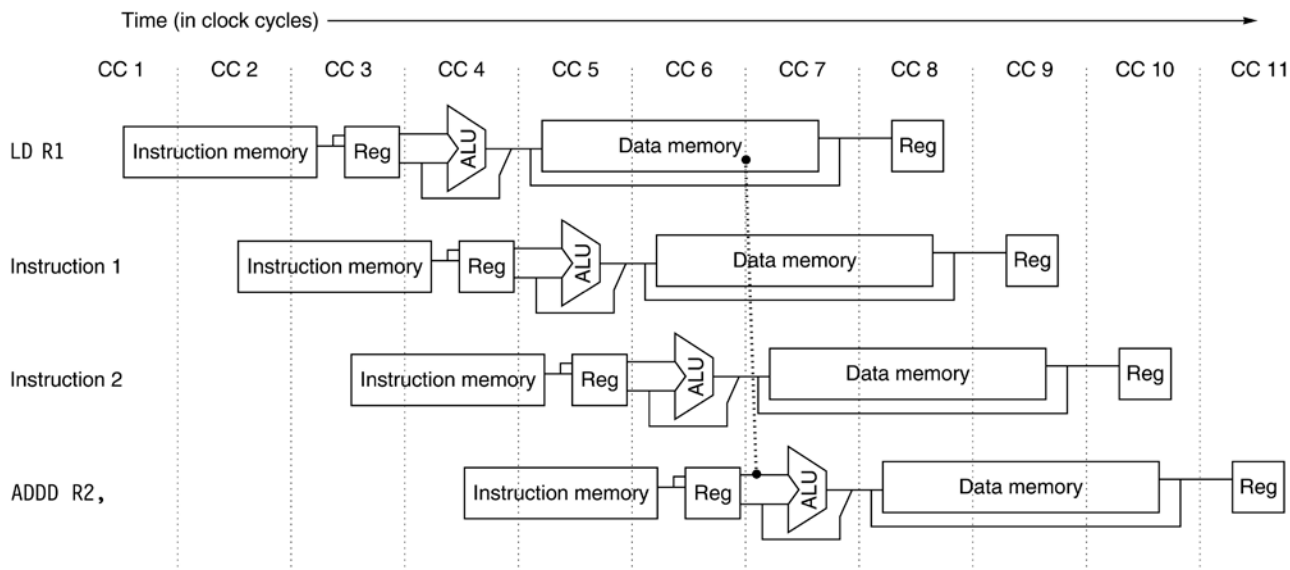

Figure C.37 The structure of the R4000 integer pipeline leads to a x1 load delay. A x1 delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

Instruction number	Clock number 1	2	3	4	5	6	7	8	9
`ld x1,...`	IF	IS	RF	EX	DF	DS	TC	WB	
`add x2,x1,...`		IF	IS	RF	Stall	Stall	EX	DF	DS
`sub x3,x1,...`			IF	IS	Stall	Stall	RF	EX	DF
`or x4,x1,...`				IF	Stall	Stall	IS	RF	EX

Figure C.38 A load instruction followed by an immediate use results in a x1 stall. Normal forwarding paths can be used after two cycles, so the `add` and `sub` get the value by forwarding after the stall. The `or` instruction gets the value from the register file. Because the two instructions after the load could be independent and hence not stall, the bypass can be to instructions that are three or four cycles after the load.

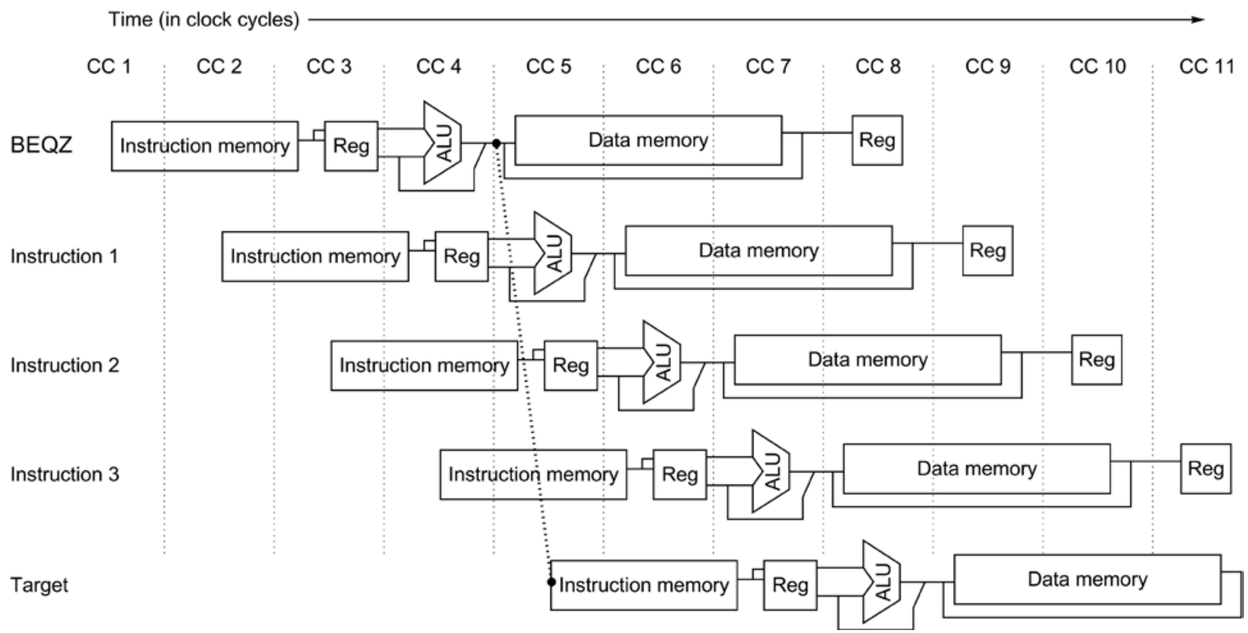

Figure C.39 The basic branch delay is three cycles, because the condition evaluation is performed during EX.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			Stall	Stall	Stall	Stall	Stall	Stall	Stall
Stall				Stall	Stall	Stall	Stall	Stall	Stall
Branch target					IF	IS	RF	EX	DF
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch instruction + 2			IF	IS	RF	EX	DF	DS	TC
Branch instruction + 3				IF	IS	RF	EX	DF	DS

Figure C.40 A taken branch, shown in the top portion of the figure, has a one-cycle delay slot followed by a x1 stall, while an untaken branch, shown in the bottom portion, has simply a one-cycle delay slot. The branch instruction can be an ordinary delayed branch or a branch-likely, which cancels the effect of the instruction in the delay slot if the branch is untaken.

Stage	Functional unit	Description
A	FP adder	Mantissa add stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

Figure C.41 The eight stages used in the R4000 floating-point pipelines.

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U, S+A, A+R, R+S
Multiply	8	4	U, E+M, M, M, M, N, N+A, R
Divide	36	35	U, A, R, D^{28}, D+A, D+R, D+A, D+R, A, R
Square root	112	111	U, E, $(A+R)^{108}$, A, R
Negate	2	1	U, S
Absolute value	2	1	U, S
FP compare	3	2	U, A, R

Figure C.42 The latencies and initiation intervals for the FP operations initiation intervals for the FP operations both depend on the FP unit stages that a given operation must use. The latency values assume that the destination instruction is an FP operation; the latencies are one cycle less when the destination is a store. The pipe stages are shown in the order in which they are used for any operation. The notation S + A indicates a clock cycle in which both the S and A stages are used. The notation D^{28} indicates that the D stage is used 28 times in a row.

		Clock cycle											
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Multiply	Issue	U	E+M	M	M	M	N	**N+A**	**R**					
Add	Issue		U	S+A	A+R	R+S								
	Issue			U	S+A	A+R	R+S							
	Issue				U	S+A	A+R	R+S						
	Stall					U	S+A	**A+R**	**R+S**					
	Stall						U	**S+A**	**A+R**	R+S				
	Issue							U	S+A	A+R	R+S			
	Issue								U	S+A	A+R	R+S		

Figure C.43 An FP multiply issued at clock 0 is followed by a single FP add issued between clocks 1 and 7. The second column indicates whether an instruction of the specified type stalls when it is issued *n* cycles later, where *n* is the clock cycle number in which the U stage of the second instruction occurs. The stage or stages that cause a stall are in bold. Note that this table deals with only the interaction between the multiply and *one* add issued between clocks 1 and 7. In this case, the add will stall if it is issued four or five cycles after the multiply; otherwise, it issues without stalling. Notice that the add will be stalled for two cycles if it issues in cycle 4 because on the next clock cycle it will still conflict with the multiply; if, however, the add issues in cycle 5, it will stall for only 1 clock cycle, because that will eliminate the conflicts.

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S+A	A+R	R+S									
Multiply	Issue		U	E+M	M	M	M	N	N+A	R				
	Issue			U	M	M	M	M	N	N+A	R			

Figure C.44 A multiply issuing after an add can always proceed without stalling, because the shorter instruction clears the shared pipeline stages before the longer instruction reaches them.

		Clock cycle											
Operation	**Issue/stall**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
Divide	Issued in cycle 0…	D	D	D	D	D	**D+A**	**D+R**	**D+A**	**D+R**	**A**	**R**	
Add	Issue		U	S+A	A+R	R+S							
	Issue			U	S+A	A+R	R+S						
	Stall				U	S+A	**A+R**	**R+S**					
	Stall					U	**S+A**	**A+R**	R+S				
	Stall						U	S+A	**A+R**	**R+S**			
	Stall							U	**S+A**	**A+R**	R+S		
	Stall								U	S+A	**A+R**	**R+S**	
	Stall									U	**S+A**	**A+R**	R+S
	Issue										U	S+A	A+R
	Issue											U	S+A
	Issue												U

Figure C.45 An FP divide can cause a stall for an add that starts near the end of the divide. The divide starts at cycle 0 and completes at cycle 35; the last 10 cycles of the divide are shown. Because the divide makes heavy use of the rounding hardware needed by the add, it stalls an add that starts in any of cycles 28–33. Notice that the add starting in cycle 28 will be stalled until cycle 36. If the add started right after the divide, it would not conflict, because the add could complete before the divide needed the shared stages, just as we saw in Figure C.44 for a multiply and add. As in the earlier figure, this example assumes *exactly* one add that reaches the U stage between clock cycles 26 and 35.

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S+A	**A+R**	**R+S**									
Divide	Stall		U	**A**	**R**	D	D	D	D	D	D	D	D	D
	Issue			U	A	R	D	D	D	D	D	D	D	D
	Issue				U	A	R	D	D	D	D	D	D	D

Figure C.46 A double-precision add is followed by a double-precision divide. If the divide starts one cycle after the add, the divide stalls, but after that there is no conflict.

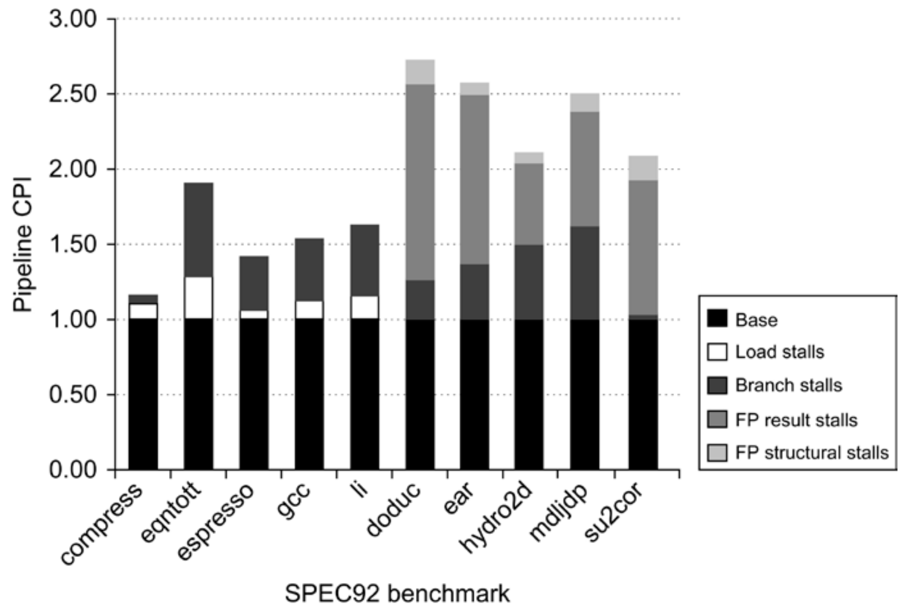

Figure C.47 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The left-most five programs are integer programs, and branch delays are the major CPI contributor for these. The right-most five programs are FP, and FP result stalls are the major contributor for these. Figure C.48 shows the numbers used to construct this plot.

Benchmark	Pipeline CPI	Load stalls	Branch stalls	FP result stalls	FP structural stalls
Compress	1.20	0.14	0.06	0.00	0.00
Eqntott	1.88	0.27	0.61	0.00	0.00
Espresso	1.42	0.07	0.35	0.00	0.00
Gcc	1.56	0.13	0.43	0.00	0.00
Li	1.64	0.18	0.46	0.00	0.00
Integer average	1.54	0.16	0.38	0.00	0.00
Doduc	2.84	0.01	0.22	1.39	0.22
Mdljdp2	2.66	0.01	0.31	1.20	0.15
Ear	2.17	0.00	0.46	0.59	0.12
Hydro2d	2.53	0.00	0.62	0.75	0.17
Su2cor	2.18	0.02	0.07	0.84	0.26
FP average	2.48	0.01	0.33	0.95	0.18
Overall average	2.00	0.10	0.36	0.46	0.09

Figure C.48 The total pipeline CPI and the contributions of the four major sources of stalls are shown. The major contributors are FP result stalls (both for branches and for FP inputs) and branch stalls, with loads and FP structural stalls adding less.

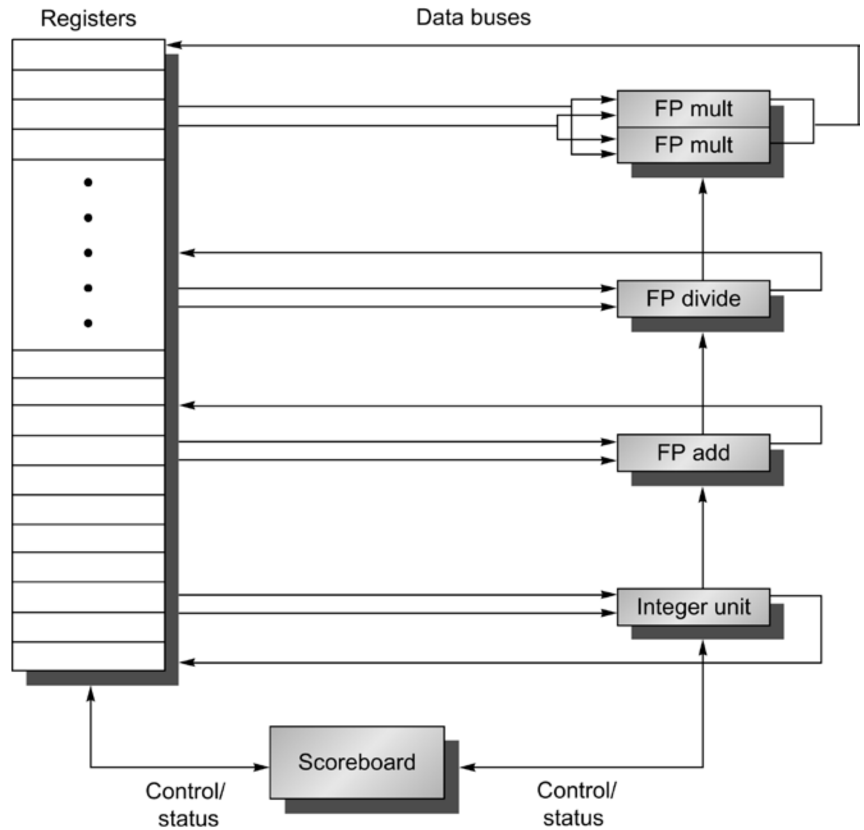

Figure C.49 The basic structure of a RISC V processor with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All of the data flow between the register file and the functional units over the buses (the horizontal lines, called *trunks* in the CDC 6600). 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. We will explore scoreboarding and its extensions in more detail in Chapter 3.