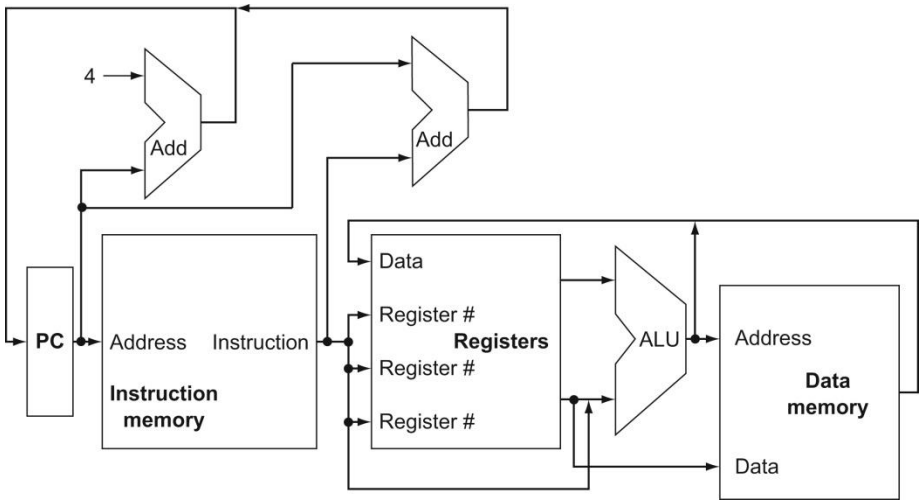


Figure 4.1 An abstract view of the implementation of the RISC-V subset showing the major functional units and the major connections between them. All instructions start by using the program counter to supply the instruction address to the instruction memory. After the instruction is fetched, the register operands used by an instruction are specified by fields of that instruction. Once the register operands have been fetched, they can be operated on to compute a memory address (for a load or store), to compute an arithmetic result (for an integer arithmetic-logical instruction), or an equality check (for a branch). If the instruction is an arithmetic-logical instruction, the result from the ALU must be written to a register. If the operation is a load or store, the ALU result is used as an address to either load a value from memory into the registers or store a value from the registers. The result from the ALU or memory is written back into the register file. Branches require the use of the ALU output to determine the next instruction address, which comes either from the adder (where the PC and branch offset are summed) or from an adder that increments the current PC by four. The thick lines interconnecting the functional units represent buses, which consist of multiple signals. The arrows are used to guide the reader in knowing how information flows. Since signal lines may cross, we explicitly show when crossing lines are connected by the presence of a dot where the lines cross.

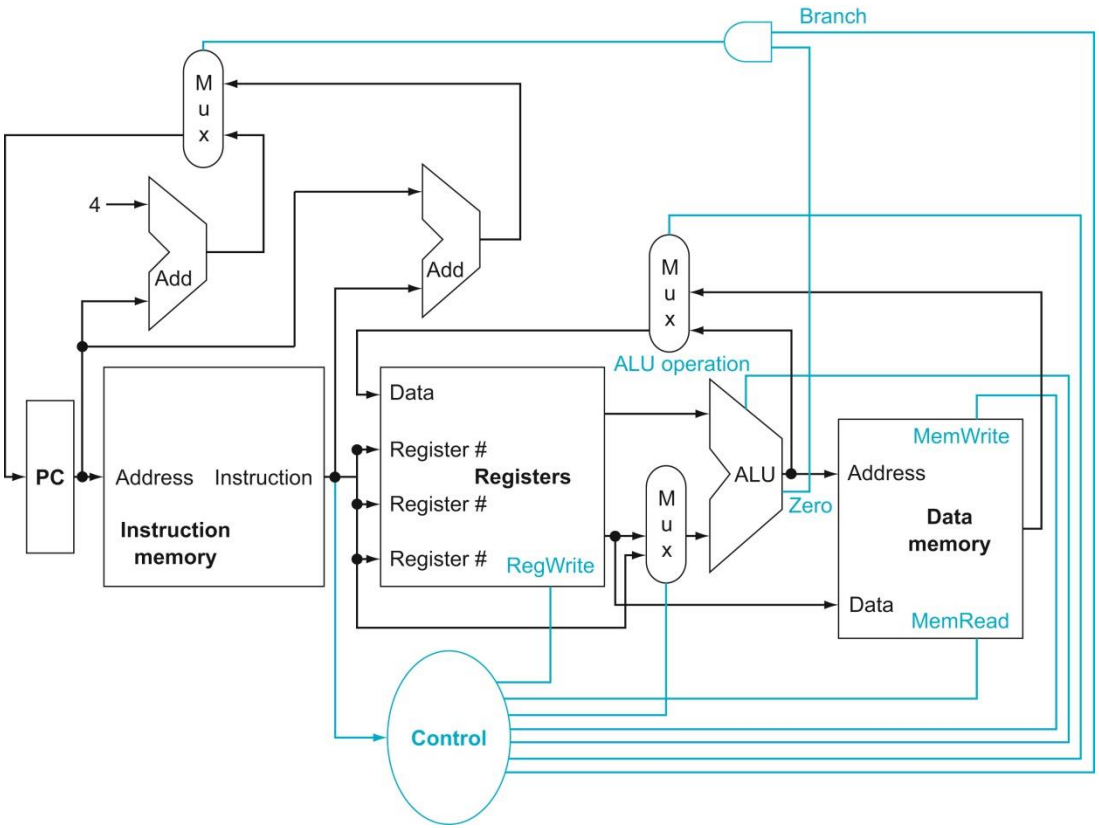


Figure 4.2 The basic implementation of the RISC-V subset, including the necessary multiplexors and control lines. The top multiplexor ("Mux") controls what value replaces the PC (PC + 4 or the branch destination address); the multiplexor is controlled by the gate that "ANDs" together the Zero output of the ALU and a control signal that indicates that the instruction is a branch. The middle multiplexor, whose output returns to the register file, is used to steer the output of the ALU (in the case of an arithmetic-logical instruction) or the output of the data memory (in the case of a load) for writing into the register file. Finally, the bottom-most multiplexor is used to determine whether the second ALU input is from the registers (for an arithmetic-logical instruction or a branch) or from the offset field of the instruction (for a load or store). The added control lines are straightforward and determine the operation performed at the ALU, whether the data memory should read or write, and whether the registers should perform a write operation. The control lines are shown in color to make them easier to see.

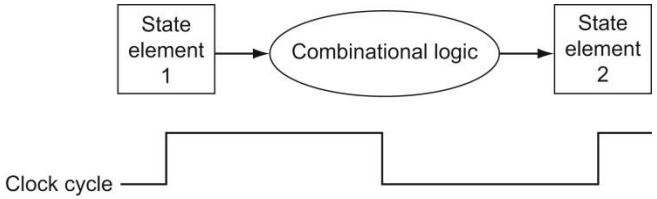


Figure 4.3 Combinational logic, state elements, and the clock are closely related. In a synchronous digital system, the clock determines when elements with state will write values into internal storage. Any inputs to a state element must reach a stable value (that is, have reached a value from which they will not change until after the clock edge) before the active clock edge causes the state to be updated. All state elements in this chapter, including memory, are assumed positive edge-triggered; that is, they change on the rising clock edge.

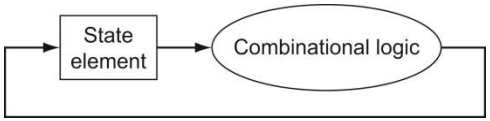


Figure 4.4 An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values. Of course, the clock cycle still must be long enough so that the input values are stable when the active clock edge occurs. Feedback cannot occur within one clock cycle because of the edge-triggeredupdate of the state element. If feedback were possible, this design could not work properly. Our designs in this chapter and the next rely on the edge-triggered timing methodology and on structures like the one shown in this figure.

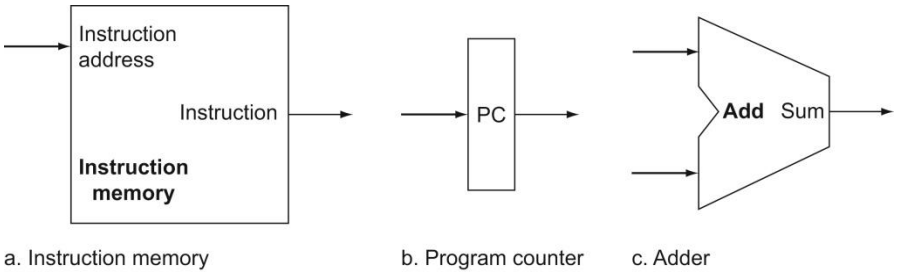


Figure 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

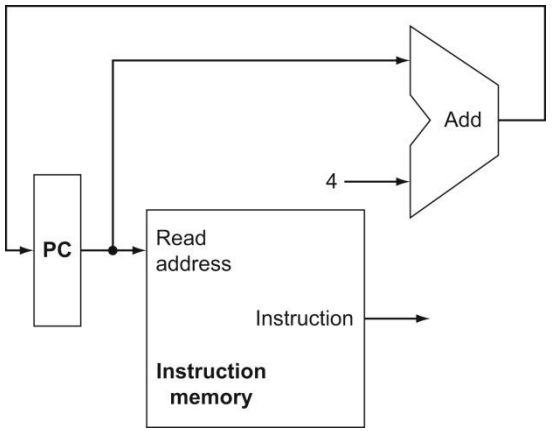


Figure 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.

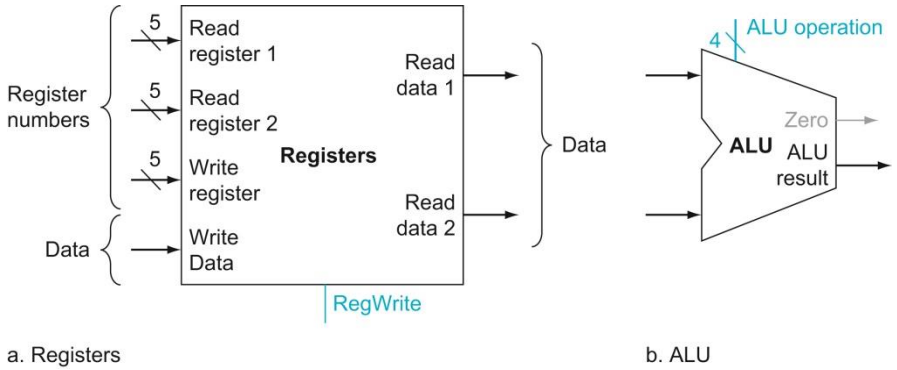


Figure 4.7 The two elements needed to implement R-format ALU operations are the register file and the ALU. The register file contains all the registers and has two read ports and one write port. The design of multiported register files is discussed in Section A.8 of Appendix A. The register file always outputs the contents of the registers corresponding to the Read register inputs on the outputs; no other control inputs are needed. In contrast, a register write must be explicitly indicated by asserting the write control signal. Remember that writes are edge-triggered, so that all the write inputs (i.e., the value to be written, the register number, and the write control signal) must be valid at the clock edge. Since writes to the register file are edge-triggered, our design can legally read and write the same register within a clock cycle: the read will get the value written in an earlier clock cycle, while the value written will be available to a read in a subsequent clock cycle. The inputs carrying the register number to the register file are all 5 bits wide, whereas the lines carrying data values are 32 bits wide. The operation to be performed by the ALU is controlled with the ALU operation signal, which will be 4 bits wide, using the ALU designed in Appendix A. We will use the Zero detection output of the ALU shortly to implement conditional branches.

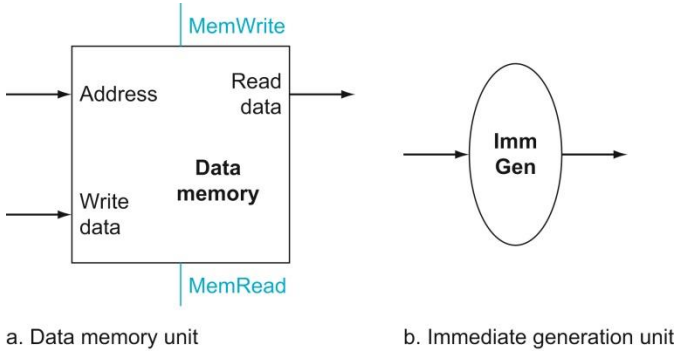


Figure 4.8 The two units needed to implement loads and stores, in addition to the register file and ALU of Figure 4.7, are the data memory unit and the immediate generation unit. The memory unit is a state element with inputs for the address and the write data, and a single output for the read result. There are separate read and write controls, although only one of these may be asserted on any given clock. The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems, as we will see in Chapter 5. The immediate generation unit (ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign-extended into a 32-bit result appearing on the output (see Chapter 2). We assume the data memory is edge-triggered for writes. Standard memory chips actually have a write enable signal that is used for writes. Although the write enable is not edge-triggered, our edge-triggered design could easily be adapted to work with real memory chips. See Section A.8 of Appendix A for further discussion of how real memory chips work.

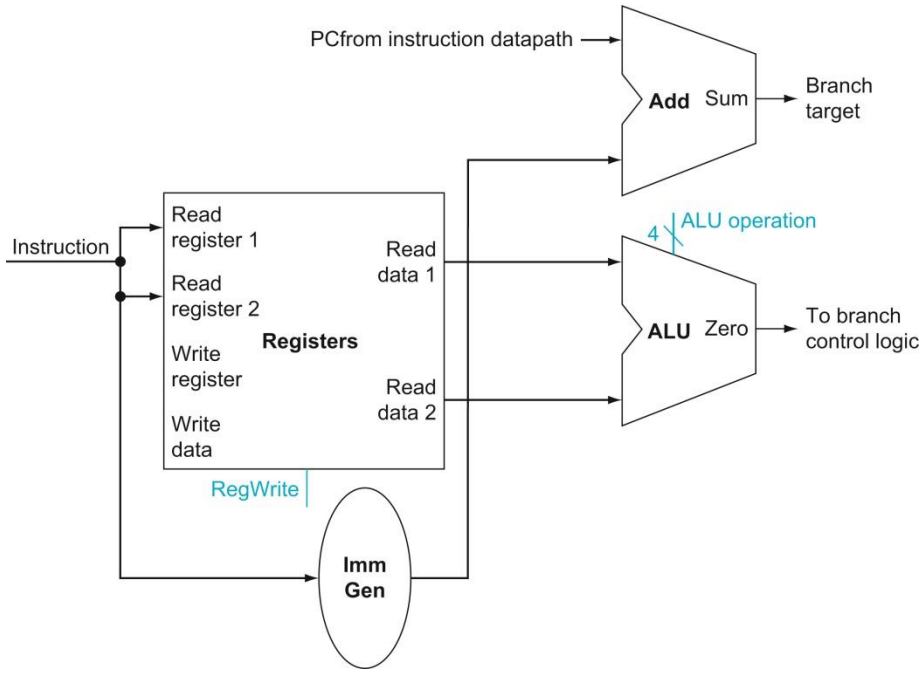


Figure 4.9 The portion of a datapath for a branch uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and immediate (the branch displacement). Control logic is used to decide whether the incremented PC or branch target should replace the PC, based on the Zero output of the ALU.

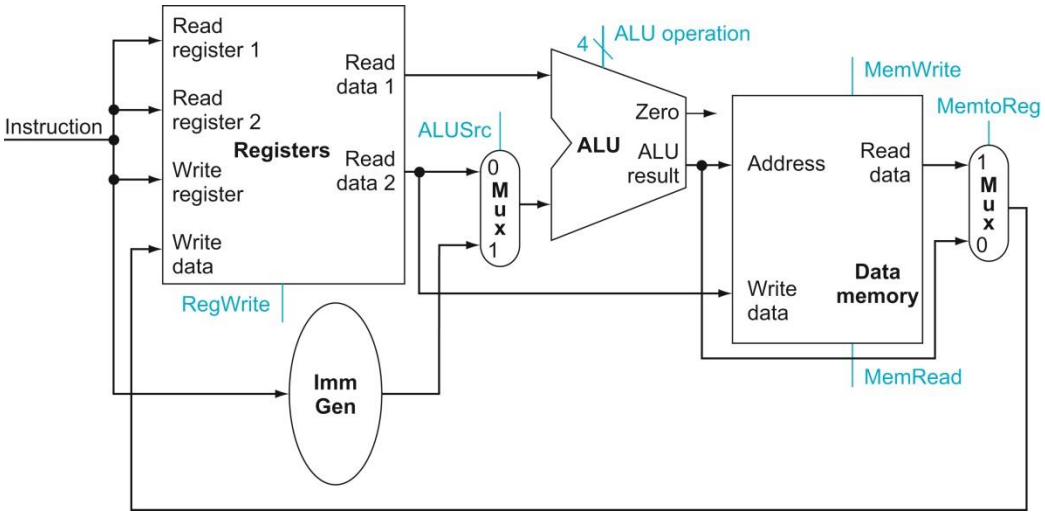


Figure 4.10 The datapath for the memory instructions and the R-type instructions. This example shows how a single datapath can be assembled from the pieces in Figures 4.7 and 4.8 by adding multiplexors. Two multiplexors are needed, as described in the example.

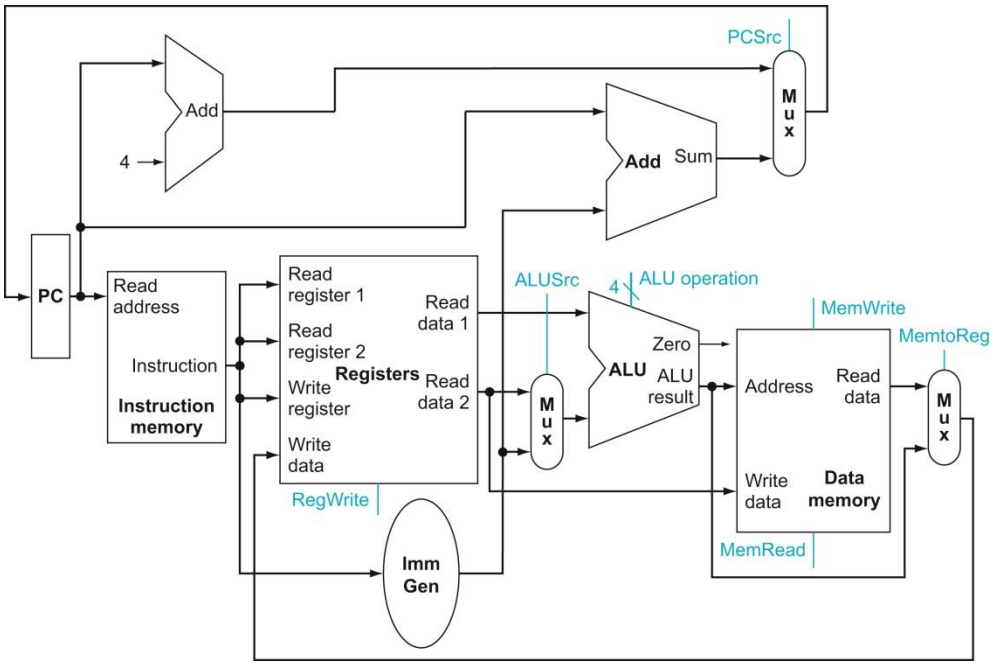


Figure 4.11 The simple datapath for the core RISC-V architecture combines the elements required by different instruction classes. The components come from Figures 4.6, 4.9, and 4.10. This datapath can execute the basic instructions (load-store register, ALU operations, and branches) in a single clock cycle. Just one additional multiplexor is needed to integrate branches.

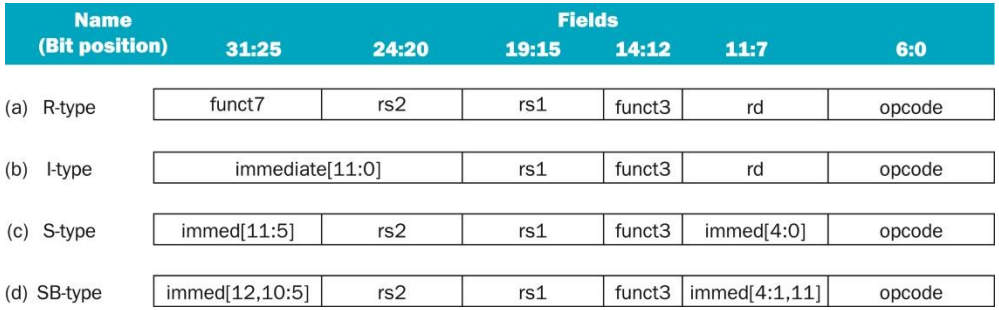
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000		AND	0000
R-type	10	or	0000000	110	OR	0001

Figure 4.12 How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction. The instruction, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the funct7 or funct3 fields; in this case, we say that we “don’t care” about the value of the opcode, and the bits are shown as Xs. When the ALUOp value is 10, then the funct7 and funct3 fields are used to set the ALU control input. See Appendix A.

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

Figure 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and funct fields. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. While we show all 10 bits of funct fields, note that the only bits with different values for the four R-format instructions are bits 30, 14, 13, and 12. Thus, we only need these four funct field bits as input for ALU control instead of all 10.



Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

Figure 4.14 The four instruction classes (arithmetic, load, store, and conditional branch) use four different instruction formats. (a) Instruction format for R-type arithmetic instructions (opcode = 51ten), which have three register operands: rs1, rs2, and rd. Fields rs1 and rd are sources, and rd is the destination. The ALU function is in the funct3 and funct7 fields and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, and, and or. (b) Instruction format for I-type load instructions (opcode = 3ten). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. Field rd is the destination register for the loaded value. (c) Instruction format for S-type store instructions (opcode = 35ten). The register rs1 is the base register that is added to the 12-bit immediate field to form the memory address. (The immediate field is split into a 7-bit piece and a 5-bit piece.) Field rs2 is the source register whose value should be stored into memory. (d) Instruction format for SB-type conditional branch instructions (opcode = 99ten). The registers rs1 and rs2 compared. The 12-bit immediate address field is sign-extended, shifted left 1 bit, and added to the PC to compute the branch target address. Figures 4.17 and 4.18 give the rationale for the unusual bit ordering for SB-type.

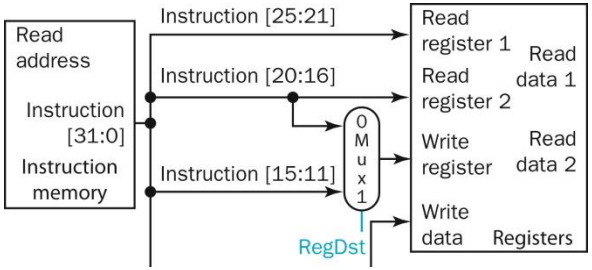


Figure 4.15 The MIPS, arithmetic instruction format, data transfer instruction format, and their impact on the MIPS datapath. For MIPS arithmetic instructions using the R format, rd is the destination register, rs is the first register operand, and rt is the second register operand. For MIPS load and immediate instructions, rs is still the first register operand, but rt is now the destination register. Hence the need of the 2:1 multiplexor to pick between the rd and rt fields to write the correct register.

Name (Field size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Figure 4.16 The actual RISC-V formats. Figure 4.16 introduces R-, I-, S-, and U-types, which are straightforward.

		Immediate Output Bit by Bit																														
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																														
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	"	"	"	"	0
Uncond. Jump	U	"	"	"	"	"	"	"	"	"	"	"	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	i19	i18	i17	i16	i15	i14	i13	i12	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4	4	4	3

Figure 4.17 Inputs to immediate if hypotheticaly conditional branches use the S format, and if jumps, use the U format.

		Immediate Output Bit by Bit																																
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Instruction	Format	Immediate Input Bit by Bit																																
Load, Arith. Imm.	I	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i31	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20
Store	S	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i11	i10	i9	i8	i7
Cond. Branch	SB	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	"	i7	"	"	"	"	"	"	"	"	"	"	"	0
Uncond. Jump	UJ	"	"	"	"	"	"	"	"	"	"	"	"	i19	i18	i17	i16	i15	i14	i13	i12	i20	"	"	"	"	"	"	"	"	"	"	"	"
Load Upper Imm.	U	"	i30	i29	i28	i27	i26	i25	i24	i23	i22	i21	i20	"	"	"	"	"	"	"	"	0	0	0	0	0	0	0	0	0	0	0	"
Unique Inputs		1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	4	2	2	2	2	2	2	3	3	3	3	3

Figure 4.18 Inputs to immediate given that branches use the SB format and jumps use the UJ format, which is what RISC-V uses.

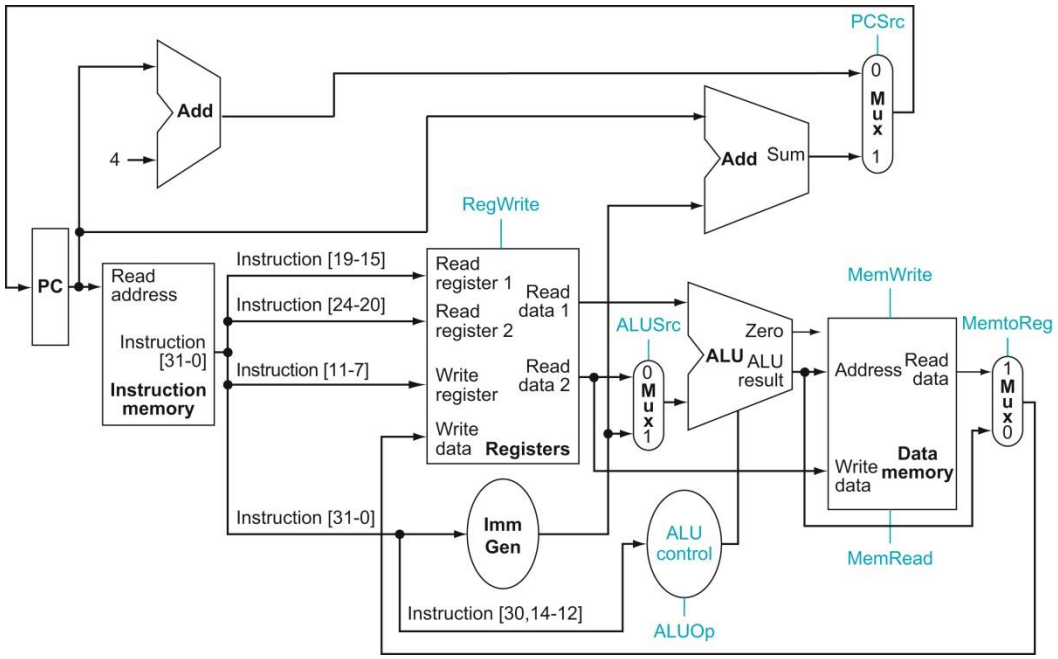


Figure 4.19 The datapath of Figure 4.11 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added, which depends on the funct3 field and part of the funct7 field. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 4.20 The effect of each of the six control signals. When the 1-bit control to a twoway multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. Gating the clock externally to a state element can create timing problems. (See Appendix A for further discussion of this problem.)

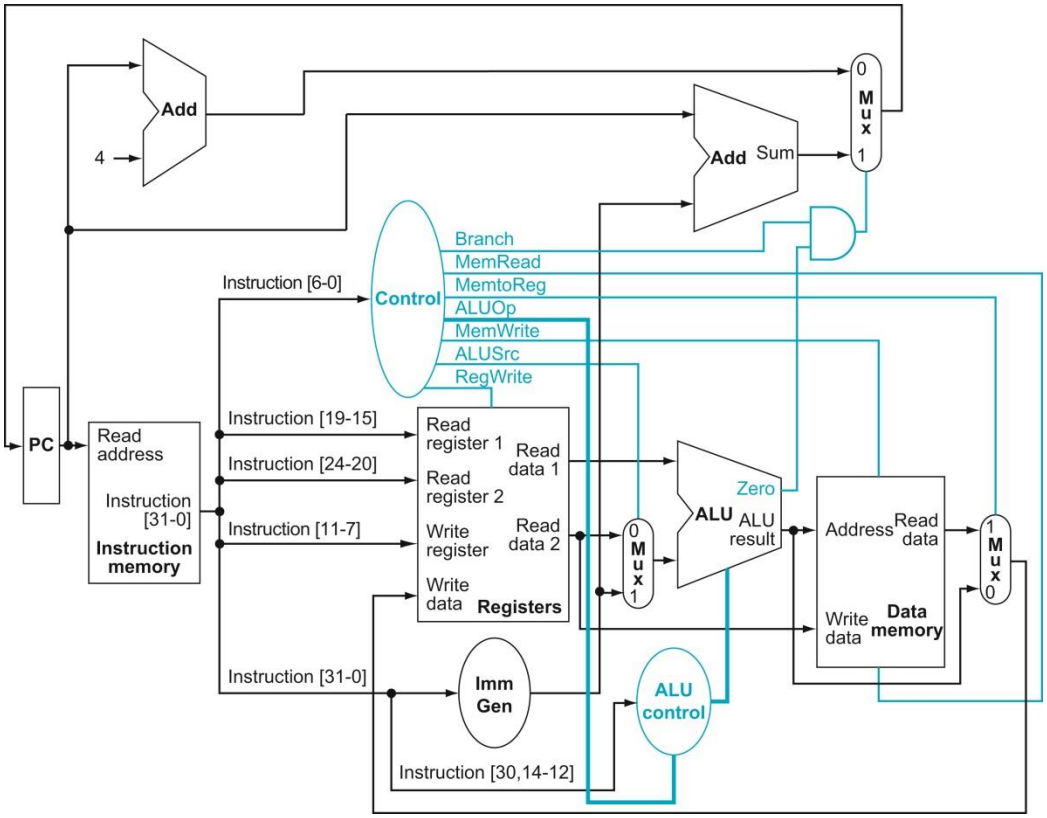


Figure 4.21 The simple datapath with the control unit. The input to the control unit is the 7-bit opcode field from the instruction. The outputs of the control unit consist of two 1-bit signals that are used to control multiplexors (ALUSrc and MemtoReg), three signals for controlling reads and writes in the register file and data memory (RegWrite, MemRead, and MemWrite), a 1-bit signal used in determining whether to possibly branch (Branch), and a 2-bit control signal for the ALU (ALUOp). An AND gate is used to combine the branch control signal and the Zero output from the ALU; the AND gate output controls the selection of the next PC. Notice that PCSrc is now a derived signal, rather than one coming directly from the control unit. Thus, we drop the signal name in subsequent figures.

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
lw	1	1	1	1	0	0	0	0
sw	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Figure 4.22 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (add, sub, and, and or). For all these instructions, the source register fields are rs1 and rs2, and the destination register field is rd; this defines how the signals ALUSrc is set. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct fields. The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegWrite is set for a load to cause the result to be stored in the rd register. The ALUOp field for branch is set for subtract (ALU control = 01), which is used to test for equality. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don’t care. This type of don’t care must be added by the designer, since it depends on knowledge of how the datapath works.

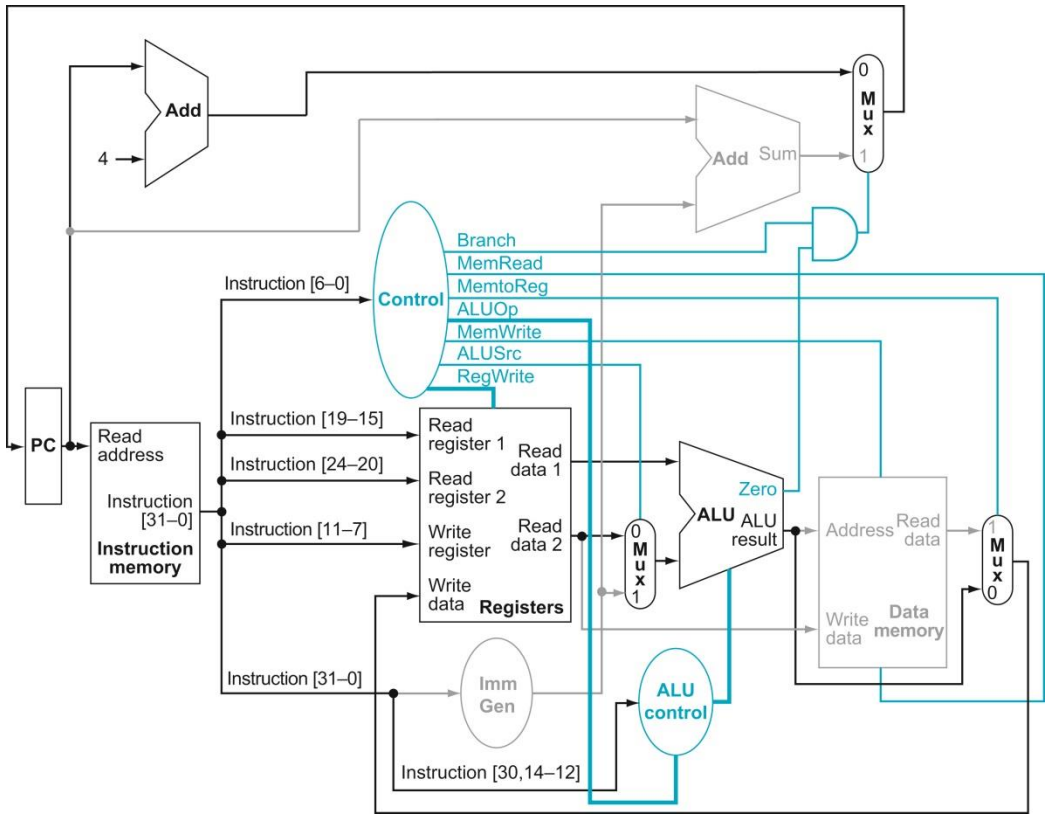


Figure 4.23 The datapath in operation for an R-type instruction, such as add x1, x2, x3. The control lines, datapath units, and connections that are active are highlighted.

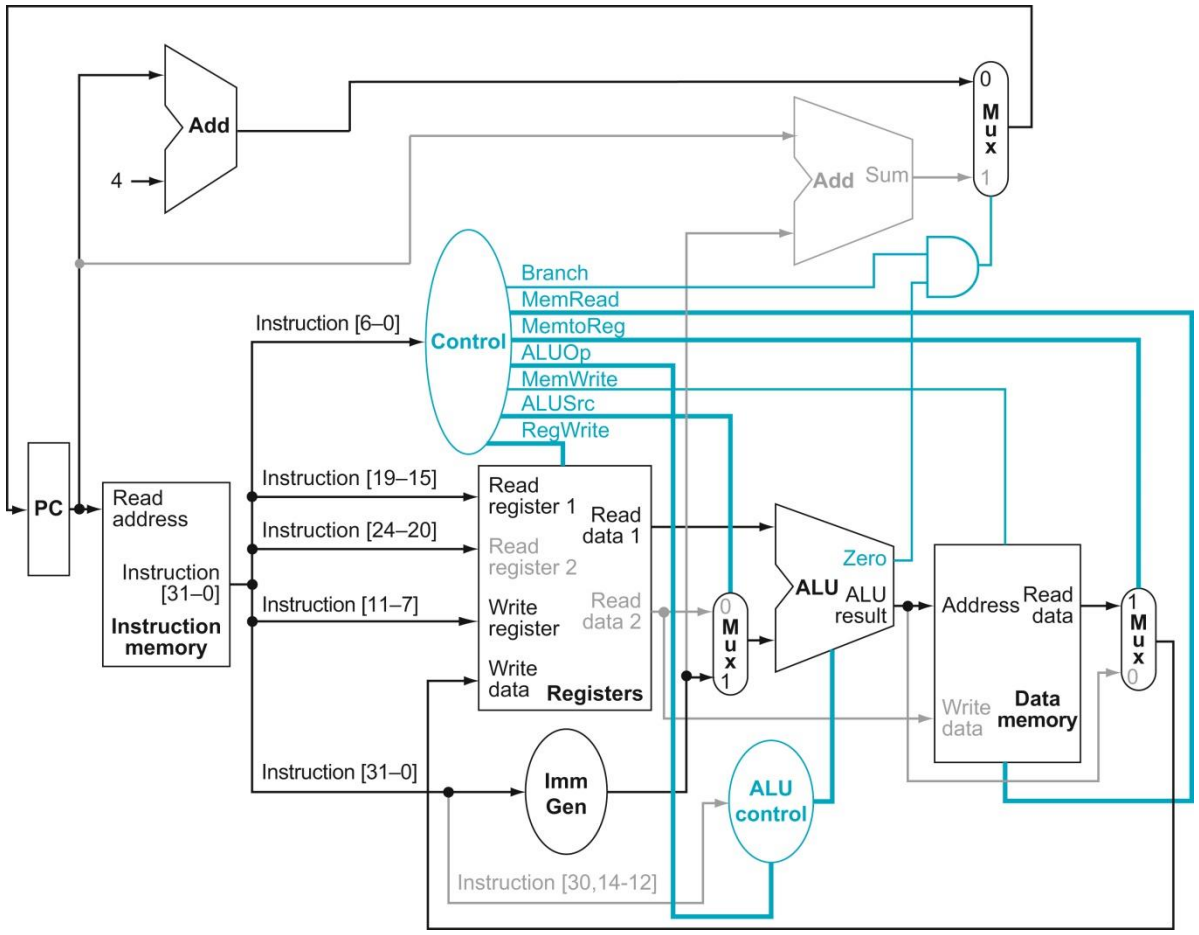


Figure 4.24 The datapath in operation for a load instruction. The control lines, datapath units, and connections that are active are highlighted. A store instruction would operate very similarly. The main difference would be that the memory control would indicate a write rather than a read, the second register value read would be used for the data to store, and the operation of writing the data memory value to the register file would not occur.

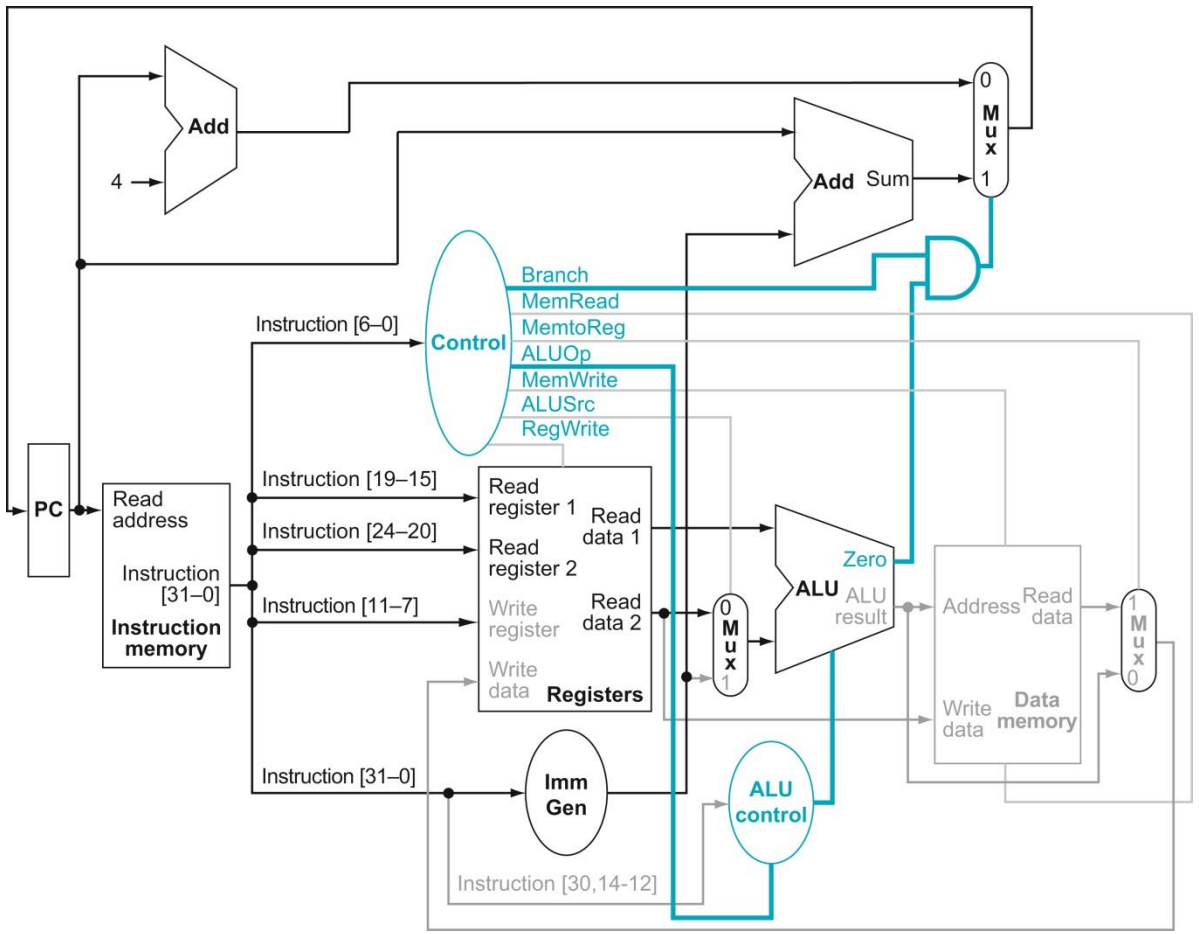


Figure 4.25 The datapath in operation for a branch-if-equal instruction. The control lines, datapath units, and connections that are active are highlighted. After using the register file and ALU to perform the compare, the Zero output is used to select the next program counter from between the two candidates.

Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Figure 4.26 The control function for the simple single-cycle implementation is completely specified by this truth table. The top seven rows of the table gives the combinations of input signals that correspond to the four instruction classes, one per column, that determine the control output settings. The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output RegWrite is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression Op4 · Op5, since this is sufficient to distinguish the R-format instructions from lw, sw, and beq. We do not take advantage of this simplification, since the rest of the RISC-V opcodes are used in a full implementation.

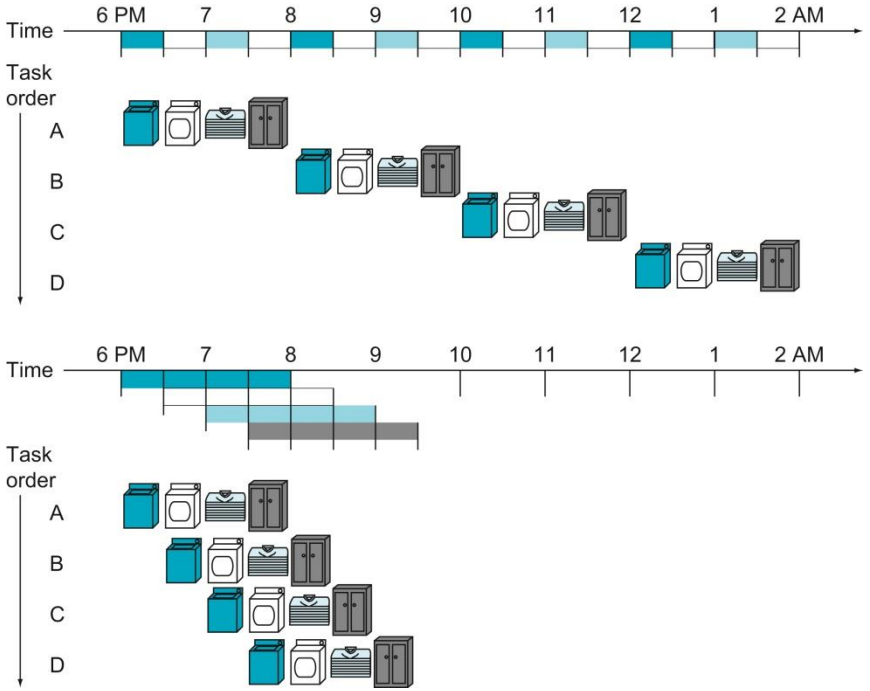


Figure 4.27 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for four loads of washing, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Figure 4.28 Total time for each instruction calculated from the time for each component.
This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

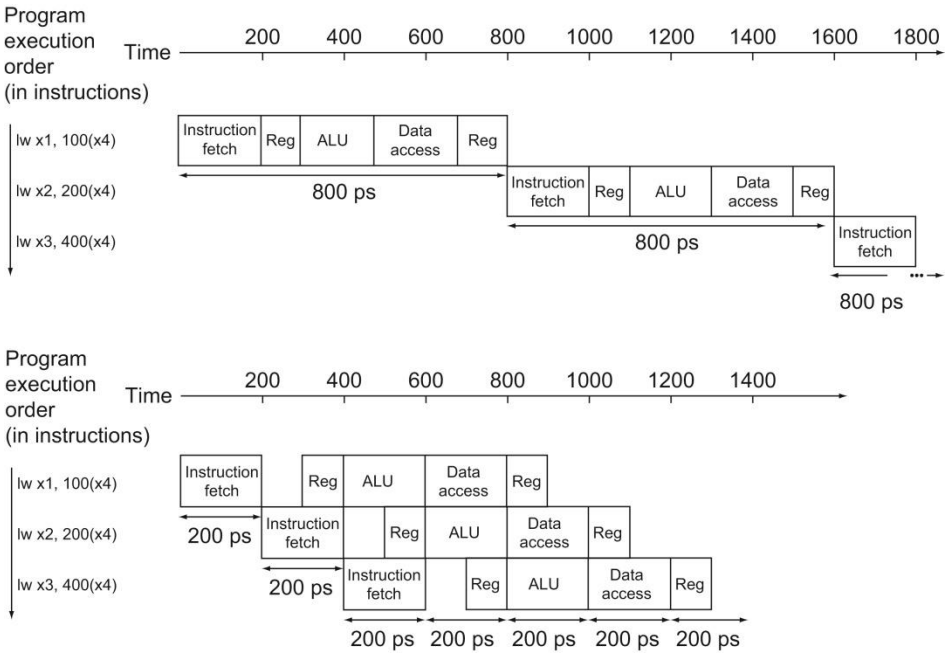


Figure 4.29 Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom). Both use the same hardware components, whose time is listed in Figure 4.28. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.27. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

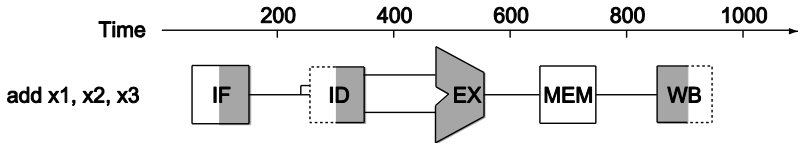


Figure 4.29 Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom). Both use the same hardware components, whose time is listed in Figure 4.28. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.27. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

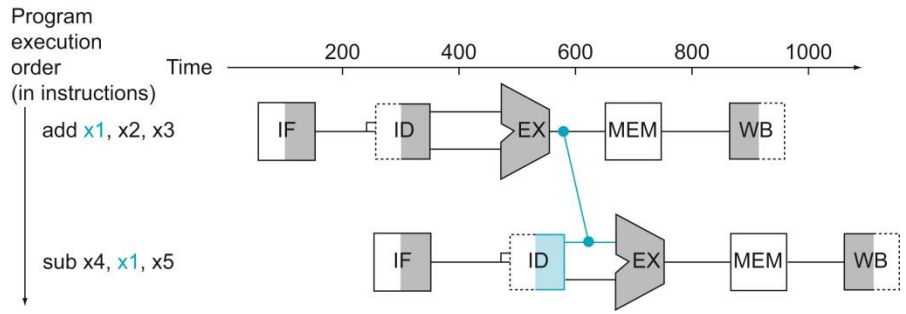


Figure 4.31 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register x1 read in the second stage of sub.

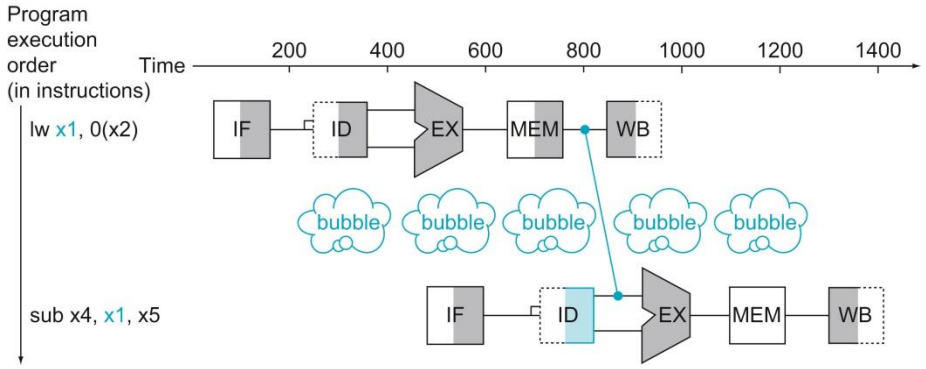


Figure 4.32 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard.

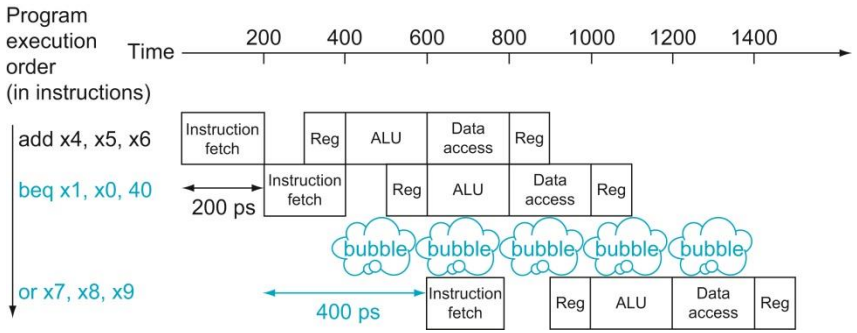


Figure 4.33 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the or instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.9. The effect on performance, however, is the same as would occur if a bubble were inserted.

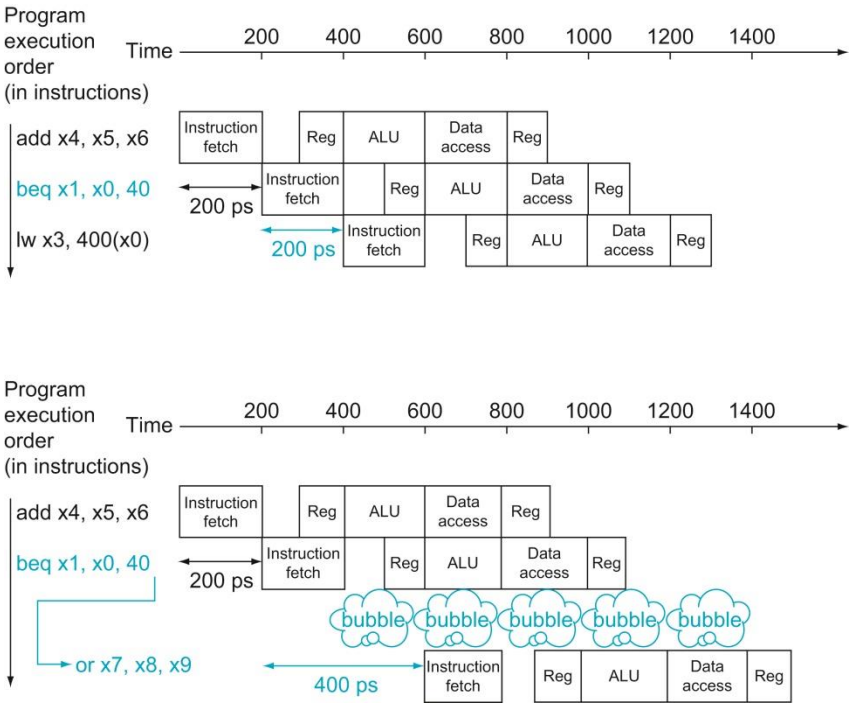


Figure 4.34 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.33, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.9 will reveal the details.

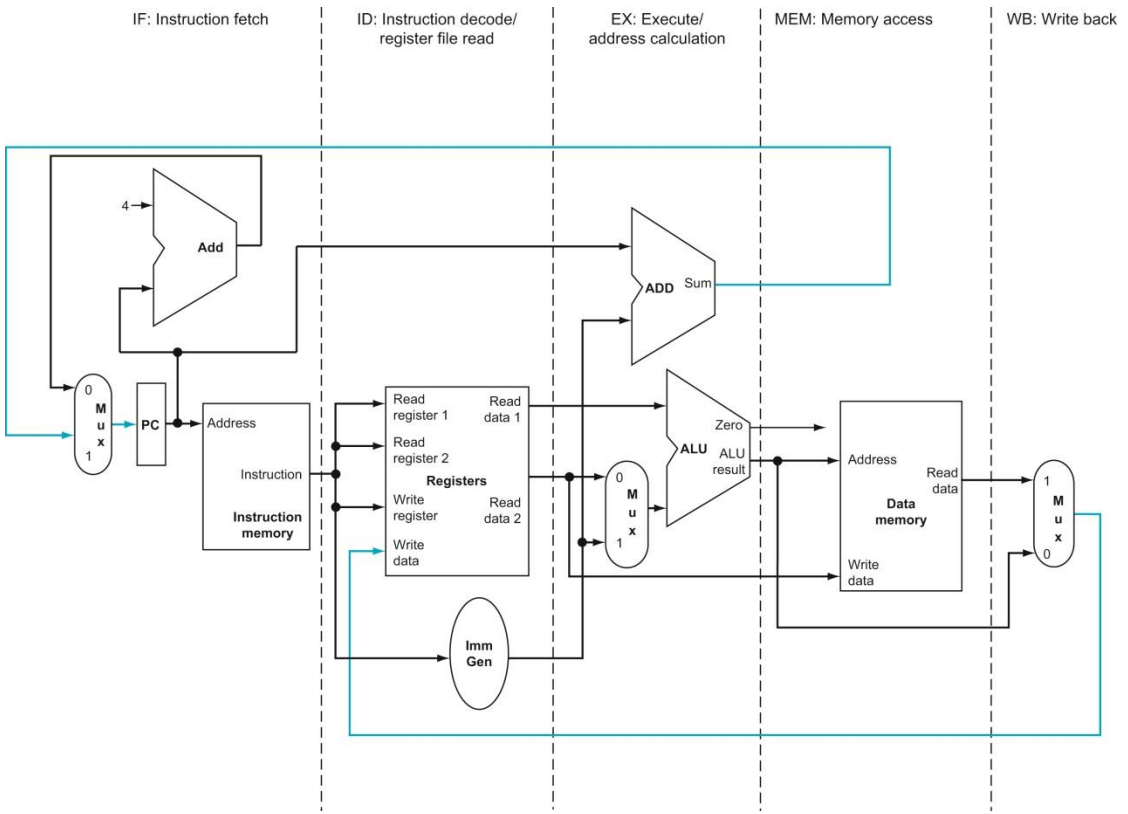


Figure 4.35 The single-cycle datapath from Section 4.4 (similar to Figure 4.21). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

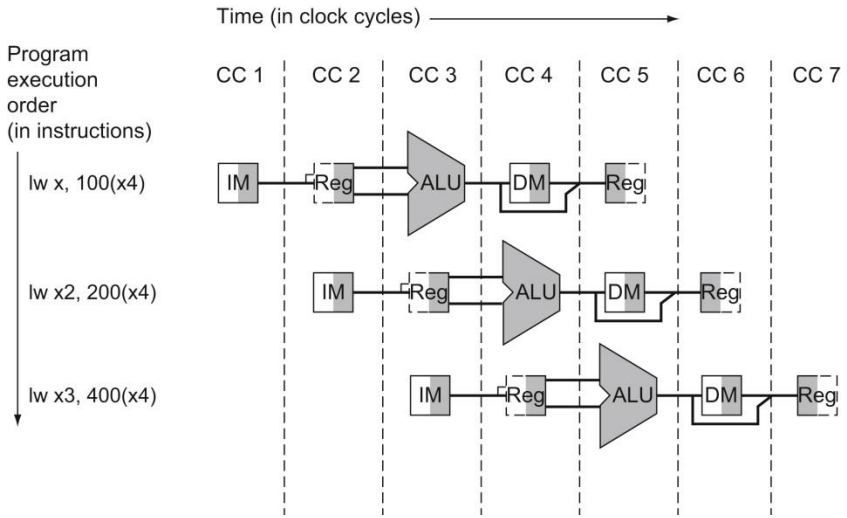


Figure 4.36 Instructions being executed using the single-cycle datapath in Figure 4.35, assuming pipelined execution. Similar to Figures 4.30 through 4.32, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.48. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

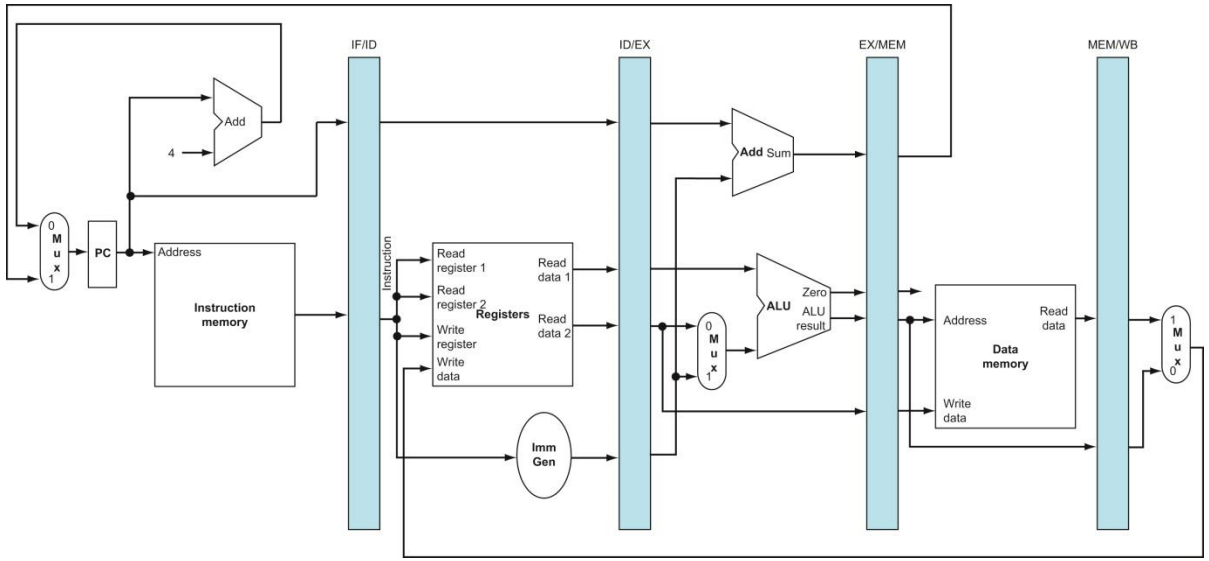


Figure 4.37 The pipelined version of the datapath in Figure 4.35. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively.

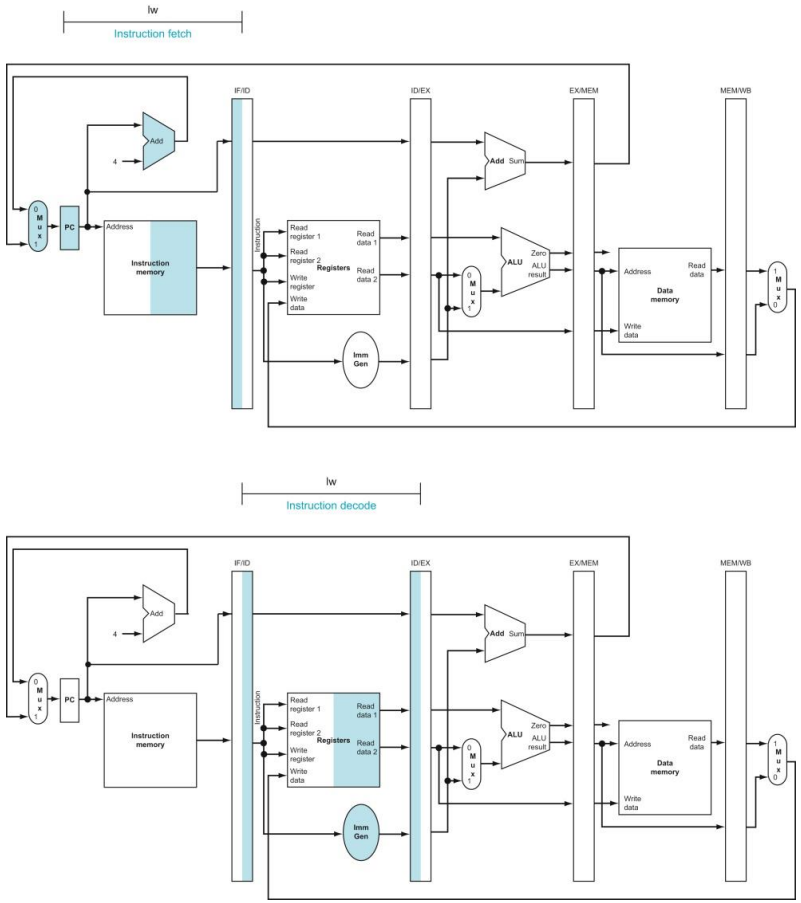


Figure 4.38 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 4.37 highlighted. The highlighting convention is the same as that used in Figure 4.30. As in Section 4.2, there is no confusion when reading and writing registers, because the contents change only on the clock edge. Although the load needs only the top register in stage 2, it doesn't hurt to do potentially extra work, so it sign-extends the constant and reads both registers into the ID/EX pipeline register. We don't need all three operands, but it simplifies control to keep all three.

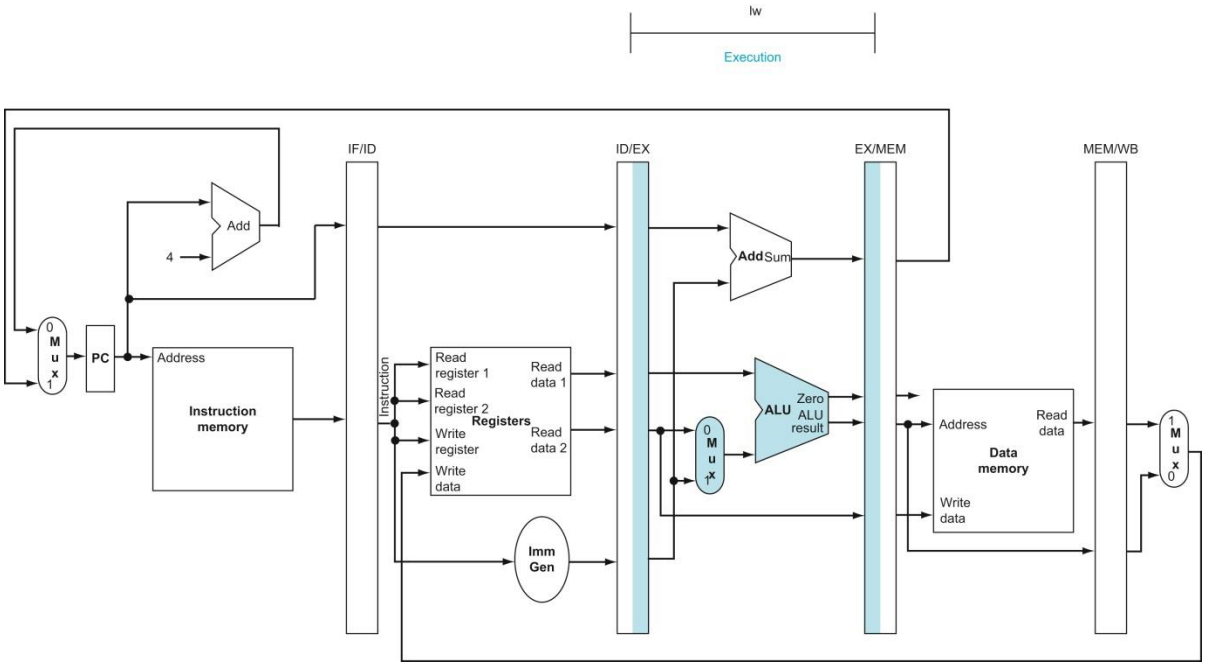


Figure 4.39 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

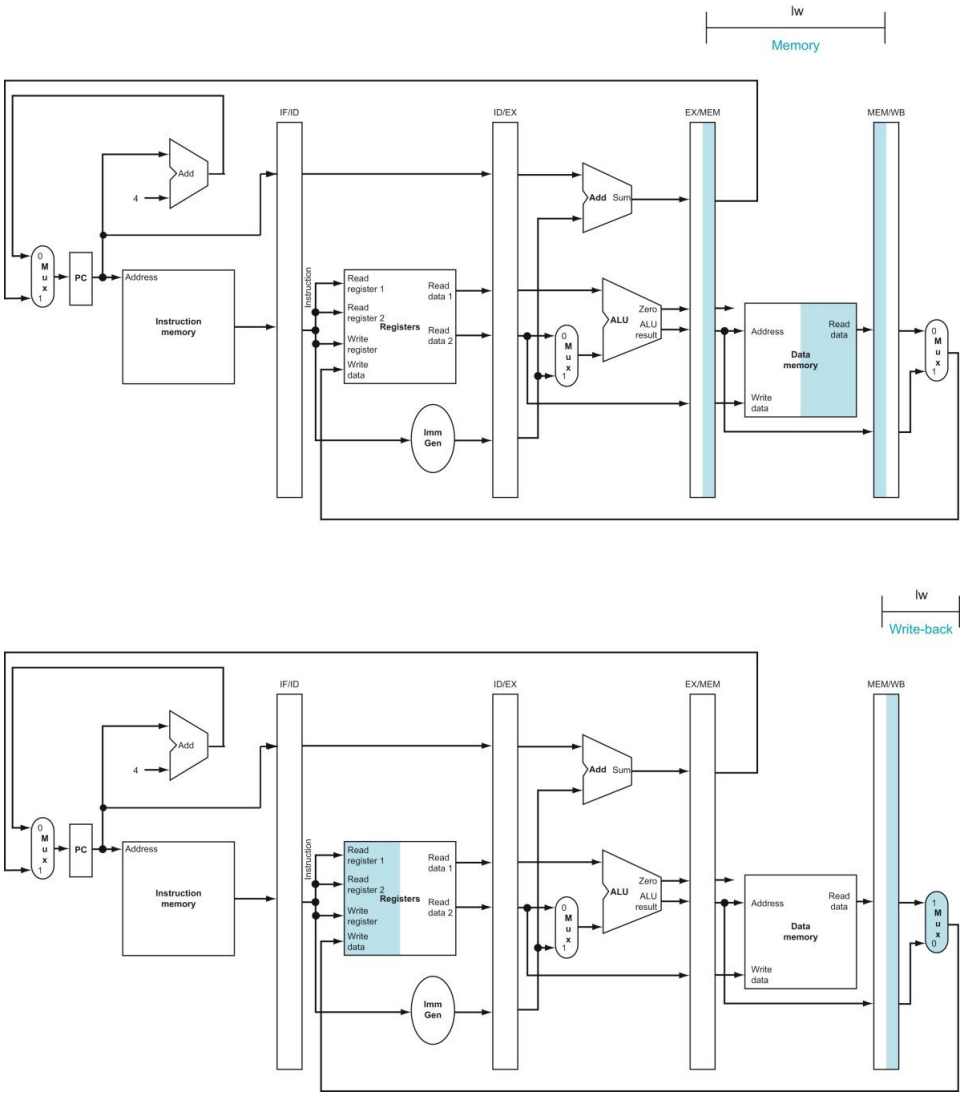


Figure 4.40 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 4.37 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data are placed in the MEM/WB pipeline register. Next, data are read from the MEM/WB pipeline register and written into the register file in the middle of the datapath. Note: there is a bug in this design that is repaired in Figure 4.43.

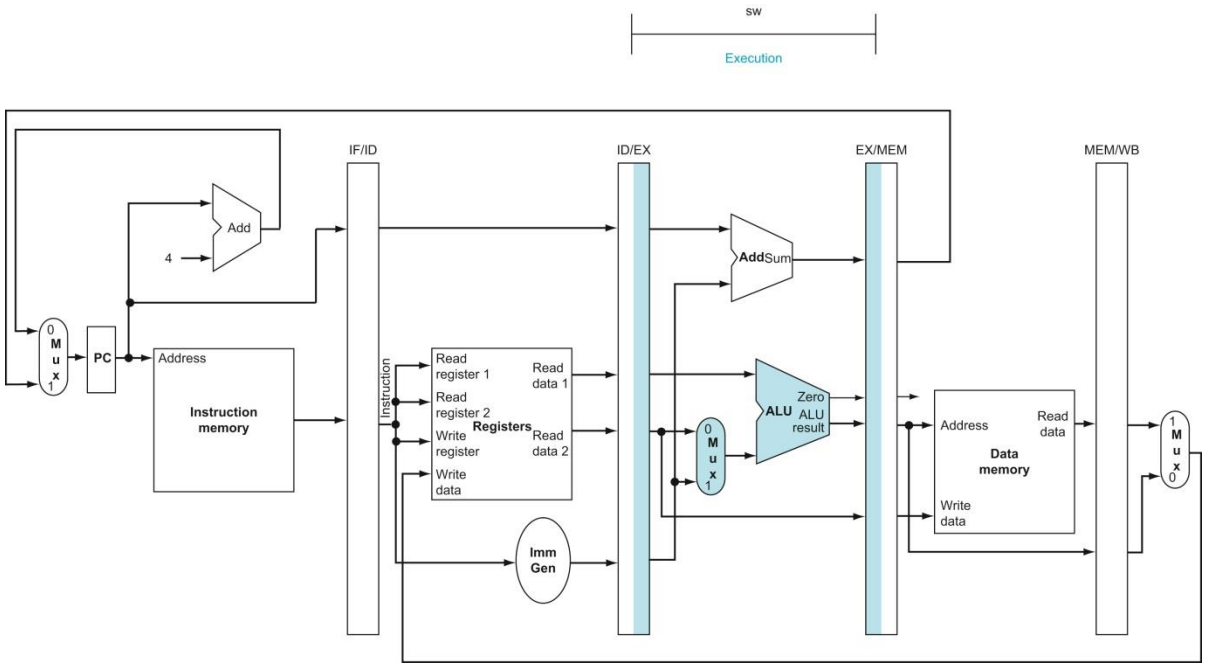


Figure 4.41 EX: The third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 4.39, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

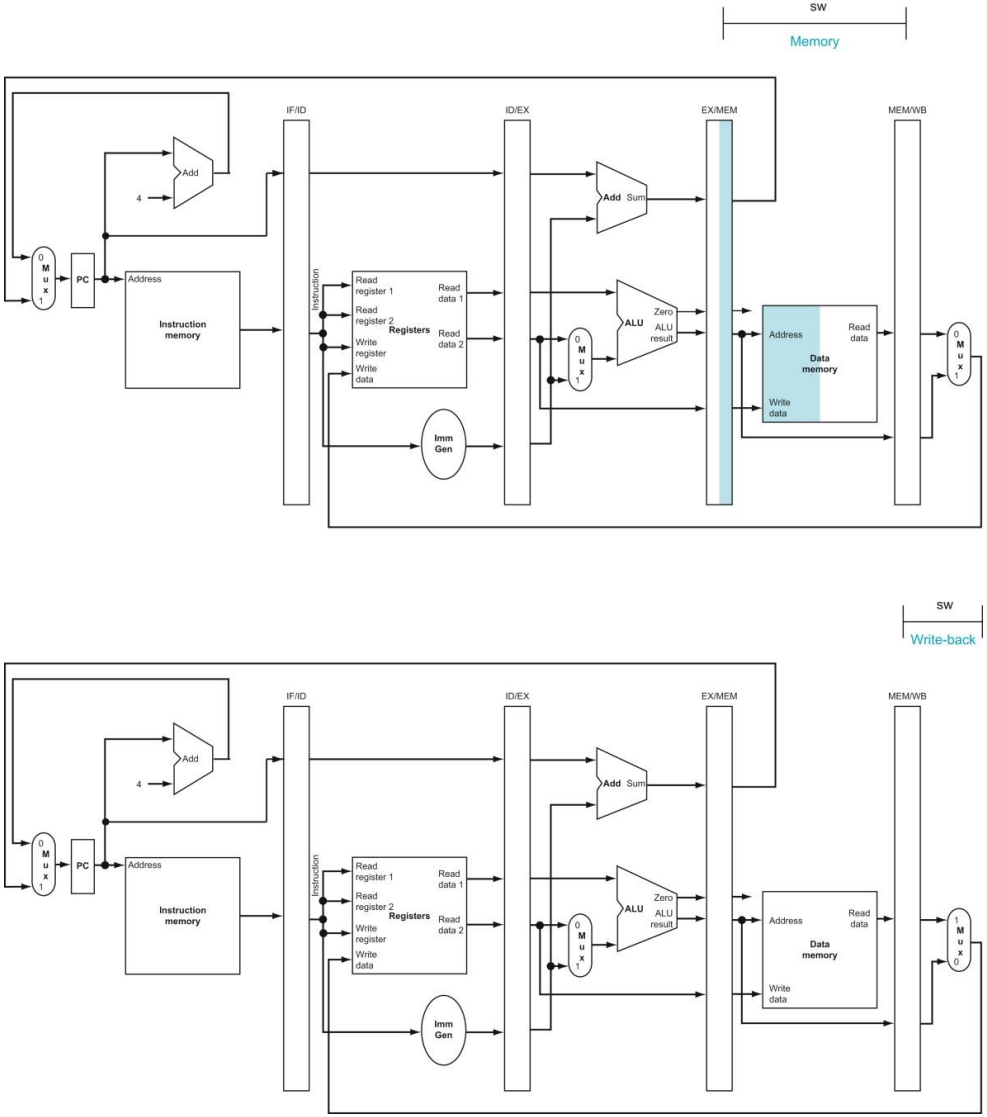


Figure 4.42 MEM and WB: The fourth and fifth pipe stages of a store instruction. In the fourth stage, the data are written into data memory for the store. Note that the data come from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data are written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

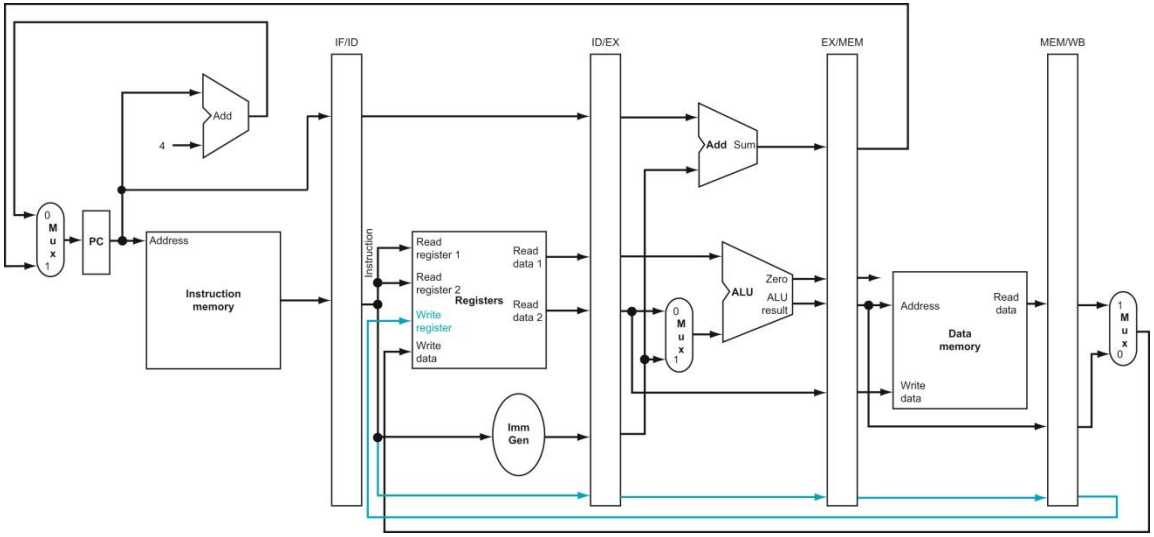


Figure 4.43 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/ WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color.

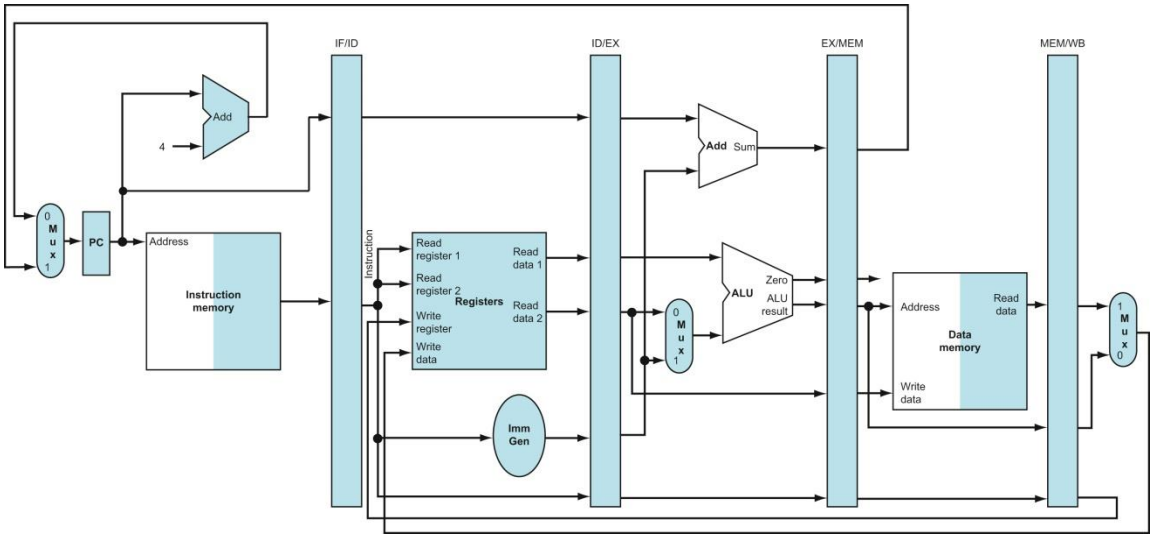


Figure 4.44 The portion of the datapath in Figure 4.43 that is used in all five stages of a load instruction.

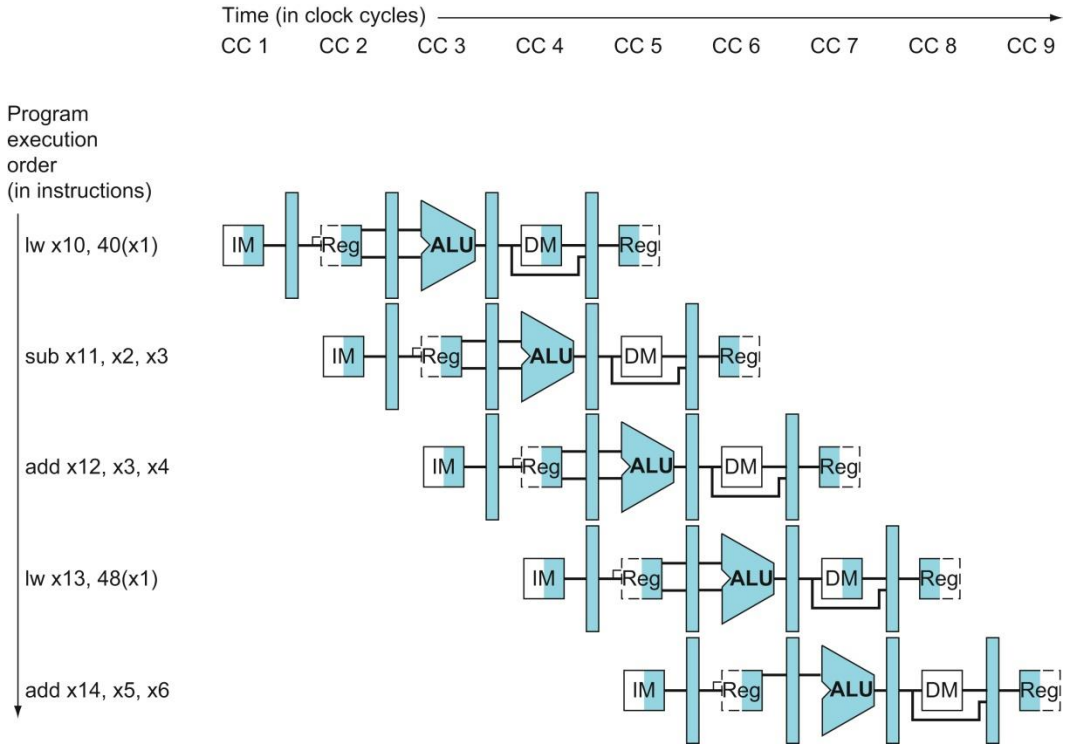


Figure 4.45 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.26, here we show the pipeline registers between each stage. Figure 4.59 shows the traditional way to draw this diagram.

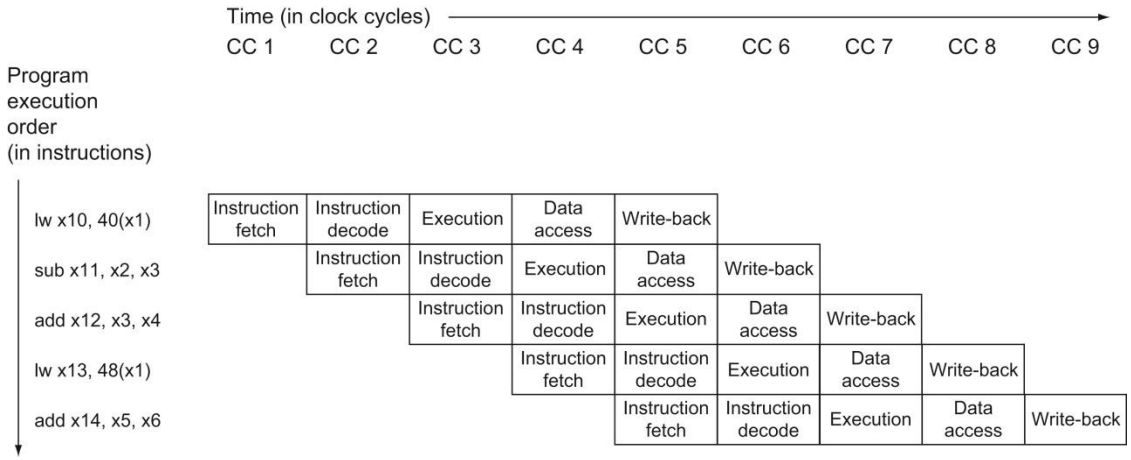


Figure 4.46 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.45.

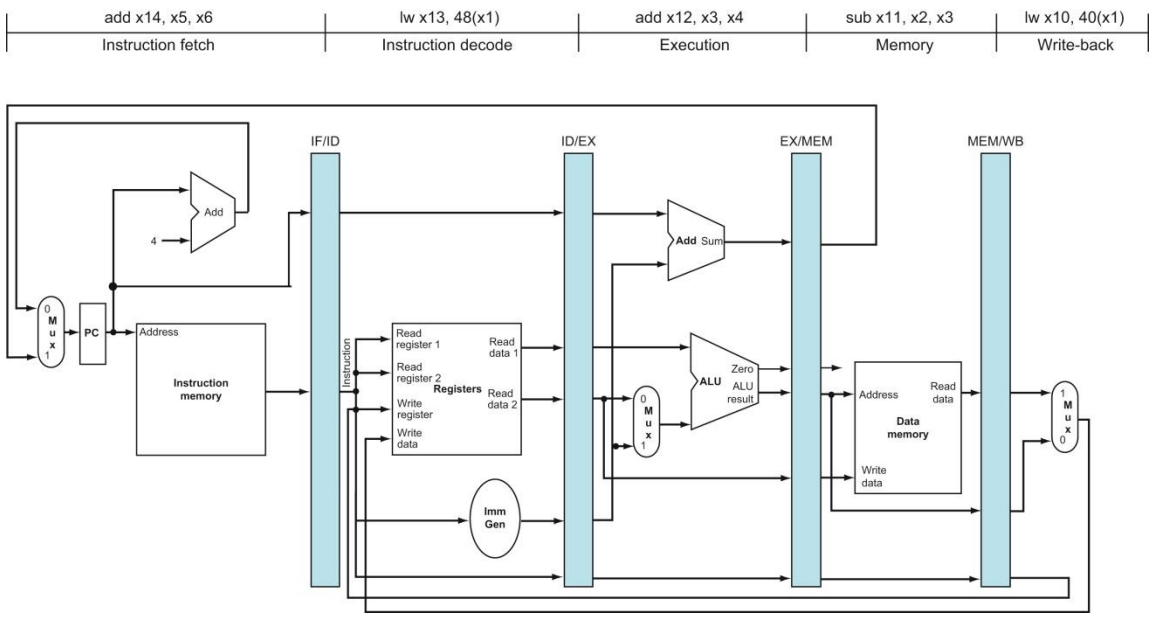


Figure 4.47 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.45 and 4.46. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

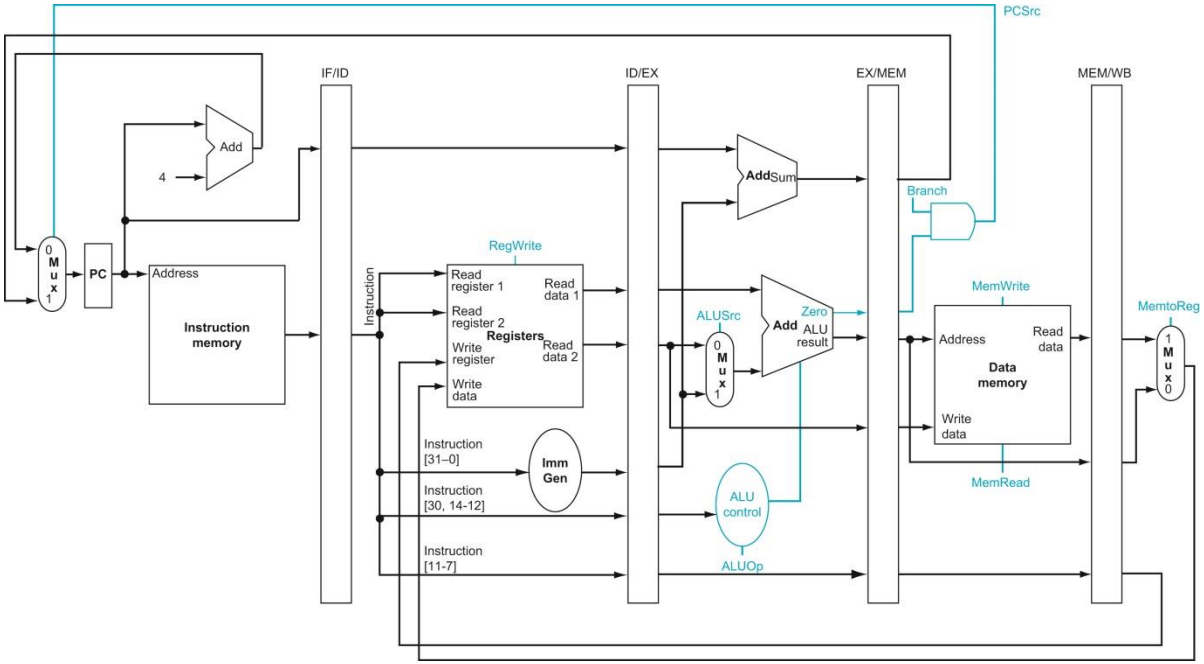


Figure 4.48 The pipelined datapath of Figure 4.43 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need funct fields of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.

Instruction	ALUOp	operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	0010
sw	00	store word	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Figure 4.49 A copy of Figure 4.12. This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different opcodes for the R-type instruction.

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 4.50 A copy of Figure 4.20. The function of each of six control signals is defined. The ALU control lines (ALUOp) are defined in the second column of Figure 4.49. When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in Figure 4.48. If the Branch signal and the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a beq instruction; otherwise, PCSrc is set to 0.

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Figure 4.51 The values of the control lines are the same as in Figure 4.22, but they have been shuffled into three groups corresponding to the last three pipeline stages.

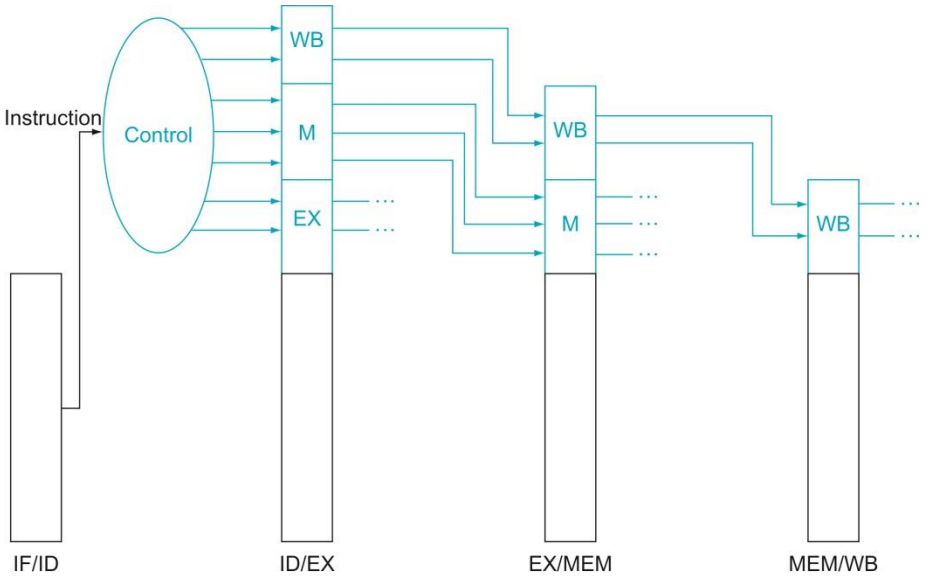


Figure 4.52 The seven control lines for the final three stages. Note that two of the seven control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

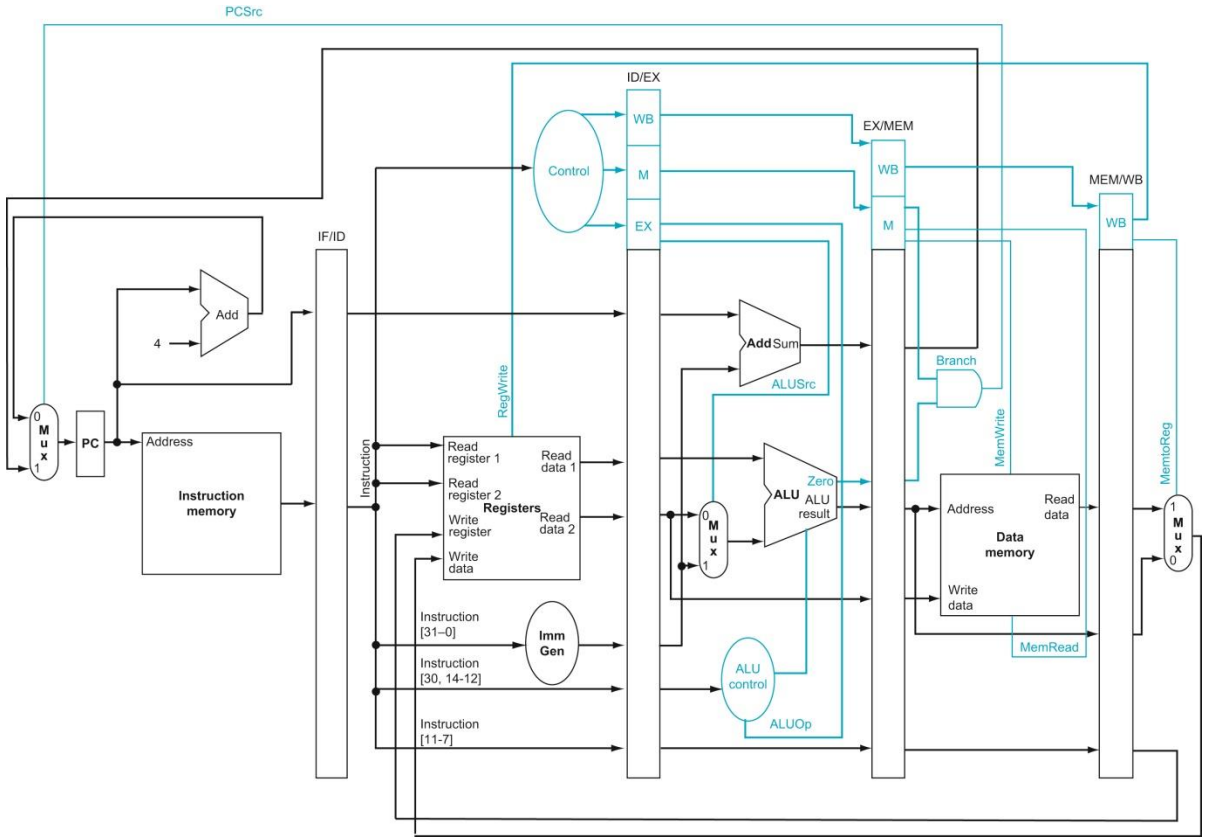


Figure 4.53 The pipelined datapath of Figure 4.48, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

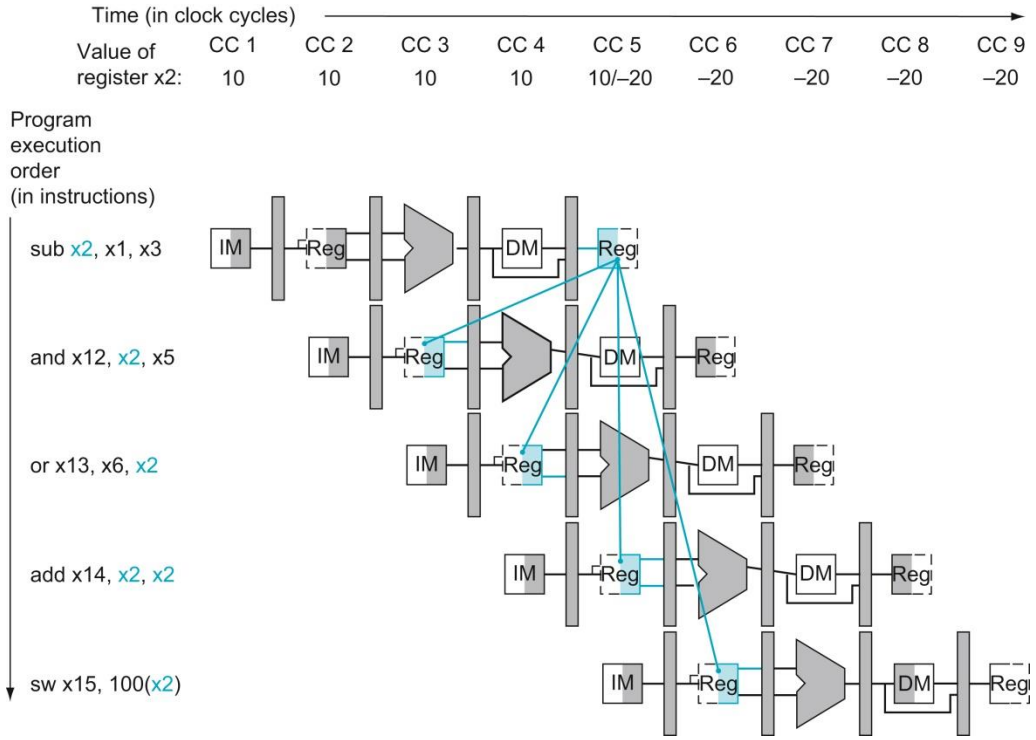


Figure 4.54 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into x2, and all the following instructions read x2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*.

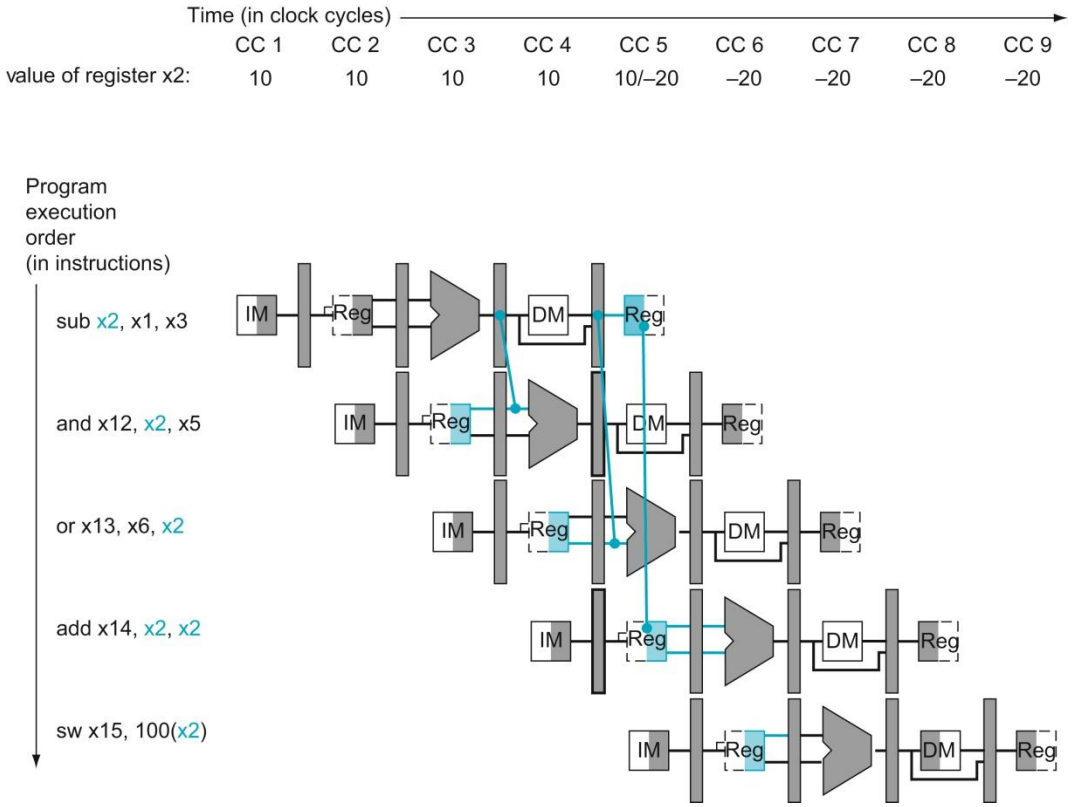


Figure 4.55 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the and instruction and or instruction by forwarding the results found in the pipeline registers. The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the add does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register x2 having the value 10 at the beginning and −20 at the end of the clock cycle.

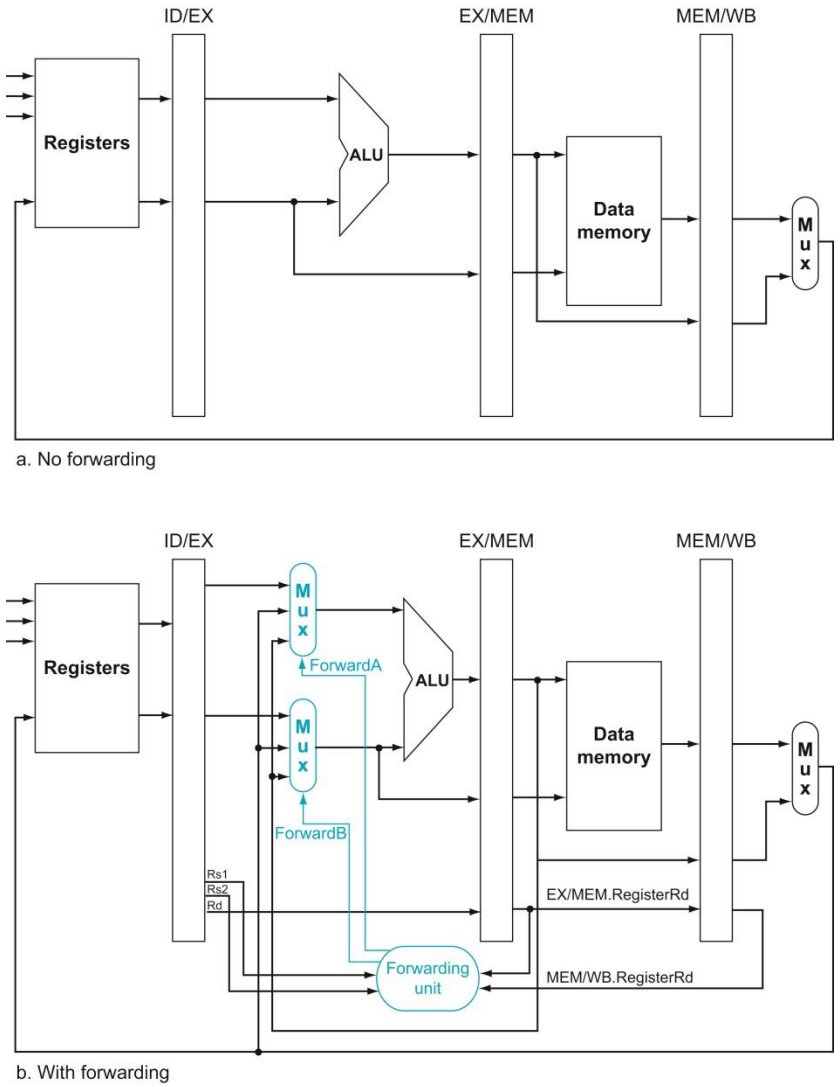


Figure 4.56 On the top are the ALU and pipeline registers before adding forwarding. On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 4.57 The control values for the forwarding multiplexors in Figure 4.56. The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

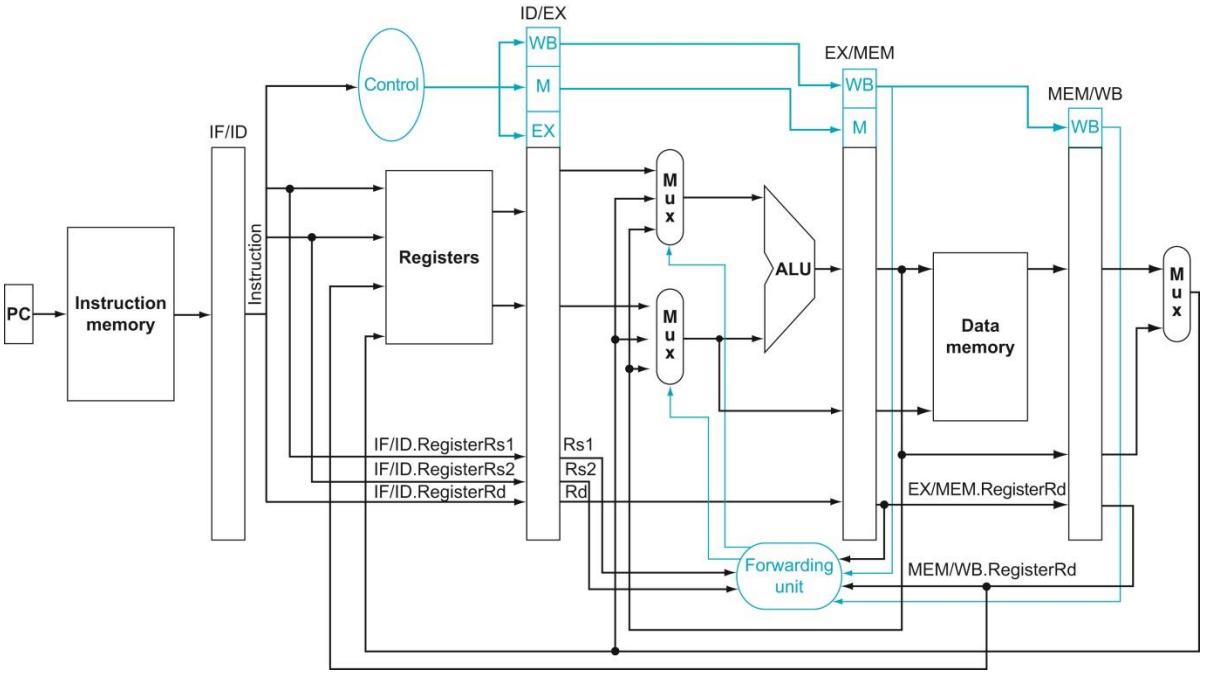


Figure 4.58 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 4.53, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.

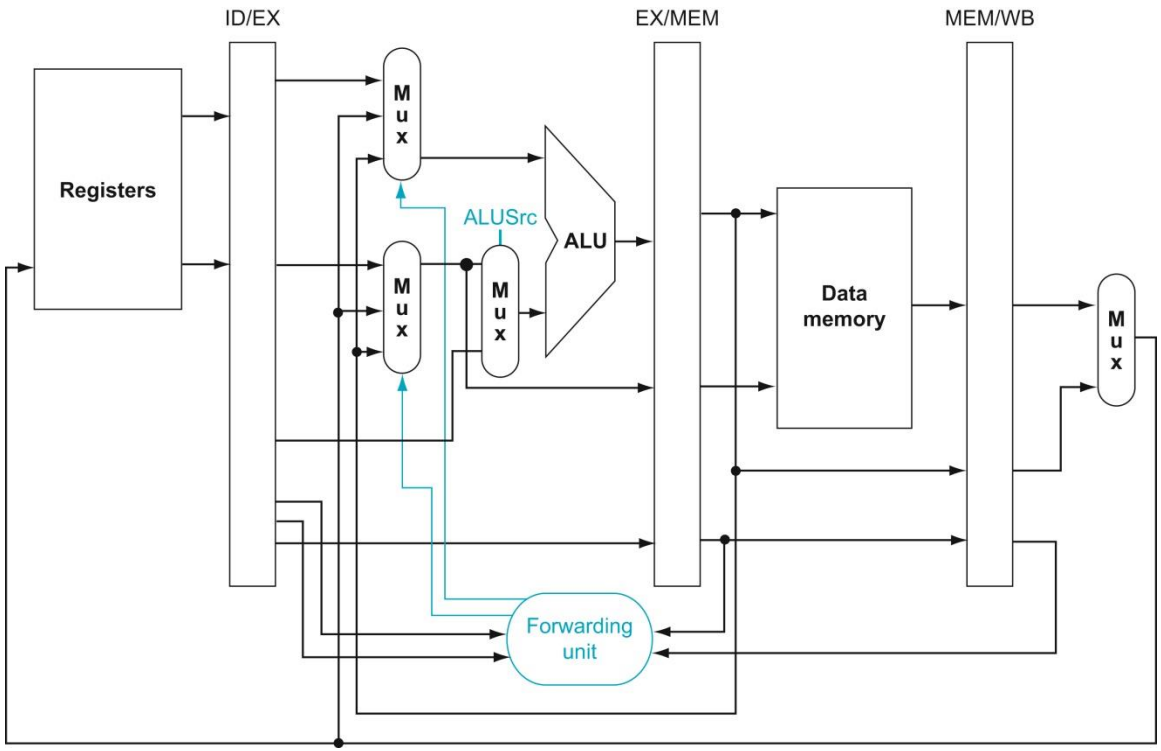


Figure 4.59 A close-up of the datapath in Figure 4.56 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.

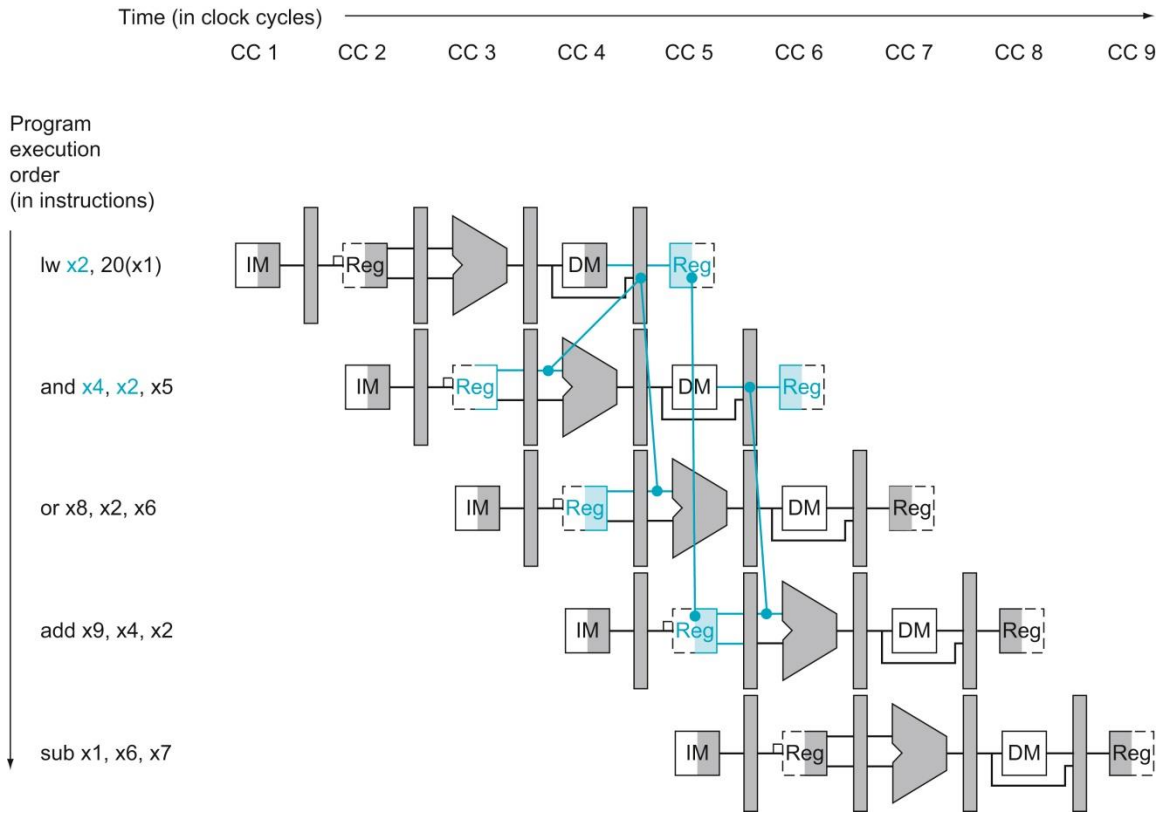


Figure 4.60 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

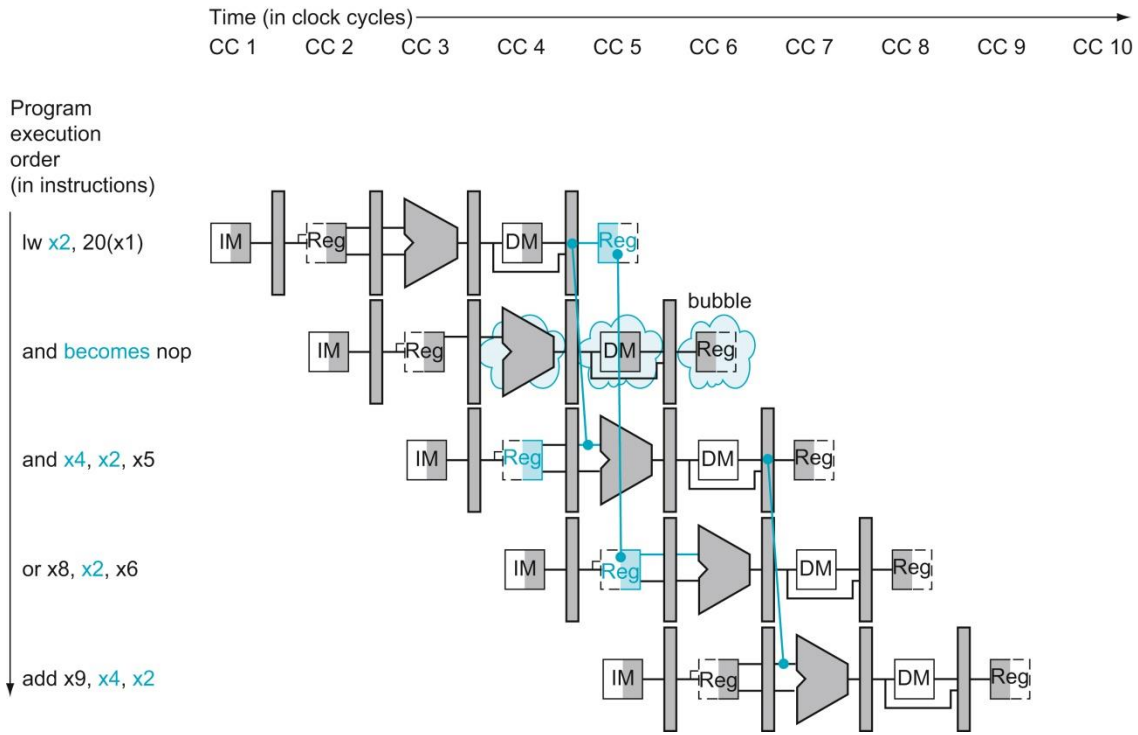


Figure 4.61 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise, the or instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

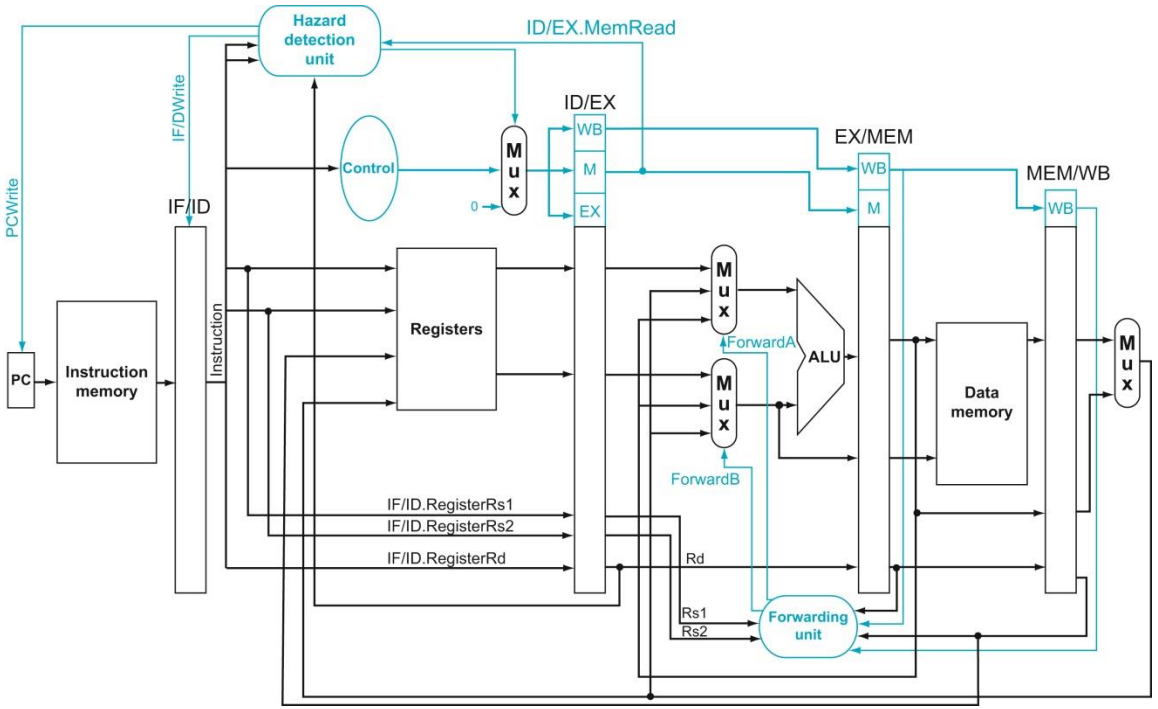


Figure 4.62 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing— this drawing gives the essence of the forwarding hardware requirements.

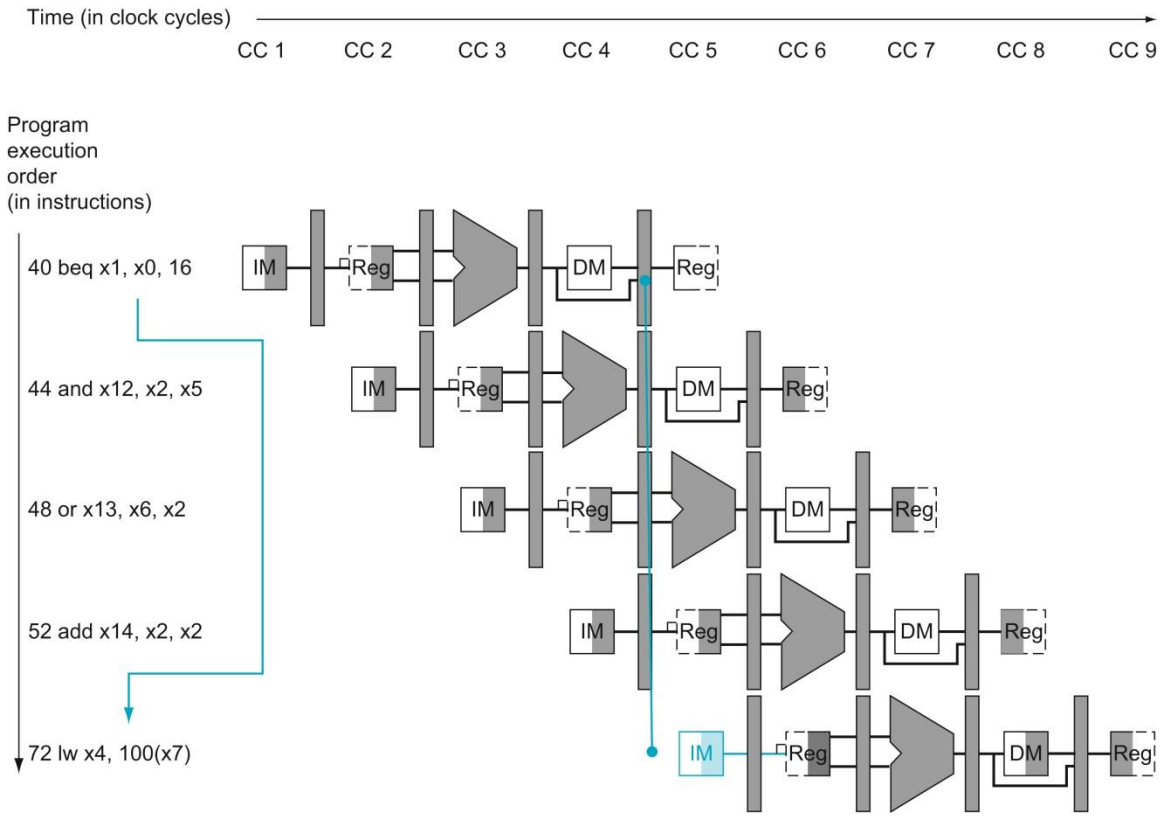


Figure 4.63 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the beq instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72. (Figure 4.33 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

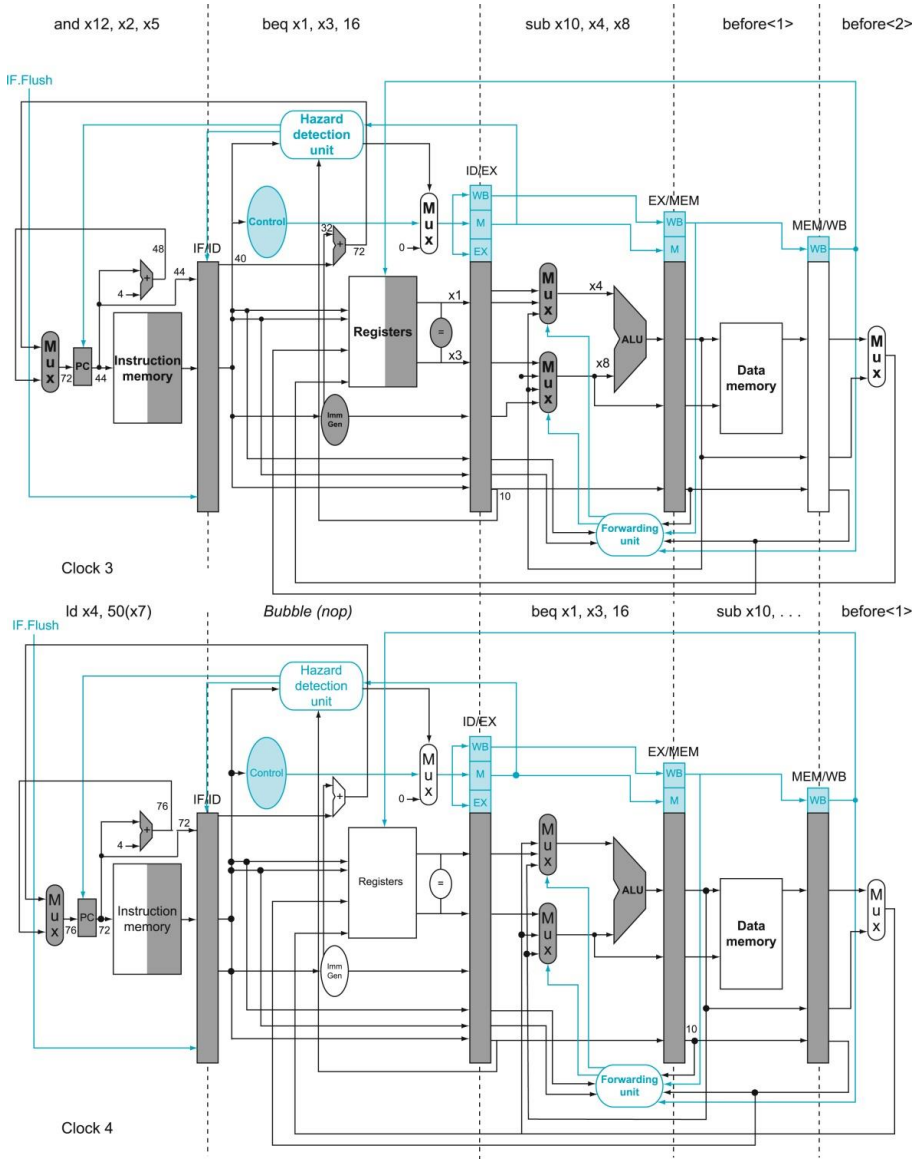


Figure 4.64 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle. Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or nop instruction in the pipeline because of the taken branch.

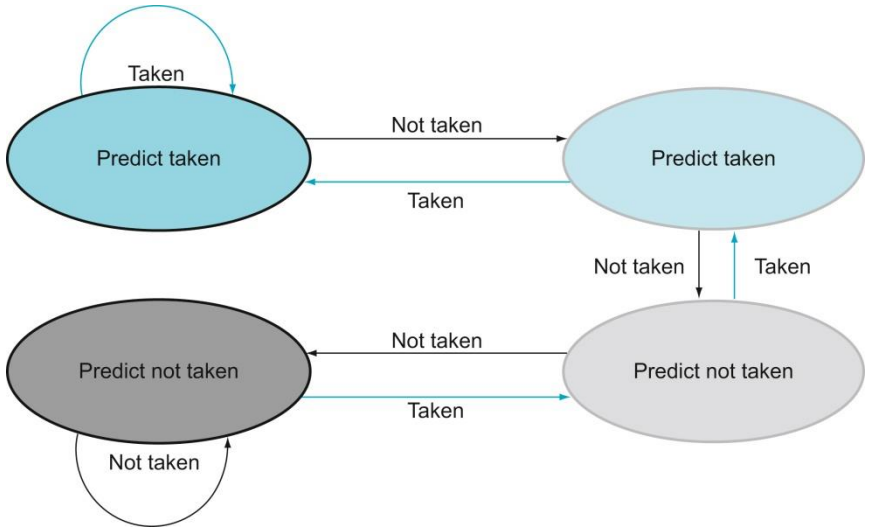


Figure 4.65 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

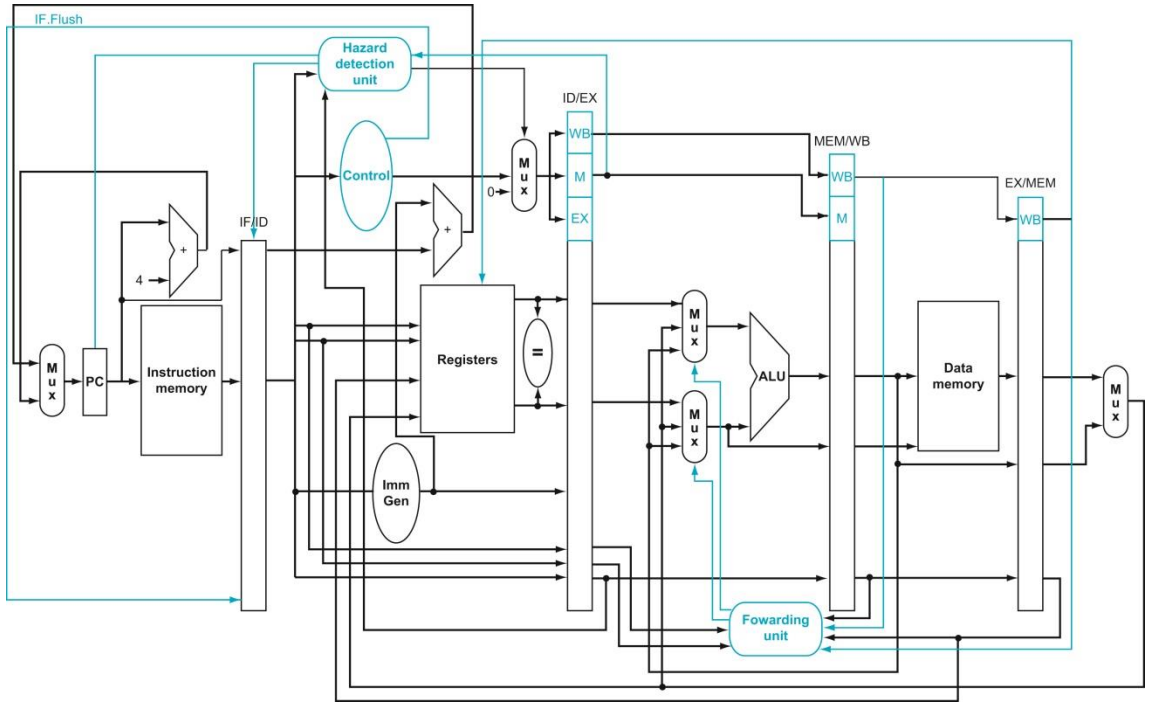


Figure 4.66 The final datapath and control for this chapter. Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from Figure 4.55 and the multiplexor controls from Figure 4.53.

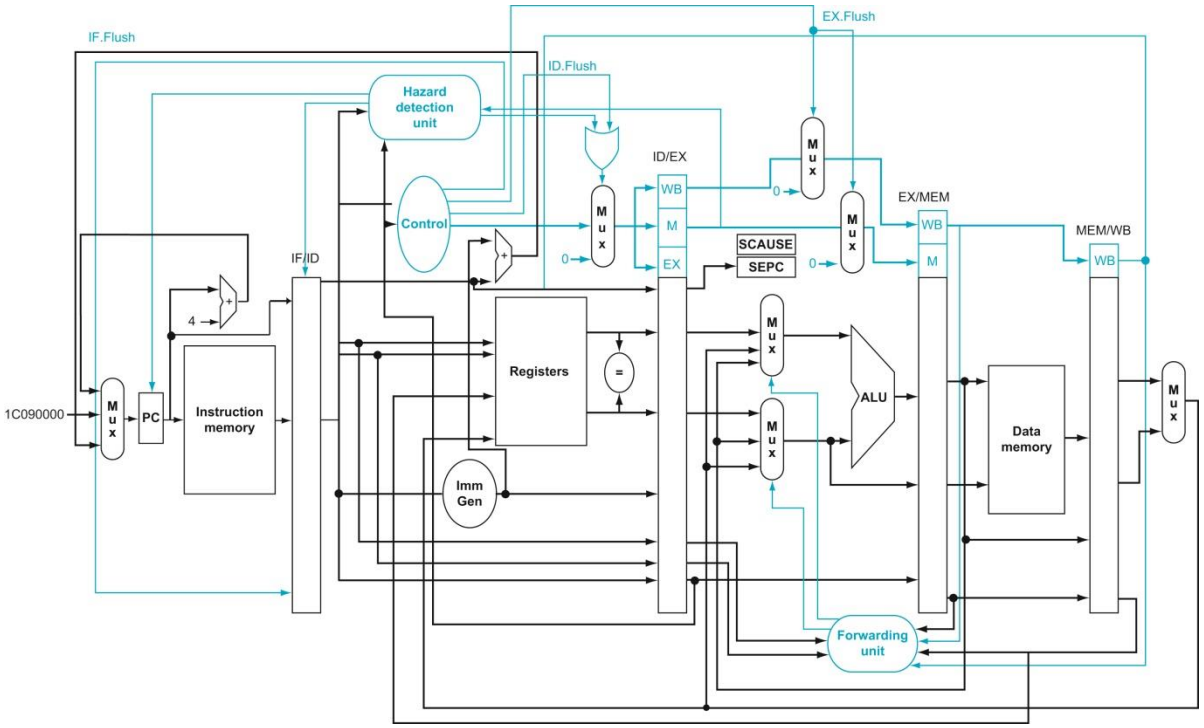


Figure 4.67 The datapath with controls to handle exceptions. The key additions include a new input with the value 0000 00001C09 0000hex in the multiplexor that supplies the new PC value; an SCAUSE register to record the cause of the exception; and an SEPC register to save the address of the instruction that caused the exception. The 0000 0000 1C09 0000hex input to the multiplexor is the initial address to begin fetching instructions in the event of an exception.

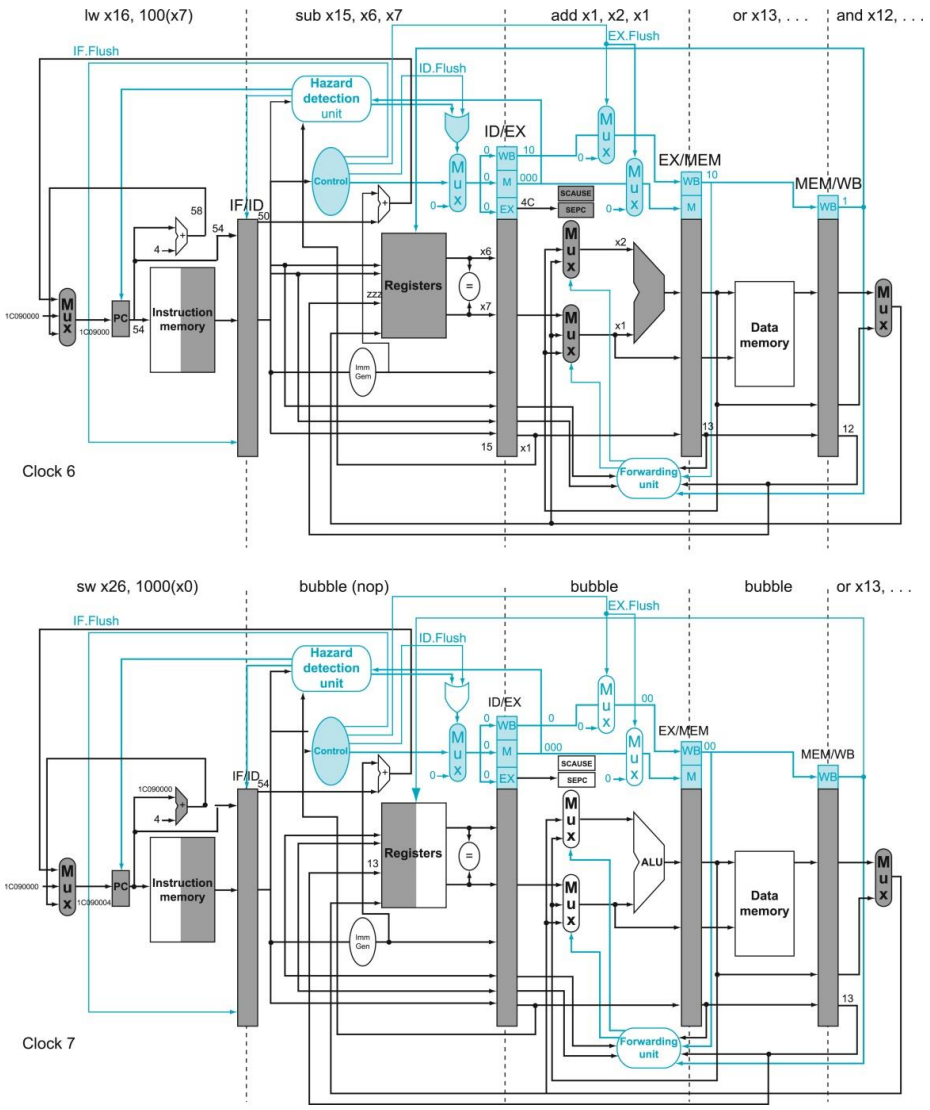


Figure 4.68 The result of an exception due to hardware malfunction in the add instruction. The exception is detected during the EX stage of clock 6, saving the address of the add instruction in the SEPC register (4Chex). It causes all the Flush signals to be set near the end of this clock cycle, deasserting control values (setting them to 0) for the add. Clock cycle 7 shows the instructions converted to bubbles in the pipeline plus the fetching of the first instruction of the exception routine—sw x26, 1000(x0)—from instruction location 0000 0000 1C09 0000hex. Note that the and and or instructions, which are prior to the add, still complete.

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

F**igure 4.69 Static two-issue pipeline in operation.** The ALU and data transfer instructions are issued at the same time. Here we have assumed the same five-stage structure as used for the single-issue pipeline. Although this is not strictly necessary, it does have some advantages. In particular, keeping the register writes at the end of the pipeline simplifies the handling of exceptions and the maintenance of a precise exception model, which become more difficult in multiple-issue processors.

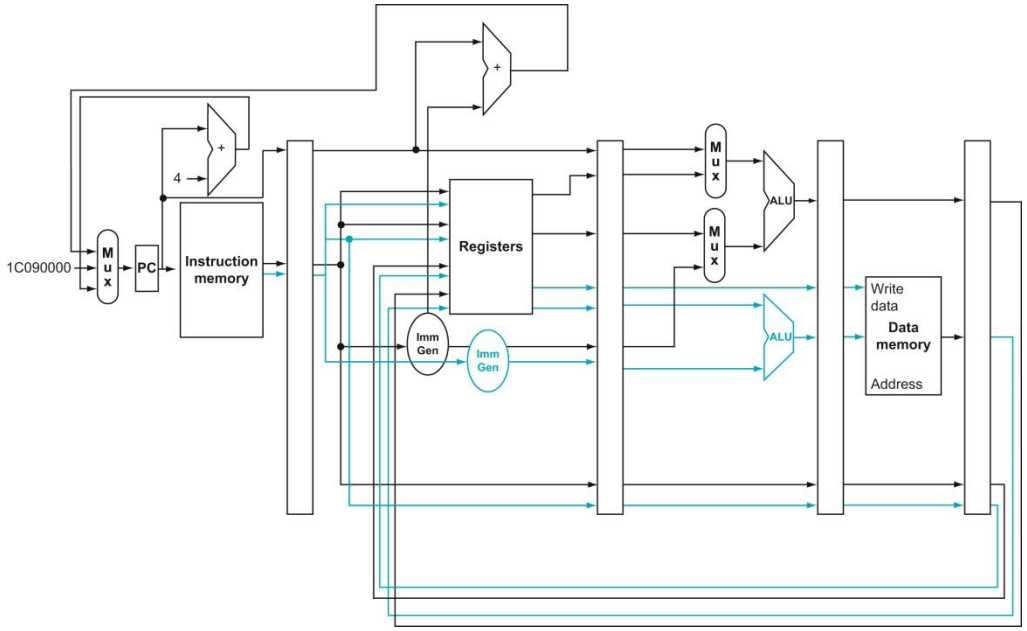


Figure 4.70 A static two-issue datapath. The additions needed for double issue are highlighted: another 32 bits from instruction memory, two more read ports and one more write port on the register file, and another ALU. Assume the bottom ALU handles address calculations for data transfers and the top ALU handles everything else.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw x31, 0(x20)	1
	addi x20, x20, -4		2
	add x31, x31, x21		3
	blt x22, x20, Loop	sw x31, 4(x20)	4

Figure 4.71 The scheduled code as it would look on a two-issue RISC-V pipeline. The empty slots are no-ops. Note that since we moved the addi before the sw, we had to adjust sw's offset by 4.

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:	addi x20, x20, -32	lw x28, 0(x20)	1
		lw x29, 12(x20)	2
	add x28, x28, x21	lw x30, 8(x20)	3
	add x29, x29, x21	lw x31, 4(x20)	4
	add x30, x30, x21	sw x28, 16(x20)	5
	add x31, x31, x21	sw x29, 12(x20)	6
		sw x30, 8(x20)	7
	blt x22, x20, Loop	sw x31, 4(x20)	8

Figure 4.72 The unrolled and scheduled code of Figure 4.71 as it would look on a static two-issue RISC-V pipeline. The empty slots are no-ops. Since the first instruction in the loop decrements x20 by 16, the addresses loaded are the original value of x20, then that address minus 4, minus 8, and minus 12.

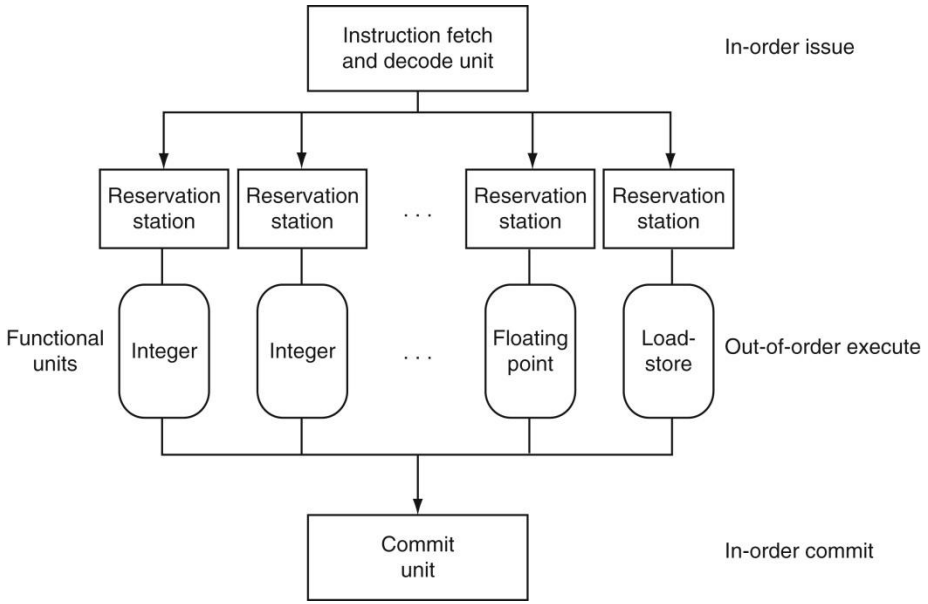


Figure 4.73 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103W
Intel Core	2006	3000 MHz	14	4	Yes	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Yes	2-4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Yes	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Yes	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Yes	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Yes	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Yes	16	185W

Figure 4.74 Record of Intel Microprocessors in terms of pipeline complexity, number of cores, and power. The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper.

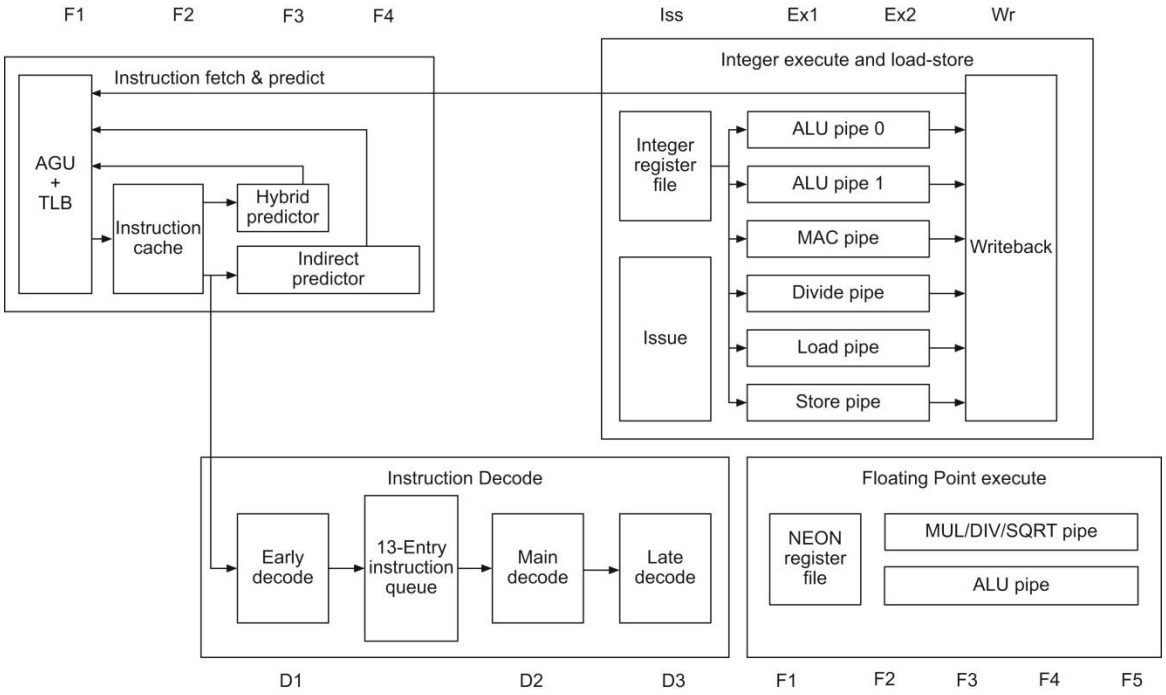


Figure 4.75 The basic structure of the A53 integer pipeline has eight stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors depending on type. The floating-point execution pipeline is 5 cycles deep in addition to the 5 cycles needed for fetch and decode, yielding 10 stages total. AGU stands for address generation unit and TLB for transaction lookaside buffer (See Chapter 5). The NEON unit performs the ARM SIMD instructions of the same name. *(From Hennessy JL, Patterson DA:* Computer architecture: A quantitative approach*, 6e, Cambridge MA, 2018, Morgan Kaufmann.)*

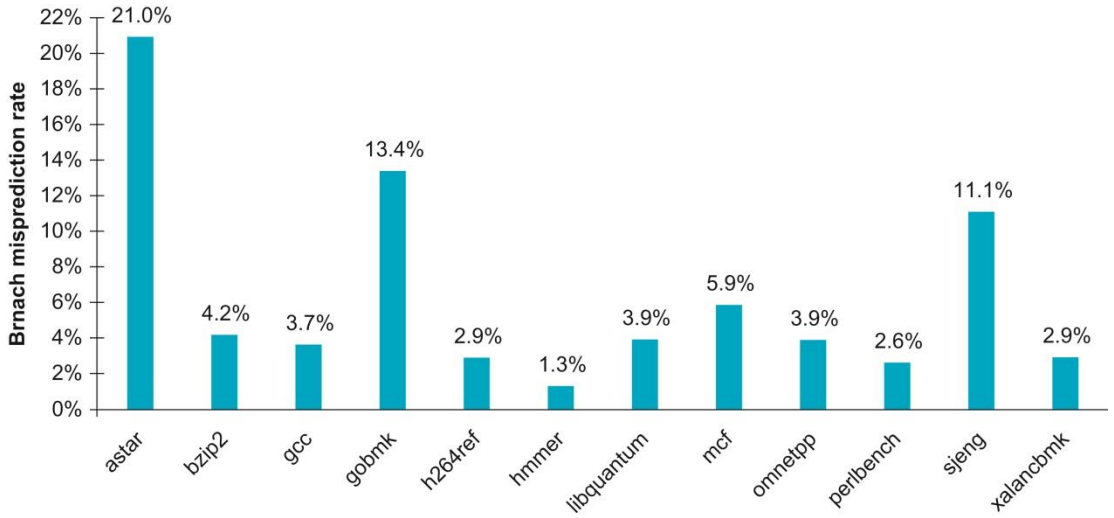


Figure 4.76 Misprediction rate of the A53 branch predictor for SPECint2006. *(Adapted from Hennessy JL, Patterson DA:* Computer architecture: A quantitative approach*, 6e, Cambridge MA, 2018, Morgan Kaufmann.)*

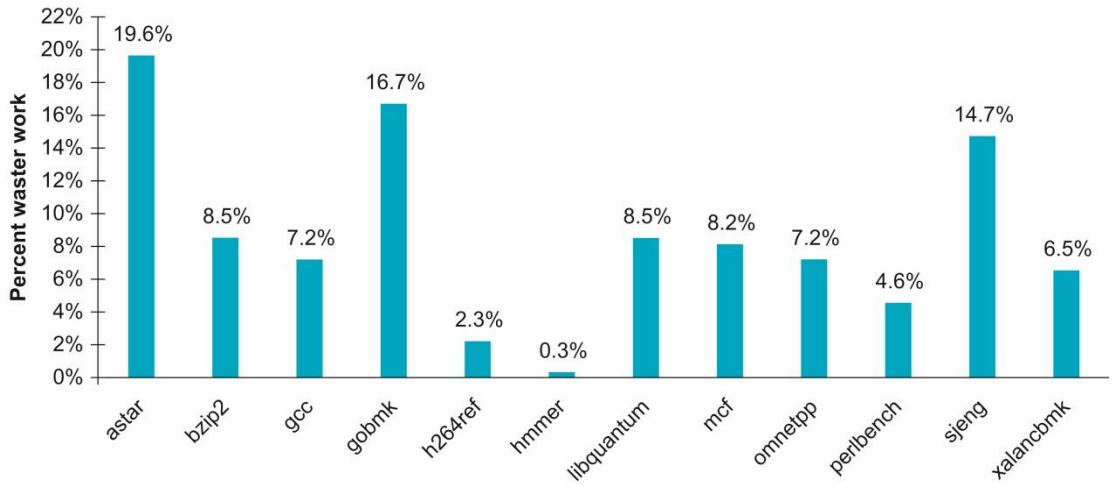


Figure 4.77 Wasted work due to branch misprediction on the A53. Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors including data dependences and cache misses, both of which will cause a stall. *(Adapted from Hennessy JL, Patterson DA:* Computer architecture: A quantitative approach*, 6e, Cambridge MA, 2018, Morgan Kaufmann.)*

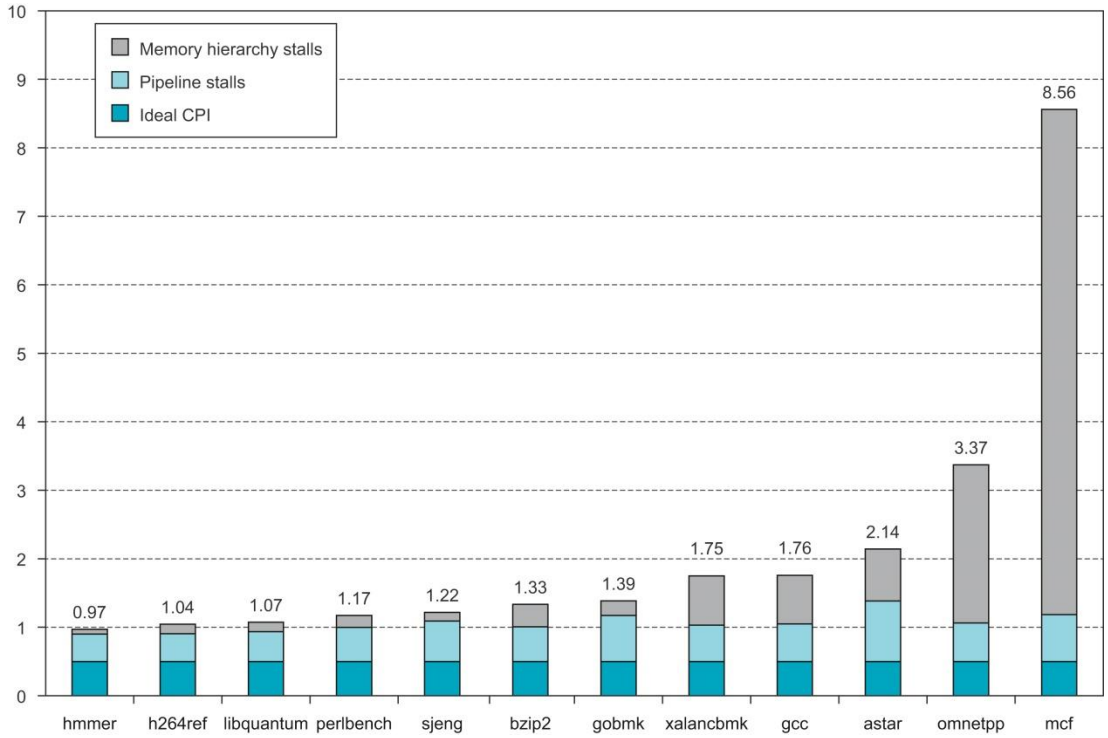


Figure 4.78 The estimated composition of the CPI on the ARM A53 shows that pipeline stalls are significant but outweighed by cache misses in the poorest-performing programs (Chapter 5). These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards. *(From Hennessy JL, Patterson DA:* Computer architecture: A quantitative approach*, 6e, Cambridge MA, 2018, Morgan Kaufmann.)*

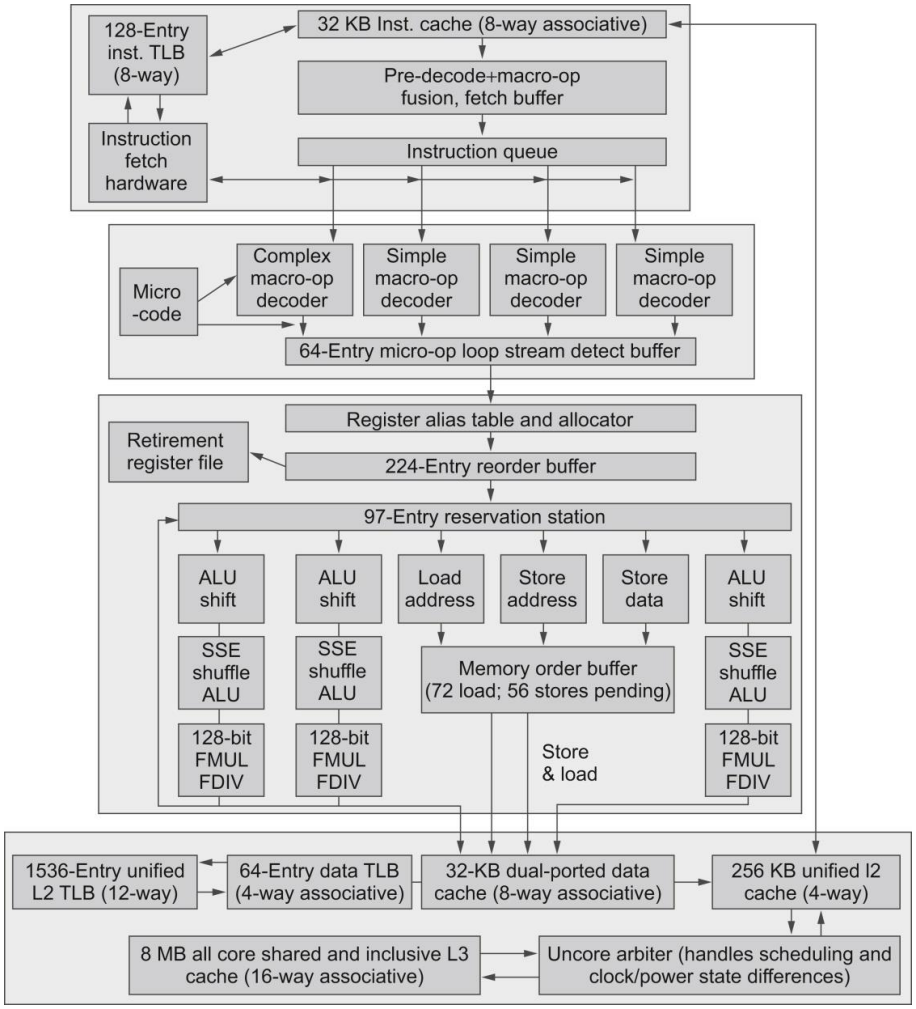


Figure 4.79 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions typically costing 17 cycles and the extra few cycles likely due to time to reset the branch predictor. This design can buffer 72 loads and 56 stores. The six independent functional units can each begin execution of a ready micro-operation in the same cycle. Up to four micro-operations can be processed in the register-renaming table. The first i7 processor was introduced in 2008; the i7 6700 is the sixth generation. The basic structure of the i7 is similar, but successive generations have enhanced performance by changing cache strategies (Chapter 5), increasing memory bandwidth, expanding the number of instructions in flight, enhancing branch prediction, and improving graphics support. *(From Hennessy JL, Patterson DA:* Computer architecture: A quantitative approach*, 6e, Cambridge MA, 2018, Morgan Kaufmann.)*

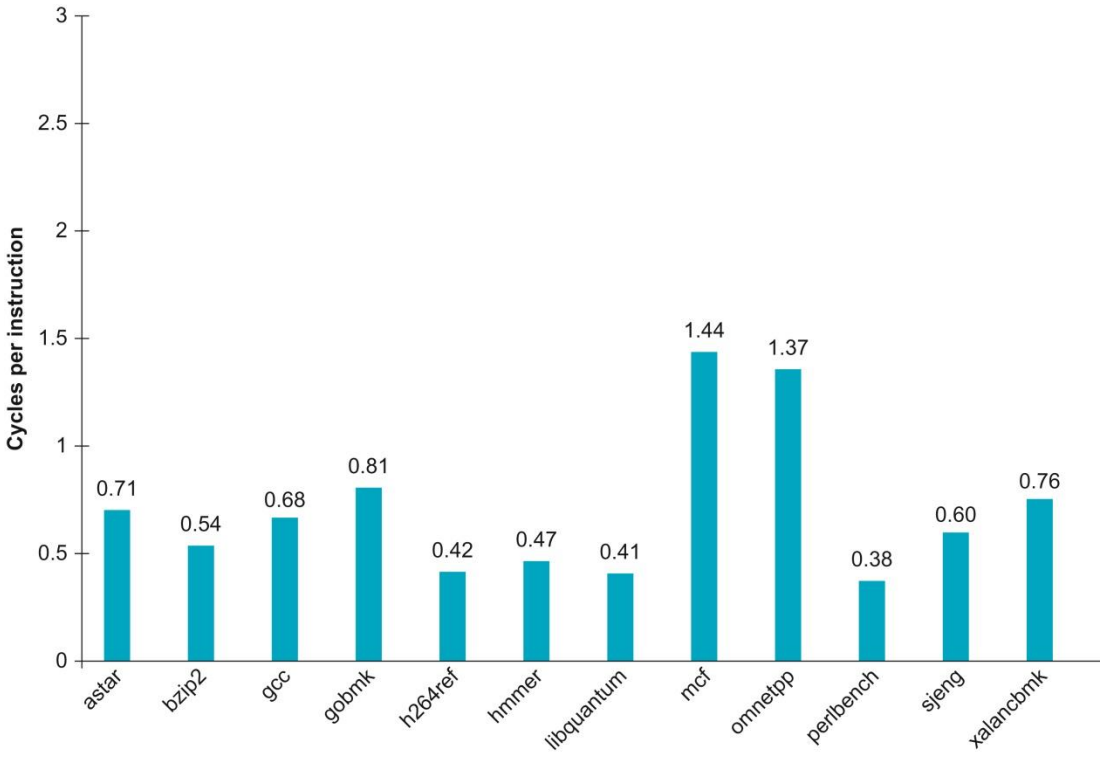


Figure 4.80 The CPI for the SPECCPUint2006 benchmarks on the i7 6700. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University. *(Adapted from Hennessy JL, Patterson DA:* Computer architecture: A quantitative approach*, 6e, Cambridge MA, 2018, Morgan Kaufmann.)*

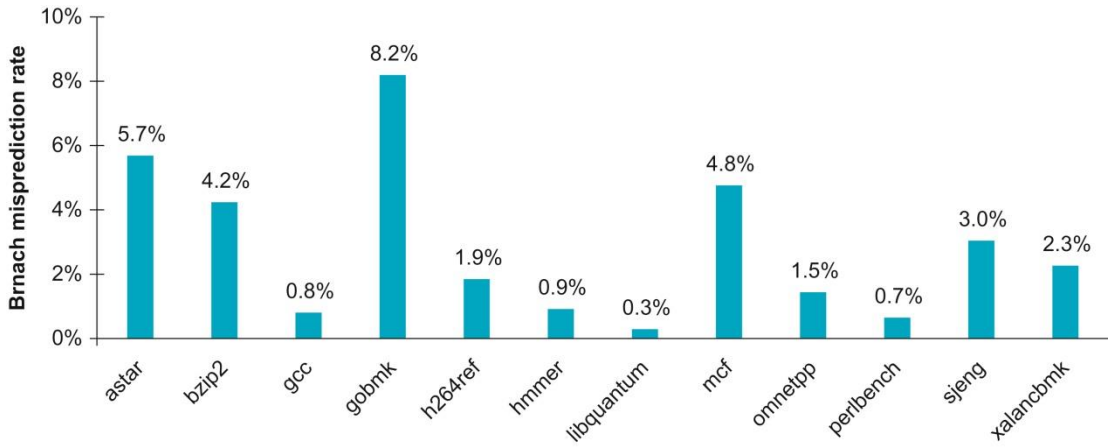


Figure 4.81 The misprediction rate for the integer SPECCPU2006 benchmarks on the Intel Core i7 6700. The misprediction rate is computed as the ratio of completed branches that are mispredicted versus all completed branches. *(Adapted from Hennessy JL, Patterson DA:* Computer architecture: A quantitative approach, *6e, Cambridge MA, 2018, Morgan Kaufmann.)*

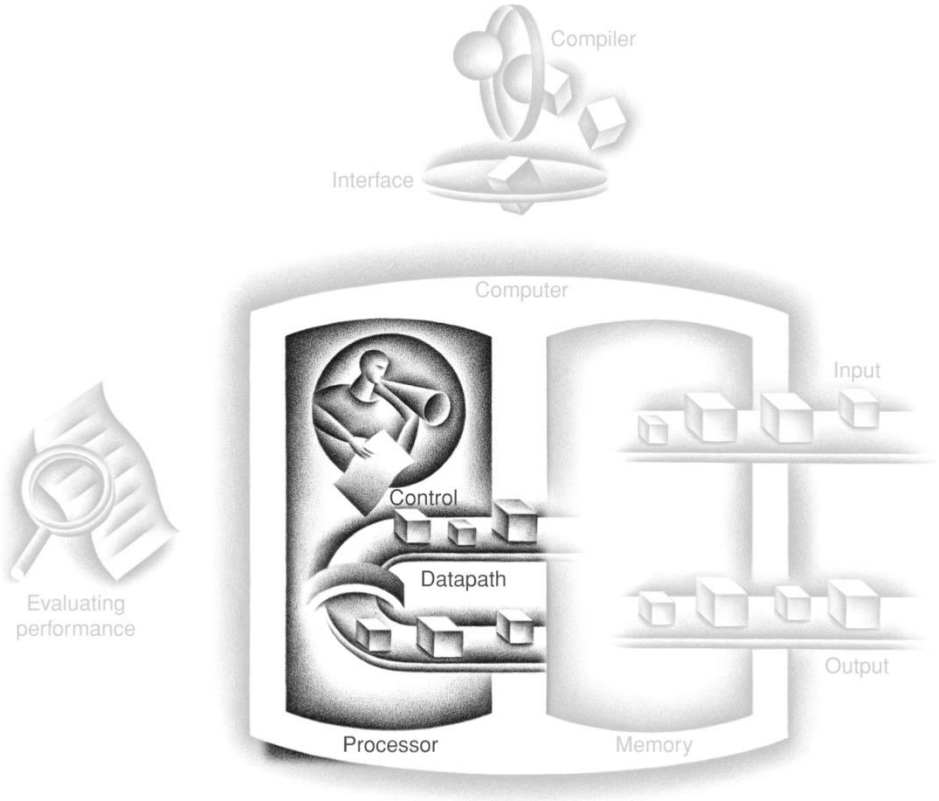
```
1.  #include <x86intrin.h>
2.  #define UNROLL (4)
3.
4.  void dgemm (int n, double* A, double* B, double* C)
5.  {
6.    for (int i = 0; i < n; i+=UNROLL*8)
7.      for (int j = 0; j < n; ++j){
8.        __m512d c[UNROLL];
9.        for (int r=0;r<UNROLL;r++)
10.         c[r] =  _mm512_load_pd(C+i+r*8+j*n); //[ UNROLL];
11.
12.       for( int k = 0; k < n; k++ )
13.       {
14.         __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B+j*n+k));
15.         for (int r=0;r<UNROLL;r++)
16.           c[r] = _mm512_fmadd_pd(_mm512_load_pd(A+n*k+r*8+i), bb, c[r]);
17.       }
18.
19.       for (int r=0;r<UNROLL;r++)
20.         _mm512_store_pd(C+i+r*8+j*n, c[r]);
21.       }
22.   }
```

Figure 4.82 Optimized C version of DGEMM using C intrinsics to generate the AVX subword-parallel instructions for the x86 (Figure 3.20) and loop unrolling to create more opportunities for instruction-level parallelism. Figure 4.96 shows the assembly language produced by the compiler for the inner loop, which unrolls the three for-loop bodies to expose instruction-level parallelism.

```
 1  .   vmovapd      (%r11),%zmm4          # Load 8 elements of C into %zmm4
 2  .   mov          %rbx,%rcx             # register %rcx = %rbx
 3  .   xor          %eax,%eax             # register %eax = 0
 4  .   vmovapd      0x20(%r11),%zmm3      # Load 8 elements of C into %zmm3
 5  .   vmovapd      0x40(%r11),%zmm2      # Load 8 elements of C into %zmm2
 6  .   vmovapd      0x60(%r11),%zmm1      # Load 8 elements of C into %zmm1
 7  .   vbroadcastsd (%rax,%r8,8),%zmm0    # Make 8 copies of B element in %zmm0

                                           # register %rax = %rax + 8
 8  .   add          $0x8,%rax
 9  .   vfmadd231pd  (%rcx),%zmm0,%zmm4    # Parallel mul & add %zmm0, %zmm4
10  .   vfmadd231pd  0x20(%rcx),%zmm0,%zmm3 # Parallel mul & add %zmm0, %zmm3
11  .   vfmadd231pd  0x40(%rcx),%zmm0,%zmm2 # Parallel mul & add %zmm0, %zmm2
12  .   vfmadd231pd  0x60(%rcx),%zmm0,%zmm1 # Parallel mul & add %zmm0, %zmm1
13  .   add          %r9,%rcx              # register %rcx = %rcx
14  .   cmp          %r10,%rax             # compare %r10 to %rax
15  .   jne          50 <dgemm+0x50>       # jump if not %r10 != %rax
16  .   add          $0x1, %esi            # register % esi = % esi + 1
17  .   vmovapd      %zmm4, (%r11)         # Store %zmm4 into 8 C elements
18  .   vmovapd      %zmm3, 0x20(%r11)     # Store %zmm3 into 8 C elements
19  .   vmovapd      %zmm2, 0x40(%r11)     # Store %zmm2 into 8 C elements
20  .   vmovapd      %zmm1, 0x60(%r11)     # Store %zmm1 into 8 C elements
```

Figure 4.83 The x86 assembly language for the body of the nested loops generated by compiling the unrolled C code in Figure 4.82.



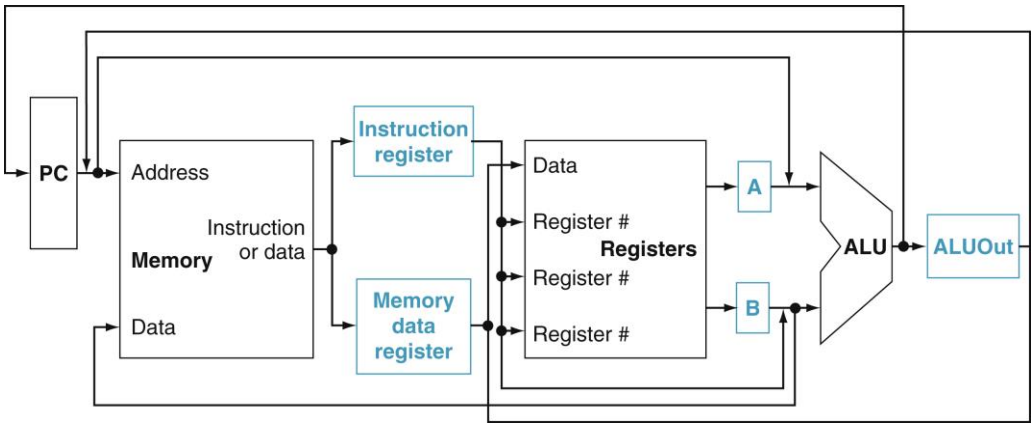


FIGURE e4.5.1 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

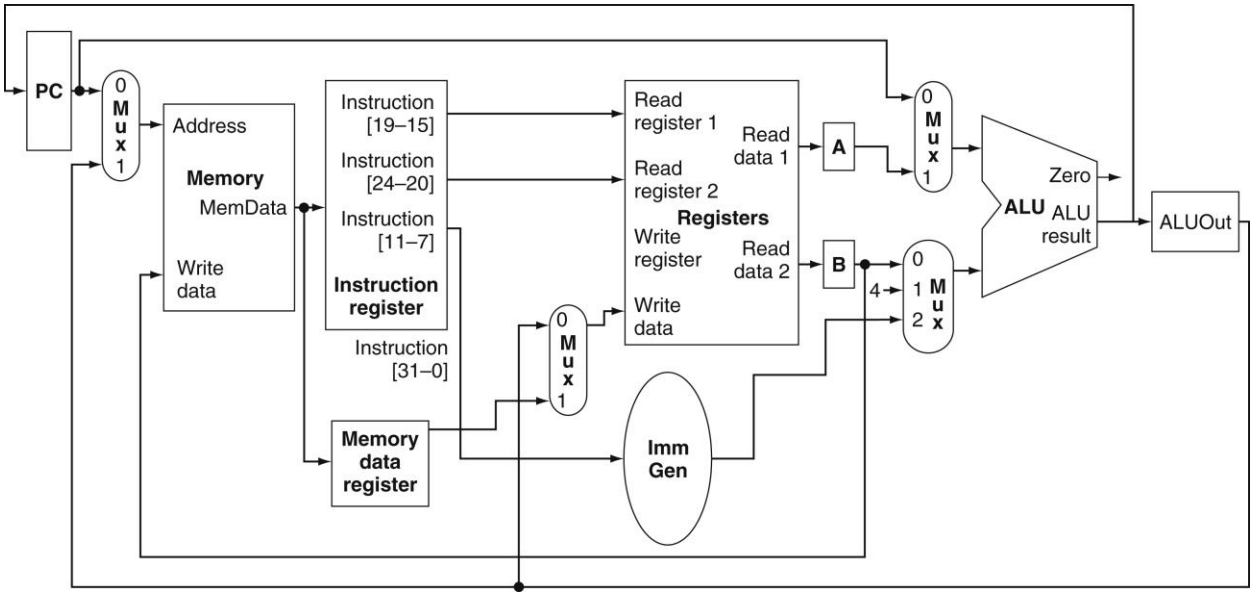


FIGURE e4.5.2 Multicycle datapath for RISC-V handles the basic instructions. Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexor will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

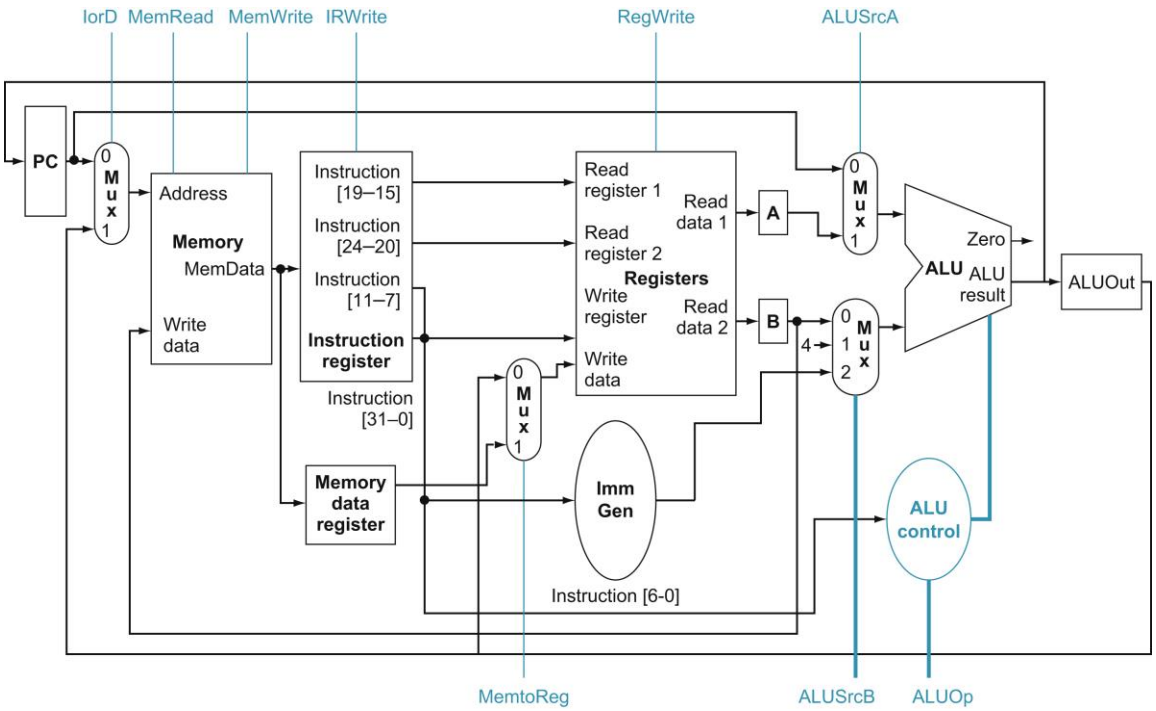


FIGURE e4.5.3 The multicycle datapath from Figure 4.28 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

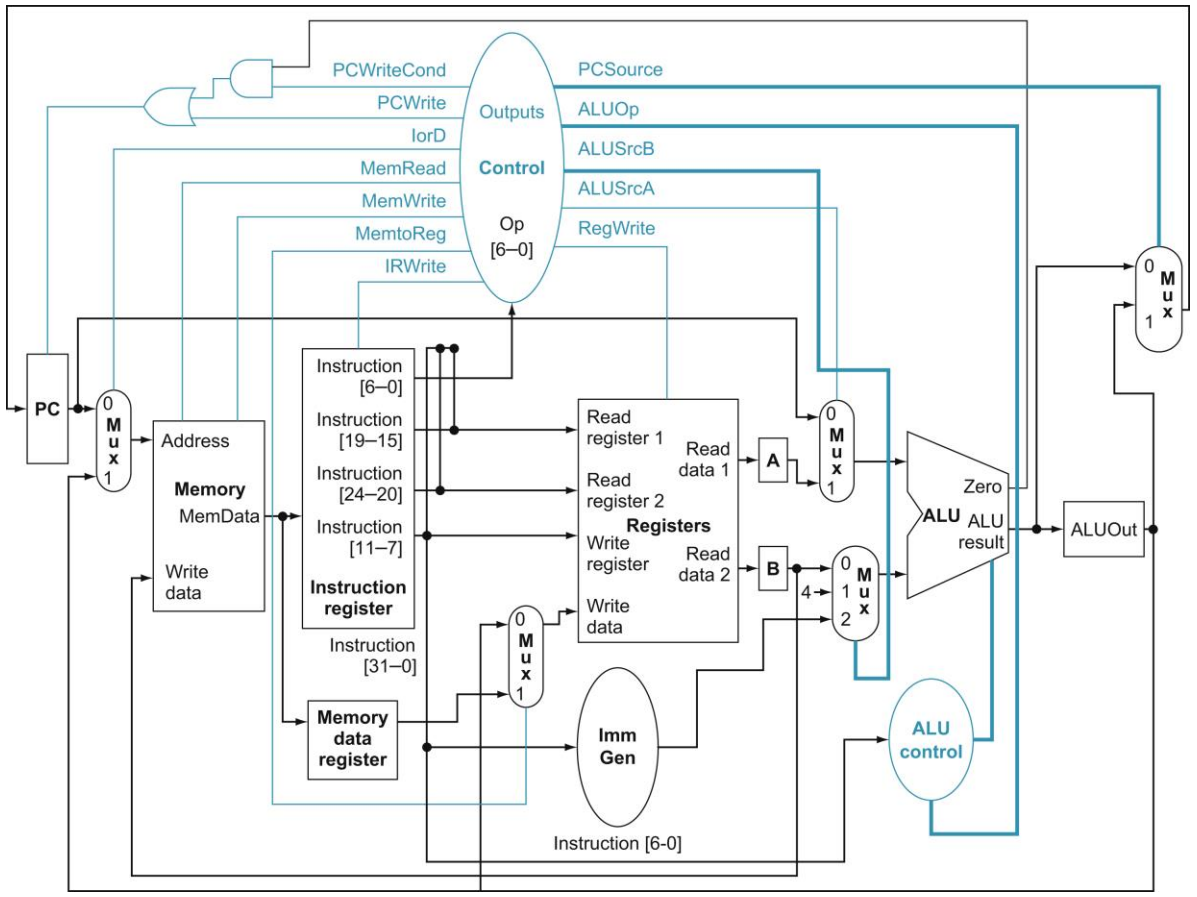


FIGURE e4.5.4 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure e4.5.3 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 4.29 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken.

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by the value on the Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the immediate generated from the IR.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

FIGURE e4.5.5 The action caused by the setting of each control signal in Figure e4.5.4 on page 323. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSource) and removes control lines that are no longer used or have been replaced (PCSrc and Branch).

Step name	Action for R-type instructions	Action for memory reference instructions	Action for branches
Instruction fetch	IR <= Memory[PC] PC <= PC + 4		
Instruction decode/register fetch	A<= Reg [IR[19:15]] B <= Reg [IR[24:20]] ALUOut <= PC + immediate		
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + immediate	if (A == B) PC <= ALUOut
Memory access or R-type completion	Reg [IR[11:7]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B	
Memory read completion		Load: Reg[IR[11:7]] <= MDR	

FIGURE e4.5.6 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

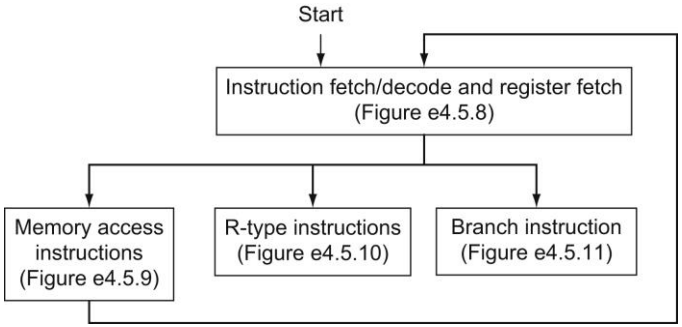


FIGURE e4.5.7 The high-level view of the finite-state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

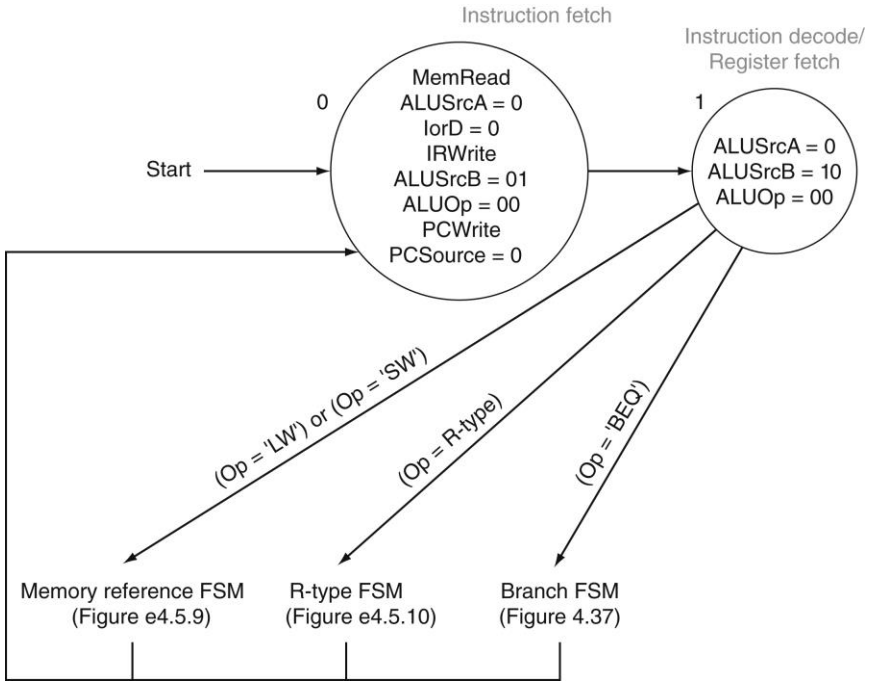


FIGURE e4.5.8 The instruction fetch and decode portion of every instruction is identical. These states correspond to the top box in the abstract finite-state machine in Figure 4.33. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (MemRead and IRWrite), and we set IorD to 0 to choose the PC as the address source. The signals ALUSrcA, ALUSrcB, ALUOp, PCWrite, and PCSource are set to compute PC + 4 and store it into the PC. (It will also be stored into ALUOut, but never used from there.) In the next state, we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of the IR to be sent to the ALU), setting ALUSrcA to 0 and ALUOp to 00; we store the result in the ALUOut register, which is written on every cycle. There are four next states that depend on the class of the instruction, which is known during this state. The control unit input, called Op, is used to determine which of these arcs to follow. Remember that all signals not explicitly asserted are deasserted; this is particularly important for signals that control writes. For multiplexor controls, lack of a specific setting indicates that we do not care about the setting of the multiplexor

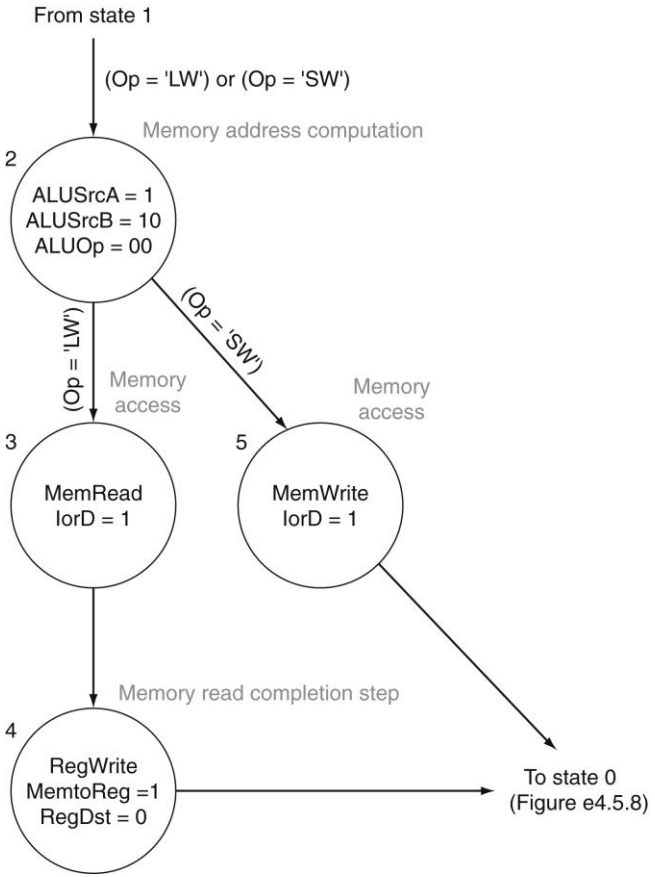


FIGURE e4.5.9 The finite-state machine for controlling memory reference instructions has four states. These states correspond to the box labeled “Memory access instructions” in Figure e4.5.7. After performing a memory address calculation, a separate sequence is needed for load and for store. The setting of the control signals ALUSrcA, ALUSrcB, and ALUOp is used to cause the memory address computation in state 2. Loads require an extra state to write the result from the MDR (where the result is written in state 3) into the register file.

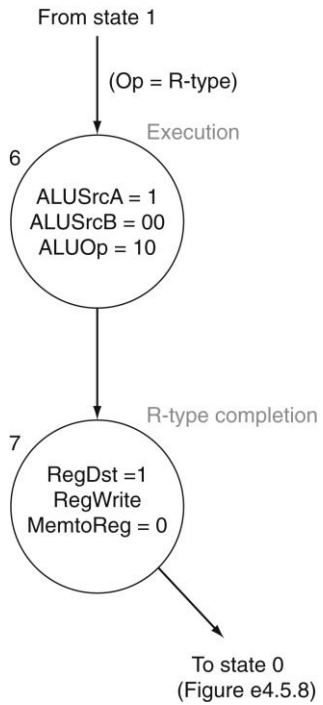


FIGURE e4.5.10 R-type instructions can be implemented with a simple two-state finitestate machine. These states correspond to the box labeled “R-type instructions” in Figure e4.5.7. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register file. The three signals asserted during state 7 cause the contents of ALUOut to be written into the register file in the entry specified by the rd field of the Instruction register.

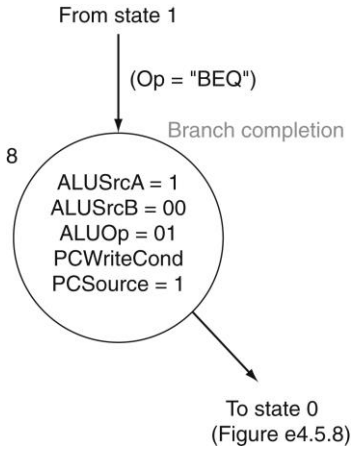


FIGURE e4.5.11 The branch instruction requires a single state. The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the conditional write if the branch condition is true. Notice that we do not use the value written into ALUOut; instead, we use only the Zero output of the ALU. The branch target address is read from ALUOut, where it was saved at the end of state 1.

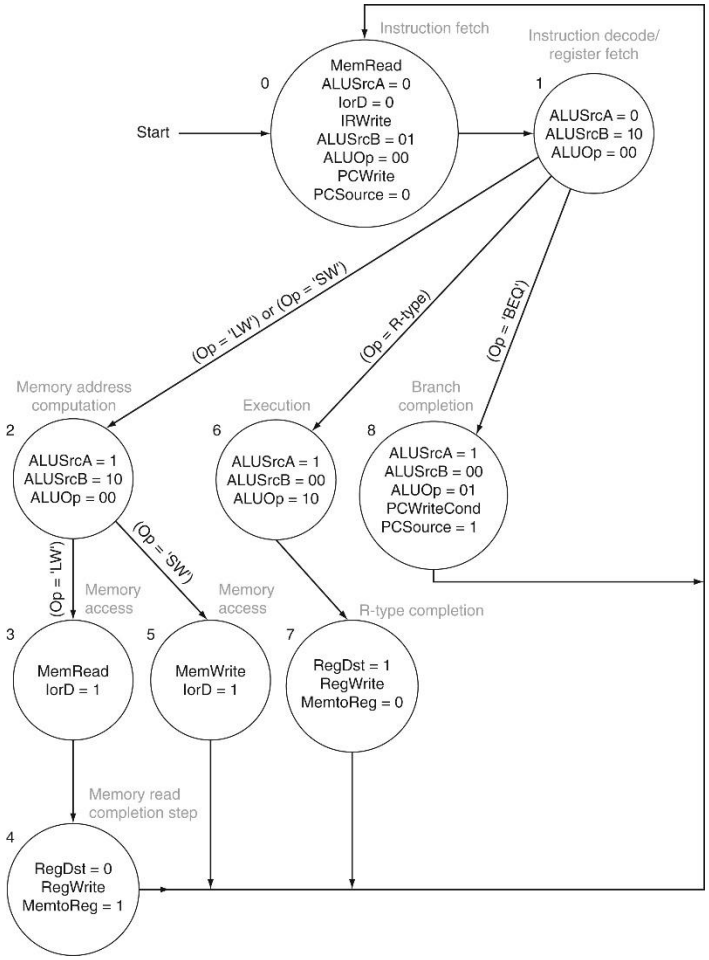


FIGURE e4.5.12 T**he complete finite-state machine control for the datapath shown** in Figure e4.5.4. The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is given. The labels inside the nodes indicate the output signals asserted during that state; we always specify the setting of a multiplexor control signal if the correct operation requires it. Hence, in some states a multiplexor control will be set to 0.

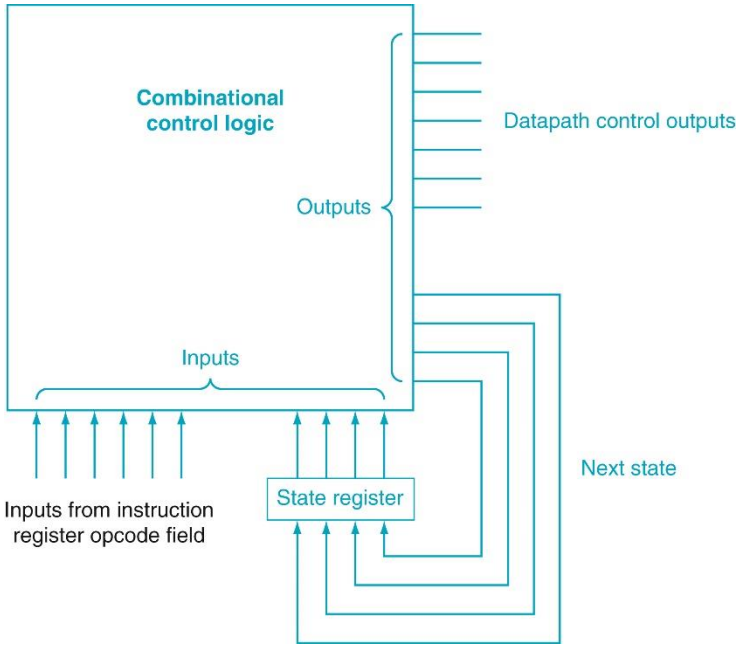


FIGURE e4.5.13 Finite-state machine controllers are typically implemented using a block of combinational logic and a register to hold the current state. The outputs of the combinational logic are the next-state number and the control signals to be asserted for the current state. The inputs to the combinational logic are the current state and any inputs used to determine the next state. In this case, the inputs are the instruction register opcode bits. Notice that in the finite-state machine used in this chapter, the outputs depend only on the current state, not on the inputs. The elaboration below explains this in more detail..

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LW = 7'b000_0011, SW = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [31:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, MEMWBrd; // Access register fields
  wire [6:0] IDEXop, EXMEMop, MEMWBop; // Access opcodes
  wire [31:0] Ain, Bin; // the ALU inputs

  // These assignments define fields from the pipeline registers
  assign IFIDrs1  = IFIDIR[19:15];  // rs1 field
  assign IFIDrs2  = IFIDIR[24:20];  // rs2 field
  assign IDEXop   = IDEXIR[6:0];    // the opcode
  assign EXMEMop  = EXMEMIR[6:0];   // the opcode
  assign MEMWBop  = MEMWBIR[6:0];   // the opcode
  assign MEMWBrd  = MEMWBIR[11:7];  // rd field
  // Inputs to the ALU come directly from the ID/EX pipeline registers
  assign Ain = IDEXA;
  assign Bin = IDEXB;

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    // first instruction in the pipeline is being fetched
    // Fetch & increment PC
    IFIDIR <= IMemory[PC >> 2];
    PC <= PC + 4;

    // second instruction in pipeline is fetching registers
    IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two registers
    IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

    // third instruction is doing address calculation or ALU operation
    if (IDEXop == LW)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
    else if (IDEXop == SW)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR [30:25],
IDEXIR[11:7]};
    else if (IDEXop == ALUop)
      case (IDEXIR[31:25]) // case for the various R-type instructions
        0: EXMEMALUOut <= Ain + Bin;  // add operation
```

```
default: ; // other R-type operations: subtract, SLT, etc.
      endcase
    EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B register

    // Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SW) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LW) || (MEMWBop == ALUop)) && (MEMWBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.14.1 A Verilog behavioral model for the RISC-V five-stage pipeline, ignoring branch and data hazards. As in the design earlier in Chapter 4, we use separate instruction and data memories, which would be implemented using separate caches as we describe in Chapter 5.

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LW = 7'b000_0011, SW = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [31:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, IDEXrs1, IDEXrs2, EXMEMrd, MEMWBrd; //
Access register fields
  wire [6:0] IDEXop, EXMEMop, MEMWBop; // Access opcodes
  wire [31:0] Ain, Bin; // the ALU inputs
  // declare the bypass signals
  wire bypassAfromMEM, bypassAfromALUinWB,
       bypassBfromMEM, bypassBfromALUinWB,
       bypassAfromLDinWB, bypassBfromLDinWB;

  assign IFIDrs1  = IFIDIR[19:15];
  assign IFIDrs2  = IFIDIR[24:20];
  assign IDEXop   = IDEXIR[6:0];
  assign IDEXrs1  = IDEXIR[19:15];
  assign IDEXrs2  = IDEXIR[24:20];
  assign EXMEMop  = EXMEMIR[6:0];
  assign EXMEMrd  = EXMEMIR[11:7];
  assign MEMWBop  = MEMWBIR[6:0];
  assign MEMWBrd  = MEMWBIR[11:7];

  // The bypass to input A from the MEM stage for an ALU operation
  assign bypassAfromMEM = (IDEXrs1 == EXMEMrd) && (IDEXrs1 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input B from the MEM stage for an ALU operation
  assign bypassBfromMEM = (IDEXrs2 == EXMEMrd) && (IDEXrs2 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input A from the WB stage for an ALU operation
  assign bypassAfromALUinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input B from the WB stage for an ALU operation
  assign bypassBfromALUinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input A from the WB stage for an LW operation
  assign bypassAfromLDinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == LW);
  // The bypass to input B from the WB stage for an LW operation
  assign bypassBfromLDinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == LW);
  // The A input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Ain = bypassAfromMEM ? EXMEMALUOut :
               (bypassAfromALUinWB || bypassAfromLDinWB) ? MEMWBValue :
IDEXA;
  // The B input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
```

```
  assign Bin = bypassBfromMEM ? EXMEMALUOut :
               (bypassBfromALUinWB || bypassBfromLDinWB) ? MEMWBValue:
IDEXB;

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    // first instruction in the pipeline is being fetched
    // Fetch & increment PC
    IFIDIR <= IMemory[PC >> 2];
    PC <= PC + 4;

    // second instruction in pipeline is fetching registers
    IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two registers
    IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

    // third instruction is doing address calculation or ALU operation
    if (IDEXop == LW)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
    else if (IDEXop == SW)
      EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:25],
IDEXIR[11:7]};
    else if (IDEXop == ALUop)
      case (IDEXIR[31:25]) // case for the various R-type instructions
        0: EXMEMALUOut <= Ain + Bin;  // add operation
        default: ; // other R-type operations: subtract, SLT, etc.
      endcase
    EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B register

    // Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SW) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LW) || (MEMWBop == ALUop)) && (MEMWBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.14.2 A behavioral definition of the five-stage RISC-V pipeline with bypassing to ALU operations and address calculations. The code added to Figure e4.14.1 to handle bypassing is highlighted. Because these bypasses only require changing where the ALU inputs come from, the only changes required are in the combinational logic responsible for selecting the ALU inputs. (*Continues on next page*)

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LW = 7'b000_0011, SW = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [31:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, IDEXrs1, IDEXrs2, EXMEMrd, MEMWBrd; //
Access register fields
  wire [6:0] IDEXop, EXMEMop, MEMWBop; // Access opcodes
  wire [31:0] Ain, Bin; // the ALU inputs
  // declare the bypass signals
  wire bypassAfromMEM, bypassAfromALUinWB,
       bypassBfromMEM, bypassBfromALUinWB,
       bypassAfromLDinWB, bypassBfromLDinWB;
  wire stall; // stall signal

  assign IFIDrs1  = IFIDIR[19:15];
  assign IFIDrs2  = IFIDIR[24:20];
  assign IDEXop   = IDEXIR[6:0];
  assign IDEXrs1  = IDEXIR[19:15];
  assign IDEXrs2  = IDEXIR[24:20];
  assign EXMEMop  = EXMEMIR[6:0];
  assign EXMEMrd  = EXMEMIR[11:7];
  assign MEMWBop  = MEMWBIR[6:0];
  assign MEMWBrd  = MEMWBIR[11:7];

  // The bypass to input A from the MEM stage for an ALU operation
  assign bypassAfromMEM = (IDEXrs1 == EXMEMrd) && (IDEXrs1 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input B from the MEM  stage for an ALU operation
  assign bypassBfromMEM = (IDEXrs2 == EXMEMrd) && (IDEXrs2 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input A from the WB stage for an ALU operation
  assign bypassAfromALUinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input B from the WB stage for an ALU operation
  assign bypassBfromALUinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input A from the WB stage for an LW operation
  assign bypassAfromLDinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == LW);
  // The bypass to input B from the WB stage for an LW operation
  assign bypassBfromLDinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == LW);
  // The A input to the ALU is  bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Ain = bypassAfromMEM ? EXMEMALUOut :
               (bypassAfromALUinWB || bypassAfromLDinWB) ? MEMWBValue :
IDEXA;
  // The B input to the ALU is bypassed from MEM if there is a bypass
there,
 // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Bin = bypassBfromMEM ? EXMEMALUOut :
               (bypassBfromALUinWB || bypassBfromLDinWB) ? MEMWBValue:
IDEXB;
```

```
  // The signal for detecting a stall based on the use of a result from
LW
  assign stall = (MEMWBop == LW) && ( // source instruction is a load
                  (((IDEXop == LW) || (IDEXop == SW)) && (IDEXrs1 ==
MEMWBrd)) || // stall for address calc
                  ((IDEXop == ALUop) && ((IDEXrs1 == MEMWBrd) ||
(IDEXrs2 == MEMWBrd)))); // ALU use

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    if (~stall)
    begin // the first three pipeline stages stall if there is a load
hazard
      // first instruction in the pipeline is being fetched
      // Fetch & increment PC
      IFIDIR <= IMemory[PC >> 2];
      PC <= PC + 4;

      // second instruction in pipeline is fetching registers
      IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two
registers
      IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

      // third instruction is doing address calculation or ALU operation
      if (IDEXop == LW)
        EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
      else if (IDEXop == SW)
        EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:25],
IDEXIR[11:7]};
      else if (IDEXop == ALUop)
        case (IDEXIR[31:25]) // case for the various R-type instructions
          0: EXMEMALUOut <= Ain + Bin;  // add operation
          default: ; // other R-type operations: subtract, SLT, etc.
        endcase
      EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B
register
    end
    else EXMEMIR <= NOP; // Freeze first three stages of pipeline; inject
a nop into the EX output

// Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SW) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LW) || (MEMWBop == ALUop)) && (MEMWBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.14.3 A behavioral definition of the five-stage RISC-V pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation.

```
module RISCVCPU (clock);
  // Instruction opcodes
  parameter LW = 7'b000_0011, SW = 7'b010_0011, BEQ = 7'b110_0011, NOP =
32'h0000_0013, ALUop = 7'b001_0011;
  input clock;

  reg [31:0] PC, Regs[0:31], IDEXA, IDEXB, EXMEMB, EXMEMALUOut,
MEMWBValue;
  reg [31:0] IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXIR, EXMEMIR, MEMWBIR; // pipeline registers
  wire [4:0] IFIDrs1, IFIDrs2, IDEXrs1, IDEXrs2, EXMEMrd, MEMWBrd; //
Access register fields
  wire [6:0] IFIDop, IDEXop, EXMEMop, MEMWBop; // Access opcodes
  wire [31:0] Ain, Bin; // the ALU inputs
  // declare the bypass signals
  wire bypassAfromMEM, bypassAfromALUinWB,
       bypassBfromMEM, bypassBfromALUinWB,
       bypassAfromLDinWB, bypassBfromLDinWB;
  wire stall; // stall signal
  wire takebranch;

  assign IFIDop   = IFIDIR[6:0];
  assign IFIDrs1  = IFIDIR[19:15];
  assign IFIDrs2  = IFIDIR[24:20];
  assign IDEXop   = IDEXIR[6:0];
  assign IDEXrs1  = IDEXIR[19:15];
  assign IDEXrs2  = IDEXIR[24:20];
  assign EXMEMop  = EXMEMIR[6:0];
  assign EXMEMrd  = EXMEMIR[11:7];
  assign MEMWBop  = MEMWBIR[6:0];
  assign MEMWBrd  = MEMWBIR[11:7];

  // The bypass to input A from the MEM stage for an ALU operation
  assign bypassAfromMEM = (IDEXrs1 == EXMEMrd) && (IDEXrs1 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input B from the MEM stage for an ALU operation
  assign bypassBfromMEM = (IDEXrs2 == EXMEMrd) && (IDEXrs2 != 0) &&
(EXMEMop == ALUop);
  // The bypass to input A from the WB stage for an ALU operation
  assign bypas sAfromALUinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input B from the WB stage for an ALU operation
  assign bypassBfromALUinWB = (IDEXrs2 == MEMWBrd) && (IDEXrs2 != 0) &&
(MEMWBop == ALUop);
  // The bypass to input A from the WB stage for an LW operation
  assign bypassAfromLDinWB = (IDEXrs1 == MEMWBrd) && (IDEXrs1 != 0) &&
(MEMWBop == LW);
  // The bypass to input B from the WB stage for an LW operation
  assign bypassBfromLDinWB = (IDEXrs2 == MEMWBrd) && (IDEX rs2 != 0) &&
(MEMWBop == LW);
  // The A input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Ain = bypassAfromMEM ? EXMEMALUOut :
               (bypassAfromALUinWB || bypassAfromLDinWB) ? MEMWBValue :
IDEXA;

 // The B input to the ALU is bypassed from MEM if there is a bypass
there,
  // Otherwise from WB if there is a bypass there, and otherwise comes
from the IDEX register
  assign Bin = bypassBfromMEM ? EXMEMALUOut :
               (bypassBfromALUinWB || bypassBfromLDinWB) ? MEMWBValue:
```

```
IDEXB;
  // The signal for detecting a stall based on the use of a result from
LW
  assign stall = (MEMWBop == LW) && ( // source instruction is a load
                  (((IDEXop == LW) || (IDEXop == SW)) && (IDEXrs1 ==
MEMWBrd)) || // stall for address calc
                  ((IDEXop == ALUop) && ((IDEXrs1 == MEMWBrd) ||
(IDEXrs2 == MEMWBrd)))); // ALU use
  // Signal for a taken branch: instruction is BEQ and registers are
equal
  assign takebranch = (IFIDop == BEQ) && (Regs[IFIDrs1] ==
Regs[IFIDrs2]);

  integer i; // used to initialize registers
  initial
  begin
    PC = 0;
    IFIDIR = NOP; IDEXIR = NOP; EXMEMIR = NOP; MEMWBIR = NOP; // put NOPs
in pipeline registers
    for (i=0;i<=31;i=i+1) Regs[i] = i; // initialize registers--just so
they aren't cares
  end

  // Remember that ALL these actions happen every pipe stage and with the
use of <= they happen in parallel!
  always @(posedge clock)
  begin
    if (~stall)
    begin // the first three pipeline stages stall if there is a load
hazard
      if (~takebranch)
      begin // first instruction in the pipeline is being fetched
normally
        IFIDIR <= IMemory[PC >> 2];
        PC <= PC + 4;
      end
      else
      begin // a taken branch is in ID; instruction in IF is wrong;
insert a NOP and reset the PC
        IFIDIR <= NOP;
        PC <= PC + {{52{IFIDIR[31]}}, IFIDIR[7], IFIDIR[30:25],
IFIDIR[11:8], 1'b0};
      end

      // second instruction in pipeline is fetching registers
      IDEXA <= Regs[IFIDrs1]; IDEXB <= Regs[IFIDrs2]; // get two
registers
      IDEXIR <= IFIDIR; // pass along IR--can happen anywhere, since this
affects next stage only!

      // third instruction is doing addre ss calculation or ALU operation
      if (IDEXop == LW)


 EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:20]};
      else if (IDEXop == SW)
        EXMEMALUOut <= IDEXA + {{53{IDEXIR[31]}}, IDEXIR[30:25],
IDEXIR[11:7]};
      else if (IDEXop == ALUop)
        case (IDEXIR[31:25]) // case for the various R-type instructions
          0: EXMEMALUOut <= Ain + Bin;  // add operation
```

```
default: ; // other R-type operations: subtract, SLT, etc.
        endcase
      EXMEMIR <= IDEXIR; EXMEMB <= IDEXB; // pass along the IR & B
register
    end
    else EXMEMIR <= NOP; // Freeze first three stages of pipeline; inject
a nop into the EX output

    // Mem stage of pipeline
    if (EXMEMop == ALUop) MEMWBValue <= EXMEMALUOut; // pass along ALU
result
    else if (EXMEMop == LW) MEMWBValue <= DMemory[EXMEMALUOut >> 2];
    else if (EXMEMop == SW) DMemory[EXMEMALUOut >> 2] <= EXMEMB; //store
    MEMWBIR <= EXMEMIR; // pass along IR

    // WB stage
    if (((MEMWBop == LW) || (MEMWBop == ALUop)) && (MEM WBrd != 0)) //
update registers if load/ALU operation and destination not 0
      Regs[MEMWBrd] <= MEMWBValue;
  end
endmodule
```

FIGURE e4.14.4 A behavioral definition of the five-stage RISC-V pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation.

```
module RISCVCPU (clock);
  parameter LW = 7'b000_0011, SW = 7'b010_0011, BEQ = 7'b110_0011, ALUop
= 7'b001_0011;
  input clock; //the clock is an external input

  // The architecturally visible registers and scratch registers for
implementation
  reg [31:0] PC, Regs[0:31], ALUOut, MDR, A, B;
  reg [31:0] Memory [0:1023], IR;
  reg [2:0] state; // processor state
  wire [6:0] opcode; // use to get opcode easily
  wire [31:0] ImmGen; // used to generate immediate

  assign opcode = IR[6:0]; // opcode is lower 7 bits
  assign ImmGen = (opcode == LW) ? {{53{IR[31]}}, IR[30:20]} :
              /* (opcode == SW) */{{53{IR[31]}}, IR[30:25], IR[11:7]};
  assign PCOffset = {{52{IR[31]}}, IR[7], IR[30:25], IR[11:8], 1'b0};

  // set the PC to 0 and start the control in state 1
  initial begin PC = 0; state = 1; end

  // The state machine--triggered on a rising clock
  always @(posedge clock)
  begin
    Regs[0] <= 0; // shortcut way to make sure R0 is always 0
    case (state) //action depends on the state
      1: begin // first step: fetch the instruction, increment PC, go to
next state
        IR <= Memory[PC >> 2];
        PC <= PC + 4;
        state <= 2; // next state
      end
      2: begin // second step: Instruction decode, register fetch, also
compute branch address
        A <= Regs[IR[19:15]];
        B <= Regs[IR[24:20]];
        ALUOut <= PC + PCOffset; // compute PC-relative branch target
        state <= 3;
      end
      3: begin // third step: Load-store execution, ALU execution, Branch
completion
        if ((opcode == LW) || (opcode == SW))
        begin
          ALUOut <= A + ImmGen; // compute effective address
          state <= 4;
        end
        else if (opcode == ALUop)
        begin
          case (IR[31:25]) // case for the various R-type instructions
            0: ALUOut <= A + B; // add operation
            default: ; // other R-type operations: subtract, SLT, etc.
          endcase
          state <= 4;
        end
        else if (opcode == BEQ)
        begin
          if (A == B) begin
            PC <= ALUOut; // branch taken--update PC

    end
```

```
          state <= 1;
        end
        else ; // other opcodes or exception for undefined instruction
would go here
      end
      4: begin
        if (opcode == ALUop)
        begin // ALU Operation
          Regs[IR[11:7]] <= ALUOut; // write the result
          state <= 1;
        end // R-type finishes
        else if (opcode == LW)
        begin // load instruction
           MDR <= Memory[ALUOut >> 2]; // read the memory
           state <= 5; // next state
        end
        else if (opcode == SW)
        begin // store instruction
          Memory[ALUOut >> 2] <= B; // write the memory
          state <= 1; // return to state 1
        end
        else ; // other instructions go here
      end
      5: begin // LW is the only instruction still in execution
        Regs[IR[11:7]] <= MDR; // write the MDR to the register
        state <= 1;
      end // complete an LW instruction
    endcase
  end
endmodule
```

FIGURE e4.14.5 A behavioral specification of the multicycle RISC-V design.

```
module Datapath (ALUOp, MemtoReg, MemRead, MemWrite, IorD, RegWrite,
IRWrite,
                 PCWrite, PCWriteCond, ALUSrcA, ALUSrcB, PCSource,
opcode, clock); // the control inputs + clock
  parameter LW = 7'b000_0011, SW = 7'b010_0011;
  input [1:0] ALUOp, ALUSrcB; // 2-bit control signals
  input MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite,
PCWriteCond,
        ALUSrcA, PCSource, clock; // 1-bit control signals
  output [6:0] opcode; // opcode is needed as an output by control
  reg [31:0] PC, MDR, ALUOut; // CPU state + some temporaries
  reg [31:0] Memory[0:1023], IR; // CPU state + some temporaries
  wire [31:0] A, B, SignExtendOffset, PCOffset, ALUResultOut, PCValue,
JumpAddr, Writedata, ALUAin,
              ALUBin, MemOut; // these are signals derived from registers
  wire [3:0] ALUCtl; // the ALU control lines
  wire Zero; // the Zero out signal from the ALU

  initial PC = 0; //start the PC at 0
  //Combinational signals used in the datapath
  // Read using word address with either ALUOut or PC as the address
source
  assign MemOut = MemRead ? Memory[(IorD ? ALUOut : PC) >> 2] : 0;
  assign opcode = IR[6:0]; // opcode shortcut
  // Get the write register data either from the ALUOut or from the MDR
  assign Writedata = MemtoReg ? MDR : ALUOut;
  // Generate immediate
  assign ImmGen = (opcode == LW) ? {{53{IR[31]}}, IR[30:20]} :
                /* (opcode == SW) */{{53{IR[31]}}, IR[30:25], IR[11:7]};
  // Generate pc offset for branches
  assign PCOffset = {{52{IR[31]}}, IR[7], IR[30:25], IR[11:8], 1'b0};
  // The A input to the ALU is either the rs register or the PC
  assign ALUAin = ALUSrcA ? A : PC; // ALU input is PC or A

  // Creates an instance of the ALU control unit (see the module defined
in Figure B.5.16
    // Input ALUOp is control-unit set and used to describe the
instruction class as in Chapter 4
    // Input IR[31:25] is the function code field for an ALU instruction
    // Output ALUCtl are the actual ALU control bits as in Chapter 4
    ALUControl alucontroller (ALUOp, IR[31:25], ALUCtl); // ALU control
unit

  // Creates a 2-to-1 multiplexor used to select the source of the next
PC
    // Inputs are ALUResultOut (the incremented PC), ALUOut (the branch
address)
    // PCSource is the selector input and PCValue is the multiplexor
output
    Mult2to1 PCdatasrc (ALUResultOut, ALUOut, PCSource, PCValue);

  // Creates a 4-to-1 multiplexor used to select the B input of the ALU
    // Inputs are register B, constant 4, generated immediate, PC offset
// ALUSrcB is the select or input
    // ALUBin is the multiplexor output
    Mult4to1 ALUBinput (B, 32'd4, ImmGen, PCOffset, ALUSrcB, ALUBin);

  // Creates a RISC-V ALU
    // Inputs are ALUCtl (the ALU control), ALU value inputs (ALUAin,
ALUBin)
    // Outputs are ALUResultOut (the 32-bit output) and Zero (zero
detection output)
    RISCVALU ALU (ALUCtl, ALUAin, ALUBin, ALUResultOut, Zero); // the ALU
```

```
  // Creates a RISC-V register file
    // Inputs are the rs1 and rs2 fields of the IR used to specify which
registers to read,
    // Writereg (the write register number), Writedata (the data to be
written),
    // RegWrite (indicates a write), the clock
    // Outputs are A and B, the registers read
    registerfile regs (IR[19:15], IR[24:20], IR[11:7], Writedata,
RegWrite, A, B, clock); // Register file

  // The clock-triggered actions of the datapath
  always @(posedge clock)
  begin
    if (MemWrite) Memory[ALUOut >> 2] <= B; // Write memory--must be a
store
    ALUOut <= ALUResultOut; // Save the ALU result for use on a later
clock cycle
    if (IRWrite) IR <= MemOut; // Write the IR if an instruction fetch
    MDR <= MemOut; // Always save the memory read value
    // The PC is written both conditionally (controlled by PCWrite) and
unconditionally
  end
endmodule
```

FIGURE e4.14.6 A Verilog version of the multicycle RISC-V datapath that is appropriate for synthesis.

```
module RISCVCPU (clock);
  parameter LW = 7'b000_0011, SW = 7'b010_0011, BEQ = 7'b110_0011, ALUop
= 7'b001_0011;
  input clock;

  reg [2:0] state;
  wire [1:0] ALUOp, ALUSrcB;
  wire [6:0] opcode;
  wire MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite,
       PCWrite, PCWriteCond, ALUSrcA, PCSource, MemoryOp;

  // Create an instance of the RISC-V datapath, the inputs are the
control signals; opcode is only output
  Datapath RISCVDP (ALUOp, MemtoReg, MemRead, MemWrite, IorD, RegWrite,
IRWrite,
                    PCWrite, PCWriteCond, ALUSrcA, ALUSrcB, PCSource,
opcode, clock);

  initial begin state = 1; end // start the state machine in state 1
  // These are the definitions of the control signals
  assign MemoryOp = (opcode == LW) || (opcode == SW); // a memory
operation
  assign ALUOp = ((state == 1) || (state == 2) || ((state == 3) &&
MemoryOp)) ? 2'b00 : // add
                ((state == 3) && (opcode == BEQ)) ? 2'b01 : 2'b10; //
subtract or use function code
  assign MemtoReg = ((state == 4) && (opcode == ALUop)) ? 0 : 1;
  assign MemRead = (state == 1) || ((state == 4) && (opcode == LW));
  assign MemWrite = (state == 4) && (opcode == SW);
  assign IorD = (state == 1) ? 0 : 1;
  assign RegWrite = (state == 5) || ((state == 4) && (opcode == ALUop));
  assign IRWrite = (state == 1);
  assign PCWrite = (state == 1);
  assign PCWriteCond = (state == 3) && (opcode == BEQ);
  assign ALUSrcA = ((state == 1) || (state == 2)) ? 0 : 1;
  assign ALUSrcB = ((state == 1) || ((state == 3) && (opcode == BEQ)))?
2'b01 :
                   (state == 2) ? 2'b11 :
                   ((state == 3) && MemoryOp) ? 2'b10 : 2'b00; // memory
operation or other
  assign PCSource = (state == 1) ? 0 : 1;

  // Here is the state machine, which only has to sequence states
  always @(posedge clock)
  begin // all state updates on a positive clock edge
    case (state)
      1: state <= 2; // unconditional next state
      2: state <= 3; // unconditional next state
      3: state <= (opcode == BEQ) ? 1 : 4; // branch go back else next
state
      4: state <= (opcode == LW) ? 5 : 1; // R-type and LW finish
      5: state <= 1; // go back
    endcase
  end
endmodule
```

FIGURE e4.14.7 The RISC-V CPU using the datapath from Figure e4.14.6.

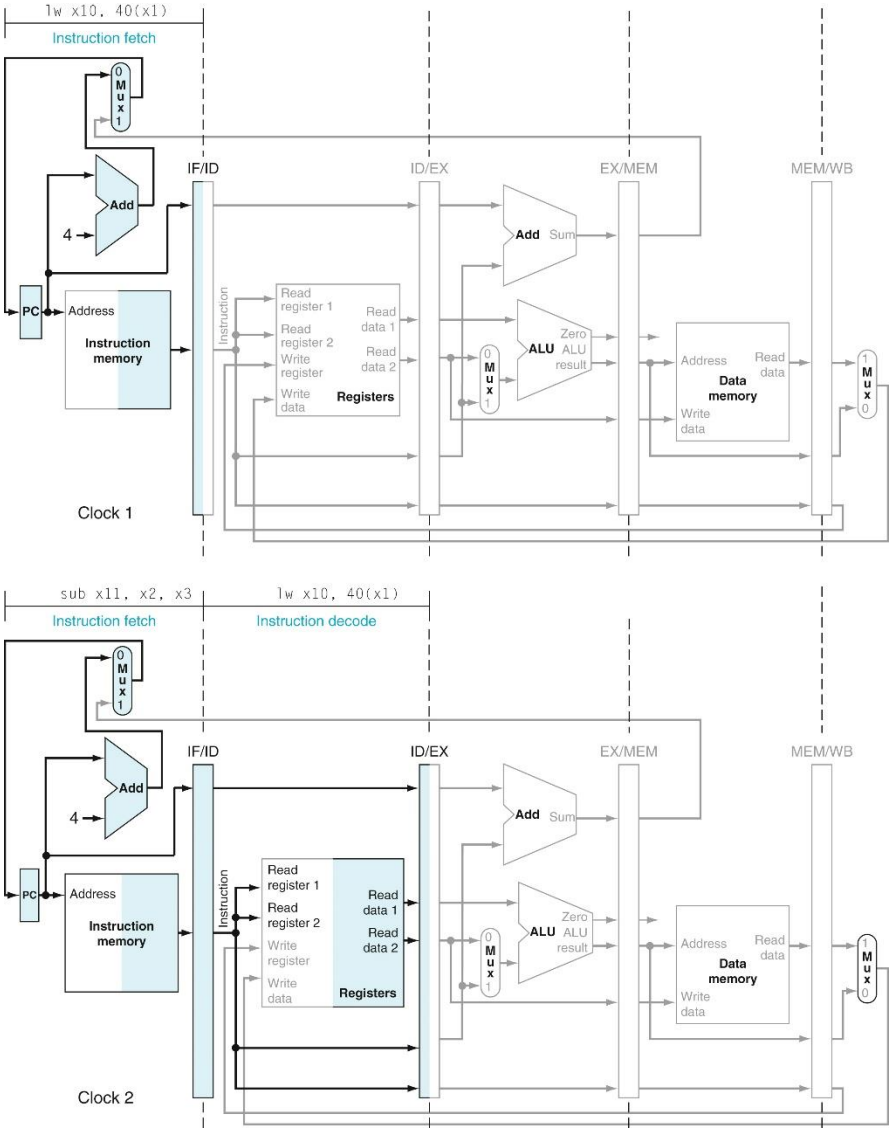


FIGURE e4.14.8 Single-cycle pipeline diagrams for clock cycles 1 (top diagram) and 2 (bottom diagram). This style of pipeline representation is a snapshot of every instruction executing during one clock cycle. Our example has but two instructions, so at most two stages are identified in each clock cycle; normally, all five stages are occupied. The highlighted portions of the datapath are active in that clock cycle. The load is fetched in clock cycle 1 and decoded in clock cycle 2, with the subtract fetched in the second clock cycle. To make the figures easier to understand, the other pipeline stages are empty, but normally there is an instruction in every pipeline stage.

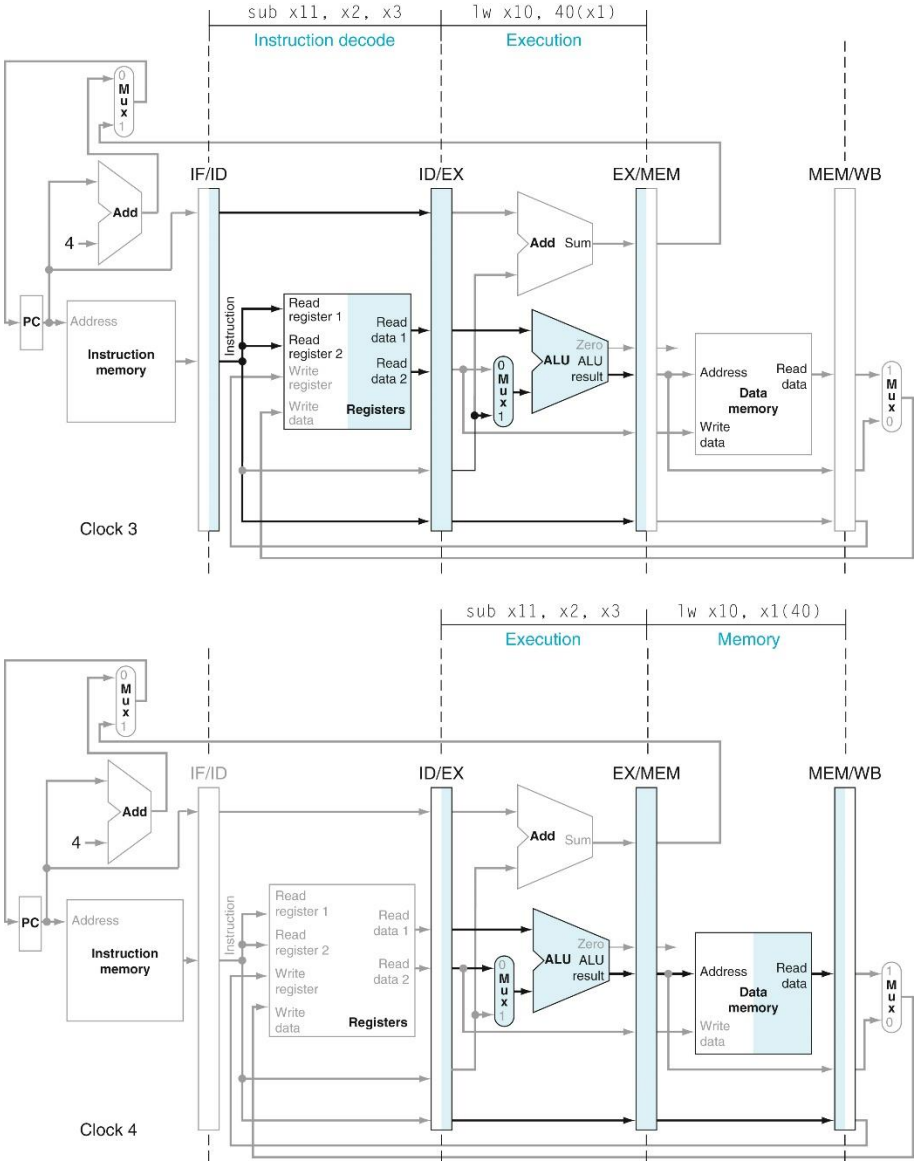


FIGURE e4.14.9 Single-cycle pipeline diagrams for clock cycles 3 (top diagram) and 4 (bottom diagram). In the third clock cycle in the top diagram, lw enters the EX stage. At the same time, sub enters ID. In the fourth clock cycle (bottom datapath), lw moves into MEM stage, reading memory using the address found in EX/MEM at the beginning of clock cycle 4. At the same time, the ALU subtracts and then places the difference into EX/MEM at the end of the clock cycle.

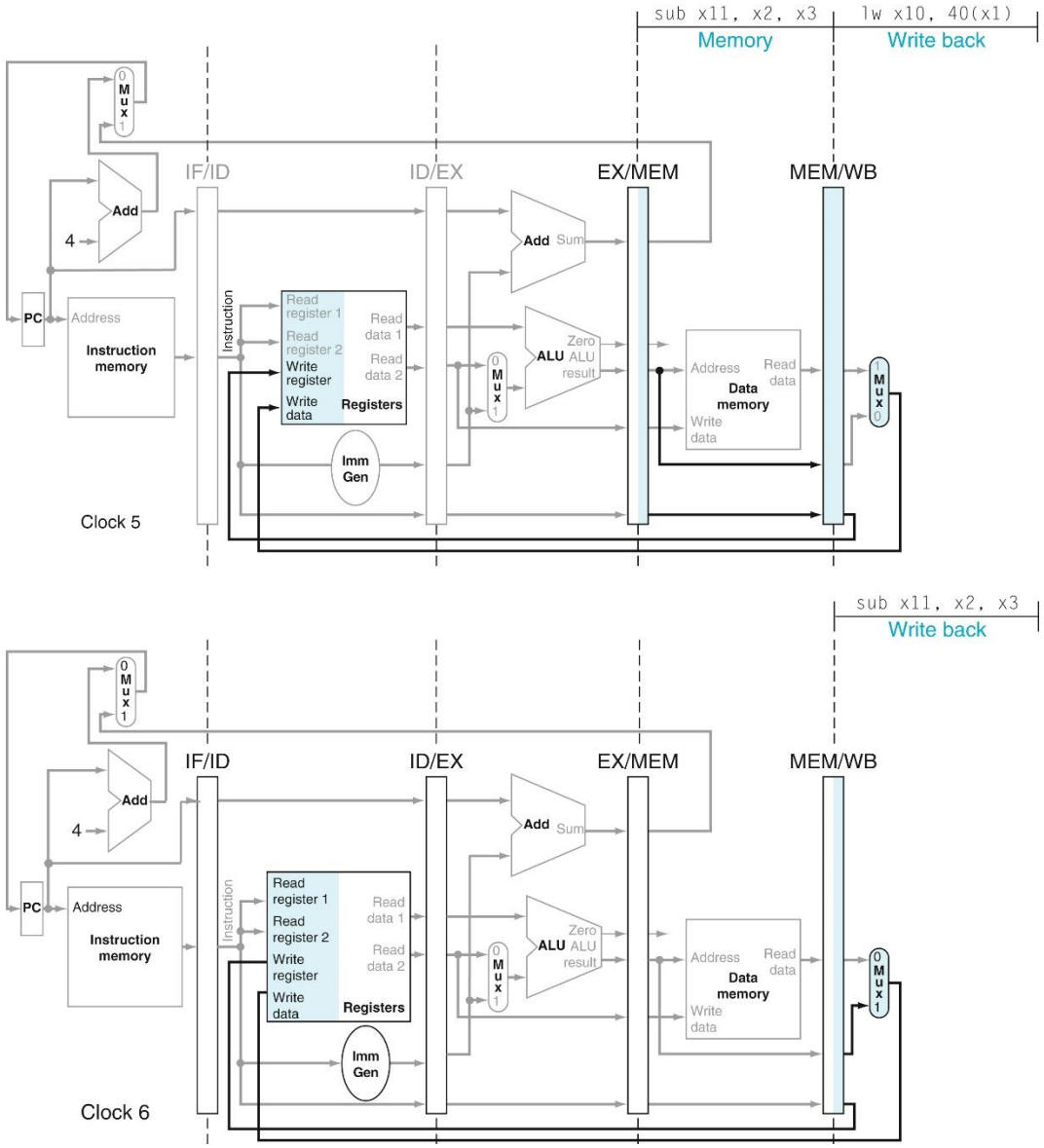


FIGURE e4.14.10 Single-cycle pipeline diagrams for clock cycles 5 (top diagram) and 6 (bottom diagram). In clock cycle 5, lw completes by writing the data in MEM/WB into register 10, and sub sends the difference in EX/MEM to MEM/WB. In the next clock cycle, sub writes the value in MEM/WB to register 11.

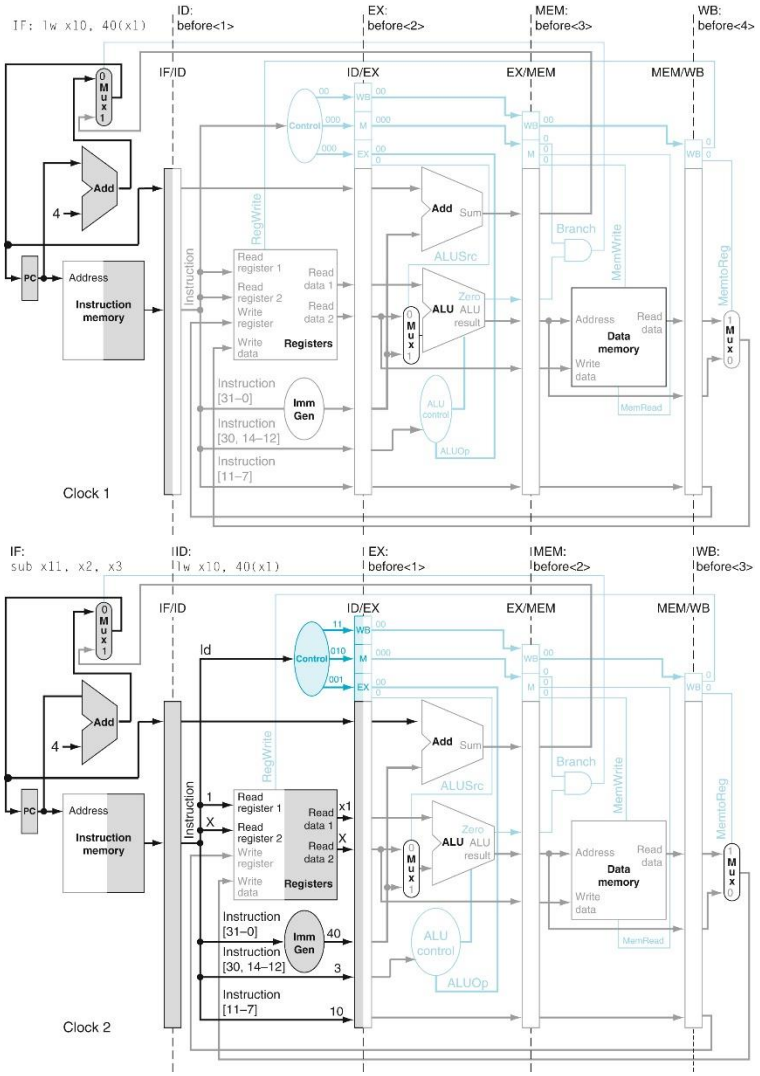


FIGURE e4.14.11 Clock cycles 1 and 2. The phrase "*before* <*i*>" means the *i*th instruction before lw. The lw instruction in the top datapath is in the IF stage. At the end of the clock cycle, the lw instruction is in the IF/ID pipeline registers. In the second clock cycle, seen in the bottom datapath, the lw moves to the ID stage, and sub enters in the IF stage. Note that the values of the instruction fields and the selected source registers are shown in the ID stage. Hence, register x1 and the constant 40, the operands of lw, are written into the ID/EX pipeline register. The number 10, representing the destination register number of lw, is also placed in ID/EX. The top of the ID/EX pipeline register shows the control values for ld to be used in the remaining stages. These control values can be read from the lw row of the table in Figure 4.22

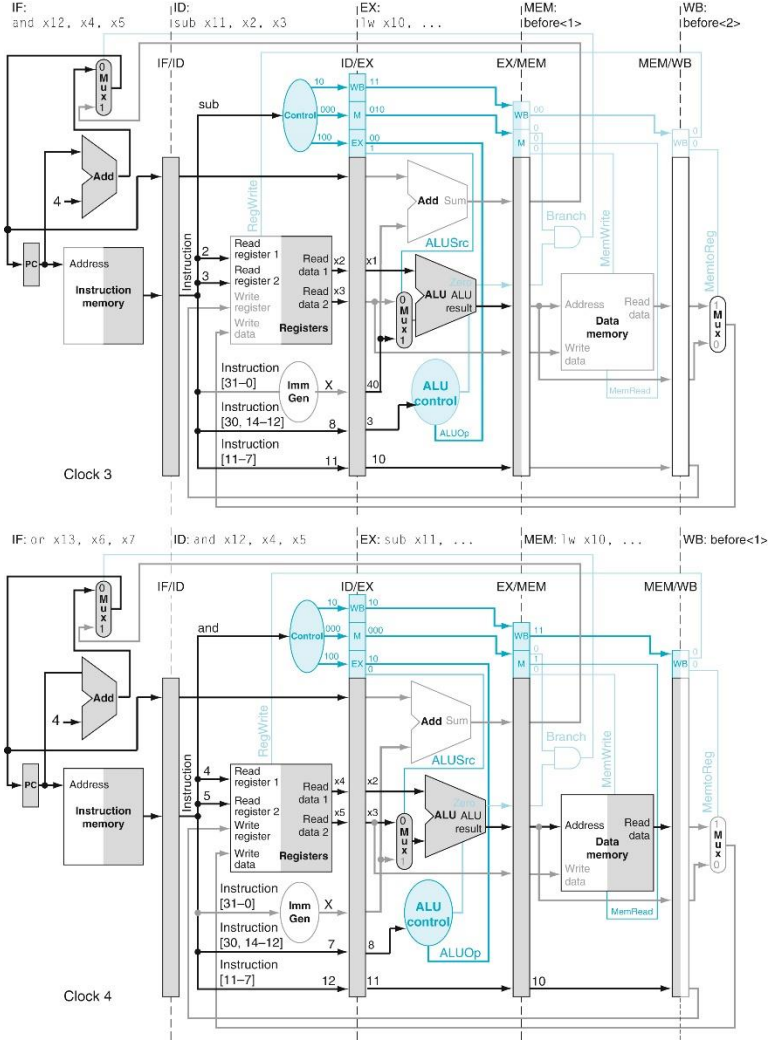


FIGURE e4.14.12 Clock cycles 3 and 4. In the top diagram, lw enters the EX stage in the third clock cycle, adding x1 and 40 to form the address in the EX/MEM pipeline register. (The lw instruction is written lw x10, … upon reaching EX, because the identity of instruction operands is not needed by EX or the subsequent stages. In this version of the pipeline, the actions of EX, MEM, and WB depend only on the instruction and its destination register or its target address.) At the same time, sub enters ID, reading registers x2 and x3, and the and instruction starts IF. In the fourth clock cycle (bottom datapath), lw moves into MEM stage, reading memory using the value in EX/ MEM as the address. In the same clock cycle, the ALU subtracts x3 from x2 and places the difference into EX/MEM, reads registers x4 and x5 during ID, and the or instruction enters IF. The two diagrams show the control signals being created in the ID stage and peeled off as they are used in subsequent pipe stages.

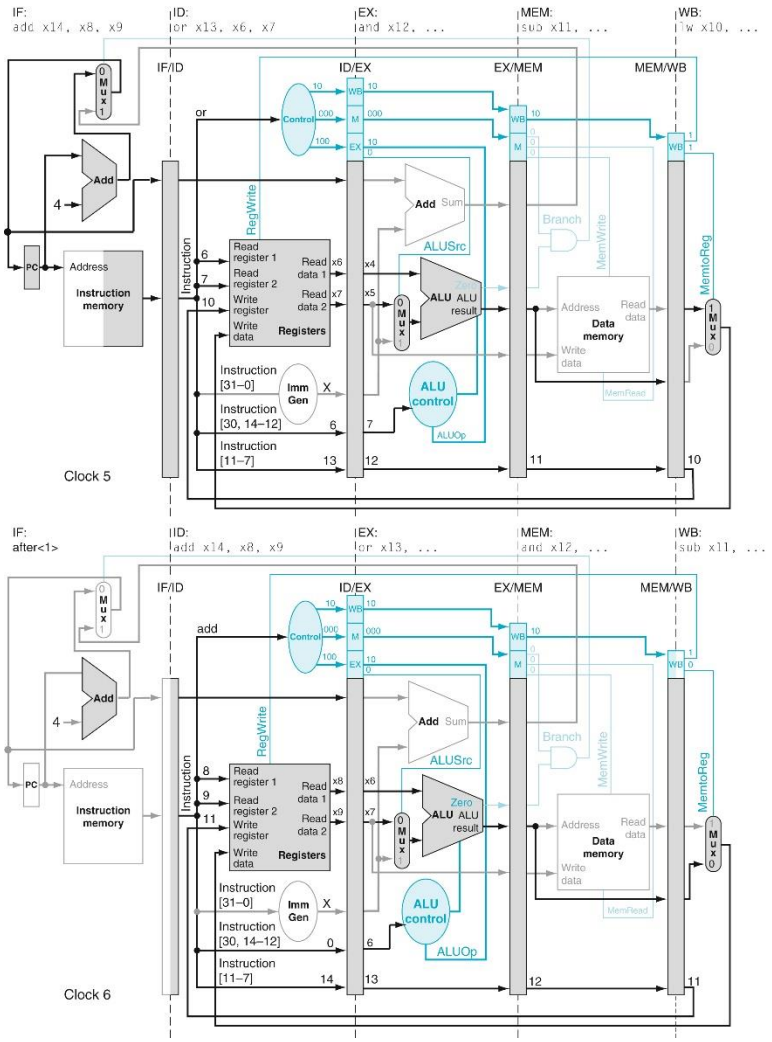


FIGURE e4.14.13 Clock cycles 5 and 6. With add, the final instruction in this example, entering IF in the top datapath, all instructions are engaged. By writing the data in MEM/WB into register 10, lw completes; both the data and the register number are in MEM/WB. In the same clock cycle, sub sends the difference in EX/MEM to MEM/WB, and the rest of the instructions move forward. In the next clock cycle, sub selects the value in MEM/WB to write to register number 11, again found in MEM/WB. The remaining instructions play follow-the-leader: the ALU calculates the OR of x6 and x7 for the or instruction in the EX stage, and registers x8 and x9 are read in the ID stage for the add instruction. The instructions after add are shown as inactive just to emphasize what occurs for the five instructions in the example. The phrase “after <i>” means the *i*th instruction after add.

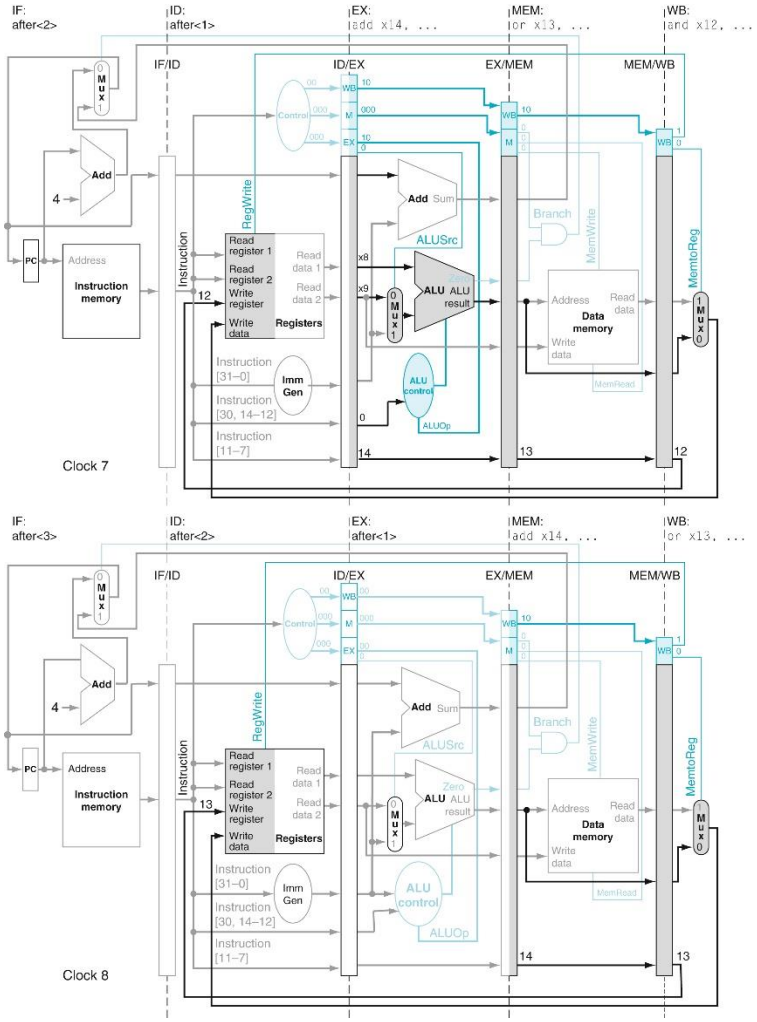


FIGURE e4.14.14 Clock cycles 7 and 8. In the top datapath, the add instruction brings up the rear, adding the values corresponding to registers x8 and x9 during the EX stage. The result of the or instruction is passed from EX/MEM to MEM/WB in the MEM stage, and the WB stage writes the result of the and instruction in MEM/WB to register x12. Note that the control signals are deasserted (set to 0) in the ID stage, since no instruction is being executed. In the following clock cycle (lower drawing), the WB stage writes the result to register x13, thereby completing or, and the MEM stage passes the sum from the add in EX/MEM to MEM/WB. The instructions after add are shown as inactive for pedagogical reasons.

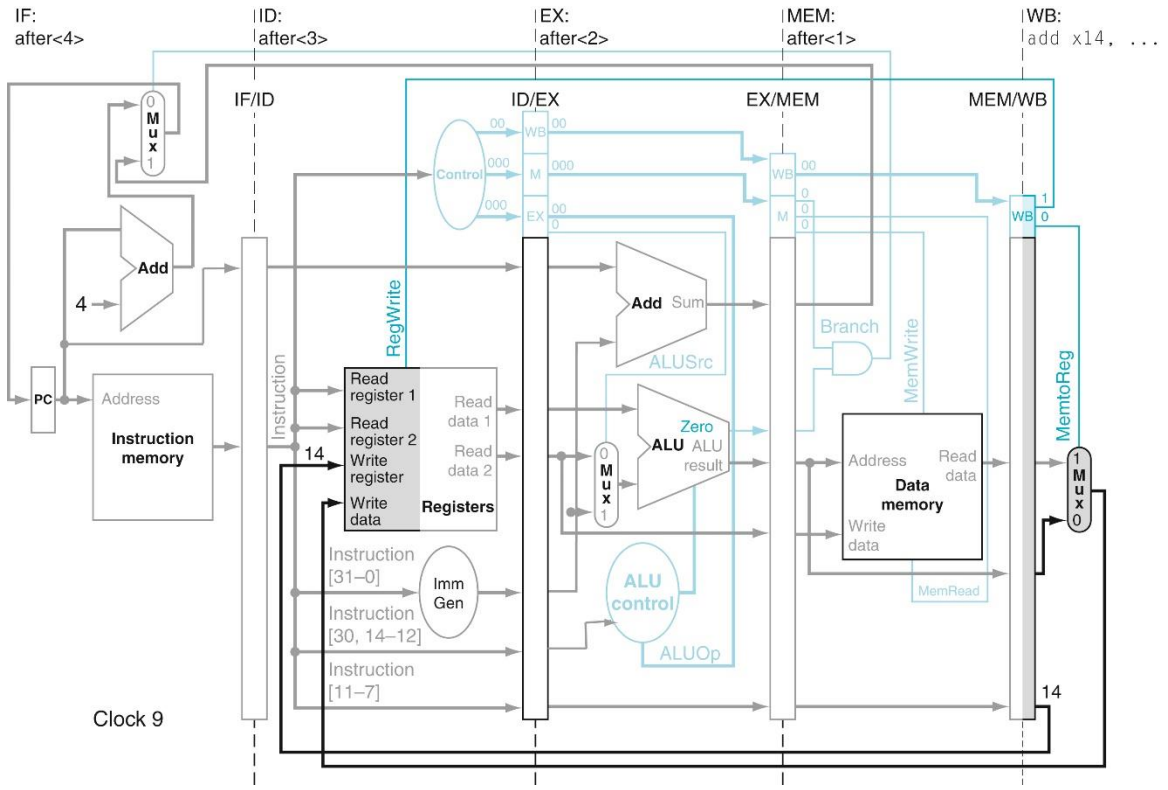


FIGURE e4.14.15 Clock cycle 9. The WB stage writes the ALU result in MEM/WB into register x14, completing add and the five-instruction sequence. The instructions after add are shown as inactive for pedagogical reasons.

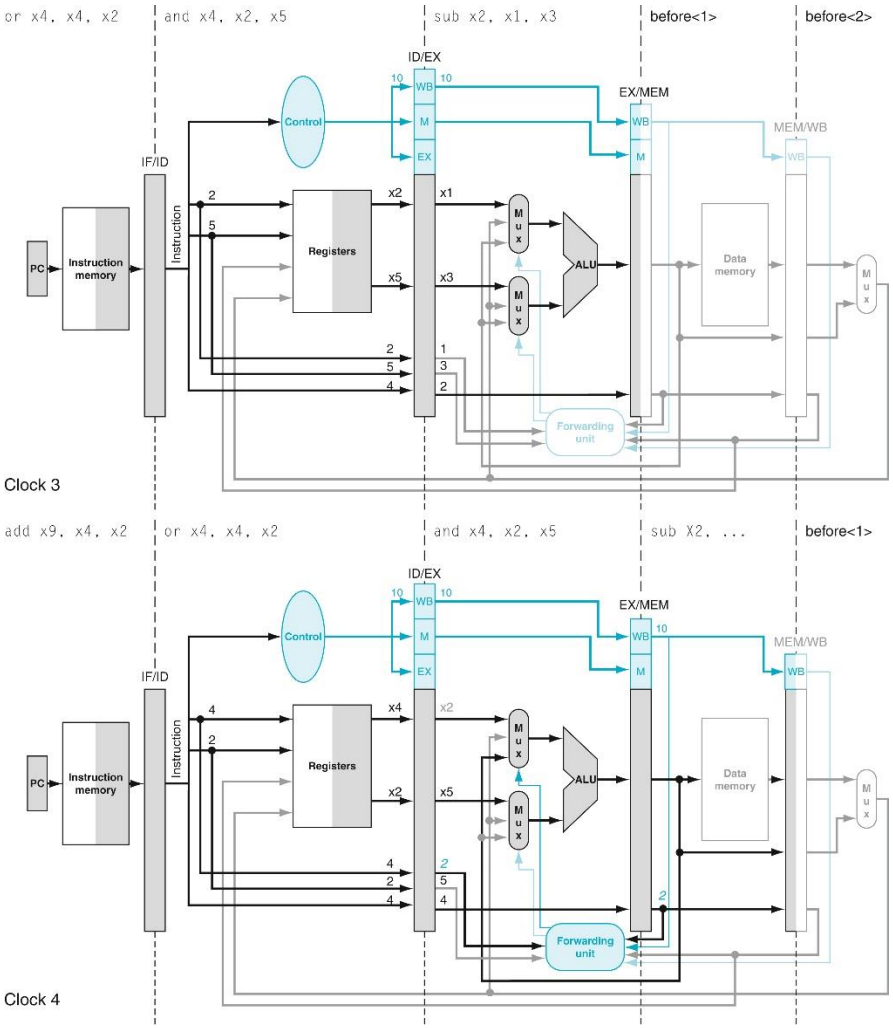


FIGURE e4.14.16 Clock cycles 3 and 4 of the instruction sequence on page 366.e26. The bold lines are those active in a cloc cycle, and the italicized register numbers in color indicate a hazard. The forwarding unit is highlighted by shading it when it is forwarding data to the ALU. The instructions before sub are shown as inactive just to emphasize what occurs for the four instructio in the example. Operand names are used in EX for control of forwarding; thus they are included in the instruction label for EX. Operand names are not needed in MEM or WB, so … is used. Compare this with Figures e4.14.12 through e4.14.15, which show the datapath without forwarding where ID is the last stage to need operand information.

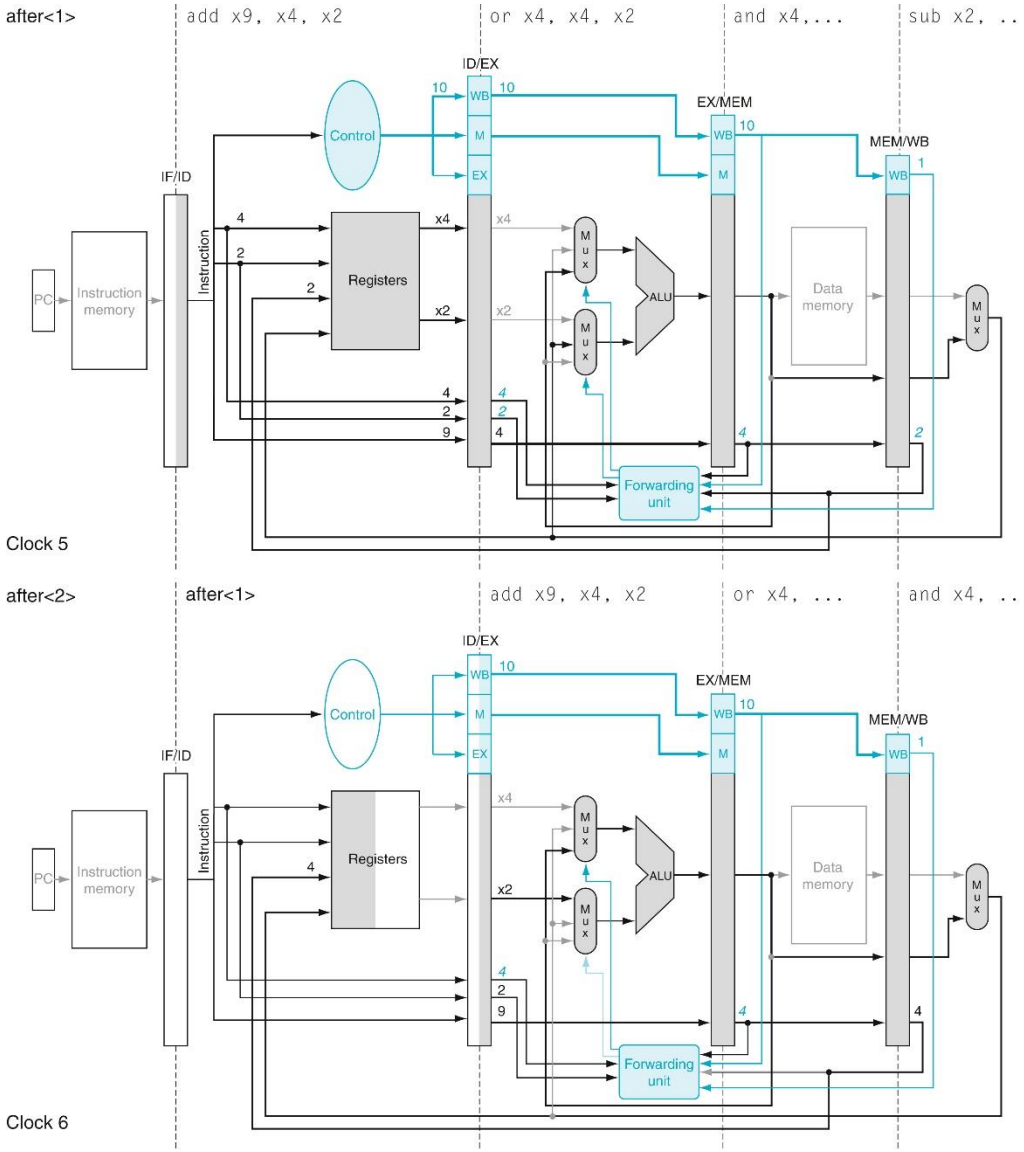


FIGURE e4.14.17 Clock cycles 5 and 6 of the instruction sequence on page 366.e26. The forwarding unit is highlighted when is forwarding data to the ALU. The two instructions after add are shown as inactive just to emphasize what occurs for the four instructions in the example. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

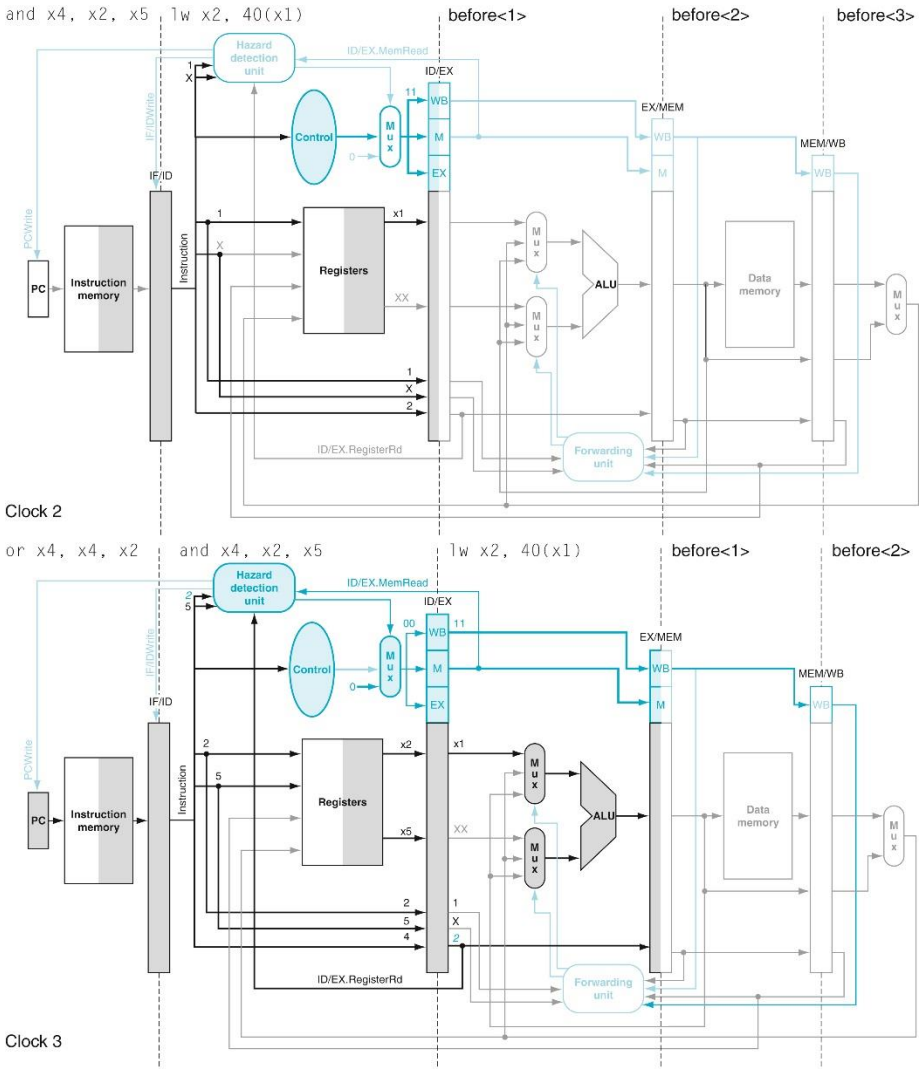


FIGURE e4.14.18 Clock cycles 2 and 3 of the instruction sequence on page 366.e26 with a load replacing sub**.** The bold line are those active in a clock cycle, the italicized register numbers in color indicate a hazard, and the … in the place of operands means that their identity is information not needed by that stage. The values of the significant control lines, registers, and register numbers are labeled in the figures. The and instruction wants to read the value created by the lw instruction in clock cycle 3, so the hazard detection unit stalls the and and or instructions. Hence, the hazard detection unit is highlighted.

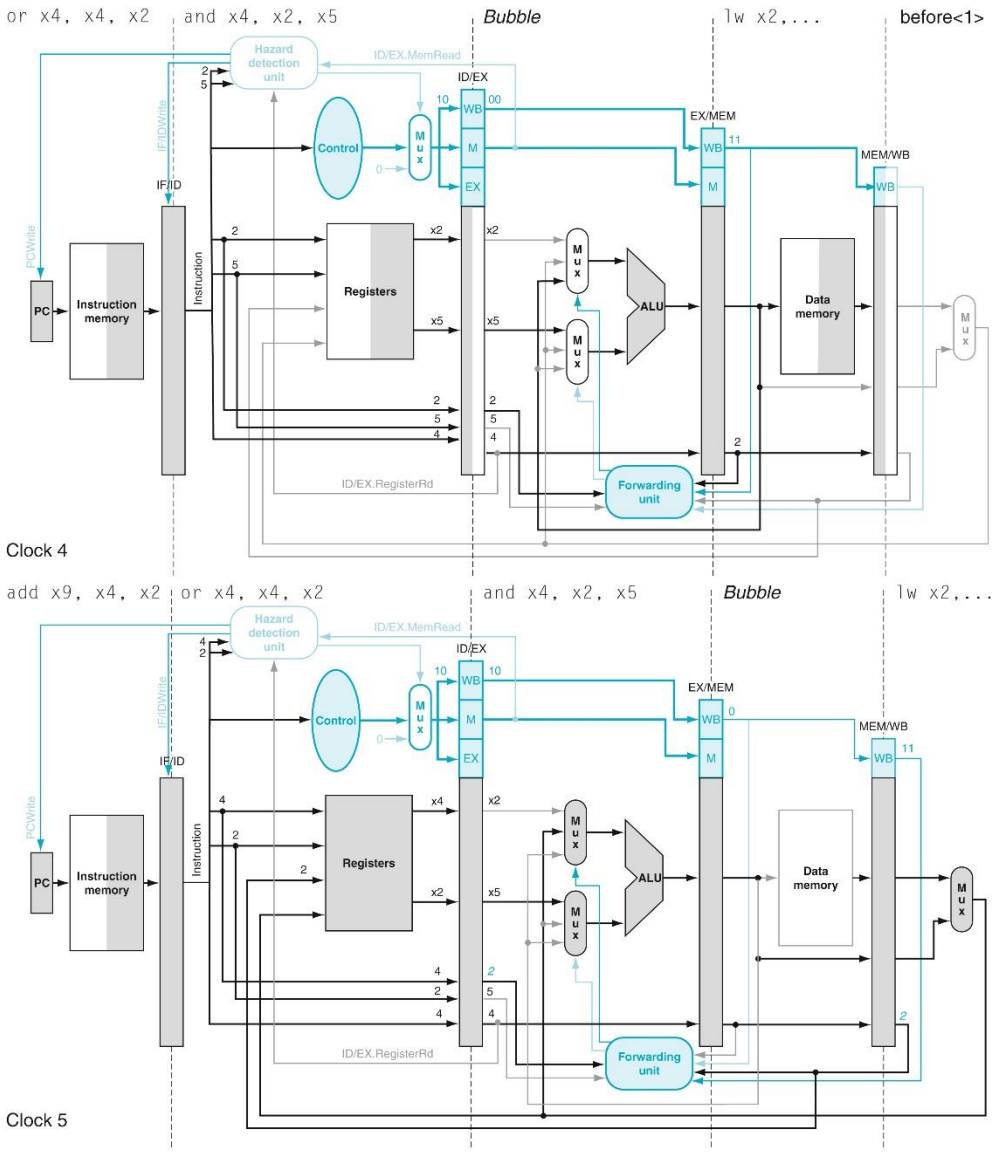


FIGURE e4.14.19 Clock cycles 4 and 5 of the instruction sequence on page 366.e26 with a load replacing sub**.** The bubble is inserted in the pipeline in clock cycle 4, and then the and instruction is allowed to proceed in clock cycle 5. The forwarding unit is highlighted in clock cycle 5 because it is forwarding data from lw to the ALU. Note that in clock cycle 4, the forwarding unit forwards the address of the lw as if it were the contents of register x2; this is rendered harmless by the insertion of the bubble. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

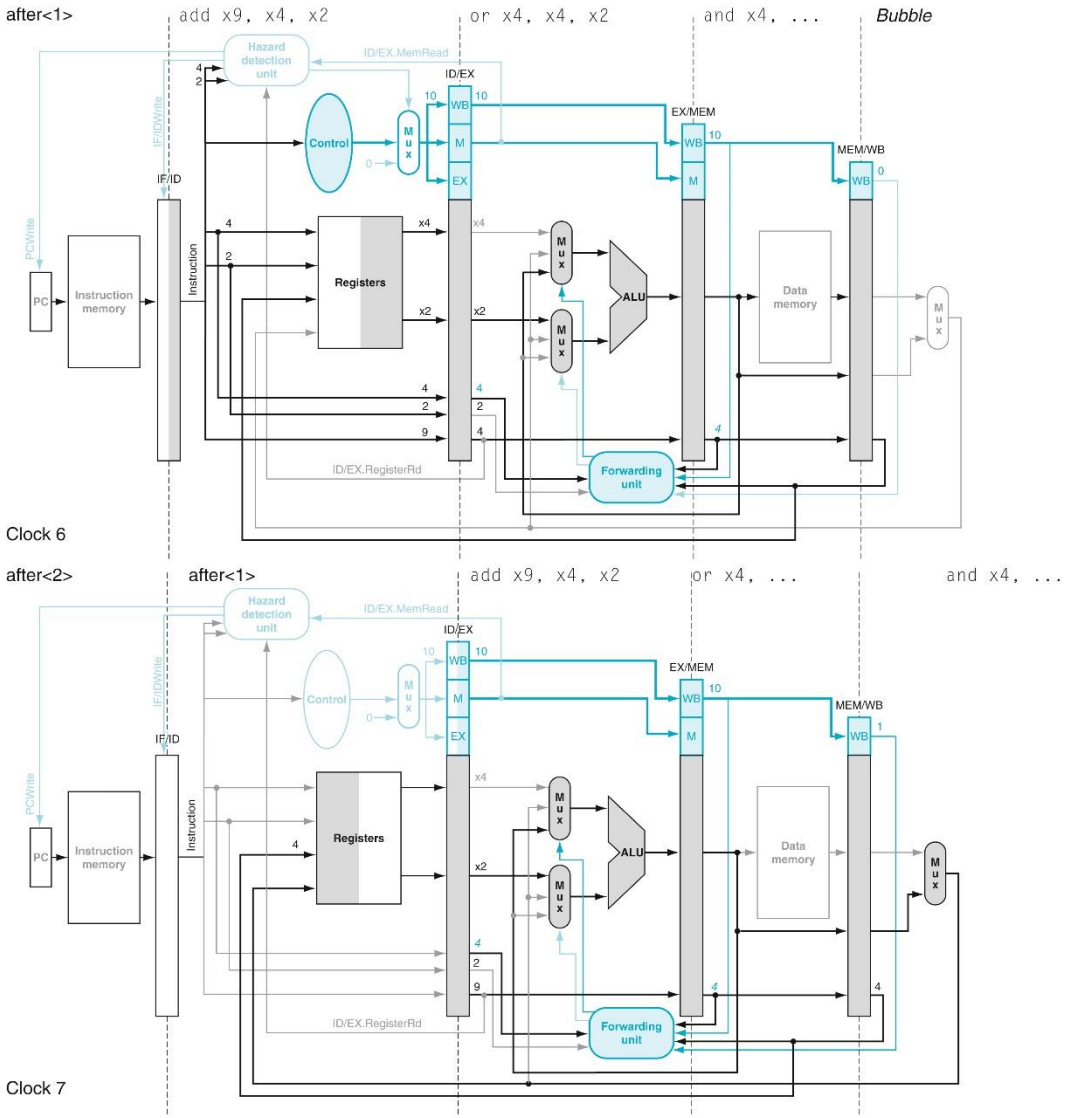


FIGURE e4.14.20 Clock cycles 6 and 7 of the instruction sequence on page 366.e26 with a load replacing sub**.** Note that unlike in Figure e4.14.17, the stall allows the lw to complete, and so there is no forwarding from MEM/WB in clock cycle 6. Register x4 for the add in the EX stage still depends on the result from or in EX/MEM, so the forwarding unit passes the result to the ALU. The bold lines show ALU input lines active in a clock cycle, and the italicized register numbers indicate a hazard. The instructions after add are shown as inactive for pedagogical reasons.

FIGURE e4.17.1 The Stretch computer, one of the first pipelined computers.

FIGURE e4.17.2 The CDC 6600, the first supercomputer.

FIGURE e4.17.3 The IBM 360/91 pushed the state of the art in pipelined execution when it was unveiled in 1966.

RegWrite	ALUSrc	ALUoperation	MemWrite	MemRead	MemToReg
true	0	“and”	false	false	0

Sign-Extend	Shift left 2
0x0000000000000014	0x0000000000000050

ALUOp	ALU Control Lines
00	0010

	No forwarding	EX/MEM	MEM/WB	Full Forwarding
CPI	1.85	1.65	1.35	1.2
Period	120	120	1.20	130
Time	222n	198n	162n	156n
Speedup	–	1.12	1.37	1.42

Cycle	1	2	3	4	5	6	7	8	
add	IF	ID	EX	ME	WB				
ld		IF	ID	EX	ME	WB			
ld			IF	ID	EX	ME	WB		
or				IF	ID	EX	ME	WB	
sd					IF	ID	EX	ME	WB

Cycle	1	2	3	4	5	6
add	IF	ID	EX	ME	WB	
ld		IF	ID	–	–	EX
ld			IF	–	–	ID

Always Taken	Always not-taken
3/5 = 60%	2/5 = 40%

Outcomes	Predictor value at time of prediction	Correct of Incorrect	Accuracy
T, NT, T, T	0,1,0,1	I,C,I,I	25%

Outcomes	Predictor value at time of prediction	Correct of Incorrect (in steady state)	Accuracy
T, NT, T, T, NT	1st occurrence: 0,1,0,1,2 2nd occurrence: 1,2,1,2,3 3rd occurrence: 2,3,2,3,3 4th occurrence: 2,3,2,3,3	C,I,C,C,I	60%

Instruction 1	Instruction 2
Invalid target address (EX)	Invalid data address (MEM)