4.14 Advanced Topic: An Introduction to Digital Design Using a Hardware Design Language to Describe and Model a Pipeline and More Pipelining Illustrations

This online section covers hardware description languages and then gives a dozen examples of pipeline diagrams, starting on page 366.e18.

As mentioned in Appendix A, Verilog can describe processors for simulation or with the intention that the Verilog specification be synthesized. To achieve acceptable synthesis results in size and speed, and a behavioral specification intended for synthesis must carefully delineate the highly combinational portions of the design, such as a datapath, from the control. The datapath can then be synthesized using available libraries. A Verilog specification intended for synthesis is usually longer and more complex.

We start with a behavioral model of the five-stage pipeline. To illustrate the dichotomy between behavioral and synthesizable designs, we then give two Verilog descriptions of a multiple-cycle-per-instruction RISC-V processor: one intended solely for simulations and one suitable for synthesis.

Using Verilog for Behavioral Specification with Simulation for the Five-Stage Pipeline

Figure e4.14.1 shows a Verilog behavioral description of the pipeline that handles ALU instructions as well as loads and stores. It does not accommodate branches (even incorrectly!), which we postpone including until later in the chapter.

Because Verilog lacks the ability to define registers with named fields such as structures in C, we use several independent registers for each pipeline register. We name these registers with a prefix using the same convention; hence, IFIDIR is the IR portion of the IFID pipeline register.

This version is a behavioral description not intended for synthesis. Instructions take the same number of clock cycles as our hardware design, but the control is done in a simpler fashion by repeatedly decoding fields of the instruction in each pipe stage. Because of this difference, the instruction register (IR) is needed throughout the pipeline, and the entire IR is passed from pipe stage to pipe stage. As you read the Verilog descriptions in this chapter, remember that the actions in the `always` block all occur in parallel on every clock cycle. Since there are no blocking assignments, the order of the events within the `always` block is arbitrary.

```
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
```

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.

```
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. (*Continued*)

Implementing Forwarding in Verilog

To extend the Verilog model further, Figure e4.14.2 shows the addition of forwarding logic for the case when the source and destination are ALU instructions. Neither load stalls nor branches are handled; we will add these shortly. The changes from the earlier Verilog description are highlighted.

Check Yourself

Someone has proposed moving the write for a result from an ALU instruction from the WB to the MEM stage, pointing out that this would reduce the maximum length of forwards from an ALU instruction by one cycle. Which of the following is accurate reasons *not to* consider such a change?

1. It would not actually change the forwarding logic, so it has no advantage.
2. It is impossible to implement this change under any circumstance since the write for the ALU result must stay in the same pipe stage as the write for a load result.
3. Moving the write for ALU instructions would create the possibility of writes occurring from two different instructions during the same clock cycle. Either an extra write port would be required on the register file or a structural hazard would be created.
4. The result of an ALU instruction is not available in time to do the write during MEM.

The Behavioral Verilog with Stall Detection

If we ignore branches, stalls for data hazards in the RISC-V pipeline are confined to one simple case: loads whose results are currently in the WB clock stage. Thus, extending the Verilog to handle a load with a destination that is either an ALU instruction or an effective address calculation is reasonably straightforward, and Figure e4.14.3 shows the few additions needed.

```
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
```

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*)

```
  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. (*Continued*)

```
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, M EMWBop; // 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;
```

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. The changes from Figure e4.14.2 are highlighted. (*Continues on next page*)

```
  // 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. (*Continued*)

Check Yourself

Someone has asked about the possibility of data hazards occurring through memory, contrary to through a register. Which of the following statements about such hazards is true?

1. Since memory accesses only occur in the MEM stage, all memory operations are done in the same order as instruction execution, making such hazards impossible in this pipeline.
2. Such hazards *are* possible in this pipeline; we just have not discussed them yet.
3. No pipeline can ever have a hazard involving memory, since it is the programmer's job to keep the order of memory references accurate.
4. Memory hazards may be possible in some pipelines, but they cannot occur in this particular pipeline.
5. Although the pipeline control would be obligated to maintain ordering among memory references to avoid hazards, it is impossible to design a pipeline where the references could be out of order.

Implementing the Branch Hazard Logic in Verilog

We can extend our Verilog behavioral model to implement the control for branches. We add the code to model branch equal using a "predict not taken" strategy. The Verilog code is shown in Figure e4.14.4. It implements the branch hazard by detecting a taken branch in ID and using that signal to squash the instruction in IF (by setting the IR to `0x00000013`, which is an effective `NOP` in RISC-V); in addition, the PC is assigned to the branch target. Note that to prevent an unexpected latch, it is important that the PC is clearly assigned on every path through the always block; hence, we assign the PC in a single *if* statement. Lastly, note that although Figure e4.14.4 incorporates the basic logic for branches and control hazards, supporting branches requires additional bypassing and data hazard detection, which we have not included.

Using Verilog for Behavioral Specification with Synthesis

To demonstrate the contrasting types of Verilog, we show two descriptions of a different, nonpipelined implementation style of RISC-V that uses multiple clock cycles per instruction. (Since some instructors make a synthesizable description of the RISC-V pipeline project for a class, we chose not to include it here. It would also be long.)

Figure e4.14.5 gives a behavioral specification of a multicycle implementation of the RISC-V processor. Because of the use of behavioral operations, it would be difficult to synthesize a separate datapath and control unit with any reasonable efficiency. This version demonstrates another approach to the control by using a Mealy finite-state machine (see discussion in Section A.10 of Appendix A). The use of a Mealy machine, which allows the output to depend both on inputs and the current state, allows us to decrease the total number of states.

```
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:
```

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. The changes from Figure e4.14.2 are highlighted. (*Continues on next page*)

```
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
```

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. (*Continued*)

```
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. (*Continued*)

Since a version of the RISC-V design intended for synthesis is considerably more complex, we have relied on a number of Verilog modules that were specified in Appendix A, including the following:

- The 4-to-1 multiplexor shown in Figure A.4.2, and the 2-to-1 multiplexor that can be trivially derived based on the 4-to-1 multiplexor.
- The RISC-V ALU shown in Figure A.5.15.
- The RISC-V ALU control defined in Figure A.5.16.
- The RISC-V register file defined in Figure A.8.11.

Now, let's look at a Verilog version of the RISC-V processor intended for synthesis. Figure e4.14.6 shows the structural version of the RISC-V datapath. Figure e4.14.7 uses the datapath module to specify the RISC-V CPU. This version also demonstrates another approach to implementing the control unit, as well as some optimizations that rely on relationships between various control signals. Observe that the state machine specification only provides the sequencing actions.

The setting of the control lines is done with a series of `assign` statements that depend on the state as well as the opcode field of the instruction register. If one were to fold the setting of the control into the state specification, this would look like a Mealy-style finite-state control unit. Because the setting of the control lines is specified using `assign` statements outside of the `always` block, most logic synthesis systems will generate a small implementation of a finite-state machine

```
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
```

FIGURE e4.14.5 A behavioral specification of the multicycle RISC-V design. This has the same cycle behavior as the multicycle design, but is purely for simulation and specification. It cannot be used for synthesis. (*Continues on next page*)

```
          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. (*Continued*)

that determines the setting of the state register and then uses external logic to derive the control inputs to the datapath.

In writing this version of the control, we have also taken advantage of a number of insights about the relationship between various control signals as well as situations where we don't care about the control signal value; some examples of these are given in the following elaboration.

More Illustrations of Instruction Execution on the Hardware

To reduce the cost of this book, starting with the third edition, we moved sections and figures that were used by a minority of instructors online. This subsection recaptures those figures for readers who would like more supplemental material to understand pipelining better. These are all single-clock-cycle pipeline diagrams, which take many figures to illustrate the execution of a sequence of instructions.

The three examples are respectively for code with no hazards, an example of forwarding on the pipelined implementation, and an example of bypassing on the pipelined implementation.

```
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
```

FIGURE e4.14.6 A Verilog version of the multicycle RISC-V datapath that is appropriate for synthesis. This datapath relies on several units from Appendix A. Initial statements do not synthesize, and a version used for synthesis would have to incorporate a reset signal that had this effect. Also note that resetting R0 to 0 on every clock is not the best way to ensure that R0 stays at 0; instead, modifying the register file module to produce 0 whenever R0 is read and to ignore writes to R0 would be a more efficient solution. (*Continues on next page*)

```
  // 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. (*Continued*)

No Hazard Illustrations

On page 285, we gave the example code sequence:

```
lw     x10, 40(x1)
sub    x11, x2, x3
add    x12, x3, x4
lw     x13, 48(x1)
add    x14, x5, x6
```

Figures 4.59 and 4.60 showed the multiple-clock-cycle pipeline diagrams for this two-instruction sequence executing across six clock cycles. Figures e4.14.8 through e4.14.10 show the corresponding single-clock-cycle pipeline diagrams for these two instructions. Note that the order of the instructions differs between these two types of diagrams: the newest instruction is at the *bottom and to the right* of the multiple-clock-cycle pipeline diagram, and it is on the *left* in the single-clock-cycle pipeline diagram.

More Examples

To understand how pipeline control works, let's consider these five instructions going through the pipeline:

```
lw      x10, 40(x1)
sub     x11, x2, x3
and     x12, x4, x5
or      x13, x6, x7
add     x14, x8, x9
```

```
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.

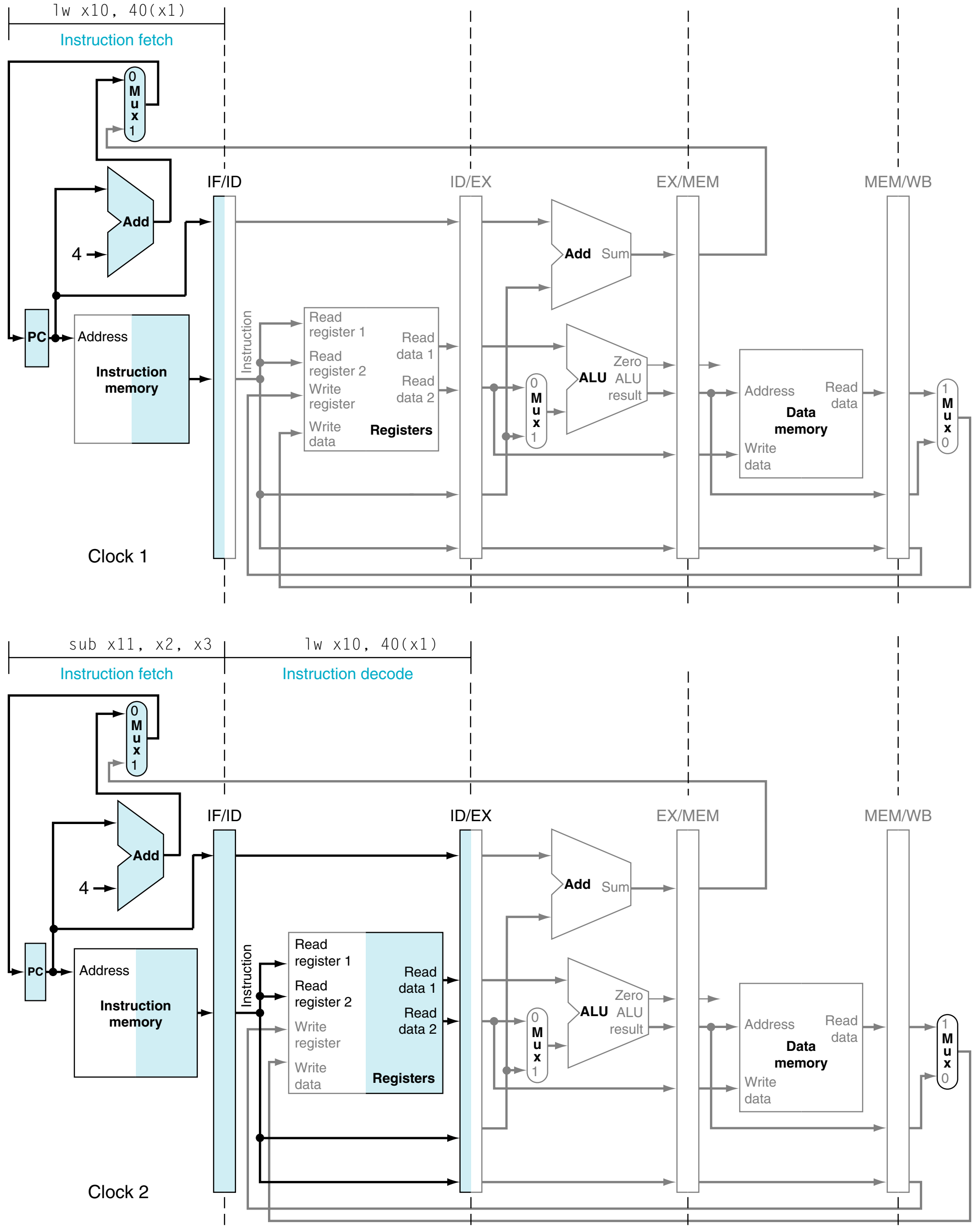


FIGURE e4.14.8 Single-cycle pipeline diagrams for clock cycles 1 (top diagram) and 2 (bottom diagram). This style of pipeline representation is a snapshot of every instruction executing during one clock cycle. Our example has but two instructions, so at most two stages are identified in each clock cycle; 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.

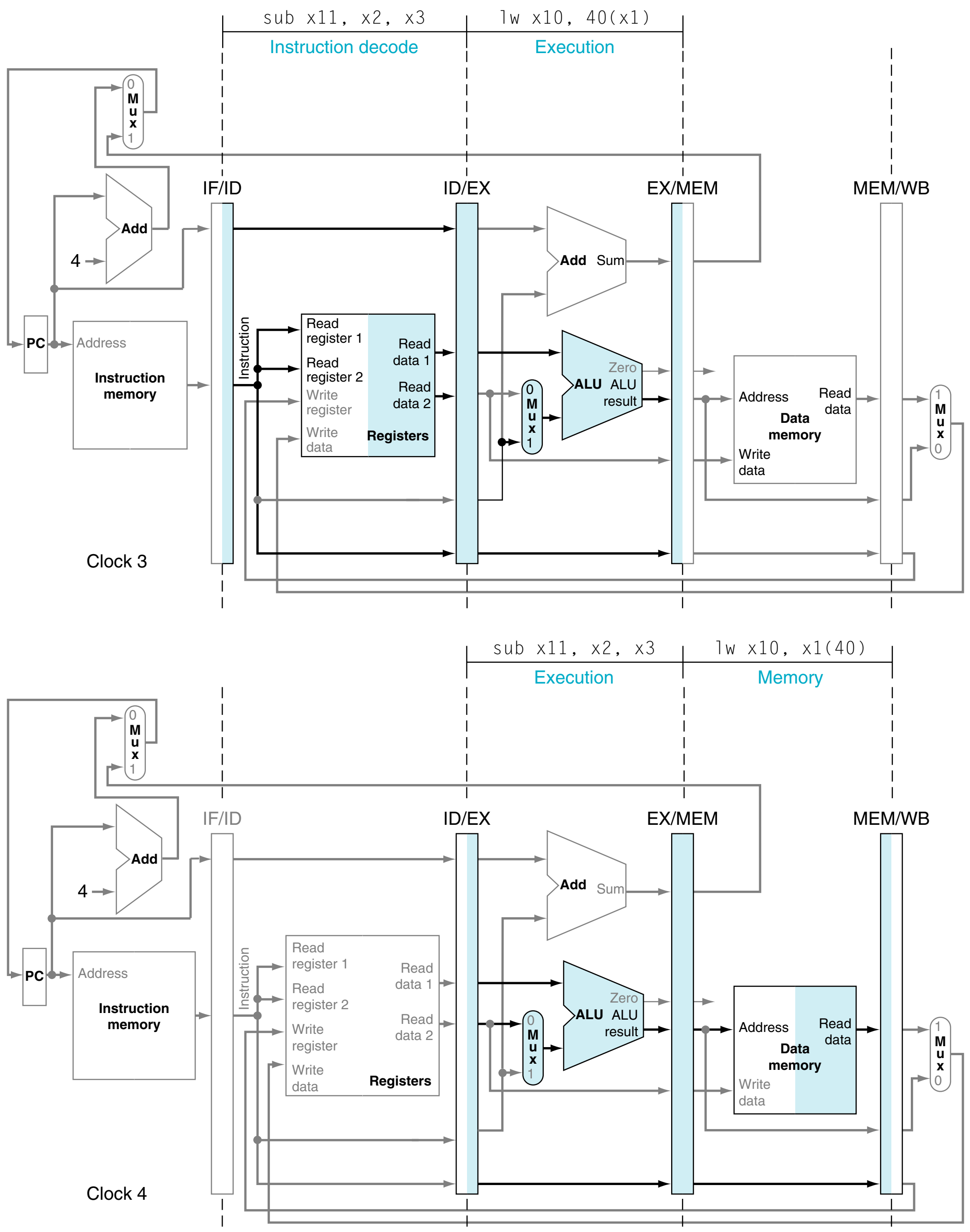


FIGURE e4.14.9 Single-cycle pipeline diagrams for clock cycles 3 (top diagram) and 4 (bottom diagram). In the third clock cycle in the top diagram, `lw` enters the EX stage. At the same time, `sub` enters ID. In the fourth clock cycle (bottom datapath), `lw` moves into MEM stage, reading memory using 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.

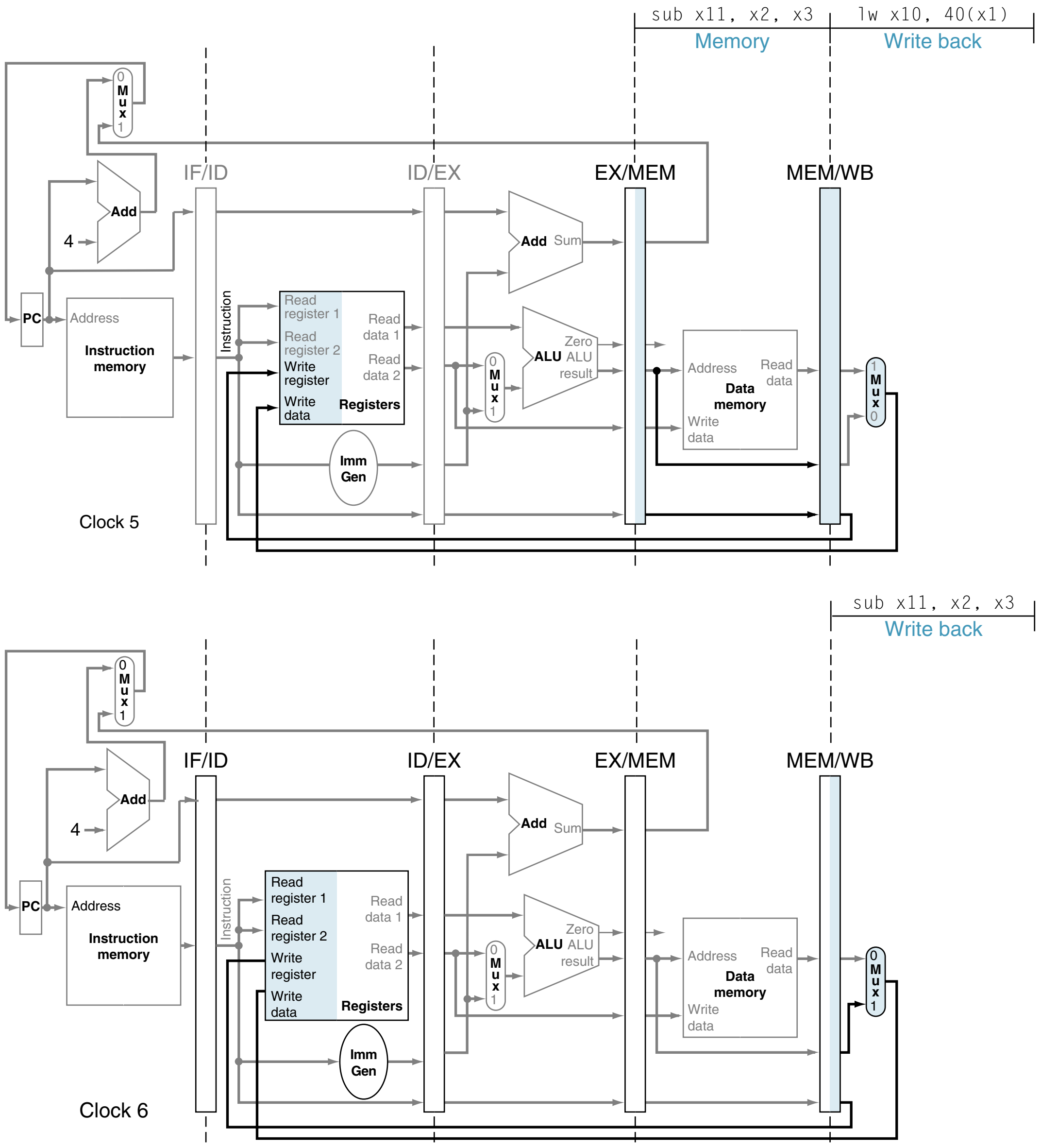


FIGURE e4.14.10 Single-cycle pipeline diagrams for clock cycles 5 (top diagram) and 6 (bottom diagram). In clock cycle 5, `lw` completes by writing the data in MEM/WB into register 10, and `sub` sends the difference in EX/MEM to MEM/WB. In the next clock cycle, `sub` writes the value in MEM/WB to register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

Figures e4.14.11 through e4.14.15 show these instructions proceeding through the nine clock cycles it takes them to complete execution, highlighting what is active in a stage and identifying the instruction associated with each stage during a clock cycle. If you examine them carefully, you may notice:

- In Figure e4.14.13 you can see the sequence of the destination register numbers from left to right at the bottom of the pipeline registers. The numbers advance to the right during each clock cycle, with the MEM/WB pipeline register supplying the number of the register written during the WB stage.
- When a stage is inactive, the values of control lines that are deasserted are shown as 0 or X (for don't care).
- Sequencing of control is embedded in the pipeline structure itself. First, all instructions take the same number of clock cycles, so there is no special control for instruction duration. Second, all control information is computed during instruction decode and then passed along by the pipeline registers.

Forwarding Illustrations

We can use the single-clock-cycle pipeline diagrams to show how forwarding operates, as well as how the control activates the forwarding paths. Consider the following code sequence in which the dependences have been highlighted:

```
sub  x2, x1, x3
and  x4, x2, x5
or   x4, x4, x2
add  x9, x4, x2
```

Figures e4.14.16 and e4.14.17 show the events in clock cycles 3–6 in the execution of these instructions.

Thus, in clock cycle 5, the forwarding unit selects the EX/MEM pipeline register for the upper input to the ALU and the MEM/WB pipeline register for the lower input to the ALU. The following `add` instruction reads both register `x4`, the target of the `and` instruction, and register `x2`, which the `sub` instruction has already written. Notice that the prior two instructions both write register `x4`, so the forwarding unit must pick the immediately preceding one (MEM stage).

In clock cycle 6, the forwarding unit thus selects the EX/MEM pipeline register, containing the result of the `or` instruction, for the upper ALU input but uses the non-forwarding register value for the lower input to the ALU.

Illustrating Pipelines with Stalls and Forwarding

We can use the single-clock-cycle pipeline diagrams to show how the control for stalls works. Figures e4.14.18 through e4.14.20 show the single-cycle diagram for clocks 2 through 7 for the following code sequence (dependences highlighted):

```
lw    x2, 40(x1)
and   x4, x2, x5
or    x4, x4, x2
add   x9, x4, x2
```

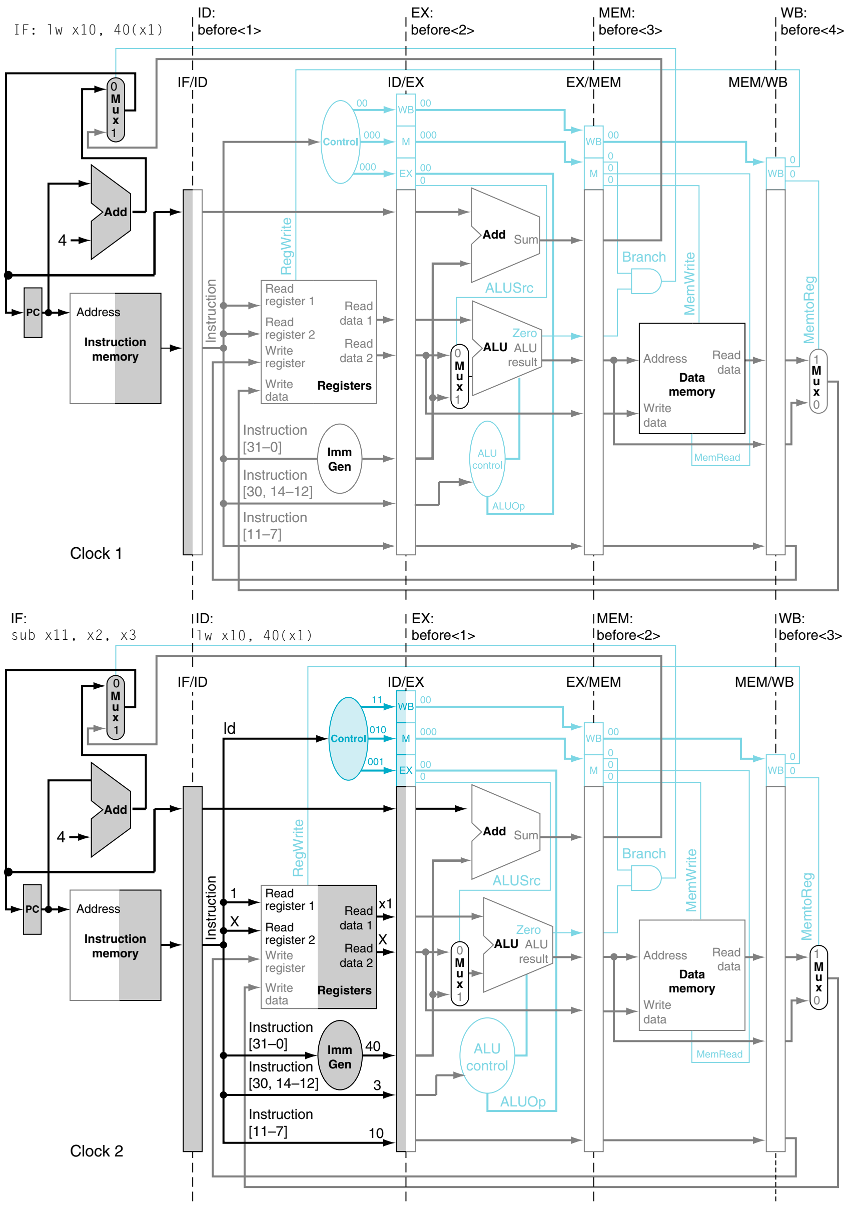


FIGURE e4.14.11 Clock cycles 1 and 2. The phrase "*before <i>*" means the *i*th instruction before `lw`. The `lw` instruction in the top datapath is in the IF stage. At the end of the clock cycle, the `lw` instruction is in the IF/ID pipeline registers. In the second clock cycle, seen in the bottom datapath, 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.

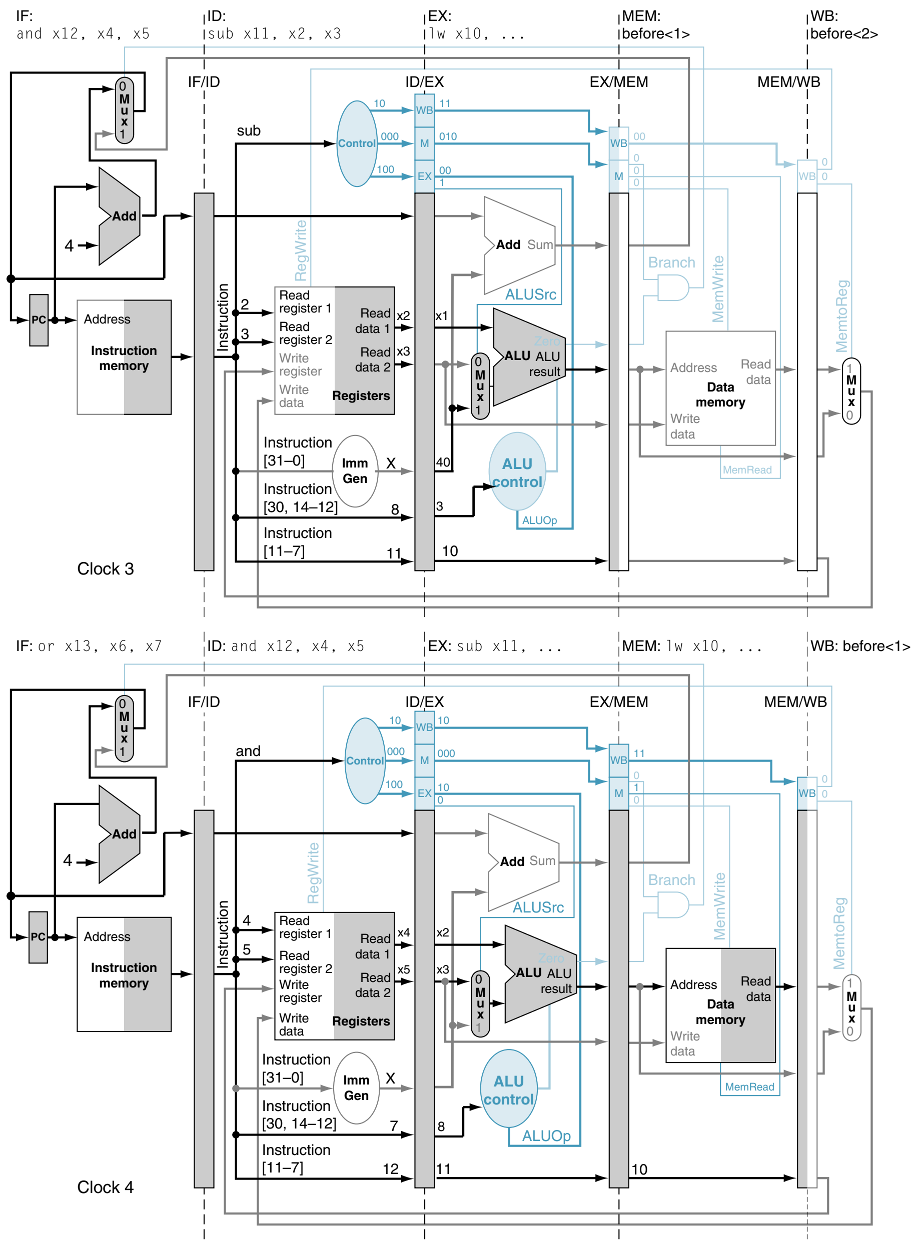


FIGURE e4.14.12 Clock cycles 3 and 4. In the top diagram, lw enters the EX stage in the third clock cycle, adding x1 and 40 to form the address in the EX/MEM pipeline register. (The lw instruction is written lw x10, … upon reaching EX, because the identity of instruction operands is not needed by EX 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.

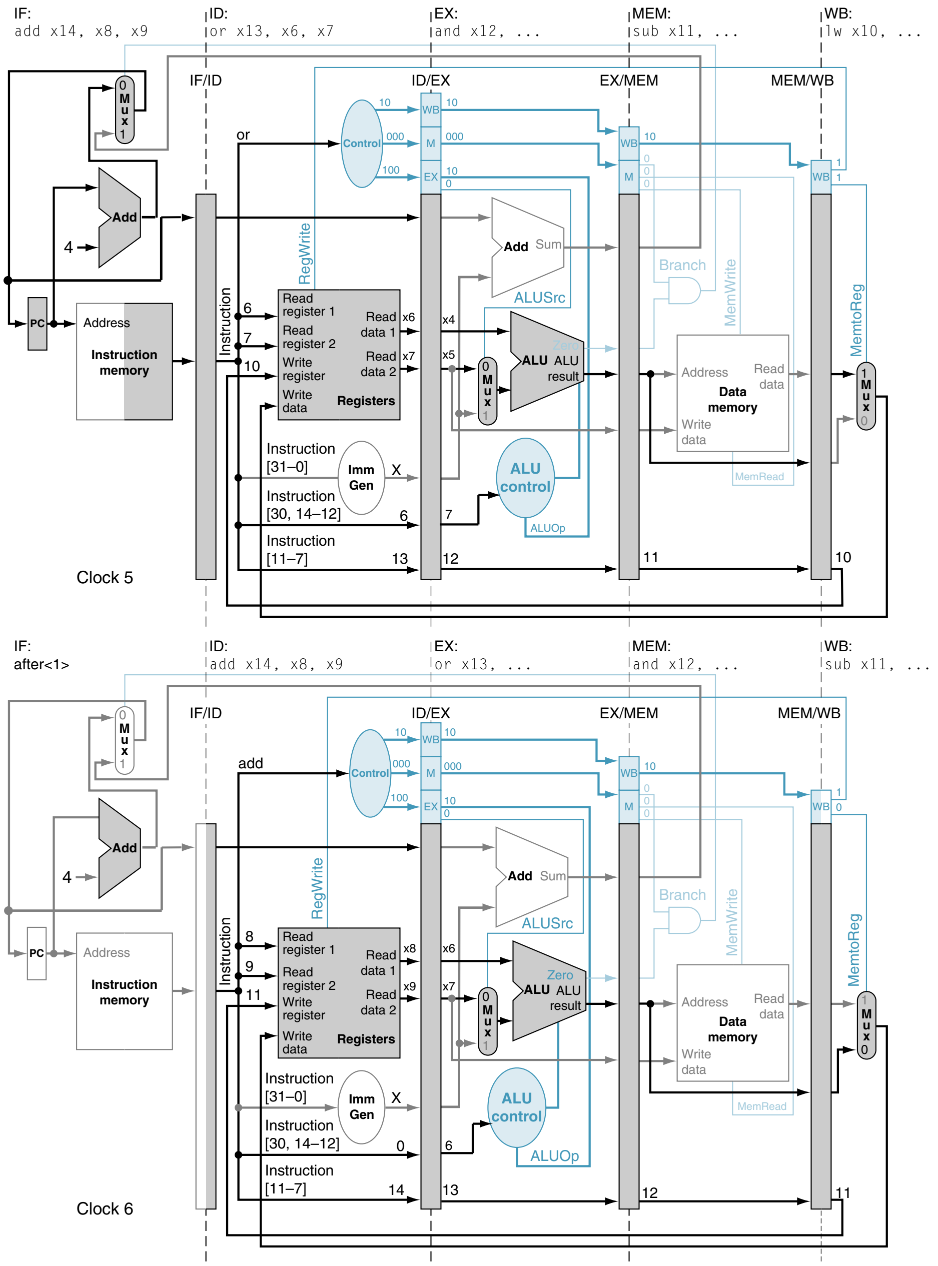


FIGURE e4.14.13 Clock cycles 5 and 6. With add, the final instruction in this example, entering IF in the top datapath, all instructions are engaged. By writing the data in MEM/WB into register 10, lw completes; both the data and the register number are in MEM/WB. In the same clock cycle, sub sends the 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.

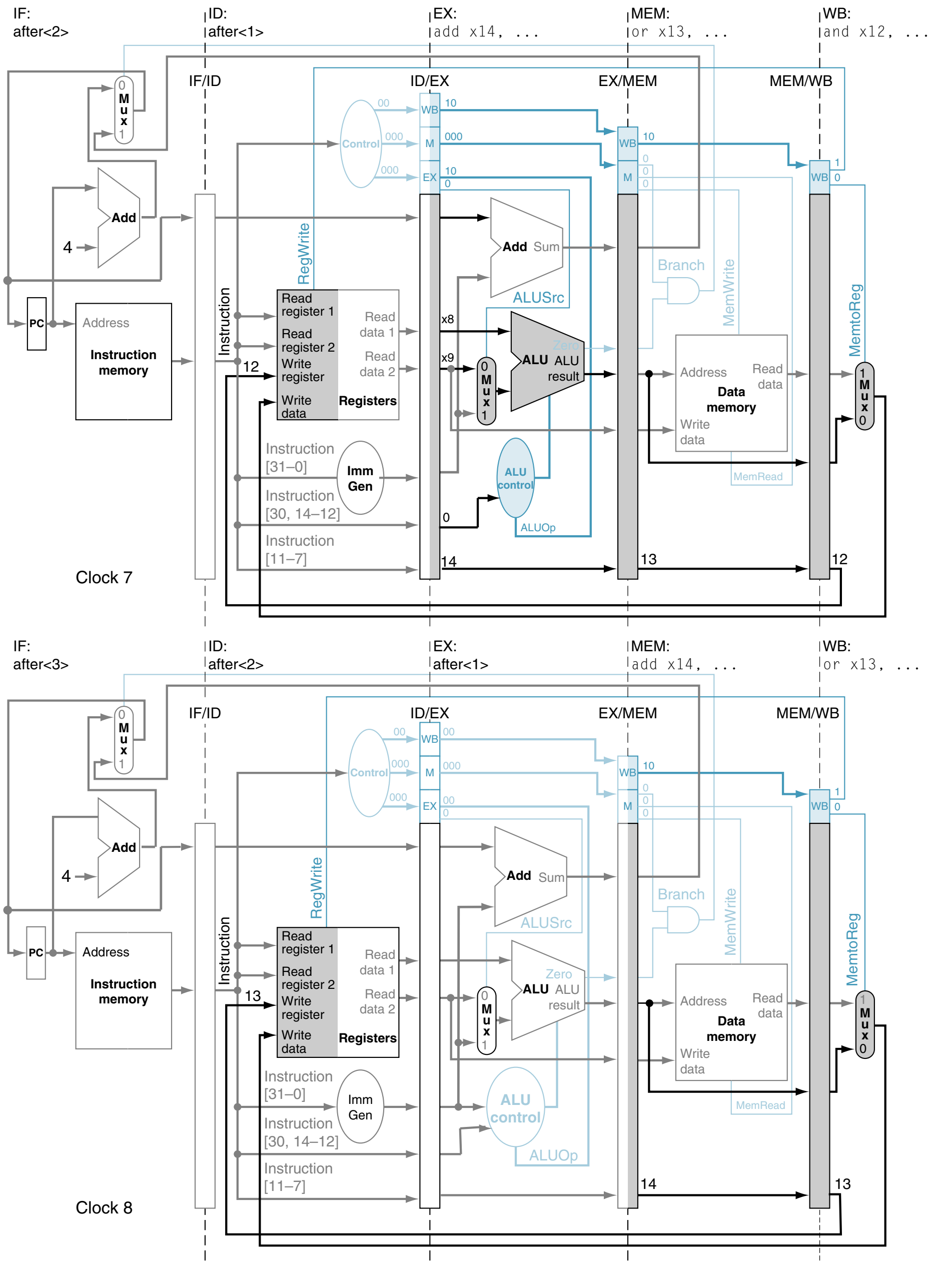


FIGURE e4.14.14 Clock cycles 7 and 8. In the top datapath, the `add` instruction brings up the rear, adding the values corresponding to registers `x8` and `x9` during the EX stage. The result of the `or` instruction is passed from EX/MEM to MEM/WB in the MEM stage, and the WB stage writes the result 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.

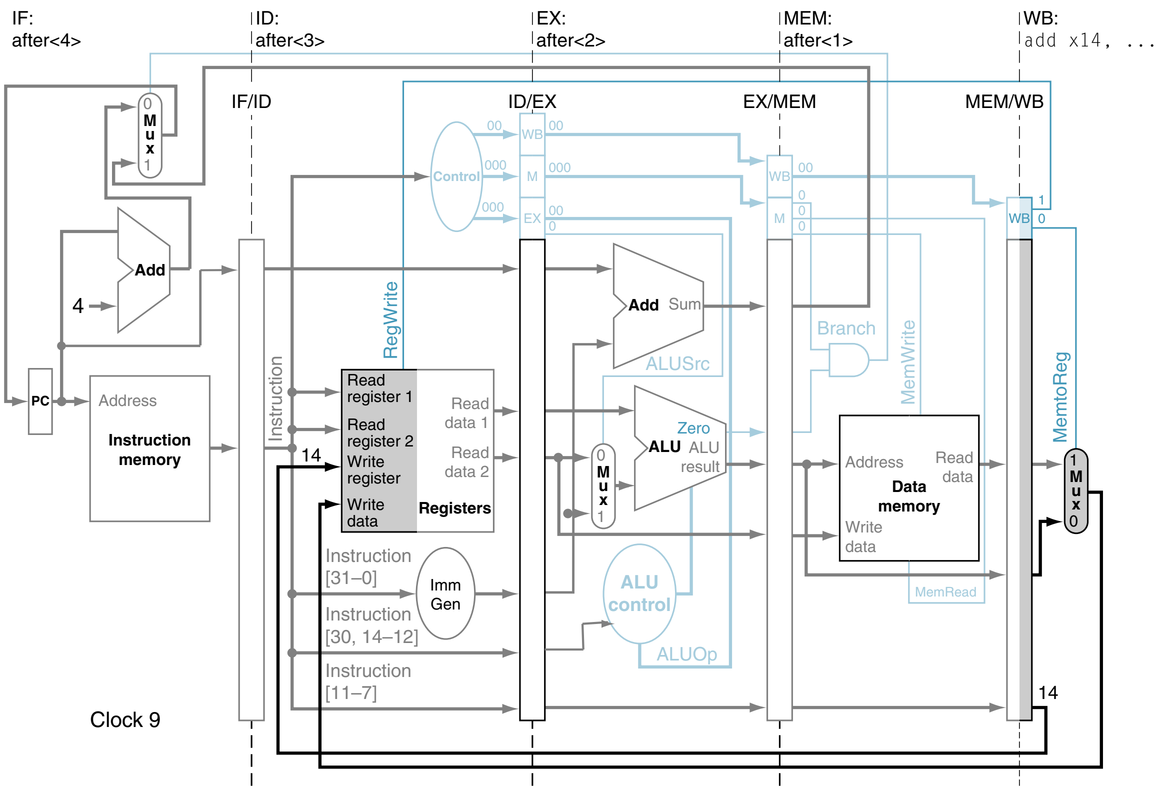
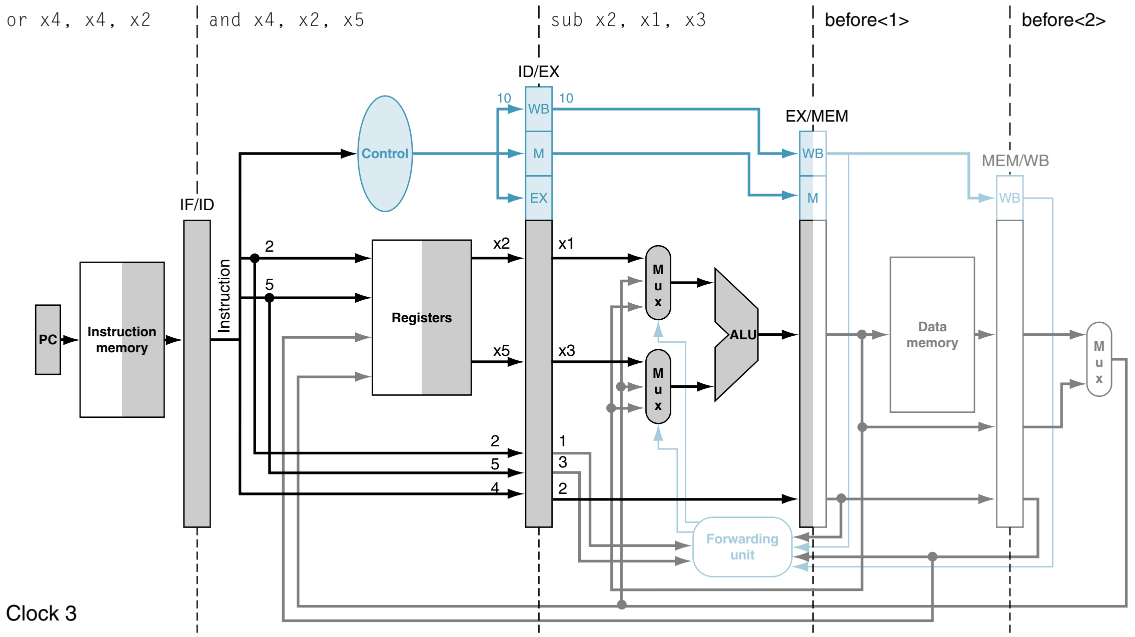


FIGURE e4.14.15 Clock cycle 9. The WB stage writes the ALU result in MEM/WB into register `x14`, completing `add` and the five-instruction sequence. The instructions after `add` are shown as inactive for pedagogical reasons.



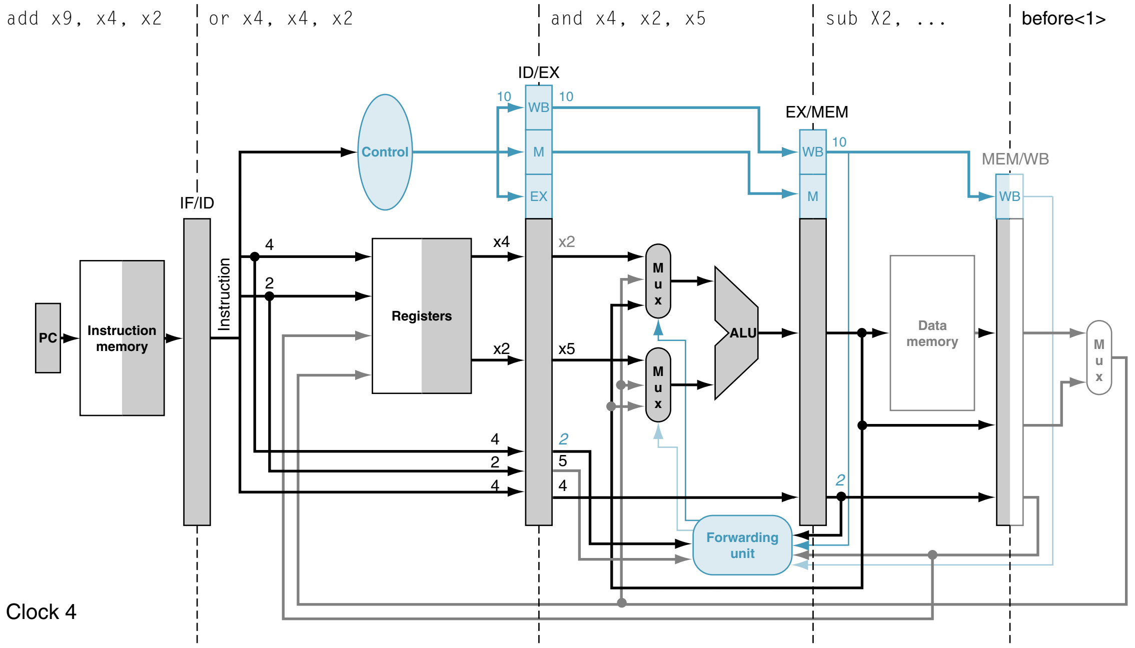


FIGURE e4.14.16 Clock cycles 3 and 4 of the instruction sequence on page 366.e26. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard. The forwarding unit is highlighted by shading it when it is forwarding data to the ALU. The instructions before `sub` are shown as inactive just to emphasize what occurs for the four instructions 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.

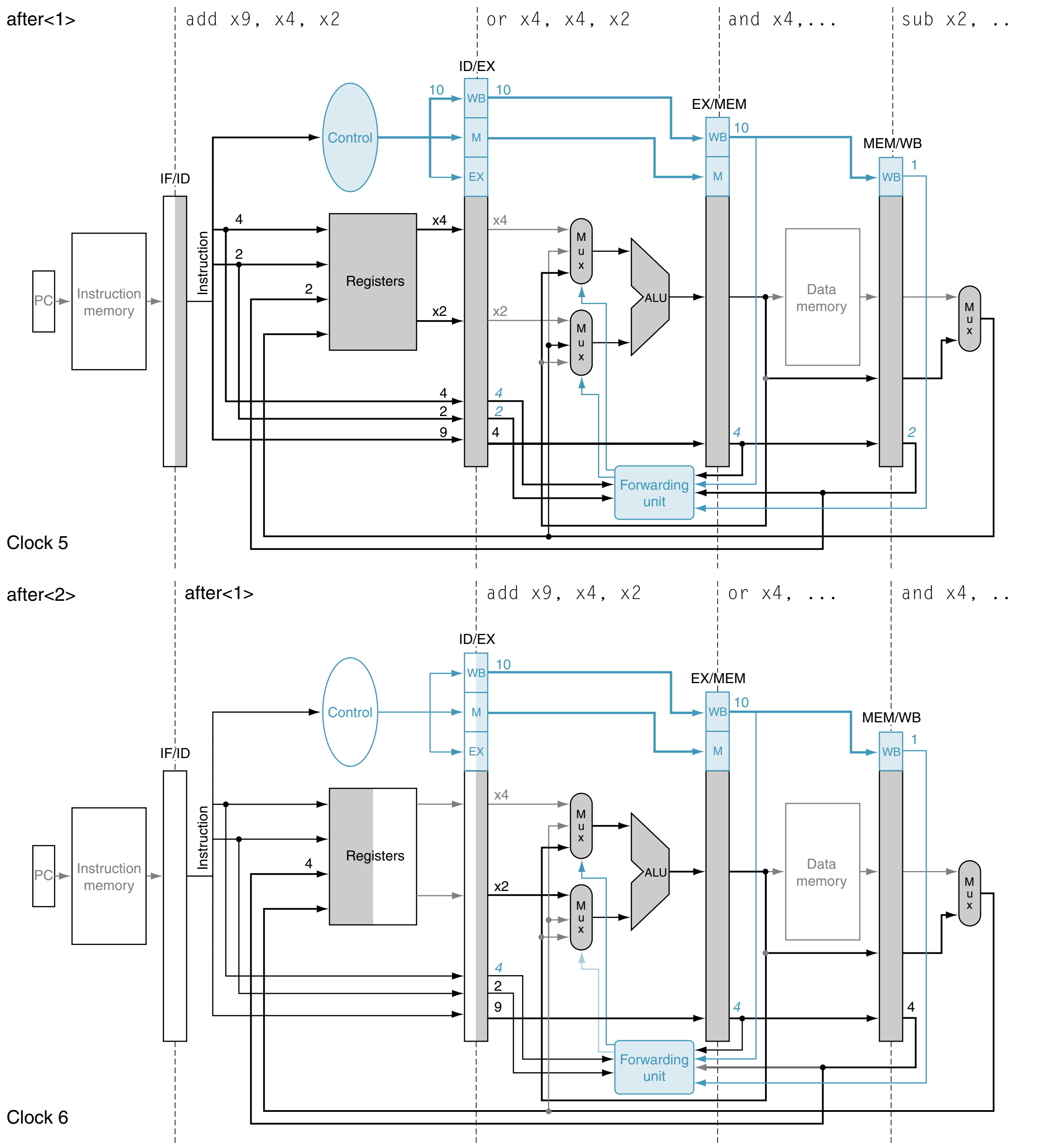


FIGURE e4.14.17 Clock cycles 5 and 6 of the instruction sequence on page 366.e26. The forwarding unit is highlighted when it is forwarding data to the ALU. The two instructions after `add` are shown as inactive just to emphasize what occurs for the four instructions in the example. The bold lines are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

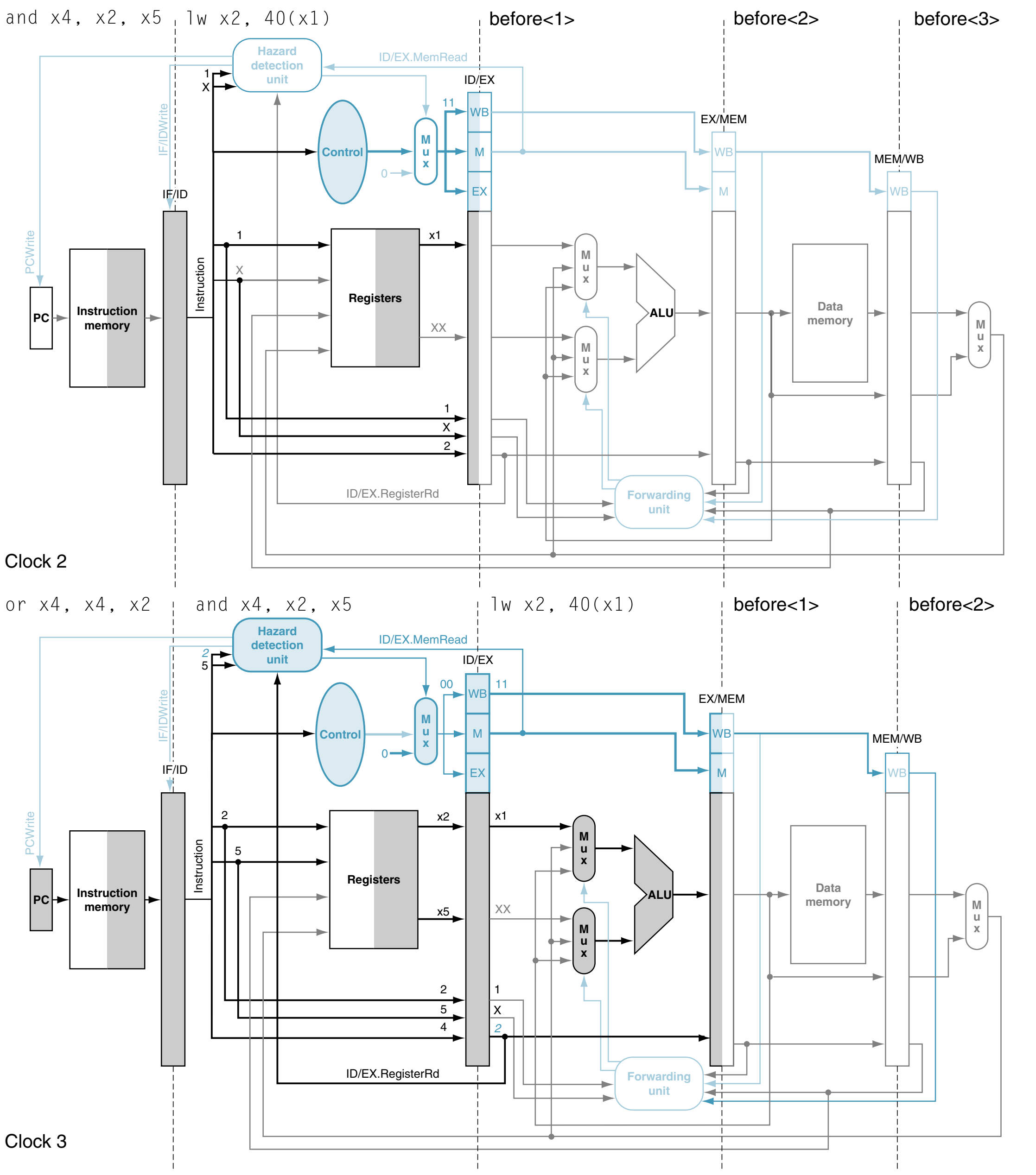


FIGURE e4.14.18 Clock cycles 2 and 3 of the instruction sequence on page 366.e26 with a load replacing `sub`. The bold lines are those active in a clock cycle, the italicized register numbers in color indicate a hazard, and the . . . in the place of operands means that their identity is information not 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.

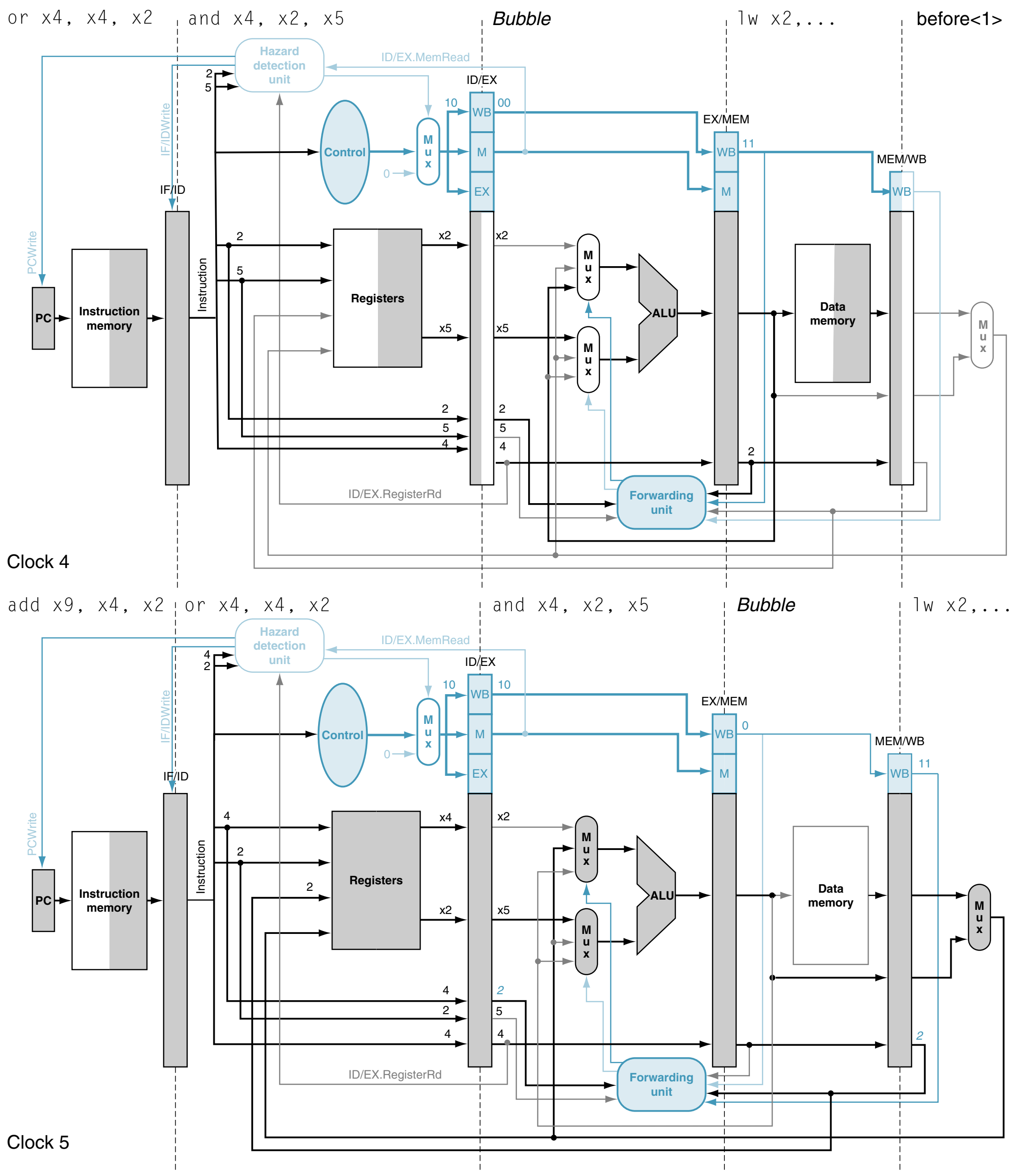


FIGURE e4.14.19 Clock cycles 4 and 5 of the instruction sequence on page 366.e26 with a load replacing `sub`. The bubble is inserted in the pipeline in clock cycle 4, and then the `and` instruction is allowed to proceed in clock cycle 5. The forwarding unit is highlighted in clock cycle 5 because it 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.

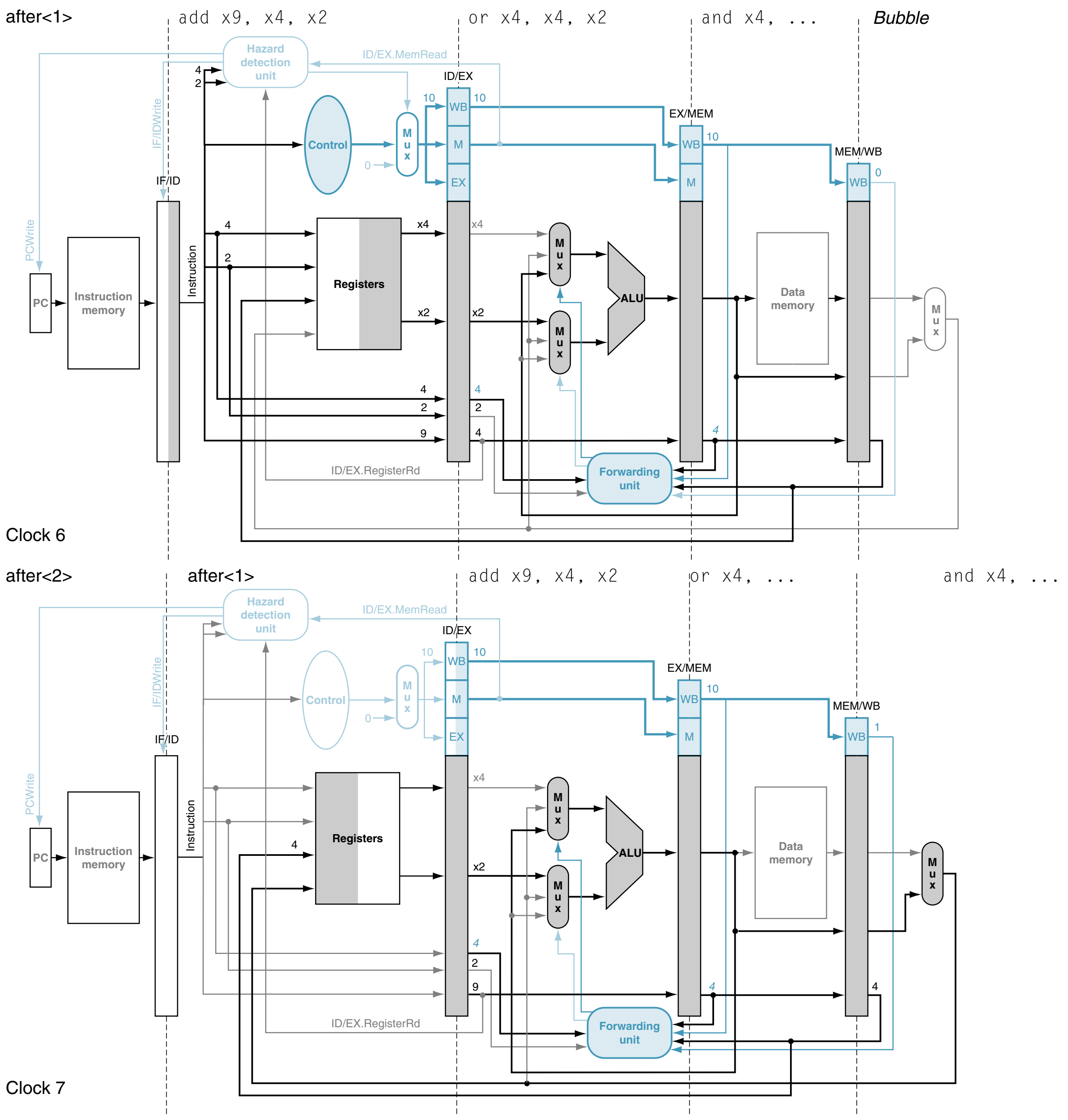


FIGURE e4.14.20 Clock cycles 6 and 7 of the instruction sequence on page 366.e26 with a load replacing `sub`. Note that unlike in Figure e4.14.17, the stall allows the `lw` to complete, and so there is no forwarding from MEM/WB in clock cycle 6. Register `x4` for the `add` in the EX stage still depends 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.