4.14 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 decription languages and then gives a dozen examples of pipeline diagrams.

As mentioned in Appendix C, 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 5-stage pipeline. To illustrate the dichotomy between behavioral and synthesizeable designs, we then give two Verilog descriptions of a multiple-cycle-per-instruction MIPS processor: one intended solely for simulations and one suitable for synthesis.

Using Verilog for Behavioral Specification with Simulation for the 5-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 CPU (clock);
  // Instruction opcodes
   parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, no-op = 32'b00000_100000, ALUop = 6'b0;
   input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
            IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
            EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd, MEMWBrt; // Access register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; // Access opcodes
wire [31:0] Ain, Bin; // the ALU inputs
// These assignments define fields from the pipeline registers
   assign IDEXrs = IDEXIR[25:21];   // rs field
   assign IDEXrt = IDEXIR[20:16];   // rt field
   assign EXMEMrd = EXMEMIR[15:11]; // rd field
   assign MEMWBrd = MEMWBIR[15:11]; //rd field
   assign MEMWBrt = MEMWBIR[20:16]; //rt field--used for loads
   assign EXMEMop = EXMEMIR[31:26]; // the opcode
   assign MEMWBop = MEMWBIR[31:26]; // the opcode
   assign IDEXop = IDEXIR[31:26];   // the opcode
   // Inputs to the ALU come directly from the ID/EX pipeline registers
   assign Ain = IDEXA;
   assign Bin = IDEXB;
   reg [5:0] i; //used to initialize registers
   initial begin
       PC = 0;
      IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
       for (i=0;i<=31;i=i+1) Regs[i] = i; //initialize registers--just so they aren't cares
   end
   always @ (posedge clock) begin
   // Remember that ALL these actions happen every pipe stage and with the use of <= they happen in parallel!
   // first instruction  in the pipeline is being fetched
          IFIDIR <= IMemory[PC>>2];
          PC <= PC + 4;
      end // Fetch & increment PC
      // second instruction in pipeline is fetching registers
         IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // 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) |(IDEXop==SW))  // address calculation
           EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
         else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
              32: EXMEMALUOut <= Ain + Bin;  //add operation
              default: ; //other R-type operations: subtract, SLT, etc.
         endcase
```

FIGURE e4.14.1 A Verilog behavorial model for the MIPS 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. (*continues on next page*)

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

FIGURE e4.14.1 A Verilog behavorial model for the MIPS five-stage pipeline, ignoring branch and data hazards. (*Continued*)

Implementing Forwarding in Verilog

To further extend the Verilog model, 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 are 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.

```
module CPU (clock);
parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, no-op = 32'b00000_100000, ALUop = 6'b0;
input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
            IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
            EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd, MEMWBrt; //hold register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; Hold opcodes
   wire [31:0] Ain, Bin;

// declare the bypass signals
   wire bypassAfromMEM, bypassAfromALUinWB,bypassBfromMEM, bypassBfromALUinWB,
        bypassAfromLWinWB, bypassBfromLWinWB;

   assign IDEXrs = IDEXIR[25:21];    assign IDEXrt = IDEXIR[15:11];    assign EXMEMrd = EXMEMIR[15:11];
   assign MEMWBrd = MEMWBIR[20:16]; assign EXMEMop = EXMEMIR[31:26];
   assign MEMWBrt = MEMWBIR[25:20];
   assign MEMWBop = MEMWBIR[31:26];  assign IDEXop = IDEXIR[31:26];

   // The bypass to input A from the MEM stage for an ALU operation
   assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop); // yes, bypass

   // The bypass to input B from the MEM stage for an ALU operation
   assign bypassBfromMEM = (IDEXrt == EXMEMrd)&(IDEXrt!=0) & (EXMEMop==ALUop); // yes, bypass

   // The bypass to input A from the WB stage for an ALU operation
   assign bypassAfromALUinWB =( IDEXrs == MEMWBrd) & (IDEXrs!=0) & (MEMWBop==ALUop);

   // The bypass to input B from the WB stage for an ALU operation
   assign bypassBfromALUinWB = (IDEXrt == MEMWBrd) & (IDEXrt!=0) & (MEMWBop==ALUop); /

   // The bypass to input A from the WB stage for an LW operation
   assign bypassAfromLWinWB =( IDEXrs == MEMWBIR[20:16]) & (IDEXrs!=0) & (MEMWBop==LW);

   // The bypass to input B from the WB stage for an LW operation
   assign bypassBfromLWinWB = (IDEXrt == MEMWBIR[20:16]) & (IDEXrt!=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 | bypassAfromLWinWB)? 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 | bypassBfromLWinWB)? MEMWBValue: IDEXB;

   reg [5:0] i; //used to initialize registers

   initial begin
       PC = 0;
      IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
       for (i = 0;i<=31;i = i+1) Regs[i] = i; //initialize registers--just so they aren't cares
   end

   always @ (posedge clock) begin

      // first instruction in the pipeline is being fetched

          IFIDIR <= IMemory[PC>>2];
          PC <= PC + 4;
      end // Fetch & increment PC
```

FIGURE e4.14.2 A behavioral definition of the five-stage MIPS 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*)

```
      // second instruction is in register fetch
         IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // 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) |(IDEXop==SW))  // address calculation & copy B
EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
           32: 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
      // the WB stage
      if ((MEMWBop==ALUop) & (MEMWBrd != 0)) Regs[MEMWBrd] <= MEMWBValue; // ALU operation
      else if ((EXMEMop == LW)& (MEMWBrt != 0)) Regs[MEMWBrt] <= MEMWBValue;
   end
endmodule
```

FIGURE e4.14.2 A behavioral definition of the five-stage MIPS pipeline with bypassing to ALU operations and address calculations. (*Continued*)

The Behavioral Verilog with Stall Detection

If we ignore branches, stalls for data hazards in the MIPS 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 4.13.3 shows the few additions needed.

Check Yourself

Someone has asked about the possibility of data hazards occurring through memory, as opposed to through a register. Which of the following statements about such hazards are 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.

```
module CPU (clock);
parameter LW = 6’b100011, SW = 6’b101011, BEQ = 6’b000100, no-op = 32’b00000_100000, ALUop = 6’b0;
input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
             EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd, MEMWBrt; //hold register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; Hold opcodes
   wire [31:0] Ain, Bin;

// declare the bypass signals
   wire stall, bypassAfromMEM, bypassAfromALUinWB,bypassBfromMEM, bypassBfromALUinWB,
        bypassAfromLWinWB, bypassBfromLWinWB;

   assign IDEXrs = IDEXIR[25:21];    assign IDEXrt = IDEXIR[15:11];    assign EXMEMrd = EXMEMIR[15:11];
   assign MEMWBrd = MEMWBIR[20:16]; assign EXMEMop = EXMEMIR[31:26];
      assign MEMWBrt = MEMWBIR[25:20];
   assign MEMWBop = MEMWBIR[31:26];  assign IDEXop = IDEXIR[31:26];
   // The bypass to input A from the MEM stage for an ALU operation
   assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop); // yes, bypass
   // The bypass to input B from the MEM stage for an ALU operation
   assign bypassBfromMEM = (IDEXrt== EXMEMrd)&(IDEXrt!=0) & (EXMEMop==ALUop); // yes, bypass
   // The bypass to input A from the WB stage for an ALU operation
   assign bypassAfromALUinWB =( IDEXrs == MEMWBrd) & (IDEXrs!=0) & (MEMWBop==ALUop);
   // The bypass to input B from the WB stage for an ALU operation
   assign bypassBfromALUinWB = (IDEXrt==MEMWBrd) & (IDEXrt!=0) & (MEMWBop==ALUop); /
   // The bypass to input A from the WB stage for an LW operation
   assign bypassAfromLWinWB =( IDEXrs ==MEMWBIR[20:16]) & (IDEXrs!=0) & (MEMWBop==LW);
   // The bypass to input B from the WB stage for an LW operation
   assign bypassBfromLWinWB = (IDEXrt==MEMWBIR[20:16]) & (IDEXrt!=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 | bypassAfromLWinWB)? 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 | bypassBfromLWinWB)? MEMWBValue: IDEXB;

   // The signal for detecting a stall based on the use of a result from LW
   assign stall = (MEMWBIR[31:26]==LW) && // source instruction is a load
        ((((IDEXop==LW)|(IDEXop==SW)) && (IDEXrs==MEMWBrd)) | // stall for address calc
   ((IDEXop==ALUop) && ((IDEXrs==MEMWBrd)|(IDEXrt==MEMWBrd)))); // ALU use

   reg [5:0] i; //used to initialize registers

   initial begin
     PC = 0;
    IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
     for (i = 0;i<=31;i = i+1) Regs[i] = i; //initialize registers--just so they aren’t cares
   end

   always @ (posedge clock) begin

     if (~stall) begin // the first three pipeline stages stall if there is a load hazard
```

FIGURE e4.14.3 A behavioral definition of the five-stage MIPS 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*)

```
      // first instruction  in the pipeline is being fetched
          IFIDIR <= IMemory[PC>>2];
          PC <= PC + 4;

         IDEXIR <= IFIDIR;  //pass along IR--can happen anywhere, since this affects next stage only!

      // second instruction is in register fetch
       IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // get two registers

      // third instruction is doing address calculation or ALU operation
          if ((IDEXop==LW) |(IDEXop==SW))  // address calculation & copy B
                 EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
         else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
              32: 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 <= no-op; /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

      // the WB stage

      if ((MEMWBop==ALUop) & (MEMWBrd != 0)) Regs[MEMWBrd] <= MEMWBValue; // ALU operation

      else if ((EXMEMop == LW)& (MEMWBrt != 0)) Regs[MEMWBrt] <= MEMWBValue;

   end
endmodule
```

FIGURE e4.14.3 A behavioral definition of the five-stage MIPS pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. (*Continued*)

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 0, which is an effective `no-op` in MIPS-32); 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, the incorporation of branches requires additional bypassing and data hazard detection, which we have not included.

```
module CPU (clock);
parameter LW = 6'b100011, SW = 6'b101011, BEQ = 6'b000100, no-op = 32'b0000000_0000000_0000000_0000000, ALUop = 6'b0;
input clock;
   reg[31:0] PC, Regs[0:31], IMemory[0:1023], DMemory[0:1023], // separate memories
             IFIDIR, IDEXA, IDEXB, IDEXIR, EXMEMIR, EXMEMB, // pipeline registers
             EXMEMALUOut, MEMWBValue, MEMWBIR; // pipeline registers
   wire [4:0] IDEXrs, IDEXrt, EXMEMrd, MEMWBrd; //hold register fields
   wire [5:0] EXMEMop, MEMWBop, IDEXop; Hold opcodes
   wire [31:0] Ain, Bin;
   // declare the bypass signals
   wire takebranch, stall, bypassAfromMEM, bypassAfromALUinWB,bypassBfromMEM, bypassBfromALUinWB,
      bypassAfromLWinWB, bypassBfromLWinWB;
   assign IDEXrs = IDEXIR[25:21];  assign IDEXrt = IDEXIR[15:11];  assign EXMEMrd = EXMEMIR[15:11];
   assign MEMWBrd = MEMWBIR[20:16]; assign EXMEMop = EXMEMIR[31:26];
   assign MEMWBop = MEMWBIR[31:26];  assign IDEXop = IDEXIR[31:26];
   // The bypass to input A from the MEM stage for an ALU operation
   assign bypassAfromMEM = (IDEXrs == EXMEMrd) & (IDEXrs!=0) & (EXMEMop==ALUop); // yes, bypass
   // The bypass to input B from the MEM stage for an ALU operation
   assign bypassBfromMEM = (IDEXrt == EXMEMrd)&(IDEXrt!=0) & (EXMEMop==ALUop); // yes, bypass
   // The bypass to input A from the WB stage for an ALU operation
   assign bypassAfromALUinWB =( IDEXrs == MEMWBrd) & (IDEXrs!=0) & (MEMWBop==ALUop);
   // The bypass to input B from the WB stage for an ALU operation
   assign bypassBfromALUinWB = (IDEXrt == MEMWBrd) & (IDEXrt!=0) & (MEMWBop==ALUop); /
   // The bypass to input A from the WB stage for an LW operation
   assign bypassAfromLWinWB =( IDEXrs == MEMWBIR[20:16]) & (IDEXrs!=0) & (MEMWBop==LW);
   // The bypass to input B from the WB stage for an LW operation
   assign bypassBfromLWinWB = (IDEXrt == MEMWBIR[20:16]) & (IDEXrt!=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 | bypassAfromLWinWB)? 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 | bypassBfromLWinWB)? MEMWBValue: IDEXB;
   // The signal for detecting a stall based on the use of a result from LW
   assign stall = (MEMWBIR[31:26]==LW) && // source instruction is a load
         ((((IDEXop==LW)|(IDEXop==SW)) && (IDEXrs==MEMWBrd)) | // stall for address calc
((IDEXop==ALUop) && ((IDEXrs==MEMWBrd)|(IDEXrt==MEMWBrd)))); // ALU use
```

FIGURE e4.14.4 A behavioral definition of the five-stage MIPS pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. The changes from Figure e4.14.3 are highlighted. (*continues on next page*)

```
// Signal for a taken branch: instruction is BEQ and registers are equal
assign takebranch = (IFIDIR[31:26]==BEQ) && (Regs[IFIDIR[25:21]]== Regs[IFIDIR[20:16]]);
   reg [5:0] i; //used to initialize registers
   initial begin
      PC = 0;
     IFIDIR = no-op; IDEXIR = no-op; EXMEMIR = no-op; MEMWBIR = no-op; // put no-ops in pipeline registers
      for (i = 0;i<=31;i = i+1) Regs[i] = i; //initialize registers--just so they aren't don't cares
   end

   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 no-op and reset the PC
         IFDIR <= no-op;
         PC <= PC + 4 + ({{16{IFIDIR[15]}}, IFIDIR[15:0]}<<2);
         end

      // second instruction is in register fetch
       IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // get two registers

      // third instruction is doing address calculation or ALU operation
         IDEXIR <= IFIDIR;  //pass along IR
if ((IDEXop==LW) |(IDEXop==SW))  // address calculation & copy B
           EXMEMALUOut <= IDEXA +{{16{IDEXIR[15]}}, IDEXIR[15:0]};
       else if (IDEXop==ALUop) case (IDEXIR[5:0]) //case for the various R-type instructions
             32: 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 <= no-op; /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

      // the WB stage
MEMWBIR <= EXMEMIR; //pass along IR
      if ((MEMWBop==ALUop) & (MEMWBrd != 0)) Regs[MEMWBrd] <= MEMWBValue; // ALU operation

      else if ((EXMEMop == LW)& (MEMWBIR[20:16] != 0)) Regs[MEMWBIR[20:16]] <= MEMWBValue;

   end
endmodule
```

FIGURE e4.14.4 A behavioral definition of the five-stage MIPS pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation. (*Continued*)

Using Verilog for Behavioral Specification with Synthesis

To demonstate the contrasting types of Verilog, we show two descriptions of a different, nonpipelined implementation style of MIPS that uses multiple clock cycles per instruction. (Since some instructors make a synthesizable description of the MIPS pipe line 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 MIPS 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 C.10 of Appendix B). 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.

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

- The 4-to-1 multiplexor shown in Figure B.4.2, and the 3-to-1 multiplexor that can be trivially derived based on the 4-to-1 multiplexor.
- The MIPS ALU shown in Figure B.5.15.
- The MIPS ALU control defined in Figure B.5.16.
- The MIPS register file defined in Figure B.8.11.

Now, let's look at a Verilog version of the MIPS processor intended for synthesis. Figure e4.14.6 shows the structural version of the MIPS datapath. Figure e4.14.7 uses the datapath module to specify the MIPS 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 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.

```
module CPU (clock);
parameter LW = 6’b100011, SW = 6’b101011, BEQ=6’b000100, J=6’d2;
input clock; //the clock is an external input
// The architecturally visible registers and scratch registers for implementation
reg [31:0] PC, Regs[0:31], Memory [0:1023], IR, ALUOut, MDR, A, B;
reg [2:0] state; // processor state
wire [5:0] opcode; //use to get opcode easily
wire [31:0] SignExtend,PCOffset; //used to get sign-extended offset field
assign opcode = IR[31:26]; //opcode is upper 6 bits
assign SignExtend = {{16{IR[15]}},IR[15:0]}; //sign extension of lower 16 bits of instruction
assign PCOffset = SignExtend << 2; //PC offset is shifted
// set the PC to 0 and start the control in state 0
initial begin PC = 0; state = 1; end

//The state machine--triggered on a rising clock
always @(posedge clock) begin
    Regs[0] = 0; //make R0 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[25:21]];
         B <= Regs[IR[20:16]];
         state = 3;
         ALUOut <= PC + PCOffset; // compute PC-relative branch target
      end

      3: begin // third step: Load-store execution, ALU execution, Branch completion
           state = 4; // default next state
           if ((opcode==LW) |(opcode==SW)) ALUOut <= A + SignExtend; //compute effective address
           else if (opcode==6’b0) case (IR[5:0]) //case for the various R-type instructions
             32: ALUOut = A + B; //add operation
             default: ALUOut = A; //other R-type operations: subtract, SLT, etc.
           endcase
```

FIGURE e4.14.5 A behavioral specification of the multicycle MIPS 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*)

```
          else if (opcode == BEQ) begin
                     if (A==B) PC <= ALUOut; // branch taken--update PC
                     state = 1;
           end
           else if (opocde=J) begin
                PC = {PC[31:28], IR[25:0],2'b00}; // the jump target PC
                state = 1;
           end  //Jumps
                else ; // other opcodes or exception for undefined instruction would go here
   end


   4: begin
       if (opcode==6'b0) begin //ALU Operation
             Regs[IR[15:11]] <= 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 == LW) begin
                      Memory[ALUOut>>2] <= B; // write the memory
                      state = 1; // return to state 1
                  end //store finishes
                       else ; // other instructions go here
         end


   5: begin // LW is the only instruction still in execution
         Regs[IR[20:16]] = 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 MIPS design. (*Continued*)

```
module Datapath (ALUOp, RegDst, MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite,
PCWrite,   PCWriteCond, ALUSrcA, ALUSrcB, PCSource, opcode, clock); // the control inputs + clock
input [1:0] ALUOp, ALUSrcB, PCSource; // 2-bit control signals
input RegDst, MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite, PCWriteCond,
ALUSrcA,    clock; // 1-bit control signals
output [5:0] opcode ;// opcode is needed as an output by control
reg [31:0] PC, Memory [0:1023], MDR,IR, ALUOut; // 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
wire[4:0] Writereg;// the signal used to communicate the destination register
 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[31:26];// opcode shortcut
// Get the write register address from one of two fields depending on RegDst
assign Writereg = RegDst ? IR[15:11]: IR[20:16];
// Get the write register data either from the ALUOut or from the MDR
assign Writedata = MemtoReg ? MDR : ALUOut;
// Sign-extend the lower half of the IR from load/store/branch offsets
assign SignExtendOffset = {{16{IR[15]}},IR[15:0]}; //sign-extend lower 16 bits;
// The branch offset is also shifted to make it a word offset
assign PCOffset = SignExtendOffset << 2;
// 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
// Compose the Jump address
assign JumpAddr = {PC[31:28], IR[25:0],2’b00}; //The jump address
```

FIGURE e4.14.6 A Verilog version of the multicycle MIPS datapath that is appropriate for synthesis. This datapath relies on several units from Appendix B. 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 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 an instance of the ALU control unit (see the module defined in Figure C.5.16 on page C-38

   // Input ALUOp is control-unit set and used to describe the instruction class as in Chapter 4
   // Input IR[5:0] 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[5:0],ALUCtl); //ALU control unit

// Creates a 3-to-1 multiplexor used to select the source of the next PC

   // Inputs are ALUResultOut (the incremented PC) , ALUOut (the branch address), the jump target address
   // PCSource is the selector input and PCValue is the multiplexor output

Mult3to1 PCdatasrc (ALUResultOut,ALUOut,JumpAddr, PCSource , PCValue);

 // Creates a 4-to-1 multiplexor used to select the B input of the ALU

   // Inputs are register B,constant 4, sign-extended lower half of IR, sign-extended lower half of IR << 2
   // ALUSrcB is the selector input
   // ALUBin is the multiplexor output

Mult4to1 ALUBinput (B,32'd4,SignExtendOffset,PCOffset,ALUSrcB,ALUBin);


 // Creates a MIPS ALU

   // Inputs are ALUCtl (the ALU control), ALU value inputs (ALUAin, ALUBin)
   // Outputs are ALUResultOut (the 32-bit output) and Zero (zero detection output)

MIPSALU ALU (ALUCtl, ALUAin, ALUBin, ALUResultOut,Zero); //the ALU

// Creates a MIPS register file

   // Inputs are
   // the rs and rt 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[25:21],IR[20:16],Writereg,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
      if (PCWrite || (PCWriteCond & Zero)) PC <=PCValue;

end
endmodule
```

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

```
module CPU (clock);
   parameter LW = 6’b100011, SW = 6’b101011, BEQ = 6’b000100, J = 6’d2; //constants
   input clock; reg [2:0] state;
   wire [1:0] ALUOp, ALUSrcB, PCSource; wire [5:0] opcode;
   wire RegDst, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite, PCWriteCond,
        ALUSrcA, MemoryOp, IRWwrite, Mem2Reg;
// Create an instance of the MIPS datapath, the inputs are the control signals; opcode is only output
Datapath MIPSDP (ALUOp,RegDst,Mem2Reg, 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 IRWrite = (state==1);
assign Mem2Reg = ~ RegDst;
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 RegDst = ((state==4)&(opcode==0)) ? 1 : 0;
   assign MemRead = (state==1) | ((state==4)&(opcode==LW));
   assign MemWrite = (state==4)&(opcode==SW);
   assign IorD = (state==1) ? 0 : (state==4) ? 1 : X;
   assign RegWrite = (state==5) | ((state==4) &(opcode==0));
   assign PCWrite = (state==1) | ((state==3)&(opcode==J));
   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) ? 2’b00 : ((opcode==BEQ) ? 2’b01 : 2’b10);
// 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: // third step: jumps and branches complete
         state = ((opcode==BEQ) | (opcode==J)) ? 1 : 4;// branch or jump go back else next state
      4: state = (opcode==LW) ? 5 : 1; //R-type and SW finish
      5: state = 1; // go back
       endcase
end
  endmodule
```

FIGURE e4.14.7 The MIPS CPU using the datapath from Figure e4.14.6.

Elaboration: When specifying control, designers often take advantage of knowledge of the control so as to simplify or shorten the control specification. Here are a few examples from the specification in Figures e4.14.6 and e4.14.7.

1. `MemtoReg` is set only in two cases, and then it is always the inverse of `RegDst`, so we just use the inverse of `RegDst`.
2. `IRWrite` is set only in state 1.
3. The ALU does not operate in every state and, when unused, can safely do anything.
4. `RegDst` is 1 in only one case and can otherwise be set to 0. In practice it might be better to set it explicitly when needed and otherwise set it to X, as we do for `IorD`. First, it allows additional logic optimization possibilities through the exploitation of don't-care terms (see Appendix B for further discussion and examples). Second, it is a more precise specification, and this allows the simulation to more closely model the hardware, possibly uncovering additional errors in the specification.

More Illustrations of Instruction Execution on the Hardware

To reduce the cost of this book, in 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 better understand pipelining. 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.

No Hazard Illustrations

On page 297, we gave the example code sequence

```
lw    $10, 20($1)
sub   $11, $2, $3
add   $12, $3, $4
lw    $13, 24($1)
add   $14, $5, $6
```

Figures 4.43 and 4.44 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.