6.7 Using a Hardware Description Language to Describe and Model a Pipeline

This section, which appears on the CD, provides a behavioral model in Verilog of the MIPS five-stage pipeline. The initial model ignores hazards, and additions to the model highlight the changes for forwarding, data hazards, and branch hazards.

For those readers who examined the use of a hardware description language in Chapter 5, we show how to use Verilog to describe the behavior of the five-stage MIPS pipeline. Figure 6.7.1 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 pipestage. Because of this difference, the instruction register (IR) is needed throughout the pipeline, and the entire IR is passed from pipestage to pipestage. 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.

Notice that we also use a different solution for writes to R0 than we did in the Verilog descriptions of the multicycle design. Instead, we detect and ignore such writes. This approach is useful because it will avoid unnecessary stalls when we add stall detection.

```
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 assignment define fields from the pipeline registers
   assign IDEXrs = IDEXIR[25:21];   // rs field
   assign IDEXrt = IDEXIR[15:11];   // rt field
   assign EXMEMrd = EXMEMIR[15:11]; // rd field
   assign MEMWBrd = MEMWBIR[20:16]; //rd field
   assign MEMWBrt = MEMWBIR[25:20]; //rt field--used for loads
   assign EXMEMop = EXMEMIR[31:26]; // the opocde
   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 pipestage 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--come 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
```

```
      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 6.7.1 A Verilog behavorial model for the MIPS five-stage pipeline, ignoring branch and data hazards. As in the design earlier in this chapter, we use separate instruction and data memories, which would be implemented using separate caches as we describe in Chapter 7.

Implementing Forwarding in Verilog

To further extend the Verilog model, Figure 6.7.2 shows the addition of forwarding logic for the case when the source instruction is an ALU instruction and the source. 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 1 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 pipestage 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 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 Bfrom 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
```

```
      // second instruction is in register fetch
         IDEXA <= Regs[IFIDIR[25:21]]; IDEXB <= Regs[IFIDIR[20:16]]; // get two registers
         IDEXIR <= IFIDIR;  //pass along IR--come 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 6.7.2 A behavioral definition of the 5-stage MIPS pipeline with bypassing to ALU operations and address calculations. The code added to Figure 6.7.1 on page 6.7-3 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.

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 6.7.3 shows the few additions needed.

```
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];
 ssign 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 Bfrom 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
```

```
        // first instruction  in the pipeline is being fetched
          IFIDIR <= IMemory[PC>>2];
          PC <= PC + 4;
         IDEXIR <= IFIDIR;  //pass along IR--come 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 6.7.3 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 6.7.2 on page 6.7-5 are highlighted.

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.
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 6.7.4. 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 6.7.4 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'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; //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 Bfrom 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
```

```
// 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 + ({{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 6.7.4 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 6.7.2 on page 6.7-5 are highlighted.