4.13 Advanced topic: An introduction to digital design using a hardware design language to describe and model a pipeline and more pipelining illustrations

This section has been set as optional by your instructor.

This section covers hardware description languages and then gives a dozen examples of pipeline diagrams.

As mentioned in COD Appendix B (Graphics and Computing GPUs), 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, 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 MIPS processor: one intended solely for simulations and one suitable for synthesis. This MIPS architecture is satisfactory to demonstrate Verilog, but if any reader builds one for LEGv8 and wants to share it with others, please contact the publisher.

Using Verilog for behavioral specification with simulation for the five-stage pipeline

The figure below 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.

Figure 4.13.1: A Verilog behavorial model for the MIPS five-stage pipeline, ignoring branch and data hazards (COD Figure 4.13.1).

As in the design earlier in COD Chapter 4 (The Processor), we use separate instruction and data memories, which would be implemented using separate caches as we describe in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy).

```
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

      //initialize registers--just so they aren't cares
      for (i = 0; i <= 31; i = i + 1)
         Regs[i] = i;
   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;                               // Fetch & increment PC
   end

      // 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

      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
         Regs[MEMWBrt] <= MEMWBValue;             // and destination not 0

   end
endmodule
```

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.

Implementing forwarding in Verilog

To extend the Verilog model further, the figure below 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.

Figure 4.13.2: A behavioral definition of the five-stage MIPS pipeline with bypassing to ALU operations and address calculations (COD Figure 4.13.2).

The code added to the figure above 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.

```
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

      // initialize registers--just so they aren't cares
      for (i = 0;i <= 31; i = i + 1)
         Regs[i] = i;
   end

   always @ (posedge clock) begin
      // first instruction in the pipeline is being fetched
      IFIDIR <= IMemory[PC >> 2];
      PC <= PC + 4;                                    // Fetch & increment PC
      end

      // 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
```

PARTICIPATION ACTIVITY 4.13.1: Check yourself: Forwarding.

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.
   - Yes
   - No

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.
   - Yes
   - No

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.
   - Yes
   - No

4) The result of an ALU instruction is not available in time to do the write during MEM.
   - Yes
   - No

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 the figure below shows the few additions needed.

Figure 4.13.3: A behavioral definition of the five-stage MIPS pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation (COD Figure 4.13.3).

The changes from the figure above are highlighted.

```
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

          // initialize registers--just so they aren't cares
          for (i = 0; i <= 31; i = i + 1)
            Regs[i] = i;
   end

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

PARTICIPATION ACTIVITY 4.13.2: Check yourself: Hazards.

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.
   - ○ True
   - ○ False

2) No pipeline can ever have a hazard involving memory, since it is the programmer's job to keep the order of memory references accurate.
   - ○ True
   - ○ False

3) 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.
   - ○ True
   - ○ False

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 the figure below. 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 the figure below incorporates the basic logic for branches and control hazards, supporting branches requires additional bypassing and data hazard detection, which we have not included.

Figure 4.13.4: A behavioral definition of the five-stage MIPS pipeline with stalls for loads when the destination is an ALU instruction or effective address calculation (COD Figure 4.13.4).

The changes from COD Figure 4.13.2 (A behavioral definition of the five-stage MIPS pipeline …) are highlighted.

```
module CPU (clock);
   parameter LW = 6'b100011,
             SW = 6'b101011,
             BEQ = 6'b000100,
             no-op = 32'b00000000_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;
   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 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

   // 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

          // initialize registers--just so they aren't don't cares
          for (i = 0; i <= 31; i = i + 1)
        Regs[i] = i;
   end

   always @ (posedge clock) begin
          // the first three pipeline stages stall if there is a load hazard
          if (~stall) begin
        // first instruction in the pipeline is being fetched normally
        if (~takebranch) begin
           IFIDIR <= IMemory[PC >> 2];
           PC <= PC + 4;
        end

        else begin                              // a taken branch is in ID; instruction in IF
is wrong;
           IFIDIR <= no-op;                     // insert a no-op and reset the PC
           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)) // ALU operation
        Regs[MEMWBrd] <= MEMWBValue;

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

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 MIPS that uses multiple clock cycles per instruction. (Since some instructors make a synthesizable description of the MIPS pipeline project for a class, we chose not to include it here. It would also be long.)

The figure below 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 COD Section A.10 (Finite-state machines)). 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.

Figure 4.13.5: A behavioral specification of the multicycle MIPS design (COD Figure 4.13.5).

This has the same cycle behavior as the multicycle design, but is purely for simulation and specification. It cannot be used for synthesis.

```
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
      // action depends on the state
      case (state)
         // first step: fetch the instruction, increment PC, go to next state
         1: begin
            IR <= Memory[PC >> 2];
            PC <= PC + 4;
            state = 2;                             // next state
         end

         // second step: Instruction decode, register fetch, also compute branch address
         2: begin
            A <= Regs[IR[25:21]];
            B <= Regs[IR[20:16]];
            state = 3;
            ALUOut <= PC + PCOffset;               // compute PC-relative branch target
         end

         // third step: Load-store execution, ALU execution, Branch completion
         3: begin
            state = 4;                             // default next state

            if ((opcode == LW) | (opcode == SW))
               ALUOut <= A + SignExtend;           // compute effective address

            else if (opcode == 6'b0)  begin
               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
            end

            else if (opcode == BEQ) begin
               if (A == B) PC <= ALUOut;           // branch taken--update PC
                  state = 1;
            end

            else if (opcode == J) begin
               PC = {PC[31:28], IR[25:0], 2'b00};  // the branch target PC
               state = 1;
            end                                    // Jumps

            else ;                                 // other opcodes or exception for undefined
         end                                       // instruction would go here

         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

         // LW is the only instruction still in execution
         5: begin
            Regs[IR[20:16]] = MDR;                 // write the MDR to the register
            state = 1;
         end                                       // complete an LW instruction
          endcase
   end
endmodule
```

Note: The MIPS ALU and LEGv8 ALU are similar except that MIPS has 32-bit input/output and LEGv8 had 64-bit input/output. Thus below references to MIPS ALU figures are actually LEGv8 ALU images in Appendix A.

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 COD Appendix A (The Basics of Logic Design), including the following:

- The 4-to-1 multiplexor shown in COD Figure A.4.2 (A Verilog definition of a 4-to-1 multiplexor with 32-bit inputs, using a case statement), and the 3-to-1 multiplexor that can be trivially derived based on the 4-to-1 multiplexor.
- The MIPS ALU shown in COD Figure A.5.15 (A Verilog behavioral definition of a LEGv8 ALU) .
- The MIPS ALU control defined in COD Figure A.5.16 (The LEGv8 ALU control: a simple piece of combinational control logic).
- The MIPS register file defined in COD Figure A.8.11 (A LEGv8 register file written in behavioral Verilog).

Now, let's look at a Verilog version of the MIPS processor intended for synthesis. The figure below shows the structural version of the MIPS datapath. COD Figure 4.13.7 (The MIPS CPU using the datapath from the figure above) 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.

Figure 4.13.6: A Verilog version of the multicycle MIPS datapath that is

Figure 4.13.6: A Verilog version of the multicycle MIPS datapath that is appropriate for synthesis (COD Figure 4.13.6).

This datapath relies on several units from COD Appendix A (The Basics of Logic Design). 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.

```
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


   // Creates an instance of the ALU control unit (see the module defined in
   // COD Figure B.5.16 (The MIPS ALU Control: a simple piece of combinational logic)
         // Input ALUOp is control-unit set and used to describe the instruction class
         // as in COD Chapter 4 (The Processor)
         // Input IR[5:0] is the function code field for an ALU instruction
         // Output ALUCtl are the actual ALU control bits as in COD Chapter 4 (The Processor)
   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)                    // Write the IR if an instruction fetch
         IR <= MemOut;

      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 4.13.7: The MIPS CPU using the datapath from the figure above (COD Figure 4.13.7).

```
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, IRWrite, 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);

   // start the state machine in state 1
   initial begin
          state = 1;
   end

   // 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)
         // unconditional next state
         1: state = 2;

         // unconditional next state
         2: state = 3;

            // third step: jumps and branches complete
         // branch or jump go back else next state
         3: state = ((opcode == BEQ) | (opcode == J)) ? 1 : 4;

         // R-type and SW finish
         4: state = (opcode == LW) ? 5 : 1;

         // go back
         5: state = 1;
          endcase
   end
endmodule
```

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.

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 the previous two figures.

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 elsewhere 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 further logic optimization possibilities through the exploitation of don't-care terms (see COD Appendix A (The Basics of Logic Design) 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.*

PARTICIPATION ACTIVITY 4.13.3: Digital design using Verilog.

1) Verilog does not support the ability to define registers with named fields.
   - O True
   - O False

2) Actions in an `always` block occur sequentially in every pipe stage.
   - O True
   - O False

3) The MIPS CPU control signals are set using `assign` statements, which are based on the state and opcode field of the instruction register.

- ○ True
- ○ False

4) When branches are ignored in the MIPS behavorial model, the only way a data hazard stall could occur is if a load result is currently occupying the WB clock stage.

- ○ True
- ○ False

5) For branch hazard logic, if a taken branch is detected in the IF stage, then the instruction in the ID stage is incorrect.

- ○ True
- ○ False

More illustrations of instruction execution on the hardware

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.

No hazard illustrations

In COD Section 4.6 (Pipelined datapath and control), we gave the example code sequence

```
LDUR  X10, [X1,#40]
SUB   X11, X2, X3
ADD   X12, X3, X4
LDUR  X13, [X1,#48]
ADD   X14, X5, X6
```

COD Figures 4.42 (Multiple-clock-cycle pipeline diagram of five instructions) and 4.43 (Traditional multiple-clock-cycle pipeline diagram of five instructions …) showed the multiple-clock-cycle pipeline diagrams for this two-instruction sequence executing across six clock cycles. The three figures below 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.

Figure 4.13.8: Single-cycle pipeline diagrams for clock cycles 1 (top diagram) and 2 (bottom diagram) (COD Figure 4.13.8).

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.

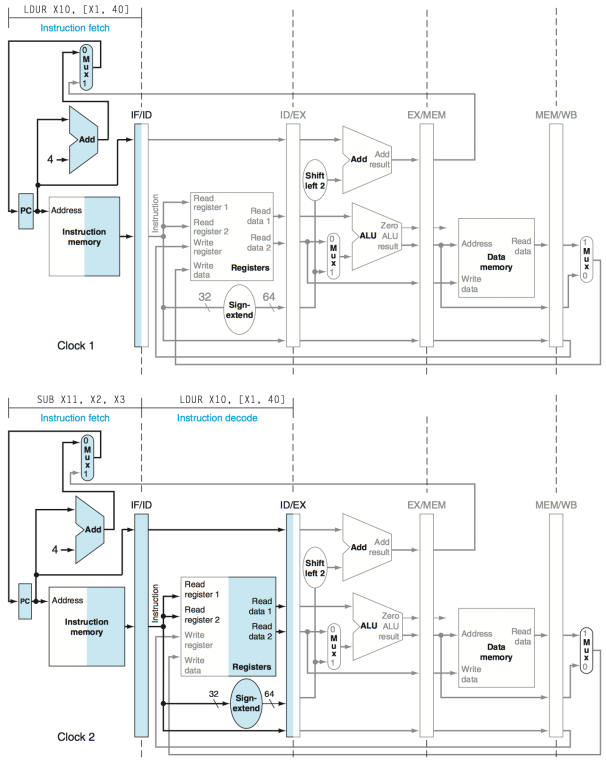


Figure 4.13.9: Single-cycle pipeline diagrams for clock cycles 3 (top diagram) and 4 (bottom diagram) (COD Figure 4.13.9).

In the third clock cycle in the top diagram, `LDUR` enters the EX stage. At the same time, `SUB` enters ID. In the fourth clock cycle (bottom datapath), `LDUR` 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.

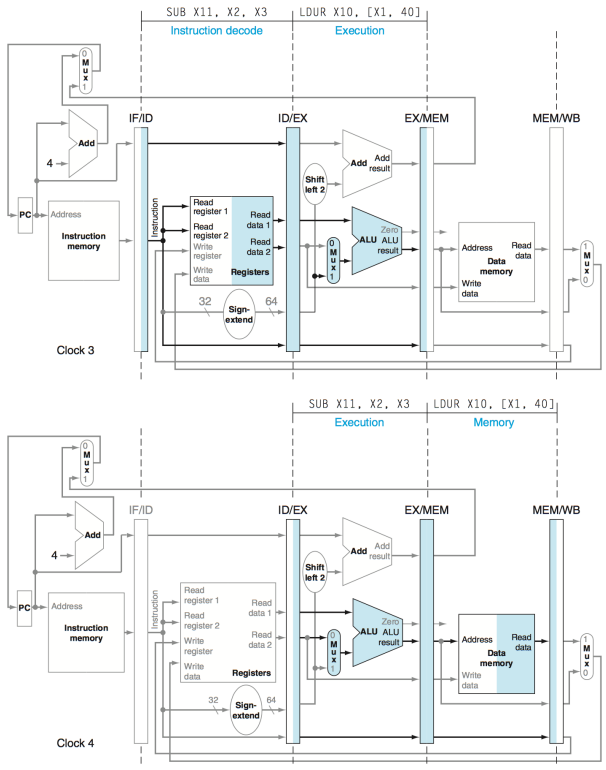
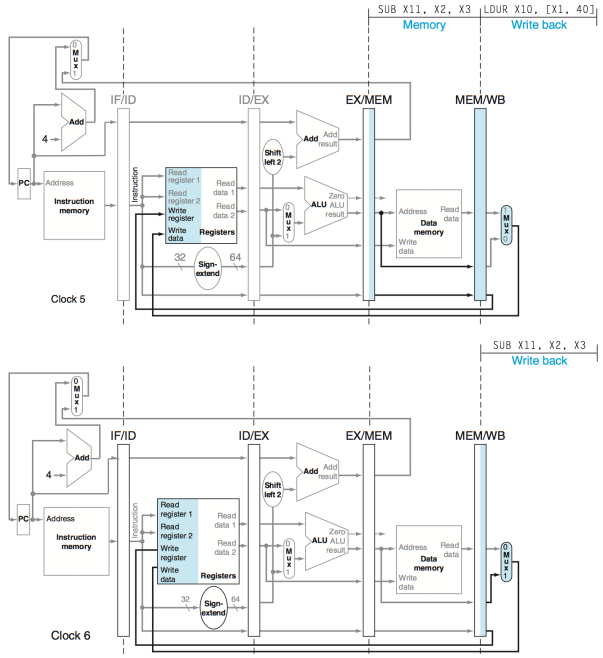


Figure 4.13.10: Single-cycle pipeline diagrams for clock cycles 5 (top diagram) and 6 (bottom diagram) (COD Figure 4.13.10).

In clock cycle 5, `LDUR` 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.



More examples

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

```
LDUR  X10, [X1,#40]
SUB   X11, X2, X3
AND   X12, X4, X5
ORR   X13, X6, X7
ADD   X14, X8, X9
```

The five figures below 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 COD Figure 4.13.13 (Clock cycles 5 and 6) 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.

Figure 4.13.11: Clock cycles 1 and 2 (COD Figure 4.13.11).

The phrase *"before <i>"* means the *i*th instruction before `LDUR`. The `LDUR` instruction in the top datapath is in the IF stage. At the end of the clock cycle, the `LDUR` instruction is in the IF/ID pipeline registers. In the second clock cycle, seen in the bottom datapath, the `LDUR` 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 `LDUR`, are written into the ID/EX pipeline register. The number 10, representing the destination register number of `LDUR`, is also placed in ID/EX. The top of the ID/EX pipeline register shows the control values for `LDUR` to be used in the remaining stages. These control values can be read from the `LDUR` row of the table in COD Figure 4.18 (The setting of the control lines is completely determined by the opcode fields of the instruction).

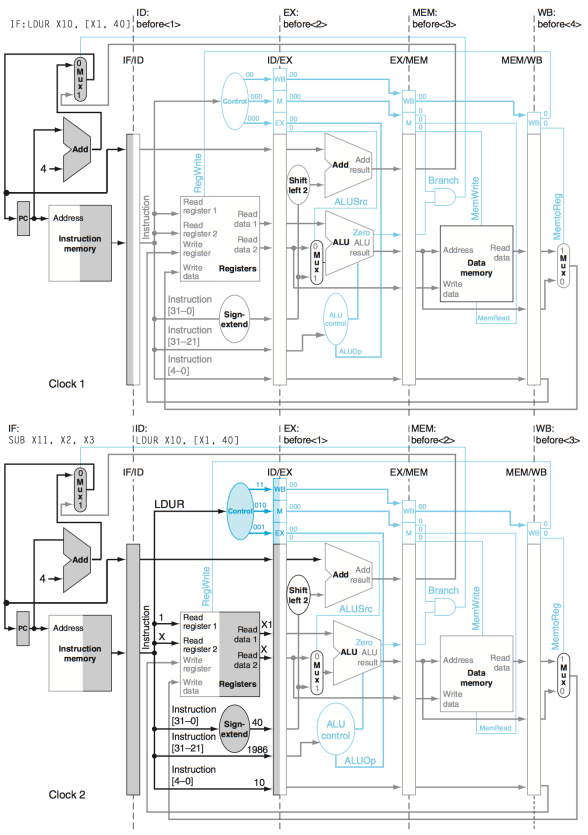


Figure 4.13.12: Clock cycles 3 and 4 (COD Figure 4.13.12).

In the top diagram, `LDUR` enters the EX stage in the third clock cycle, adding `X1` and 40 to form the address in the EX/MEM pipeline register. (The `LDUR` instruction is written `LDUR 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), `LDUR` 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 `ORR` 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.

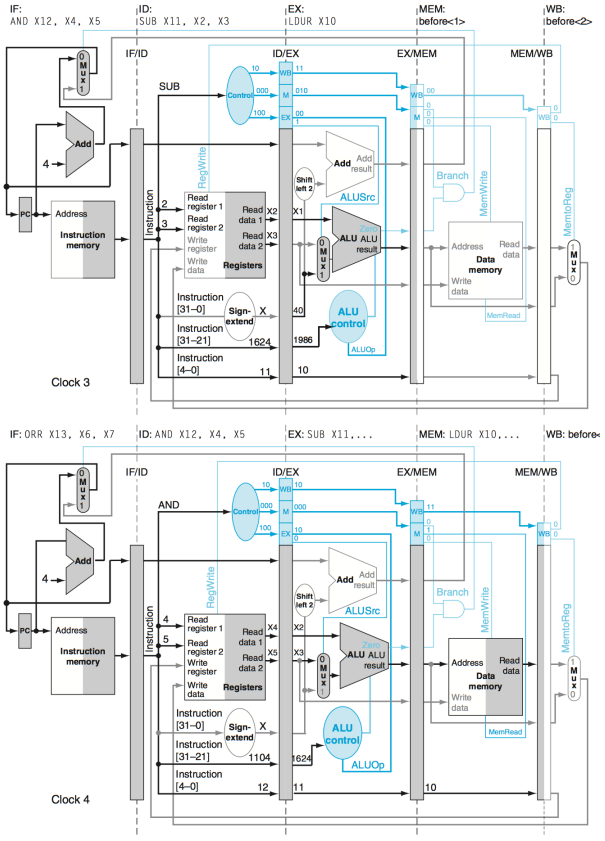


Figure 4.13.13: Clock cycles 5 and 6 (COD Figure 4.13.13).

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, `LDUR` 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 `ORR` 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`.

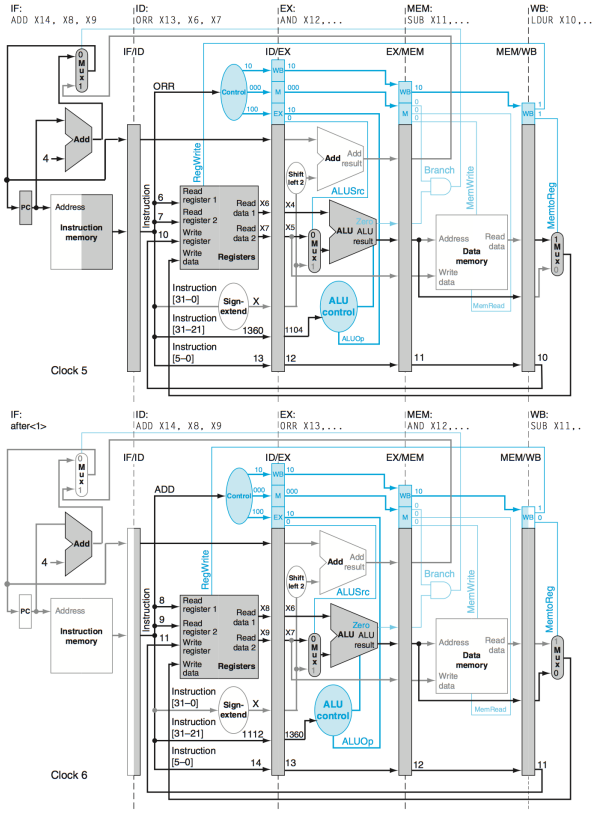


Figure 4.13.14: Clock cycles 7 and 8 (COD Figure 4.13.14).

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 `ORR` 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 `ORR`, 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.

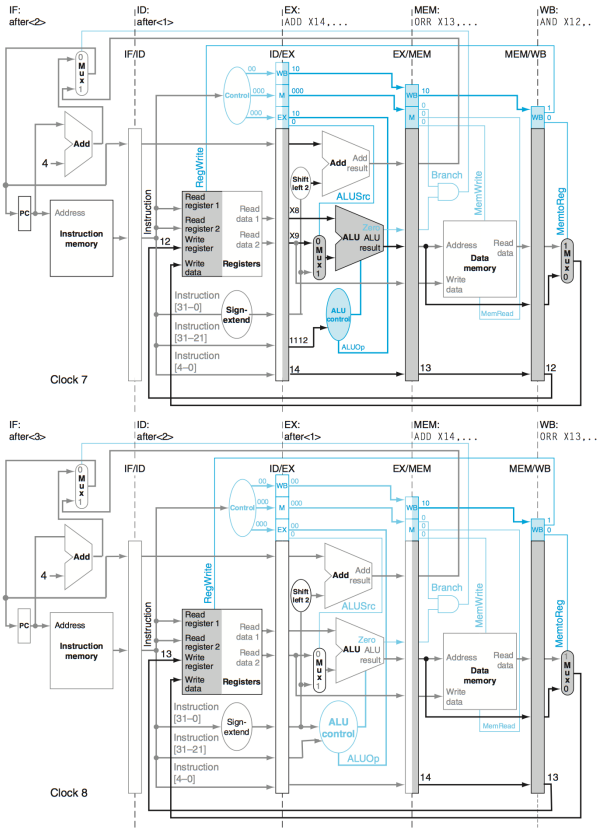
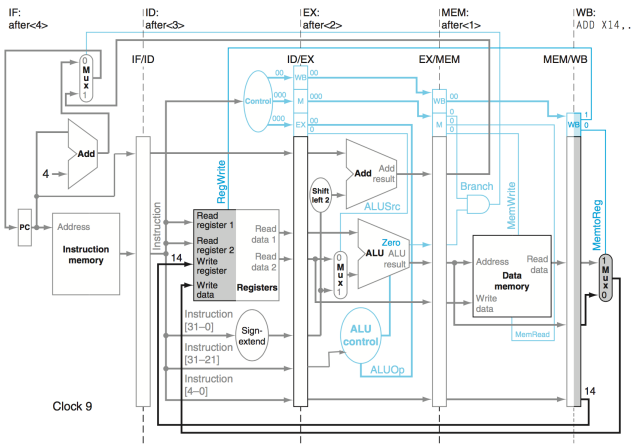


Figure 4.13.15: Clock cycle 9 (COD Figure 4.13.15).

The WB stage writes the `sum` in MEM/WB into register `X14`, completing `ADD` and the five-instruction sequence. The instructions after `ADD` are shown as inactive for pedagogical reasons.



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:

```
SUB  X2, X1, X3
AND  X4, X2, X5
ORR  X4, X4, X2
ADD  X9, X4, X2
```

The two figures below show the events in clock cycles 3—6 in the execution of these instructions.

Figure 4.13.16: Clock cycles 3 and 4 of the above instruction sequence (COD Figure 4.13.16).

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 COD Figures 4.13.12 (Clock cycles 3 and 4) through 4.13.15 (Clock cycle 9), which show the datapath without forwarding where ID is the last stage to need operand information.

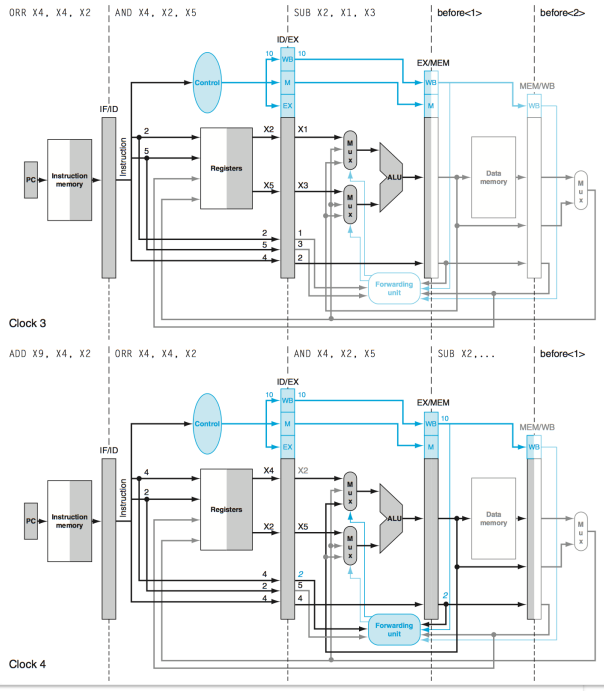
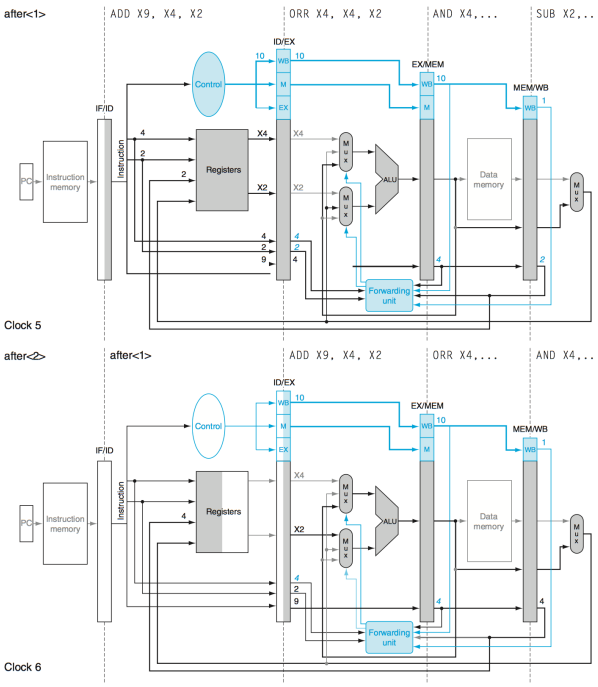


Figure 4.13.17: Clock cycles 5 and 6 of the above instruction sequence (COD Figure 4.13.17).

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.



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 `ORR` 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. The three figures below show the single-cycle diagram for clocks 2 through 7 for the following code sequence:

```
LDUR X2, [X1,#40]
AND  X4, X2,X5
ORR  X4, X4,X2
ADD  X9, X4,X2
```

Figure 4.13.18: Clock cycles 2 and 3 of the above instruction sequence with a load replacing SUB (COD Figure 4.13.18).

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 **LDUR** instruction in clock cycle 3, so the hazard detection unit stalls the **AND** and **ORR** instructions. Hence, the hazard detection unit is highlighted.

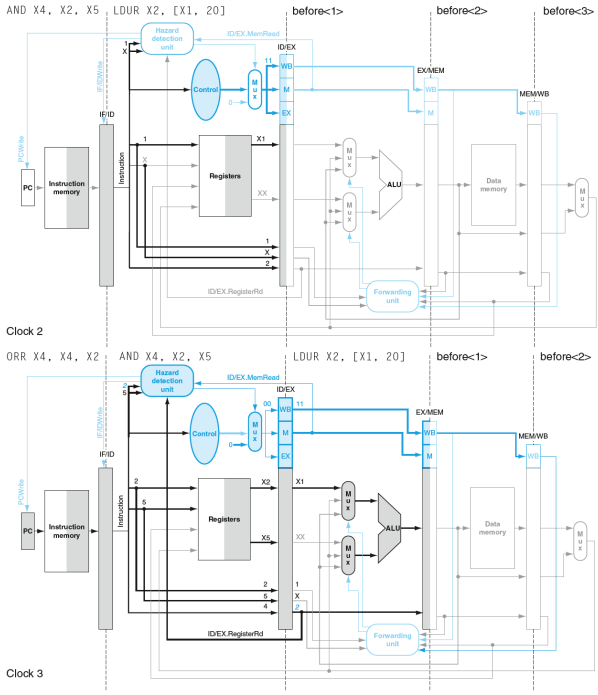


Figure 4.13.19: Clock cycles 4 and 5 of the above instruction sequence with a load replacing SUB (COD Figure 4.13.19).

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 **LDUR** to the ALU. Note that in clock cycle 4, the forwarding unit forwards the address of the **LDUR** 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.

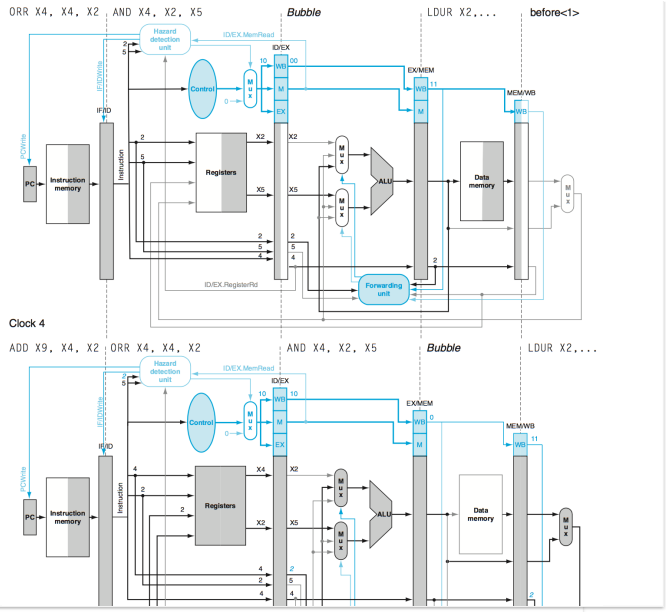
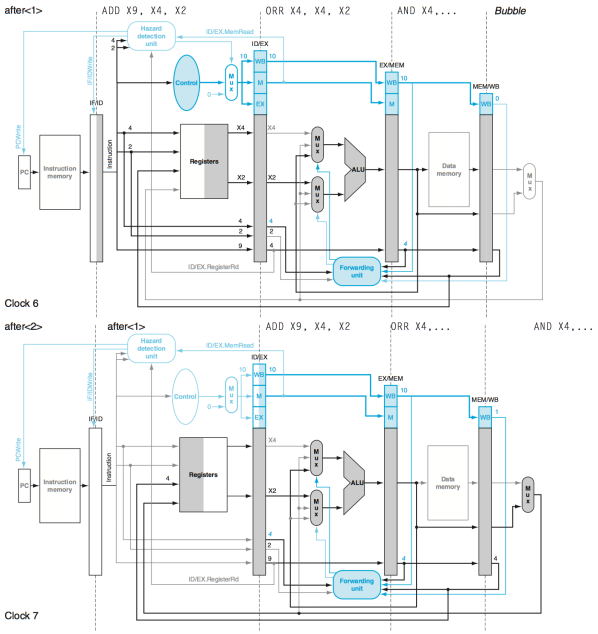


Figure 4.13.20: Clock cycles 6 and 7 of the above instruction sequence with a load replacing SUB (COD Figure 4.13.20).

Note that unlike in COD Figure 4.13.17 (Clock cycles 5 and 6 of the above instruction sequence), the stall allows the `LDUR` 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 `ORR` 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.



PARTICIPATION ACTIVITY 4.13.4: Pipeline diagrams.

1) A _____ unit is introduced to the pipeline to make an ALU instruction result available as an operand to the next instruction.

Check Show answer

2) If an ALU instruction attempts to read a value created by an `LDUR` instruction, the _____ unit stalls subsequent instructions.

Check Show answer

Provide feedback on this section