5.8 Using a Hardware Description Language to Design and Simulate a Processor

As mentioned in Appendix B, 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 illustrate this dichotomy with two Verilog descriptions of a MIPS processor: one intended solely for simulations and one suitable for synthesis.

Using Verilog for Behavioral Specification with Simulation

To illustrate the use of Verilog for describing behavioral implementations, Figure 5.8.1 gives a behavioral specification of the multicycle implementation of the MIPS processor. This version has the same cycle count behavior as the multicycle implementation earlier in this chapter, but it does not use the structural datapath. 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 also demonstrates another approach to the control by using a Mealy finite state machine (see discussion in Section B.10 of Appendix B). The use of a Mealy machine, which allows the output to depend both on inputs and the current state, allow us to decrease the total number of states to 5.

Using Verilog for Behavioral Specification and Synthesis

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 on page B-24, 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 on page B-37.
- The MIPS ALU Control defined in Figure B.5.16 on page B-37.
- The MIPS register file defined in Figure B.8.11 on page B-57.

```
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 //short-cut 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 5.8.1 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 a LW instruction
     endcase
  end
  endmodule
```

FIGURE 5.8.1 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.

Now, let's look at a Verilog version of the MIPS processor intended for synthesis. Figure 5.8.2 shows the structural version of the MIPS datapath. Figure 5.8.3 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 decomposition between the main control and the datapath is the same as was done in Section 5.5.

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 external logic to derive the control inputs to the datapath.

```
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 5.8.2 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 B.5.16 on page B.37
   // Input ALUOp is control-unit set and used to describe the instruction class as in Chapter 5
   // Input IR[5:0] is the function code field for an ALU instruction
   // Output ALUCtl are the actual ALU control bits as in Chapter 5
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,{PC[31:28],IR[25:0], 2'b00}, 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
   // ALUSourceB 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 conditionally
   if (PCWrite || (PCWriteCond & Zero)) PC <=PCValue;
end
endmodule
```

FIGURE 5.8.2 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.

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 Figures 5.8.2 and 5.8.3.

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 C 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.

```
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)&(opocde==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 5.8.3 The MIPS CPU using the datapath from Figure 5.8.2.