**LSU EE 3755** **Homework 6** Solution **Due: Not Collected**

*The following questions are based on the Fall 2012 Final Exam. The first two questions on that exam asked about the Hardwired Control (or Multi Cycle) MIPS Implementation. The questions here ask about the Very Simple MIPS Implementation. The two implementations (Very Simple and Hardwired Control) are very similar. Very Simple is written in a Verilog style that makes it easier to see what the synthesized hardware will be and it is easier to distinguish between datapath and control logic. Another difference is that the Hardwired Control MIPS uses more states for some instructions.*

*The following questions ask about an implementation that is similar to the Very Simple MIPS implementation covered in class, but includes some small datapath changes and control logic changes to implement a new instruction. Verilog for the implementation follows the questions. It can also be viewed at* `http://www.ece.lsu.edu/ee3755/2013f/mips-vsi-fe12.v.html`. *The solution is at* `http://www.ece.lsu.edu/ee3755/2013f/mips-vsi-fe12-sol.v.html`.

**Problem 1:** The MIPS implementation can execute a new instruction, `xxx`. Lines relevant to the instruction have `XXX` on the right hand side.

First, lets figure out what it does. In the Verilog code which updates the values of "real" registers (as opposed to things declared `reg` in Verilog) we notice that a new one has been added, named `md` connected to `mem_data_out`. (Maybe `md` is for Memory Data.)

```
///
/// Register Write
///
// Write "real" registers at end of clock cycle. We expect
// these to be the only registers that will be synthesized.
//
always @( posedge clk )
  if ( reset ) begin
     pc <= 32'h400000;
     npc <= 32'h400004;
     ir <= 32'd0;
     state <= ST_if;
  end else begin
     if ( epc ) pc <= npc;
     if ( enpc ) npc <= alu_out;
     if ( eir ) ir <= mem_data_out;
     if ( emd ) md <= mem_data_out;                                    // XXX
     if ( gpr_dst_reg ) gpr[gpr_dst_reg] <= gpr_data_in;
     state <= next_state;
  end
```

In the code below we see a new input to the upper ALU input connected to the new `md` register. That means we can perform arithmetic and other operations from data freshly loaded from memory.

```
// Upper ALU Input Multiplexer
//
always @*
  case ( xalu1 )
    SRC_rs: alu_1 = rs_val;
    SRC_sa: alu_1 = sa_val;
    SRC_npc: alu_1 = npc;
    SRC_md: alu_1 = md;                                               // XXX
    default: begin
```

```
      alu_1 = rs_val;
      // cadence translate_off
      $display("Unexpected ALU 1 source, %d\n", xalu1); $stop;
      // cadence translate_on
    end
  endcase
```

Below we see ALU control signals for the new `xxx` instruction. It's commanding the ALU to add the rs value to the immediate. The value is not being written back to the register file (since `xwr` is set to `DST_0`). (But wait, wasn't there a new upper ALU input?!? So why isn't `xalu1` being set to that new signal `SRC_md`? Okay, I'll be patient.)

```
    // I- and J-Format Instructions
    //              xrwr    xalu1   oalu    xalu2
    O_lbu, O_lw:
            bndl = {DST_rt, SRC_rs, OP_add, SRC_si };
    O_xxx:  bndl = {DST_0,  SRC_rs, OP_add, SRC_si };          // XXX
    O_sb:   bndl = {DST_0 , SRC_rs, OP_add, SRC_si };
    O_lui:  bndl = {DST_rt, SRC_rs, OP_or,  SRC_li };
```

It looks like the new instruction is setting the memory control signals to load something of size 3 (meaning 32 bits). Based on the ALU settings above, it's using the same kind of memory address as load and store instructions.

```
    // Determine values for omem_size, me_wb, and xrws.
    //
    case ( opcode )
      O_lbu   : begin omem_size = 1;  omem_wr = 0;  xrws = 0; end
      O_xxx   : begin omem_size = 3;  omem_wr = 0;  xrws = 0; end// XXX
      O_lw    : begin omem_size = 3;  omem_wr = 0;  xrws = 0; end
```

Whatever our new instruction reads from memory gets written into the new register, `md`. (Actually, every instruction now writes that register, but presumably `xxx` needs it.) We can also see that our new instruction has its very own state, `ST_xxx`. That state must do something with the value in the `md` register.

```
    emd = 1;                                                   // XXX

    // Determine value for next_state.
    //
    case ( opcode )
      O_bne  : next_state = alu_out_z ? ST_bt : ST_ni;
      O_beq  : next_state = alu_out_z ? ST_ni : ST_bt;
      O_xxx  : next_state = ST_xxx;                            // XXX
      O_j    : next_state = ST_jt;
```

Ah, we've found the control signals for the new state. It looks like it's using the ALU to add the contents of `md` to the value of the `rt` register and then writing the sum back into the `rt` register. For the next state it selected `ST_ni`, which prepares to fetch the next instruction.

```
  ST_xxx:                                                      // XXX
    begin                                                      // XXX
      xalu1 = SRC_md;                                          // XXX
      xalu2 = SRC_rt;                                          // XXX
      oalu = OP_add;                                           // XXX
      xrws = 1;                                                // XXX
      xrwr = DST_rt;                                           // XXX
      next_state = ST_ni;                                      // XXX
```

```
        end                                                          // XXX
```

Okay, let's review. The instruction loads something from memory in the same way a `lw` would. Instead of storing the result into the `rt` register it adds the loaded value to whatever is already in the `rt` register.

(*a*) Describe instruction `xxx` as it might be described in an assembly language manual. Remember to describe this as a MIPS instructions, don't describe implementation details such as states or control signals.

☑ Which instruction format is `xxx`?

Format I. It must be Format I because it is using an immediate value and the `rs` and `rt` fields.

☑ Suggest a name and assembly language syntax for `xxx`.

Let's call it `lwa` for load-word-add.

```
# Solution
lwa RT, IMMED(RS)  # RT = RT + Mem[RS + IMMED]
```

☑ Describe what `xxx` does.

It loads a word from memory address `rs` + `simmed` and adds that word (interpreted as an integer) to the contents of register `rt`.

(*b*) Show an example (one instruction is fine) of the use of `xxx`, then show how to do the same thing without `xxx`.

☑ Code example with `xxx`:

```
# Solution
lwa r3, 4(r2)   # r3 = r3 + Mem[ r2 + 4 ]
```

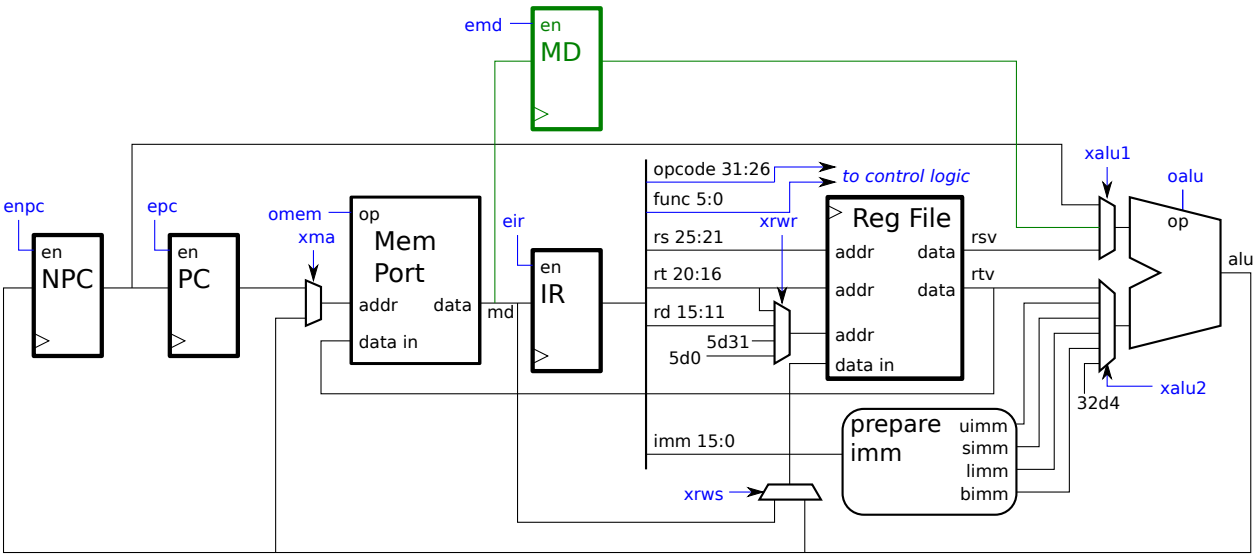
☑ Code doing same thing but without `xxx`:

```
# Solution.  Note that without lwa two instructions are needed.
lw r1, 4(r2)       # r1 = Mem[ r2 + 4 ]
add r3, r3, r1     # r3 = r3 + r1
```

**Problem 2:** Appearing below is the datapath hardware for the Very Simple MIPS. The hardware does not include the new datapath elements added for instruction `xxx`. Based on the Verilog at the end of this assignment, show the new datapath elements that were added for `xxx`. For convenience, changed lines have an `XXX` in the right side.

☑ Show added hardware.

Solution appears below in green. The added datapath includes the MD register which holds the output of the memory port and the additional input to the upper ALU multiplexor.



**Problem 3:** The following new instruction is to be implemented on the Very Simple MIPS implementation at the end of this assignment. The instruction, `lsb RT, (RS), IMMED`, loads the byte from memory at the address in register RS and puts the unsigned byte in register RT, it also writes the memory location with IMMED. For example, in the code below memory location `0x1000` initially holds a 7. After the execution of the instruction the 7 is placed in the destination register, `r1`, and the memory location is written with 3 (the immediate). The behavior of the instruction is undefined when `RS` and `RT` are the same registers. (That is, don't worry about, say, `lsb r1, (r1), 5`.)

```
# Before:  r2 = 0x1000   Mem[0x1000] = 7
lsb $r1, ($r2), 3
# After:   r1 = 7        Mem[0x1000] = 3
```

(*a*) Add this new instruction to the MIPS implementation attached to this assignment by modifying the Verilog.

- Note that the immediate is not used to compute the address.
- The memory port cannot simultaneously read and write.
- Try to minimize the number of new registers used.

☑ Modify the Verilog so that the implementation can execute the new `lsb` instruction.

Look for `SOL` in the right margin of the Verilog listing at the end of this assignment. Here we'll walk through highlights of the changes.

Our new instruction must do both a load and a store. The memory port does not have a combined read/write operation so we'll need to do the load and store in different states. As with other loads, we'll do the load part in `ST_id` and use a new state, `ST_lsb`, to do the store. What makes this store different than existing MIPS stores is that it stores the immediate value (other stores store the `rt` register value). So we'll need to add that capability.

Enough planning, lets start. We'll start with easy stuff, declaring a new `opcode` and state constants:

```
parameter   O_sb   = 6'h28;
parameter   O_xxx  = 6'h30;                                          // XXX
parameter   O_lsb  = 6'h31;                                          // SOL

...

parameter   ST_jt = 5;
parameter   ST_xxx = 6;                                              // XXX
parameter   ST_lsb = 7;                                              // SOL
```

Another easy thing to do is adding a multiplexor to the memory data in port. Here's the code, with new control signal `xmin` and constant `MI_rt` (memory data in select rt). (Before this change `mem_data_in` was connected only to `rt_val`.)

```
// Connect memory data in port to rt_val output of register file.
//
assign    mem_data_in = xmin == MI_rt ? rt_val : uimm;               // SOL
```

In the `ID` state we'll add control signals to command the ALU to compute the memory address. Hmm, for `lsb` the memory address is just the `rs` value, we shouldn't add the immediate. What to do? Okay, let's add a new ALU operation called *pass 1*, which passes the upper ALU input to the output unchanged. Below we use that operation, and we've made a new op constant `OP_p1`. Notice that since the lower ALU input is ignored the `xalu2` control signal has no effect.

```
            //              xrwr    xalu1   oalu    xalu2
            O_lbu, O_lw:
                    bndl = {DST_rt, SRC_rs, OP_add, SRC_si };
```

```
      O_xxx:  bndl = {DST_0,  SRC_rs, OP_add, SRC_si };          // XXX
      O_lsb:  bndl = {DST_rt, SRC_rs, OP_p1,  SRC_si };          // SOL
      O_sb:   bndl = {DST_0 , SRC_rs, OP_add, SRC_si };
```

We need to add control signals to do the load. We'll just copy the `lbu` values:

```
    // Determine values for omem_size, me_wb, and xrws.
    //
    case ( opcode )
      O_lbu   : begin omem_size = 1;  omem_wr = 0;  xrws = 0; end
      O_lsb   : begin omem_size = 1;  omem_wr = 0;  xrws = 0; end// SOL
```

We've set things up so that at the end of `ID` we write `rt` with the loaded value. Here we set the next state to our new state, where we'll store the immediate.

```
    // Determine value for next_state.
    //
    case ( opcode )
      O_bne  : next_state = alu_out_z ? ST_bt : ST_ni;
      O_beq  : next_state = alu_out_z ? ST_ni : ST_bt;
      O_xxx  : next_state = ST_xxx;                              // XXX
      O_lsb  : next_state = ST_lsb;                              // SOL
```

In our new state we set the ALU control signals to once again compute the memory address, and we also set control signals to command the memory to store a byte. Here we set our new memory-data-in multiplexor to write the immediate (by setting `xmin`). Finally, we set the next state to `ST_ni`, which starts the process of fetching the next instruction.

```
    ST_lsb:                                                      // SOL
      begin                                                      // SOL
        xalu1 = SRC_rs;                                          // SOL
        oalu = OP_p1;                                            // SOL
        xma = MA_alu;                                            // SOL
        xmin = MI_ui;                                            // SOL
        omem_size = 1;                                           // SOL
        omem_wr = 1;                                             // SOL
        next_state = ST_ni;                                      // SOL
      end                                                        // SOL
```

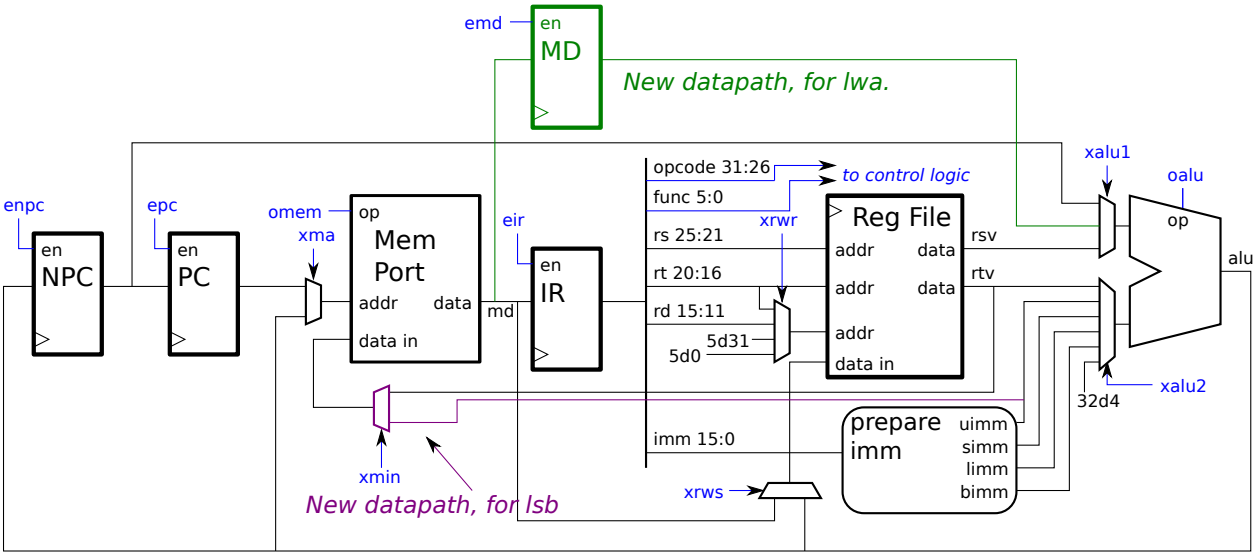
We're not quite done. We need to modify the ALU for our new pass 1 operation. That's easy:

```
always @*
  case ( alu_op )
    OP_add  : alu_out = alu_1 + alu_2;
    OP_p1   : alu_out = alu_1;                                   // SOL
    OP_and  : alu_out = alu_1 & alu_2;
    OP_or   : alu_out = alu_1 | alu_2;
```

(*b*) Add the new datapath components needed for `lsb` to the diagram below.

☑ Show new datapath.

Solution appears below in purple.



# Very Simple MIPS Implementation

*Also available at* `http://www.ece.lsu.edu/ee3755/2013f/mips-vsi-fe12-sol.v.html`.

```
module cpu(exc,
           mem_data_in, mem_addr, omem_size, omem_wr, mem_data_out,
           mem_error_in, reset, clk);
   input wire [31:0] mem_data_out;
   input wire [2:0]  mem_error_in;
   input wire reset,clk;
   output reg [7:0] exc;
   output wire [31:0] mem_data_in;
   output reg [31:0] mem_addr;
   output reg [1:0]  omem_size;
   output reg  omem_wr;

   // Values for the MIPS funct field.
   //
   parameter   F_sll = 6'h0;
   parameter   F_srl = 6'h2;
   parameter   F_add = 6'h20;
   parameter   F_sub = 6'h22;
   parameter   F_or  = 6'h25;
   parameter   F_swap = 6'h26;

   // Values for the MIPS opcode field.
   //
   parameter   O_rfmt = 6'h0;
   parameter   O_j    = 6'h2;
   parameter   O_beq  = 6'h4;
   parameter   O_bne  = 6'h5;
   parameter   O_addi = 6'h8;
   parameter   O_slti = 6'ha;
   parameter   O_andi = 6'hc;
   parameter   O_ori  = 6'hd;
   parameter   O_lui  = 6'hf;
   parameter   O_lw   = 6'h23;
   parameter   O_lbu  = 6'h24;
   parameter   O_sw   = 6'h2b;
   parameter   O_sb   = 6'h28;
   parameter   O_xxx  = 6'h30;                                          // XXX
   parameter   O_lsb  = 6'h31;                                          // SOL

   // Processor Control Logic States
   //
   parameter   ST_if = 1;
   parameter   ST_id = 2;
   parameter   ST_ni = 3;
   parameter   ST_bt = 4;
   parameter   ST_jt = 5;
   parameter   ST_xxx = 6;                                              // XXX
   parameter   ST_lsb = 7;                                              // SOL

   // ALU Operations
   //
   parameter   OP_nop = 6'd0;
   parameter   OP_sll = 6'd1;
   parameter   OP_srl = 6'd2;
   parameter   OP_add = 6'd3;
   parameter   OP_sub = 6'd4;
   parameter   OP_or  = 6'd5;
   parameter   OP_and = 6'd6;
   parameter   OP_slt = 6'd7;
   parameter   OP_seq = 6'd8;
```

```
parameter   OP_jp  = 6'd9;
parameter   OP_p1  = 6'd10;                                          // SOL

// Control Settings for alu_1 Multiplexer
//
parameter SRC_rs = 2'd0;
parameter SRC_sa = 2'd1;
parameter SRC_npc = 2'd2;
parameter SRC_md = 2'd3;                                            // XXX

// Control Settings alu_2 Multiplexer
//
parameter SRC_rt = 3'd0;
parameter SRC_ui = 3'd1;
parameter SRC_si = 3'd2;
parameter SRC_li = 3'd3;
parameter SRC_bi = 3'd4;
parameter SRC_ji = 3'd5;
parameter SRC_4 = 3'd6;

// Control Setting for Writeback Register Number Multiplexer
//
parameter DST_0 = 2'd0;
parameter DST_rt = 2'd1;
parameter DST_rd = 2'd2;
parameter DST_31 = 2'd3;

// Control Setting for Memory Address Multiplexer
//
parameter MA_pc = 1'b0;
parameter MA_alu = 1'b1;

/// SOLUTION - Control Settings for Data In Mux
//
parameter MI_rt = 1'b0;                                              // SOL
parameter MI_ui = 1'b1;                                              // SOL


///
/// Datapath Registers
///

reg [31:0] pc, npc;
reg [31:0] ir;               // Instruction Register
reg [31:0] gpr [0:31];

reg [31:0] md;                                                       // XXX

///
/// Datapath "Wires"
///

// Instruction Fields
//
reg [4:0]  rs, rt, rd, sa;
reg [5:0]  opcode, func;
wire [25:0] ii;
wire [15:0] immed;

// Values Derived From Instruction Fields and Read From Register File
//
wire [31:0] simm, uimm, limm, bimm, jimm;
wire [31:0] rs_val, rt_val, sa_val;
```

```
reg [4:0]   gpr_dst_reg;
wire [31:0] gpr_data_in;


//
// ALU and ALU Connections
//
wire [31:0] alu_out;
reg [31:0]  alu_1, alu_2;
reg [5:0]   oalu;

alu our_alu(alu_out, alu_1, alu_2, oalu);


//
/// Control Logic Declarations
//

// Write-enable signals for npc, pc, and ir.
reg         enpc, epc, eir;
reg         emd;                                                    // XXX

// Memory Address Multiplexer Control Signal
reg         xma;

/// SOLUTION - Memory Data In Multiplexer
reg         xmin;                                                   // SOL

// Register File Multiplexer Control Signals
reg    xrws;
reg [1:0]   xrwr;

// ALU Multiplexers Control Signals
reg [1:0]   xalu1;
reg [2:0]   xalu2;

// Processor Control Logic State
//
reg [2:0]  state, next_state;

reg [75:0] bndl; // Collection of control signals.


///
/// Initialization (Simulator Only)
///
// cadence translate_off
initial begin
   exc = 0;
   state = ST_if;
   func = 0;
   opcode = 0;
   ir = 0;
   xalu1 = 0;
   xalu2 = 0;
   oalu = 0;
end
// cadence translate_on

///
/// Register Write
///
```

```
// Write "real" registers at end of clock cycle. We expect
// these to be the only registers that will be synthesized.
//
always @( posedge clk )
  if ( reset ) begin
     pc <= 32'h400000;
     npc <= 32'h400004;
     ir <= 32'd0;
     state <= ST_if;
  end else begin
     if ( epc ) pc <= npc;
     if ( enpc ) npc <= alu_out;
     if ( eir ) ir <= mem_data_out;
     if ( emd ) md <= mem_data_out;                                    // XXX
     if ( gpr_dst_reg ) gpr[gpr_dst_reg] <= gpr_data_in;
     state <= next_state;
  end


///
/// Memory Port Connections
///

// Memory Address Multiplexer
//
always @*
  case ( xma )
    MA_pc:  mem_addr = pc;
    MA_alu: mem_addr = alu_out;
  endcase

// Connect memory data in port to rt_val output of register file.
//
//  assign    mem_data_in = rt_val;
assign    mem_data_in = xmin == MI_rt ? rt_val : uimm;

///
/// Extract IR Fields and Compute Some Values
///

// Extract fields from IR (for convenience).
always @* {opcode,rs,rt,rd,sa,func} = ir;
assign ii = ir[25:0];
assign immed  = ir[15:0];

assign uimm = { 16'h0, immed };
assign simm = { immed[15] ? 16'hffff : 16'h0, immed };
assign limm = { immed, 16'h0 };
assign bimm = { immed[15] ? 14'h3fff : 14'h0, immed, 2'b0 };
assign jimm = { 4'b0, ii, 2'b0 };

assign sa_val = {26'd0,sa};

//
/// Register File (GPR) Connections
//

assign rs_val = gpr[rs];
assign rt_val = gpr[rt];

// GPR Destination Register Number Multiplexer
//
always @*
```

```
   case ( xrwr )
     DST_rt: gpr_dst_reg = rt;
     DST_rd: gpr_dst_reg = rd;
     DST_31: gpr_dst_reg = 5'd31;
     DST_0 : gpr_dst_reg = 5'd0;
   endcase

// Source of data written to the register file.
//
assign gpr_data_in = xrws ? alu_out : mem_data_out;


///
/// ALU Connections
///

// Upper ALU Input Multiplexer
//
always @*
  case ( xalu1 )
    SRC_rs: alu_1 = rs_val;
    SRC_sa: alu_1 = sa_val;
    SRC_npc: alu_1 = npc;
    SRC_md: alu_1 = md;                                                  // XXX
    default: begin
      alu_1 = rs_val;
      // cadence translate_off
      $display("Unexpected ALU 1 source, %d\n", xalu1); $stop;
      // cadence translate_on
    end
  endcase

// Lower ALU Input Multiplexer
//
always @*
  case ( xalu2 )
    SRC_rt: alu_2 = rt_val;
    SRC_si: alu_2 = simm;
    SRC_ui: alu_2 = uimm;
    SRC_li: alu_2 = limm;
    SRC_bi: alu_2 = bimm;
    SRC_ji: alu_2 = jimm;
    SRC_4 : alu_2 = 32'd4;
    default: begin
      alu_2 = bimm;
      // cadence translate_off
      $display("Unexpected ALU 2 source, %d\n", xalu2); $stop;
      // cadence translate_on
    end
  endcase

// Set to 1 if output of ALU is zero.
//
wire        alu_out_z;
assign      alu_out_z = alu_out[0] == 0;


///
/// Control Logic
///
always @* begin

   ///
```

```
/// Default Values
///

// The "enable" signals which control whether a register is written.
enpc = 0;
epc = 0;
eir = 0;
emd = 0;                                                          // XXX

// Memory Control Signals
xma = MA_pc;    // Multiplexer connected to memory address port.
xmin = MI_rt;   // Memory data in.                                 // SOL
omem_wr = 0;    // If 1, write, if 0 read.
omem_size = 0;  // Size of value loaded from memory; 0 means do nothing.

// Register File Signals
xrwr = DST_0;   // Where to get the register number from.
xrws = 1;       // Source of data to write; 0, memory;  1, alu.

// ALU Control Signals
xalu1 = SRC_rs; // Upper ALU input.
xalu2 = SRC_rt; // Lower ALU input.
oalu = OP_add;  // ALU operation.

case ( state )

  /// IF: Instruction Fetch State.
  //
  ST_if:
    begin
       xma = MA_pc;
       omem_wr = 0;
       omem_size = 3;
       eir = 1;
       next_state = ST_id;
    end

  /// ID: Instruction Decode, Register Read, Execute, Writeback State
  //     Note: It would be better to break this into multiple steps.
  ST_id:
    begin

       // Determine values for xrwr, xalu1, oalu, and xalu2.
       //
       case ( opcode )

         O_rfmt:
           // R-Format Instructions
           case ( func )
             //            xrwr    xalu1   oalu    xalu2
             F_add: bndl = {DST_rd, SRC_rs, OP_add, SRC_rt};
             F_or:  bndl = {DST_rd, SRC_rs, OP_or,  SRC_rt};
             F_sub: bndl = {DST_rd, SRC_rs, OP_sub, SRC_rt};
             F_sll: bndl = {DST_rd, SRC_sa, OP_sll, SRC_rt};
             F_srl: bndl = {DST_rd, SRC_sa, OP_srl, SRC_rt};
             default:
               // Unrecognized instruction. Set exc to alert testbench.
               begin bndl = {DST_rd, SRC_sa, OP_sll, SRC_rt}; exc = 1; end
           endcase

         // I- and J-Format Instructions
         //             xrwr    xalu1   oalu    xalu2
         O_lbu, O_lw:
```

```
            bndl = {DST_rt, SRC_rs, OP_add, SRC_si };
      O_xxx:  bndl = {DST_0,  SRC_rs, OP_add, SRC_si };          // XXX
      O_lsb:  bndl = {DST_rt, SRC_rs, OP_p1,  SRC_si };          // SOL
      O_sb:   bndl = {DST_0 , SRC_rs, OP_add, SRC_si };
      O_lui:  bndl = {DST_rt, SRC_rs, OP_or,  SRC_li };
      O_addi: bndl = {DST_rt, SRC_rs, OP_add, SRC_si };
      O_andi: bndl = {DST_rt, SRC_rs, OP_and, SRC_ui };
      O_ori:  bndl = {DST_rt, SRC_rs, OP_or,  SRC_ui };
      O_slti: bndl = {DST_rt, SRC_rs, OP_slt, SRC_si };
      O_j:    bndl = {DST_0,  SRC_rs, OP_add, SRC_si };
      O_bne, O_beq:
            bndl = {DST_0,  SRC_rs, OP_seq, SRC_rt };
      default:
        // Unrecognized instruction. Set exc to alert testbench.
        begin bndl = {DST_0, SRC_rs, OP_seq, SRC_rt }; exc = 1; end
    endcase

    { xrwr, xalu1, oalu, xalu2 } = bndl;


    // Determine values for omem_size, me_wb, and xrws.
    //
    case ( opcode )
      O_lbu   : begin omem_size = 1;  omem_wr = 0;  xrws = 0; end
      O_lsb   : begin omem_size = 1;  omem_wr = 0;  xrws = 0; end// SOL
      O_xxx   : begin omem_size = 3;  omem_wr = 0;  xrws = 0; end// XXX
      O_lw    : begin omem_size = 3;  omem_wr = 0;  xrws = 0; end
      O_sb    : begin omem_size = 1;  omem_wr = 1;  xrws = 1; end
      default : begin omem_size = 0;  omem_wr = 0;  xrws = 1; end
    endcase

    // Determine value for xma.
    //
    // Note: for loads and stores xma must be MA_alu. For
    // other instructions it doesn't matter, so it's being
    // set to MA_alu to simplify the logic.
    //
    xma = MA_alu;

    emd = 1;                                                     // XXX

    // Determine value for next_state.
    //
    case ( opcode )
      O_bne  : next_state = alu_out_z ? ST_bt : ST_ni;
      O_beq  : next_state = alu_out_z ? ST_ni : ST_bt;
      O_xxx  : next_state = ST_xxx;                              // XXX
      O_lsb  : next_state = ST_lsb;                              // SOL
      O_j    : next_state = ST_jt;
      default: next_state = ST_ni;
    endcase
  end

ST_xxx:                                                          // XXX
  begin                                                          // XXX
    xalu1 = SRC_md;                                              // XXX
    xalu2 = SRC_rt;                                              // XXX
    oalu = OP_add;                                               // XXX
    xrws = 1;                                                    // XXX
    xrwr = DST_rt;                                               // XXX
    next_state = ST_ni;                                          // XXX
  end                                                            // XXX
```

```
        ST_lsb:                                                         // SOL
          begin                                                         // SOL
             xalu1 = SRC_rs;                                            // SOL
             oalu = OP_p1;                                              // SOL
             xma = MA_alu;                                              // SOL
             xmin = MI_ui;                                              // SOL
             omem_size = 1;                                             // SOL
             omem_wr = 1;                                               // SOL
             next_state = ST_ni;                                        // SOL
          end                                                           // SOL

        /// NI: Next Instruction State
        //  Compute the address of the next instruction.
        ST_ni:
          begin
             xalu1 = SRC_npc;
             xalu2 = SRC_4;
             oalu = OP_add;
             enpc = 1;
             epc = 1;
             next_state = ST_if;
          end

        /// BT: Branch Target State
        //  Compute the address of the branch target.
        ST_bt:
          begin
             xalu1 = SRC_npc;
             xalu2 = SRC_bi;
             oalu = OP_add;
             enpc = 1;
             epc = 1;
             next_state = ST_if;
          end

        /// JT: Jump Target State
        //  Compute the address of the jump target.
        ST_jt:
          begin
             xalu1 = SRC_npc;
             xalu2 = SRC_ji;
             oalu = OP_jp;
             enpc = 1;
             epc = 1;
             next_state = ST_if;
          end

        /// Illegal State
        //  It should be impossible to reach this state, but
        //  if we do print a helpful error message.
        default:
          begin
             next_state = ST_if;  // Avoid register for next_state;
             // cadence translate_off
             $display("Unexpected state, %d.\n", state);
             $stop;
             // cadence translate_on
          end

      endcase
    end

endmodule
```

```
module alu(alu_out,alu_1,alu_2,alu_op);
   output reg [31:0] alu_out;
   input wire [31:0] alu_1, alu_2;
   input wire [5:0]  alu_op;

   // Control Signal Value Names
   parameter  OP_nop = 0;
   parameter  OP_sll = 1;
   parameter  OP_srl = 2;
   parameter  OP_add = 3;
   parameter  OP_sub = 4;
   parameter  OP_or  = 5;
   parameter  OP_and = 6;
   parameter  OP_slt = 7;
   parameter  OP_seq = 8;
   parameter  OP_jp  = 9;
   parameter  OP_p1  = 6'd10;                                          // SOL

   always @*
     case ( alu_op )
       OP_add  : alu_out = alu_1 + alu_2;
       OP_and  : alu_out = alu_1 & alu_2;
       OP_or   : alu_out = alu_1 | alu_2;
       OP_sub  : alu_out = alu_1 - alu_2;
       OP_slt  : alu_out = {alu_1[31],alu_1} < {alu_2[31],alu_2};
       OP_sll  : alu_out = alu_2 << alu_1;
       OP_srl  : alu_out = alu_2 >> alu_1;
       OP_seq  : alu_out = alu_1 == alu_2;
       OP_jp   : alu_out = { alu_1[31:26], alu_2[25:0] };
       OP_p1   : alu_out = alu_1;                                      // SOL
       OP_nop  : alu_out = 0;
       default :
         begin
            alu_out = 0;
            // cadence translate_off
            $display("Unrecognized alu op, %d",alu_op);
            $stop;
            // cadence translate_on
         end
     endcase

endmodule
```