Name Solution

Computer Organization
EE 3755
Final Examination

Tuesday, 4 December 2012, 7:30–9:30 CST

Problem 1		(15 pts)
Problem 2		(15 pts)
Problem 3		(20 pts)
Problem 4		(20 pts)
Problem 5		(5 pts)
Problem 6		(5 pts)
Problem 7		(20 pts)
Exam Total		(100 pts)

Alias Just Me

Good Luck!

Note: The hardwired-control MIPS implementation which is the subject of Problems 1 and 2 was not covered in the Fall 2013 semester. To prepare for the 2013 (and future) final exams an alternative practice question which is similar to the two following problems but uses the Very Simple MIPS covered in Fall 2013 and been posted as Fall 2013 Homework 6. The two implementations (Very Simple and Hardwired Control) are very similar. Very Simple is written in a Verilog style that makes it easier to 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.

Problem 1: [15 pts] The MIPS implementation attached to this exam can execute a new instruction, `xxx`. Lines relevant to the instruction have `XXX` on the right hand side.

(*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`.

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

☑ Describe what `xxx` does.

It loads a word from memory address `rs` + `immed` and adds that word (interpreted as an integer) to the contents of `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
addmto r3, 4(r2)   # r3 = r3 + Mem[ r2 + 4 ]
```

☑ Code doing same thing but without `xxx`:

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

Problem 2: [15 pts] The following new instruction is to be implemented on the multi-cycle MIPS implementation attached to the exam. The instruction, `lsb RT, (RS), IMMED`, loads the byte from memory at the address in register RS and puts the 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).

```
# 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 exam.

- 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.

☑ Add the `lsb` instruction to attached implementation.

Look for the word **SOLUTION** in the right margin of the Verilog listing at the end of this exam.

Problem 3: [20 pts] Answer each of the following MIPS programming questions.

(*a*) Show the shortest sequence of MIPS instructions needed to load the following constants or memory locations into register `t0`. The solution for the first constant is given.

☑ Instruction(s) to load `0x7` into `t0`.

```
# Example Solution
addi $t0, $0, 7
```

☑ Instruction(s) to load `0xa30bf18a` into `t0`.

```
# Solution
lui t0, 0xa30b
ori t0, t0, 0xf18a
```

☑ Instruction(s) to load `0xa30b` into `t0`.

```
# Solution
ori t0, r0, 0xa30b
```

☑ Instruction(s) to load `0xa30b0000` into `t0`.

```
# Solution
lui t0, 0xa30b
```

☑ Instruction(s) to load word at memory address `0xa30b018c` into `t0`.

```
# Solution
lui t1, 0xa30b
lw t0, 0x18c(t1)
```

(*b*) The code fragment below loads two items from memory, adds them together, then stores the sum. It does so using more instructions than are necessary. Re-write the code so that it uses fewer instructions.

- A correct solution has only four instructions.
- Solution should take into account that MIPS byte order is big-endian.

```
lw $t0, 0($t1)
andi $t0, $t0, 0xffff
addi $t1, $t1, 6
lh $t2, 0($t1)
srl $t2, $t2, 8
andi $t2, $t2, 0xff
add $t3, $0, $0
add $t3, $t0, $t2
addi $t1, $t1, 2
sw $t3, 0($t1)
# Registers $t1-$t3 no longer used at this point.
```

☑ Re-written code using as few instructions as possible.

```
# SOLUTION
lhu $t0, 2($t1)
lbu $t2, 6($t1)
add $t3, $t0, $t2
sw $t3, 8($t1)
```

(*c*) Fill the delay slot in the MIPS code below by moving an instruction (without changing what the code does, of course).

☑ Fill delay slot.

```
        addi $t4, $t4, 5
        add $t1, $t1, $t4
        beq $t0, $t1, SKIP
        nop
        add $t2, $t2, $t3
SKIP:
        addi $t3, $t3, 1
        addi $t4, $t4, 1
```

```
        # SOLUTION
        addi $t4, $t4, 5
        add $t1, $t1, $t4
        beq $t0, $t1, SKIP
        addi $t4, $t4, 1    # This instruction replaces the nop.
        add $t2, $t2, $t3
SKIP:
        addi $t3, $t3, 1
```

Problem 4: [20 pts] Consider the logic that would be synthesized for `the_A_block` in the `multiplier` module below.

```
module multiplier(product,ready,multiplicand,multiplier,start,clk);
   input [15:0]  multiplicand, multiplier;
   input         start, clk;
   output        product, ready;
   reg [31:0]    product;
   reg [4:0]     bit;
   wire          ready = !bit;
   wire [17:0]   multiplicand_X_1 = {2'b0,multiplicand};
   wire [17:0]   multiplicand_X_2 = {1'b0,multiplicand,1'b0};
   wire [17:0]   multiplicand_X_3 = multiplicand_X_2 + multiplicand_X_1;

   initial bit = 0;

   always @( posedge clk )

     if ( ready && start ) begin

        bit     = 8;
        product = { 16'd0, multiplier };

     end else if ( bit ) begin:the_A_block

        reg [17:0]    hs;

        case ( product[1:0] )
          2'd0: hs = {2'b0, product[31:16] };
          2'd1: hs = {2'b0, product[31:16] } + multiplicand_X_1;
          2'd2: hs = {2'b0, product[31:16] } + multiplicand_X_2;
          2'd3: hs = {2'b0, product[31:16] } + multiplicand_X_3;
        endcase

        product = { hs, product[15:2] };
        bit     = bit - 1;

     end
endmodule
```

Problem 4, continued:

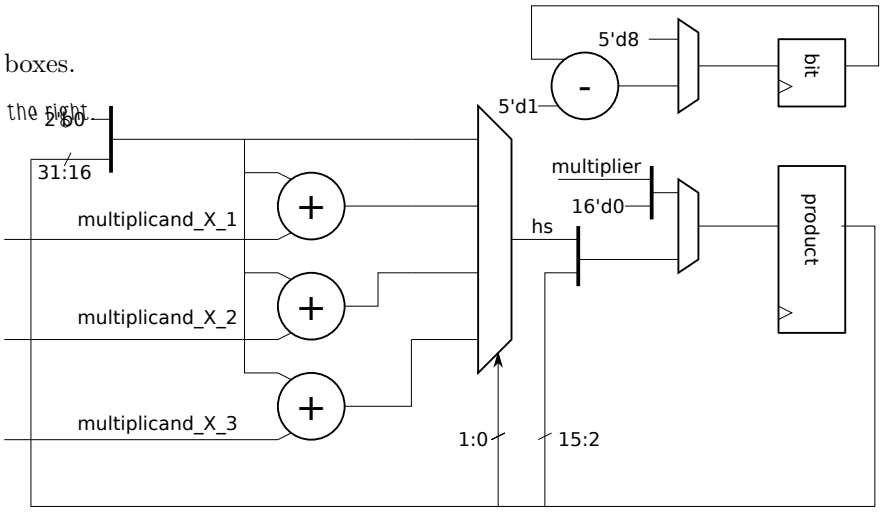
(*a*) Sketch the logic that would be synthesized for `the_A_block` without optimization. Treat `multiplicand_X_1`, `multiplicand_X_2`, and `multiplicand_X_3` as inputs to this logic.

☑ Show logic for `the_A_block`.

☑ Clearly mark registers with edge-trigger symbols.

☑ Show adders as boxes.

Solution appears to the right.



(*b*) An engineer fears that even with optimization the logic for `the_A_block` will contain more adders than necessary because of the way the `case` statement is used. Re-write the `case` statement and surrounding code so that even without optimization one adder is used. (Don't count the adders used to compute `multiplicand_X_3` and `bit`.)

☑ Re-write block to eliminate chance of unnecessary adders.

Solution appears below. The addition is done after the case statement, rather than in each case item. This avoids the chance that the synthesis program will use three adders instead of just one.

```
end else if ( bit ) begin:the_A_block

   reg [17:0]    hs;
   reg [17:0]    pp;

   case ( product[1:0] )
     2'd0: pp = 18'b0;
     2'd1: pp = multiplicand_X_1;
     2'd2: pp = multiplicand_X_2;
     2'd3: pp = multiplicand_X_3;
   endcase

   hs = {2'b0, product[31:16] } + pp;
   product = { hs, product[15:2] };
   bit     = bit - 1;

end
```

The material in Problems 5 and 6 was covered only one time and will not be covered again. **Please ignore these questions.**

Problem 5: [5 pts] A new MIPS implementation is being designed for a customer. Energy consumption can be reduced by retrieving register values only for those instructions that use them. The logic to detect whether the registers will be used requires 100 gates. The static power usage of these gates will reduce the energy savings by 50%. For a MIPS-like instruction set which is identical except for encoding (that is, the assembly language is the same but the encoded instruction differ) almost no logic is needed to detect whether a register is used and so the full energy savings can be realized. Since the MIPS-like instruction set has a different encoding than MIPS, programs will have to be recompiled before they can run on the implementation. An implementation of just MIPS will run existing code.

In summary, the ordinary MIPS implementation will save some energy, but can run existing code unmodified. The MIPS-like implementation will save more energy, but code needs to be re-compiled.

Consider two types of customers: one that runs large data centers, and one that makes set-top boxes for cable companies (which are government regulated utilities).

(*a*) Describe how receptive the data-center operator would be to the MIPS-like implementation. What arguments would you need to make in its favor?

☑ Data center customer receptiveness to MIPS-like implementation?

The data center operator would be receptive because energy is a big part of their costs. Re-compiling their code would be little trouble and the savings would be large.

☑ Arguments that can be made for it to them:

Since they knowledgeable about energy consumption one could provide technical data, perhaps the energy used to run some sample programs.

(*b*) Describe how receptive the cable box manufacturer will be to the MIPS-like implementation. What might persuade them to choose the MIPS-like implementation?

☑ Cable box customer receptiveness to MIPS-like implementation?

Cable boxes are bought by cable operators (such as Cox, Comcast, Eatel) and rented to their subscribers (that's us). Very few people cancel their cable service because the cable box uses too much power. So there is no incentive to develop a lower-power box.

☑ Arguments that can be made for it to them:

If you don't do it, the government will make you do it.

Problem 6: [5 pts] Analysis of a new MIPS implementation indicates that if registers `r1` to `r9` are written with a particular set of values then the contents of `r31` will replaced with the contents of `r10`. This will only occur with one exact set of values in `r1`-`r9`, that's one set out of $2^{256} \approx 1.16 \times 10^{77}$ possible. Such a set of values are essentially impossible to occur by chance. Fixing this problem will delay the release of the MIPS implementation by four months.

☑ Should this problem be fixed? Explain.

Register `r31` is the return address register, if its value is "accidentally" changed then the program will jump to an "unexpected" place. Suppose the new implementation is used in a computer running a Web server. Someone who is familiar with a Web sites code and the source of the Web server, might figure out how to fill in blanks on a Web page in such a way to get the particular values in `r1`-`r9`, and also how to get their preferred value in `r10`, and due to the flaw, in `r31`. That preferred value will cause the return to jump to their own code, thus compromising the server.

So even though the values for `r1` to `r9` might never occur by accident, they could be discovered and placed their intentionally for some mischievous, or worse, purpose.

Problem 7: [20 pts] Answer each question below.

(*a*) What is wrong with the following statement: "An assembler should recognize just a few pseudo instructions, such as `nop` for MIPS, but adding too many more pseudo instructions would make the hardware too complicated."

☑ Why statement is wrong:

A pseudo instruction is not a machine instruction at all, so it has no impact on hardware complexity. A pseudo instruction is a convenience provided by an assembler to make it easier on an assembly language programmer. An example in MIPS is `nop`. Programmers use pseudo instructions in assembler code as though they were real instructions, but the assembler will substitute a true machine instruction for them. For example, `nop` is replaced by `sll r0, r0, 0`.

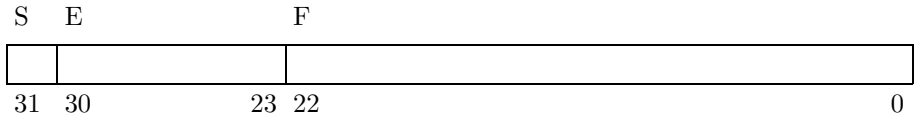
(*b*) *Technology mapping* is one of the steps taken by a typical synthesis program.

☑ What happens during technology mapping?

The generic gates from the user-written Verilog (or libraries) code are replaced by devices that are available in the target (the type of chip being synthesized). For example, the target might have 2-, 4-, and 8-input AND gates, so in technology mapping three-input AND gates would be replaced by perhaps a 4-input gate with one input connected to logic 1. (Or perhaps to two 2-input AND gates.)

(*c*) Show the IEEE 754 single-precision representation of 1280 (which is $2^{10} + 2^8$). Just show the different parts, sign, biased exponent, and significand; there is no need to show it as a single hexadecimal number.

☑ Show IEEE 754 single-precision rep. of 1280.



First, re-write 1280 in binary scientific notation:

$$1280_{10} = 101\,0000\,0000_2 = 1.01_2 \times 2^{10}$$

Since it's positive the sign bit is $S = 0$. To obtain the biased exponent add 127 to the exponent: $E = 10 + 127 = 137$. To obtain the fraction remove the leading 1 and add enough trailing zeros to make it 23 bits long: $F = 01000000000000000000000_2$.

(*d*) The functional simulation (single-cycle) implementation of MIPS presented in class uses more hardware than the multi-cycle implementation.

☑ Provide an example of how it uses more hardware.

There is a separate adder for branch targets and for the addition needed by instructions.

☑ Explain why the single-cycle implementation must use more hardware than the multi-cycle implementation.

Since everything is done in one cycle, there is no way to first use a unit, such as an adder, for one purpose and then switch to another purpose. A unit's input might come from several possible sources, connected to multiplexors at the unit inputs. The multiplexor control signal selects which of those inputs to use. To use the unit for two purposes one would need to switch the multiplexors from one input to a different input *in the middle of a clock cycle* and that can't be done reliably.

EE 3755 Fall 2012 Final Exam Appendix
Multi-cycle MIPS Implementation

Studying for the Fall 2013 or later final? See note at the beginning of Problem 1.

☑ Name: _____

```
module cpu(exc,data_out,addr,size,we,data_in,mem_error_in,reset,clk);
   input [31:0] data_in;
   input [2:0]  mem_error_in;
   input        reset,clk;
   output [7:0] exc;
   output [31:0] data_out, addr;
   output [1:0]  size;
   output        we;

   reg [31:0]    data_out, addr;
   reg [1:0]     size;
   reg           we;
   reg [7:0]     exc;

   // MIPS Registers
   //
   reg [31:0] gpr [0:31];
   reg [31:0] pc, npc;
   reg [31:0] ir;

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

   // Values Derived From Immediates and Read From Register File
   //
   reg [31:0] simmed, uimmed;
   reg [31:0] sa_val;

   reg [31:0] rs_val, rt_val;
   reg [75:0]     bndl;

   // ALU Connections
   //
   wire [31:0] alu_out;
   reg [31:0]  alu_a, alu_b;
   reg [5:0]   alu_op;

   // Processor Control Logic State
   //
   reg [3:0] state;

   reg [4:0] wb_rd;     // Register number to write.
   reg       me_we;     // we value to use in state st_me
   reg [1:0] me_size;   // size value to use in state st_me

   alu our_alu(alu_out, alu_a, alu_b, alu_op);

   // Values for the MIPS funct field.
   //
```

```
parameter  F_sll = 6'h0;     parameter  F_add = 6'h20;
parameter  F_srl = 6'h2;     parameter  F_sub = 6'h22;
parameter  F_or  = 6'h25;

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

// Processor Control Logic States
//
parameter  ST_if = 1;     parameter  ST_ex_addr = 5;
parameter  ST_id = 2;     parameter  ST_ex_cond = 6;
parameter  ST_ex = 3;     parameter  ST_ex_targ = 7;
parameter  ST_me = 4;
parameter  ST_xxx_1 = 8;                                            // XXX
parameter  ST_xxx_2 = 9;                                            // XXX

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

parameter  R0 = 5'd0;

/// Set Memory Connection Values: addr, we, and size.
//
always @( state or pc or alu_out or me_size or me_we )
  case ( state )
    ST_if   : begin addr = pc;       we = 0;      size = 3;       end
    ST_xxx_2: begin addr = alu_out;  we = me_we;  size = me_size; end  // XXX

    // Problem 2 Solution                                           // SOLUTION
    // Set memory connections for read portion of lsb instruction.
    ST_lsb_mr: begin addr = alu_out;  we = 0;      size = me_size; end
    // Problem 2 Solution                                           // SOLUTION
    // Set memory connections for write portion of lsb instruction.
    ST_lsb_mw: begin addr = alu_out;  we = 1;      size = me_size; end

    ST_me   : begin addr = alu_out;  we = me_we;  size = me_size; end
    default : begin addr = pc;       we = 0;      size = 0;       end
  endcase

always @( posedge clk )
  if ( reset ) begin
     state = ST_if;
     exc   = 0;
     pc    = 32'h400000;
     npc   = pc + 4;
  end else
  case ( state )

    ///       Instruction Fetch
    ST_if:
```

```
  begin
     ir    = data_in;
     state = ST_id;
  end

///       Instruction Decode (and Register Read)
ST_id:
  begin

     {opcode,rs,rt,rd,sa,func} = ir;
     ii     = ir[25:0];
     immed  = ir[15:0];

     simmed = { immed[15] ? 16'hffff : 16'h0, immed };
     uimmed = { 16'h0, immed };

     rs_val = gpr[rs];
     rt_val = gpr[rt];
     sa_val = {26'd0,sa};

     // Set alu_a, alu_b, alu_op, and wb_rd.
     //
     case ( opcode )

       O_rfmt:
         // R-Format Instructions
         case ( func )
           F_add: bndl = {rd, rs_val, OP_add, rt_val};
           F_sub: bndl = {rd, rs_val, OP_sub, rt_val};
           F_sll: bndl = {rd, sa_val, OP_sll, rt_val};
           default:
             begin bndl = {rd, sa_val, OP_sll, rt_val}; exc = 1; end
         endcase

       // I- and J-Format Instructions

       // Problem 2 solution.                                  // SOLUTION
       // Provide ALU settings to compute load and store address.
       // The operation OP_pa means that the ALU output is input A.
       // Input B to the ALU (simmed) is ignored.
       O_lsb:  bndl = {rt, rs_val, OP_pa,  simmed };         // SOLUTION

       O_lbu:  bndl = {rt, rs_val, OP_add, simmed };
       O_sb:   bndl = {R0, rs_val, OP_add, simmed };
       O_lui:  bndl = {rt, 32'd16, OP_sll, uimmed };
       O_addi: bndl = {rt, rs_val, OP_add, simmed };
       O_andi: bndl = {rt, rs_val, OP_and, uimmed };
       O_ori:  bndl = {rt, rs_val, OP_or,  uimmed };
       O_slti: bndl = {rt, rs_val, OP_slt, simmed };
       O_j:    bndl = {R0, rs_val, OP_nop, simmed };
       O_bne, O_beq: bndl = {R0, rs_val, OP_seq, rt_val };
       O_xxx:  bndl = {rt, rs_val, OP_add, simmed };               // XXX
       default: begin bndl = {R0, rs_val, OP_seq, rt_val }; exc = 1; end
     endcase

     { wb_rd, alu_a, alu_op, alu_b } = bndl;

     // Problem 2 solution.                                  // SOLUTION
     // Set data_out, used for memory write, to immediate if
     // this is an lsb instruction. Otherwise set it to the
     // rt value.
     data_out = opcode == O_lsb ? uimmed : rt_val;          // SOLUTION
```

```
    // Set me_size and me_wb
    //
    case ( opcode )
      O_lsb   : begin me_size = 1;  me_we = 0; end          // SOLUTION
      O_lbu   : begin me_size = 1;  me_we = 0; end
      O_sb    : begin me_size = 1;  me_we = 1; end
      O_xxx   : begin me_size = 3;  me_we = 0; end               // XXX
      default : begin me_size = 0;  me_we = 0; end
    endcase

    pc = npc;

    // Set npc, branch instruction may change npc.
    //
    case ( opcode )
      O_j     : npc = { pc[31:28], ii, 2'b0 };
      default : npc = pc + 4;
    endcase

    case ( opcode )
      // Problem 2 solution: Choose next state for lsb.
      O_lsb        : state = ST_lsb_mr;                     // SOLUTION
      O_lbu, O_sb  : state = ST_ex_addr;
      O_bne, O_beq : state = ST_ex_cond;
      O_j          : state = ST_if;
      O_xxx        : state = ST_xxx_1;                           // XXX
      default      : state = ST_ex;
    endcase

  end

// SOLUTION
/// LSB Memory Read State.
// Put loaded value in destination register.
// This logic is identical to ST_me (used for load instructions)
// except that the next state is ST_lsb_mw instead of ST_if.
ST_lsb_mr:                                                  // SOLUTION
  begin                                                     // SOLUTION
     if ( wb_rd ) gpr[wb_rd] = data_in;                     // SOLUTION
     state = ST_lsb_mw;                                     // SOLUTION
  end                                                       // SOLUTION
                                                            // SOLUTION

// SOLUTION
/// LSB Memory Write State.
//  See the assignment to data_out, above, to see how the
//  immediate is written to memory.
ST_lsb_mw:                                                  // SOLUTION
  begin                                                     // SOLUTION
     state = ST_if;                                         // SOLUTION
  end                                                       // SOLUTION

///       Execute -- ALU instructions
ST_ex:
  begin
     if ( wb_rd ) gpr[wb_rd] = alu_out;
     state = ST_if;
  end

///       Execute -- Compute Effective Address for Loads and Stores
ST_ex_addr:
  begin
     state = ST_me;
```

```
          end

        ///       Execute -- Compute Branch Condition
        ST_ex_cond:
          begin
             if ( opcode == O_beq && alu_out
                  || opcode == O_bne && !alu_out ) begin
                alu_a  = pc;
                alu_b  = simmed << 2;
                alu_op = OP_add;
                state  = ST_ex_targ;
             end else begin
                state = ST_if;
             end
          end

        ///       Execute -- Compute Branch Target
        ST_ex_targ: begin npc = alu_out; state = ST_if; end

        ///       Memory
        ST_me:
          begin
             if ( wb_rd ) gpr[wb_rd] = data_in;
             state = ST_if;
          end

        ///       XXX
        ST_xxx_1: begin state = ST_xxx_2; end                              // XXX
        ST_xxx_2:                                                          // XXX
          begin                                                            // XXX
             alu_a = data_in; alu_b = rt_val; alu_op = OP_add;             // XXX
             state = ST_ex;                                                // XXX
          end                                                              // XXX

        default:
          begin
             $display("Unexpected state.");
             $stop;
          end

      endcase

endmodule

module alu(alu_out,alu_a,alu_b,alu_op);
   output [31:0] alu_out;
   input [31:0]  alu_a, alu_b;
   input [5:0]   alu_op;

   reg [31:0]    alu_out;

   // 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_pa  = 9;                                          // SOLUTION
```

```
   // Problem 2 Solution
   // Add the pass-a operation to the ALU. All the pass a operation
   // does is set the ALU output to the A input.

   always @( alu_a or alu_b or alu_op )
     case ( alu_op )
       OP_add  : alu_out = alu_a + alu_b;
       OP_and  : alu_out = alu_a & alu_b;
       OP_or   : alu_out = alu_a | alu_b;
       OP_sub  : alu_out = alu_a - alu_b;
       OP_slt  : alu_out = {alu_a[31],alu_a} < {alu_b[31],alu_b};
       OP_sll  : alu_out = alu_b << alu_a;
       OP_srl  : alu_out = alu_b >> alu_a;
       OP_seq  : alu_out = alu_a == alu_b;
       OP_pa   : alu_out = alu_a;                                  // SOLUTION
       OP_nop  : alu_out = 0;
       default : begin alu_out = 0;  $stop;  end
     endcase

endmodule
```