Computer Organization
EE 3755
Final Examination
16 May 2002, 12:30-14:30 CDT

Problem 1 ________ (20 pts)
Problem 2 ________ (20 pts)
Problem 3 ________ (20 pts)
Problem 4 ________ (20 pts)
Problem 5 ________ (20 pts)

Exam Total ________ (100 pts)

Alias Edge Triggered

Good Luck!
Problem 1: The proposed (just for this exam) swap instruction swaps the contents of its register (GPR) operands. For example:

```
# Before:  $4 = 123,  $5 = 456
swap $4, $5
# After:   $4 = 456,  $5 = 123
```

Modify the MIPS implementation in the appendix so that it can execute the swap instruction.

(20 pts)

☑ Make up any field values that are needed.

☑ MIPS registers cannot be written in the ID state.

☑ At most one MIPS register can be written per cycle.

Grading notes: Better solutions make fewer changes to the code, synthesize into less hardware, and do not increase the critical path. The simplest strategy is to add two new states, write rs in one \((\text{if}(\text{rs})\text{gpr}[\text{rs}] = \text{rt}.\text{val})\) and rt in the other. That would add a new path into the register file. That is the approach used in the solution given here (at the end of the exam).

There is an existing path to the register file through the ALU. Several people tried using that path (not all of them did it for efficiency reasons) but none did it efficiently. The best way to use the ALU is to define two new ALU operations, Pass A \((\text{alu}.\text{out} = \text{alu}.a)\) and Pass B \((\text{alu}.\text{out} = \text{alu}.b)\). In decode assign \(\text{alu}.a = \text{rs}.\text{val}\) and \(\text{alu}.b = \text{rt}.\text{val}\), just like other Type I instructions.
Problem 2: The incomplete `split` procedure below copies a list of integers using a threshold. In the first part of the copy all numbers must be no greater than the threshold, in the second part all numbers must be above the threshold. For example, suppose the threshold were 5 and the list was 10, 3, 5, 12, 2. Then the copy would be: 3, 5, 2, 10, 12. Numbers in each part can be in any order, for example, the following is also a correct copy: 5, 3, 2, 12, 10. (20 pts)

- Fill as many delay slots as possible.
- The only pseudo instruction allowed is `nop`.
- Points will be deducted for obviously unnecessary instructions.

The solution appears below.

```
split:

    ## Register Usage
    #
    # $a0: Procedure input: Address of start of list of integers (words).
    # $a1: Procedure input: Number of integers.
    # $a2: Procedure input: Threshold for splitting list.
    # $a3: Procedure input: Address for copy (the list to be created).
    # No return value.
    # Any register can be modified.
    # A correct solution uses 17 instructions. More are okay.

    addi $t1, $a3, 0   # Start of new list.
    sll $t2, $a1, 2
    add $t4, $a0, $t2 # End of original list.
    add $t2, $t2, $a3 # End of new list.

    LOOP:
    beq $a0, $t4, DONE
    lw $t0, 0($a0)
    slt $t3, $a2, $t0
    bne $t3, $0, SMALL_END

    addi $a0, $a0, 4
    addi $t2, $t2, -4
    j LOOP
    sw $t0, 0($t2)

    SMALL_END:
    sw $t0, 0($t1)
    j LOOP
    addi $t1, $t1, 4

    jr $ra
    nop
```
Problem 3: The MIPS implementation in the appendix can execute two brand new instructions, xxx and yyy. So that you can find them quickly, some of the new lines added for these instructions have XXX or YYY comments in the right margin.

(a) What is xxx? (8 pts)

Choose a mnemonic (name) for xxx, show a possible assembly language syntax, and give a brief description of what it does.

naddi rt, rs, simmed

Instruction naddi writes gpr[rt] with simmed - gpr[rs].

Show an example of how xxx is used in an assembly language program, then show how the same thing is done using non-fictional MIPS instructions.

# Using nadd
naddi $t0, $t1, 123

# Using existing instructions.
sub $at, $0, $t1
addi $t0, $at, 123

(b) What is yyy? (12 pts)

Choose a mnemonic (name) for yyy, show a possible assembly language syntax, and give a brief description of what it does.

addm simmed(rs), rt

Instruction addm adds gpr[rt] to the half-word starting at address gpr[rs]+simmed. (That is, mem[gpr[rs]+simmed] = mem[gpr[rs]+simmed] + gpr[rt].) Note: Because it both accesses memory and performs arithmetic this is not the kind of instruction one would add to MIPS or any other RISC ISA.

Show an example of how yyy is used in an assembly language program, then show how the same thing is done using non-fictional MIPS instructions.

# Using yyy
addm 4($s1), $s2

# Existing instructions.
lh $at, 4($s1)
add $at, $at, $s2
sh $at, 4($s1)
Problem 4: Answer each synthesis question below.

(a) Show the hardware that will be synthesized for nnpc in the MIPS implementation at the end of the exam. Clearly show any registers and indicate whether they are edge- or level-triggered. All lines containing nnpc have an NNPC comment in the right margin. (12 pts)

Here is the Verilog code where nnpc is assigned:

```verilog
nnpc = npc + 4;
case( opcode )
  0_tyr :
    case( func )
      F_jr : nnpc = rs_val;
    endcase
  0_j :
    nnpc = {npc[31:28],ii,2'd0};
  0_jal : {gpr[31],nnpc} = {nnpc,npc[31:28],ii,2’d0};
endcase
bimmed = npc + ( simmed << 2 );
pc = npc;  npc = nnpc;
```

Consider the hardware with optimization. Note that by the time the last line is reached nnpc must be assigned one of three possible values, npc+4, rs_val, and (for both the jump and jump and link) a concatenation of the high bits of npc, the ii field and two zeros. The synthesis program will use a three-input mux to select the correct value for nnpc. Because it is assigned to nnpc and not used again no register is needed for nnpc, though a register is needed for npc. The control input for the mux is computed from the opcode and func fields, in the solution below that is shown by an oval. The problem only asked about the logic for the nnpc value, but the solution below also shows inputs to the npc registers.

Without optimization the synthesis program might use two multiplexors, because there are two case statements. That
(b) A reasonable synthesis program would not synthesize a register for $bndl$ but would synthesize a register for $exc$, both used in the ID state of the MIPS implementation. Why? (8 pts)

No register is needed for $bndl$ because there is no need to remember a value. That is, the value computed in the positive edge of a clock is used in the positive edge, but not after so there is no need to remember it.

A register is needed for $exc$ because it is assigned in Form 2 code and its value is needed after the clock edge (in $exc$'s case because it is a module output).

Because the code for NNPC, $bndl$, and $exc$ is in an always block in Form 2 it does not matter whether it is sometimes are always assigned, that only matters for Form 1.
Problem 5: Answer each question below.

(a) Show the intermediate values produced by the simple binary integer division algorithm when computing $64_{16} \div 1_{16}$. Indicate the quotient and the remainder. (5 pts)

Solution:

0x64 = 0110 0100
0x1a = 0001 1010

0110 0100 = 0x64
- 1101 0000 = 0x1a << 3

0110 0100 Unchanged
- 0110 1000 = 0x1a << 2

0110 0100 Unchanged
- 0011 0100 = 0x1a << 1

0011 0000 Subtract
- 0001 1010 = 0x1a << 0

Remainder -> 0001 0110 Subtract 0011 <- Quotient
(b) Modify the multiplier below so that it uses 16-bit adder module \texttt{add16} to do the partial-product addition (on the if( \texttt{lsb} ) line). Module \texttt{add16} has two 16-bit inputs and a 16-bit output and performs an unsigned addition using combinational logic. (5 pts)

- ✔ Show the instantiation of the adder.
- ✔ Declare any new wires or registers that are used.
- ✔ Use it to perform the addition.

Solution appears below. Note that \texttt{product} is shifted at the end of the cycle but is needed at an input to the adder near the beginning of the cycle and so one is added to the bit indices (30:15).

```verilog
module streamlined_mult(product,ready,multiplier,multiplicand,start,clk);
  input [15:0] multiplier, multiplicand;
  input start, clk;
  output product;
  output ready;
  reg [31:0] product;
  reg [4:0] bit;
  wire ready = !bit;
  initial bit = 0;
  wire [15:0] sum; // Sol
  add16 ppadder(sum,product[31:16],multiplicand); // Sol
  always @( posedge clk )
    if( ready && start ) begin
      bit = 16;
      product = { 16'd0, multiplier };
    end else if( bit ) begin:
      reg lsb;
      lsb = product[0];
      product = product >> 1;
      bit = bit - 1;
      if( lsb ) product[31:15] = sum; // Sol
    end
endmodule
```
(c) Write the assembly language for the following MIPS instruction: 01d82025₁₆. *Hint: The MIPS implementation in the appendix should be helpful.* (5 pts)

Be sure to include any registers and immediates present. Register numbers are okay.

```
0x01d82025
= 0000 0001 1101 1000 0010 0000 0010 0101
= 000000 01110 11000 00100 00000 100101

opcode rs rt rd sa func

Since opcode is zero must by type R.

or $4, $14, $24
```
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;

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

    reg [31:0] pc, npc, nnpc, npc_plus_4;
    reg [31:0] ir;
    wire [5:0] opcode, func;
    wire [4:0] rs, rt, rd, sa;
    wire [15:0] immed;
    wire [25:0] ii;

    reg [31:0] gpr [0:31];

    wire [31:0] simmed, uimmed;
    wire [31:0] limmed;
    reg [31:0] bimmed;

    reg [3:0] state, state_after_me; // Prob 1

    reg [4:0] dst;
    reg [74:0] bndl;
    reg [31:0] rs_val, rt_val, sa_val;

    parameter F_add = 6'h20;
    parameter F_sll = 6'h0;
    parameter F_srl = 6'h2;
    parameter F_sub = 6'h22;
    parameter F_or = 6'h25;
    parameter F_jr = 6'h8;
    parameter F_swp = 6'h26; // Prob 1
Hardwired Control MIPS Implementation

```verbatim
parameter 0_tyr = 6'h0;
parameter 0_addi = 6'h8;
parameter 0_j = 6'h2;
parameter 0_jal = 6'h3;
parameter 0_beq = 6'h4;
parameter 0_bne = 6'h5;
parameter 0_slti = 6'ha;
parameter 0_andi = 6'hc;
parameter 0_ori = 6'hd;
parameter 0_lui = 6'hf;
parameter 0_lw = 6'h23;
parameter 0_lbu = 6'h24;
parameter 0_lb = 6'h20;
parameter 0_sw = 6'h2b;
parameter 0_sb = 6'h28;
parameter 0_xxx = 6'h29; // XXX
parameter 0_yyy = 6'h2a; // YYY

// Control Signal Value Names
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_sne = 6'd9;
parameter OP_xxx = 6'd10; // XXX

parameter ST_if = 1; // Instruction Fetch
parameter ST_id = 2; // Instruction Decode
parameter ST_ex = 3; // Executing Arithmetic or Logical Instruction
parameter ST_eb = 4; // Executing Branch (Computing condition.)
parameter ST_ea = 5; // Executing Memory (Computing address.)
parameter ST_me = 6; // Performing load or store.
parameter ST_yy = 7; // YYY
parameter ST_s1 = 8; // Prob 1
parameter ST_s2 = 9; // Prob 1

parameter R0 = 5'd0; // Neat constant for register 0's register number.

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

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

reg [1:0] me_size;
```
Hardwired Control MIPS Implementation

```verilog
reg me_we, me_se;

reg [31:0] ma, md; // YYY
assign data_out = state == ST_yy ? alu_out : rt_val; // YYY

always @( state or pc or alu_out or me_we or me_size or ma )
case( state )
  ST_if: begin addr = pc; size = 3; we = 0; end
  ST_me: begin addr = alu_out; size = me_size; we = me_we; end
  ST_yy: begin addr = ma; size = me_size; we = 1; end // YYY
  default: begin addr = pc; size = 0; we = 0; end
endcase

assign immed = ir[15:0];

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

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

always @( posedge clk )
  if( reset ) begin
    state = ST_if;
    pc = 'h400000;
    npc = pc + 4;
    exc = 0;
  end else
  case( state )
    /// IF: Instruction Fetch
    ST_if:
      begin
        ir = data_in;
        state = ST_id;
      end
    /// Instruction Decode
    ST_id:
      begin
        rs_val = gpr[rs];
        rt_val = gpr[rt];
        sa_val = { 27'd0, sa };  
case( opcode )
      O_tyr:
        case( func )
          F_add: bndl = {rd, rs_val, OP_add, rt_val };
          F_swp: bndl = {rs, rs_val, OP_add, rt_val }; // Prob 1
```
Hardwired Control MIPS Implementation

\[
\begin{align*}
F_{\text{sub}}: & \quad \text{bndl} = \{\text{rd}, \text{rs\_val}, \text{OP}\_\text{sub}, \text{rt\_val}\}; \\
F_{\text{sll}}: & \quad \text{bndl} = \{\text{rd}, \text{sa\_val}, \text{OP}\_\text{sll}, \text{rt\_val}\}; \\
F_{\text{jr}}: & \quad \text{bndl} = \{\text{R0}, \text{rs\_val}, \text{OP}\_\text{add}, \text{rt\_val}\}; \quad \text{// Dummy} \\
\text{default}: & \quad \text{exc} = 1; \\
endcase
\end{align*}
\]

\[
\begin{align*}
0_{\text{addi}}: & \quad \text{bndl} = \{\text{rt}, \text{rs\_val}, \text{OP}\_\text{add}, \text{simmed}\}; \\
0_{\text{andi}}: & \quad \text{bndl} = \{\text{rt}, \text{rs\_val}, \text{OP}\_\text{and}, \text{uimmed}\}; \\
0_{\text{xxx}}: & \quad \text{bndl} = \{\text{rt}, \text{rs\_val}, \text{OP}\_\text{xxx}, \text{simmed}\}; \quad \text{// XXX} \\
0_{\text{yyy}}: & \quad \text{bndl} = \{\text{R0}, \text{rs\_val}, \text{OP}\_\text{add}, \text{simmed}\}; \quad \text{// YYY} \\
0_{\text{ori}}: & \quad \text{bndl} = \{\text{rt}, \text{rs\_val}, \text{OP}\_\text{or}, \text{uimmed}\}; \\
0_{\text{lui}}: & \quad \text{bndl} = \{\text{rt}, \text{rs\_val}, \text{OP}\_\text{or}, \text{limmed}\}; \\
0_{\text{bne}}: & \quad \text{bndl} = \{\text{R0}, \text{rs\_val}, \text{OP}\_\text{sne}, \text{rt\_val}\}; \\
0_{\text{beq}}: & \quad \text{bndl} = \{\text{R0}, \text{rs\_val}, \text{OP}\_\text{seq}, \text{rt\_val}\}; \\
0_{\text{lb}}, 0_{\text{lbu}}, 0_{\text{lw}}: & \quad \text{bndl} = \{\text{rt}, \text{rs\_val}, \text{OP}\_\text{add}, \text{simmed}\}; \\
0_{\text{jal}}, 0_{\text{j}}: & \quad \text{bndl} = \{\text{R0}, \text{rs\_val}, \text{OP}\_\text{sll}, \text{simmed}\}; \quad \text{// Dummy} \\
0_{\text{sb}}: & \quad \text{bndl} = \{\text{R0}, \text{rs\_val}, \text{OP}\_\text{add}, \text{simmed}\}; \\
\text{default}: & \quad \text{exc} = 1; \\
endcase
\end{align*}
\]

\[
\{\text{dst}, \text{alu\_a}, \text{alu\_op}, \text{alu\_b}\} = \text{bndl};
\]

\[
\begin{align*}
\text{case( opcode )} \\
0_{\text{lb}}: & \quad \text{begin me\_size} = 1; \text{ me\_se} = 1; \text{ me\_we} = 0; \text{ end} \\
0_{\text{lbu}}: & \quad \text{begin me\_size} = 1; \text{ me\_se} = 0; \text{ me\_we} = 0; \text{ end} \\
0_{\text{sb}}: & \quad \text{begin me\_size} = 1; \text{ me\_se} = 0; \text{ me\_we} = 1; \text{ end} \\
0_{\text{lw}}: & \quad \text{begin me\_size} = 3; \text{ me\_se} = 0; \text{ me\_we} = 0; \text{ end} \\
0_{\text{yyy}}: & \quad \text{begin me\_size} = 2; \text{ me\_se} = 0; \text{ me\_we} = 0; \text{ end} \quad \text{// YYY} \\
\text{default}: & \quad \text{begin me\_size} = 0; \text{ me\_se} = 0; \text{ me\_we} = 0; \text{ end} \quad \text{endcase}
\end{align*}
\]

\[
\begin{align*}
\text{case( opcode )} \\
0_{\text{yyy}}: & \quad \text{state\_after\_me} = \text{ST}_{\text{yy}}; \quad \text{// YYY} \\
\text{default}: & \quad \text{state\_after\_me} = \text{ST}_{\text{if}}; \quad \text{endcase}
\end{align*}
\]

\[
\begin{align*}
\text{case( opcode )} \\
0_{\text{tyr}}: & \quad \text{\text{case( func )}} \\
& \quad \text{F_{jr}}: \quad \text{state} = \text{ST}_{\text{if}}; \\
& \quad \text{F_{swp}}: \quad \text{state} = \text{ST}_{\text{s1}}; \quad \text{// Prob 1} \\
& \quad \text{default}: \quad \text{state} = \text{ST}_{\text{ex}}; \quad \text{endcase}
\end{align*}
\]

\[
\begin{align*}
0_{\text{beq}}, 0_{\text{bne}}: & \quad \text{state} = \text{ST}_{\text{eb}}; \\
0_{\text{sb}}, 0_{\text{lb}}, 0_{\text{lbu}}, 0_{\text{lw}}, 0_{\text{yyy}}: & \quad \text{state} = \text{ST}_{\text{ea}}; \\
0_{\text{j}}, 0_{\text{jal}}: & \quad \text{state} = \text{ST}_{\text{if}}; \\
\text{default}: & \quad \text{state} = \text{ST}_{\text{ex}}; \quad \text{endcase}
\end{align*}
\]
Hardwired Control MIPS Implementation

npc_plus_4 = npc + 4;

\[
\begin{align*}
\text{case( opcode )} \\
\text{\hspace{1em} 0\_tyr :} \\
\text{\hspace{2em} case( func )} \\
\text{\hspace{3em} F\_jr : nnpc = rs\_val;} \\
\text{\hspace{3em} default : nnpc = npc\_plus_4;} \\
\text{endcase} \\
\text{0\_j : nnpc = \{npc[31:28],ii,2'd0\};} \\
\text{0\_jal : \{gpr[31],nnpc\} = \{npc\_plus_4,npc[31:28],ii,2'd0\};} \\
\text{default : nnpc = npc\_plus_4;} \\
\text{endcase}
\end{align*}
\]

bimmed = npc + ( simmed << 2);

pc = npc;
npc = nnpc;

end

ST_s1: // Prob 1
begin
\text{if( rs ) gpr[rs] = rt\_val;} // Prob 1
state = ST_s2; // Prob 1
end // Prob 1

ST_s2: // Prob 1
begin
\text{if( rt ) gpr[rt] = rs\_val;} // Prob 1
state = ST_if; // Prob 1
end // Prob 1

// EX: Execute
ST_ex:
begin
\text{if( dst ) gpr[dst] = alu\_out;} \\
state = ST_if;
end

// EXB: Execute Branch
ST_eb:
begin
\text{if( alu\_out[0] ) npc = bimmed;} \\
state = ST_if;
end

// EA: Effective Address
ST_ea:
begin
\text{state = ST_me;} \\
ma = alu\_out; // YYY
end
Hardwired Control MIPS Implementation

```verilog
// ME: Memory
ST_me:
begin
    case( size )
        1: md = { me_se & data_in[7] ? 24'hffffff : 24'h0, data_in[7:0] };
        2: md = { me_se & data_in[15] ? 16'hffff : 16'h0, data_in[15:0] };
        3: md = data_in;
    endcase
    if( dst != 0 ) gpr[dst] = md;
    alu_a = md; alu_b = rt_val; // YYY
    state = state_after_me; // YYY
end

// YYY
ST_yy:
begin
    state = ST_if;
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 = 6'd0;
    parameter OP_sll = 6'd1;
    parameter OP_srl = 6'd2;
    parameter OP_add = 6'd3;
    parameter OP_or = 6'd5;
    parameter OP_and = 6'd6;
    parameter OP_sub = 6'd4;
    parameter OP_seq = 6'd8;
    parameter OP_sne = 6'd9;
    parameter OP_xxx = 6'd10; // XXX

    always @( alu_a or alu_b or alu_op )
        case( alu_op )
            OP_xxx : alu_out = -alu_a + alu_b; // XXX
            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;
        endcase
endmodule
```
Hardwired Control MIPS Implementation

```verilog
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_sne : alu_out = alu_a != alu_b;
OP_nop : alu_out = 0;
default : begin alu_out = 0; $stop; end
endcase
endmodule
```