Computer Organization
EE 3755
Practice Final Examination
5 December 2001

Problem 1  ________  (15 pts)
Problem 2  ________  (20 pts)
Problem 3  ________  (20 pts)
Problem 4  ________  (15 pts)
Problem 5  ________  (15 pts)
Problem 6  ________  (15 pts)

Alias  ___________________________  Exam Total  ________  (100 pts)

Good Luck!
Problem 1: Add a rotate left (\texttt{rol}) instruction to the MIPS implementation at the end of this exam. This instruction is similar to shift left except that bits that are shifted off one end are placed on the other end; see the example below.

\begin{verbatim}
# $2 = 0x12345678
rol $1, $2, 4
# $1 <- 0x23456781
\end{verbatim}

(a) Show the coding for the \texttt{rol} used in the example above. Indicate what each field is used for and its value for the \texttt{rol} above. The coding can be shown as a collection of fields, there is no need to put them together. Make up values for the opcode and any other needed fields. \textit{Code the instruction to minimize the amount of added hardware. Look closely at how similar instructions are coded and implemented.}

(b) Add the instruction to the MIPS implementation at the end of this exam.
Problem 2: The incomplete atoi procedure below returns, in $v0, the integer corresponding to the ASCII string starting at the address in $a0. Complete the routine. Avoid using a multiply instruction, instead use an efficient alternative. Computing $a \times b$ by computing the additions $a + a + \cdots + a$ ($b$ times) is not efficient if $b > 3$. See the comments for additional details.

.data
SEVEN_HUNDRED_THIRTEEN:
    .asciiz "713"

.text
.globl __start
__start:
    la $a0, SEVEN_HUNDRED_THIRTEEN
    jal atoi
    nop
    # $v0 should contain 713 when execution reaches this point.
    ...

atoi:
    # $a0: The address of an ASCII null-terminated string.
    # The string contains only digits, no whitespace, no - or +.
    # Number in string is in decimal.
    # This routine is allowed to modify $a0.
    # $v0: Used for return value, the value of the string.
    #
    # The ASCII code for '0' is 48.
    # Do not use pseudo instructions, except for nop.
    # A correct solution contains 11 instructions.
    # Do not use any multiply instruction, use an *efficient* substitute.

    jr $ra
    nop
Problem 3: The MIPS implementation at the end of this exam can execute two brand new instructions, xxx and yyy. So that you can find them quickly, the new lines added for these instructions has XXX or YYY comments in the right margins.

(a) Show a possible assembly language syntax for xxx with a brief description of what it does.

(b) 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.

(c) Show a possible assembly language syntax for yyy with a brief description of what it does.

(d) Show an example of how yyy is used, then show how the same thing is done using non-fictional MIPS instructions.

(e) Which of these instructions is an unlikely candidate for a future MIPS instruction? Explain.
Problem 4: Show the hardware that generates the value of state in the MIPS implementation at the end of the exam that would be synthesized based on the rules given in class.

- In your drawing clearly indicate which boxes are registers.
- Show each multiplexor, do not combine them.
- Show the logic used to generate the multiplexor control signals.
- All non-constant signals must originate at a register output or a module input port.
Problem 5: The synthesized processor at the end of this test runs at a clock frequency of 1 GHz. On a particularly sunny day your fab informs you that their newest process runs twice as fast. Alas, the speed of the memory module has not changed.

(a) With no changes to the Verilog code how much faster would programs run on the processor synthesized for the new process? Explain.

(b) How would you modify the Verilog code to take advantage of the new process? Now how much faster would programs run?
Problem 6: Answer each question below.

(a) The slti instruction in the MIPS implementation at the end of this exam does not work. Fix it. *Hint: it does what sltiu should do.*

(b) What are the characteristics of an ISA that make it suitable for an implementation using microprogrammed control?

(c) Describe the three ways that the microprogram program counter is advanced in the microprogrammed control MIPS implementation described in class and in the text.

(d) The microprogram program counter is incremented by one while the MIPS program counter is incremented by four. Why is the amount different?
(e) Name three sections of an object file and explain their contents and how they are used.

(f) A new pseudo-instruction is available. (Pretend.) What needs to be replaced?

(g) Show two ways a branch can be used as a no-op instruction.
Here it is: The MIPS Implementation at the End of This Exam!
To see it in color visit [http://www.ece.lsu.edu/ee3755/2001f/fepp3.html](http://www.ece.lsu.edu/ee3755/2001f/fepp3.html).

```vhdl
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 [31:0] data_out, addr;
  output [31:0] exc;
  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, limmed, jimmed;
  reg [31:0] rs_val, rt_val;

  // 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_yyy = 6'h3f; // YYY
  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;

  // Values for the MIPS opcode field.
  //
  parameter o_xxx = 6'h3e; // XXX
  parameter o_rfmt = 6'h0;
  parameter o_j = 6'h2;
  parameter o_beq = 6'h4;
```

9
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;

// Processor Control Logic States
//
parameter st_yyy = 8; // YYY
parameter st_if = 1;
parameter st_id = 2;
parameter st_ex = 3;
parameter st_ex_addr = 5;
parameter st_ex_cond = 6;
parameter st_ex_targ = 7;
parameter st_me = 4;

// ALU Operations
//
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;

/// 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_yyy : begin addr = alu_out; we = me_we; size = me_size; end // YYY
  st_me : begin addr = alu_out; we = me_we; size = me_size; end
  default : begin addr = pc; we = 0; size = 0; end // Note: addr is set for default case to simplify synthesized hardware.
endcase

always @( posedge clk )
if( reset ) begin
  state = st_if;
  exc = 0;
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;
```java
ii = ir[25:0];
immed = ir[15:0];

simmed = { immed[15] ? 16'hffff : 16'h0, immed };
uimmed = { 16'h0, immed }; 
jimmed = { 6'h0, ii }; // XXX
limmed = { immed, 16'h0 }; 

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

// Set alu_a, alu_b, alu_op, and wb_rd. 

// case( opcode )

o_rfmt:
  // R-Format Instructions
  case ( func )
  f_yyy : begin alu_a = sa; alu_op = op_add; // YYY
         alu_b = rt_val; wb_rd = rt; end
  f_add : begin alu_a = rs_val; alu_op = op_add;
         alu_b = rt_val; wb_rd = rd; end
  f_sub : begin alu_a = rs_val; alu_op = op_sub;
         alu_b = rt_val; wb_rd = rd; end
  f_sll : begin alu_a = sa; alu_op = op_sll;
         alu_b = rt_val; wb_rd = rd; end
  default : begin alu_a = rs_val; alu_op = op_nop;
            alu_b = rt_val; wb_rd = 0; exc = 1; end
endcase

// I- and J-Format Instructions
o_xxx: begin alu_a = 0; alu_op = op_add; // XXX
       alu_b = jimmed; wb_rd = 25; end
o_lbu: begin alu_a = rs_val; alu_op = op_add;
       alu_b = simmed; wb_rd = rt; end
o_snb: begin alu_a = rs_val; alu_op = op_add;
       alu_b = limmed; wb_rd = 0; end
o_lui: begin alu_a = rs_val; alu_op = op_or;
       alu_b = limmed; wb_rd = rt; end
o_addi: begin alu_a = rs_val; alu_op = op_add;
        alu_b = simmed; wb_rd = rt; end
o_andi: begin alu_a = rs_val; alu_op = op_and;
        alu_b = uimmed; wb_rd = rt; end
o_ori: begin alu_a = rs_val; alu_op = op_or;
       alu_b = uimmed; wb_rd = rt; end
o_slti: begin alu_a = rs_val; alu_op = op_slt;
        alu_b = simmed; wb_rd = rt; end
o_j: begin alu_a = rs_val; alu_op = op_nop;
    alu_b = simmed; wb_rd = 0; end
o_bne, o_beq:
begin alu_a = rs_val; alu_op = op_seq;
    alu_b = rt_val; wb_rd = 0; end
default:begin alu_a = rs_val; alu_op = op_nop;
        alu_b = simmed; wb_rd = 0; exc = 1; end
endcase

// Needed for a store instruction, doesn’t hurt others.
data_out = rt_val;

// Set me_size and me_wb

// case( opcode )

o_lbu : begin me_size = 1; me_we = 0; end
o_snb : begin me_size = 1; me_we = 1; end
o_rfmt : case ( func )
  f_yyy : begin me_size = 3; me_we = 0; end // YYY
  default : begin me_size = 0; me_we = 0; end
```
endcase
    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 )
    o_lbu, o_sb : state = st_ex_addr;
    o_bne, o_beq : state = st_ex_cond;
    o_j : state = st_if;
    o_rfmt : case ( func )
        f_yyy : state = st_yyy; // YYY
        default : state = st_ex;
    endcase
    default : state = st_ex;
endcase

end

/// Instruction YYY
st_yyy:
begin
    if( wb_rd ) gpr[wb_rd] = alu_out;
    alu_a = rs_val;
    alu_b = data_in;
    wb_rd = rd;
    state = st_ex; // Yes, st_ex.
end

/// 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[0] ) 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
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;
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 < 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_nop : alu_out = 0;
default : begin alu_out = 0; $stop; end
endcase
endmodule