Computer Organization
EE 3755
Practice Final Examination
9 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)

Exam Total (100 pts)

Good Luck!
Problem 1: Add a rotate left (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.

# $2 = 0x12345678
rol $1, $2, 4
# $1 <- 0x23456781

(a) Show the coding for the rol used in the example above. Indicate what each field is used for and its value for the 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. Code the instruction to minimize the amount of added hardware. Look closely at how similar instructions are coded and implemented.

Code rol similarly to sll since both perform shifts and have a small immediate operand. Use func value 26_{16} since it's not being by other instructions. The complete coding is shown below:

<table>
<thead>
<tr>
<th>Coding:</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>15</td>
<td>11</td>
<td>10</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x26</td>
</tr>
</tbody>
</table>

(b) Add the instruction to the MIPS implementation at the end of this exam.

The changes are at the end of the exam. The line changed in the ALU is repeated below.

```c
op_rol : alu_out = ( alu_b << alu_a ) | ( alu_b >> ( 32 - alu_a ) );
```
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.

The solution appears below. For a runnable (and color) version visit http://www.ece.lsu.edu/ee3755/2001f/fepp2.html.

.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.
  # Solution
  j START
  add $v0, $0, $0

  LOOP:
    addi $t1, $t1, -48  # '0' = 48
    sll $t2, $v0, 3
    sll $t3, $v0, 1
    add $v0, $t2, $t3
    add $v0, $v0, $t1

  START:
    lb $t1, 0($a0)
    bne $t1, $0, LOOP
    addi $a0, $a0, 1

    jr $ra
    nop
Problem 3: The MIPS implementation at the end of this exam can execute two brand new instructions, \texttt{xxx} and \texttt{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 \texttt{xxx} with a brief description of what it does.

Assembler Syntax: XXX \texttt{ummed26}, where \texttt{ummed26} is a 26-bit unsigned immediate. Instruction loads register \texttt{t9} with the immediate.

(b) Show an example of how \texttt{xxx} is used in an assembly language program, then show how the same thing is done using non-fictional MIPS instructions.

\begin{verbatim}
# Using xxx
xxx 0x123456

# Using existing instructions.
lui $t9, 0x12
ori $t9, $t9, 0x3456
\end{verbatim}

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

Assembler Syntax: YYY \texttt{rd, rs, (rt)+sa}, load register \texttt{rd} with \texttt{rs} plus the word at address \texttt{rt}, then increment \texttt{rt} by \texttt{sa}.

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

\begin{verbatim}
# Using yyy
yyy $s4, $s3, ($t0)+4

# Existing instructions.
lw $at, 0($t0)
addi $t0, $t0, 4
add $s4, $s3, $at
\end{verbatim}

(e) Which of these instructions is an unlikely candidate for a future MIPS instruction? Explain.

Instruction \texttt{yyy} is the less likely MIPS candidate because it performs arithmetic \textit{and} it accesses memory.
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.

Solution:

```
31:26 (opcode)
5:0 (func)

YY
EX
default

IF
ID
YY
EX
default

EX_ADDR
EX_COND
IF
rfmt
default

alu_out(0)
EX_TARG
IF
0
ME
EX_ADDR
EX_COND

reset
(input)
State

IR

default

alu_out
EX
bne, beq
j

EX_ADDR
EX_COND

alu_out[0]
IF
0
ME
EX_ADDR
EX_COND

alu_out[0]
EX_TARG
IF
1
ME
EX_ADDR
EX_COND

alu_out
EX_TARG
IF
0
ME
EX_ADDR
EX_COND
```


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\_3 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.

It would go no faster because the speed of memory hasn’t changed. If the clock frequency were increased then memory would not be able to produce a result in time.

(b) How would you modify the Verilog code to take advantage of the new process? Now how much faster would programs run?

Replace each state that access memory with a pair of states, in particular replace \texttt{st\_if} with \texttt{st\_if\_1} and \texttt{st\_if\_2} and replace \texttt{st\_me} with \texttt{st\_me\_1} and \texttt{st\_me\_2}. The memory access starts in the first state and the result is clocked into a register at the end of the second state.

The programs would be faster, but not twice as fast as the old version. The speedup depends on the mix of instructions, including the percentage of memory accesses.
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.*

Instruction `slti` is supposed to do a signed comparison, but the `alu` module uses the Verilog comparison operator, `<`, on a register which is done using unsigned arithmetic. One solution is to subtract sign-extended versions of the operands and look at the most significant bit.

```plaintext
# Replace
op_slt : alu_out = alu_a < alu_b;
# With
op_slt : alu_out = ( {alu_a[31],alu_a} - {alu_b[31],alu_b} ) >> 32;
```

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

Complicated instructions. To implement such instructions using hardwired control would require a lot of logic, a microprogrammed controller in such a case would be simpler.

(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.

It's incremented, set to 1 (instruction fetch), or set to a value provided by the dispatch table.

(d) The microprogram program counter is incremented by one while the MIPS program counter is incremented by four. Why is the amount different?

Because the MIPS program counter is a memory address and memory is addressed by characters, four of which make up an instruction. The microprogram counter is an index into the micro ROM, which is used only for microinstructions, and so each microaddress can conveniently be made to store one microinstruction.
(e) Name three sections of an object file and explain their contents and how they are used.

Text: Contains the encoded instructions. A linker would assemble text segments from multiple files into a larger one. The larger text segment is loaded into memory before the program is run.

Data: Contains data constants. These are also assembled by the linker into a larger data segment and ultimately loaded into memory.

Symbol Table: A list of symbols defined in the assembler file. There are many types of symbols. Some symbols are globally visible procedure names, the linker might resolve a symbol used for a procedure call (jal FOO) in one file with the procedure itself in another file.

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

The assembler. The assembler would replace the new pseudo-instruction with existing machine instructions. Nothing else needs to be replaced, in particular a new processor is not needed.

(g) Show two ways a branch can be used as a no-op instruction.

# Solution
bne $0, $0, ANYWHERE

beq $0, $0, 1  # The 1 is the branch displacement.
Here it is: The MIPS Implementation at the End of This Exam!
To see the unsolved version visit [http://www.ece.lsu.edu/ee3755/2001f/fepp3.html](http://www.ece.lsu.edu/ee3755/2001f/fepp3.html). To see the solution version visit [http://www.ece.lsu.edu/ee3755/2001f/fepv.html](http://www.ece.lsu.edu/ee3755/2001f/fepv.html).

```verilog
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, 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;
  parameter f_rol = 6'h26; // Prob 1
  parameter f_slt = 6'h2a;

  // Values for the MIPS opcode field.
  //
  parameter o_xxx = 6'h3e; // XXX
```

9
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;

// 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;
parameter op rol = 9; // Prob 1

/// 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_b; 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)
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};
jimmed = {6'h0, ii};
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
    f_rol: begin
      alu_a = sa;
      alu_op = op_rol; // Prob 1
      alu_b = rt_val;
      wb_rd = rd;
    end
    f_slt: begin
      alu_a = rs_val;
      alu_op = op_slt;
      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_sb: begin
  alu_a = rs_val;
  alu_op = op_add;
  alu_b = simmed;
  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_sb : 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: // YYY
begin
  if( wb_rd ) gpr[wb_rd] = alu_out;
  alu_a = rs_val;
  alu_b = data_in;
  wb_rd = rd;
  state = 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;
parameter op_rol = 9;

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_slt : alu_out = alu_a < alu_b;
    // op_sll : alu_out = ( {alu_a[31],alu_a} - {alu_b[31],alu_b} ) >> 32;
    op_sll : alu_out = alu_a << alu_a;
    op_srl : alu_out = alu_b >> alu_a;
    op_rol : alu_out = ( alu_b << alu_a ) | ( alu_b >> ( 32 - alu_a ) );
    op_seq : alu_out = alu_a == alu_b;
    op_nop : alu_out = 0;
    default : begin alu_out = 0; $stop; end
endcase
endmodule