Name ____

Computer Organization
EE 3755
Final Examination

Monday, 9 December 2013, 10:00–12:00 CST

Problem 1	__________	(16 pts)
Problem 2	__________	(16 pts)
Problem 3	__________	(16 pts)
Problem 4	__________	(16 pts)
Problem 5	__________	(20 pts)
Problem 6	__________	(16 pts)
Exam Total	__________	(100 pts)

Alias ____

Good Luck!

Problem 1: [16 pts] Write MIPS code corresponding to the following fragments of C code. Assume that C variables have the same name as the MIPS registers to which they were assigned. That is, in the first example, the C variable `r2` has been assigned to MIPS register `r2`. The C compiler used for the code below uses 4-byte `int`s, 2-byte `short`s, and 4-byte pointers.

☐ Efficient MIPS code for corresponding C:

```
int r1, r2, r3;
r1 = r2 + r3 - 4;
```

☐ Efficient MIPS code for corresponding C:

```
int r1, r2, r3;
r1 = r2 & 0xabcdef01;
r3 = r2 && r1;
```

☐ Efficient MIPS code for corresponding C:

```
int r1, r2;
r1 = r2 > 21;
```

☐ Efficient MIPS code for corresponding C:

```
int r1, r3;
int* r2; // sizeof(int) = 4
r1 = r2[r3] + r2[r3+10];
r2[0] = r1;
```

Problem 2: [16 pts] The MIPS procedure below returns an element from a table of word-sized (4 byte) integers. The total size of the table in bytes is 4× the number of elements.

(*a*) Modify the routine below so that it will return a -1 if the requested element number is out of range, meaning that it is less than 0 or ≥ the value in `a2`.

☐ Modify code above so -1 is returned when `a1` is out of range.

(*b*) Also modify the routine so that it operates on a table of 3-byte values. Yes, 3-byte values. The size of the table now will be 3× the number of elements. (Don't modify the table itself.)

☐ Modify for table of 3-byte values.

☐ Don't use a 3-byte load instruction (which MIPS lacks anyway).

☐ Don't forget about alignment restrictions.

☐ Be efficient.

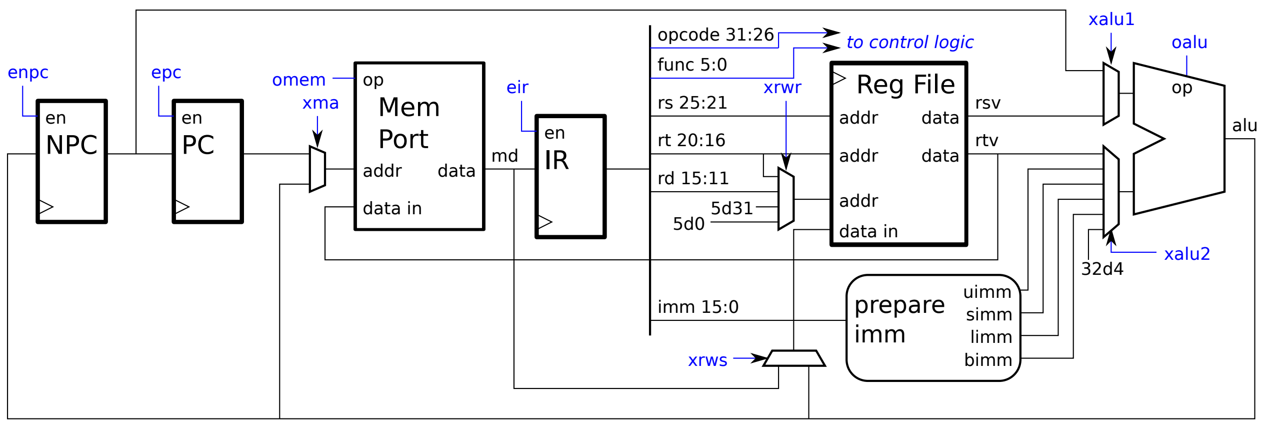
```
# Call Arguments
#   $a0: Address of start of table.
#   $a1: Element number to retrieve.
#   $a2: Number of elements.
# Return Argument
#   $v0: Element from table.

sll $t1, $a1, 2
add $t0, $a0, $t1
jr $ra
lw $v0, 0($t0)
```

Problem 3: [16 pts] A Verilog description of a MIPS implementation similar to the Very Simple MIPS is attached to this exam. The attached implementation takes a different approach to implementing the shift instructions. The relevant lines have a comment with `SSS` to the right.

(*a*) Appearing below is a diagram of the Very Simple MIPS datapath hardware which omits the shift hardware. Add the shift hardware to this diagram based on the Verilog code.

☐ Show the shift instructions' datapath hardware based on Verilog.

☐ Show shift control signal names using names in Verilog.



(*b*) Appearing below are the control signal settings needed to implement the `add` instruction for the VS MIPS presented in class. Show a similar table for the `sll` (shift left logical using `sa` for shift amount) instruction and `srlv` (shift right logical using `rs` register for shift amount) instructions. Insert columns as needed. No need to copy `IF` and any other unchanged states.

State	SubSt	xma	omem	eir	xalu1	xalu2	oalu	xrwr	xrws	enpc	epc	Next
IF		pc	rd32	1								ID
ID	add				rsv	rtv	add	rd	alu			NI
NI					npc	4	add			1	1	IF

☐ Add columns for new control signals.

☐ Control signal settings for ☐ `sll` and for ☐ `srlv`.

State	SubSt	xma	omem	eir	xalu1	xalu2	oalu	xrwr	xrws	enpc	epc	Next

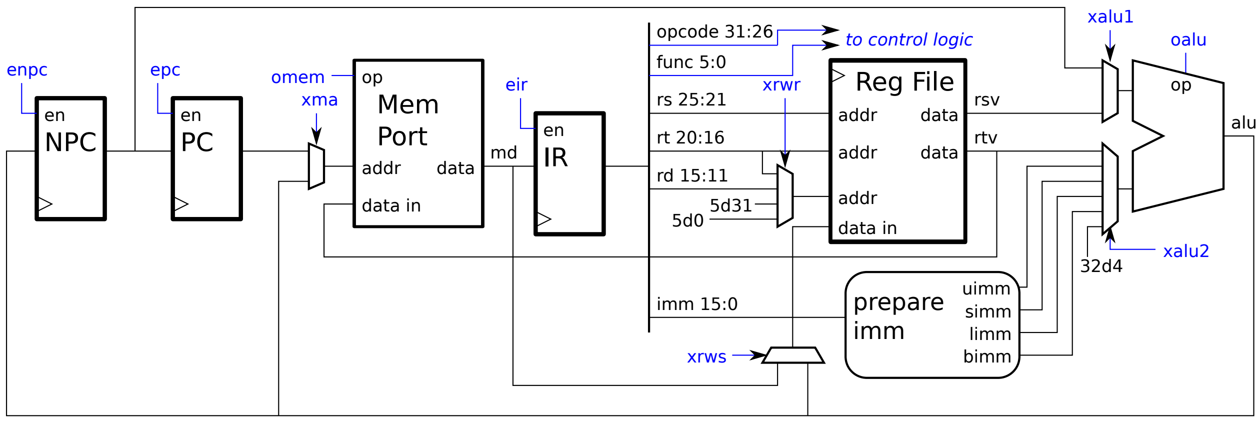
Problem 4: [16 pts] A *counted branch* instruction, `beq.cnt rs, rt, TARG`, works like a `beq` (branch equal) instruction, except that if the branch is taken it subtracts 1 from the `rs` register (and writes the decremented value back into the register file). Note that it is the `rs` register that gets modified.

(*a*) Modify the MIPS implementation at the end of this exam to implement the counted branch. For your convenience the Verilog already has an opcode for this branch, `O_beqcnt`, and in case you need it, a state constant, `ST_beqcnt`.

☐ Modify Verilog to implement counted branch.

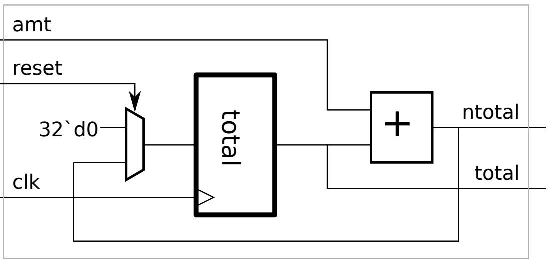
(*b*) In the diagram below sketch in any datapath changes that were made. *Hint: They should be minor.*

☐ Show datapath changes.



Problem 5: [20 pts] Answer each Verilog question below.

(*a*) Write a Verilog module corresponding to the hardware illustrated on the right.



☐ Write a Verilog description of the module to the right.

☐ Choose `reg` and `wire` sizes based on whatever information is given the diagram.

Problem 5, continued:

(*b*) Sketch the unoptimized hardware that would by inferred by a synthesis program for the module below. As we have done in class, assume that the synthesis program will infer to common units such as adders, multiplexers, etc.

```
module show_syn2(w, x,  s, u,  a, b, clk);
   input wire [7:0] a, b;    input wire [1:0] s;    input wire u, clk;
   output reg [7:0] w, x;
   reg [7:0] y, z;

   always @ ( posedge clk ) begin
      z = a + b;
      if ( u == 1 ) begin
         x = z;
         y = a - b;
      end else begin
         x = a + 8'd20;
      end
      case ( s )
        0: w = z;
        1: w = x;
        2: w = y;
        3: w = x + y;
      endcase
      if ( b < 8'd30 ) x = x + 8'd1;
   end
endmodule
```

☐ Show hardware for the module above. ☐ Show module inputs and outputs.

☐ Be careful about what gets synthesized as a register.

Problem 6: [16 pts] Answer each question below.

(*a*) What is the approximate cost of an n-bit ripple adder? Specify the cost model that you are using. (Don't give the name of the model, briefly explain what it is.)

☐ Cost of n-bit ripple adder.

☐ Briefly describe cost model.

(*b*) What is the delay of an n-bit ripple adder?

☐ Delay of an n-bit ripple adder.

(*c*) When we evaluated the delay of the carry lookahead adder we used both an *unrealistic* and *conservative* model. Why did we need to consider both models for evaluating a carry lookahead adder while only one model was adequate for a ripple adder?

☐ Briefly explain *unrealistic model* and ☐ briefly explain *conservative model*.

☐ Why are both models needed to get a good picture of carry lookahead delay?

☐ Why is only one model sufficient for evaluating ripple adder delay?

(*d*) Fill in the values for `x` below, any radix will do.

☐ Fill in blanks for `x` below.

```
module misc();
   reg [15:0] x;
   reg [3:0] a, b, c;
   initial begin

      a = 4'hd;
      x = { 2'd3, 6'o60, a };       // x =

      a = 4'b1010;
      b = 4'b0001;

      x = a | b;                   // x =

      x = a || b;                 // x =

      x = a < b ? -1 : 1;           // x =

   end

endmodule
```

(*e*) When performing floating-point addition one of the operands needs to be scaled. Show this with an example for the addition of numbers $1.111_2 \times 2^{10} + 1.001_2 \times 2^{12}$.

☐ Show how numbers above are added emphasizing scaling.

EE 3755 Fall 2013 Final Exam Appendix—Very Simple MIPS Implementation

```
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 func field.
   parameter   F_sll = 6'h0,  F_srl = 6'h2,  F_sra = 6'h3,  F_sllv = 6'h4;
   parameter   F_srlv = 6'h6, F_add = 6'h20, F_sub = 6'h22, F_or  = 6'h25;
   parameter   F_swap = 6'h26;

   // Values for the MIPS opcode field.
   parameter   O_rfmt = 6'h0, O_j    = 6'h2,  O_beq  = 6'h4,  O_bne  = 6'h5;
   parameter   O_addi = 6'h8, O_slti = 6'ha,  O_andi = 6'hc,  O_ori  = 6'hd;
   parameter   O_lui  = 6'hf, O_lw   = 6'h23, O_lbu  = 6'h24, O_sw   = 6'h2b;
   parameter   O_sb   = 6'h28, O_beqcnt = 6'h3f;

   // Processor Control-Logic States
   parameter   ST_if = 1, ST_id = 2, ST_ni = 3, ST_bt = 4, ST_jt = 5, ST_beqcnt = 6;

   // ALU Operations
   parameter   OP_nop = 6'd0, OP_add = 6'd3, OP_sub = 6'd4;
   parameter   OP_or = 6'd5,  OP_and = 6'd6, OP_slt = 6'd7, OP_seq = 6'd8;
   parameter   OP_jp = 6'd9;

   // Control Settings for xalu1: ALU Upper Input Mux
   parameter SRC_rs = 2'd0, SRC_npc = 2'd2;

   // Control Settings for xalu2: ALU Lower Input Mux
   parameter SRC_rt = 3'd0, SRC_ui = 3'd1, SRC_si = 3'd2, SRC_li = 3'd3;
   parameter SRC_bi = 3'd4, SRC_ji = 3'd5, SRC_4 = 3'd6;

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

   // Control Settings for xrws: Register Writeback Source
   parameter RWS_mdo  = 2'd0;
   parameter RWS_alu = 2'd1;
   parameter RWS_su  = 2'd2;                                              // SSS

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

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

   /// 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;    reg [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;

// Memory Address Multiplexer Select Signal
reg         xma;

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

// ALU Multiplexers Select 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.

///
/// Shift Unit                                                      // SSS
///
// Values for xsu: shift amount multiplexer.                        // SSS
parameter  SAMT_rs = 1'd0;                                          // SSS
parameter  SAMT_sa = 1'd1;                                          // SSS
parameter  SUOP_left = 2'd0;                                        // SSS
parameter  SUOP_right_ar = 2'd1;                                    // SSS
parameter  SUOP_right_lo = 2'd2;                                    // SSS

reg        xsu;                                                     // SSS
reg [1:0]  osu;                                                     // SSS
wire [31:0] su_data_out;                                            // SSS
wire [4:0]  su_amt_in;                                              // SSS
assign      su_amt_in = xsu == SAMT_rs ? rs_val[4:0] : sa;          // SSS

shift_unit our_shift_unit(su_data_out, rt_val, su_amt_in, osu);     // SSS

///
/// 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 ( 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;

/// Extract IR Fields and Compute Some Values
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

// Register File Writeback Source Multiplexer
always @* case ( xrws )
           RWS_mdo: gpr_data_in = mem_data_out;
           RWS_alu: gpr_data_in = alu_out;
           RWS_su:  gpr_data_in = su_data_out;                       // SSS
           default: gpr_data_in = alu_out;
         endcase

///
/// ALU Connections
///
// Upper ALU Input Multiplexer
always @* case ( xalu1 )
           SRC_rs: alu_1 = rs_val;
           SRC_npc: alu_1 = npc;
           default: alu_1 = rs_val;
         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: alu_2 = bimm;
  endcase

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

```
/// Set Control Signals to Default Values

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

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

// Register File Control Signals
xrwr = DST_0;    // Where to get the register number from.
xrws = RWS_alu;  // Register Writeback Source

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

// Shift Unit Control Signals                                      // SSS
xsu = SAMT_sa;                                                     // SSS
osu = SUOP_left;                                                   // SSS

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
  ST_id:
    begin

       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, F_srl, F_sllv, F_srlv: // Note: ALU ignored    // SSS
                    bndl = {DST_rd, SRC_rs, OP_add, SRC_rt};       // SSS
             default:// Unrecognized instruction.
               begin bndl = {DST_rd, SRC_rs, OP_add, SRC_rt};
                  exc = 1; end
           endcase

         // I- and J-Format Instructions
         //             xrwr    xalu1   oalu    xalu2
         O_lbu:  bndl = {DST_rt, SRC_rs, OP_add, SRC_si };
         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;

      if ( opcode == O_rfmt )                                   // SSS
        case ( func )                                           // SSS
          F_sll: {xsu,osu} = { SAMT_sa,  SUOP_left     };         // SSS
          F_srl: {xsu,osu} = { SAMT_sa,  SUOP_right_lo };         // SSS
          F_sra: {xsu,osu} = { SAMT_sa,  SUOP_right_ar };         // SSS
          F_sllv:{xsu,osu} = { SAMT_rs,  SUOP_left     };         // SSS
          F_srlv:{xsu,osu} = { SAMT_rs,  SUOP_right_lo };         // SSS
          default:{xsu,osu}= { SAMT_sa,  SUOP_left     };         // SSS
        endcase                                                 // SSS

      // Set xrws: register write source multiplexer select signal.
      case ( opcode )
        O_rfmt:
          case ( func )
            F_srl, F_sra, F_sllv, F_srlv, F_sll: xrws = RWS_su;   // SSS
            default:                             xrws = RWS_alu;
          endcase
        O_lbu:                                   xrws = RWS_mdo;
        default:                                 xrws = RWS_alu;
      endcase

      // Determine values for omem_size and me_wb.
      case ( opcode )
        O_lbu   : begin omem_size = 1;  omem_wr = 0; end
        O_sb    : begin omem_size = 1;  omem_wr = 1; end
        default : begin omem_size = 0;  omem_wr = 0; end
      endcase

      xma = MA_alu;

      // Determine value for next_state.
      case ( opcode )
        O_bne  : next_state = alu_out[0] == 0 ? ST_bt : ST_ni;
        O_beq  : next_state = alu_out[0] == 0 ? ST_ni : ST_bt;
        O_j    : next_state = ST_jt;
        default: next_state = ST_ni;
      endcase
   end
   /// End of ID: Instruction Decode State (ID appears above)

 /// 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.
        default: next_state = ST_if;
      endcase
   end

endmodule
```