

Name Solution\_\_\_\_\_

# Computer Organization

EE 3755

## Final Examination

15 December 2001, 12:30-14:30 CST

Problem 1 \_\_\_\_\_ (20 pts)

Problem 2 \_\_\_\_\_ (17 pts)

Problem 3 \_\_\_\_\_ (20 pts)

Problem 4 \_\_\_\_\_ (17 pts)

Problem 5 \_\_\_\_\_ (26 pts)

Alias It's a Processor!\_\_\_\_\_

Exam Total \_\_\_\_\_ (100 pts)

*Good Luck!*

Problem 1: The `bcloop` instruction is a combination subtract-by-one and branch-if-not-zero instruction. It works as follows: if the register operand (`$t1` in the example below) is zero the branch is not taken and the register operand is not changed. If the register operand is non-zero it is decremented and the branch is taken. The target of `bcloop` (the place it branches to) is computed the same way as existing branch instructions and like other branches it has a delay slot.

Modify the first MIPS implementation in the appendix so that it can execute the `bcloop` instruction. (20 pts)

- Make up any field values that are needed.
- All arithmetic must be done by the ALU.
- As does the current branch code, only a single bit of `alu_out` can be examined by the control logic. (For example, `if( alu_out == 123 )...` is not allowed but `if( alu_out[10] == 0 )` is.)

```
    addi $t1, $0, 3
LOOP: # Loop iterates 4 times.
    lw $t2, 0($t3)
    addi $t3, $t3, 4
    bcloop $t1, LOOP
    add $t4, $t4, $t2
```

Changed lines shown in the appendix.

Problem 2: The incomplete `max` procedure below returns, in `$v0`, the largest integer in the `$a1`-element array of word-sized signed integers which starts at the address in `$a0`. Write the `max` procedure. (17 pts)

- Use `beq` and `bne` for conditional branches, do not use `blt` (branch less than), etc.
- Fill as many delay slots as possible.
- The only pseudo instruction allowed is `nop`.

*Hint: The following instructions may come in handy: `slt` (set less than) and `lw` (load word).*

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

```
max:
    # $a0: Call argument: address of the 1st element of an integer array.
    # $a1: Call argument: number of elements in the array.
    #       Number of elements always greater than one.
    # $v0: Return value: the largest element in the array.
    # $a0 and $a1 can be modified.

    beq $a1, $0, DONE
    nop
    lw $v0, 0($a0)
LOOP:
    lw $t0, 0($a0)
    slt $t1, $v0, $t0
    beq $t1, $0, SKIP
    addi $a1, $a1, -1
    add $v0, $t0, $0
SKIP:
    bne $a1, $0, LOOP
    addi $a0, $a0, 4
DONE:

    jr $ra
    nop
```

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

(a) What is `xxx`? (8 pts)

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

Possible syntax: `xxx rd, rs, rt`

Write register `rd` with the bitwise AND of the contents of `rs` and the bitwise NOT of the contents of `rt`.

This instruction comes in handy when masking out bits.

- 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 xxx
xxx $t0, $t1, $t2
```

```
# Using existing instructions.
nor $at, $t2, $0
and $t0, $t1, $at
```

(b) What is `yyy`? (12 pts)

- Show a possible assembly language syntax for `yyy` with a brief description of what it does.

Assembler Syntax: `YYY rs,rt`. Jump to address stored in location `Mem[rs+rt*2]`.

Note: The instruction would be useful for implementing C-style switch statements.

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

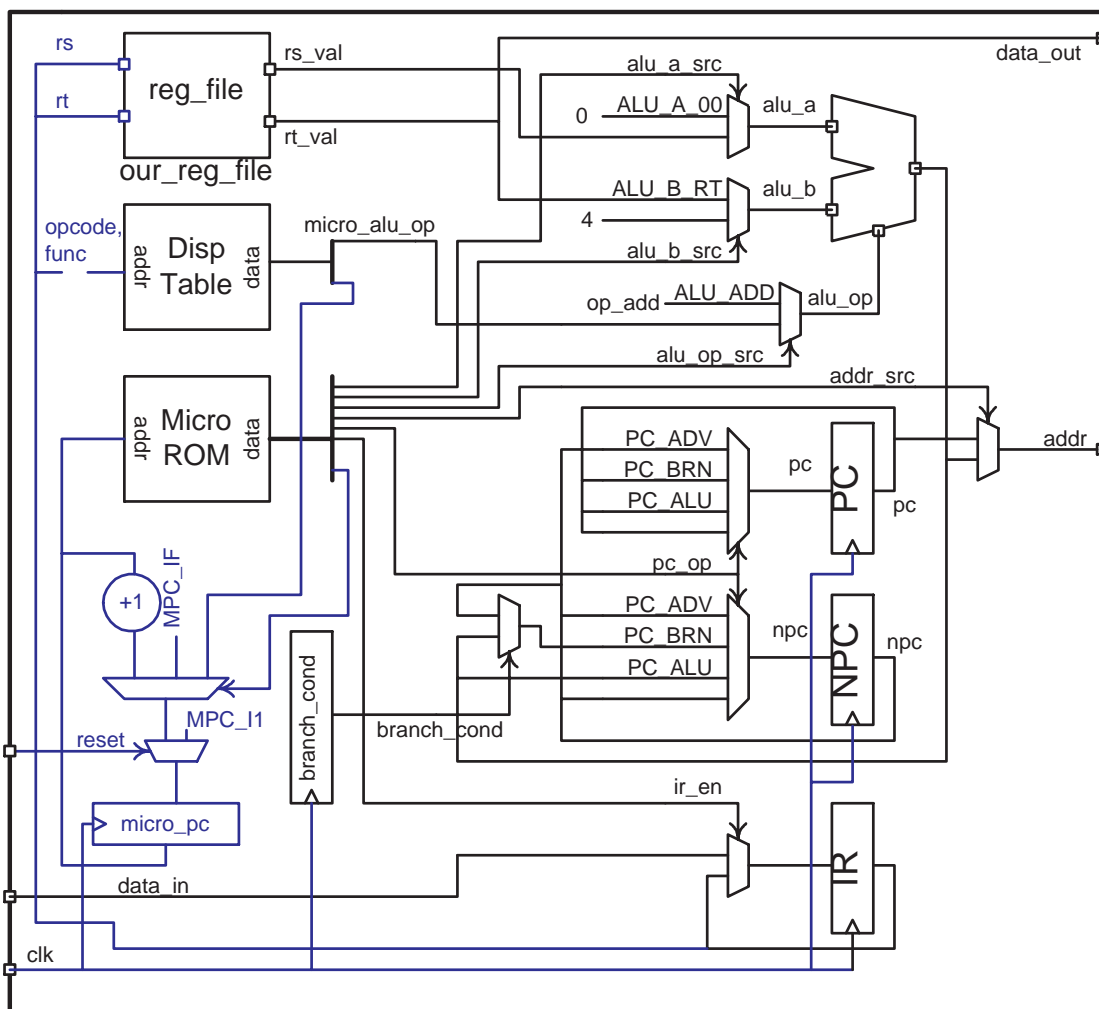
```
# Using yyy
yyy $t0, $t1
```

```
# Existing instructions.
sll $at, $t1, 2
add $at, $t0, $t1
lw $at, 0($at)
jr $at
nop
```

Problem 4: At the end of the appendix is part of the microcoded control MIPS implementation. Show the hardware that will be synthesized as described below. (17 pts)

- ✓ The code does not show the micro ROM or the dispatch table but certain signals used in the code connect to them. Show the micro ROM and dispatch table in your diagram below and show the connections to the hardware that are described in the code. (Some micro ROM and dispatch table connections will be unused.) *Hint: The micro ROM and dispatch table in the diagram should each have an address input and a data output.*
- ✓ Show the ALU and its connections.
- ✓ Clearly indicate all registers.
- ✓ Label all wires using symbols from the Verilog code.
- ✓ Show all multiplexers **and** the connections to their control inputs. Do not combine multiplexers.
- ✓ All non-constant signals must originate at a register output or a module connection or port.

Diagram appears below. The material in blue is not part of the solution but is included for completeness.



Problem 5: Answer each question below.

(a) The module below is complete except for the logic that computes overflow.

(5 pts) Complete it.

Use only basic gates or logical operators. **Do not** compute a 5-bit sum and compare it to the 4-bit sum, or something similar.

A correct answer is one line.

```
module adder(sum,overflow,a,b);
    input [3:0] a,b; // Two's complement signed integers.
    output [3:0] sum;
    output      overflow;

    assign      sum = a + b;
    assign      overflow = a[3] == b[3] && b[3] != sum[3];

endmodule
```

(b) Write the assembly language for the following MIPS instruction:  $90a90007_{16}$ . (8 pts) *Hint: The first MIPS implementation in the appendix should be helpful.*

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

It's `lbu t1, 7(a1)`.

(c) What is the value in decimal of the following IEEE 754 single-precision-encoded floating point number:  $40500000_{16}$ . (8 pts)(The bias for IEEE 754 singles is 127.)

It's 3.25.

(d) The code below is from the combinational floating-point adder with an important step between Step 1 and Step 7 removed. (5 pts)

Identify the missing step.

Step 3, un-normalize.

Briefly explain what the missing step is supposed to do.

In preparation for adding the significands, shift b's significand and adjust b's exponent so that a and b's exponent are the same.

Insert the missing code. (Two statements.)

See step 3 below.

```
/// Compute IEEE 754 Double Floating-Point Sum in Seven Easy Steps

/// Step 1: Copy inputs to a and b so that a's exponent not smaller than b's.
if( a_original[62:52] < b_original[62:52] ) begin
    a = b_original; b = a_original;
end else begin
    a = a_original; b = b_original;
end

/// Step ?: Break operand into sign (neg), exponent, and significand.
aneg = a[63];    bneg = b[63];
aexp = a[62:52]; bexp = b[62:52];
// Put a 0 in bits 53 and 54 (later used for sign).
// Put a 1 in bit 52 of significand if exponent is non-zero.
// Copy significand into remaining bits.
asig = { 2'b0, aexp ? 1'b1 : 1'b0, a[51:0] };
bsig = { 2'b0, bexp ? 1'b1 : 1'b0, b[51:0] };

/// Step ?: If necessary, negate significands.
if( aneg ) asig = -asig;
if( bneg ) bsig = -bsig;

/// Solution: Step 3: Un-normalize b so that aexp == bexp.
//
diff = aexp - bexp;
bsig = bsig >> diff;

/// Step ?: Compute the sum.
sumsig = asig + bsig;
```



```
/// Step ?: Take absolute value of sum.
sumneg = sumsig[54];
if( sumneg ) sumsig = -sumsig;

/// Step 7: Normalize sum. (Three cases.)
// Code omitted from exam, but it's not the missing step.
/// Step 3: Un-normalize b so that aexp == bexp.
//
diff = aexp - bexp;
bsig = bsig >> diff;
//
// Note: bexp no longer used. If it were would need to set bexp = aexp;
```

EE 3755 Fall 2001 Final Exam Appendix  
Hardwired Control MIPS Implementation

Name: \_\_\_\_\_

Lines changed for Problem 1 have Prob 1 on the right side.

```

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;
    reg [31:0] rs_val, rt_val, rt_vx4; // YYY

    // 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;

```

# Hardwired Control MIPS Implementation

For Problems 1,3,5

```

reg [4:0]  wb_rd;    // Register number to write.
reg        wb_npc;    // YYY
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_xxx = 6'h3f;    // XXX
parameter f_yyy = 6'h3e;    // 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_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;
parameter o_bclp = 6'h3f;    // Prob 1

// Processor Control Logic States
//
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_xxx = 9; // XXX

/// 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_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;
        ii = ir[25:0];
        immed = ir[15:0];

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

        rs_val = gpr[rs];
        rt_val = gpr[rt];
        rt_vx4 = {rt_val[29:0],2'b0};

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

```

```

case( opcode )

o_rfmt:
  // R-Format Instructions
  case ( func )
    f_xxx  : begin alu_a = rs_val;  alu_op = op_xxx;      // XXX
              alu_b = rt_val;  wb_rd = rd;              end
    f_yyy  : begin alu_a = rs_val;  alu_op = op_add;     // YYY
              alu_b = rt_vx4;  wb_rd = 0;              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_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_bclp: // Prob 1
        begin alu_a = rs_val;  alu_op = op_sub;
        alu_b = 1;          wb_rd = rs;          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, me_wb, and wb_npc
  //

```

```

case( opcode )
  o_rfmt :
    case( func )
      f_yyy : begin me_size = 3; me_we = 0; wb_npc = 1; end// YYY
      default : begin me_size = 0; me_we = 0; wb_npc = 0; end
    endcase
  o_lbu   : begin me_size = 1; me_we = 0; wb_npc = 0; end
  o_sb    : begin me_size = 1; me_we = 1; wb_npc = 0; end
  default : begin me_size = 0; me_we = 0; wb_npc = 0; end
endcase

pc = npc;

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

case( opcode )
  o_rfmt:
    case( func )
      f_yyy : state = st_ex_addr; // YYY
      default : state = st_ex;
    endcase
  o_lbu, o_sb : state = st_ex_addr;
  o_bcloop, o_bne, o_beq // Prob 1
                : state = st_ex_cond;
  o_j          : state = st_if;
  default      : state = st_ex;
endcase

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]
    || opcode == o_bne && !alu_out[0]
    || opcode == o_bcloop && !alu_out[31] ) begin // Prob 1
    alu_a = pc;
    alu_b = simmed << 2;
    alu_op = op_add;
    state = st_ex_targ;
    if( wb_rd ) gpr[wb_rd] = alu_out; // Prob 1
  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;
  if( wb_npc ) npc = data_in; // YYY
  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_xxx = 9;                                     // XXX

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_xxx  : alu_out = alu_a & ~alu_b;                   // XXX
    op_nop  : alu_out = 0;
    default : begin alu_out = 0;  $stop;  end
  endcase

endmodule
```



## Microcoded Control MIPS Implementation Excerpt For Problem 4 Only

```
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 [7:0] exc;
  // Code omitted.
  wire [31:0] alu_out;
  reg [31:0] alu_a, alu_b;
  reg [5:0] alu_op;
  reg [31:0] pc, npc, ir;

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

  reg_file our_reg_file(rs_val,rt_val,...);

  // Code omitted.

  always @( alu_a_src or rs_val or pc or sa or reset )
    case( alu_a_src )
      ALU_A_00: alu_a = 32'h0;
      default: alu_a = rs_val;
    endcase

  always @( alu_b_src or rt_val or immed or pc or ii or reset )
    case( alu_b_src )
      ALU_B_RT: alu_b = rt_val;
      default: alu_b = 32'h4;
    endcase

  always @( alu_op_src or micro_alu_op )
    case( alu_op_src )
      ALU_ADD: alu_op = op_add;
      default: alu_op = micro_alu_op; // Use dispatch table operation.
    endcase

  always @( addr_src or pc or alu_out )
    case( addr_src )
      ADDR_PC: addr = pc;
      default: addr = alu_out;
    endcase

  assign data_out = rt_val;

  always @( posedge clk ) if( ir_en ) ir = data_in;

  always @( posedge clk )
    case( pc_op )
      PC_ADV: pc = npc;
      PC_BRN: if( branch_cond ) npc = alu_out;
      PC_ALU: npc = alu_out;
      default: ; // No nothing.
    endcase

  // Code omitted.
  // End of microcoded MIPS Verilog code (for his exam).
```