CS 3220: Processor Design
Fall 2014

Midterm Exam

Explain your answers, but be concise. You should write your answers in the space provided with each question. Write legibly – illegible answers are wrong answers.
(1) A stack machine implements registers with a stack. The operands of the arithmetic logic unit (ALU) are usually the top two registers of the stack and the result from the ALU is stored in the top register of the stack. "Stack machine" commonly refers to computers which use a Last-in, First-out stack to hold short-lived temporary values while executing individual program instructions. Stack machines have much smaller instructions than the other styles of machines.

The following instructions implement the depicted expression tree: This is just an example to give you a feeling about the stack machine. These instruction are not the same as the instruction in the ISA.

```
push x
push y
push z
mult
add
push u
add
```

Design the ISA for a stack machine that supports the following instructions. Your ISA should have the minimum number of bits for the instruction word. You instruction word should have a primary opcode and secondary opcode. The operations are all 32bit and the immediate are 10 bits that need to be sign extended.

Group 1: add, sub, neg, mult, and, or, xor, not
Example: add: pops the two top registers of the stacks; adds them; pushes the result;

Group 2, addi, subi, negi, multi, andi, ori, xori, noti
Example: addi Imm: pops the top register of the stacks; adds the sign extended Imm to top register; pushes the result;

Group 3: push
Example: push: pops the top register of the stacks; the top register contains the address; loads the value stored in the address and pushes it to the stack

Group 4: pop
Example: push: pops the two top registers of the stacks; the top register contains the address; the next to top is the value; stores the value in the stack

Group 5: eq, gt, leq
Example: eq: pops the two top registers of the stacks; compares them; pushes 1 if they are equal otherwise pushes 0

Group 6: branch_zero branch_nzero
Example: branch_zero: pops the two top registers of the stacks; the top register contains the branch target address; if the next to top register contains zero; it jumps to the address

Group 7: push_pc
Example: push_pc: pushes PC + 2

Group 8:
Example: pop_pc: pops the top of the stack and stores it in the PC
The minimum number of bits needed for this ISAs is 16 as described below.

<table>
<thead>
<tr>
<th>Primary Opcode</th>
<th>Secondary Opcode</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>sub</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>neg</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>mult</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>and</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>or</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>xor</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>not</td>
<td>000</td>
<td>0000000000</td>
</tr>
<tr>
<td>addi</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>subi</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>negi</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>multi</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>andi</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>ori</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>xori</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>noti</td>
<td>100</td>
<td>0000000000 Imm</td>
</tr>
<tr>
<td>push</td>
<td>010</td>
<td>0000000000</td>
</tr>
<tr>
<td>pop</td>
<td>011</td>
<td>0000000000</td>
</tr>
<tr>
<td>eq</td>
<td>100</td>
<td>0000000000</td>
</tr>
<tr>
<td>gt</td>
<td>100</td>
<td>0000000000</td>
</tr>
<tr>
<td>leq</td>
<td>100</td>
<td>0000000000</td>
</tr>
<tr>
<td>branch_zero</td>
<td>101</td>
<td>0000000000</td>
</tr>
<tr>
<td>branch_nzero</td>
<td>101</td>
<td>0000000000</td>
</tr>
<tr>
<td>push_pc</td>
<td>110</td>
<td>0000000000</td>
</tr>
<tr>
<td>pop_pc</td>
<td>111</td>
<td>0000000000</td>
</tr>
</tbody>
</table>
(2) Design a stack with 16, 32-bit registers and implement it with Verilog and make your code parametric.

```verilog
module Stack (clk, rst, push, pop, pop2, dataIn, dataOut1, dataOut2, empty, full);

  parameter ADDR_BITWIDTH = 4;
  parameter DATA_BITWIDTH = 32;
  parameter N_ENTRIES = (1 << ADDR_BITWIDTH);

  input clk;
  input rst;
  input push;
  input pop;
  input pop2;
  input [DATA_BITWIDTH - 1: 0] dataIn;
  output [DATA_BITWIDTH - 1: 0] dataOut1;
  output [DATA_BITWIDTH - 1: 0] dataOut2;
  output empty;
  output full;

  reg [ADDR_BITWIDTH: 0] head = 0;
  reg [DATA_BITWIDTH - 1: 0] data [0: N_ENTRIES - 1];

  wire [1: 0] popCmd;
  assign popCmd = {pop2, pop};

  always @(posedge clk) begin
    if(rst) head <= 0;
    else begin
      if(push == 1'b1 & head[ADDR_BITWIDTH] != 1'b1) begin
        head <= head + 1;
        data[head] <= dataIn;
      end
      else begin
        if (head >= popCmd)
          head <= head - popCmd;
        else
          head <= head;
      end
    end

    assign dataOut1 = (head >= 1) ? data[head - 1] : 0;
    assign dataOut2 = (head >= 2) ? data[head - 2] : 0;

    assign empty = (head == 0);
    assign full = (head == N_ENTRIES);
  endmodule
```
(3) Design the single-cycle stack machine and show its detailed diagram.
Implement the ALU in Verilog with only one 32-bit adder. That is, the subtraction and negation should use the same adder, which is used by the addition. The same adder should be used for comparison operations. You can use Verilog's `*` operator to implement the multiplication.

module Alu (ctrl, dataIn1, dataIn2, dataOut);
  parameter DATA_BIT_WIDTH = 32;
  parameter CTRL_BIT_WIDTH = 4;
  parameter
    CMD_ADD = 3'b000,
    CMD_SUB = 3'b001,
    CMD_NEG = 3'b010,
    CMD_MUL = 3'b011,
    CMD_AND = 3'b100,
    CMD_OR = 3'b101,
    CMD_XOR = 3'b110,
    CMD_NOT = 3'b111,
    CMD_EQ = 3'b000,
    CMD_GT = 3'b001,
    CMD_LEQ = 3'b010;
  input [CTRL_BIT_WIDTH - 1: 0] ctrl;
  input [DATA_BIT_WIDTH - 1: 0] dataIn1, dataIn2;
  output [DATA_BIT_WIDTH - 1: 0] dataOut;

  wire isCmp;
  assign isCmp = ctrl[CTRL_BIT_WIDTH - 1];

  wire [CTRL_BIT_WIDTH - 2: 0] command;
  assign command = ctrl [CTRL_BIT_WIDTH - 2: 0];

  wire signed [DATA_BIT_WIDTH - 1: 0] signedDataIn1, signedDataIn2;
  assign signedDataIn1 = dataIn1;
  assign signedDataIn2 = dataIn2;

  wire signed [DATA_BIT_WIDTH - 1: 0] data1, data2;
  assign data1 = (command == CMD_NEG) ? ~signedDataIn1 : signedDataIn2;
  assign data2 = (command == CMD_NEG) ? 0 :
    ((command == CMD_SUB) | (isCmp == 1'b1)) ? ~signedDataIn2 :
    signedDataIn2;

  wire carryIn;
  assign carryIn = ((command == CMD_SUB) | (command == CMD_NEG) | (isCmp == 1'b1))
    ? 1 : 0;

  wire [DATA_BIT_WIDTH - 1: 0] adderOut;
  assign adderOut = dataIn1 + dataIn2 + carryIn;
wire allZero;
assign allZero = ~adderOut;

reg [DATA_BIT_WIDTH - 1: 0] tmpOut;
always @(*) begin
    tmpOut = 0;
    if (isCmp == 1'b1) begin
        if (command == CMD_EQ & allZero) begin
            tmpOut = 32'b1;
        end
    end else if (command == CMD_GT & !adderOut [DATA_BIT_WIDTH - 1] & !allZero) begin
        tmpOut = 32'b1;
    end else if (command == CMD_LEQ & (adderOut [DATA_BIT_WIDTH - 1] | allZero)) begin
        tmpOut = 32'b1;
    end else begin
        tmpOut = 32'b0;
    end
endcase
end

assign dataOut = tmpOut;
endmodule
(5) Implement the instruction decoder and the controller in Verilog.

module Controller (inst, dataOut2, aluCtrl, imm, push, pop, pop2, dataWrtEn, immSel, resultSel, pcSel);
  parameter DATA_BIT_WIDTH = 32;
  parameter
    GROUP1 = 3'b000,
    GROUP2 = 3'b001,
    GROUP3 = 3'b010,
    GROUP4 = 3'b011,
    GROUP5 = 3'b100,
    GROUP6 = 3'b101,
    GROUP7 = 3'b110,
    GROUP8 = 3'b111;

  input [15: 0] inst;
  input [DATA_BIT_WIDTH - 1: 0] dataOut2;
  output [3: 0] aluCtrl;
  output [9: 0] imm;
  output push;
  output pop;
  output pop2;
  output dataWrtEn;
  output immSel;
  output [1: 0] resultSel;
  output pcSel;

  wire [2: 0] fstOpcode, sndOpcode;
  assign fstOpcode = inst[15: 13];
  assign sndOpcode = inst[12: 10];

  wire [9: 0] imm;
  assign imm = inst[9: 0];

  assign aluCtrl[3] = (fstOpcode == GROUP5) ? 1 : 0;
  assign aluCtrl[2: 0] = sndOpcode;

  wire push;
  assign push = (fstOpcode == GROUP1 ||
                   fstOpcode == GROUP2 ||
                   fstOpcode == GROUP3 ||
                   fstOpcode == GROUP5 ||
                   fstOpcode == GROUP7) ? 1'b1 : 1'b0;

  wire pop;
  assign pop = (fstOpcode == GROUP2 ||
                 fstOpcode == GROUP3 ||
                 fstOpcode == GROUP8) ? 1'b1 : 1'b0;
wire pop2;
assign pop2 = (fstOpcode == GROUP1 ||
fstOpcode == GROUP4 ||
fstOpcode == GROUP5 ||
fstOpcode == GROUP6) ? 1'b1 : 1'b0;

wire dataWrtEn;
assign dataWrtEn = (fstOpcode == GROUP4) ? 1'b1 : 1'b0;

wire immSel;
assign immSel = (fstOpcode == GROUP2) ? 1'b1 : 1'b0;

wire resultSel;
assign resultSel = (fstOpcode == GROUP4) ? 2'b00 :
(fstOpcode == GROUP7) ? 2'b01 : 2'b10;

reg pcSel;
always @ (*) begin
  pcSel = 0;
  if (fstOpcode == GROUP6) begin
    if ((sndOpcode[0] == 1'b0) & (dataOut2 == 32'b0))
      pcSel = 1'b1;
    else if ((sndOpcode[0] == 1'b1) & (dataOut2 != 32'b0))
      pcSel = 1'b1;
  end
end
endmodule