University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

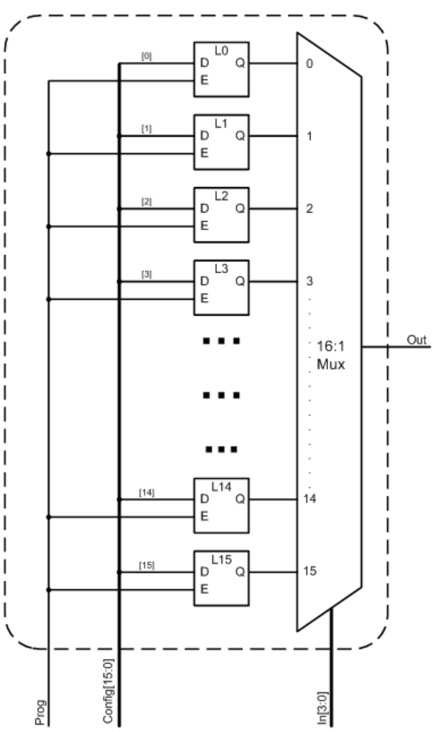
EECS150, Spring 2010

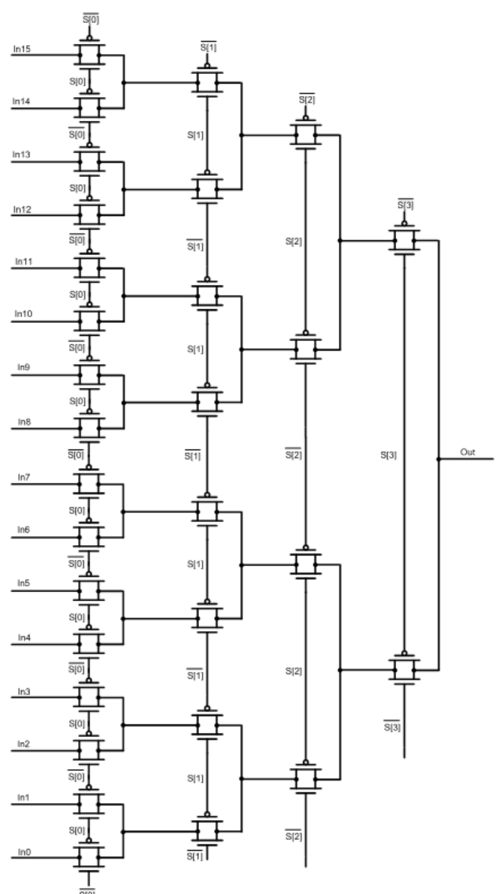
Homework 5 Solutions: More Transistors and Single Cycle Processor Implementations

Homework submission will only be through SVN. Email submissions will not be accepted! Please format your homework as plain text with either PNG or PDF for any necessary figures. Microsoft Visio is installed on the machines in 125 Cory, and is a useful tool for drawing figures of all kinds.

1. Implement a basic 4-input LUT down to the transistor level. Allow the LUT to be programmable with a new function whenever the `prog` signal is asserted (there is no clock). You may implement this in any way you wish, however, the goal is to minimize the number of transistors used. How many configuration bits does your LUT need?
   Hint: You may want to consider latches, which, unlike flip-flops, may be triggered using signals besides `clock`. Transmission gate muxes may also be useful.

   See solution below. We will need 16 configuration bits for the LUT, one bit for each latch. See lecture notes for Lecture #9 if you need to review how to make a level-sensitive latch.



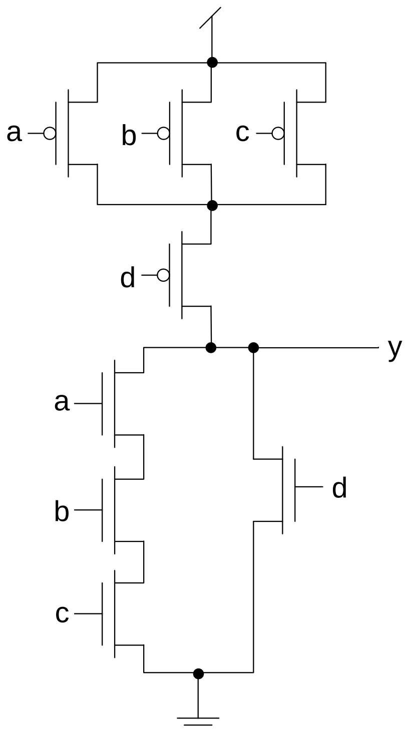


2. Using the minimum number of transistors, draw the Static CMOS implementations for the following functions:

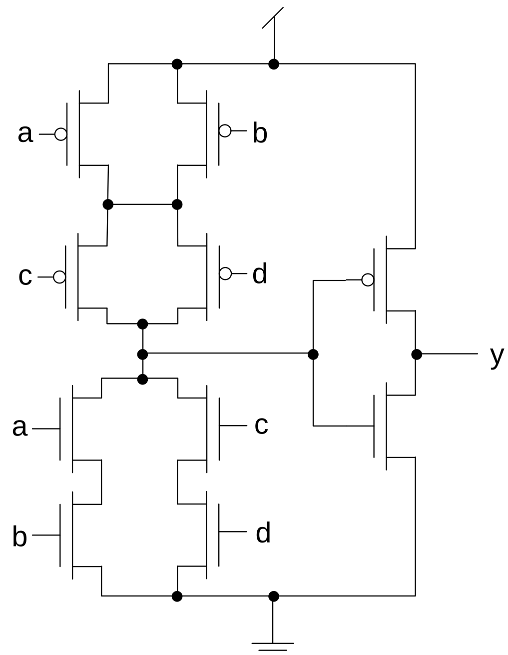
   (a) `y = (abc + d)'`

   (b) `y = ab + cd`

   See solution below.

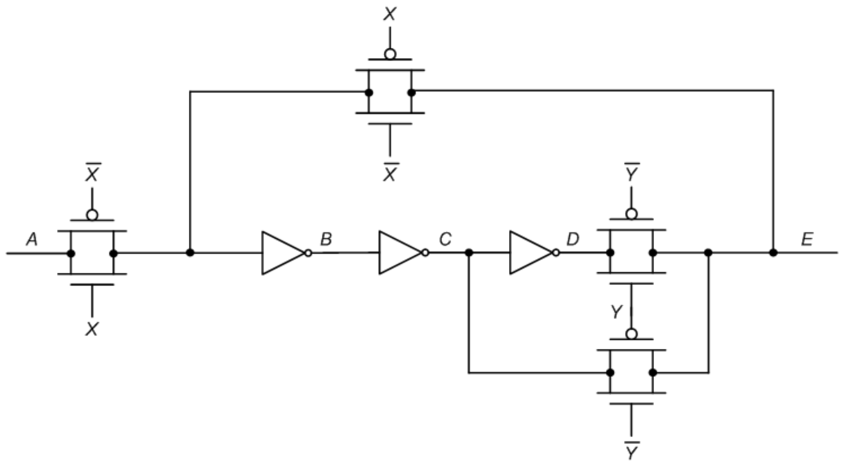


(a) `y = (abc + d)'`



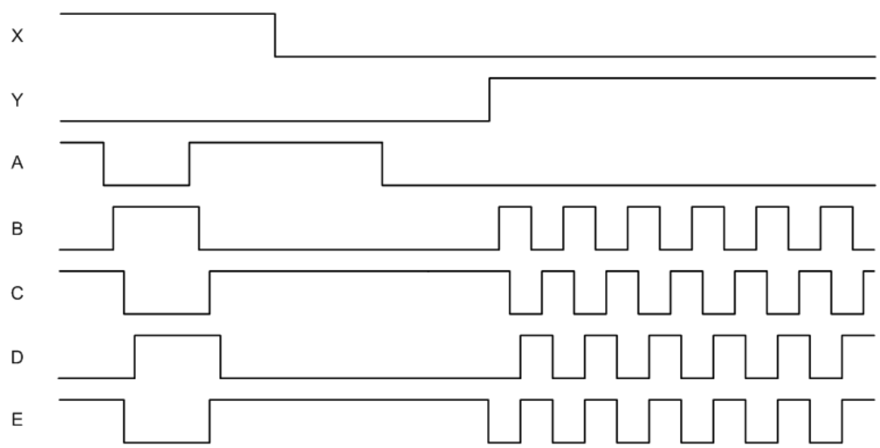
(b) `y = ab + cd`

3. Consider the circuit pictured here:



(a) Complete the timing diagram below for the circuit. Note that it may be helpful for you to think about the inverters as having a little bit of delay. What does this circuit do if **X** is 0 and **Y** is 1? Why?

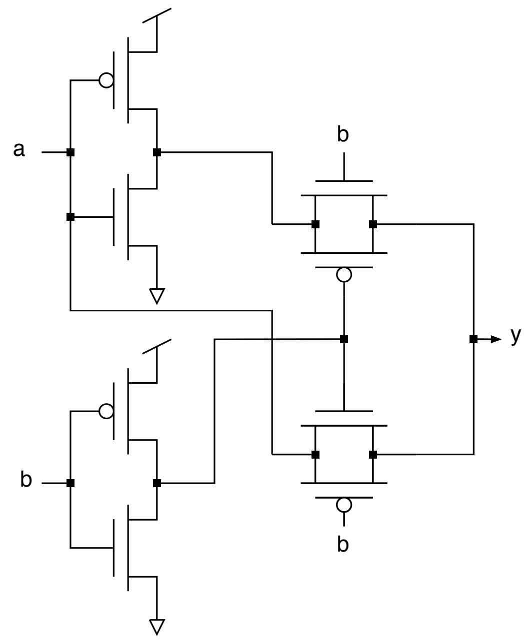
The circuit forms a combinational loop of 3 inverters when **X** is 0 and **Y** is 1. When we make a combinational loop with an odd number of inverters, we form an unstable circuit that will oscillate at a certain frequency. In fact, a circuit with an odd number of inverters in a loop is called a **ring oscillator**. See completed figure below for what the waveform will look like.



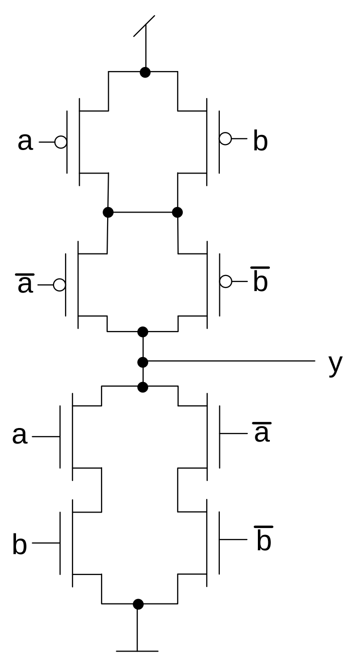
(b) How would the waveform for signal **E** change if the circuit had 5 inverters in series instead of 3? Why?

Adding more inverters in series in a ring oscillator simply increases the period (decreases the frequency) of the oscillation, since now it takes a longer time for a signal to propagate all the way through the ring.

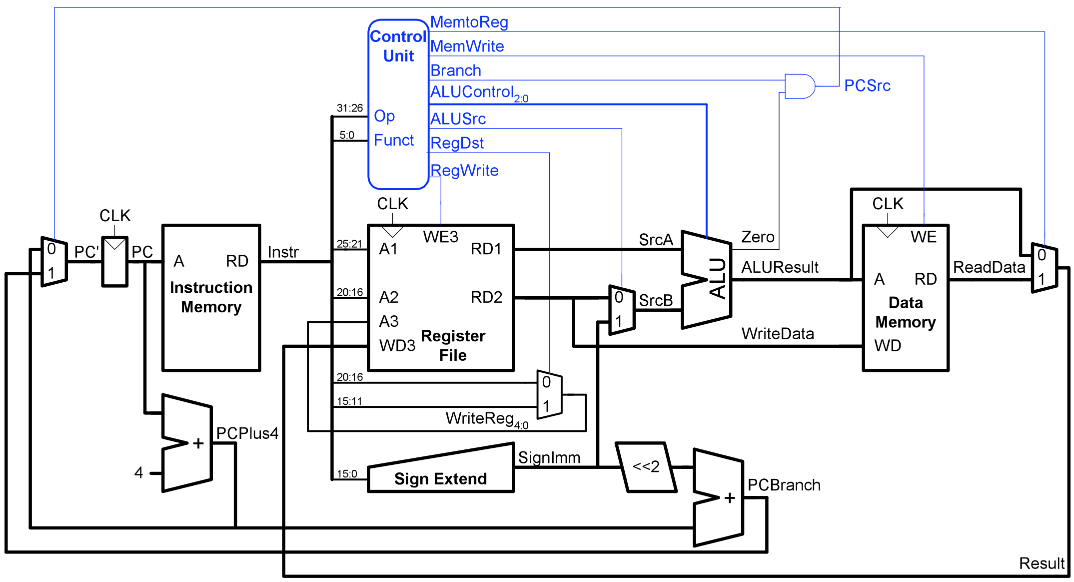
4. For the CMOS circuit shown in below, write the truth table, give a Boolean expression that corresponds to the circuit, and draw an alternative implementation that doesn't use transmission gates:



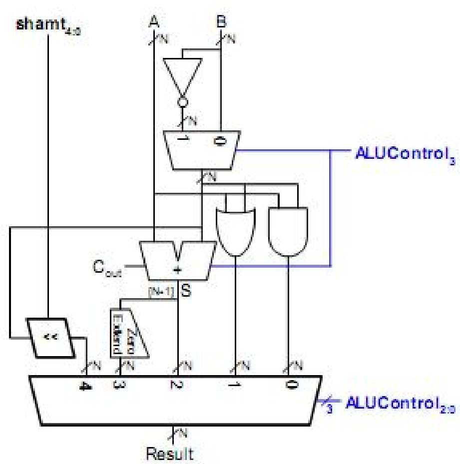
This is an XOR circuit with a corresponding Boolean expression, `y = a ⊕ b`. The circuit can be represented in Static CMOS as follows (inverters for **a** and **b** are not shown):



5. (based on DDCA 7.3) - Modify the datapath shown below to add functionality for **`sll`** and **`jal`**:



Modifications to the ALU for **`sll`**:



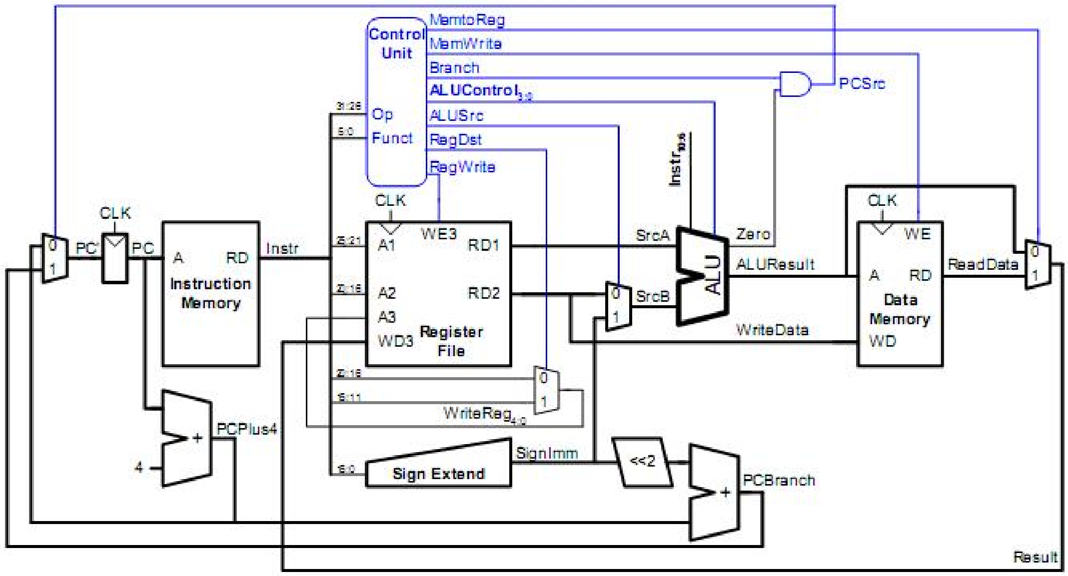
Modified ALU Control for **`sll`**:

$ALUControl_{3:0}$	Function
0000	A AND B
0001	A OR B
0010	A + B
0011	not used
1000	A AND $\overline{B}$
1001	A OR $\overline{B}$
1010	A - B
1011	SLT
0100	SLL

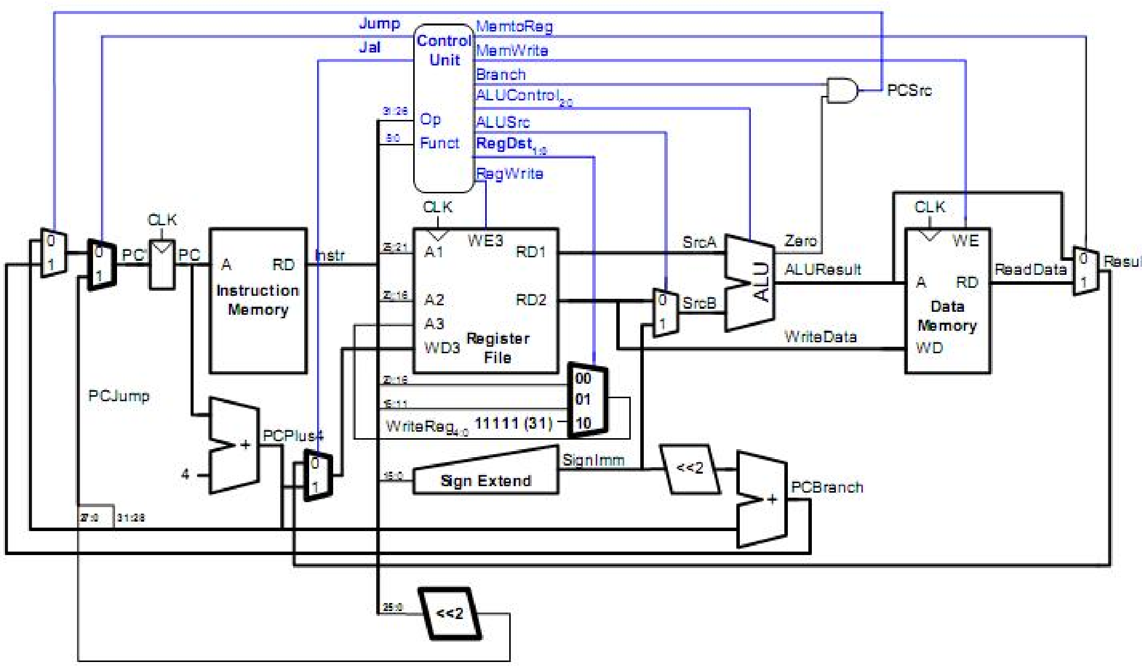
Modified ALU Decoder for **`sll`**:

ALUOp	Funct	ALUControl
00	X	0010 (add)
X1	X	1010 (subtract)
1X	100000 (`add`)	0010 (add)
1X	100010 (`sub`)	1010 (subtract)
1X	100100 (`and`)	0000 (and)
1X	100101 (`or`)	0001 (or)
1X	101010 (`slt`)	1011 (set less than)
1X	000000 (`sll`)	0100 (shift left logical)

Modified Datapath for **`sll`**:



Modified Datapath for **`jal`**:



Modified Decoder for **`jal`**:

Instruction	opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump	Jal
R-type	000000	1	01	0	0	0	0	10	0	0
lw	100011	1	00	1	0	0	1	00	0	0
sw	101011	0	XX	1	0	1	X	00	0	0
beq	000100	0	XX	0	1	0	X	01	0	0
addi	001000	1	00	1	0	0	0	00	0	0
j	000010	0	XX	X	X	0	X	XX	1	0
jal	000011	1	10	X	X	0	X	XX	1	1

6. (based on DDCA 7.10) - Do the following for **sll** and **jal** only.

- For parts **(a)** through **(e)**, make the necessary changes in the Verilog to implement the changes made to the datapath in Problem 5. Changes shown below.
- Study parts **(f)** through **(i)** and write a simple MIPS program to test the functionality of the changes you implemented. Individual answers may vary greatly.

(a) Single-cycle MIPS Processor:

```
  module mips(input          clk, reset,
              output [31:0] pc,
              input  [31:0] instr,
              output        memwrite,
              output [31:0] aluout, writedata,
              input  [31:0] readdata);

  wire          memtoreg, branch,
                pcsrc, zero,
                alusrc, regwrite, jump;
  wire [3:0]    alucontrol;
  wire [1:0]    regdst;
  wire          jal;

  controller c(instr[31:26], instr[5:0], zero,
                memtoreg, memwrite, pcsrc,
                alusrc, regdst, regwrite, jump,
                alucontrol,
                jal
                );
  datapath dp(clk, reset, memtoreg, pcsrc,
              alusrc, regdst, regwrite, jump,
              alucontrol,
              zero, pc, instr,
              aluout, writedata, readdata,
              jal
              );
endmodule
```

(b) Controller:

```
module controller(input  [5:0] op, funct,
                  input        zero,
                  output       memtoreg, memwrite,
                  output       pcsrc, alusrc, regwrite,
                  output [1:0] regdst,
                  output       jump,
                  output [3:0] alucontrol,
                  output       jal
                  );

  wire [1:0] aluop;
  wire       branch;

  maindec md(op, memtoreg, memwrite, branch,
             alusrc, regdst, regwrite, jump,
             jal,
             aluop);
  aludec  ad(funct, aluop, alucontrol);

  assign pcsrc = branch & zero;
endmodule
```

(c) Main Decoder:

```
module maindec(input  [5:0] op,
               output       memtoreg, memwrite,
               output       branch, alusrc, regwrite,
               output [1:0] regdst,
               output       jal,
               output       jump,
               output [1:0] aluop);

  reg [10:0] controls;

  assign {regwrite, regdst, alusrc,
          branch, memwrite,
          memtoreg, jump, aluop,
          jal
          } = controls;

  always @(*)
    case(op)
      6'b000000: controls <= 11'b10100000100; //Rtype
      6'b100011: controls <= 11'b10010010000; //LW
      6'b101011: controls <= 11'b00010100000; //SW
```

```
      6'b000100: controls <= 11'b00001000010; //BEQ
      6'b001000: controls <= 11'b10010000000; //ADDI
      6'b000010: controls <= 11'b00000001000; //J
      6'b000011: controls <= 11'b11000001001; //JAL
      default:   controls <= 9'bxxxxxxxxx; //???
    endcase
endmodule
```

(d) ALU Decoder:

```
module aludec(input      [5:0] funct,
              input      [1:0] aluop,
              output reg [3:0] alucontrol);

  always @(*)
    case(aluop)
      2'b00: alucontrol <= 4'b0010;  // add
      2'b01: alucontrol <= 4'b1010;  // sub
      default: case(funct)          // RTYPE
          6'b100000: alucontrol <= 4'b0010; // ADD
          6'b100010: alucontrol <= 4'b1010; // SUB
          6'b100100: alucontrol <= 4'b0000; // AND
          6'b100101: alucontrol <= 4'b0001; // OR
          6'b101010: alucontrol <= 4'b1011; // SLT
          6'b000000: alucontrol <= 4'b0100; // SLL
          default:   alucontrol <= 4'bxxxx; // ???
        endcase
    endcase
endmodule
```

(e) Datapath:

```
module datapath(input          clk, reset,
                input          memtoreg, pcsrc,
                input          alusrc,
                input   [1:0]  regdst,
                input   [3:0]  alucontrol,
                input          jal,
                input          regwrite, jump,
                output         zero,
                output  [31:0] pc,
                input   [31:0] instr,
                output  [31:0] aluout, writedata,
                input   [31:0] readdata);

  wire [4:0]  writereg;
  wire [31:0] pcnext, pcnextbr, pcplus4, pcbranch;
  wire [31:0] signimm, signimmsh;
```

```
  wire [31:0] srca, srcb;
  wire [31:0] result;
  wire [31:0] writeresult;

  // next PC logic
  flopr #(32) pcreg(clk, reset, pcnext, pc);
  adder       pcadd1(pc, 32'b100, pcplus4);
  sl2         immsh(signimm, signimmsh);
  adder       pcadd2(pcplus4, signimmsh, pcbranch);
  mux2 #(32)  pcbrmux(pcplus4, pcbranch, pcsrc,
                      pcnextbr);
  mux2 #(32)  pcmux(pcnextbr, {pcplus4[31:28],
                    instr[25:0], 2'b00},
                    jump, pcnext);

  // register file logic
  regfile     rf(clk, regwrite, instr[25:21],
                 instr[20:16], writereg,
                 result, srca, writedata);

  mux2 #(32)  wamux(result, pcplus4, jal, writeresult);

  mux3 #(5)   wrmux(instr[20:16], instr[15:11], 5'd31,
                    regdst, writereg);
  mux2 #(32)  resmux(aluout, readdata,
                     memtoreg, result);
  signext     se(instr[15:0], signimm);

  // ALU logic
  mux2 #(32)  srcbmux(writedata, signimm, alusrc,
                      srcb);
  alu         alu(srca, srcb, alucontrol,
                  instr[10:6],
                  aluout, zero);
endmodule
```

(f) MIPS Testbench:

```
module testbench();

  reg         clk;
  reg         reset;

  wire [31:0] writedata, dataadr;
  wire memwrite;

  // instantiate device to be tested
  top dut(clk, reset, writedata, dataadr, memwrite);
```

```
  // initialize test
  initial
    begin
      reset <= 1; # 22; reset <= 0;
    end

  // generate clock to sequence tests
  always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end

  // check that 7 gets written to address 84
  always@(negedge clk)
    begin
      if(memwrite) begin
        if(dataadr === 84 & writedata === 7) begin
          $display("Simulation succeeded");
          $stop;
        end else if (dataadr !== 80) begin
          $display("Simulation failed");
          $stop;
        end
      end
    end
endmodule
```

(g) MIPS Top-Level Module:

```
module top(input         clk, reset,
           output [31:0] writedata, dataadr,
           output        memwrite);

  wire [31:0] pc, instr, readdata;

  // instantiate processor and memories
  mips mips(clk, reset, pc, instr, memwrite, dataadr, writedata, readdata);
  imem imem(pc[7:2], instr);
  dmem dmem(clk, memwrite, dataadr, writedata, readdata);

endmodule
```

(h) MIPS Data Memory:

```
module dmem(input         clk, we,
            input  [31:0] a, wd,
            output [31:0] rd);

  reg  [31:0] RAM[63:0];

  assign rd = RAM[a[31:2]]; // word aligned

  always @(posedge clk)
    if (we)
      RAM[a[31:2]] <= wd;
endmodule
```

(i) MIPS Instruction Memory:

```
module imem(input  [5:0] a,
            output [31:0] rd);

  reg  [31:0] RAM[63:0];

  initial
    begin
      $readmemh("memfile.dat",RAM);
    end

  assign rd = RAM[a]; // word aligned
endmodule
```