

```

//-----
// mips.v
// Max Yi (byyi@hmc.edu) and David_Harris@hmc.edu 12/9/03
// Model of subset of MIPS processor described in Ch 1
//-----

// top level design for testing
module top #(parameter WIDTH = 8, REGBITS = 3)();

    reg                clk;
    reg                reset;
    wire               memread, memwrite;
    wire   [WIDTH-1:0] adr, writedata;
    wire   [WIDTH-1:0] memdata;

    // instantiate devices to be tested
    mips #(WIDTH,REGBITS) dut(clk, reset, memdata, memread, memwrite, adr,
writedata);

    // external memory for code and data
    exmemory #(WIDTH) exmem(clk, memwrite, adr, writedata, memdata);

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

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

    always@(negedge clk)
        begin
            if(memwrite)
                if(adr == 5 & writedata == 7)
                    $display("Simulation completely successful");
                else $display("Simulation failed");
        end
endmodule

```

```

// external memory accessed by MIPS
module exmemory #(parameter WIDTH = 8)
    (clk, memwrite, adr, writedata, memdata);

    input          clk;
    input          memwrite;
    input  [WIDTH-1:0] adr, writedata;
    output reg [WIDTH-1:0] memdata;

    reg  [31:0] RAM [(1<<WIDTH-2)-1:0];
    wire [31:0] word;

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

    // read and write bytes from 32-bit word
    always @(posedge clk)
        if(memwrite)
            case (adr[1:0])
                2'b00: RAM[adr>>2][7:0] <= writedata;
                2'b01: RAM[adr>>2][15:8] <= writedata;
                2'b10: RAM[adr>>2][23:16] <= writedata;
                2'b11: RAM[adr>>2][31:24] <= writedata;
            endcase

    assign word = RAM[adr>>2];
    always @(*)
        case (adr[1:0])
            2'b00: memdata <= word[31:24];
            2'b01: memdata <= word[23:16];
            2'b10: memdata <= word[15:8];
            2'b11: memdata <= word[7:0];
        endcase
endmodule

```

```

// simplified MIPS processor
module mips #(parameter WIDTH = 8, REGBITS = 3)
    (input          clk, reset,
     input  [WIDTH-1:0] memdata,
     output        memread, memwrite,
     output  [WIDTH-1:0] adr, writedata);

    wire [31:0] instr;
    wire        zero, alusrca, memtoreg, iord, pcen, regwrite, regdst;
    wire [1:0]  aluop, pcsource, alusrcb;
    wire [3:0]  irwrite;
    wire [2:0]  alucont;

    controller  cont(clk, reset, instr[31:26], zero, memread, memwrite,
                    alusrca, memtoreg, iord, pcen, regwrite, regdst,
                    pcsource, alusrcb, aluop, irwrite);
    alucontrol  ac(aluop, instr[5:0], alucont);
    datapath    dp(WIDTH, REGBITS)
                dp(clk, reset, memdata, alusrca, memtoreg, iord, pcen,
                  regwrite, regdst, pcsource, alusrcb, irwrite, alucont,
                  zero, instr, adr, writedata);
endmodule

```

```

module controller(input clk, reset,
                 input      [5:0] op,
                 input      zero,
                 output reg memread, memwrite, alusrca, memtoreg, iord,
                 output      pcen,
                 output reg  regwrite, regdst,
                 output reg [1:0] pcsource, alusrcb, aluop,
                 output reg [3:0] irwrite);

parameter  FETCH1  = 4'b0001;
parameter  FETCH2  = 4'b0010;
parameter  FETCH3  = 4'b0011;
parameter  FETCH4  = 4'b0100;
parameter  DECODE  = 4'b0101;
parameter  MEMADR  = 4'b0110;
parameter  LBRD   = 4'b0111;
parameter  LBWR   = 4'b1000;
parameter  SBWR   = 4'b1001;
parameter  RTYPEEX = 4'b1010;
parameter  RTYPEWR = 4'b1011;
parameter  BEQEX  = 4'b1100;
parameter  JEX    = 4'b1101;

parameter  LB      = 6'b100000;
parameter  SB      = 6'b101000;
parameter  RTYPE   = 6'b0;
parameter  BEQ     = 6'b000100;
parameter  J       = 6'b000010;

reg [3:0] state, nextstate;
reg      pcwrite, pcwritecond;

```

```

// state register
always @(posedge clk)
    if(reset) state <= FETCH1;
    else state <= nextstate;

// next state logic
always @(*)
    begin
        case(state)
            FETCH1: nextstate <= FETCH2;
            FETCH2: nextstate <= FETCH3;
            FETCH3: nextstate <= FETCH4;
            FETCH4: nextstate <= DECODE;
            DECODE: case(op)
                LB:      nextstate <= MEMADR;
                SB:      nextstate <= MEMADR;
                RTYPE:  nextstate <= RTYPEEX;
                BEQ:    nextstate <= BEQEX;
                J:      nextstate <= JEX;
                default: nextstate <= FETCH1; // should never
happen

                endcase
            MEMADR: case(op)
                LB:      nextstate <= LBRD;
                SB:      nextstate <= SBWR;
                default: nextstate <= FETCH1; // should never
happen

                endcase
            LBRD:  nextstate <= LBWR;
            LBWR:  nextstate <= FETCH1;
            SBWR:  nextstate <= FETCH1;
            RTYPEEX: nextstate <= RTYPEWR;
            RTYPEWR: nextstate <= FETCH1;
            BEQEX: nextstate <= FETCH1;
            JEX:   nextstate <= FETCH1;
            default: nextstate <= FETCH1; // should never happen
        endcase
    end

```

```

always @(*)
  begin
    // set all outputs to zero, then conditionally assert just the
appropriate ones
    irwrite <= 4'b0000;
    pcwrite <= 0; pcwritecond <= 0;
    regwrite <= 0; regdst <= 0;
    memread <= 0; memwrite <= 0;
    alusrca <= 0; alusrcb <= 2'b00; aluop <= 2'b00;
    pcsource <= 2'b00;
    iord <= 0; memtoreg <= 0;
    case(state)
      FETCH1:
        begin
          memread <= 1;
          irwrite <= 4'b1000;
          alusrcb <= 2'b01;
          pcwrite <= 1;
        end
      FETCH2:
        begin
          memread <= 1;
          irwrite <= 4'b0100;
          alusrcb <= 2'b01;
          pcwrite <= 1;
        end
      FETCH3:
        begin
          memread <= 1;
          irwrite <= 4'b0010;
          alusrcb <= 2'b01;
          pcwrite <= 1;
        end
      FETCH4:
        begin
          memread <= 1;
          irwrite <= 4'b0001;
          alusrcb <= 2'b01;
          pcwrite <= 1;
        end
      DECODE: alusrcb <= 2'b11;
      MEMADR:
        begin
          alusrca <= 1;
          alusrcb <= 2'b10;
        end
      LBRD:

```

```

        begin
            memread <= 1;
            iord     <= 1;
        end
LBWR:
        begin
            regwrite <= 1;
            memtoreg <= 1;
        end
SBWR:
        begin
            memwrite <= 1;
            iord     <= 1;
        end
RTYPEEX:
        begin
            alusrca <= 1;
            aluop   <= 2'b10;
        end
RTYPEWR:
        begin
            regdst  <= 1;
            regwrite <= 1;
        end
BEQEX:
        begin
            alusrca <= 1;
            aluop   <= 2'b01;
            pcwritecond <= 1;
            pcsource  <= 2'b01;
        end
JEX:
        begin
            pcwrite <= 1;
            pcsource <= 2'b10;
        end
    endcase
end
    assign pcen = pcwrite | (pcwritecond & zero); // program counter enable
endmodule

```

```

module alucontrol(input      [1:0] aluop,
                  input      [5:0] funct,
                  output reg [2:0] alucont);

always @(*)
  case(aluop)
    2'b00: alucont <= 3'b010; // add for lb/sb/addi
    2'b01: alucont <= 3'b110; // sub (for beq)
    default: case(funct) // R-Type instructions
      6'b100000: alucont <= 3'b010; // add (for add)
      6'b100010: alucont <= 3'b110; // subtract (for sub)
      6'b100100: alucont <= 3'b000; // logical and (for
and)
      6'b100101: alucont <= 3'b001; // logical or (for or)
      6'b101010: alucont <= 3'b111; // set on less (for
slt)
      default: alucont <= 3'b101; // should never happen
    endcase
  endcase
endmodule

```



```

module datapath #(parameter WIDTH = 8, REGBITS = 3)
    (input          clk, reset,
     input  [WIDTH-1:0] memdata,
     input          alusrca, memtoreg, iord, pcen,
regwrite, regdst,
     input  [1:0]    pcsource, alusrcb,
     input  [3:0]    irwrite,
     input  [2:0]    alucont,
     output         zero,
     output [31:0]   instr,
     output [WIDTH-1:0] adr, writedata);

    // size of the parameters must be changed to match the WIDTH parameter
    parameter CONST_ZERO = 8'b0;
    parameter CONST_ONE  = 8'b1;

    wire [REGBITS-1:0] ra1, ra2, wa;
    wire [WIDTH-1:0] pc, nextpc, md, rd1, rd2, wd, a, src1, src2, aluresult,
        aluout, constx4;

    // shift left constant field by 2
    assign constx4 = {instr[WIDTH-3:0],2'b00};

    // register file address fields
    assign ra1 = instr[REGBITS+20:21];
    assign ra2 = instr[REGBITS+15:16];
    mux2      #(REGBITS) regmux(instr[REGBITS+15:16], instr[REGBITS+10:11],
regdst, wa);

    // independent of bit width, load instruction into four 8-bit registers
over four cycles
    flopen    #(8)      ir0(clk, irwrite[0], memdata[7:0], instr[7:0]);
    flopen    #(8)      ir1(clk, irwrite[1], memdata[7:0], instr[15:8]);
    flopen    #(8)      ir2(clk, irwrite[2], memdata[7:0], instr[23:16]);
    flopen    #(8)      ir3(clk, irwrite[3], memdata[7:0], instr[31:24]);

    // datapath
    flopenr   #(WIDTH)  pcreg(clk, reset, pcen, nextpc, pc);
    flop     #(WIDTH)  mdr(clk, memdata, md);
    flop     #(WIDTH)  areg(clk, rd1, a);
    flop     #(WIDTH)  wrd(clk, rd2, writedata);
    flop     #(WIDTH)  res(clk, aluresult, aluout);
    mux2     #(WIDTH)  adrmux(pc, aluout, iord, adr);
    mux2     #(WIDTH)  src1mux(pc, a, alusrca, src1);
    mux4     #(WIDTH)  src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],
        constx4, alusrcb, src2);

```

```
    mux4          #(WIDTH)    pcmux(alurestult, aluout, constx4, CONST_ZERO,
pcsource, nextpc);
    mux2          #(WIDTH)    wdmux(aluout, md, memtoreg, wd);
    regfile      #(WIDTH,REGBITS) rf(clk, regwrite, ra1, ra2, wa, wd, rd1,
rd2);
    alu          #(WIDTH)    alunit(src1, src2, alucont, alurestult);
    zerodetect   #(WIDTH)    zd(alurestult, zero);
endmodule
```

```

module alu #(parameter WIDTH = 8)
    (input      [WIDTH-1:0] a, b,
     input      [2:0]      alucont,
     output reg [WIDTH-1:0] result);

    wire      [WIDTH-1:0] b2, sum, slt;

    assign b2 = alucont[2] ? ~b:b;
    assign sum = a + b2 + alucont[2];
    // slt should be 1 if most significant bit of sum is 1
    assign slt = sum[WIDTH-1];

    always@(*)
        case(alucont[1:0])
            2'b00: result <= a & b;
            2'b01: result <= a | b;
            2'b10: result <= sum;
            2'b11: result <= slt;
        endcase
endmodule

```

```

module regfile #(parameter WIDTH = 8, REGBITS = 3)
    (input          clk,
     input          regwrite,
     input  [REGBITS-1:0] ra1, ra2, wa,
     input  [WIDTH-1:0]  wd,
     output [WIDTH-1:0]  rd1, rd2);

    reg  [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];

    // three ported register file
    // read two ports combinatorially
    // write third port on rising edge of clock
    // register 0 hardwired to 0
    always @(posedge clk)
        if (regwrite) RAM[wa] <= wd;

    assign rd1 = ra1 ? RAM[ra1] : 0;
    assign rd2 = ra2 ? RAM[ra2] : 0;
endmodule

module zerodetect #(parameter WIDTH = 8)
    (input  [WIDTH-1:0] a,
     output          y);

    assign y = (a==0);
endmodule

module flop #(parameter WIDTH = 8)
    (input          clk,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(posedge clk)
        q <= d;
endmodule

```

```

module flopen #(parameter WIDTH = 8)
    (input          clk, en,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(posedge clk)
        if (en) q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
    (input          clk, reset, en,
     input  [WIDTH-1:0] d,
     output reg [WIDTH-1:0] q);

    always @(posedge clk)
        if      (reset) q <= 0;
        else if (en)    q <= d;
endmodule

module mux2 #(parameter WIDTH = 8)
    (input  [WIDTH-1:0] d0, d1,
     input          s,
     output [WIDTH-1:0] y);

    assign y = s ? d1 : d0;
endmodule

module mux4 #(parameter WIDTH = 8)
    (input  [WIDTH-1:0] d0, d1, d2, d3,
     input  [1:0]      s,
     output reg [WIDTH-1:0] y);

    always @(*)
        case(s)
            2'b00: y <= d0;
            2'b01: y <= d1;
            2'b10: y <= d2;
            2'b11: y <= d3;
        endcase
endmodule

```