```
//                          -*- Mode: Verilog -*-
// Filename        : controller.v
// Description     : Description of the controller logic for our simplified MIPS processor.
// Author          : Pallav Gupta
// Created On      : Thu Oct 23 20:34:04 2008
// Last Modified On: Time-stamp: <2009-05-05 17:17:50 pgupta>
// Update Count    : 0
// Status          : Unknown, Use with caution!

// This module describes a controller for a multicycle MIPS processor like that
// given in Patterson and Hennessy. It contains the control FSM and the
// additional AND/OR logic for branching. It does not contain the alucontrol
// logic.


// Using busses would be cleaner, but is not supported well by the Electric
// Silicon Compiler.

// reference time unit is 1 nanosecond
`timescale 1ns/1ps

module controller(/*AUTOARG*/
   // Outputs
   memread, memwrite, irwrite3, irwrite2, irwrite1, irwrite0, pcen, regwrite,
   aluop1, aluop0, alusrca, alusrcb1, alusrcb0, pcsource1, pcsource0, iord,
   memtoreg, regdst,
   // Inputs
   ph1, ph2, reset, op5, op4, op3, op2, op1, op0, zero
   );

   input ph1, ph2;
   input reset;
   input op5, op4, op3, op2, op1, op0;
   input zero;
   output reg memread;
   output reg memwrite;
   output reg irwrite3, irwrite2, irwrite1, irwrite0;
   output     pcen;
   output reg regwrite;
   output reg aluop1, aluop0;
   output reg alusrca;
   output reg alusrcb1, alusrcb0;
   output reg pcsource1, pcsource0;
```

```
output reg iord;
output reg memtoreg;
output reg regdst;

// multicycle state machine state definitions
parameter    FETCH1 = 4'b0000;  // instruction fetch (4 cycles, 8-bit datapath)
parameter    FETCH2 = 4'b0001;
parameter    FETCH3 = 4'b0010;
parameter    FETCH4 = 4'b0011;
parameter    DECODE = 4'b0100;  // instruction decode
parameter    MEMADR = 4'b0101;  // memory address computation
parameter    LBRD   = 4'b0110;  // load byte read
parameter    LBWR   = 4'b0111;  // load byte writeback
parameter    SBWR   = 4'b1000;  // store writeback
parameter    RTYPEEX= 4'b1001;  // R-type instruction execution
parameter    RTYPEWR= 4'b1010;  // R-type instruction writeback
parameter    BEQEX  = 4'b1011;  // branch on equal execution
parameter    JEX    = 4'b1100;  // jump execution
// add extra states for ADDI here, use unique code (continue above sequence)
parameter    ADDIEX = 4'b1101;
parameter    ADDIWR = 4'b1110;

// instruction opcodes (do not modify!)
parameter    LB       = 6'b100000;
parameter    SB       = 6'b101000;
parameter    RTYPE    = 6'b000000;
parameter    BEQ      = 6'b000100;
parameter    J        = 6'b000010;
parameter    ADDI     = 6'b001000;

// internal signals
reg [3:0] nextstate_s2;
reg [3:0] state_s1, state_s2;
reg       pcwrite, pcwritecond;

// update state regiter (comprised of master/slave latch)
always @(/*AUTOSENSE*/nextstate_s2 or ph2) // update master latch
  if (ph2) state_s1 = nextstate_s2;
always @(/*AUTOSENSE*/ph1 or state_s1) // update slave latch
  if (ph1) state_s2 = state_s1;

// next state logic
always @(/*AUTOSENSE*/op0 or op1 or op2 or op3 or op4 or op5 or reset
```

```
      or state_s2)
  // state transition figure is given in the write up
  if (reset) nextstate_s2 = FETCH1; // synchronous reset
  else case (state_s2)
      // fetch
      FETCH1: nextstate_s2 = FETCH2;
      FETCH2: nextstate_s2 = FETCH3;
      FETCH3: nextstate_s2 = FETCH4;
      FETCH4: nextstate_s2 = DECODE;

      // decode, look at opcode
      DECODE: case ({op5, op4, op3, op2, op1, op0})
          LB: nextstate_s2 = MEMADR;
          SB: nextstate_s2 = MEMADR;
          RTYPE: nextstate_s2 = RTYPEEX;
          BEQ: nextstate_s2 = BEQEX;
          J: nextstate_s2 = JEX;
          ADDI: nextstate_s2 = ADDIEX;
          default: nextstate_s2 = FETCH1;
        endcase // case ({op5, op4, op3, op2, op1, op0})

      // memory adress compute (for load/store)
      MEMADR: case ({op5, op4, op3, op2, op1, op0}) // synopsys full_case
          LB: nextstate_s2 = LBRD;
          SB: nextstate_s2 = SBWR;
          // no default needed becaus of full_case directive
        endcase // case ({op5, op4, op3, op2, op1, op0})

      LBRD: nextstate_s2 = LBWR;
      LBWR: nextstate_s2 = FETCH1;
      SBWR: nextstate_s2 = FETCH1;
      RTYPEEX: nextstate_s2 = RTYPEWR;
      RTYPEWR: nextstate_s2 = FETCH1;
      BEQEX: nextstate_s2 = FETCH1;
      JEX: nextstate_s2 = FETCH1;
      ADDIEX: nextstate_s2 = ADDIWR;
        ADDIWR: nextstate_s2 = FETCH1;

        default: nextstate_s2 = FETCH1;
       endcase

// output logic
always @(/*AUTOSENSE*/state_s2)
```

```
begin
  // provide default values for signals not specified
  memread = 0;
  memwrite = 0;
  irwrite3 = 0; irwrite2 = 0; irwrite1 = 0; irwrite0 = 0;
  pcwrite = 0;
  pcwritecond = 0;
  regwrite = 0;
  alusrca = 0;
  alusrcb1 = 0; alusrcb0 = 0;
  aluop1 = 0; aluop0 = 0;
  pcsource1 = 0; pcsource0 = 0;
  iord = 0;
  memtoreg = 0;
  regdst = 0;

  // specify the outputs for reach state according to FSM
  case (state_s2)
   FETCH1: begin
      memread = 1;
      alusrca = 0;
      iord = 0;
      irwrite3 = 1; // endianness
      alusrcb1 = 0; alusrcb0 = 1;
         aluop1 = 0; aluop0 = 0;
         pcwrite = 1;
         pcsource1 = 0; pcsource0 = 0;
       end
   FETCH2: begin
         memread = 1;
         alusrca = 0;
         iord = 0;
         irwrite2 = 1; // endianness
         alusrcb1 = 0; alusrcb0 = 1;
         aluop1 = 0; aluop0 = 0;
         pcwrite = 1;
         pcsource1 = 0; pcsource0 = 0;
       end
       FETCH3: begin
         memread = 1;
         alusrca = 0;
         iord = 0;
         irwrite1 = 1; // endianness
```

```
  alusrcb1 = 0; alusrcb0 = 1;
  aluop1 = 0; aluop0 = 0;
  pcwrite = 1;
  pcsource1 = 0; pcsource0 = 0;
end
FETCH4: begin
  memread = 1;
  alusrca = 0;
  iord = 0;
  irwrite0 = 1; // endianness
  alusrcb1 = 0; alusrcb0 = 1;
  aluop1 = 0; aluop0 = 0;
  pcwrite = 1;
  pcsource1 = 0; pcsource0 = 0;
end
DECODE: begin
  alusrca = 0;
  alusrcb1 = 1; alusrcb0 = 1;
  aluop1 = 0; aluop0 = 0;
end
MEMADR: begin
  alusrca = 1;
  alusrcb1 = 1; alusrcb0 = 0;
  aluop1 = 0; aluop0 = 0;
end
LBRD: begin
  memread = 1;
  iord = 1;
end
LBWR: begin
  regdst = 0;
  regwrite = 1;
  memtoreg = 1;
end
SBWR: begin
  memwrite = 1;
  iord = 1;
end
RTYPEEX: begin
  alusrca = 1;
  alusrcb1 = 0; alusrcb0 = 0;
  aluop1 = 1; aluop0 = 0;
end
```

```
            RTYPEWR: begin
              regdst = 1;
              regwrite = 1;
              memtoreg = 0;
            end
            BEQEX: begin
              alusrca = 1;
              alusrcb1 = 0; alusrcb0 = 0;
              aluop1 = 0; aluop0 = 1;
              pcwritecond = 1;
              pcsource1 = 0; pcsource0 = 1;
            end
          JEX: begin
              pcwrite = 1;
              aluop0 = 1; // not logically required, but a hack to ensure aluop0
                    // and pcsource0 aren't always identical.  If they
                    // were identical, Synopsys would optimize one away,
                    // which confuses the Silicon Compiler.
            pcsource1 = 1; pcsource0 = 0;
            end
            ADDIEX: begin
              alusrca = 1;
              alusrcb1 = 1; alusrcb0 = 0;
              aluop1 = 0; aluop0 = 0;
            end
            ADDIWR: begin
              regdst = 0;
              regwrite = 1;
              memtoreg = 0;
            end
            default: begin
            end
        endcase // case (state_s2)
      end // always @ (state_s2)

    // compute pcen, the write enable for the program counter
    assign pcen = pcwrite | (pcwritecond & zero);
endmodule



// Controller testbench: Supplies test vectors and dumps the results to file and
// standard output. Do not modify!
```

```
module controller_tb;
  reg ph1, ph2, reset;
  reg [5:0] opcode;
  reg       zero;
  wire memread;
  wire memwrite;
  wire [3:0] irwrite;
  wire       pcen;
  wire       regwrite;
  wire [1:0] aluop;
  wire       alusrca;
  wire [1:0] alusrcb;
  wire [1:0] pcsource;
  wire       iord;
  wire       memtoreg;
  wire       regdst;
  wire       pcwritecond;
  wire       pcwrite;

  // instatiate device under test and connect the signals
  controller U0(
              // inputs
              .ph1 (ph1),
              .ph2 (ph2),
              .reset (reset),
              .op5 (opcode[5]), .op4 (opcode[4]), .op3 (opcode[3]),
              .op2 (opcode[2]), .op1 (opcode[1]), .op0 (opcode[0]),
              .zero (zero),
              // ouputs
              .memread (memread),
              .memwrite (memwrite),
              .irwrite3 (irwrite[3]), .irwrite2 (irwrite[2]),
              .irwrite1 (irwrite[1]), .irwrite0 (irwrite[0]),
              .pcen (pcen),
              .regwrite (regwrite),
              .aluop1 (aluop[1]), .aluop0 (aluop[0]),
              .alusrca (alusrca),
              .alusrcb1 (alusrcb[1]), .alusrcb0 (alusrcb[0]),
              .pcsource1 (pcsource[1]), .pcsource0 (pcsource[0]),
              .iord (iord),
              .memtoreg (memtoreg),
              .regdst (regdst));
```

```
  // initialization (reset is high)
  initial
    begin
      ph1 <= 0;
      ph2 <= 0;
      reset <= 1;
      zero = 0;
    end

  // generate a two-phase non-overlapping clock (period is 8 units)
  always begin
    #2 ph1 = 1;
    #2 ph1 = 0;
    #4 ph1 = 0;

  end
  always begin
    #6 ph2 = 1;
    #2 ph2 = 0;
  end

  // dump all the signals info file. use with a waveform viewer to see the signals
  initial begin
    $dumpfile("controller.vcd");
    $dumpvars;
  end

  initial begin
    // bring out of reset and supply the first opcode lb
    #2 reset = 0; opcode = 6'b100000;
    $display("%s %s %s %s %s %s %s %s %s %s %s %s", "memread", "alusrca", "iord",
"irwrite",
            "alusrcb", "aluop", "pcen", "pcsource", "regdst", "regwrite",
            "memtoreg", "memwrite");
    $display("opcode = %b", opcode);

    // sb
    #72 opcode = 6'b101000;
    $display("opcode = %b", opcode);

    // r-type instructions
    #56 opcode = 6'b000000;
    $display("opcode = %b", opcode);
```

```
    // beq
    #56 opcode = 6'b000100;
    $display("opcode = %b", opcode);
    #40 zero = 1'b1;      // check that zero works in BEQEX

    // jump
    #8 opcode = 6'b000010;
    $display("opcode = %b", opcode);

    // addi
    #48 opcode = 6'b001000;
    $display("opcode = %b", opcode);

    // terminate simulation
    #56 $finish;
  end

  // print the values of all relevant signals every clock period
  always
    #8 $display("%5b %6b %6b %6b %7b %5b %4b %8b %7b %6b %8b %7b", memread,
alusrca, iord, irwrite, alusrcb, aluop,
                pcen, pcsource, regdst, regwrite, memtoreg, memwrite);
endmodule
```