Department of Electrical and Computer Engineering

Author: Zainab Hussein

Title: Multicycle Processor (Part 1)

Date: 4-11-2017

1. Time spent: Lab period
2. Table 4

State (Name)	PCWrite	MemWrite	IRWrite	RegWrite	ALUSrcA	Branch	IorD	MemtoReg	RegDst	ALUSrcB[1:0]	PCRsc[1:0]	ALUOp[1:0]	FSM Control Word
0 (Fetch)	1	0	1	0	0	0	0	0	0	01	00	00	0x5010
1 (Decode)	0	0	0	0	0	0	0	0	0	11	00	00	0x0030
2 (MemAdr)	0	0	0	0	1	0	0	0	0	10	00	00	0x0420
3 (MemRd)	0	0	0	0	0	0	1	0	0	00	00	00	0x0100
4 (MemWB)	0	0	0	1	0	0	0	1	0	00	00	00	0x0880
5 (MemWr)	0	1	0	0	0	0	1	0	0	00	00	00	0x2100
6 (RtypeEx)	0	0	0	0	1	0	0	0	0	00	00	10	0x0402
7 (RtypeWB)	0	0	0	1	0	0	0	0	1	00	00	00	0x0840
8 (BeqEx)	0	0	0	0	1	1	0	0	0	00	01	01	0x0605
9 (AddiEx)	0	0	0	0	1	0	0	0	0	10	00	00	0x0420
10 (AddiWB)	0	0	0	1	0	0	0	0	0	00	00	00	0x0800
11 (JEx)	1	0	0	0	0	0	0	0	0	00	10	00	0x4008

3. Controller, maindec and aludec code

```
module controller(input  logic       clk, reset,
                  input  logic [5:0] op, funct,
                  input  logic       zero,
                  output logic       pcen, memwrite, irwrite, regwrite,
                  output logic       alusrca, iord, memtoreg, regdst,
                  output logic [1:0] alusrcb, pcsrc,
                  output logic [2:0] alucontrol);
  logic [1:0] aluop;
  logic       branch, pcwrite;
```

```
  assign pcen = pcwrite|(branch&zero);
  // Main Decoder and ALU Decoder subunits.
  maindec md(clk, reset, op,
             pcwrite, memwrite, irwrite, regwrite,
             alusrca, branch, iord, memtoreg, regdst,
             alusrcb, pcsrc, aluop);
  aludec  ad(funct, aluop, alucontrol);
  // ADD CODE HERE
  // Add combinational logic (i.e. an assign statement)
  // to produce the PCEn signal (pcen) from the branch,
  // zero, and pcwrite signals
endmodule
module maindec(input  logic       clk, reset,
               input  logic [5:0] op,
               output logic       pcwrite, memwrite, irwrite, regwrite,
               output logic       alusrca, branch, iord, memtoreg, regdst,
               output logic [1:0] alusrcb, pcsrc,
               output logic [1:0] aluop);
  parameter   FETCH   = 4'b0000;    // State 0
  parameter   DECODE  = 4'b0001;    // State 1
  parameter   MEMADR  = 4'b0010; // State 2
  parameter   MEMRD   = 4'b0011; // State 3
  parameter   MEMWB   = 4'b0100; // State 4
  parameter   MEMWR   = 4'b0101; // State 5
  parameter   RTYPEEX = 4'b0110; // State 6
  parameter   RTYPEWB = 4'b0111; // State 7
  parameter   BEQEX   = 4'b1000; // State 8
  parameter   ADDIEX  = 4'b1001; // State 9
  parameter   ADDIWB  = 4'b1010; // state 10
  parameter   JEX     = 4'b1011; // State 11
  parameter   LW      = 6'b100011;      // Opcode for lw
  parameter   SW      = 6'b101011;      // Opcode for sw
  parameter   RTYPE   = 6'b000000;      // Opcode for R-type
```

```
parameter   BEQ     = 6'b000100;      // Opcode for beq
parameter   ADDI    = 6'b001000;      // Opcode for addi
parameter   J       = 6'b000010;      // Opcode for j
logic [3:0]  state, nextstate;
logic [14:0] controls;
// state register
always_ff @(posedge clk or posedge reset)
  if(reset) state <= FETCH;
  else state <= nextstate;
// ADD CODE HERE
// Finish entering the next state logic below.  We've completed the first
// two states, FETCH and DECODE, for you.
// next state logic
always_comb
  case(state)
    FETCH:   nextstate <= DECODE;
    DECODE:  case(op)
               LW:      nextstate <= MEMADR;
               SW:      nextstate <= MEMADR;
               RTYPE:   nextstate <= RTYPEEX;
               BEQ:     nextstate <= BEQEX;
               ADDI:    nextstate <= ADDIEX;
               J:       nextstate <= JEX;
               default: nextstate <= 4'bx; // should never happen
             endcase
            // Add code here
    MEMADR:   case(op)
                  LW:         nextstate <= MEMRD;
                  SW:         nextstate <= MEMWR;
                  default:    nextstate <= 4'bx; // should never happen
              endcase
    MEMRD:    nextstate <= MEMWB;
    MEMWB:    nextstate <= FETCH;
    MEMWR:    nextstate <= FETCH;
```

```
      RTYPEEX:  nextstate <= RTYPEWB;
      RTYPEWB:  nextstate <= FETCH;
      BEQEX:    nextstate <= FETCH;
      ADDIEX:   nextstate <= ADDIWB;
      ADDIWB:   nextstate <= FETCH;
      JEX:      nextstate <= FETCH;
      default: nextstate <= 4'bx; // should never happen
    endcase
  // output logic
  assign {pcwrite, memwrite, irwrite, regwrite,
          alusrca, branch, iord, memtoreg, regdst,
          alusrcb, pcsrc, aluop} = controls;
  // ADD CODE HERE
  // Finish entering the output logic below.  We've entered the
  // output logic for the first two states, S0 and S1, for you.
  always_comb
    case(state)
      FETCH:   controls <= 15'h5010;
      DECODE:  controls <= 15'h0030;
    // your code goes here
      MEMADR:  controls <= 15'h0420;
      MEMRD:   controls <= 15'h0100;
      MEMWB:   controls <= 15'h0880;
      MEMWR:   controls <= 15'h2100;
      RTYPEEX: controls <= 15'h0402;
      RTYPEWB: controls <= 15'h0840;
      BEQEX:   controls <= 15'h0605;
      ADDIEX:  controls <= 15'h0420;
         ADDIWB:  controls <= 15'h0800;
         JEX:     controls <= 15'h4008;
      default: controls <= 15'hxxxx; // should never happen
    endcase
endmodule
module aludec(input  logic [5:0] funct,
```

```
              input  logic [1:0] aluop,
              output logic [2:0] alucontrol);
  // ADD CODE HERE
  // Complete the design for the ALU Decoder.
  // Your design goes here.  Remember that this is a combinational
  // module.
  // Remember that you may also reuse any code from previous labs.
  // beq, addi, j
    always_comb
      case(aluop)
        2'b00: alucontrol <= 3'b010;  // add, lw, sw
        2'b10: alucontrol <= 3'b110;  // sub
        2'b01: alucontrol <= 3'b110;  // beq
        2'b11: alucontrol <= 3'b001;  // ori  //ADDED ORI CONTROL SIGNAL
        default: case(funct)          // RTYPE
            6'b100000: alucontrol <= 3'b010; // ADD
            6'b100010: alucontrol <= 3'b110; // SUB
            6'b100100: alucontrol <= 3'b000; // AND
            6'b100101: alucontrol <= 3'b001; // OR
            6'b101010: alucontrol <= 3'b111; // SLT
            default:   alucontrol <= 3'bxxx; // ???
          endcase
      endcase
endmodule
```

4. Controllertest testbench

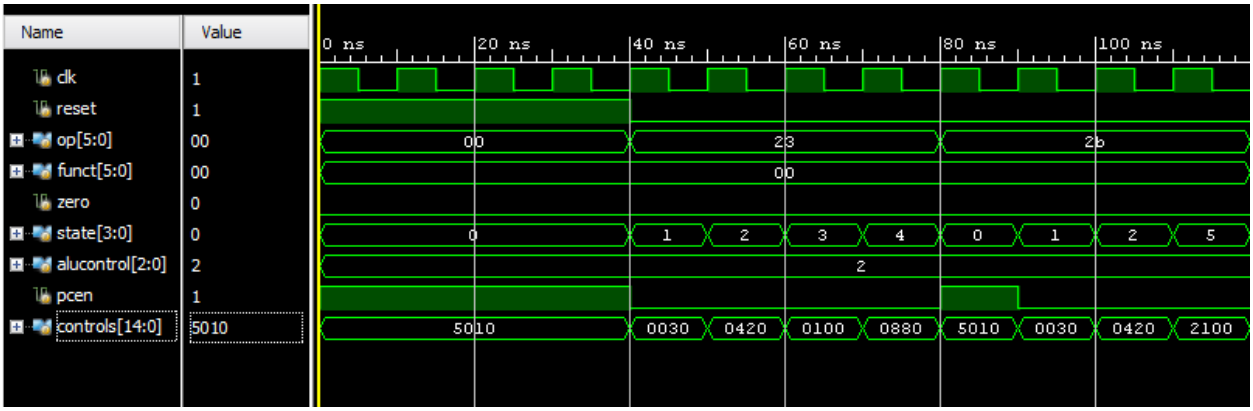
```
module controllertest();
    //inputs
    logic       clk, reset;
    logic [5:0] op, funct;
    logic       zero;
    //outputs
    logic       pcen, memwrite, irwrite, regwrite;
    logic       alusrca, iord, memtoreg, regdst;
```

```
logic [1:0] alusrcb, pcsrc;
logic [2:0] alucontrol;

// instantiate device to be tested
controller dut(clk, reset, op, funct, zero,
               pcen, memwrite, irwrite, regwrite,
               alusrca, iord, memtoreg, regdst,
               alusrcb, pcsrc,alucontrol);
// generate clock to sequence tests
always
    begin
      clk <= 1; # 5; clk <= 0; # 5;
    end
// initializet est
initial begin
    reset= 1 ;
    op= 6 'b000000;
    zero =0 ;
    funct= 6 'b000000;
    //lw test
    @(posedge clk)#40;
    begin
        reset= 0 ;
        op = 6 'b100011;
        zero= 0 ;
        funct= 6 'b000000;
    end
    //sw test
    @(posedge clk)#40;
    begin
        reset= 0 ;
        op = 6 'b101011;
        zero= 0 ;
        funct= 6 'b000000;
```

```
end

//add test
@(posedge clk)#40;
begin
    reset= 0 ;
    op = 6 'b000000;
    zero= 0 ;
    funct= 6 'b100000;
end
//sub test
@(posedge clk)#40;
begin
    reset= 0 ;
    op = 6 'b000000;
    zero= 0 ;
    funct= 6 'b100010;
end
//or test
@ (posedge clk)#40;
begin
    reset =0 ;
    op= 6 'b000000;
    zero= 0 ;
    funct =6 'b100101;
end
//slt test
@(posedge clk)#40;
begin
    reset =0 ;
    op= 6 'b000000;
    zero= 0 ;
    funct =6 'b101010;
end
```

```
        //beq test
        @(posedge clk)#40;
        begin
            reset= 0 ;
            op = 6 'b000100;
            zero= 1 ;
            funct= 6 'b000000;
        end
        //addi test
        @(posedge clk)#40;
        begin
            reset= 0 ;
            op = 6 'b001000;
            zero= 0 ;
            funct= 6 'b000000;
        end
        //j test
        @(posedge clk)#40;
        begin
            reset= 0 ;
            op = 6 'b000010;
            zero= 0 ;
            funct= 6 'b000000;
        end
    end
endmodule
```

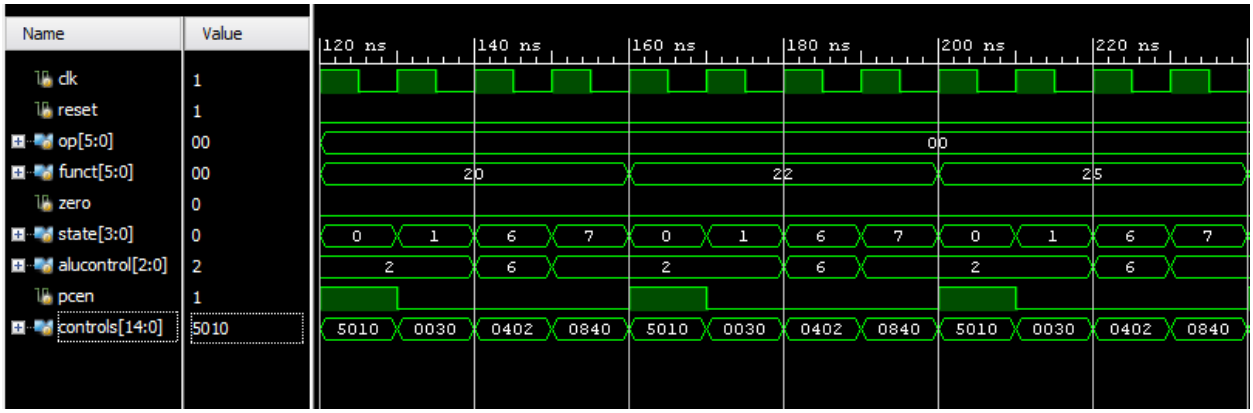
5. Simulation waveforms



Expected:

lw => Fetch, Decode, MemAddr, MemRead, MemWriteback (states 0,1,2,3,4)

Sw => Fetch, Decode, MemAddr, MemWrite (states 0,1,2,5)

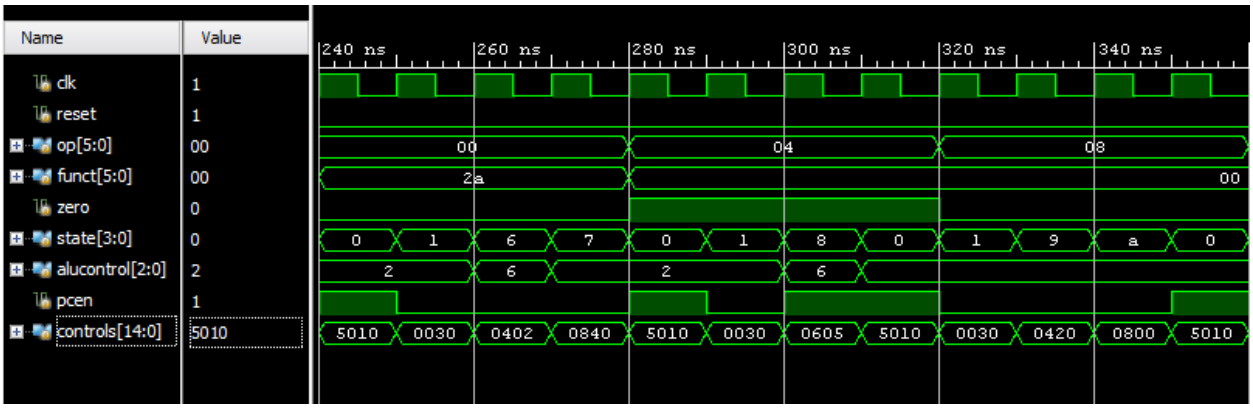


Expected:

add => Fetch, Decode, RtypeEx, RtypeWb (states 0,1,6,7 & funct 20)

sub => Fetch, Decode, RtypeEx, RtypeWb (states 0,1,6,7,0 & funct 22)

or => Fetch, Decode, RtypeEx, RtypeWb (states 0,1,6,7,0 & funct 25)

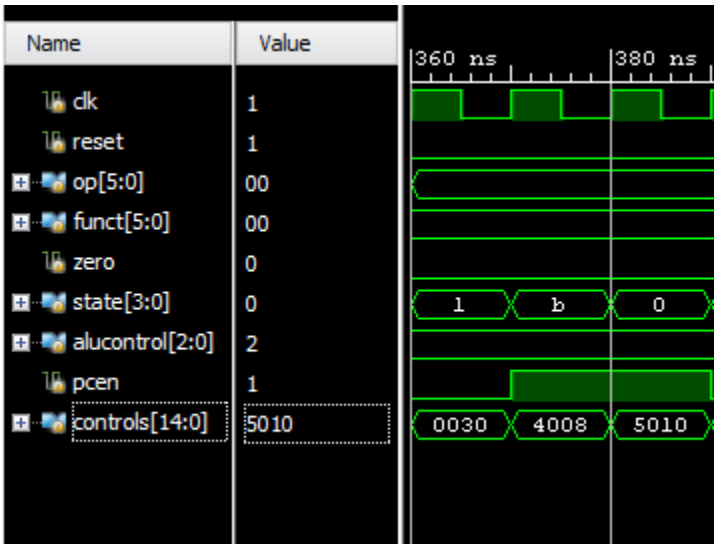


Expected:

slt => Fetch, Decode, RtypeEx, RtypeWb (states 0,1,6,7)

beq => Fetch, Decode, Branch (states 0,1,8)

addi => Fetch, Decode, AddiEx, AddiWb (states 0,1,9,a)



Expected:

j => Fetch, Decode, Jump (states 0,1,b)

Yes, all the simulation matches the expected values outlined in the captions, which correspond to the table 4 and processor state diagram.