MIPS with Pipeline and Data Cache

Aniqa Abid, Nathan Pen, Yimin Wang

ECE 6213 Final Report

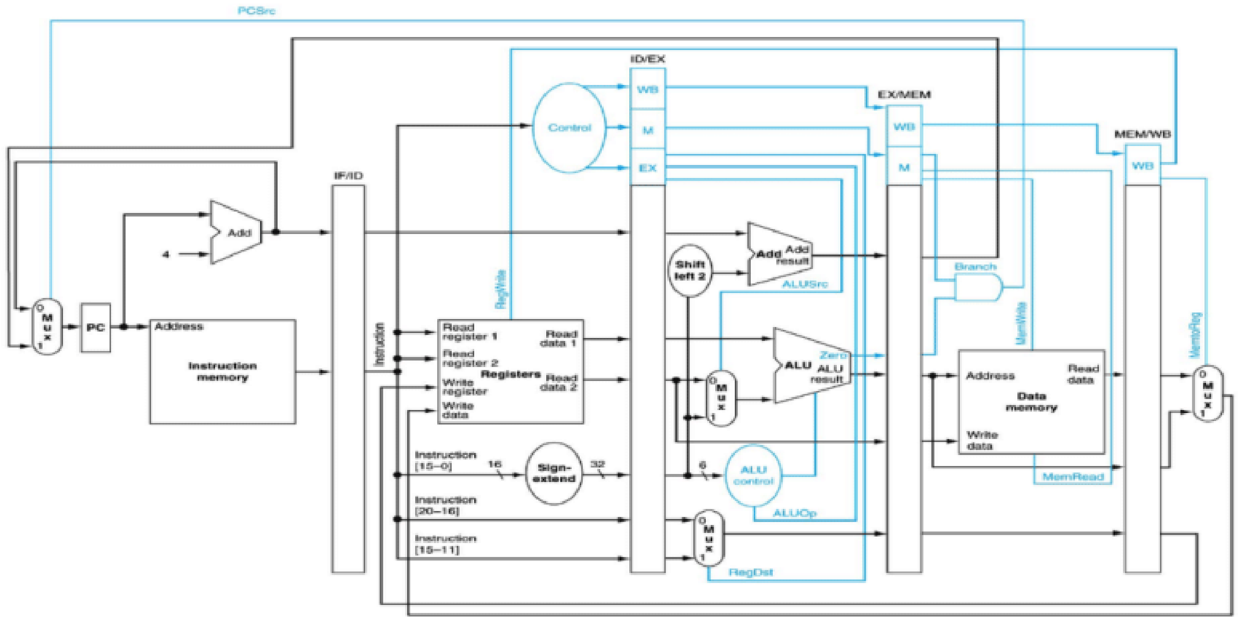
December 18, 2022

Introduction

The purpose of this project is to implement the Pipelined MIPS CPU with Data Cache Architecture in Verilog code, synthesize the codes and implement the code on the FPGA board.The team spent numerous hours debugging the project until we could get a working product that was synthesizable and able to be loaded onto an Altera FPGA board with function inputs and outputs.

Architectural

a. Block Diagram



b. Assembly instructions the processor implement

i. Load Word
ii. Store word
iii. ADDi
iv. Branch if Equal
v. R Type Add
vi. R Type Subtract
vii. R Type Logical AND
viii. R Type Logical OR
ix. R Type Logical NOR
x. R Type Set Less Than

Verilog Code

a. Controller

```
module controller(alusrc, aluop,
                  memread, memwrite, memtoreg,
                 branch, regwrite, regdst,
                  op, clk, reset) ;
      //12/1 need to finalize opcodes and corresponding outputs
   // INPUTS (3)
```

```
input  [5:0]   op         ;       // OPCODE
input          clk, reset;
//input       zero;

// OUTPUTS (12)
output         alusrc    ;// ALUSrcA
output [1:0]   aluop      ;       // ALUOp
output         memread    ;       // MemRead
output         memwrite   ;       // MemWrite
output         memtoreg   ;       // MemtoReg
//output      pcsource   ; // PCSrc
output         regwrite   ;       // RegWrite
output        regdst     ; // RegDst
output        branch     ;

// REGISTERS
reg           alusrc     ;
reg [1:0]      aluop      ;       // ALUOp
reg            branch     ;       // Branch
reg            memread    ;       // MemRead
reg            memwrite   ;       // MemWrite
reg            memtoreg   ;       // MemtoReg
reg            regwrite   ;       // RegWrite
reg           regdst     ; // RegDst



// OPCODES  -- Make input decoding simpler
//
parameter      LW      =  6'b100011;     // load word
parameter      SW      =  6'b101011; // store word
parameter      RTYPE   =  6'b0; // Regsiter type instuction TODO - see lecture 4b
parameter      BEQ     =  6'b000100; // Branch if Equal instruction i-type TODO
parameter      J       =  6'b000010; //  Jump, not used in this
parameter     ADDI    =  6'b001000; // addi op

always @(op or reset)
   begin

     if(reset)
     begin
            alusrc <= 0;
            aluop <= 2'b00;
            memread <= 0;
            memwrite <= 0;
            memtoreg <= 0;
            regwrite <= 0;
            regdst <= 0;
            branch <= 0;
     end

     else
```

```
begin
case(op)

LW:
   begin
        alusrc <= 1;
       aluop <= 2'b00;
       memread <= 1;
       memwrite <= 0;
       memtoreg <= 1;
       regwrite <= 1;
       regdst <= 0;
       branch <= 0;
   end
SW:
   begin
        alusrc <= 1;
       aluop <= 2'b00;
       memread <= 0;
       memwrite <= 1;
       memtoreg <= 0;
       regwrite <= 0;
       //regdst <= 1'bx;
       branch <= 0;
   end
RTYPE:
   begin
        alusrc <= 0;
       aluop <= 2'b10;
       memread <= 0;
       memwrite <= 0;
       memtoreg <= 0;
       regwrite <= 1;
       regdst <= 1;
       branch <= 0;
   end
BEQ:
   begin
        alusrc <= 0;
       aluop <= 2'b01;
       memread <= 0;
       memwrite <= 0;
       memtoreg <= 0;
       regwrite <= 1;
       //regdst <= 1'bx;
       branch <= 1;
   end
J: //not used
   begin
        alusrc <= 0;
       aluop <= 2'b10;
       memread <= 0;
```

```
                memwrite <= 0;
                memtoreg <= 0;
                regwrite <= 1;
                regdst <= 1;
                branch <= 0;
            end
         ADDI:
            begin
                 alusrc <= 1;
                aluop <= 2'b11;
                memread <= 0;
                memwrite <= 0;
                memtoreg <= 0;
                regwrite <= 1;
                regdst <= 0;
                branch <= 0;
            end
         endcase
         end
       end

endmodule
```

b. Instruction Memory & Regfile

```
module imem(clk, adr,memdata,rst);

   input clk,rst;
   input [31:0] adr;
   output reg [31:0] memdata;
   reg [31:0] mips_ram [0:255];
   reg [31:0] temp;
   integer i;

   always@(posedge rst)
       begin
              for(i=0; i<256; i=i+1)
              begin
                     mips_ram[i]=32'b0;
              end
              mips_ram[0] = 32'h01095020;
              mips_ram[1] = 32'hAC0A8000;
              mips_ram[2] = 32'h8EB10000;
              mips_ram[3] = 32'h20040001;
              mips_ram[4] = 32'h2005ffff;
       end
   always@(posedge clk)
       begin
              memdata <= mips_ram[adr>>2];
       end

endmodule
```

```
module regfile #(parameter WIDTH = 32, REGBITS = 5)
                (input                clk, rst,
                 input                regwrite,
                 input  [REGBITS-1:0] ra1, ra2, wa,
                 input  [WIDTH-1:0]   wd,
                 output reg [WIDTH-1:0]   rd1, rd2);

   reg  [31:0] memory [(1<<REGBITS)-1:0];

   // three ported register file
   // read two ports combinationally
   // write third port on rising edge of clock
   // register 0 hardwired to 0

   always @(posedge clk or posedge rst)
   begin
       if(rst)
       begin
              memory[0] <= 32'h00000000;
              memory[1] <= 32'h00000000;
              memory[2] <= 32'h00000000;
              memory[3] <= 32'h00000000;
              memory[4] <= 32'h00000000;
              memory[5] <= 32'h00000000;
              memory[6] <= 32'h00000000;
              memory[7] <= 32'h00000000;
              memory[8] <= 32'h00000001;
              memory[9] <= 32'h00000002;
              memory[10] <= 32'h00000000;
              memory[11] <= 32'h00000000;
              memory[12] <= 32'h00000000;
              memory[13] <= 32'h00000000;
              memory[14] <= 32'h00000000;
              memory[15] <= 32'h00000000;
              memory[16] <= 32'h00000000;
              memory[17] <= 32'h00000000;
              memory[18] <= 32'h00000003;
              memory[19] <= 32'h00000003;
              memory[20] <= 32'h00000004;
              memory[21] <= 32'h00000000;
              memory[22] <= 32'h00000008;
              memory[23] <= 32'h00000000;
              memory[24] <= 32'h00000000;
              memory[25] <= 32'h00000000;
              memory[26] <= 32'h00000000;
              memory[27] <= 32'h00000000;
              memory[28] <= 32'h00000000;
              memory[29] <= 32'h00000000;
              memory[30] <= 32'h00000000;
              memory[31] <= 32'h00000000;
       end
```

```
        else if(regwrite)
        begin
               memory[wa] <= wd;
        end
   end

   always @(ra1,ra2)
   begin
        rd1 = memory[ra1];
        rd2 = memory[ra2];
   end
endmodule
```

c. ALU

```
module alu #(parameter WIDTH = 32)
            (input      [WIDTH-1:0] a, b,
             input      [3:0]       alucont,
             output reg [WIDTH-1:0] result,
            output zero);

   wire     [WIDTH-1:0] sum, slt;
   assign zero = (alucont==0); //Zero is true if aluout is 0;
   // slt should be 1 if most significant bit of sum is 1

   always@(*)
      case(alucont)
         4'b0000: result <= a & b;
         4'b0001: result <= a | b;
         4'b0010: result <= a + b;
         4'b0110: result <= a - b;
        4'b0111: result <= a < b ? 1:0;
        4'b1100: result <= ~(a | b); //result is nor
        default: result <= 0; //default to 0 which shouldn't happen
      endcase
endmodule
```

d. ALU Control

```
module alucontrol( alucont, aluop, funct ) ;

// decodes the 'funct' field from the assembly instruction
// determines the type of instruction it is: R, I, J
// creates a 3-bit control line (alucont) for the ALU itself.

   input  [1:0] aluop   ;
   input  [5:0] funct   ;
   output reg [3:0] alucont ;
   always @(*)
      case(aluop)
         2'b00: alucont <= 4'b0010;  // add for lw/sw
         2'b01: alucont <= 4'b0110;  // sub (for beq)
         2'b11: alucont <= 4'b0010;  // addi
         2'b10: case(funct)       // R-Type instructions
                     6'b100000: alucont <= 4'b0010; // add (for add)
                     6'b100010: alucont <= 4'b0110; // subtract (for sub)
                     6'b100100: alucont <= 4'b0000; // logical and (for and)
                     6'b100101: alucont <= 4'b0001; // logical or (for or)
                      6'b100111: alucont <= 4'b1100; // nor
                     6'b101010: alucont <= 4'b0111; // set on less (for slt)
                     default:   alucont <= 4'b0101; // should never happen
                endcase
       default: alucont <= 4'b1111; //shouldn't happen
      endcase

endmodule
```

e. Sign Extension

```
module signextend(in ,out);

input  [15:0] in;
output [31:0] out;

assign out =  (in[15] == 1)? {16'hffff , in} :
              (in[15] == 0)? {16'h0000 ,in}  : 16'hxxxx;

endmodule
```

f. Shifter

```
module shiftleft2(out, in);

input [31:0]in;
output [31:0]out;

assign out = in << 2;

endmodule
```

g. Adder

```
module adder(out, in1, in2);

input [31:0] in1,in2;
output [31:0] out;

assign out = in1 + in2;
```

```
endmodule
```

h. MUX

```
module mux2_5(input  [4:0] d0, d1,
              input        s,
              output [4:0] y);

   assign y = s ? d1 : d0;
endmodule

module mux2 #(parameter WIDTH = 32)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```

i. PC

```
module PC (nextpc,outpc,reset,enable,clk);
       input [31:0] nextpc;
       input reset,enable,clk;
       output reg [31:0] outpc;

       always@(posedge clk or posedge reset)
       begin
              if(reset)
              begin
                     outpc <= 32'b0;
              end
              else if(enable == 1)
              begin
                     outpc <= nextpc;
              end
              else
              begin
                     outpc <= outpc;
              end
       end
endmodule
```

j. Data Cache

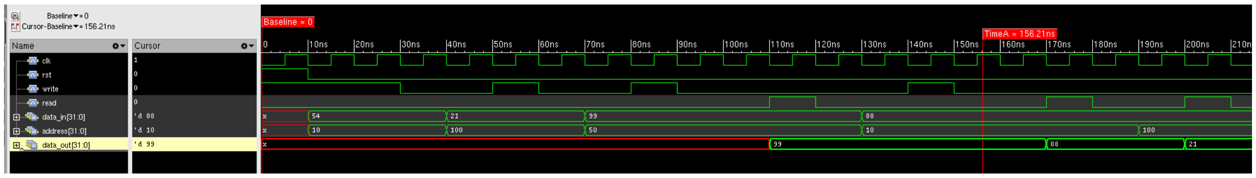
The group designed three different types of data cache in the process and finally decided to implement a write through data cache. The first idea was to design a 13-stage finite state machine. Then the group reduced the size of the structure by deleting some of the stages. However after finishing the 5-stage finite state machine code, the group found out there were too many triple driving errors in the code. The most time-saving method is to implement a write-through data cache.

```
`timescale 1ns / 1ps
module datacache(address, data_in, clk, rst, write, read, temp_out);

input [31:0] address;// 24-bit tag, 4-bit offset
input [31:0] data_in;
input clk,rst;
input write,read;
output [31:0] temp_out;

reg [31:0] pre_adr = 0;
reg [31:0] pre_data = 0;
reg [31:0] temp_out;

reg [31:0] cache_arr [0:15];
reg [31:0] mem_arr [0:255];
reg [0:0] v_arr [0:15];
reg [3:0] t_arr [0:15];

integer i;

always @(*)
begin
       if(rst)
       begin
              for(i = 0; i<16; i = i+1)
              begin
                     v_arr[i] <= 1'b0;
                     t_arr[i] <= 5'b00000;
              end
       end
       else if(!rst && write)////write, which means sw
       begin
              begin
                     mem_arr[pre_adr] <= data_in;
                     v_arr[{pre_adr[7:4]}] <= 1;
                     t_arr[{pre_adr[7:4]}] <= {pre_adr[31:8]};
                     cache_arr[{pre_adr[7:4]}] <= pre_data;
              end
       end
       else if(!rst && read)//read, which means lw
       begin
              begin
              if(v_arr[{pre_adr[7:4]}] == 1)
                     begin
                            if( t_arr[{pre_adr[7:4]}] == {pre_adr[31:8]})
                            temp_out <= cache_arr[{pre_adr[7:4]}];
                            else
                            begin
                                   cache_arr[{pre_adr[7:4]}] <= mem_arr[pre_adr];
```

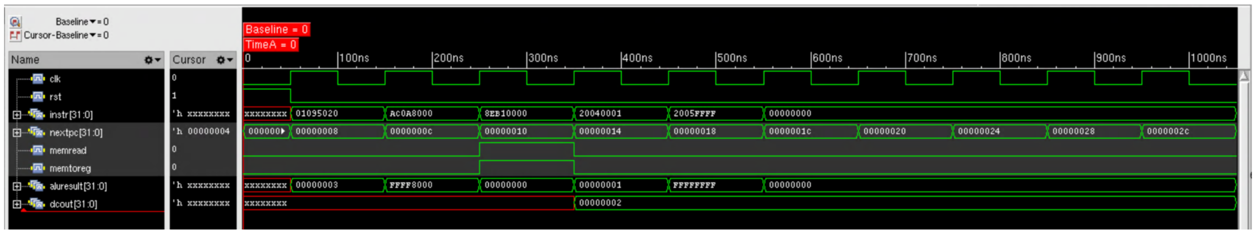
```
                    temp_out <= mem_arr[pre_adr];
                end
            end
        else
            begin
                cache_arr[{pre_adr[7:4]}] <= mem_arr[pre_adr];
                temp_out <= mem_arr[pre_adr];
            end

        end
    end
    else
    begin
        pre_adr <= address;
        pre_data <= data_in;
    end
end
endmodule
```

Simulation

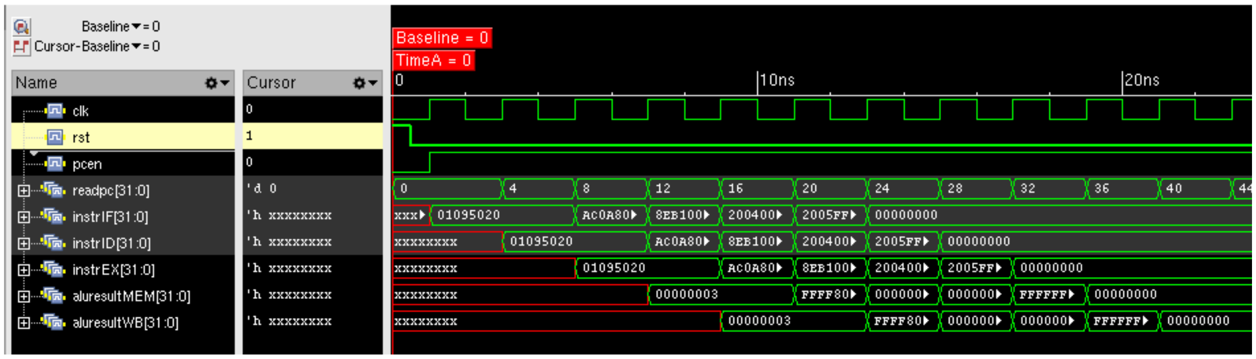
a. Data Cache



b. Non-pipelined MIPS with data cache



c. Pipelined MIPS with data cache



FPGA

a. Clock Divider

This module was created so the students could see their I/O pins on the board at a slower frequency, this allowed the students to troubleshoot better and made visualization easier.

```
module clkdiv
#(
parameter WIDTH = 32, // Width of the register required
parameter N = 100000000// We will divide by 10 for example in this case
)
(clk,reset,clk_out);

input clk;
input reset;
output reg clk_out;

reg [WIDTH-1:0] count;
//wire [WIDTH-1:0] r_nxt;
//reg clk_track;

always @(posedge clk or posedge reset)

begin
  if (reset)
     begin
        count <= 32'b0;
                 clk_out <= 1'b0;
     end

  else if (count == N - 1)
          begin
            count <= 0;
            clk_out <= ~clk_out;
          end
  else
              begin
                     count <= count + 1;
                     clk_out <= clk_out;
              end
end
endmodule
```

b. 7-segment Display

```
module bintobcd
 #( parameter                W = 8)  // input width
  ( input      [W-1      :0] bin   ,  // binary
    output reg [W+(W-4)/3:0] bcd   ); // bcd {...,thousands,hundreds,tens,ones}

  integer i,j;
  always @(bin) begin
    for(i = 0; i <= W+(W-4)/3; i = i+1) bcd[i] = 0;     // initialize with zeros
    bcd[W-1:0] = bin;                                   // initialize with input vector
    for(i = 0; i <= W-4; i = i+1)                       // iterate on structure depth
      for(j = 0; j <= i/3; j = j+1)                     // iterate on structure width
        if (bcd[W-i+4*j -: 4] > 4)                      // if > 4
          bcd[W-i+4*j -: 4] = bcd[W-i+4*j -: 4] + 4'd3; // add 3
  end

endmodule


module sevenseg(readpc,segTens,segOnes,reset);
       input [6:0] readpc;
       output reg [6:0] segTens;
       output reg [6:0] segOnes;
       input reset;
       wire [7:0] bcd;


       bintobcd btod(.bin(readpc),.bcd(bcd));

       always @(bcd)

       begin
       if(reset)
       begin
              segOnes = 7'b0000000;
              segTens = 7'b0000000;
       end

       else
       begin
              case (bcd[7:4])// tens place
                     4'h0: // for display 0 on seven segTensment
                     begin
                            segTens[6] = 1'b1;
                            segTens[5] = 1'b0;
                            segTens[4] = 1'b0;
                            segTens[3] = 1'b0;
                            segTens[2] = 1'b0;
                            segTens[1] = 1'b0;
                            segTens[0] = 1'b0;
                     end

                     4'h1: // for display 1 on seven segTensment
                     begin
                            segTens[6] = 1'b1;
                            segTens[5] = 1'b1;
                            segTens[4] = 1'b1;
                            segTens[3] = 1'b1;
                            segTens[2] = 1'b0;
        segTens[1] = 1'b0;
        segTens[0] = 1'b1;
end

4'h2: // for display 2 on seven segTensment
begin
        segTens[6] = 1'b0;
        segTens[5] = 1'b1;
        segTens[4] = 1'b0;
        segTens[3] = 1'b0;
        segTens[2] = 1'b1;
        segTens[1] = 1'b0;
        segTens[0] = 1'b0;
end

4'h3: // for display 3 on seven segTensment
begin
        segTens[6] = 1'b0;
        segTens[5] = 1'b1;
        segTens[4] = 1'b1;
        segTens[3] = 1'b0;
        segTens[2] = 1'b0;
        segTens[1] = 1'b0;
        segTens[0] = 1'b0;
end

4'h4: // for display 4 on seven segTensment
begin
        segTens[6] = 1'b0;
        segTens[5] = 1'b0;
        segTens[4] = 1'b1;
        segTens[3] = 1'b1;
        segTens[2] = 1'b0;
        segTens[1] = 1'b0;
        segTens[0] = 1'b1;
end

4'h5: // for display 5 on seven segTensment
begin
        segTens[6] = 1'b0;
        segTens[5] = 1'b0;
        segTens[4] = 1'b1;
        segTens[3] = 1'b0;
        segTens[2] = 1'b0;
        segTens[1] = 1'b1;
        segTens[0] = 1'b0;
end

4'h6: // for display 6 on seven segTensment
begin
        segTens[6] = 1'b0;
        segTens[5] = 1'b0;
```

```
                segTens[4] = 1'b0;
                segTens[3] = 1'b0;
                segTens[2] = 1'b0;
                segTens[1] = 1'b1;
                segTens[0] = 1'b0;
        end

        4'h7: // for display 7 on seven segTensment
        begin
                segTens[6] = 1'b1;
                segTens[5] = 1'b1;
                segTens[4] = 1'b1;
                segTens[3] = 1'b1;
                segTens[2] = 1'b0;
                segTens[1] = 1'b0;
                segTens[0] = 1'b0;
        end

        4'h8: // for display 8 on seven segTensment
        begin
                segTens[6] = 1'b0;
                segTens[5] = 1'b0;
                segTens[4] = 1'b0;
                segTens[3] = 1'b0;
                segTens[2] = 1'b0;
                segTens[1] = 1'b0;
                segTens[0] = 1'b0;
        end

        4'h9: // for display 9 on seven segTensment
        begin
                segTens[6] = 1'b0;
                segTens[5] = 1'b0;
                segTens[4] = 1'b1;
                segTens[3] = 1'b0;
                segTens[2] = 1'b0;
                segTens[1] = 1'b0;
                segTens[0] = 1'b0;
        end

        default:
        begin
                segTens[6] = 1'b0;
                segTens[5] = 1'b0;
                segTens[4] = 1'b0;
                segTens[3] = 1'b0;
                segTens[2] = 1'b0;
                segTens[1] = 1'b0;
                segTens[0] = 1'b0;
        end
endcase
```

```
case (bcd[3:0])// ones place
        4'h0: // for display 0 on seven segTensment
        begin
                segOnes[6] = 1'b1;
                segOnes[5] = 1'b0;
                segOnes[4] = 1'b0;
                segOnes[3] = 1'b0;
                segOnes[2] = 1'b0;
                segOnes[1] = 1'b0;
                segOnes[0] = 1'b0;
        end


        4'h1: // for display 1 on seven segTensment
        begin
                segOnes[6] = 1'b1;
                segOnes[5] = 1'b1;
                segOnes[4] = 1'b1;
                segOnes[3] = 1'b1;
                segOnes[2] = 1'b0;
                segOnes[1] = 1'b0;
                segOnes[0] = 1'b1;
        end

        4'h2: // for display 2 on seven segTensment
        begin
                segOnes[6] = 1'b0;
                segOnes[5] = 1'b1;
                segOnes[4] = 1'b0;
                segOnes[3] = 1'b0;
                segOnes[2] = 1'b1;
                segOnes[1] = 1'b0;
                segOnes[0] = 1'b0;
        end

        4'h3: // for display 3 on seven segTensment
        begin
                segOnes[6] = 1'b0;
                segOnes[5] = 1'b1;
                segOnes[4] = 1'b1;
                segOnes[3] = 1'b0;
                segOnes[2] = 1'b0;
                segOnes[1] = 1'b0;
                segOnes[0] = 1'b0;
        end

        4'h4: // for display 4 on seven segTensment
        begin
                segOnes[6] = 1'b0;
                segOnes[5] = 1'b0;
                segOnes[4] = 1'b1;
        segOnes[3] = 1'b1;
        segOnes[2] = 1'b0;
        segOnes[1] = 1'b0;
        segOnes[0] = 1'b1;
end

4'h5: // for display 5 on seven segTensment
begin
        segOnes[6] = 1'b0;
        segOnes[5] = 1'b0;
        segOnes[4] = 1'b1;
        segOnes[3] = 1'b0;
        segOnes[2] = 1'b0;
        segOnes[1] = 1'b1;
        segOnes[0] = 1'b0;
end

4'h6: // for display 6 on seven segTensment
begin
        segOnes[6] = 1'b0;
        segOnes[5] = 1'b0;
        segOnes[4] = 1'b0;
        segOnes[3] = 1'b0;
        segOnes[2] = 1'b0;
        segOnes[1] = 1'b1;
        segOnes[0] = 1'b0;
end

4'h7: // for display 7 on seven segTensment
begin
        segOnes[6] = 1'b1;
        segOnes[5] = 1'b1;
        segOnes[4] = 1'b1;
        segOnes[3] = 1'b1;
        segOnes[2] = 1'b0;
        segOnes[1] = 1'b0;
        segOnes[0] = 1'b0;
end

4'h8: // for display 8 on seven segTensment
begin
        segOnes[6] = 1'b0;
        segOnes[5] = 1'b0;
        segOnes[4] = 1'b0;
        segOnes[3] = 1'b0;
        segOnes[2] = 1'b0;
        segOnes[1] = 1'b0;
        segOnes[0] = 1'b0;
end

4'h9: // for display 9 on seven segTensment
begin
```

```
                              segOnes[6] = 1'b0;
                              segOnes[5] = 1'b0;
                              segOnes[4] = 1'b1;
                              segOnes[3] = 1'b0;
                              segOnes[2] = 1'b0;
                              segOnes[1] = 1'b0;
                              segOnes[0] = 1'b0;
                      end

                      default:
                      begin
                              segOnes[6] = 1'b0;
                              segOnes[5] = 1'b0;
                              segOnes[4] = 1'b0;
                              segOnes[3] = 1'b0;
                              segOnes[2] = 1'b0;
                              segOnes[1] = 1'b0;
                              segOnes[0] = 1'b0;
                      end
              endcase
              end
       end
endmodule
```

c. MIPS

```
//`timescale 1ns/10ps
module mips #(parameter WIDTH = 32, REGBITS = 5)( clk, reset,pcen);//,IF,ID,EX,MEM,WB);
input clk, reset,pcen;
//output IF,ID,EX,MEM,WB;
   wire [WIDTH-1:0] sl2out,wd,nextpc,readpc,alub;
   wire [3:0]  alucont;

   wire           pcsrc,
memreadID,memwriteID,alusrcID,regdstID,branchID,regwriteID,memtoregID; //1 bit ID regs
   wire [WIDTH-1:0] instrIF,instrID,pcplus4IF,pcplus4ID, signextendoutID,rd1ID,rd2ID; //32
bit IF and ID regs
   wire [1:0]     aluopID,aluopEX; //2 bit ID and EX
   wire [WIDTH-1:0] pcplus4EX,signextendoutEX,instrEX,aluresultEX,branchpcEX,rd1EX,rd2EX;
//32 bit EX regs
   wire           alusrcEX,regdstEX,branchEX,memwriteEX,memreadEX, zeroEX,
regwriteEX,memtoregEX, memreadMEM,memwriteMEM,branchMEM,zeroMEM,regwriteMEM,memtoregMEM; //
1 bit EX and MEM regs
   wire [WIDTH-1:0] aluresultMEM,branchpcMEM,rd2MEM,dcoutMEM,dcoutWB,aluresultWB; //32 bit
MEM and WB regs
   wire           memtoregWB, regwriteWB; //1 bit WB regs
   wire  [4:0]    waEX, waMEM, waWB;
flag
flags(.reset(reset),.instrIF(instrIF),.instrID(instrID),.instrEX(instrEX),.aluresultMEM(alur
esultMEM),.aluresultWB(aluresultWB),.flagIF(IF),.flagID(ID),.flagEX(EX),.flagMEM(MEM),.flagW
B(WB));
```

```
//clkdiv divider(.clk(fclk),.reset(reset),.clk_out(clk));

//Instruction Fetch
mux2 #(WIDTH)  pcmux(.d0(pcplus4IF), .d1(branchpcMEM), .s(pcsrc), .y(nextpc));
PC             pcreg(.nextpc(nextpc),.outpc(readpc),.reset(reset),.enable(pcen),.clk(clk));
adder          pcadder(.out(pcplus4IF), .in1(readpc), .in2(32'h00000004));
imem           instrmem(.clk(clk), .adr(readpc), .memdata(instrIF),.rst(reset));

//Instruction Decode
controller     cont(.alusrc(alusrcID), .aluop(aluopID), .memread(memreadID),
.memwrite(memwriteID), .memtoreg(memtoregID), .branch(branchID), .regwrite(regwriteID),
.regdst(regdstID), .op(instrID[31:26]), .clk(clk), .reset(reset));
regfile #(WIDTH, REGBITS) rf(.clk(clk), .regwrite(regwriteWB), .ra1(instrID[25:21]),
.ra2(instrID[20:16]), .wa(waWB), .wd(wd), .rd1(rd1ID), .rd2(rd2ID),.rst(reset));//NEED to
add wa and wd
signextend     se(.in(instrID[15:0]) , .out(signextendoutID));

//Execution
alucontrol     aluc(.alucont(alucont), .aluop(aluopEX), .funct(signextendoutEX[5:0]));
//should be the same as instr[5:0]
shiftleft2     sl2(.out(sl2out), .in(signextendoutEX));
//adder        branchadder(.out(branchpcEX), .in1(pcplus4EX), .in2(sl2out));
mux2_5         regmux(.d0(instrEX[20:16]), .d1(instrEX[15:11]), .s(regdstEX), .y(waEX));
mux2 #(WIDTH)  alumux(.d0(rd2EX), .d1(signextendoutEX), .s(alusrcEX), .y(alub));
alu #(WIDTH)   alumain(.a(rd1EX), .b(alub), .alucont(alucont), .result(aluresultEX),
.zero(zeroEX));

//Memory
datacache      dc(.address(aluresultMEM),
.data_in(rd2MEM),.temp_out(dcoutMEM),.read(memreadMEM),.write(memwriteMEM),.clk(clk),.rst(re
set));

//Writeback
mux2 #(WIDTH)  wdmux(.d0(dcoutWB), .d1(aluresultWB), .s(memtoregWB), .y(wd));


//pipelinestages
IFID #(WIDTH)  ifid(instrIF,instrID,pcplus4IF, pcplus4ID,clk);

IDEX #(WIDTH)
idex(clk,memreadID,memwriteID,alusrcID,regdstID,branchID,regwriteID,memtoregID,instrID,pcplu
s4ID,
signextendoutID,rd1ID,rd2ID,aluopID,aluopEX,pcplus4EX,signextendoutEX,instrEX,rd1EX,rd2EX,al
usrcEX,regdstEX,branchEX,memwriteEX,memreadEX, regwriteEX,memtoregEX);//zeroEX,branchpcEX

EXMEM #(WIDTH) exmemp(clk,branchEX,memwriteEX,memreadEX, zeroEX,
regwriteEX,memtoregEX,aluresultEX,pcplus4EX,rd2EX,waEX,
memreadMEM,memwriteMEM,branchMEM,zeroMEM,regwriteMEM,memtoregMEM,waMEM,aluresultMEM,branchpc
MEM,rd2MEM);//,dcoutMEM);
```

```
MEMWB #(WIDTH)
memwb(clk,regwriteMEM,memtoregMEM,dcoutMEM,aluresultMEM,waMEM,waWB,memtoregWB,
regwriteWB,dcoutWB,aluresultWB);


assign pcsrc = branchEX & zeroMEM;
//assign branchpcEX = pcplus4EX; //| sl2out;
endmodule

module IFID #(parameter WIDTH = 32)(instrIF,instrID,pcplus4IF, pcplus4ID,clk);
       input [WIDTH-1:0] instrIF,pcplus4IF;
       input clk;
       output reg [WIDTH-1:0] instrID,pcplus4ID;
       always@(posedge clk)
       begin
              //Fetch -> Decode
              instrID <= instrIF;
              pcplus4ID <= pcplus4IF;
       end
endmodule

module IDEX #(parameter WIDTH =
32)(clk,memreadID,memwriteID,alusrcID,regdstID,branchID,regwriteID,memtoregID,instrID,pcplus
4ID,
signextendoutID,rd1ID,rd2ID,aluopID,aluopEX,pcplus4EX,signextendoutEX,instrEX,rd1EX,rd2EX,al
usrcEX,regdstEX,branchEX,memwriteEX,memreadEX, regwriteEX,memtoregEX);//branchcEX, zeroEX;
       input clk;
       input memreadID,memwriteID,alusrcID,regdstID,branchID,regwriteID,memtoregID;
       input [WIDTH-1:0] instrID,pcplus4ID, signextendoutID,rd1ID,rd2ID;
       input [1:0] aluopID;
       output reg [1:0] aluopEX;
       output reg [WIDTH-1:0] pcplus4EX,signextendoutEX,instrEX,rd1EX,rd2EX;//branchpcEX
       output reg alusrcEX,regdstEX,branchEX,memwriteEX,memreadEX,regwriteEX,memtoregEX;//
zeroEX,
       always@(posedge clk)
       begin
              //Decode -> Execute
              pcplus4EX <= pcplus4ID;
              signextendoutEX <= signextendoutID;
              instrEX <= instrID;
              rd1EX <= rd1ID;
              rd2EX <= rd2ID;
              regdstEX <= regdstID;
              alusrcEX <= alusrcID;
              aluopEX <= aluopID;
              branchEX <= branchID;
              regwriteEX <= regwriteID;
              memreadEX <= memreadID;
              memwriteEX <= memwriteID;
              memtoregEX <= memtoregID;
       end
endmodule
```

```
module EXMEM#(parameter WIDTH = 32)(clk,branchEX,memwriteEX,memreadEX, zeroEX,
regwriteEX,memtoregEX,aluresultEX,branchpcEX,rd2EX,waEX,
memreadMEM,memwriteMEM,branchMEM,zeroMEM,regwriteMEM,memtoregMEM,waMEM,aluresultMEM,branchpc
MEM,rd2MEM);//,dcoutMEM
	input clk;
	input branchEX,memwriteEX,memreadEX, zeroEX, regwriteEX,memtoregEX;
	input[WIDTH-1:0] aluresultEX,branchpcEX,rd2EX;
	input [4:0]	waEX;
	output reg memreadMEM,memwriteMEM,branchMEM,zeroMEM,regwriteMEM,memtoregMEM;
	output reg [4:0]   waMEM;
	output reg [WIDTH-1:0] aluresultMEM,branchpcMEM,rd2MEM;//,dcoutMEM;

	always@(posedge clk)
	begin
		//Execute -> Memory
		waMEM <= waEX;
		rd2MEM <= rd2EX;
		aluresultMEM <= aluresultEX;
		branchMEM <= branchEX;
		zeroMEM <= zeroEX;
		branchpcMEM <= branchpcEX;
		memreadMEM <= memreadEX;
		memwriteMEM <= memwriteEX;
		regwriteMEM <= regwriteEX;
		memtoregMEM <= memtoregEX;
	end
endmodule

module MEMWB#(parameter WIDTH =
32)(clk,regwriteMEM,memtoregMEM,dcoutMEM,aluresultMEM,waMEM,waWB,memtoregWB,
regwriteWB,dcoutWB,aluresultWB);
	input clk;
	input regwriteMEM,memtoregMEM;
	input [WIDTH-1:0] dcoutMEM,aluresultMEM;
	input [4:0]	waMEM;
	output reg [4:0]   waWB;
	output reg memtoregWB, regwriteWB;
	output reg [WIDTH-1:0] dcoutWB,aluresultWB;

	always@(posedge clk)
	begin
		//Memory -> Write Back
		regwriteWB <= regwriteMEM;
		dcoutWB <= dcoutMEM;
		aluresultWB <= aluresultMEM;
		waWB <= waMEM;
		memtoregWB <= memtoregMEM;
	end
endmodule

module alu #(parameter WIDTH = 32)
```

```
            (input      [WIDTH-1:0] a, b,
             input      [3:0]       alucont,
             output reg [WIDTH-1:0] result,
            output zero);

   wire     [WIDTH-1:0] sum, slt;
   assign zero = (alucont==0); //Zero is true if aluout is 0;
   // slt should be 1 if most significant bit of sum is 1

   always@(*)
      case(alucont)
         4'b0000: result <= a & b;
         4'b0001: result <= a | b;
         4'b0010: result <= a + b;
         4'b0110: result <= a - b;
        4'b0111: result <= a < b ? 1:0;
        4'b1100: result <= ~(a | b); //result is nor
        default: result <= 0; //default to 0 which shouldn't happen
      endcase
endmodule

module regfile #(parameter WIDTH = 32, REGBITS = 5)
                (input                clk, rst,
                 input                regwrite,
                 input  [REGBITS-1:0] ra1, ra2, wa,
                 input  [WIDTH-1:0]   wd,
                 output reg [WIDTH-1:0]   rd1, rd2);

   reg  [31:0] memory [(1<<REGBITS)-1:0];

   // three ported register file
   // read two ports combinationally
   // write third port on rising edge of clock
   // register 0 hardwired to 0

   always @(posedge clk or posedge rst)
   begin
       if(rst)
       begin
              memory[0] <= 32'h00000000;
              memory[1] <= 32'h00000000;
              memory[2] <= 32'h00000000;
              memory[3] <= 32'h00000000;
              memory[4] <= 32'h00000000;
              memory[5] <= 32'h00000000;
              memory[6] <= 32'h00000000;
              memory[7] <= 32'h00000000;
              memory[8] <= 32'h00000001;
              memory[9] <= 32'h00000002;
              memory[10] <= 32'h00000000;
              memory[11] <= 32'h00000000;
              memory[12] <= 32'h00000000;
```

```
                memory[13] <= 32'h00000000;
                memory[14] <= 32'h00000000;
                memory[15] <= 32'h00000000;
                memory[16] <= 32'h00000000;
                memory[17] <= 32'h00000000;
                memory[18] <= 32'h00000003;
                memory[19] <= 32'h00000003;
                memory[20] <= 32'h00000004;
                memory[21] <= 32'h00000000;
                memory[22] <= 32'h00000008;
                memory[23] <= 32'h00000000;
                memory[24] <= 32'h00000000;
                memory[25] <= 32'h00000000;
                memory[26] <= 32'h00000000;
                memory[27] <= 32'h00000000;
                memory[28] <= 32'h00000000;
                memory[29] <= 32'h00000000;
                memory[30] <= 32'h00000000;
                memory[31] <= 32'h00000000;
        end

        else if(regwrite)
        begin
                memory[wa] <= wd;
        end
    end

    always @(ra1,ra2)
    begin
        rd1 = memory[ra1];
        rd2 = memory[ra2];
    end
endmodule


module signextend(in ,out);

input  [15:0] in;
output [31:0] out;

assign out =  (in[15] == 1)? {16'hffff , in} :
              (in[15] == 0)? {16'h0000 ,in}  : 16'hxxxx;

endmodule

module shiftleft2(out, in);

input [31:0]in;
output [31:0]out;

assign out = in << 2;
```

```
endmodule

module mux2_5(input  [4:0] d0, d1,
              input        s,
              output [4:0] y);

   assign y = s ? d1 : d0;
endmodule

module mux2 #(parameter WIDTH = 32)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule

module adder(out, in1, in2);

input [31:0] in1,in2;
output [31:0] out;

assign out = in1 + in2;
endmodule

module PC (nextpc,outpc,reset,enable,clk);
	input [31:0] nextpc;
	input reset,enable,clk;
	output reg [31:0] outpc;

	always@(posedge clk or posedge reset)
	begin
		if(reset)
		begin
			outpc <= 32'b0;
		end
		else if(enable == 1)
		begin
			outpc <= nextpc;
		end
		else
		begin
			outpc <= outpc;
		end
	end
endmodule
```

d. Flag

In order to observe the stage change on the FPGA board, the group created a flag module which would turn on the LED lights when the corresponding stage received an input.

```
module flag(reset, instrIF, instrID, instrEX, aluresultMEM, aluresultWB, flagID, flagIF,
flagEX, flagMEM, flagWB);
        input reset;
        input instrIF, instrID, instrEX;
        input aluresultMEM, aluresultWB;
        output reg flagIF, flagID, flagEX, flagMEM, flagWB;

        always @(*)
        begin
                if (reset)
                        begin
                                flagIF <= 1'b0;
                                flagID <= 1'b0;
                                flagEX <= 1'b0;
                                flagMEM <= 1'b0;
                                flagWB <= 1'b0;
                        end
                else if (instrIF)
                        flagIF <= 1'b1;
                else if (instrID)
                        flagID <= 1'b1;
                else if (instrEX)
                        flagEX <= 1'b1;
                else if (aluresultMEM)
                        flagMEM <= 1'b1;
                else if (aluresultWB)
                        flagWB <= 1'b1;
                else
                        begin
                                flagIF <= 1'b0;
                                flagID <= 1'b0;
                                flagEX <= 1'b0;
                                flagMEM <= 1'b0;
                                flagWB <= 1'b0;
                        end
        end
endmodule
```

Problems and Debuggings

a. Clock Divider(frequency)

From the DE2-115 Board I/O Pin Assignments, the group noticed that the frequency of the clock on the board was 50 MHz, which would make it impossible to observe any changes happening. The group decided to use a clock divider to slow the process down to 1Hz.

b. Flags

The DE2-115 board has certain I/O pins. The group noticed that a ‘wire’ cannot be assigned to these pins. The team then decided to create a module that would convert the ‘wire’ to a reg so it can then be assigned to a pin on the board.

c. Double Drive

This issue was persistent throughout the entire project. The students would get a double drive issue that required them to go line by line to understand each variable and what the inputs to those variables would be. Once the students figured out what was causing the issue, debugging became a lot easier and the issues of the double drive became apparent. The team was able to find the double drive issues and resolve them by updating the code.

Conclusions

In all, the students were able to create a 32 Bit MIPS processor that was synthesized on an Altera FPGA board. The inputs were a clock, a reset switch, and an enable switch that was used to stall the code. A manual stall was implemented due to time constraints. The outputs of the board were the program counter on the seven segment display, the pipeline stages on LEDs (Instruction Fetch, Instruction Decode, Execution, Memory, and Write Back), and an LED output for every time the divided clock was high so the human eye could see when each new clock cycle was. There were a lot of issues the students ran into while creating this project such as double drive errors, interfacing with the DE2-115 board, and timing of the testbench to verify functionality. In all the students learned a lot about Verilog coding, computer architecture design, static timing analysis, and design verification. All these skills will be extremely beneficial for the students in their final semester of graduate school and in the industry once they graduate and start working.