# CNG 331 – CAD2 Assignment

Fatma Erem Aksoy – 2315075

Muhammad Khizr Shahid – 2413235

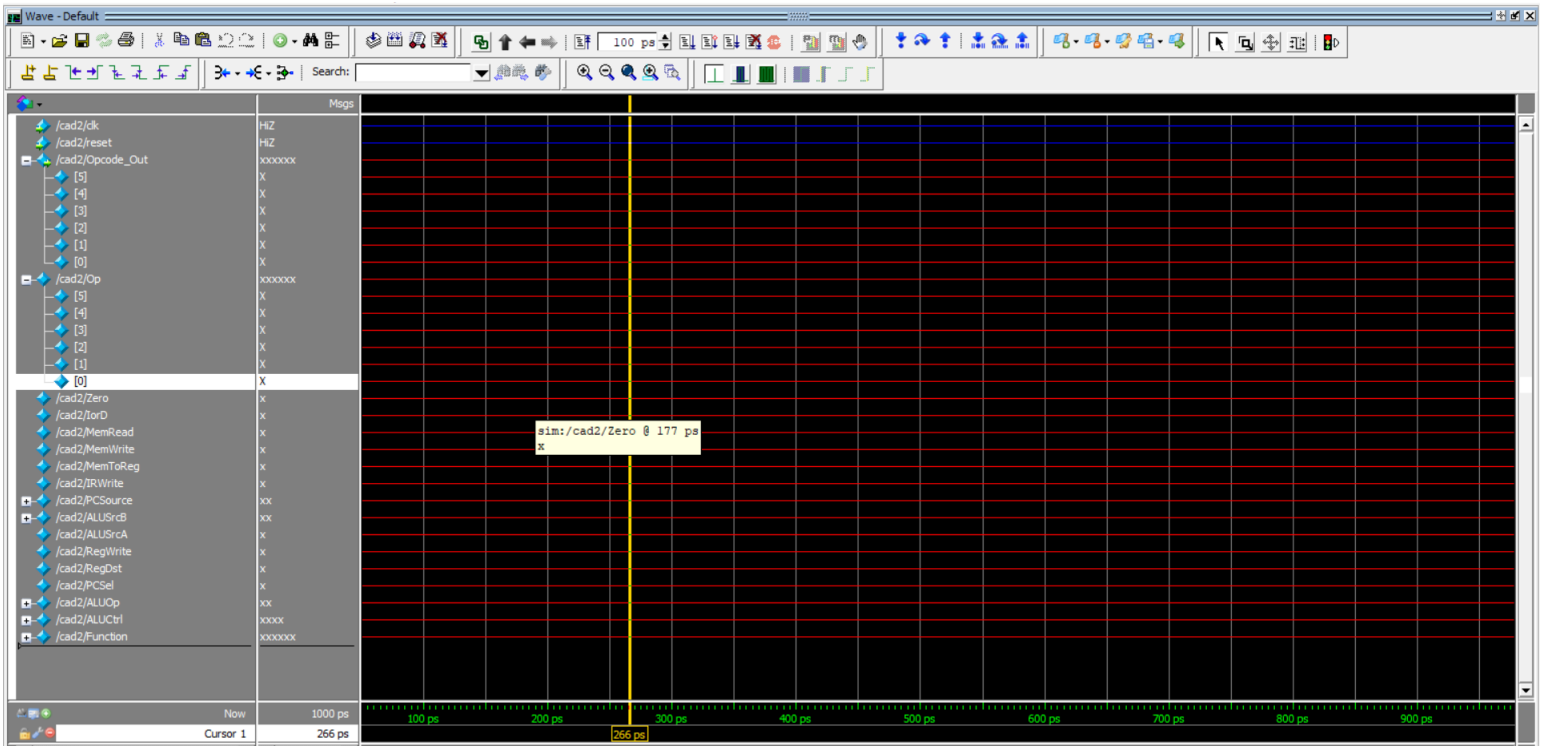
**INTRODUCTION:**

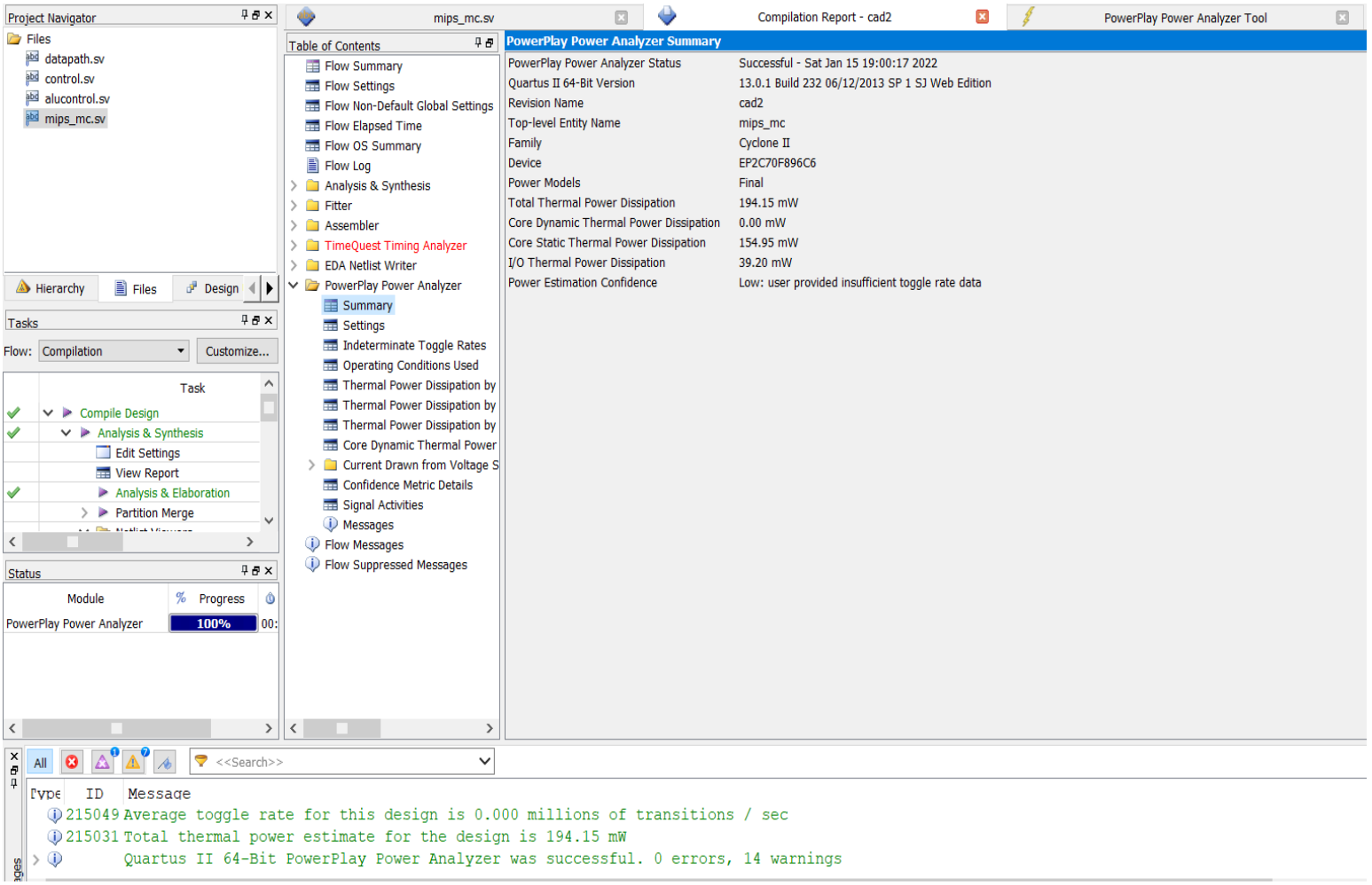
The goal of this CAD assignment is to design multicycle and pipeline MIPS Processors with the knowledge gained from CAD1. We need to calculate some values such as maximum clock rate (frequency), average CPI, execution time, energy consumption and thermal power dissipation to compare those two designs and decide which one is better when a specific requirement is considered. To achieve that, we also need to simulate some code and compare them based on the simulation results.

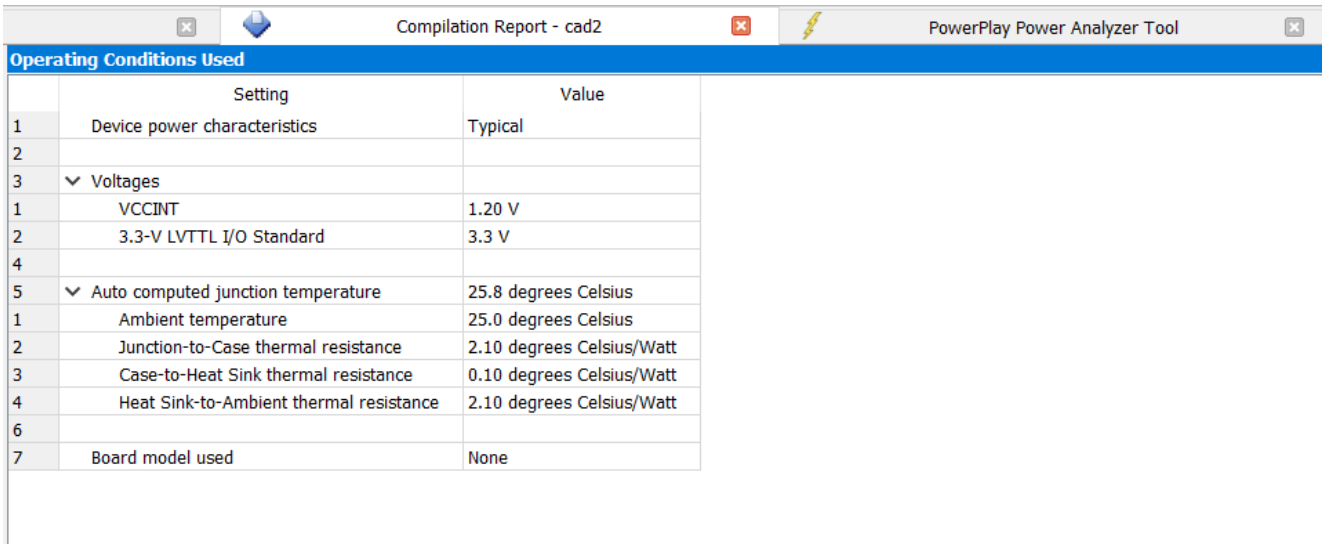
**METHOD:**

This CAD assignment was really challenging for us as it's a practical assignment and even though we are familiar and confident with the theoretical concepts, applying them to practice was an arduous process for both of us. We did not really split the task between each other to make sure that we both worked together in each step to learn new things from each other's knowledge. We worked together during the whole process, but still it was hard for us to complete both parts multicycle and pipeline design. We only managed to do multicycle design. Our code works properly on Quartus, but it does not work on Modelsim for some reason that we could not detect. It gives red lines, and we could not find the issue there to fix it to get the required waveforms. We did loads of debugging, and it was fun to analyze the code step by step with the hope that we could fix the waveforms, but it did not work, so it was very tiring and time-consuming. As Modelsim did not work as desired, we mostly worked on Quartus to get values using Timequest Analysis and Powerplay Analysis tools. We checked the summary after simulating the analysis, and we were able to get logical results from there.

## FUNCTIONAL VERIFICATION:






| | Setting | Value |
|---|---|---|
| 1 | Device power characteristics | Typical |
| 2 | | |
| 3 | Voltages | |
| 1 | VCCINT | 1.20 V |
| 2 | 3.3-V LVTTL I/O Standard | 3.3 V |
| 4 | | |
| 5 | Auto computed junction temperature | 25.8 degrees Celsius |
| 1 | Ambient temperature | 25.0 degrees Celsius |
| 2 | Junction-to-Case thermal resistance | 2.10 degrees Celsius/Watt |
| 3 | Case-to-Heat Sink thermal resistance | 0.10 degrees Celsius/Watt |
| 4 | Heat Sink-to-Ambient thermal resistance | 2.10 degrees Celsius/Watt |
| 6 | | |
| 7 | Board model used | None |

**RESULTS:**

| Parameter | Multi Cycle MIPS |
|---|---|
| **$F_{max}$ (TimeQuest Timing Analyzer)** | **92.8 MHz** |
| **$CPI_{avg}$** | |
| **Execution Time (50MHz)** | **(CPI * 10) / 50MHz** |
| **Core Total Thermal Power Dissipation (PowerPlay Power Analyzer)** | **194.15 mW** |
| **Energy Consumption (50MHz)** | **194.15 mW * Exec.Time** |

**CPI (avg) =** # of clock cycles/# of instructions

Execution Time = (CPI * # of instructions) / Clock rate = (CPI * 10) / 50MHz

Energy Consumption = Power * Execution Time = 194.15 mW * Exec.Time

**CONCLUSION:**

Basically we were able to complete almost all of multi cycle design, but unfortunately we were not able to do the pipeline design due to the complexity and difficulty level of that part. We were able to compile the multi cycle design successfully, however, we weren't getting the desired simulation outputs. We were unable to calculate the CPI for the multi cycle design so we were unable to calculate the rest of the parts such as Execution Time and Energy consumption but we had an idea on how to calculate them so we have shown the formulas and available calculations above in the table. As we were unable to do the pipeline design, we couldn't compare are results from multi cycle design to the pipeline design.

**APPENDICIES:**

datapath.sv:

```
module datapath(input logic clk, reset, IorD, MemRead, MemWrite, MemToReg, input logic IRWrite, ALUSrcA, RegWrite, RegDst, PCSel,
                input logic [1:0] PCSource, input logic [1:0] ALUSrcB,input logic [3:0] ALUCtrl, output logic Zero,output logic [5:0] Op,
                output logic [5:0] Function);

    parameter PCSTART = 128; //starting address of instruction memory
    logic [7:0] PC;
    logic [31:0] ALUOut;
    logic [31:0] ALUResult;
    logic [31:0] OpA;
    logic [31:0] OpB;
    logic [31:0] A;
    logic [31:0] B;
    logic [31:0] Instruction;
    logic [31:0] MDR;
    logic [31:0] dA;
    logic [31:0] dB;
    logic [7:0] address;
    logic [31:0] MemData;
    logic [31:0] RF_WriteData;
    logic [4:0] RF_rd;
    reg[31:0] mem[255:0];
    reg[31:0] registers[31:0];


    assign Op = Instruction[31:26];
    assign Function = Instruction[5:0];

    assign address = (IorD) ? ALUOut : PC;
    initial $readmemh("unified_memory.dat", mem);


    always @(posedge clk) begin
       if(MemWrite)
          mem[address] <= B;
    end

    assign MemData = (MemRead)?mem[address] : 32'bx;

    always @(posedge clk)begin
       if(reset)
          PC <= PCSTART;
       else
          if(PCSel)begin
             case (PCSource)
                1'b0: PC <= ALUResult;
                1'b1: PC <= ALUOut;
             endcase
          end
    end


    always @(posedge clk) begin
       if (IRWrite)
          Instruction <= MemData;
    end


    always @(posedge clk) begin
       MDR <= MemData;
    end

    //Register File
    //$r0 is always 0
    assign dA = (Instruction[25:21]!=0) ? registers[Instruction[25:21]] : 0;
    assign dB = (Instruction[20:16]!=0) ? registers[Instruction[20:16]] : 0;
    assign RF_WriteData = (MemToReg) ? MDR : ALUOut;
    assign RF_rd = (RegDst) ? Instruction[15:11] : Instruction[20:16];
```

```
70    always @(posedge clk) begin
71       if (RegWrite) begin
72          registers[RF_rd] <= RF_WriteData;
73       end
74    end
75
76
77    //A and B registers
78    always @(posedge clk) begin
79       if (reset)
80          A <= 0;
81       else
82          A<=dA;
83    end
84
85    always @(posedge clk) begin
86       if (reset)
87          B <= 0;
88       else
89          B<=dB;
90    end
91
92
93    //ALU
94    assign OpA = (ALUSrcA) ? A : PC;
95
96    always_comb
97    begin
98       case(ALUSrcB)
99       2'b00: OpB = B;
100      2'b01: OpB = 4;
101      2'b10: OpB = {{(16){Instruction[15]}}, Instruction[15:0]};
102      2'b11: OpB = {{(14){Instruction[15]}}, Instruction[15:0],2'b00};
103      endcase
104   end
105
106   assign Zero = (ALUResult==0); //Zero == 1 when ALUResult is 0 (for branch)
107
108   always_comb
109   begin
110      case(ALUCtrl)
111      4'b0000: ALUResult = OpA & OpB;  //and
112      4'b0001: ALUResult = OpA | OpB;  //or
113      4'b0010: ALUResult = OpA ^ OpB;  //add
114      4'b0011: ALUResult = ~(OpA | OpB);
115      4'b0110: ALUResult = OpA + OpB;  //branch
116      4'b1110: ALUResult = OpA - OpB;
117      4'b1111: ALUResult = OpA < OpB?1:0; //ALUResult is 1 when OpA<Opb, otherwise 0
118      default: ALUResult = OpA + OpB;
119      endcase
120   end
121
122
123   always@(posedge clk) begin
124      ALUOut <= ALUResult;
125   end
126
127 endmodule
```

alucontrol.sv

```
module alucontrol(ALUOp,Function,ALUCtrl);

   input [1:0] ALUOp;
   input [5:0] Function;

   output reg [3:0] ALUCtrl;  //for R-type instructions

   always@(ALUOp or Function)begin
      casex({ALUOp,Function})
         8'b00_xxxxxx : ALUCtrl=4'b0010;     //lw or sw
         8'b01_xxxxxx : ALUCtrl=4'b0110;  //beq
         8'b1x_xx0000 : ALUCtrl=4'b0010;  //add
         8'b1x_xx0010 : ALUCtrl=4'b0110;  //sub
         8'b1x_xx0100 : ALUCtrl=4'b0000;  //and
         8'b1x_xx0101 : ALUCtrl=4'b0001;  //or
         8'b1x_xx1010 : ALUCtrl=4'b0111;  //slt
      endcase
   end

endmodule
```

mips_mc.sv:

```
module mips_mc(input logic clk,
               input logic reset,
            output logic [5:0] Opcode_Out);

   logic [5:0] Op;
   logic Zero;
   logic IorD;
   logic MemRead;
   logic MemWrite;
   logic MemToReg;
   logic IRWrite;
   logic [1:0] PCSource;
   logic [1:0] ALUSrcB;
   logic ALUSrcA;
   logic RegWrite;
   logic RegDst;
   logic PCSel;
   logic [1:0] ALUOp;
   logic [3:0] ALUCtrl;
   logic [5:0] Function;
   assign Opcode_Out = Op[5:0];

   control u0(.clk(clk), .reset(reset), .Zero(Zero), .Op(Op), .IorD(IorD), .MemRead(MemRead),
            .MemWrite(MemWrite), .MemToReg(MemToReg), .IRWrite(IRWrite), .ALUSrcA(ALUSrcA),
            .RegWrite(RegWrite), .RegDst(RegDst), .PCSel(PCSel), .PCSource(PCSource),
            .ALUSrcB(ALUSrcB), .ALUOp(ALUOp));

   alucontrol u1(.ALUOp(ALUOp), .Function(Function), .ALUCtrl(ALUCtrl));

   datapath u2(.clk(clk), .reset(reset), .IorD(IorD), .MemRead(MemRead), .MemWrite(MemWrite),
            .MemToReg(MemToReg), .IRWrite(IRWrite), .ALUSrcA(ALUSrcA), .RegWrite(RegWrite),
            .RegDst(RegDst), .PCSel(PCSel), .PCSource(PCSource), .ALUSrcB(ALUSrcB),
            .ALUCtrl(ALUCtrl), .Zero(Zero), .Op(Op), .Function(Function));
endmodule
```

tb_mips_mc.sv:

```
module tb_mips_mc;
   logic clk;
   logic reset;
   logic [5:0] Opcode_Out;

   mips_mc utb(.clk(clk), .reset(reset), .Opcode_Out(Opcode_Out));

   initial
      forever #10000 clk=~clk;
      initial begin
         clk = 0; #20000;
         reset = 1;
         reset = 0;
         #1000000 $finish;
       end

endmodule
```

control.sv :

```
module control (input logic clk, reset, Zero,input logic [5:0] Op,output logic IorD,
                  MemRead, MemWrite, MemToReg, IRWrite, output logic ALUSrcA, RegWrite,
                  RegDst, PCSel,output logic [1:0] PCSource,output logic [1:0] ALUSrcB,
                  output logic [1:0] ALUOp);

   reg PCWrite;
   reg PCWriteCond;

   assign PCSel = (PCWrite | (PCWriteCond & Zero));

   parameter Fetch = 4'b0000; //Stage 0 - fetch
   parameter Decode = 4'b0001;   //Stage 1 - decode
   parameter LoadORStore = 4'b0010; //Stage 2 - load or store
   parameter MemAccessLoad = 4'b0011;  //Stage 3 - L1
   parameter WriteBack = 4'b0100;   //Stage 4 - Write Back
   parameter MemAccessStore = 4'b0101; //Satge 5 - Store
   parameter Execution = 4'b0110;   //Stage 6 - R-type
   parameter RType = 4'b0111;    //Stage 7
   parameter Beq = 4'b1000;   //Stage 8 - Branch
   parameter J = 4'b1001;  //Stage 9 - Jump

   reg [3:0] state;
   reg [3:0] nextstate;

    always@(posedge clk)

       if (reset)
         state <= Fetch;
       else
         state <= nextstate;

   always@(state or Op) begin
      case (state)
        Fetch:
```

```
            nextstate = Decode;

        Decode:
         case(Op)
             6'b100011: nextstate = LoadORStore;    //load(lw)
             6'b101011: nextstate = LoadORStore;    //store(sw)
             6'b000000: nextstate = Execution;   //r
             6'b000100: nextstate = Beq;   //branch(beq)
             6'b000010: nextstate = J;  //jump(j)
             default: nextstate = Fetch;
         endcase

        LoadORStore: case(Op)
                        6'b100011: nextstate = MemAccessLoad;//lw
                        6'b101011: nextstate = MemAccessStore;//sw
                        default: nextstate = Fetch;
                    endcase

        MemAccessLoad:     nextstate = WriteBack;
        WriteBack:         nextstate = Fetch;
        MemAccessStore:    nextstate = Fetch;
        Execution:         nextstate = RType;
        RType:             nextstate = Fetch;
        Beq:               nextstate = Fetch;
        J:                 nextstate = Fetch;
        default: nextstate = Fetch;
      endcase
    end

     always@(state) begin
       IorD=1'b0; MemRead=1'b0; MemWrite=1'b0; MemToReg=1'b0; IRWrite=1'b0;
       PCSource=1'b0; ALUSrcB=2'b00; ALUSrcA=1'b0; RegWrite=1'b0; RegDst=1'b0;
       PCWrite=1'b0; PCWriteCond=1'b0; ALUOp=2'b00;

       case (state)
          Fetch:
             begin
                MemRead = 1'b1;
                IRWrite = 1'b1;
                ALUSrcB = 2'b01;
                PCWrite = 1'b1;
             end

          Decode:
             ALUSrcB = 2'b11;

          LoadORStore:
             begin
                ALUSrcA = 1'b1;
                ALUSrcB = 2'b10;
             end

          MemAccessLoad:
             begin
                MemRead = 1'b1;
                IorD    = 1'b1;
             end

          WriteBack:
             begin
                RegWrite = 1'b1;
                MemToReg = 1'b1;
                RegDst = 1'b0;
             end

          MemAccessStore:
```

```
            begin
                MemWrite = 1'b1;
                IorD     = 1'b1;
            end

        Execution:
            begin
                ALUSrcA = 1'b1;
                ALUOp   = 2'b10;
            end

        RType:
            begin
                RegDst = 1'b1;
                RegWrite = 1'b1;
            end

        Beq:
            begin
                ALUSrcA = 1'b1;
                ALUOp   = 2'b01;
                PCWriteCond = 1'b1;
                PCSource = 2'b01;
            end

        J:
            begin
                PCSource = 2'b10;
                PCWrite = 1'b1;
            end
      endcase
    end

endmodule
```