Implementation of 32-bit 5-stage pipelined RISC-V processor

by

Ali Sajjad
2018-EE-63

Advisor:
Mr. Umer Shahid

FALL 2021

Department of Electrical Engineering
University of Engineering and Technology, Lahore

Course Code and Title: EE-475: Computer Architecture
Semester: Fall 2021 (7th Semester)
Instructor: Dr. Muhammad Tahir & Mr. Umer Shahid
Total Marks: 50 (in Lab)
Deadline: End of Semester

CLOs and PLOs for Complex Engineering Problem

Below stated are the CLOs and PLOs addressed in the complex engineering problem along with domain and level. These are the CLOs from the theory/lab course which are already defined.

CLOs		Description	Domains & Levels	PLOs, Levels
CLO1	Theory	Demonstrate the skills to design datapath of single-cycle, multi-cycle, and pipeline microprocessor architecture using a variety of techniques	Cognitive, 2. Understand	PLO1
CLO2	Lab	Formulate the control portion of a processor, and its data path consisting of the general-purpose register file, ALU, the memory interface, internal registers between stages, and multiplexers.	Psychomotor, 4. Articulate	PLO3
CLO3	Lab	Design and emulate a pipelined CPU by given specifications using Verilog	Psychomotor, 5. Naturalization	PLO3

Problem Statement

In this open-ended design lab, students will investigate and design a 32-bit RISC-V processor with five stage pipeline.

- Implement a processor that supports a subset of the RISC-V ISA. The designed processor does not support all RISC-V ISA instruction, however it will have most parts that a real processor has: A fetch unit, decode logic, functional units, a register file, I/O support, access to memory, interrupt handling, hazard elimination protocols, and CSR Support.

- Students will be implementing the datapath while designing a pipelined RISC-V processor which will work with five stages mainly, It must contain a file or block RAM of FPGA that will be used as Random Access Memory. It will have 32 Registers working as General purpose Register. While some special purpose registers (Program Counter, CSR registers) will be used to hold the address of the instruction and will handle the interrupts. Each Register must be 32 bit wide and clock edge triggered.

ABSTRACT

The pipeline implementation of RISC 5 processor architecture consists of several pipeline registers along with memories , control unit , hazard unit and CSR. These units are interconnected with each other efficiently . All types of instructions in RISV32I architecture can be implemented in the proposed datapath. Finally the designed processor is verified using cadence xcellium.

.

ACKNOWLEDGMENTS

The project Pipeline RISCV processor architecture is acknowledged to our instructor Dr. Muhammed Tahir and our lab instructor Sir Umer Shahid. The lab has been handled professionally and all the work was linked with industry standards. The Course work was manageable and was divided into phases to help students complete the complex engineering problem till the end of semester.

Lastly huge respect and acknowledgements to all those students and other teachers who helped to understand and implement the lab work.

STATEMENT OF ORGINALITY

It is stated that this CEP presented in this course consists of my own ideas and hands-on working. The contributions and ideas from others have been duly acknowledged and cited in the dissertation. This complete report and code is written by me.

[Ali Sajjad]

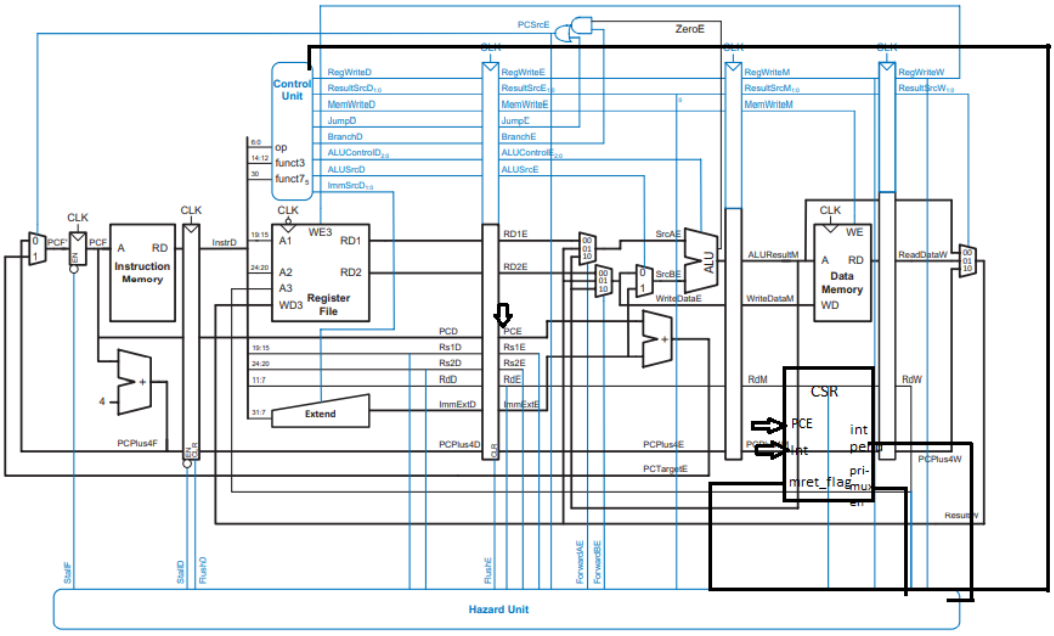
TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1. Design Description

The complete datapath of the implemented processor is as follows:

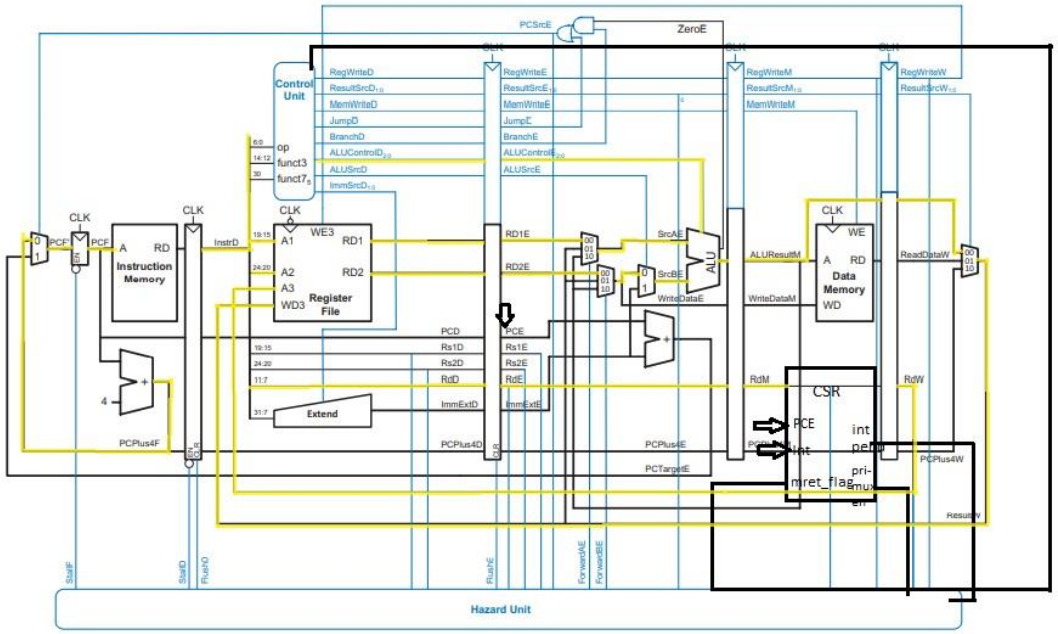


1.2. Potential Hazards

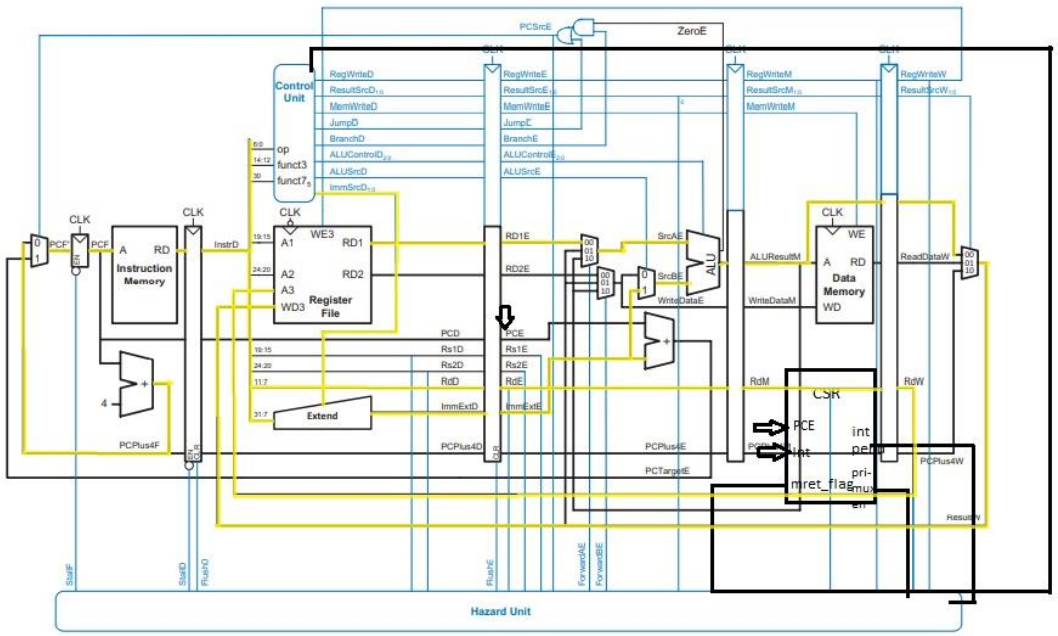
There are no potential hazard for the implemented datapath but it only have limitations.

1.3. Datapath Explanation

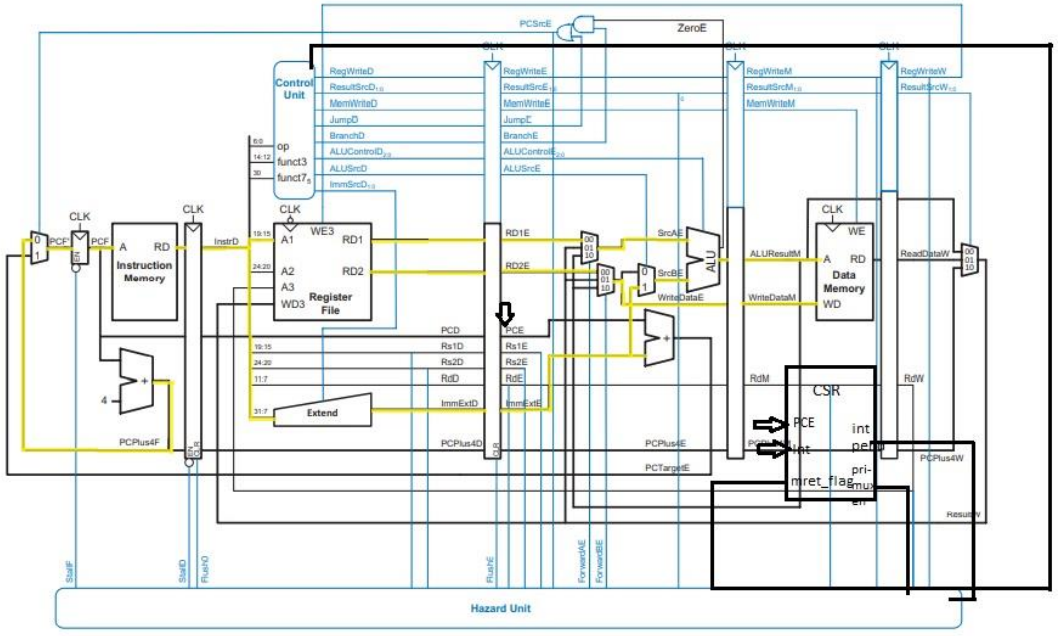
1.3.1. Highlighted Datapath for R-type Instructions



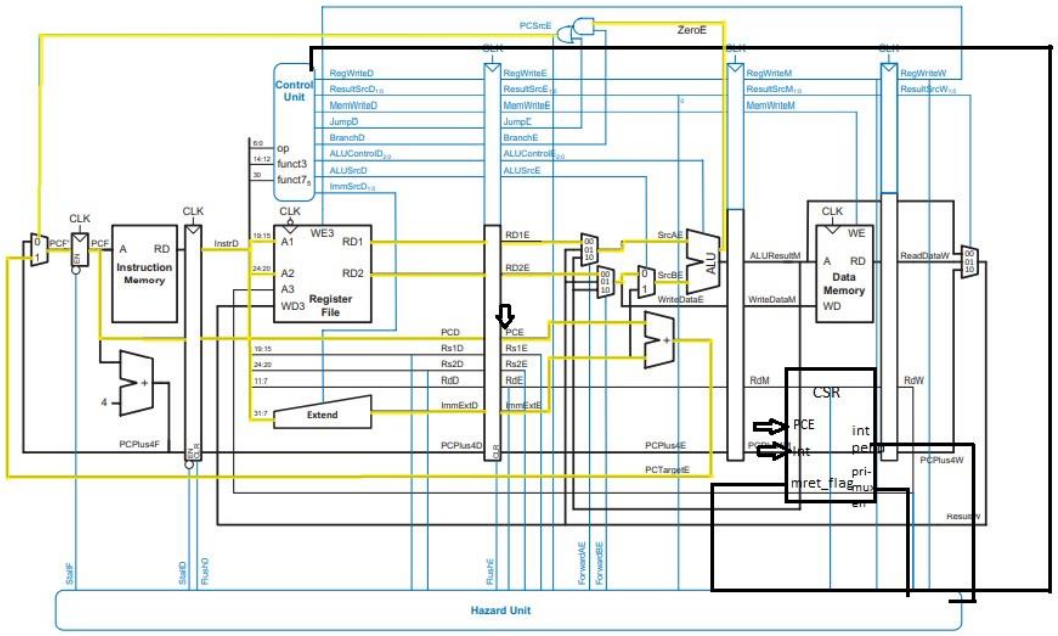
1.3.2. Highlighted Datapath for I-type Instructions



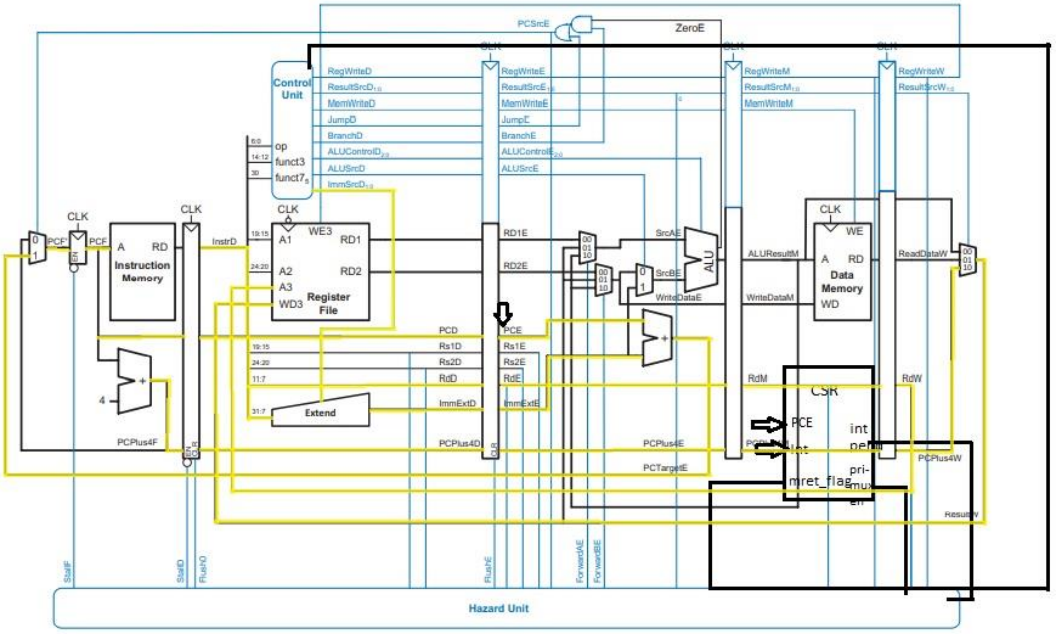
1.3.3. Highlighted Datapath for S-type Instructions



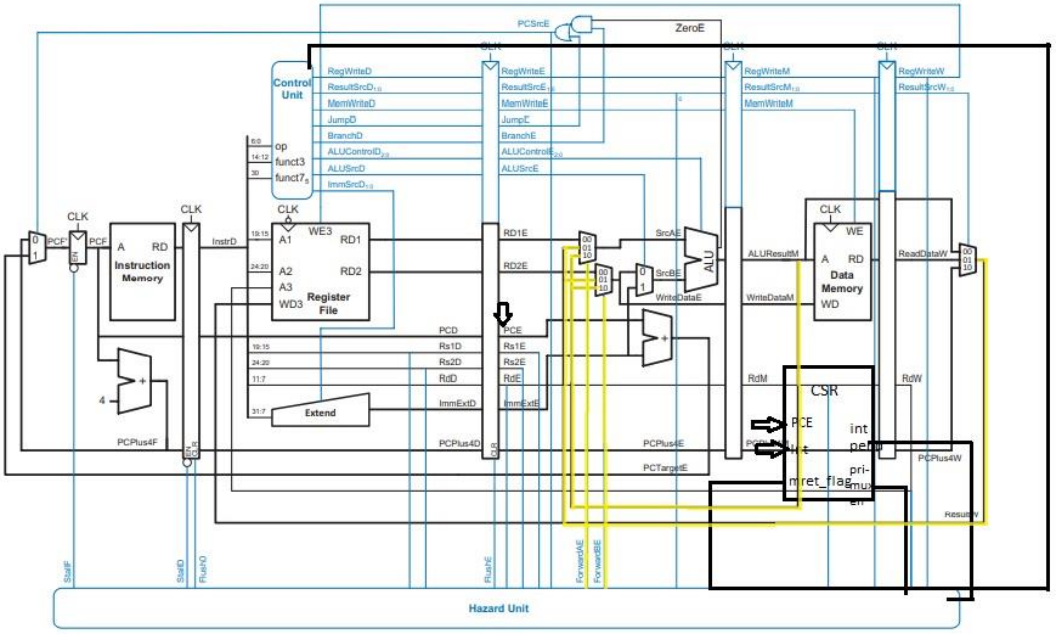
1.3.4. Highlighted Datapath for B-type Instructions



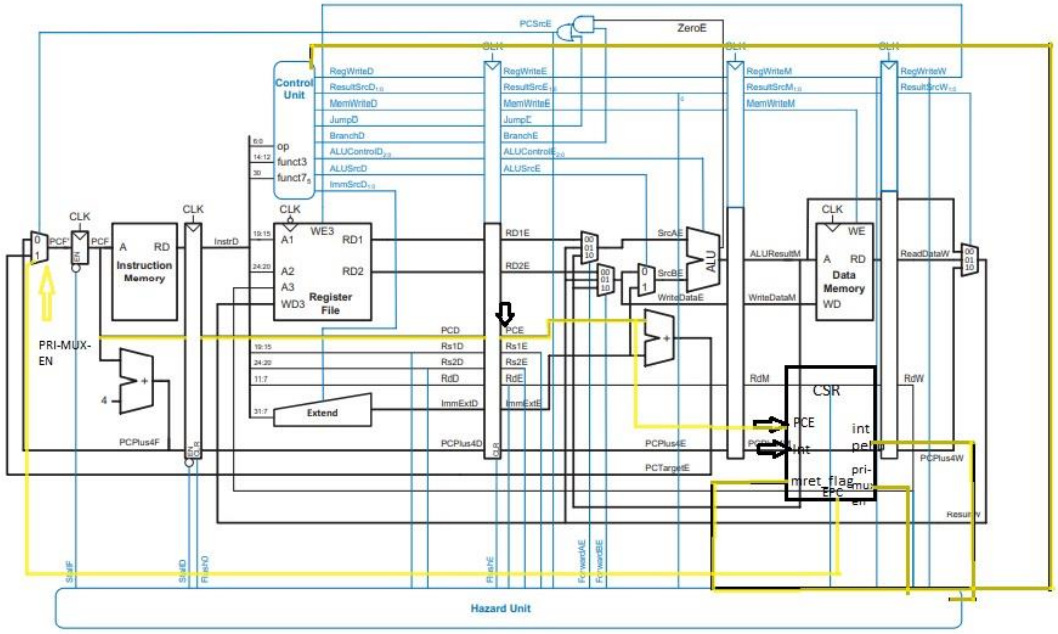
1.3.5. Highlighted Datapath for J-type Instructions



1.3.6. Highlighted Datapath for Hazards



1.3.7. Highlighted Datapath for CSR Instructions



2. Modules

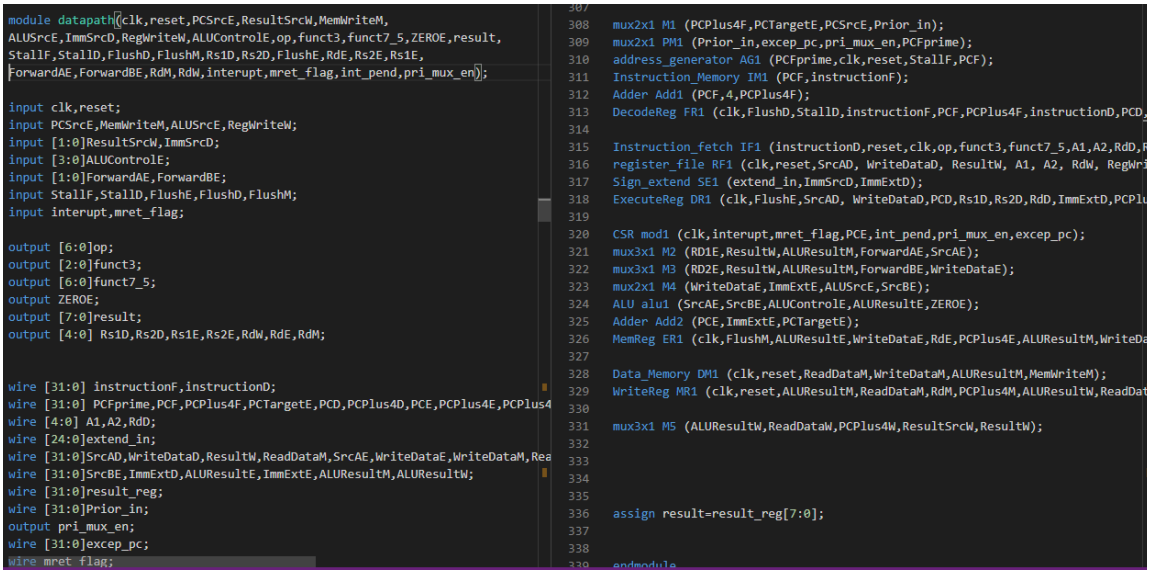
The step by step modular description is as follows:

2.1. The Top module

The pipeline processor is subdivided into three big modules which are the Datapath, controller and the hazard unit.

```
module pipeline_riscv(clk,reset,interupt,result);
input clk,reset,interupt;
output [7:0] result;


wire PCSrcE,MemWriteM,ALUSrcE,RegWriteW;
wire [1:0]ResultSrcW,ImmSrcD;
wire [3:0]ALUControlE;
wire [6:0]op;
wire [2:0]funct3;
wire [6:0]funct7_5;
wire ZEROE;
wire [4:0] Rs1E,Rs2E,RdM,RdW,RdE,Rs1D,Rs2D;
wire [1:0]ForwardAE,ForwardBE;
wire RegWriteM;
wire StallF,StallD,FlushE,FlushD,int_pend,pri_mux_en,FlushM;
wire [1:0] ResultSrcE;

datapath D1 (clk,reset,PCSrcE,ResultSrcW,MemWriteM,ALUSrcE,ImmSrcD,RegWriteW,ALUControlE,
op,funct3,funct7_5,ZEROE,result,StallF,StallD,FlushD,FlushM,Rs1D,Rs2D,FlushE,RdE,Rs2E,Rs1E,
ForwardAE,ForwardBE,RdM,RdW,interupt,mret_flag,int_pend,pri_mux_en);

controller C1(clk,reset,op,funct3,funct7_5,ZEROE,PCSrcE,ResultSrcW,MemWriteM,ALUSrcE,ImmSrcD,
RegWriteW,ALUControlE,ResultSrcE,RegWriteM,FlushE,mret_flag);
HazardUnit HU1 (StallF,StallD,FlushD,Rs1D,Rs2D,FlushE,RdE,Rs2E,Rs1E,PCSrcE,ForwardAE,
ForwardBE,ResultSrcE[0],RdM,RegWriteM,RdW,RegWriteW,int_pend,pri_mux_en,FlushM);

endmodule
```

The data path consists of complete components such as data memory, instruction memory, register file, ALU, muxes, program counter, control and status register. The control unit consists of all the control signals that are used to regulate the working of these components during the flow of the program. Different wires are attached to each components for interconnection.



```
module datapath(clk,reset,PCSrcE,ResultSrcW,MemWriteM,
ALUSrcE,ImmSrcD,RegWriteW,ALUControlE,op,funct3,funct7_5,ZEROE,result,
StallF,StallD,FlushD,FlushM,Rs1D,Rs2D,FlushE,RdE,Rs2E,Rs1E,
ForwardAE,ForwardBE,RdM,RdW,interupt,mret_flag,int_pend,pri_mux_en);

input clk,reset;
input PCSrcE,MemWriteM,ALUSrcE,RegWriteW;
input [1:0]ResultSrcW,ImmSrcD;
input [3:0]ALUControlE;
input [1:0]ForwardAE,ForwardBE;
input StallF,StallD,FlushE,FlushD,FlushM;
input interupt,mret_flag;

output [6:0]op;
output [2:0]funct3;
output [6:0]funct7_5;
output ZEROE;
output [7:0]result;
output [4:0] Rs1D,Rs2D,Rs1E,Rs2E,RdW,RdE,RdM;


wire [31:0] instructionF,instructionD;
wire [31:0] PCFprime,PCF,PCPlus4F,PCTargetE,PCD,PCPlus4D,PCE,PCPlus4E,PCPlus4
wire [4:0] A1,A2,RdD;
wire [24:0]extend_in;
wire [31:0]SrcAD,WriteDataD,ResultW,ReadDataM,SrcAE,WriteDataE,WriteDataM,Rea
wire [31:0]SrcBE,ImmExtD,ALUResultE,ImmExtE,ALUResultM,ALUResultW;
wire [31:0]result_reg;
wire [31:0]Prior_in;
output pri_mux_en;
wire [31:0]excep_pc;
wire mret_flag;
```

```
mux2x1 M1 (PCPlus4F,PCTargetE,PCSrcE,Prior_in);
mux2x1 PM1 (Prior_in,excep_pc,pri_mux_en,PCFprime);
address_generator AG1 (PCFprime,clk,reset,StallF,PCF);
Instruction_Memory IM1 (PCF,instructionF);
Adder Add1 (PCF,4,PCPlus4F);
DecodeReg FR1 (clk,FlushD,StallD,instructionF,PCF,PCPlus4F,instructionD,PCD,

Instruction_fetch IF1 (instructionD,reset,clk,op,funct3,funct7_5,A1,A2,RdD,R
register_file RF1 (clk,reset,SrcAD, WriteDataD, ResultW, A1, A2, RdW, RegWri
Sign_extend SE1 (extend_in,ImmSrcD,ImmExtD);
ExecuteReg DR1 (clk,FlushE,SrcAD, WriteDataD,PCD,Rs1D,Rs2D,RdD,ImmExtD,PCPlu

CSR mod1 (clk,interupt,mret_flag,PCE,int_pend,pri_mux_en,excep_pc);
mux3x1 M2 (RD1E,ResultW,ALUResultM,ForwardAE,SrcAE);
mux3x1 M3 (RD2E,ResultW,ALUResultM,ForwardBE,WriteDataE);
mux2x1 M4 (WriteDataE,ImmExtE,ALUSrcE,SrcBE);
ALU alu1 (SrcAE,SrcBE,ALUControlE,ALUResultE,ZEROE);
Adder Add2 (PCE,ImmExtE,PCTargetE);
MemReg ER1 (clk,FlushM,ALUResultE,WriteDataE,RdE,PCPlus4E,ALUResultM,WriteDa

Data_Memory DM1 (clk,reset,ReadDataM,WriteDataM,ALUResultM,MemWriteM);
WriteReg MR1 (clk,reset,ALUResultM,ReadDataM,RdM,PCPlus4M,ALUResultW,ReadDat

mux3x1 M5 (ALUResultW,ReadDataW,PCPlus4W,ResultSrcW,ResultW);




assign result=result_reg[7:0];


endmodule
```

2.2. CPU Control Unit

The control unit is mainly divided into the main Decoder and the ALU Decorder. The main Decoder gives the 12 bits control signals which comprises of the register write decode stage,immediate source decode stage, ALU source decode stage,memory write,result source,branch operation jump and mret flag. The control signals are decided on the basis of 7 bit value from the instruction. The control signals for the other registered stages which are of execute and memory stage are decided on the operation.

The ALU decoder takes the ALU operation of 2 bit value and decides whether the operation is load word store word, branching or ALU operations. The decision of which branch is decided on the 3 bit Func value that tells whether the branch is equal,not equal or less than operation.

```
module controller(clk,reset,op,funct3,funct7_5,ZEROE,PCSrcE,ResultSrcW,MemWriteM,ALUSrcE,ImmSrcD,RegWriteW,ALUControlE,
ResultSrcE,RegWriteM,FlushE,mret_flag);

input clk,reset;
output PCSrcE,MemWriteM,ALUSrcE;
output [1:0]ResultSrcW,ImmSrcD;
output [3:0]ALUControlE;
input [6:0]op;
input [2:0]funct3;
input [6:0]funct7_5;
input ZEROE;
input FlushE;

output [1:0] ResultSrcE;
output RegWriteM,RegWriteW,mret_flag;

wire [1:0] ResultSrcD,ResultSrcM;
wire MemWriteD,ALUSrcD,RegWriteD,MemWriteE,RegWriteE;
wire [3:0] ALUControlD;
wire BranchD,JumpD,BranchE,JumpE;
wire [1:0]ALUOp;

main_decoder MD1 (op, ResultSrcD, MemWriteD, BranchD,
 ALUSrcD, RegWriteD, JumpD, ImmSrcD, ALUOp,mret_flag);
ALU_decoder ALD1 (op[5], funct3, funct7_5, ALUOp, ALUControlD);
ExecuteCtrReg DCR1 (clk,FlushE,ResultSrcD, MemWriteD, BranchD,
 ALUSrcD, RegWriteD, JumpD,ALUControlD,ResultSrcE, MemWriteE, BranchE,
 ALUSrcE, RegWriteE, JumpE,ALUControlE);
MemCtrReg ECR1 (clk,reset,RegWriteE,ResultSrcE,MemWriteE,RegWriteM,ResultSrcM,MemWriteM);
WriteCtrReg MCR1 (clk,reset,RegWriteM,ResultSrcM,RegWriteW,ResultSrcW);
assign PCSrcE = (BranchE & ZEROE ) | JumpE;
```

```
module ALU_decoder(opb5, funct3, funct7_5, ALUOp, ALUControlD);
input opb5;
input [2:0]funct3;
input [6:0]funct7_5;
input [1:0] ALUOp;
output reg [3:0]ALUControlD;

wire Rtypesub;
assign Rtypesub = funct7_5[5] & opb5;

always @(*) begin
    casex (ALUOp)
        2'b00: ALUControlD = 4'b0000; //lw and sw
        2'b01: case (funct3)
            3'b000:  ALUControlD = 4'b0001; //branch equal
            3'b001:  ALUControlD = 4'b1111; //branch not equal
            3'b100:  ALUControlD = 4'b1110; //branch less than
            default: ALUControlD = 4'b0000;
        endcase


        2'b10: casex (funct3)
            3'b000: case (Rtypesub)
                0: ALUControlD = 4'b0000;  //add
                1: ALUControlD =  4'b0001;  //sub

                default: ALUControlD =4'b0000;
            endcase
            3'b100: ALUControlD = 4'b0011; //div
            3'b111: ALUControlD = 4'b1000; //and
            3'b110: ALUControlD = 4'b1001; //or
```

```
always @(*) begin
    casex (ALUOp)
        2'b00: ALUControlD = 4'b0000; //lw and sw
        2'b01: case (funct3)
            3'b000:  ALUControlD = 4'b0001; //branch equal
            3'b001:  ALUControlD = 4'b1111; //branch not equal
            3'b100:  ALUControlD = 4'b1110; //branch less than
            default: ALUControlD = 4'b0000;
        endcase


        2'b10: casex (funct3)
            3'b000: case (Rtypesub)
                0: ALUControlD = 4'b0000;  //add
                1: ALUControlD =  4'b0001;  //sub

                default: ALUControlD =4'b0000;
            endcase
            3'b100: ALUControlD = 4'b0011; //div
            3'b111: ALUControlD = 4'b1000; //and
            3'b110: ALUControlD = 4'b1001; //or
            3'b100: ALUControlD = 4'b1010; //xor
            3'b001: ALUControlD = 4'b0100; //sll
            3'b101: ALUControlD = 4'b0101; //srl
            //3'b101: ALUControlD = 4'b1110; //sgt
            default: ALUControlD = 3'bxxx;
        endcase
        default: ALUControlD = 3'bxxx;
    endcase

end
endmodule
```

```
module main_decoder(op, ResultSrcD, MemWriteD, BranchD,
 ALUSrcD, RegWriteD, JumpD, ImmSrcD, ALUOp,mret_flag);

input [6:0] op;
output [1:0] ResultSrcD;
output MemWriteD;
output BranchD, ALUSrcD;
output RegWriteD, JumpD;
output [1:0] ImmSrcD;
output [1:0] ALUOp;
output mret_flag;

reg [11:0] controls;
assign {RegWriteD, ImmSrcD, ALUSrcD, MemWriteD,
ResultSrcD, BranchD, ALUOp, JumpD,mret_flag} = controls;
//assign mret_flag = op == 7'b1110011;
always @(*) begin
casex (op)
// RegWrite_ImmSrc_ALUSrc_MemWrite_ResultSrc_Branch_ALUOp_Jump_mretflag
7'b0000011: controls = 12'b1_00_1_0_01_0_00_0_0; // lw
7'b0100011: controls = 12'b0_01_1_1_00_0_00_0_0; // sw
7'b0110011: controls = 12'b1_xx_0_0_00_0_10_0_0; // R-type
7'b1100011: controls = 12'b0_10_0_0_00_1_01_0_0; // branch
7'b0010011: controls = 12'b1_00_1_0_00_0_10_0_0; // I-type ALU
7'b1101111: controls = 12'b1_11_0_0_10_0_00_1_0; // jal
7'b1110011: controls = 12'b0_00_0_0_00_0_00_0_1; //CSR mret

    default: controls = 12'b0_00_0_0_00_0_00_0; // ???
endcase
end

endmodule
```

2.3. Memory Module

There are 2 memory modules. One is instruction memory and other one is data memory. In the instruction memory, we will include the mem file. Each instruction is fetched by dividing the program counter value with 4.The instructions in the Mem file are in hex. The instruction memory also consists of the interrupt handler code as well.

The data memory consists of 32 bits of each memory block and about 256 memory blocks. The data is stored in the respective memory block when an address is given to it. The data can also be retrieved by giving the same respective address.

```
module Instruction_Memory(input [31:0] pc,output [31:0] instruction);
// Write your code here
    reg [31:0] instruct [2825:0];
    integer i=0;
    initial begin
        $readmemh("Verify.mem",instruct);
    end
    assign instruction = instruct [pc/4];
endmodule
```

```
module Data_Memory(input clk,input reset,output reg [31:0] Data_Out, input [31:0] Data_In, input [31:0] D_Addr, input wr);
      reg [31:0] Mem [255:0];        // Data Memory

   // Write your code here
   integer i;
   always @(negedge clk) begin
      if (reset) begin
         for (i =0 ;i<256 ;i=i+1 ) begin
            Mem[i]<= 32'h0;

         end
      end
      if (wr) begin
         Mem[D_Addr] <= Data_In;
      end
      Data_Out <= Mem[D_Addr];

   end



endmodule
```

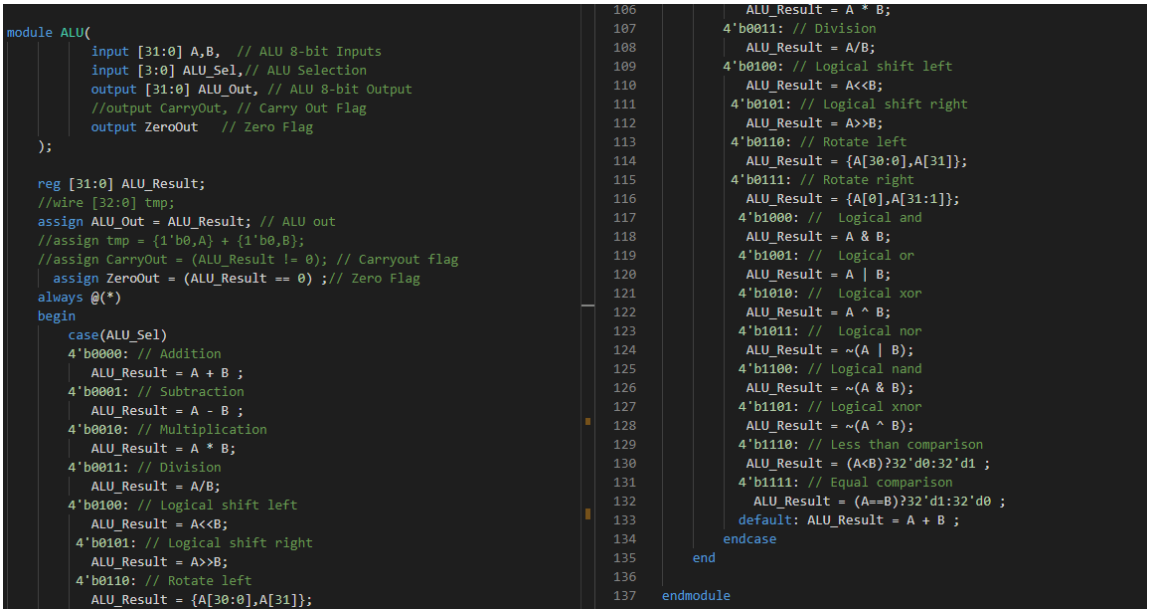
2.4. Register File

Register file consists of 32 registers in which the first register x0 is hardwired to ground. The input to register file three lines of 5 Bits each which are address A, address B and address Write. It has two output ports each of 32 bits. The input is also a 32 bit data which will be written to the third address line. The register file will be controlled by the register write control signal. The Verilog code of the register file is:

```
module register_file(clk,reset,Port_A, Port_B, Din, Addr_A, Addr_B, Addr_Wr, wr,result_reg);
            output reg [31:0] Port_A, Port_B;           // Data to be provided from register to execute the instruction
            output [31:0]result_reg;
            input [31:0] Din;                       // Data to be loaded in the register
            input [4:0] Addr_A, Addr_B, Addr_Wr;    // Address (or number) of register to be written or to be read
            input wr,clk,reset;                             // input wr flag input
            reg [31:0] Reg_File [31:0];             // Register file
            integer i=0;
// Write your code here

        always @(negedge clk) begin


            if(reset)begin
                for(i=0;i<32;i=i+1)
                    Reg_File[i]<=32'h0;
            end else if (wr && (Addr_Wr!=0)) begin
            Reg_File[Addr_Wr]=Din;
            end


    end
    always @(*) begin
        Reg_File[0]=0;
        Port_A=Reg_File[Addr_A];
        Port_B=Reg_File[Addr_B];

    end

    assign result_reg = Reg_File[5];
endmodule
```

2.5. ALU

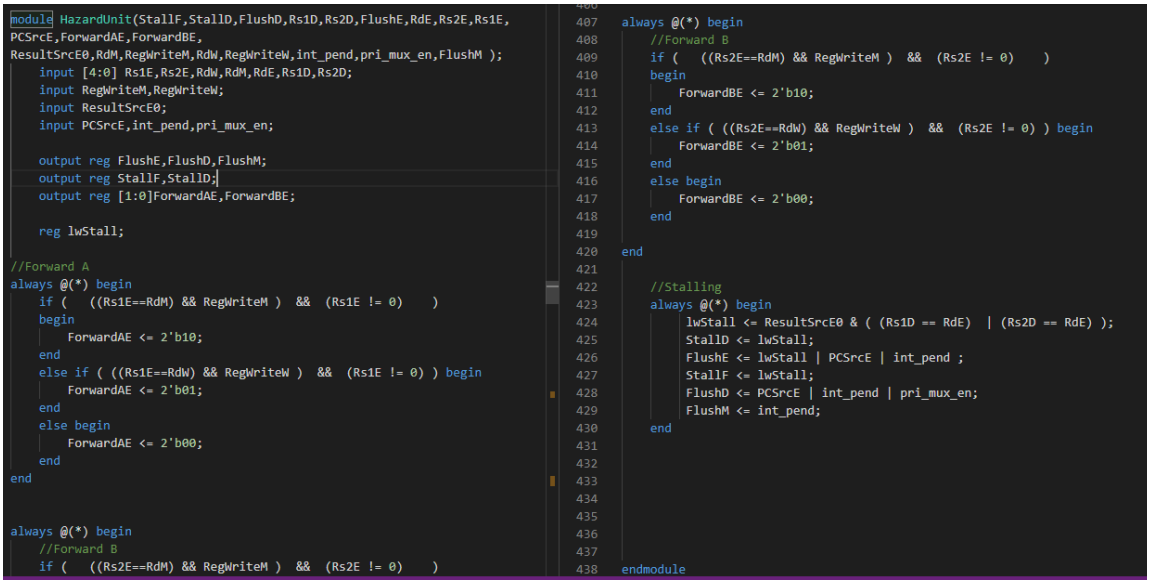
The ALU known as arithmetic logic unit consists of about 16 operations. The operations can be accessed from the 4 bit Alu selection values.These 4 bit selection values are actually provided by the control unit which are determined by the instruction type. As explained in the ALU decoder above the 2 bit value of ALUop tells which type of instruction it is ,if it is an R- type instruction the further operation is determined from the funct3 values. These funct3 bit values then gives the output of the four bit control selection values of the ALU unit.



```
module ALU(
        input [31:0] A,B,  // ALU 8-bit Inputs
        input [3:0] ALU_Sel,// ALU Selection
        output [31:0] ALU_Out, // ALU 8-bit Output
        //output CarryOut, // Carry Out Flag
        output ZeroOut   // Zero Flag
    );

    reg [31:0] ALU_Result;
    //wire [32:0] tmp;
    assign ALU_Out = ALU_Result; // ALU out
    //assign tmp = {1'b0,A} + {1'b0,B};
    //assign CarryOut = (ALU_Result != 0); // Carryout flag
      assign ZeroOut = (ALU_Result == 0) ;// Zero Flag
    always @(*)
    begin
        case(ALU_Sel)
        4'b0000: // Addition
           ALU_Result = A + B ;
        4'b0001: // Subtraction
           ALU_Result = A - B ;
        4'b0010: // Multiplication
           ALU_Result = A * B;
        4'b0011: // Division
           ALU_Result = A/B;
        4'b0100: // Logical shift left
           ALU_Result = A<<B;
         4'b0101: // Logical shift right
           ALU_Result = A>>B;
         4'b0110: // Rotate left
           ALU_Result = {A[30:0],A[31]};
           ALU_Result = A * B;
        4'b0011: // Division
           ALU_Result = A/B;
        4'b0100: // Logical shift left
           ALU_Result = A<<B;
         4'b0101: // Logical shift right
           ALU_Result = A>>B;
         4'b0110: // Rotate left
           ALU_Result = {A[30:0],A[31]};
         4'b0111: // Rotate right
           ALU_Result = {A[0],A[31:1]};
         4'b1000: //  Logical and
           ALU_Result = A & B;
         4'b1001: //  Logical or
           ALU_Result = A | B;
         4'b1010: //  Logical xor
           ALU_Result = A ^ B;
         4'b1011: //  Logical nor
           ALU_Result = ~(A | B);
         4'b1100: // Logical nand
           ALU_Result = ~(A & B);
         4'b1101: // Logical xnor
           ALU_Result = ~(A ^ B);
         4'b1110: // Less than comparison
           ALU_Result = (A<B)?32'd0:32'd1 ;
         4'b1111: // Equal comparison
            ALU_Result = (A==B)?32'd1:32'd0 ;
         default: ALU_Result = A + B ;
        endcase
    end

endmodule
```

2.6. Hazard Controller

In case of pipeline processor, three types of hazards are there. Data control and structural hazard. The structural hazard can not occur in the prescribed implementation of the processor. There are three types of data hazards that can occur which are RAW,WAR and WAW. The most common one is RAW (read after write). In this hazard the next instruction actually needs the data which should be written by the previous instruction. To resolve this hazard forwarding is used. In forwarding that data which is to be used by the next instruction is given after the execute stage . In this way the processor doesn't have to stall itself for the result to be written in the register file.



```
module HazardUnit(StallF,StallD,FlushD,Rs1D,Rs2D,FlushE,RdE,Rs2E,Rs1E,
PCSrcE,ForwardAE,ForwardBE,
ResultSrcE0,RdM,RegWriteM,RdW,RegWriteW,int_pend,pri_mux_en,FlushM );
    input [4:0] Rs1E,Rs2E,RdW,RdM,RdE,Rs1D,Rs2D;
    input RegWriteM,RegWriteW;
    input ResultSrcE0;
    input PCSrcE,int_pend,pri_mux_en;

    output reg FlushE,FlushD,FlushM;
    output reg StallF,StallD;
    output reg [1:0]ForwardAE,ForwardBE;

    reg lwStall;

//Forward A
always @(*) begin
    if (   ((Rs1E==RdM) && RegWriteM )  &&  (Rs1E != 0)    )
    begin
        ForwardAE <= 2'b10;
    end
    else if ( ((Rs1E==RdW) && RegWriteW )  &&  (Rs1E != 0) ) begin
        ForwardAE <= 2'b01;
    end
    else begin
        ForwardAE <= 2'b00;
    end
end


always @(*) begin
    //Forward B
    if (   ((Rs2E==RdM) && RegWriteM )  &&  (Rs2E != 0)    )
    begin
        ForwardBE <= 2'b10;
    end
    else if ( ((Rs2E==RdW) && RegWriteW )  &&  (Rs2E != 0) ) begin
        ForwardBE <= 2'b01;
    end
    else begin
        ForwardBE <= 2'b00;
    end

end

    //Stalling
    always @(*) begin
        lwStall <= ResultSrcE0 & ( (Rs1D == RdE)  | (Rs2D == RdE) );
        StallD <= lwStall;
        FlushE <= lwStall | PCSrcE | int_pend ;
        StallF <= lwStall;
        FlushD <= PCSrcE | int_pend | pri_mux_en;
        FlushM <= int_pend;
    end


endmodule
```

2.7. Forwarding Unit

The concept of forwarding unit is a subset of hazard controller. The condition of the hazard unit actually detects the hazard. The condition compares the memory and write back stage of source registers. If they match the result is forwarded from the execute stage. So bye the detection of the hazard is done by controller and forwarding unit executes the forwarding.

2.8. CSR Controller

The CSR (control and status register) controller is used to detect the interrupt and executing the respective interrupt handler. When an interrupt occurs, the current program counter value is stored. The interrupt pending flag is enabled which enables the priority mux. When the priority mux is enabled to the CSR, the address of the interrupt handler is given to the mux which is then given to the program counter. The processor then executes the interrupt service routine. Once the interrupt service routine is completed machine

return instruction is used (mret). The mret flag is enabled and the EPC (Exception PC) is given the the return address of the stored program counter value. The processor will then continue the normal operation.

```
module CSR(clk,interupt,mret_flag,PCE,int_pend,pri_mux_en,excep_pc);
input clk,interupt,mret_flag;
input [31:0]PCE;

output reg int_pend;
output pri_mux_en;
output reg [31:0]excep_pc;

reg [31:0] ret_addr;

always @(posedge interupt) begin
    int_pend <= 1'b1;
    ret_addr <= PCE;
    excep_pc <= 96;
end

assign pri_mux_en = int_pend | mret_flag;

always @(posedge clk) begin
    int_pend <= 1'b0;
end

always @(*) begin
    if (mret_flag) begin
        excep_pc <= ret_addr;
    end
    else begin
        excep_pc <= 96;
    end
end
endmodule
```

2.9. CSR Control Signals

The CSR control signals are interrupt,mret_flag,program counter value of execute stage,interrupt pending flag, priority mux enable and exception PC address.

3. Tests

In this chapter, the processor verification is explained which is done using Cadence Xcelium.

3.1. Simulation Tool Description

Xcelium is the EDA industry's first production-ready third generation simulator. Based on innovative multi-core technology, Xcelium allows SoCs to get from design to market in record time. With Xcelium, one can expect up to 5X improved multi-core performance, and up to 2X speed-up for single-core use cases.
Sometimes for simulation purposes, Xilinx Vivado is also used which is the core tool for designing as well as implementing testbenches of different modules in this course.

3.2. Steps to run a basic modular test

The Cadence Xcelium runs on Linux terminal. Following are the steps used for implementation:

- "xrun" utility command is used to run Xcelium. The syntax of the command is:
  xrun pipeline_riscv.v pipeline_riscv_tb.v verify.mem -access +rwc -gui &
- -access +rwc provides probing access to all the signals in the design hierarchy.
- Once the simulator is opened, the modules are run and the waveform windows is shown with complete signals.
- The code and block coverage is then determined using the integrated metrics centre (IMC). The percentage graphs show how well the design is exercised by the testbench.

3.3. Test performed

The test bench consist of different types of instruction which are I type ,R type,store word , load word and toggle instructions etc. The instructions are generated randomly using a python script. The generated instructions are more than 2800. Sample of a python script is as:

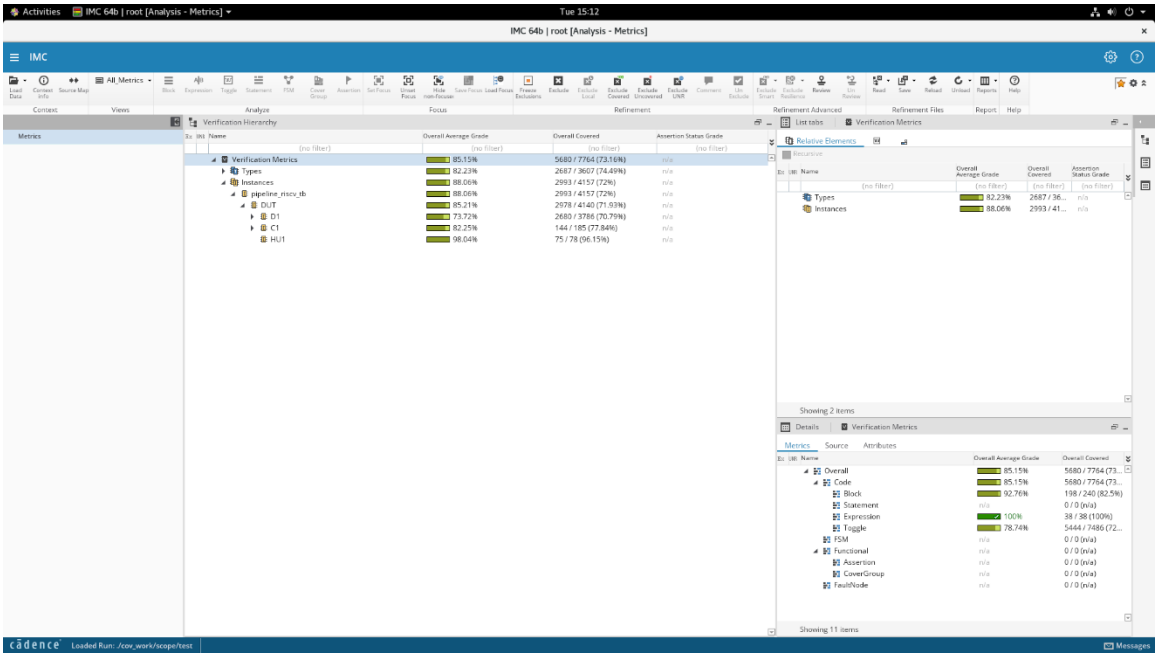
```python
#Sw.py
import random
import time

def sw_r():
    out = ""
    random.seed(time.time())
    for i in range(0, 256):
        register = random.randint(0, 31)
        out += ("sw x{}, {}(x0)".format(register, i)) + "\n"
    return out
```

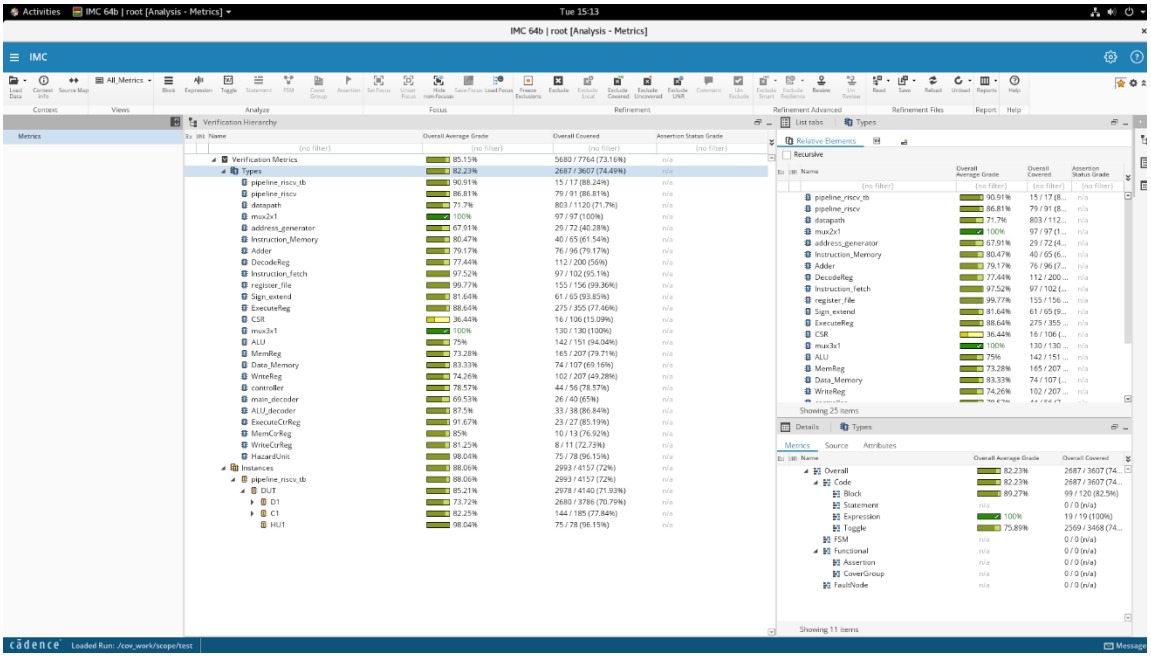
```python
#Toggle
def toggle_r():
    out = ""
    out += "ADDI x1, x0, 1" + "\n"
    for i in range(2, 32):
        out += ("ADDI x{}, x0, 1".format(i)) + "\n"
        for j in range(0, 32):
            out += ("SLL x{}, x{}, x1".format(i, i)) + "\n"
    out += "ADDI x2, x0, 1" + "\n"
    for j in range(0, 32):
        out += "SLL x1, x1, x2" + "\n"
    return out
```

```python
#itype.py
def itype_r():
    out = ""
    for i in range(0, 32):
        for j in range(0, 32):
            out += ("ADDI x{}, x{}, {}".format(i, j, random.randint(-2048, 2047))) + "\n"
    return out
```

3.4. Correctness Chart

The correctness chart shows and overall coverage and correctness of the processor up to 85.15%. This is a pretty decent coverage. The code coverage is 85% consisting of 92.7 6% marriage and 100% expression. The toggle coverage is a little less of up to 78.74%. The toggle coverage can be improved by including more toggle instructions that can toggle each and every bit of the register files,the data memory of the processor.



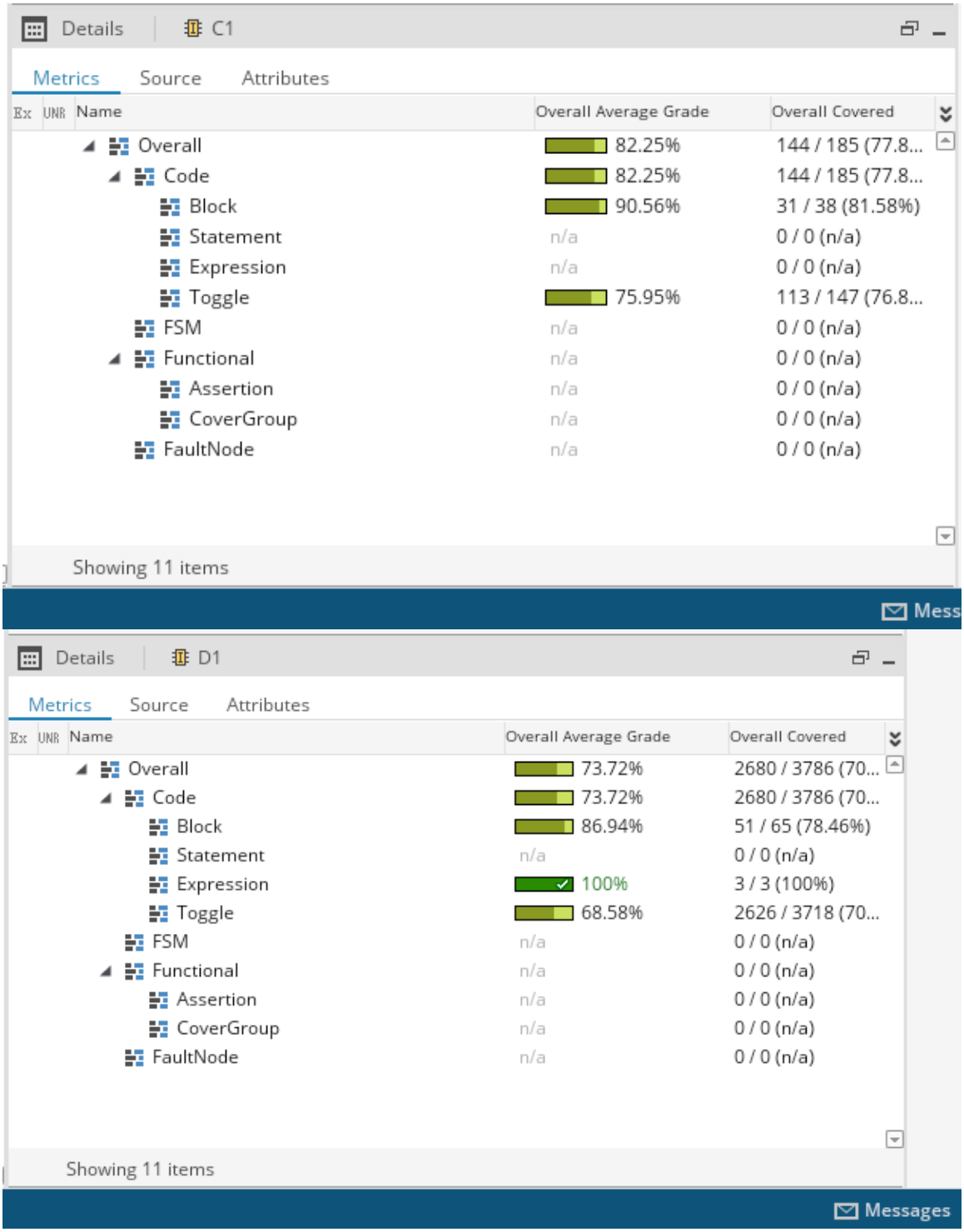


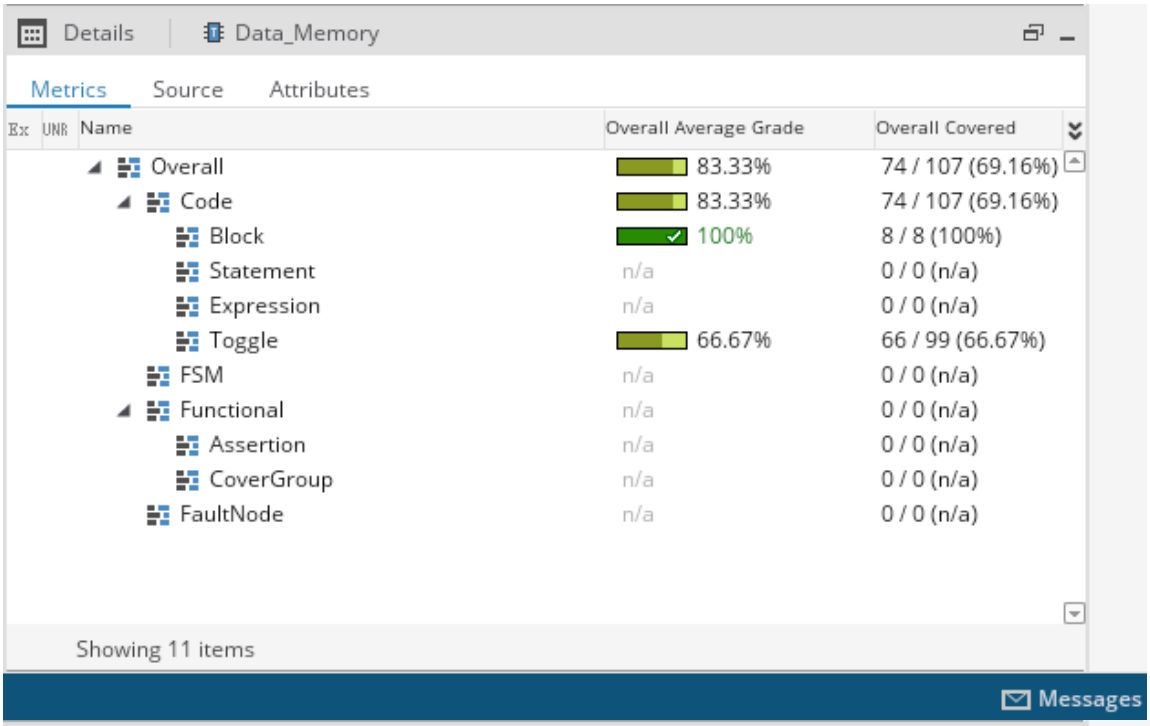
The correctness chart shows and overall coverage and correctness of the processor up to 85.15%. This is a pretty decent coverage. The code coverage is 85% consisting of 92.7 6% marriage and 100% expression. The toggle coverage is a little less of up to 78.74%. The toggle coverage can be improved by including more toggle instructions that can toggle each and every bit of the register files,the data memory of the processor. The coverage of each type of component of the processor is also shown in which the Muxes give the maximum 100% coverage, next are the pipeline registers and then the memories. Unfortunately the CSR gave very minimum coverage. The reason is due to no including of sufficient test instructions to check it rigorously.

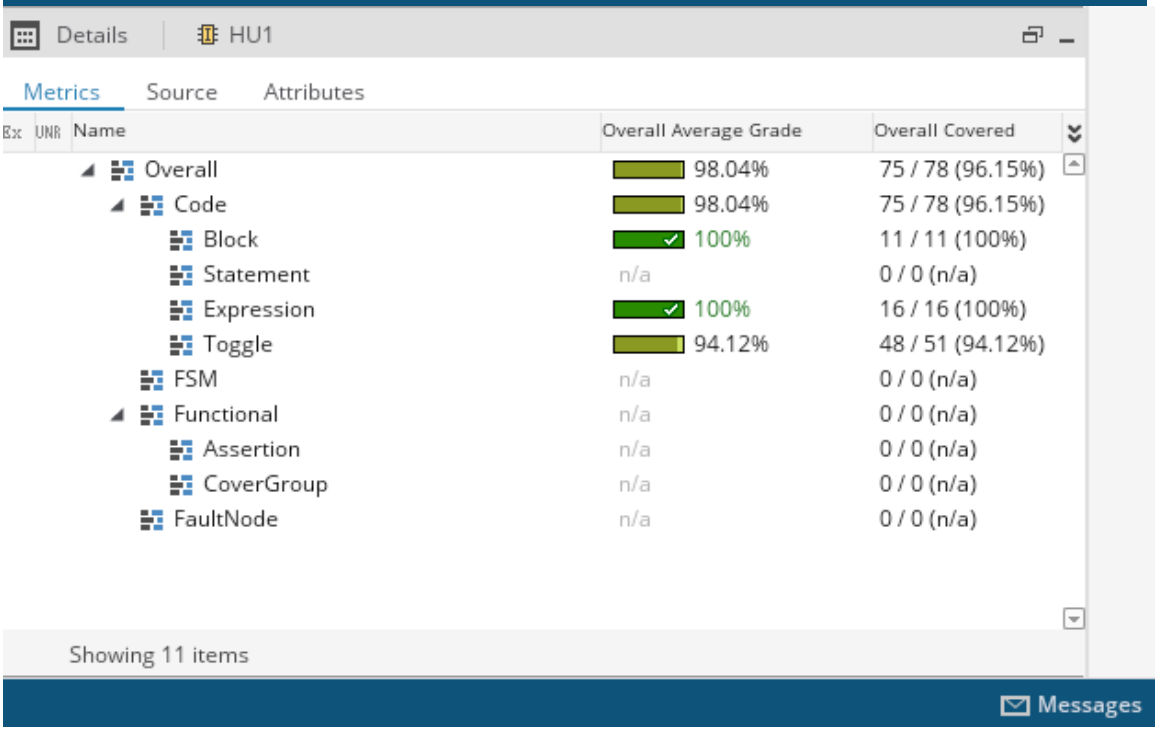
3.5. Coverage Report

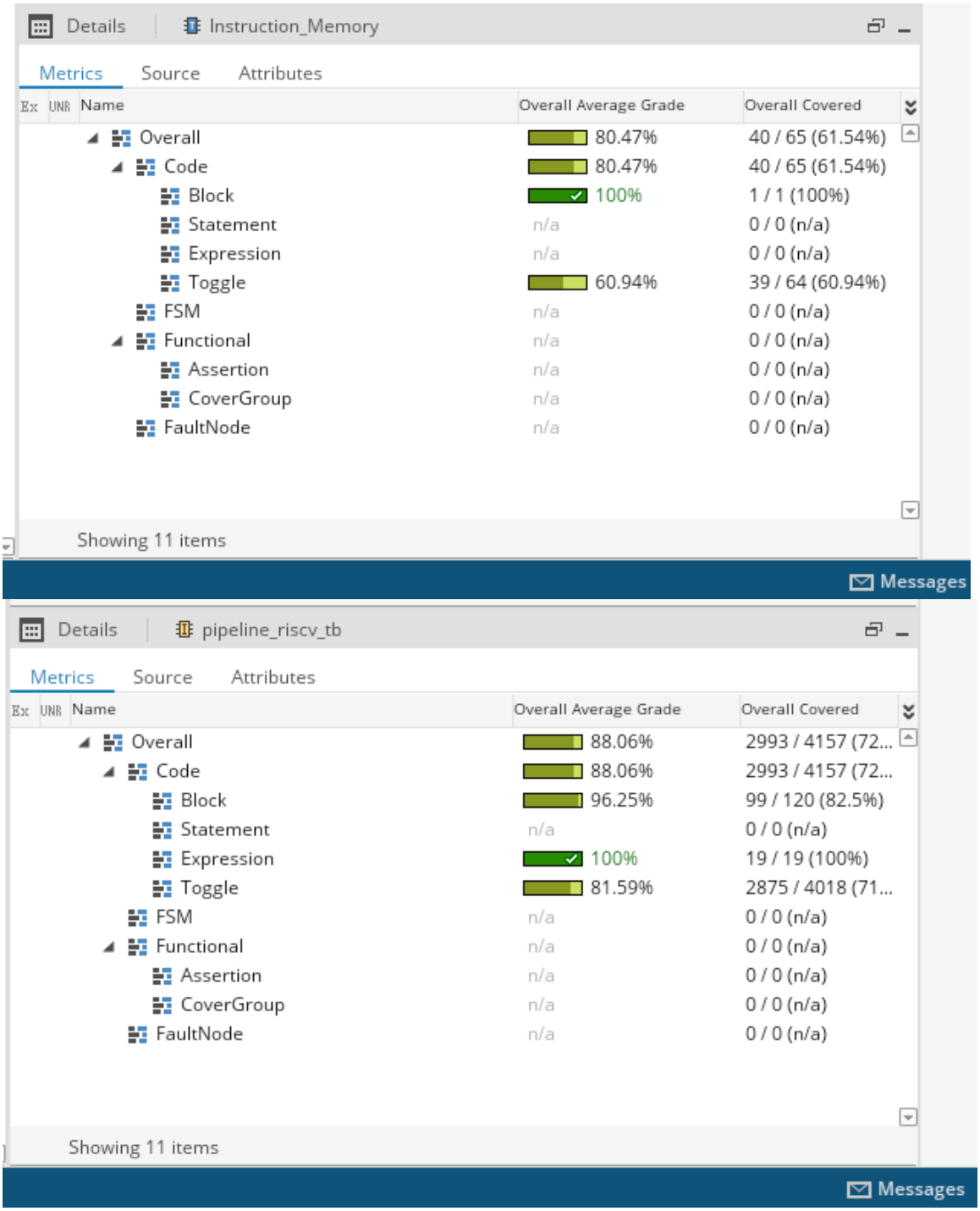
The coverage report of main components is given below. Each component's overall coverage consists of block ,expression and toggle coverage. The coverage report of

controller,datapath,hazard unit,instruction and data memory is given along with pipeline test bench also.




Details
Data_Memory
Metrics
Source
Attributes
Ex
UNR
Name
Overall Average Grade
Overall Covered
Overall 83.33% 74 / 107 (69.16%)
Code 83.33% 74 / 107 (69.16%)
Block 100% 8 / 8 (100%)
Statement n/a 0 / 0 (n/a)
Expression n/a 0 / 0 (n/a)
Toggle 66.67% 66 / 99 (66.67%)
FSM n/a 0 / 0 (n/a)
Functional n/a 0 / 0 (n/a)
Assertion n/a 0 / 0 (n/a)
CoverGroup n/a 0 / 0 (n/a)
FaultNode n/a 0 / 0 (n/a)
Showing 11 items
Messages


Details
HU1
Metrics
Source
Attributes
Ex
UNR
Name
Overall Average Grade
Overall Covered
Overall 98.04% 75 / 78 (96.15%)
Code 98.04% 75 / 78 (96.15%)
Block 100% 11 / 11 (100%)
Statement n/a 0 / 0 (n/a)
Expression 100% 16 / 16 (100%)
Toggle 94.12% 48 / 51 (94.12%)
FSM n/a 0 / 0 (n/a)
Functional n/a 0 / 0 (n/a)
Assertion n/a 0 / 0 (n/a)
CoverGroup n/a 0 / 0 (n/a)
FaultNode n/a 0 / 0 (n/a)
Showing 11 items
Messages


Details
Instruction_Memory
Metrics
Source
Attributes
Ex
UNR
Name
Overall Average Grade
Overall Covered
Overall 80.47% 40 / 65 (61.54%)
Code 80.47% 40 / 65 (61.54%)
Block 100% 1 / 1 (100%)
Statement n/a 0 / 0 (n/a)
Expression n/a 0 / 0 (n/a)
Toggle 60.94% 39 / 64 (60.94%)
FSM n/a 0 / 0 (n/a)
Functional n/a 0 / 0 (n/a)
Assertion n/a 0 / 0 (n/a)
CoverGroup n/a 0 / 0 (n/a)
FaultNode n/a 0 / 0 (n/a)
Showing 11 items
Messages
Details
pipeline_riscv_tb
Metrics
Source
Attributes
Ex
UNR
Name
Overall Average Grade
Overall Covered
Overall 88.06% 2993 / 4157 (72...
Code 88.06% 2993 / 4157 (72...
Block 96.25% 99 / 120 (82.5%)
Statement n/a 0 / 0 (n/a)
Expression 100% 19 / 19 (100%)
Toggle 81.59% 2875 / 4018 (71...
FSM n/a 0 / 0 (n/a)
Functional n/a 0 / 0 (n/a)
Assertion n/a 0 / 0 (n/a)
CoverGroup n/a 0 / 0 (n/a)
FaultNode n/a 0 / 0 (n/a)
Showing 11 items
Messages

3.6. Challenges and Limitations

3.6.1. Challenges

- The lab work was manageable but due to weak background of verilog implementation the students have to face a lot of difficulties and a lot of time is consumed to actually master the verilog in order for the single cycle implementation.

- Writing verilog is very time consuming. A GUI tool must be practiced that can make the work easy by generating the required verilog code just by dragging and dropping the required components.

- The CSR implementation was challenging because of its complexity.

3.6.2. Limitations

- The CSR implementation is not so efficient. The CSR that I have implemented does not consist of CSR registers such as mcause and mstatus etc. The implementation of the presented CSR is only a prototype of how interupts work.

- The processor does not support complete riscV ise architecture. That means it does not support complete integer instructions. Only a few important of them are supported.

- The processor does not support nested vector interrupt control.

- The data memory is not byte addressable the user has the only option of load and store word. Halfword cannot be accessed.

- The memory is implemented using read memh function which considers the memory ideal so the processor is actually not tested and implemented on the hardware FPGA.

BIBLIOGRAPHY

[1] Functional Verification Blogs, Cadence <https://community.cadence.com/

APPENDIX

Complex Engineering Problem Attributes

<table>
<tr><td rowspan="5">WP1: Depth of knowledge
WP2: Range of conflicting requirements
WP3: Depth of analysis
WP4: Familiarity of issues
WP5: Extent of applicable codes
WP6: Extent of stakeholders
WP7: Interdependence</td><td colspan="2">WP1: Depth of Knowledge Requires knowledge of Digital Systems and Computer Architecture (WK4), design of Datapath and Controller for pipelined architecture (WK5), use of Modern Tools (Xilinx Vivado 2020.1) to program and test the code on FPGA (WK6) and engagement in research literature (WK8)
WP3: Depth of analysis Numerous approaches can be adopted. Choice of the selected algorithm requires in-detail analysis
WP5: Extent of applicable codes Requires going beyond the use of standardized features of Standard Verilog modules.</td></tr>
<tr><td colspan="2">Rubrics</td></tr>
<tr><td>Methodically investigates different design approaches and techniques to design datapath of pipelined microprocessor architecture. (CLO1)</td><td>WP1, WP3</td></tr>
<tr><td>Selects an optimal methodology to design the control portion of a processor, and its data path consisting of the general-purpose register file, ALU, the memory interface, internal registers between stages, and multiplexers with proper justification (CLO2)</td><td>WP3, WP5</td></tr>
<tr><td>Designs a comprehensive testbench system to test and verify the designed processor requirements subject to the given constraints (CLO3)</td><td>WP1, WP5</td></tr>
<tr><td>Identifies limitations and implications of the proposed solution (CLO3)</td><td>WP1</td></tr>
</table>

EA1: Range of resources EA2: Level of interaction EA3: Innovation EA4: Consequences for society and environment EA5: Familiarity	• ***EA1: Range of resources*** The design involves internet resources, information related to software usage and technology (modern end tools including Cadence). • ***EA3: Innovation*** Addressing sustainability and optimization of the design based on engineering principles and knowledge.	
	Rubrics	
	Methodically investigates different design approaches and techniques to design datapath of pipelined microprocessor architecture. (CLO1)	**EA1, EA3**
	Selects an optimal methodology to design the control portion of a processor, and its data path consisting of the general-purpose register file, ALU, the memory interface, internal registers between stages, and multiplexers with proper justification (CLO2)	**EA1, EA3**
	Designs a comprehensive testbench system to test and verify the designed processor requirements subject to the given constraints (CLO3)	**EA1, EA3**
	Identifies limitations and implications of the proposed solution (CLO3)	**EA3**

Complex Engineering Problem Rubrics Distribution

	Assessment Tool	**CLOs**	**WP**	**EA**	**Marks Distribution**
Grading Policy	Phase – I: Basic Constraint Design (Software Evaluation)	CLO1	**WP1, WP3**	**EA1, EA3**	5 (In Lab)
	Phase – II: Adding Hazards (Software Evaluation & Viva)	CLO2	**WP3, WP5**	**EA1, EA3**	5 (In Lab)
	Phase – III: Adding CSR Support (Software Evaluation)	CLO3	**WP1, WP5**	**EA1, EA3**	5 (In Lab)
	Final Hardware Evaluation on FPGA with External Viva	CLO3	**WP1**	**EA3**	25 (In Lab)

	Comprehensive Report	CLO1, CLO2	**WP1, WP3, WP5**	**EA1, EA3**	10 (In Lab)

Complex Engineering Problem Rubrics

	Good (8-10)	Average (4-7)	Bad (0-3)
Report (10 Marks)	References properly added in IEEE Format Template has been followed with all required sections.	References properly added in IEEE Format Template has not been followed and formatting is not properly done but all sections have been covered	If any section is missing.
Hardware Viva (10 Marks)	100% answer rate.	60-90% answer rate	0-50% answer rate
Software Viva (10 Marks)	100% Understanding.	60-90% Understanding	0-50% Understanding
Software Evaluation (15 Marks)	All Tests passed	RTL submitted with no errors but missing some tests	RTL has errors or is not complete
Hardware Working (5 Marks)	Perfectly/Minor issues in working with all constraints fulfilled	Working with not able to fulfill few constraints	Not Working/Working with not able to fulfill many constraints