RISC V Pipelined Processor

Final Report

Submitted by:
Yousuf Aijaz
Shem Eziekel
Sufiyan Aman

Research Assistant:
Ms. Saima Shaheen

Computer Engineering
Habib University
Fall Semester'24

Date: December 02, 2024

A report submitted in fulfillment
of the requirements for the lab project
of EE/CS - 371/330: Computer Architecture

1 Introduction

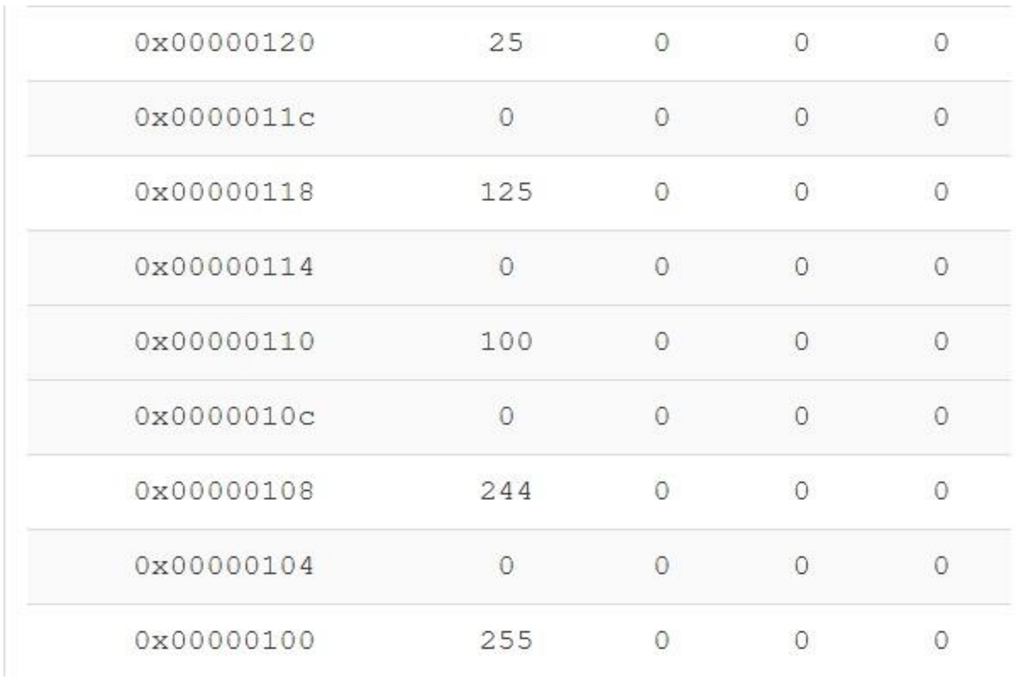
Our project involved building a 5-stage pipelined RISC-V processor capable of executing a bubble sorting algorithm. The tasks we performed included:

1. Converting the bubble sort pseudocode into RISC-V assembly code and verifying it on Venus, as well as modifying the Lab 11 single-cycle processor to execute the bubble sort code.
2. Pipelining the processor and testing individual instructions separately to ensure the functionality of the pipelined version.
3. Implementing hazard detection circuitry to identify hazards (data, control, and structural) and handling them in hardware through forwarding, stalling, and flushing the pipeline.
4. Comparing the execution time of the array sorting program on a single-cycle processor with that on a pipelined RISC-V processor.

2 Task 1

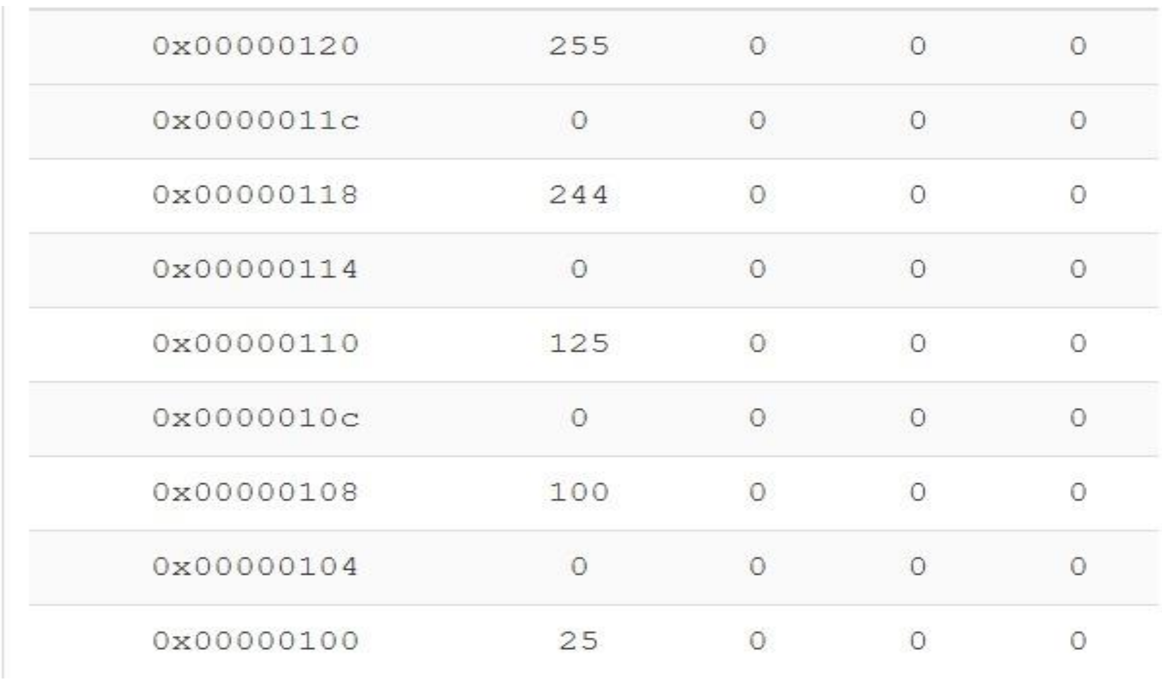
2.1 Bubble Sort Pseudocode to Machine Code

We first implemented the sorting algorithm in RISC V assembly on the Venus simulator. We had also previously tried implementing sorting in one of the labs so we modified that lab code from the pseudocode.



0x00000120	25	0	0	0
0x0000011c	0	0	0	0
0x00000118	125	0	0	0
0x00000114	0	0	0	0
0x00000110	100	0	0	0
0x0000010c	0	0	0	0
0x00000108	244	0	0	0
0x00000104	0	0	0	0
0x00000100	255	0	0	0

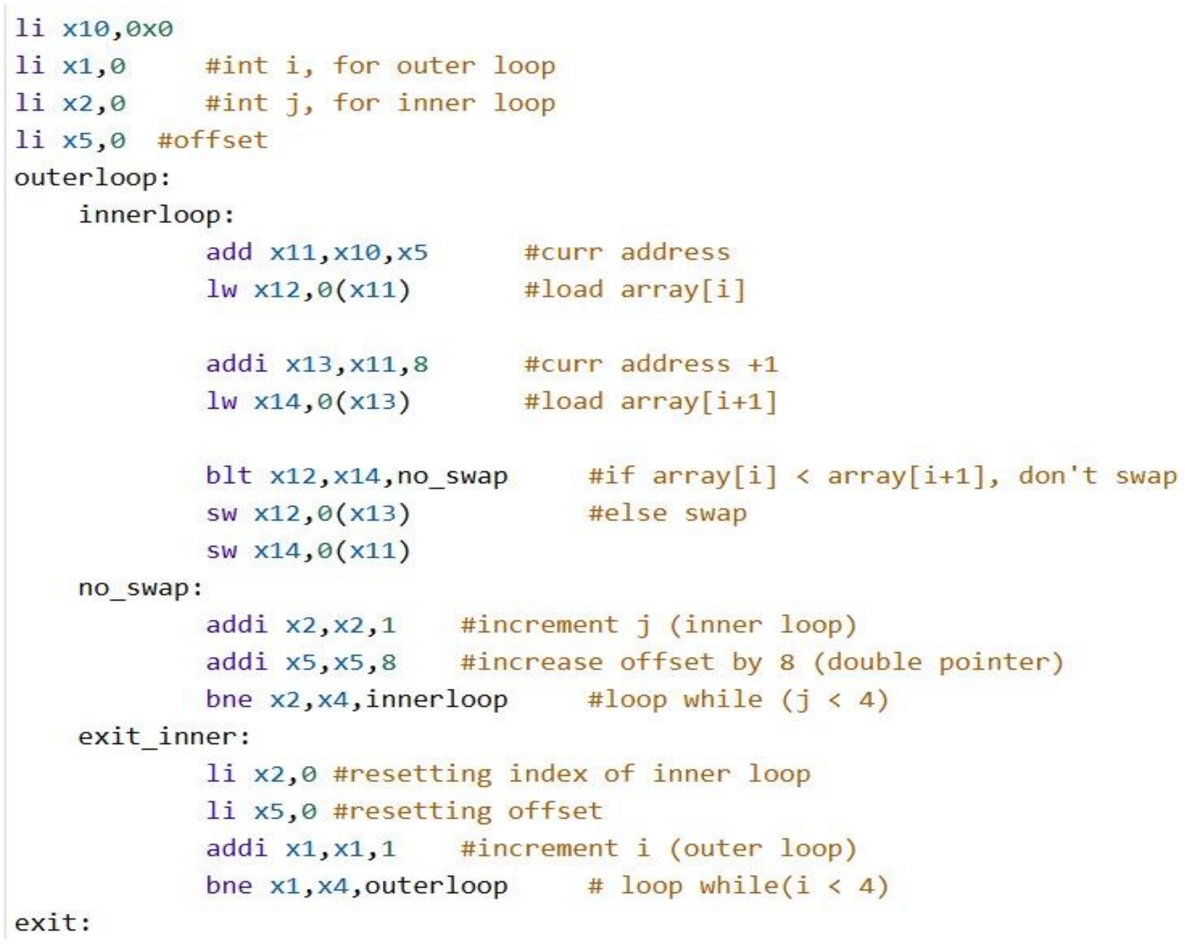
Initial Array (Before Sorting)



0x00000120	255	0	0	0
0x0000011c	0	0	0	0
0x00000118	244	0	0	0
0x00000114	0	0	0	0
0x00000110	125	0	0	0
0x0000010c	0	0	0	0
0x00000108	100	0	0	0
0x00000104	0	0	0	0
0x00000100	25	0	0	0

Sorted Array

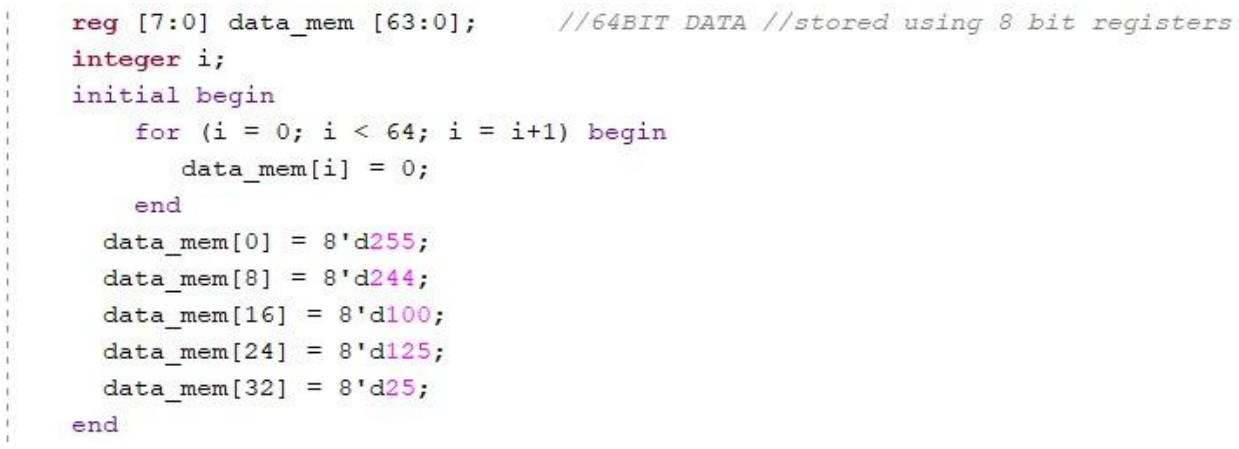
2.2 Bubble Sort Implementation Risc V:



```
li x10,0x0
li x1,0     #int i, for outer loop
li x2,0     #int j, for inner loop
li x5,0  #offset
outerloop:
    innerloop:
            add x11,x10,x5      #curr address
            lw x12,0(x11)       #load array[i]

            addi x13,x11,8      #curr address +1
            lw x14,0(x13)       #load array[i+1]

            blt x12,x14,no_swap     #if array[i] < array[i+1], don't swap
            sw x12,0(x13)           #else swap
            sw x14,0(x11)
    no_swap:
            addi x2,x2,1    #increment j (inner loop)
            addi x5,x5,8    #increase offset by 8 (double pointer)
            bne x2,x4,innerloop     #loop while (j < 4)
    exit_inner:
            li x2,0 #resetting index of inner loop
            li x5,0 #resetting offset
            addi x1,x1,1    #increment i (outer loop)
            bne x1,x4,outerloop     # loop while(i < 4)
exit:
```

2.3 Initializing values in Data Memory:

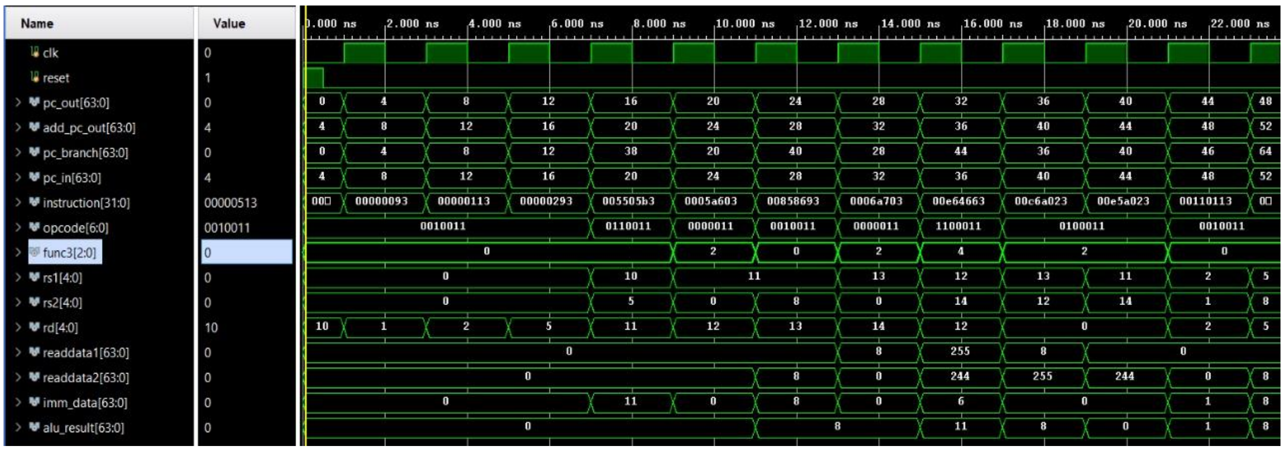
Our Data Memory initializes values in the array starting from 0x0.

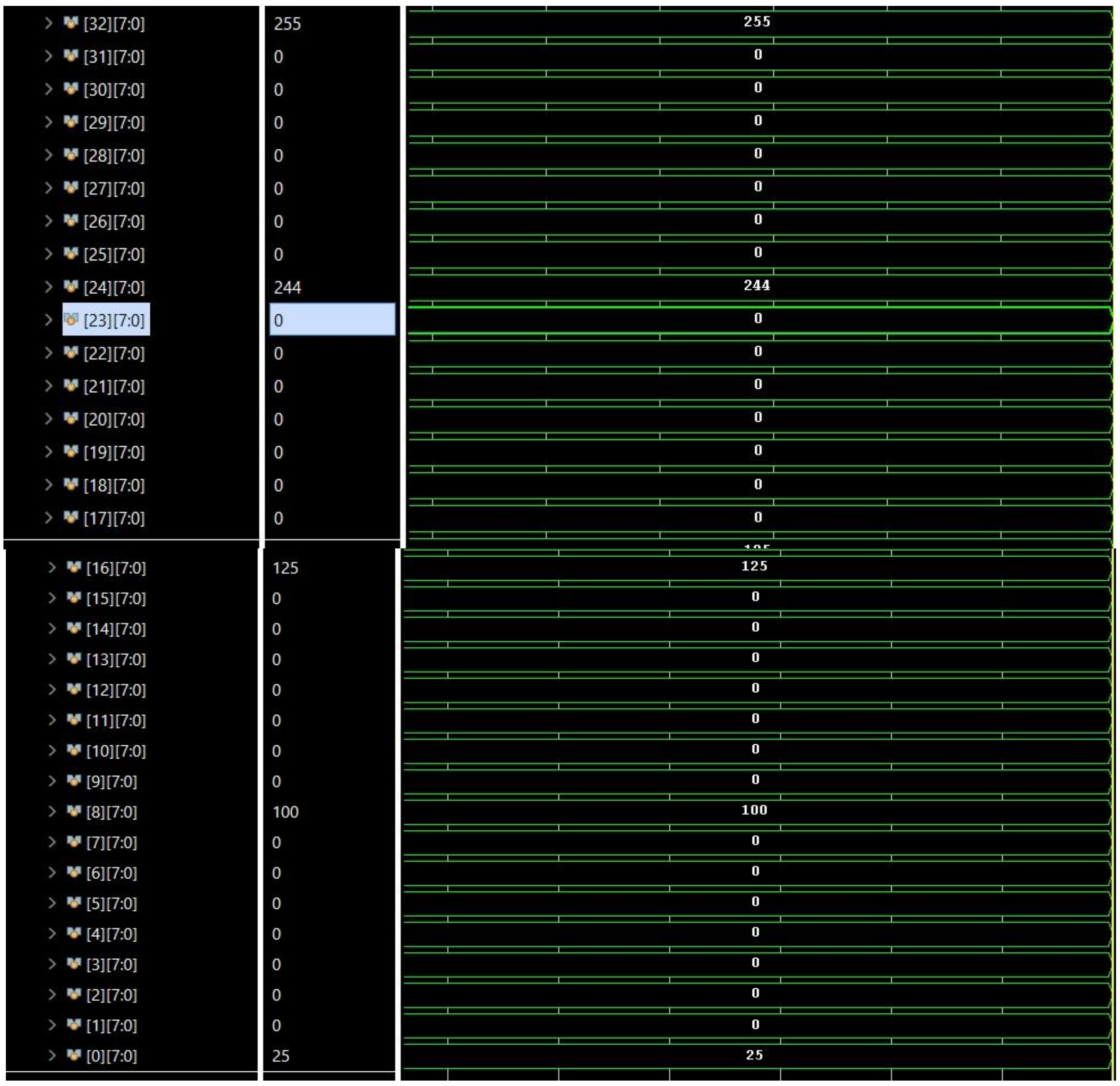


```
reg [7:0] data_mem [63:0];      //64BIT DATA //stored using 8 bit registers
integer i;
initial begin
    for (i = 0; i < 64; i = i+1) begin
      data_mem[i] = 0;
    end
  data_mem[0] = 8'd255;
  data_mem[8] = 8'd244;
  data_mem[16] = 8'd100;
  data_mem[24] = 8'd125;
  data_mem[32] = 8'd25;
end
```

Figure 2: Snippet of Data Memory

2.4 Simulation:

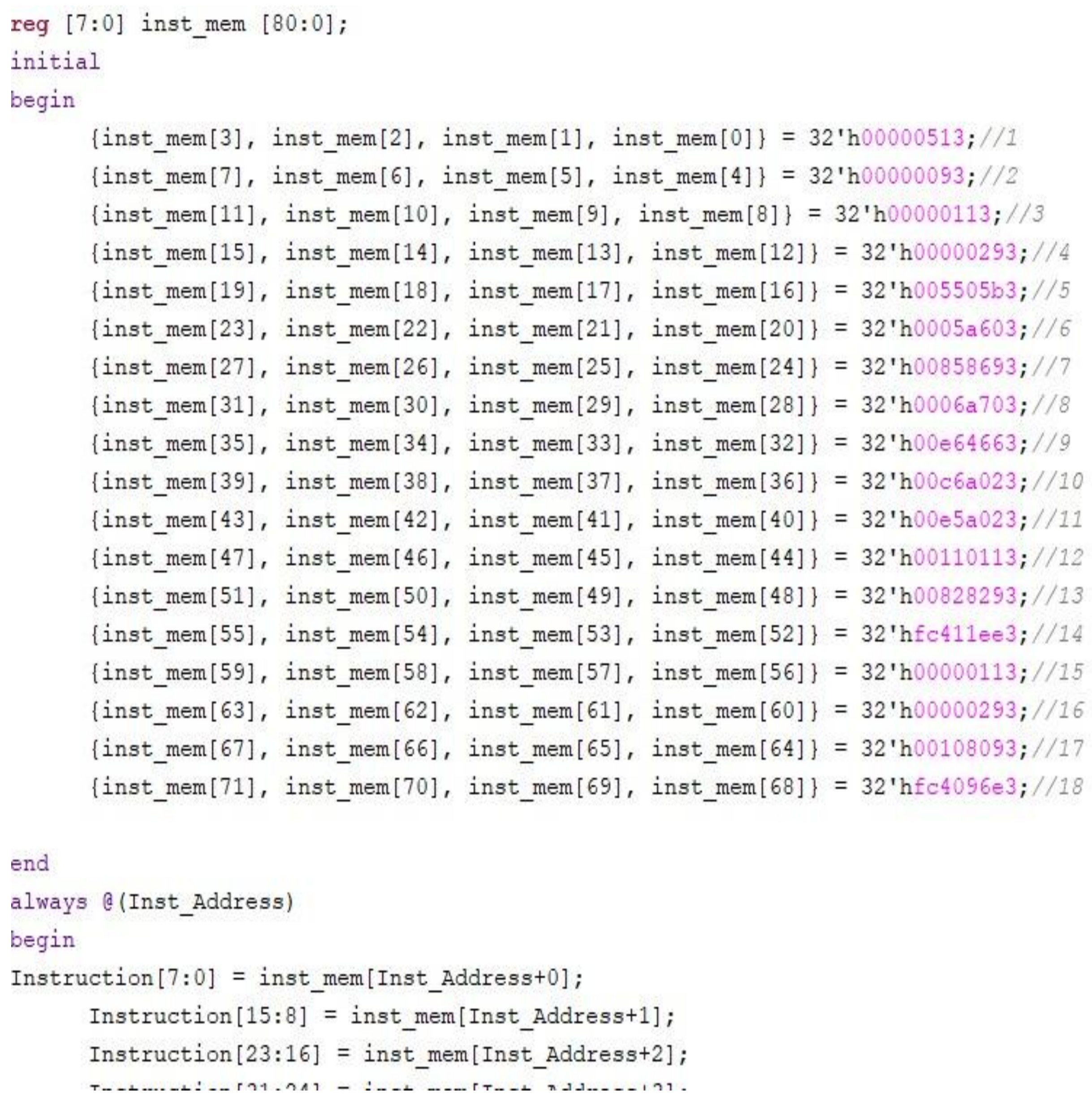




```
[32][7:0]  255
[31][7:0]  0
[30][7:0]  0
[29][7:0]  0
[28][7:0]  0
[27][7:0]  0
[26][7:0]  0
[25][7:0]  0
[24][7:0]  244
[23][7:0]  0
[22][7:0]  0
[21][7:0]  0
[20][7:0]  0
[19][7:0]  0
[18][7:0]  0
[17][7:0]  0
[16][7:0]  125
[15][7:0]  0
[14][7:0]  0
[13][7:0]  0
[12][7:0]  0
[11][7:0]  0
[10][7:0]  0
[9][7:0]   0
[8][7:0]   100
[7][7:0]   0
[6][7:0]   0
[5][7:0]   0
[4][7:0]   0
[3][7:0]   0
[2][7:0]   0
[1][7:0]   0
[0][7:0]   25
```

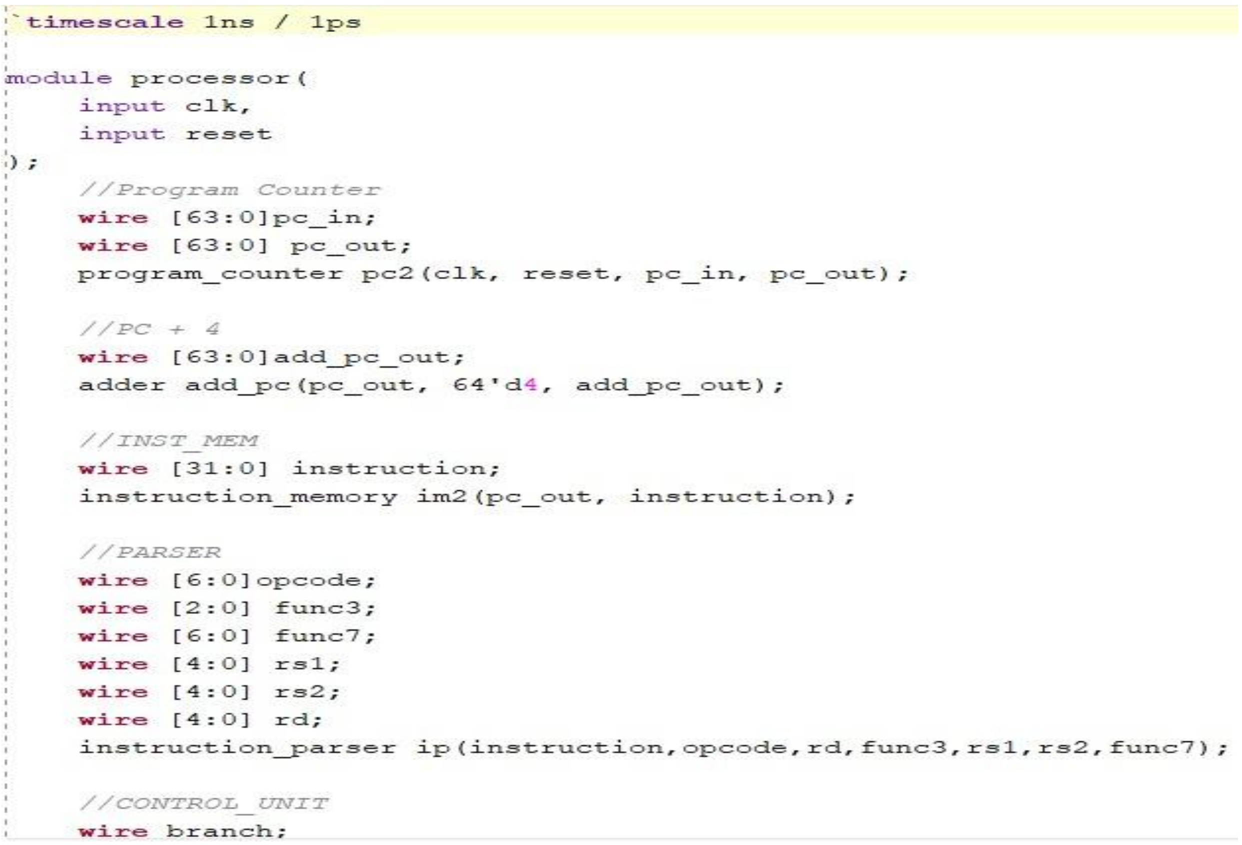
Figure 3: Sorted array

2.5 Instruction Memory initialized:

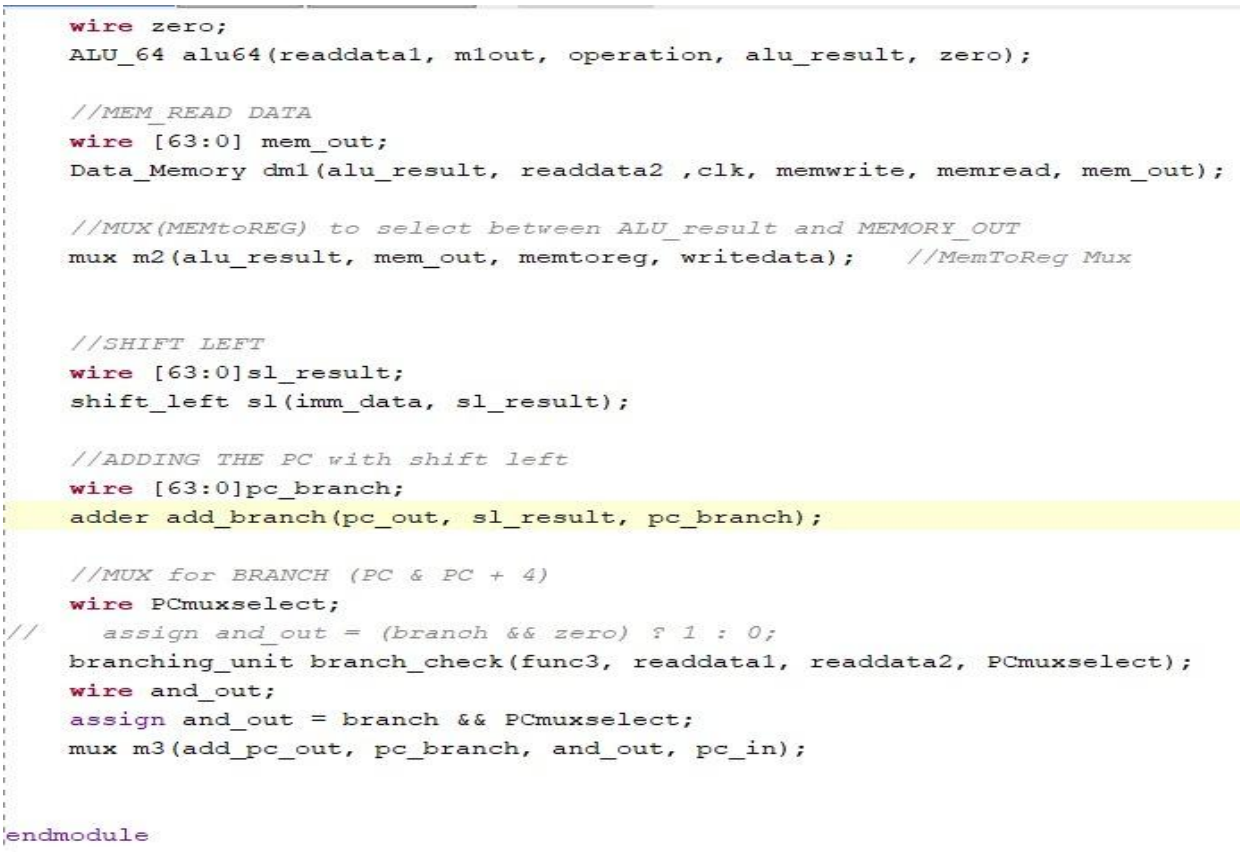


```
reg [7:0] inst_mem [80:0];
initial
begin
      {inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h00000513;//1
      {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h00000093;//2
      {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h00000113;//3
      {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h00000293;//4
      {inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'h005505b3;//5
      {inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'h0005a603;//6
      {inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} = 32'h00858693;//7
      {inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h0006a703;//8
      {inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} = 32'h00e64663;//9
      {inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} = 32'h00c6a023;//10
      {inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'h00e5a023;//11
      {inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h00110113;//12
      {inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'h00828293;//13
      {inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'hfc411ee3;//14
      {inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'h00000113;//15
      {inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'h00000293;//16
      {inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} = 32'h00108093;//17
      {inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'hfc4096e3;//18

end
always @(Inst_Address)
begin
Instruction[7:0] = inst_mem[Inst_Address+0];
      Instruction[15:8] = inst_mem[Inst_Address+1];
      Instruction[23:16] = inst_mem[Inst_Address+2];
```

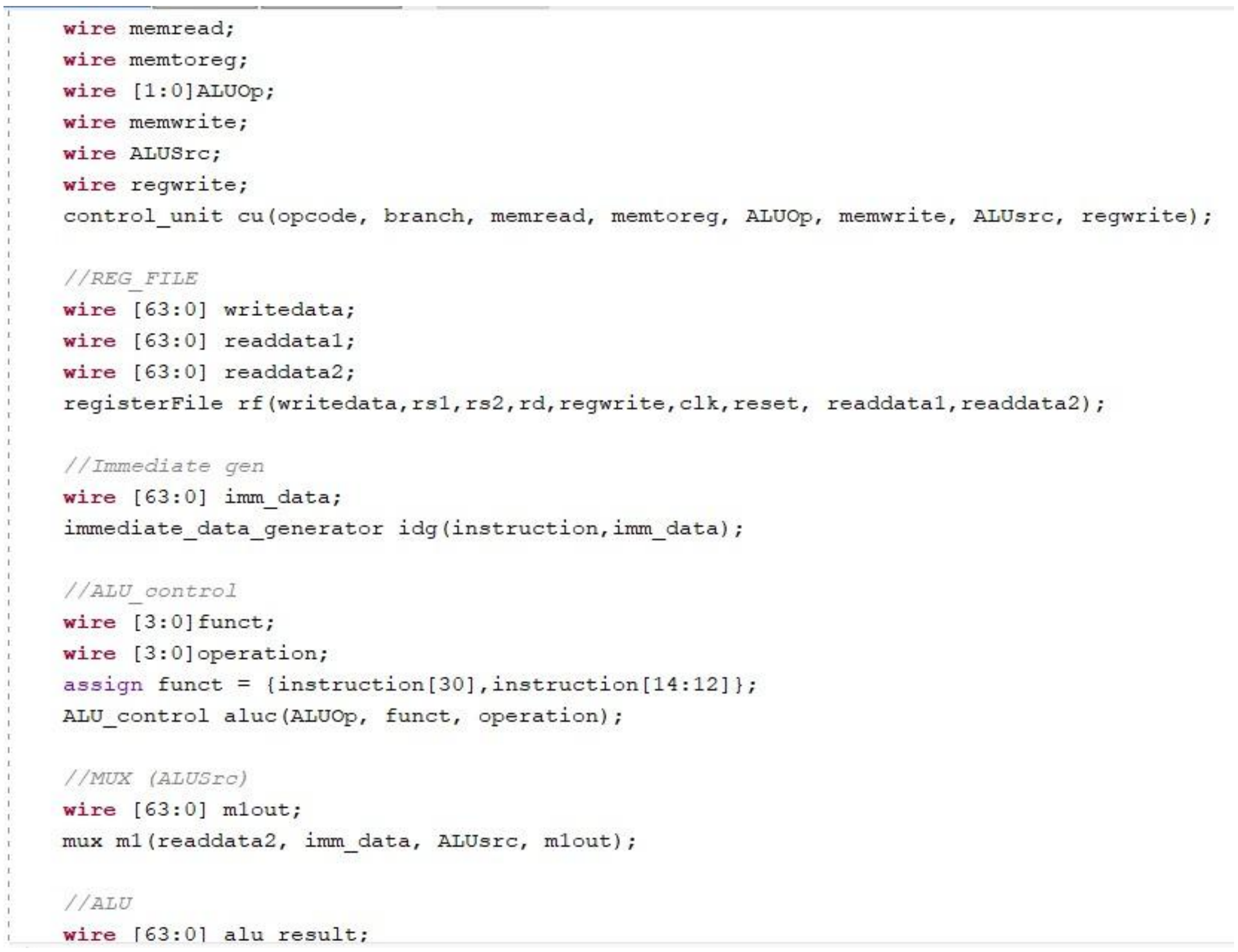
2.6 Main processor code:



```
`timescale 1ns / 1ps

module processor(
    input clk,
    input reset
);
    //Program Counter
    wire [63:0]pc_in;
    wire [63:0] pc_out;
    program_counter pc2(clk, reset, pc_in, pc_out);

    //PC + 4
    wire [63:0]add_pc_out;
    adder add_pc(pc_out, 64'd4, add_pc_out);

    //INST_MEM
    wire [31:0] instruction;
    instruction_memory im2(pc_out, instruction);

    //PARSER
    wire [6:0]opcode;
    wire [2:0] func3;
    wire [6:0] func7;
    wire [4:0] rs1;
    wire [4:0] rs2;
    wire [4:0] rd;
    instruction_parser ip(instruction,opcode,rd,func3,rs1,rs2,func7);

    //CONTROL_UNIT
    wire branch;
```

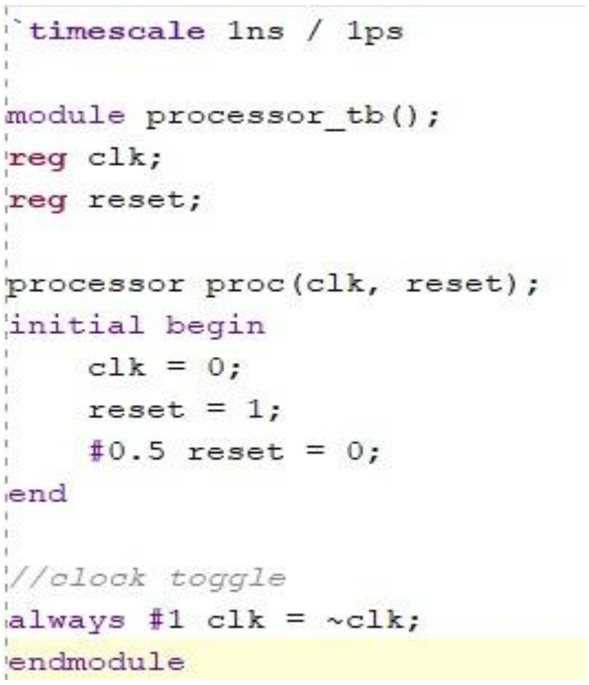


```
    wire zero;
    ALU_64 alu64(readdata1, m1out, operation, alu_result, zero);

    //MEM_READ DATA
    wire [63:0] mem_out;
    Data_Memory dm1(alu_result, readdata2 ,clk, memwrite, memread, mem_out);

    //MUX(MEMtoREG) to select between ALU_result and MEMORY_OUT
    mux m2(alu_result, mem_out, memtoreg, writedata);   //MemToReg Mux


    //SHIFT LEFT
    wire [63:0]sl_result;
    shift_left sl(imm_data, sl_result);

    //ADDING THE PC with shift left
    wire [63:0]pc_branch;
    adder add_branch(pc_out, sl_result, pc_branch);

    //MUX for BRANCH (PC & PC + 4)
    wire PCmuxselect;
//    assign and_out = (branch && zero) ? 1 : 0;
    branching_unit branch_check(func3, readdata1, readdata2, PCmuxselect);
    wire and_out;
    assign and_out = branch && PCmuxselect;
    mux m3(add_pc_out, pc_branch, and_out, pc_in);


endmodule
```



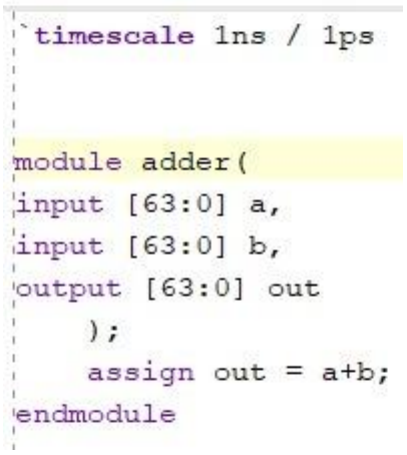
```
    wire memread;
    wire memtoreg;
    wire [1:0]ALUOp;
    wire memwrite;
    wire ALUSrc;
    wire regwrite;
    control_unit cu(opcode, branch, memread, memtoreg, ALUOp, memwrite, ALUsrc, regwrite);

    //REG_FILE
    wire [63:0] writedata;
    wire [63:0] readdata1;
    wire [63:0] readdata2;
    registerFile rf(writedata,rs1,rs2,rd,regwrite,clk,reset, readdata1,readdata2);

    //Immediate gen
    wire [63:0] imm_data;
    immediate_data_generator idg(instruction,imm_data);

    //ALU_control
    wire [3:0]funct;
    wire [3:0]operation;
    assign funct = {instruction[30],instruction[14:12]};
    ALU_control aluc(ALUOp, funct, operation);

    //MUX (ALUSrc)
    wire [63:0] m1out;
    mux m1(readdata2, imm_data, ALUsrc, m1out);

    //ALU
    wire [63:0] alu_result;
```

2.7 Processor TestBench:



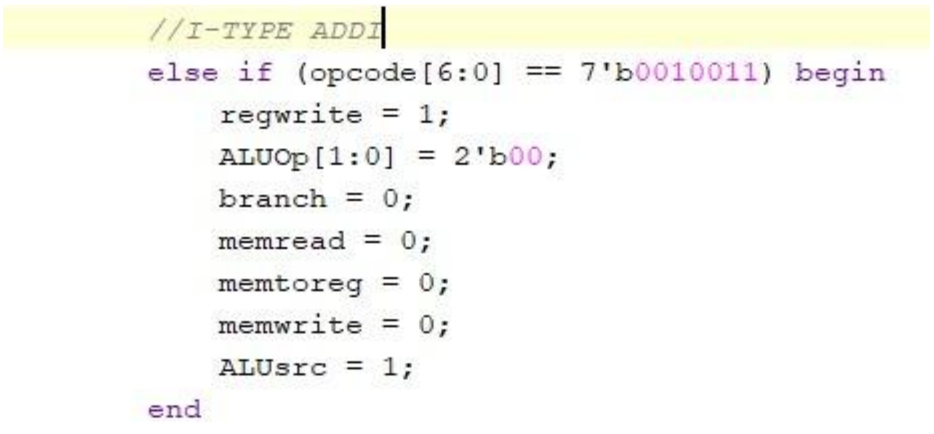
```
`timescale 1ns / 1ps

module processor_tb();
reg clk;
reg reset;

processor proc(clk, reset);
initial begin
    clk = 0;
    reset = 1;
    #0.5 reset = 0;
end

//clock toggle
always #1 clk = ~clk;
endmodule
```

Changes Made:



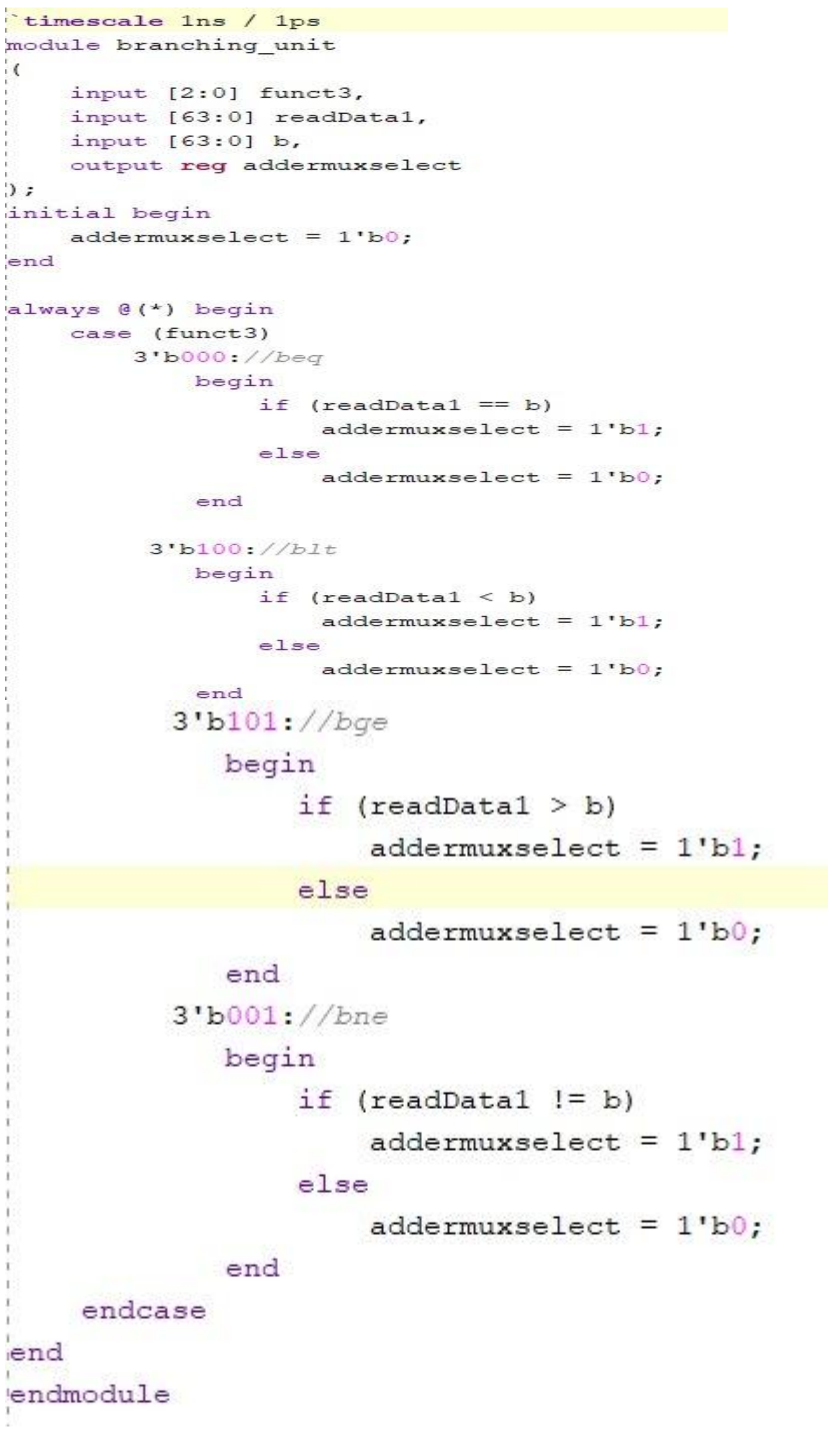
```
`timescale 1ns / 1ps


module adder(
input [63:0] a,
input [63:0] b,
output [63:0] out
    );
    assign out = a+b;
endmodule
```

64 bit-adder.
Used to add pc+4,
And to add PC + Imm for branch (pc_branch)



```
        //I-TYPE ADDI
        else if (opcode[6:0] == 7'b0010011) begin
            regwrite = 1;
            ALUOp[1:0] = 2'b00;
            branch = 0;
            memread = 0;
            memtoreg = 0;
            memwrite = 0;
            ALUsrc = 1;
        end
```

In the control_unit module
We initially used ALUOp = 10, which was incorrect as it is meant for R-Type instructions, so we changed it to ALUOp = 00 for Addi. We successfully implemented Addi instructions; otherwise, we faced issues adding negative values, essentially subtracting, e.g., (addi, x6, x6, -1).



```
`timescale 1ns / 1ps
module branching_unit
(
    input [2:0] funct3,
    input [63:0] readData1,
    input [63:0] b,
    output reg addermuxselect
);
initial begin
    addermuxselect = 1'b0;
end

always @(*) begin
    case (funct3)
        3'b000://beq
            begin
                if (readData1 == b)
                    addermuxselect = 1'b1;
                else
                    addermuxselect = 1'b0;
            end

         3'b100://blt
            begin
                if (readData1 < b)
                    addermuxselect = 1'b1;
                else
                    addermuxselect = 1'b0;
            end
         3'b101://bge
            begin
                if (readData1 > b)
                    addermuxselect = 1'b1;
                else
                    addermuxselect = 1'b0;
            end
         3'b001://bne
            begin
                if (readData1 != b)
                    addermuxselect = 1'b1;
                else
                    addermuxselect = 1'b0;
            end
    endcase
end
endmodule
```

We removed zero from our ALU_64 and created a separate branching unit that takes func3, readdata1, and readdata2 to compare the two provided registers. We implemented 4 different types using various Func3 codes based on the RISC-V Card:

000 = beq
100 = blt
101 = bge
001 = bne

3 Task 2

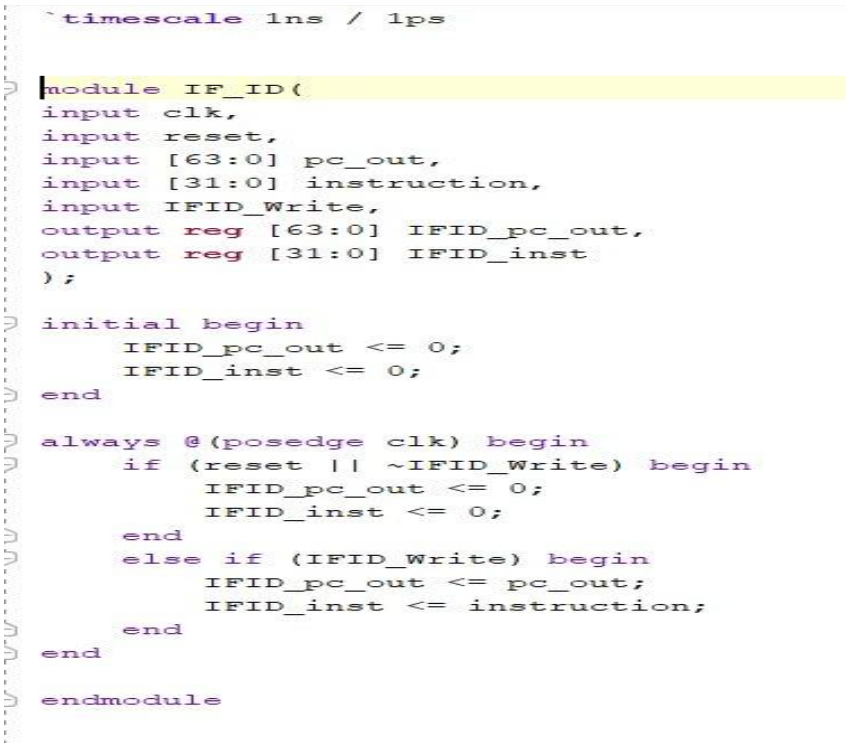
3.1 Pipelined RISC V Processor

After implementing the algorithm on the single-cycle processor, we proceeded to modify the code to pipeline the modules. To achieve this, we referred to our course book and added the following modules to serve as intermediate registers for the pipeline:

- **IF/ID**
- **ID/EX**
- **EX/MEM**
- **MEM/WB**

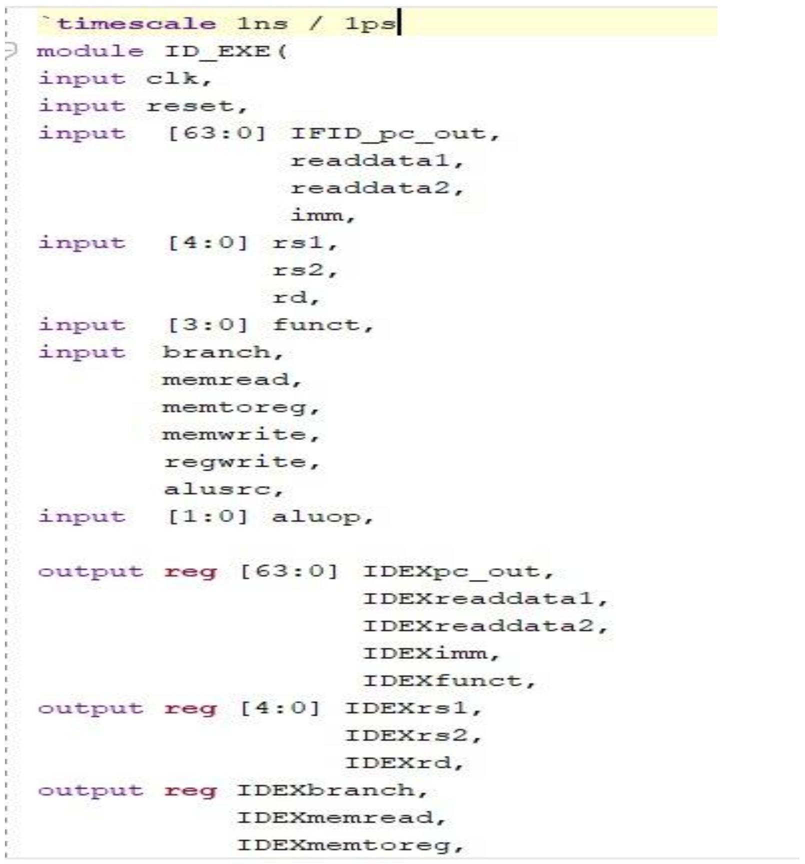
These registers store the data of the previous instruction and pass it along the pipeline. We tested each instruction individually to ensure the pipeline operated correctly. Additionally, we implemented the forwarding unit by modifying the previous code to allow data forwarding within the pipeline. The forwarding unit is integrated with the hazard detection unit, which determines where the data should be forwarded.

3.2 IF/ID Register:

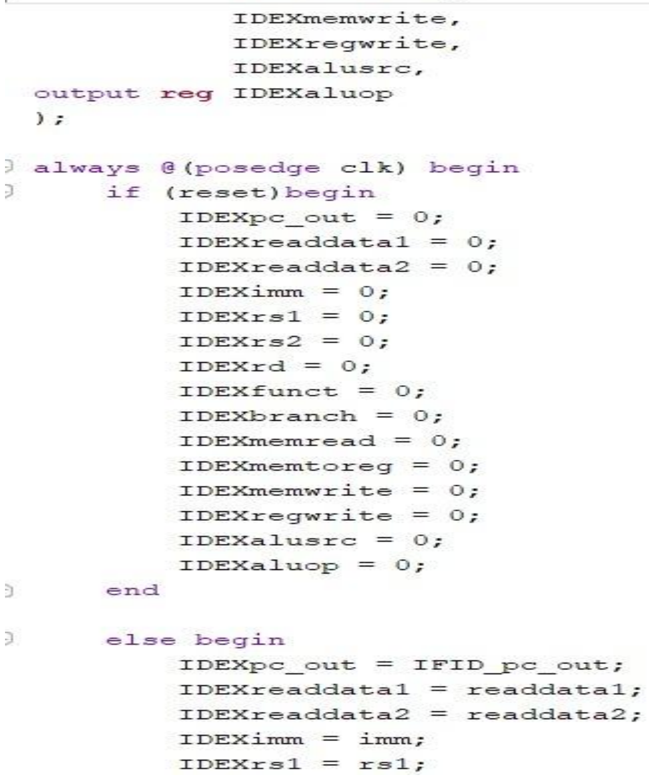


```
`timescale 1ns / 1ps

module IF_ID(
input clk,
input reset,
input [63:0] pc_out,
input [31:0] instruction,
input IFID_Write,
output reg [63:0] IFID_pc_out,
output reg [31:0] IFID_inst
);

initial begin
     IFID_pc_out <= 0;
     IFID_inst <= 0;
end

always @(posedge clk) begin
     if (reset || ~IFID_Write) begin
          IFID_pc_out <= 0;
          IFID_inst <= 0;
     end
     else if (IFID_Write) begin
          IFID_pc_out <= pc_out;
          IFID_inst <= instruction;
     end
end

endmodule
```

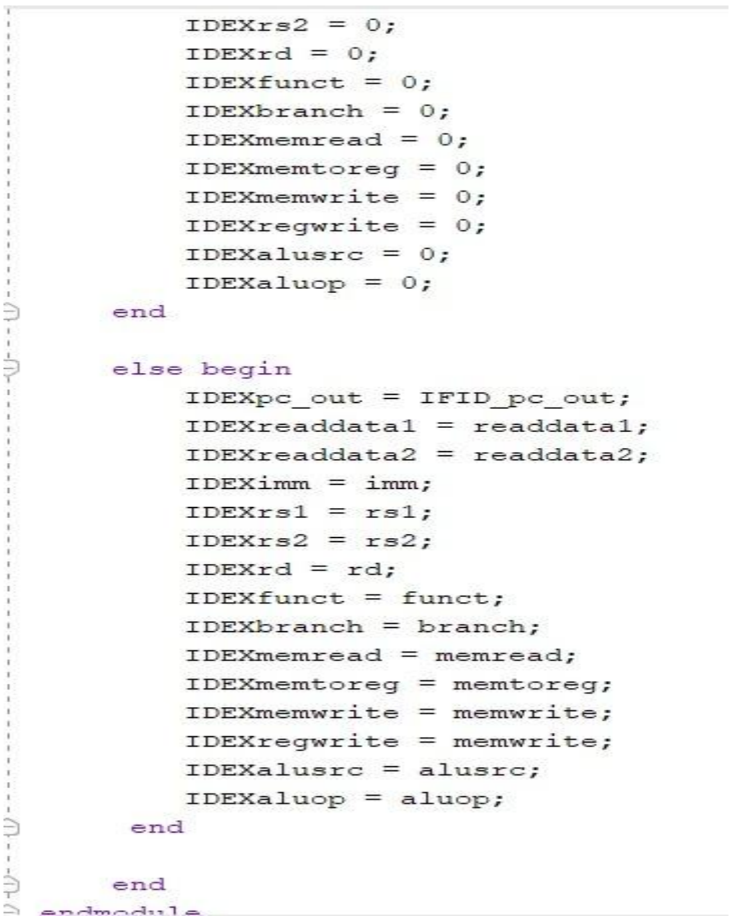
3.3 ID/EXE Register:



```
`timescale 1ns / 1ps
module ID_EXE(
input clk,
input reset,
input   [63:0] IFID_pc_out,
               readdata1,
               readdata2,
               imm,
input   [4:0] rs1,
              rs2,
              rd,
input   [3:0] funct,
input   branch,
        memread,
        memtoreg,
        memwrite,
        regwrite,
        alusrc,
input   [1:0] aluop,

output reg [63:0] IDEXpc_out,
                  IDEXreaddata1,
                  IDEXreaddata2,
                  IDEXimm,
                  IDEXfunct,
output reg [4:0] IDEXrs1,
                 IDEXrs2,
                 IDEXrd,
output reg IDEXbranch,
           IDEXmemread,
           IDEXmemtoreg,
```

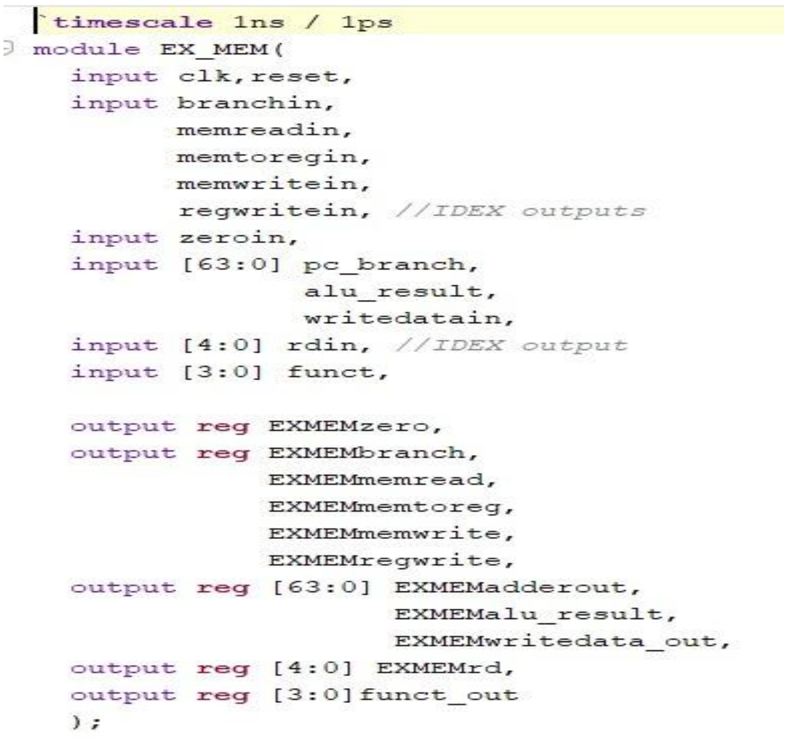


```
           IDEXmemwrite,
           IDEXregwrite,
           IDEXalusrc,
output reg IDEXaluop
);

always @(posedge clk) begin
    if (reset)begin
        IDEXpc_out = 0;
        IDEXreaddata1 = 0;
        IDEXreaddata2 = 0;
        IDEXimm = 0;
        IDEXrs1 = 0;
        IDEXrs2 = 0;
        IDEXrd = 0;
        IDEXfunct = 0;
        IDEXbranch = 0;
        IDEXmemread = 0;
        IDEXmemtoreg = 0;
        IDEXmemwrite = 0;
        IDEXregwrite = 0;
        IDEXalusrc = 0;
        IDEXaluop = 0;
    end

    else begin
        IDEXpc_out = IFID_pc_out;
        IDEXreaddata1 = readdata1;
        IDEXreaddata2 = readdata2;
        IDEXimm = imm;
        IDEXrs1 = rs1;
```

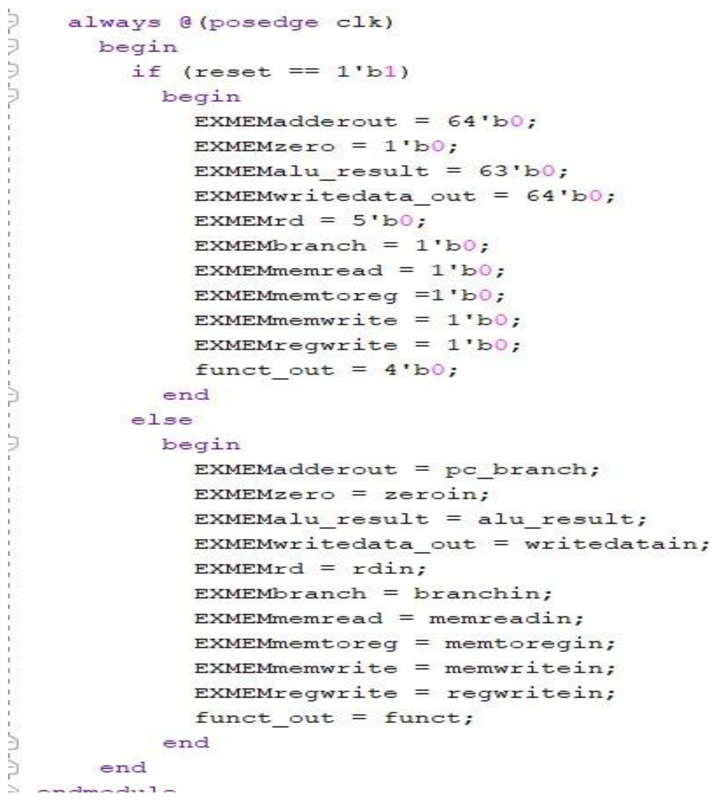


```
            IDEXrs2 = 0;
            IDEXrd = 0;
            IDEXfunct = 0;
            IDEXbranch = 0;
            IDEXmemread = 0;
            IDEXmemtoreg = 0;
            IDEXmemwrite = 0;
            IDEXregwrite = 0;
            IDEXalusrc = 0;
            IDEXaluop = 0;
        end

        else begin
            IDEXpc_out = IFID_pc_out;
            IDEXreaddata1 = readdata1;
            IDEXreaddata2 = readdata2;
            IDEXimm = imm;
            IDEXrs1 = rs1;
            IDEXrs2 = rs2;
            IDEXrd = rd;
            IDEXfunct = funct;
            IDEXbranch = branch;
            IDEXmemread = memread;
            IDEXmemtoreg = memtoreg;
            IDEXmemwrite = memwrite;
            IDEXregwrite = memwrite;
            IDEXalusrc = alusrc;
            IDEXaluop = aluop;
         end

        end
endmodule
```

3.4 EX/MEM Register:

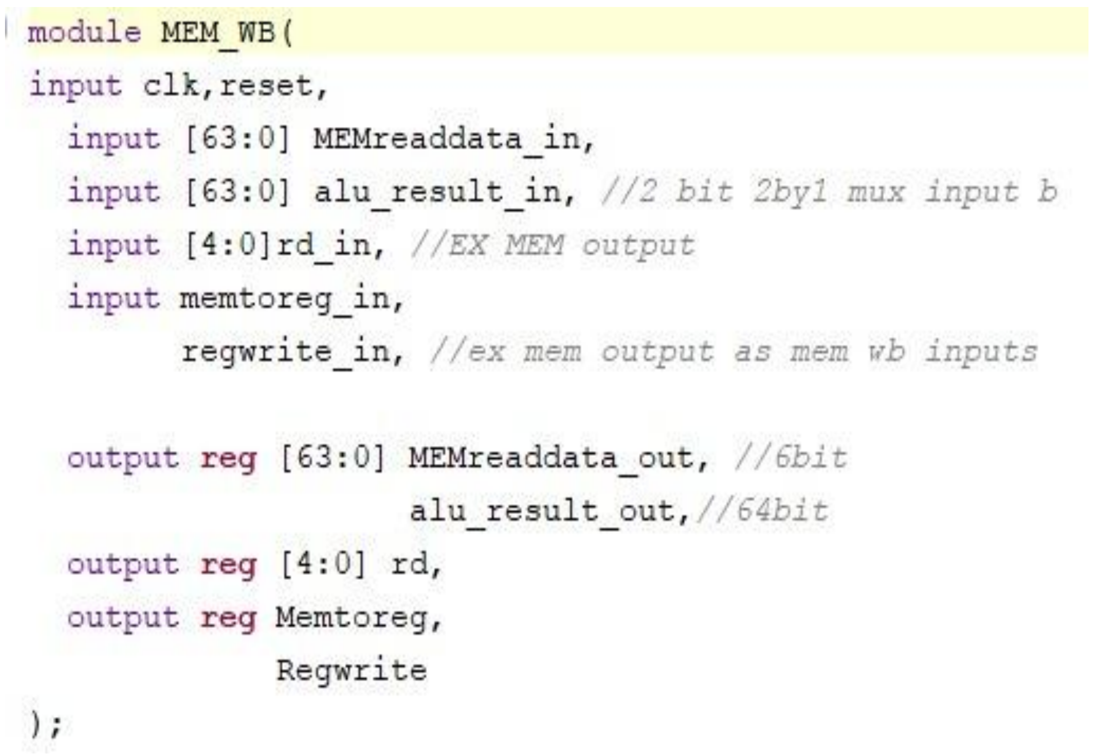


```
`timescale 1ns / 1ps
module EX_MEM(
    input clk,reset,
    input branchin,
          memreadin,
          memtoregin,
          memwritein,
          regwritein, //IDEX outputs
    input zeroin,
    input [63:0] pc_branch,
                 alu_result,
                 writedatain,
    input [4:0] rdin, //IDEX output
    input [3:0] funct,

    output reg EXMEMzero,
    output reg EXMEMbranch,
               EXMEMmemread,
               EXMEMmemtoreg,
               EXMEMmemwrite,
               EXMEMregwrite,
    output reg [63:0] EXMEMadderout,
                      EXMEMalu_result,
                      EXMEMwritedata_out,
    output reg [4:0] EXMEMrd,
    output reg [3:0]funct_out
    );
```

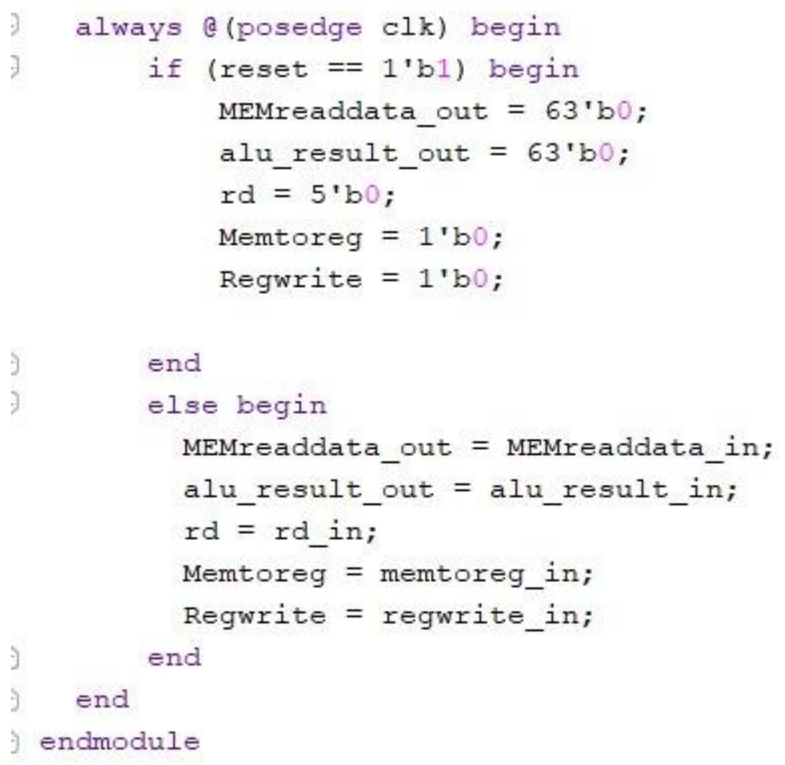


```
  always @(posedge clk)
    begin
      if (reset == 1'b1)
        begin
          EXMEMadderout = 64'b0;
          EXMEMzero = 1'b0;
          EXMEMalu_result = 63'b0;
          EXMEMwritedata_out = 64'b0;
          EXMEMrd = 5'b0;
          EXMEMbranch = 1'b0;
          EXMEMmemread = 1'b0;
          EXMEMmemtoreg =1'b0;
          EXMEMmemwrite = 1'b0;
          EXMEMregwrite = 1'b0;
          funct_out = 4'b0;
        end
      else
        begin
          EXMEMadderout = pc_branch;
          EXMEMzero = zeroin;
          EXMEMalu_result = alu_result;
          EXMEMwritedata_out = writedatain;
          EXMEMrd = rdin;
          EXMEMbranch = branchin;
          EXMEMmemread = memreadin;
          EXMEMmemtoreg = memtoregin;
          EXMEMmemwrite = memwritein;
          EXMEMregwrite = regwritein;
          funct_out = funct;
        end
    end
```

3.5 MEM/WB Register:

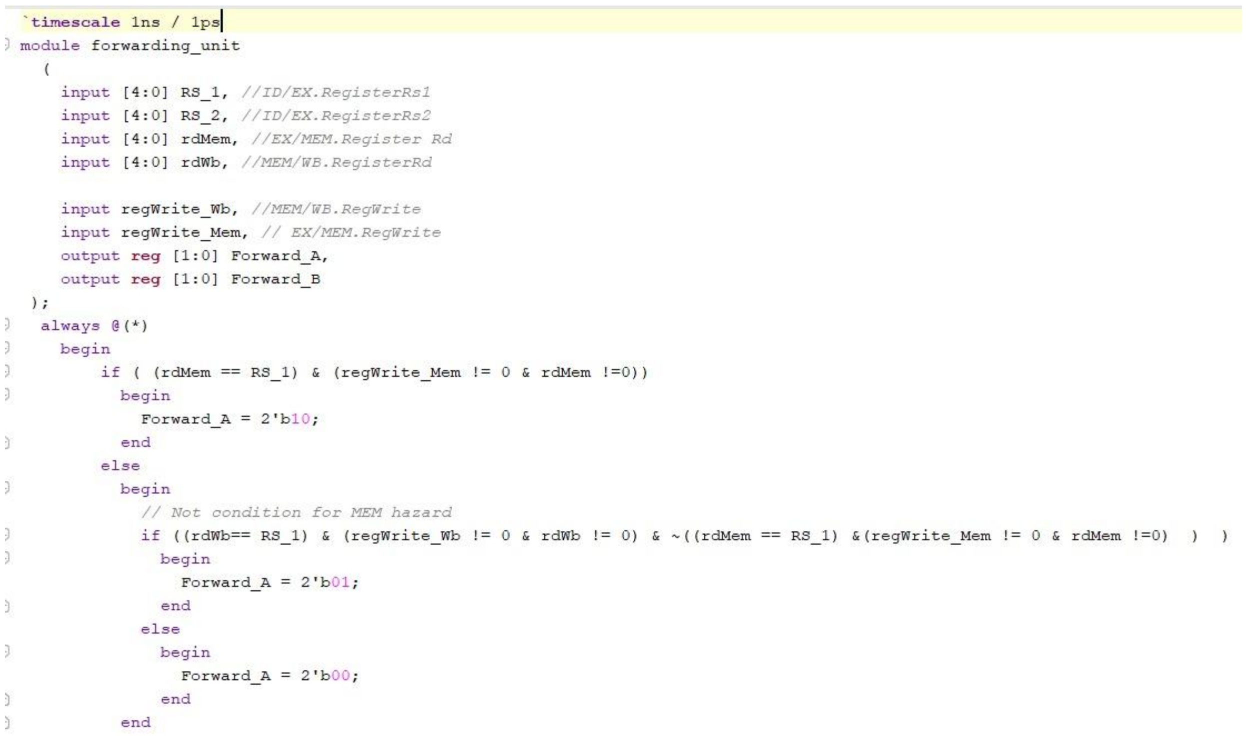


```
module MEM_WB(
input clk,reset,
  input [63:0] MEMreaddata_in,
  input [63:0] alu_result_in, //2 bit 2by1 mux input b
  input [4:0]rd_in, //EX MEM output
  input memtoreg_in,
        regwrite_in, //ex mem output as mem wb inputs

  output reg [63:0] MEMreaddata_out, //6bit
                    alu_result_out,//64bit
  output reg [4:0] rd,
  output reg Memtoreg,
             Regwrite
);
```

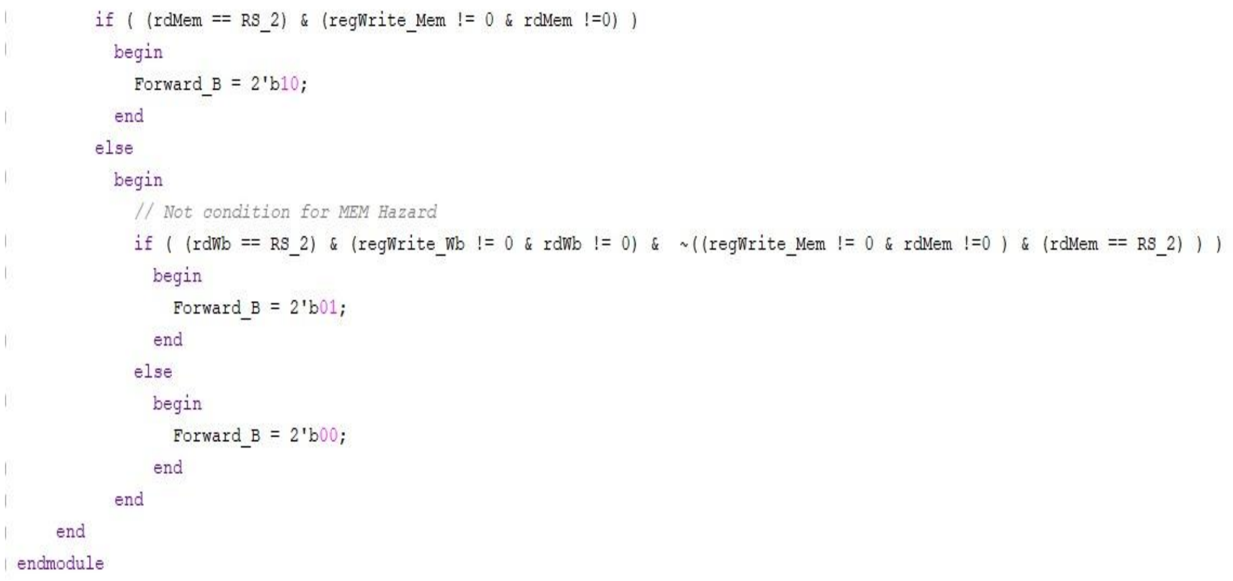


```
    always @(posedge clk) begin
        if (reset == 1'b1) begin
            MEMreaddata_out = 63'b0;
            alu_result_out = 63'b0;
            rd = 5'b0;
            Memtoreg = 1'b0;
            Regwrite = 1'b0;

        end
        else begin
          MEMreaddata_out = MEMreaddata_in;
          alu_result_out = alu_result_in;
          rd = rd_in;
          Memtoreg = memtoreg_in;
          Regwrite = regwrite_in;
        end
    end
endmodule
```

3.6 Forwarding Unit:

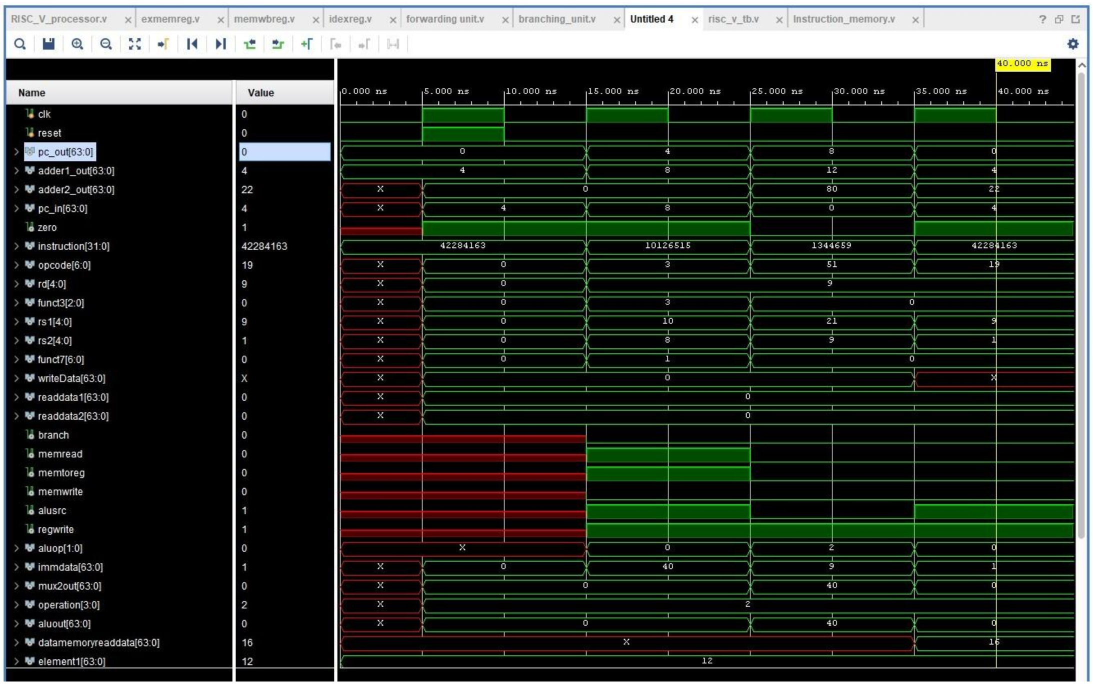


```
 `timescale 1ns / 1ps
module forwarding_unit
   (
     input [4:0] RS_1, //ID/EX.RegisterRs1
     input [4:0] RS_2, //ID/EX.RegisterRs2
     input [4:0] rdMem, //EX/MEM.Register Rd
     input [4:0] rdWb, //MEM/WB.RegisterRd

     input regWrite_Wb, //MEM/WB.RegWrite
     input regWrite_Mem, // EX/MEM.RegWrite
     output reg [1:0] Forward_A,
     output reg [1:0] Forward_B
  );
   always @(*)
     begin
         if ( (rdMem == RS_1) & (regWrite_Mem != 0 & rdMem !=0))
           begin
             Forward_A = 2'b10;
           end
         else
           begin
             // Not condition for MEM hazard
             if ((rdWb== RS_1) & (regWrite_Wb != 0 & rdWb != 0) & ~((rdMem == RS_1) &(regWrite_Mem != 0 & rdMem !=0)  )  )
               begin
                 Forward_A = 2'b01;
               end
             else
               begin
                 Forward_A = 2'b00;
               end
           end
```



```
         if ( (rdMem == RS_2) & (regWrite_Mem != 0 & rdMem !=0) )
           begin
             Forward_B = 2'b10;
           end
         else
           begin
             // Not condition for MEM Hazard
             if ( (rdWb == RS_2) & (regWrite_Wb != 0 & rdWb != 0) &  ~((regWrite_Mem != 0 & rdMem !=0 ) & (rdMem == RS_2) ) )
               begin
                 Forward_B = 2'b01;
               end
             else
               begin
                 Forward_B = 2'b00;
               end
           end
     end
endmodule
```

Simulation Output



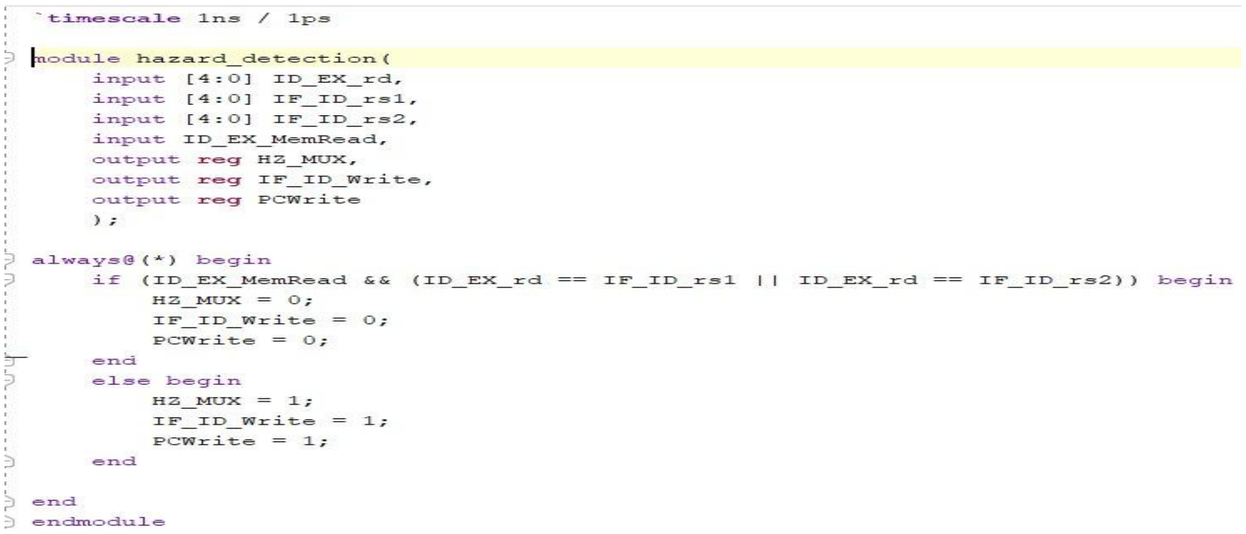
Name	Value
clk	0
reset	0
pc_out[63:0]	0
adder1_out[63:0]	4
adder2_out[63:0]	22
pc_in[63:0]	4
zero	1
instruction[31:0]	42284163
opcode[6:0]	19
rd[4:0]	9
funct3[2:0]	0
rs1[4:0]	9
rs2[4:0]	1
funct7[6:0]	0
writeData[63:0]	X
readdata1[63:0]	0
readdata2[63:0]	0
branch	0
memread	0
memtoreg	0
memwrite	0
alusrc	1
regwrite	1
aluop[1:0]	0
immdata[63:0]	1
mux2out[63:0]	0
operation[3:0]	2
aluout[63:0]	0
datamemoryreaddata[63:0]	16
element1[63:0]	12

Figure 4: Snippet of simulation output

4 Task 3

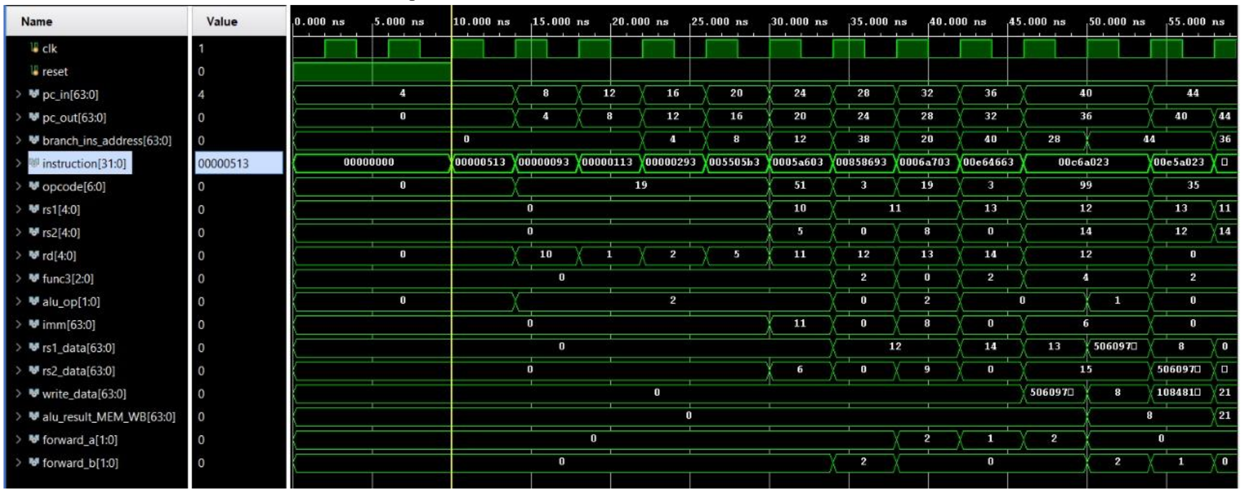
4.1 Implementing Hazard Detection Circuitry

Hazards such as data, structural, and control are addressed in the code by implementing hazard detection circuitry and introducing pipeline stalls. These hazards primarily occur due to code dependencies or when data needs to be forwarded at a specific stage. To handle this, we implemented a hazard detection unit that determines when to stall the pipeline or forward the data, signaling the forwarding unit to either stall or flush the pipeline accordingly



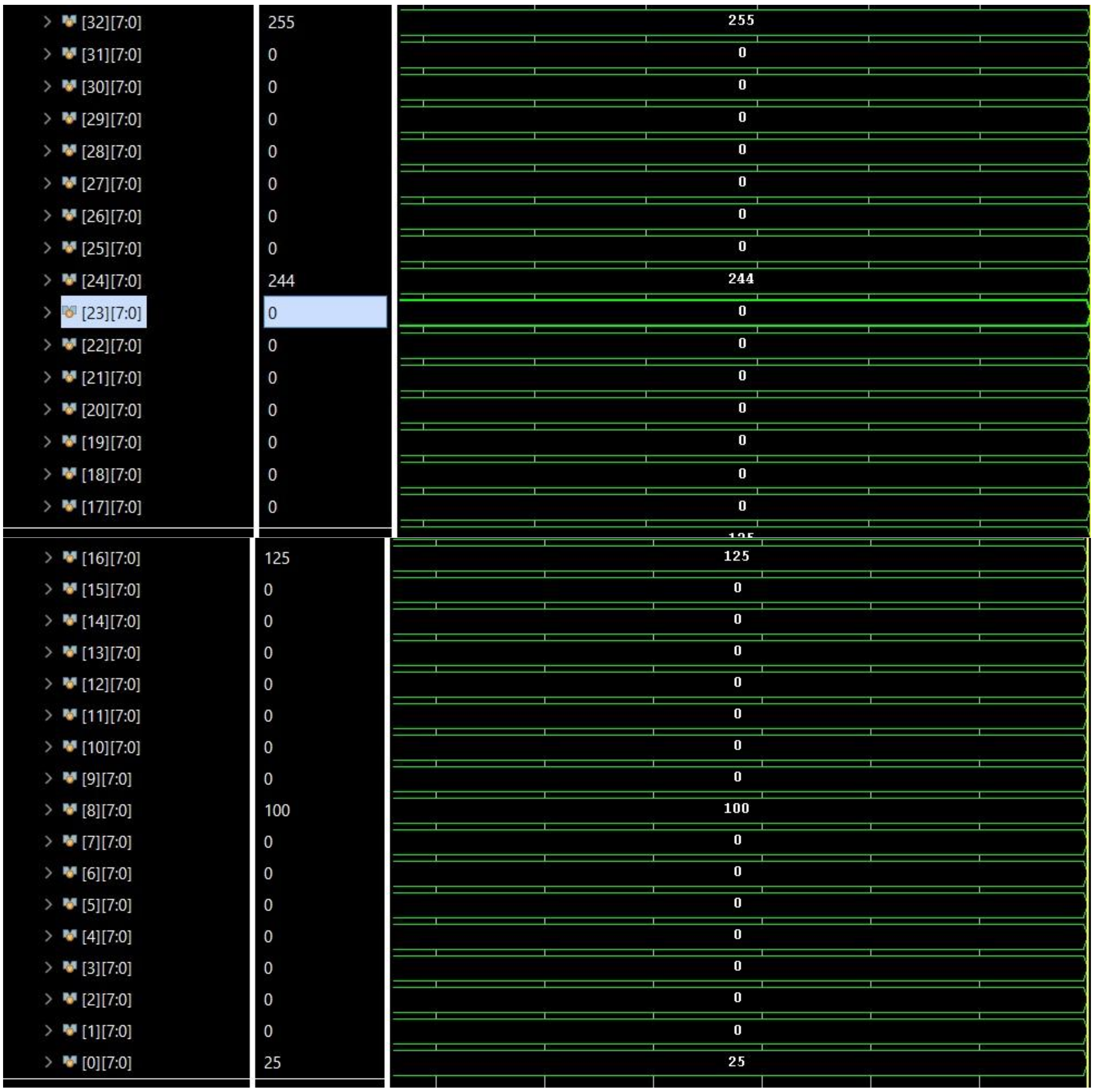
```
`timescale 1ns / 1ps

module hazard_detection(
        input  [4:0]  ID_EX_rd,
        input  [4:0]  IF_ID_rs1,
        input  [4:0]  IF_ID_rs2,
        input ID_EX_MemRead,
        output reg HZ_MUX,
        output reg IF_ID_Write,
        output reg PCWrite
        );

always@(*) begin
        if (ID_EX_MemRead && (ID_EX_rd == IF_ID_rs1 || ID_EX_rd == IF_ID_rs2)) begin
                HZ_MUX = 0;
                IF_ID_Write = 0;
                PCWrite = 0;
        end
        else begin
                HZ_MUX = 1;
                IF_ID_Write = 1;
                PCWrite = 1;
        end

end
endmodule
```

4.2 Simulation Output



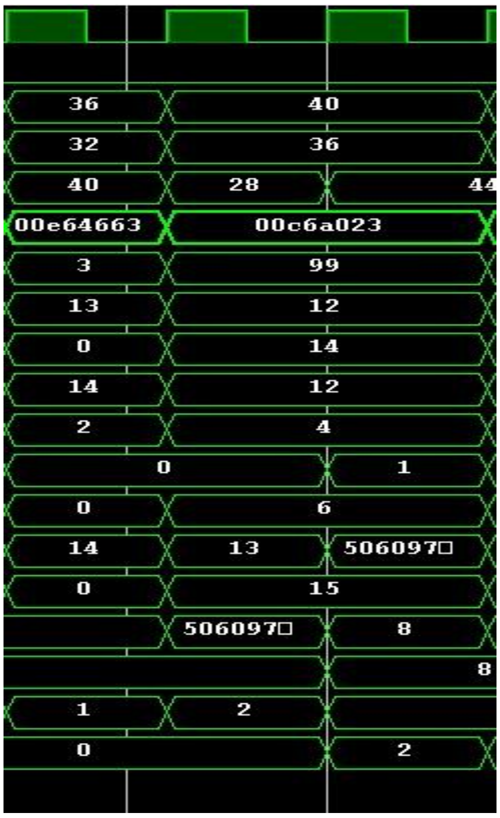
Name	Value
clk	1
reset	0
pc_in[63:0]	4
pc_out[63:0]	0
branch_ins_address[63:0]	0
instruction[31:0]	00000513
opcode[6:0]	0
rs1[4:0]	0
rs2[4:0]	0
rd[4:0]	0
func3[2:0]	0
alu_op[1:0]	0
imm[63:0]	0
rs1_data[63:0]	0
rs2_data[63:0]	0
write_data[63:0]	0
alu_result_MEM_WB[63:0]	0
forward_a[1:0]	0
forward_b[1:0]	0

Figure 5: Snippet of simulation output



[32][7:0]	255
[31][7:0]	0
[30][7:0]	0
[29][7:0]	0
[28][7:0]	0
[27][7:0]	0
[26][7:0]	0
[25][7:0]	0
[24][7:0]	244
[23][7:0]	0
[22][7:0]	0
[21][7:0]	0
[20][7:0]	0
[19][7:0]	0
[18][7:0]	0
[17][7:0]	0
[16][7:0]	125
[15][7:0]	0
[14][7:0]	0
[13][7:0]	0
[12][7:0]	0
[11][7:0]	0
[10][7:0]	0
[9][7:0]	0
[8][7:0]	100
[7][7:0]	0
[6][7:0]	0
[5][7:0]	0
[4][7:0]	0
[3][7:0]	0
[2][7:0]	0
[1][7:0]	0
[0][7:0]	25

Figure 6: Sorted Array



This shows the stall in the cycle, due to loading in an instruction and reading in the other.

5 Performance Comparison

Pipelined processors are typically faster than non-pipelined ones. However, in our case, the pipelined processor is underperforming, with a latency of approximately 6000ns—double the 3000ns latency of the non-pipelined processor. While we successfully implemented pipelining, it introduced several efficiency challenges. To enhance the performance of our pipelined processor, a thorough analysis and optimization of its design are necessary.

6 Conclusion and Challenges

Building the processor was quite challenging, as we encountered several issues, including frequent crashes in Vivado and problems with zipped files that failed to open on other PCs or laptops. The primary goal of pipelining was to enhance the processor's efficiency, enabling it to handle multiple tasks seamlessly. However, we faced significant difficulties during simulations, particularly in implementing stalls at critical points. Making the simulation run successfully to completion proved to be a tough task. Incorporating branch conditions added to the complexity, and dependency issues led to frequent stall problems. Despite these challenges, we successfully developed a pipelined processor that effectively managed most hazards.

7 References

Book. *Course Book*. Computer Organization and Design: The Hardware/Software Interface RISC-V Edition by David A. Patterson, John L. Hennessy