Final Project Report

Pipelined Processor

Ali Asghar Yousuf - ay06993
Abdul Majid - at06616
Muneeb Shafique - ms06373

May 18, 2022

EE/CS 371L/330L Computer Architecture
Supervisors: Muhammad Umer Tariq - Aiman Najeeb

Contents

1 Task 1

1.1 Assembly code

```
addi x10, x0, 6     #initializing size of array
addi x6, x0, 0      #initializing i
addi x7, x0, 0      #initializing offset
addi x23, x0, 12    #initializing arbitary number


#Inputting arbitary values in the array
ARRAY: sw x23, 0x100(x7)            #storing in memory
           addi x6, x6, 1           #i+=1
           addi x7, x7, 4           #offset+=4
           addi x23, x23, -2        #generating arbitary values
           beq x10, x6, Array_Done  #i<size of array
           beq x0, x0, ARRAY

Array_Done: #selection sort will start now

addi x7, x0, 0 #use for ILOOP
addi x8, x7, 1 #use for JLOOp
addi x6, x0, 0

LOOP_I: beq x10, x7, Sorting_Done
       add x11, x6, x0      #[min_idx]=i
       addi x8, x7, 1       #j+=1
       addi x17, x6, 4      #off_j+=4

LOOP_J: beq x8, x10, SWAPPING
        lw x12, 0x100(x17)      #array[j]
        lw x13, 0x100(x11)      #array[min_idx]
        blt x12, x13, Enter_if  #if A[min_idx]>A[j]
        back: addi x8, x8, 1    #increment j
        addi x17, x17, 4        #off_j+=14
        beq x0, x0, LOOP_J

       return: addi x7, x7, 1   #i+=1
       addi x6, x6, 4   #off_i+=4
       beq x0, x0, LOOP_I

Enter_if: addi x11, x17, 0      #[min_idx]=j
       beq x0, x0, back

SWAPPING:  lw x15, 0x100(x11)   #A[min_idx]
           lw x14, 0x100(x6)    #A[i]
           sw x14, 0x100(x11)   #Changing locations
           sw x15, 0x100(x6)
           beq x0,x0, return

Sorting_Done:
```

Figure 1: Risc-V assembly code

1.2 Equivalent Python code

The following is the Sorting Algorithm in Python:

```
# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]
```

Figure 2: Sorting Algorithm in Python

1.3 Testing Assembly on Venus Simulator

Registers Memory

Address	+0	+1	+2	+3
0x00000120	0	0	0	0
0x0000011c	0	0	0	0
0x00000118	0	0	0	0
0x00000114	12	0	0	0
0x00000110	10	0	0	0
0x0000010c	8	0	0	0
0x00000108	6	0	0	0
0x00000104	4	0	0	0
0x00000100	2	0	0	0

Figure 3: Snapshot of Memory

1.4 Explanation: Testing Sorting Algorithm on Single Cycle Processor

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two sub-arrays in a given array, the sub-array which is already sorted, remaining sub-array which is unsorted. In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted sub-array is picked and moved to the sorted sub-array.

To implement this algorithm in assembly the following approach has been used. Firstly, we initialize an array in the memory with arbitrary values which has been done using the initial loop. Once the array is populated with values two loops have been made use of to sort the already present values in the array. The minimum (min-index) to is initialized to location 0 after the array is traversed to locate the min element. While traversing if and element is found to be smaller that min-index both values have been swapped. At this stage min-index is incremented to point to the next element. This process has been repeated until the entire array is sorted.

To test the working of this assembly code on our processor alterations were made to the module Instruction Memory. Hex codes for each instruction were found and loaded into the subsequent locations of the array(Figure 5). To verify our results additional registers in our case 6 (number of values stored in the array) were defined in the module Data Memory and their values were compared which demonstrated sorting by showcasing each switch of element in the array. (Figure 6) The following is demonstrated in the following snapshot of the EP-wave generated after running the code.

EDA playground Link for Task 1:
https://www.edaplayground.com/x/k9ej

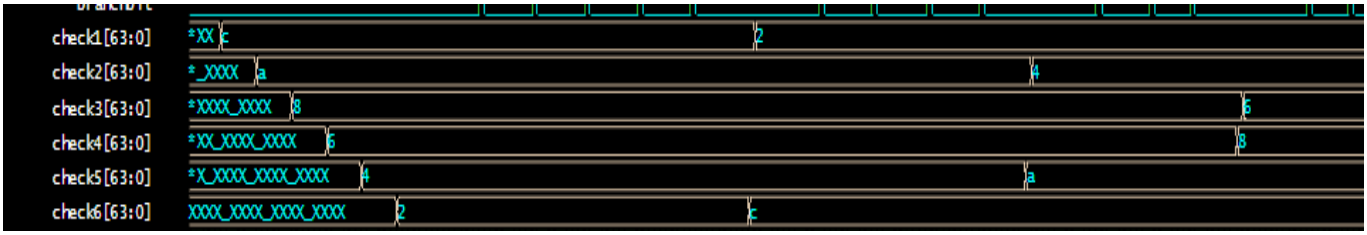

Figure 4: Snapshot of EP-Wave: Result Testing

1.5 Changes in Architecture

```
initial
  begin
    {Array[138], Array[137], Array[136], Array[135]} = 32'hfc000ee3; //beq x0 x0 -36
    {Array[134], Array[133], Array[132], Array[131]} = 32'h10F33023; //sw x15 256(x6)
    {Array[130], Array[129], Array[128], Array[127]} = 32'h10E5B023; //sw x14 256(x11)
    {Array[126], Array[125], Array[124], Array[123]} = 32'h10033703; //lw x14 256(x6)
    {Array[122], Array[121], Array[120], Array[119]} = 32'h1005B783; //lw x15 256(x11)

    {Array[118], Array[117], Array[116], Array[115]} = 32'hfe0002e3; //beq x0 x0 -28
    {Array[114], Array[113], Array[112], Array[111]} = 32'h00088593;//addi x11 x17 0
    {Array[110], Array[109], Array[108], Array[107]} = 32'hfc0006e3;//beq x0 x0 -52
    {Array[106], Array[105], Array[104], Array[103]} = 32'h00830313; //addi x6 x6 8
    {Array[102], Array[101], Array[100], Array[99]} = 32'h00138393; //addi x7 x7 1
    {Array[98], Array[97], Array[96], Array[95]} = 32'hfe0004e3;//beq x0 x0 -24
    {Array[94], Array[93], Array[92], Array[91]} = 32'h00888893; //addi x17 x17 8
    {Array[90], Array[89], Array[88], Array[87]} = 32'h00140413;  //addi x8 x8 1
    {Array[86], Array[85], Array[84], Array[83]} = 32'h00d64e63; //blt x12 x13 28
    {Array[82], Array[81], Array[80], Array[79]} = 32'h1005B683; //lw x13 256(x11)

    {Array[78], Array[77], Array[76], Array[75]} = 32'h10088603; //lw x12 256(x17)
    {Array[74], Array[73], Array[72], Array[71]} = 32'h02a40863; //beq x8 x10 48
    {Array[70], Array[69], Array[68], Array[67]} = 32'h00830893; //addi x17 x6 8
    {Array[66], Array[65], Array[64], Array[63]} = 32'h00138413; //addi x8 x7 1
    {Array[62], Array[61], Array[60], Array[59]} = 32'h000305b3; //add x11 x6 x0
    {Array[58], Array[57], Array[56], Array[55]} = 32'h04750a63; //beq x10 x7 84
    {Array[54], Array[53], Array[52], Array[51]} = 32'h00000313; //addi x6 x0 0
    {Array[50], Array[49], Array[48], Array[47]} = 32'h00138413; //addi x8 x7 1
    {Array[46], Array[45], Array[44], Array[43]} = 32'h00000393; //addi x7 x0 0
    {Array[42], Array[41], Array[40], Array[39]} = 32'hfe0006e3; //beq x0 x0 -20
    {Array[38], Array[37], Array[36], Array[35]} = 32'h00650463; //beq x10 x6 8
    {Array[34], Array[33], Array[32], Array[31]} = 32'hffeb8b93; //addi x23 x23 -2

    {Array[30], Array[29], Array[28], Array[27]} = 32'h00838393; //addi x7 x7 8

    {Array[26], Array[26], Array[25], Array[24]} = 32'h00130313; //addi x6 x6 1
    {Array[23], Array[22], Array[21], Array[20]} = 32'h1173B023; //sw x23 256(x7)
    {Array[19], Array[18], Array[17], Array[16]} = 32'h00c00b93;  // addi x23 x0 12
    {Array[11], Array[10], Array[9], Array[8]} = 32'h00000393; //addi x7 x0 0
    {Array[7], Array[6], Array[5], Array[4]} = 32'h00000313; //addi x6 x0 0
    {Array[3], Array[2], Array[1], Array[0]} = 32'h00600513; //addi x10 x0 6

  end

endmodule
```

Figure 5: Changes in Instruction Memory Module

```
module DataMemory(input clk, Memread, Memwrite,
                  input [63:0] Writedata, address,
                  output reg [63:0] readdata,
                  output reg [63:0] check1,
                  output reg [63:0] check2,
                  output reg [63:0] check3,
                  output reg [63:0] check4,
                  output reg [63:0] check5,
                  output reg [63:0] check6);



  reg [7:0] Registers [1233:0];

  assign check1 = Registers[256];
  assign check2 = Registers[264];
  assign check3 = Registers[272];

  assign check4 = Registers[280];
  assign check5 = Registers[288];
  assign check6 = Registers[296];
```

Figure 6: Changes in Data Memory Module

Further Changes in Control Unit:

As shown in Figure 1 the assembly code includes a blt instruction which is used to find the minimum value from the entire array. The following modifications were made the to the Control Unit to cater for this additional type of assembly. Along with the 7 outputs leaving the Control Unit an eighth one was added by the name of branch-blt. This signal is activated, and inputted into an AND gate along with the output from the ALU whose most significant is 1 is the first value is lesser than the second. After passing through the AND gate it further enters the OR gate with the regular signal before it goes into the MUX.

```
module ControlUnit(input[6:0] opcode,
                   output reg branch_eq, MemRead, MemtoReg, MemWrite, ALUsrc, RegWrite,
                   output reg [1:0] ALU_Op,input[2:0] func3,
                   output reg branch_blt);

      always @(*)
          begin
            case(opcode)
                7'b0110011 :
                  begin
                    ALUsrc=1'b0;MemtoReg=1'b0;RegWrite=1'b1;MemRead=1'b0;MemWrite=1'b0;
                    branch_eq=1'b0;branch_blt=1'b0;
                    ALU_Op[1:0]=2'b10;
                  end // If sel=0, output can be a
                7'b0000011 :
                  begin
                    ALUsrc=1'b1;MemtoReg=1'b1;RegWrite=1'b1;MemRead=1'b1;MemWrite=1'b0;
                    branch_eq=1'b0;branch_blt=1'b0;
                    ALU_Op[1:0]=2'b00;
                  end

      // If sel=1, output is b
                7'b0100011 :
                  begin
                  ALUsrc=1'b1;MemtoReg=1'bx; RegWrite=1'b0; MemRead=1'b0; MemWrite=1'b1;
                  branch_eq=1'b0;branch_blt=1'b0;
                  ALU_Op[1:0]=2'b00;
                  end

      // If sel=2, output is c
                7'b1100011:
                  begin
                    ALUsrc=1'b0;MemtoReg=1'bx;RegWrite=1'b0;MemRead=1'b0;MemWrite=1'b0;
                    if (func3==3'b000)
                    begin
                      branch_eq=1'b1;branch_blt=1'b0;
                    end
                    if (func3==3'b100)
                    begin
                      branch_eq=1'b0;branch_blt=1'b1;
                    end
      ALU_Op[1:0]=2'b01;
      end
                  7'b0010011:
                  begin
                    ALUsrc=1'b1;MemtoReg=1'b0;RegWrite=1'b1;MemRead=1'b0;MemWrite=1'b0;
                    branch_eq=1'b0;branch_blt=1'b0;
                    ALU_Op[1:0]=2'b00;
                  end
      // If sel is something, out is commonly
      endcase
      end
      endmodule
```

Figure 7: Changes in Control Unit for blt

2 Task 2: Modifying Processor: Pipelined Processor

2.1 Pipeline Registers

The following figures represent pipeline registers which are used to store values between stages. They take their new values on the positive edge of the clk and if "reset" equals 1 then their values are initialized to 0. Connections have been made between these 4 different register to include control information and circuitry.

```
initial
    begin
      rd_out = 0;ALU_result_out = 0;read_data_out = 0;reg_write_out= 0;mem_to_reg_out= 0;
    end

    always @(posedge clk or reset)
        begin
            if (reset)
                begin
                  rd_out = 0;
                  ALU_result_out = 0;
                  read_data_out = 0;
                  reg_write_out= 0;
                  mem_to_reg_out= 0;
                end
            else if (clk)
                begin
                  rd_out = rd;
                  ALU_result_out = ALU_result;
                  read_data_out = read_data;
                  reg_write_out= reg_write;
                  mem_to_reg_out= memtoreg;
                end
        end
endmodule
```

Figure 8: Changes in MEM/Write Back Register

```
module IF_ID(input clk, reset,
             input [31:0] instruction,
             input [63:0] PC_Out,
             output reg [31:0] IF_ID__instruction,
             output reg [63:0] IF_ID__PC_Out);

  initial
  begin
    IF_ID__instruction = 0;
    IF_ID__PC_Out = 0;
  end

    always @(posedge clk or reset)
        begin
            if (reset == 1'b1)
                begin
                    IF_ID__instruction = 0;
                    IF_ID__PC_Out = 0;
                end
          else if (  clk==1)
                begin
                    IF_ID__instruction = instruction;
                    IF_ID__PC_Out = PC_Out;
                end
        end
endmodule
```

Figure 9: Changes in instruction Fetch/Instruction Decode Register

```
module ID_EX( input brancheq,branchblt, MemRead, MemtoReg, MemWrite, clk,reset,ALUsrc, RegWrite,
              input [1:0] ALU_Op, input [63:0] readdata1,readdata2,immediate,pc_out,
             input [4:0] rs1, rs2, rd ,
             input[3:0] func,
             output reg brancheq_out,branchblt_out, MemRead_out, MemtoReg_out, MemWrite_out, ALUsrc_out, RegWrite_out,
             output reg [1:0] ALU_Op_out,
             output reg [63:0]  readdata1_out,readdata2_out,immediate_out,pc_out_out,
             output reg [4:0] rs1_out, rs2_out, rd_out ,
             output reg [3:0] func_out);

  initial
  begin
      brancheq_out=0;branchblt_out=0;MemRead_out=0;ALU_Op_out=0;
      MemWrite_out=0;ALUsrc_out=0;ALU_Op_out=0;RegWrite_out=0; readdata1_out=0;
      readdata2_out=0;immediate_out=0;pc_out_out=0;rs1_out= 0;rs2_out=0;rd_out=0;func_out=0;
    end

  always @(*)
    begin
  if (reset==1'b1)
    begin
      brancheq_out=0;branchblt_out=0;MemRead_out=0;MemtoReg_out=0;
      MemWrite_out=0;ALUsrc_out=0;ALU_Op_out=0;RegWrite_out=0;readdata1_out=0;
      readdata2_out=0;immediate_out=0;pc_out_out=0;rs1_out= 0;rs2_out=0;rd_out=0;func_out=0;
    end

  else if (clk==1)
    begin
        MemRead_out=MemRead;MemtoReg_out=MemtoReg;MemWrite_out=MemWrite;ALUsrc_out=ALUsrc;ALU_Op_out=ALU_Op;RegWrite_out=RegWrite;
        readdata1_out=readdata1;readdata2_out=readdata2;immediate_out=immediate;pc_out_out=pc_out;rs1_out= rs1;rs2_out=rs2;rd_out=rd;func_out=func;
    end
    end
endmodule
```

Figure 10: Changes in Instruction Decode/Execute Register

```
module EX_MEM(input clk, reset,input [4:0] rd,input [63:0] write_data , input branch_MUX,input [63:0] ALU_result, PC_out,
input zero,branch,MemRead, MemWrite,RegWrite,MemtoReg, output reg [4:0] rd_out ,output reg [63:0] write_data_out, output reg branch_MUX_out,output reg [63:0] ALU_result_out,
             output reg zero_out,branch_out,MemRead_out, MemWrite_out,RegWrite_out,MemtoReg_out, output reg [63:0] PC_out_out
);

initial
 begin
        PC_out_out=0;
        rd_out = 0;
        write_data_out=0;
        ALU_result_out = 0;
        branch_MUX_out=0;
        zero_out=0;
        branch_out=0;
        MemRead_out=0;
        MemWrite_out=0;
        RegWrite_out=0;
        MemtoReg_out=0;
    end

    always @(posedge clk ,posedge reset)

        begin

          if (reset==1)

                begin
                  PC_out_out=0;
                  rd_out = 0;
                  write_data_out=0;
                  ALU_result_out = 0;
                  branch_MUX_out=0;
                  zero_out=0;
                  branch_out=0;
                  MemRead_out=0;
                  MemWrite_out=0;
                  RegWrite_out=0;
                  MemtoReg_out=0;

                end

  else if (clk==1)
    begin
        PC_out_out=PC_out;
         rd_out=rd ;
        write_data_out=write_data;
         ALU_result_out=ALU_result ;
       branch_MUX_out=ALU_result ;
        zero_out= zero;
        branch_out=branch ;
        MemRead_out=MemRead;
        MemWrite_out=MemWrite;
        RegWrite_out= RegWrite ;
        MemtoReg_out=MemtoReg ;
    end
    end
endmodule
```

Figure 11: Changes in Execute/Mem Register

2.2 Forwarding Unit

The Forward Unit is responsible for comparing the rd values of the EX-MEM and MEM-WB instructions with rs1 and rs2 values of ID-EX. If they equal each other, the value of the regwrite is observed for EX-MEM and MEM-WB. In the case where it is 1 i.e. the next instructions is writng to the register Forward A and Forward B signals are sent accordingly to resolve the haazard.

Figure 13 shows the different requirements that must be fulfilled to generate the required forwarding signals. Once the necessary Forwarding signals have been generated they are sent to the MUX (Figure 14) which is responsible for deciding the signals that are to be used as inputs in the ALU unit.

```
607 module Forwarding_Unit(input [4:0] ID_EX__Rs1, ID_EX__Rs2,
608                       |input [4:0] EX_MEM__Rd,
609                        input EX_MEM__RegWrite,
610                        input [4:0] MEM_WB__Rd,
611                        input MEM_WB__RegWrite,
612                        output reg [1:0] Forward_A, Forward_B);
613
614     always @(*)
615         begin
616           //checking with the next instruction (rs1) with rd
617         if (EX_MEM__Rd == ID_EX__Rs1 && EX_MEM__RegWrite == 1 && EX_MEM__Rd != 0)
618           begin
619                 Forward_A = 2'b10;   //10
620             end
621           //checking with the next to next instruction (rs1) with rd and generating control signal for rs1 mux according to the textbook
622         else if (MEM_WB__Rd == ID_EX__Rs1 && MEM_WB__RegWrite == 1 && MEM_WB__Rd != 0
623                 && !(EX_MEM__RegWrite == 1 && EX_MEM__Rd != 0 && EX_MEM__Rd == ID_EX__Rs1))
624             begin
625                 Forward_A = 2'b01;  //01
626             end
627         else
628             begin
629                 Forward_A = 2'b00; //00
630             end
631
632         //FORWARD B LOGIC
633         //checking with the next instruction (rs2) with rd and generating control signal according to the textbook table
634
635         if (EX_MEM__Rd == ID_EX__Rs2 && EX_MEM__RegWrite == 1 && EX_MEM__Rd != 0)
636             begin
637                 Forward_B = 2'b10;   //10
638             end
639             //checking with the next instruction (rs2) with rd and generating control signal according to the textbook table
640         else if (MEM_WB__Rd == ID_EX__Rs2 && MEM_WB__RegWrite == 1 && MEM_WB__Rd != 0
641                 &&!(EX_MEM__RegWrite == 1 && EX_MEM__Rd != 0 && EX_MEM__Rd == ID_EX__Rs2))
642             begin
643                 Forward_B = 2'b01;  //01
644             end
645         // No Hazard because of rs2
646         else
647             begin
648                 Forward_B = 2'b00;  //00
649             end
650         end
651 endmodule
652
```

Figure 12: Fowarding Unit

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Figure 13: Mux Control Signals

```
module fourinputmux(input [63:0] one, two ,three, four , input [1:0] sel_pin, output reg [63:0] mux_result)
  always @(*)
    begin
      if (sel_pin==2'b00)
        mux_result=one;
      else if (sel_pin ==2'b01)
        mux_result=two;
        else if (sel_pin==2'b10)
          mux_result=three;
      else if (sel_pin==2'b11)
        mux_result=four;

    end

endmodule
```

Figure 14: Mux

```
0x00a00493          addi x9 x0 10                addi x9,x0,10
0x00700413          addi x8 x0 7                 addi x8 , x0,7
0x008484b3          add x9 x9 x8                 add x9,x9,x8
0x008484b3          add x9 x9 x8                 add x9,x9,x8
```

Figure 15: Test Cases used to test forwarding

TASK 2:

EDA playground link: https://edaplayground.com/x/jBrB

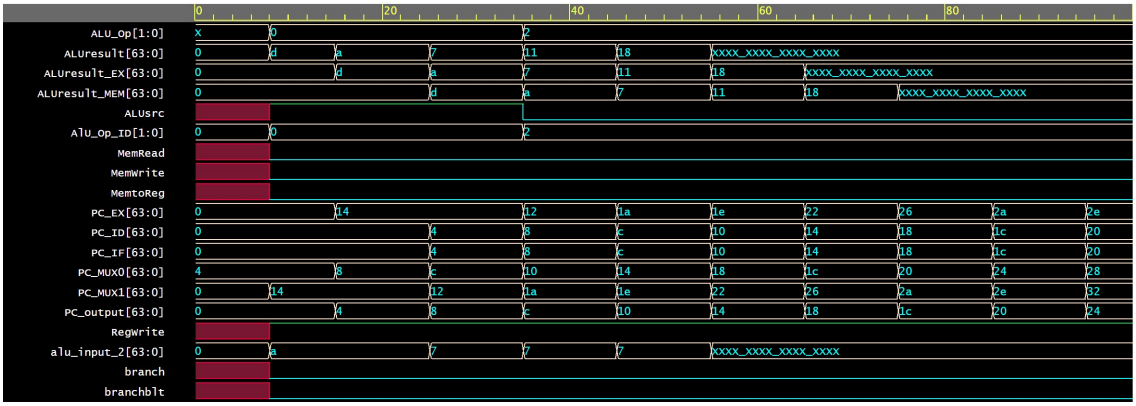

Figure 16: Test Cases used to test forwarding

3 Task 3: Detecting Hazards

3.1 Hazard Detection Module

The hazard detection(Figure 15) has the ability to introduce the effect equivalent to that of a bubble. This is done when the next instruction is a load hence a valid hazard. To achieve this result the mux controller is set to zero thereby setting the control signals to zero making the instruction an effective zero. Along with this unit, a seperate mux (Figure16) is used which takes input as the mux controller pin and changes them to zero if the need be. Furthermore, pc-write and if-write signals which prevent writing to pc and pipeline registers when the hazard detection unit is activated to introduce the bubble effect.

```
module hazard_detection_unit(input [4:0] if_id_rs1, if_id_rs2, id_ex_rd, input Memread, output reg mux_controller, PC_write, if_id_write
  initial
    begin
       mux_controller=1;
        PC_write=1;
      if_id_write=1;

    end
  always @(*)
    begin
      if ((if_id_rs1==id_ex_rd || if_id_rs2==id_ex_rd) &&  Memread==1       )
        begin
        mux_controller=0;
        PC_write=0;


        if_id_write=0;
        end

      else
      begin
      mux_controller=1;
        PC_write=1;
      if_id_write=1;
        end

    end
endmodule
```

Figure 17: Hazard Detection

This section of the code present in the module IF-ID specifically prevents it from writing back.

```
module Hazard_detection_MUX( input  sel, branch_eq, MemRead, MemtoReg, MemWrite, ALUsrc, RegWrite,
                            input [1:0] ALU_Op,input branch_blt,
                            output reg branch_eq_haz, MemRead_haz, MemtoReg_haz, MemWrite_haz, ALUsrc_haz, RegWrite_haz,
                            output reg [1:0] ALU_Op_haz,
                            output reg branch_blt_haz);

  always @ (*)
    begin
      if (sel==0)
        begin
          branch_eq_haz=0;
          MemRead_haz=0;
          MemtoReg_haz=0;
         MemWrite_haz=0;
          ALUsrc_haz=0;
          RegWrite_haz=0;
          ALU_Op_haz=0;
          branch_blt_haz=0;
        end
      if (sel==1)
        begin
          branch_eq_haz=branch_eq;
          MemRead_haz=MemRead;
          MemtoReg_haz=MemtoReg;
         MemWrite_haz=MemWrite;
          ALUsrc_haz=ALUsrc;
          RegWrite_haz=RegWrite;
          ALU_Op_haz=ALU_Op;
          branch_blt_haz=branch_blt;
          end

    end
endmodule
```

Figure 18: Hazard Detection Mux

```
            end
    else if ( if_id_write==1)
            begin
                IF_ID__instruction = instruction;
                IF_ID__PC_Out = PC_Out;
            end
end
```

Figure 19: Hazard Detection Mux

3.2 Testing Performance of Modules

The EP_Wave shows that the final result is 17 in hexadecimal i.e 23 which is the expected result hence proving that the alterations made to modules in section 3.1 are correct. Despite the apparent hazard in the assembly our pipelined processor resolves the issue.

```
addi x10,x0,0
lw x9,50(x10)
addi x9,x9,1
```

(a) Assembly Code

```
Registers[48]=0;
Registers[49]=0;
  Registers[50]=64'd22;
Registers[51]=0;
Registers[52]=0;
```

(b) Data-Memory

Figure 20: Testing

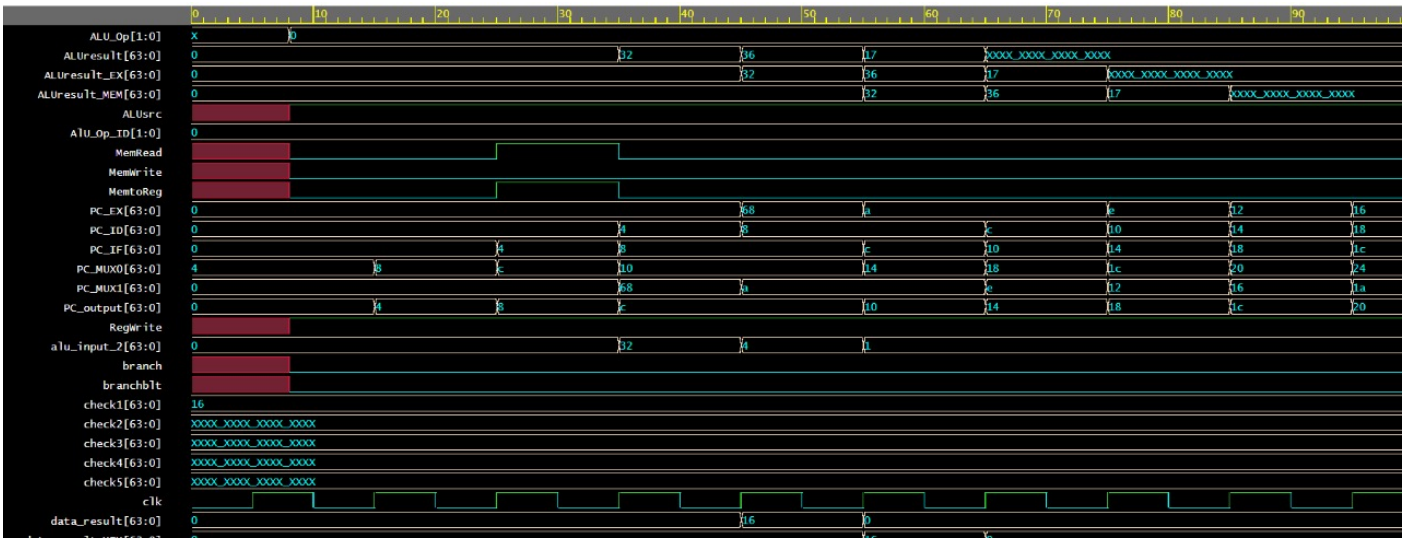

Figure 21: EP-Wave results

TASK 3:

EDA playground link: https://edaplayground.com/x/Xg_i