# Habib University


## Dhanani School of Science and Engineering

### Computer Architecture
### CE/CS 321/330

### T3

**Final Project Report**

Fakeha Faisal (08288)
Marium Zeeshan (08488)
Syeda Yashfeen Zehra Zaidi (08469)

Research Assistant: Saima Shaheen

30 Apr 2024

# Contents

# Introduction

## 1.1 Objective

To build a 5-stage pipelined processor capable of executing any one array sorting algorithm, we will need to convert a single-cycle processor into a pipelined one. The following implementations will be done to achieve this goal.

1) Implementing branch module ( beq, bgt, blt)
2) Applying a sorting algorithm (bubble sort in our case)
3) Forwarding
4) Hazard detection
5) Pipelining

### 1.1.1 Sorting Algorithm

We will be using Bubble Sort to test our processor functionality.

```
add x21,x0,x0 #start adress of the array #0
addi x22,x0,8 #size of the array to sort #4
add x19,x0,x0 #i initialized to 0
for1tst: #outer loop
blt x19,x22, here #if i < size jump to here #12
beq x0, x0, exit1 # if not jumpt out of outer loop #16
here:
addi x20,x19,-1 #j=i-1 #20
for2tst: #inner loop
blt x20,x0,exit2 #if j<0 exit from innerloop #24
add x1,x20,x20 #28
add x5,x20,x20 #32
add x5,x5,x1 #36
add x5,x5,x1 #48
add x5,x5,x1 #x5=x20 slli 3 #44
add x5,x21,x5 # x5+= adress of array #48
lw x6,0(x5) # load elements from th and j+1th location and if jth item is lesser than jump to exit2
lw x7,8(x5) #56
blt x6,x7,exit2 #60
```

```
lw x7,8(x5) #56
blt x6,x7,exit2 #60
swap: #else swap
add x8,x20,x20 #64
add x13,x20,x20 #68
add x8, x8, x13 #72
add x8,x8,x13 #76
add x8, x8, x13 #80
add x8,x8,x21#84
lw x9,0(x8)#88
lw x12,8(x8)#92
sw x12,0(x8)#96
sw x9,8(x8)#100
addi x20,x20,-1#104
beq x0, x0, for2tst#108
exit2:
addi x19,x19,1 #112
beq x0, x0, for1tst #116
exit1:
```

Fig 1.1.1: Bubble Sort Algorithm in Assembly

# Tasks

## 2.1 Task 1

### 2.1.1 RISC V Implementation (Single Cycle)

We implemented the converted bubble sort algorithm on the RISC V single-cycle processor developed in our lab. To make the processor compatible with the algorithm, we made some adjustments. Specifically, we made changes to the single-cycle processor to be able to detect branch instructions and execute the code accordingly. We then tested that the bubble sort ran correctly on the single-cycle processor. These modifications enabled us to run the sorting algorithm effectively on the RISC V processor.

### 2.1.2 Changes

We modified the bubble sort algorithm to accommodate the limitations of the Venus simulator. First, we wrote our bubble sort code in the RISC-V assembly language on Venus, then we converted that code to machine language using the website here. The converted code was written in called instruction memory and the element values to be sorted were written in data memory. Furthermore, changes were made to the single-cycle processor to detect branch instructions and execute the code accordingly. As shown below, a Branch Unit module was added to support the bgt and blt instructions

```
module branch
  (
   input [2:0] funct3,
    input signed [63:0] readData1,
    input signed [63:0] b,
   output reg addermuxselect
  );

  initial
    begin
      addermuxselect = 1'b0;
    end

  always @(*)
    begin
      case (funct3)
        3'b000:
          begin
            if (readData1 == b)
              addermuxselect = 1'b1;
            else
              addermuxselect = 1'b0;
            end
         3'b100:
            begin
              if (readData1 < b)begin
              addermuxselect = 1'b1;
              end

            else begin
              addermuxselect = 1'b0;
              end
            end
```

Fiq 2.1.2: Branch Unit Module

```
`timescale 1ns / 1ps


module Data_Memory
(
input [63:0] Mem_Addr,
input [63:0] Write_Data,
input clk, MemWrite, MemRead,
output reg [63:0] Read_Data,
 output [63:0] element1,
  output [63:0] element2,
  output [63:0] element3,
  output [63:0] element4,
  output [63:0] element5,
  output [63:0] element6,
  output [63:0] element7,
  output [63:0] element8
);
reg [7:0] DataMemory [255:0];
integer i;
initial
begin
for (i = 0;i<256;i = i + 1)begin
    DataMemory[i] = 0;
end
      DataMemory[0] = 8'd53;
      DataMemory[8] = 8'd156;
      DataMemory[16] = 8'd13;
      DataMemory[24] = 8'd63;
      DataMemory[32] = 8'd4;
      DataMemory[40] = 8'd14;
      DataMemory[48] = 8'd99;
      DataMemory[56] = 8'd20;
end

 assign element1 = {DataMemory[7],DataMemory[6],DataMemory[5],DataMemory[4],DataMemory[3],DataMemory[2],DataMemory[1],DataMemory[0]};
 assign element2 = {DataMemory[15],DataMemory[14],DataMemory[13],DataMemory[12],DataMemory[11],DataMemory[10],DataMemory[9],DataMemory[8]};
 assign element3 = {DataMemory[23],DataMemory[22],DataMemory[21],DataMemory[20],DataMemory[19],DataMemory[18],DataMemory[17],DataMemory[16]};
 assign element4 = {DataMemory[31],DataMemory[30],DataMemory[29],DataMemory[28],DataMemory[27],DataMemory[26],DataMemory[25],DataMemory[24]};
 assign element5 = {DataMemory[39],DataMemory[38],DataMemory[37],DataMemory[36],DataMemory[35],DataMemory[34],DataMemory[33],DataMemory[32]};
 assign element6 = {DataMemory[47],DataMemory[46],DataMemory[45],DataMemory[44],DataMemory[43],DataMemory[42],DataMemory[41],DataMemory[40]};
 assign element7 = {DataMemory[55],DataMemory[54],DataMemory[53],DataMemory[52],DataMemory[51],DataMemory[50],DataMemory[49],DataMemory[48]};
 assign element8 = {DataMemory[63],DataMemory[62],DataMemory[61],DataMemory[60],DataMemory[59],DataMemory[58],DataMemory[57],DataMemory[56]};


always @ (*)
begin
if (MemRead)
Read_Data =
{DataMemory[Mem_Addr+7],DataMemory[Mem_Addr+6],DataMemory[Mem_Addr+5],DataMemory[Mem_Addr+4],DataMemory[Mem_Addr+3],DataMemory[Mem_Addr+2],DataMemory[Mem_Addr+1]
end
always @ (posedge clk)
begin
if (MemWrite)
begin
DataMemory[Mem_Addr] = Write_Data[7:0];
DataMemory[Mem_Addr+1] = Write_Data[15:8];
DataMemory[Mem_Addr+2] = Write_Data[23:16];
DataMemory[Mem_Addr+3] = Write_Data[31:24];
DataMemory[Mem_Addr+4] = Write_Data[39:32];
DataMemory[Mem_Addr+5] = Write_Data[47:40];
DataMemory[Mem_Addr+6] = Write_Data[55:48];
DataMemory[Mem_Addr+7] = Write_Data[63:56];
end
```

Fig: 2.1.2a: Data Memory

```
`timescale 1ns / 1ps

module Instruction_Memory
(
input [63:0] Inst_Address,
output reg [31:0] Instruction
);
reg [7:0] inst_mem [135:0];
initial
begin
{inst_mem[3], inst_mem[2], inst_mem[1], inst_mem[0]} = 32'h00000AB3;
      {inst_mem[7], inst_mem[6], inst_mem[5], inst_mem[4]} = 32'h00800B13;
      {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h000009B3;
      {inst_mem[15], inst_mem[14], inst_mem[13], inst_mem[12]} = 32'h0169c463;
      {inst_mem[19], inst_mem[18], inst_mem[17], inst_mem[16]} = 32'h06000c63;
      {inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'hfff98a13;
      {inst_mem[27], inst_mem[26], inst_mem[25], inst_mem[24]} = 32'h040a4c63;
      {inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h014a00b3;
      {inst_mem[35], inst_mem[34], inst_mem[33], inst_mem[32]} = 32'h014a02b3;
      {inst_mem[39], inst_mem[38], inst_mem[37], inst_mem[36]} = 32'h001282b3;
      {inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'h001282b3;
      {inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h001282b3;
      {inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'h005a82b3;
      {inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'h0002B303;
      {inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'h0082B383;
      {inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'h02734a63;
      {inst_mem[67], inst_mem[66], inst_mem[65], inst_mem[64]} = 32'h014a0433;
      {inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'h014a06b3;
      {inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} = 32'h00d40433;
      {inst_mem[79], inst_mem[78], inst_mem[77], inst_mem[76]} = 32'h00d40433;
      {inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]} = 32'h00d40433;
      {inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} = 32'h01540433;
      {inst_mem[91], inst_mem[90], inst_mem[89], inst_mem[88]} =32'h00043483;
      {inst_mem[95], inst_mem[94], inst_mem[93], inst_mem[92]}=32'h00843603;
      {inst_mem[99], inst_mem[98], inst_mem[97], inst_mem[96]}=32'h00c43023; //
      {inst_mem[103], inst_mem[102], inst_mem[101], inst_mem[100]}=32'h00943423; //
      {inst_mem[107], inst_mem[106], inst_mem[105], inst_mem[104]}=32'hfffa0a13;
      {inst_mem[111], inst_mem[110], inst_mem[109], inst_mem[108]}=32'hfa0006e3;
     {inst_mem[115], inst_mem[114], inst_mem[113], inst_mem[112]}= 32'h00198993;
      {inst_mem[119], inst_mem[118], inst_mem[117], inst_mem[116]}=32'hf8000ce3;
      {inst_mem[123], inst_mem[122], inst_mem[121], inst_mem[120]}=32'h00000000;
      {inst_mem[127], inst_mem[126], inst_mem[125], inst_mem[124]}=32'h00000000;
      {inst_mem[131], inst_mem[130], inst_mem[129], inst_mem[128]}=32'h00000000;
      {inst_mem[135], inst_mem[134], inst_mem[133], inst_mem[132]}=32'h00000000;
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];
      Instruction[31:24] = inst_mem[Inst_Address+3];
end
endmodule
```

Fig 2.1.2b: Instruction Memory

### 2.1.3 Code and Waveform

The link for the code can be found here.

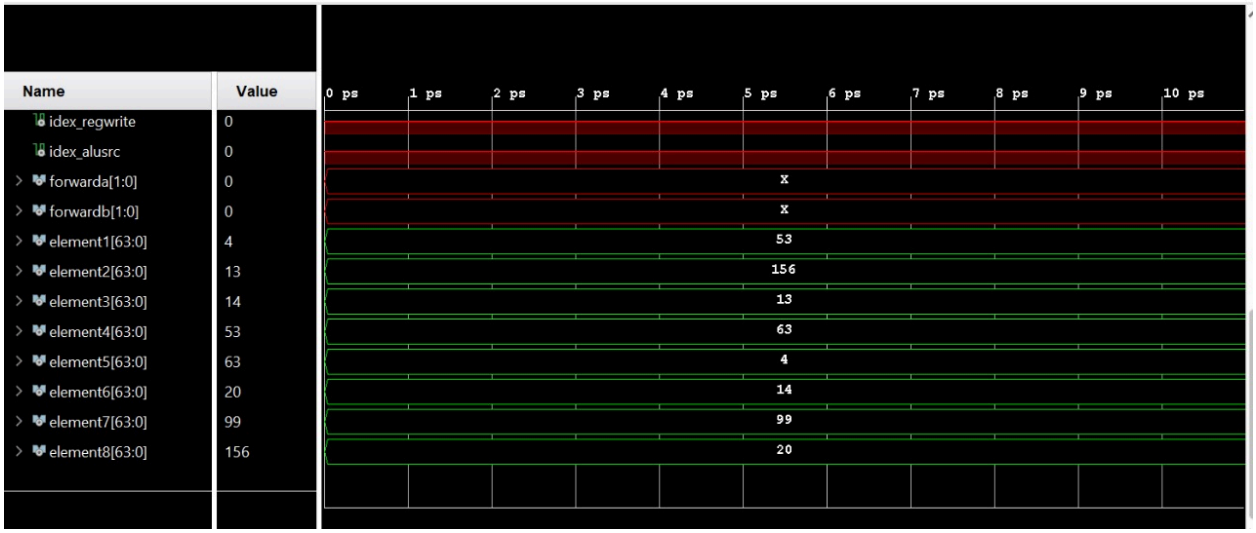



Fig2.1.3a: Before sorting

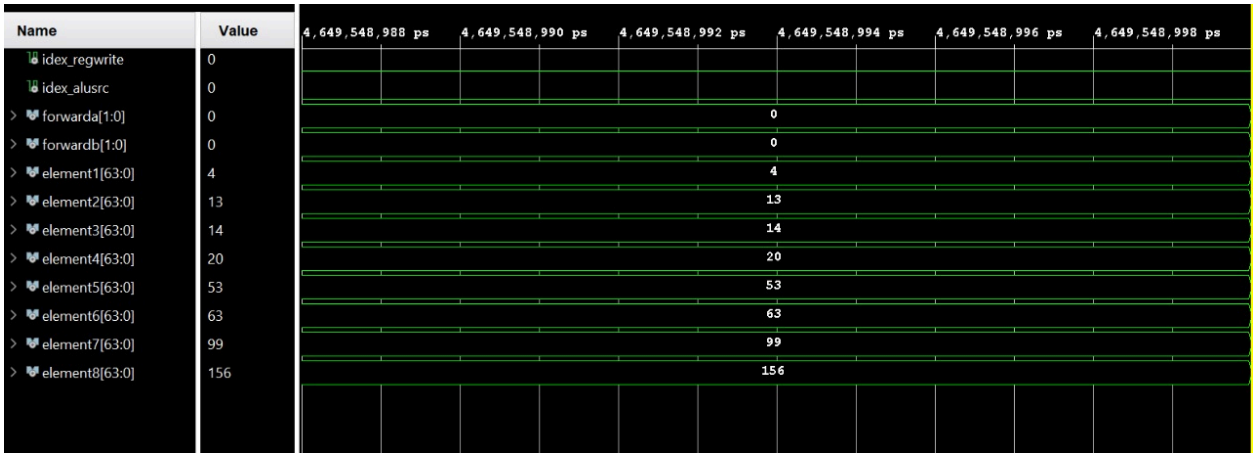



Fig 2.1.3b: After sorting

## 2.2 Task 2

### 2.2.1 Testing 5 stage/Pipelined RISC V Processor for Sorting Algorithm

After implementing the algorithm on the single-cycle processor, our next step was to introduce pipeline stages. To do this, we referred to our course materials and incorporated intermediate registers for the pipeline, namely IF/ID, ID/EX, EX/MEM, and MEM/WB. These registers help store and pass on data from preceding instructions along the pipeline.

### 2.2.2 Changes

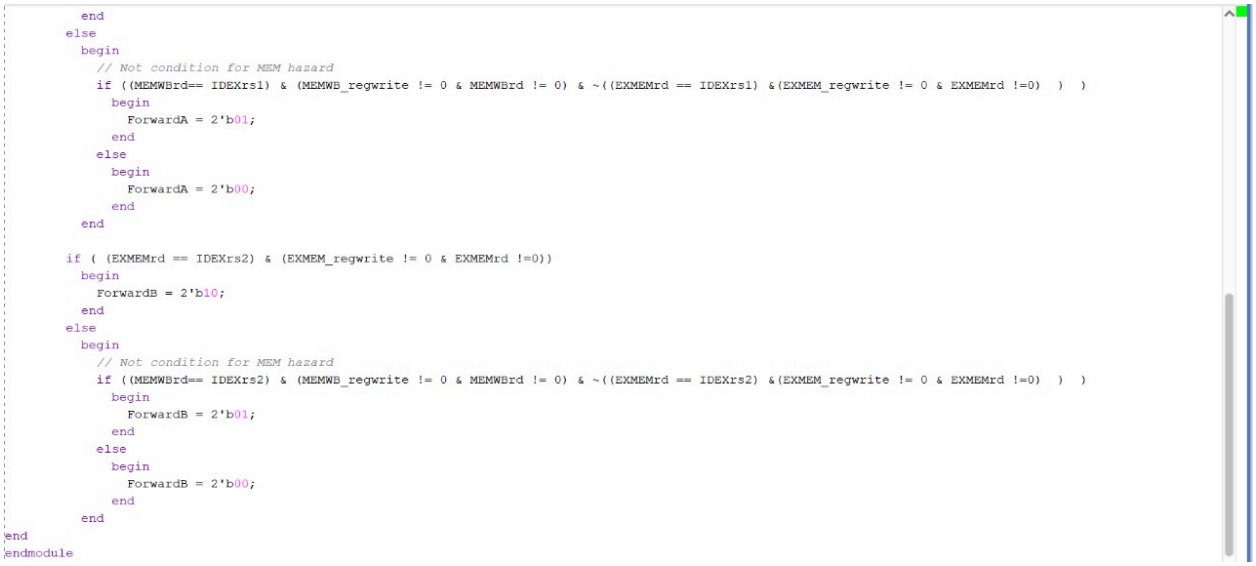
To implement a pipelined design, Instead of creating separate modules for these registers, we chose to implement our approach in risc V file to ensure the retention of register values across clock cycles, crucial for achieving pipelining. Because, when we implemented the modules, because of the wires, the data was lost. To test the forwarding unit's functionality, we devised a test case consisting of three instructions stored in the instruction memory.

Additionally, we included a forwarding unit that facilitates data transmission within the pipeline. We have chosen test cases, that do not require stalls and therefore no need for a hazard detection unit to be called however we had already implemented it.

```
module ForwardingUnit( input [4:0] IDEXrs1,input [4:0] IDEXrs2,input [4:0]EXMEMrd,//rdmem
input [4:0]MEMWBrd,
input MEMWB_regwrite,input EXMEM_regwrite,
output reg [1:0] ForwardA,ForwardB
);

  always @(*)
    begin
        if ( (EXMEMrd == IDEXrs1) & (EXMEM_regwrite != 0 & EXMEMrd !=0))
          begin
            ForwardA = 2'b10;
          end
        else
          begin
            // Not condition for MEM hazard
            if ((MEMWBrd== IDEXrs1) & (MEMWB_regwrite != 0 & MEMWBrd != 0) & ~((EXMEMrd == IDEXrs1) &(EXMEM_regwrite != 0 & EXMEMrd !=0)  )  )
              begin
                ForwardA = 2'b01;
              end
            else
              begin
                ForwardA = 2'b00;
              end
          end

        if ( (EXMEMrd == IDEXrs2) & (EXMEM_regwrite != 0 & EXMEMrd !=0))
          begin
            ForwardB = 2'b10;
          end
        else
          begin
            // Not condition for MEM hazard
```



```
        end
      else
        begin
          // Not condition for MEM hazard
          if ((MEMWBrd== IDEXrs1) & (MEMWB_regwrite != 0 & MEMWBrd != 0) & ~((EXMEMrd == IDEXrs1) &(EXMEM_regwrite != 0 & EXMEMrd !=0)  )  )
            begin
              ForwardA = 2'b01;
            end
          else
            begin
              ForwardA = 2'b00;
            end
        end

      if ( (EXMEMrd == IDEXrs2) & (EXMEM_regwrite != 0 & EXMEMrd !=0))
        begin
          ForwardB = 2'b10;
        end
      else
        begin
          // Not condition for MEM hazard
          if ((MEMWBrd== IDEXrs2) & (MEMWB_regwrite != 0 & MEMWBrd != 0) & ~((EXMEMrd == IDEXrs2) &(EXMEM_regwrite != 0 & EXMEMrd !=0)  )  )
            begin
              ForwardB = 2'b01;
            end
          else
            begin
              ForwardB = 2'b00;
            end
        end
end
endmodule
```

Fig 2.2.2a: Forwarding Unit

The forwarding unit in this RISC-V processor module is in charge of managing data hazards within the pipeline stages. It takes in information about the current and previous pipeline stages, such as register identifiers and control signals. The unit evaluates conditions to determine if data forwarding is necessary for each source operand of the instruction in the ID/EX stage. It first checks if the destination register of the instruction in the EX/MEM stage matches the source register of the current instruction; if so, and if the instruction in the EX/MEM stage is writing back to a register, data forwarding is enabled from the EX/MEM stage to the ID/EX stage. If no forwarding is required from the EX/MEM stage, it checks for a similar condition with the MEM/WB stage. If forwarding is needed from the MEM/WB stage and not from the EX/MEM stage, data is forwarded from the MEM/WB stage to the ID/EX stage. If no forwarding is necessary from either stage, the forwarding unit signals (forwardA/forwardB=0) that no forwarding is required. This unit helps mitigate stalls in the pipeline by ensuring that the processor uses the most recent data available for executing instructions, enhancing overall performance.

## Risc_Processor.v module (Top Module)

```
`timescale 1ns / 1ps

module RISC_V_Processor(
input clk,
input reset,
output wire wire_stall,
output reg stall,
output reg flush,
output reg exmem_branch,exmem_branchfinale,
output wire [63:0] ex_alu2in,
  output wire [63:0] pc_temp,
output wire [63:0] pc_out,
output reg [63:0] pc_in, //wire

output wire [31:0] if_inst,
output reg [31:0] ifid_inst,
output reg   idex_branch, idex_memread, idex_memtoreg, idex_memwrite, idex_regwrite, idex_alusrc,
 output wire [1:0] forwarda, output wire [1:0] forwardb,
output wire [63:0] element1,
  output wire [63:0] element2,
  output wire [63:0] element3,
  output wire [63:0] element4,
  output wire [63:0] element5,
```

```
output wire [63:0] element1,
  output wire [63:0] element2,
  output wire [63:0] element3,
  output wire [63:0] element4,
  output wire [63:0] element5,
  output wire [63:0] element6,
  output wire [63:0] element7,
  output wire [63:0] element8
    );
    //if
   // reg stall;
    wire [63:0] if_adder4;
    //ifid
   reg [63:0] ifid_pcout;

   //id
      wire [6:0] id_opcode;
wire [4:0] id_rd;
wire [2:0] id_funct3;
wire [4:0] id_rs1;
wire [4:0] id_rs2;
wire [6:0] id_funct7;
wire  id_branch, id_memread, id_memtoreg, id_memwrite, id_alusrc, id_regwrite;
```

```
 wire [6:0] id_funct7;
 wire  id_branch, id_memread, id_memtoreg, id_memwrite, id_alusrc, id_regwrite;
wire [1:0] id_aluop;
    wire [63:0] id_readdata1,id_readdata2,id_immdata;

   //ex
   wire [63:0] ex_adderimm,ex_alu1in,ex_muxB;

  wire [3:0] ex_operation;
  wire ex_zero,ex_branchfinale;
  wire [63:0] ex_aluresult;


    //idex
     reg [63:0] idex_pcout,idex_readdata1,idex_readdata2, idex_immdata;
     reg [4:0] idex_rs1, idex_rs2, idex_rd;
     reg [3:0] idex_funct;
    // reg   idex_branch, idex_memread, idex_memtoreg, idex_memwrite, idex_regwrite, idex_alusrc;
     reg [1:0]idex_aluop;
    //exmem
    reg [63:0] exmem_adderimm, exmem_aluresult,exmem_writedataout;
    reg exmem_regwrite,exmem_zero,exmem_memread, exmem_memtoreg, exmem_memwrite;
    reg [4:0] exmem_rd;
     reg [1:0]idex_aluop;
     //exmem
     reg [63:0] exmem_adderimm, exmem_aluresult,exmem_writedataout;
     reg exmem_regwrite,exmem_zero,exmem_memread, exmem_memtoreg, exmem_memwrite;
     reg [4:0] exmem_rd;
     //mem
     wire [63:0] datamemreaddata;
     //memwb
     reg [4:0] memwb_rd;
   reg memwb_regwrite,memwb_memtoreg;
reg [63:0] memwb_readdataout, memwb_aluresult;
  //wb
//  wire [63:0] mem_writedata;
  reg [63:0] pc_reg;
   wire [63:0] mem_writedata;
  //ifstage


  //  Mux m1(if_adder4,exmem_adderimm,(exmem_branch && exmem_branchfinale) , pc_in);
always@(*)begin
if (stall==1'b0) begin
if ((exmem_branch && exmem_branchfinale)==1'b0)
pc_in = if_adder4;
```

```
  //ifstage


  //  Mux m1(if_adder4,exmem_adderimm,(exmem_branch && exmem_branchfinale) , pc_in);
always@(*)begin
if (stall==1'b0) begin
if ((exmem_branch && exmem_branchfinale)==1'b0)
pc_in = if_adder4;
else if ((exmem_branch && exmem_branchfinale)==1'b1)
pc_in = exmem_adderimm;
end
end
    prog_counter pc(clk, reset,pc_reg,pc_temp, pc_in,pc_out,stall);
//    always@(*) begin
//    pc_reg<=pc_temp;end
    Instruction_Memory im(pc_out, if_inst);
    Adder a1(pc_out, 64'd4, if_adder4);
    always @(posedge clk) begin
    if (reset == 1'b1 || flush==1'b1) begin
        ifid_pcout <= 0;
        ifid_inst <= 0;
        end
     else if (stall==1'b1)begin
```

```
        end
     else if (stall==1'b1)begin
     end
    else if (stall==1'b0|| stall===1'bx) begin
       ifid_pcout <= pc_out;
        ifid_inst <= if_inst;
        end
end
   //IF_ID ifid(stall,clk, reset, pc_out,if_inst, ifid_pcout, ifid_inst);

  //id stage

  instructionparser ip(ifid_inst, id_opcode, id_rd, id_funct3, id_rs1, id_rs2, id_funct7);
  Hazard_Detection hd(id_rs1, id_rs2, idex_rd,
    idex_memread, wire_stall // IFID_Write, PC_Write, IDEX_MuxOut,
    );

      always@(*) begin
 stall=wire_stall;end
//HAZARDD
//always @(posedge clk) begin
//    if (idex_memread && (idex_rd == id_rs1 || idex_rd == id_rs2)) begin
//        stall = 1; // IDEX_MuxOut = 1; IFID_Write = 0; PC_Write = 0;
```

```
//    end
//end
control_unit cu(id_opcode,stall, id_branch, id_memread, id_memtoreg, id_memwrite, id_alusrc, id_regwrite, id_aluop);

register_file rf(mem_writedata, id_rs1, id_rs2, memwb_rd, memwb_regwrite, clk, reset, id_readdata1, id_readdata2);
imm_gen imm_gen(ifid_inst, id_immdata);

//ID_EX idex( clk, reset,  ifid_pcout,id_readdata1,id_readdata2, id_immdata,   id_rs1, id_rs2, id_rd,
//                {ifid_inst[30],ifid_inst[14:12] },  id_branch, id_memread, id_memtoreg, id_memwrite, id_regwrite, id_alus
//               id_aluop,  idex_pcout,idex_readdata1,idex_readdata2, idex_immdata,
//               idex_rs1, idex_rs2, idex_rd,   idex_funct,   idex_branch, idex_memread,
//               idex_memtoreg, idex_memwrite, idex_regwrite, idex_alusrc,   idex_aluop);
always @(posedge clk) begin
    if (reset == 1'b1 ||flush==1'b1  )begin
        idex_pcout <= 0;
        idex_readdata1 <= 0;
        idex_readdata2 <= 0;
         idex_immdata <= 0;
        idex_rs1 <= 0;
        idex_rs2 <= 0;
        idex_rd <= 0;
        idex_funct <= 0;
        idex_branch <= 0;
```

```
        idex_readdata1 <= 0;
        idex_readdata2 <= 0;
         idex_immdata <= 0;
        idex_rs1 <= 0;
        idex_rs2 <= 0;
        idex_rd <= 0;
        idex_funct <= 0;
        idex_branch <= 0;
        idex_memread <= 0;
        idex_memtoreg <= 0;
        idex_memwrite <= 0;
        idex_regwrite <= 0;
        idex_alusrc <= 0;
        idex_aluop <= 0;
        end
    else if (stall==1'b1) begin
    idex_pcout <= ifid_pcout;
        idex_readdata1 <= id_readdata1;
        idex_readdata2 <= id_readdata2;
        idex_immdata <= id_immdata;
        idex_rs1 <= id_rs1;
        idex_rs2 <= id_rs2;
        idex_rd <= id_rd;
```

```
        idex_readdata2 <= id_readdata2;
        idex_immdata <= id_immdata;
        idex_rs1 <= id_rs1;
        idex_rs2 <= id_rs2;
        idex_rd <= id_rd;
        idex_funct <= {ifid_inst[30],ifid_inst[14:12] };
        idex_branch <= 0;
        idex_memread <= 0;
        idex_memtoreg <= 0;
        idex_memwrite <= 0;
        idex_regwrite<= 0;
        idex_alusrc <= 0;
        idex_aluop <= 0;
    end
    else begin
         idex_pcout <= ifid_pcout;
        idex_readdata1 <= id_readdata1;
        idex_readdata2 <= id_readdata2;
        idex_immdata <= id_immdata;
        idex_rs1 <= id_rs1;
        idex_rs2 <= id_rs2;
        idex_rd <= id_rd;
        idex_funct <= {ifid_inst[30],ifid_inst[14:12] };
```

```
      idex_immdata <= id_immdata;
      idex_rs1 <= id_rs1;
      idex_rs2 <= id_rs2;
      idex_rd <= id_rd;
      idex_funct <= {ifid_inst[30],ifid_inst[14:12] };
      idex_branch <= id_branch;
      idex_memread <= id_memread;
      idex_memtoreg <= id_memtoreg;
      idex_memwrite <= id_memwrite;
      idex_regwrite<= id_regwrite;
      idex_alusrc <= id_alusrc;
      idex_aluop <= id_aluop;
      end

   end

//ex
Adder a2(idex_pcout, idex_immdata*2, ex_adderimm);
ForwardingUnit fu(idex_rs1, idex_rs2, exmem_rd, memwb_rd, memwb_regwrite, exmem_regwrite, forwarda, forwardb);

Mux_3 mm(idex_readdata1, mem_writedata, exmem_aluresult, forwarda, ex_alu1in);

Mux_3 mn(idex_readdata2, mem_writedata, exmem_aluresult, forwardb, ex_muxB);
```

```
//ex
Adder a2(idex_pcout, idex_immdata*2, ex_adderimm);
ForwardingUnit fu(idex_rs1, idex_rs2, exmem_rd, memwb_rd, memwb_regwrite, exmem_regwrite, forwarda, forwardb);

Mux_3 mm(idex_readdata1, mem_writedata, exmem_aluresult, forwarda, ex_alu1in);

Mux_3 mn(idex_readdata2, mem_writedata, exmem_aluresult, forwardb, ex_muxB);

Mux m2(ex_muxB, idex_immdata, idex_alusrc, ex_alu2in);
alu_control alu_c(idex_aluop, idex_funct, ex_operation);
ALU_64_bit alu(ex_alu1in, ex_alu2in, ex_operation, ex_aluresult, ex_zero);
branch bu(idex_funct[2:0], ex_alu1in, ex_alu2in, ex_branchfinale);

  //branch bu({ifid_inst[30],ifid_inst[14:12] }, id_readdata1, idreaddata2, id_branchfinale);
```

```
    always @(posedge clk)
    begin
      if (reset == 1'b1 ||flush==1'b1 )
        begin
          exmem_adderimm <= 64'b0;
          exmem_zero <= 1'b0;
          exmem_aluresult <= 63'b0;
          exmem_writedataout<= 64'b0;
          exmem_rd <= 5'b0;
          exmem_branch <= 1'b0;
          exmem_memread <= 1'b0;
          exmem_memtoreg <=1'b0;
          exmem_memwrite <= 1'b0;
          exmem_regwrite <= 1'b0;
          exmem_branchfinale<=1'b0;

        end
      else
        begin
         exmem_adderimm <= ex_adderimm;
          exmem_zero <= ex_zero;
          exmem_aluresult <= ex_aluresult;
          exmem_writedataout<= ex_muxB;
```

```
          exmem_aluresult <= ex_aluresult;
          exmem_writedataout<= ex_muxB;
          exmem_rd <= idex_rd;
          exmem_branch <= idex_branch;
          exmem_memread <= idex_memread;
          exmem_memtoreg <=idex_memtoreg;
          exmem_memwrite <= idex_memwrite;
          exmem_regwrite <= idex_regwrite;
          exmem_branchfinale<=ex_branchfinale;

        end
    end
//mem

Data_Memory dm(exmem_aluresult, exmem_writedataout, clk, exmem_memwrite, exmem_memread, datamemreaddata,element1,element2,

always @(*)
    begin
      if (exmem_branch == 1'b1 && exmem_branchfinale==1'b1)
        flush = 1'b1;
      else
        flush = 1'b0;
    end
```

```
//                          memwb_readdataout, memwb_aluresult, memwb_rd, memwb_memtoreg, memwb_regwrite);
always @(posedge clk)
    begin
      if (reset == 1'b1 )
        begin
          memwb_readdataout <= 63'b0;
          memwb_aluresult <= 63'b0;
          memwb_rd <= 5'b0;
          memwb_memtoreg <= 1'b0;
          memwb_regwrite <= 1'b0;

        end
      else
        begin
          memwb_readdataout <= datamemreaddata;
          memwb_aluresult <=  exmem_aluresult;
          memwb_rd <= exmem_rd;
          memwb_memtoreg <= exmem_memtoreg;
          memwb_regwrite <=  exmem_regwrite;
        end
    end

Mux m3(memwb_aluresult, memwb_readdataout, memwb_memtoreg, mem_writedata);
```

Fig 2.2.2b: Risc_Processor.v module

### 2.2.3 Code and Waveform

The link for the code can be found here.

```
// //Addi x1,x0,8
 Instruction_Memory[3] = 8'b00000000;
 Instruction_Memory[2] = 8'b10000000;
 Instruction_Memory[1] = 8'b00000000;
 Instruction_Memory[0] = 8'b10010011;

 //Addi x4,x0, x1
 Instruction_Memory[7] = 8'b00000000;
 Instruction_Memory[6] = 8'b00010000;
 Instruction_Memory[5] = 8'b00000010;
 Instruction_Memory[4] = 8'b00110011;

 //ld x6,0(x1) 0000B303
  Instruction_Memory[11] = 8'b00000000;
 Instruction_Memory[10] = 8'b00000000;
 Instruction_Memory[9] = 8'b10110011;
 Instruction_Memory[8] = 8'b00000011;
```

- For the **Addi x1, x0, 8** instruction, no forwarding is required. There are no data dependencies between this instruction and any previous instructions in the pipeline, as there is no previous instruction.

- For the **Addi x4, x0, x1** instruction, forwarding is needed from the EX/MEM stage to the ID/EX stage for the first source operand (x1). The result of the previous instruction (**Addi x1, x0, 8**) is written back to register **x1** in the EX/MEM stage. Since the next instruction **Addi x4, x0, x1** instruction needs the updated value of **x1** as its source operand, forwarding is necessary from the EX/MEM stage to the ID/EX stage for the first source operand (**x1**). This ensures that the instruction has the correct value of **x1** available for execution without any stalls in the pipeline. For this forwarding, the value of forwardB is 2.

- For the **ld x6, 0(x1)** instruction, forwarding is needed from the MEM/WB stage to the ID/EX stage for the second source operand (x1). Since the previous instruction (**Addi x4, x0, x1**) updates **x1** and writes back its result to the MEM/WB stage, forwarding is necessary from the MEM/WB stage to the ID/EX stage for the second source operand (**x1**). This ensures that the instruction has the updated value of **x1** available for execution without any pipeline stalls. For this forwarding, the value of forwardA is 1.

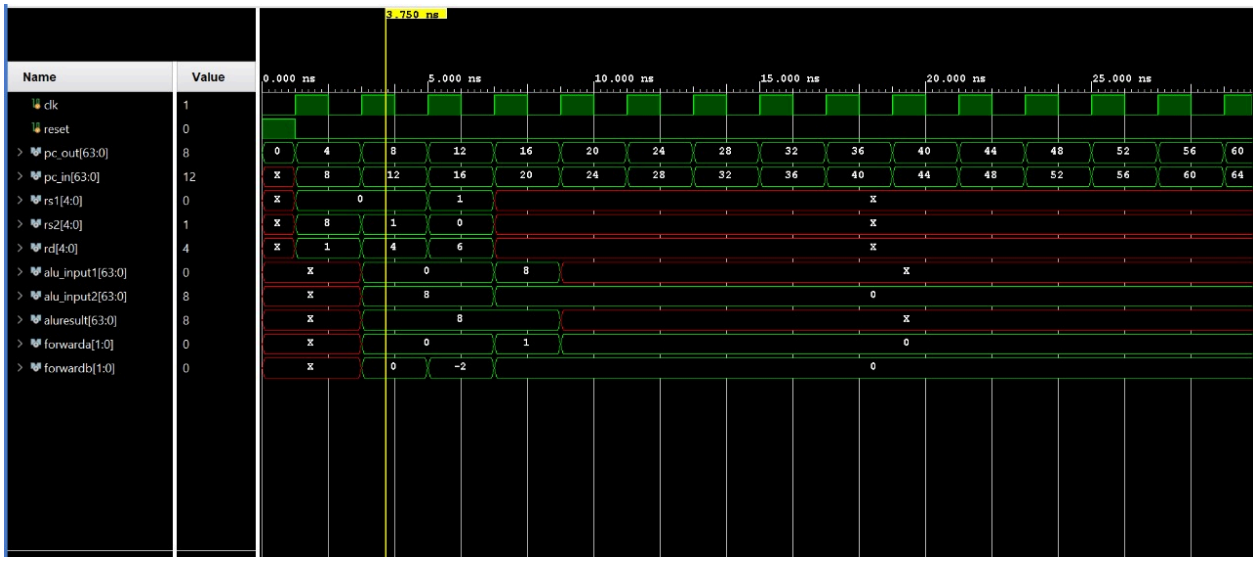



Fig 2.2.3: Test cases

## 2.3 Task 3

### 2.3.1 Handling Data Hazards

For data hazards that don't involve load (ld) instructions, the hazard detection unit is unnecessary as there's no need to stall the pipeline. However, when data hazards include ld instructions, the hazard detection unit is sometimes required as stalling may be necessary.

If there's another instruction between the ld instruction and the one causing the data hazard, forwarding can resolve the issue without needing the hazard detection unit. However, when the ld instruction and the one causing the hazard are consecutive, stalling becomes necessary and we have to use the hazard detection unit.

### 2.3.2 Changes

In computer architecture, hazards refer to conflicts that arise when attempting to execute instructions in a pipelined processor. These conflicts can occur due to dependencies between instructions, leading to stalls or incorrect results.

To address this issue, we added a Hazard Detection Unit module to our processor. The hazard detection module detects the data hazard which occurs due to being dependent on memory read instructions that attempt to read from a register that is about to be written by a previous instruction in the pipeline. If the hazard is detected the output is set to 0 which creates a stall.

### 2.3.3 Code and Waveform

The link for the code can be found here.

```
module hazard_detection_unit
  (
    input memread,
    input [31:0] instruction,
    input [4:0] rd,
    output reg stall
  );

  initial
    begin
      stall = 1'b0;
    end

  always @(*)
    begin
      if (memread == 1'b1 && ((rd == instruction[19:15]) || (rd == instruction[24:20])))
        stall = 1'b1;
      else
        stall = 1'b0;
    end
endmodule
```

Fig 2.3.3: Hazard Detection Module

## 2.4 Task 4

### 2.4.1 Comparison

Investigating the performance of bubble sort on a pipelined processor has revealed unexpected results. Conventionally, pipelined architectures are renowned for their efficiency, often outpacing their non-pipelined counterparts. However, our analysis has uncovered a discrepancy: our pipelined processor exhibits a longer elapsed time of 700 ns compared to the non-pipelined processor's 500 ns.

Despite successfully implementing pipeline techniques, our system faces efficiency challenges, notably coming from the absence of dynamic branch prediction. When the processor encounters a branch instruction in the memory stage, it necessitates flushing out preceding instructions. Consequently, fetching additional instructions takes an extra 200 ns in the case of branch instructions.

This revelation underscores the critical role of dynamic branch prediction in optimizing pipelined processor performance, prompting us to explore avenues for its integration to mitigate these inefficiencies.

## Conclusion and Challenges

We successfully executed the bubble sort algorithm on both the single-cycle processor and the RISC-V pipelined processor. Our results revealed that the single-cycle processor outperforms the pipelined processor in bubble sort. This discrepancy arises due to data dependencies inherent in the bubble sort algorithm, causing stalls in our processor.

Additionally, we encountered difficulties while implementing the IF/ID, ID/EX, EX/MEM, and MEM/WB modules. These modules were intended to pass values to the next cycle, but this functionality was not functioning as expected. Consequently, we improvised and integrated the functions of these modules directly into the Risc_Processor.V module.

Furthermore, testing the code after each implementation proved challenging as Vivado frequently froze. We had to restart the program repeatedly, leading to significant time consumption in the testing process.

## Task Division

- Task 1 → Fakeha Faisal
- Task 2 → Mariam Zeeshan
- Task 3 → Syeda Yashfeen Zehra Zaidi
- Task 4 → Syeda Yashfeen Zehra Zaidi
- Project Report → All group members

## References

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

[2] Encoding Simulator

[3] Venus

# Appendix

The link to our code can be found here.