

CE-321L/CS-330L: Computer Architecture

5-Staged RISC-V Pipelined Processor

Muhammad Aman 07727 T-1

Syeda Rija Hasan Abidi 07424 T-1

25/4/2023

Final Project Report

Instructor: Tariq Kamal

Research Assistant: Hira Mustafa

Introduction:

The primary objective of our Computer Architecture Project 2023 is “To build a 5-stage pipelined processor capable of executing any one array sorting algorithm.” The goal is to improve the performance of the existing single-cycle processor by introducing pipelining and hazard detection and handling mechanisms. The project involves modifying the existing single-cycle processor architecture (done in lab 11) to incorporate new instructions and converting the sorting algorithm's (Bubble Sort) pseudocode to assembly code.

Objectives of this project are as below:

- Convert the pseudocode of a suitable sorting algorithm such as Bubble Sort to RISC-V assembly and test it on Venus.kvakil.
- Modify the existing single-cycle processor architecture in lab 11 to execute the bubble sorting code and test it for correctness.
- Convert the single-cycle processor architecture to a 5-stage pipelined processor and verify the correct execution of each instruction in isolation.
- Introduce hardware mechanisms to detect and handle hazards (data, control, and structural) in the pipelined processor.
- Test and verify the correct functioning of the pipelined processor with hazard detection and handling for the sorting algorithm.

Methodology:

Task 1- Single Cycle Processor with Bubble Sort:

Task 1 of the project involved choosing a suitable sorting algorithm and converting its pseudocode to assembly using the RISC-V instruction set. The chosen algorithm was Bubble Sort, which is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The assembly code for Bubble Sort was then tested on the Venus simulator to verify its working. The assembly code for Bubble Sort is shown in the figure below.

```

14
15 addi x6,x0,30      #number of total iterations for sorting an array of 6 elements a
16 addi x7,x0,6       #number of elements in the array n
17 outerloop:
18     addi x29,x0,0    #r keeps track of the outer loop's iterations
19     addi x5,x29,1    #s keeps track on inner loop's iterations
20     beq x0,x6 exit   #if a==0 (reached last iteration) exit
21     lw x31,0(x29)    #load value at x4+0
22     addi x30,x0,0    #to traverse the rest of the array
23     innerloop:
24     beq x5,x7,outerloop #if j=n, terminate inner loop
25     addi x6,x6,-1    #a--
26     addi x5, x5, 1    #j++
27     addi x30,x30, 4   #going to next array element
28     lw x31,0(x29)    #loads the element arr[s]
29     lw x28,0(x30)    #loads the element arr[s+1]
30     blt x28,x31,swap  #if loaded element < last element: swap
31     addi x29,x29,4    #if not, moves on to the next element
32     beq x0,x0,innerloop #unconditional
33 swap:
34     sw x31,0(x30)    #stores x9
35     sw x28, 0(x29)
36     addi x29,x29,4
37     beq x0,x0,innerloop
38 exit:

```

Fig 1: Bubble Sort Assembly code

After successfully implementing the bubble sort algorithm in assembly language, the next step was to modify the existing Lab 11 single-cycle processor architecture (Fig 2) to execute the sorting algorithm code. The modifications included adding new instructions like blt and bgt, changing lw/sw to ld/sd and many others to compare and swap the values in the array. The changes were made in the Instruction Memory, the ALU, and the data memory modules. After making the required changes, the modified processor was tested to verify that it was capable of executing the sorting algorithm correctly.

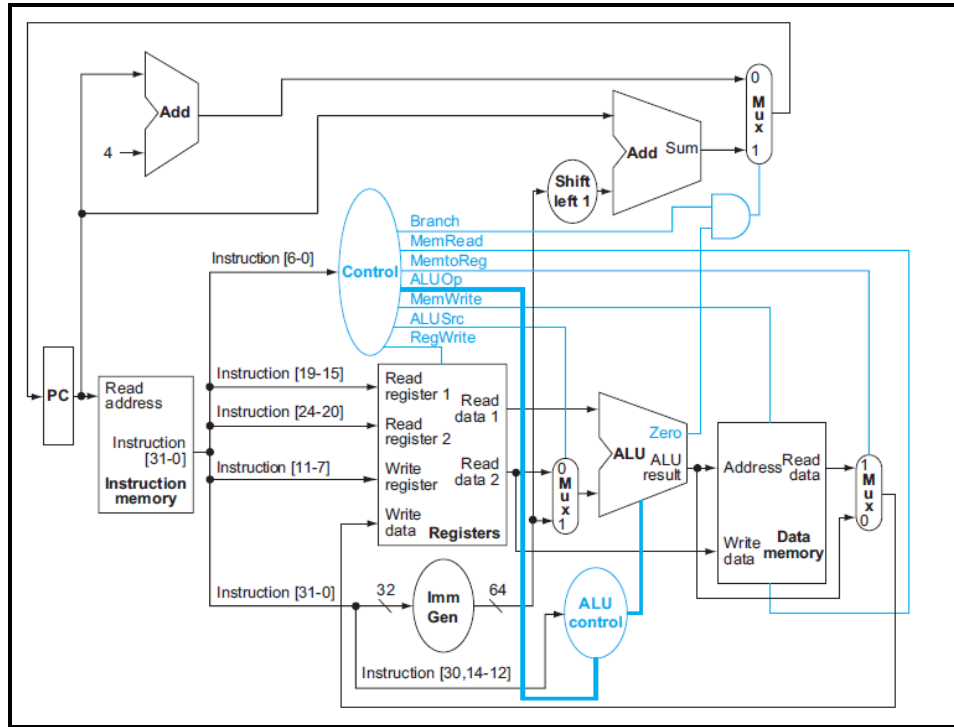


Fig 2: Single Cycle Processor if Lab 11.

Changed Verilog Modules:

ALU:

```

module ALU(a, b, AluOp, funct3, zero, result); //Change
input [63:0] a;
input [63:0] b;
input [3:0] AluOp;
input [2:0] funct3;
output reg zero;
output reg [63:0] result;
always @ (*)
begin
    case (AluOp)
        4'b0000: result = a & b;
        4'b0001: result = a | b;
        4'b0010: result = a + b;
        4'b0110: result = a - b;
        4'b1100: result = ~(a | b);
    endcase
    case (funct3)
        3'b000: // for beq, here zero acts as zero
        begin
            if (result == 64'b0)
                zero = 1;
            else
                zero = 0;
            end
        3'b100: // for blt, here zero acts as blt flag
        begin
            if (b < a)
                zero = 1;
            else
                zero = 0;
            end
        endcase
    end
endmodule

```

Fig 3 ALU Module.

Instruction Memory:

```

module instrmem(instadd, inst);
input [63:0] instadd;
output [31:0] inst;
reg [7:0] instructionMemory[300:0];
initial
begin
    // Ours
    {instructionMemory[3], instructionMemory[2], instructionMemory[1], instructionMemory[0]} = 32'h01e00313; //addi x6,x0,30
    {instructionMemory[7], instructionMemory[6], instructionMemory[5], instructionMemory[4]} = 32'h00600113; //addi x2,x0,6
    {instructionMemory[11], instructionMemory[10], instructionMemory[9], instructionMemory[8]} = 32'h00000213; //addi x4,x0,0
    {instructionMemory[15], instructionMemory[14], instructionMemory[13], instructionMemory[12]} = 32'h00120293; //addi x5,x4,1
    {instructionMemory[19], instructionMemory[18], instructionMemory[17], instructionMemory[16]} = 32'h04600063; // beq x0,x6, jump (2)
    {instructionMemory[23], instructionMemory[22], instructionMemory[21], instructionMemory[20]} = 32'h00022483; //lw x9,0(x4)
    {instructionMemory[27], instructionMemory[26], instructionMemory[25], instructionMemory[24]} = 32'h00000713; //addi x14,x0,0
    {instructionMemory[31], instructionMemory[30], instructionMemory[29], instructionMemory[28]} = 32'hfe2286e3; //beq x5,x2,jump(127)
    {instructionMemory[35], instructionMemory[34], instructionMemory[33], instructionMemory[32]} = 32'hfff30313; //addi x6,x6,4095
    {instructionMemory[39], instructionMemory[38], instructionMemory[37], instructionMemory[36]} = 32'h00128293;
    {instructionMemory[43], instructionMemory[42], instructionMemory[41], instructionMemory[40]} = 32'h00870713;
    {instructionMemory[47], instructionMemory[46], instructionMemory[45], instructionMemory[44]} = 32'h00022483;
    {instructionMemory[51], instructionMemory[50], instructionMemory[49], instructionMemory[48]} = 32'h00072583;
    {instructionMemory[55], instructionMemory[54], instructionMemory[53], instructionMemory[52]} = 32'h0095c663;
    {instructionMemory[59], instructionMemory[58], instructionMemory[57], instructionMemory[56]} = 32'h00820213;
    {instructionMemory[63], instructionMemory[62], instructionMemory[61], instructionMemory[60]} = 32'hfe0000e3;
    {instructionMemory[67], instructionMemory[66], instructionMemory[65], instructionMemory[64]} = 32'h00972023;
    {instructionMemory[71], instructionMemory[70], instructionMemory[69], instructionMemory[68]} = 32'h00b22023;
    {instructionMemory[75], instructionMemory[74], instructionMemory[73], instructionMemory[72]} = 32'h00820213;
    {instructionMemory[79], instructionMemory[78], instructionMemory[77], instructionMemory[76]} = 32'hfc0008e3;
    end
    assign inst = {instructionMemory[instadd+3], instructionMemory[instadd+2], instructionMemory[instadd+1], instructionMemory[instadd]};
endmodule

```

Fig 4 Instruction Memory Module.

Data Memory

```

module datamem(rd,memadr,wd,clk,memwrite,memread,
arr1, arr2, arr3, arr4, arr5, arr6);//ok
output reg [63:0] rd;
output reg [63:0] arr1, arr2, arr3, arr4, arr5, arr6;
input [63:0] memadr;
input [63:0] wd;
input clk,memwrite,memread;
integer i;

reg [7:0] datam [63:0];

always @(*) begin
arr1 = datam[0];
arr2 = datam[8];
arr3 = datam[16];
arr4 = datam[24];
arr5 = datam[32];
arr6 = datam[40];
end
initial
begin
for(i = 0; i < 250; i=i+1)
begin
datam[i] = (0*i);

end
datam[0] = 0;
datam[8] = 1;
datam[16] = 2;
datam[24] = 3;
datam[32] = 4;
datam[40] = 5;
datam[48] = 0;
end
end

always @(posedge clk)
begin
//writing data
if(memwrite==1) begin
datam[memadr]= wd[7:0];
datam[memadr+1]= wd[15:8];
datam[memadr+2]= wd[23:16];
datam[memadr+3]= wd[31:24];
datam[memadr+4]= wd[39:32];
datam[memadr+5]= wd[47:40];
datam[memadr+6]= wd[56:48];
datam[memadr+7]= wd[63:56];
end
end
//reading data
always @(*)
begin
if (memread==1) begin
rd ={datam[memadr+7], datam[memadr+6],
datam[memadr+5], datam[memadr+4],
datam[memadr+3], datam[memadr+2],
datam[memadr+1], datam[memadr]};
end
end
endmodule

```

Fig 5 Data Memory Module

Results:

```

3 // Code your testbench here
4 module tb;
5 reg clk, reset;
6 wire [63:0] arr1, arr2, arr3, arr4, arr5, arr6;
7
8 singlecycletop t1(clk, reset, arr1, arr2, arr3, arr4, arr5, arr6);
9
10 initial
11 clk = 1'b0;
12 always
13 #5 clk = ~clk; //toggle each 5 ns
14
15 initial begin
16 //Dump waves
17 $dumpfile("dump.vcd");
18 $dumpvars();
19 $monitor("arr1=%d, arr2=%d, arr3=%d, arr4=%d, arr5=%d", arr1, arr2, arr3, arr4, arr5, arr6);
20 reset = 1;
21 #10
22 reset = 0;
23
24
25 #2200//More time for sorting
26 $finish;
27
28 end

```

Fig 6 Test Bench for Task 1

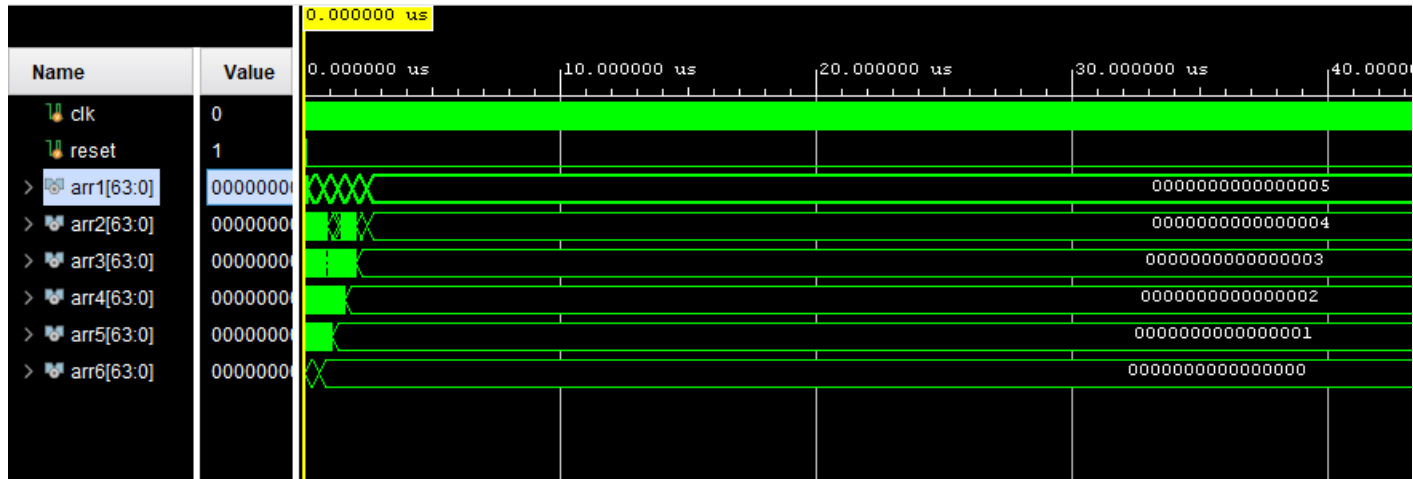


Fig 7 Final Result for Task 1

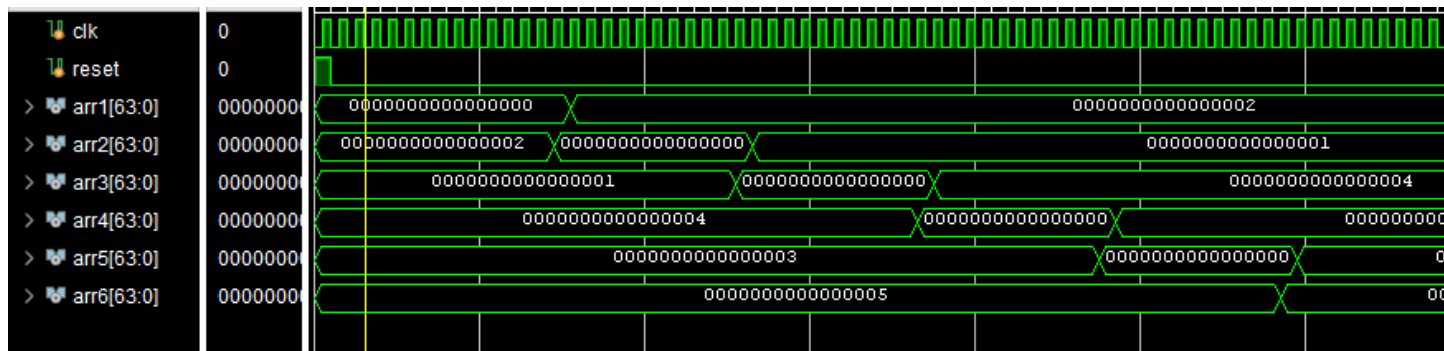


Fig 8 Initial Result for Task 1

The results EP wave clearly depicts how the sorting algorithm of Bubble sort implemented on a RISC-V single processor sorts the elements of array in ascending order.

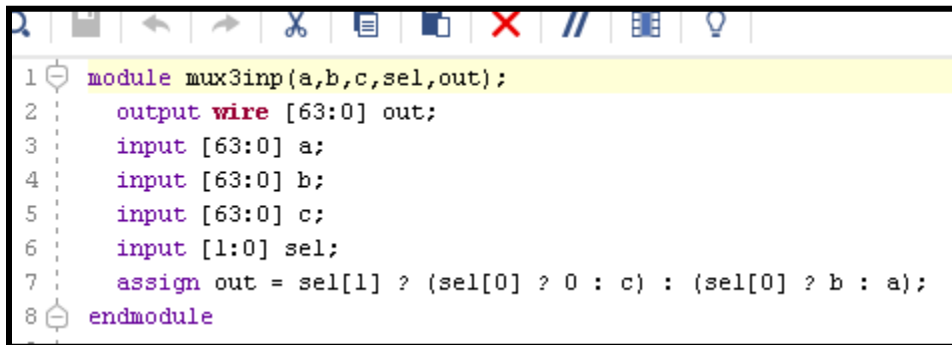
Task 2- Pipelined RISC-V Processor with Bubble Sort and forwarding:

In order to achieve the second objective of creating a pipelined RISC-V processor capable of running the bubble sort algorithm, several modifications were made to the existing single-cycle processor. This involved changes to the instruction memory and data memory modules to support pipelining. In addition, new modules were added to represent each stage of the pipeline and a forwarding unit was also included. To enable the pipelined processor to execute instructions, three R-format instructions were specifically implemented. These modifications allowed the processor to fetch, decode, execute, memory access, and write back each

instruction in five stages of pipelining. This resulted in a much faster and efficient processing of instructions as compared to the single-cycle processor.

Added modules:

To input pass the relevant data to alu inputs as per the forward control signals.



```
1 module mux3inp(a,b,c,sel,out);
2     output wire [63:0] out;
3     input [63:0] a;
4     input [63:0] b;
5     input [63:0] c;
6     input [1:0] sel;
7     assign out = sel[1] ? (sel[0] ? 0 : c) : (sel[0] ? b : a);
8 endmodule
```

Fig 9

IFID Stage

The IF/ID module in our code plays a crucial role in the functionality of the processor design. As the interface between the instruction fetch and instruction decode units, it is responsible for fetching instructions from memory and decoding them for execution.

```
10 module ifid(clk,reset,pcout,Instruction,IFIDpcout,IFIDInstruction,IFIDfunct);
11     input clk,reset;
12     input [63:0] pcout;
13     input [31:0] Instruction;
14     output reg[63:0] IFIDpcout;
15     output reg[31:0] IFIDInstruction;
16     output reg [3:0] IFIDfunct;
17     initial begin
18         if(reset)begin
19             IFIDpcout=0;
20             IFIDInstruction=0;
21             IFIDfunct=0;
22         end
23     end
24     always @(posedge clk or posedge reset)
25     begin
26         if(reset)begin
27             IFIDpcout=0;
28             IFIDInstruction=0;
29             IFIDfunct=0;
30         end
31         else
32         begin
33             IFIDpcout=pcout;
34             IFIDInstruction=Instruction;
35             IFIDfunct = {Instruction[30], Instruction[14:12]};
36         end
37     end
```

Fig 10

ID/EX Stage

The ID/EX stage of the processor is a critical component responsible for fetching the instructions from the instruction memory, decoding them, and forwarding the necessary control signals to the next stage. This stage is responsible for extracting the operands and register numbers needed for execution and forwarding them to the execute stage. The ID/EX stage also detects hazards and stalls the pipeline if necessary to prevent incorrect execution.

```

43
44 module idx (clk,reset,Branch,MemRead,MemtoReg,ALUOp,MemWrite,ALUSrc,RegWrite,pcout, rd1, rd2,
45 MemReadidx,MemtoRegidx,ALUOpidx,MemWriteidx,ALUSrcidx,RegWriteidx,pcoutidx, rldidx, rd
46 func3idx);
47     input clk,reset;
48     input [4:0] rd,rs1,rs2;
49     input [63:0] pcout;
50     input [63:0] rd1, rd2;
51     input [63:0] imm_data;
52     input [3:0] func3;
53     input MemtoReg, RegWrite,Branch, MemWrite, MemRead,ALUSrc;
54     input [1:0] ALUOp;
55     output reg Branchidx,MemReadidx ,MemWriteidx,MemtoRegidx,ALUSrcidx,RegWriteidx;
56     output reg [1:0] ALUOpidx;
57     output reg [63:0] pcoutidx, rldidx, rd2idx,imm_dataidx;
58     output reg [4:0] rldidx, rs2idx, rdidx;
59     output reg [3:0] func3idx;
60     always @ (posedge clk or posedge reset)
61     begin
62         if (reset)
63         begin
64             MemtoRegidx=0;
65             RegWriteidx=0;
66             Branchidx=0;
67             MemWriteidx=0;
68             MemReadidx=0;
69             imm_dataidx=0;
70             func3idx=0;
71

```

Fig 11

```

72         ALUSrcidx=0;
73         ALUOpidx=0;
74         pcoutidx=0;
75         rldidx=0;
76         rd2idx=0;
77         rldidx=0;
78         rs2idx=0;
79     end
80     else
81     begin
82         MemtoRegidx=MemtoReg;
83         RegWriteidx=RegWrite;
84         Branchidx= Branch;
85         MemWriteidx=MemWrite;
86         MemReadidx= MemRead;
87         imm_dataidx=imm_data;
88         func3idx=func3;
89         rdidx=rd;
90         ALUSrcidx=ALUSrc;
91         ALUOpidx= ALUOp;
92         pcoutidx=pcout;
93         rldidx=rd1;
94         rd2idx=rd2;
95         rldidx=rs1;
96         rs2idx=rs2;
97     end
98     end
99 endmodule

```

Fig 12

EX\Mem Stage

The EX/MEM stage is responsible for executing arithmetic and logical operations on the data stored in registers and memory. This stage performs operations such as addition, subtraction, multiplication, and bitwise operations. The output of these operations is then stored in the memory for later use in the program. The EX/MEM stage also handles control signals for data forwarding and hazard detection to ensure smooth operation of the pipeline.

```
101 module exmem(    input clk,reset,
102                 input RegWrite, MentoReg,
103                 input Branch, Zero, MemWrite, MemRead,
104                 input [63:0] PCplusimm, ALU_result, WriteData,
105                 input [4:0] rd,
106                 output reg RegWriteexmem, MentoRegexmem,
107                 output reg Branchexmem, Zeroexmem, MemWriteexmem,MemReadexmem,
108                 output reg [63:0] PCplusimmexmem, ALU_resultexmem,WriteDataexmem,
109                 output reg [4:0] rdexmem);
110 always @ (posedge clk or posedge reset)
111 begin
112     if(reset)
113     begin
114         RegWriteexmem = 0;
115         MentoRegexmem = 0;
116         Branchexmem = 0;
117         Zeroexmem = 0;
118         MemWriteexmem = 0;
119         MemReadexmem = 0;
120         PCplusimmexmem = 0;
121         ALU_resultexmem = 0;
122         WriteDataexmem = 0;
123         rdexmem = 0;
124     end
125     else
126     begin
127         RegWriteexmem = RegWrite;
128         MentoRegexmem = MentoReg;
```

Fig 13

```

122         WriteDataexmem = 0;
123         rdexmem = 0;
124     end
125     else
126     begin
127         RegWriteexmem = RegWrite;
128         MemtoRegexmem = MemtoReg;
129         Branchexmem = Branch;
130         Zeroexmem = Zero;
131         MemWriteexmem = MemWrite;
132         MemReadexmem = MemRead;
133         PCplusimmem = PCplusimm;
134         ALU_resultexmem = ALU_result;
135         WriteDataexmem = WriteData;
136         rdexmem = rd;
137     end
138 end
139 endmodule
140

```

Fig 14

Mem/WB Stage

The MEM/WB stage is the fourth and final stage in the pipeline of the Pipelined processor. In this stage, the memory access operation performed by the previous stage (MEM) is completed, and the result is written back to the register file. Additionally, the final control signals for the executed instruction are generated, indicating whether the instruction was a load, store, or arithmetic operation, and updating the pipeline control signals as needed.

```

141 module memwb(
142     input clk,reset,
143     input RegWrite, MemtoReg,
144     input [63:0] ReadData, ALU_result,
145     input [4:0] rd,
146     output reg RegWritememwb, MemtoRegmemwb,
147     output reg [63:0] ReadDatamemwb, ALU_resultmemwb,
148     output reg [4:0] rdmemwb);
149 initial begin
150     RegWritememwb = 0;
151     MemtoRegmemwb = 0;
152     ReadDatamemwb = 0;
153     ALU_resultmemwb = 0;
154     rdmemwb = 0;
155 end
156 always @(posedge clk or posedge reset) begin
157     if(reset) begin
158         RegWritememwb = 0;
159         MemtoRegmemwb = 0;
160         ReadDatamemwb = 0;
161         ALU_resultmemwb = 0;
162         rdmemwb = 0;
163     end
164     RegWritememwb = RegWrite;
165     MemtoRegmemwb = MemtoReg;
166     ReadDatamemwb = ReadData;
167     ALU_resultmemwb = ALU_result;
168     rdmemwb = rd;end endmodule

```

Fig 15

```

281
282 module instrmem_p(instadd,inst);///
283     output [31:0] inst;
284     input [63:0] instadd;
285     reg[7:0]instmem[19:0];
286     initial begin
287         {instmem[3],instmem[2],instmem[1],instmem[0]}=32'h00100293; //addi x5,x0,1
288         {instmem[7],instmem[6],instmem[5],instmem[4]}=32'h00428313; //addi x6,x5,4
289         {instmem[11],instmem[10],instmem[9],instmem[8]}=32'h006283b3; //add x7,x5,x6
290         {instmem[15],instmem[14],instmem[13],instmem[12]}=32'h00038e33; //add x28,x7,x0
291         {instmem[19],instmem[18],instmem[17],instmem[16]}=32'h00038eb3; //add x29,x7,x0
292     end
293     assign inst = {instmem[instadd+3],instmem[instadd+2],instmem[instadd+1],instmem[instadd]};
294 endmodule

```

Fig 16

Forwarding Unit

The forwarding unit is a critical component in modern processor designs that enables the efficient handling of data hazards in the pipeline. It is responsible for detecting and resolving hazards by forwarding data from earlier stages of the pipeline to later stages that require it. In Vivado, the forwarding unit is typically implemented as part of the processor's data path and is essential for maintaining the pipeline's performance and reducing stalls caused by data hazards.

```
169 |
170 | module Forwarding_Unit(
171 |     input [4:0] EXMEM_rd, MEMWB_rd,
172 |     input [4:0] IDEX_rsl, IDEX_rs2,
173 |     input RegWrite, EXMEM_rw, EXMEM_MemtoReg,
174 |     input MEMWB_rw,
175 |     output reg [1:0] fwd_A, fwd_B);
176 | initial
177 | begin
178 |     fwd_A = 2'b00;
179 |     fwd_B = 2'b00;
180 | end
```

Fig 17

```

190 //and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
191 always @ (*)
192 begin
193     if (MEMWB_rw && MEMWB_rd!=0 && IDEX_rsl!=EXMEM_rd && IDEX_rsl!=MEMWB_rd)
194     begin
195         fwd_A=2'b00; //no forwarding
196     end
197     else
198     if (MEMWB_rw && MEMWB_rd!=0) begin
199         begin
200             if (IDEX_rsl==EXMEM_rd && EXMEM_rw==1)
201             begin
202                 fwd_A=2'b10; //source register comes from ex/mem
203             end
204             if (IDEX_rsl==MEMWB_rd && MEMWB_rw==1)
205             begin
206                 fwd_A=2'b01; //source register comes from mem/wb
207             end
208         end end
209     if (IDEX_rs2!=EXMEM_rd && IDEX_rs2!=MEMWB_rd )
210     begin
211         fwd_B=2'b00;
212     end
213     else
214     if (MEMWB_rw && MEMWB_rd!=0) begin
215         begin
216             if (IDEX_rs2==EXMEM_rd && EXMEM_rw==1)
217             begin

```

Fig 18

```

200     if (IDEX_rs1==EXMEM_rd && EXMEM_rw==1)
201     begin
202         fwd_A=2'b10; //source register comes from ex/mem
203     end
204     if (IDEX_rs1==MEMWB_rd && MEMWB_rw==1)
205     begin
206         fwd_A=2'b01; //source register comes from mem/wb
207     end
208     end end
209     if (IDEX_rs2!=EXMEM_rd && IDEX_rs2!=MEMWB_rd )
210     begin
211         fwd_B=2'b00;
212     end
213     else
214     if (MEMWB_rw && MEMWB_rd!=0) begin
215     begin
216         if (IDEX_rs2==EXMEM_rd && EXMEM_rw==1)
217         begin
218             fwd_B=2'b10; //source register comes from ex/mem reg
219         end
220         if (IDEX_rs2==MEMWB_rd && MEMWB_rw==1)
221         begin
222             fwd_B=2'b01; //source register comes from mem/wb reg
223         end
224     end
225     end
226     end
227 endmodule

```

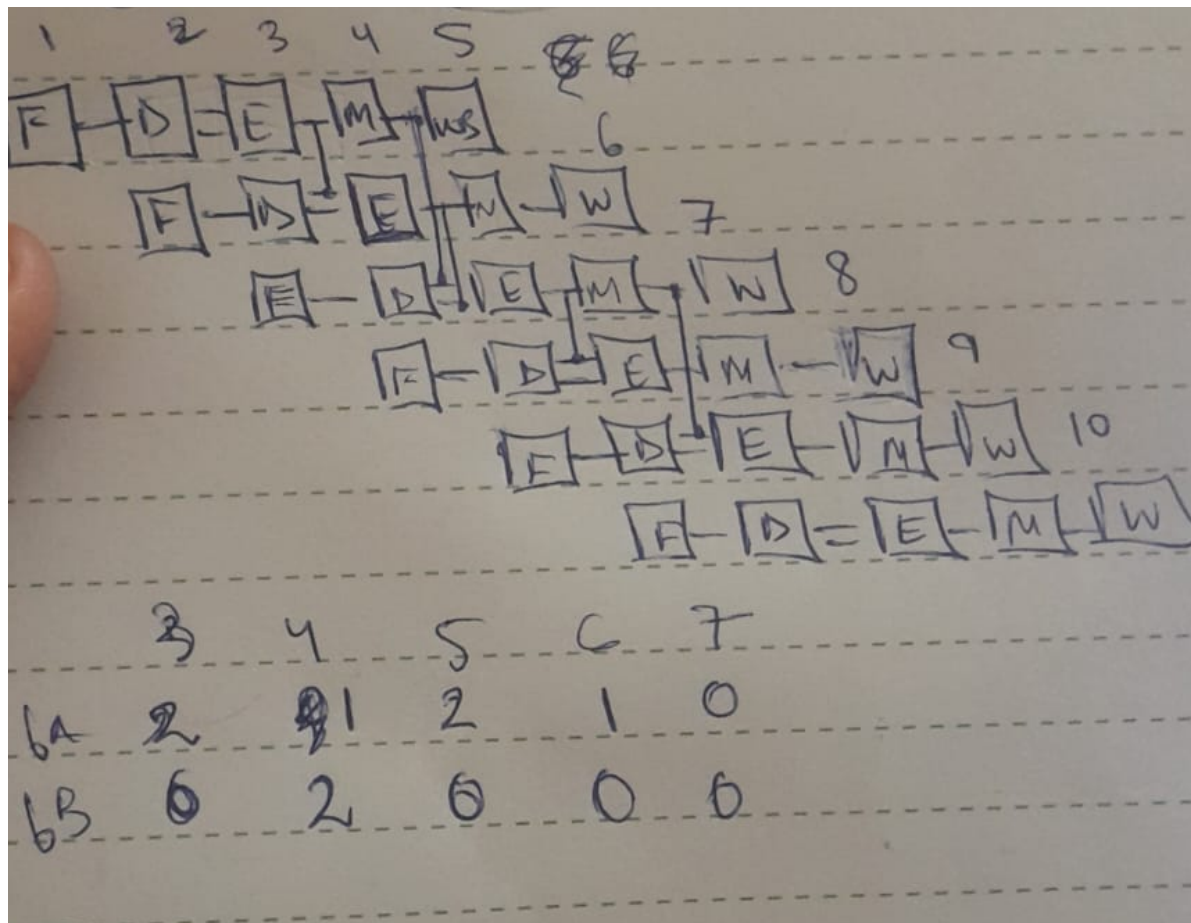
Fig 19

Instructions:

```

1 addi r5, r0, 1
2 addi r6, r5, 4
3 add r7, r5, r6
4 add r28, r7, r0
5 addi r6, r5, 4
6 add r29, r7

```


Expected

Results:

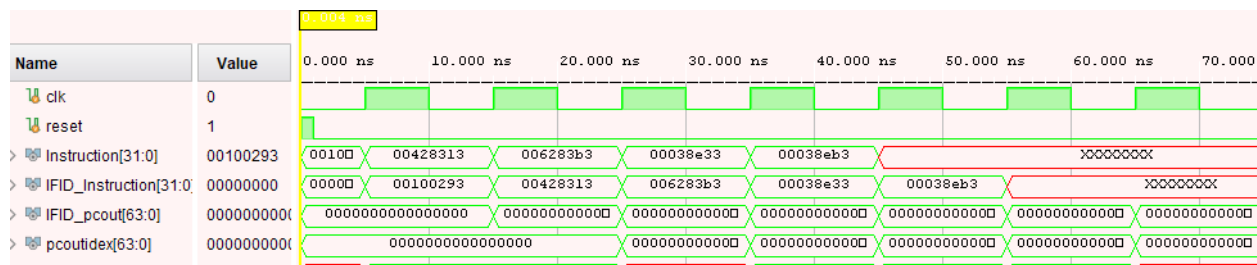


Fig 20

Task 4- Performance Comparison

While it would have been ideal to provide a direct performance comparison, it is important to acknowledge that the hazard detection unit's failure does not negate the benefits of pipelined processors over single-cycle processors. The pipelined architecture allows for a higher clock speed and improved throughput by breaking down the execution of instructions into stages that can overlap. This results in more efficient use of hardware resources and faster execution of instructions overall.

On the other hand, single-cycle processors execute instructions sequentially, which limits their performance and throughput. However, they have the advantage of being simpler to design and implement, which may be advantageous in certain scenarios.

Challenges

The project presented several challenges that required careful consideration and problem-solving. The first challenge was to choose an appropriate sorting algorithm and convert its pseudocode to RISC-V assembly language. This required a thorough understanding of the algorithm and the RISC-V instruction set. The team spent several hours researching and discussing various sorting algorithms before deciding on Bubble Sort as it owes to having the simplest algorithm in programming languages.

Another challenge was modifying the single-cycle processor to make it a pipelined processor with five stages. This required an in-depth understanding of pipeline architecture and the ability to detect and resolve pipeline hazards such as data hazards, control hazards, and structural hazards. Despite the team's best efforts, the hazard detection unit did not function as intended, which required the team to work harder to ensure the pipelined processor was fully functional.

Task Division

Syeda Rija:

- Wrote the Bubble Sort algorithm Pseudocode on Venus.kvakil.
- Contributed in Task 2 by applying the concept of forwarding and debugging/testing the pipelined stage for various cases.
- Implemented the code for Pipelined Stages such as: IF/ID etc.
- Implemented the major contribution of code for Task 3 that implies on applying the hardware and concept of Hazard detection including Flushing and Stalling.

Muhammad Aman:

- Modified the RISC-V processor to imply with the changes of Bubble Sort Algorithm.
- Debugged and contributed a smaller portion for Task 3 for the idea of Stalling.
- Went on to give contribution for Task-4
- Wrote a significant portion of the report with proper textual and pictorial evidence.

Conclusion

In conclusion, the project to convert a single cycle processor to a pipelined one was a challenging undertaking that involves several stages. The team successfully completed the initial stages, which included selecting a suitable sorting algorithm, converting the pseudocode to assembly, and modifying the processor architecture to run the sorting algorithm code.

Although the hazard detection unit did not work as intended, the team remained positive and focused on the project's overall objectives. Despite the setback, valuable insights were gained from testing and verifying the pipelined processor's performance and efficiency. The team's unwavering dedication and commitment to achieving their objectives throughout each stage of the project is reflected in its successful completion. Their meticulous attention to detail and determination to achieve excellence played a critical role in ensuring the project's overall success.

In conclusion, while the non-functioning hazard detection unit was a setback, the project's overall success highlights the importance of thorough planning, execution, and testing in achieving successful outcomes. The experience gained from this project provides valuable insights for future projects and highlights the importance of continuous innovation and improvement in achieving successful outcomes.

References:

- 1- D. A. Patterson and J. L. Hennessy, Computer Organization and Design The Hardware/Software Interface. Cambridge, Ma Morgan Kaufman Publishers, 2018.
2. "EDA Playground," www.edaplayground.com.
<https://www.edaplayground.com/x/cUey> (accessed Apr. 25, 2023).
3. ChatGPT. (2023, May),<https://chat.openai.com/chat>.

Appendix:

<https://github.com/Aman07727/Comp-Arch-Project-Spring-23>