COMPUTER ARCHITECTURE

Risc V Single Cycle Processor & Risc V Pipelined Processor

By:
Syeda Alishba Zaidi 08339
Huzaifa Ahmed Khan 08437

CONTENTS

1. Introduction

For this project, we were tasked to build a Risc V Processor that is able to execute Bubble Sort Algorithm. Our Project included building a single cycle processor and then a pipelined version which is able execute the same task. Our aim was to improve the performance of the single-cycle processor by pipelining it and also implementing the Hazard detection and then comparing the 2 processors to find out which is better for bubble sort.

2. Methodology

We divided our project in 4 main tasks:

1. Firstly we chose the bubble sort algorithm, tested it on Kwakil Venus to see if it was working or not, Then we converted each line of code from Assembly to Binary and created our Instruction memory to be used for this project and then we modified the single cycle processor we created in lab 11 to run the sorting algorithm.
2. Then we made changes to our single cycle processor to create the pipelined processor and tested if it was able to execute each instruction separately.
3. We then modified our pipelined processor to be able to detect hazards and solve hazards by implementing the forwarding unit, so that it can run our bubble sort algorithm completely.
4. At last, We compared the performance of running the bubble sort on Single Cycle Processor and On Pipelined Processor in terms of execution time to find which one is better.

Bubble Sort Algorithm in Assembly

```
addi x10, x0, 0x100
addi x11, x0, 3
bne x10, x0, else
bne x11, x0, else
else: addi x18, x0, 0 # i
Loop1: beq x18, x11, exit1
    add x19, x0, x18 # j = i
    Loop2:
        beq x19, x11, exit2
        slli x5, x18, 2 # calculating offset of i
        slli x6, x19, 2 # calculating offset of j
        add x5, x5, x10
        add x6, x6, x10
        lw x28, 0(x5) # accessing value of a[i]
        lw x29, 0(x6) # accessing value of a[j]
        bge x28, x29, loop2_continued #if a[i] >= a[j]
        add x30, x0, x28 # temp = a[i]
        add x28, x0, x29 # a[i] = a[j]
        add x29, x0, x30 # a[j] = temp
        sw x28, 0(x5)
        sw x29, 0(x6)
        loop2_continued: addi x19, x19, 1 # j += 1
        beq x0, x0, Loop2
    exit2: addi x18, x18, 1 # i += 1
    beq x0, x0, Loop1
exit1:
```

Array Initialized

```
1 #initializing array of length 3
2 addi x5, x0, 8
3 sw x5, 0x100 (x0)
4 addi x5, x0, 96
5 sw x5, 0x104(x0)
6 addi x5, x0, 48
7 sw x5, 0x108(x0)
```

Array in memory before and after sorting

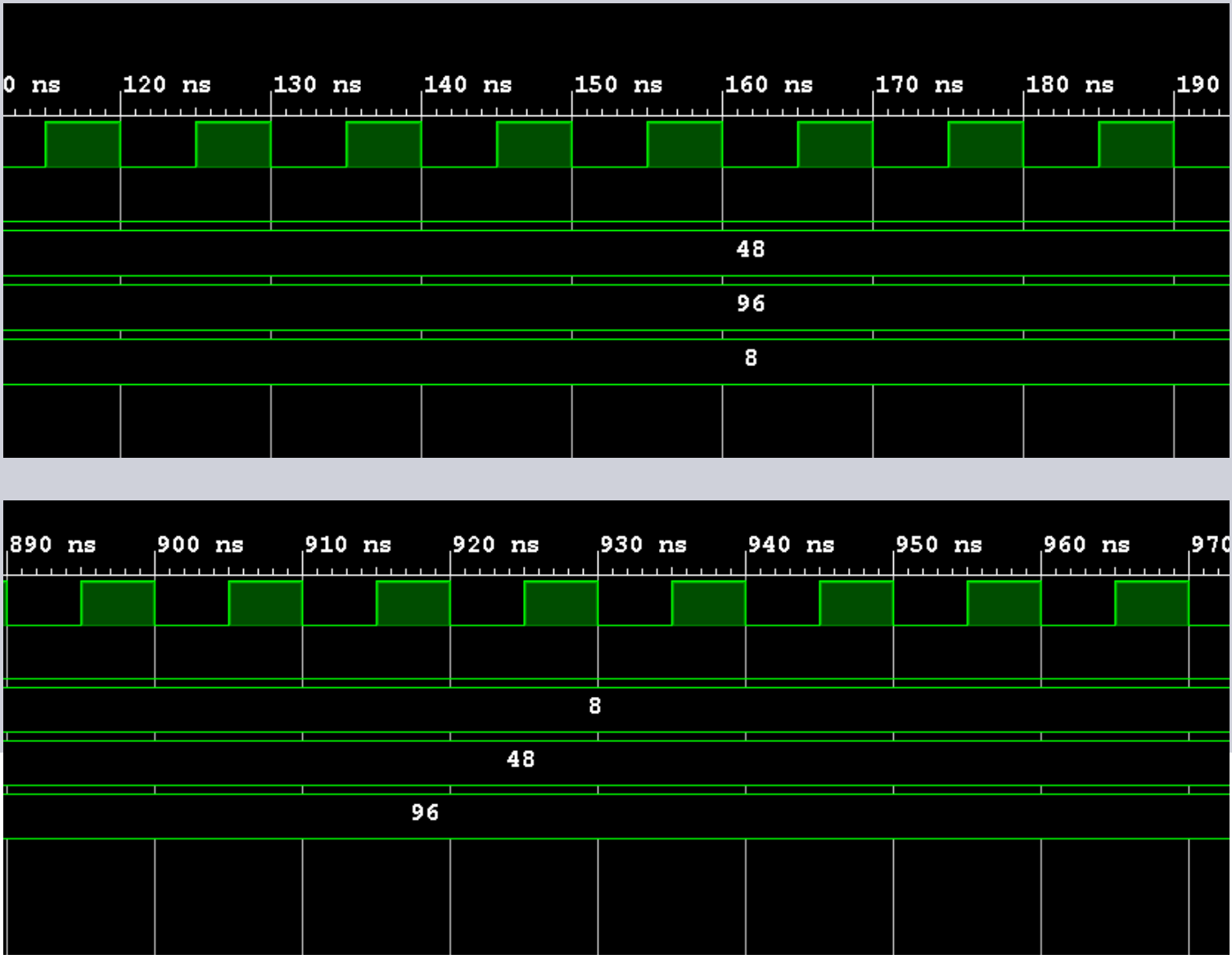
0x00000108	48	0	0	0
0x00000104	96	0	0	0
0x00000100	8	0	0	0

BEFORE

0x00000108	8	0	0	0
0x00000104	48	0	0	0
0x00000100	96	0	0	0

AFTER

TASK 1



SINGLE CYCLE PROCESSOR

We used the Venus simulator to find the machine code for all the instructions of bubble sort, But Venus does not support instructions such as store double word and load double word, so we had to modify bubble sort and separately generate machine code for all the ‘sd’ and ‘ld’ instructions. After which 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 runs correctly on the single cycle processor. As shown above, the sorting is being done correctly on the processor and the elements inside the data memory are sorted at the end.

BRANCH MODULE

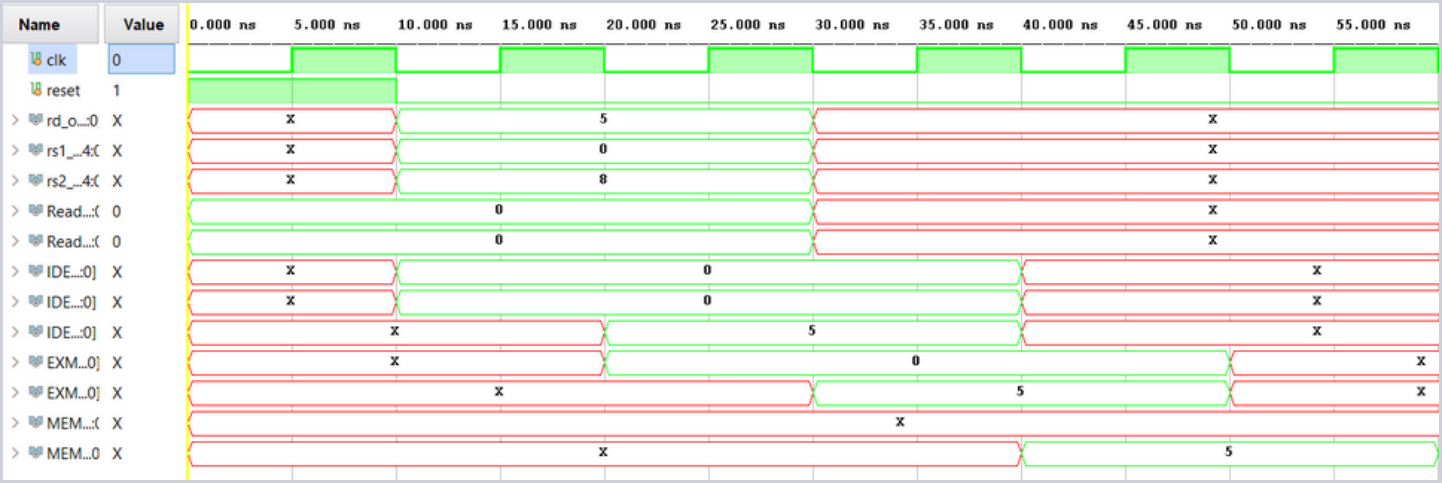
```
module Branch(
    input Branch,
    input ZERO,
    input Isgreater,
    input [3:0] funct,
    output reg switch_branch
    );

    always @(*) begin
        if(Branch) begin
            case({funct[2:0]})
                3'b000: switch_branch = ZERO ? 1:0;
                3'b001: switch_branch = ZERO ? 0:1;
                3'b101: switch_branch =Isgreater ? 1:0;
            endcase
          end
        else
           switch_branch=0;
     end
endmodule
```

TASK 2

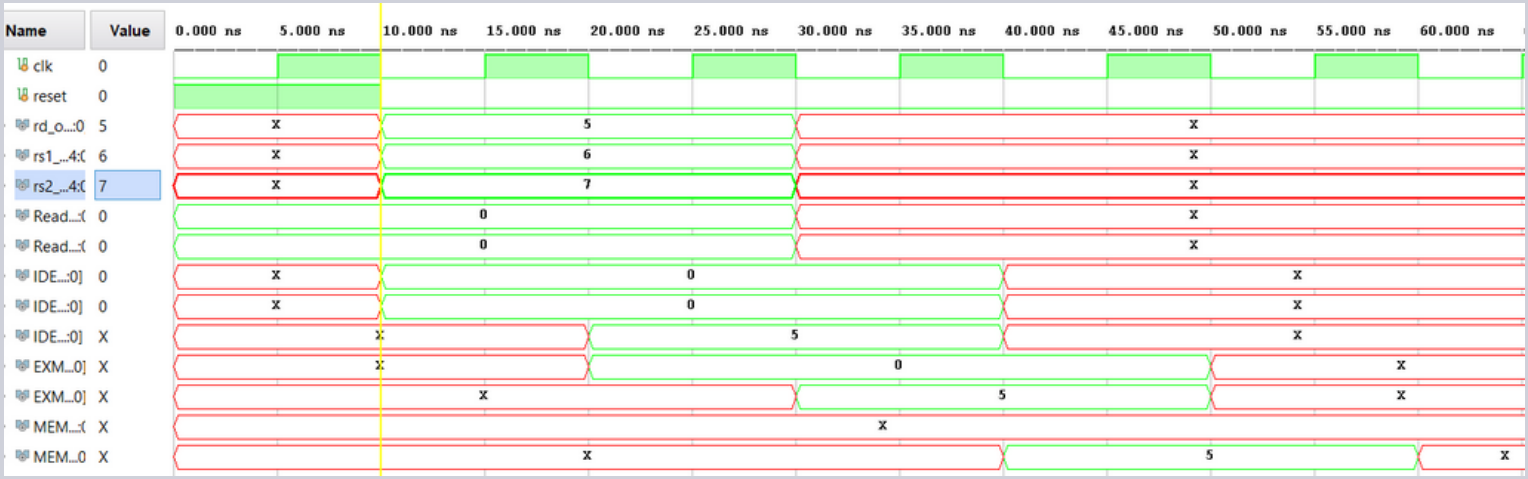
PIPELINED PROCESSOR

We modified the single cycle processor by adding modules for IFID, IDEX, EXMEM and MEMWB registers. These modules are basically the 5 stages of a pipelined processor and they store as well as forward the information to the next stage for executing the instruction properly. We tested 2 instructions in isolation and checked if the processor is working for one instruction or not.

TESTCASE 1: ADDI X5, X0, 8



TESTCASE 2: ADD X5, X6, X7



HAZARD DETECTION

For detecting hazards, we created the modules for hazard detection, forwarding unit and Mux3to1. 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. The forwarding module decides if the current instruction is dependent upon the previous instruction or not and then forward the data accordingly. The result from the forwarding unit is sent to the 2 mux's which then select the data and then send to the ALU.

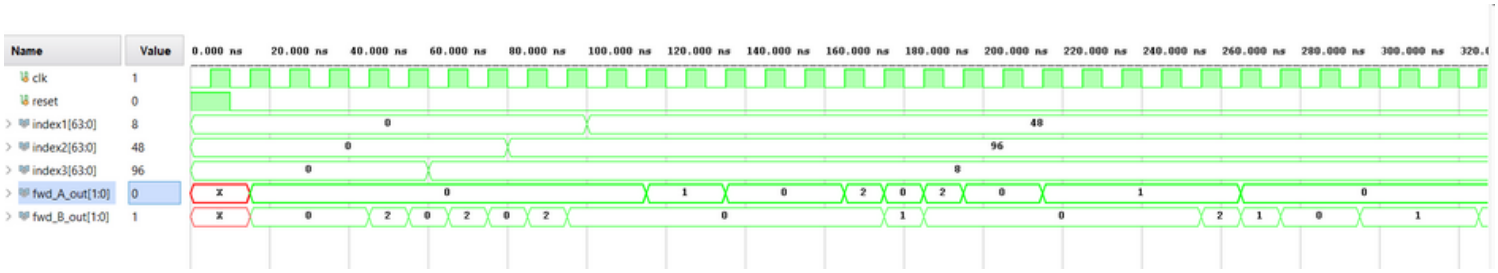
```
module Hazard_Detection
(
    input [4:0] IDEX_rd, IFID_rs1, IFID_rs2,
    input IDEX_MemRead,
    output reg IDEX_mux_out,
    output reg IFID_Write, PCWrite
);

always@(*) begin

    if (IDEX_MemRead && (IDEX_rd == IFID_rs1 || IDEX_rd == IFID_rs2))
    begin
        IDEX_mux_out = 0;
        IFID_Write = 0;
        PCWrite = 0;
    end
    else begin
        IDEX_mux_out = 1;
        IFID_Write = 1;
        PCWrite = 1;
    end
end
endmodule // Hazard_Detection
```

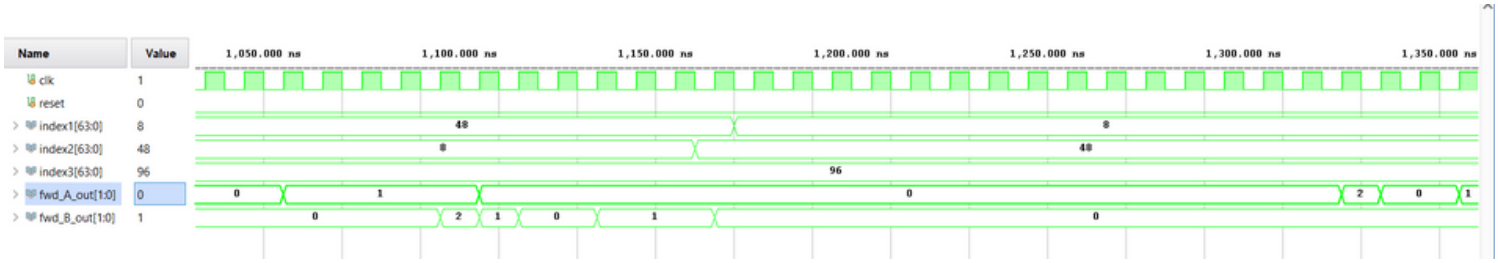
FORWARDING MODULE

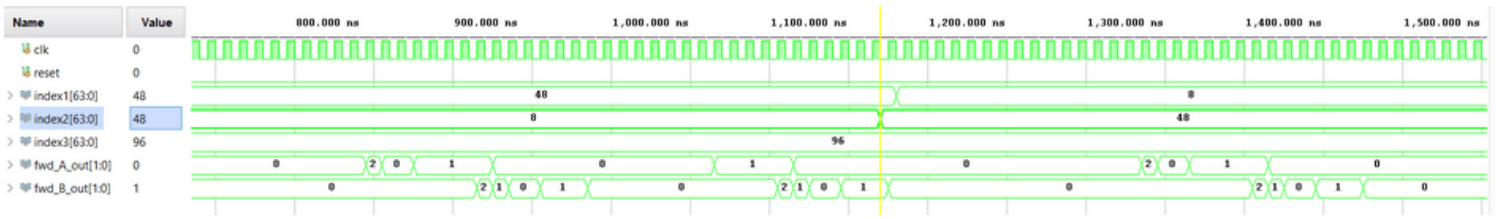
```
module Forwarding_Unit
(
    input [4:0] EXMEM_rd, MEMWB_rd,
    input [4:0] IDEX_rs1, IDEX_rs2,
    input EXMEM_RegWrite, EXMEM_MemtoReg,
    input MEMWB_RegWrite,
    output reg [1:0] fwd_A, fwd_B
);

always @(*) begin

    if (EXMEM_rd == IDEX_rs1 && EXMEM_RegWrite && EXMEM_rd != 0)
        begin
            fwd_A = 2'b10;
        end
    else if (((MEMWB_rd == IDEX_rs1) && MEMWB_RegWrite && (MEMWB_rd != 0))
            &&
            !(EXMEM_RegWrite && (EXMEM_rd != 0) && (EXMEM_rd == IDEX_rs1)))
        begin
            fwd_A = 2'b01;
        end
    else
        begin
            fwd_A = 2'b00;
        end
```

```
    if ((EXMEM_rd == IDEX_rs2) && (EXMEM_RegWrite) && (EXMEM_rd != 0))
        begin
            fwd_B = 2'b10;
        end

    else if (
                ( (MEMWB_rd == IDEX_rs2) && (MEMWB_RegWrite == 1) && (MEMWB_rd != 0) )
            &&
                !( EXMEM_RegWrite && ( EXMEM_rd != 0 ) && ( EXMEM_rd == IDEX_rs2 ) )
            )
        begin
            fwd_B = 2'b01;
        end

    else
        begin
            fwd_B = 2'b00;
        end
end

endmodule // Forwarding_Unit
```

SORTING ON PIPELINED PROCESSOR







PERFORMANCE COMPARISON

The pipelined RISC V processor takes 1180 ns to complete sorting whereas the single cycle processor takes 860 ns to complete sorting. The pipeline processor is thus slower than the single cycle processor and the reason why the pipelined processor performs poorly on bubble sort id due to pipeline stalls which cannot be avoided after hazard detection and forwarding. Since there are no stalls in the single cycle processor, it takes less time.

CHALLENGES

We faced a lot of challenges while doing this project, one of them being that Venus Simulator does not support the double word instructions, so we had to spend some time trying to figure out how will we modify the code and make changes to run the load double and the store double instructions. It was also challenging to figure out that how to implement branch equals to instructions and for solving that we spent a lot of time on internet and at last found a solution which suited and worked as we wanted. Moreover, while implementing pipelined processor with hazard detection unit we found out that syntax error can also occur in Vivado not like a normal one but it just shocked us both the error was if we simulate our code in the version 2020.1 it works fine but as we run it in the older version it fails to simulate.

TASK DIVISION

We worked on the entire project together, as every task was connected to the previous one, and so we both were involved in the working of each task.

CONCLUSION

We were able to execute the bubble sort algorithm successfully on both single cycle processor as well on the Risc V pipelined processor. Our results showed that single cycle processor performs better than the pipelined processor on bubble sort, this is due to data being dependent as bubble sort is a data dependent algorithm which causes stalls in our processor.

APPENDIX

LINK TO GITHUB REPOSITORY

Link