CE-321L/CS-330L

Computer Architecture

RISC V PIPELINE PROCESSOR

Final Report

Team Members:

Naveen Zehra Zaidi

Rasib Qazi

Kulsoom Asim

Date: 20 April 2024

Introduction

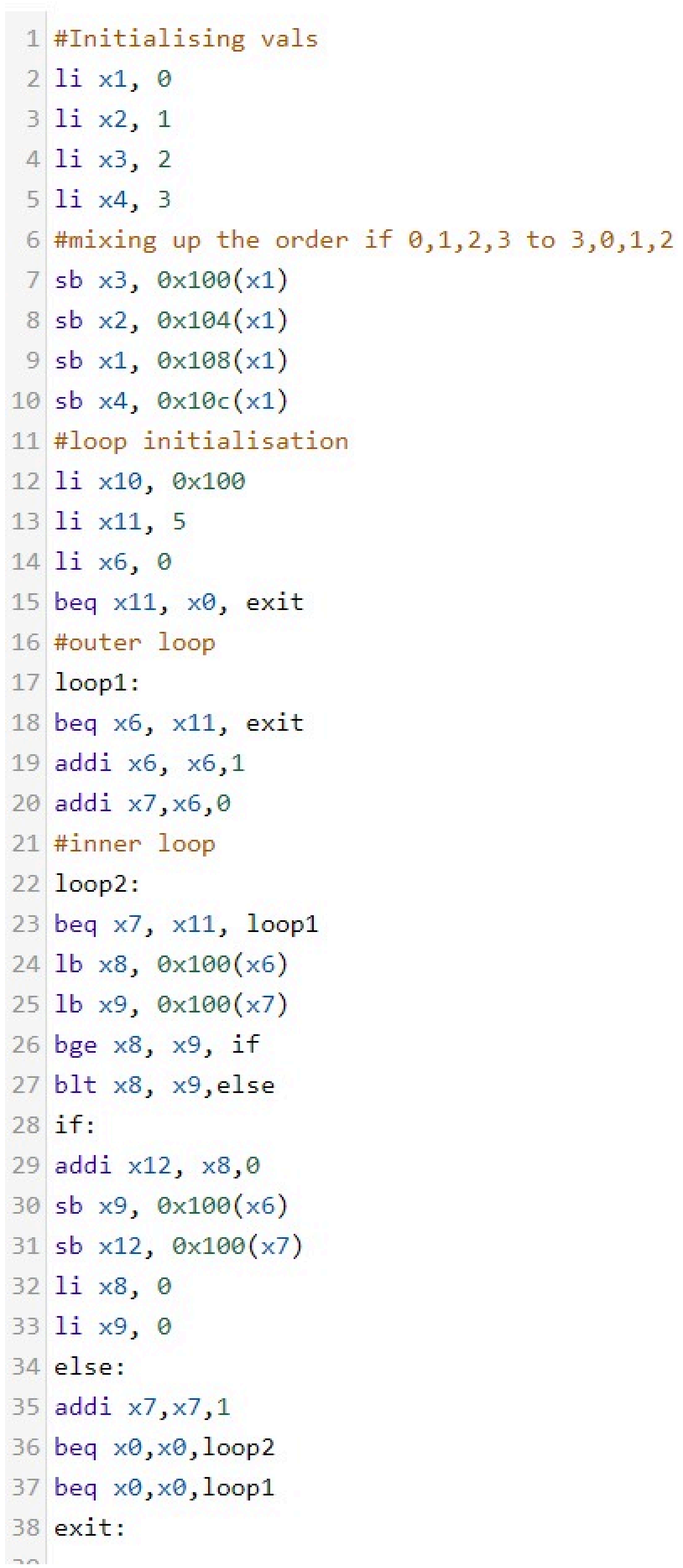
The main aim of this project was to build a 5-stage pipelined RISC V processor that can execute a sorting algorithm. We chose bubble sort as our sorting algorithm. The tasks we had to work on were:

1. Converting bubble sort pseudocode to RISC V assembly code and verifying it on Venus. Then, we modify the lab 11 single-cycle processor to run the bubble sort code.
2. Pipelining the processor and then performing some test instructions separately to ensure the pipelined version works correctly.
3. Introducing hazard detection to detect data/control/structural hazards and trying to handle them in hardware by forwarding, stalling, and flushing the pipeline.
4. Lastly, we had to compare the performance of running the array sorting program on single-cycle processor with a pipelined RISC V processor in terms of execution time.

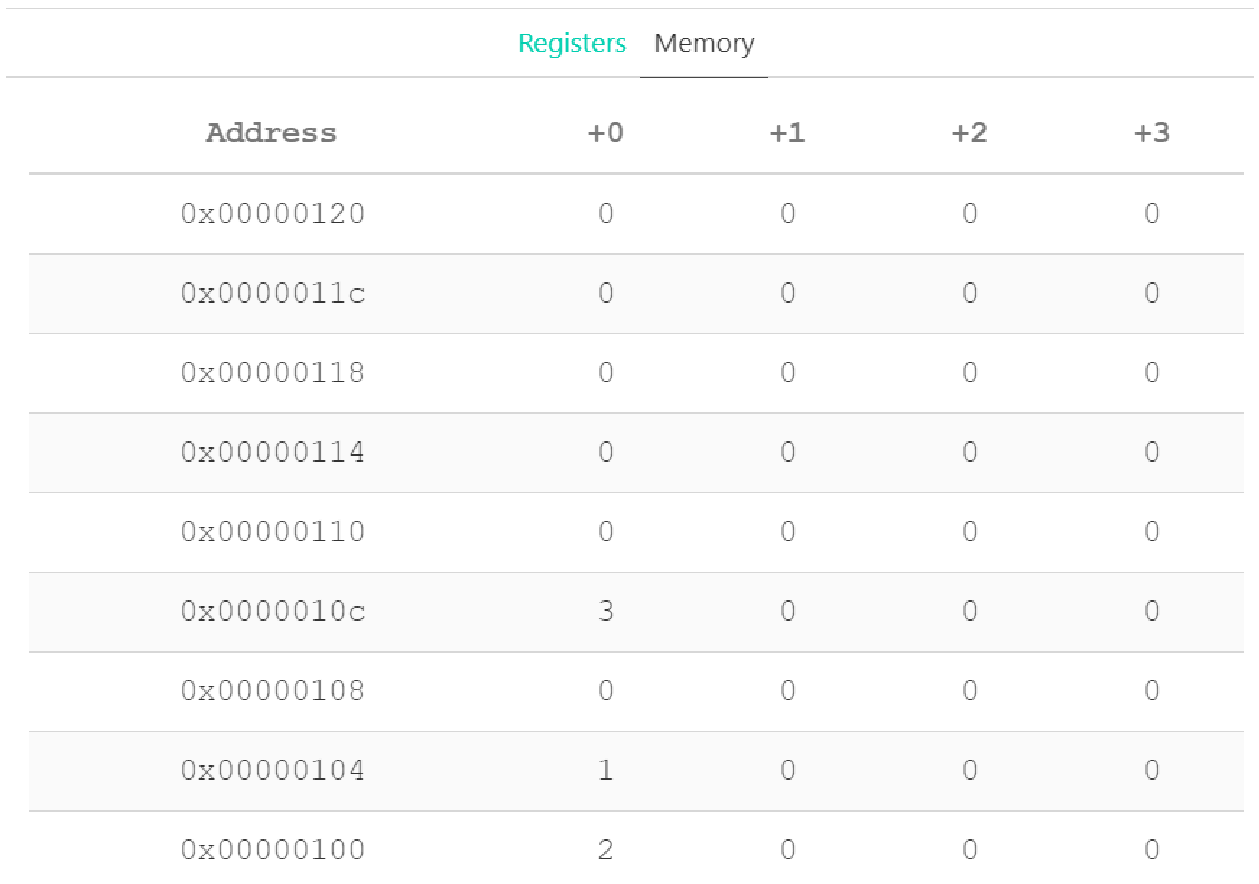
Task 1

- **Bubble Sort Pseudocode to Machine Code**

  Firstly, we had to implement bubble sort in RISC-V assembly on Venus.



```
#Initialising vals
li x1, 0
li x2, 1
li x3, 2
li x4, 3
#mixing up the order if 0,1,2,3 to 3,0,1,2
sb x3, 0x100(x1)
sb x2, 0x104(x1)
sb x1, 0x108(x1)
sb x4, 0x10c(x1)
#loop initialisation
li x10, 0x100
li x11, 5
li x6, 0
beq x11, x0, exit
#outer loop
loop1:
beq x6, x11, exit
addi x6, x6,1
addi x7,x6,0
#inner loop
loop2:
beq x7, x11, loop1
lb x8, 0x100(x6)
lb x9, 0x100(x7)
bge x8, x9, if
blt x8, x9,else
if:
addi x12, x8,0
sb x9, 0x100(x6)
sb x12, 0x100(x7)
li x8, 0
li x9, 0
else:
addi x7,x7,1
beq x0,x0,loop2
beq x0,x0,loop1
exit:
```

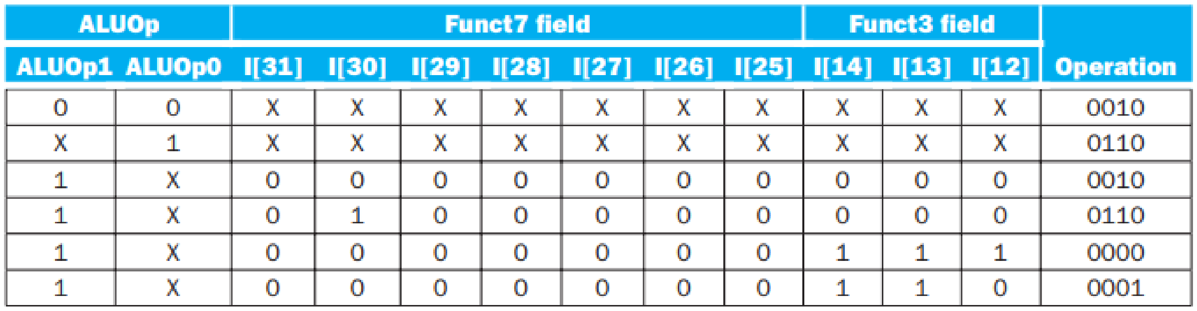

Registers Memory

Address	+0	+1	+2	+3
0x00000120	0	0	0	0
0x0000011c	0	0	0	0
0x00000118	0	0	0	0
0x00000114	0	0	0	0
0x00000110	0	0	0	0
0x0000010c	3	0	0	0
0x00000108	0	0	0	0
0x00000104	1	0	0	0
0x00000100	2	0	0	0

The following screenshot is the result of the bubble sort code taken from Lab 4. The values 0,1,2,3 stored in registers x1,x2,x3,x4.

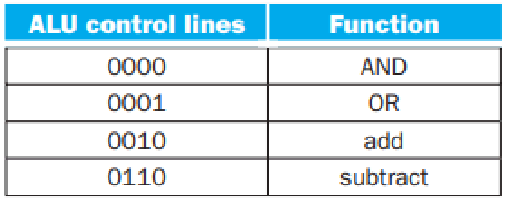
- **Bubble Sort Implementation in Single Cycle**

  The instruction memory module has been expanded to store 22 instructions (from 15 to 87) with 4 bytes each (32-bit instructions). The inst_mem array now has a size of 88 (from 16 to 88), with each element being a byte (8-bit) wide. The values to be sorted through bubble sort were initialised in data memory. Further changes made were that 8-bit wide memory with 256 entries was initialised. Output ports that provided specific values from the memory were also added. A new module, Branch_unit was added, the inputs for it were funct3, a 3 bit signal which determines the type of branch instructions, readData1, 64-bit signal that represents the first operand of the branch instruction and lastly b, a 64-bit signal that represents the second operand of the branch instruction. In the Control_Unit we added more instruction formats, the two I-type format are for load and addi respectively.

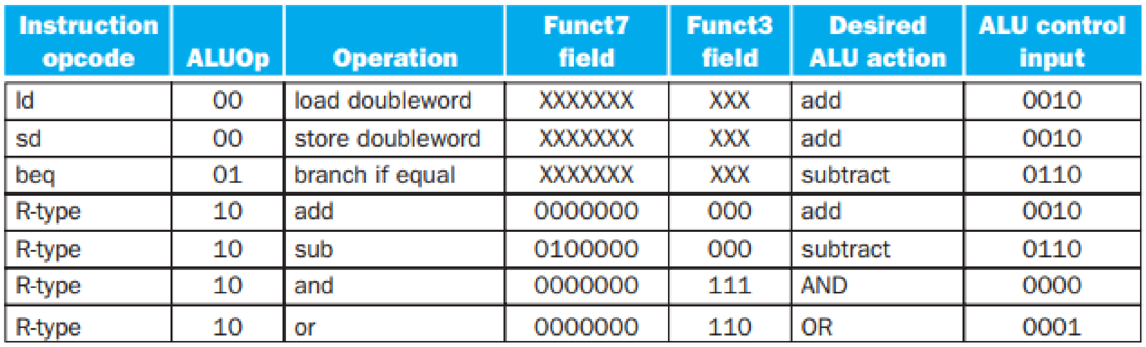


ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

In AlU64bit and ALUcontrol we changed according to the book



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

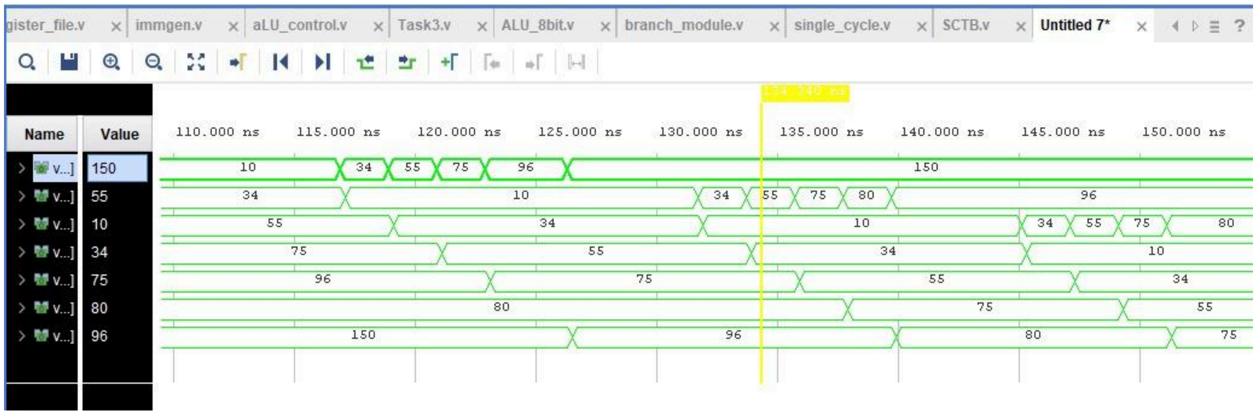


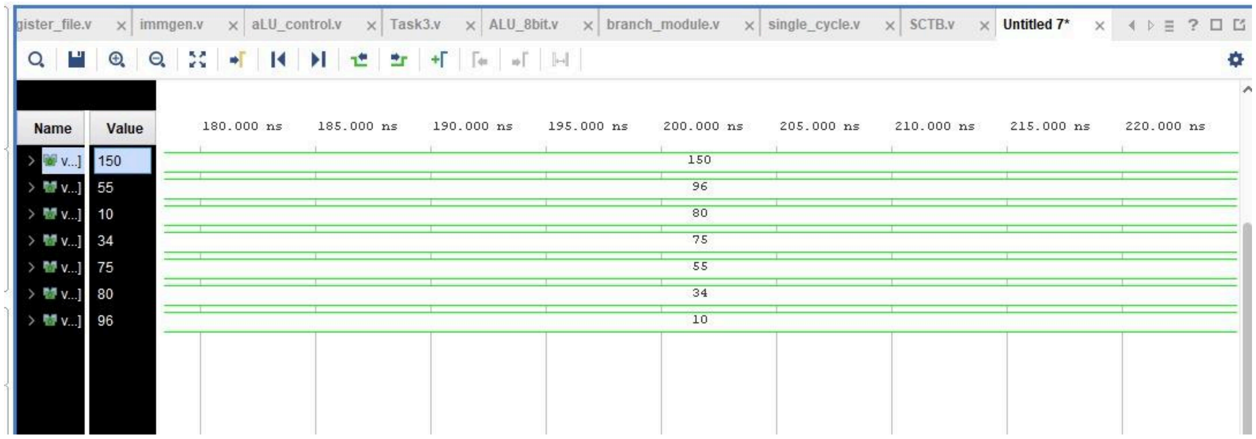
Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Branch Unit Code:

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

  always @(*)
      begin
      case (funct3)
        3'b000:
          begin
            if (readData1 == b)
              addermuxselect = 1'b1;
            else
              addermuxselect = 1'b0;
            end
         3'b100:
           begin
              if (readData1 < b)
              addermuxselect = 1'b1;
            else
              addermuxselect = 1'b0;
            end
        3'b101:
          begin
            if (readData1 > b)
           addermuxselect = 1'b1;
           else
              addermuxselect = 1'b0;
          end
      endcase
     end
endmodule
```

- **Simulation Output**





Task 2

- **Pipelined RISC V Processor**

After implementing bubble sort on the single-cycle processor, we made changes to the code to add pipelining to our modules. We added the following pipeline registers that hold data from one stage of the pipeline to the next:

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

Creating these modules for our registers helped us store the values of previous instruction data and pass it along the pipeline. After this, in order to test that our pipelining was working fine, we had to test each instruction one by one. The forwarding unit was also implemented by modifying the previous code so that data can be forwarded in the pipeline. In order to determine where the data should be forwarded, this forwarding unit will be integrated with the hazard detection unit.

The modules for the pipeline registers:

```
module IF_ID_reg
(
      input[63:0] PC_Out, //Input: Program Counter value
      input[31:0] Instruction,
      input clk, reset, if_id_write,flush,
      output reg[63:0] PC_Out1,
      output reg[31:0] Instruction1
);

      always @ (posedge clk)
      begin
//if reset or flush signal is asserted, PC_Out1 is reset to zero
aswell as Instruction1
            if (reset || flush)
            begin
                  PC_Out1 <= 64'd0;
                  Instruction1 <= 32'd0;
            end

           else if(if_id_write)
            begin
// Update the registered PC_Out and the instruction 1 with the input
value


                  PC_Out1 <= PC_Out;
                  Instruction1 <= Instruction;
            end
      end

endmodule
```

IF/ID

```
module ID_EX_reg(
// Input signals from the previous pipeline stage
      input[63:0] PC_Out1, ReadData1, ReadData2, imm_data,
      input[3:0] Funct,
// Source and destination register addresses
      input[4:0] rs1, rs2, rd,
//control signals
      input RegWrite, Branch, ALUSrc, MemRead, MemWrite, MemtoReg,
      input clk, reset, flush,
      input[1:0] ALUOp,
      output reg[63:0] PC_Out2, ReadData1_1, ReadData2_1, imm_data1,
      output reg[3:0] Funct_1,
      output reg[4:0] rs1_1, rs2_1, rd_1,
      output reg RegWrite_1, Branch_1, ALUSrc_1,
      output reg MemRead_1, MemWrite_1, MemtoReg_1,
      output reg[1:0] ALUOp_1
);

      always @ (posedge clk)// Register values on the rising edge of
the clock
      begin
// Register input signals to the registered outputs
            if (reset || flush)// Reset or flush condition
                  begin
                        PC_Out2 <= 64'd0;
                        ReadData1_1 <= 64'd0;
                        ReadData2_1 <= 64'd0;
                        imm_data1 <= 64'd0;
                        Funct_1 <= 4'd0;
                        rd_1 <= 5'd0;
                        rs1_1 <= 5'd0;
                        rs2_1 <= 5'd0;
                        RegWrite_1 <= 1'b0;
```

```
                    Branch_1 <= 1'b0;
                    ALUSrc_1 <= 1'b0;
                    ALUOp_1 <= 2'b00;
                    MemRead_1 <= 1'b0;
                    MemWrite_1 <= 1'b0;
                    MemtoReg_1 <= 1'b0;
               end

          else
               begin
                    PC_Out2 <= PC_Out1;
                    ReadData1_1 <= ReadData1;
                    ReadData2_1 <= ReadData2;
                    imm_data1 <= imm_data;
                    Funct_1 <= Funct;
                    rd_1 <= rd;
                    rs1_1 <= rs1;
                    rs2_1 <= rs2;
                    RegWrite_1 <= RegWrite;
                    Branch_1 <= Branch;
                    ALUSrc_1 <= ALUSrc;
                    ALUOp_1 <= ALUOp;
                    MemRead_1 <= MemRead;
                    MemWrite_1 <= MemWrite;
                    MemtoReg_1 <= MemtoReg;
               end
     end

 endmodule
```

ID/EX

```
module EX_MEM_reg(
// Input signals from the execute stage
     input[63:0] out, Result, ReadData2_1,
     input[4:0] rd_1,
     input MemtoReg_1, MemWrite_1, MemRead_1, RegWrite_1,
     input branch_op, clk, Branch_1, reset, flush,
// Output signals to the memory stage
     output reg[63:0] out_1, Result_1, ReadData2_2,
     output reg[4:0] rd_2,
     output reg MemtoReg_2, MemWrite_2, MemRead_2,
     output reg RegWrite_2, Branch_2, branch_op1
);

     always @(posedge clk)
     begin
           if (reset || flush) // Check if reset or flush signal is
asserted
                  begin
// Reset all registers and outputs to zero


                        out_1 <= 64'd0;
                        Result_1 <= 64'd0;
                        ReadData2_2 <= 64'd0;
                        rd_2 <= 5'd0;
                        MemtoReg_2 <= 1'b0;
                        MemWrite_2 <= 1'b0;
                        MemRead_2 <= 1'b0;
                        RegWrite_2 <= 1'b0;
                        branch_op1 <= 1'b0;
                        Branch_2 <= 1'b0;
                  end

           else
                  begin
                  // Assign output regs to 1
                        out_1 <= out;
                        Result_1 <= Result;
                        ReadData2_2 <= ReadData2_1;
                        rd_2 <= rd_1;
                        MemtoReg_2 <= MemtoReg_1;
                        MemWrite_2 <= MemWrite_1;
                        MemRead_2 <= MemRead_1;
                        RegWrite_2 <= RegWrite_1;
                        branch_op1 <= branch_op;
                        Branch_2 <= Branch_1;
                end
        end

endmodule
```

EX/MEM

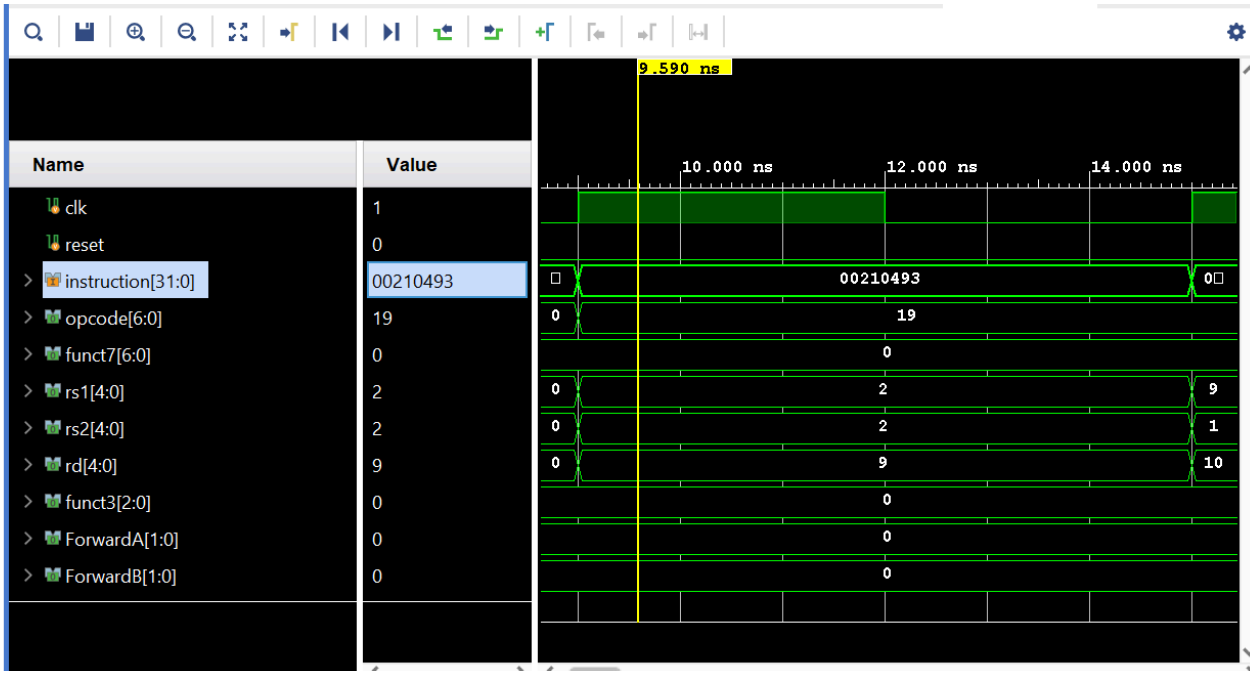
```
module MEM_WB_reg(
        input[63:0] Read_Data, Result_1,//input data from mem
        input RegWrite_2, MemtoReg_2, clk, reset,//control signals
        input[4:0] rd_2,//dest reg
        output reg[63:0] Read_Data_1, Result_2,
        output reg RegWrite_3, MemtoReg_3,//control signals
        output reg[4:0] rd_3 //dest reg
);

        always @(posedge clk)
        begin
                if (reset)//Check if reset
                        begin
                        //Reset all registers and outputs to zero
                                Read_Data_1 <= 64'd0;
                                Result_2 <= 64'd0;
                                RegWrite_3 <= 1'b0;
                                MemtoReg_3 <= 1'b0;
                                rd_3 <= 5'd0;
                        end

                else
                        begin
                        // Assign output regs to 1
                                Read_Data_1 <= Read_Data;
                                Result_2 <= Result_1;
                                RegWrite_3 <= RegWrite_2;
                                MemtoReg_3 <= MemtoReg_2;
                                rd_3 <= rd_2;
                        end
        end
```

MEM/WB

MEM/WB module does not require flush as in the stage the results are written back to the memory, does not have the need to use. Also because hazards are resolved in the previous pipeline stages.

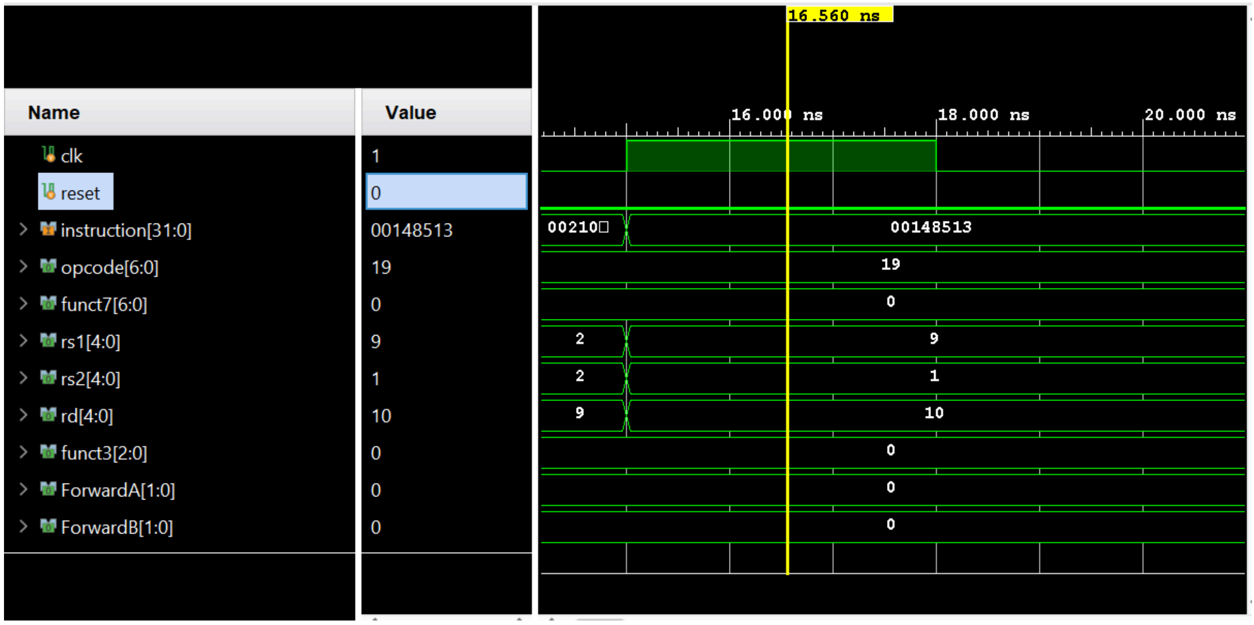
We applied 3 test cases(instructions) in order to be sure that our code works well:

1. **addi x9, x2, 2**



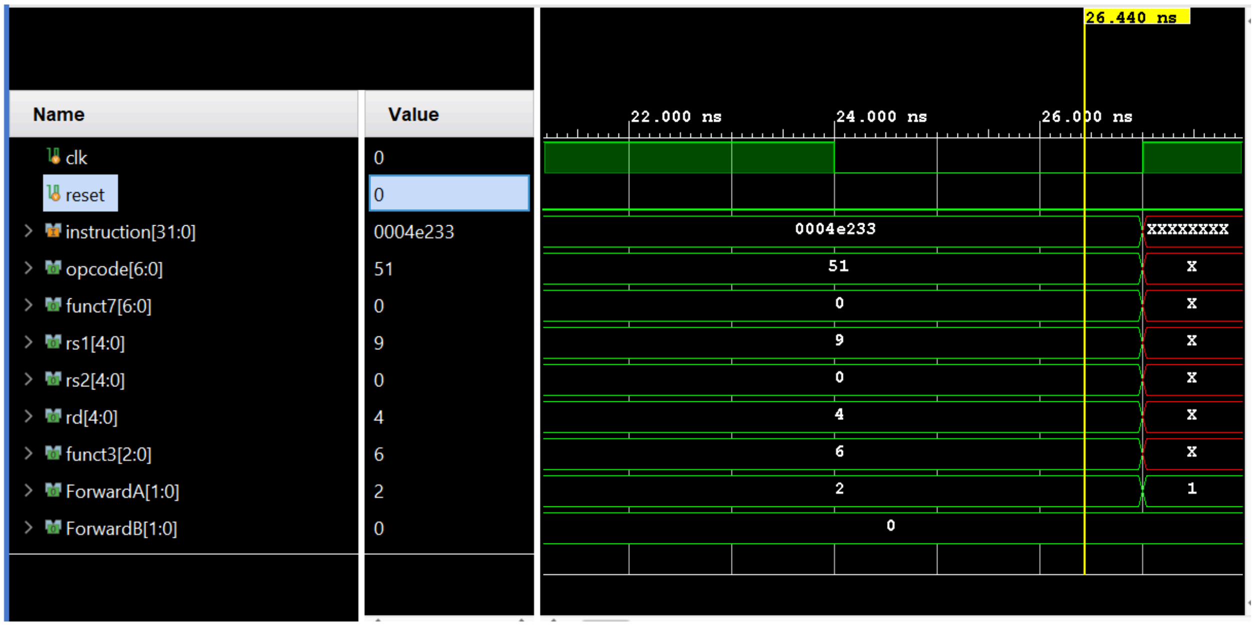
Name	Value
clk	1
reset	0
instruction[31:0]	00210493
opcode[6:0]	19
funct7[6:0]	0
rs1[4:0]	2
rs2[4:0]	2
rd[4:0]	9
funct3[2:0]	0
ForwardA[1:0]	0
ForwardB[1:0]	0

2. **addi x10, x9, 1**

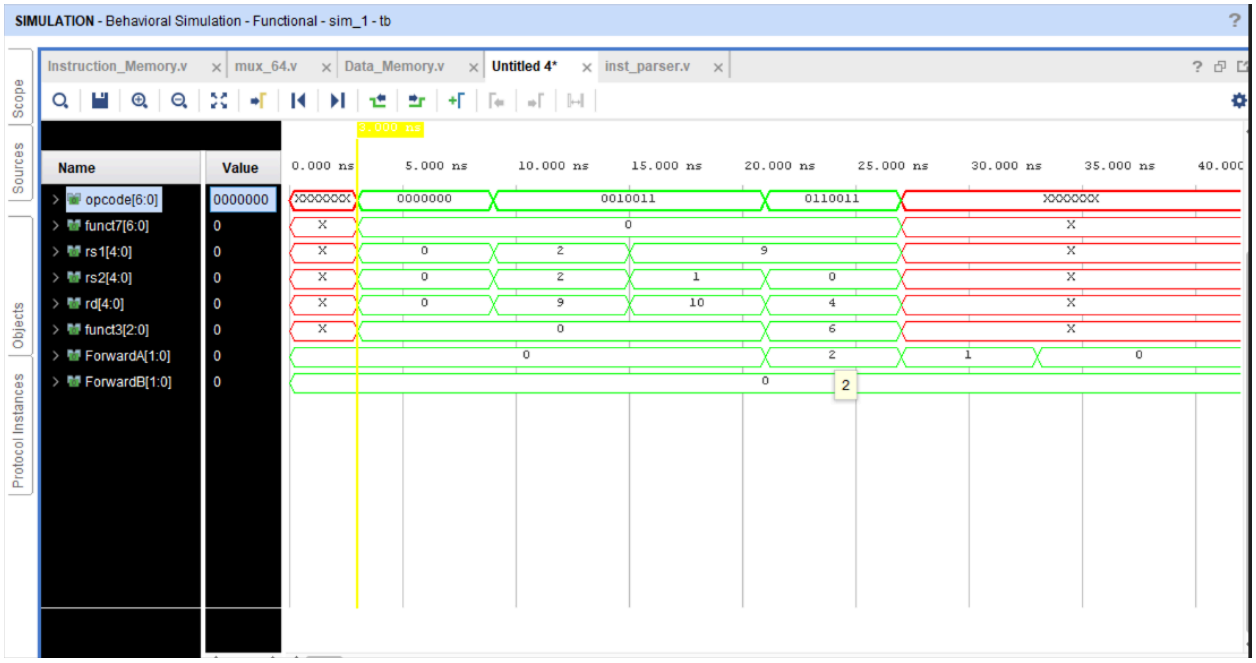


Name	Value
clk	1
reset	0
instruction[31:0]	00148513
opcode[6:0]	19
funct7[6:0]	0
rs1[4:0]	9
rs2[4:0]	1
rd[4:0]	10
funct3[2:0]	0
ForwardA[1:0]	0
ForwardB[1:0]	0

3. **or x4, x9, x0**



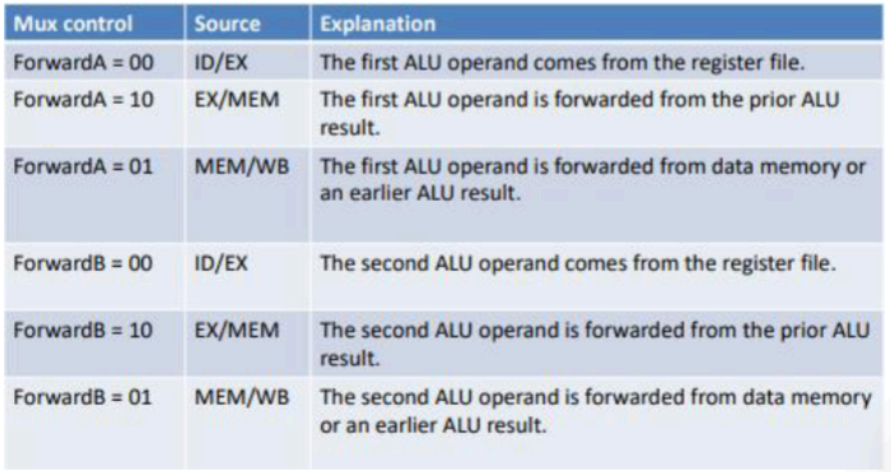
Name	Value
clk	0
reset	0
instruction[31:0]	0004e233
opcode[6:0]	51
funct7[6:0]	0
rs1[4:0]	9
rs2[4:0]	0
rd[4:0]	4
funct3[2:0]	6
ForwardA[1:0]	2
ForwardB[1:0]	0



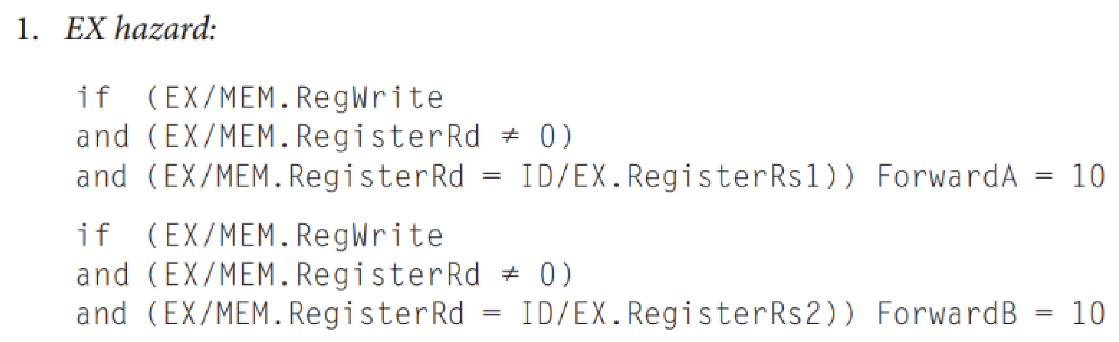
In the above graphs we can see that the instructions can be decoded easily. We can see the forwarding working as the rd register x9 for the first instruction is used in the addition operation of the second operation as rs1. So, forwarding is applied here, we can see that the value for the forward A has changed to 2 which is what is should be for the data hazard in the above 2 instructions. Furthermore, forwarding is also required again where the value of forward A has changed to 1 as it should. This occurs as the rd register, x9, of instruction 1 is needed as the rs1 of the or instruction.

Forwarding Unit:

It was implemented by using the information given in the book and the table given in the project guidelines, which are attached below, we implemented the forwarding unit.

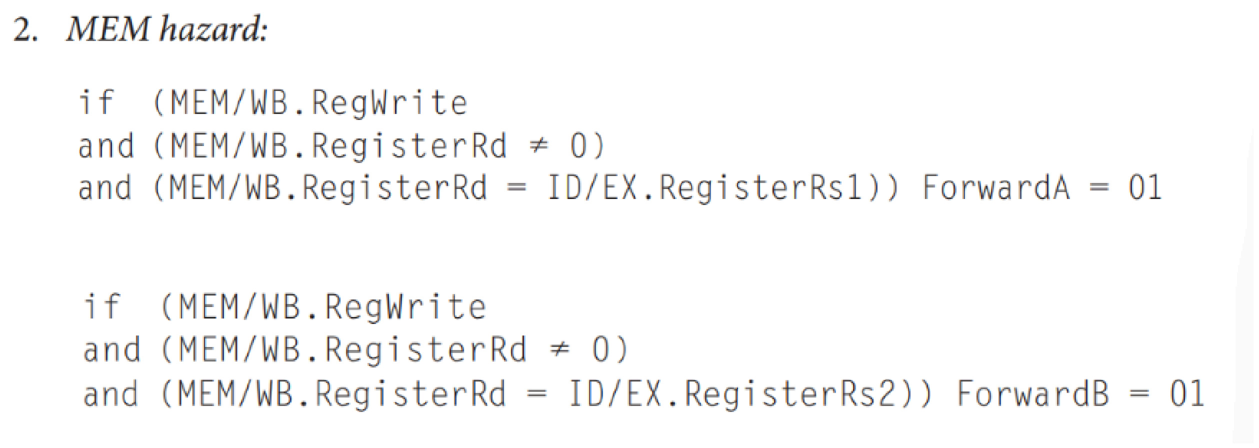


Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX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



1. *EX hazard:*

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```



2. *MEM hazard:*

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01


if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

```
module Forwarding_Unit(
      input[4:0] id_ex_rs1,//Source reg 1 from the ID/EX pipeline stage
input[4:0] id_ex_rs2,// Input: Source reg2 from the ID/EX pipeline stage
input[4:0] ex_mem_rd, // Input: Destination register from the EX/MEM
input[4:0] mem_wb_rd,//Input: Destination register from the MEM/WB
input ex_mem_regwrite,// Signal indicating register write in the EX/MEM
input mem_wb_regwrite,//Signal indicating register write in the MEM/WB
output reg[1:0] ForwardA,//Control signal for forwarding to source reg 1
output reg[1:0] ForwardB//Control signals for forwarding to source reg2



);
    initial
      begin
```

```
            ForwardA = 2'b00;
            ForwardB = 2'b00;
      end


      always @ (*)
      begin
//Forwarding logic for Source Register 1 (ForwardA)
            if (ex_mem_regwrite && (ex_mem_rd != 5'b0) &&
               (ex_mem_rd == id_ex_rs1))
                  ForwardA <= 2'b10;
            else if (mem_wb_regwrite && (mem_wb_rd != 5'b0) &&
                     !(ex_mem_regwrite && (ex_mem_rd != 5'b0)
                         && (ex_mem_rd == id_ex_rs1))
                         && (mem_wb_rd == id_ex_rs1))
                  ForwardA <= 2'b01;
            else
                  ForwardA <= 2'b00;
//Forwarding logic for Source Register 2 (ForwardB)
            if (ex_mem_regwrite && (ex_mem_rd != 5'b0) &&
               (ex_mem_rd == id_ex_rs2))
                  ForwardB <= 2'b10;
            else if (mem_wb_regwrite && (mem_wb_rd != 5'b0) &&
                   !(ex_mem_regwrite && (ex_mem_rd != 5'b0)
                     && (ex_mem_rd == id_ex_rs2))
                     && (mem_wb_rd == id_ex_rs2))
                  ForwardB <= 2'b01;
            else
                  ForwardB <= 2'b00;
      end
endmodule
```

Task 3

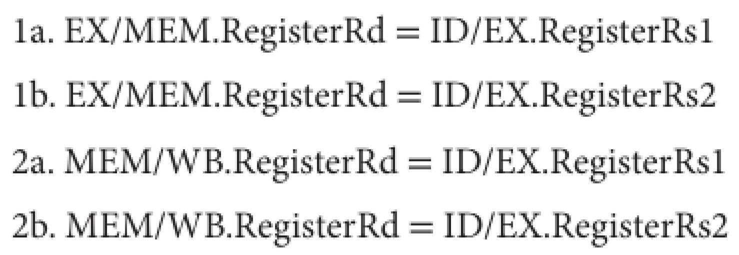
- **Implementing Hazard Detection Circuitry**

Hazard detection circuitry and stalling the pipeline were implemented to deal with all types of hazards, ie. data, structural, and control. Our hazard detection unit control signals the forwarding unit to stall or flush the pipeline, whenever needed.

For this particular task, we made use of the resources provided to us, ie, the table provided in the project guidelines and our text book. The main purpose of the hazard detection module is to stall the pipeline if it detects any dependency.

*The testing for the hazard detection module can be seen to be working fine in the waveform of task 2. Forwarding occurs only when a hazard has been detected, which can be seen in the Task2 waveform.

Using the following resources, given in our book on how to detect the occurring of a hazard we made our hazard detection unit :

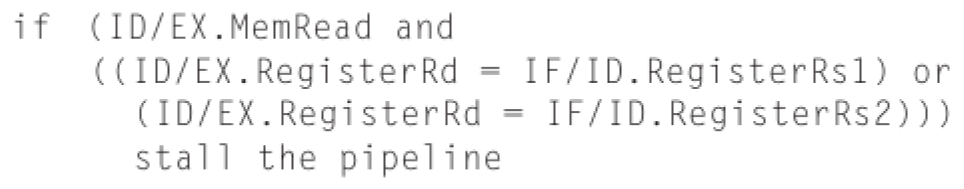


1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1

1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

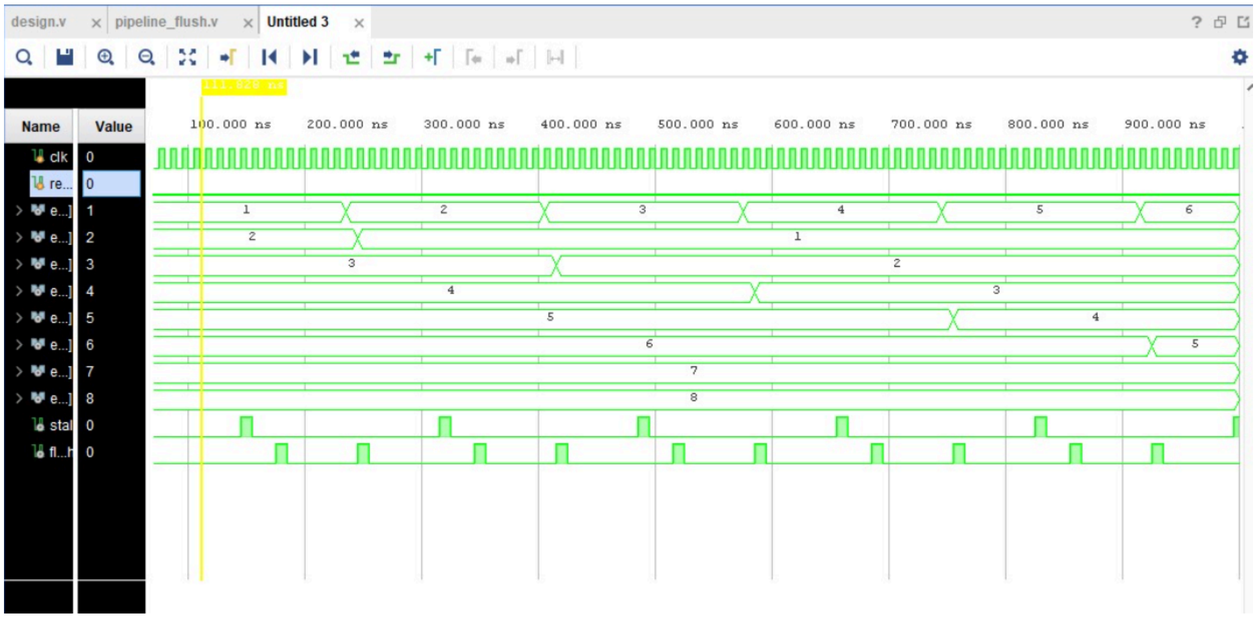


```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
      stall the pipeline
```

```
module Hazard_Detection(
input id_ex_memread,//Indicates whether a memory read is happening in
the ID/EX stage
input[4:0] id_ex_rd,//Destination register in the ID/EX stage
input[4:0] if_id_rs1,// Source register 1 in the IF/ID stage
input[4:0] if_id_rs2,// Source register 2 in the IF/ID stage
output reg PCWrite,// Control signal for enabling PC write
output reg if_id_write,// Control signal for enabling IF/ID write
output reg control_mux_sel,// Control signal for selecting mux
output reg stall //A reg/variable to show stalls in WAVEFORM
);

        initial
        begin
```

```
            PCWrite = 1'b0;
            if_id_write = 1'b0;
            control_mux_sel = 1'b0;
            stall = 1'b0;

      end

//logic for hazard detection
      always @ (*)
      begin
// If there's a memory read in the ID/EX stage and the destination
register
// matches with either of the source registers in the IF/ID stage,
hazard detected
            if ((id_ex_memread) && ((id_ex_rd == if_id_rs1) ||
(id_ex_rd == if_id_rs2)))
            begin
                  PCWrite = 1'b0;
                  if_id_write = 1'b0;
                  control_mux_sel = 1'b0;
                 stall = 1'b1;

            end
          else
            begin// No hazard detected
                  PCWrite = 1'b1;
                  if_id_write = 1'b1;
                  control_mux_sel = 1'b1;
                 stall= 1'b0;


            end
      end
endmodule
```

- **Simulation Output**



The values are sorted in descending order. We see that ‘1’ is swapped with ‘2’ below, and swapped with 3, 4, 5, and 6, and further 2 being swapped with 1. The final output should have been 8,7,6,5,4,3,2,1 however our bubble sort codes swaps partially through., not swapping values. We think, the reason why complete sorting in pipelined processor does not take place can be, what we believe, due to the issue in our stalling or flushing implementation.

Task 4

A pipelined RISC V processor is better than a single-cycle processor for executing array sorting programs. Even though, a single-cycle processor offers simplicity, it forces each instruction to wait for a full cycle to complete, which leads to wasted time, especially in sorting algorithms that involve numerous comparisons and swaps. On the other hand, a pipelined RISC V processor breaks down the instruction cycle into stages, allowing for the simultaneous execution of different stages on separate instructions. This pipelined approach has a significant advantage as the operations are overlapped, thus drastically reducing wasted cycles. This results in a substantial reduction in the overall execution time it takes to sort as compared to the single-cycle processor.

Contribution

Task 1 was done by Kulsoom, Task 2 and 4 by Naveen, and Task 3 by Rasib. We filled in the report for our individual tasks.

Conclusion, and Challenges

In conclusion, the main purpose of this project was to utilise pipelining to increase the efficiency of our processor, thus allowing us to perform multiple tasks with efficiency and ease. The challenges we faced were mainly to implement the changes in our processor. We had issues incorporating the branch instructions. In our task 2, there are issues in our forwarding. Apart from that, only one of us had Verilog installed on our laptop which led to a lot of dependencies being created. For task 3, we were not able to properly make bubble sort work on our code.

References

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

Appendices

https://github.com/Rasib1/realCAProject.git