Computer Architecture CE/CS-321/330

Final Project Report

RISC V Processor



Group Members

Afeera Umair au08735 Mikaal Imam mi08753 Emaan Naveed Sheikh es09163

Instructor

Saima Shaheen

Contents

1 Introduction

- 1.1 Objective
- 1.2 Sorting Algorithm

2 Tasks

2.1 Task 1

- 2.1.1 RISC V Implementation (Single Cycle)
- 2.1.2 Changes
- 2.1.3 Waveform

2.2 Task 2

- 2.2.1 Pipelined RISC V Processor
- 2.2.2 Changes
- 2.2.3 Waveform

2.3 Task 3

- 2.3.1 Handling Data Hazards
- 2.3.2 Changes
- 2.3.3 Waveform

2.4 Task 4

- 2.4.1 Performance Comparison
- 3 Challenges
- **4 Task Division**
- **5** Appendix
 - A: Bubble Sort Code
 - B: Codes for Task 1
 - C: Codes for Task 2
 - D: Codes for Task 3

1) Introduction

1.1 Objective

The objective of this project is to design and implement a **5-stage pipelined RISC-V processor** that can execute an array sorting algorithm. The project involves converting a previously built **single-cycle processor** into a pipelined one. This pipelined architecture allows instructions to overlap in execution through stages: Fetch, Decode, Execute, Memory, and Write Back.

To test the processor's functionality, we selected the **Bubble Sort algorithm**, a straightforward sorting method well-suited for verifying control flow and data dependencies in the pipeline. All relevant codes can be found in the Appendix at the end of the report.

1.11 Sorting Algorithm

We used the bubble sort algorithm to sort our array. We wrote the code and tested it on https://venus.kvakil.me/

2) Tasks

2.1 Task 1

2.1.1 RISC V Implementation Single Cycle

The algorithm was implemented on the RISC-V single-cycle processor developed in our lab by making some necessary changes. To ensure compatibility with the algorithm, the instruction memory, ALU, and ALU Control were modified. Additionally, the branching mux was adjusted for the jump instructions used in blt.

2.1.2 Changes

The processor implemented in the lab did not work for I-type and SB-type instructions. The following modules were modified to make the bubble sort work:

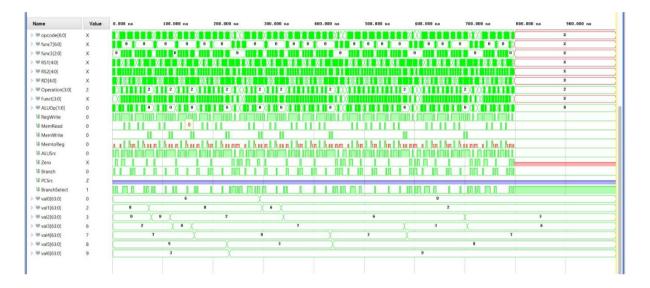
- ALU Control
- Control Unit
- Top level module for RISCV Processor

The slli instruction was added to support left-shifting operations during element comparisons and swaps in Bubble Sort. It is handled within the ALU Control unit when ALUOp = 2'b00 and Funct[2:0] = 4'b001. The blt (branch if less than) instruction is used to enable conditional branching during element comparisons. A branch is triggered if the left element is greater than the right element, prompting a swap.

The ALU Control unit was updated to support both slli and blt, ensuring that proper shifting and branching occur during the sorting process. Support for I-type instructions was also added to handle immediate values and operations.

The Branch Multiplexer was modified to control the flow for the blt instruction, enabling the processor to iterate over the array and perform necessary swaps. Through these modifications, the processor was enabled to efficiently execute Bubble Sort, handling shifts, comparisons, and conditional branching

2.1.3 Waveforms:



The waveform shows the step-by-step sorting of values using Bubble Sort, where initially unsorted values (like 8, 0, 6, etc.) are gradually swapped over time to achieve a sorted order. The values are sorted in ascending order: 0, 2, 6, 7, 8.

2.2 Task 2

2.2.1 Testing 5-Stage/Pipelined RISC V Processor for Sorting Algorithm

In this part, we upgraded our developed processor for Pipeline Execution. We added four new modules and made modifications in three of them.

2.2.2 Changes

To implement pipelining in the processor, new modules were introduced to handle the separation of stages effectively:

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

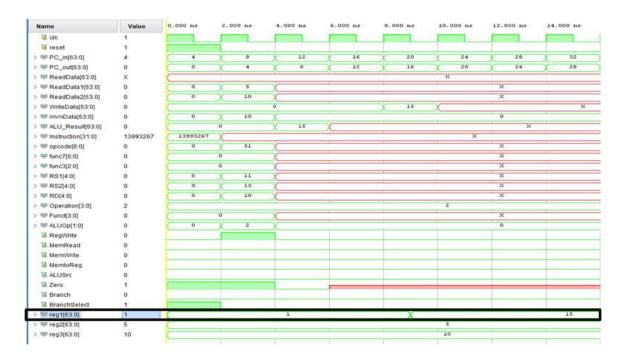
These pipeline registers are responsible for transferring both data and control signals between stages, ensuring proper timing and coordination. This structure forms the foundation of the pipelined architecture and helps minimize data hazards. Along with this, updates were made to the **Register File**, **Instruction Memory**, and the main **Processor Logic** to fully support pipelined operation.

2.2.3 Waveforms

The pipelined processor was tested with individual instructions.

1. add x10 x11 x13

In the waveform below, reg1 = x10, reg2 = x11 and reg3 = x13. The values of reg2 and reg3 are added and stored in reg1 as shown.



2. sub x10 x11 x13

In the waveform below, reg1 = x10, reg2 = x11 and reg3 = x13. The values of reg2 and reg3 are subtracted and stored in reg1 as shown.

Name	Value	0.000 ns	2.000 ns	4.000 ns	6.000 ns	8.000 ns	10.000 ns	12.000 ns	14.000 ns
₩ dk	1				1	1000			
¼ reset	1							-	
> NF PC_in[63:0]	4	4	Х 8	12	16	20	24	28	32
> W PC_out[63:0]	0	0	¥ 4	8	12	16	20	24	28
> MReadData[63:0]	×		-14 8-			-11	×		
> 16 ReadData1[63:0]	0	0	× 5	X				×	
> W ReadData2[63:0]	0	0	10	X				×	100
> WriteData[63:0]	0	K		0		× -5	X		
> 1 ImmData[63:0]	0	0	1034	X			Laboratoria de la companya de la com	0	
> M ALU_Result[63:0]	0	(0	χ -ε	X			×	
> 1 Instruction[31:0]	1087735091	1087735091	X				×		
> 💖 opcode[6:0]	0	0	51	X				×	
> 16# func7[6:0]	0	0	у 32	X				×	
> 16F func3[2:0]	0		0	X				×	
> W RS1[4:0]	0	0	× 11	X				×	
> W RS2[4:0]	0	0	13	X		-		×	
> 10 RD[4:0]	0	0	10	X				×	
> ® Operation[3:0]	2		2	¥ 6	×	1		2	
> % Funct[3:0]	0	0	¥ 8	X				×	
> ₩ ALUOp[1:0]	0	0	χ 2	X			11	0	
¼ RegWrite	0								
14 MemRead	0			-					
1 MemWrite	0								
14 MemtoReg	0								
18 ALUSTO	0								
14 Zero	1								
1 Branch	0				10 11				
M BranchSelect	1		in the second						
> 163:0] > 163:0]	1			1		X			-5
> 15 reg2[63:0]	5	X	14		N.		5	_	10
> 10 reg3[63:0]	10	X-				-	10		

3. slli x10 x11 3

In the waveform below, reg1 = x10, reg2 = x11. The value of reg2 is multiplied by 8 and stored in reg1 as shown.

Name	Value	0.000 ns	2.000 ns	4.000 ns	6.000 ns	8.000 ns	10.000 ns	12.000 ns	14.000 ns
U dk	1				48 (1997)				Garage Co.
[™] reset	1			- 11		11	-		
PC_in[63:0]	4	4	X 8	12	16	20	24	28	32
PC_out[63:0]	0	0	X 4	X 8	12	16	20	24	28
ReadData[63:0]	X	(×		-1
№ ReadData1[63:0]	0	0	X 5	X				×	-
▶ ■ ReadData2[63:0]	0		0	×				х	
> WriteData[63:0]	0			0		¥ 40	X		×
> 1 mmData[63:0]	0	0) з	X	10			0	
MALU_Result[63:0]	0	(0	¥ 40	X			×	
> Instruction[31:0]	3511571	3511571	X				×		
> ® opcode[6:0]	0	0	¥ 19	X				×	
> 16 func7[6:0]	0	(0	X				×	
> [®] func3[2:0]	0	0	¥ 1					×	
> ₩ RS1[4:0]	0	0	11	X	- 10		- 1	×	
> ₩ RS2[4:0]	0	0	у з	-X				×	
> 169 RD[4:0]	0	0	10	X				×	
> SP Operation[3:0]	2		2	8	X			2	
> № Funct[3:0]	0	0	¥ 1					×	
₩ ALUOp[1:0]	0			100			0		
¼ RegWrite	0								
₩ MemRead	0								
1 MemWrite	0								
1 MemtoReg	0								
M ALUSTO	0		d	-					
₩ Zero	1								
1å Branch	0								
1 BranchSelect	1								
₩ reg1[63:0]	1	C		1		X			40
> 10 reg2[63:0]	5	(5		
≥ № reg3[63:0]	10	(10		

4. addi x10 x11 3

In the waveform below, reg1 = x10, reg2 = x11. The value of reg2 + 3 is stored in reg1 as shown.

Name	Value	0.000 ns	2.000 ns	4.000 ns	6.000 ns	8.000 ns	10.000 ns	12.000 ns	14.000 ns
⅓ clk	1			£		4		(a	
₩ reset	1								
PC_in[63:0]	4	4	¥ 8	12	¥ 16	20	24	28	32
PC_out[63:0]	0	0	X 4	χ 8	12	16	20	24	28
№ ReadData[63:0]	X						×		
₩ ReadData1[63:0]	0	0	\$	X				×	
> ■ ReadData2[63:0]	0	¥	0	X			- 0	×	13.1
WriteData[63:0]	0	k		0		χ 8	X		>
> № ImmData[63:0]	0	0	Х 3	X	101			0	
■ ALU_Result[63:0]	0	V	0	χ 8	X			×	
> [∰] Instruction[31:0]	3507475	3507475	X				×		
⇒ w opcode[6:0]	0	0	19	X				×	
> ₩ func7[6:0]	0	V	0	X				×	
> 10 func3[2:0]	0	E	0	X				×	
> № RS1[4:0]	0	0	11	X				×	
> 19 RS2[4:0]	0	(0	Х 3	X				×	
> ® RD[4:0]	0	0	10	X			10	×	
> W Operation[3:0]	2	(2		
> W Funct[3:0]	0	X	0	X	11			×	
> W ALUOp[1:0]	0	X			-111		0		
1 RegWrite	0								
18 MemRead	0								
18 MemWrite	0								
18 MemtoReg	0								
18 ALUSrc	0								
1ª Zero	1								
14 Branch	0			-					
18 BranchSelect	1	3-							
> № reg1[63:0]	1			1		X			8
> ₩ reg2[63:0]	5	(-					5		
⇒ reg3[63:0]	10	¥					10		- 11-

5. beq x13 x11 84 (untaken)

In the waveform below, reg2 = x11 and reg3 = x13. Since the values of reg2 and reg3 are unequal, the branching condition is false, and we observe no change in PC values.

Name	Value	0.000 ns	2.000 ns	4.000 ns	6.000 ns	8.000 ns	10.000 ns	12.000 ns	14.000 ns
¹ d clk	1				-	2			-
¼ reset	1			-					
₩ PC_in[63:0]	4	4	X 8	12	16	20	24	28	32
PC_out[63:0]	0	0	X 4	X 8	12	16	20	24	28
ReadData[63:0]	X	(×		
■ ReadData1[63:0]	0	0	10	X				×	
■ ReadData2[63:0]	0	0	X 5	X	12			×	
WriteData[63:0]	0	<u> </u>	1	0			X		
> № ImmData[63:0]	0	0	42	X				0	
ALU_Result[63:0]	0	X	0	× 5	X			×	
Instruction[31:0]	79071843	79071843	X		100	117	×		
> ® opcode[6:0]	0	0) 99	X				×	
> 💖 func7[6:0]	0	0	Х 2	X		l.		×	
> 10 func3[2:0]	0	V.	0 X					×	
> NF RS1[4:0]	0	0	¥ 13	X				×	
> SP RS2[4:0]	0	0	11	X				×	
> W RD[4:0]	0	0	X 20	X				×	
> W Operation[3:0]	2	-	2	χ 6	X			2	
> ₩ Funct[3:0]	0	\	0	X		1.11		×	
> ₩ ALUOp[1:0]	0	0	X 1	×				0	-
18 RegWrite	0								
18 MemRead	0								
1å MemWrite	0								
lå MemtoReg	0								
1ª ALUSIC	0								
1å Zero	1								
1 Branch	0				1				
1 BranchSelect	1								
₩ reg1[63:0]	1	(1		
№ reg2[63:0]	5						s		
> 16 reg3[63:0]	10	7					10		

6. beq x2 x2 80 (taken)

The branching condition is true, and we observe a change in PC values.

Name	Value	0.000 ns	2.000 ns	4.000 ns	6.000 ns	8.000 ns	10.000 ns	12.000 ns
U dk	1				H-MAN S			
⅓ reset	1					- 1		
PC_in[63:0]	4	4	Х 8	12	80	84	88	92
■ PC_out[63:0]	0	0	X 4	8	12	80	X 84	88
■ ReadData[63:0]	Х						×	- 0.1
ReadData1[63:0]	0		0	X				×
ReadData2[63:0]	0		0	X				ж
WriteData[63:0]	0			0		-	X	
ImmData[63:0]	0	0	40	X			10 min	0
₩ ALU_Result[63:0]	0		0		X			×
Instruction[31:0]	69273699	69273699	X				×	
₩ opcode[6:0]	0	0	99	X				×
¶ func7[6:0]	0	0	χ 2	X				×
№ func3[2:0]	0		0	X				×
₩ RS1[4:0]	0	0	2	X				×
₩ RS2[4:0]	0	0	χ 2	X				×
W RD[4:0]	0	0	16	X				×
SP Operation[3:0]	2		2	6	X	1	11	2
₩ Funct[3:0]	0	(0	X				×
₩ ALUOp[1:0]	0	0	1	X				0
₩ RegWrite	0							
1 MemRead	0							
¼ MemWrite	0							
MemtoReg	0			_				
14 ALUSIC	0							
1 Zero	1							
1 Branch	0							
☑ BranchSelect	1							
™ reg1[63:0]	1						1	
₩ reg2[63:0]	5						5	
** reg3[63:0]	10	V	-				10	

7. blt x13 x11 76 (untaken)

In the waveform below, reg2 = x11 and reg3 = x13. Since the value of reg3 is greater than reg2, the branching condition is false, and we observe no change in PC values.

Name	Value	0.000 ns	2.000 ns	4.000 ns	6.000 ns	8.000 ns	10.000 ns	12.000 ns
U dk	1			- B				
18 reset	1							
PC_in[63:0]	4	4	8	12	16	20	24	28
₩ PC_out[63:0]	0	0	X 4	8	12	16	20	24
ReadData[03.0]	X							
ReadData1[63:0]	0	0	10	X				×
≫ ReadData2[63:0]	0	0	X 5	X				×
₩ WriteData[63:0]	0		and a	0	1		X	
➡ ImmData[63:0]	0	0	38	X				0
ALU_Result[63:0]	0	<	0	5	X			×
Instruction[31:0]	79087203	79087203	X		1042		×	
opcode[6:0]	0	0	X 99	X	1			×
> 15 func7[6:0]	0	0	2	X				×
₩ func3[2:0]	0	0	X 4	X	,			×
₩ RS1[4:0]	0	0	13	X				×
≫ RS2[4:0]	0	0	11	X				×
> № RD[4:0]	0	0	12					×
Operation[3:0]	2		2	X 6	X			2
> ₩ Funct[3:0]	0	0	X 4	X				×
	0	0	X 1	X				0
¼ RegWrite	0							
14 MemRead	0							
1 MemWrite	0							
¼ MemtoReg	0		0.					
1∄ ALUSrc	0							
₩ Zero	1				0.			
1 Branch	0							
BranchSelect	1							
™ reg1[63:0]	1	<			1		1	
₩ reg2[63:0]	5	7				- 1	5	
≥ ® reg3[63:0]	10	7					10	

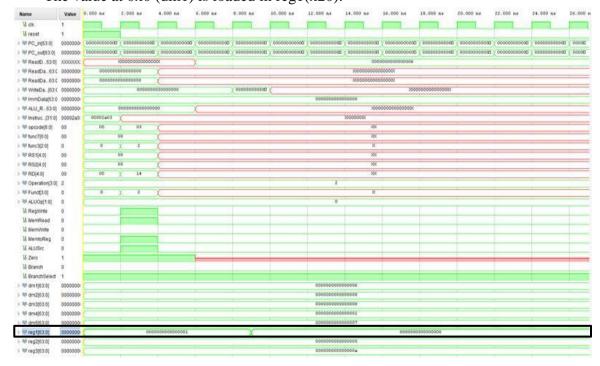
8. blt x11 x13 72 (taken)

In the waveform below, reg2 = x11 and reg3 = x13. Since the value of reg2 is less than reg3, the branching condition is true, and we observe a change in PC values.

Name	Value	0.000 ns	2.000 ns	4.000 ns	6.000 ns	8.000 ns	10.000 ns	12.000 ns
18 dk	1							
₩ reset	1				1.00		- 13	- 1
> III PC_in[63:0]	4	4	8	12	72	76	80	84
PC_out[63:0]	0	0	X 4	8	12	72	76	80
ReadData[63:0]	X	(×	-10
■ ReadData1[63:0]	0	0	5					×
■ ReadData2[63:0]	0	0	10	X				×
WriteData[63:0]	0			0	111		X	
№ ImmData[63:0]	0	0	36	×	0.0		1.84	0
M ALU_Result[63:0]	0		0	-5	X			>
Instruction[31:0]	81118307	81118307	X				×	
popcode[6:0]	0	0	X 99	X				×
> ₩ func7[6:0]	0	0	2		.hii	17		×
₩ func3[2:0]	0	0	X 4					×
> ® RS1[4:0]	0	0	X 11	X				×
> ₩ RS2[4:0]	0	0	13					×
→ NF RD[4:0]	0	0	× 8	X				×
→ W Operation[3:0]	2		2	¥ 6	X	77		2
₩ Funct[3:0]	0	0	X 4	X				×
₩ ALUOp[1:0]	0	0	X 1	X				0
¹ RegWrite	0							
1 MemRead	0							
1å MemWrite	0							
₩ MemtoReg	0							
1å ALUSrc	0							
18 Zero	1							
1 Branch	0							
1 BranchSelect	1							
⇒ 1 reg1[63:0]	1	0					1	
₩ reg2[63:0]	5	(-	- W		il e	5	-1-
® reg3[63:0]	10	7	-10			01	10	

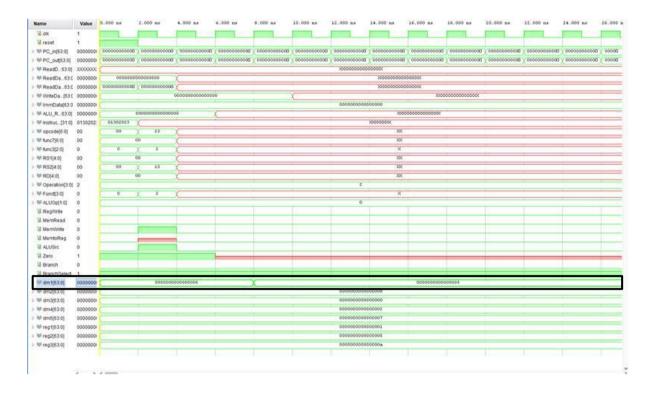
9. lw x20, 0(x0)

The value at 0x0 (dm1) is loaded in reg1(x20).



10. sw x19, 0(x0)

The value at x19(reg2) is stored in dm1(0x0).



2.3 Task 3

2.3.1 Handling Data Hazards

To manage data hazards in our circuit design, we implemented modules that can both detect hazards and control pipeline stalling. These detection units help determine when data forwarding is needed. Once a hazard is identified, the forwarding logic is activated to resolve it.

2.3.2 Changes

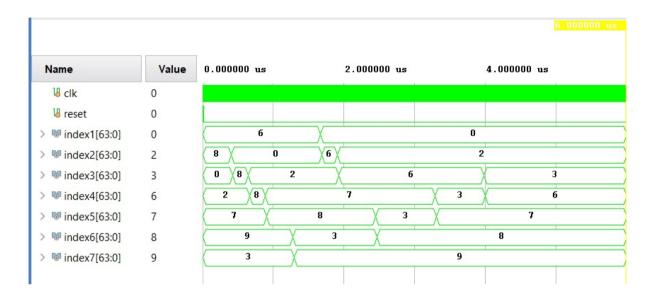
The following modules were added/modified:

- Hazard Detection Unit
- Forwarding Unit

We added a Hazard_Detection module to our processor to manage data hazards, particularly load-use hazards. This module takes IDEX_rd, IFID_rs1, IFID_rs2, and IDEX_MemRead as inputs to detect potential conflicts between consecutive instructions. If a hazard is detected—specifically when the instruction in the ID/EX stage is reading from memory and its destination register matches one of the source registers of the instruction in the IF/ID stage—the module asserts control signals to stall the pipeline.

When such a hazard is detected, the Hazard_Detection module sets IDEX_mux_out, IFID_Write, and PCWrite to 0, which prevents updates to the PC and IF/ID pipeline register and inserts a bubble in the pipeline. Otherwise, all signals are set to 1, allowing normal instruction flow. This addition improves the processor's ability to handle hazards efficiently, ensuring correct execution and enhancing overall performance.

2.3.3 Waveform



2.4 Task 4

2.4.1 Performance Comparison

Single Cycle Processor:

- Clock cycle time = 2 ns
- Execution time = 640 ns
- Clock cycles = 640/2 = 320cc

Pipelined Processor:

- Clock cycle time = 2/5 = 0.4ns
- Execution time = 911 ns
- Clock cycles = 911/0.4 = 2,277.5 cc

Speed up:

- Speedup = Single Cycle Execution Time / Pipelined Execution Time
- Speed up = 640/911 = 0.7

The pipelined processor for our project takes more time than the single cycle processor. It is 0.7 times slower.

3) Challenges

In task 1, our initial bubble sorting code was incorrect. A lot of time was spent on debugging the processor even though our instruction memory was incorrect. We figured out that our instruction memory was wrong and updated it. For task 2, we struggled in making the branch instructions work. Task 3 overall was very complex and writing the top level module took a lot of time and a lot of debugging.

4) Task Division

Task 1 : Mikaal, Afeera, EmaanTask 2: Mikaal, Emaan, Afeera

Task 3: Mikaal, AfeeraTask 4: Mikaal, AfeeraReport: Afeera, Emaan

5) Appendix

A: Bubble Sort Code

```
1 #assuming all regs are initialised at 0
 2 addi x18 x0 0x100 #base addy
 3 addi x19, x0, 7
                        # length of the array
4 outer_loop:
 5
      beq x8, x19, exit # exit the program
                       # j = 0
 6
      addi x9, x0, 0
     addi x10, x0, 0 # reseting swapping flag
 7
      addi x6, x19, -1 # Limit for the inner loop
 8
9
      sub x6, x6, x8  # making the inner loop smaller
10 inner_loop:
11
     beq x9, x6, inner_loop_exit
     slli x7, x9, 2 # Multiply i by 4
12
      add x7, x7, x18
                        # Calc addr of a[j]
13
14
     lw x5, 0(x7)
                   # Load a[j] into x5
                        # j+1
15
     addi x29, x9, 1
     slli x28, x29, 2 # Multiply i+1 by 4
16
     add x28, x28, x18 # calc addr of a[j+1]
17
                       # load a[j+1]
18
     lw x30, 0(x28)
19
     bge x5 x30 swap
                       #branch if a[j] > a[j+1]
      #blt x30, x5, swap # Branch if a[j+1] < a[j] then swap
20
                        # j += 1
21
      addi x9, x9, 1
22
      beq x0, x0, inner_loop # next iter inner_loop
23 swap:
24
      addi x31, x5, 0
                        # swapping starts
25
      addi x5, x30, 0
     sw x5, 0(x7)
26
27
     addi x30, x31, 0
28
     sw x30, 0(x28)
29
     addi x10, x0, 1
                        # swapped falg = true
      addi x9, x9, 1
30
      beq x0, x0, inner_loop # next iter inner_loop
31
32 inner_loop_exit:
                        # i += 1
33
      addi x8, x8, 1
34
      beq x10, x0, exit # if no swapps exit program
35
      beq x0, x0, outer_loop # next iter outer_loop
36 exit:
```

B: Codes for Task 1

ALU Control Module:

```
3  module ALU_Control(ALUOp, Funct, Operation);
 5 input [1:0] ALUOp;
 6 | input [3:0] Funct;
 7 output reg [3:0] Operation;
9 - always@ (*)
10 🖯 begin
11 🖯
         case (ALUOp)
12 🖯
             2'b00: // I Type
13 🖯
                 begin
14 🗇
                 case (Funct[2:0])
15 🖨
                     4'b001 :
16 🗀
                         Operation = 4'b1000; // slli
17 🖨
                     default:
18 🖨
                        Operation = 4'b0010; // sd n ld
19 🖨
                endcase
20 🗀
                 end
21 🖯
             2'b01: // SB Type
22 🖨
                 begin
23
                     Operation = 4'b0110; //beq
24 🖒
25 🗇
             2'b10: // R Type
26 🖨
                 begin
27 🖯
                     case (Funct)
28 🖨
                         4'b0000:
29 🖨
                             Operation = 4'b0010; // add
30 □
                         4'b1000:
31 🖒
                             Operation = 4'b0110; // sub
32 ⊖
                         4'b0111:
13 🖨
                             Operation = 4'b00000; // and
4 🖯
                         4'b0110:
15 🖒
                             Operation = 4'b0001; // or
16 🖨
                      endcase
i7 🖨
                 end
18 🖨
         endcase
19 🗀 end
10 endmodule
```

Control Unit:

```
`timescale lns / lps
 3   module Control_Unit(
        input [6:0] opcode,
 5
         output reg [1:0] ALUOp,
        output reg Branch,
 6
 7
        output reg MemRead,
 8
        output reg MemtoReg,
        output reg MemWrite,
9
10
       output reg ALUSrc,
11
         output reg RegWrite);
12
13 🖨 always@(*)begin
14 🖯
      case (opcode)
15 Ö
            7'b0110011: // R type
16 🖯
               begin
17
                ALUSrc = 0;
18
19
20
                 MemtoReg = 0;
                 RegWrite = 1;
                MemRead = 0;
21
                MemWrite = 0;
22
                Branch = 0;
23
               ALUOp = 2'b10;
24 🖨
                 end
25 🖨
             7'b0000011: // I type (ld)
26 🗇
                begin
27
28
                 ALUSrc = 1;
                 MemtoReg = 1;
29
                RegWrite = 1;
30
                MemRead = 1;
               MemWrite = 0;
31
32
33
                Branch = 0;
                ALUOp = 2'b00;
 34 🖨
                end
           7'b0100011: // S type (sd)
 35 🖯
 36 🖨
                begin
 37
                ALUSrc = 1;
38
39
40
               MemtoReg = 1'bx;
                RegWrite = 0;
               MemRead = 0;
 41
42
43
               MemWrite = 1;
                Branch = 0;
                ALUOp = 2'b00;
 44 🖨
               end
 45 🖨
           7'b1100011: // SB type (beq)
 46 🖯
               begin
 47
48
49
                ALUSrc = 0;
                MemtoReg = 1'bx;
               RedWrite = 0;
50
51
52
               MemRead = 0;
                MemWrite = 0;
                Branch = 1;
 53
                ALUOp = 2'b01;
 54 🖨
                end
             7'b0010011: // I type (addi)
 55 ⊕
 56 🖯
                begin
57
58
59
60
                ALUSrc = 1;
                MemtoReg = 0;
                RegWrite = 1;
                MemRead = 0;
```

```
61
                 MemWrite = 0;
62
                  Branch = 0;
63 :
                  ALUOp = 2'b00;
64 🖒
65 E
             default: // initializing
66 🖨
                 begin
67
                  ALUSrc = 0;
68
                  MemtoReg = 0;
69
                  RegWrite = 0;
70
                 MemRead = 0;
71
                 MemWrite = 0;
72
                  Branch = 0;
73
                  ALUOp = 2'b00;
74 🗀
                  end
75 🖨
         endcase
76 🖨 end
77 endmodule
78
```

Instruction Memory:

```
1 | `timescale lns / lps
 3 - module Instruction Memory(
       input [63:0] Inst_Address,
        output reg [31:0] Instruction);
 5
     reg [7:0] inst_mem [120:0];
9 - initial
10 🖯
         begin
           \{inst\_mem[3], inst\_mem[2], inst\_mem[1], inst\_mem[0]\} = 32'h00500993; // 1
11
12
            {inst mem[7], inst mem[6], inst mem[5], inst mem[4]} = 32'h07340663; // 2
            {inst_mem[11], inst_mem[10], inst_mem[9], inst_mem[8]} = 32'h00000493; // 3
13
           \{inst\_mem[15], inst\_mem[14], inst\_mem[13], inst\_mem[12]\} = 32'h00000513; // 4
14
           {inst mem[19], inst mem[18], inst mem[17], inst mem[16]} = 32'hfff98313; // 5
16
           {inst_mem[23], inst_mem[22], inst_mem[21], inst_mem[20]} = 32'h40830333; // 6
17
           \{inst\_mem[27], inst\_mem[26], inst\_mem[25], inst\_mem[24]\} = 32'h04648663; // 7
18
           {inst_mem[31], inst_mem[30], inst_mem[29], inst_mem[28]} = 32'h00349393; // 8
19
           {inst mem[35], inst mem[34], inst mem[33], inst mem[32]} = 32'h012383b3; // 9
           \{inst\_mem[39], inst\_mem[38], inst\_mem[37], inst\_mem[36]\} = 32'h0003b283; // 10
20
           {inst_mem[43], inst_mem[42], inst_mem[41], inst_mem[40]} = 32'h00148e93; // 11
22
            {inst_mem[47], inst_mem[46], inst_mem[45], inst_mem[44]} = 32'h003e9e13; // 12
23
            {inst_mem[51], inst_mem[50], inst_mem[49], inst_mem[48]} = 32'h012e0e33; // 13
            {inst_mem[55], inst_mem[54], inst_mem[53], inst_mem[52]} = 32'h000e3f03; // 14
25
            {inst_mem[59], inst_mem[58], inst_mem[57], inst_mem[56]} = 32'h005f4663; // 15
           {inst_mem[63], inst_mem[62], inst_mem[61], inst_mem[60]} = 32'h00148493; // 16
26
           {inst mem[67], inst mem[66], inst mem[65], inst mem[64]} = 32'hfc000ce3; // 17
28
            {inst_mem[71], inst_mem[70], inst_mem[69], inst_mem[68]} = 32'h00028f93; // 18
            {inst_mem[75], inst_mem[74], inst_mem[73], inst_mem[72]} = 32'h000f0293; // 19
29
            {inst mem[79], inst mem[78], inst mem[77], inst mem[76]} = 32'h0053b023; // 20
```

```
31
           \{inst_mem[83], inst_mem[82], inst_mem[81], inst_mem[80]\} = 32'h000f8f13; // 21
32
           {inst_mem[87], inst_mem[86], inst_mem[85], inst_mem[84]} = 32'h01ee3023; // 22
33
           {inst mem[91], inst mem[90], inst mem[89], inst mem[88]} = 32'h00100513; // 23
           \{inst\_mem[95], inst\_mem[94], inst\_mem[93], inst\_mem[92]\} = 32'h00148493; // 24
34
35
           {inst mem[99], inst mem[98], inst mem[97], inst mem[96]} = 32'hfa000ce3; // 25
36
           [103], inst_mem[102], inst_mem[101], inst_mem[100]} = 32'h00140413; // 26
            {inst_mem[107], inst_mem[106], inst_mem[105], inst_mem[104]} = 32'h00050463; // 27
37
38
            {inst_mem[111], inst_mem[110], inst_mem[109], inst_mem[108]} = 32'hf8000ce3; // 28
39 🖨 end
40
41 
always @(Inst_Address)begin
         Instruction[31:24] <= inst mem[Inst Address + 3];</pre>
43
         Instruction[23:16] <= inst_mem[Inst_Address + 2];</pre>
44
         Instruction[15:8] <= inst mem[Inst Address + 1];</pre>
45
         Instruction[7:0] <= inst_mem[Inst_Address];</pre>
46 🖒 end
47 endmodule
```

RISC V Single Cycle Processor:

```
1 | 2 |
      timescale lns / lps
3 ⊝ module RISCV_SCProcessor(
         input clk,
         input reset,
         output [63:0] PC in, PC out, ReadData, ReadDatal, ReadData2, WriteData, ImmData, Result,
7
         output [63:0] shifted_data, Data_Out, Out1, Out2,
8
         output [31:0] Instruction,
         output [6:0] opcode, func7,
10
         output [2:0] func3,
11
         output [4:0] RS1, RS2, RD,
12
         output [3:0] Operation, Funct,
13
         output [1:0] ALUOp,
14
         output RegWrite, MemRead, MemWrite, MemtoReg, ALUSrc, Zero, Branch, PCSrc, BranchSelect ,
15
         output [63:0] dml, dm2, dm3, dm4, dm5
16
         );
17
18
         // Fetching
19
         Adder FOURADDER(PC_out, 64'd4, Outl);
20
21
         Mux 2x1 BRANCH(Out1, Out2, (Branch & BranchSelect), PC in);
22
         Program_Counter PC(clk, reset, PC_in, PC_out);
23
         Instruction_Memory IM(PC_out, Instruction);
24
25
         // Decode
26
         Instruction_Parser IP(Instruction, opcode, RD, func3, RS1, RS2, func7);
27
         Imm Gen IG(Instruction, ImmData);
28
         Control_Unit CU(opcode, ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite);
29
         Register_File RF(WriteData, RS1, RS2, RD, RegWrite, clk, reset, ReadData1, ReadData2);
30
         Branch Unit BU(func3, ReadData1, ReadData2, BranchSelect);
31
32
         // Execute
33
         assign shifted_data = ImmData << 1;</pre>
         Adder BRANCHADDER(PC_out, shifted_data, Out2);
34
         Mux_2x1 ALUSRC(ReadData2, ImmData, ALUSrc, Data_Out);
35
36
          assign Funct = {Instruction[30],Instruction[14:12]};
         ALU Control ALUC(ALUOp, Funct, Operation);
37
38
         ALU64 ALU(ReadDatal, Data_Out, Operation, Result, Zero);
39
          // Memory
40
41
         Data_Memory DM(Result, ReadData2, clk, MemWrite, MemRead, ReadData, dm1, dm2, dm3, dm4, dm5);
42
43
          // Write Back
44
         Mux_2x1 WB(Result, ReadData, MemtoReg, WriteData);
45
46 endmodule
```

C: Codes for Task 2

Pipelined RISC-V Processor:

```
timescale lns / lps
 2 module RISC_V_Pipelined(
           input reset,
           output wire [63:0] PC in, PC out, ReadData, ReadDatal, ReadData2, WriteData, ImmData, ALU Result,
           output wire [31:0] Instruction,
           output wire [6:0] opcode, func7,
           output wire [2:0] func3,
          output wire [4:0] RS1, RS2, RD,
output wire [3:0] Operation, Funct,
10
11
          output wire [1:0] ALUOp,
           output wire RegWrite, MemRead, MemWrite, MemtoReg, ALUSrc, Zero, Branch, BranchSelect, // PCSrc
          output wire [63:0] dml, dm2, dm3, dm4, dm5,
14
          output wire [63:0] regl, reg2, reg3);
15
17
          wire [63:0] shifted_data, Data_Out, Outl, Adder_out;
          //wire [63:0] dm1, dm2, dm3, dm4, dm5;
19
           // IF_ID wires
21
          wire [63:0] IF_ID_PC_out;
22
          wire [31:0] IF_ID_Instruction;
23
25
          wire ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_Branch, ID_EX_ALUSrc;
26
          wire [1:0] ID EX ALUOp;
27
          wire [63:0] ID EX PC out, ID EX ReadDatal, ID EX ReadData2, ID EX ImmData;
28
          wire [3:0] ID_EX_Funct;
29
           wire [4:0] ID_EX_RS1, ID_EX_RS2, ID_EX_RD;
30
31
32
33
         wire EX_MEM_RegWrite, EX_MEM_MemRead, EX_MEM_MemtoReg, EX_MEM_MemWrite, EX_MEM_Branch, EX_MEM_Zero; wire [4:0] EX_MEM_RD;
34
35
          wire [63:0] EX_MEM_Adder_out, EX_MEM_ALU_Result, EX_MEM_ReadData2;
36
37
         wire MEM WB RegWrite, MEM_WB_MemtoReg;
          wire [4:0] MEM_WB_RD;
39
         wire [63:0] MEM_WB_ReadData, MEM_WB_ALU_Result;
41
42
         Adder FOURADDER(64'd4, PC_out, Out1);
43
44
         Mux_2xl BRANCH(Outl, EX_MEM_Adder_out, (EX_MEM_Branch & EX_MEM_Zero), PC_in);
Program_Counter PC(clk, reset, PC_in, PC_out);
45
          Instruction_Memory_Pipelined IMP(PC_out, Instruction);
46
47
         IF_ID Pipelinel(clk, reset, PC_out, Instruction, IF_ID_PC_out, IF_ID_Instruction);
48
          Assign Funct = {IF_ID_Instruction[30],IF_ID_Instruction[14:12]};
Instruction_Parser IP(IF_ID_Instruction, opcode, RD, func3, RS1, RS2, func7);
50
51
52
          Imm_Gen IG(IF_ID_Instruction, ImmData);
53
          Control_Unit CU(opcode, ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite);
54
55
          register_file_pipelined RF(WriteData, RS1, RS2, MEM_WB_RD, MEM_WB_RegWrite, clk, reset, ReadData1, ReadData2,reg1,reg2,reg3);
         Branch_Unit BU(func3, ReadData1, ReadData2, BranchSelect);
         ID EX Pipeline2(clk, reset,
                  RegWrite, MemRead, MemtoReg, MemWrite, Branch, ALUOp, ALUSrc, // control signals
59
                  IF_ID_PC_out, ReadData1, ReadData2, ImmData, RS1, RS2, RD, Funct, // inputs
ID EX RegWrite, ID EX MemRead, ID EX MemtoReg, ID EX MemWrite, ID EX Branch, ID EX ALUOp, ID EX ALUSrc, // output control signals
```

```
ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_Branch, ID_EX_ALUOp, ID_EX_ALUSrc, // output control signals ID_EX_PC_out, ID_EX_ReadDatal, ID_EX_ReadData2, ID_EX_ImmData, ID_EX_RS1, ID_EX_RS2, ID_EX_RD, ID_EX_Funct); // outputs
62
63
             // EXE Stage
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
80
81
82
83
84
85
86
87
88
             assign shifted_data = ID_EX_ImmData << 1;
Adder BRANCHADDER(ID_EX_PC_out, shifted_data, Adder_out);</pre>
             Mux_2x1 MuxALUSrc(ID_EX_ReadData2, ID_EX_ImmData, ID_EX_ALUSrc, Data_Out);
ALU_Control ALUC(ID_EX_ALUOp, ID_EX_Funct, Operation);
             ALU64 ALU(ID_EX_ReadDatal, Data_Out, Operation, ALU_Result, Zero);
             EX_MEM Pipeline3(clk, reset,
                        ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemtoReg, ID_EX_MemWrite, ID_EX_Branch, // input control signals
                        Adder_out, ALU Result, BranchSelect, ID_EX_ReadData2, ID_EX_RD, // inputs

EX_MEM_RegWrite, EX_MEM_MemRead, EX_MEM_MemtoReg, EX_MEM_MemWrite, EX_MEM_Branch, // output control signals
                        EX_MEM_Adder_out, EX_MEM_ALU_Result, EX_MEM_Zero, EX_MEM_ReadData2, EX_MEM_RD // outputs
             // MEM Stage
             Data_Memory DM(EX_MEM_ALU_Result, EX_MEM_ReadData2, clk, EX_MEM_MemWrite, EX_MEM_MemRead, ReadData,
             dm1, dm2, dm3, dm4, dm5);
            MEM WB Pipeline4(clk, reset,
                        EX_MEM_RegWrite, EX_MEM_MemtoReg, // input control signals
                        ReadData, EX_MEM_ALU_Result, EX_MEM_RD, // inputs
MEM_WB_RegWrite, MEM_WB_MemtoReg, // output control signals
                        MEM_WB_ReadData, MEM_WB_ALU_Result, MEM_WB_RD // outputs
             Mux 2x1 WB(MEM WB ALU Result, MEM WB ReadData, MEM WB MemtoReg, WriteData);
```

Pipelined Register File:

```
1 'timescale lns / lps
 2 module register_file_pipelined(
        input clk, reset, RegWrite,
         input [63:0] WriteData,
 5
        input [4:0] RS1, RS2, RD,
 6
         output reg [63:0] ReadDatal, ReadData2,
        output [63:0] regl, reg2, reg3);
 8
 9
10    reg [63:0] Registers [31:0];
11    integer k;
12  initial begin
13 🗀
        for (k = 0 ; k < 31 ; k = k + 1)
14 🗇
             Registers[k] = 0;
             Registers[10] = 64'd1;
15
             Registers[11] = 64'd5;
16
             Registers[13] = 64'd10;
17
18 🖨 end
19  assign regl = Registers[10];
20 assign reg2 = Registers[11];
21 assign reg3 = Registers[13];
22
23 \stackrel{\cdot}{\bigcirc} always @(negedge clk) begin
24 🗇
       if (RegWrite) begin //writing data to memory
             Registers[RD] <= WriteData;</pre>
25 '
26 🖨
       end
27 🖨 end
28
29
30 🖯 always @(*) begin
30 🖂 always @(*) begin
31 🖨
         if (reset)begin //resseting
32
               ReadData1 = 0;
33
               ReadData2 = 0;
34 🗀
          end
           else begin //reading datafrom mem into rs1 read data 1 and 2
35 🖨
36
               ReadDatal <= Registers[RS1];
37
               ReadData2 <= Registers[RS2];
38 🖨
          end
39 @ end
40 endmodule
```

IF/ID:

```
`timescale lns / lps
3 ⊝ module IF_ID(
4
        input clk, reset,
5
        input [31:0] Instruction,
 6
        input [63:0] PC out,
 7
        output reg [31:0] IF_ID_Instruction,
 8
        output reg [63:0] IF_ID_PC_out);
10 🖨 always @ (posedge clk)
11 🖯 begin
12 🖯
        case (reset)
13 🖨
            1'b1:
14 🖯
                 begin //resseting
15
                 IF_ID_Instruction <= 0;</pre>
16
                IF_ID_PC_out <= 0;</pre>
17 🖒
                end
18 🖨
             1'b0:
19 🖯
                begin //taking the instruction and giving it to IF ID
20
                 IF_ID_Instruction <= Instruction;</pre>
21
                 IF_ID_PC_out <= PC_out;</pre>
22 🖒
23 🗀
         endcase
24 🖨 end
25 endmodule
```

ID/EX:

```
`timescale lns / lps
2 module ID_EX (
         input clk, reset, RegWrite, MemRead, MemToReg, MemWrite, Branch, ALUSrc,
         input [1:0] ALUOp,
         input [63:0] IF_ID_PC_out, ReadData1, ReadData2, ImmData,
         input [3:0] Funct,
         input [4:0] RS1, RS2, RD,
         output reg ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemToReg, ID_EX_MemWrite, ID_EX_Branch, ID EX ALUSrc,
 8
         output reg [1:0] ID_EX_ALUOp,
9
10
         output reg [63:0] ID_EX_PC_out, ID_EX_ReadData1, ID_EX_ReadData2, ID_EX_ImmData,
11
         output reg [3:0] ID_EX_Funct,
12
         output reg [4:0] ID_EX_RS1, ID_EX_RS2, ID_EX_RD );
13
14
15 🖨 always @(posedge clk) begin
16 🖯
       case (reset)
17 🖨
            1'b1:
18 🖯
             begin //resseting
19
                ID_EX_RegWrite <= 0;</pre>
20
                 ID_EX_MemRead <= 0;</pre>
21
                 ID EX MemToReg <= 0;
                 ID EX MemWrite <= 0;
22
23
                 ID_EX_Branch <= 0;</pre>
24
                 ID_EX_ALUSrc <= 0;
25
                 ID_EX_ALUOp <= 0;</pre>
                 ID_EX_PC_out <= 0;</pre>
27
                 ID_EX_ReadData1 <= 0;</pre>
28
                 ID EX ReadData2 <= 0;
                 ID_EX_ImmData <= 0;</pre>
29
                 ID EX Funct <= 0;
30 :
```

```
31 ;
                  ID_EX_RS1 <= 0;
32
                  ID EX RS2 <= 0;
33
                  ID EX RD <= 0;
34 🖨
              end
35
36 ⊡
             1'b0:
37 🖨
              begin //decoding the instruction and giving it to ID EX
38 !
                  ID EX RegWrite <= RegWrite;
                  ID_EX_MemRead <= MemRead;</pre>
39
40
                 ID EX MemToReg <= MemToReg;
41
                 ID EX MemWrite <= MemWrite;
42
                 ID EX Branch <= Branch;
43
                  ID_EX_ALUSrc <= ALUSrc;</pre>
                  ID EX ALUOp <= ALUOp;
45
                  ID_EX_PC_out <= IF_ID_PC_out;</pre>
46
                 ID_EX_ReadData1 <= ReadData1;</pre>
47
                 ID EX ReadData2 <= ReadData2;</pre>
48
                 ID EX ImmData <= ImmData;
49
                 ID EX Funct <= Funct;
50
                  ID_EX_RS1 <= RS1;</pre>
51
                  ID_EX_RS2 <= RS2;
52
                  ID_EX_RD <= RD;</pre>
53 🖒
              end
54 🖨
         endcase
56 🖨 end
57 @ endmodule
```

EX/MEM:

```
`timescale lns / lps
2 nodule EX_MEM(
          input clk, reset, ID_EX_RegWrite, ID_EX_MemRead, ID_EX_MemToReg, ID_EX_MemWrite, ID_EX_Branch, Zero,
          input [4:0] ID_EX_RD,
          input [63:0] Adder_out, ALU_Result, ID_EX_ReadData2,
          output reg EX_MEM_RegWrite, EX_MEM_MemRead, EX_MEM_MemToReg, EX_MEM_MemWrite, EX_MEM_Branch, EX_MEM_Zero,
          output reg [4:0] EX MEM RD,
         output reg [63:0] EX_MEM_Adder_out, EX_MEM_ALU_Result, EX_MEM_ReadData2);
10 \buildrel \ominus always @(posedge clk) begin
11 🗇
        case (reset)
12 🖯
             1'b1:
13 🖨
             begin //resseting
14
                 EX_MEM_RegWrite <= 0;</pre>
                  EX_MEM_MemRead <= 0;</pre>
15
                  EX MEM MemToReg <= 0;
16
17
                 EX_MEM_MemWrite <= 0;
                 EX_MEM_Branch <= 0;</pre>
18
19
                 EX_MEM_Zero <= 0;</pre>
20
                  EX MEM RD <= 0;
21
                  EX_MEM_Adder_out <= 0;</pre>
22
                  EX_MEM_ALU_Result <= 0;</pre>
23
                  EX_MEM_ReadData2 <= 0;</pre>
25 🖯
             1'b0:
             begin //takes data from ID EX stage and gives it to EX MEM stage
26 🖯
27
                 EX_MEM_RegWrite <= ID_EX_RegWrite;</pre>
                  EX_MEM_MemRead <= ID_EX_MemRead;</pre>
28
29
                  EX_MEM_MemToReg <= ID_EX_MemToReg;</pre>
                  EX MEM MemWrite <= ID EX MemWrite;
30 :
```

```
31
                EX MEM Branch <= ID EX Branch;
32
                 EX MEM Zero <= Zero;
33
                 EX MEM RD <= ID EX RD;
34
                 EX MEM Adder out <= Adder out;
35
                 EX MEM ALU Result <= ALU Result;
36
                 EX_MEM_ReadData2 <= ID_EX_ReadData2;</pre>
37 🖨
            end
38 🖨
       endcase
39
40 🗀 end
41 endmodule
```

MEM/WB:

```
`timescale lns / lps
 2 module MEM WB(
        input clk, reset, EX_MEM_RegWrite, EX_MEM_MemToReg,
        input [4:0] EX_MEM_RD,
        input [63:0] ReadData, EX MEM ALU Result,
       output reg MEM WB RegWrite, MEM WB MemToReg,
       output reg [4:0] MEM WB RD,
 8
        output reg [63:0] MEM_WB_ReadData, MEM_WB_ALU_Result);
10 \bigcirc always @(posedge clk) begin
11 (case (reset)
12 🖯
             1'b1:
13 🖨
                 begin //resseting
14
                    MEM_WB_RegWrite <= 0;</pre>
15
                    MEM_WB_MemToReg <= 0;
16
                    MEM_WB_ReadData <= 0;</pre>
17
                    MEM WB ALU Result <= 0;
18
                     MEM WB RD <= 0;
19 🖨
20
21 🖯
               1'b0:
22 🖨
               begin //takes data changed from EX MEM to MEM WB
23
                    MEM WB RegWrite <= EX MEM RegWrite;
24
25
                     MEM WB MemToReg <= EX MEM MemToReg;
                     MEM WB ReadData <= ReadData;</pre>
26
                     MEM WB ALU Result <= EX MEM ALU Result;
                     MEM_WB_RD <= EX_MEM_RD;</pre>
27
28 🖨
                 end
29 🖒
        endcase
30 :
```

D: Codes for Task 3

RISC V Processor

```
23 🖨
         module RISCV PP forwarding (
24
            input clk, reset,
25
            output[63:0] PC_to_INSTMEM,
            output [31:0] INSTMEM to IF ID,
26
27
            output [63:0] element1, element2, element3, element4, element5, element6, element7);
28
29
        wire [1:0] forwardA, forwardB;
30 ⊜
         //wire [63:0] PC to INSTMEM;
         ://wire [31:0] INSTMEM to IF ID;
31 🖨
32
         wire [6:0] CONTROL IN;
33
        wire[4:0] rd;
34
        wire [2:0] funct3;
35
         wire [6:0] funct7;
36
        wire [4:0] rs1, rs2;
37
         wire Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite;
3.8
         wire Is_Greater;
39
         wire [1:0] ALUOp;
40
         wire [63:0] mux_to_reg;
41
         wire [63:0] mux to pc in;
42
         wire [3:0] ALU_C_Operation;
        wire [63:0] ReadData1, ReadData2;
43
44
        wire [63:0] imm data;
45
46
47
48
     Wire [63:0] fixed_4 = 64'd4;
49
         wire [63:0] PC_plus_4_to_mux;
50
51
         wire [63:0] alu_mux;
52
53
        wire [63:0] alu_result;
54
         wire zero;
55
56
         wire [63:0] imm_to_adder;
         wire [63:0] imm_adder_to_mux;
57
58
59
         wire [63:0] DM Read Data;
60
         wire pc_mux_sel_wire;
61
62
         wire PCWrite;
```

```
64 !
              ://IF ID WIRES
 65
               wire [63:0] IF ID PC addr;
 66
               wire [31:0] IF ID IM to parse;
 67
               wire IF ID Write;
 68
 69
               //ID EX WIRES
 70
 71
               wire ID EX Branch, ID EX MemRead, ID EX MemtoReg;
 72
               wire ID EX MemWrite, ID EX ALUSTO, ID EX RegWrite;
 73
 74
               wire [63:0] ID EX PC addr, ID EX ReadData1, ID EX ReadData2,
 75
                                ID EX imm data;
               wire [3:0] ID_EX_funct_in;
 76
 77
               wire [4:0] ID_EX_rd, ID_EX_rs1, ID_EX_rs2;
  78
               wire [1:0] ID EX ALUOp;
  79
         O assign imm_to_adder = ID_EX_imm_data<< 1;
 80
 81
 82
 83
               //EX MEM WIRES
 84
               wire EX_MEM_Branch, EX_MEM_MemRead, EX_MEM_MemtoReg;
 85
               wire EX_MEM_MemWrite, EX_MEM_RegWrite;
 86
               wire EX_MEM_zero, EX_MEM_Is_Greater;
               wire [63:0] EX_MEM_PC_plus_imm, EX_MEM_alu_result, EX_MEM_ReadData2;
 87
 88
               wire [3:0] EX MEM funct in;
 89
               wire [4:0] EX_MEM_rd;
 90
               wire NOP_Check;
 91
 92
              1//MEM WB WIRES
 93
               wire MEM WB MemtoReg, MEM WB RegWrite;
 94
              wire [63:0] MEM_WB_DM_Read_Data, MEM_WB_alu_result;
 95
              wire [4:0] MEM WB rd;
 96
 97
 98
               Mux_2x1 pcsrcmux(EX_MEM_PC_plus_imm, PC_plus_4_to_mux, pc_mux_sel_wire, mux_to_pc_in);
 99
100
               PC_for_forwarding PC (clk, reset, PCWrite, mux_to_pc_in, PC_to_INSTMEM);
101
102
               Adder pcadder (PC to INSTMEM, fixed 4, PC plus 4 to mux);
103
104
105
106
         Instruction_mem_forwarding insmem(PC_to_INSTMEM, INSTMEM_to_IF_ID);
107
108
109
        IF_ID_forwarding IFIDreg(.clk(clk), .Flush(NOP_Check), .IFID_Write(IF_ID_Write), .PC_addr(PC_to_INSTMEM),
                       .Instruc(INSTMEM_to_IF_ID), .FC_store(IF_ID_FC_addr),
Instr_store(IF_ID_IM_to_parse));
110
111
112
113
114
115
116
117
         wire control mux sel;
         Hazard_Detection hazarddetectionunit(ID_EX_rd, rs1, rs2, ID_EX_MemRead, control_mux_sel,
                                             IF ID Write, PCWrite);
        Instruction_Parser insparser(IF_ID_IM_to_parse, CONTROL_IN, rd, funct3, rs1, rs2, funct7);
         wire [3:0] funct_in;
      O assign funct_in = (IF_ID_IM_to_parse[30], IF_ID_IM_to_parse[14:12]);
125
126
127
128
129
         Control_Unit cunit(CONTROL_IN, ALUOp, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite);
        Register_File registerfiles(mux_to_reg, rs1, rs2, MEM_WB_rd, MEM_WB_RegWrite, clk, reset,
ReadData1, ReadData2);
130
         Imm Gen immgen (IF ID IM to parse, imm data);
133
134
         wire MemtoReg_ID_EXin, RegWrite_ID_EXin, Branch_ID_EXin, MemWrite_ID_EXin, MemRead_ID_EXin, ALUSrc_ID_EXin;
         //if a hazard is detected then stop taking in instructions
assign MemtoReg_ID_EXin = control_mux_sel ? MemtoReg : 0;
     assign Memtokeg ID_EXin = control_mux_sel ? RegWrite : 0;
assign Branch_ID_EXin = control_mux_sel ? Branch : 0;
assign MemWrite_ID_EXin = control_mux_sel ? MemWrite : 0;
assign MemRead_ID_EXin = control_mux_sel ? MemRead : 0;
assign ALUSrc_ID_EXin = control_mux_sel ? ALUSrc : 0;
wire [1:0] ALUOp_ID_EXin;
138
141
```

```
143
             wire [1:0] ALUop ID EXin;
144
        O assign ALUop_ID_EXin = control_mux_sel ? ALUOp : 2'b00;
145
146
147
             // ID/EX STAGE
            ID_EX_new ID_EX1(.clk(clk), .Flush(NOP_Check), .PC_addr(IF_ID_PC_addr), .read_datal(ReadDatal),
148
                            .read_data2(ReadData2), .imm_val(imm_data), .funct_in(funct_in), .rd_in(rd),
149
150
                            .rs1_in(rs1), .rs2_in(rs2), .RegWrite(RegWrite_ID_EXin),
151
                            .MemtoReg (MemtoReg_ID_EXin), .Branch (Branch_ID_EXin),
                            .MemWrite(MemWrite_ID_EXin), .MemRead(MemRead_ID_EXin), .ALUSrc(ALUSrc_ID_EXin),
152
153
                            .ALU_op(ALUop_ID_EXin), .PC_addr_store(ID_EX_PC_addr),
                            .read_data1_store(ID_EX_ReadData1), .read_data2_store(ID_EX_ReadData2),
154
155
                            .imm_val_store(ID_EX_imm_data), .funct_in_store(ID_EX_funct_in),
156
                            .rd_in_store(ID_EX_rd), .rs1_in_store(ID_EX_rs1), .rs2_in_store(ID_EX_rs2),
157
                            .RegWrite_store(ID_EX_RegWrite), .MemtoReg_store(ID_EX_MemtoReg),
158
                            .Branch_store(ID_EX_Branch), .MemWrite_store(ID_EX_MemWrite),
159
                            .MemRead_store(ID_EX_MemRead), .ALUSrc_store(ID_EX_ALUSrc),
160
                            .ALU_op_store(ID_EX_ALUOp));
161
162
             ALU_Control ALU_Control1(ID_EX_ALUOp, ID_EX_funct_in, ALU_C_Operation);
163
164
             wire [63:0] triplemux_to_a, triplemux_to_b;
165
166
             Mux_2x1 ALU_mux(ID_EX_imm_data, triplemux_to_b, ID_EX_ALUSrc, alu_mux);
167
168
169
170
             Mux 3x1 mux for a(ID EX ReadData1, mux to reg, EX MEM alu result, forwardA, triplemux to a);
171
172
            Mux_3x1 mux_for_b(ID_EX_ReadData2, mux_to_reg, EX_MEM_alu_result, forwardB, triplemux_to_b);
173
174
            ALUC4 ALU_64(triplemux_to_a, alu_mux, ALU_C_Operation, alu_result, zero, Is_Greater);
175
176
177
178
             Forwarding_Unit Fwd_unit(EX_MEM_rd, MEM_WB_rd, ID_EX_rs1, ID_EX_rs2, EX_MEM_RegWrite,
179
                                           EX MEM MemtoReg, MEM WB RegWrite, forwardA, forwardB);
180
181
182
             wire [63:0] pc_add_imm_to_EX_MEM;
183
184
          Adder PC_plus_imm(ID_EX_PC_addr, imm_to_adder, pc_add_imm_to_EX_MEM);
186
          // EX/MEM STAGE
188
189
          EX_MEM_new EX_MEM1(.clk(clk),.Flush(NOP_Check),.RegWrite(ID_EX_RegWrite),.MemtoReg(ID_EX_MemtoReg),
                         .Branch (ID EX Branch) , .Zero (zero) , .Is Greater (Is Greater) ,
190
191
                         .MemWrite(ID_EX_MemWrite),.MemRead(ID_EX_MemRead),.PCplusimm(pc_add_imm_to_EX_MEM),
.ALU_result(alu_result),.WriteData(triplemux_to_b),.funct_in(ID_EX_funct_in),
192
193
                         .rd(ID_EX_rd),.RegWrite_store(EX_MEM_RegWrite),.MemtoReg_store(EX_MEM_MemtoReg),.Branch_store(EX_MEM_Branch),.Zero_store(EX_MEM_zero),
194
195
                         .Is_Greater_store(EX_MEM_Is_Greater),.MemWrite_store(EX_MEM_MemWrite),.MemRead_store(EX_MEM_MemRead),.PCplusimm_store(EX_MEM_PC_plus_imm),
196
197
                         .ALU_result_store(EX_MEM_alu_result),.WriteData_store(EX_MEM_ReadData2),.funct_in_store(EX_MEM_funct_in),.rd_store(EX_MEM_rd));
198
201
          Branch_Control Branch_Control(.Branch(EX_MEM_Branch), .Flush(NOP_Check), .Zero(EX_MEM_zero), .Is_Greater(EX_MEM_Is_Greater),
                                        .funct(EX_MEM_funct_in),.switch_branch(pc_mux_sel_wire));
204
205
          Data_Memory dm(EX_MEM_alu_result, EX_MEM_ReadData2, clk,EX_MEM_MemWrite,EX_MEM_MemRead,
206
             DM Read Data, element1, element2, element3, element4, element5, element6, element7);
207
209
          // MEM/WB STAGE
211
          MEM WB new MEM WB1(.clk(clk), .RegWrite(EX MEM RegWrite), .MemtoReg(EX MEM MemtoReg),
213
214
                         .ReadData(DM Read Data), .ALU result(EX MEM alu result), .rd(EX MEM rd), .RegWrite_store(MEM WB RegWrite), .MemtoReg_store(MEM WB MemtoReg),
215
                         .ReadData_store(MEM_WB_DM_Read_Data), .ALU_result_store(MEM_WB_alu_result),
                         .rd store (MEM WB rd));
217
218
219
220
          Mux_2x1 mux2(MEM_WB_DM_Read_Data, MEM_WB_alu_result, MEM_WB_MemtoReg, mux_to_reg);
222 🖒
          endmodule
```

Hazard Detection Unit:

```
1 \ \  \, \neg \  \, \text{module Hazard\_Detection}
3
         input [4:0] IDEX_rd, IFID_rs1, IFID_rs2,
        input IDEX_MemRead,
        output reg IDEX_mux_out,
        output reg IFID_Write, PCWrite
9  always@(*) begin
10
        if (IDEX_MemRead && (IDEX_rd == IFID_rs1 || IDEX_rd == IFID_rs2))
12 🖨
       begin //assigning ZERO value means it is deasserted and the values for PC and Instruction won't be updated
           IDEX_mux_out = 0;
13
            IFID Write = 0:
14
15
            PCWrite = 0;
16 🖨 end
      else begin
17 👨
18
         IDEX_mux_out = 1;
           IFID_Write = 1;
20
            PCWrite = 1;
23 endmodule // Hazard Detection
```

Forwarding Unit

```
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 ReqWrite && EXMEM rd != 0)
       begin
                                                input [4:0]
          fwd A = 2'b10;
   else if (MEMWB RegWrite && MEMWB rd!=0 && MEMWB rd==IDEX rs1 )
          fwd A = 2'b01;
       end
   else
          fwd_A = 2'b00;
       end
   if ((EXMEM rd == IDEX rs2) && (EXMEM RegWrite) && (EXMEM rd != 0))
       begin
          fwd_B = 2'b10;
       end
```