CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Nathan Reff

Logan Roe

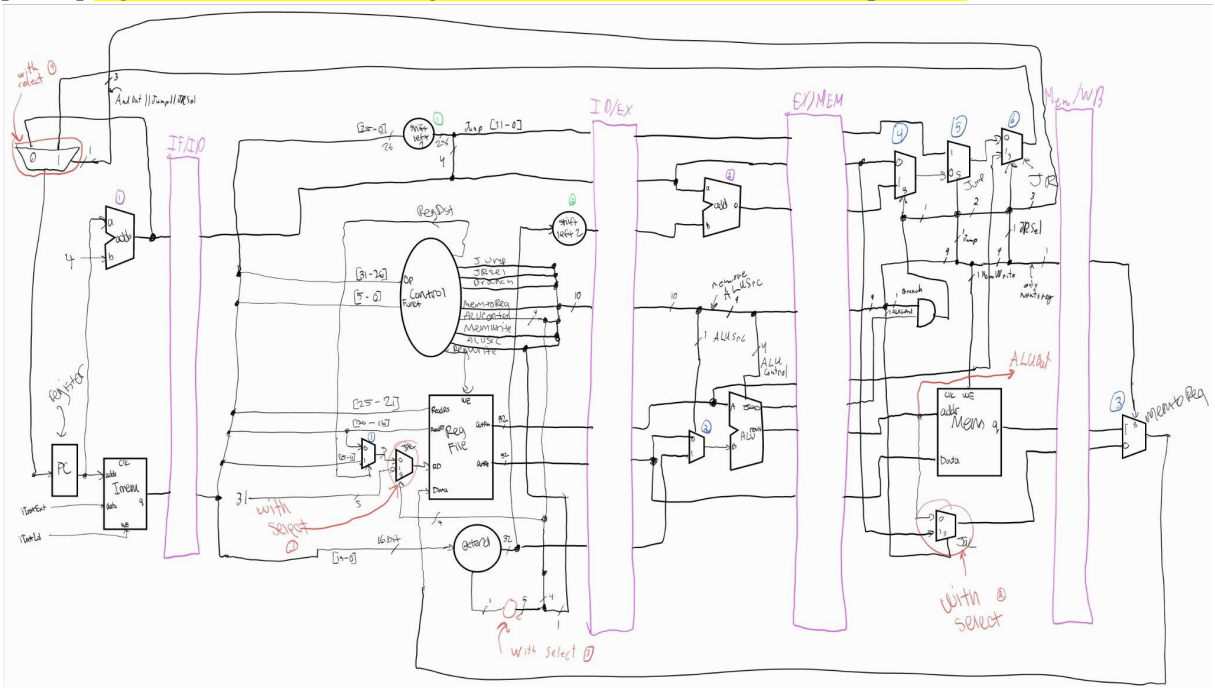
Project Teams Group #: Project2 - 5-7

Refer to the highlighted language in the project 1 instruction for the context of the following questions.

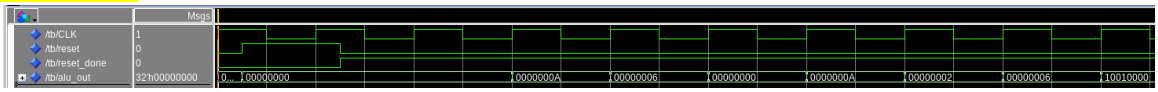
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

IF/ID Stage				ID/EX Stage				Mem/WB Stage					Ex/MEM Stage			
Inputs		Outputs		Inputs		Outputs		Inputs		Outputs			Inputs		Outputs	
Name	Bit-Width	Name	Bit-Width	Name	Bit-Width	Name	Bit-Width	Name	Bit-Width	Name	Bit-Width	Notes	Name	Bit-Width	Name	Bit-Width
NextAddr (PC+4)	32	NextAddr (PC+4)	32	JumpAddr	32	JumpAddr	32	JumpAddr	32	JumpAddr	32		memtoReg	1	memtoReg	1
InstrAddr	32	InstrAddr	32	NextAddr (PC+4)	32	NextAddr (PC+4)	32	NextAddr (PC+4	32	NextAddr (PC+4	32		MemOut	32	MemOut	32
				BranchImm	32	BranchImm	32	BranchAddr	32	BranchAddr	32		JalOut	32	JalOut	32
				controlBus	11	controlBus	11	controlBus	10	controlBus	10	Drop ALUSrc				
				RegfileRS	32	RegfileRS	32	RegfileRS	32	RegfileRS	32					
				RegfileRT	32	RegfileRT	32	RegfileRT	32	RegfileRT	32					
				ImmExt	32	ImmExt	32	ALUZero	1	ALUZero	1					
				Shamt	5	Shamt	5	ALUResult	32	ALUResult	32					

[1.b.ii] high-level schematic drawing of the interconnection between components.

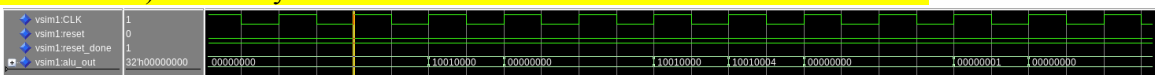


[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

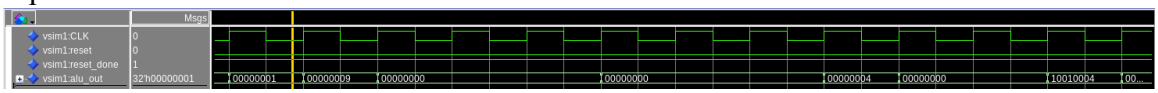


The above waveform is a snip from the start of the waveform output of the base test. This includes the following instructions, in order: addi, addi, nop, addu, and, andi, lui. The outputs of these instructions are as follows: 10, 6, 0, 10, 2, 6, 0x10010000. Given this, it can be seen from the waveform, that the processor performs the instructions as intended. Furthermore, the rest of the waveform, not shown here, shows that the entire program works as intended including at least one example of each instruction, including branching and jumping as well.

[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.



In the above waveform, the first iteration of the "loop" portion of the bubblesort program can be seen working properly. From 400 ns to 600 ns in the waveform, as seen in the snip above, shows this first iteration. The final result is a branch to "increment" which is expected.



In the above waveform, this shows the area from 700 ns to around 940 ns which is the first iteration of the "increment" portion of the program and then the start of the second iteration of the "loop" portion of the program. It was expected to branch on the first bne in increment in this iteration so that worked as intended. As can be seen near the end of the waveform, it did branch and started executing the instructions in "loop" again.

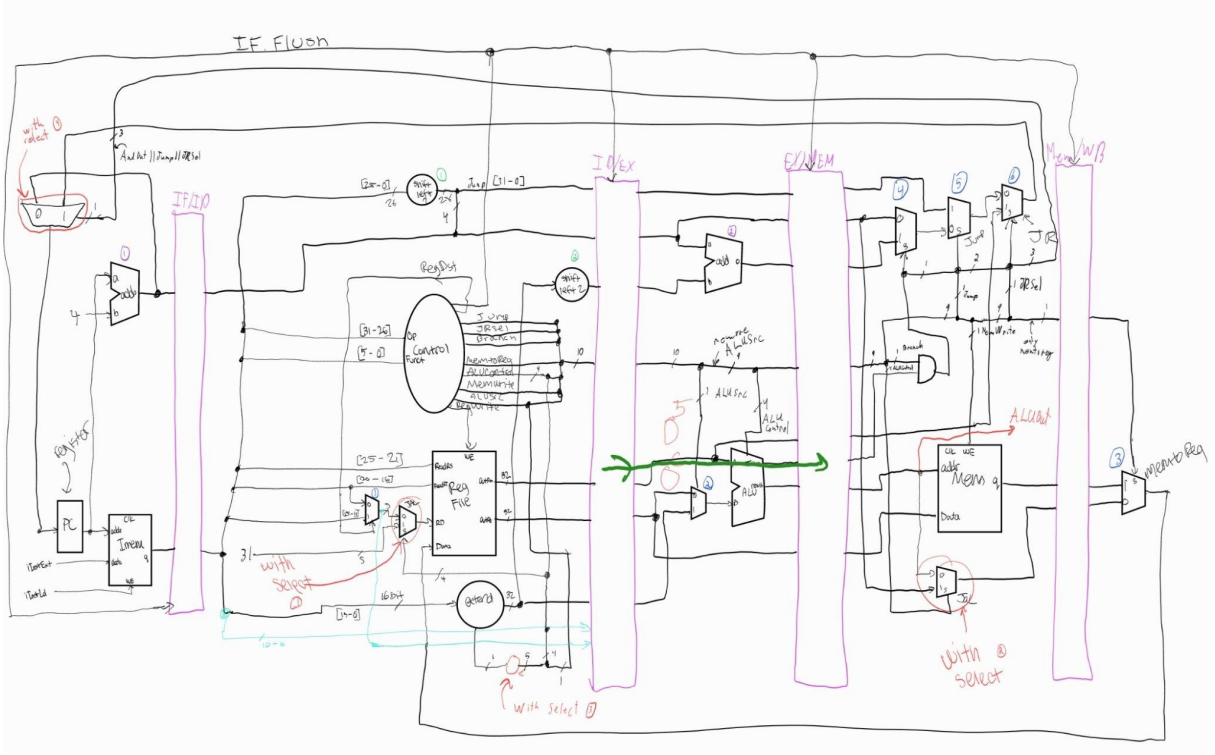
```
lw $t0, 0($t7)                          #load numbers[j]
lw $t1, 4($t7)                          #load numbers[j+1]

nop
nop

slt $t2, $t0, $t1                       #if t0 < t1

nop
nop

bne $t2, $zero, increment
nop
nop
nop

sw $t1, 0($t7)                          #swap
```

As can be seen in the above code, only two nops work used after a lw despite a dependency in the next instruction, slt, which is three less than the maximum number of five. Next, a set less than can be seen with two nops right after it despite a dependency in bne, the next instruction. For the control-flow one, bne can be seen with only three nops, since we branch from the memory stage, instead of the maximum number of five nops.

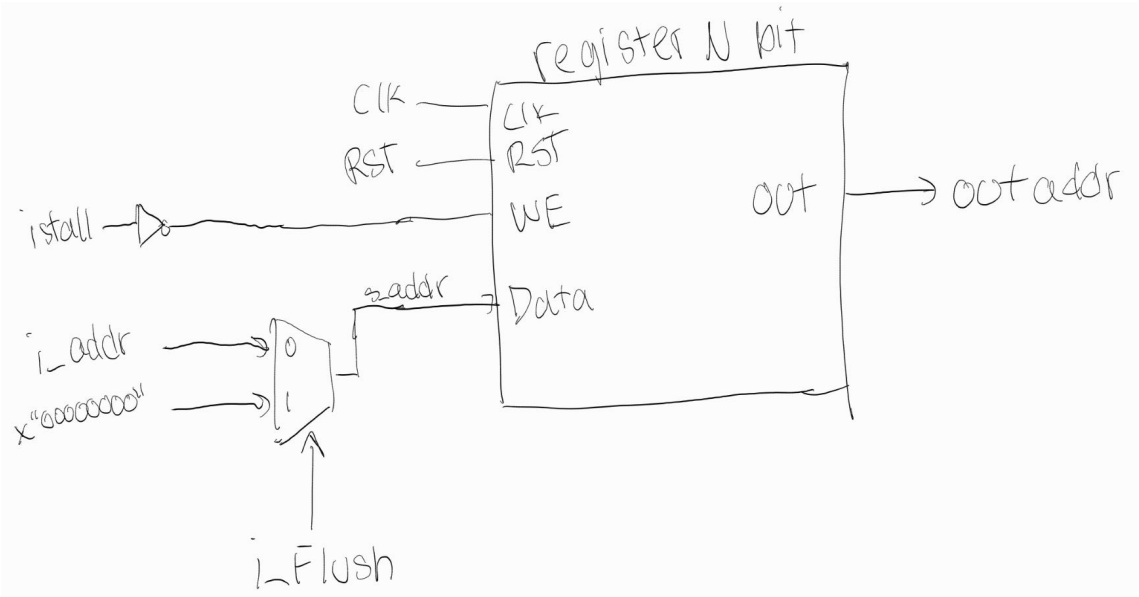
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

Maximum Frequency: 50.80 mhz

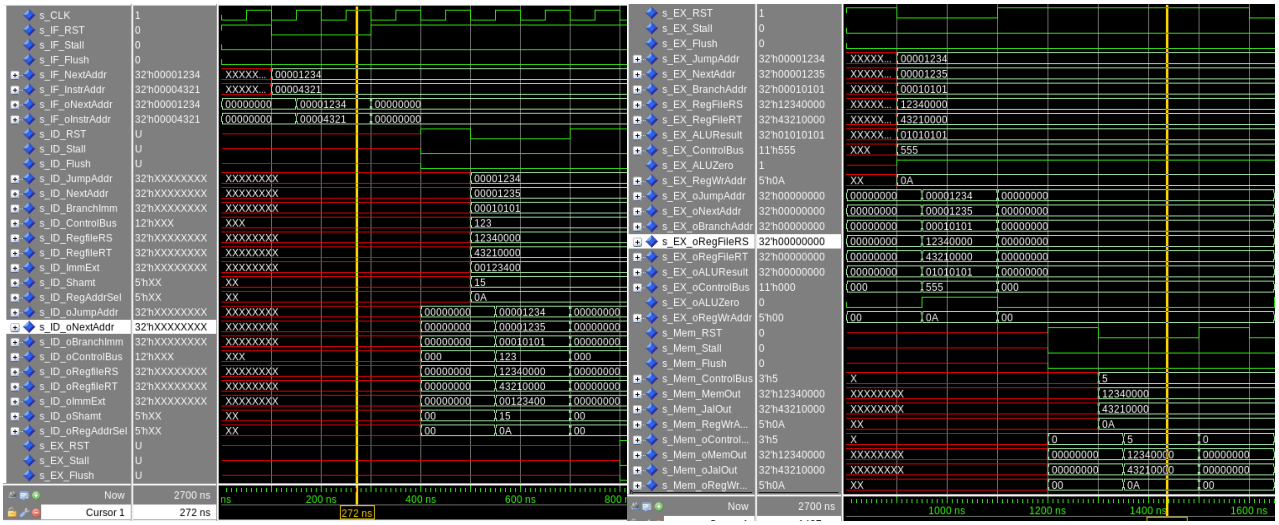
Critical Path: ID_EXRegister → ALU → EX_MemRegister



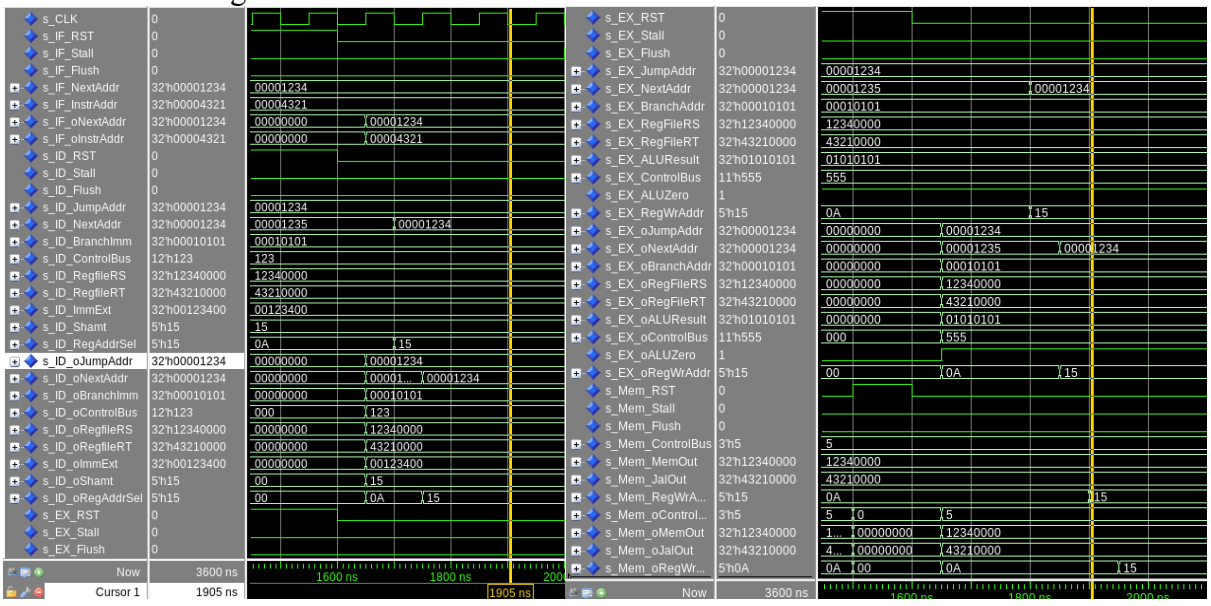
[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.



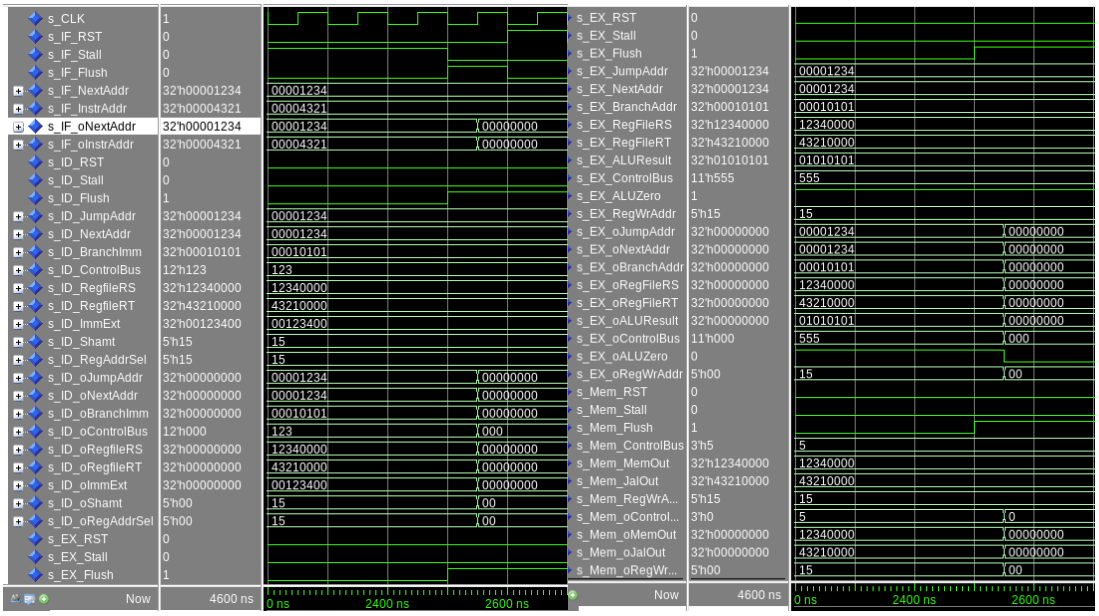
[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.



In the two above waveforms, the basic functionality of the state registers can be seen as they can store values and reset them as expected from any sort of register. The left picture shows IF/ID and ID/EX state registers and the right picture shows EX/Mem and Mem/WB state registers



In the above waveforms, it can be seen that data can be passed between the state registers and it takes the expected number of cycles to go from one state register to another state register. For example, it takes 4 cycles to go from the beginning to the end. The left waveform shows the IF/ID and ID/EX state registers and the right picture shows EX/Mem and Mem/WB state registers.



In the above waveforms, it can be seen that flushing synchronously works. It writes 0's to all portions of the state registers once it hits a positive edge on the clock cycle. The left waveform shows the IF/ID and ID/EX state registers and the right picture shows EX/Mem and Mem/WB state registers.

[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.

	A	B	C	D
1		Stages		
2	Instruction	Execute	Memory	Writeback
3	and	s_ALUResult	s_oALUResult	s_WithSel2
4	andi	s_ALUResult	s_oALUResult	s_WithSel2
5	or	s_ALUResult	s_oALUResult	s_WithSel2
6	ori	s_ALUResult	s_oALUResult	s_WithSel2
7	xor	s_ALUResult	s_oALUResult	s_WithSel2
8	xori	s_ALUResult	s_oALUResult	s_WithSel2
9	nor	s_ALUResult	s_oALUResult	s_WithSel2
10	add	s_ALUResult	s_oALUResult	s_WithSel2
11	addi	s_ALUResult	s_oALUResult	s_WithSel2
12	sw	s_ALUResult	s_oALUResult	s_WithSel2
13	addu	s_ALUResult	s_oALUResult	s_WithSel2
14	addiu	s_ALUResult	s_oALUResult	s_WithSel2
15	sub	s_ALUResult	s_oALUResult	s_WithSel2
16	subu	s_ALUResult	s_oALUResult	s_WithSel2
17	slt	s_ALUResult	s_oALUResult	s_WithSel2
18	slti	s_ALUResult	s_oALUResult	s_WithSel2
19	sll	s_ALUResult	s_oALUResult	s_WithSel2
20	srl	s_ALUResult	s_oALUResult	s_WithSel2
21	sra	s_ALUResult	s_oALUResult	s_WithSel2
22	lui	s_ALUResult	s_oALUResult	s_WithSel2
23	lw	s_ALUResult	s_DMemOut	s_RegWrData
24	beq	s_Zero	s_Mux6	----
25	bne	s_Zero	s_Mux6	----

[2.b.ii] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

	A	B	C	D
1		Stages		
2		Consumption		
3	Instruction	Decode	Execute	Memory
4	and	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
5	andi	s_OutRS, s_ExtOut	s_WithSel5, s_Mux2	----
6	or	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
7	ori	s_OutRS, s_ExtOut	s_WithSel5, s_Mux2	----
8	xor	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
9	xori	s_OutRS, s_ExtOut	s_WithSel5, s_Mux2	----
10	nor	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
11	add	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
12	addi	s_OutRS, s_ExtOut	s_WithSel5, s_Mux2	----
13	sw	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	s_DMemAddr, s_DMemData, s_DMemWr
14	addu	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
15	addiu	s_OutRS, s_ExtOut	s_WithSel5, s_Mux2	----
16	sub	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
17	subu	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
18	slt	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
19	slti	s_OutRS, s_ExtOut	s_WithSel5, s_Mux2	----
20	sll	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
21	srl	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
22	sra	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
23	lui	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	----
24	lw	s_OutRS, s_OutRT	s_WithSel5, s_Mux2	s_DMemAddr, s_DMemData, s_DMemWr
25	beq	s_OutRS, s_OutRT	s_WithSel5, s_Mux2, s_ooAdder1, s_oShiftLeft2	s_oAdder2, s_ooJumpAddress, s_JRForwardAOutput
26	bne	s_OutRS, s_OutRT	s_WithSel5, s_Mux2, s_ooAdder1, s_oShiftLeft2	s_oAdder2, s_ooJumpAddress, s_JRForwardAOutput

[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

	A	B	C
1		Forwarding	
2	Instruction	Receiving	Forwarding
3	and	Execute	Memory, Writeback
4	andi	Execute	Memory, Writeback
5	or	Execute	Memory, Writeback
6	ori	Execute	Memory, Writeback
7	xor	Execute	Memory, Writeback
8	xori	Execute	Memory, Writeback
9	nor	Execute	Memory, Writeback
10	add	Execute	Memory, Writeback
11	addi	Execute	Memory, Writeback
12	sw	Execute	Memory, Writeback
13	addu	Execute	Memory, Writeback
14	addiu	Execute	Memory, Writeback
15	sub	Execute	Memory, Writeback
16	subu	Execute	Memory, Writeback
17	slt	Execute	Memory, Writeback
18	slti	Execute	Memory, Writeback
19	sll	Execute	Memory, Writeback
20	srl	Execute	Memory, Writeback
21	sra	Execute	Memory, Writeback
22	lui	Execute	Memory, Writeback
23	lw	Execute	Writeback
24	beq	Execute	----
25	bne	Execute	----

[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	IF/ID Stage				ID/EX Stage				EX/Mem Stage					Mem/WB Stage			
2	Inputs		Outputs		Inputs		Outputs		Inputs		Outputs		Notes	Inputs		Outputs	
3	Name	Bit-Width	Name	Bit-Width	Name	Bit-Width	Name	Bit-Width	Name	Bit-Width	Name	Bit-Width		Name	Bit-Width	Name	Bit-Width
4	NextAddr (PC+4)	32	NextAddr (PC+4)	32	JumpAddr	32	JumpAddr	32	JumpAddr	32	JumpAddr	32		ControlBus	3	ControlBus	3
5	InstrAddr	32	InstrAddr	32	NextAddr (PC+4)	32	NextAddr (PC+4)	32	NextAddr (PC+4)	32	NextAddr (PC+4)	32		MemOut	32	MemOut	32
6	Stall	1			BranchImm	32	BranchImm	32	BranchAddr	32	BranchAddr	32		JalOut	32	JalOut	32
7	Flush	1			controlBus	12	controlBus	12	controlBus	11	controlBus	11	Drop ALUSrc	RegWrAddr	5	RegWrAddr	5
8					RegfileRS	32	RegfileRS	32	RegfileRS	32	RegfileRS	32		Ovfl	1	Ovfl	1
9					RegfileRT	32	RegfileRT	32	RegfileRT	32	RegfileRT	32		Stall	1		
10					ImmExt	32	ImmExt	32	ALUZero	1	ALUZero	1		Flush	1		
11					Shamt	5	Shamt	5	ALUResult	32	ALUResult	32					
12					RegAddrSel	5	RegAddrSel	5	RegLocRS	5	RegLocRS	5					
13					RegLocRS	5	RegLocRS	5	RegWrAddr	5	RegWrAddr	5					
14					RegLocRT	5	RegLocRT	5	Stall	1							
15					Stall	1			Flush	1							
16					Flush	1											

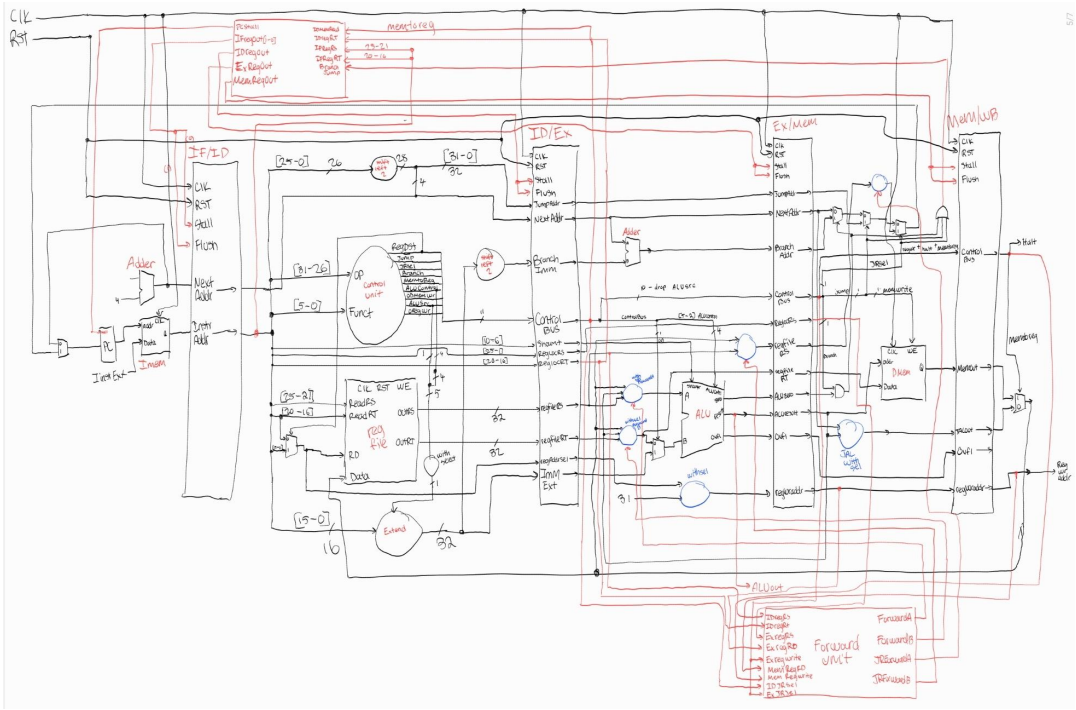
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

	A	B
1	Instructions	Stage
2	beq	Memory
3	bne	Memory
4	j	Memory
5	jr	Memory
6	jal	Memory

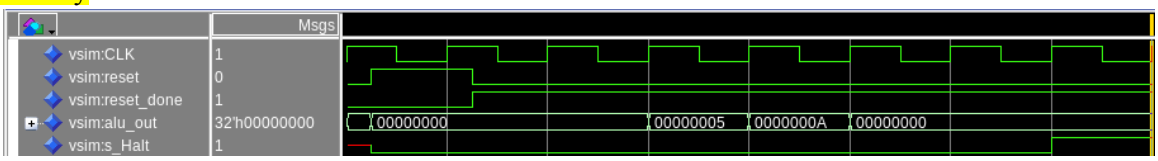
[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.

	A	B	C	D
1	Instructions	Stage	Stages To Stall	Stages To Flush
2	beq	Memory	---	IF/ID, ID/EX, EX/Mem
3	bne	Memory	---	IF/ID, ID/EX, EX/Mem
4	j	Memory	---	IF/ID, ID/EX, EX/Mem
5	jr	Memory	---	IF/ID, ID/EX, EX/Mem
6	jal	Memory	---	IF/ID, ID/EX, EX/Mem

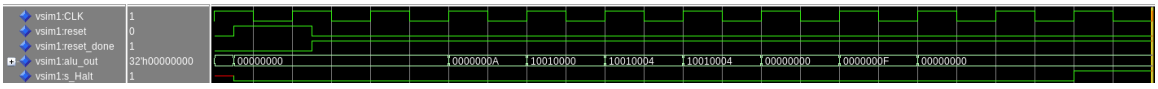
[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.



[2.e – i, ii, and iii] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.



The above waveform shows the data forwarding test which shows that data forwarding works. As can be seen, cycle one and two do not have an ALU output as the pipeline is being filled with the instructions. Cycles 3 and 4 show that the ALU gets the result from the previous instruction and uses it immediately, with no delays being added.



The above waveform shows the data hazard testing working as intended. Cycle 7 shows a stall happening after the lw, but before the addi so that the addi can have the new $t1 value, which is intended. Cycle 8 shows the addi, using the previously obtained $t1 from the lw instruction, outputting the expected 15 value, which would only happen if the $t1 had the correct value in it.

The above waveform shows that control hazard detection works as intended. Cycles 6-8 show the state registers getting flushed due to the beq branch being taken and our branching being done in the memory stage. Cycle 9 shows the addi, which places 2 into register $t2, running so it clearly branched properly. Then, the jump occurs and the state registers are flushed, then halt occurs as expected.

[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

- Data Forwarding Test Cases
  - Any non-J format and non-branch instruction back to back with dependencies

```
1  addi $t0, $zero, 5
2  addi $t1, $t0, 5
3  halt
```

- Data Hazard Test Cases
  - Load word as it takes an extra stage thus requiring one stall

```
1  addi $t0, $zero, 10
2  lui $s0, 0x1001
3  sw $t0, 4($s0)
4  lw $t1, 4($s0)
5  addi $t2, $t1, 5
6  halt
```

[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

- Control Hazard Test Cases
  - When a branch is taken or a jump occurs

```
1  addi $t0, $zero, 5
2  addi $t1, $zero, 5
3  beq $t0, $t1, test
4  end:
5          halt
6  test:
7          addi $t2, $zero, 2
8          j end
```

[2.f] report the maximum frequency your hardware-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through

Maximum Frequency: 47.44 mhz

Critical Path: DMem → MemToReg MUX → Imm Sel MUX → ALU → EX_MemReg