CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Gurumanie Singh Dhiman
Dawud Benedict

Project Teams Group #: C_01

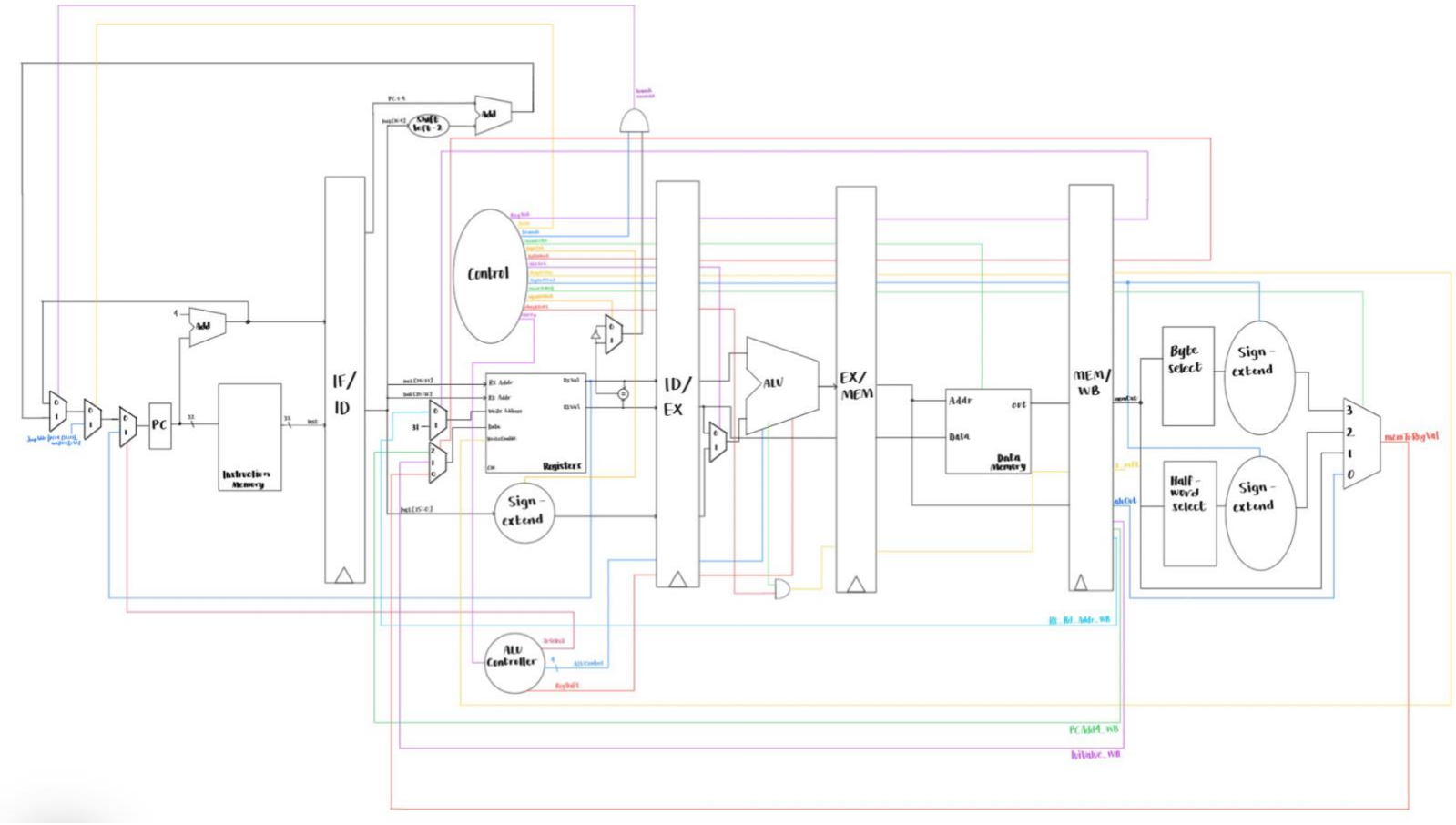
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

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

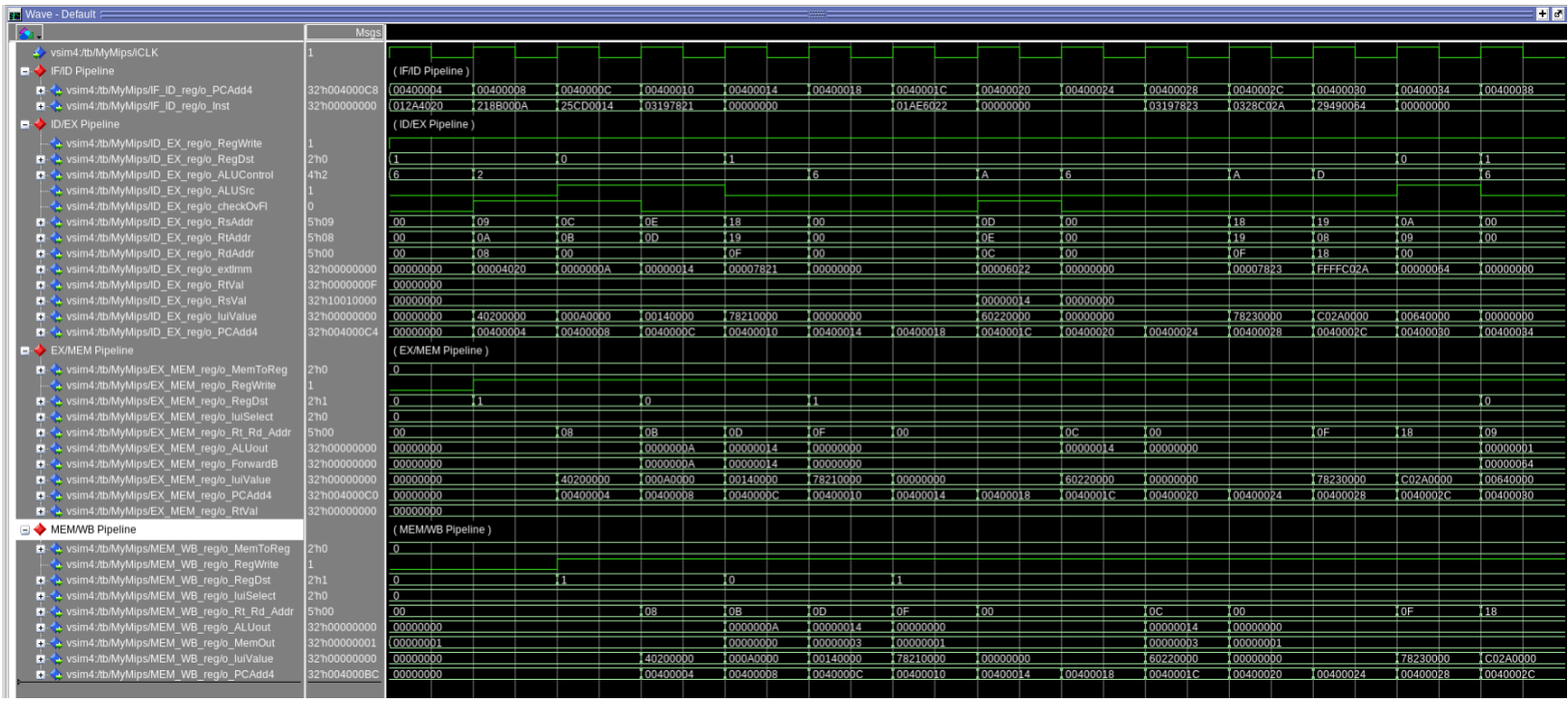
Pipeline Inputs			
IF/ID	ID/EX	EX/MEM	MEM/WB
PC + 4	PC + 4	PC + 4	PC + 4
Instruction	MemToReg	MemToReg	MemToReg
	RegWrite	RegWrite	RegWrite
	RegDst	RegDst	RegDst
	LuiSelect	LuiSelect	LuiSelect
	s_Halt	s_Halt	s_Halt
	SignedLoad	SignedLoad	SignedLoad
	DMemWr	DMemWr	MemOut
	ALUControl	ALUOut	ALUOut
	ALUSrc	Rd/Rt Addr	Rd/Rt Addr
	CheckOvFl	luiVal	luiVal
	RegShift	RtVal	RtVal
	RsAddr		
	RtAddr		
	SignExtImmd		
	RdAddr		
	RtVal		
	RsVal		
	luiVal		

	Used in WB
	Used in MEM
	Used in EX

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

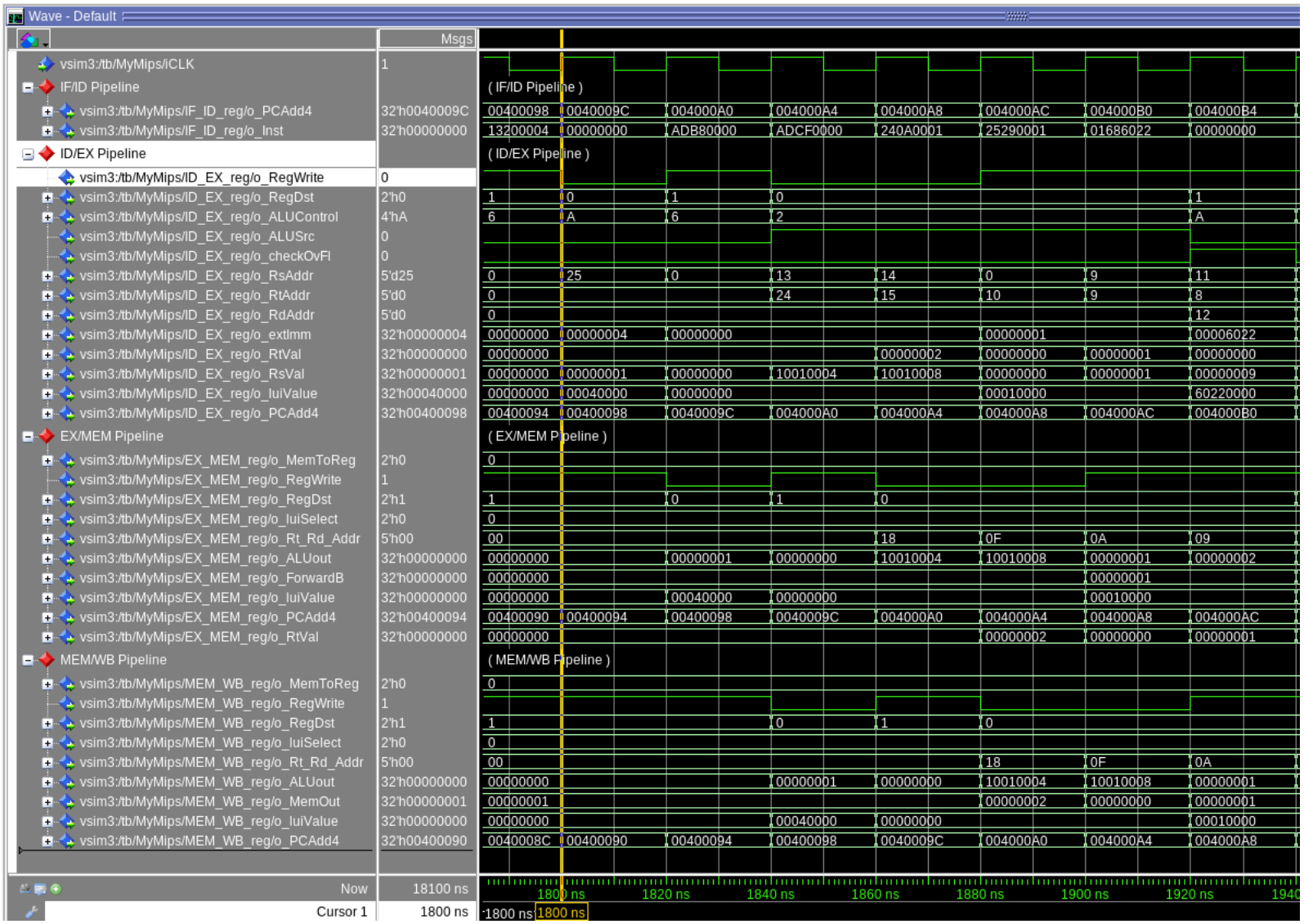


[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.



We can see that the instruction exiting IF/ID at the beginning of the screenshot corresponds to the machine code for the add instruction. On the next cycle, the add instruction propagates through the next stage and makes way for the addi instruction that is now exiting the IF/ID stage of the pipeline. In the next cycle, the add is finally at MEM/WB, and we can see that register $t0 (register 8 in waveform) is being written to during this cycle. The following cycle shows the same, but for the addi instruction that writes to $t3 instead, which is shown in the waveform as register 11 (hex 0B). The same applies to all the other instructions propagating through the pipeline.

[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) where you did not have to use the maximum number of NOPs.



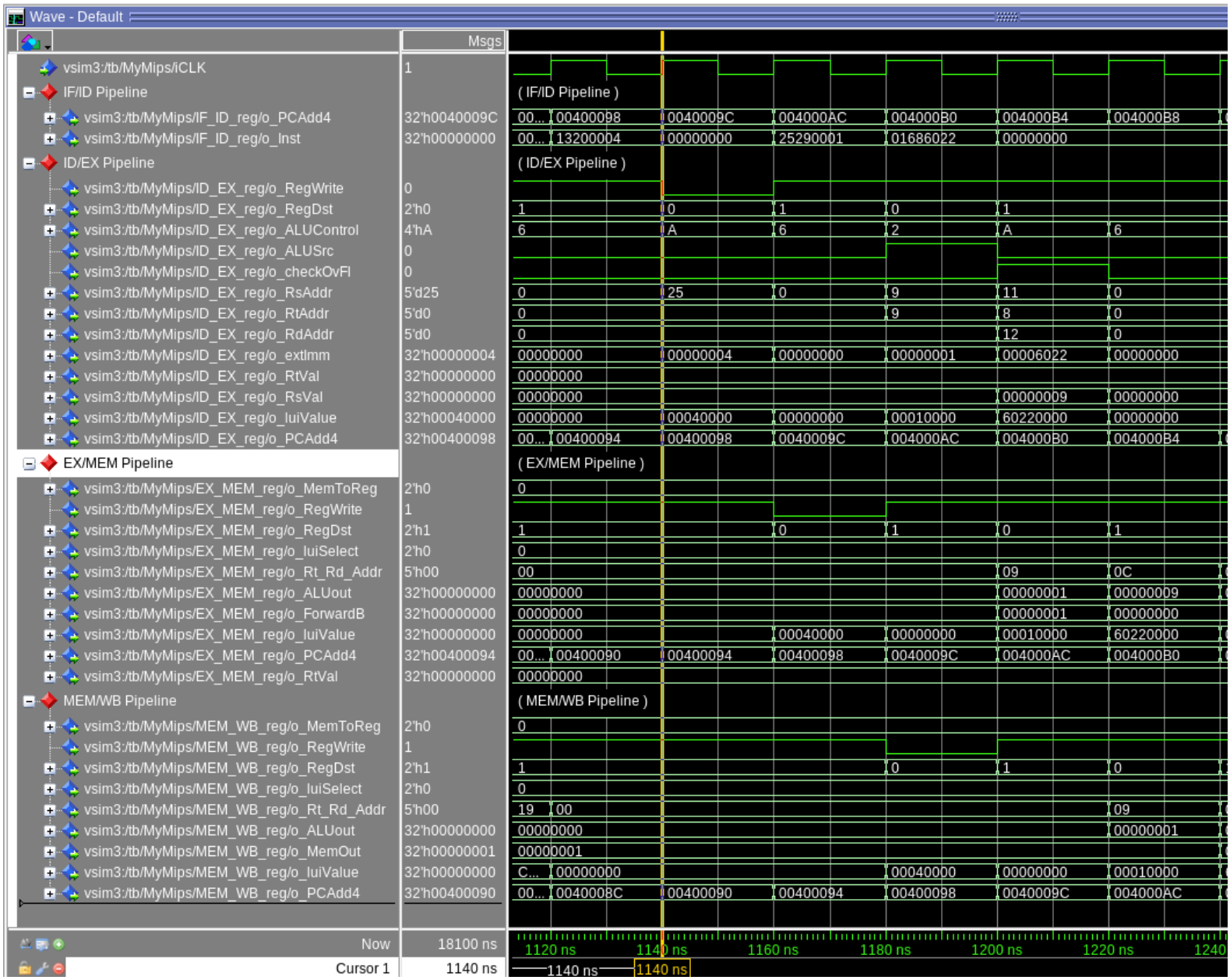
The waveform above shows the code given below when branch is not executed:

```
beq $t9, $0, skip    # if not less, skip swap
nop                  # delay slot

swap:
sw $t8, 0($t5)    # arr[j] <= arr[j+1]
sw $t7, 0($t6)    # arr[j+1] <= arr[j]
addiu $t2, $0, 1  # swapFlag = 1

skip:
addiu $t1, $t1, 1 # j+
```

The waveform above also shows all the pipeline outputs. We can see that the instruction exiting IF/ID at the beginning of the screenshot before the yellow line corresponds to the machine code for the branch instruction. On the next cycle, the $t9 register (register 25) propagates through each pipeline in the next cycle. Since this branch is not taken, we can see the store word is fetched after the delay slot nop. The next cycle shows that sw is being decoded with registers 13 and 24, which are $t5 and $t8, respectively. The extended immediate is still 0 since the offset is 0. Next cycle shows ALUOut exiting the EX/MEM pipeline, which is the same value as RsVal from the previous cycle, since Rs + Immd corresponds to the memory address. The data input is then the RtVal (turned into ForwardB signal), which is just 0 for our example.



(DESCRIPTION FOR THIS WAVEFORM IS BELOW)

The waveform above shows the same code when a branch is executed:

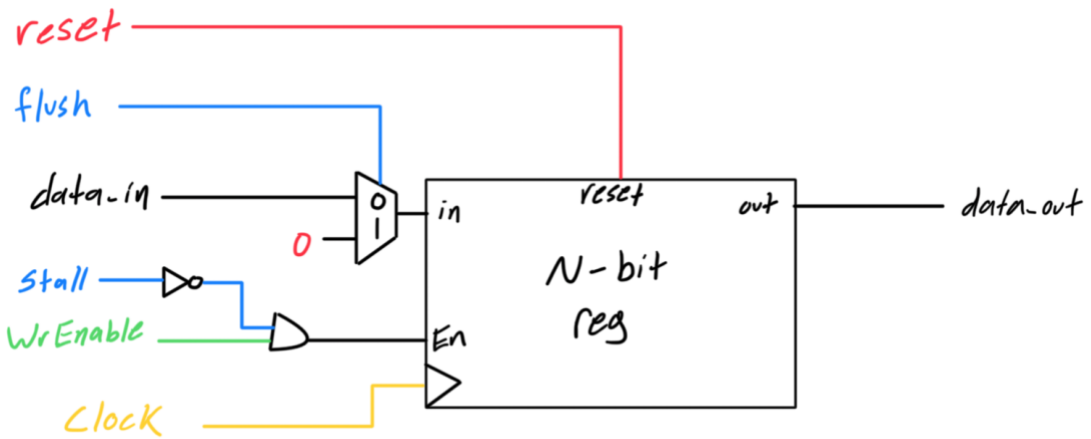
Both waveforms show that the PC has the same value for the branch instruction. This time, the PC after the nop instruction (the yellow line) is different because it jumps to the skip label. The nop is still there because it is a delay slot not flushed in our software-scheduled pipeline. However, PC jumps from 0040009C to 004000AC. We can see that the cycle after the skipped PC is RtAddr and RsAddr are both 9. This corresponds to register $t1, which is used in the addiu after the skip label to increment j. In the addiu decode cycle (cycle 1180ns), we can see that Rs and Rt are both 0. Next cycle ALUOut is then 1 because it added the 0 from Rs + the 1 from ExtImmd. The next cycle goes to WB stage, and the ALUOut is 1, and the RtRdAddr is 9, which is the write address for the RegFile. RegDst is set to 0, meaning it uses Rt as this value.

[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).

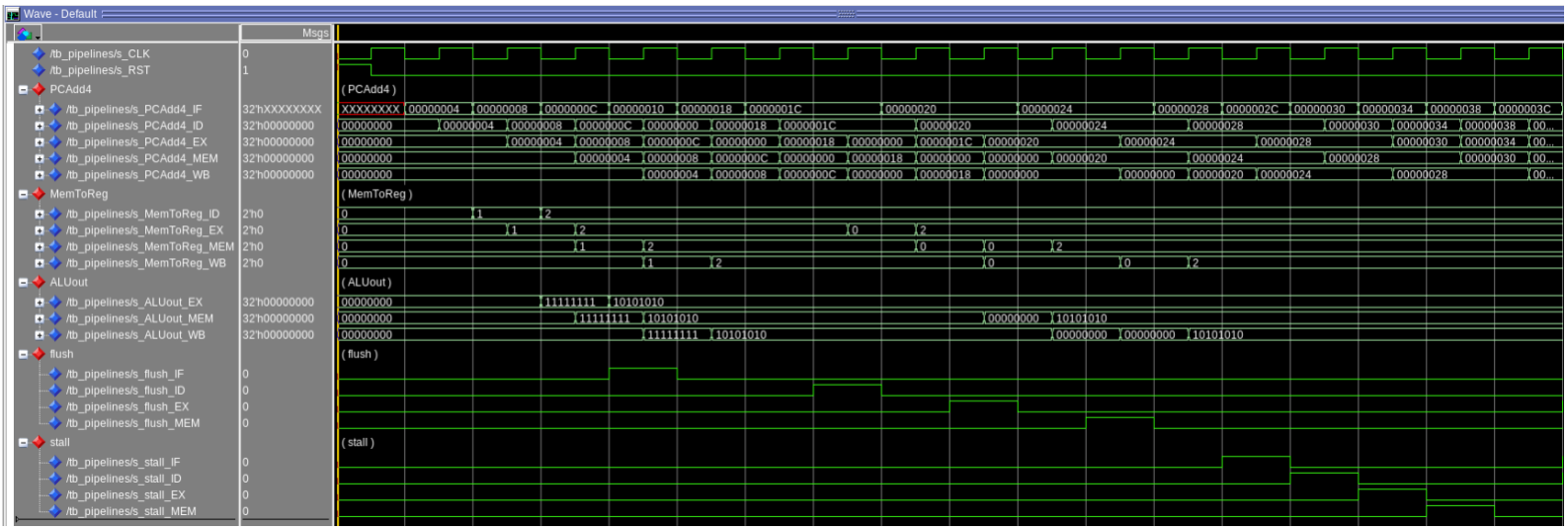
52.08 MHz is the Fmax of our processor. The critical path is shown below, along with the total time. We could move ALUSrc in the Decode instead of the Execute stage to make it faster. The most significant change would be implementing a faster Adder, because we currently use a ripple carry adder. The ALU is the slowest component, with the Full Adder inside the ALU taking 15.72ns. So, this indicates instructions such as LW, SW, Add, and Sub are on the critical path.

Critical Path:
Pipeline_ID_EX (0.232ns) -> ALUSrcMux (1.072ns) -> ALU (adder) (15.72ns) -> ALU (Output mux) (2.105ns) -> Pipeline_EX_MEM (0.104ns)
Total Time:
22.171ns

[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.



The waveform above shows the successful flushing and stalling implementation. When the flush signal is set for each register, the output of the register would be set to 0 on the next positive clock edge. When the stall signal is set for each register, the output of the register will hold the same as previous cycle.

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

Add, Addi, Addiu, Addu, And, Andi, nor, or, ori, sll, sllv, slt, slti, sra, srl, srlv, srav, sub, subu, xor, xori :

ALUOut_EX, ALUOut_MEM, ALUOut_WB, memToRegValue, RegWrData

lw :

MemOut_MEM, MemOut_WB, memToRegValue, RegWrData

lb, lbu, lh, lhu :

memToRegValue, RegWrData

lui :

luiValue_ID, luiValue_EX, luiValue_MEM, luiValue_WB, RegWrData

jal : PCAdd4_ID, PCAdd4_EX, PCAdd4_MEM, PCAdd4_WB, RegWrData

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

add, addu, and, nor, or, sllv, slt, srlv, srav, sub, subu, xor :
ALU_inputA, ALU_inputB

addi, addiu, andi, ori, sll, slti, srl, sra, xori, lw, lb, lbu, lh, lhu :
ALU_inputA

sw :
ALU_inputA, DMemData_MEM

beq, bne :
RsVal_ID, RtVal_ID

jr :
RsVal_ID

[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.

P1: ALUOut_MEM
P2: RegWrData_WB
P3: PC+4_MEM
P4: luiValue_MEM

C1: ALUInput_A_EX
C2: ALUInput_B_EX
C3: DMemData_MEM
C4: RsVal_ID
C5: RtVal_ID
C6: RtVal_EX

Data Hazards (with new read after write):

1:
Add $1 …
Add … $1 …
Forward P1 to C1/C2

2:
Add $1 … / lw $1 …
…
Add … $1 … / lw … 0($1)
Forward P2 to C1/C2

3:
Add $1 …
Beq $1 … … / jr $1
Stall ID

4:
Add $1 …
…
Beq $1 … … / jr $1
Forward P1 to C4/C5

5:
Add $1 … / lw $1 …
Sw $1 …
Forward P2 to C3

6:
Add $1 … / lw $1 …
…
Sw $1 …
Forward P2 to C6

7:
lw $1 …
Add … $1 … / lw … 0($1)
Stall EX
8:
Lw $1 …
Beq … $1 … / jr $1
Stall ID

9:
Lw $1 …
…
Beq … $1 … / jr $1
Stall ID

10:
Lui $1 …
Add … $1 …
Forward P4 to C1/C2

11:
Lui $1 …
…

Add … $1 …
Forward P2 to C1/C2

12:
Lui $1 …
sw $1 …
Forward P2 to C3

13:
Lui $1 …
…
sw $1 …
Forward P2 to C6

14:
Lui $1 …
Beq $1 … / jr $1
~~Forward P3 to C4/C5~~
Stall ID (Possible to forward, but to get rid of a producing point for a rare case)

15:
Lui $1 …
…
Beq $1 … / jr $1
Forward P4 to C4/C5

16:
jal jump
…
jump:
Add … $31 …
Forward P2 to C1/C2

17:
Jal jump
…
jump:
beq … $31 … / jr $31
Forward P3 to C4/C5

18:
jal jump
…
jump:
sw $31 …
Forward P2 to C6

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

Pipeline Inputs			
IF/ID	ID/EX	EX/MEM	MEM/WB
PC + 4	PC + 4	PC + 4	PC + 4
Instruction		MemToReg	
		RegWrite	
		RegDst	
		LuiSelect	
		s_Halt	
		SignedLoad	
	MemRead		MemOut
	DMemWr		
	ALUControl	ALUOut	ALUOut
	ALUSrc		
	CheckOvFl		
	RegShift		
	RsAddr		
	RtAddr	Rd/Rt Addr	Rd/Rt Addr
	RdAddr		
	SignExtImmd		
	RtVal	RtVal	
	RsVal		
	luiVal	luiVal	luiVal

	Used in WB
	Used in MEM
	Used in EX

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

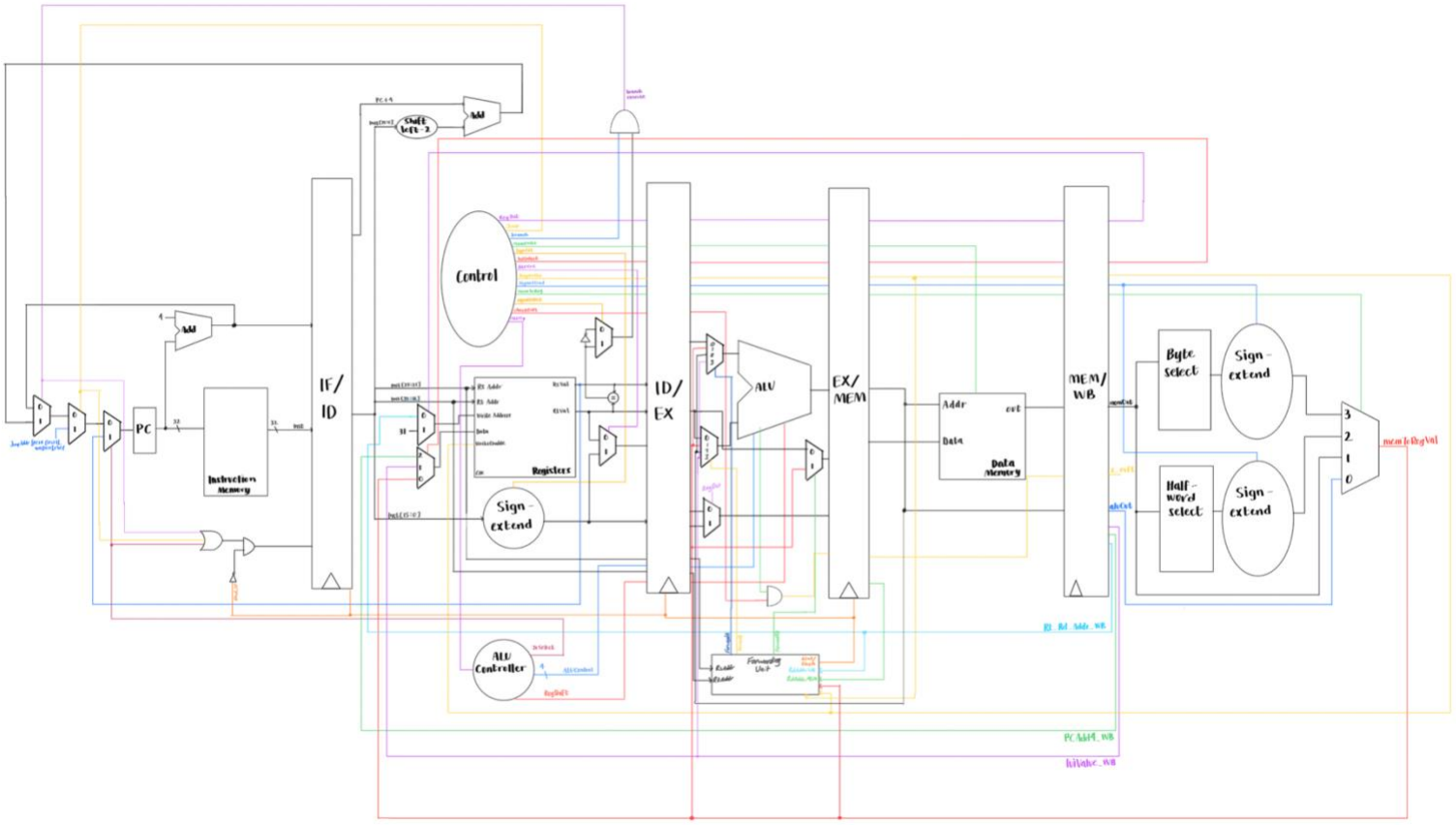
The instructions that may result in a non-sequential PC update are:
Beq
Bne
J
Jal
Jr

The pipeline stage in which the update occurs is Decode.

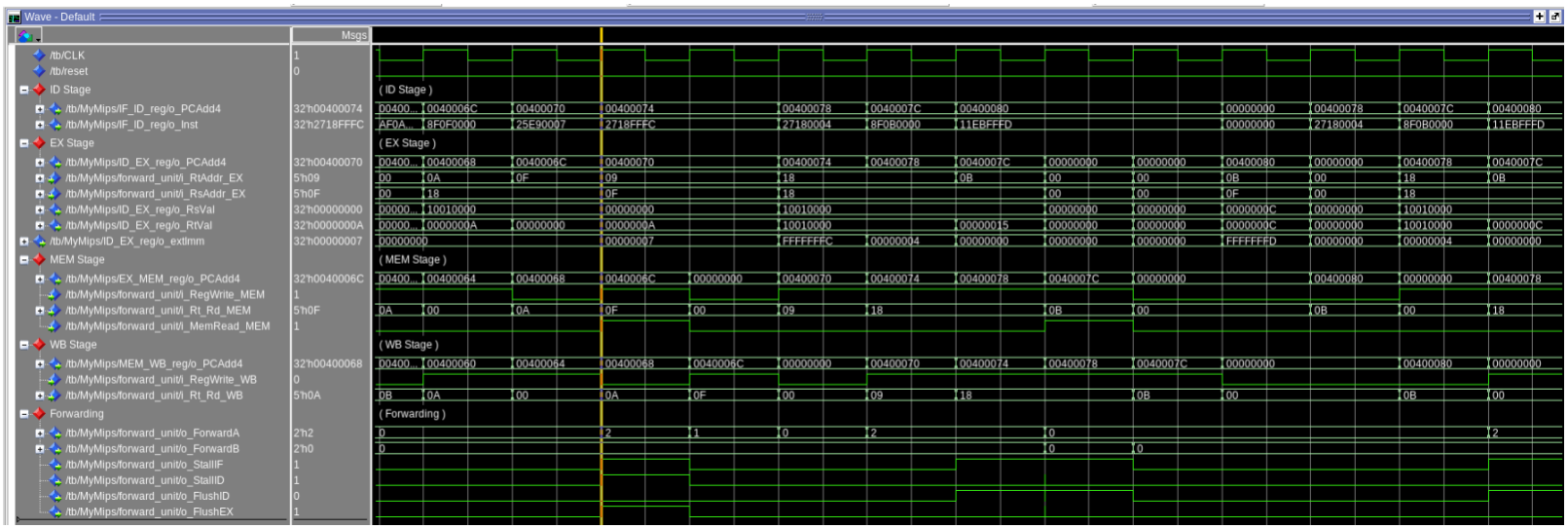
[2.c.ii] For these instructions, list which stages need to be stalled, and which stages need to be squashed/flushed relative to the stage each of these instructions is in.

Instruction	PC Update Stage	Stalls Needed	Stages Flushed / Stalled
Beq	Decode	None	Instruction Fetch
Bne	Decode	None	Instruction Fetch
J	Decode	None	Instruction Fetch
Jal	Decode	None	Instruction Fetch
Jr	Decode	None	Instruction Fetch

[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 QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.



lw $t7, 0($t8)
addiu $t1, $t7, 7

Starting at the cycle after the cursor (yellow bar), lw is in the MEM stage, and addiu is in the EX stage. Since the lw has not fetched its value from DMEM yet, a stall must occur. This is checked since MemRead_MEM is 1. Rt_Rd_MEM is 0F, and RsAddr_EX is also 0F. Finally, since RegWrite_MEM is 1, there will be a stall. This is shown with a FlushEX, StallID, and StallIF being set. This is seen on the waveform since the MEM stage next cycle is flushed with 0's and ID/EX stage keeps its values.
Additionally, now the lw is in the WB stage, and a data hazard must be forwarded. Because RegWrite_WB is 1 and Rt_Rd_WB equals RsAddr_EX, forwarding will occur as seen on the ForwardA signal set to 1. Then, in the next cycle, addiu propagates to the MEM stage. This shows correct execution of the instructions mentioned above.

[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.

Test 1	An add instruction writes to $1, and the result is used immediately in another add — this tests data forwarding from the EX/MEM stage to EX stage (P1 to C1/C2).
Test 2	An add or lw writes to $1, followed by an instruction that uses $1 in a dependent add or lw — this tests forwarding from the MEM/WB stage to the EX stage (P2 to C1/C2).
Test 3	An add instruction writes to $1, followed immediately by a beq or jr using $1 — this tests a stall in the decode (ID) stage because forwarding to a branch decision isn't available in time.

Test 4	An add writes to $1, a delay of one instruction, then a beq or jr uses $1 — this tests forwarding from the EX/MEM stage to the branch comparator (P1 to C4/C5).
Test 5	An add or lw writes to $1, and the very next instruction stores $1 to memory — this tests forwarding to the memory address input in MEM stage (P2 to C3).
Test 6	An add or lw writes to $1, then after a delay, a sw stores $1 — this tests forwarding from MEM/WB to memory address input (P2 to C6).
Test 7	A lw loads into $1, and an instruction immediately uses $1 in an arithmetic or load — this causes a stall in the EX stage because the value is not yet available.
Test 8	A lw loads into $1, and a beq or jr uses $1 right after — this results in a stall in the ID stage due to a read-after-load hazard on a branch.
Test 9	A lw loads into $1, with one instruction delay before a beq or jr uses $1 — still causes a stall in ID, as the value is needed too early in branch evaluation.
Test 10	A lui loads a constant into $1, immediately followed by an add using $1 — this tests forwarding from the WB stage (P4) to the EX stage (C1/C2).
Test 11	A lui is followed by one instruction, then an add uses $1 — this tests forwarding from MEM/WB (P2) to EX stage (C1/C2).
Test 12	A lui writes to $1, and the next instruction stores $1 — this tests forwarding from MEM/WB (P2) to MEM stage input (C3).
Test 13	A lui is followed by one instruction, then a sw uses $1 — this tests forwarding from MEM/WB (P2) to memory stage input (C6).
Test 14	A lui writes to $1, and the very next instruction is a beq or jr using $1 — forwarding from MEM stage (P3) to branch comparator (C4/C5) is possible, but usually stalls in ID.
Test 15	A lui followed by one instruction, then a beq or jr using $1 — this tests forwarding from WB (P4) to branch comparator (C4/C5).
Test 16	A jal sets $31 (return address), followed by an add using $31 — this tests forwarding from MEM/WB (P2) to EX stage (C1/C2).
Test 17	A jal sets $31, and after one instruction, a beq or jr uses $31 — this tests forwarding from MEM stage (P3) to branch comparator (C4/C5).
Test 18	A jal sets $31, followed by a store that writes $31 to memory — this tests forwarding from MEM/WB (P2) to memory stage input (C6).
Double Hazard Test	A loop increments a counter and performs two consecutive dependent operations (sub and slt) on that counter before conditionally branching, testing multiple back-to-back data hazards.
Mini Grendel Test	Loads a memory address, reads and modifies the data at that address using multiple register reuses and memory accesses, then writes the result back, testing memory and register data forwarding.

Hazard Detection and Forwarding.xlsx

[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.

Test Case	Instructions under test	Explanation
main	N/A	Initializes registers and directs control to beq or j tests.
test_beq	beq	Tests if beq correctly branches when the condition is true.
test_bne	bne, beq	Validates bne after a beq in the same pipeline path.
test_j	j	Tests unconditional jump to a target label.
test_j_target	jal	Tests jump and link to subroutine from a j target.
test_jal	jal, jr	Tests nested jal calls with jr returning via the stack.
test_nested	jal, jr	Tests another level of jal to confirm return address handling.
test_jr	jr	Tests register-based jump using return address from jal.
test_comb1	beq	Begins a combined sequence with a conditional branch.
test_comb2	bne	Tests conditional branch after beq with changed value.
test_comb3	jal, j	Tests call to a function via jal and immediate jump after.
test_comb4	beq, jr	Tests branching followed by immediate return using jr.
test_comb5	j	Verifies that j transfers control correctly to the next test.
test_comb6	jal	Tests a function call without return logic to verify call flow.
test_comb7	jr	Returns control from a jal with a clean jr.
exit	N/A	Terminates the program to finalize the test sequence.

Control Hazard Avoidance.xlsx

[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).

47.37 MHz is the Fmax of our processor. The critical path is shown below, along with the total time. We moved the ALUSrc to the ID stage versus the software-scheduled design. To make it faster, we should still work on reducing the time the ripple carries adder takes by using a different adder design. We also added high latency from the DMem due to forwarding. Fixing this dataflow of forwarding could reduce the timing overhead. It also now needs to wait for MemToReg and LuiMux to finish. Still, these are necessary unless we make many new producing points, which would probably add more latency to the forwarding muxes.

Critical Path:

DMem (3.112ns) –> memToRegMux (1.543ns) –> luiMux (0.376ns) –> ForwardAndStall (0.9ns) –> ALU/adder (14.217ns) –> ALU/mux (0.839ns) –> Pipeline_EX_MEM (0.087ns).

Total Time:
23.504ns