

CprE 3810: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Zevan Gustafson

Shane Nebraska

Project Teams Group #: 4

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.

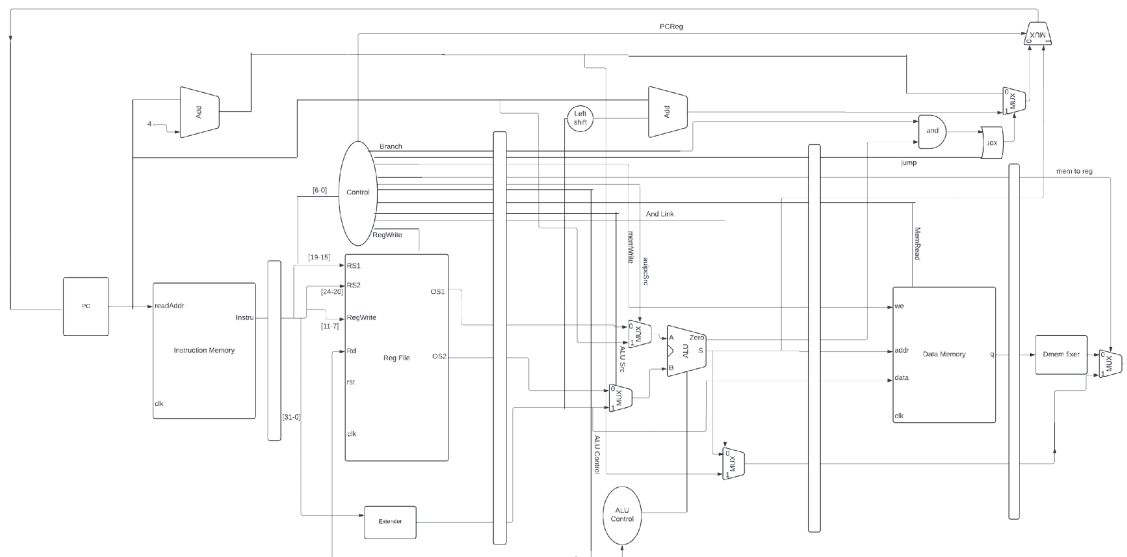
IF/ID: ALUSrc, ALUControl, MemWrite, Branch, MemtoReg, RegWrite, ImmType, imm_sel, branchType, jump, andLink, PCReg, auipcSrc, instruction

ID/EX: ALUSrc, ALUControl, MemWrite, Branch, MemtoReg, RegWrite, branchType, jump, andLink, PCReg, auipcSrc, instruction, dataout1, dataout2

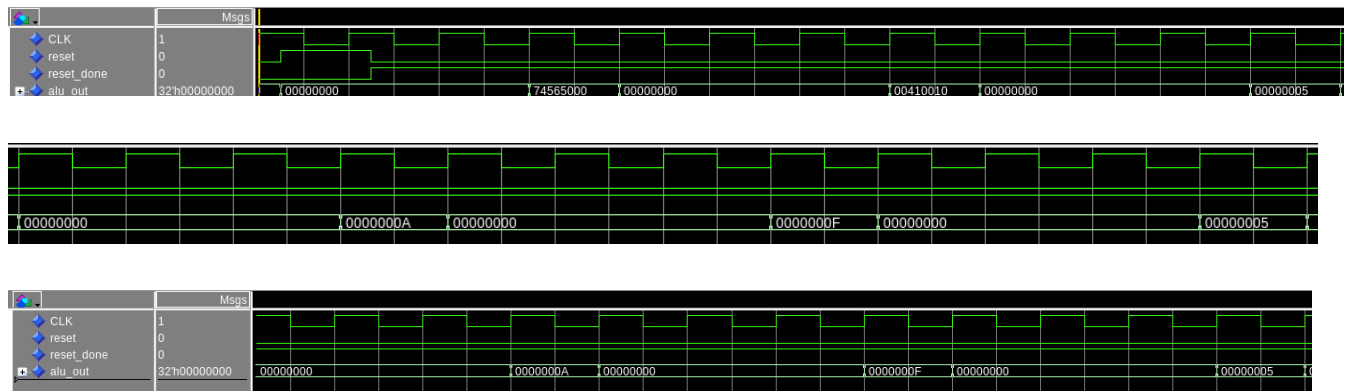
EX/MEM: MemWrite, Branch, MemtoReg, RegWrite, jump, andLink, PCReg, auipcSrc, dataout1, data out2

MEM/WB: MemtoReg, RegWrite

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

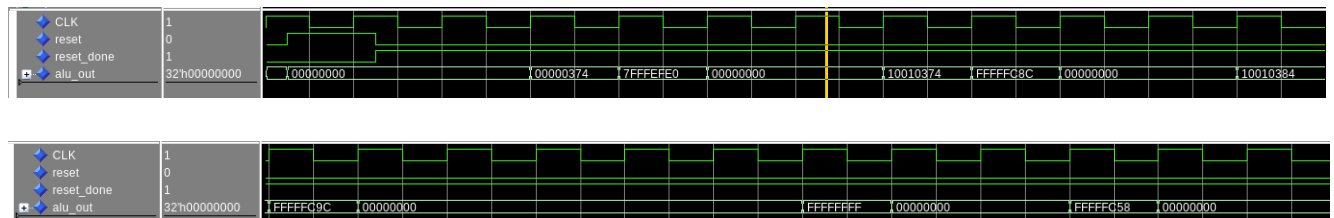


[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.



The resulting waveform matches the expected behavior expected for the included instructions. In addition it passes in the toolflow every instruction and passes all other tests in the toolflow.

[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.

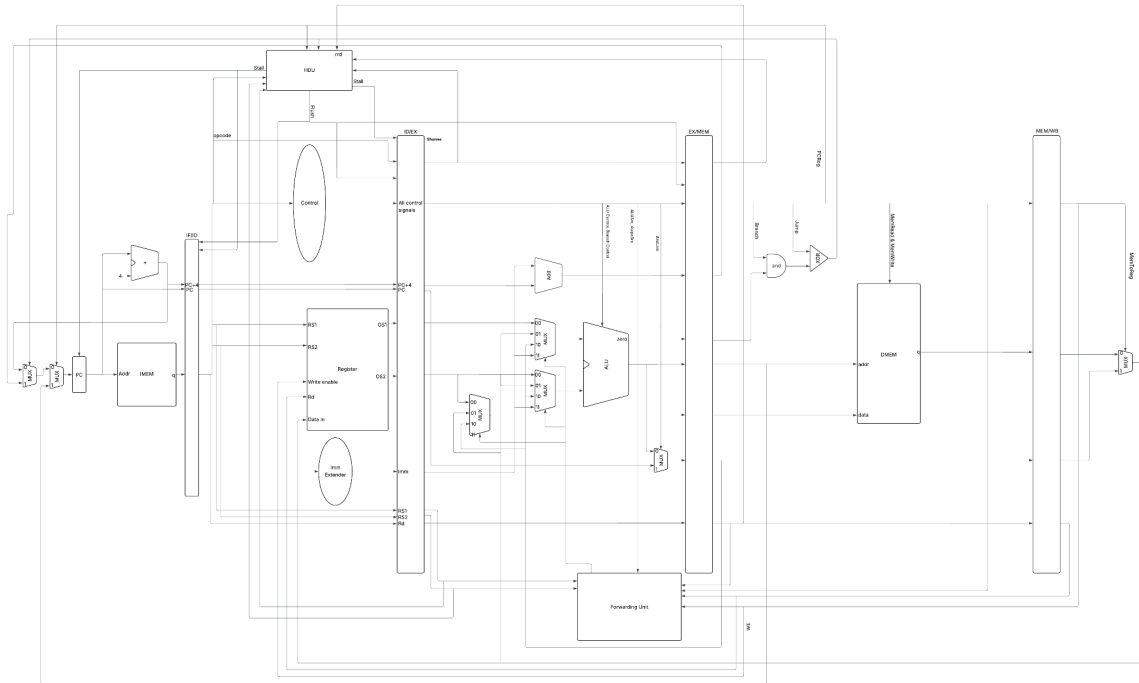


The resulting waveform matches the expected behavior expected for the included instructions and passes in the toolflow. Additionally, it has the same results as the waveform of the regular merge sort on the single-cycle processor. There was no scenario where we did not need to use the maximum number of NOPs. One example of a control-flow hazard was blt x11, x15, right_side followed by addi x0, x0, 0. One data-flow hazard example was an add x14, x12, x13 followed by srl x14, x14, 1. A third example was slli x24, x17, 2 followed by add x25, x14, x24.

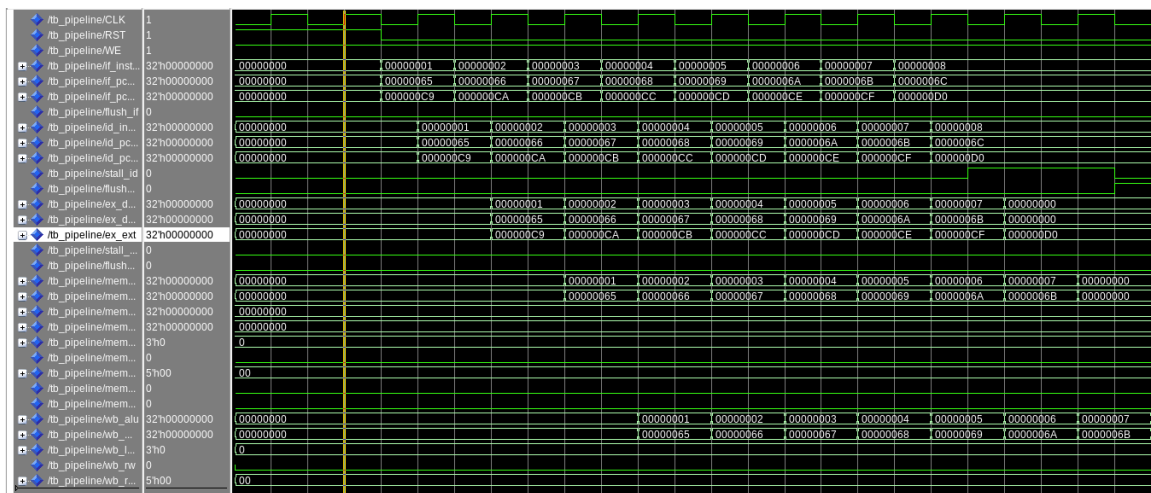
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

The maximum frequency of the software-scheduled pipeline is 54.50 mhz. The critical path is: register file, ID/EX, mux, ALU, EX/MEM, data memory, dmem fixer, mux, MEM/WB.

[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.



srai - in_alu
auipc - in_alu

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

Addi - in_data1
add - in_data1, in_data2
and - in_data1, in_data2
andi - in_data1
lw - in_data1, in_data2, in_dmem
xor - in_data1, in_data2
xori - in_data1
or - in_data1, in_data2
ori - in_data1
slt - in_data1, in_data2
slti - in_data1
sll - in_data1, in_data2
srl - in_data1, in_data2
sra - in_data1, in_data2
sw - in_data1, in_data2
sub - in_data1, in_data2
jalr - in_data1, in_data2
lb - in_data1, in_data2, in_dmem
lh - in_data1, in_data2, in_dmem
lbu - in_data1, in_data2, in_dmem
lhu - in_data1, in_data2, in_dmem
slli - in_data1
srli - in_data1
srai - in_data1

[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.

Addi - EX to ID
add - EX to ID
and - EX to ID
andi - EX to ID
lw - stall
xor - EX to ID
xori - EX to ID
or - EX to ID
ori - EX to ID
slt - EX to ID
slti - EX to ID
sll - EX to ID
srl - EX to ID
sra - EX to ID
sw - EX to ID

sub - EX to ID
jalr - EX to ID
lb - stall
lh - stall
lbu - stall
lhu - stall
slli - EX to ID
srli - EX to ID
srai - EX to ID

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

IF/ID: ALUSrc, ALUControl, MemWrite, Branch, MemtoReg, RegWrite, ImmType, imm_sel, branchType, jump, andLink, PCReg, auipcSrc, instruction, s_stall_pc, s_flush_fetch
ID/EX: ALUSrc, ALUControl, MemWrite, Branch, MemtoReg, RegWrite, branchType, jump, andLink, PCReg, auipcSrc, instruction, dataout1, dataout2, s_stall_decode, s_flush_decode
EX/MEM: MemWrite, Branch, MemtoReg, RegWrite, jump, andLink, PCReg, auipcSrc, dataout1, data out2, s_stall_execute, s_flush_execute
MEM/WB: MemtoReg, RegWrite

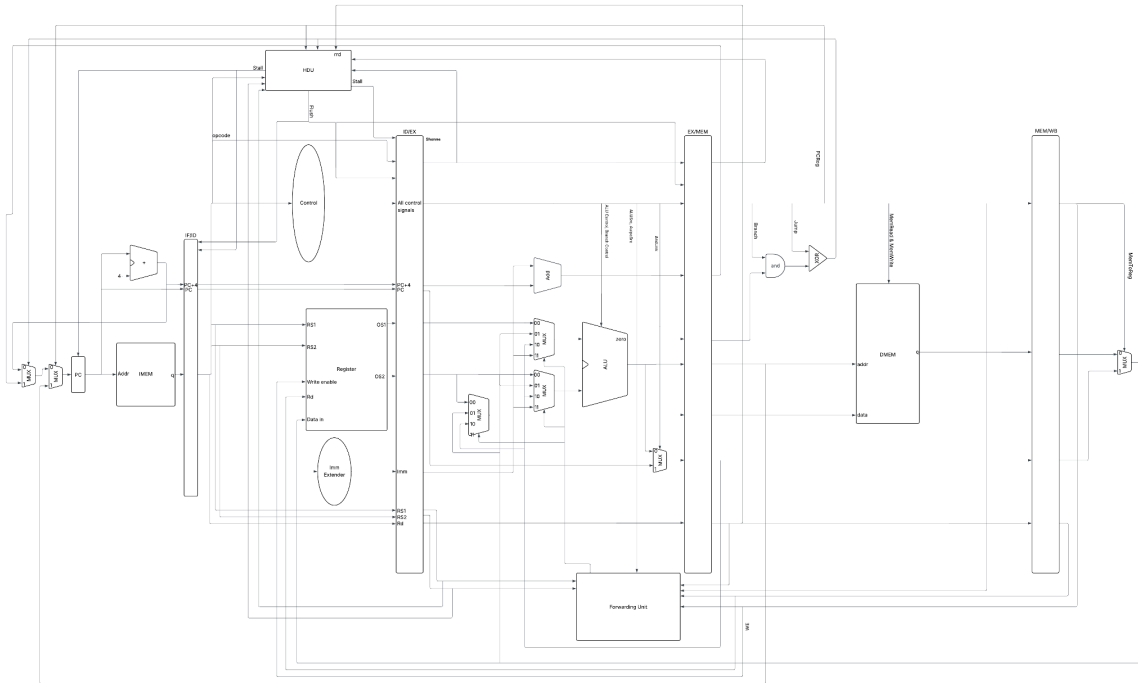
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

Instructions that may result in a non-sequential pc update would be beq, bne, blt, bge, bltu, bgeu, jal, and jalr. For all of these instructions the pipeline stage where the update occurs is the execution stage.

[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.

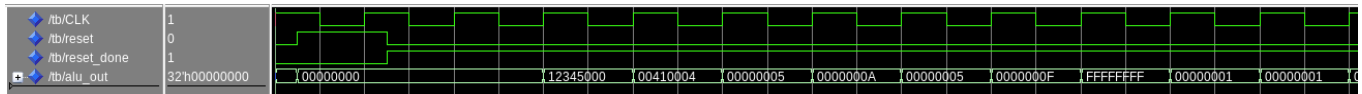
Once the instruction reaches the execute stage the fetch stage would need to stall for one cycle. Once the new PC is known in execute the instructions in both fetch and decode get flushed.

[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.



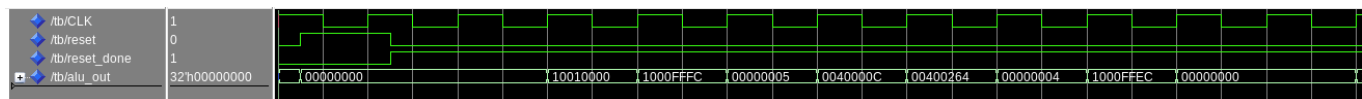
[2.e – i, ii, and iii] In your writeup, show the QuestaSim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

Base Test:



This test passes in the toolflow and matches the output produced by the single-cycle processor.

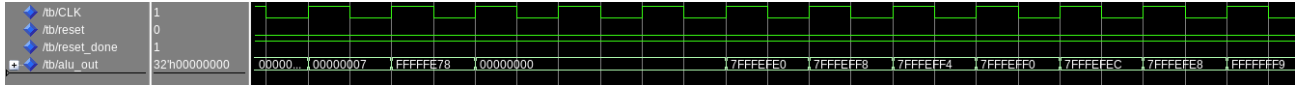
Cf Test:



This test passes in the toolflow and matches the output produced by the single-cycle processor.

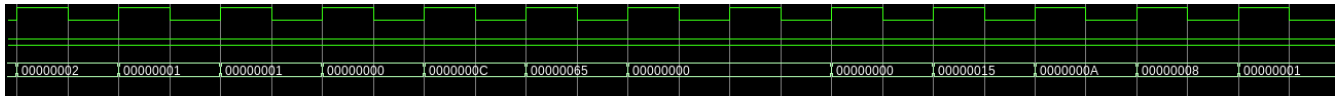
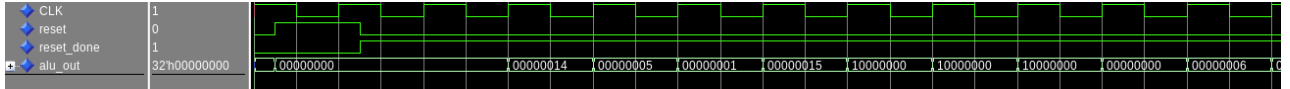
Mergeshort Test:



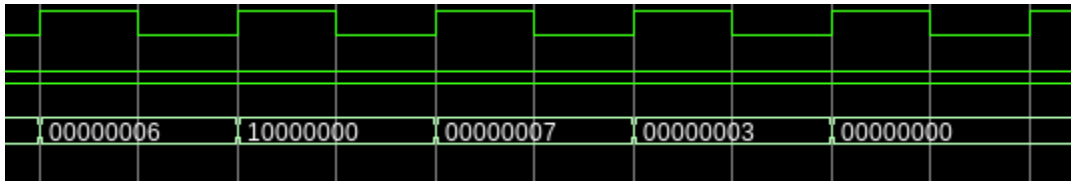


This test passes in the toolflow and matches the output produced by the single-cycle processor.

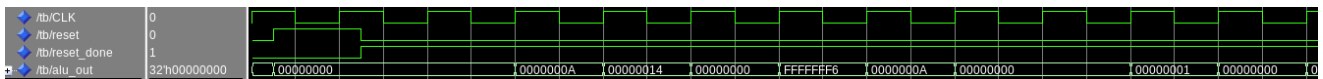
Mem_to_Ex_forwarding Test:



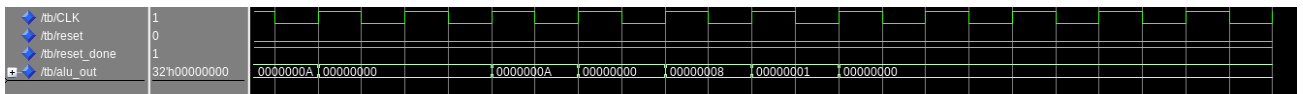
WB_to_Ex_forwarding Test:



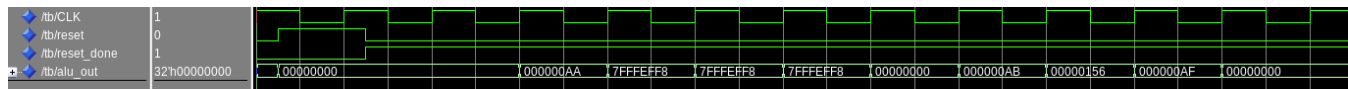
Flush_branch Test:



Flush_jump Test:



Load_after_use Test:



[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.

Hazard Case	Instructions Triggering it (I1 → I2)	Example Code	Expected Action
ALU/PC-to-EX (1-Cycle)	I1 (Producer) is in MEM, I2 (Consumer) is in EX.	add x1, x2, x3	Forward result from the EX/MEM pipeline register to the EX stage input for I2.
	I1 is a {R-Type, I-Type ALU, Lui/AUIPC, JAL/JALR}	sub x4, x1, x5	This resolves the dependency without a stall.
	I1 is {add, sub, and, or, xor, slt, sll, srl, sra, addi, andi, ori, xori, slti, sltiu, slli, srli, srai, lui, auipc, jal, jalr}		
	I2 is a {ALU, Store, Branch, JALR} (Any instruction reading Rs1 or Rs2)		
	I2 is {add, sub, and, or, xor, slt, sll, srl, sra, sw, beq, bne, blt, bge, bltu, bgeu, lb, lh, lbu, lhu, addi, andi, ori, xori, slti, sltiu, slli, srli, srai, jalr}		
Full Load-to-EX (2-Cycle)	I1 (Load Producer) is in WB, I2 (Consumer) is in EX.	lw x1, 0(x2)	Forward result from the MEM/WB pipeline register to the EX stage input for I2.
	I1 is a {Load Instructions}	addi x3, x3, 10	This resolves the dependency without a stall.
	I1 is {lw, lb, lh, lbu, lhu}	and x4, x1, x5	
	I2 is a {ALU, Store, Branch, JALR} (Any instruction reading Rs1 or Rs2)		

Full ALU/PC-to-EX (2-Cycle)	I1 (ALU/PC Producer) is in WB, I2 (Consumer) is in EX.	addi x1, x2, 10	Forward result from the MEM/WB pipeline register to the EX stage input for I2.
	I1 is a {R-Type, I-Type ALU, LUI/AUIPC, JAL/JALR}	sub x3, x4, x5	This resolves the dependency without a stall.
	I1 is {add, sub, and, or, xor, slt, sll, srl, sra, addi, andi, ori, xori, slti, sltiu, slli, srli, srai, lui, auipc, jal, jalr}	or x4, x1, x5	
	I2 is in {ALU, Store, Branch, JALR} (Any instruction reading Rs1 or Rs2)		

[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.

Hazard Case	Instructions Triggering it (I1 → I2)	Example Code	Expected Action
Load-Use Data Stall	I1 (Load Producer) is in EX, I2 (Consumer) is in ID.	lw x1, 0(x2)	HDU detects the immediate dependency (I1.Rd == I2.Rs1 or I2.Rs2) and forces a 1-cycle stall (NOP) on I2 and I3, waiting for the load data to exit MEM.
	I1 is a {lw, lb, lh, lbu, lhu}	add x3, x1, x4	The result is then forwarded from MEM/WB.
	I2 is in {ALU, Store, Branch, JALR} (Any instruction reading Rs1 or Rs2)		
Control Hazard Flush (Jumps)	I1 is a {JAL, JALR} (Resolved in MEM)	jal x1, label	HDU detects the jump and forces a 3-cycle flush on the instructions currently in IF, ID, and EX stages (I2, I3, I4) after I1 exits MEM.
	I1 is executed.	addi x2, x2, 1	The PC is then redirected to the calculated target.

Control Hazard Flush (Branches)	I1 is a {Branch Instructions} (Resolved in MEM)	beq x1, x2, label	Assuming a misprediction (or no prediction), HDU forces a 3-cycle flush on I2, I3, and I4 (IF, ID, EX) after I1 exits MEM.
	I1 is {beq, bne, blt, bge, bltu, bgeu}	sub x3, x4, x5	The PC is then redirected to the correct target (if taken) or next instruction (if not taken).

[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).

The maximum frequency of the hardware-scheduled pipeline is 43.29 mhz. The critical path is: register file, ID/EX, mux, forwarder, ALU, EX/MEM, data memory, dmux, mux, MEM/WB.