CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Josh Arceo

Joseph Barnes III

Evan Shiber

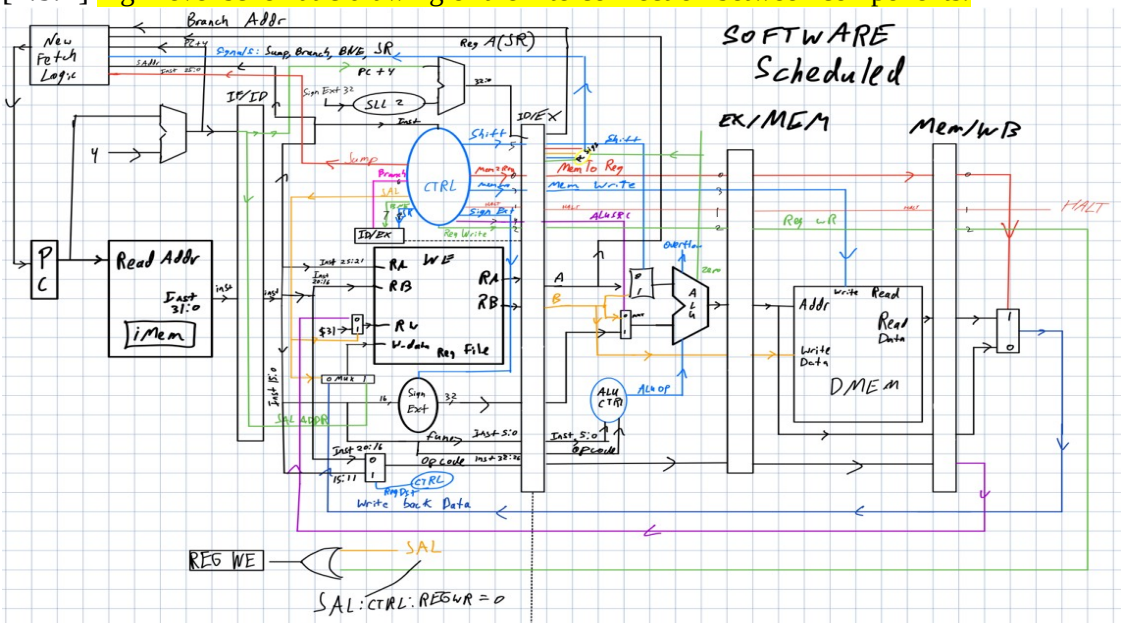
Project Team Group #: Sec C 03

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

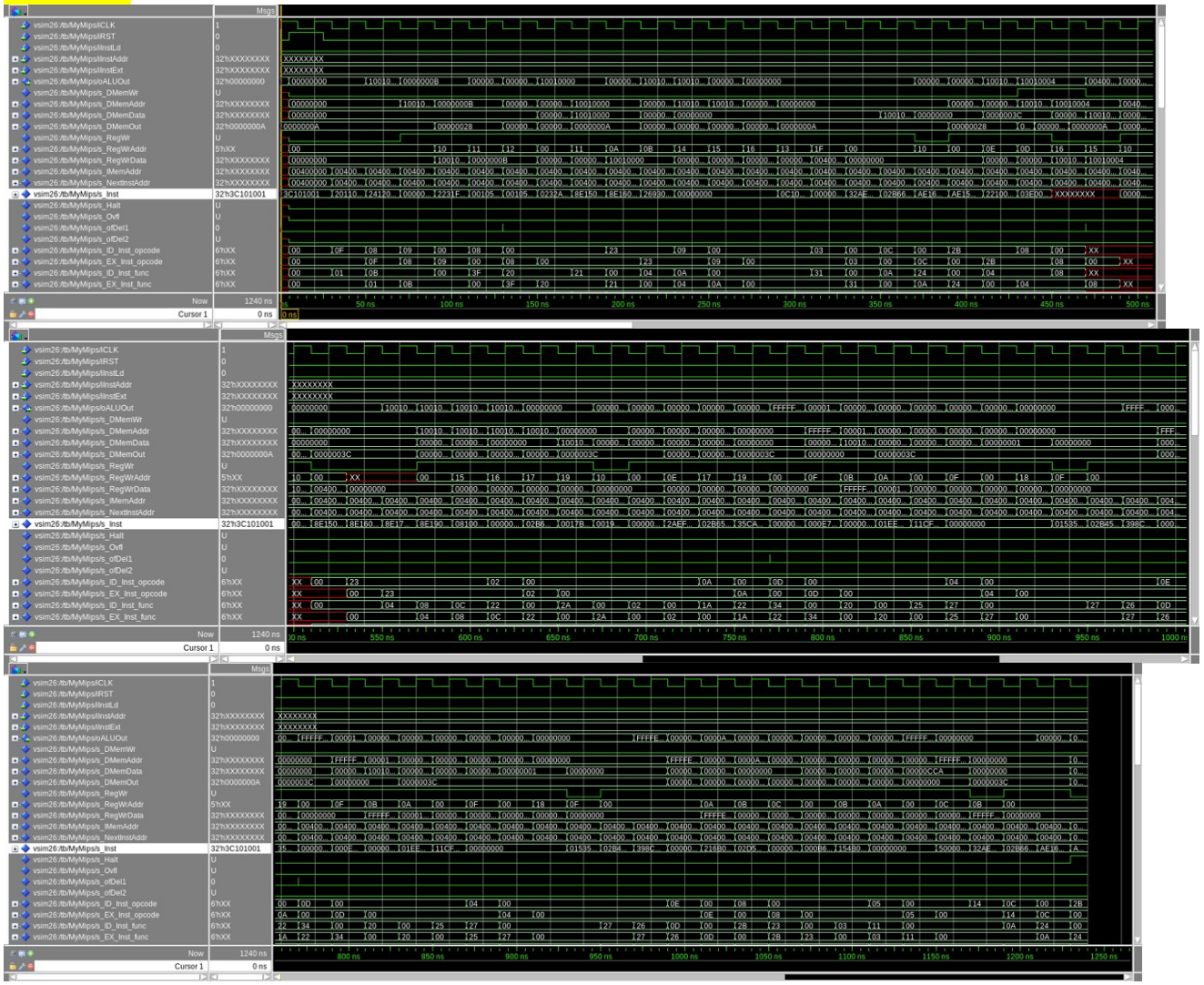
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

DECODE		EXECUTE		MEMORY		WRITEBACK	
HALT	**GENERATED**	HALT		HALT		HALT	**CONSUMED**
		OVERFLOW	**GENERATED**	OVERFLOW		OVERFLOW	**CONSUMED**
MEM TO REG	**GENERATED**	MEM TO REG		MEM TO REG		MEM TO REG	**CONSUMED**
REG WRITE	**GENERATED**	REG WRITE		REG WRITE		REG WRITE	**CONSUMED**
MEM WRITE	**GENERATED**	MEM WRITE		MEM WRITE	**CONSUMED**		
ALU SRC	**GENERATED**	ALU SRC	**CONSUMED**				
SHIFT	**GENERATED**	SHIFT	**CONSUMED**				
BRANCH	**GENERATED**	BRANCH	**CONSUMED**				
BRANCH NE	**GENERATED**	BRANCH NE	**CONSUMED**				
JUMP RET	**GENERATED**	JUMP RET	**CONSUMED**				
JUMP	**GEN+CONSUMED**						
JAL	**GEN+CONSUMED**						
SIGNEXT	**GEN+CONSUMED**						

[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.

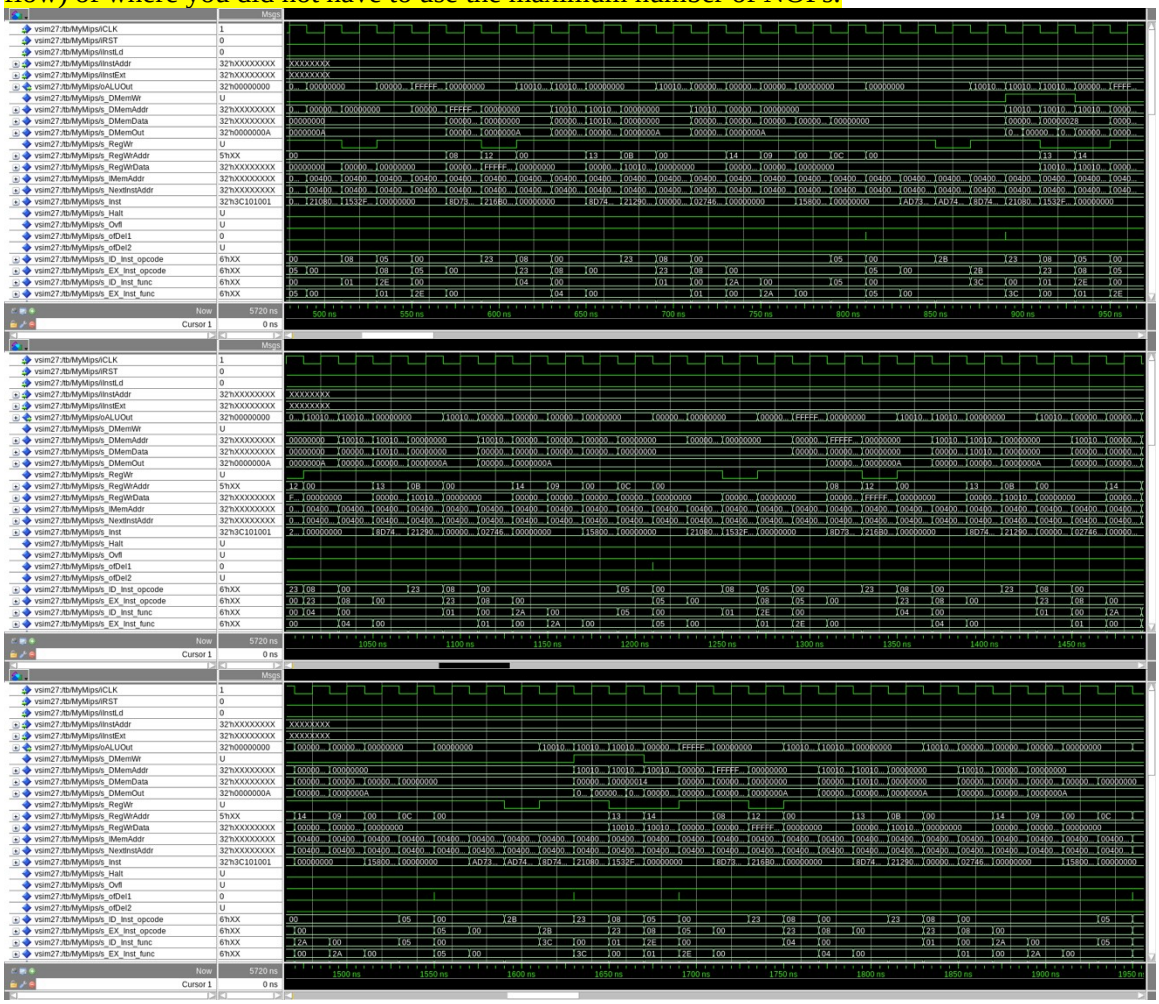


We can see the current Instrution in the IF stage in s_Inst, and since it's software scheduled they're is not flushing or stalling so we can assume that in the next cycle it moves to ID, in the 1'st image the first instruction is lui $s0, 0x1001, then 4 cycles later in it's WB (100ns) stage we see RegWrAddr is 10 ($s0) and RegWrdata is 0x10010000. We can follow the rest of the waveform to see all writes will be properly processed.

JAL (300ns) has 3 NOPs ahead of it to ensure all r type instructions before it have finished using the regfile to write and then JAL can jump and write the correct addr into $31 in the ID stage.

Branches have 2 nops after them to correctly calculate whether it will branch or not before following next instrucitons. (850 ns)

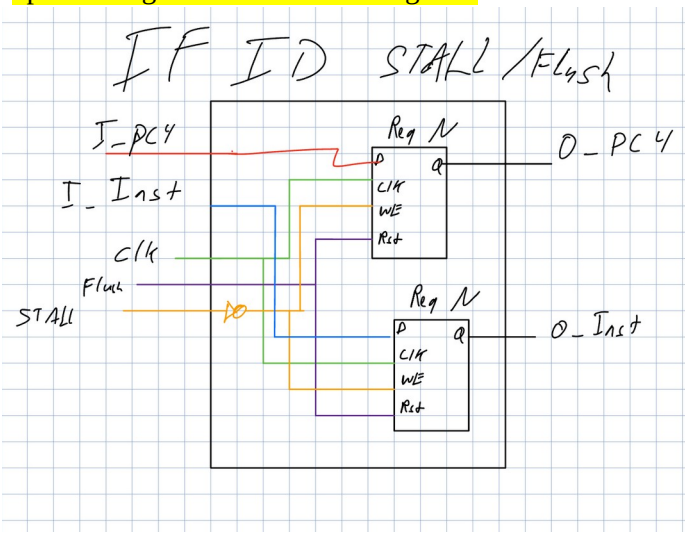
[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did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



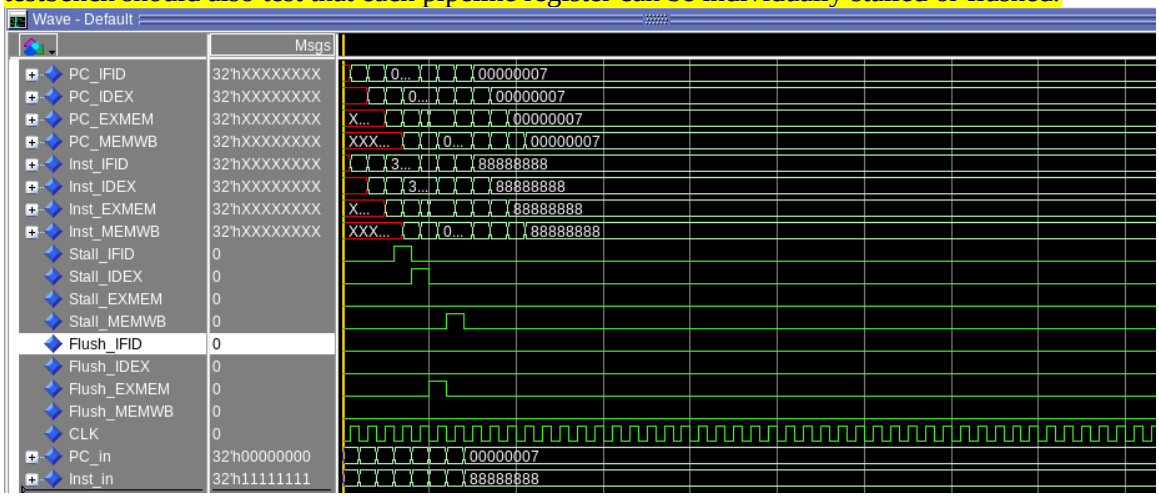
The array is properly sorted at the end of execution. Which can be seen when we compare Dmem values at the beginning vs end of execution. In our cond label instead of adding NOPs after incrementing $t0 before using it in a branch comparison, we instead moved it to the beginning of the label and moved another branch comparison after it to give room before we compared it. (1200ns) Similarly, we moved addi $t0 to the beginning of the print_loop only requiring 1 nop instead of 2 before the branch condition. This did require us to add a nop before this loop however, since it isn't inside of the loop it actually saves us 1 cycle every time that loop runs. (This part is near the end and isn't shown in the waveform)

[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).
50.39mhz
Critical Path:
PC_Reg → iMem → IFID → PC Fetch → Reg File → IDEX → ALU A MUX/ALU B MUX → ALU → EXMEM → DMEM → MEMWB → WB MUX → RegDst Mux → JAL MUX

[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.



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

add/addi/addiu ALU Out → DMEM ADDR→ WB
and/andi/nor/xor/xori/or/ori ALU Out → DMEM ADDR → WB
lui/lw ALU Out → DMEM ADDR // DMEM Out → WB
slt/slti ALU Out → DMEM ADDR → WB
sll/srl/sra ALU Out → DMEM ADDR → WB
sub/subu ALU Out → DMEM ADDR → WB

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

add Consumes RS, RT → RegA/RegB → ALU A/ALU B
addi/addiu Consumes RS→ RegA → ALU A
and/nor/xor/or/ Consumes RS, RT → RegA/RegB → ALU A/ALU B
andi/ xori/ ori Consumes RS→ RegA → ALU A
lui Consumes imm → ALU A (shifted)
lw Consumes RS → ALU A → ALU OUT → DMEM ADDR
sw Consumes RS → ALU A → ALU OUT → DMEM ADDR Then Consumes RD → DMEM DATA
slt Consumes RS, RT → RegA/RegB → ALU A/ALU B
slti Consumes RS→ RegA → ALU A
sll/srl/sra Consumes RS, RT → RegA/RegB → ALU A/ALU B
sub/subu Consumes RS, RT → RegA/RegB → ALU A/ALU B
jr Consumes RS → Reg A → RegA Out → Next Addr

[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.

Instruction that writes to a register followed by an instruction that consumes the value of that register as either the next instruction or the 2nd instruction after.

Instruction lw that writes to a register followed by an instruction that consumes that register value, for this we will need one stall to ensure that lw can read the data from memory then it can be forwarded from MEM to the EX.

Instruction that writes to a register followed by LW that uses that register as the dataAddr, the value can be forwarded like the first data dependency listed without stalling

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

FETCH	DECODE	EXECUTE	MEMORY	WRITEBACK
	HALT **GENERATED**	HALT	HALT	HALT **CONSUMED**
PC Stall		OVERFLOW **GENERATED**	OVERFLOW	OVERFLOW **CONSUMED**
	MEM TO REG **GENERATED**	MEM TO REG	MEM TO REG	MEM TO REG **CONSUMED**
IFID Stall	REG WRITE **GENERATED**	REG WRITE	REG WRITE	REG WRITE **CONSUMED**
IFID Flush	MEM WRITE **GENERATED**	MEM WRITE	MEM WRITE **CONSUMED**	
	ALU SRC **GENERATED**	ALU SRC **CONSUMED**		
	SHIFT **GENERATED**	SHIFT **CONSUMED**	FWD Data **CONSUMED**	
	BRANCH **GENERATED**	BRANCH **CONSUMED**		
	BRANCH NE **GENERATED**	BRANCH NE **CONSUMED**		
	JUMP RET **GENERATED**	JUMP RET **CONSUMED**		
	JUMP **GEN+CONSUMED**			
	JAL **GEN+CONSUMED**	FWD A **GEN+CONSUMED**		
	SIGNEXT **GEN+CONSUMED**	FWD B **GEN+CONSUMED**		
	IDEX Stall	EXMEM Stall	MEMWB Stall	
	IDEX Flush	EXMEM Flush	MEMWB Flush	

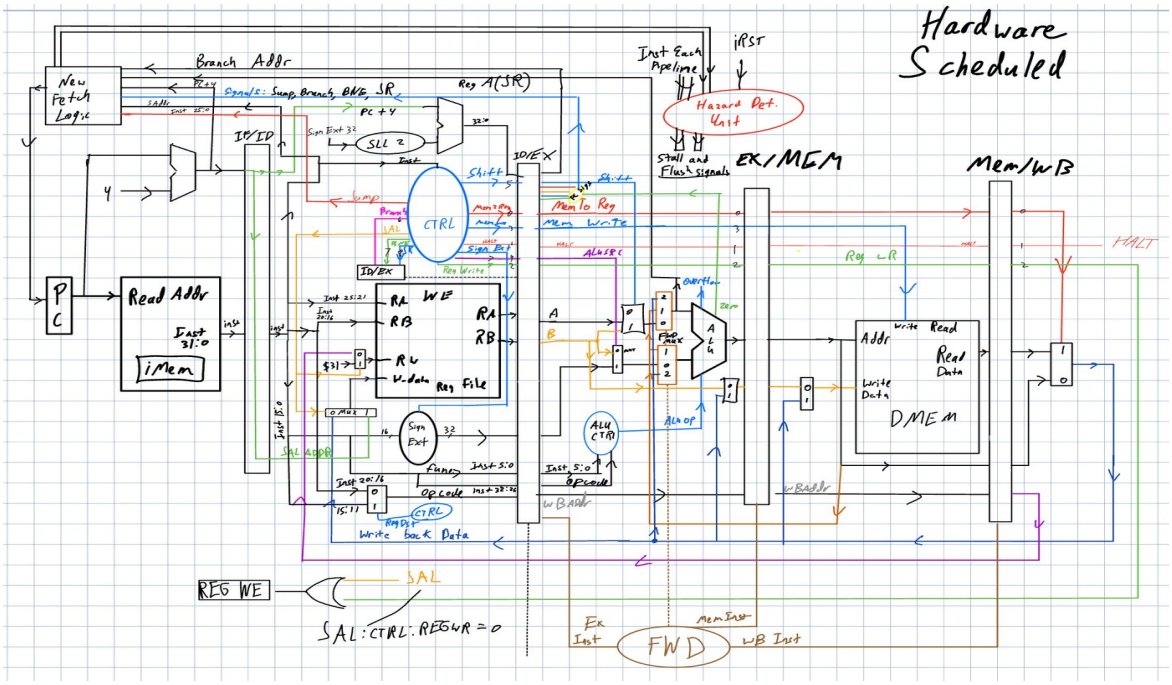
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

Jump: ID
JAL: ID
JR: EX
BEQ: EX
BNE: EX

[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.

Jump: Stall PC and flush IFID when reaches EX
JAL: Stall in IF when another instruction is writing to reg, until they're done. Then stall IFID when in ID and flush IFID, IDEX, EXMEM when reaches EX. (Need to be done the way our JAL is implemented because it writes in ID and not WB)
JR: EX
BEQ: EX
BNE: EX

[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.

[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	Data Hazard	Add	Test read after write hazard with just adds/addis
Test 2	Data Hazard	Add/LW	Test read after write hazard with add/addi and lw
Test 3	Data Hazard	Add/LW/SW	Test read after write hazard with add/addi and lw and sw

[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.

ControlTest1	Control Hazard	BEQ	Tests to make sure BEQ branches correctly and does not run instructions that it branched past
ControlTest2	Control Hazard	BEQ/JUMP	Tests to make sure it BEQ wont run jumps it branches past since jumps execute in ID
ControlTest3	Used all Branch/Jump Base tests		These tests ensured comprenehsive testing for all control hazards that we could come across
			and lead to us discovering a rare edge case in JR_5 where the program would flush the halt
			after a JR thinking that it was misread because the Jaddr was = PC + 4

[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).

35.60mhz

PC_Reg → iMem → IFID → Control → Hazard Detection → Reg File → IDEX → Forward → PC Fetch → FWD MUX A/FWD MUX B → ALU A MUX/ALU B MUX → ALU → EXMEM → FWD MUX DMEM Data → DMEM → MEMWB → WB MUX → RegDst Mux → JAL MUX