CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Jake Hafele

Thomas Gaul

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

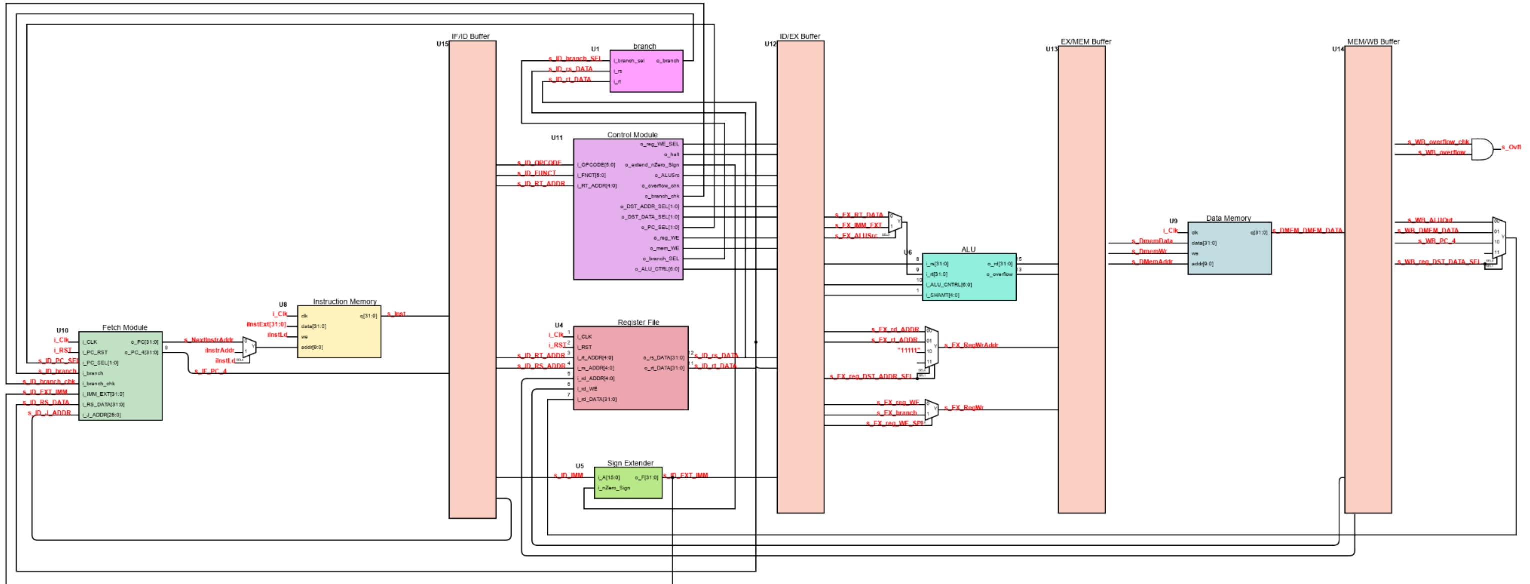
Control Signal						
Control Signal	IF (Instruction Fetch)	ID (Instruction Decode)	EX (Execute)	DMEM (Data Memory)	WB (Writeback)	Notes
reg_WE_SEL	NO	s_ID_reg_WE_SEL	s_EX_reg_WE_SEL	NO	NO	
Halt	NO	s_ID_Halt	s_EX_Halt	s_DMEM_Halt	s_WB_Halt	
nZero_Sign	NO	s_ID_nZero_Sign	NO	NO	NO	Sign extender in ID stage
ALUSrc	NO	s_ID_ALUSrc	s_EX_ALUSrc	NO	NO	
overflow_chk	NO	s_ID_overflow_chk	s_EX_overflow_chk	s_DMEM_overflow_chk	s_WB_overflow_chk	Overflow checked in last stage
branch_chk	NO	s_ID_branch_chk	NO	NO	NO	Branch control in ID stage
reg_DST_ADDR_SEL [1:0]	NO	s_ID_reg_DST_ADDR_SEL [1:0]	s_EX_reg_DST_ADDR_SEL [1:0]	NO	NO	Writeback in last stage
reg_DST_DATA_SEL [1:0]	NO	s_ID_reg_DST_DATA_SEL [1:0]	s_EX_reg_DST_DATA_SEL [1:0]	s_DMEM_reg_DST_DATA_SEL [1:0]	s_WB_reg_DST_DATA_SEL [1:0]	Writeback in last stage
PC_SEL[1:0]	NO	s_ID_PC_SEL[1:0]	NO	NO	NO	
reg_WE	NO	s_ID_reg_WE	s_EX_reg_WE	NO	NO	MUXed in ID stage and carried through as regWr
mem_WE	NO	s_ID_mem_WE	s_EX_mem_WE	s_DMEM_mem_WE	NO	
branch_SEL[2:0]	NO	s_ID_branch_SEL[2:0]	NO	NO	NO	
nAdd_Sub	NO	s_ID_nAdd_Sub	s_EX_nAdd_Sub	NO	NO	
shift_SEL[1:0]	NO	s_ID_shift_SEL[1:0]	s_EX_shift_SEL[1:0]	NO	NO	
logic_SEL[1:0]	NO	s_ID_logic_SEL[1:0]	s_EX_logic_SEL[1:0]	NO	NO	
out_SEL[1:0]	NO	s_ID_out_SEL[1:0]	s_EX_out_SEL[1:0]	NO	NO	

SW Pipeline Control Signals

Our Datapath Signals						
Datapath Signal	**IF (Instruction Fetch)**	**ID (Instruction Decode)**	**EX (Execute)**	**DMEM (Data Memory)**	**WB (Writeback)**	**Notes**
branch	NO	s_ID_branch	s_EX_branch	NO	NO	
PC [31:0]	s_IF_PC [31:0]	NO	NO	NO	NO	Assign to nextInstrAddr
PC_4 [31:0]	s_IF_PC_4 [31:0]	s_ID_PC_4 [31:0]	s_EX_PC_4 [31:0]	s_DMEM_PC_4 [31:0]	s_WB_PC_4 [31:0]	Has to propogate through for writeback for jal to set r[31] = PC + 4
INSTR [31:0]	NO	s_ID_INSTR [31:0]	NO	NO	NO	Use given Instr signal for IF
OPCODE [5:0]	NO	s_ID_OPCODE [5:0]	NO	NO	NO	
FUNCT [5:0]	NO	s_ID_FUNCT [5:0]	NO	NO	NO	
rt_ADDR [4:0]	NO	s_ID_rt_ADDR [4:0]	s_EX_rt_ADDR [4:0]	NO	NO	
rs_ADDR [4:0]	NO	s_ID_rs_ADDR [4:0]	NO	NO	NO	
rd_ADDR [4:0]	NO	s_ID_rd_ADDR [4:0]	s_EX_rd_ADDR [4:0]	NO	NO	Needed for WB
SHAMT [4:0]	NO	s_ID_SHAMT [4:0]	s_EX_SHAMT [4:0]	NO	NO	
IMM [15:0]	NO	s_ID_IMM [15:0]	NO	NO	NO	Not needed since sign extender in ID and turns into sign extended immediate 32 bit
J_ADDR [25:0]	NO	s_ID_J_ADDR [25:0]	NO	NO	NO	
rs_DATA [31:0]	NO	s_ID_rs_DATA [31:0]	s_EX_rs_DATA [31:0]	NO	NO	
rt_DATA [31:0]	NO	s_ID_rt_DATA [31:0]	s_EX_rt_DATA [31:0]	s_DMEM_rt_DATA [31:0]	NO	DMemData used in DMEM stage
ALUOut [31:0]	NO	NO	s_EX_ALUOut [31:0]	s_DMEM_ALUOut [31:0]	s_WB_ALUOut [31:0]	DMemAddr used in DMEM stage. When to assign for output? WB?
rt_DATA_MUX [31:0]	NO	NO	s_EX_rt_DATA_MUX [31:	NO	NO	MUX for second ALU operand between RT DATA and EXT IMM
IMM_EXT [31:0]	NO	s_ID_IMM_EXT [31:0]	s_EX_IMM_EXT [31:0]	NO	NO	
overflow	NO	NO	s_EX_overflow	s_DMEM_overflow	s_WB_overflow	Overflow should propogate to end of final stage (WB)
DMEM_DATA [31:0]	NO	NO	NO	s_DMEM_DMEM_DATA [3	s_WB_DMEM_DATA [31	Needed between EX/MEM buffer to writeback MUX
MUX'd signals						
RegWr	NO	NO	s_EX_RegWr	s_DMEM_RegWr	s_WB_RegWr	Final write signal, MUXXED out
RegWrAddr [4:0]	NO	NO	s_EX_RegWrAddr [4:0]	s_DMEM_RegWrAddr [4:0	s_WB_RegWrAddr [4:0]	MUX with rd_ADDR, rt_ADDR, and decimal 31 ($ra reg)

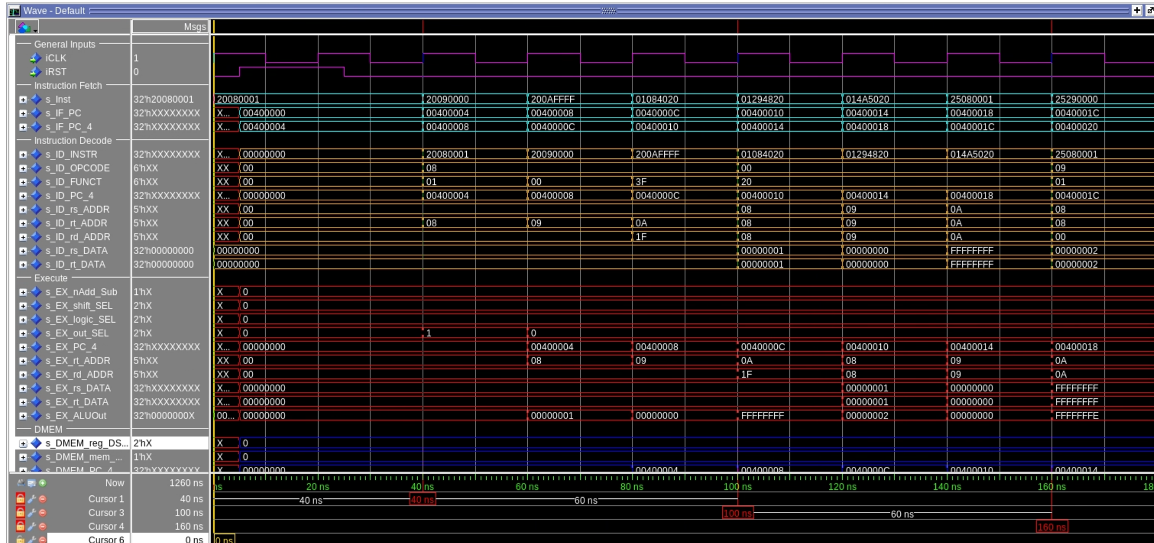
SW Pipeline Datapath Signals

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

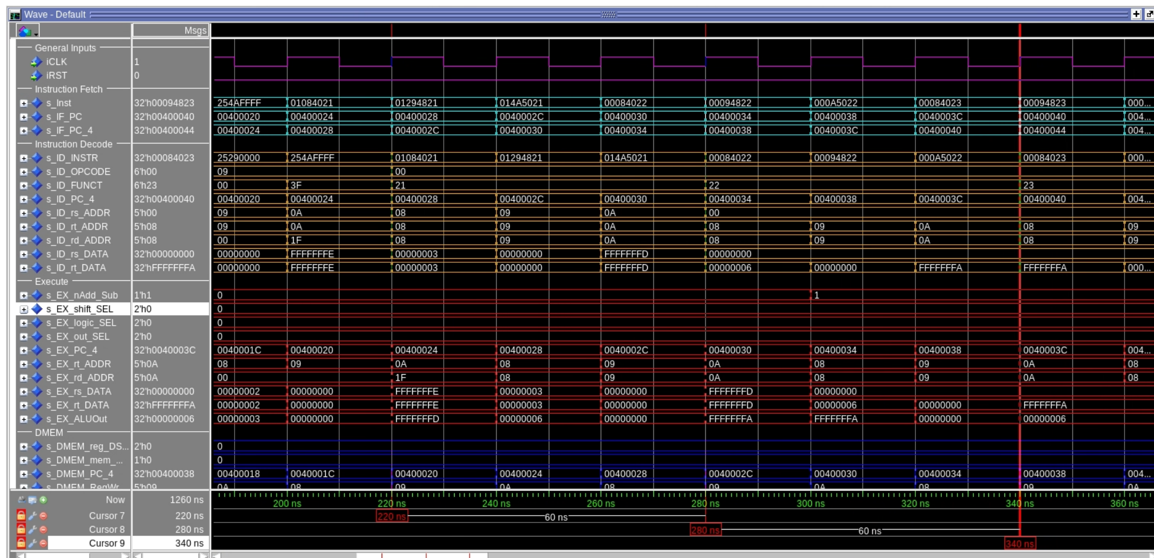


[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

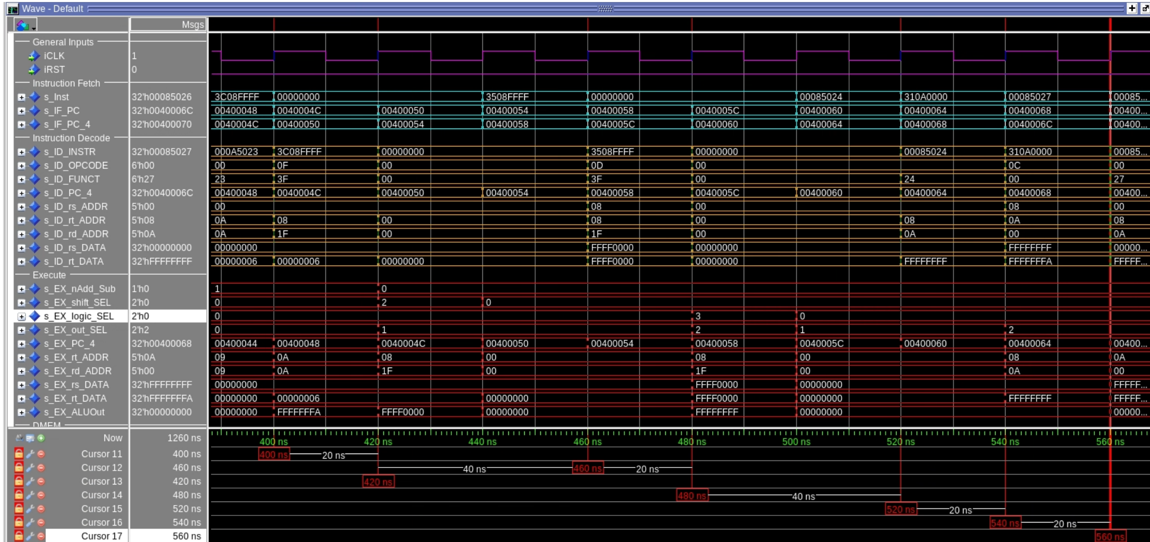
The assembly file ran is titled Proj2SW_cf_test.s



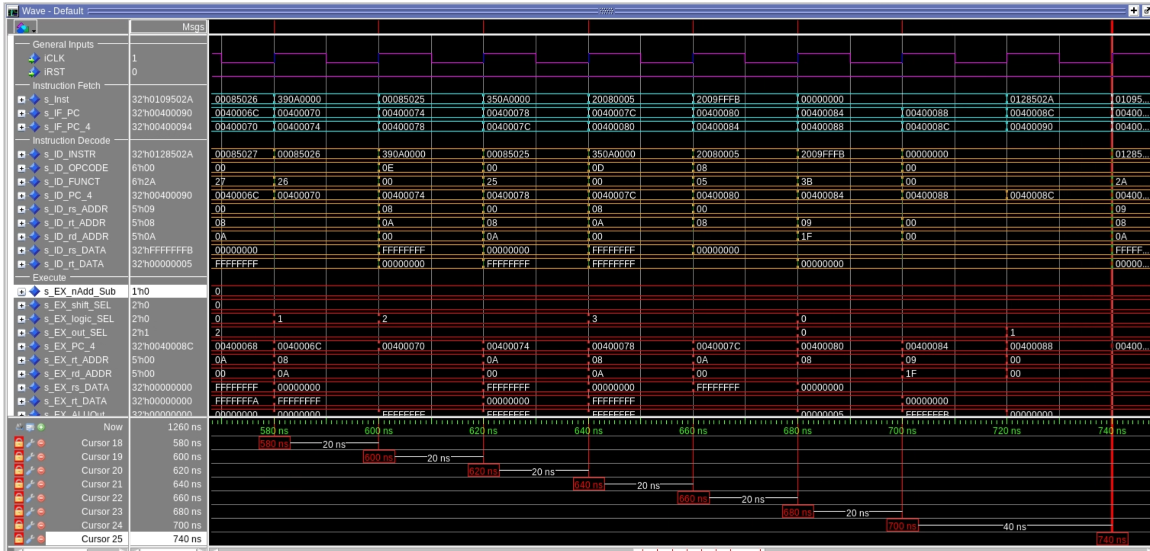
40ns- addi
100ns- add
160ns- addiu



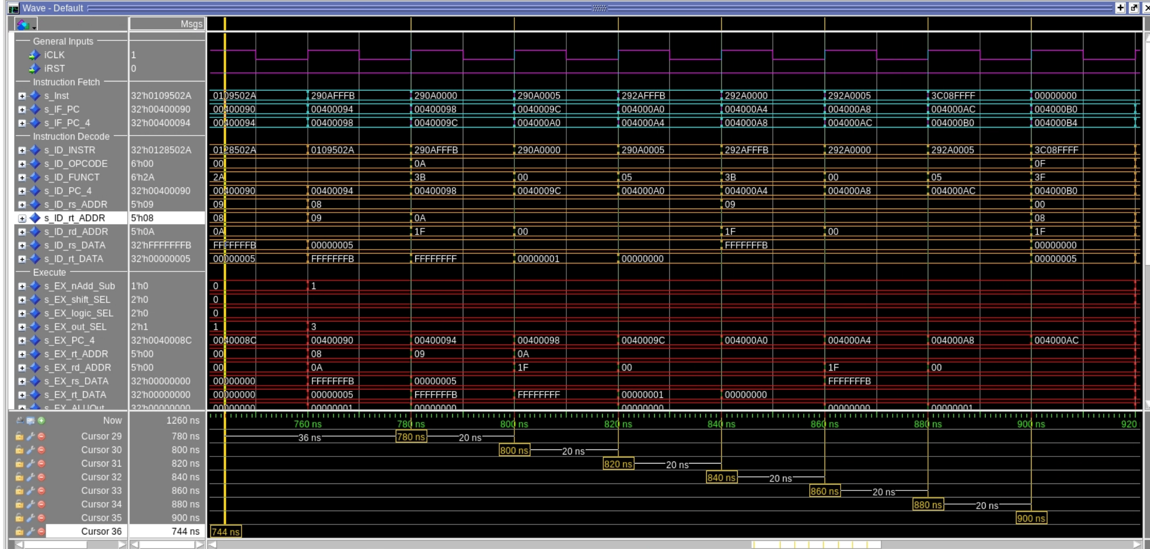
220ns- addu
280ns- sub
340ns- subu



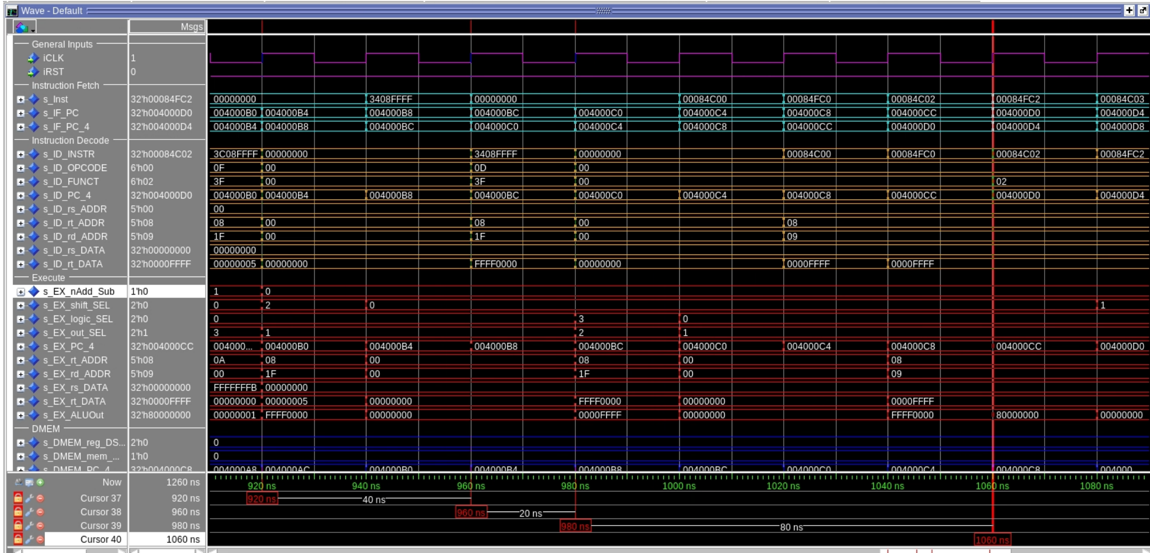
400ns- lui
420ns- nop
460ns- ori
480ns- nop
520ns- and
540ns- andi
560ns- nor



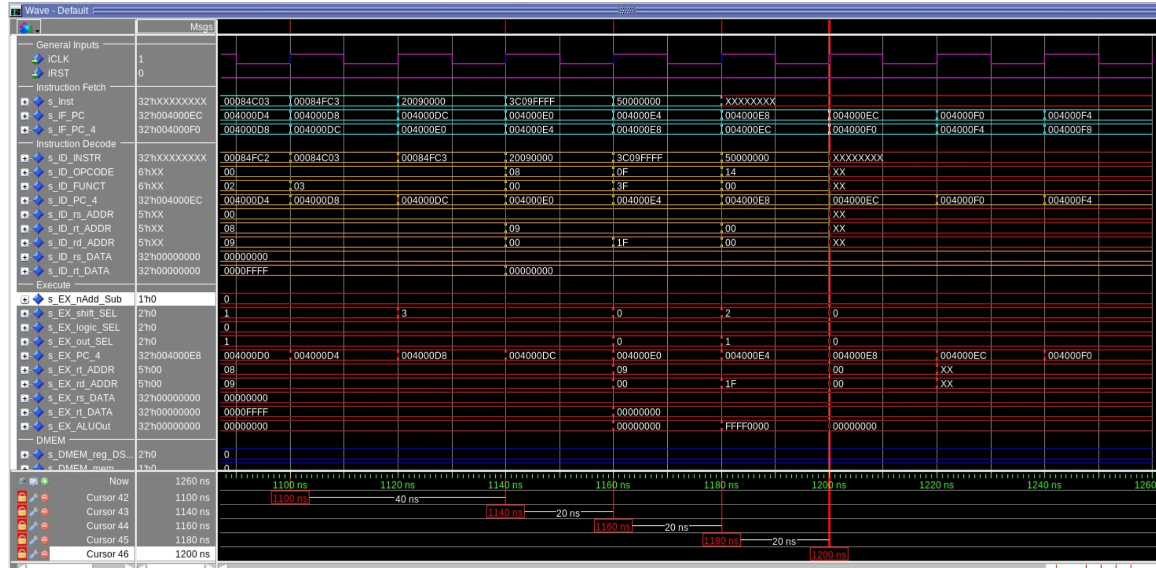
580ns- xor
600ns- xori
620ns- or
640ns- ori
660ns- andi
700ns- nop
740ns- slt



780ns- slti
900ns- lui



920ns- nop
960ns- ori
980ns - nop
1020ns- sll
1060ns- srl



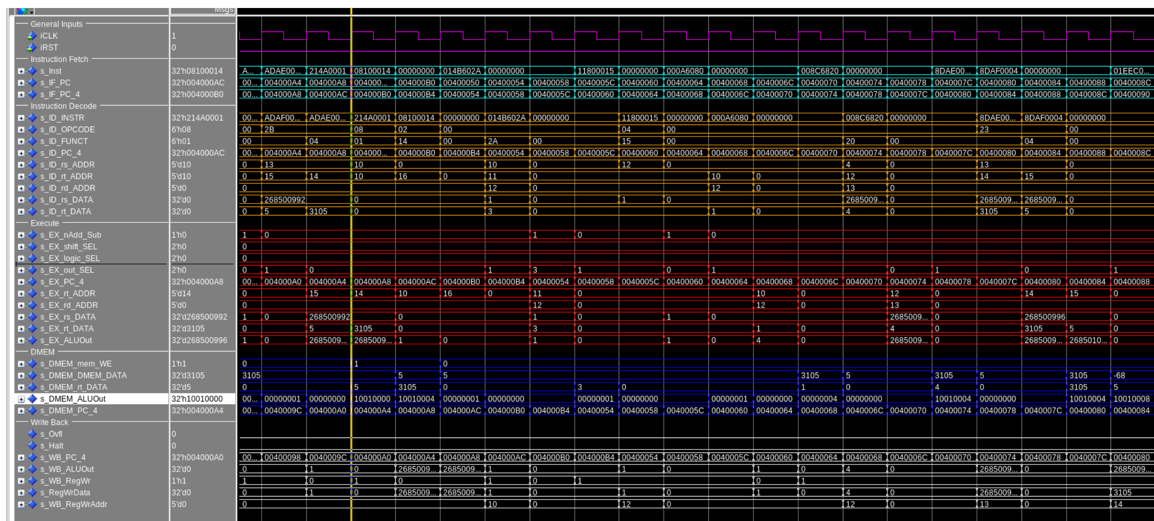
1100ns- sra
1140ns- addi
1160ns- lui
1180ns- halt

[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.

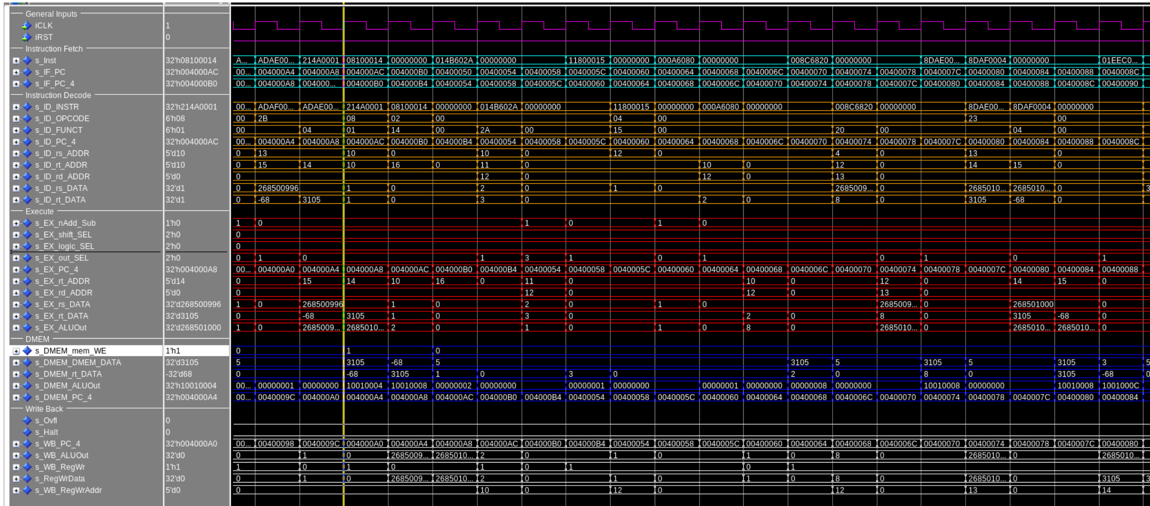
The assembly file ran is Proj2SW_bubblesort.s

	0x0000	0x0004	0x0008	0x000c
step	array[0]	array[1]	array[2]	array[3]
null	3105	5	-68	3
0	5	-68	3	3105
1	-68	3	5	3105
2	-68	3	5	3105
3	-68	3	5	3105

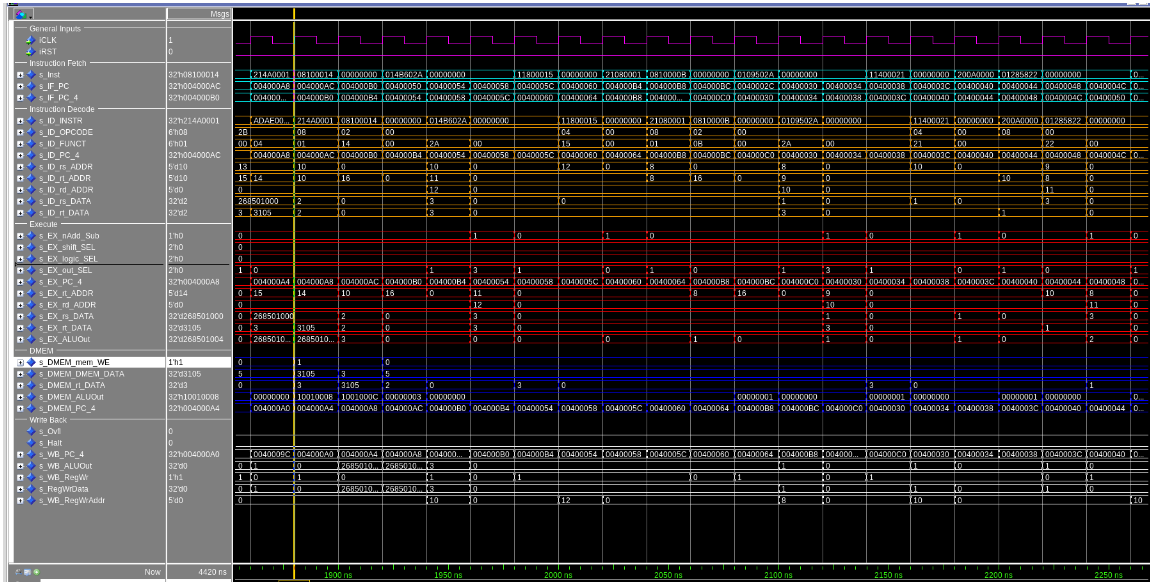
Each step in the above table represents a full sweep through each array value to compare with the next index, to see if it is larger than the following array index. If it is the case, then the two values swap positions and are rewritten into the data memory with swapped indices. The following five screenshots show when s_DMEM_mem_WE is asserted to 1, indicating a write to the data memory of the array has occurred. We can prove our design works as intended by confirming that the smaller of the two compared array values is written first to the lower array memory address, and in turn getting "sorted" to a lower index. Given the above table, this swap occurs 5 times between the first two loops. Each screenshot below shows one swap at the cursor position, taking two consecutive instructions to write back to the data memory.



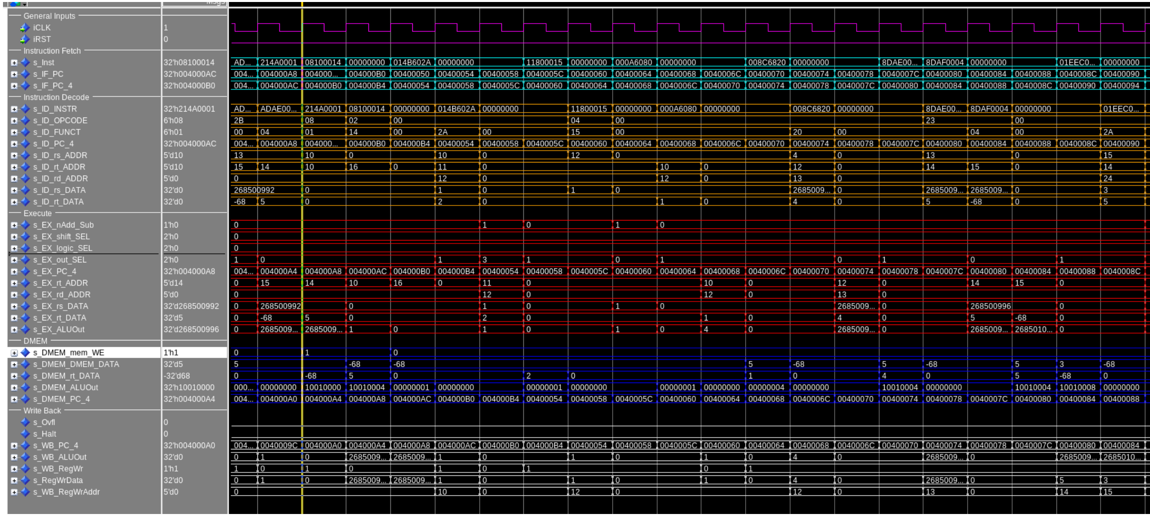
Bubble Sort Array Swap 1



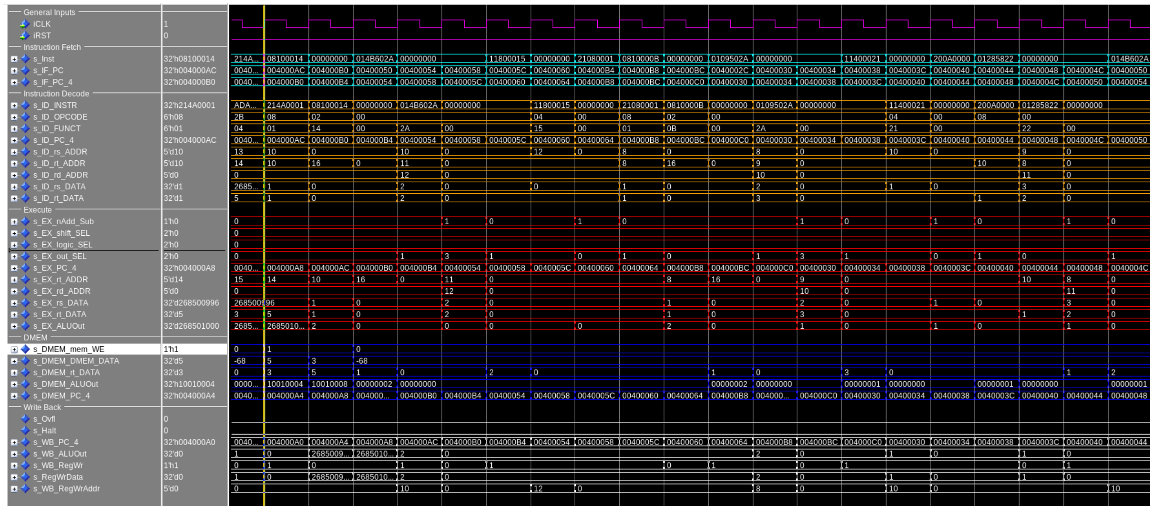
Bubble Sort Array Swap 2



Bubble Sort Array Swap 3



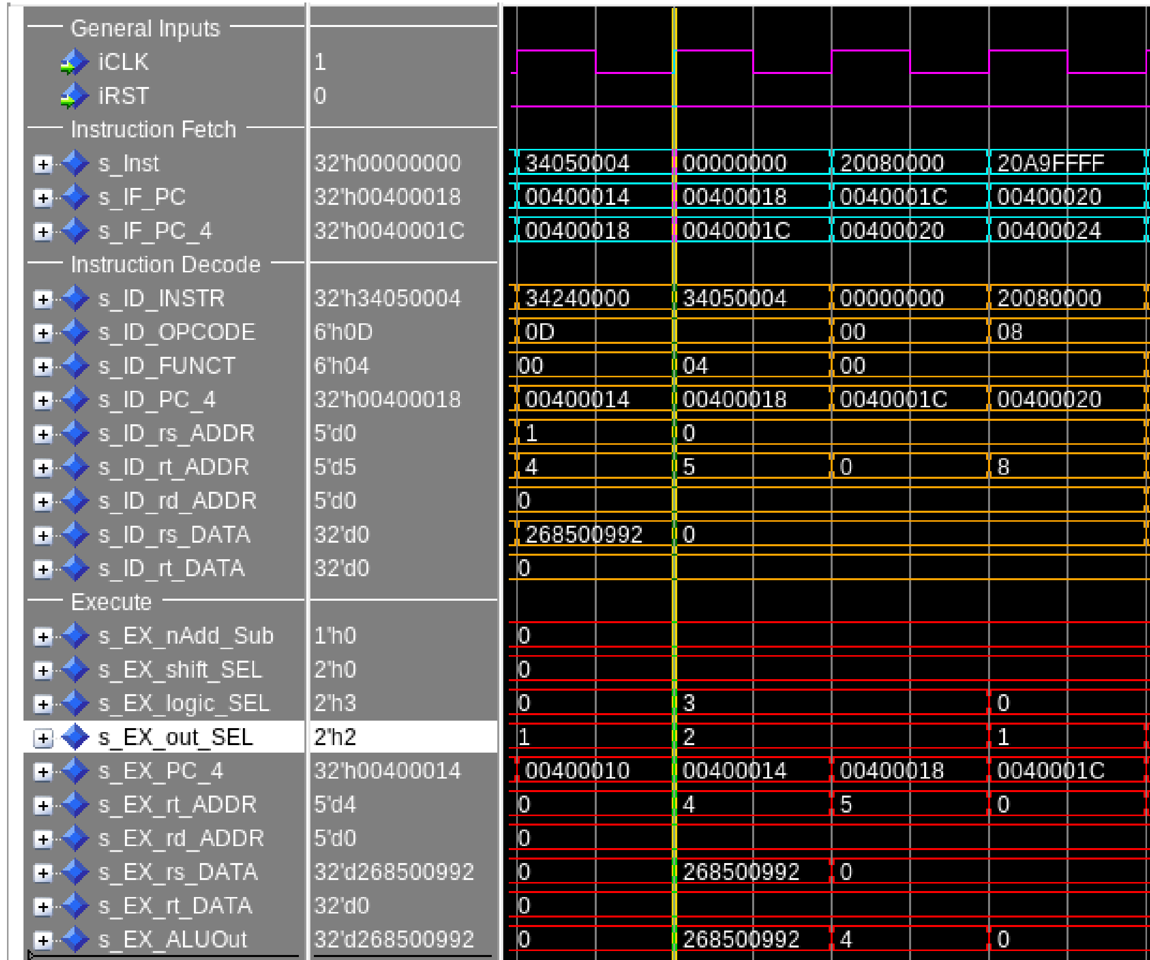
Bubble Sort Array Swap 4



Bubble Sort Array Swap 5

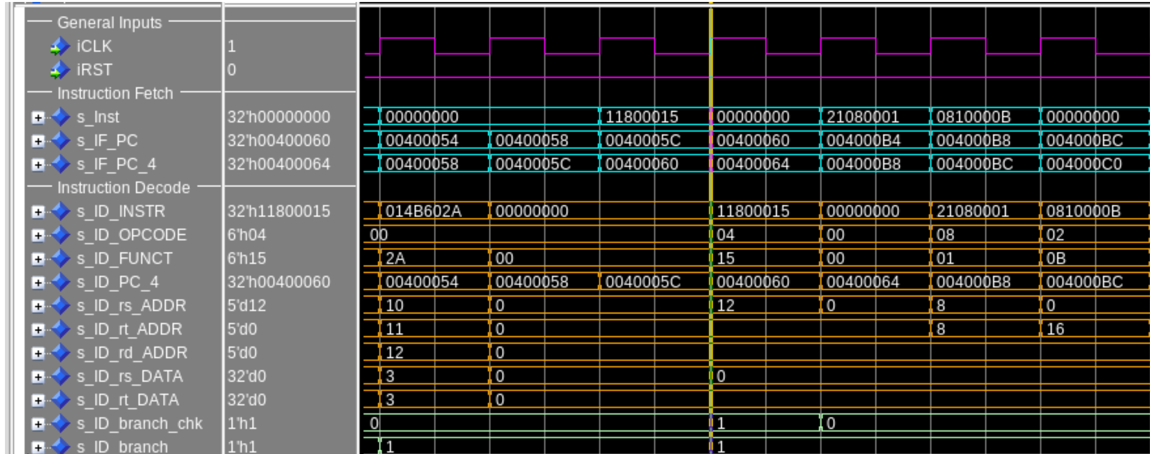
In the below screenshot, the instruction decoded in the ID stage following the cursor is the ori instruction, described by the s_ID_OPCODE signal being 0x0D. After this instruction, we can see that only one NOP instruction is decoded next as s_ID_INSTR is 0x00000000. This shows that we did not use 5 NOP instructions for every single signal. We updated our register file such that the register read would output the incoming memory that was being written before the clock cycle, which reduced our potential of

NOP’s from 3 to 2 for any arithmetic or ALU operation. In other cases, like seen below, it can still help to reduce some NOP’s from 2 down to 1, or even 1 to using none!



Bubble Sort Data Flow NOP Proof

Based on the below screenshot, it can be seen that we are branching because the s_ID_branch_chk control signal is asserted to 1 after the cursor. After the branch instruction, it can be seen that the next decoded instruction is 0x00000000, which is a NOP. After the NOP, the program resumes with a regular instruction that is intended for the Bubble Sort algorithm. Since we moved our branch module from the ALU in the EX stage to the top level in the ID stage, we only need to wait for one cycle with a NOP to determine a branch, and in turn find the next PC.



Bubblesort Control Flow NOP Proof

[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).

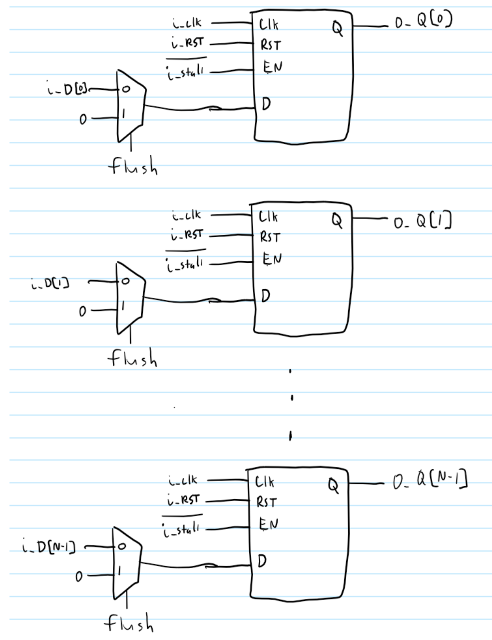
Our maximum frequency reported was 55.05 MHz.
The critical path follows as:

1. Data Memory
2. Reg Address MUX
3. Register File
4. Branch module
5. Prefetch Module to update PC

The longest path occurred in our Instruction Decode stage. The path followed decoding a given register, and passing that along to the branch module which determined if a branch should be made or not. Then, the PC finally got updated. We moved the branch conditional check outside of the ALU and into the ID stage to help prevent less NOP's in our test code and stalling later on. Now, we can see the tradeoff of having to wait for the register file contents for the RS and RT DATA to propagate and output from the register file before checking the branch conditions. For project 3, we could analyze the tradeoffs on reducing the CPI of branch instructions by 1 versus the added propagation delay after the register file in our instruction decode stage. It could be the case that the next slowest propagation is only slightly faster, in which case would lead to a much better benefit in keeping the conditional branch module in the ID phase, to reduce CPI.

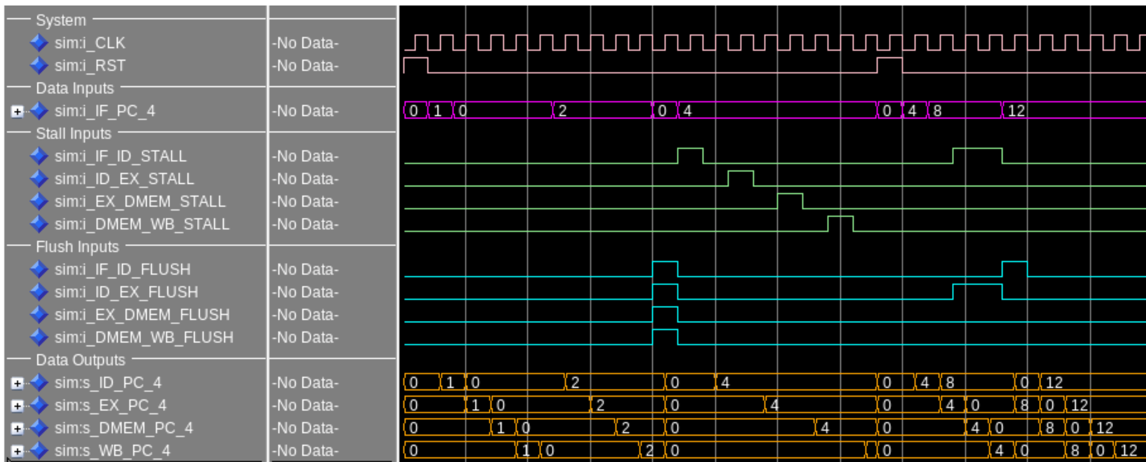
[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

The following example depicts an N-bit register that has both stalling and flushing operations. To stall a pipeline register for one cycle, the i_stall input control signal will be asserted to 1. This means that each DFF in the pipeline register will not be written on the next positive edge clock cycle, instead HOLDING it's previous contents. To flush, we will MUX 0's in for every bit so that on the POSITIVE EDGE of the next clock cycle, the pipeline register will be filled with 0's and not affect any registers or memory.



[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 following waveform depicts our testbench that instantiates all four pipeline register buffers in a single testbench. For this test, we only used one sample datapath signal that propagates throughout each pipeline register, which was PC_4. Initially, we began by buffering decimal 1 through the IF_ID_PC_4 pipeline register, and then waiting 4 clock cycles to confirm decimal 1 is propagated through each pipeline register. The order of the pipeline registers follows the stages between them, following IF_ID, ID_EX, EX_DMEM, and finally DMEM_WB. If we write a signal decimal 2 and hold that value through multiple clock cycles, 2's are confirmed to be filled through each pipeline register as expected. When we wrote decimal 4 for IF_PC_4, we drove the stall control for each pipeline register on the clock cycle; it was expected to write a new value for PC_4, indicating that the old contents of 0 were not written over. Finally, we simulated a condition that had two data hazards and a control hazard, which led to two stalls and then a flush. This was very beneficial to test since a sample set of instructions with a read/write data dependency and a flush was how we derived the logic for our data hazard module.



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

Instruction	Format	Decode	Produce Execute	Produce Demem	Produce Writeback
add	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
addi	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
addiu	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
addu	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
and	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
andi	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
lui	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
lw	I	-	s_EX_ALUOut	s_DMEM_DMEM_DATA	s_RegWrData
nor	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
xor	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
xori	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
or	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
ori	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
slt	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
slti	I	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
sll	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
srl	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
sra	R	-	s_EX_ALUOut	s_DMEM_ALUOut	s_RegWrData
sw	I	-	s_EX_ALUOut	-	-
sub	R	-	s_EX_ALUOut	s_EX_ALUOut	s_RegWrData
subu	R	-	s_EX_ALUOut	s_EX_ALUOut	s_RegWrData
beq	I	s_ID_branch	-	-	-
bne	I	s_ID_branch	-	-	-
j	J	s_ID_IMM_EXT	-	-	-
jal	J	s_ID_IMM_EXT	s_EX_PC_4	s_DMEM_PC_4	s_RegWrData
jr	R	s_ID_IMM_EXT	-	-	-
bgez	I	s_ID_IMM_EXT	-	-	-
bgezal	I	s_ID_branch	s_EX_PC_4	s_DMEM_PC_4	-
bgtz	I	s_ID_branch	-	-	-
blez	I	s_ID_branch	-	-	-
bltzal	I	s_ID_branch	s_EX_PC_4	s_DMEM_PC_4	-
bltz	I	s_ID_branch	-	-	-
halt		s_ID_Halt,	s_EX_Halt,	s_DMEM_Halt,	s_WB_Halt,

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

Instruction	Format	Decode	Consume Decode	Consume Decode
add	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
addi	I	-	s_ID_rs_DATA	-
addiu	I	-	s_ID_rs_DATA	-
addu	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
and	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
andi	I	-	s_ID_rs_DATA	-
lui	I	-	s_ID_rs_DATA	-
lw	I	-	s_ID_rs_DATA	-
nor	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
xor	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
xori	I	-	s_ID_rs_DATA	-
or	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
ori	I	-	s_ID_rs_DATA	-
slt	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
slti	I	-	s_ID_rs_DATA	-
sll	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
srl	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
sra	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
sw	I	-	s_ID_rs_DATA	s_ID_rt_DATA,
sub	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
subu	R	-	s_ID_rs_DATA	s_ID_rt_DATA,
beq	I	s_ID_branch	s_ID_rs_DATA	s_ID_rt_DATA,
bne	I	s_ID_branch	s_ID_rs_DATA	s_ID_rt_DATA,
j	J	s_ID_IMM_EXT	-	-
jal	J	s_ID_IMM_EXT	-	-
jr	R	s_ID_IMM_EXT	s_ID_rs_DATA	-
bgez	I	s_ID_IMM_EXT	s_ID_rs_DATA	-
bgezal	I	s_ID_branch	s_ID_rs_DATA	-
bgtz	I	s_ID_branch	s_ID_rs_DATA	-
blez	I	s_ID_branch	s_ID_rs_DATA	-
bltzal	I	s_ID_branch	s_ID_rs_DATA	-
bltz	I	s_ID_branch	s_ID_rs_DATA	-
halt		s_ID_Halt,	-	-

[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.

	Consuming Signals				
	EX RS ALU INPUT	EX RT ALU INPUT	DMEM DATA INPUT		
Producing Signal	s_EX_RS_DATA_MUX_FWD	s_EX_RT_DATA_MUX_FWD	s_EX_DMEM_DATA_MUX_FWD	Branch	JR
s_EX_RS_DATA [31:0]	s_EX_RS_MUX_FWD_SEL = 00	NO	NO	NOT ADDRESSED	NOT ADDRESSED
s_EX_RT_DATA_MUX [31:0]	NO	s_EX_RT_MUX_FWD_SEL = 00	s_DMEM_DATA_FWD_SEL = 00	NOT ADDRESSED	NOT ADDRESSED
s_DMEM_ALUOut [31:0]	s_EX_RS_MUX_FWD_SEL = 10	s_EX_RT_MUX_FWD_SEL = 10	s_DMEM_DATA_FWD_SEL = 10	NOT ADDRESSED	NOT ADDRESSED
s_RegWrData	s_EX_RS_MUX_FWD_SEL = 01	s_EX_RT_MUX_FWD_SEL = 01	s_DMEM_DATA_FWD_SEL = 01	NOT ADDRESSED	NOT ADDRESSED
Notes				Would need to be forwared to the comparator module	Would need to forward both producing signals to jump adder

Legend
EX
DMEM
WB

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

Control Signal						
Control Signal	IF (Instruction Fetch)	ID (Instruction Decode)	EX (Execute)	DMEM (Data Memory)	WB (Writeback)	Notes
reg_WE_SEL	NO	s_ID_reg_WE_SEL	s_EX_reg_WE_SEL	NO	NO	
Halt	NO	s_ID_Halt	s_EX_Halt	s_DMEM_Halt	s_WB_Halt	
nZero_Sign	NO	s_ID_nZero_Sign	NO	NO	NO	Sign extender in ID stage
ALUSrc	NO	s_ID_ALUSrc	s_EX_ALUSrc	NO	NO	
overflow_chk	NO	s_ID_overflow_chk	s_EX_overflow_chk	s_DMEM_overflow_chk	s_WB_overflow_chk	Overflow checked in last stage
branch_chk	NO	s_ID_branch_chk	NO	NO	NO	Branch control in ID stage
reg_DST_ADDR_SEL [1:0]	NO	s_ID_reg_DST_ADDR_SEL [1:0]	s_EX_reg_DST_ADDR_SEL [1:0]	NO	NO	Writeback in last stage
reg_DST_DATA_SEL [1:0]	NO	s_ID_reg_DST_DATA_SEL [1:0]	s_EX_reg_DST_DATA_SEL [1:0]	s_DMEM_reg_DST_DATA_SEL [1:0]	s_WB_reg_DST_DATA_SEL [1:0]	Writeback in last stage
PC_SEL[1:0]	NO	s_ID_PC_SEL[1:0]	NO	NO	NO	
reg_WE	NO	s_ID_reg_WE	s_EX_reg_WE	NO	NO	MUXed in ID stage and carried through as regWr
mem_WE	NO	s_ID_mem_WE	s_EX_mem_WE	s_DMEM_mem_WE	NO	
branch_SEL[2:0]	NO	s_ID_branch_SEL[2:0]	NO	NO	NO	
nAdd_Sub	NO	s_ID_nAdd_Sub	s_EX_nAdd_Sub	NO	NO	
shift_SEL[1:0]	NO	s_ID_shift_SEL[1:0]	s_EX_shift_SEL[1:0]	NO	NO	
logic_SEL[1:0]	NO	s_ID_logic_SEL[1:0]	s_EX_logic_SEL[1:0]	NO	NO	
out_SEL[1:0]	NO	s_ID_out_SEL[1:0]	s_EX_out_SEL[1:0]	NO	NO	
Hazard Detection						
IF_ID_STALL	s_IF_ID_STALL	NO	NO	NO	NO	
ID_EX_STALL	NO	s_ID_EX_STALL	NO	NO	NO	
EX_DMEM_STALL	NO	NO	s_EX_DMEM_STALL	NO	NO	
DMEM_WB_STALL	NO	NO	NO	s_DMEM_WB_STALL	NO	
PC_STALL	s_PC_STALL	NO	NO	NO	NO	
IF_ID_FLUSH	s_IF_ID_FLUSH	NO	NO	NO	NO	
ID_EX_FLUSH	NO	s_ID_EX_FLUSH	NO	NO	NO	
EX_DMEM_FLUSH	NO	NO	s_EX_DMEM_FLUSH	NO	NO	
DMEM_WB_FLUSH	NO	NO	NO	s_DMEM_WB_FLUSH	NO	
Forwarding						
EX_RS_DATA_FWD_SEL	NO	s_EX_RS_DATA_FWD_SEL	NO	NO	NO	Select line for ALU RS forwarding
EX_RT_DATA_FWD_SEL	NO	s_EX_RT_DATA_FWD_SEL	NO	NO	NO	Select line for ALU RT forwarding
DMEM_DATA_FWD_SEL	NO	s_DMEM_DATA_FWD_SEL	NO	NO	NO	Select line for DMEM Data forwarding
LW_HAZARD_CHK	NO	s_LW_HAZARD_CHK	s_LW_HAZARD_CHK	NO	NO	indicator for SW in EX stage, indicating stall to hazard detection

Legend
Test
Top Level
Fetch Logic
ALU
Stall
Flush
Forwarding

HW Pipeline Control Signals

Our Datapath Signals						
Datapath Signal	IF (Instruction Fetch)	ID (Instruction Decode)	EX (Execute)	DMEM (Data Memory)	WB (Writeback)	Notes
branch	NO	s_ID_branch	s_EX_branch	NO	NO	
PC [31:0]	s_IF_PC [31:0]	NO	NO	NO	NO	Assign to nextInstrAddr
PC_4 [31:0]	s_IF_PC_4 [31:0]	s_ID_PC_4 [31:0]	s_EX_PC_4 [31:0]	s_DMEM_PC_4 [31:0]	s_WB_PC_4 [31:0]	Has to propogate through for writeback for jal to set r[31] = PC + 4
INSTR [31:0]	NO	s_ID_INSTR [31:0]	NO	NO	NO	Use given Instr signal for IF
OPCODE [5:0]	NO	s_ID_OPCODE [5:0]	s_EX_OPCODE [5:0]	NO	NO	
FUNCT [5:0]	NO	s_ID_FUNCT [5:0]	s_EX_FUNCT [5:0]	NO	NO	
rt_ADDR [4:0]	NO	s_ID_rt_ADDR [4:0]	s_EX_rt_ADDR [4:0]	NO	NO	rt_ADDR needed in EX for forwarding
rs_ADDR [4:0]	NO	s_ID_rs_ADDR [4:0]	s_EX_rs_ADDR [4:0]	NO	NO	rs_ADDR needed in EX for forwarding
rd_ADDR [4:0]	NO	s_ID_rd_ADDR [4:0]	s_EX_rd_ADDR [4:0]	NO	NO	Needed for WB
SHAMT [4:0]	NO	s_ID_SHAMT [4:0]	s_EX_SHAMT [4:0]	NO	NO	
IMM [15:0]	NO	s_ID_IMM [15:0]	NO	NO	NO	Not needed since sign extender in ID and turns into sign extended immediate 32 bit
J_ADDR [25:0]	NO	s_ID_J_ADDR [25:0]	NO	NO	NO	
rs_DATA [31:0]	NO	s_ID_rs_DATA [31:0]	s_EX_rs_DATA [31:0]	NO	NO	
rt_DATA [31:0]	NO	s_ID_rt_DATA [31:0]	s_EX_rt_DATA [31:0]	s_DMEM_rt_DATA [31:0]	NO	DMemData used in DMEM stage
ALUOut [31:0]	NO	NO	s_EX_ALUOut [31:0]	s_DMEM_ALUOut [31:0]	s_WB_ALUOut [31:0]	DMemAddr used in DMEM stage. When to assign for output? WB?
rt_DATA_MUX [31:0]	NO	NO	s_EX_rt_DATA_MUX [31:0]	NO	NO	MUX for second ALU operand between RT DATA and EXT IMM
IMM_EXT [31:0]	NO	s_ID_IMM_EXT [31:0]	s_EX_IMM_EXT [31:0]	NO	NO	
overflow	NO	NO	s_EX_overflow	s_DMEM_overflow	s_WB_overflow	Overflow should propogate to end of final stage (WB)
DMEM_DATA [31:0]	NO	NO	NO	s_DMEM_DMEM_DATA [31:0]	s_WB_DMEM_DATA [31	Needed between EX/MEM buffer to writeback MUX
MUX'd signals						
RegWr	NO	NO	s_EX_RegWr	s_DMEM_RegWr	s_WB_RegWr	Final write signal, MUXXED out
RegWrAddr [4:0]	NO	NO	s_EX_RegWrAddr [4:0]	s_DMEM_RegWrAddr [4:0]	s_WB_RegWrAddr [4:0]	MUX with rd_ADDR, rt_ADDR, and decimal 31 ($ra reg)
MUX'd Forwarding signals						
RS_DATA_FWD_MUX [31:0]	NO	s_ID_RS_DATA_FWD_MUX [31	NO	NO	NO	
RT_DATA_FWD_MUX [31:0]	NO	s_ID_RT_DATA_FWD_MUX [31	NO	NO	NO	
DMEM_DATA_MUX_FWD [31:0]	NO	NO	s_EX_DMEM_DATA_MUX_FWD [31:0]	s_DMEM_DMEM_DATA_MUX_FWD [31:0]	NO	

Legend
fetch
Instruction Decode
ALU
RT DATA MUX
Sign Extender
Overflow
Data Memory
MUX

HW Pipeline Datapath Signals

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

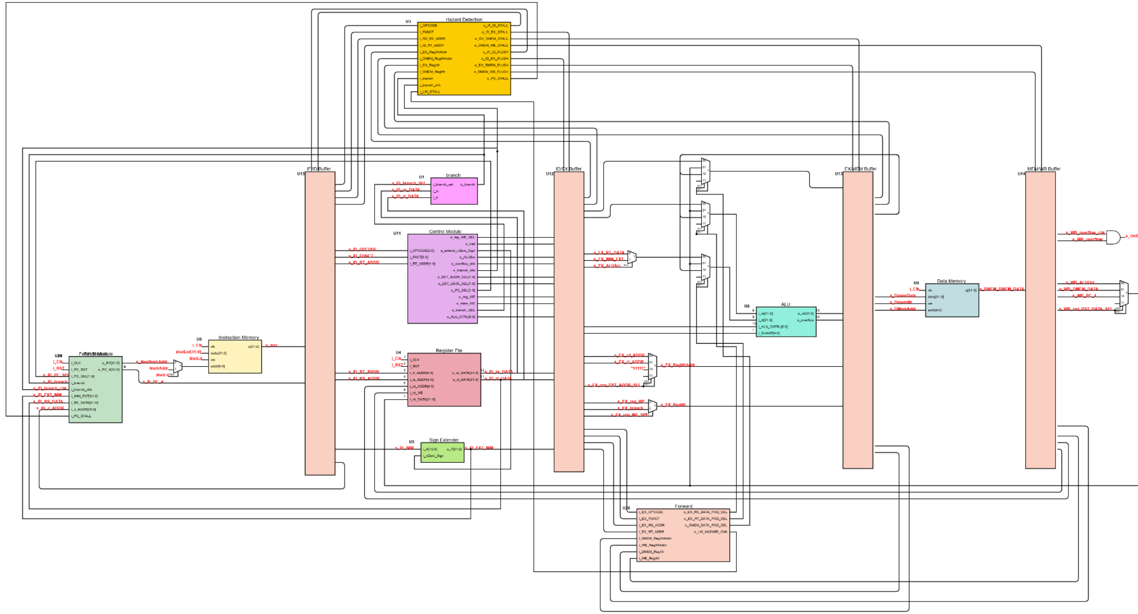
beq- decode
bne- decode
j- decode
jal- decode
jr- decode
bgez- decode
bgezal- decode
bgtz- decode
blez- decode
bltzal- decode
bltz- decode

[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.

beq- if branch taken instruction IF_ID flush
bne- if branch taken instruction IF_ID flush
j- instruction IF_ID flush
jal- instruction IF_ID flush
jr- instruction IF_ID flush
bgez- if branch taken instruction IF_ID flush
bgezal- if branch taken instruction IF_ID flush
bgtz- if branch taken instruction IF_ID flush
blez- if branch taken instruction IF_ID flush
bltzal- if branch taken instruction IF_ID flush
bltz- if branch taken instruction IF_ID flush

Any time an unconditional or conditional branch is taken, we will flush the IF_ID register ONLY, since we have moved our branch checking to the ID stage. We load PC+4 after the branch and jump instructions, predicting that the branch will NOT be taken, and will NOT flush the instruction if it is not.

[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.

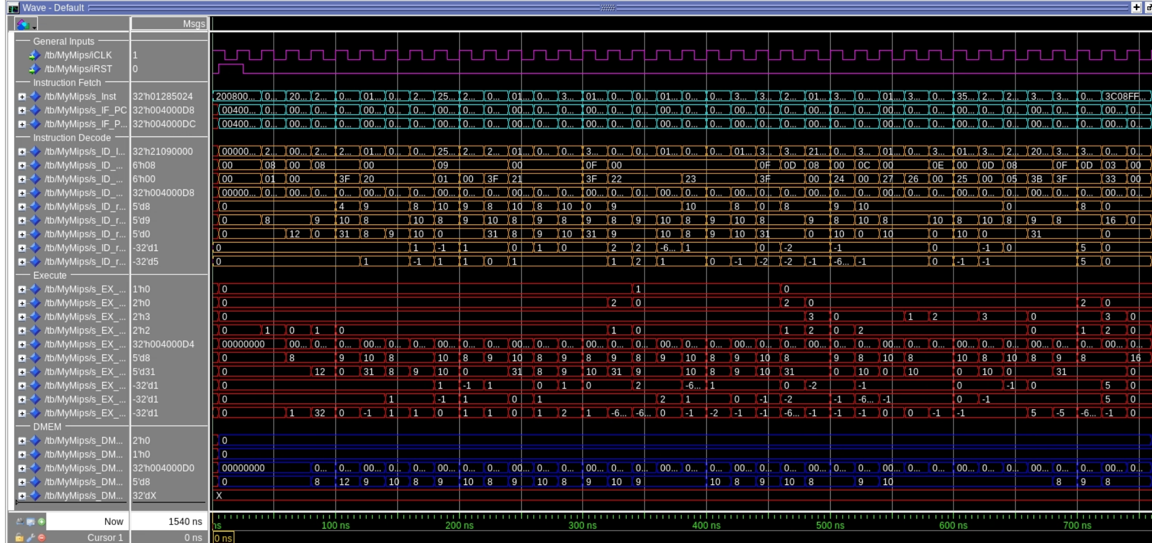


[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table. Show the Questasim output for the following test

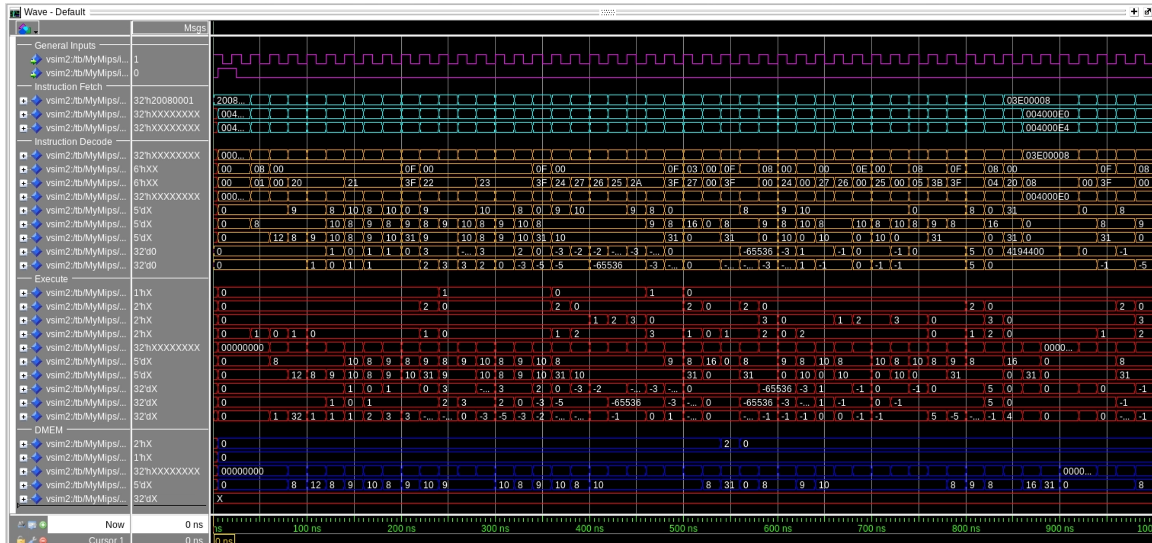
Test data forwarding and hazard detection here

Instruction	TEST CASE	Reasoning
add	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
addi	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX
addiu	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX
addu	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
and	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
andi	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX
lui	ALU_RS, ALU_RT	No data dependencies can only cause them
lw	DMEM	Has delayed producing timing from all other commands
nor	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
xor	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
xori	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX
or	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
ori	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX
slt	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
slti	ALU_RS	Forwarding From DMEM ALU out or WB Mux to EX
sll	ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
srl	ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
sra	ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
sw	DMEM	Has different consuming points from ALU operations
sub	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
subu	ALU_RS, ALU_RT	Forwarding From DMEM ALU out or WB Mux to EX
beq	Control/Jump	Stall for all data dependencies
bne	Control/Jump	Stall for all data dependencies
j	NA	No Data dependies
jal	ALU_RS, ALU_RT	Can cause data dependencies
jr	ALU_RS, ALU_RT	Needs stalling from data dependencies
bgez	Control/Jump	Stall for all data dependencies
bgezal	Control/Jump	Stall for all data dependencies
bgtz	Control/Jump	Stall for all data dependencies
blez	Control/Jump	Stall for all data dependencies
bltzal	Control/Jump	Stall for all data dependencies
bltz	Control/Jump	Stall for all data dependencies
halt	NA	No data dependencies

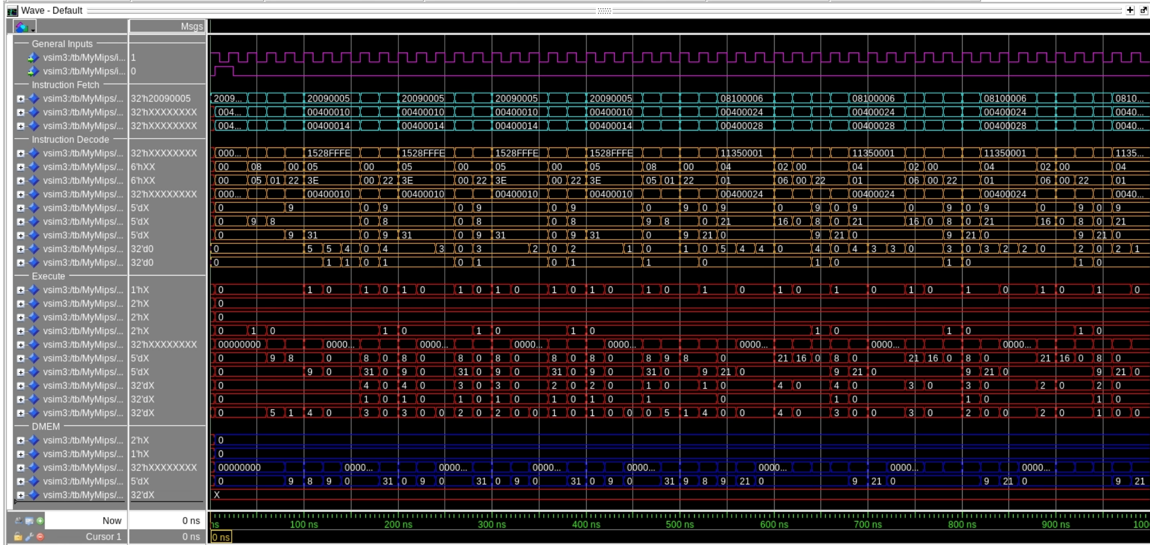
Data Hazard Test Cases



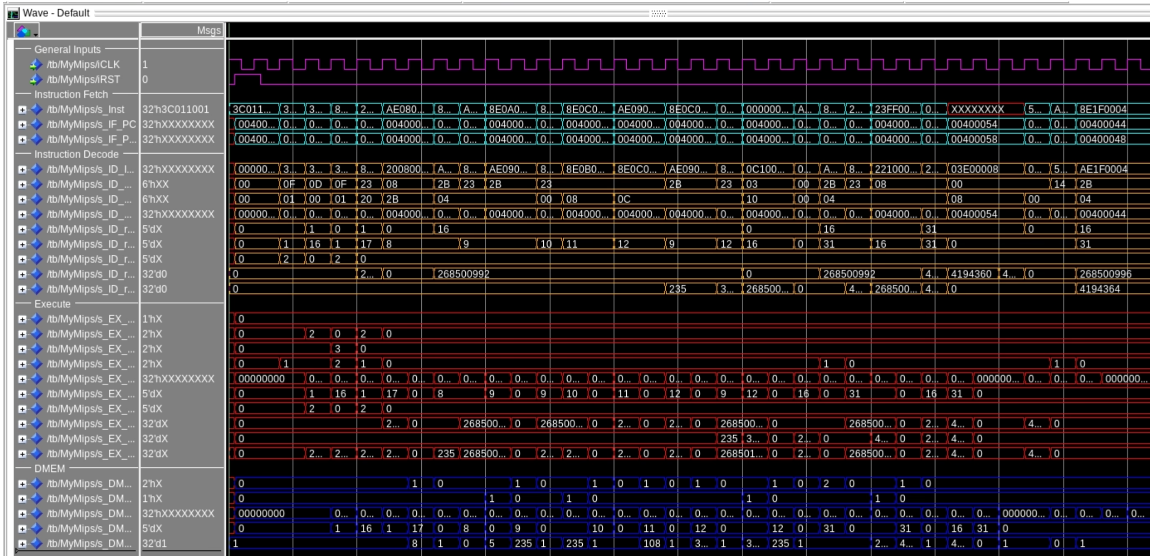
ALU_RS Data Hazard Test Cases



ALU_RT Data Hazard Test Cases



Control/Jump Data Hazard Test Cases



DMEM Data Hazard Test Case

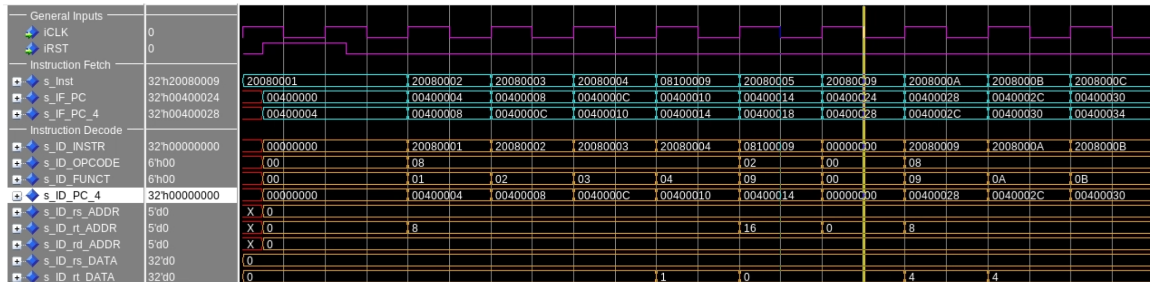
[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table. Show the Questasim output for the following test

Create a set of assembly programs that exhaustively tests control hazard avoidance. Minimally include one test program per control flow instruction. Then you should create a set of test programs that activate combinations of these instructions in the pipeline.

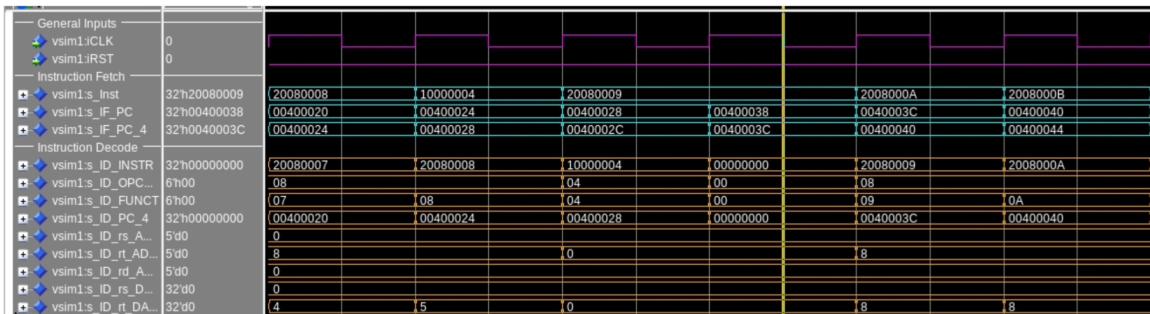
Instruction	Assembly File	Branch Taken	Branch Not Taken
j	j_controlHazard.s	Flush IF/ID	N/A
jal	jal_controlHazard.s	Flush IF/ID	N/A
jr	jr_controlHazard.s	Flush IF/ID	N/A
beq	beq_controlHazard.s	Flush IF/ID	No Action
	branch_explosion_controlHazard.s	Flush IF/ID	No Action
bne	bne_controlHazard.s	Flush IF/ID	No Action
bgez	bgez_controlHazard.s	Flush IF/ID	No Action
bgezal	bgezal_controlHazard.s	Flush IF/ID	No Action
bgtz	bgtz_controlHazard.s	Flush IF/ID	No Action
blez	blez_controlHazard.s	Flush IF/ID	No Action
bltzal	bltzal_controlHazard.s	Flush IF/ID	No Action
bltz	bltz_controlHazard.s	Flush IF/ID	No Action

Control Hazard Test Cases

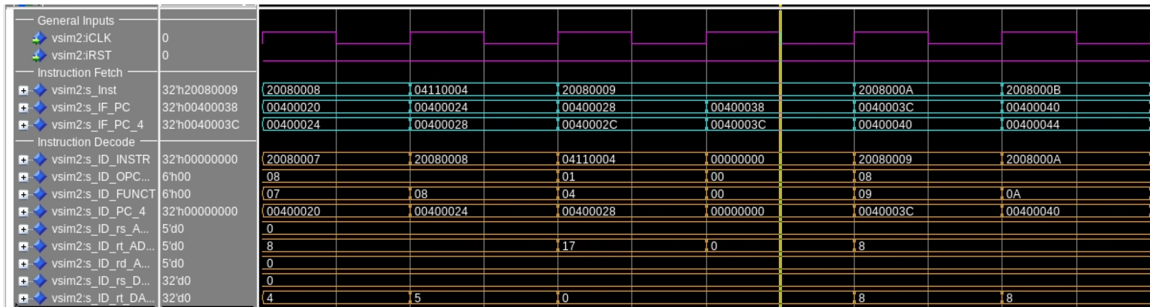
The following waveforms demonstrate that when a control hazard occurs, the IF/ID pipeline register flushes all of its contents. The control hazard occurs when a branch is taken, since we will already be fetching the PC + 4 instruction after the branch or jump instruction. We designed our prediction module to not flush the IF/ID Buffer register when we do not branch, to help improve our CPI. In the following three waveforms, near the cursor, it can be seen that one NOP is propagated ONLY when a branch is taken.



J Instruction Control Hazard Test Case

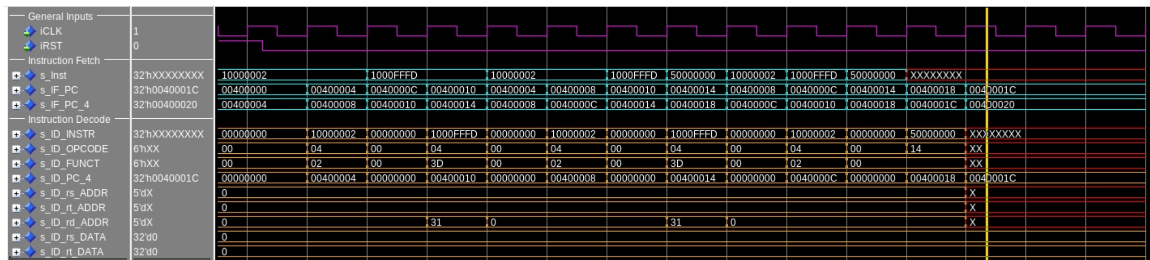


BEQ Instruction Control Hazard Test Case



BGEZAL Instruction Control Hazard Test Case

In our final test, we created the assembly file branch_explosion_controlHazard.s, which simulates 5 consecutive branch taken conditions in a row. Amazingly, this passed by the end of everything. As with our other validation and waveforms, the instruction that was fetched in the IF stage after the branch instruction would get flushed on every branch taken, as seen below in the following waveforms.



Multiple Consecutive Branch Instructions Control Hazard Test Case

[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 WITH forwarding: 48.66 MHz

Critical Path:

1. Data Memory (?)
2. MUX to ALU
3. ALU (Ripple Adder)
4. ALU (Ripple SLL)
5. EX/DMEM Pipeline Register (MUX to write)

We assumed that the critical path changed from our ID stage to our EX stage due to adding more multiplexers from the pipeline registers to the ALU inputs, leading to the increased timing delay before the ALU processes its operations. Similar to the single cycle, the ALU ripple carry adder led to a very high propagation delay, leading to our critical path again once we added more hardware for forwarding. Despite this, our CPI dropped to an average of 1.50 for our tests, which is a good tradeoff for decreasing the maximum frequency by around 6 MHz from a CPI average of near 2.1.