CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: _____Owen Jewell_____

_____Corey Heithoff_____

_____Luca Cano_____

_____ Jason Di Giovanni _____

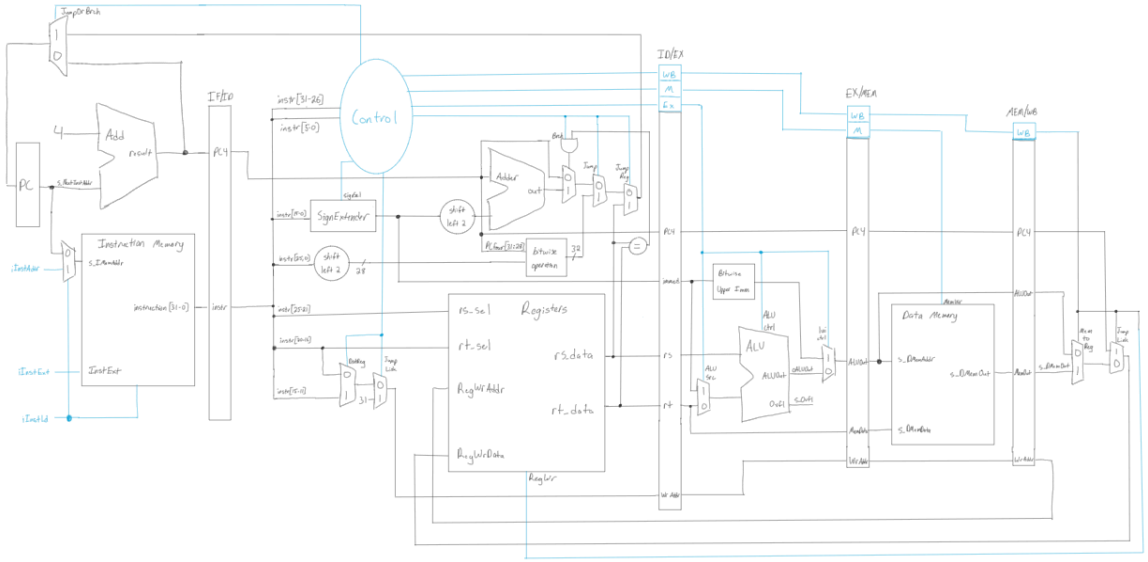
Project Teams Group #:_Section C, Group 4_____

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

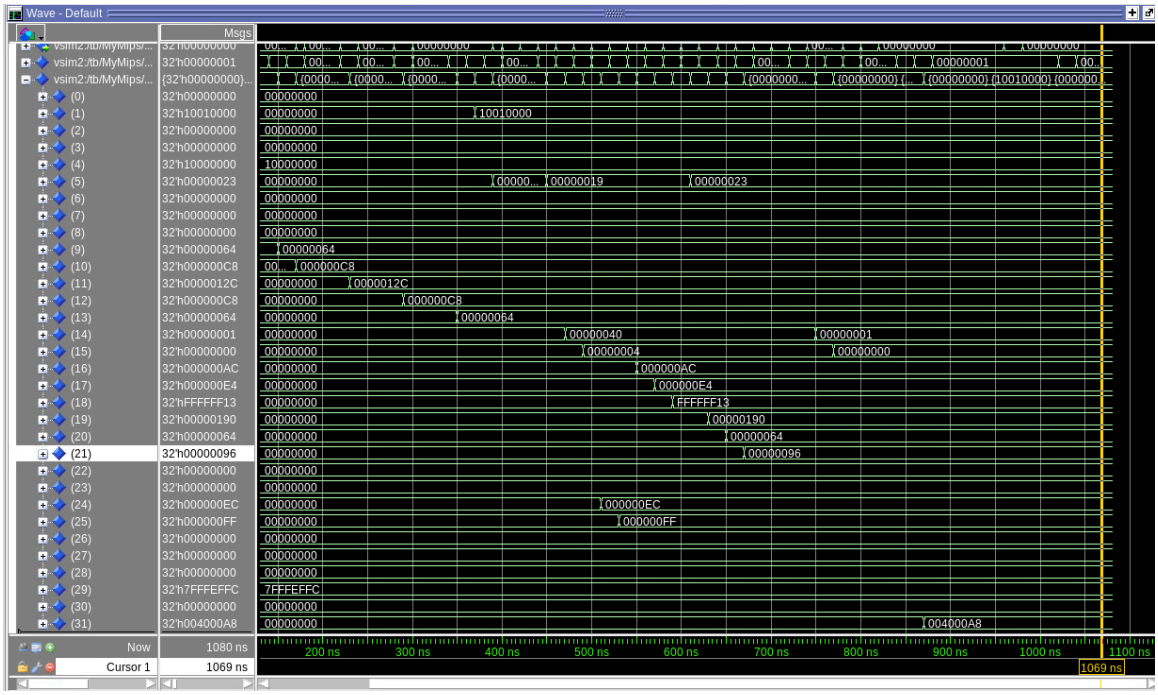
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

IFID Register		**IDEX Register**		**EXMEM Register**		**MEMWB Register**	
Consume	Produce	Consume	Produce	Consume	Produce	Consume	Produce
i_CLK	o_PC	i_CLK	o_Halt	i_CLK	o_Halt	i_CLK	o_Halt
i_RST	o_Instr	i_RST	o_MemToReg	i_RST	o_MemToReg	i_RST	o_MemToReg
i_WE		i_WE	o_RegWr	i_WE	o_RegWr	i_WE	o_RegWr
i_PC		i_Halt	o_MemWr	i_Halt	o_MemWr	i_Halt	o_isJump
i_Instr		i_MemToReg	o_isJump	i_MemToReg	o_isJump	i_MemToReg	o_RegWrAddr
		i_RegWr	o_luiCntrl	i_RegWr	o_RegWrAddr	i_RegWr	o_MemData
		i_MemWr	o_AluSrc	i_MemWr	o_AluOut	i_isJump	o_AluOut
		i_isJump	o_AluCntrl	i_isJump	o_RdDataB	i_RegWrAddr	o_PC
		i_isJumpReg	o_RegWrAddr	i_RegWrAddr	o_PC	i_MemData	
		i_luiCntrl	o_Imm	i_MemData		i_AluOut	
		i_AluSrc	o_Instr	i_AluOut		i_PC	
		i_AluCntrl	o_A	i_RdDataB			
		i_RegWrAddr	o_B	i_PC			
		i_Imm	o_SignExt				
		i_Instr	o_PC				
		i_A					
		i_B					
		i_SignExt					
		i_PC					

[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.



Expected final register states:

$t0=$8: 0x0

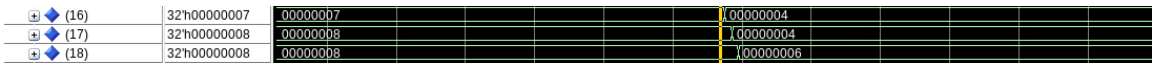
$t1=$9: 0x64
$t2=$10: 0xC8
$t3=$11: 0x12C
$t4=$12: 0xC8
$t5=$13: 0x64
$t6=$14: 0x1
$t7=$15: 0x0
$s0=$16: 0xAC
$s1=$17: 0xE4
$s2=$18: 0xFFFFFF13
$s3=$19: 0x190
$s4=$20: 0x64
$s5=$21: 0x96
$t8=$24: 0xEC
$t9=$25: 0xFF

As can be seen in the waveforms, the expected final register states match the actual final register states. For a breakdown of how the actual values are calculated, see the comments below the actual program.

[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.

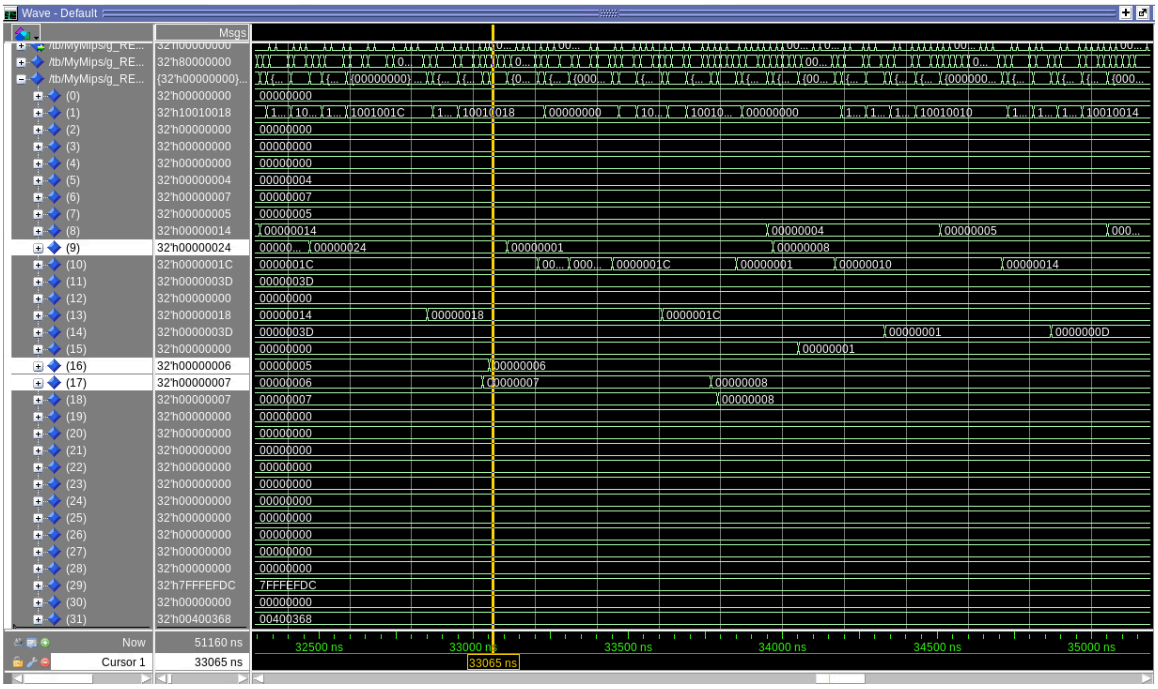
Merge Sort Screenshots

Data Hazard:



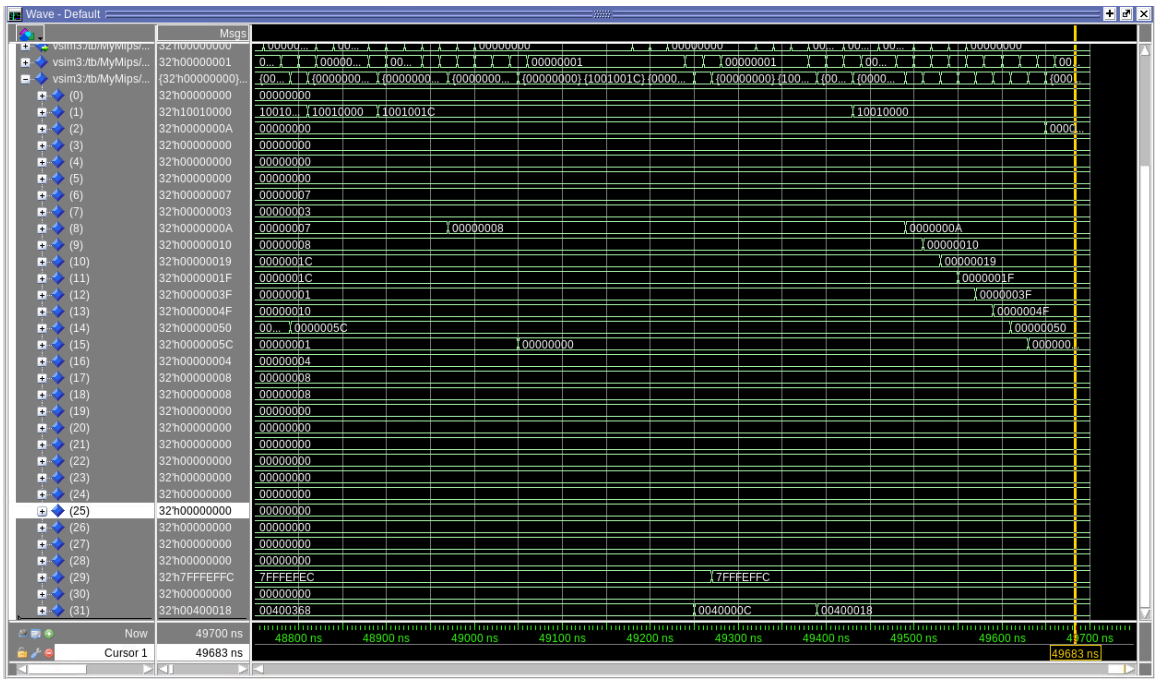
In the screenshot above, I avoided using NOPs after add instructions by setting multiple variables before using any of the data. This gives time for $s0 to be available before needing to be used next.

Control-Flow:

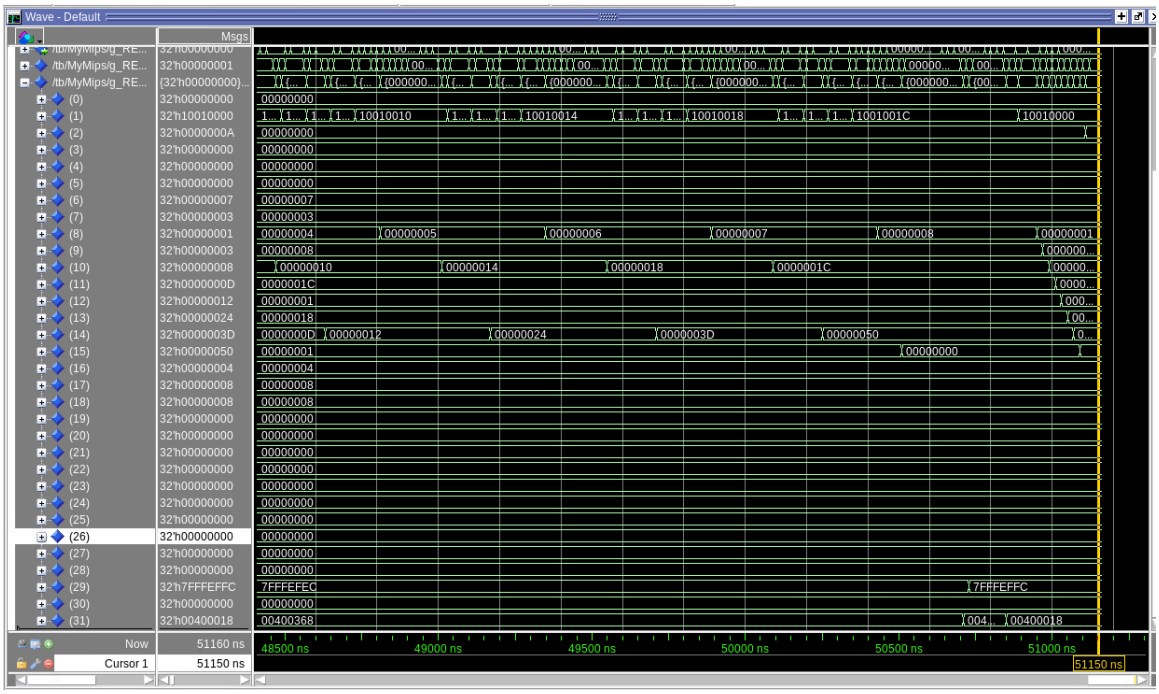


For the screenshot above, a control hazard is avoided after 17 and 16 are set there is a jump followed by a nop the 9 is set with an slt.

Final Results:



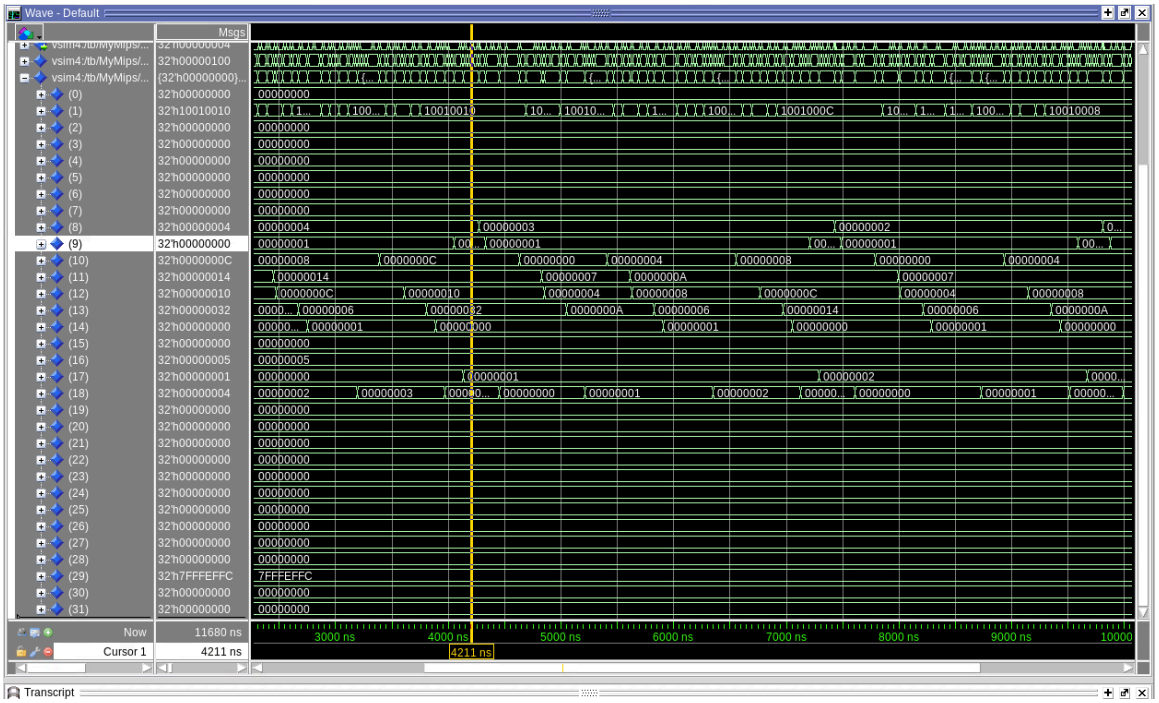
For the screenshot above, I used 25, 92, 79, 80, 31, 63, 16, 10, and the final array is successfully stored in 9-15 in ascending order.



For the screenshot above, I used 18, 3, 8, 80, 13, 36, 61, 1 and the final sorted array is stored in 9-15. A data hazard is avoided without using maximum NOPs when setting the final values.

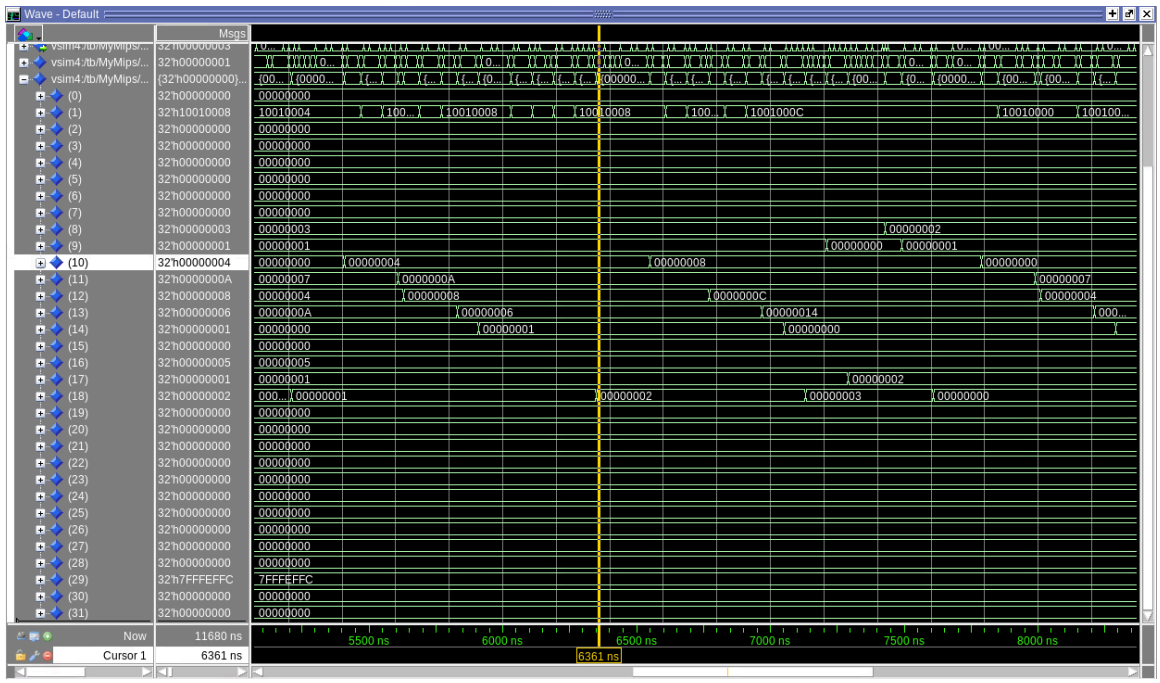
Bubble Sort Screenshots

Control Flow:

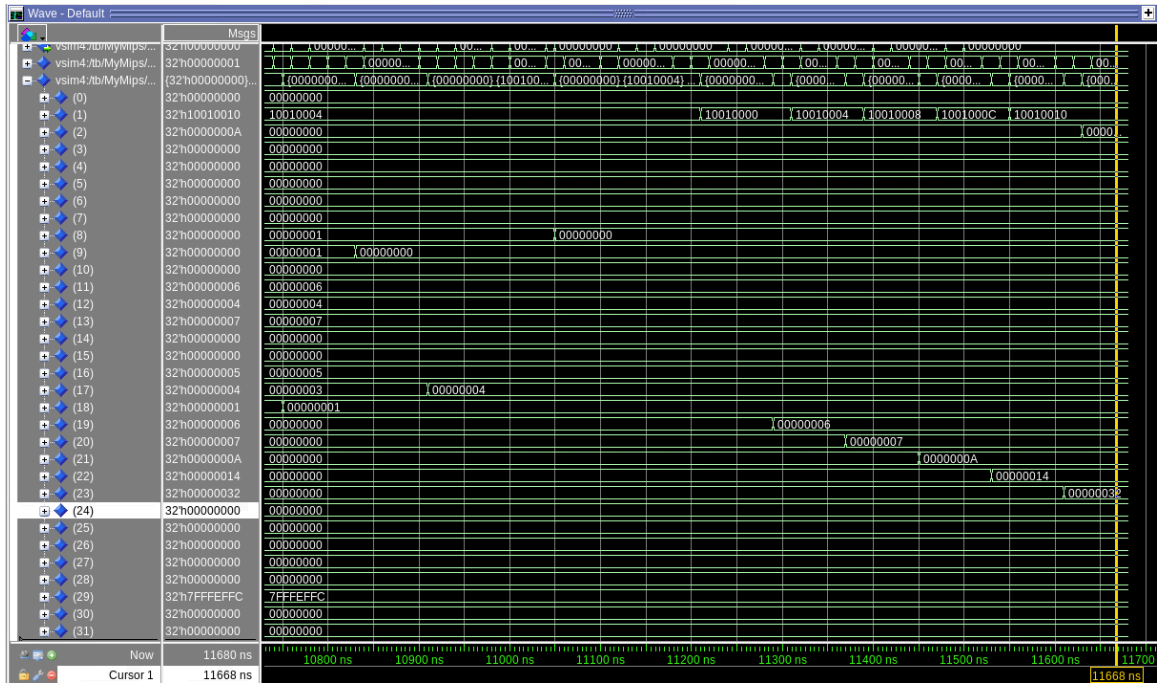


In this screenshot, 9 is set to 1 for an slt. A nop is used after a beq checking 9 and 0, and after this hazard avoidance, $t2(10) is sll'd to be multiplied.

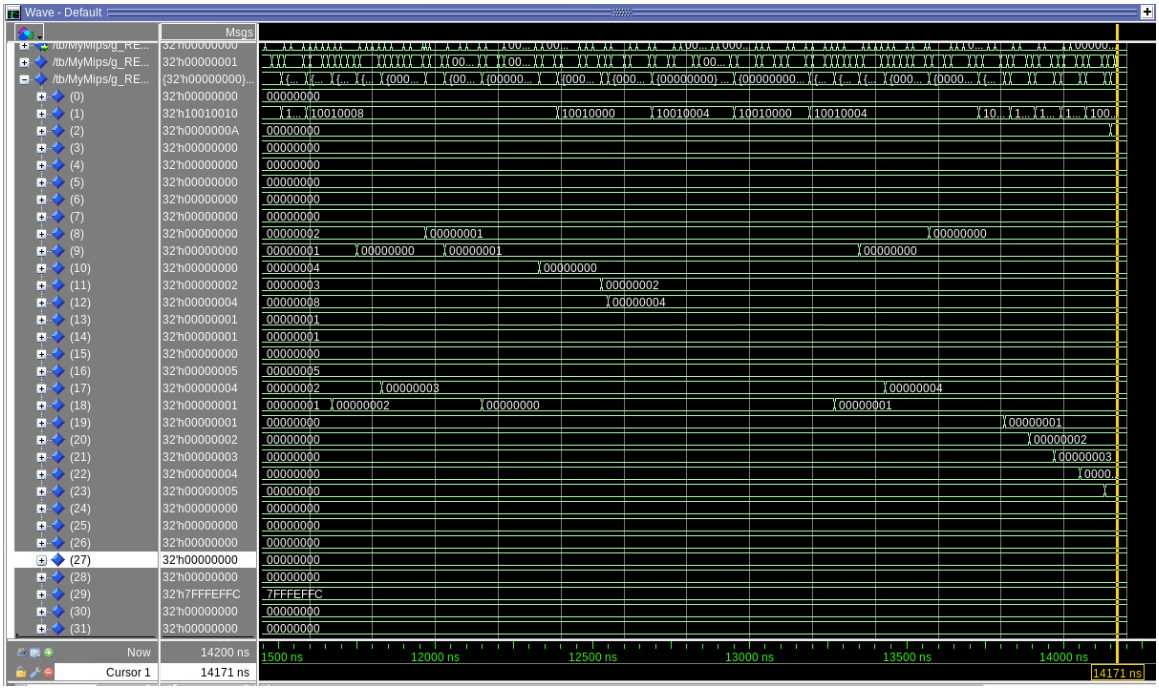
Data Flow:



In this screenshot, only 2 nops are used after a sll on $10 for an lui on $1 instead of using more nops



For the screenshot above I used 7, 10, 20, 6, 50 as the initial array, and the final array is stored in ascending order in 19-23.



For the screenshot above I used 5, 4, 3, 2, 1 as the initial array, and the final ascending array is stored in 19-23 again.

Both final screenshots also use fewer nops than maximum while avoiding a data hazard while setting 19-23 to the final array.

All programs using nops took longer than the programs used later in this project.

[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).

Software Timing:

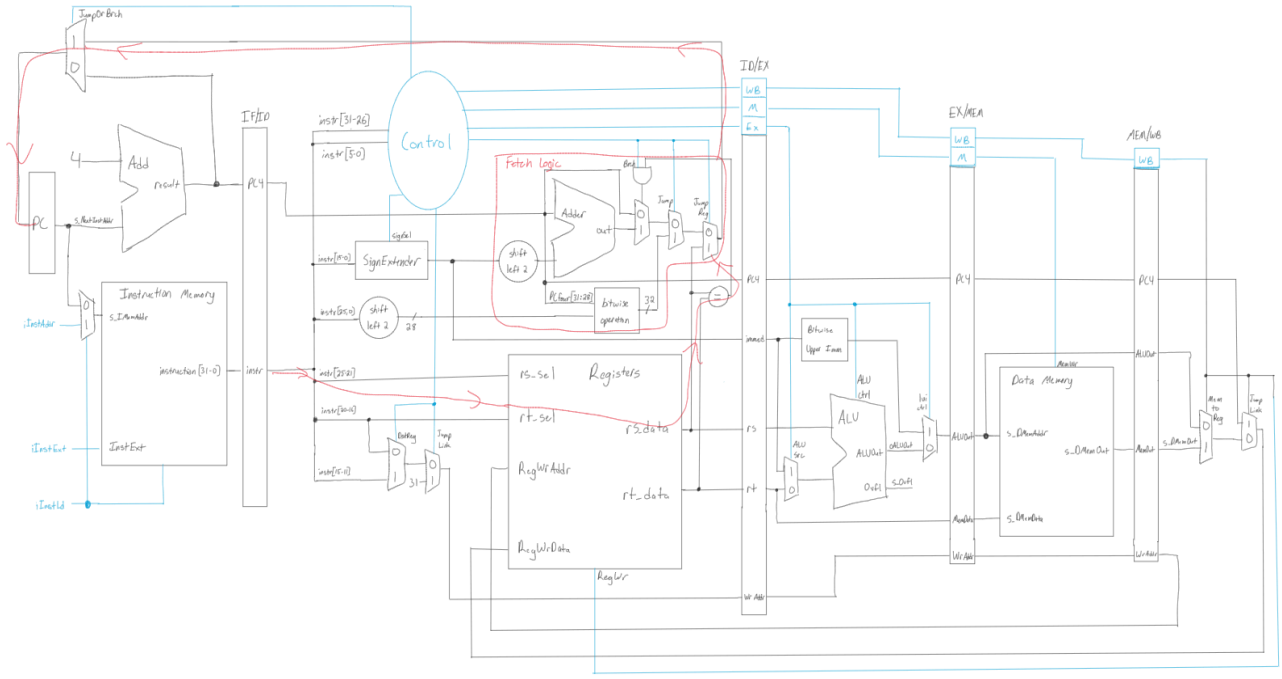
```
#
# CprE 381 toolflow Timing dump
#

FMax: 50.95mhz Clk Constraint: 20.00ns Slack: 0.37ns
```

Max frequency: 50.95 Mhz
Cycle Time: 1/50.95 Mhz = 19.63 nanoseconds

Software Critical Path:

Decode Stage:
Register File --> Equality Module (checks if two values are equal for branching) --> Fetch Logic --> PcNextAddr Mux

Critical Path Diagram (red line):

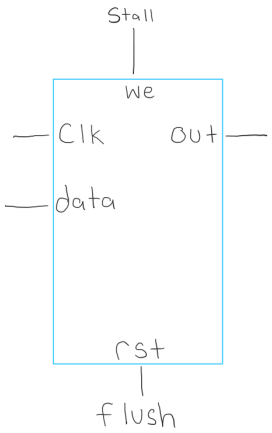


Synthesis Output:

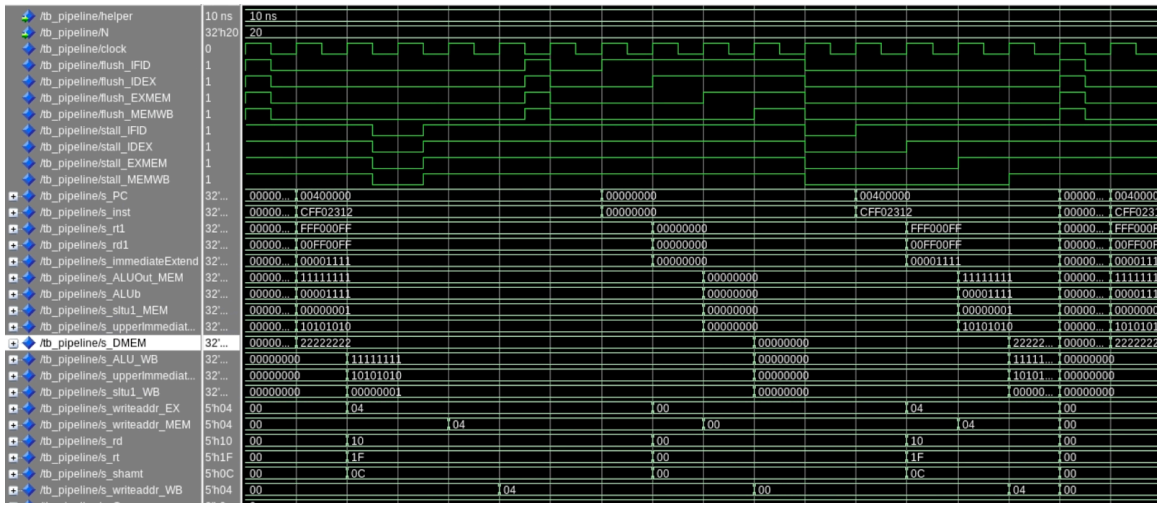
```
============================================================
From Node    : regFile:g_REGFILE|reg_N:\G_N_Reg:17:REGI|dffg:\G_NBit_Reg:0:REGI|s_Q
To Node      : reg_NPC:g_NBITREG|dffg:\G_NBit_Reg0:18:REGI|s_Q
Launch Clock : iCLK (INVERTED)
Latch Clock  : iCLK
Data Arrival Path:
Total (ns)  Incr (ns)     Type  Element
==========  ========= ==  ====  ================================
    10.000     10.000           launch edge time
    13.596      3.596  F        clock network delay
    13.828      0.232     uTco  regFile:g_REGFILE|reg_N:\G_N_Reg:17:REGI|dffg:\G_NBit_Reg:0:REGI|s_Q
    13.828      0.000 FF  CELL  g_REGFILE|\G_N_Reg:17:REGI|\G_NBit_Reg:0:REGI|s_Q|q
    14.395      0.567 FF    IC  g_REGFILE|g_MUX_RS|Mux31~11|datab
    14.799      0.404 FF  CELL  g_REGFILE|g_MUX_RS|Mux31~11|combout
    15.174      0.375 FF    IC  g_REGFILE|g_MUX_RS|Mux31~12|datad
    15.324      0.150 FR  CELL  g_REGFILE|g_MUX_RS|Mux31~12|combout
    16.275      0.951 RR    IC  g_REGFILE|g_MUX_RS|Mux31~15|datac
    16.562      0.287 RR  CELL  g_REGFILE|g_MUX_RS|Mux31~15|combout
    16.766      0.204 RR    IC  g_REGFILE|g_MUX_RS|Mux31~18|datad
    16.921      0.155 RR  CELL  g_REGFILE|g_MUX_RS|Mux31~18|combout
    17.925      1.004 RR    IC  g_REGFILE|g_MUX_RS|Mux31~19|datac
    18.210      0.285 RR  CELL  g_REGFILE|g_MUX_RS|Mux31~19|combout
    18.836      0.626 RR    IC  e_equalityModule|Equal0~0|datab
    19.268      0.432 RF  CELL  e_equalityModule|Equal0~0|combout
    19.494      0.226 FF    IC  e_equalityModule|Equal0~4|datad
    19.619      0.125 FF  CELL  e_equalityModule|Equal0~4|combout
    19.891      0.272 FF    IC  e_equalityModule|Equal0~20|datab
    20.241      0.350 FF  CELL  e_equalityModule|Equal0~20|combout
    20.482      0.241 FF    IC  g_FETCHLOGIC|g_ADD|o_F|datad
    20.632      0.150 FR  CELL  g_FETCHLOGIC|g_ADD|o_F|combout
    20.860      0.228 RR    IC  g_NBITREG|\G_NBit_Reg0:21:REGI|s_Q~0|datad
    21.015      0.155 RR  CELL  g_NBITREG|\G_NBit_Reg0:21:REGI|s_Q~0|combout
    21.443      0.428 RR    IC  g_NBITMUX_PCnextAddr|\G_NBit_MUX:18:MUXI|g_Or|o_F~5|datac
    21.730      0.287 RR  CELL  g_NBITMUX_PCnextAddr|\G_NBit_MUX:18:MUXI|g_Or|o_F~5|combout
    22.442      0.712 RR    IC  g_NBITMUX_PCnextAddr|\G_NBit_MUX:18:MUXI|g_Or|o_F~6|datad
    22.597      0.155 RR  CELL  g_NBITMUX_PCnextAddr|\G_NBit_MUX:18:MUXI|g_Or|o_F~6|combout
    22.801      0.204 RR    IC  g_NBITMUX_PCnextAddr|\G_NBit_MUX:18:MUXI|g_Or|o_F~7|datad
    22.956      0.155 RR  CELL  g_NBITMUX_PCnextAddr|\G_NBit_MUX:18:MUXI|g_Or|o_F~7|combout
    22.956      0.000 RR    IC  g_NBITREG|\G_NBit_Reg0:18:REGI|s_Q|d
    23.043      0.087 RR  CELL  reg_NPC:g_NBITREG|dffg:\G_NBit_Reg0:18:REGI|s_Q
Data Required Path:
Total (ns)  Incr (ns)     Type  Element
==========  ========= ==  ====  ================================
    20.000     20.000           latch edge time
    23.385      3.385  R        clock network delay
    23.417      0.032           clock pessimism removed
    23.397     -0.020           clock uncertainty
    23.415      0.018     uTsu  reg_NPC:g_NBITREG|dffg:\G_NBit_Reg0:18:REGI|s_Q
Data Arrival Time  :    23.043
Data Required Time :    23.415
Slack              :     0.372
============================================================
```

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

Stalling operations use the existing write enable function of the N-bit register, and flushing operations use the existing reset function of the 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.



This test bench demonstrates that values stored in the initial IF/ID stage are correctly available four cycles later, as expected. Additionally, all four registers 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.

Instruction	Produces
General R format (add, or, and, sub, etc)	regWrData
General I format (addi, ori, andi, subi, etc)	regWrData
beq, bne	PcNext
jr	PcNext
j	PcNext
jal	PcNext, regWrData
lw	regWrData
sw	memWrData

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

Instruction	Consumes
General R format (add, or, and, sub, etc)	rs, rt, regDst, regWrAddr
General I format (addi, ori, andi, subi, etc)	rs, imm, regDst, regWrAddr
beq, bne	rs, rt, imm, brnch
jr	rs, imm, jump
j	imm, jump
jal	imm, jump, regDst, regWrAddr, currPc
lw	rs, regDst, regWrAddr, memToReg
sw	rs, rt, memToReg, memWrite, memWrAddr

[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.

Case Number	Case Description	Instructions	Requires Stall	Can be Forwarded	Forwad From Stage	Forward To Stage
Case 1a	R format -> R format (Rs hazard)	add t1 x x add x t1 x	No	Yes	Memory	Execute
Case 1b	R format -> R format (Rt hazard)	add t1 x x add x x t1	No	Yes	Memory	Execute
Case 2a	R format -> gap -> R format (Rs hazard)	add t1 x x NOP add x t1 x	No	Yes	Writeback	Execute
Case 2b	R format -> gap -> R format (Rt hazard)	add t1 x x NOP add x t1 x	No	Yes	Writeback	Execute
Case 3a	I format -> R format (Rs hazard)	addi t1 x x add x t1 x	No	Yes	Memory	Execute
Case 3b	I format -> R format (Rt hazard)	addi t1 x x add x x t1	No	Yes	Memory	Execute
Case 4a	I format -> gap -> R format (Rs hazard)	addi t1 x x NOP add x t1 x	No	Yes	Writeback	Execute
Case 4b	I format -> gap -> R format (Rt hazard)	addi t1 x x NOP add x t1 x	No	Yes	Writeback	Execute
Case 5	R format -> I format (Rs hazard)	add t1 x x addi x t1 x	No	Yes	Memory	Execute
Case 6	R format -> gap -> I format (Rs hazard)	add t1 x x NOP addi x t1 x	No	Yes	Writeback	Execute
Case 7	Lw -> R format (Rs hazard)	lw t1 x(x) add x t1 x	Yes	Yes (After Stall)	Writeback	Execute
Case 8	Lw -> R format (Rt hazard)	lw t1 x(x) add x x t1	Yes	Yes (After Stall)	Writeback	Execute
Case 9	Lw -> I format (Rs hazard)	lw t1 x(x) addi x t1 x	Yes	Yes (After Stall)	Writeback	Execute
Case 10	R format -> jr or bne or beq	add t1 x x jr t1 label	No	Yes	Writeback	Decode
Case 11	I format -> jr or bne or beq	addi t1 x x beq t1 x label	No	Yes	Writeback	Decode
Case 12	Lw -> jr or bne or beq	lw t1 x x bne t1 x label	Yes (two)	Yes (After a stall)	Writeback	Decode

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

IFID Register	IDEX Register		EXMEM Register	MEMWB Register
i_CLK	i_CLK	i_SignExt	i_CLK	i_CLK
i_Flush	i_Flush	i_PC	i_Flush	i_Flush
i_Stall	i_Stall	i_RegDst	i_Stall	i_Stall
i_PC	i_Halt	i_forwardA	i_Halt	i_Halt
i_PCNext	i_MemToReg	i_forwardB	i_MemToReg	i_MemToReg
i_Instr	i_RegWr	i_RsAddr	i_RegWr	i_RegWr
	i_MemWr	i_RtAddr	i_MemWr	i_isJump
	i_isJump	i_RegWrAddr	i_isJump	i_RegWrAddr
	i_isJumpReg	i_Imm	i_RegWrAddr	i_MemData
	i_luiCntrl	i_Instr	i_MemData	i_AluOut
	i_AluSrc		i_AluOut	i_PC
	i_AluCntrl		i_AluZero	i_Overflow
	i_A		i_PC	
	i_B		i_RdDataB	
	i_overflowCtrl		i_Overflow	

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

Predict Taken		
beq	Updates PC to Label in Fetch Stage	If incorrect Jump, flush and update PC in Decode Stage
bne	Updates PC to Label in Fetch Stage	If incorrect Jump, flush and update PC in Decode Stage
j	Updates PC to label in Decode Stage	
jr	Updates PC to label in Decode Stage	
jal	Updates PC to label in Decode Stage	

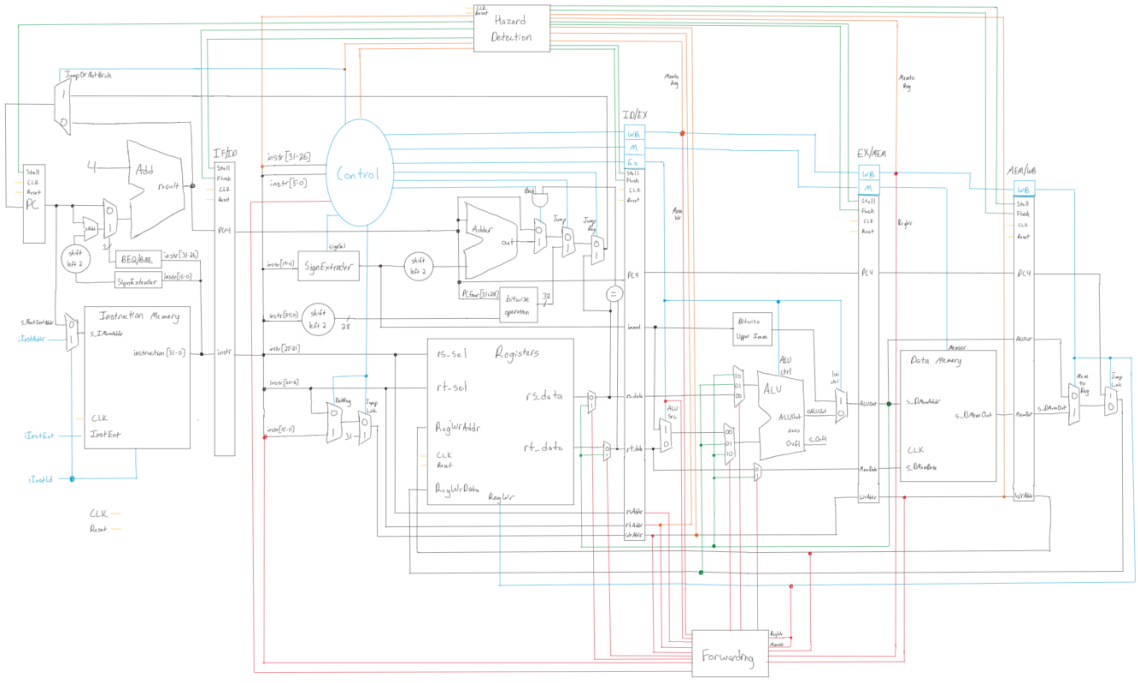
[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.

Predict Taken		Flushes/Stalls
	Updates PC to Label in Fetch Stage	None
beq	If incorrect Jump, flush and update PC in Decode Stage	Flush IFID, Stall PC
	Updates PC to Label in Fetch Stage	None
bne	If incorrect Jump, flush and update PC in Decode Stage	Flush IFID, Stall PC
j	Updates PC to label in Decode Stage	Flush IFID, Stall PC
jr	Updates PC to label in Decode Stage	Flush IFID, Stall PC
jal	Updates PC to label in Decode Stage	Flush IFID, Stall PC

[2.c.iii] As a start, identify and justify the maximum possible benefit per control-flow instruction you will see over your baseline design.

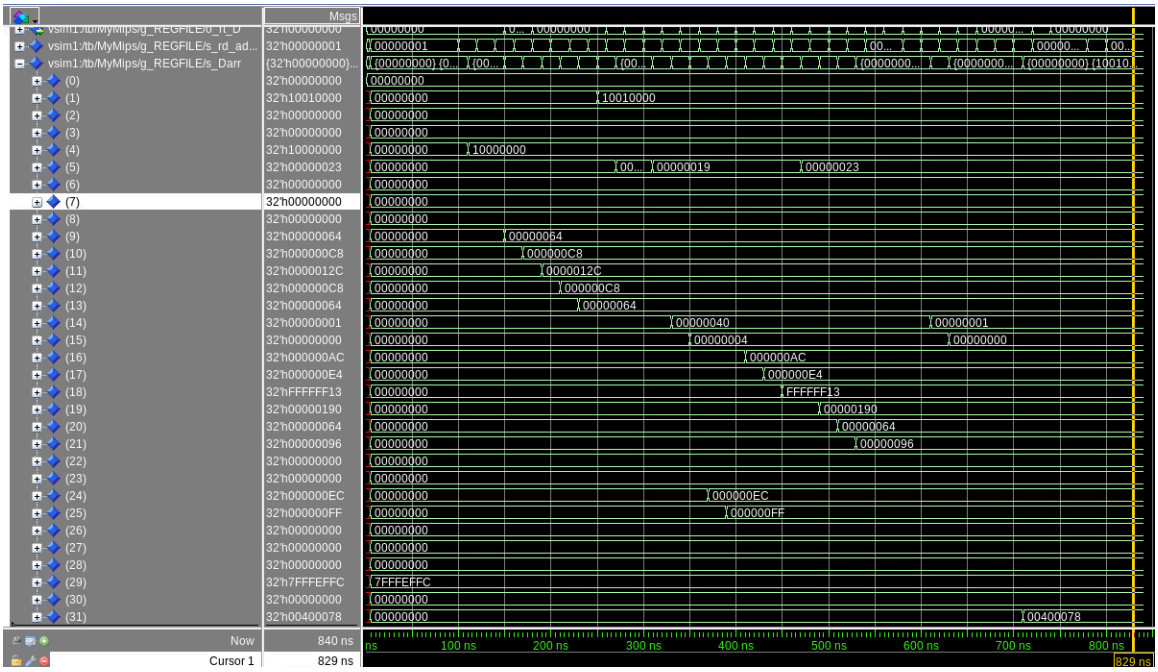
Predict Taken Benefit			
Instruction	Baseline (Predict Not Taken)	Predict Taken	Maximum Benefit
beq (taken)	Flush IFID and Stall PC	None	one less stall
bne (taken)	Flush IFID and Stall PC	None	one less stall
j	Flush IFID and Stall PC	Flush IFID and Stall PC	no change
jr	Flush IFID and Stall PC	Flush IFID and Stall PC	no change
jal	Flush IFID and Stall PC	Flush IFID and Stall PC	no change

[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.

Base Test:



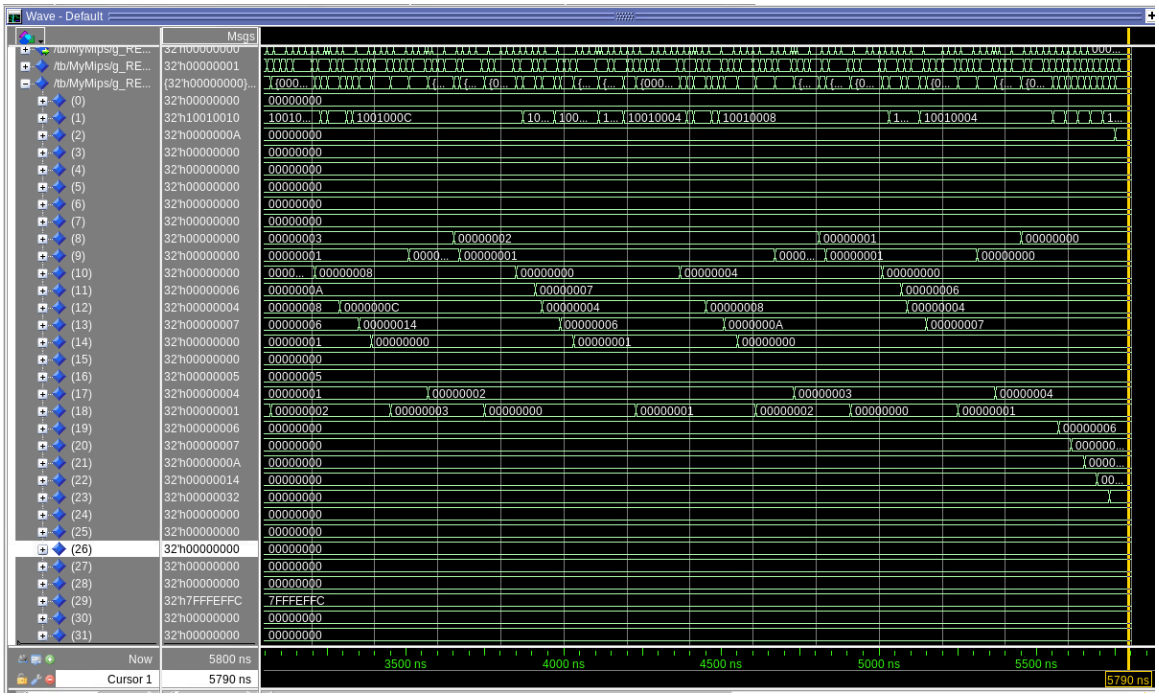
Expected final register states:

$t0=$8: 0x0

$t1=$9: 0x64
$t2=$10: 0xC8
$t3=$11: 0x12C
$t4=$12: 0xC8
$t5=$13: 0x64
$t6=$14: 0x1
$t7=$15: 0x0
$s0=$16: 0xAC
$s1=$17: 0xE4
$s2=$18: 0xFFFFFF13
$s3=$19: 0x190
$s4=$20: 0x64
$s5=$21: 0x96
$t8=$24: 0xEC
$t9=$25: 0xFF

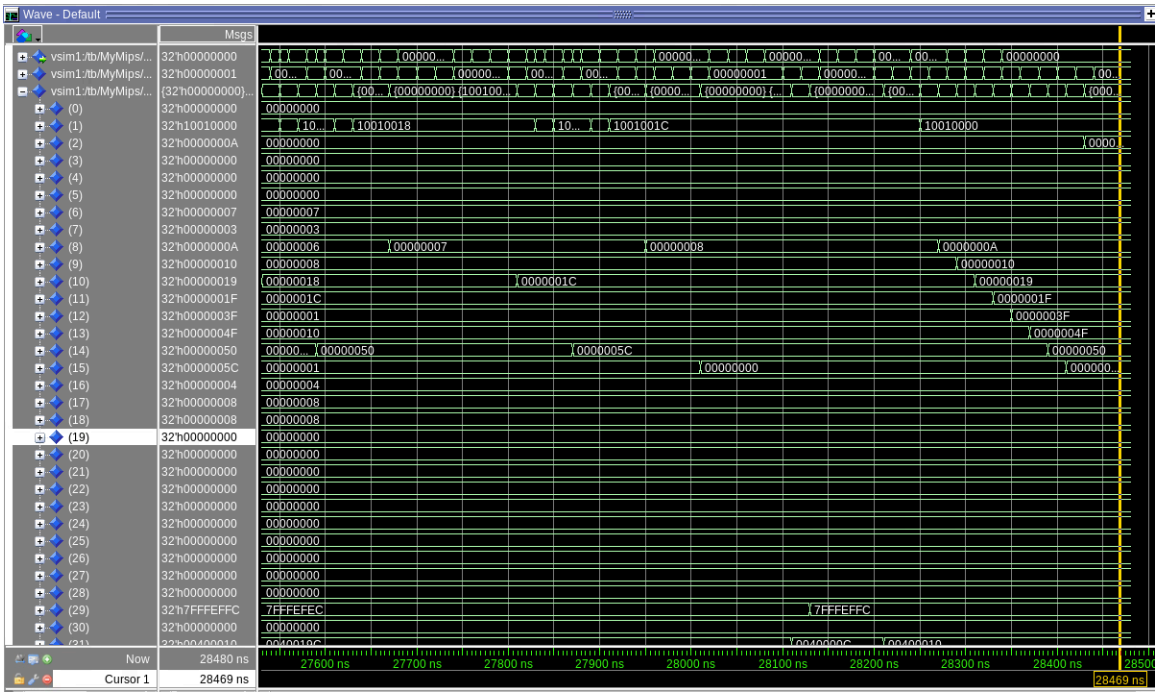
As can be seen in the waveforms, the expected final register states match the actual final register states. For a breakdown of how the actual values are calculated, see the comments below the actual program.

Bubble Sort:



As can be seen, I set $s3-$s7 (19-23) to be the final array in sorted order. For this test, the array started as 7, 10, 20, 6, 50, and when the program was finished, it was sorted to 6, 7, 10, 20, 50

Bubble Sort without NOP Screenshot



For this test, the array began as 25, 92, 79, 80, 31, 63, 16, 10 and was sorted successfully as shown in the screenshot above (Stored in 8-15).

[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.

Case	Testing Approach Description	Justification
RAW_1	Test RAW hazard with adjacent instructions where the result of one is used by the next	Ensures forwarding logic correctly identifies and resolves data dependencies in back-to-back instructions.
RAW_2	Test RAW hazard with a gap of one instruction between dependent instructions	Validates forwarding across multiple pipeline stages (e.g., EX/MEM and MEM/WB).
WAW_1	Test WAW hazard where two instructions write to the same register consecutively	Ensures write serialization to prevent overwriting in-flight data.
WAW_2	Test WAW hazard with a gap of one instruction between writes	Verifies pipeline logic prevents data corruption when writes occur in non-adjacent instructions.
WAR_1	Test WAR hazard where a register read is followed by a write to the same register	Checks that writes are delayed until earlier reads are completed.
WAR_2	Test WAR hazard with an instruction gap between the read and write	Ensures correct handling of deferred writes across pipeline stages.
COMBO_1	Test a combination of RAW and WAR hazards in consecutive instructions	Validates simultaneous resolution of multiple hazards without conflicts.

[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.

BRANCH_1	Test a taken branch with instructions after the branch flushed	Verifies pipeline correctly handles branch predictions and flushes invalid instructions.
BRANCH_2	Test a not-taken branch with instructions executed sequentially	Ensures that the pipeline continues smoothly without unnecessary stalls.
JUMP_1	Test an unconditional jump to a label	Checks proper PC update and pipeline flushing.
JUMP_2	Test a jump and link instruction with a return to the caller	Validates correct updates to the link register and return operation.
CTRL_COMBO_1	Test back-to-back branches, both taken	Verifies the pipeline can handle consecutive branch mispredictions and flushes.

[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).

Hardware Predict Branch Not Taken:

```
#
# CprE 381 toolflow Timing dump
#

FMax: 41.82mhz Clk Constraint: 20.00ns Slack: -3.91ns
```

Max frequency: 41.82 Mhz
Cycle Time: 1/41.82 Mhz = 23.91 nanoseconds

Hardware Predict Branch Taken Timing:

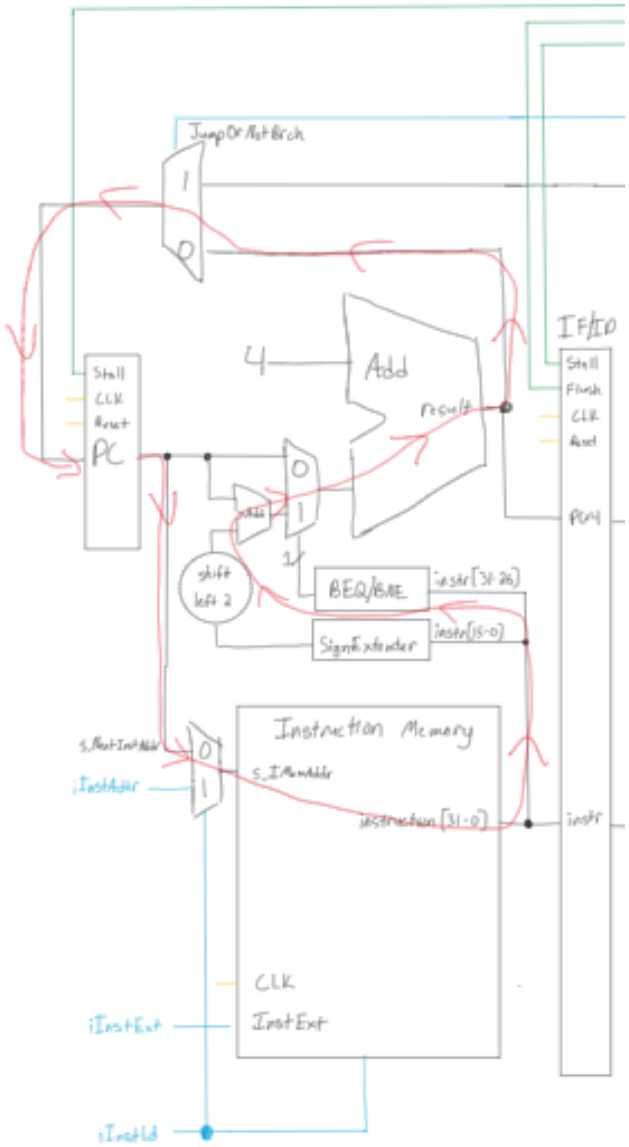
```
#
# CprE 381 toolflow Timing dump
#

FMax: 44.48mhz Clk Constraint: 20.00ns Slack: -2.48ns
```

Max frequency: 44.48 Mhz
Cycle Time: 1/44.48 Mhz = 22.48 nanoseconds

Hardware Predict Branch Taken Critical Path:

Fetch Stage:
PC --> IMEM --> Branch Target Adder --> Branch Taken Mux --> Branch Target +4 Adder --> PC + 4 Adder -> PC next address mux --> PC

Critical Path Diagram (red line): Fetch Stage



Synthesis Output:

Total (ns)	Incr (ns)		Type	Element
0.000	0.000			launch edge time
3.087	3.087	R		clock network delay
3.319	0.232		uTco	reg_NPC:g_NBITREG_PC|dffg:\G_NBit_Reg0:2:REGI|s_Q
3.319	0.000	FF	CELL	g_NBITREG_PC|\G_NBit_Reg0:2:REGI|s_Q|q
3.722	0.403	FF	IC	s_IMemAddr[2]~6|datad
3.847	0.125	FF	CELL	s_IMemAddr[2]~6|combout
6.416	2.569	FF	IC	IMem|ram~46349|dataa
6.828	0.412	FR	CELL	IMem|ram~46349|combout
7.826	0.998	RR	IC	IMem|ram~46350|datad
7.981	0.155	RR	CELL	IMem|ram~46350|combout
9.575	1.594	RR	IC	IMem|ram~46351|datab
9.963	0.388	RR	CELL	IMem|ram~46351|combout
10.168	0.205	RR	IC	IMem|ram~46352|datad
10.323	0.155	RR	CELL	IMem|ram~46352|combout
10.527	0.204	RR	IC	IMem|ram~46363|datad
10.666	0.139	RF	CELL	IMem|ram~46363|combout
10.893	0.227	FF	IC	IMem|ram~46406|datad
11.018	0.125	FF	CELL	IMem|ram~46406|combout
13.062	2.044	FF	IC	IMem|ram~46449|datac
13.343	0.281	FF	CELL	IMem|ram~46449|combout
13.580	0.237	FF	IC	IMem|ram~46450|datac
13.861	0.281	FF	CELL	IMem|ram~46450|combout
15.190	1.329	FF	IC	g_NBITADDER_PC|\G_NBit_Adder:5:ADDERI|g_OR|o_F~1|datad
15.315	0.125	FF	CELL	g_NBITADDER_PC|\G_NBit_Adder:5:ADDERI|g_OR|o_F~1|combout
15.584	0.269	FF	IC	g_NBITADDER_PC|\G_NBit_Adder:5:ADDERI|g_OR|o_F~2|datab
15.977	0.393	FF	CELL	g_NBITADDER_PC|\G_NBit_Adder:5:ADDERI|g_OR|o_F~2|combout
17.328	1.351	FF	IC	g_NBITADDER_PC|\G_NBit_Adder:6:ADDERI|g_XOR2|o_F|datac
17.609	0.281	FF	CELL	g_NBITADDER_PC|\G_NBit_Adder:6:ADDERI|g_XOR2|o_F|combout
17.878	0.269	FF	IC	g_NBITMUX_BrnchCheckMUX|\G_NBit_MUX:6:MUXI|g_Or|o_F~2|datab
18.271	0.393	FF	CELL	g_NBITMUX_BrnchCheckMUX|\G_NBit_MUX:6:MUXI|g_Or|o_F~2|combout
19.876	1.605	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:7:ADDERI|g_ADD2|o_F|datac
20.157	0.281	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:7:ADDERI|g_ADD2|o_F|combout
20.409	0.252	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:10:ADDERI|g_ADD2|o_F|datad
20.534	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:10:ADDERI|g_ADD2|o_F|combout
20.787	0.253	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:13:ADDERI|g_ADD2|o_F|datad
20.912	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:13:ADDERI|g_ADD2|o_F|combout
21.216	0.304	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:16:ADDERI|g_ADD2|o_F|datad
21.341	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:16:ADDERI|g_ADD2|o_F|combout
21.590	0.249	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:19:ADDERI|g_ADD2|o_F|datad
21.715	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:19:ADDERI|g_ADD2|o_F|combout
21.991	0.276	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:21:ADDERI|g_ADD2|o_F|datad
22.116	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:21:ADDERI|g_ADD2|o_F|combout
22.372	0.256	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:23:ADDERI|g_ADD2|o_F|datad
22.497	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:23:ADDERI|g_ADD2|o_F|combout
22.792	0.295	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:25:ADDERI|g_ADD2|o_F|datac
23.073	0.281	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:25:ADDERI|g_ADD2|o_F|combout
23.771	0.698	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:27:ADDERI|g_ADD2|o_F|datad
23.896	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:27:ADDERI|g_ADD2|o_F|combout
24.148	0.252	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:31:ADDERI|g_XOR2|o_F~1|datad
24.273	0.125	FF	CELL	g_NBITADDER_PC2|\G_NBit_Adder:31:ADDERI|g_XOR2|o_F~1|combout
24.500	0.227	FF	IC	g_NBITADDER_PC2|\G_NBit_Adder:31:ADDERI|g_XOR2|o_F~2|datad
24.650	0.150	FR	CELL	g_NBITADDER_PC2|\G_NBit_Adder:31:ADDERI|g_XOR2|o_F~2|combout
24.855	0.205	RR	IC	g_NBITMUX_PCnextAddr|\G_NBit_MUX:31:MUXI|g_Or|o_F~0|datad
25.010	0.155	RR	CELL	g_NBITMUX_PCnextAddr|\G_NBit_MUX:31:MUXI|g_Or|o_F~0|combout
25.217	0.207	RR	IC	g_NBITMUX_PCnextAddr|\G_NBit_MUX:31:MUXI|g_Or|o_F~2|datad
25.356	0.139	RF	CELL	g_NBITMUX_PCnextAddr|\G_NBit_MUX:31:MUXI|g_Or|o_F~2|combout
25.356	0.000	FF	IC	g_NBITREG_PC|\G_NBit_Reg2:31:REGI|s_Q|d
25.460	0.104	FF	CELL	reg_NPC:g_NBITREG_PC|dffg:\G_NBit_Reg2:31:REGI|s_Q

Data Required Path:

Total (ns)	Incr (ns)		Type	Element
20.000	20.000			latch edge time
22.971	2.971	R		clock network delay
22.979	0.008			clock pessimism removed
22.959	-0.020			clock uncertainty
22.977	0.018		uTsu	reg_NPC:g_NBITREG_PC|dffg:\G_NBit_Reg2:31:REGI|s_Q

Data Arrival Time : 25.460
Data Required Time : 22.977
Slack : -2.483 (VIOLATED)