CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: _____John Brose_____

_____Andrew Deick_____

_____Rolf Anderson_____

Project Teams Group #:_____2-4_____

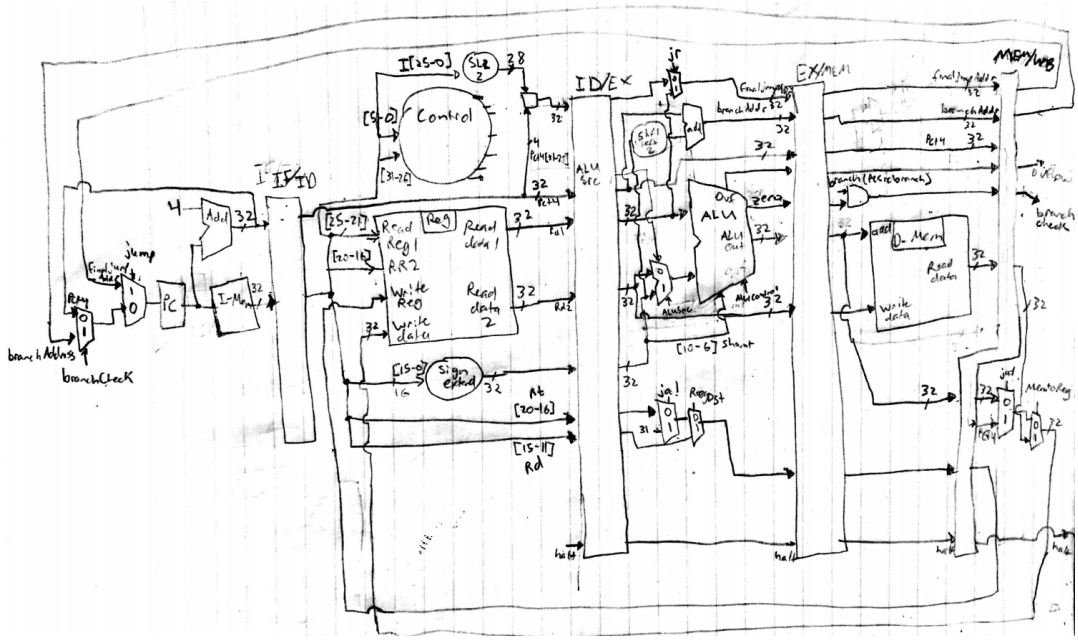
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

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

	A	B	C	D	E	F	G
1	**Pipeline Stage**						
2	IF/ID		32 bits: PC+4 (for branch)	32 bits: Instruction			
3	ID/EX		32 bits: PC+4 (for branch)	32 bits: Read data 1(A)	32 bits: Read data 2	32 bits: sign extended immediate	32 bits: jumpAddress
4	EX/MEM		32 bits: PC+4 (for branch)	32 bits: final_jump_address	32 bits: branchAddr	1 bit: overflow	1 bit: zero(for branching)
5	MEM/WB		32 bits: PC+4 (for branch)	32 bits: final_jump_address	32 bits: branchAddr	1 bit: overflow	1 bit: branch_check(for branching)

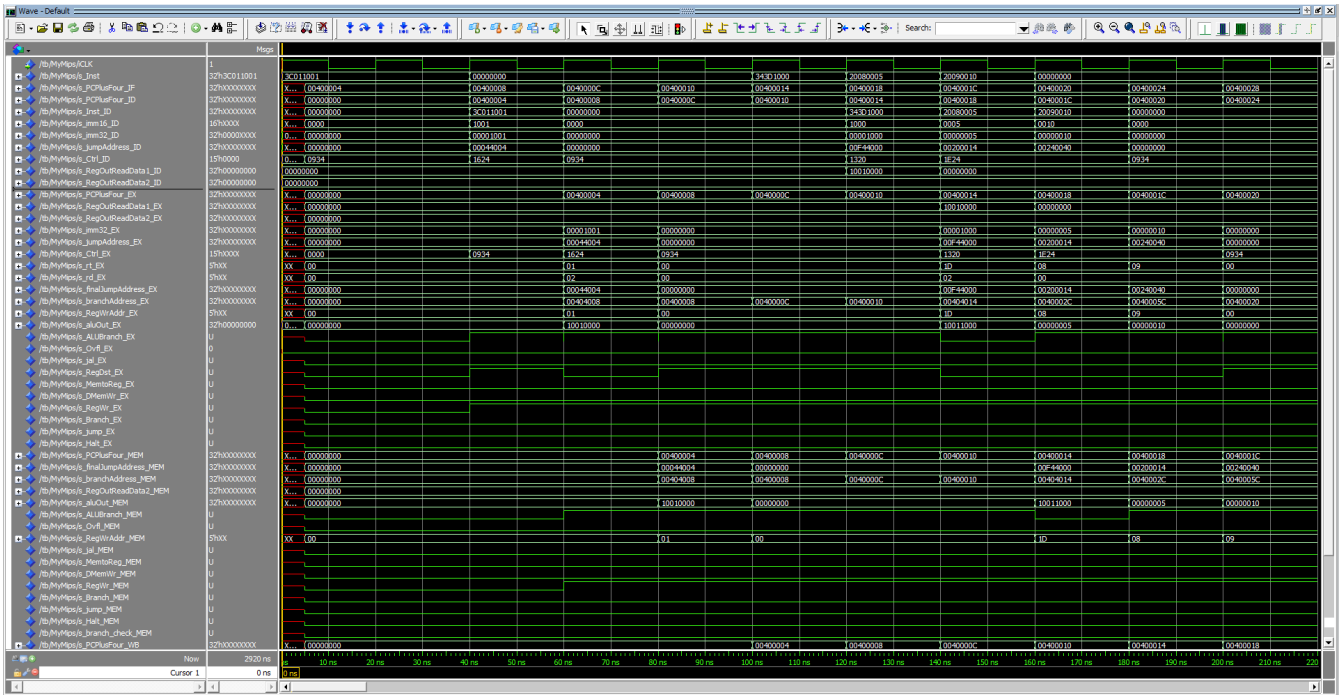
	A	H	I	J	K	L	M	N	O	P	Q
1	**Pipeline Stage**										
2	IF/ID										
3	ID/EX	5 bits: instruction 20-16 (rt)	5 bits: instruction 15-11 (rd)	15 bits: control_bits(SignExt kept cause easier to implement)							
4	EX/MEM	32 bits: alu_out	32 bits: read_data_2	5 bits: write_register	jal	MemtoReg	we_mem	we_reg	branch	j	halt
5	MEM/WB	32 bits: alu_out	32 bits: mem_read_data	5 bits: write_register	jal	MemtoReg		we_reg		j	halt

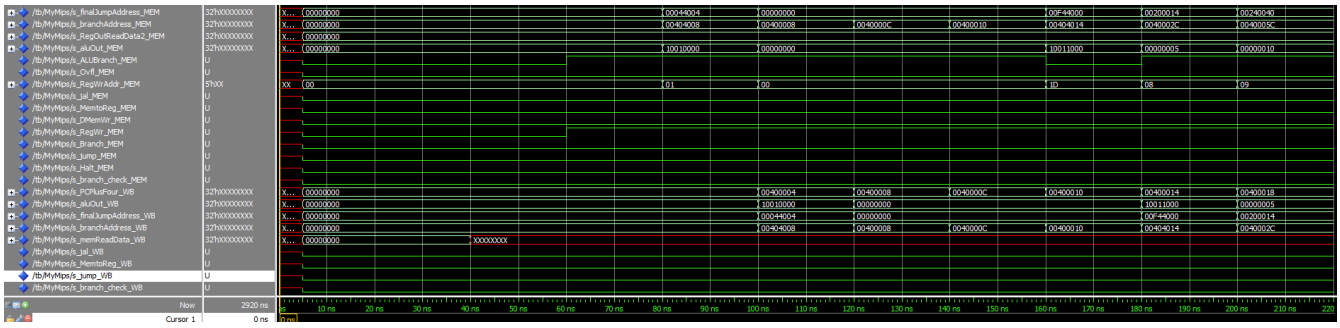
[1.b.ii] *high-level schematic drawing of the interconnection between components.*



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

```
aStarting Simulation for : mips\proj1_base_test_ss.s
aSkipping compilation
 Starting VHDL Simulation...
Successfully simulated program!
Victory!! Your processes matches MARS expected output with no mismatches!!
```





The base test is a rework of the base tests from project 1. It uses all of the operations supported by our pipeline, and we removed all data hazards from the assembly program. The waveforms above show the beginning of the program, where $sp is populated and a couple addi operations are performed and saved to $t0 and $t1. I separated the waveforms by pipeline stage and cycle. The operations flow down and to the left like it should be. Each PC+4 value gets passed down the line while the next PC+4, PC+8, is populating the pipeline stage before it. There are no errors in the testing, and the waveform analysis shows all values where they should be. In the section where active pipelining is performed (i.e. multiple operations going down the line without flushing) each stage is isolated from the other, and no errors arise from stages interfering with one another.

[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.*

```
Starting Simulation for : mips\proj1_bubblesort_ss.s
Skipping compilation
Starting VHDL Simulation...
Successfully simulated program!
Victory!! Your processes matches MARS expected output with no mismatches!!
```

Beginning of program:

Halfway through execution:

In the above waveforms, the following operations are visible at various points in the pipeline:

```
PC[-5] |lw    $t0, 0($t8)          #not visible but helpful for context
PC[-4] |lw    $t1, 4($t8)
PC[-3] |nop
PC[-2] |nop
PC[-1] |nop
PC[0]  |slt   $t2, $t1, $t0
PC[1]  |addi  $at, $zero, 1
PC[2]  |nop
PC[3]  |nop
PC[4]  |nop
PC[5]  |bne   $t2, $at, preloop   #branch not taken
PC[6]  |nop
PC[7]  |nop
PC[8]  |nop
PC[9]  |sw    $t1, 0($t8)
PC[10] |sw    $t0, 4($t8)
```

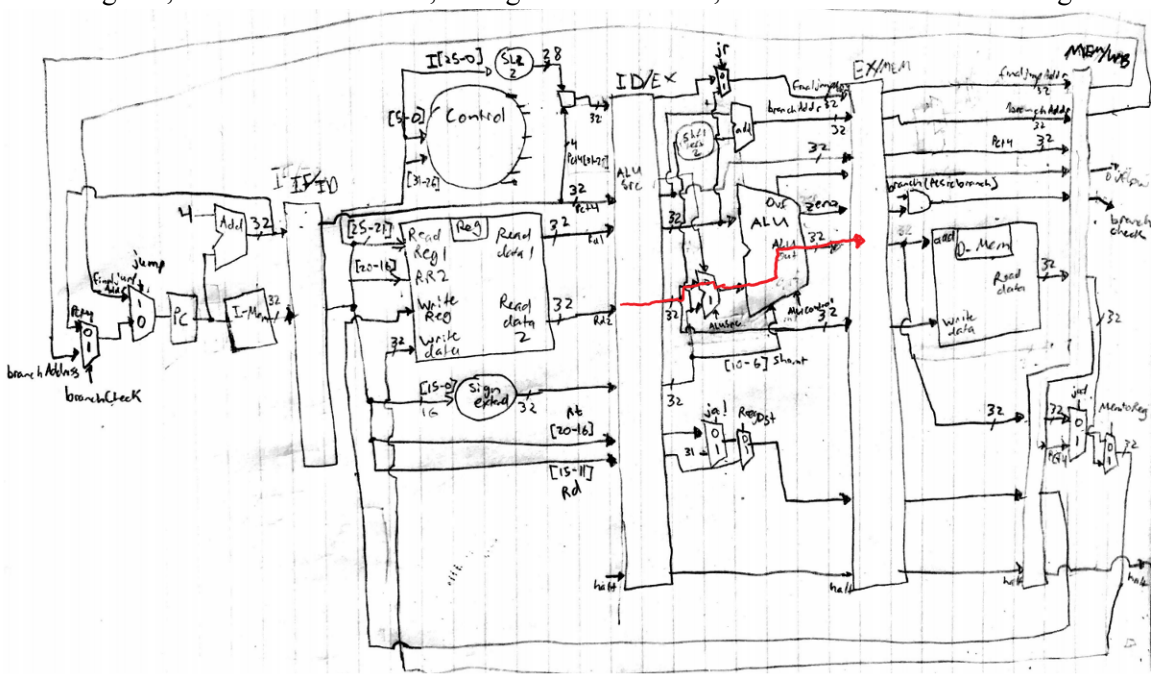
As with the base tests, the waveforms are exactly as expected. In our implementation of bubble sort, the processor loads $t0 = a[i] and $t1 = a[i+1]. If $t1 is less than $t0, then it does NOT branch and proceeds to save a[i] = $t1 and a[i+1] = $t0. In the above waveform, a[i] = 3 and a[i+1] = 2, so it does not branch and swaps the values.

Analyzing the waveforms show no inconsistencies with our expectations. Each pipeline stage remains isolated and free from interference as long as there is not a data inconsistency. Flushing the system allows branches and jumps to function correctly.

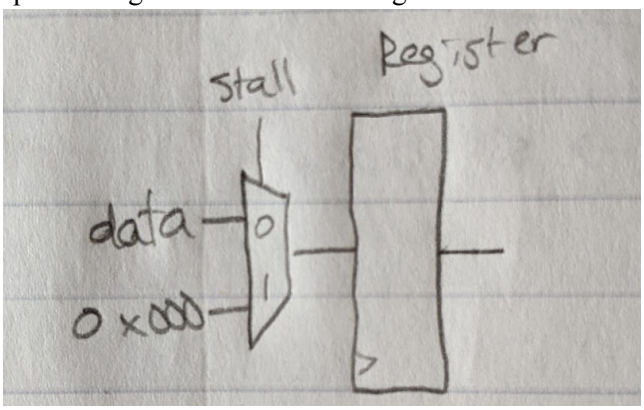
Regular instructions do not need a full flush for data dependencies, and the waveform shows that only 3 nops are needed for those situations.

[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 critical path of the software pipelined processor is highlighted in red, and numerically comes out to 17.23 nanoseconds which is a frequency of 58.05 MHz. The critical path goes from the ID/EX register, into the ALUsrc mux, through the main ALU, and ends at the EX/MEM register.



[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.*

Our hardware-scheduled processor does not have the ability to stall or flush individual registers.

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

<table>
<tr><th>Instruction</th><th colspan="2">Signals Produced</th></tr>
<tr><td>add, addu, and, nor, xor, or, slt,sll, srl, sra, sub, subu</td><td rowspan="2">s_writeAddr_ID ,
s_writeAddr_EX ,
s_writeAddr_MEM,
s_writeEnable_ID,
s_writeEnable_EX,
s_writeEnable_MEM,</td><td>s_readAddr1,
s_readAddr2,</td></tr>
<tr><td>addi, addiu, andi, xori, ori, slti, lui, jr</td><td>s_readAddr1</td></tr>
<tr><td>beq, bne</td><td colspan="2">s_branch_ID ,
s_branch_EX ,
s_branch_MEM,
s_branch_WB ,</td></tr>
<tr><td>j, jal, jr</td><td colspan="2">s_jump_ID ,
s_jump_EX ,
s_jump_MEM,
s_jump_WB ,</td></tr>
</table>

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

Instruction	Signals Consumed	
add, addu, and, nor, xor, or, slt,sll, srl, sra, sub, subu	s_writeAddr_ID , s_writeAddr_EX , s_writeAddr_MEM, s_writeEnable_ID, s_writeEnable_EX, s_writeEnable_MEM,	s_readAddr1, s_readAddr2,
addi, addiu, andi, xori, ori, slti, lui, jr		s_readAddr1
beq, bne	s_branch_ID , s_branch_EX , s_branch_MEM, s_branch_WB ,	
j, jal, jr	s_jump_ID , s_jump_EX , s_jump_MEM, s_jump_WB ,	

[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.*

Dependency which Require Hazard Stalls		Dependency that can be forwarded	
s_writeAddr_ID, s_writeAddr_EX, s_writeEnable_ID, s_writeEnable_EX,	s_readAddr1, s_readAddr2,	s_writeAddr_MEM, s_writeEnable_MEM,	s_readAddr1, s_readAddr2,

In order for a dependency to be registered, either of the readAddr wires must match a writeAddr value (in any cycle), and the writeEnable in the same cycle must be enabled.

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

s_PCPlusFour_ID, s_DecodeData1_ID, s_DecodeData2_ID, s_imm32_ID, s_jumpAddress_ID, s_RegWrAddr_ID, s_Ctrl_ID,	s_PCPlusFour_EX, s_finalJumpAddress_EX, s_branchAddress_EX, s_RegOutReadData2_EX, s_aluOut_EX, s_RegWrAddr_EX, s_ALUBranch_EX, s_Ovfl_EX,	s_PCPlusFour_MEM, s_finalJumpAddress_MEM, s_branchAddress_MEM, s_DMemOut, s_aluOut_MEM, s_RegWrAddr_MEM, s_branch_check_MEM, s_Ovfl_MEM,	s_PCPlusFour_WB, s_finalJumpAddress_WB , s_branchAddress_WB, s_memReadData_WB, s_aluOut_WB, s_RegWrAddr, s_branch_check_WB, s_Ovfl,

	s_jal_EX, s_MemtoReg_EX, s_DMemWr_EX, s_RegWr_EX, s_Branch_EX, s_jump_EX, s_Halt_EX,	s_jal_MEM, s_MemtoReg_MEM, s_RegWr_MEM, s_jump_MEM, s_Halt_MEM,	s_jal_WB, s_MemtoReg_WB, s_RegWr, s_jump_WB, s_Halt

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

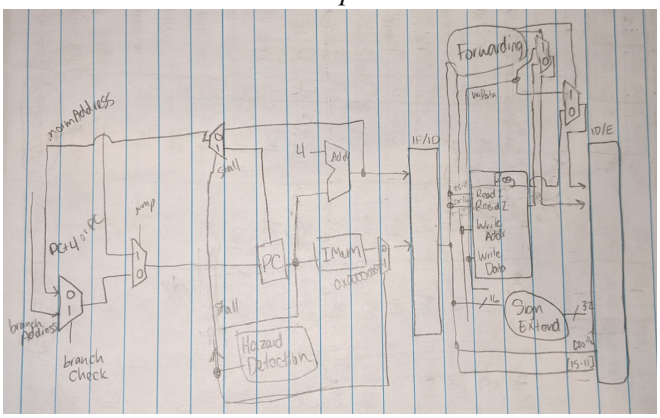
Instruction	Pipeline Stage
beq, bne, j, jal, jr	WB

[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.*

Because we placed our Hazard Detection module inside the Fetch stage of the pipeline, the processor will not load an instruction if there is any branching or jumping instructions in the pipeline already. Therefore, no stages need to be squashed or flushed, and the processor will stall itself and wait for the hazard to resolve itself.

To answer the question, it is possible that stages IF, ID, EX, and MEM may need to wait while the jump instruction is in the WB stage.

[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 set of assembly programs that exhaustively test the data forwarding and hazard detection capabilities of your pipeline. Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.*

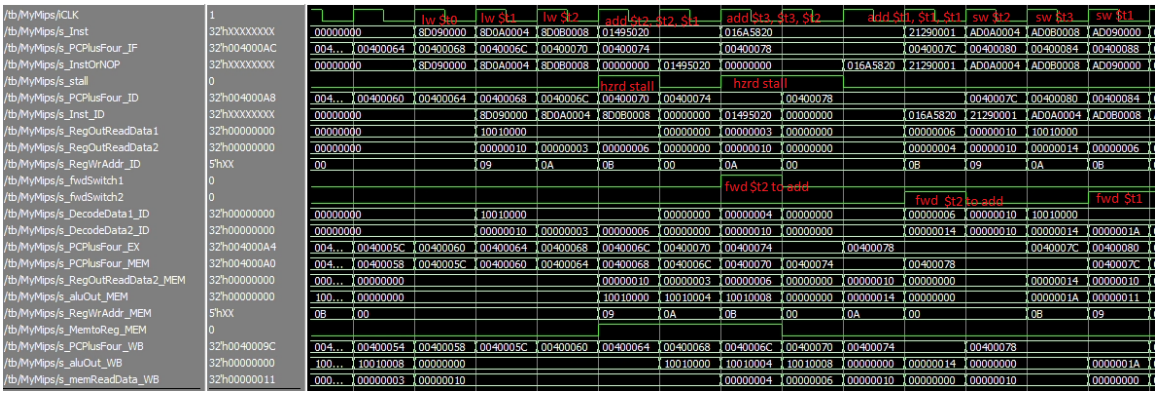
Type	Description	Example
IF -> ID	Instruction in IF depends on data in ID	addi $t0, $zero, 0x1 sw $t0, 4($sp)

IF -> EX	Instruction in IF depends on data in EX	lw $t1, 4($sp)
		lw $t2, 8($sp)
		add $t3, $t1, $t1

IF -> MEM	Instruction in IF depends on data in MEM	add $t2, $t1, $t0
		add $t1, $t0, $t0
		sw $t3, 4($sp)
		sw $t2, 8($sp)

The comprehensive testing program we created throws examples of each at the processor, and throws an example of all three at once at the processor. It runs successfully on our processor.

```
Starting Simulation for : mips\proj2test_data-fwd_hzd-det.s
Skipping compilation
Starting VHDL Simulation...
Successfully simulated program!
Victory!! Your processes matches MARS expected output with no mismatches!!
```

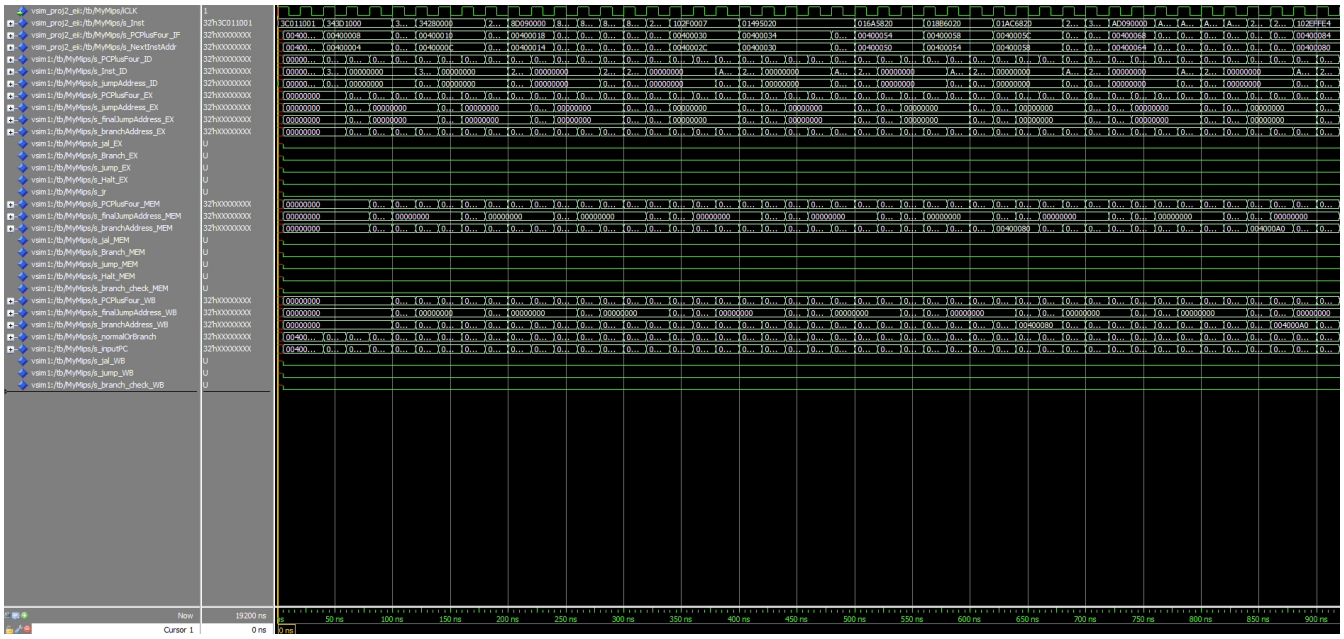


In the above example, we see examples of stalling and forwarding from EX/MEM and MEM/WB.

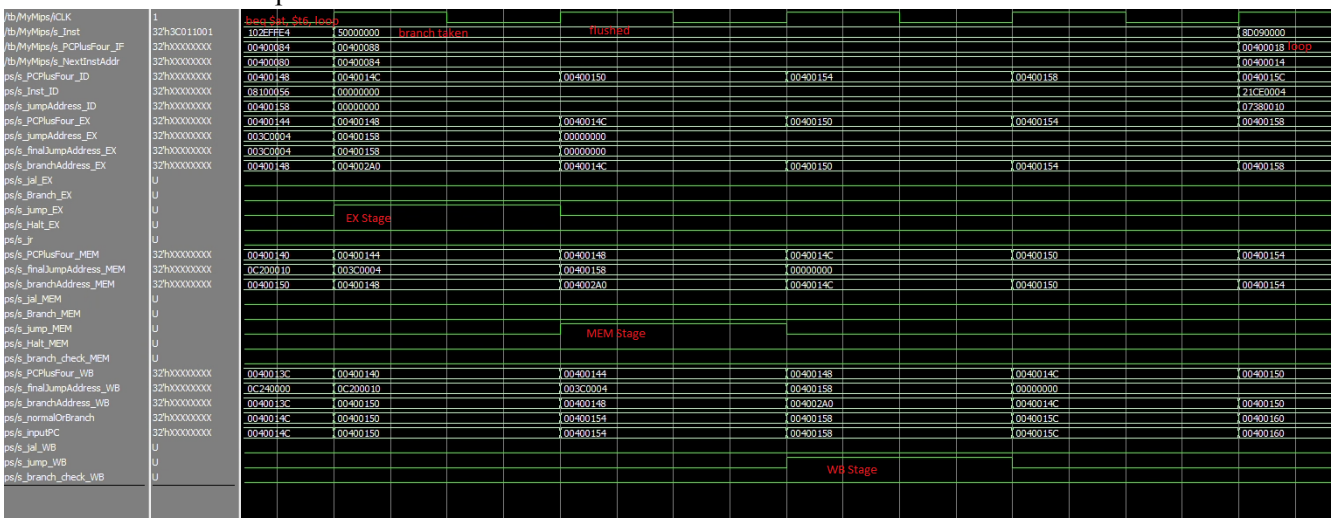
[2.e.ii] *Create a set of assembly programs that exhaustively tests control hazard avoidance. Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.*

Type	Description	Example
Jump	Invariably change PC	j loop
Branch	Conditionally change PC	bne $t0, $zero, example

```
Starting Simulation for : mips\proj2test_ctrl-haz.s
Skipping compilation
Starting VHDL Simulation...
Successfully simulated program!
Victory!! Your processes matches MARS expected output with no mismatches!!
```



Branch example:



The testing program used plenty of jumps and conditional branches, the processor did not encounter any errors, and analysis of the waveforms shows no problems.

[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 critical path of the hardware pipelined processor comes out to 18.41 nanoseconds which is a frequency of 54.32 MHz. The critical path goes from the pcReg register, into the Instruction memory, through the hazard detection unit, and ends at the IF/ID register.