

44CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Conner Ohnesorge

Aidan Foss

Daniel Vergara

Project Teams Group #: _____

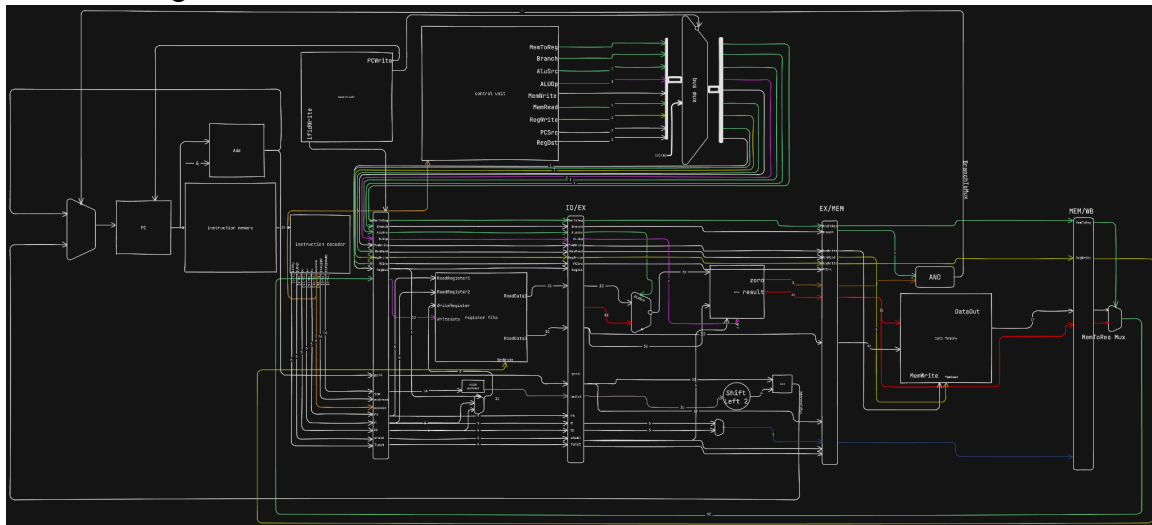
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

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

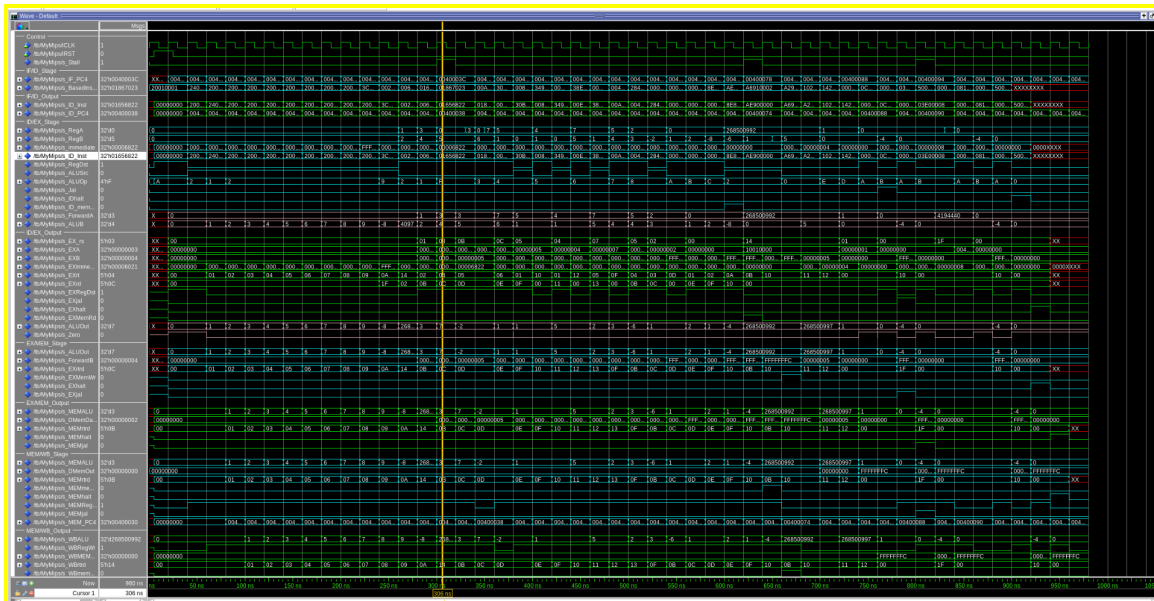
IF/ID STAGE		ID/EX STAGE		EX/MEM STAGE		MEM/WB STAGE	
Inputted Signal	New Signals	Inputted Signal	New Signals	Inputted Signal	New Signals	Inputted Signal	New Signals
MemToReg	ReadData1	MemToReg	ALUZero	MemToReg	BranchToMux	MemToReg	MemToRegMux
Branch	ReadData2	Branch	ALUResult	Branch	DataOut	RegWrite	MEMWBRegRD
AluSrc	immExt	AluSrc	AddressToMux	MemWrite	EXMEMRegRD	DataOut	
AluOp		AluOp	FwdBOut	MemRead		ALUResult	
MemWrite		MemWrite	RegisterRD	RegWrite		RegisterRD	
RegWrite		RegWrite		PCSrc			
PCSrc		PCSrc		ALUZero			
RegDst		RegDst		ALUResult			
PC+4		ReadData1		ReadData2			
Imm		ReadData2		FwdBOut			
Address		PC+4		PC+4			
Opcode		immExt		RegisterRD			
rs		rs					
rt		rt					
rd		rd					
shamt		shamt					
funct		funct					

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

Software Diagram



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

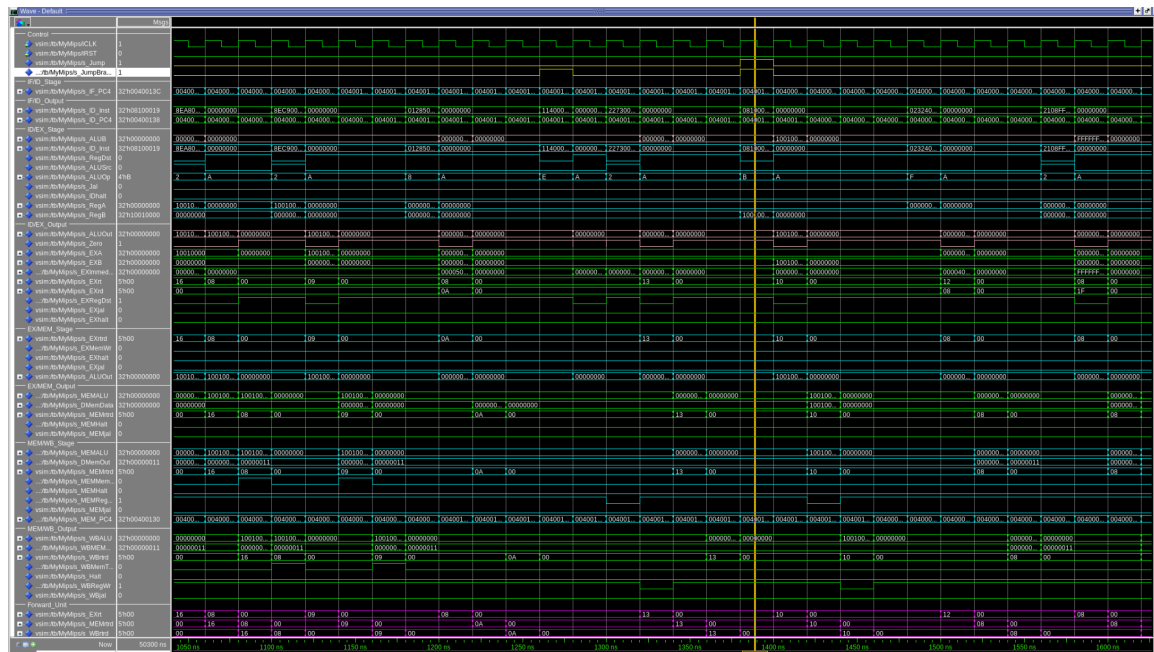


At the cursor, the fifth instruction listed below is started:

```
add  $11, $1, $2    # $11 = $1 + $2    (1+2 = 3)
addu $12, $3, $4    # $12 = $3 + $4    (3+4 = 7)
nop
nop
sub  $13, $11, $5    # $12 = $11 - $5 (3-5 = -2)
```

The processor correctly ends up with the value -2, shown in pink to the right of the cursor. Using ALUOp 4'hF, or 1111, it subtracts the values and sends them to ALUOut, which is then passed into the input of the next stage in the pipeline. This is used in the

[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs (bubblesort) 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 cursor is hovering over a jump instruction. If you follow the pipeline stages down, separated with dividers, you can see where the jump executes.
The following are excerpts from our bubble sort assembly program.

```

.data
.align 2
vals: .word 17 38 20 8008 69 420 100 60 0 12 # address: 0x10010000
vals_length: .word 10 # address: 0x10010028
.text
.globl main

main:
    lui $s1, 0x1001
    lui $s0, 0x1001
    addi $s2, $zero, 0
    nop
    ori $s1, $s1, 0x0028
    nop
    nop
    nop
    lw $s1, 0($s1)
outer_loop_cond:
    nop
    nop
    nop
    addi $t0, $s1, -1
    nop
    nop
    nop
    slt $t1, $s2, $t0
    nop
    nop
    nop
    bne $t1, $zero, outer_loop_body
    nop
    j exit_outer_loop
outer_loop_body:
    add $s4, $zero, $zero
    add $s3, $zero, $zero
inner_loop_cond:
    sub $t0, $s1, $s2
    nop
    nop
    nop
    addi $t0, $t0, -1
    nop
    nop
    nop
    slt $t0, $s3, $t0
    nop
    nop
    nop
    bne $t0, $zero, inner_loop_body
    nop
    nop
    j exit_inner_loop
    nop
    nop
    nop
inner_loop_body:
    sll $t0, $s3, 2
    nop
    nop
    nop
    add $s5, $s0, $t0
    nop
    nop
    nop
    addi $s6, $s5, 4
    lw $t0, 0($s5)
    nop
    nop
    lw $t1, 0($s6)
    nop
    nop
    nop
    slt $t2, $t1, $t0
    nop
    nop
    nop
    beq $t2, $zero, inner_loop_footer
    nop
    sw $t0, 0($s6)
    sw $t1, 0($s5)
    addi $s4, $zero, 1
inner_loop_footer:
    addi $s3, $s3, 1
    nop
    nop
    nop
    j inner_loop_cond
exit_inner_loop:
    nop
    beq $s4, $zero, exit_outer_loop
outer_loop_footer:
    nop
    addi $s2, $s2, 1
    nop
    nop
    j outer_loop_cond
    nop
exit_outer_loop:
    j exit
exit:
    halt

```

1. Data-Flow Example:

```
```\mips
lw $t0, 0($s5)
nop
nop # Only 2 NOPs used
lw $t1, 0($s6)
```\
```

Here only 2 NOPs are needed after the load instruction because the value from the first load (\$t0) isn't used immediately by the second load instruction. The second load is independent of the first load's result.

2. Control-Flow Example:

```
```\mips
beq $s4, $zero, exit_outer_loop
outer_loop_footer:
nop # Only 1 NOP used
addi $s2, $s2, 1
```\
```

Only 1 NOP is used here because the branch instruction and increment are independent - the increment operation doesn't affect or depend on the branch condition (\$s4).

3. Another Data-Flow Example:

```
```\mips
nop # Only 2 NOPs used
nop
beq $t2, $zero, inner_loop_footer
```\
```

Before the branch instruction, only 2 NOPs are used because the comparison value in \$t2 was already stable and the branch doesn't require the full 3-NOP delay in this case.

[For teams with >4]

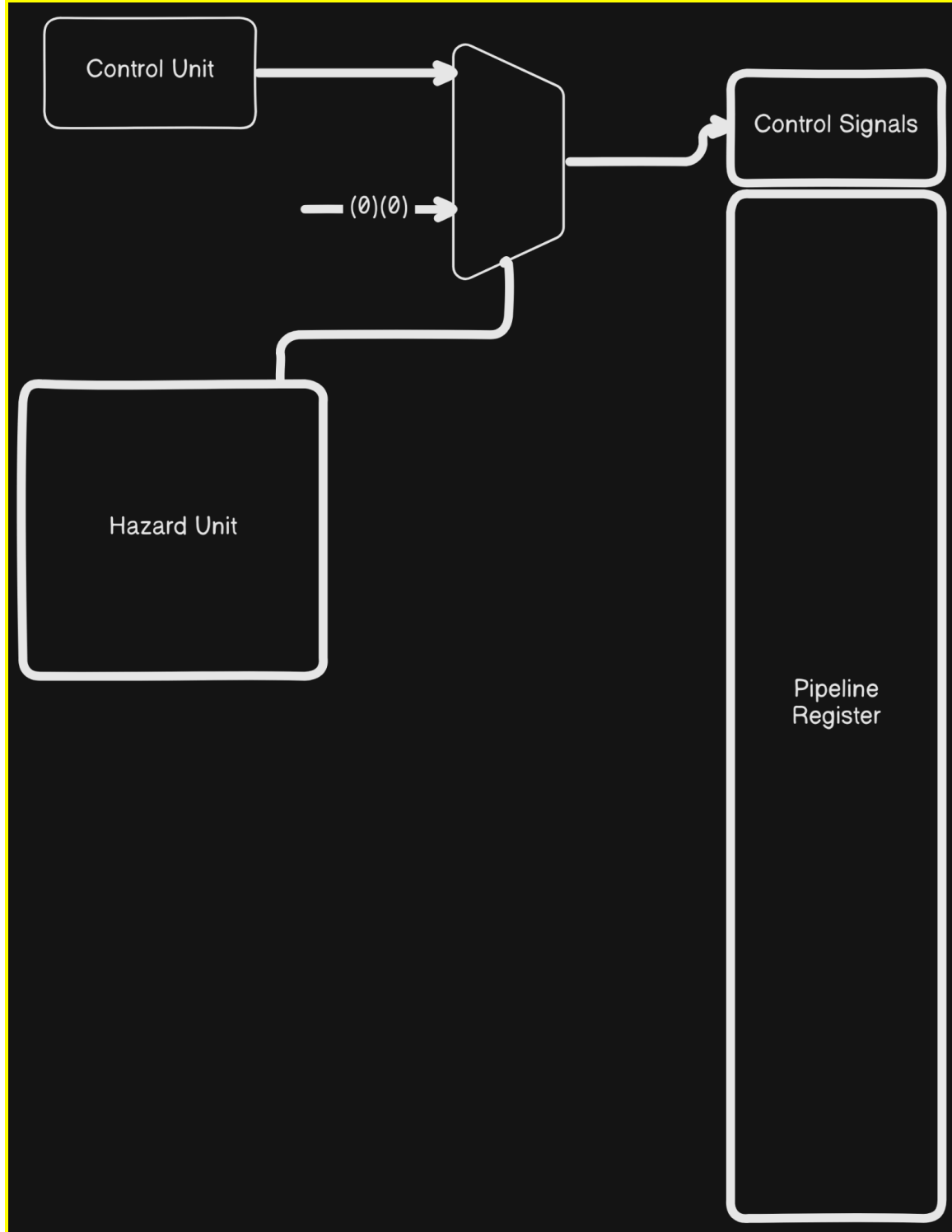
[1.c.iii] Include an annotated waveform in your writeup of two iterations or recursions of these programs (mergesort) executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of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

[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).

```
FMax: 52.86mhz  
From Node      :  
mem:IMem|altsyncram:ram_rtl_0|altsyncram_eg81:auto_generated|ra  
m_block1a0~porta_we_reg  
To Node        : dffg_n:instPC|dffg:\G1:13:flipflop|s_Q
```

Thus, we infer that our critical path is from the PC through the registerfile as we spend over 7ns in that region during instruction fetch. More specifically, our branch/jump detection unit (zero detector) is the slowest part of the instruction fetch process.

[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

		Msgs				
+	/tb_stages/S_CLK	0				
+	/tb_stages/IF_PC4	32'h55555555	DDD... CCC... AAA... 55555555 33333333			
+	/tb_stages/ID_PC4	32'hAAAAAAAA	FFFF... DDD... CCC... AAAAAAAAAA 00000000			
+	/tb_stages/EX_PC4	32'hCCCCCCCC	XXX... FFFF... DDD... CCCCCCCC 00000000			
+	/tb_stages/MEM_...	32'hDDDDDDDD	XXXXXXXXXX FFFF... DDDDDDDD 00000000			
+	/tb_stages/WB_P...	32'hFFFFFFF	0000... XXXXXXXX FFFFFFFF 00000000			
+	/tb_stages/MEM...	32'hXXXXXXXX	XXXXXXXXXX 00000000			

Called `tb_MipsProcessor.vhd`.

Instruction Format	Instruction Type	Instruction	Produced Values	Pipeline Signals
R-Type	Arithmetic and Logical	add, addu	s_ALUOut, s_ALUB	s_ALUOp, s_RegWr
I-Type	Arithmetic and Logical	addi, addiu	s_ALUOut, s_ALUB	s_ALUOp, s_RegWr
R-Type	Arithmetic and Logical	sub, subu	s_ALUOut, s_ALUB	s_ALUOp, s_RegWr
R-Type	Arithmetic and Logical	and, or, xor, nor	s_ALUOut, s_ALUB	s_ALUOp, s_RegWr
I-Type	Arithmetic and Logical	andi, ori, xori	s_ALUOut, s_ALUB	s_ALUOp, s_RegWr
R-Type	Arithmetic and Logical	slt	s_ALUOut, s_ALUB	s_ALUOp, s_RegWr
I-Type	Arithmetic and Logical	slti	s_ALUOut, s_ALUB	s_ALUOp, s_RegWr
R-Type	shift	sll, srl, sra	s_ALUOut, s_ALUB	s_ALUOp, s_ShiftDirection, s_ShiftType, s_RegWr
I-Type	Mem Access	lw, lh, lb, lbu	s_ForwardA, s_ForwardB	s_memToReg, s_ALUSrc, s_WE, s_muxRegWr, s_ALUOp
I-Type	Mem Access	lui	s_RegA, s_ForwardA	s_memToReg, s_ALUSrc, s_WE, s_muxRegWr, s_ALUOp
I-Type	Mem Access	sw	No "Produced Value", but it updates a memory address with new content	s_ALUSrc, s_Stall, s_MemWr
J-Type	Control Flow	j	BranchToMux	s_J, s_Jump,
J-Type	Control Flow	jr	BranchToMux	s_Jr, s_Jump,
J-Type	Control Flow	jal	BranchToMux	s_Jal, s_Jump,
J-Type	Control Flow	beq, beq	BranchToMux	s_Branch, s_Jump, s_Signed, s_Beq

Instruction Type	Instruction	Consumed Values	Pipeline Signals
Arithmetic and Logical	add, addu	Values from rs and rt	s_rs, s_rt
Arithmetic and Logical	addi, addiu	Value from rs and immediate	s_rs, s_imm
Arithmetic and Logical	sub, subu	Values from rs and rt	s_rs, s_rt
Arithmetic and Logical	and, andi	Values from rs and rt/imm	s_rs, s_rt/imm
Arithmetic and Logical	or, ori	Values from rs and rt/imm	s_rs, s_rt/imm
Arithmetic and Logical	xor, xori	Values from rs and rt/imm	s_rs, s_rt/imm
Arithmetic and Logical	nor	Values from rs and rt	s_rs, s_rt
Arithmetic and Logical	sll, slli	Values from rs and rt/imm	s_rs, s_rt/imm
Shift	sll, srl, sra	Value from rt and shift amount	s_rt, shamt
Shift	sllv, srlv, sra	Values from rs and rt	s_rs, s_rt
Memory Access	lw, lh, lbu, lb, lbu	Base address (rs) and offset	s_rs, s_imm
Memory Access	sw	Base address (rs) and data (rt)	s_rs, s_rt
Control Flow	jr	Jump address from rs	s_rs
Control Flow	beq, bne	Values from rs and rt	s_rs, s_rt

[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.

Load-Store:

Data dependencies can come up when a store instruction comes right after a load instruction. The store instructions need data to be loaded in the previous cycle.

Immediate to Immediate:

Dependencies can happen when the oneImmediate uses the result from the other.

Register to Register to Register:

Dependencies exist when multiple register-using instructions need the register results or use the same registers in different instructions.

Register to Register:

Dependencies can exist between two register-using instructions.

Immediate to Register:

Dependencies can come up when an immediate instruction writes to a register that a following register-using instructions will need the result.

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

IF/ID STAGE		ID/EX STAGE		EX/MEM STAGE		MEM/WB STAGE	
Inputted Signal	New Signals	Inputted Signal	New Signals	Inputted Signal	New Signals	Inputted Signal	New Signals
MemToReg	ReadData1	MemToReg	ALUZero	MemToReg	BranchToMux	MemToReg	MemToRegMux
Branch	ReadData2	Branch	ALUResult	Branch	DataOut	RegWrite	MEMWBRegRD
AluSrc	immExt	AluSrc	AddressToMux	MemWrite	EXMEMRegRD	DataOut	
AluOp		AluOp	FwdBOut	MemRead		ALUResult	
MemWrite		MemWrite	RegisterRD	RegWrite		RegisterRD	
RegWrite		RegWrite		PCSrc			
PCSrc		PCSrc		ALUZero			
RegDst		RegDst		ALUResult			
PC+4		ReadData1		ReadData2			
Imm		ReadData2		FwdBOut			
Address		PC+4		PC+4			
Opcode		immExt		RegisterRD			
rs		rs					
rt		rt					
rd		rd					
shamt		shamt					
funct		funct					

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

Instruction	Instruction Type	Pipeline Stage	Reason for Non-Sequential PC Update
j	Jump	Decode (ID)	PC updated to the jump target address (computed from instruction).
jal	Jump and Link	Decode (ID)	PC updated to the jump target address; return address stored in \$ra.
jr	Jump Register	Execute (EX)	PC updated to the address stored in rs.
jalr	Jump and Link Register	Execute (EX)	PC updated to the address in rs; return address stored in \$ra.
beq	Branch on Equal	Execute (EX)	PC updated to branch target if $rs == rt$.
bne	Branch on Not Equal	Execute (EX)	PC updated to branch target if $rs \neq rt$.
bltz	Branch on Less Than 0	Execute (EX)	PC updated to branch target if $rs < 0$.
bgez	Branch on Greater/Equal	Execute (EX)	PC updated to branch target if $rs \geq 0$.
beql	Branch on Equal Likely	Execute (EX)	Similar to beq, but skips next instruction if branch condition is false.
bnel	Branch on Not Equal Likely	Execute (EX)	Similar to bne, but skips next instruction if branch condition is false.

[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.

Stages Stalled: ID/EX

Stages Squashed/Flushed: IF/ID, ID/EX, EX/MEM

For the instructions BNE, BEQ, J, JAL, and JR that result in a non-sequential PC update, the following stages need to be stalled and squashed/flushed relative to the stage each instruction is in:

Stalling:

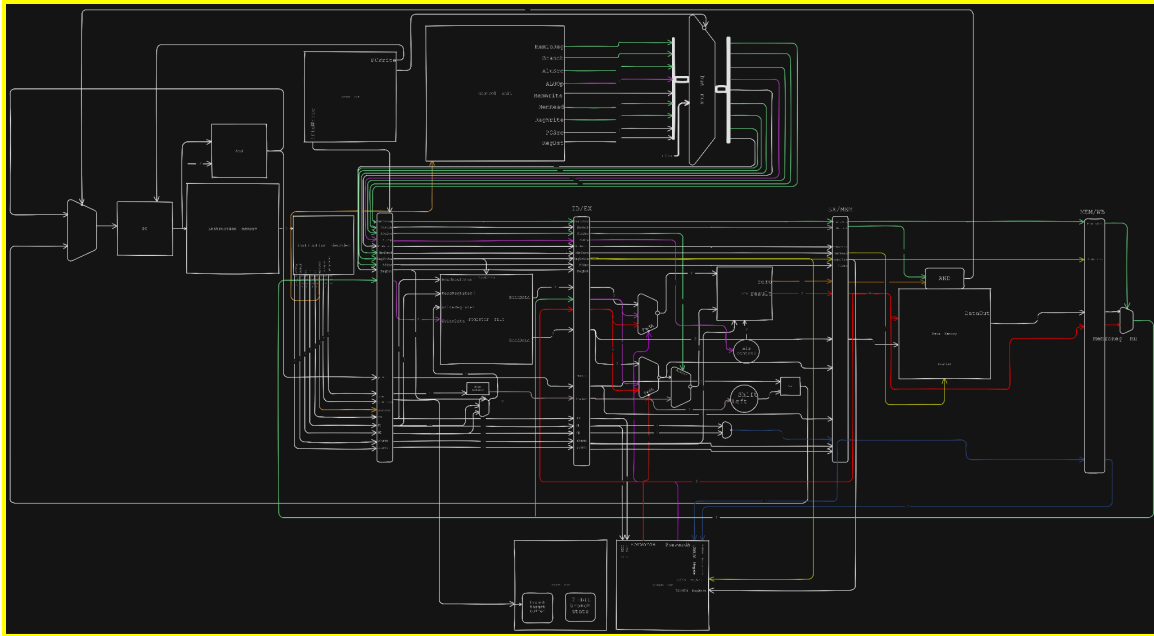
- Stalling occurs for 1 cycle.
- The IF/ID pipeline register is stalled to prevent the next instruction from entering the Decode (ID) stage.
- The ID/EX pipeline register is stalled to prevent the instruction in the Decode (ID) stage from moving to the Execute (EX) stage.

Flushing:

- Flushing occurs for 1 cycle.
- Flushing takes place in the Fetch (IF) stage to discard the instruction that was fetched based on the sequential PC.
- Instructions producing values on the ALU Result signal in the execute stage:
 - add, addi, addiu, addu
 - sub, subu
 - and, andi
 - or, ori
 - xor, xori
 - nor
 - slt, slti
 - sll, srl, sra
- Instructions producing/using values on the mem data signal in the memory stage:
 - lw, lh, lhu, lb, lbu (load instructions)
 - sw, sh, sb (store instructions)
- Instructions producing values on the branchLogic signal going into the fetch logic unit:
 - beq, bne
- Instructions producing values on the JumpLogic and JregLogic signals going into the fetch logic unit:
 - j, jal
 - jr (uses iJreg signal specifically for register jumping)

[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.

Hardware Diagram



Implemented in source_hw

[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

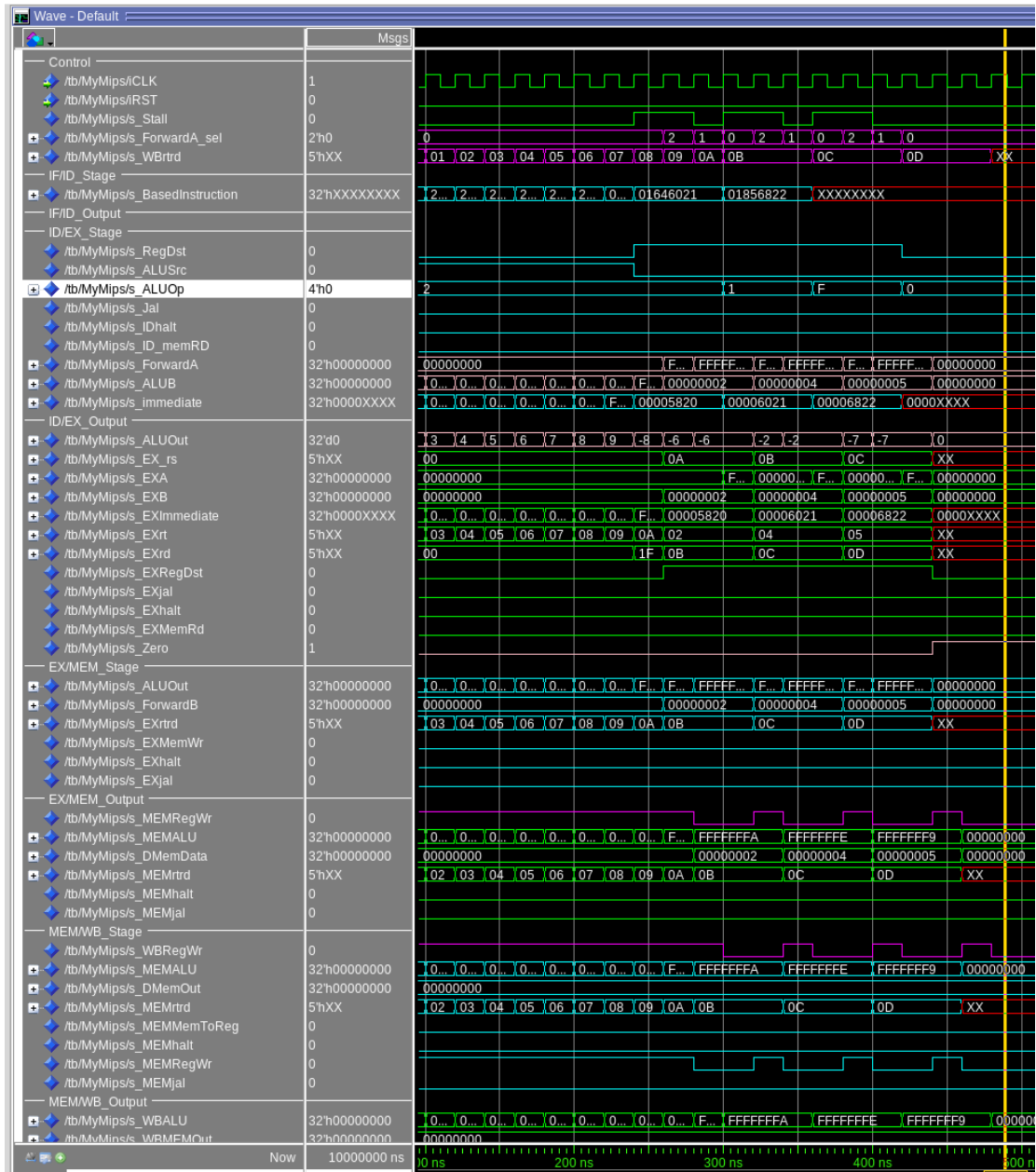


add \$11, \$1, \$2 # \$11 = \$1 + \$2 (1+2 = 3)

addu \$12, \$3, \$4 # \$12 = \$3 + \$4 (3+4 = 7)

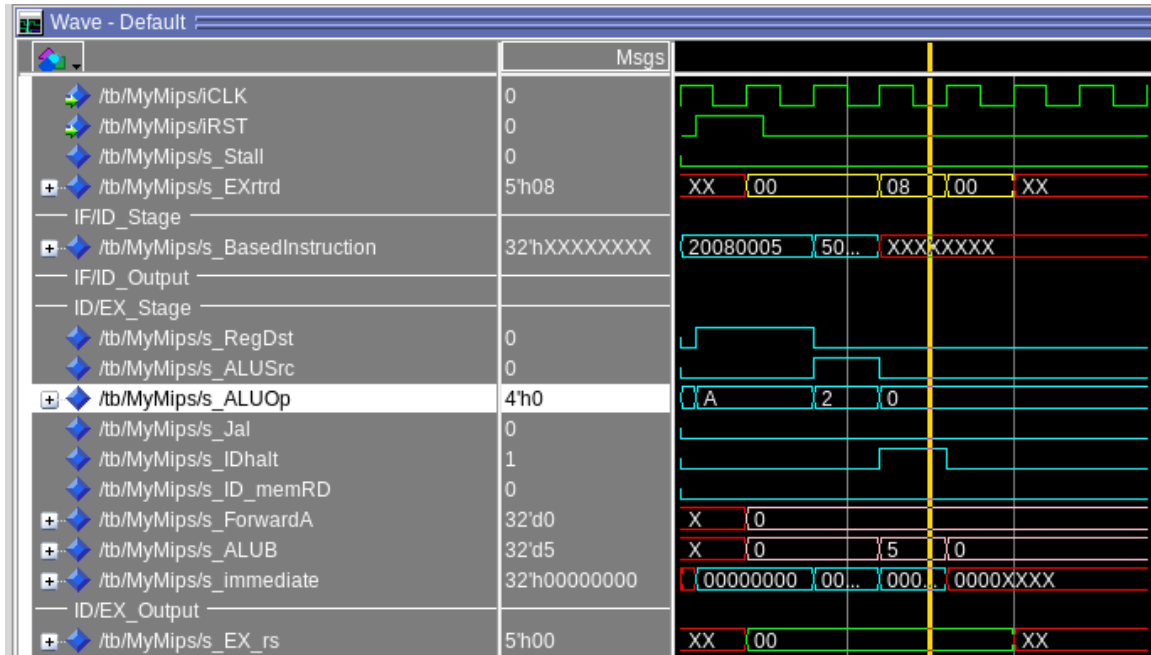
sub \$13, \$11, \$5 # \$12 = \$11 - \$5 (3-5 = -2) RegToReg Hazard Here (The cursor is over this instruction)

The processor correctly detects the RegToReg Hazard, and the forwarding unit forwards the value of register 11 to the last instruction after a short stall. The forwarding unit is in magenta, and the hazard unit is in yellow near the top. Their values are pipelined properly to pass the relevant information (in this case, that register 11 ended up being equal to 3) to the appropriate location

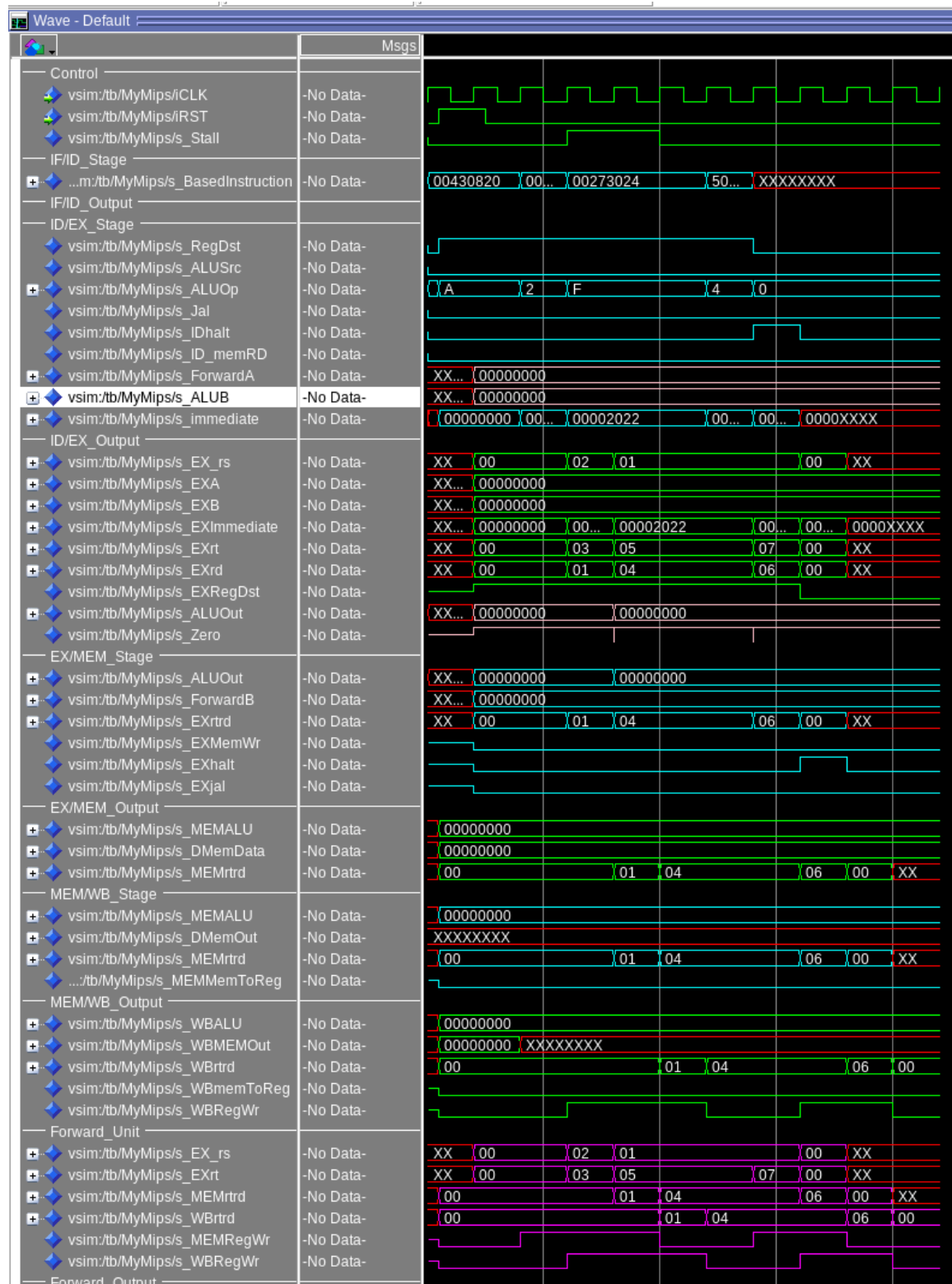


This image is of RegToRegToReg.s, which is testing the RegToRegToReg Hazard, In this case, there are multiple stalls that are correctly handled. You can see the forwarding unit in purple. Each line below has a hazard except for the addi. The processor needs to stall for two cycles each, which is correctly handled. These are all ReadAfterWrite Hazards.

```
addi $10, $0, -8      # Place "10" in $10
add  $11, $10, $2      # $11 = -8 + 2 = -6 RAW
addu $12, $11, $4      # $12 = -6 + 4 = -2 RAW
sub  $13, $12, $5      # $12 = -2 - 5 = -7 RAW
```

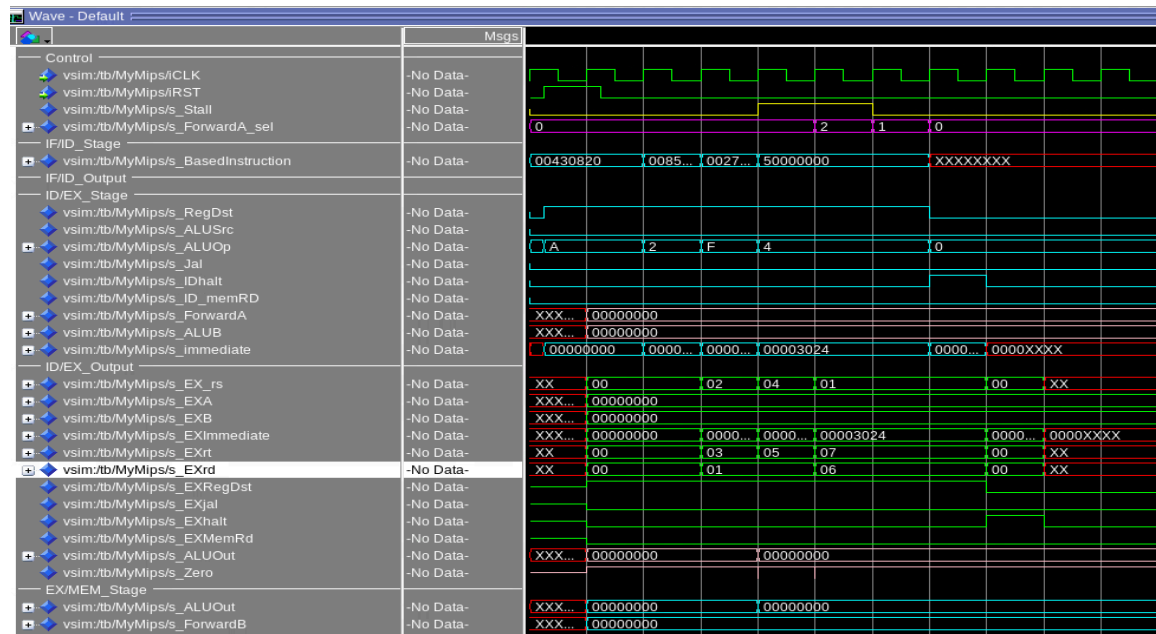


In this image, I have ImmtoImm.s shown. this just runs one instruction: `addi $t0, $zero, 5`. This is problematic, as If it were possible to change the set value of 0 to something else, it could cause major issues. Our processor handles this instruction as a nop, as it doesn't actually change any values in memory or in the register file. Hazard Detector unit signal is shown in yellow.

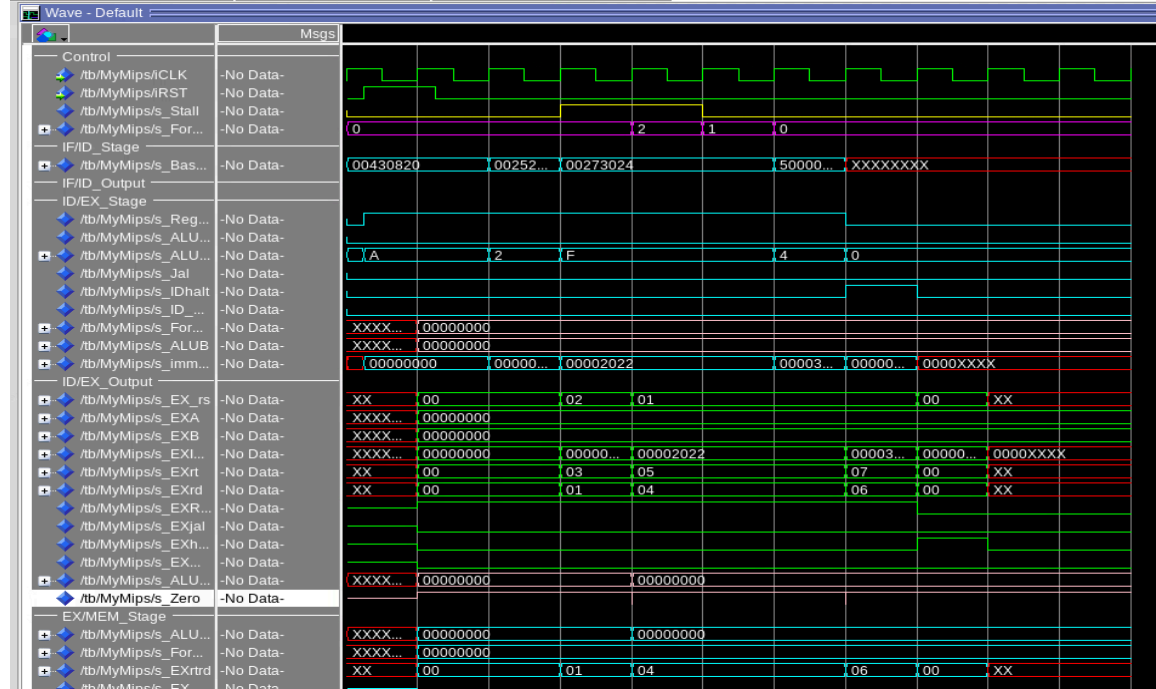


Here is a WAR Hazard Getting Detected. I used WAR.s in my MIPS programs, which ran the instructions below. It properly detects the hazard and forwards anything relevant, as well as running a stall.

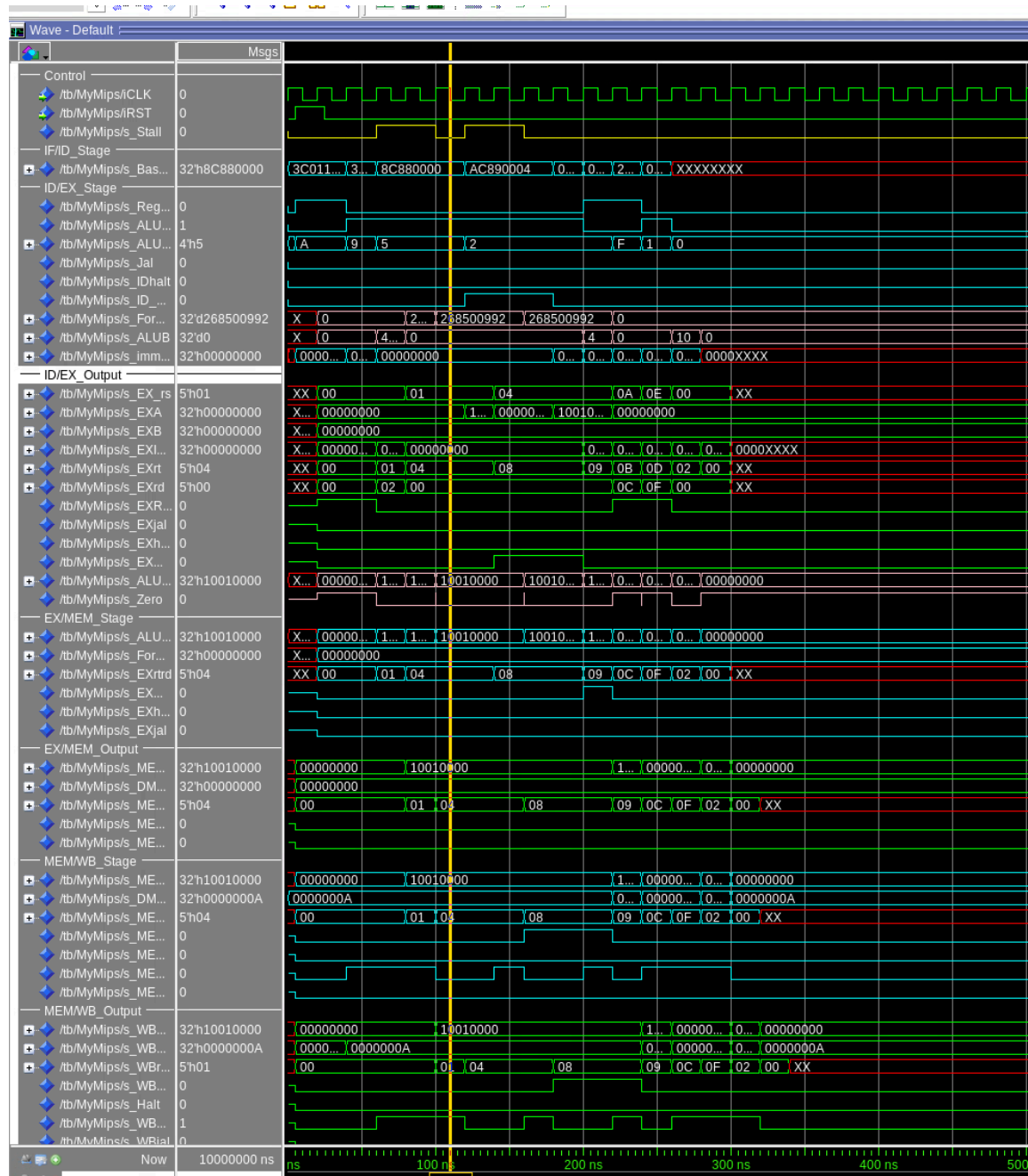
add \$1, \$2, \$3
sub \$3, \$4, \$5
and \$6, \$3, \$7



Here is a WAW hazard being detected. A stall is used to allow time for the forwarding unit to forward the value that is depended on. Stall is shown in yellow, and the forwarded register is shown in magenta



Here is a RAW hazard being detected (RAW.s) A stall is used to allow time for the forwarding unit to forward the value that is depended on. Stall is shown in yellow, and the forwarded register is shown in magenta

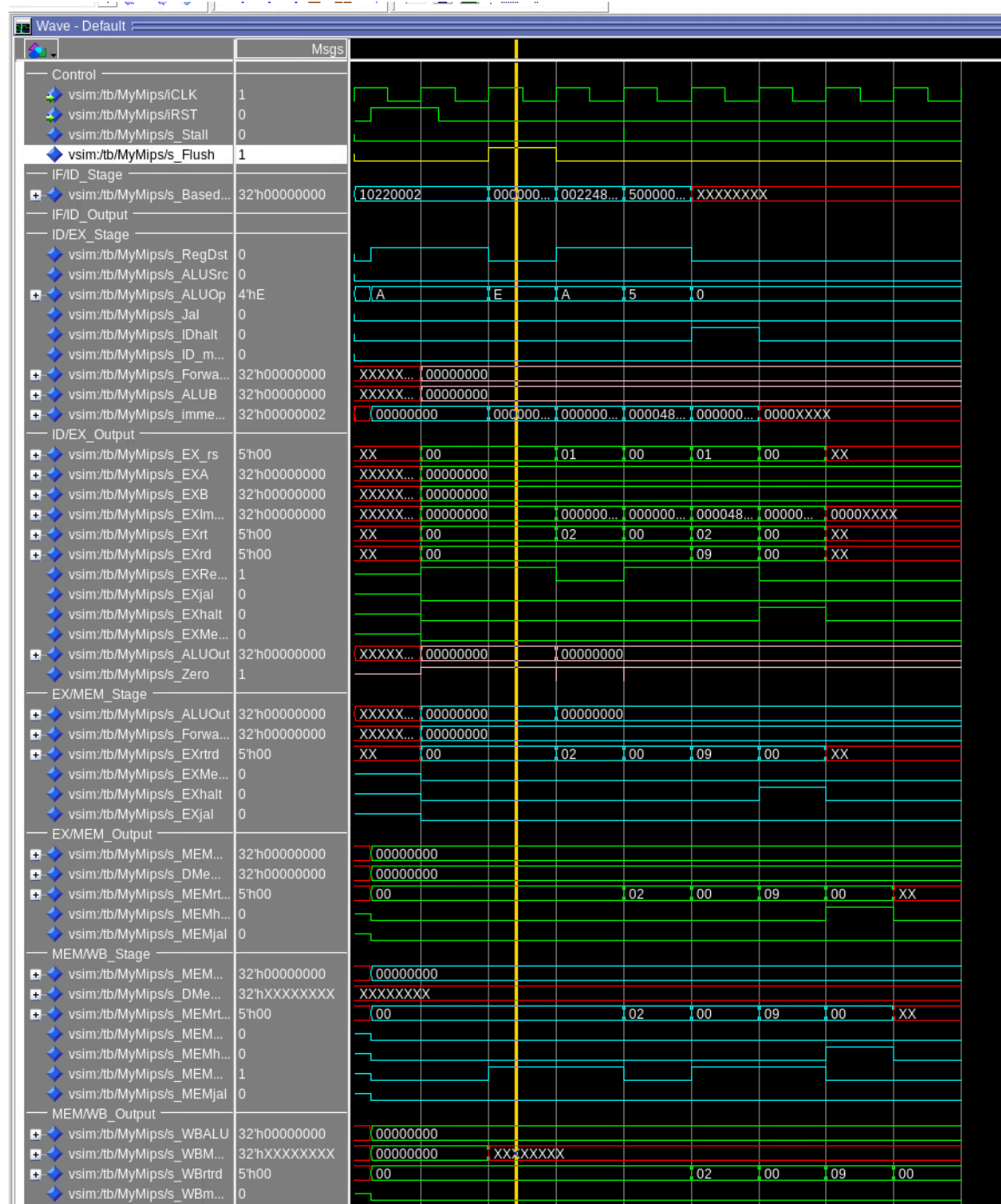


Here is an implementation of a Structural Hazard, as the LW and SW instructions rely on the register file at the same time, so we have to stall. The stall is in yellow at the top.

```

la $a0, array          # Load the address of the $a0 array
lw $t0, 0($a0)         # Load the first word from the array into $t0
sw $t1, 4($a0)         # Store the value from $t1 into the second word of the array
add $t4, $t2, $t3      # Add $t2 and $t3, storing the result in $t4
sub $t7, $t6, $t5      # Subtract $t5 from $t6, storing the result in $t7

```



Here is an implementation of a control hazard. In this case, no stalls were needed, but an instruction did need to get flushed from the pipeline. This is because the next instruction got loaded, and then the Hazard Detection realized that the instruction shouldnt have ever been loaded, so it wipes it from the pipeline.

```

beq $1, $2, Label
add $3, $4, $5
sub $6, $7, $8
beq $1, $2, Label
Label:
or $9, $1, $2
halt

```

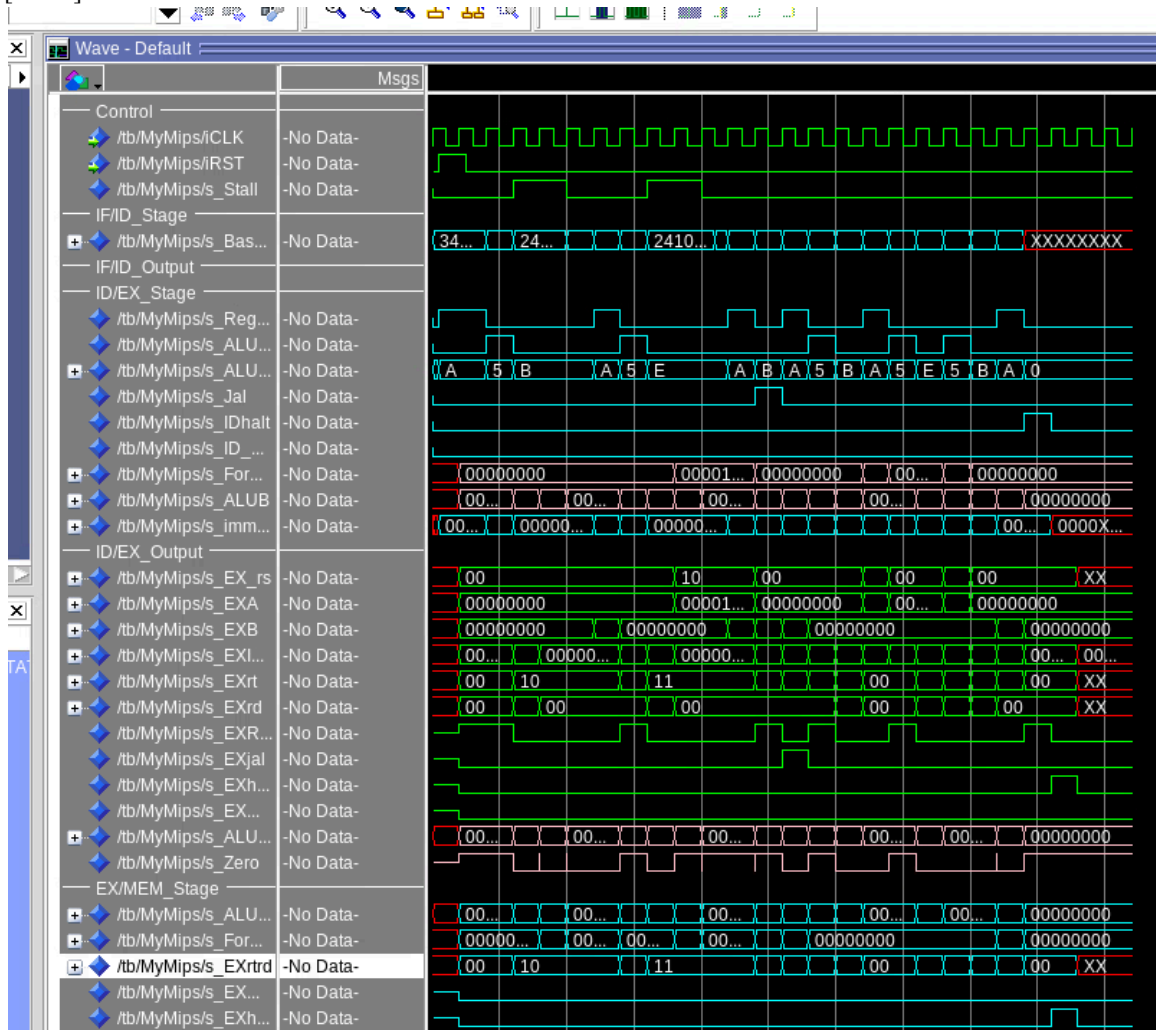
[2.e.i]

Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

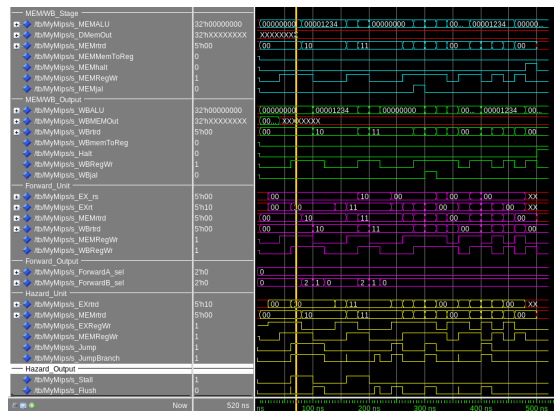
A	B	C	D
Hazard	MIPS File	Instructions Example	Justification
WAR Hazard	WAR.s	add \$1, \$2, \$3 sub \$3, \$4, \$5 and \$6, \$3, \$7	For all three of these, we need to insure that forwarding paths are implemented properly to minimize the length of pipeline stalls
RAW Hazard	RAW.s	add \$1, \$2, \$3 sub \$4, \$1, \$5 and \$6, \$1, \$7	
WAW Hazard	WAW.s	add \$1, \$2, \$3 sub \$1, \$4, \$5 and \$6, \$1, \$7	
Structural Hazard	StructHaz.s	lw \$1, 0(\$2) sw \$1, 4(\$3) add \$4, \$1, \$5	Tests resource conflicts, priorities must be assigned or maintained. Stalls must be kept at a minimum
Control Hazard	CtrlHaz.s	beq \$1, \$2, Label add \$3, \$4, \$5 sub \$6, \$7, \$8 Label: or \$9, \$1, \$2 jr \$ra	Corrects issues with incorrect speculation or assumptions. Validate branch decisions

Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

[2.e.iii]



Here is our simplebranch.s implementation from Project1, which is the script shown to the right. The signals below show that the new system is able to run all those instructions pipelined,



```
main:
    ori $s0, $zero 0x1234
    j skip
    li $s0 0xffffffff

skip:
    ori $s1 $zero 0x1234
    beq $s0 $s1 skip2
    li $s0 0xffffffff

skip2:
    jal fun
    ori $s3 $zero 0x1234

    beq $s0, $zero exit
    ori $s4 $zero 0x1234
    j exit

fun:
    ori $s2 $zero 0x1234
    jr $ra

exit:
    halt
```

[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).

```
FMax: 48.38mhz
```

```
From Node      :
```

```
mem:DMem|altsyncram:ram_rtl_0|altsyncram_eg81:auto_generated|ram_block1a0~porta_we_reg
```

```
To Node        :
```

```
EX_MEM:instEXMEM|dffg:\G_ALU_Reg:31:ALUDFFGI|s_Q
```

48.38 was our maximum frequency for our hardware implementation. As for our critical path, we inferred from the synthesis output that;