CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members:Justin Sebahar

Jackson Collalti

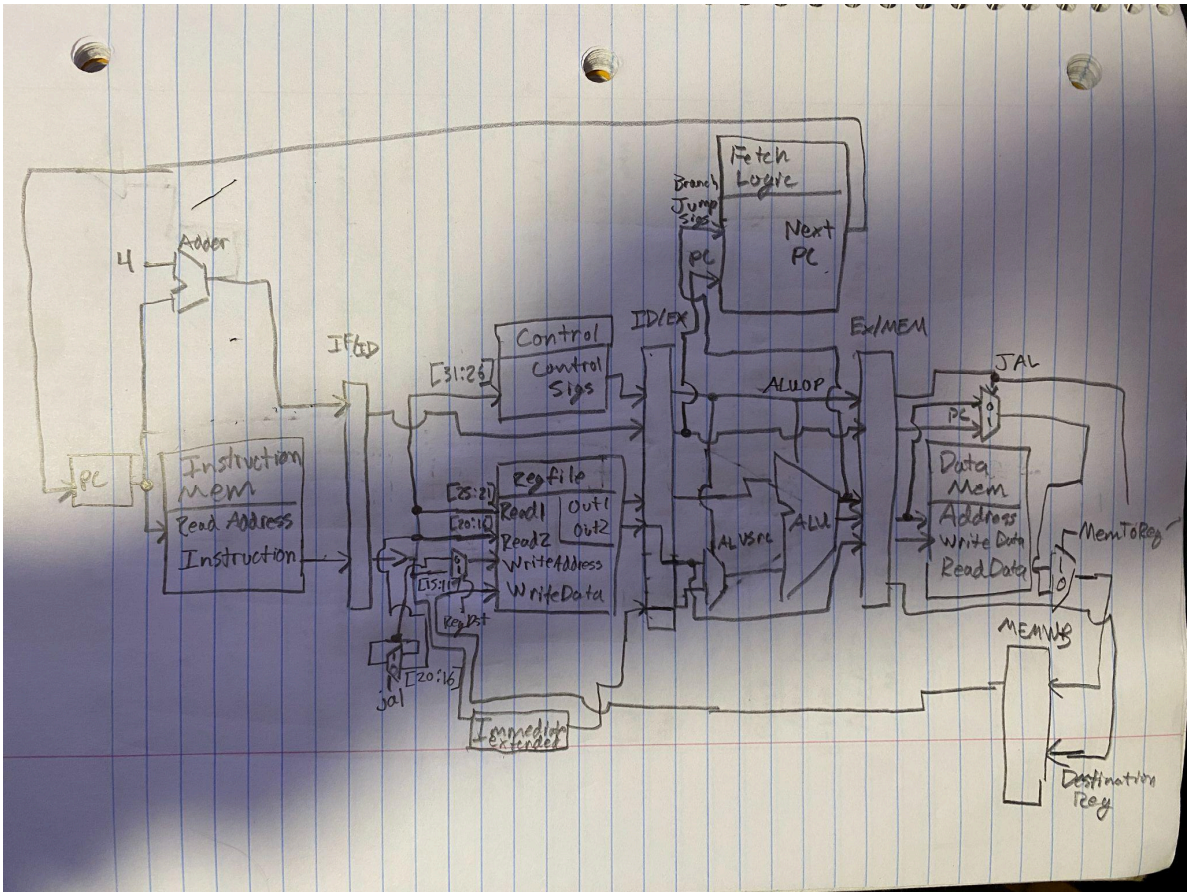
Project Teams Group #: D_04

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.

Fetch	Decode	Execute	DataMem	WriteBack
zero (ex) IMemAddr regOutRs (ex) JR (ex) Inst (ex) JUMP_EN (ex) imm_extended (ex) BRANCH (ex) BRANCH_TYPE (ex) NextInstAddr	Inst IMemAddr ALUOP DST_REG Reg_WR Mem_WR ALUSrc EXTENSION MEM_TO_REG BRANCH BRANCH_TYPE JUMP_EN JR JAL Halt regOutRs regOutRt imm_extended	IMemAddr Inst ALUOP DST_REG ALUSrc MEM_TO_REG Reg_WR Mem_WR BRANCH BRANCH_TYPE JUMP_EN JR JAL Halt regOutRt zero imm_extended Ovfl	IMemAddr Reg_WR Mem_WR MEM_TO_REG DST_REG JAL Halt ALURes DMemOut Inst regOutRs regOutRt Ovfl	MEM_TO_REG Reg_WR Halt DMemOut DST_REG Inst jal_mux2_out JAL Ovfl

[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 simple program used was the base test from Project 1, which used all supported instructions, and NOPs were added where necessary.

Test Results

```
[collalti@linuxvdi-40 proj2_sw]$ ./381_tf.sh test  proj/mips/Proj1_base_test.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Failed to remove output/Proj1_base_test.s. Is questasim open?
Saving instead to "output/Proj1_base_test.s_1"
Testing file: proj/mips/Proj1_base_test.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 50
Processor Cycles: 68
CPI:  1.36
Results in: output/Proj1_base_test.s_1
-----------------------------------------------
[collalti@linuxvdi-40 proj2_sw]$
```

Since Modelsim waveforms can't easily show all stages and all values present, I will break down the log file from Modelsim.

```
main:
        # Initialize some registers with values
        lui $1,4097
        nop
        nop
        ori $19,$1,0
        addi $s0, $0, 10           # $s0 = 10
        addi $s1, $0, 20           # $s1 = 20
        addi $s2, $0, -1           # $s2 = -1
        # One no opperation for $s1 to update for add $t0, $s0, $s1
        nop

        # Arithmetic operations
        add $t0, $s0, $s1    # $t0 = $s0 + $s1 = 10 + 20 = 30
        addi $t1, $s0, 5     # $t1 = $s0 + 5 = 10 + 5 = 15
        addiu $t2, $s2, 1    # $t2 = $s2 + 1 = -1 + 1 = 0
        addu $t3, $s0, $s1   # $t3 = $s0 + $s1 = 10 + 20 = 30
        sub $t4, $s1, $s0    # $t4 = $s1 - $s0 = 20 - 10 = 10
        subu $t5, $s1, $s0   # $t5 = $s1 - $s0 = 20 - 10 = 10
```

Corresponding Trace for instructions above

```
In clock cycle: 3
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 4
Register Write to Reg: 0x01 Val: 0x10010000
In clock cycle: 5
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 6
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 7
Register Write to Reg: 0x13 Val: 0x10010000
In clock cycle: 8
Register Write to Reg: 0x10 Val: 0x0000000A
In clock cycle: 9
Register Write to Reg: 0x11 Val: 0x00000014
In clock cycle: 10
Register Write to Reg: 0x12 Val: 0xFFFFFFFF
In clock cycle: 11
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 12
Register Write to Reg: 0x08 Val: 0x0000001E
In clock cycle: 13
Register Write to Reg: 0x09 Val: 0x0000000F
In clock cycle: 14
Register Write to Reg: 0x0A Val: 0x00000000
In clock cycle: 15
Register Write to Reg: 0x0B Val: 0x0000001E
In clock cycle: 16
Register Write to Reg: 0x0C Val: 0x0000000A
In clock cycle: 17
```

Cycle 3 represents the start of the program. Cycle 4 is lui instruction since the next “actual” instruction requires the LUI value 2 NOP’s are placed in between to wait for lui to evaluate. Cycle 7 is the ori using the lui value. Cycles 8-11 show standard addi’s without any dependencies. Cycle 12 is a nop waiting from Cycle 11 to be able to use Cycle 8’s value. The remaining cycle is testing of all other arithmetic operations without dependencies.

```
# Logical operations
and $t0, $s0, $s1    # $t0 = $s0 & $s1 = 0
andi $t7, $s0, 15    # $t7 = $s0 & 15 = 10
or $t8, $s0, $s1     # $t8 = $s0 | $s1 = 30
ori $t9, $s0, 25     # $t9 = $s0 | 25 = 27
xor $t3, $s0, $s1    # $t8 = $s0 ^ $s1 = 30
xori $t2, $s0, 15    # $t1 = $s0 ^ 15 = 5
nor $t1, $s0, $s1    # $t2 = ~($s0 | $s1) = FFFFFFE1 = -31

# Shift operations
sll $t3, $s0, 2      # $t3 = $s0 << 2 = 40
srl $t4, $s1, 2      # $t4 = $s1 >> 2 (logical shift) = 5
sra $t3, $t1, 1      # $t3 = $t1 >> 2 (arithmetic shift) = FFFFFFF0 = -16

# Comparison operations
slt $t6, $s0, $s1    # $t6 = ($s0 < $s1) true
slti $t7, $s0, 5    # $t7 = ($s0 < 5) false


# Branch and Jump instructions
bne $s0, $s1, label1 # If $s0 != $s1, jump to label1 true
nop
nop
```

```
In clock cycle: 17
Register Write to Reg: 0x0D Val: 0x0000000A
In clock cycle: 18
Register Write to Reg: 0x08 Val: 0x00000000
In clock cycle: 19
Register Write to Reg: 0x0F Val: 0x0000000A
In clock cycle: 20
Register Write to Reg: 0x18 Val: 0x0000001E
In clock cycle: 21
Register Write to Reg: 0x19 Val: 0x0000001B
In clock cycle: 22
Register Write to Reg: 0x0B Val: 0x0000001E
In clock cycle: 23
Register Write to Reg: 0x0A Val: 0x00000005
In clock cycle: 24
Register Write to Reg: 0x09 Val: 0xFFFFFFE1
In clock cycle: 25
Register Write to Reg: 0x0B Val: 0x00000028
In clock cycle: 26
Register Write to Reg: 0x0C Val: 0x00000005
In clock cycle: 27
Register Write to Reg: 0x0B Val: 0xFFFFFFF0
In clock cycle: 28
Register Write to Reg: 0x0E Val: 0x00000001
In clock cycle: 29
Register Write to Reg: 0x0F Val: 0x00000000
In clock cycle: 31
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 32
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 33
```

All these cycles are testing of logical operations and shifts getting expected values two NOPs at the end are for bne to evaluate to decide the next PC value.

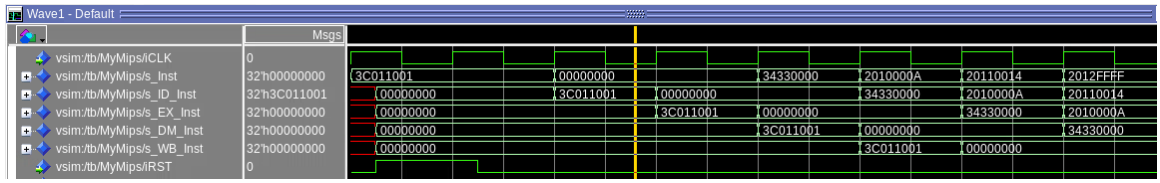
```
        nop
Loops:
        addi $s0, $s0, 10           # $s0 = 20
        # two no opperations for waiting for result of addi for beq
        nop
        nop

        beq $s0, $s1, label2 # If $s0 == $s1, jump to label2 true
        nop
        nop
        j end
        nop
        nop

label1:
        lui $t8, 0x1000      # $t8 = 0x10000000 (upper immediate)
        lw $t9, 0($s3)       # Load word from array (10)
        sw $t4, 4($s3)       # Store 5 in memory at $t3 + 4
        j Loops
        nop
        nop

label2:
        addi $t1, $t1, 1     # Increment $t1 if branch taken
        j Loops
        nop
        nop

end:
        jal JumpAndLink
        nop
        nop
        halt

JumpAndLink:
        addi $t1, $t1, 3     # Increment $t1 by 3 if jal taken
        jr $ra
        nop
        nop
```

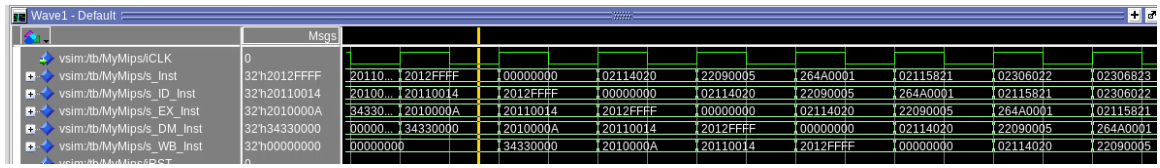
```
In clock cycle: 33
Register Write to Reg: 0x18 Val: 0x10000000
In clock cycle: 34
Register Write to Reg: 0x19 Val: 0x0000000A
In clock cycle: 34
Memory Write to Addr: 0x10010004 Val: 0x00000005
In clock cycle: 37
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 38
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 39
Register Write to Reg: 0x10 Val: 0x00000014
In clock cycle: 40
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 41
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 43
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 44
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 45
Register Write to Reg: 0x09 Val: 0xFFFFFFE2
In clock cycle: 47
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 48
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 49
Register Write to Reg: 0x10 Val: 0x0000001E
In clock cycle: 50
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 51
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 53
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 54
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 56
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 57
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 58
Register Write to Reg: 0x1F Val: 0x004000C4
In clock cycle: 59
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 60
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 61
Register Write to Reg: 0x09 Val: 0xFFFFFFE5
In clock cycle: 63
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 64
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 65
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 66
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 67
Register Write to Reg: 0x00 Val: 0x00000000
Execution is stopping! Clock Cycle: 67
```

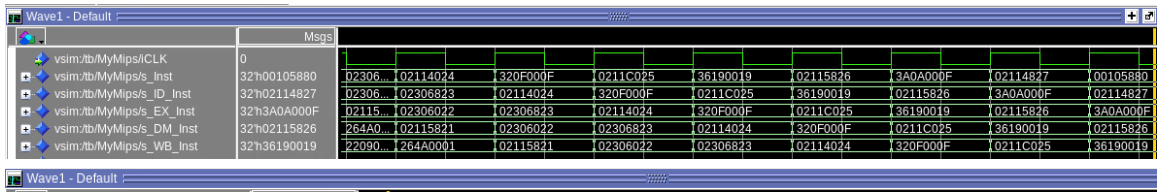
The remainder of execution testing the different jumps types and beq. These all pass and get expected values since jumps and branches cause control hazards this section requires a large number of NOPs to wait for the evaluation of jump location or decide whether a branch needs to be taken or not taken.

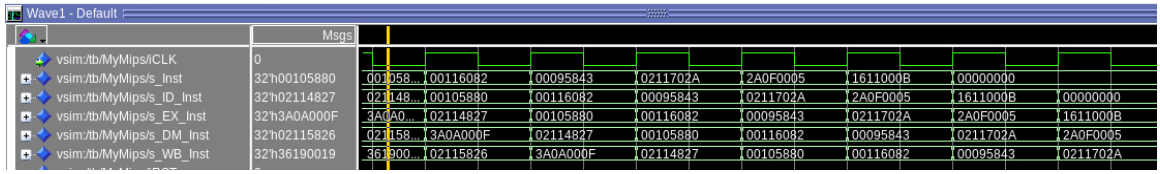
Actual Waveform showing instruction in each pipeline

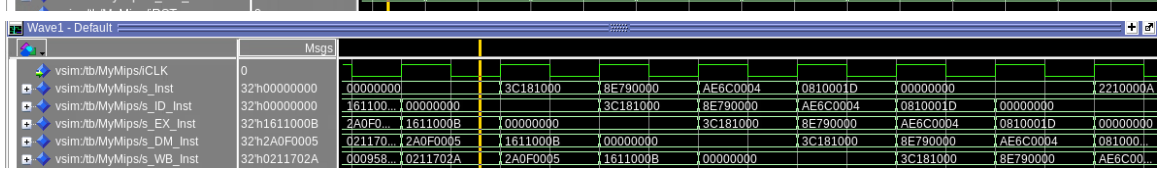
Start

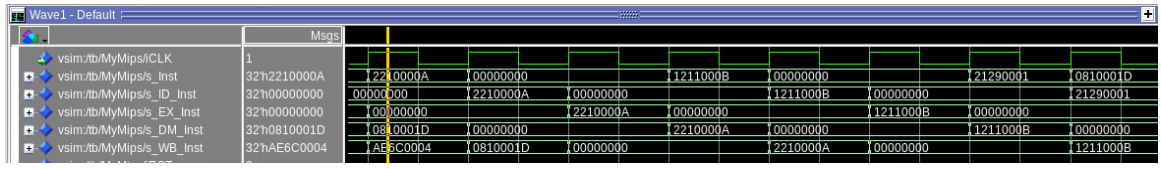

Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
vsim:/tb/MyMips/iRST
0
32'h00000000
32'h3C011001
3C011001 00000000 34330000 2010000A 20110014 2012FFFF
00000000 3C011001 00000000 34330000 2010000A 20110014
00000000 3C011001 00000000 34330000 2010000A
00000000 3C011001 00000000 34330000
00000000 3C011001 00000000
00000000

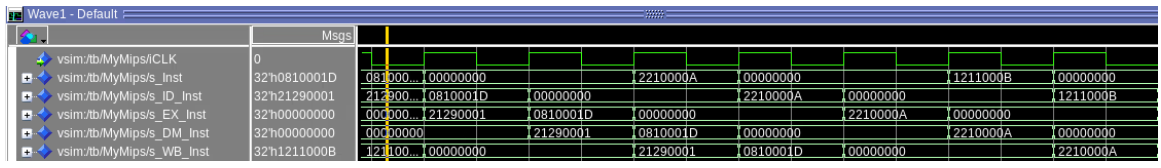

Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
0
32'h2012FFFF
32'h20110014
32'h2010000A
32'h34330000
32'h00000000
20110... 2012FFFF 00000000 02114020 22090005 264A0001 02115821 02306022 02306823
20100... 20110014 2012FFFF 00000000 02114020 22090005 264A0001 02115821 02306022
34330... 2010000A 20110014 2012FFFF 00000000 02114020 22090005 264A0001 02115821
00000... 34330000 2010000A 20110014 2012FFFF 00000000 02114020 22090005 264A0001
00000000 34330000 2010000A 20110014 2012FFFF 00000000 02114020 22090005

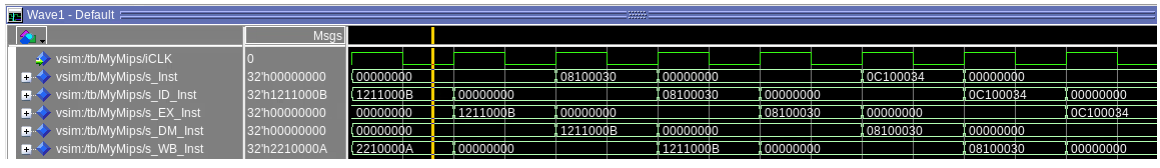

Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
0
32'h00105880
32'h02114827
32'h3A0A000F
32'h02115826
32'h36190019
02306... 02114024 320F000F 0211C025 36190019 02115826 3A0A000F 02114827 00105880
02306... 02306823 02114024 320F000F 0211C025 36190019 02115826 3A0A000F 02114827
02115... 02306022 02306823 02114024 320F000F 0211C025 36190019 02115826 3A0A000F
264A0... 02115821 02306022 02306823 02114024 320F000F 0211C025 36190019 02115826
22090... 264A0001 02115821 02306022 02306823 02114024 320F000F 0211C025 36190019

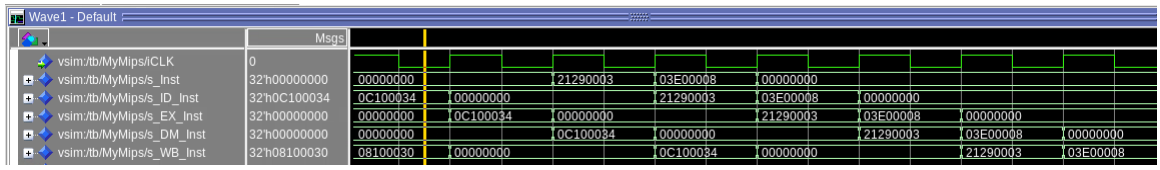

Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
0
32'h00105880
32'h02114827
32'h3A0A000F
32'h02115826
32'h36190019
00105... 00116082 00095843 0211702A 2A0F0005 1611000B 00000000
02114... 00105880 00116082 00095843 0211702A 2A0F0005 1611000B 00000000
3A0A0... 02114827 00105880 00116082 00095843 0211702A 2A0F0005 1611000B
02115... 3A0A000F 02114827 00105880 00116082 00095843 0211702A 2A0F0005
36190... 02115826 3A0A000F 02114827 00105880 00116082 00095843 0211702A

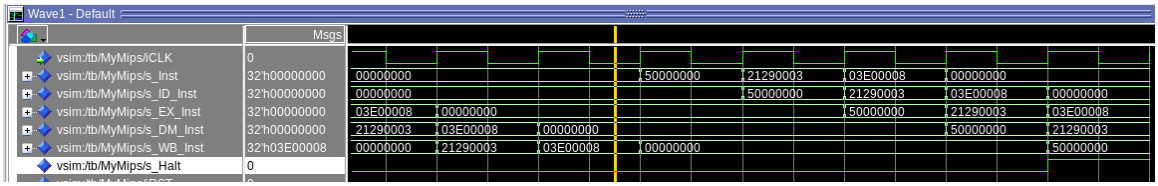

Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
0
32'h00000000
32'h00000000
32'h1611000B
32'h2A0F0005
32'h0211702A
00000000 3C181000 8E790000 AE6C0004 0810001D 00000000 2210000A
16110... 00000000 3C181000 8E790000 AE6C0004 0810001D 00000000
2A0F0... 1611000B 00000000 3C181000 8E790000 AE6C0004 0810001D 00000000
02117... 2A0F0005 1611000B 00000000 3C181000 8E790000 AE6C0004 081000
00095... 0211702A 2A0F0005 1611000B 00000000 3C181000 8E790000 AE6C00


Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
1
32'h2210000A
32'h00000000
32'h00000000
32'h0810001D
32'hAE6C0004
2210000A 00000000 1211000B 00000000 21290001 0810001D
00000000 2210000A 00000000 1211000B 00000000 21290001
00000000 2210000A 00000000 1211000B 00000000
08100... 00000000 2210000A 00000000 1211000B 00000000
AE6C0004 0810001D 00000000 2210000A 00000000 1211000B


Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
0
32'h0810001D
32'h21290001
32'h00000000
32'h00000000
32'h1211000B
081000... 00000000 2210000A 00000000 1211000B 00000000
212900... 0810001D 00000000 2210000A 00000000 1211000B
000000... 21290001 0810001D 00000000 2210000A 00000000
00000000 21290001 0810001D 00000000 2210000A 00000000
121100... 00000000 21290001 0810001D 00000000 2210000A


Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
0
32'h00000000
32'h1211000B
32'h00000000
32'h00000000
32'h2210000A
00000000 08100030 00000000 0C100034 00000000
1211000B 00000000 08100030 00000000 0C100034 00000000
00000000 1211000B 00000000 08100030 00000000 0C100034
00000000 1211000B 00000000 08100030 00000000
2210000A 00000000 1211000B 00000000 08100030 00000000


Wave1 - Default
Msgs
vsim:/tb/MyMips/iCLK
vsim:/tb/MyMips/s_Inst
vsim:/tb/MyMips/s_ID_Inst
vsim:/tb/MyMips/s_EX_Inst
vsim:/tb/MyMips/s_DM_Inst
vsim:/tb/MyMips/s_WB_Inst
0
32'h00000000
32'h0C100034
32'h00000000
32'h00000000
32'h08100030
00000000 21290003 03E00008 00000000
0C100034 00000000 21290003 03E00008 00000000
00000000 0C100034 00000000 21290003 03E00008 00000000
00000000 0C100034 00000000 21290003 03E00008
08100030 00000000 0C100034 00000000 21290003 03E00008



End

[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.

Bubble sort passing.

```
[collalti@linuxvdi-40 proj2_sw]$ ./381_tf.sh test  proj/mips/Proj1_bubblesort.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Proj1_bubblesort.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 469
Processor Cycles: 563
CPI:   1.2
Results in: output/Proj1_bubblesort.s
-------------------------------------------------
```

The same format will be used for proof of correctness as 1.c.i

Two iterations will be shown for the inner loop of bubble sort

```
outer_loop:
        addi $s2, $s2, -1           # Decrement outer loop counter

        # Two no opperations for $t1 to update for beq
        nop
        nop

        beq  $s2, $0, sorted     # If $s2 == 0, array is sorted
        nop
        nop

        # Inner loop to go through the array
        # Moved up add by one
        add  $s4, $s0, $0           # Load size into $s4 for inner loop bounds
        li   $s3, 0                 # $s3 = inner loop counter
        nop
        addi $s4, $s4, -1           # Subtract 1 from size for inner loop
```

```
inner_loop:
        beq  $s3, $s4, outer_loop   # If inner loop counter equals size-1, go to outer loop
        nop
        nop
        # Load array elements for comparison
        sll  $t0, $s3, 2            # $t0 = $s3 * 4 (calculate offset)

        # Two no opperations for $t0 to update for add instruction can't move sll up before beq and can't move add down because
        # add $t1 depends on $t0 and $t1 is a dependenant for $t2
        nop
        nop

        addi $t3, $t0, 4            # Offset to the next element (array[$t1+1])
        add  $t1, $s1, $t0          # $t1 = base address + offset
        nop
        add  $t4, $s1, $t3          # $t4 = base address + offset
        nop
        # Moved lw   $t2, 0($t1) down to break up one nop
        lw   $t2, 0($t1)            # Load array[$t1] into $t2
        lw   $t5, 0($t4)            # Load array[$t4] into $t5

        nop
        nop

        # Compare array[$t1] and array[$t1+1]
        bne  $t2, $t5, check_swap   # If array[$t1] != array[$t1+1], check if swap needed
        nop
        nop
```

```
check_swap:
        slt  $t7, $t2, $t5      # $t7 = array[$t1] < array[$t1+1], check if swap needed

        nop
        nop

        bne  $t7, $0, no_swap       # If $t7 != 0 (array[$t1] < array[$t1+1] is true), no swap
        nop
        nop
        add  $t6, $t2, $zero        # Temporarily store array[$t1] in $t6
        sw   $t5, 0($t1)            # Move array[$t1+1] to array[$t1]
        nop
        sw   $t6, 0($t4)            # Move array[$t1] to array[$t1+1]

no_swap:
        addi $s3, $s3, 1            # Increment inner loop counter
        j    inner_loop             # Jump back to the start of the inner loop
        #Time for jump to evaluate
        nop
        nop
```

Iteration 1:

```
In clock cycle: 22
Register Write to Reg: 0x14 Val: 0x00000005
In clock cycle: 23
Register Write to Reg: 0x13 Val: 0x00000000
In clock cycle: 24
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 25
Register Write to Reg: 0x14 Val: 0x00000004
In clock cycle: 27
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 28
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 29
Register Write to Reg: 0x08 Val: 0x00000000
In clock cycle: 30
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 31
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 32
Register Write to Reg: 0x0B Val: 0x00000004
In clock cycle: 33
Register Write to Reg: 0x09 Val: 0x10010000
In clock cycle: 34
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 35
Register Write to Reg: 0x0C Val: 0x10010004
In clock cycle: 36
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 37
Register Write to Reg: 0x0A Val: 0x00000008
In clock cycle: 38
Register Write to Reg: 0x0D Val: 0x00000004
In clock cycle: 39
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 40
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 42
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 43
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 44
Register Write to Reg: 0x0F Val: 0x00000000
In clock cycle: 45
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 46
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 48
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 49
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 50
Register Write to Reg: 0x0E Val: 0x00000008
In clock cycle: 50
Memory Write to Addr: 0x10010000 Val: 0x00000004
In clock cycle: 52
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 52
Memory Write to Addr: 0x10010004 Val: 0x00000008
In clock cycle: 54
Register Write to Reg: 0x13 Val: 0x00000001
In clock cycle: 56
```

The first couple of cycles set the loop boundaries and go into the loop. All get expected values Cycles 45-49 is four nops because it's back-to-back load word and branch instruction dependent on load word. Cycles 50 and 52 show the first swap in bubble sort since 0x00000008 is greater than 0x00000004, leading to a swap. Cycle 54 represents the end of the first iteration by incrementing register 0x13.

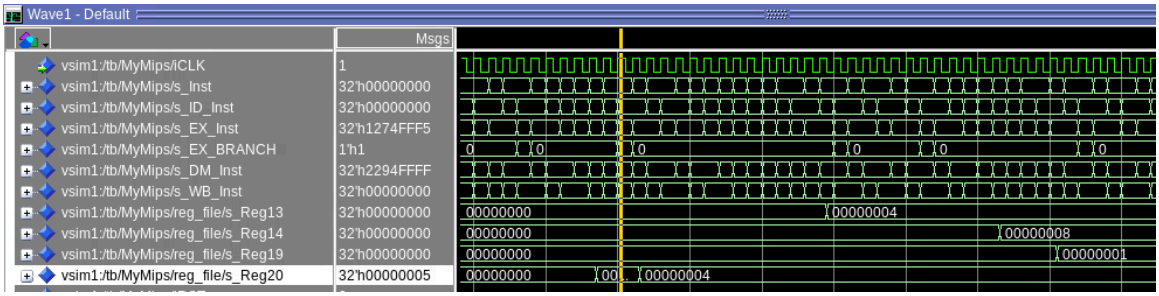
Iteration 2:

```
In clock cycle: 54
Register Write to Reg: 0x13 Val: 0x00000001
In clock cycle: 56
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 57
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 59
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 60
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 61
Register Write to Reg: 0x08 Val: 0x00000004
In clock cycle: 62
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 63
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 64
Register Write to Reg: 0x0B Val: 0x00000008
In clock cycle: 65
Register Write to Reg: 0x09 Val: 0x10010004
In clock cycle: 66
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 67
Register Write to Reg: 0x0C Val: 0x10010008
In clock cycle: 68
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 69
Register Write to Reg: 0x0A Val: 0x00000008
In clock cycle: 70
Register Write to Reg: 0x0D Val: 0xFFFFFFFA
In clock cycle: 71
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 72
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 74
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 75
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 76
Register Write to Reg: 0x0F Val: 0x00000000
In clock cycle: 77
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 78
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 80
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 81
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 82
Register Write to Reg: 0x0E Val: 0x00000008
In clock cycle: 82
Memory Write to Addr: 0x10010004 Val: 0xFFFFFFFA
In clock cycle: 84
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 84
Memory Write to Addr: 0x10010008 Val: 0x00000008
In clock cycle: 86
Register Write to Reg: 0x13 Val: 0x00000002
In clock cycle: 88
```

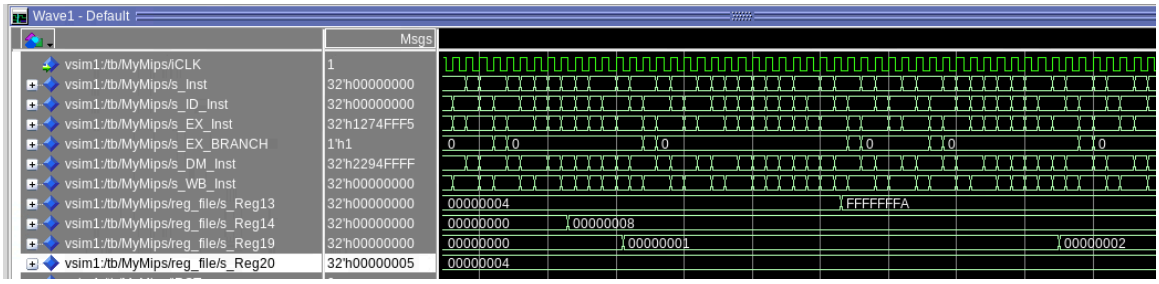
Cycles 82 and 84 show a swap happening when 0x00000008 is greater than 0xFFFFFFFA, which means they swap places. Cycle 86 shows the loop counter 0x13 again increasing, signifying the end of iteration 2 of the inner loop.

Questa Waveform for the above
Reg13 and Reg 14 used for swaps
Reg 19 is the inner loop counter
Reg 20 is the outer loop counter

Setup, start, and swap of first iteration swap register hold 0x00000004 and 0x00000008



End of iteration 1 start and end of iteration 2 with swap registers holding 0xFFFFFFFA and 0x00000008



3 examples of where you did not have to use the maximum number of NOPs

1) Through reordering only two nops are need for this segment ori is dependent on lui. Lui can't be moved up because it's the first instruction in the program and ori can't be moved down since the lw's (not lw $s0, size) all depend on the ori value so ori can't be moved down. In the same vain lw $s0, size I was able to move up before the nop so I wouldn't need to have a nop before the add.

```
main:
        # Load the size of the array into $s0
        lui $1,4097
        lw   $s0, size # Load the size of the array into $s0
        nop
        ori $s1,$1,0
        add  $s2, $s0, $zero        # Copy size into $s2 for outer loop (outer loop counter)
        lw   $t0, 0($s1)
        lw   $t1, 4($s1)
        lw   $t2, 8($s1)
        lw   $t3, 12($s1)
        lw   $t4, 16($s1)
```

2) By having the order below, only one nop is needed between addi and add $t4 instructions since the add $t1 is above the nop, no nop is needed for the first lw. The second lw needs one nop because of the dependency on add $t4, which can't be moved up because of the addi dependency. The lw can't be moved down because it also has a dependency for bne

```
addi $t3, $t0, 4            # Offset to the next element (array[$t1+1])
add  $t1, $s1, $t0          # $t1 = base address + offset
nop
add  $t4, $s1, $t3          # $t4 = base address + offset
nop
# Moved lw   $t2, 0($t1) down to break up one nop
lw   $t2, 0($t1)            # Load array[$t1] into $t2
lw   $t5, 0($t4)            # Load array[$t4] into $t5

nop
nop

# Compare array[$t1] and array[$t1+1]
bne  $t2, $t5, check_swap   # If array[$t1] != array[$t1+1], check if swap needed
```

3) Control

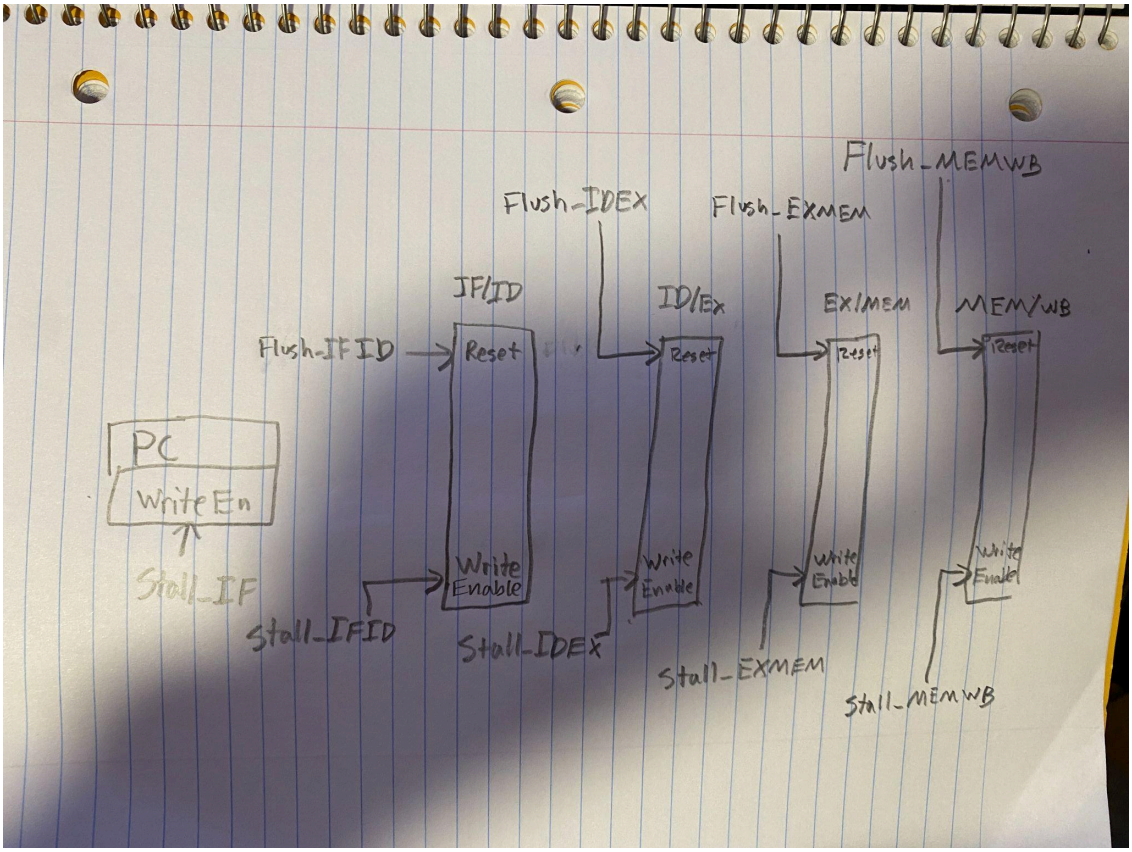
```
outer_loop:
        addi $s2, $s2, -1           # Decrement outer loop counter

        # Two no opperations for $t1 to update for beq
        nop
        nop

        beq  $s2, $0, sorted     # If $s2 == 0, array is sorted
        nop
        nop
```

Controls can't be optimized well because they will always need 2 NOP's after the instruction to decide if the branch taken or not taken. Above is an example that shows the 2 NOP's that given time to the beq to see if the array is sorted or not.

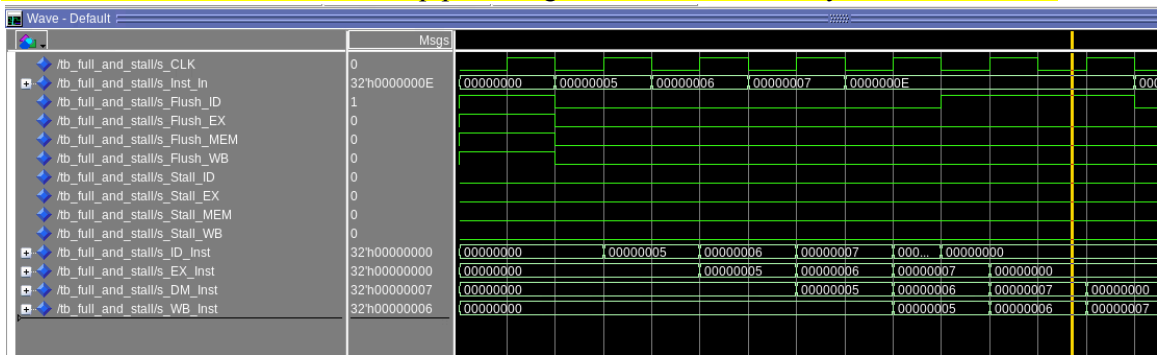
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

FMax = 55.44
Critical path: ALU stage
Specifically, our data moves out of the IF/EX register block and then into the ALU where the output is written into the EX/DM registers.

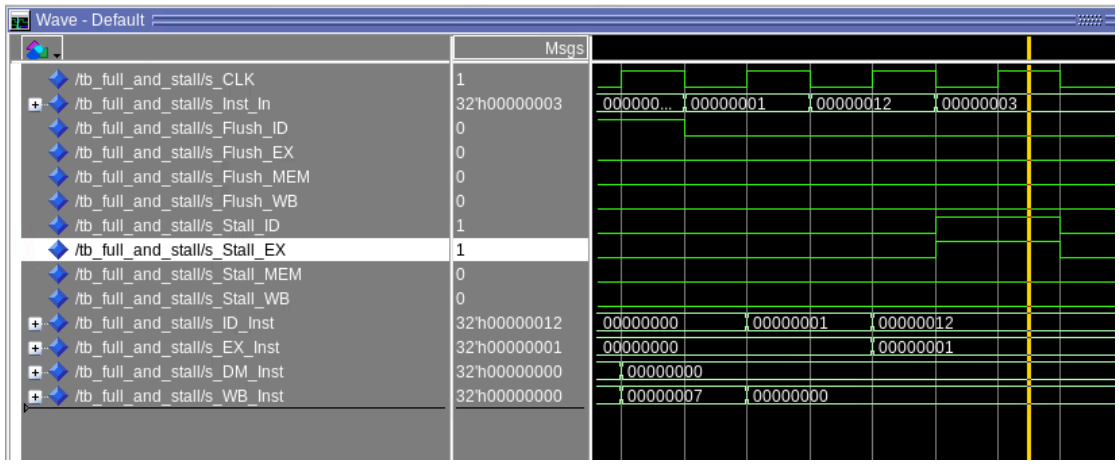
[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.
Stalling can be implemented using the write enable of a register. If a register has a write enable of 1 it acts as normal, but if write enable is 0, then the contents of the register will not change. If IF/ID and ID/EX are stalled then The current pc isn't changed, and ID/EX isn't updated, this would represent a standard stall. Flushing can be done by resetting the register and clearing the data inside, which is what flushing is meant to do. In the finished processor, the stalling and flushing signals are done on the falling edge to avoid conflicts and clarity of data values in each clock cycle



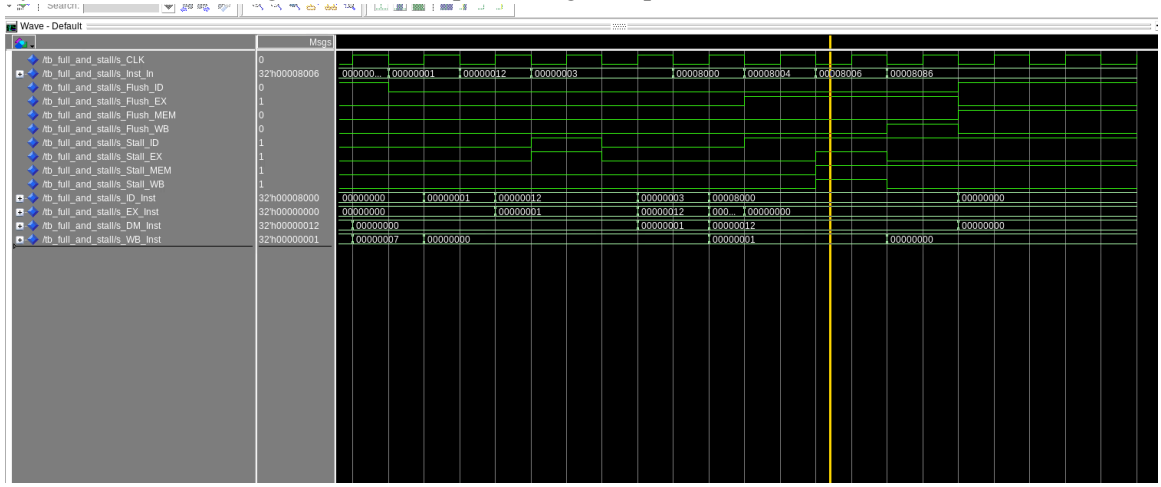
[2.a.iii] Create a testbench that instantiates all four of the registers in a single design. Show that values that are stored in the initial IF/ID register are available as expected four cycles later, and that new values can be inserted into the pipeline every single cycle. Most importantly, this testbench should also test that each pipeline register can be individually stalled or flushed.



First cycle above: all registers are flushed, setting all register instructions to 0x00000000. Cycles 2-5 fill pipeline with 0x00000005, 0x00000006, 0x00000007, and 0x00000008 to show standard operation. Cycles 6 and 7 show back-to-back ID flushing this sets ID register to 0x00000000 and then propagates that in cycle 7.



The first cycle shows seven from the previous image. Cycles 8 and 9 starts refilling pipeline, then Cycle 10 shows stalling of ID and EX preventing the update



The first third of the cycles are from the previous screenshot. Then, the remainder is testing a combination of flushes and stalling.

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

Signal: alu output

add
addu
and
nor
xor
or
slt
sll
srl
sra
sub
subu
addi
addiu

andiu
lui
lw
xori
ori
slti
sw
jal

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

Signals: ALU input A (rs), ALU input B(rt)
add
addu
and
nor
xor
or
slt
sll
srl
sra
sub
subu
addi
addiu
andiu
lui
lw
xori
ori
slti
sw
jal

[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.

Forward ALU output from data memory stage if…

- Reg write is enabled at data memory stage and
- The destination register at the data memory stage is the same as the source register (rs or rt) in the execute stage and
- Destination register in data memory stage is not zero

Forward ALU output from write back stage if…

- Reg write is enabled at write back stage and
- The destination register at the write back stage is the same as the source register (rs or rt) in the execute stage and
- Data will not be forwarded from the memory stage and

- Destination register in write back stage is not zero

The load-use hazard is an additional hazard that will require a stall, as detected by the hazard detection unit. This occurs when mem read in IF/EX stage is enabled and rt at ID/EX is the same as rs or rt in the IF/ID stage.

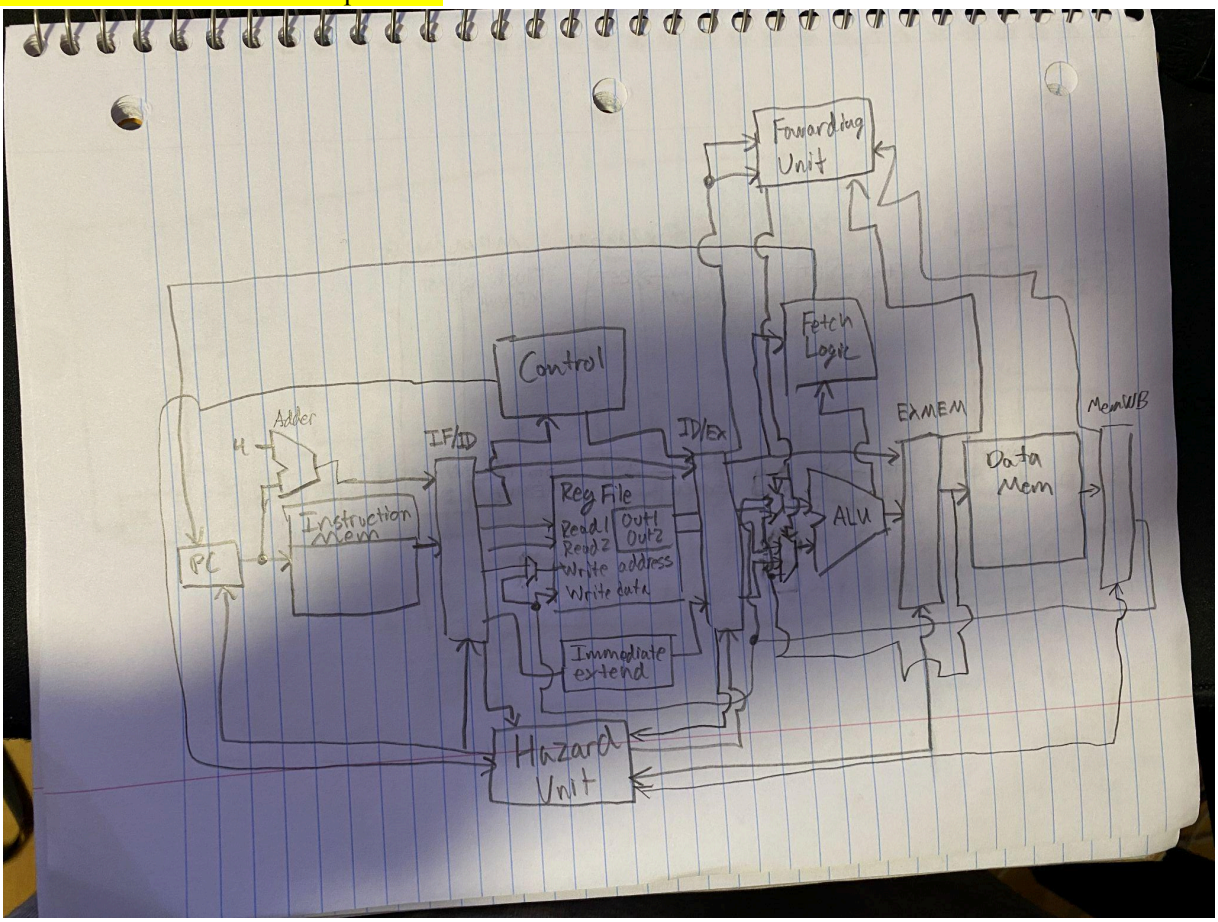
[2.b.iv] global list of the datapath values and control signals that are required during each pipeline stage

Fetch	Decode	Execute	DataMem	WriteBack
zero (ex) IMemAddr regOutRs (ex) JR (ex) Inst (ex) JUMP_EN (ex) imm_extended (ex) BRANCH (ex) BRANCH_TYPE (ex) NextInstAddr Flush_IFID Flush_IFID_IN Stall_IF Stall_IFID	Inst IMemAddr ALUOP DST_REG Reg_WR Mem_WR ALUSrc EXTENSION MEM_TO_REG BRANCH BRANCH_TYPE JUMP_EN JR JAL Halt regOutRs regOutRt imm_extended Flush_IDEX Flush_IDEX_IN Stall_IDEX	IMemAddr Inst ALUOP DST_REG ALUSrc MEM_TO_REG Reg_WR Mem_WR BRANCH BRANCH_TYPE JUMP_EN JR JAL Halt regOutRs regOutRt zero imm_extended Ovfl select_rs (forwarding value) select_rt (forwarding value) Flush_EXMEM Flush_EXMEM_IN Stall_EXMEM	IMemAddr Reg_WR Mem_WR MEM_TO_REG DST_REG JAL Halt ALURes DMemOut Inst regOutRs regOutRt Ovfl Flush_MEMWB Flush_MEMWB_IN Stall_MEMWB	MEM_TO_REG Reg_WR Halt DMemOut DST_REG Inst jal_mux2_out JAL Ovfl

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.
Jump, jump and link, jump register, branch equal, and branch not equal are all discovered in the decode stage. Stalls also can cause non-sequential PC updates by holding/ not updating the PC value when the PC is stalled.
[2.c.ii] For these instructions, list which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each of these instructions is in.
Jump and JAL - Since Jump and Jal are known in the decode stage, flush IF/ID and stall pc on the falling edge of the clock cycle. Flush IF/ID, Stall_IF

Jr - known in the decode stage if RS is the destination register in any stages ahead of Jr instruction stall until RS value is updated. Jr, jump location is known in Execute stage Flush IF/ID, Stall_IF.
Branches - known in the decode stage if RS or RT is the destination register in any stages ahead of the Branch instruction stall until the RS or RT value is updated. Then, if the branch instruction condition is met in the decode stage, we can Flush IF/ID, Stall_IF, getting rid of the two following instructions of incorrect prediction
Load Words- If LW destination is needed in earlier stages, stall previous stages to give time for lw to evaluate.

[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.



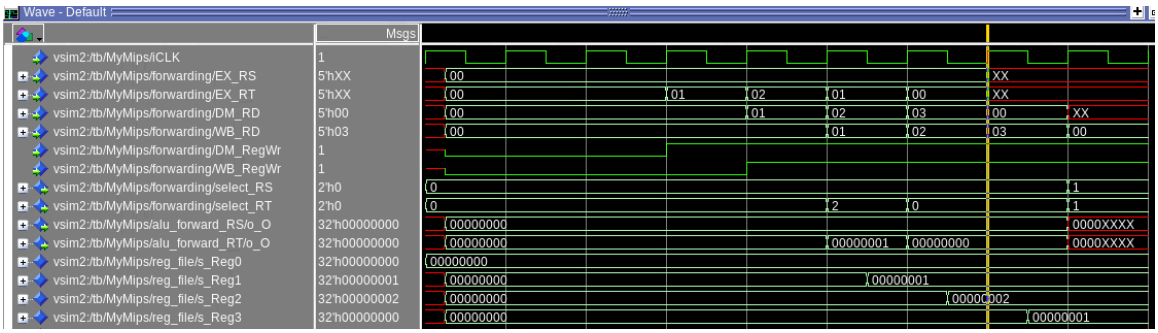
[2.e – i, ii] In your writeup, show the Modelsim output for each of the following tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.
Four Cases for Forwarding

Forwarding							
Test 1: MEM/WB.RegisterRd = ID/EX.RegisterRt							
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Addi $1, $0, 1	IF	ID	EX	MEM	WB		
Addi $2, $0, 2		IF	ID	EX	MEM	WB	
Add $3, $0, $1			IF	ID	EX	MEM	WB
Test 2: MEM/WB.RegisterRd = ID/EX.RegisterRs							
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
Addi $1, $0, 1	IF	ID	EX	MEM	WB		
Addi $2, $0, 2		IF	ID	EX	MEM	WB	
Add $3, $1, $0			IF	ID	EX	MEM	WB
Test 3: EX/MEM.RegisterRd = ID/EX.RegisterRt							
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	
Addi $1, $0, 1	IF	ID	EX	MEM	WB		
Add $2, $0, $1		IF	ID	EX	MEM	WB	
Test 4: EX/MEM.RegisterRd = ID/EX.RegisterRs							
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	
Addi $1, $0, 1	IF	ID	EX	MEM	WB		
Add $2, $1, $0		IF	ID	EX	MEM	WB	

Test 1:

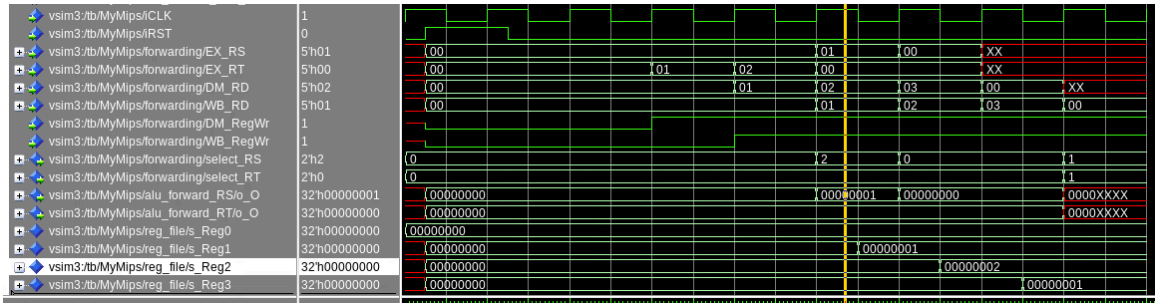
```
[collalti@linuxvdi-40 proj2_hw]$ ./381_tf.sh test  proj/mips/Test1_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test1_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 4
Processor Cycles: 8
CPI:   2.0
Results in: output/Test1_Collalti.s
```



alu_forward_RT/o_O is the forwarding signal and forwards Addi $1, $0, 1 result to Add $3, $0, $1.

Test 2:

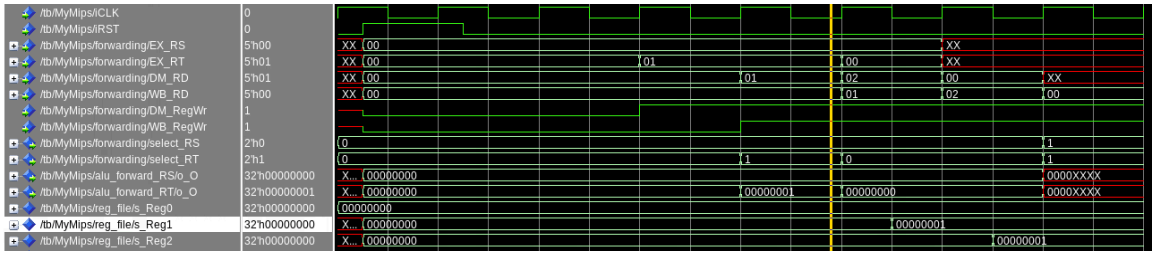
```
[collalti@linuxvdi-40 proj2_hw]$ ./381_tf.sh test  proj/mips/Test2_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test2_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 4
Processor Cycles: 8
CPI:   2.0
Results in: output/Test2_Collalti.s
```



alu_forward_RS/o_O is the forwarding signal and forwards Addi $1, $0, 1 result to Add $3, $1, $0.

Test 3:

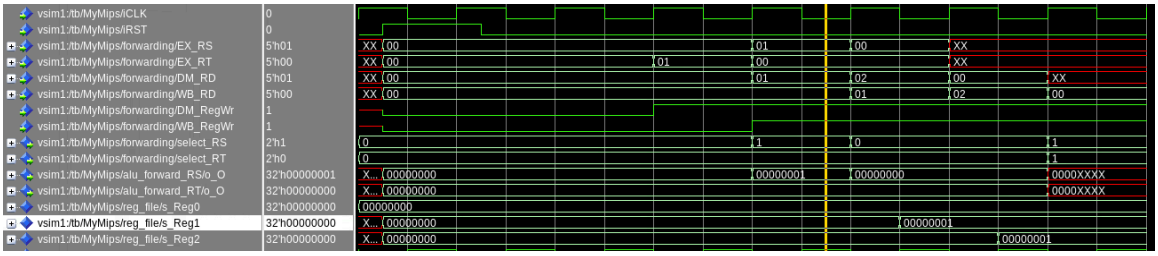
```
[collalti@linuxvdi-28 proj2_hw]$ ./381_tf.sh test  proj/mips/Test3_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test3_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 3
Processor Cycles: 7
CPI:  2.33
Results in: output/Test3_Collalti.s
```



alu_forward_RT/o_O is the forwarding signal and forwards Addi $1, $0, 1 result to Add $2, $0, $1.

Test 4:

```
------------------------------------------------
[collalti@linuxvdi-28 proj2_hw]$ ./381_tf.sh test  proj/mips/Test4_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test4_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 3
Processor Cycles: 7
CPI:  2.33
Results in: output/Test4_Collalti.s
```

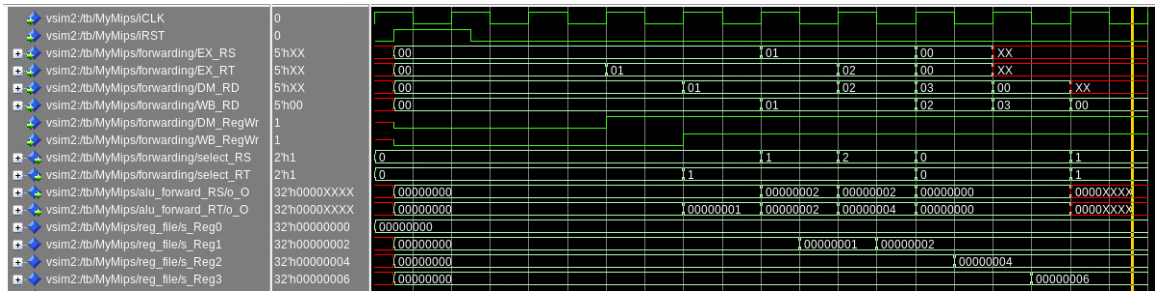


alu_forward_RS/o_O is the forwarding signal and forwards Addi $1, $0, 1 result to Add $2, $1, $0.

Test 5: Multiple hazards

Test 5: Multiple												
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7					
Addi $1, $0, 1	IF	ID	EX	MEM	WB							
Addi $1, $0, 2		IF	ID	EX	MEM	WB						
Add $2, $1, $1			IF	ID	EX	MEM	WB		Should forward most recent Addi $1,$0,2. Forwards both RS and RT			
Add $3, $1, $2				IF	ID	EX	MEM	WB	Depends on instructions in different stages.			

```
------------------------------------------------
[collalti@linuxvdi-28 proj2_hw]$ ./381_tf.sh test  proj/mips/Test5_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test5_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 5
Processor Cycles: 9
CPI:   1.8
Results in: output/Test5_Collalti.s
```



The second instruction output is forwarded to the third instruction because it's the more recent update to $1 compared to the first instruction. Then, the fourth instruction tests the forwarding of Rs from WB and the forwarding of Rt from MEM in the same instruction.

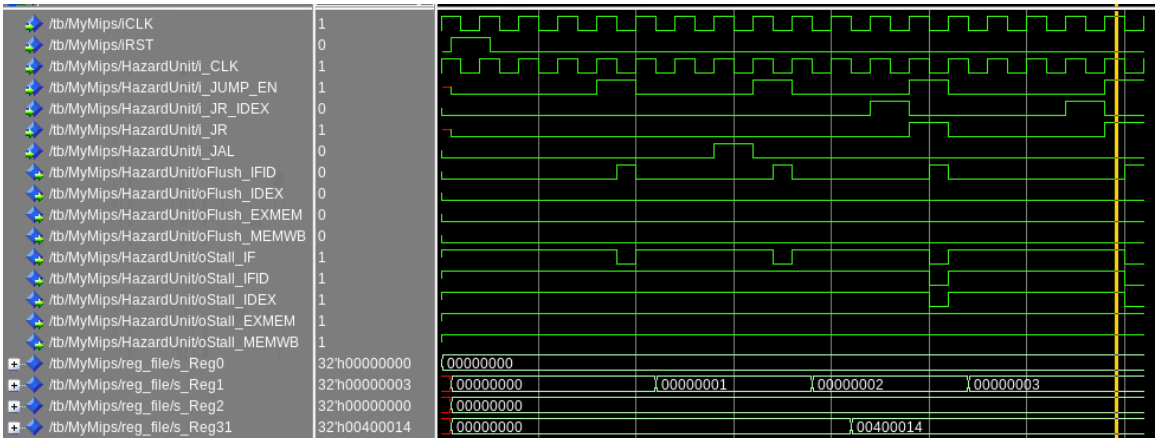
[2.e.ii] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

Test 6: Jumps (Jump, Jal, Jr)

```
.data
.text
.globl main
main:
    # Start
    addi $1, $0, 1
    j Jump
    addi $2, $0, 1

Jump:
addi $1,$1,1
jal Link
halt

Link:
        addi $1, $1,1
        jr $ra #Important peice
        addi $2,$0,1
```

Test 6: Jumps																	
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11	Cycle 12	Cycle 13	Cycle 14	Cycle 15	Cycle 16	Cycle 17
Addi $1, $0, 1	IF	ID	EX	MEM	WB												
j Jump		IF	ID	EX	MEM	WB											
Addi $2, $0, 1			IF	ID	EX	MEM	WB										
Jump				IF stall (NOP)													
Addi $1, $1, 1					IF	ID	EX	MEM	WB								
jal Link						IF	ID	EX	MEM	WB							
Addi $2, $0, 0							IF	Stall	ID	EX	MEM		IF	Stall	ID	EX	MEM
halt							IF Stall (NOP)										
Link:								IF	Stall	ID	EX	MEM					
Addi $1, $1, 1									IF	Stall	ID	EX	MEM				
jr $ra										IF	Stall	ID	EX	MEM			
Addi $2, $0, 1											IF	Stall	ID	EX	MEM	FLUSHED	
												IF Stall (NOP)					

```
[collalti@linuxvdi-06 proj2_hw]$ ./381_tf.sh test  proj/mips/Test6_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test6_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 7
Processor Cycles: 17
CPI:  2.43
Results in: output/Test6_Collalti.s
```



From the waveform, Reg 1 shows when jumps happen, and Reg 2 should stay at zero because all instructions involving Reg2 should be flushed since they come after a jump instruction. Each time a jump-type instruction is in the hazard unit, it outputs the correct stalls and flush values to the pipeline registers.

Branch Testing

Test 7: BEQ false								
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	
Addi $1, $0, 1	IF	ID	EX	MEM	WB			
beq $1, $0, Exit		IF	ID	EX	MEM	WB		FALSE
Addi $2, $0, 1			IF	ID	EX	MEM	WB	Runs
Exit								

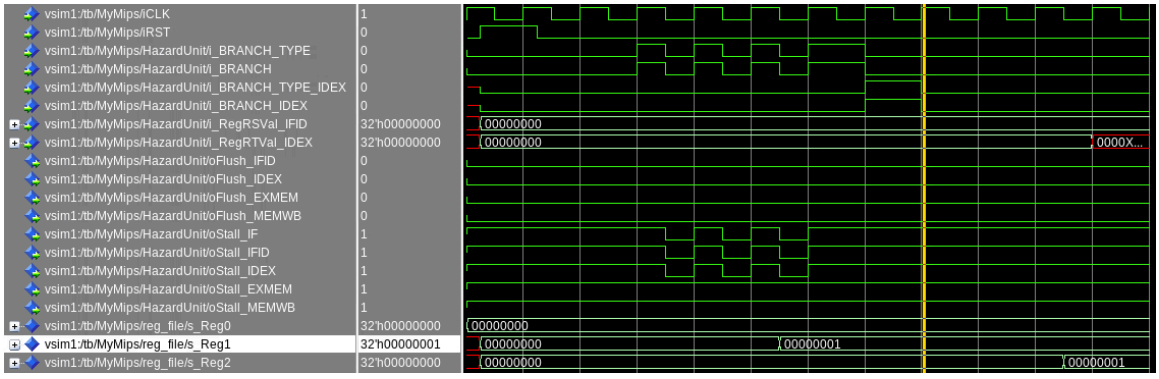
Test 8: BEQ true								
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	
Addi $1, $0, 0	IF	ID	EX	MEM	WB			
beq $1, $0, Exit		IF	ID	EX	MEM	WB		TRUE
Addi $2, $0, 1			IF	ID	EX	MEM	WB	FLUSHED
Exit								

Test 9: BNE false								
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	
Addi $1, $0, 0	IF	ID	EX	MEM	WB			
bne $1, $0, Exit		IF	ID	EX	MEM	WB		FALSE
Addi $2, $0, 1			IF	ID	EX	MEM	WB	Runs
Exit								

Test 10: BNE true								
Instruction IN	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	
Addi $1, $0, 1	IF	ID	EX	MEM	WB			
bne $1, $0, Exit		IF	ID	EX	MEM	WB		TRUE
Addi $2, $0, 1			IF	ID	EX	MEM	WB	FLUSHED
Exit								

Test 7: BEQ false

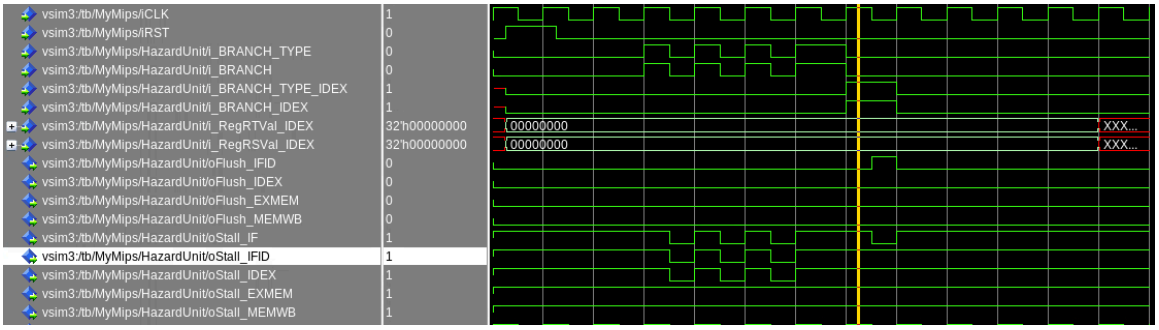
```
[collalti@linuxvdi-06 proj2_hw]$ ./381_tf.sh test  proj/mips/Test7_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test7_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 4
Processor Cycles: 11
CPI:  2.75
Results in: output/Test7_Collalti.s
```



Since the branch condition will be false, then the reg 2 will be 1

Test 8: BEQ false

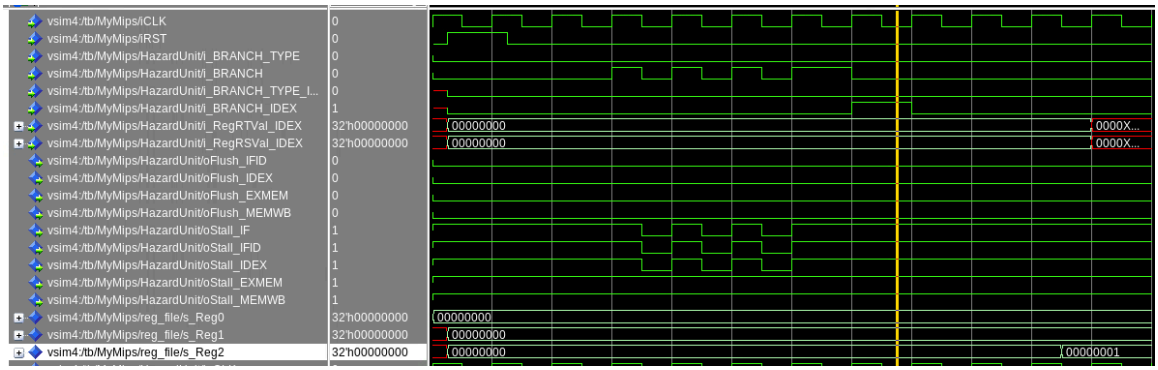
```
[collalti@linuxvdi-06 proj2_hw]$ ./381_tf.sh test  proj/mips/Test8_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test8_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 3
Processor Cycles: 12
CPI:   4.0
Results in: output/Test8_Collalti.s
```



Reg 1 is equal to Reg 0 so the branch condition is met, so the add to reg 2 is flushed, and all registers are 0

Test 9: BNE false

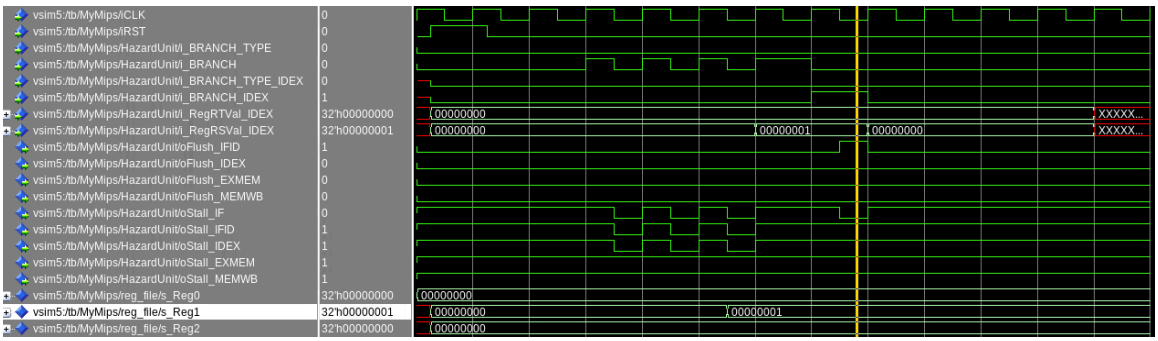
```
------------------------------------------------
[collalti@linuxvdi-06 proj2_hw]$ ./381_tf.sh test  proj/mips/Test9_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test9_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 4
Processor Cycles: 11
CPI:  2.75
Results in: output/Test9_Collalti.s
------------------------------------------------
```



The branch condition isn't met, so Reg 2 gets added to since the instruction isn't flushed

Test 10: BNE true

```
------------------------------------------------
[collalti@linuxvdi-06 proj2_hw]$ ./381_tf.sh test  proj/mips/Test10_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test10_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 3
Processor Cycles: 12
CPI:   4.0
Results in: output/Test10_Collalti.s
```



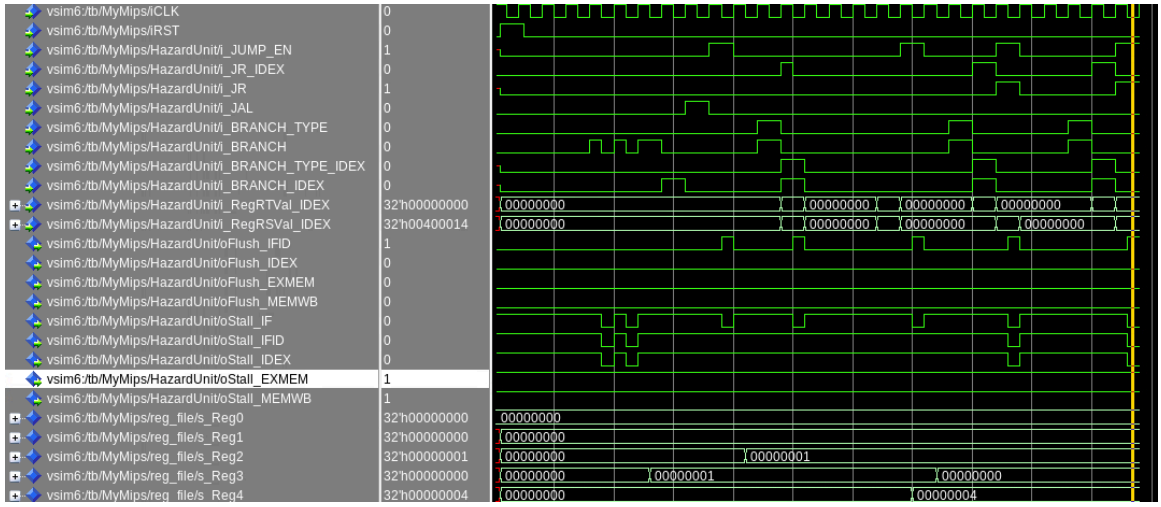
BNE condition is met so reg 2 add instruction is flushed.

Test 11: Mixed

```
.data
.text
.globl main
main:
    # Start
    addi $1, $0, 0
    addi $3, $0, 1
    bne $1, $0, Exit
    addi $2, $0, 1
    jal Out
    halt
Out:
    beq $3, $2, Loop
    jr $31

Loop:
   addi $4, $4, 4
   addi $3, $3, -1
   j Out
Exit:
    # Exit program
    halt
```

```
[collalti@linuxvdi-06 proj2_hw]$ ./381_tf.sh test  proj/mips/Test11_Collalti.s
Using VDI Python Environment
Testing
All VHDL src files compiled successfully
Testing file: proj/mips/Test11_Collalti.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: pass
Mars Instructions: 12
Processor Cycles: 26
CPI:  2.17
Results in: output/Test11_Collalti.s
```

due to 26 cycles, I opted not to do a spreadsheet and instead will explain. Addi's starts by setting initial values and is forwarded to bne which will be false, continuing to jal. Jal will jump to Out where beq will be true flushing jr instruction and going into loop at the end of loop jump Out returns back to the bne which will now be false so jr returns back to the jal instruction then runs the halt instruction.



The register value for reg 1 will always be 0 reg 2 will 1, reg 3 will start with 1, but once the loop is entered reg 3 decreases by 1. Also in loop reg 4 is increased by 4. The all jumps were used in this test as well as both branch types where one was true and the other was false.

[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 = 48.94mhz
Critical path: ALU stage
Specifically, the critical path occurs at an instance of data forwarding from the memory stage. That is, the value read from memory is forwarded to the execute stage and then moves through that entire stage.