CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Kevin Dickey

Brock Dykhuis

Owen Parker

Project Teams Group #: Proj2_6_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.

NOTE: for this processor I followed the diagram on figure 4.65 of the lab description pdf, which had a weird naming structure for the pipeline register, so it's a little goofy.

(IF/ID -> ID/EX -> MEM/WB -> EX/MEM)

That PDF should probably be updated 😀

Needed for the Fetch stage:
branch address
jump address
R[rs]

Needed for the Decode stage:
Instruction
Next PC address

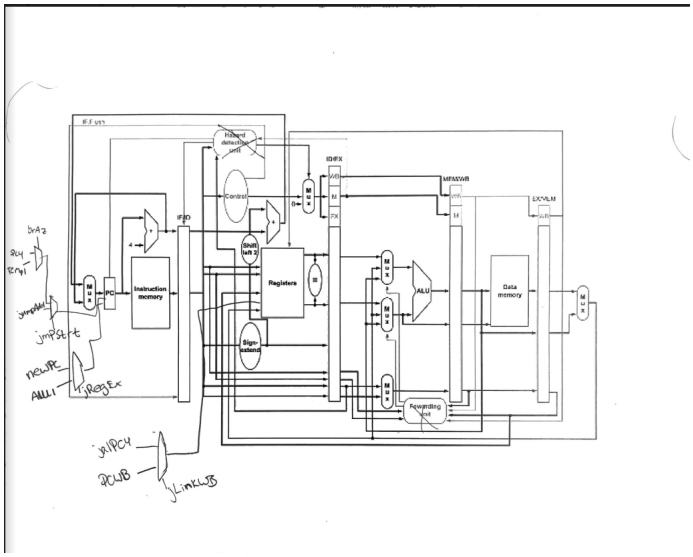
Needed for the Execute stage:
MemtoReg
RegWrite
memWrite
memRead
branch
jump
AluSrc
ctl
halt
Both register read data values

PC value
SignExtended value
instruction
rs
rt
destination register

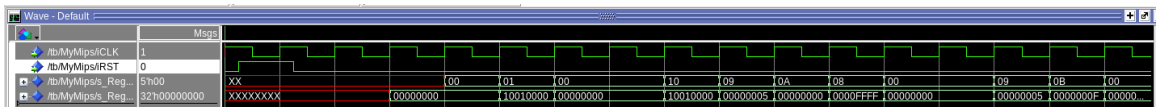
Needed for the Memory stage:
MemtoReg
RegWrite
memWrite
memRead
halt
jlink
PC value
alu output
second alu input
destination register

Needed for the Writeback stage:
MemtoReg
RegWrite
halt
jlink
PC value
ALU output
data memory output
destination register

[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.



This is the waveform for our "Proj2_base_tests.s" file. This waveform's values match the result from mars.

[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.

Test: bubble sort

s_RegWrAddrD- signal for destination register of the instruction

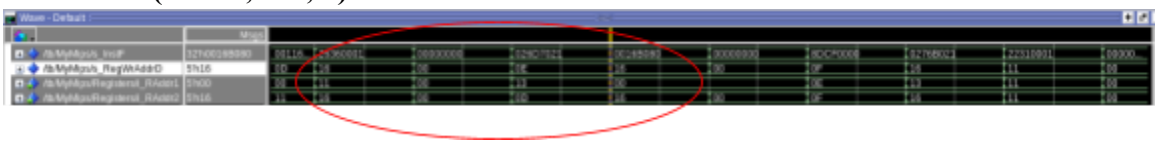
i_RAddr1- signal for the rs register used in the instruction

i_RAddr2- signal for the rt register used in the instruction

s_InstF - instruction during the decode stage(when the other values are being compared)

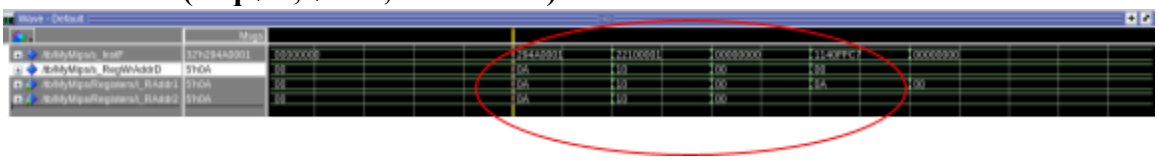
Dataflow:

Line 86-89 (sll $s6, $s6, 2)



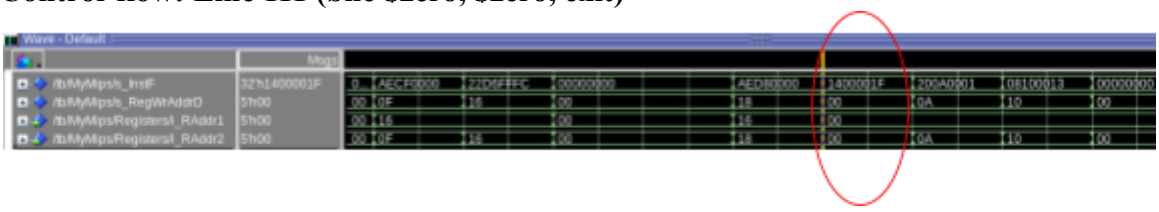
This was a data-flow example because the rightmost instruction used the rt value that gets set in the leftmost instruction. In this instance we only had to add one nop because there was another instruction in between which let the value of the sixteenth register be set correctly before it was used.

Line 119-122 (beq $t2, $zero, outerCond)



This was a data-flow example because the rightmost instruction used the rs value that gets set in the leftmost instruction. In this instance we only had to add one nop because there was another instruction in between which let the value of the tenth register be set correctly before it was used.

Control-flow: Line 111 (bne $zero, $zero, exit)



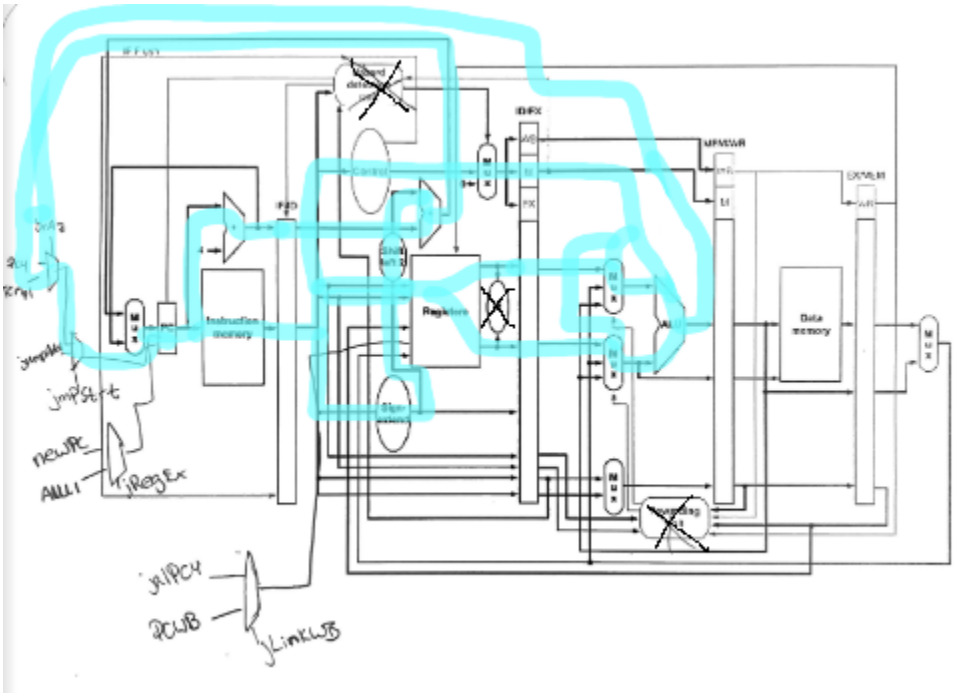
This was a Control-flow example because the instruction is a branch instruction which will cause a control hazard if the branch is taken since the next instructions will also run before the PC is updated. For this branch no nops are needed because this branch never changes the PC.

[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).

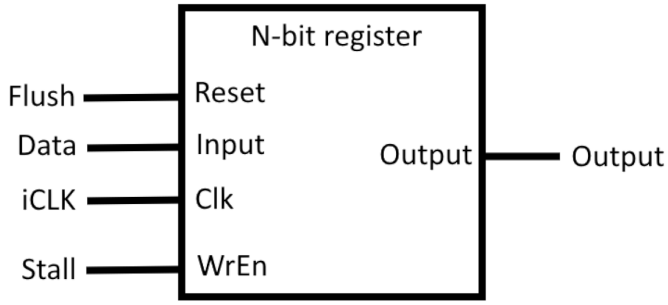
54.78 MHz

Critical Path: branch instruction

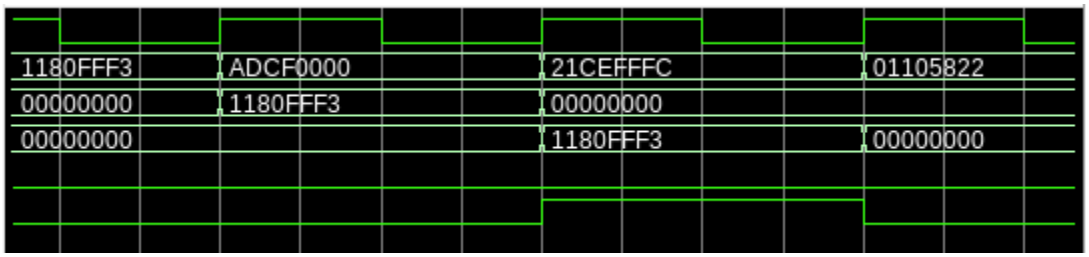
(path drawn below, zero flag (brAz signal for mux on far left) and branch address calculated in execute stage)



[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.



This shows instructions going through the fetch, decode, and execute stages. This also displays a flush(bottom signal) which happens to the decode stage which resets the instruction.

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

add, **addi**, **addu**, **addiu**, **sub**, **subu**, **xor**, **xori**, **nor**, **and**, **andi**, **or**, **ori**, **sw**, **slt**, **slti**, **sll**, **srl**, **sra**, **lui**, **lw, jal**,

I-type and R-type: values done calculating at the wire: s_ALU

sw: s_ALU (mem address to store at), s_B (value to store)

lw: s_DMemOut (data to store), s_B/s_RegWrAddr (location to store at)

jal: s_RegWrAddrD (which will be $ra, reg 31)

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

add, addi, addu, addiu, and, andi, lw, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sw, sub, subu: produce and consume from register file and/or data memory

(lui and jal don't consume anything which isn't in the register file or the data memory (though they do produce). beq, bne, j, and jr consume values but don't produce)

I-type and R-type → s_A and s_B (consumes), s_ALU (produces)

lw and sw → s_A and s_B (consumes), s_DMemOut (produces for sw)

[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.

Can be Forwarded:

- **ALU operand from prior ALU result (ALU output forwarded to inputs of ALU for next instruc)**
- **ALU operand from data memory (DMEM output forwarded to inputs of ALU for next instruc)**

Requires Stall:

- **RAW – instruction following something that accesses the memory (like lw) will need a stall**
- **Branches will need to stall and flush instructions in earlier stages of the pipeline <u>when they are being taken</u>**
- **Jumps will need to stall and flush**

basic arithmetic instructions can have their ALU output forwarded to the instructions behind them.
Memory reads will be stalled

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

IF-ID:

- **s_Instr, s_pc4, s_pc**

ID-EX:

- **s_Instr, s_pc4, s_MemtoRegD, s_RegWrD, s_DMemWrD, s_MemReadD, s_BranchD, s_BranchEx, s_zeroD, s_JumpD, s_ALUsrcD, s_ctl, s_jLink, s_jReg, s_HaltD, s_SoZexrend, s_ALU1D, s_ALU2D, s_PC, s_signExtendedD, s_instF, s_RegWrAddrD**

EX-MEM:

- **s_Instr, s_pc4, s_MemtoRegM, s_RegWriteM, s_HaltM, s_jLinkM, s_OverflowM, s_PCM, s_DMemAddr, s_DMemOut, s_InstM**

MEM-WB:

- **s_Instr, s_pc4, s_MemtoRegEx, s_RegWriteEx, s_memWriteEx, s_memReadEx, s_HaltEx, s_jLinkEx, s_OverflowEx, s_PCEX, s_ALU, s_forward2, s_rdEx**

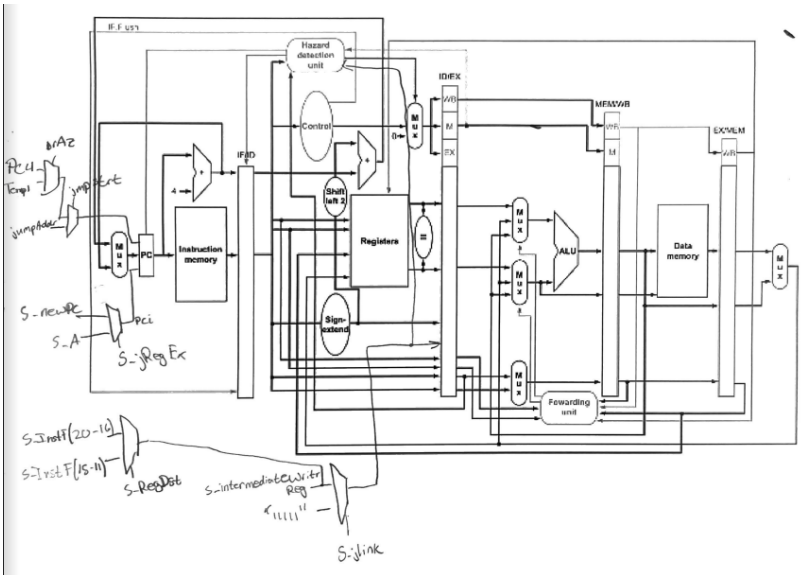
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

Bne, beq, j, jal, jr

[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.

the processor checks if a branch or jump is taken, if a branch is taken the IF/ID register is both stalled and flushed, if a jump is taken the IF/ID register is just flushed

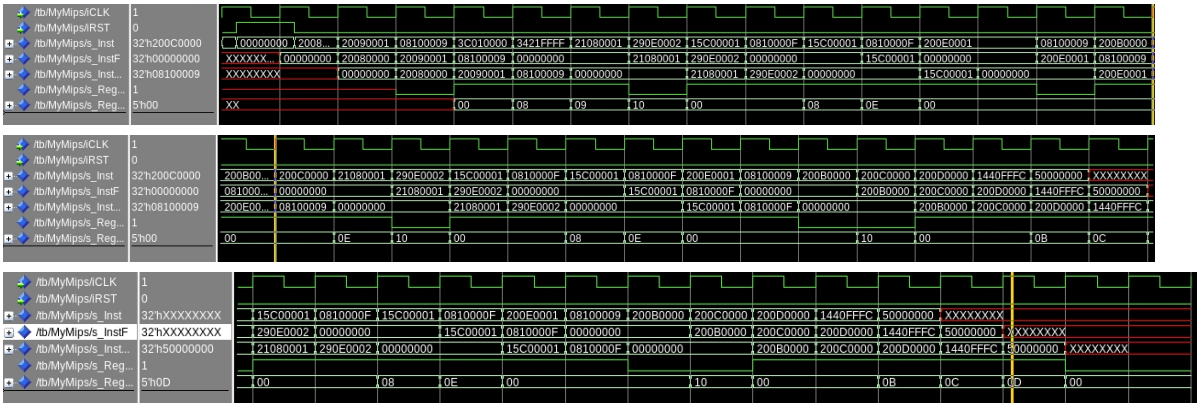
[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.



[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table. (include modelsim output and tiny discussion of result correctness)

Create a set of assembly programs that exhaustively tests the data forwarding and hazard detection capabilities of your pipeline. Minimally you should create one assembly program for each of your hazard detection and forwarding cases

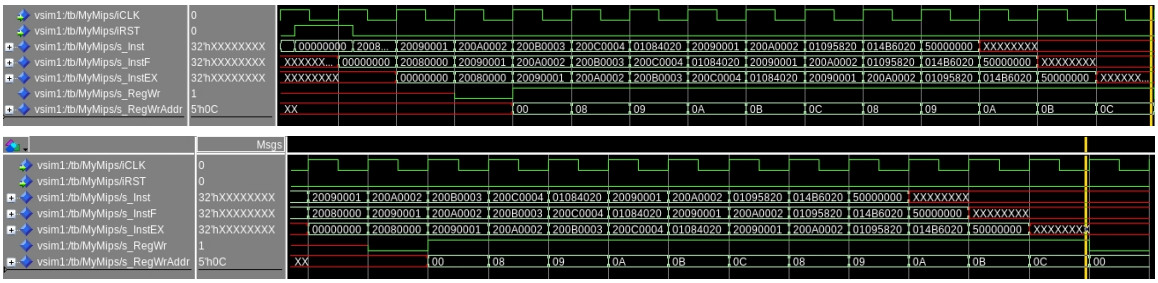
(forwarding) just use base_tests and bubblesort and cover the data forwarding and hazard detection capabilities of the pipeline



```
3  main:
4          addi $t0, $zero, 0x0 # t0 = 0
5          addi $t1, $zero, 0x1 # t1 = 1
6    j loop
7
8    skip:
9    addi $t0, $zero, 0xFFFF # should be skipped
10   addi $t1, $zero, 0xFFFF # should be skipped
11
12   loop:
13   addi $t0, $t0, 0x1
14   slti $t6, $t0, 2
15   bne $t6, $0, branch
16   j end
17
18   branch:
19   addi $t6, $0, 0x1
20   j loop
21
22   end:
23   addi $t3, $zero, 0x0 # t3 = 0
24   addi $t4, $zero, 0x0 # t4 = 0
25   addi $t5, $zero, 0x0 # t5 = 0
26   bne $t6, $0, end
```

thx

6: jump
15: branch taken
26: branch not taken



```
4   addi $t0, $zero, 0x0 # t0 = 0
5   addi $t1, $zero, 0x1 # t1 = 1
6   addi $t2, $zero, 0x2 # t2 = 2
7   addi $t3, $zero, 0x3 # t3 = 3
8   addi $t4, $zero, 0x4 # t4 = 4
9
10  add $t0, $t0, $t0 # t0 = 0
11  addi $t1, $zero, 0x1 # t1 = 1
12  addi $t2, $zero, 0x2 # t2 = 2
13  add $t3, $t0, $t1 # t3 = 1
14  add $t4, $t2, $t3 # t4 = 3
```

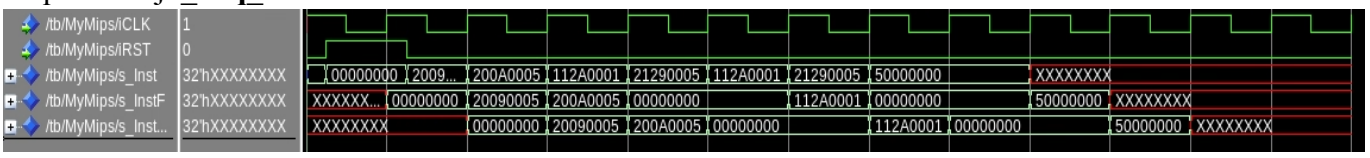
8: No Hazards
13: forwarding
14: forwarding and stall

instruction	i_type	r_type	branch/jump
sequential	1st 2nd	1st 2nd	
individual	1st 2nd	1st 2nd	1st

[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. (include modelsim output and tiny discussion of result correctness)

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

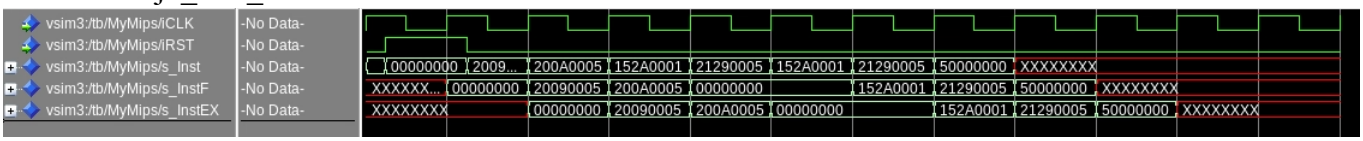
Beq : Proj2_**beq**_test.s




```
[inst #1] addi $9,$0,5
Register Write to Reg: 0x09 Val: 0x00000005
[inst #2] addi $10,$0,5
Register Write to Reg: 0x0A Val: 0x00000005
[inst #3] beq $9,$10,1
[inst #4] halt
```

For this test, our processor goes instruction by instruction and then halts in the correct order avoiding all control hazards. After it detects a branch hazard it flushes and stalls the next instruction till the branch is calculated.

Bne : Proj2_**bne**_test.s




```
[inst #1] addi $9,$0,5
Register Write to Reg: 0x09 Val: 0x00000005
[inst #2] addi $10,$0,5
Register Write to Reg: 0x0A Val: 0x00000005
[inst #3] bne $9,$10,1
[inst #4] addi $9,$9,5
Register Write to Reg: 0x09 Val: 0x0000000A
[inst #5] halt
```

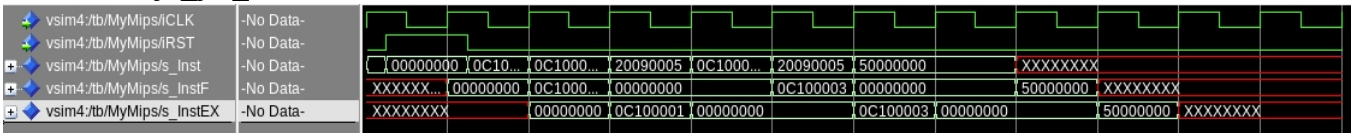
For this test, our processor goes instruction by instruction and then halts in the correct order avoiding all control hazards. This program doesn't need to flush or stall the next instructions since it doesn't have any control hazards

J : Proj2_**j**_test.s


```
[inst #1] j 4194316
[inst #2] addi $9,$0,3
Register Write to Reg: 0x09 Val: 0x00000003
[inst #3] addi $10,$0,7
Register Write to Reg: 0x0A Val: 0x00000007
[inst #4] halt
```

For this test, our processor goes instruction by instruction and then halts in the correct order avoiding all control hazards. After detecting a jump our program flushes the next two instructions in order to have the correct PC value and run the correct instruction.

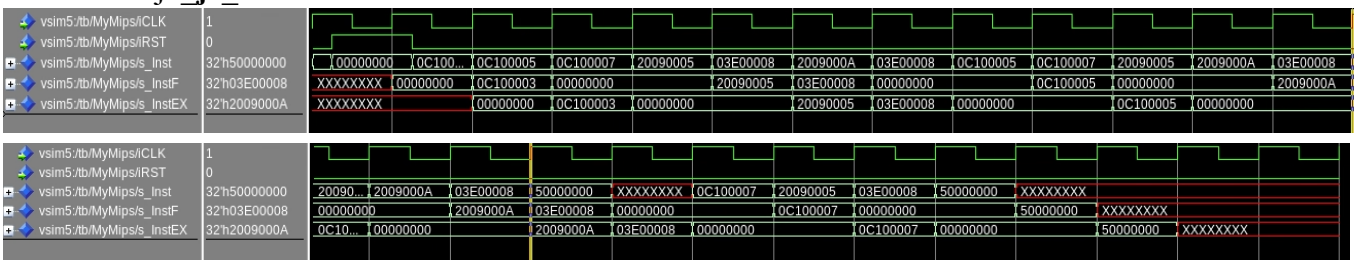
Jal : Proj2_**jal**_test.s




```
[inst #1] jal 4194308
Register Write to Reg: 0x1F Val: 0x00400004
[inst #2] jal 4194316
Register Write to Reg: 0x1F Val: 0x00400008
[inst #3] halt
```

For this test, our processor goes instruction by instruction and then halts after avoiding control hazards. After jal our program flushes in order to have the correct pc value.

Jr : Proj2_**jr**_test.s

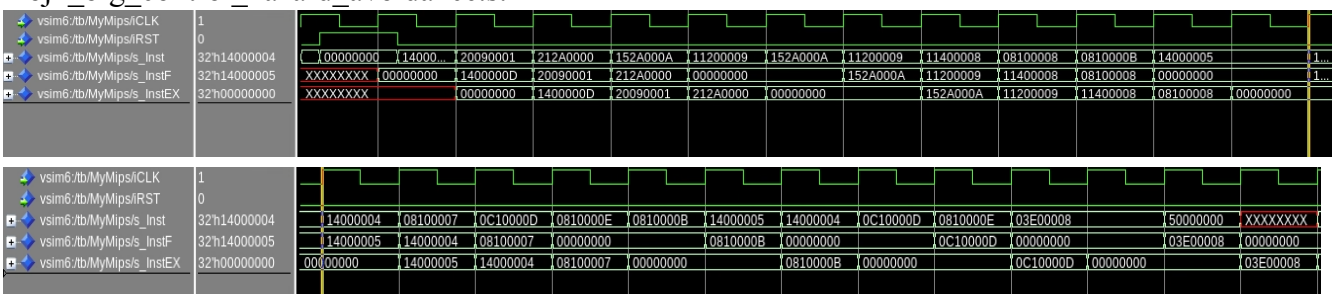


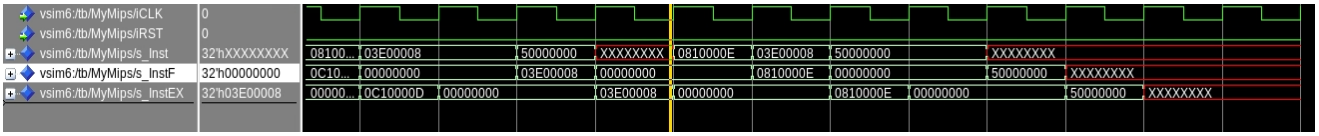

```
[inst #1] jal 4194316
Register Write to Reg: 0x1F Val: 0x00400004
[inst #2] addi $9,$0,5
Register Write to Reg: 0x09 Val: 0x00000005
[inst #3] jr $31
[inst #4] jal 4194324
Register Write to Reg: 0x1F Val: 0x00400008
[inst #5] addi $9,$0,10
Register Write to Reg: 0x09 Val: 0x0000000A
[inst #6] jr $31
[inst #7] jal 4194332
Register Write to Reg: 0x1F Val: 0x0040000C
[inst #8] halt
```

This test simply starts out with a jal instruction, which properly takes and has its stalls/flushes reflected properly in the waveform. After this $t1 gets a value of 5 added to it, which is shown in the waveform, and then a jr instruction occurs. Then another jal instruction occurs which adds 10 onto $t1 again, then returns. After this the program finally jumps and link's (though it could just jump) to the halt signal and finishes execution.

My assembly program: Proj2_**big**_control_hazard_avoidance.s tests each of these instructions in combination.

Proj2_big_control_hazard_avoidance.s:






```
[inst #1] bne $0,$0,13
[inst #2] addi $9,$0,1
Register Write to Reg: 0x09 Val: 0x00000001
[inst #3] addi $10,$9,0
Register Write to Reg: 0x0A Val: 0x00000001
[inst #4] bne $9,$10,10
[inst #5] beq $9,$0,9
[inst #6] beq $10,$0,8
[inst #7] j 4194336
[inst #8] bne $0,$0,5
[inst #9] bne $0,$0,4
[inst #10] j 4194332
[inst #11] j 4194348
[inst #12] jal 4194356
Register Write to Reg: 0x1F Val: 0x00400030
[inst #13] jr $31
[inst #14] j 4194360
[inst #15] halt
```

For this last big test, the first bne should not take and is reflected as such in the waveform. After this two addi's are performed properly, followed by another bne and two more beq's that do not take. After this a jump occurs, which inserts the correct amount of stalls/flushes, which causes two more bne's to occur – neither of which take either. After this another jump occurs wherein a sequential jump occurs, which causes a jal to occur, then a jr, then finally a jump to exit wherein the halt signal propagates through and finishes execution.

instruction	beq	bne	j	jal	jr
sequential execution	**big**	**big**	**big**	**jal**	**jr**
individual execution	**beq**	**big/bne**	**big/j**	**big**	**big**

The entries in the spreadsheet show where each of the cases are tested (which assembly file tests them).

[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).

54.60 MHz
Critical path: Branch
(zero flag (brAz) calculated in the decode stage (calculated in the top-level, not its own component). branch address calculated in execute stage (it does not need to be but we didn't want to bother changing it))