# **CprE 381: Computer Organization and Assembly-Level Programming**

# **Project Part 2 Report**

Team Members: Connor Hand

Zach Scurlock

## Project Teams Group #: 3

***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.

In the fetch stage, the datapath value needed is the instruction, there are no control signals.

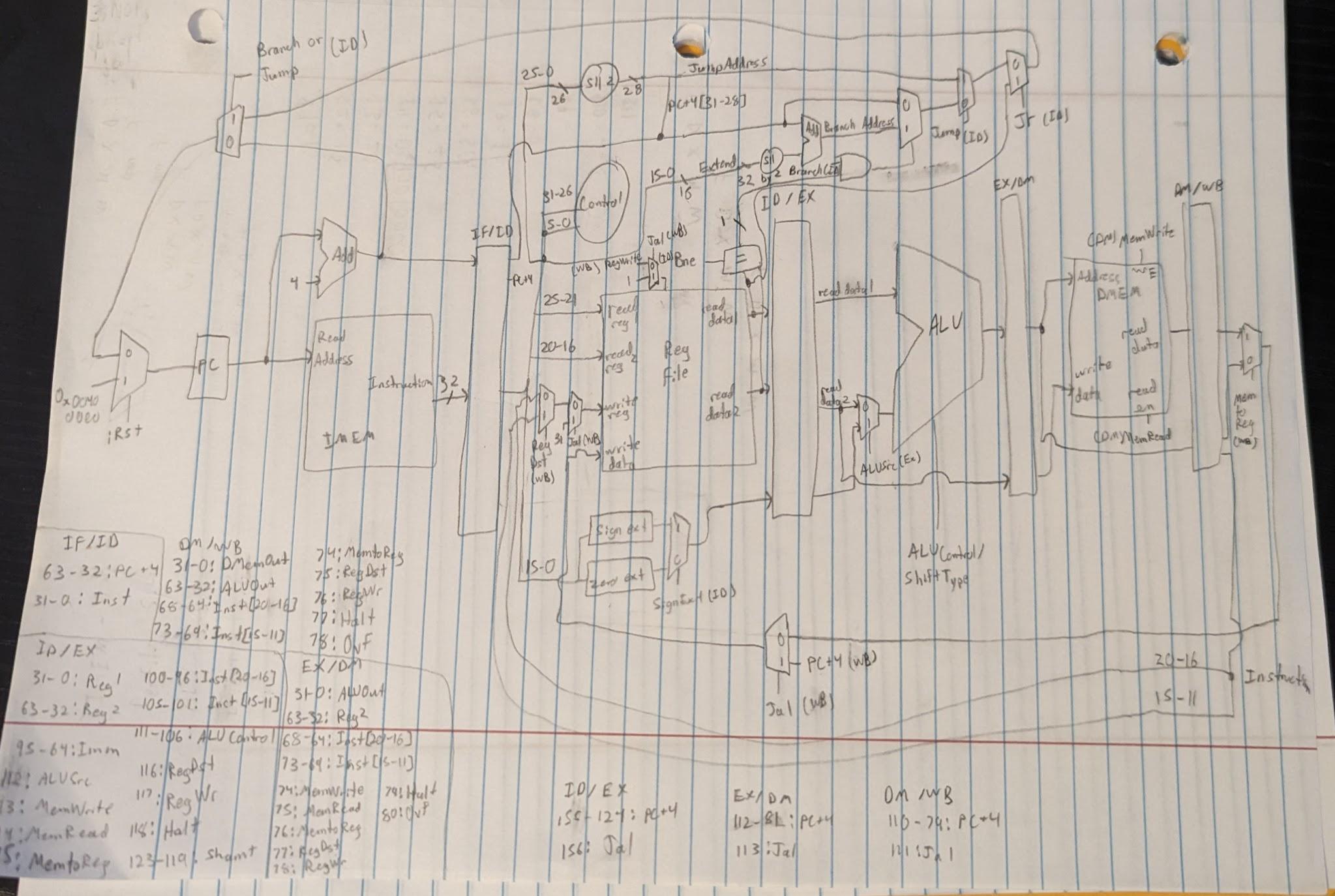
In the decode stage, the datapath values needed are the jump address, branch address, new PC value, entire instruction, register read 1, register read 2, and the immediate value. The control signals needed in the decode stage are SignExt, Bne, Branch, Jump, Jr, BranchOrJump, and iRST is used as a control signal in the MUX that goes directly to the PC register.

In the execute stage, the datapath values needed are reg read 1, reg read 2, immediate value, and ALUout. The control signals needed are ALUControl/ShiftType and ALUSrc.

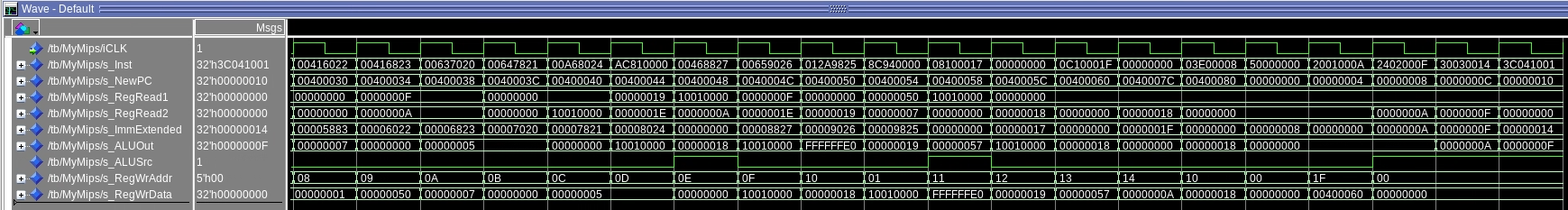
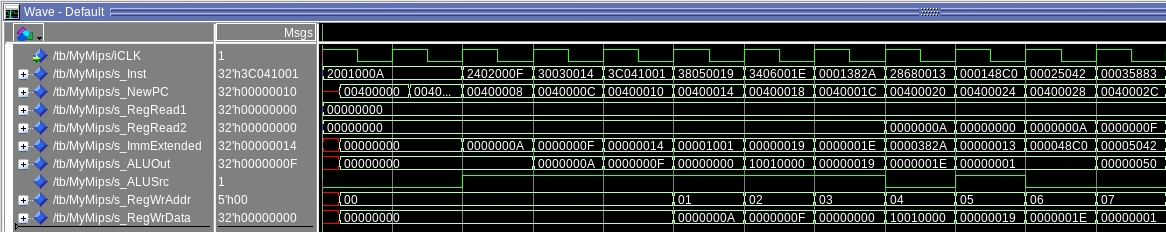
In the memory stage, the datapath values needed are ALUout, reg read 2, and read data signal from memory output. The control signals needed are MemWrite and MemRead.

In the writeback stage, the datapath values needed are ALUout, memory data, PC+4, and the instruction’s rt and rd values. The control signals needed are MemToReg, Jal, RegDst, and RegWrite.

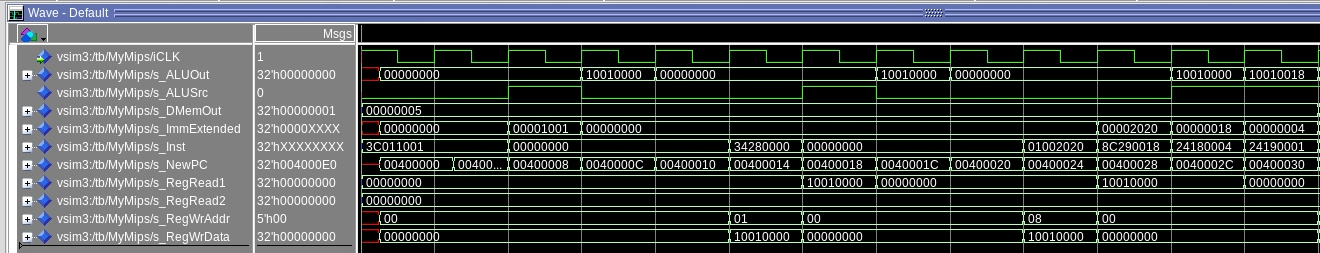
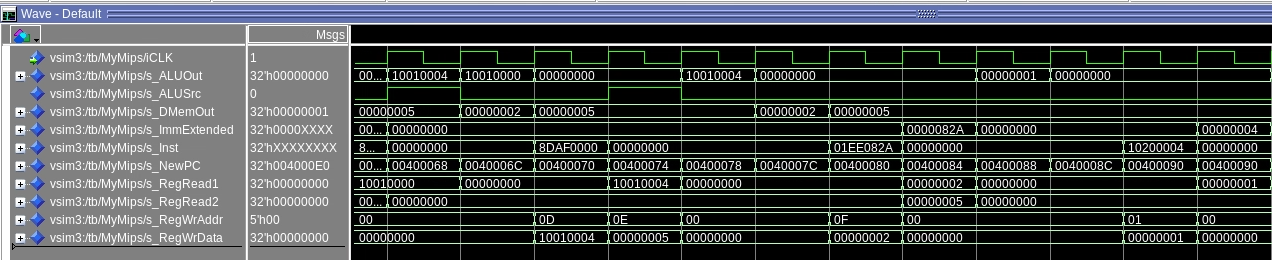
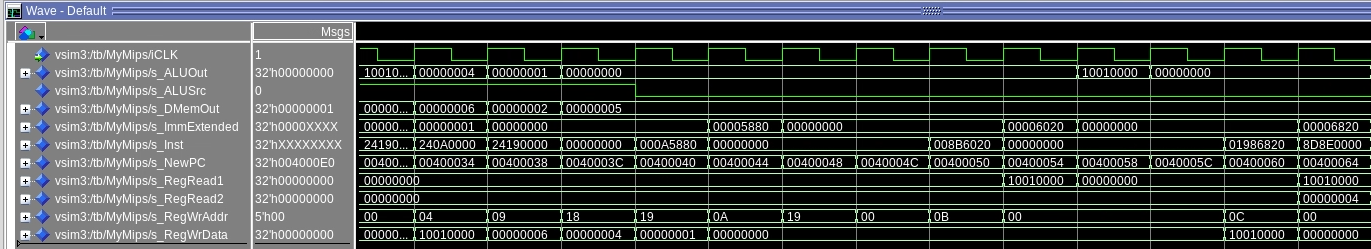
[1.b.ii] high-level schematic drawing of the interconnection between components.



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

Looking at all of these signals and comparing to the MIPS assembly program, you can see that all of the signal values are correct and it is working as intended.

[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.

Looking at all of the signals, you can see that the results are correct.   
  
 # Increment the inner loop counter

addi $t2, $t2, 1

nop

nop

nop

# Check if the inner loop is finished

bne $t2, $t1, inner\_loop

nop

Above is an example from the bubblesort program. Did not have to use the maximum number of nops after addi (data-flow), and did not have to use the maximum number of nops after bne (control).

j outer\_loop

nop

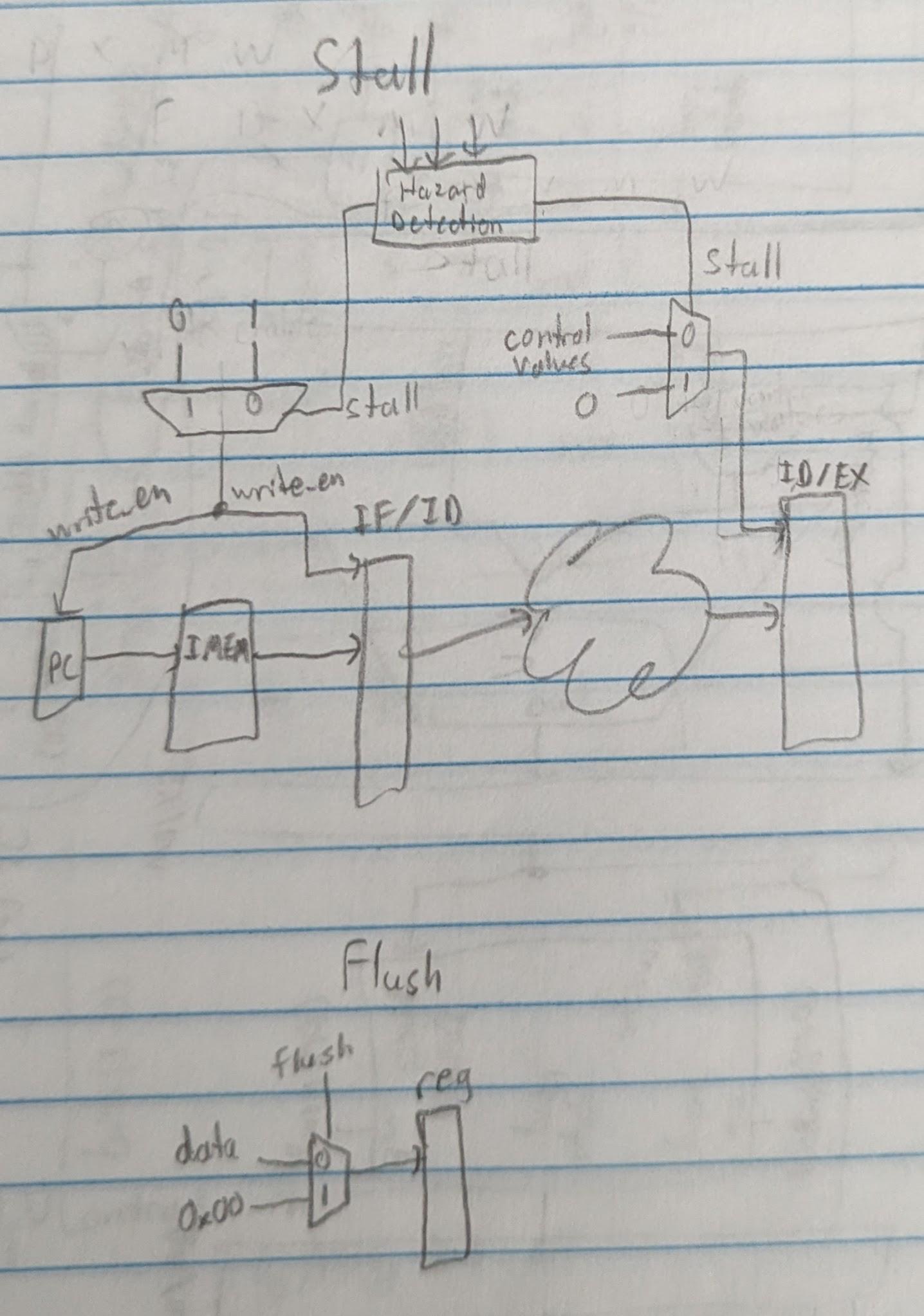
This is another example of not having to use the maximum number of nops .

[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).

56.48 MHz

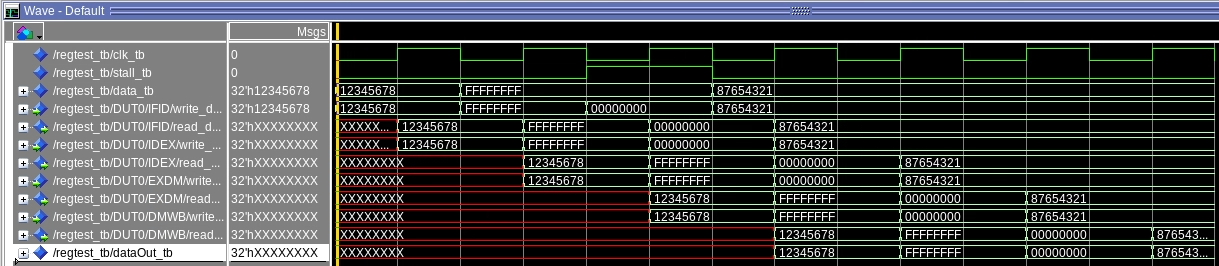
The critical path is the lw instruction. It goes through IMEM, RegFile, ALU, DMEM, RegFile

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



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

Our implementation does not require individually stalling or flushing registers, it only requires stalling the processor as a whole. This is because our hazard detector will detect any branching or jumping instructions and stall the pipeline. This testbench shows that values stored in the initial IF/ID register are available as expected four cycles later, and that the pipeline can be stalled.



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

| Instructions | Signals Produced |
| --- | --- |
| add, addi, addu, addiu, and, andi, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sub, subu, lui | s\_WriteAddrID, s\_WriteAddrEX, s\_WriteAddrMEM, s\_WriteEnID, s\_WriteEnEX, s\_WriteEnMEM, |
| beq, bne | s\_Branch, s\_Bne |
| j, jal, jr | s\_Jump, s\_Jr, s\_JalID, s\_JalEX, s\_JalMEM, s\_JalWB |

[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, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sub, subu, lui  
  
The above instructions consume s\_ReadAddr1 and s\_ReadAddr2

[2.b.iii] generalized list of potential data dependencies. From this generalized list, select those dependencies that can be forwarded (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

Dependency which requires hazard stalls: s\_WriteAddrID, s\_WriteAddrEx, s\_WriteEnID, s\_WriteEnEX, s\_ReadAddr1, s\_ReadAddr2

Dependency that can be forwarded: s\_WriteEnMEM, s\_WriteAddrMEM, s\_ReadAddr1, s\_ReadAddr2

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

IF - s\_NewPC, s\_NextInstAddr, s\_PCPlusFour, Hazard Detection signals

ID - s\_DINST, s\_Reg1Forwarded, s\_Reg2Forwarded, SignExt, FwdSel1/2, s\_Bne, s\_Branch, s\_Jump, s\_BranchOrJump, s\_JumpAddress, s\_BranchAddress, s\_ImmExtended

EX - s\_ALUSrc, s\_ALUControl, Alu inputs A and B (most signals come from s\_IDEXoutput), s\_ALUout

DM - s\_MemWrite, s\_MemRead, s\_DMemWr, s\_DMemAddr, s\_DMemData (most signals come from s\_EXDMoutput)

WB - s\_Jal, s\_PCPlusFour, s\_MemToReg, s\_WriteAddr, s\_RegWr, s\_RegWrAddr, s\_RegWrData (most signals come from s\_DMWBoutput)

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

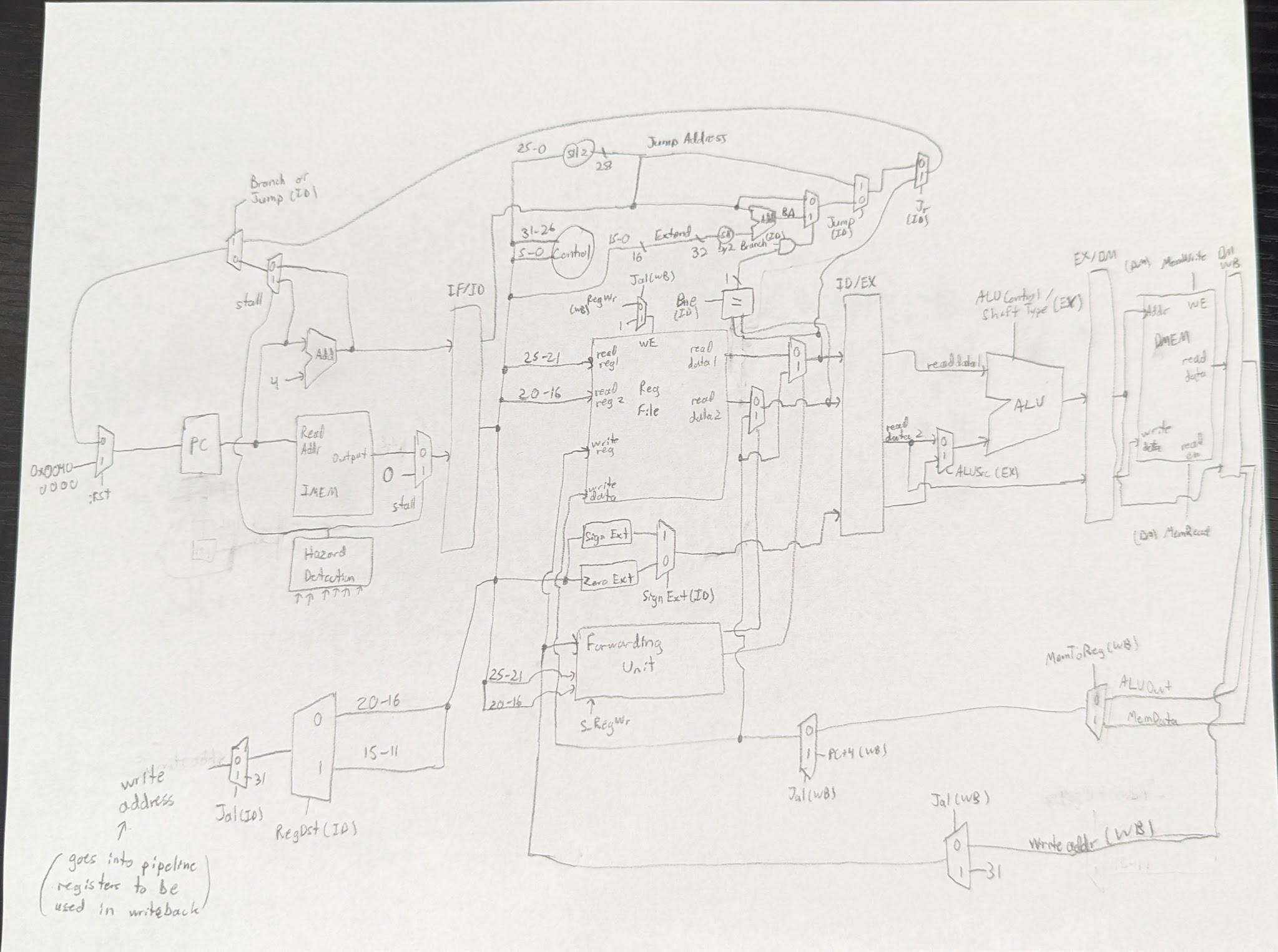
beq, bne, j, jr - ID stage

jal - WB stage

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

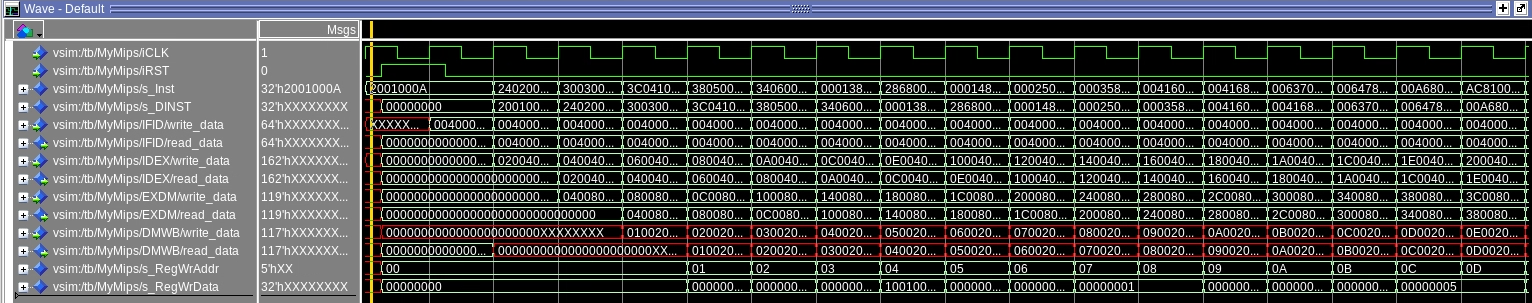
After beq, bne, j, and jr, there will be one stall inserted at the beginning (in the IF/ID register). After jal, there will need to be three stalls so that the return address can be saved into the register file.

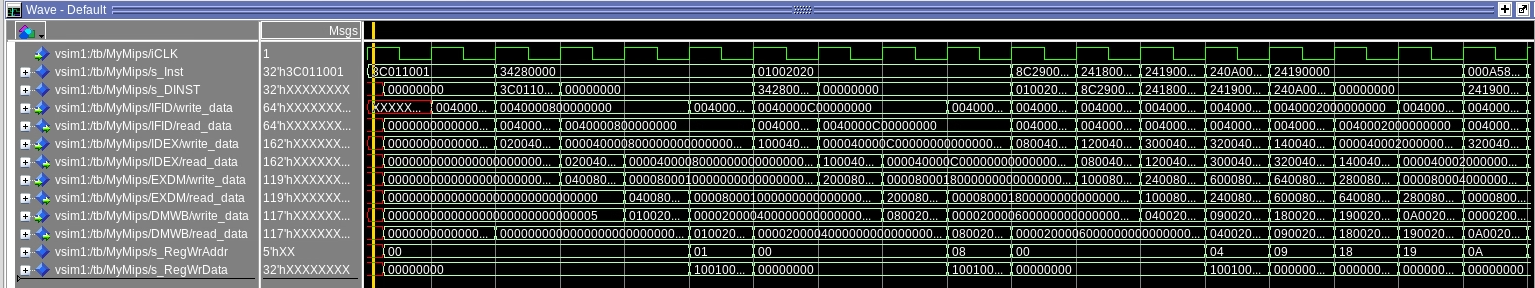
[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.



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

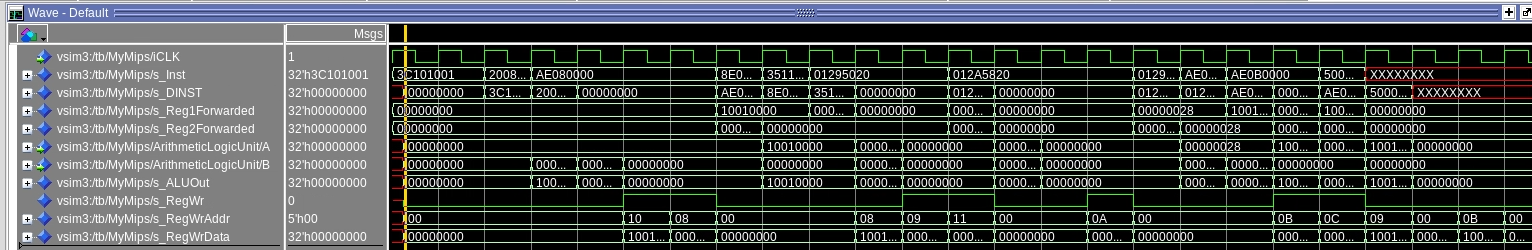
Proj2\_basetest:

Proj2\_bubblesort:



[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.

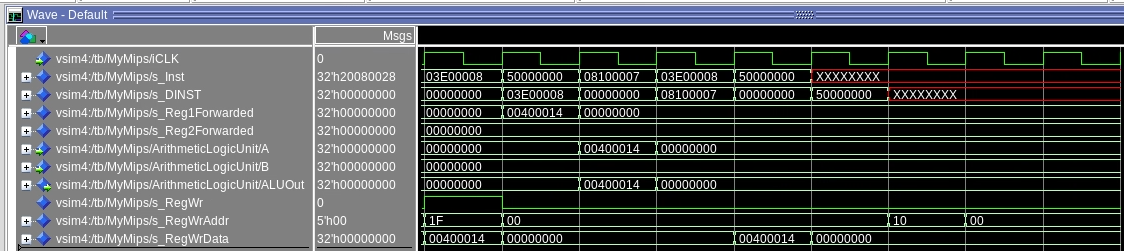
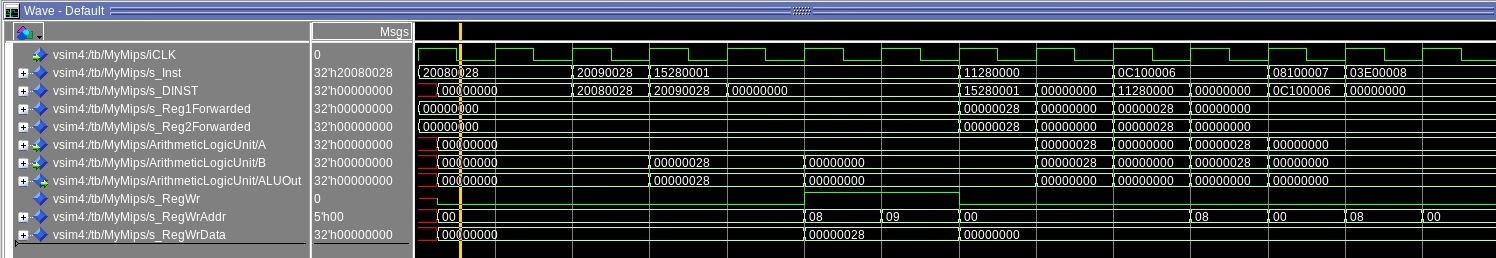
| Data Hazard | Assembly |
| --- | --- |
| Instruction in IF depends on data in ID | lui $s0, 0x1001  addi $t0, $zero, 40  sw $t0, 0($s0) |
| Instruction in IF depends on data in EX | lw $t1, 0($s0)  ori $s1, $t0, 20  add $t2, $t1, $t1 |
| Instruction in IF depends on data in DM | add $t3, $t1, $t2  add $t4, $t1, $t1  sw $t1, 0($s0)  sw $t3, 0($s0) |



[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.

| Control Hazard | Assembly |
| --- | --- |
| jump, jal, jr, beq, bne | addi $t0, $0, 40  addi $t1, $0, 40  bne $t1, $t0, jumper  beq $t1, $t0, jumper  jumper:  jal jaler  j exit  jaler:  jr $ra  exit:  halt |

This approach thoroughly tests control hazards by testing each branching possibility.



[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).

54.06 MHz

The critical path is when lw is used. The path goes through IMEM, RegFile, ALU, DMEM, and the RegFile again.