# CprE 381: Computer Organization and Assembly-Level Programming

## Hardware and Software Scheduled Pipelined Processor Design Report

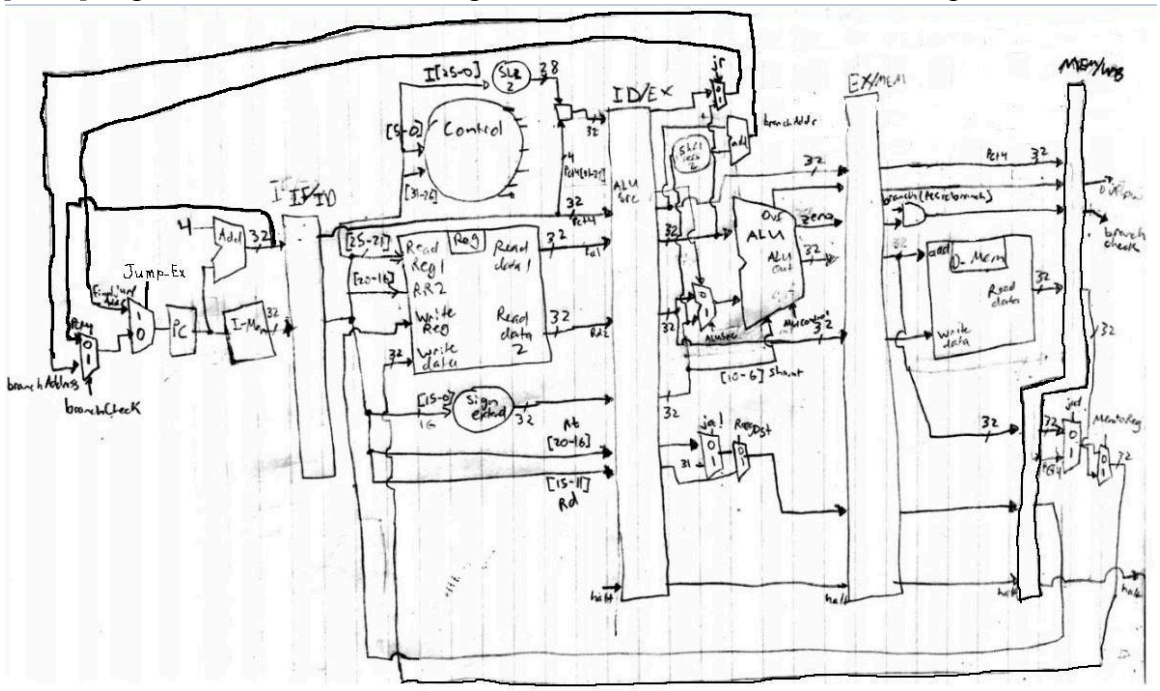
**Team Members:** Lex Somers & Remington Greatline

**Project Teams Group #:** 2_01

**[1.a]** Global list of control signals required for each pipeline stage.

Control Signals:jr, jal, ALUSrc, ALUControl, MemtoReg, we_mem, we_reg, RegDst, Branch, SignExt, j, Halt

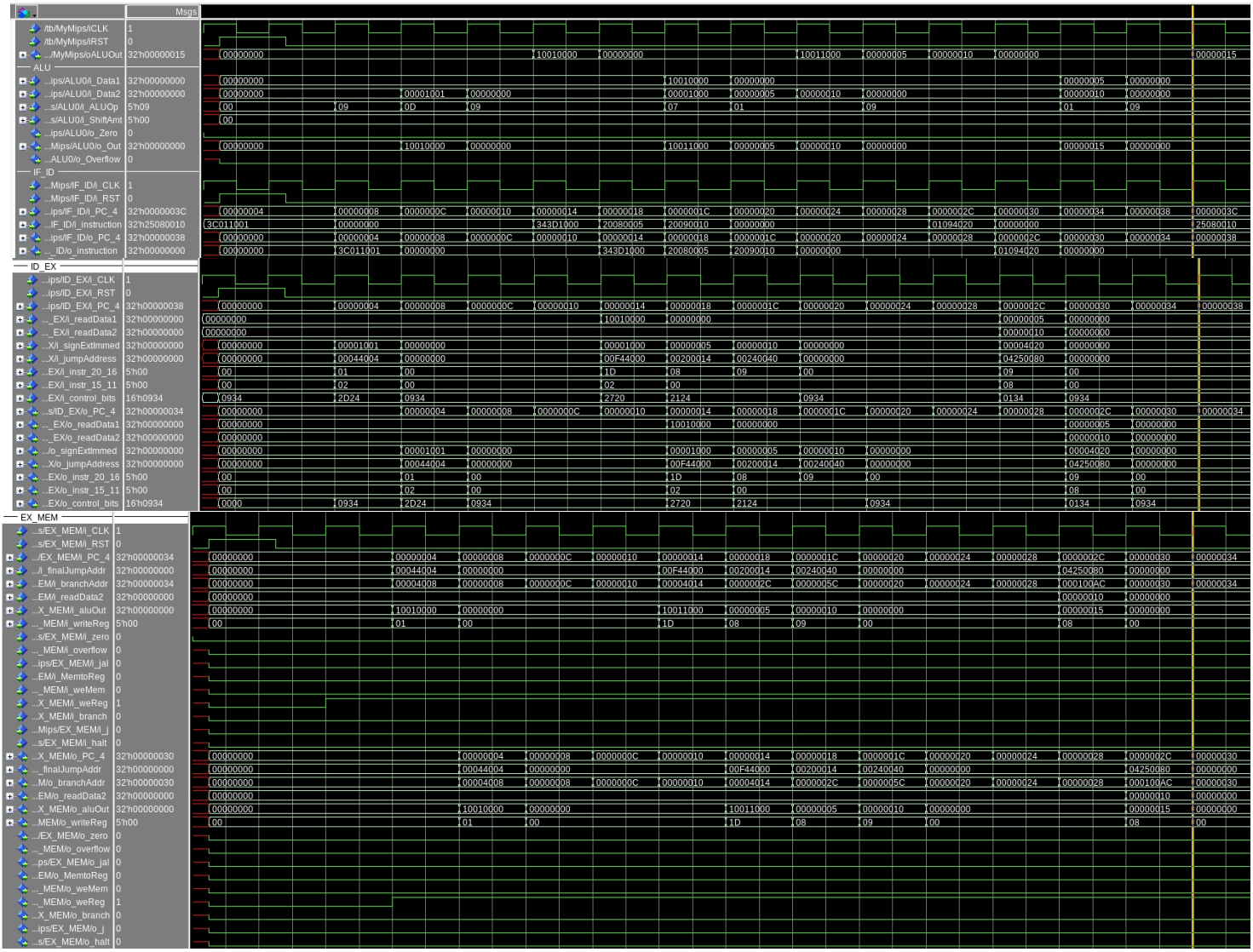
**[1.b.ii]** High-level schematic drawing of the interconnection between components.

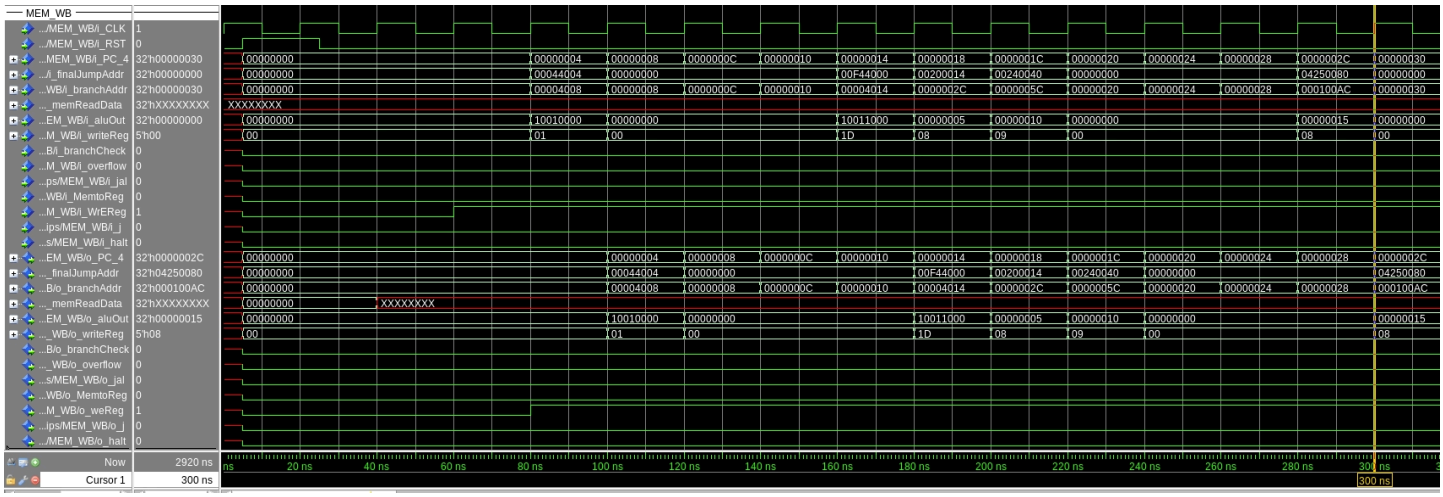



**[1.c.i]** Annotated waveform from tests and brief discussion of result correctness.
Test result:

```
Testing file: proj/mips/proj2_base_test_ss.s
Mars simulation: pass
Modelsim simulation: pass
Test Result: fail
Oh no...

Cycle: 117
MARS instruction number: 110    Instruction: jal 4194840
Expected:       Register Write to Reg: 0x1F Val: 0x004001C8
Got     :       Register Write to Reg: 0x1F Val: 0x000001C8
Incorrect write

Almost! your processor completed the program with  1/3 allowed mismatches

Helpful resources for Debugging:
ms.trace : output from the VHDL testbench during program execution on your processor
mars.trace : output from MARS containing expected output
vsim.wlf: waveform file generated by processor simulation, you can display this simulation in ModelSim without resimulating your processor by hand


Results in: output/proj2_base_test_ss.s
-----------------------------------------------
[rg2@vlinux-34 cpre381-toolflow_sw]$
```

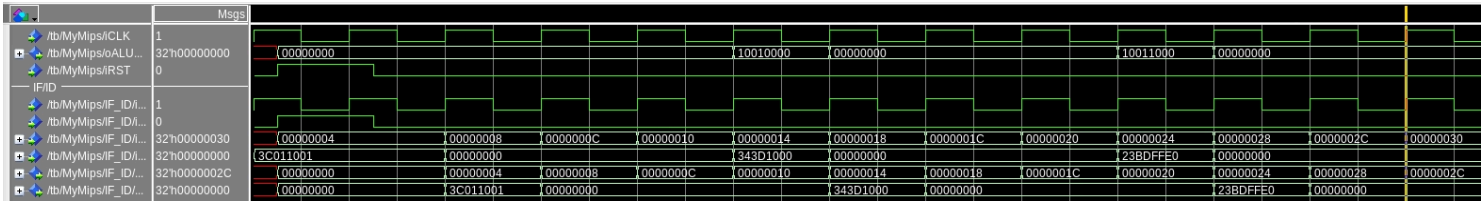
Waveform:

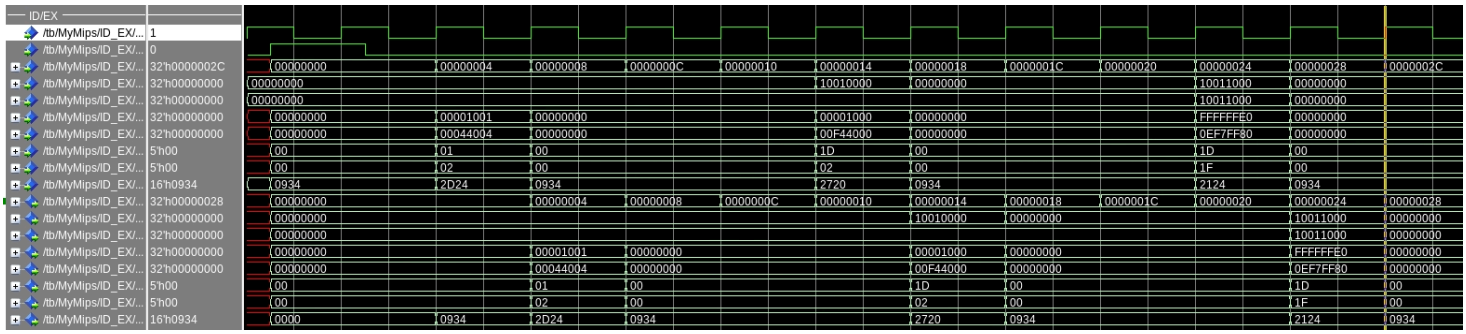


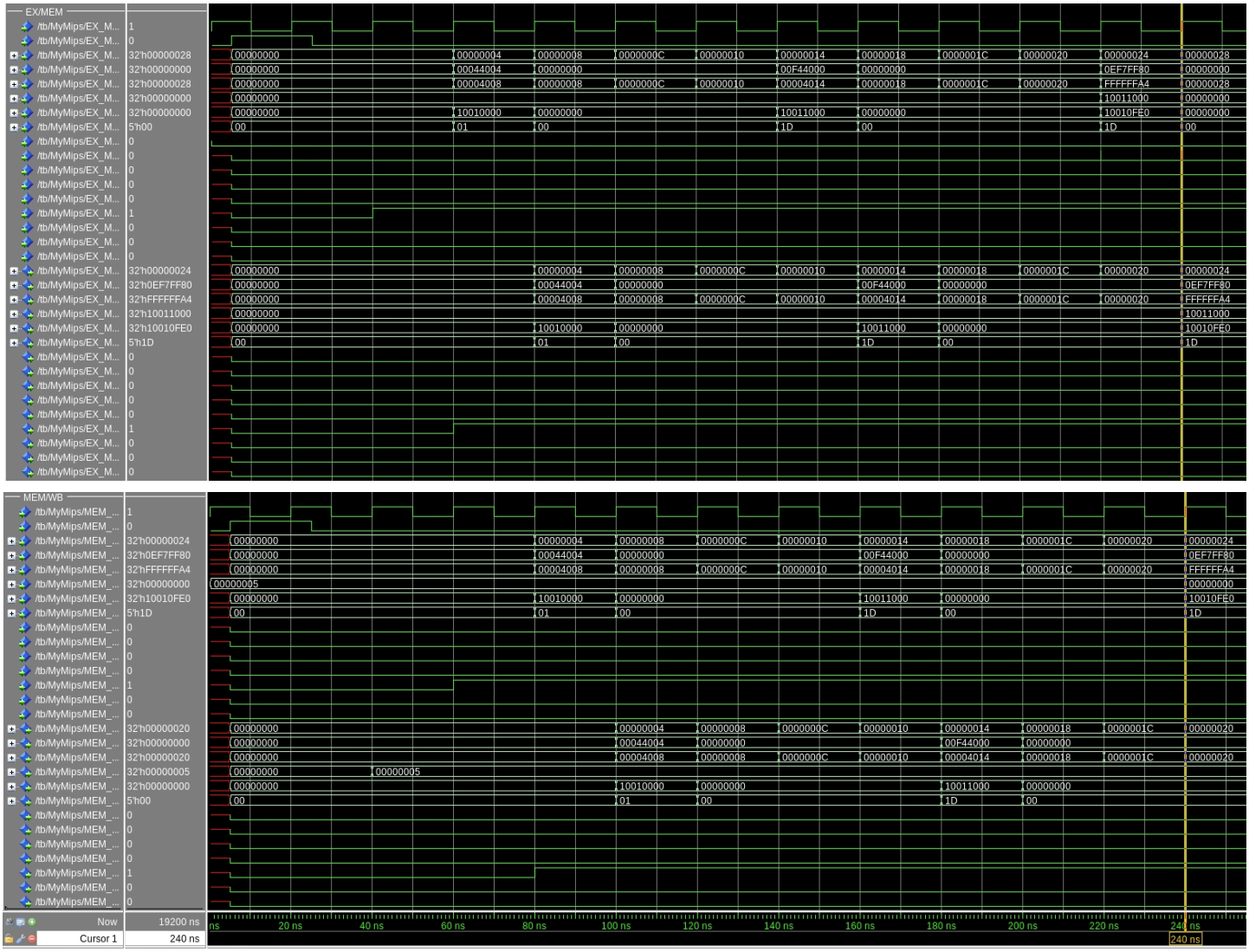



The base test is a rework of the project 1 base tests. The pipeline uses all the currently supported operations, and we removed all the data hazards from the assembly program. The wave form shows the ALU, and all the stages of the pipeline. Each PC+4 is passed down the line while the next (PC+4, PC+8) populates the pipeline stage before it. There were a few errors with the jal instruction just like in the single cycle processor because we were unable to get the base address for the instructions to load into the pipeline for some reason.. Otherwise there are no errors in the testing and the waveform analysis shows all the correct values.

**[1.c.ii]** Annotated waveform of two iterations of programs executing correctly and brief discussion of result correctness. Three examples are given in the waveforms for which we did not require the maximum number of NOPs, with at least one being in the data-flow and control-flow each.



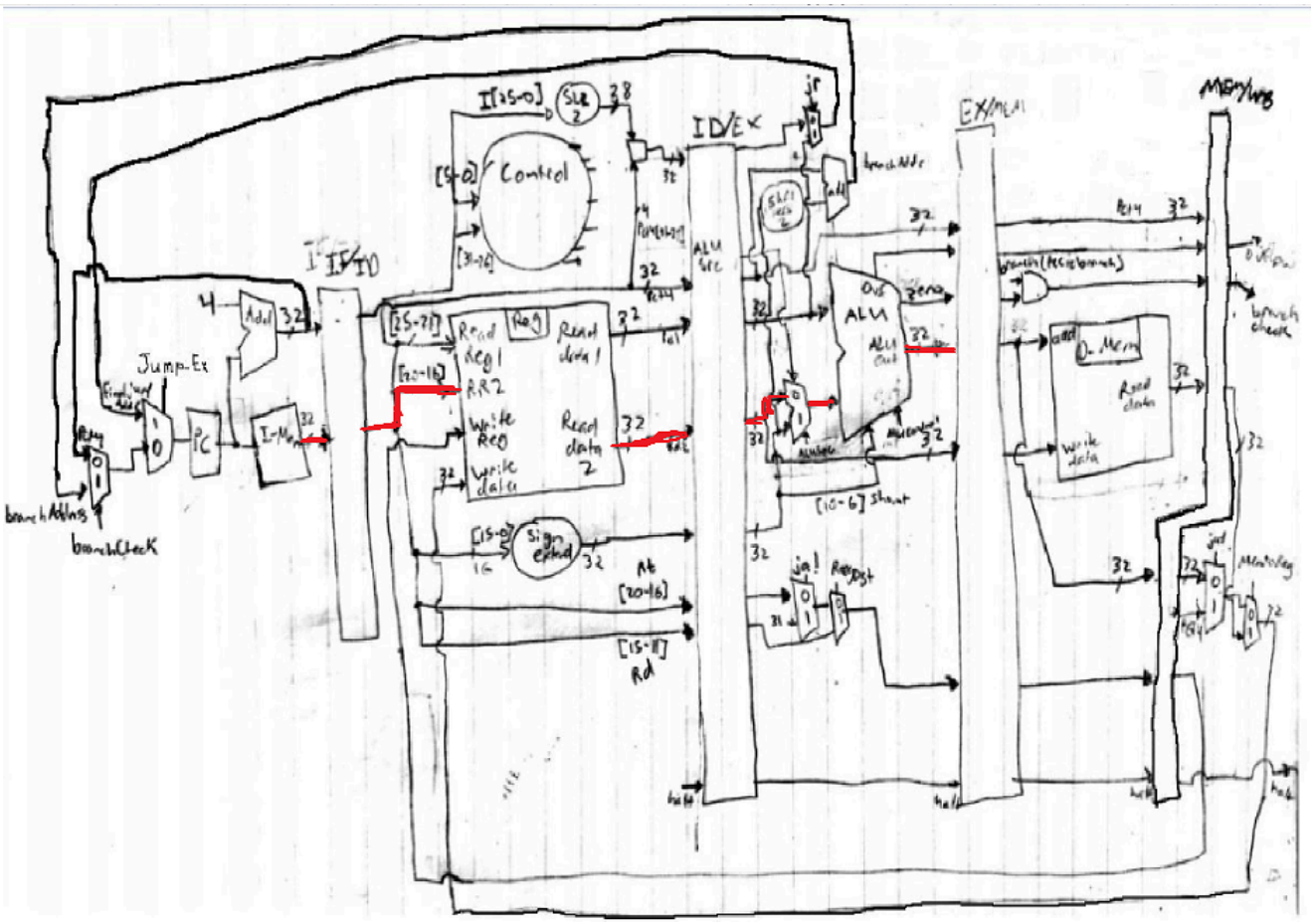



The waveforms are as expected. In our implementation, the processor loads $t0 = a[i] and $t1 = a[i+1]. if $t1 is less than $t0, then it does NOT branch and proceeds to save a[i] = $t1 and a[i+1] = t0. In the above waveform, a[i] =3 and a[i+1] = 2, so it does not branch and swaps the values.

Each pipeline state remains isolated while there are no data inconsistencies. When we flush the system, it allows branches and jumps to function properly.

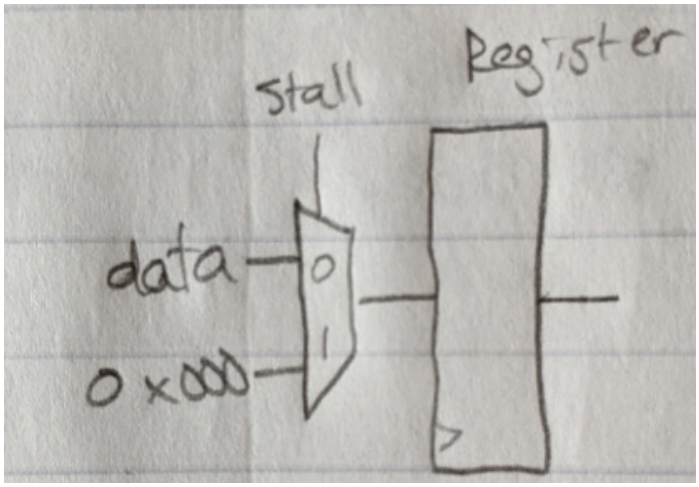
Regular instructions don’t need a flush for data dependencies. In those situations, only three nops are needed, as shown in the waveform.

**[1.d]** Maximum frequency the software-scheduled pipelined multi-cycle processor can run at, its critical path (highlighted in red), and each module/component the path goes through.

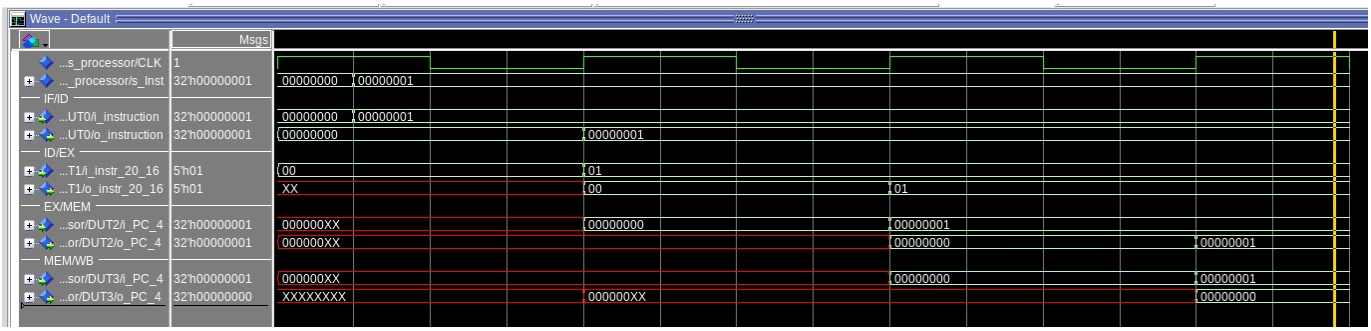



The maximum frequency for our processor is 50.7 MHz
Critical path goes as follows: The critical path consists of:
ID_EX_reg, MUX_ALUSrc, ALU, EX_MEM

**[2.a.ii]** Simple schematic of how to implement stalling and flushing with an ideal N-bit register.



**[2.a.iii]** Testbench instantiating all four registers in a single design. Shows that value 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 cycle. Tests each pipeline register's ability to be individually stalled and or flushed.




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

| **Instruction** | **Signals Produced** |
|---|---|
| add, addu, and, nor, xor, or, slt, sll, srl, sra, sub, subu | s_writeAddr_ID , s_writeAddr_EX , s_writeAddr_MEM, s_writeEnable_ID, s_writeEnable_EX, s_writeEnable_MEM |
| addi, addiu, andi, xori, ori, slti, lui, jr | |
| beq, bne | s_branch_ID , s_branch_EX |
| j, jal, jr | s_jump_ID , s_jump_EX |

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

| **Instruction** | **Signals Consumed** |
|---|---|
| add, addu, and, nor, xor, or, slt, sll, srl, sra, sub, subu | s_readAddr1, s_readAddr2 |
| addi, addiu, andi, xori, ori, slti, lui, jr | s_readAddr1 |
| beq, bne | |
| j, jal, jr | |

**[2.b.iii]** Generalized list of potential data dependencies that can be forwarded (and the corresponding pipeline stages that will be forwarding and receiving the data) and those that will require hazard stalls.

| **Need Hazard Stalls** | | **Can be Forwarded** | |
|---|---|---|---|
| s_writeAddr_ID, s_writeAddr_EX, s_writeEnable_ID, s_writeEnable_EX, | s_readAddr1, s_readAddr2 | s_writeAddr_MEM, s_writeEnable_MEM | s_readAddr1, s_readAddr2, |

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

Decode stage:

- s_PCPlusFour_ID, s_DecodeData1_ID, s_DecodeData, s_imm32_ID, s_jumpAddress_ID, s_RegWrAddr_ID, s_Ctrl_ID

Execute Stage:

- S_PCPlusFour_EX, s_finalJumpAddress_EX, s_branchAddress_EX, s_RegOutReadData2_EX, s_aluOut_EX, s_RegWrAddr_EX, s_ALUBranch_EX, s_Ovfl_EX, s_jal_EX, s_MemtoReg_EX, s_DMemWr_EX, s_RegWr_EX, s_Branch_EX, s_jump_EX, s_Halt_EX

Memory Stage:

- s_PCPlusFour_MEM, s_DMemOut, s_aluOut_MEM, s_RegWrAddr_MEM, s_Ovfl_MEM, s_jal_MEM, s_MemtoReg_MEM, s_RegWr_MEM,, s_Halt_MEM

Write Back Stage:

- s_PCPlusFour_WB, s_memReadData_WB, s_aluOut_WB, s_RegWrAddr, s_Ovfl, s_MemtoReg_WB, s_RegWr, s_Halt

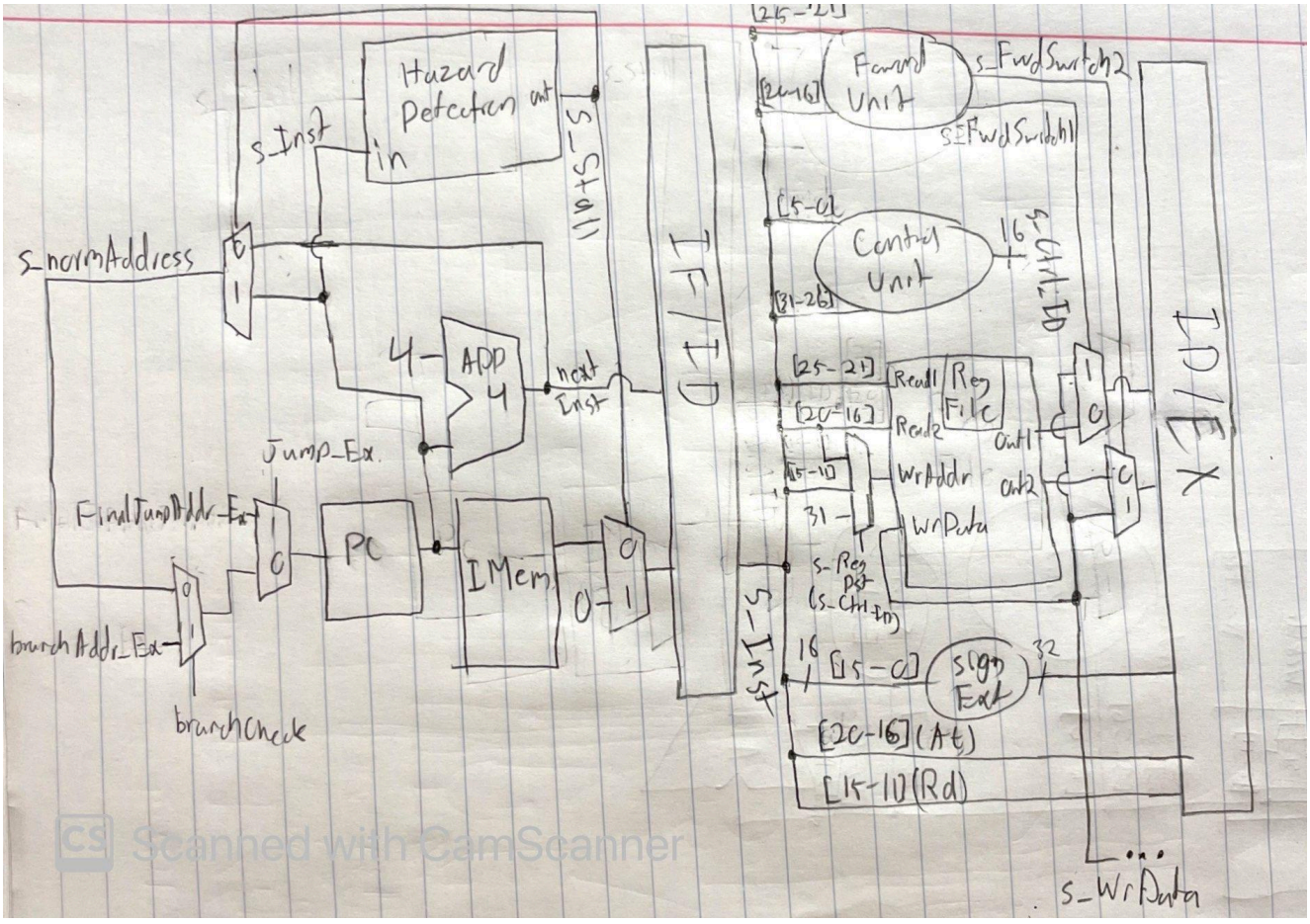
**[2.c.i]** List of all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

The instructions that may cause a non-sequential PC update are beq, bne, j, jr, and jal. This update would occur within the Execute stage (EX), as that is where we calculate branching and jumping addressing.

**[2.c.ii]** List of which stages need to be stalled and which stages need to be squashed/flushed relative to the stage each instruction is in for the instructions a from part 2.c.ii.

Because the hazard detection is within the fetch stage, the processor won't load an instruction if there is a branch or jump instruction in the pipeline already. Therefore, no stages would need to be squashed or flushed, and the processor will stall itself and wait for the hazard to resolve itself. It is possible that the IF, ID, Ex, and Mem stages need to wait while the jump instruction is within the WB stage.

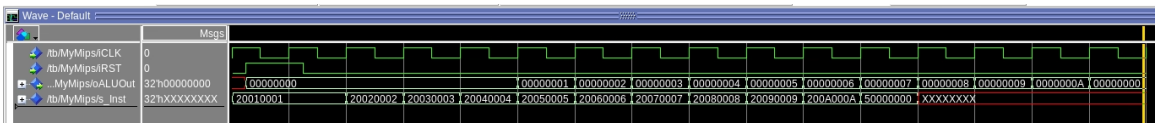
**[2.d]** High-level schematic drawing of the interconnection between components for the hardware-scheduled pipelined processor.




In this drawing, I only drew the first two stages, Fetch and decode, as they are the only two stages that were changed from the software implementation. Execute, memory, and Write back are essentially the exact same as before.

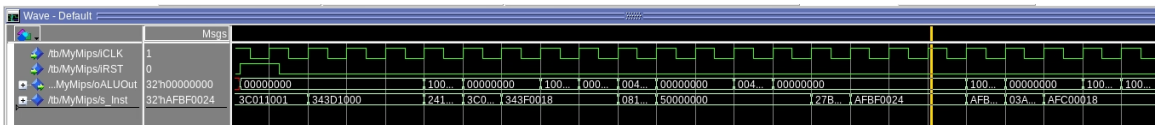
**[2.e – i, ii, and iii]** QuestaSim output for each of the following tests and a brief discussion of result correctness.

addiseq.s

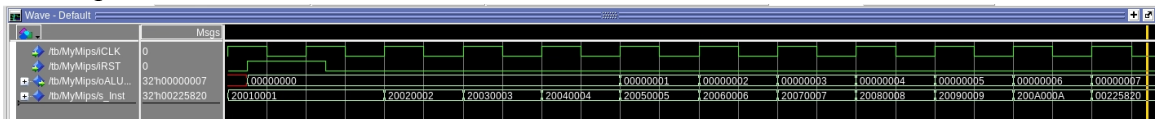



fibonacci.s

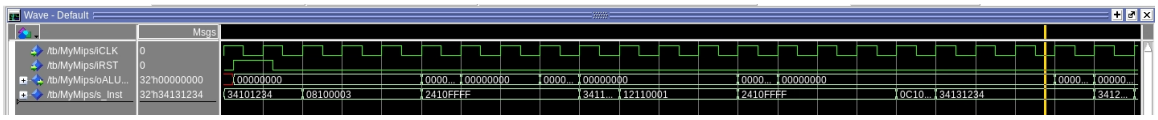

grendel.s




lab3Seq.s

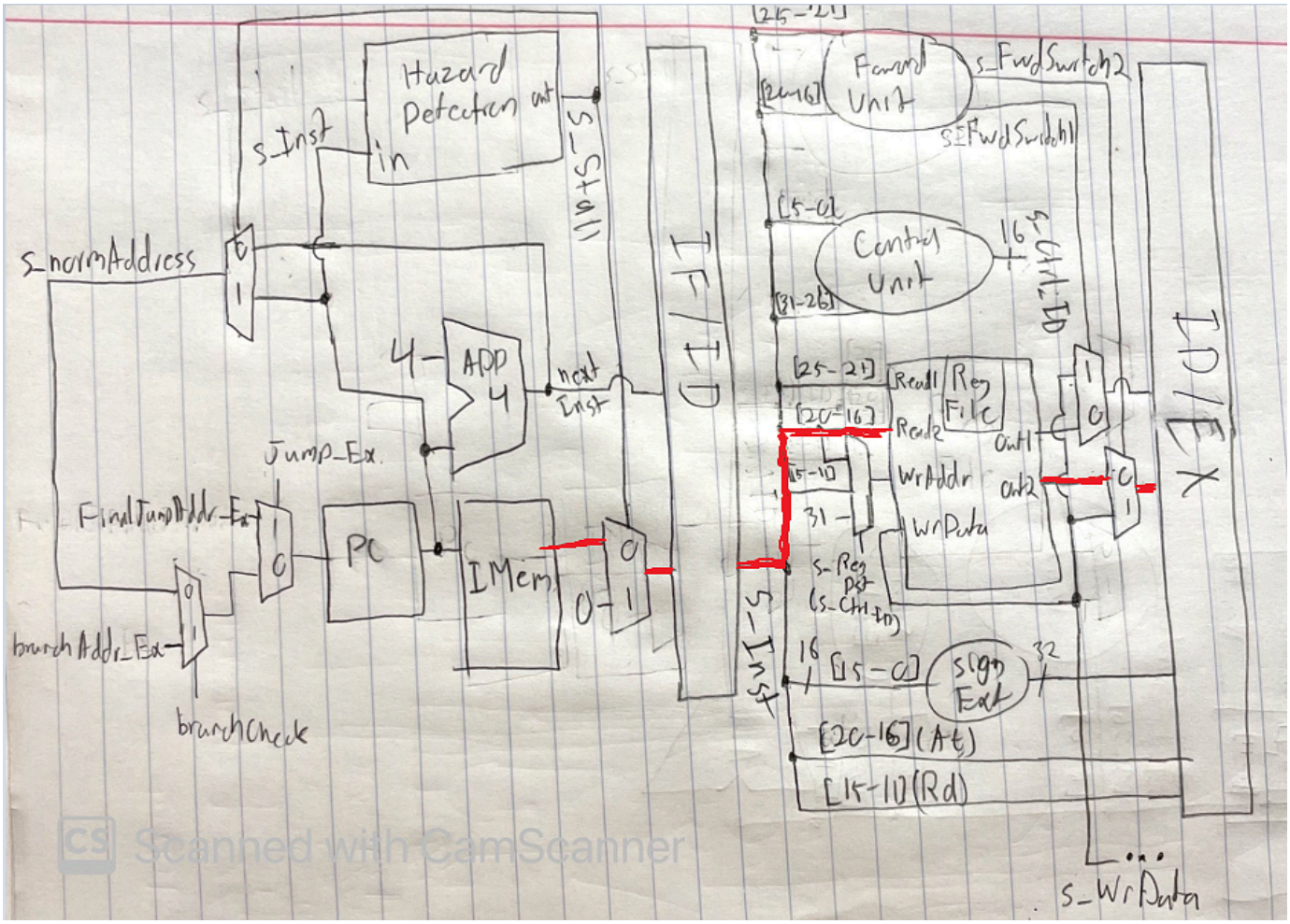



simplebranch.s




All of the tests ran correctly. The only issue we experienced with any of them was during jump and link instructions because we did not know how to fix the bug of the base address not being in the value we are linking. Otherwise, the waveforms looked as we expected them to. All results functioned as expected.

**[2.f]** Maximum frequency the hardware-scheduled pipelined multi-cycle processor can run at, its critical path (highlighted in red), and each module/component the path goes through.




The critical path consists of:
ID_EX_reg, MUX_ALUSrc, ALU, EX_MEM
46.7 MHz