CprE 381 – Computer Organization and Assembly-Level Programming

Proj-C Report

Lab Partners	______________________Muhamed Stilic________________
	___________________Trevor Rowland__________________
	____________________Felipe V. Carvalho________________
Section / Lab Time	____________Section 1/ 8am-9:50am______

Submit a typeset pdf version of this on Canvas by the due date. Refer to the highlighted language in the Proj-C instructions for the context of the following questions.

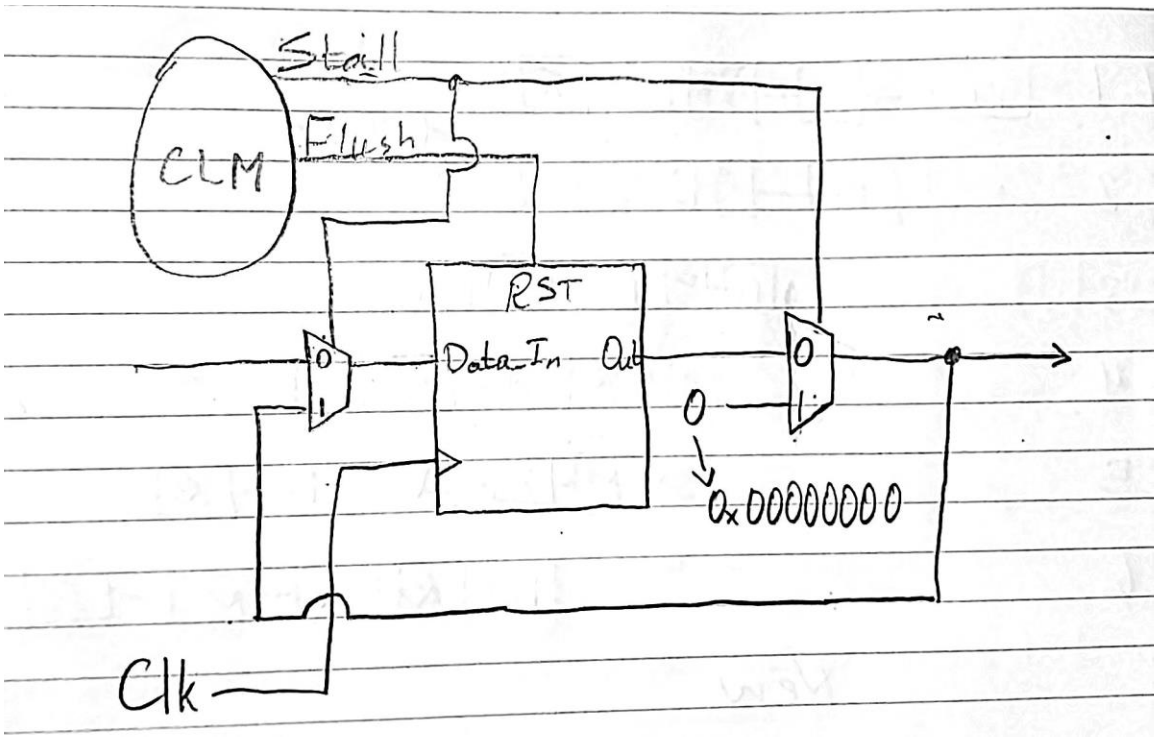
a. [Part 0] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

Global List:

Control Signal	Datapath Values
Stage: IF/ID(IF/ID Register to Main Control to ID/EX Register)	
ExtOp	1
ALUSrc	1
ALUOp	1
RegDst	1
MemWr	1
Branch	1
MemtoReg	1
RegWr	1
Stage: EX (ID/EX Register to Ex/MEM Register)	
ExtOp	X
ALUSrc	X
ALUOp	X
RegDst	X
MemWr	1
Branch	1
MemtoReg	1
RegWr	1
Stage: Mem(Ex/MEM Register to MEM/WB Register)	
MemWr	X
Branch	X
MemtoReg	1
RegWr	1
Stage: WB(MEM/WB Register to loop back around)	
MemtoReg	1
RegWr	1

b. [Part 1 (a)] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.

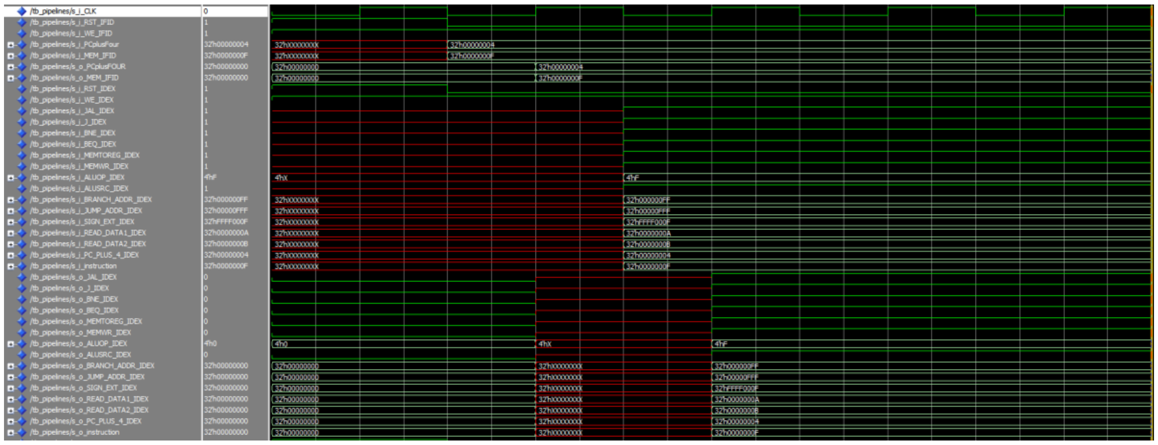
Schematic:



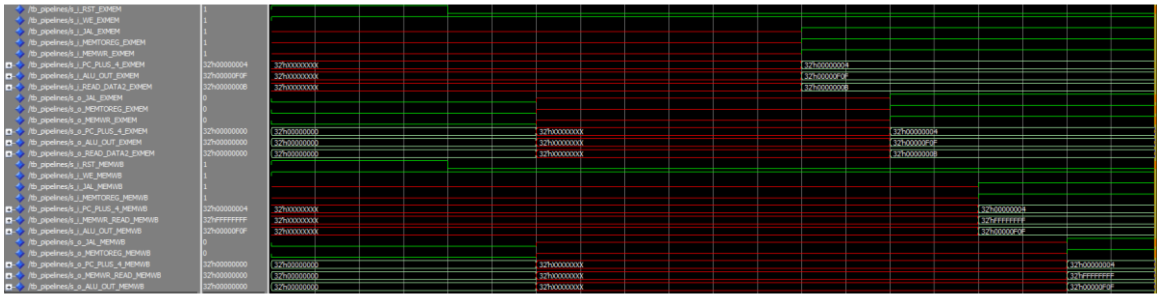
Explanations:
- Stalling is done above by multiplexing the data input to the register to either use the value coming in for the 0 input on the multiplexer, and the 1 input is the last used output of the register.
- Flushing is performed simply by resetting a register to 0 to clear out the data in the register.

c. [Part 1 (b)] 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. *[Please include waveforms and explanations.]*

Waveform:



```
/tb_pipelines/s_i_CLK                  0
/tb_pipelines/s_i_RST_IFID             1
/tb_pipelines/s_i_WE_IFID              1
/tb_pipelines/s_i_PCplusFour           32'h00000004
/tb_pipelines/s_i_MEM_IFID             32'h0000000F
/tb_pipelines/s_o_PCplusFOUR           32'h00000000
/tb_pipelines/s_o_MEM_IFID             32'h00000000
/tb_pipelines/s_i_RST_IDEX             1
/tb_pipelines/s_i_WE_IDEX              1
/tb_pipelines/s_i_JAL_IDEX             1
/tb_pipelines/s_i_J_IDEX               1
/tb_pipelines/s_i_BNE_IDEX             1
/tb_pipelines/s_i_BEQ_IDEX             1
/tb_pipelines/s_i_MEMTOREG_IDEX        1
/tb_pipelines/s_i_MEMWR_IDEX           1
/tb_pipelines/s_i_ALUOP_IDEX           4'hF
/tb_pipelines/s_i_ALUSRC_IDEX          1
/tb_pipelines/s_i_BRANCH_ADDR_IDEX     32'h000000FF
/tb_pipelines/s_i_JUMP_ADDR_IDEX       32'h00000FFF
/tb_pipelines/s_i_SIGN_EXT_IDEX        32'hFFFF000F
/tb_pipelines/s_i_READ_DATA1_IDEX      32'h0000000A
/tb_pipelines/s_i_READ_DATA2_IDEX      32'h0000000B
/tb_pipelines/s_i_PC_PLUS_4_IDEX       32'h00000004
/tb_pipelines/s_i_instruction          32'h0000000F
/tb_pipelines/s_o_JAL_IDEX             0
/tb_pipelines/s_o_J_IDEX               0
/tb_pipelines/s_o_BNE_IDEX             0
/tb_pipelines/s_o_BEQ_IDEX             0
/tb_pipelines/s_o_MEMTOREG_IDEX        0
/tb_pipelines/s_o_MEMWR_IDEX           0
/tb_pipelines/s_o_ALUOP_IDEX           4'h0
/tb_pipelines/s_o_ALUSRC_IDEX          0
/tb_pipelines/s_o_BRANCH_ADDR_IDEX     32'h00000000
/tb_pipelines/s_o_JUMP_ADDR_IDEX       32'h00000000
/tb_pipelines/s_o_SIGN_EXT_IDEX        32'h00000000
/tb_pipelines/s_o_READ_DATA1_IDEX      32'h00000000
/tb_pipelines/s_o_READ_DATA2_IDEX      32'h00000000
/tb_pipelines/s_o_PC_PLUS_4_IDEX       32'h00000000
/tb_pipelines/s_o_instruction          32'h00000000
```



```
/tb_pipelines/s_i_RST_EXMEM            1
/tb_pipelines/s_i_WE_EXMEM             1
/tb_pipelines/s_i_JAL_EXMEM            1
/tb_pipelines/s_i_MEMTOREG_EXMEM       1
/tb_pipelines/s_i_MEMWR_EXMEM          1
/tb_pipelines/s_i_PC_PLUS_4_EXMEM      32'h00000004
/tb_pipelines/s_i_ALU_OUT_EXMEM        32'h00000F0F
/tb_pipelines/s_i_READ_DATA2_EXMEM     32'h0000000B
/tb_pipelines/s_o_JAL_EXMEM            0
/tb_pipelines/s_o_MEMTOREG_EXMEM       0
/tb_pipelines/s_o_MEMWR_EXMEM          0
/tb_pipelines/s_o_PC_PLUS_4_EXMEM      32'h00000000
/tb_pipelines/s_o_ALU_OUT_EXMEM        32'h00000000
/tb_pipelines/s_o_READ_DATA2_EXMEM     32'h00000000
/tb_pipelines/s_i_RST_MEMWB            1
/tb_pipelines/s_i_WE_MEMWB             1
/tb_pipelines/s_i_JAL_MEMWB            1
/tb_pipelines/s_i_MEMTOREG_MEMWB       1
/tb_pipelines/s_i_PC_PLUS_4_MEMWB      32'h00000004
/tb_pipelines/s_i_MEMWR_READ_MEMWB     32'hFFFFFFFF
/tb_pipelines/s_i_ALU_OUT_MEMWB        32'h00000F0F
/tb_pipelines/s_o_JAL_MEMWB            0
/tb_pipelines/s_o_MEMTOREG_MEMWB       0
/tb_pipelines/s_o_PC_PLUS_4_MEMWB      32'h00000000
/tb_pipelines/s_o_MEMWR_READ_MEMWB     32'h00000000
/tb_pipelines/s_o_ALU_OUT_MEMWB        32'h00000000
```

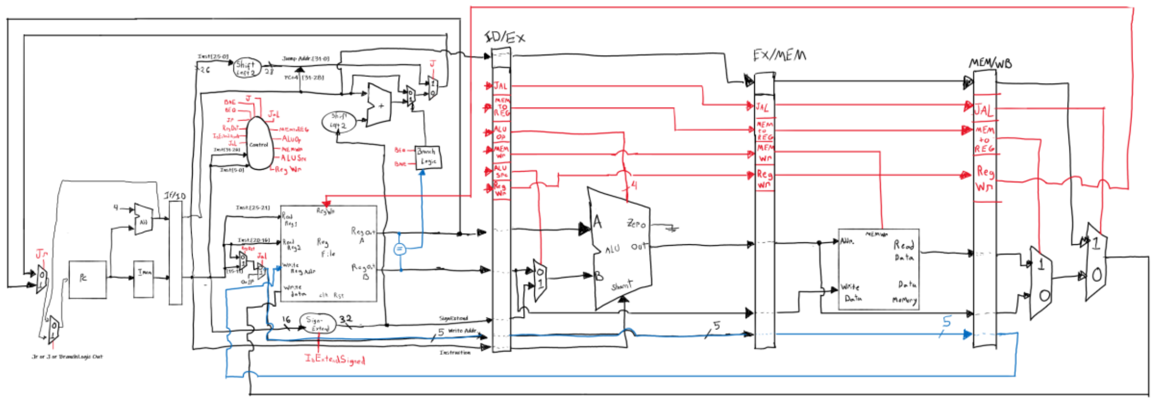
Explanation:

Above are the waveforms for the Pipeline test bench. Pipelines were connected in their designed order. The test bench starts by resetting all of the pipelines (equivalent to a flush). Then, throughout its four cycles, it inputs artificial values into the pipelines to simulate how the pipeline hardware would be implemented with our already designed components from project B. It is possible to see that the first pipeline register (IF/ID) keeps its initial assigned value during the entire duration of the simulation. The registers used in the pipelines are the same as the ones used in the Register File with the only difference being the bit size of some of them.

It is possible to see that the pipelines work by evaluating the input and output signals of each register and how they are being held. The progressive "stair" shape of these waveforms indicate that the values are progressively being inputted into the pipelines and are being correctly held by the registers. The values are being passed on correctly with no change.

d. [Part 2] In your writeup, provide your schematic for this part, describe what challenges (if any) you faced in implementing this module.

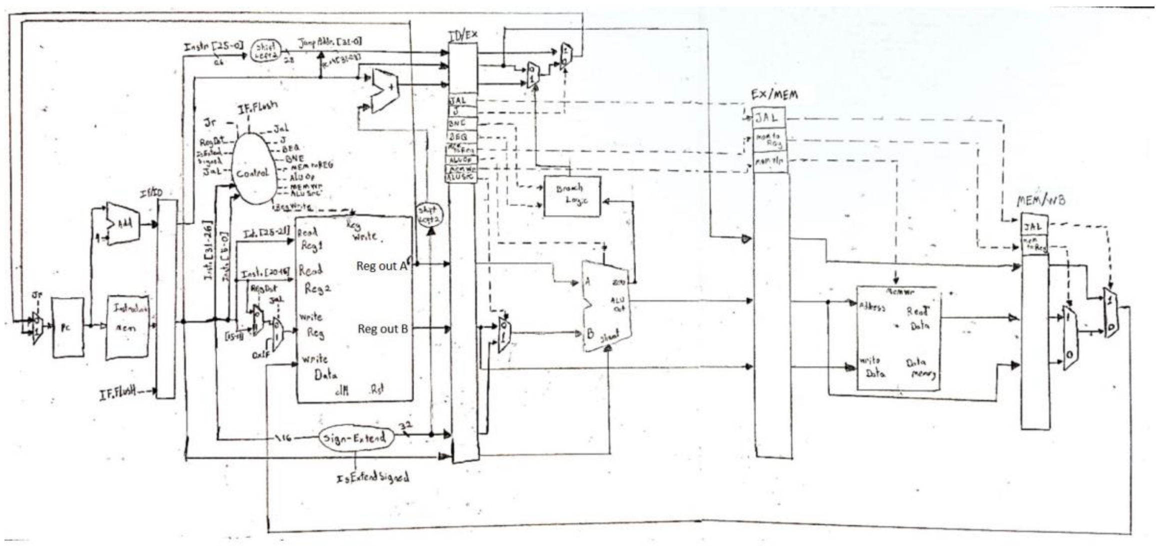
Schematic:



Challenges:

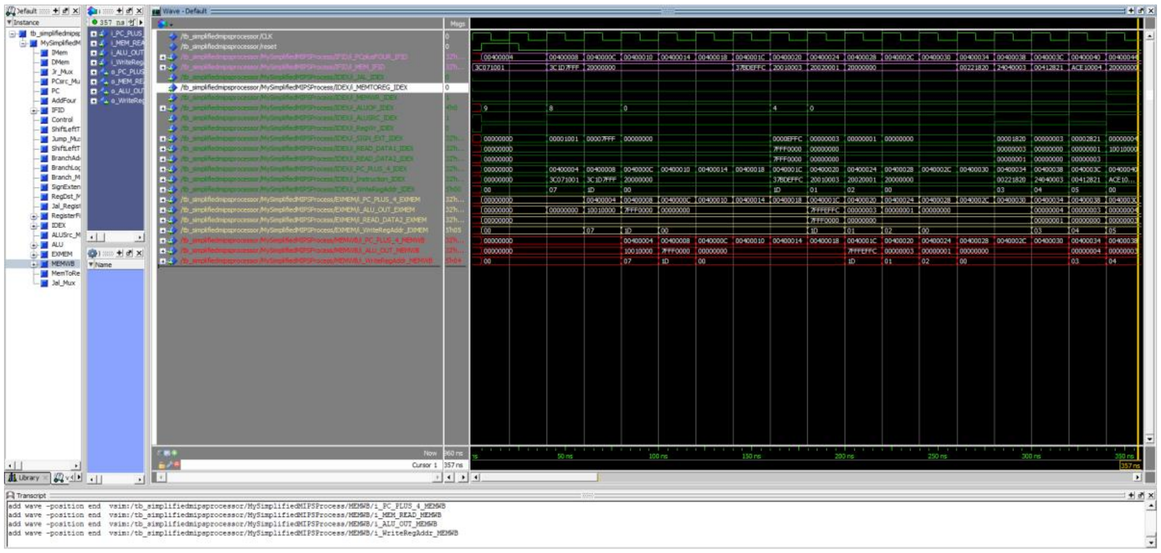
The Challenges that the team has faced were adding in a CLM signal for the Register Write Address and adding the Write Address signal to each pipelined register. After an initial run of tests, the team realized that a separate Register Write(RegWr) signal needed to be added to the Control Logic and have it run through each of the pipelined registers, then circle back to the Instruction Memory Registers. The second thing the group had to adjust was the connection from the two multiplexers close to the Write Address Register(Write Data on the diagram) in the picture above. Instead of wiring the output of the multiplexers right up to the Write Data Register, it must instead be sent through each of the pipeline registers, and then send it back to the Write Data port on the register after passing through the Pipeline Registers.

Some minor additions that needed to be made in the schematic were just adding the control signals to the pipeline registers and to the registers in the ModelSim files as well. The first iteration of the schematic is shown below, with the additions and edits made to the final document, shown above, in red and blue.



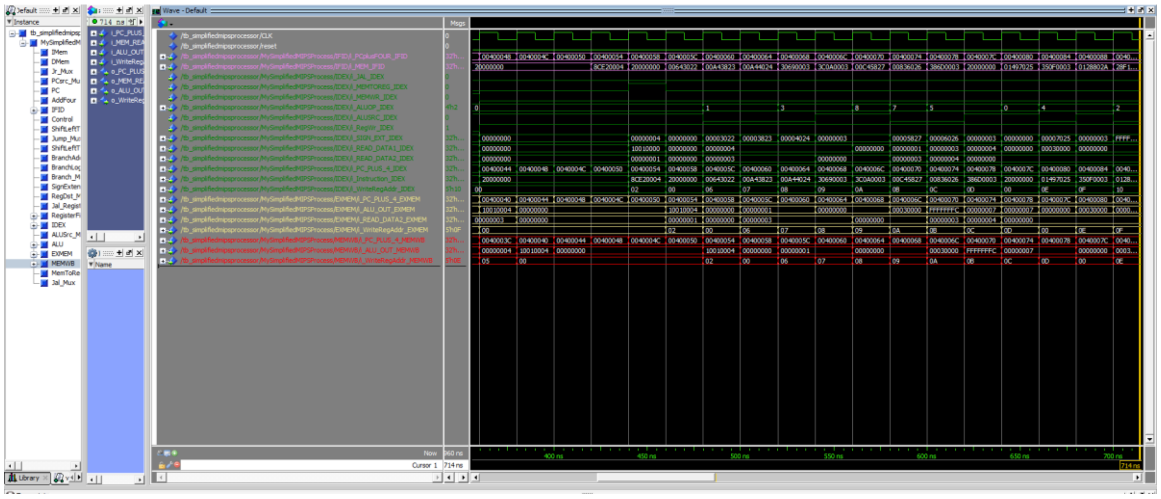
e. [Part 3 (a)] In your writeup, show the ModelSim output for the individual instruction tests, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

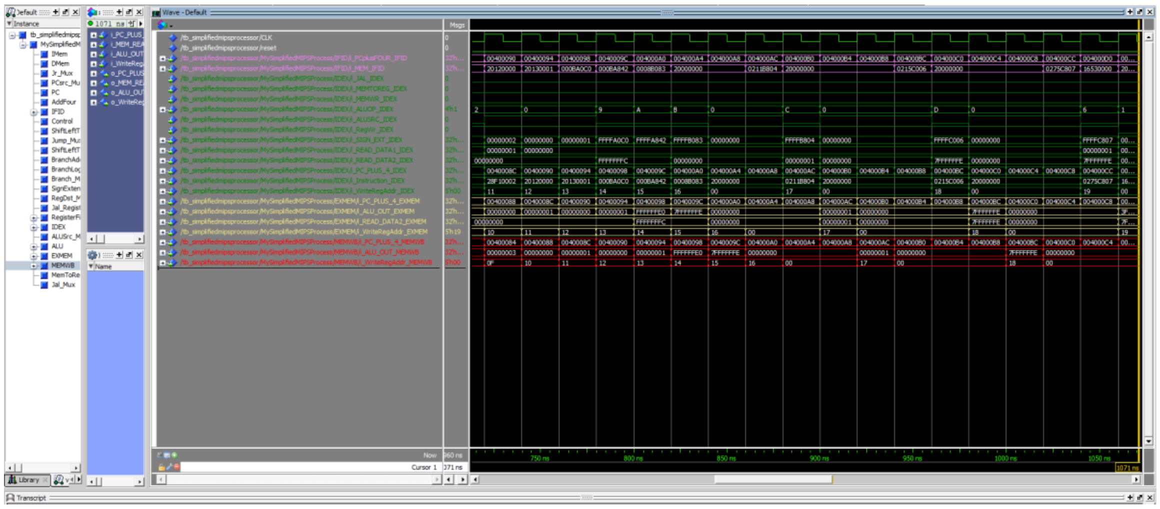
Waveform:

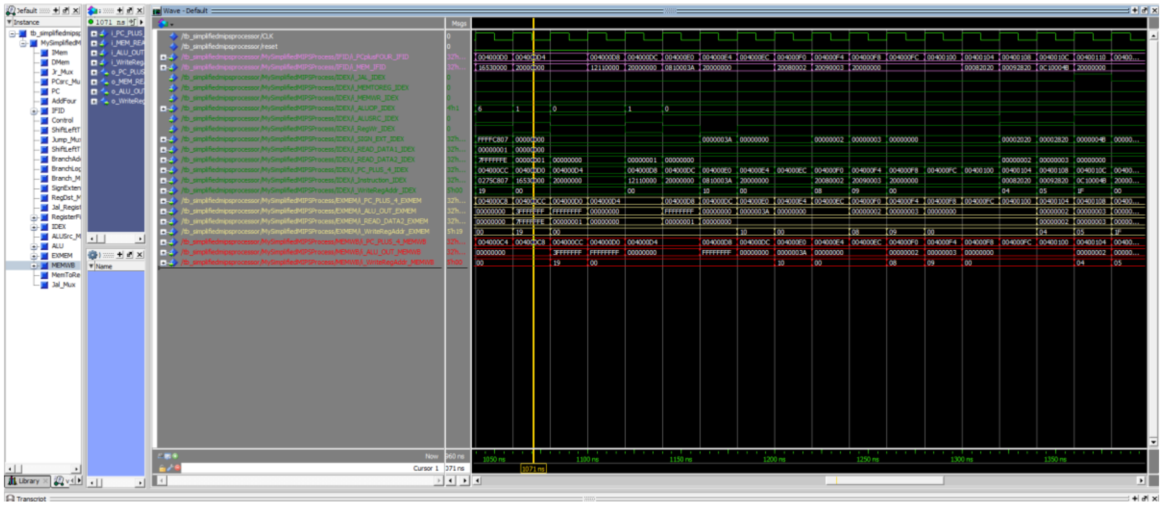


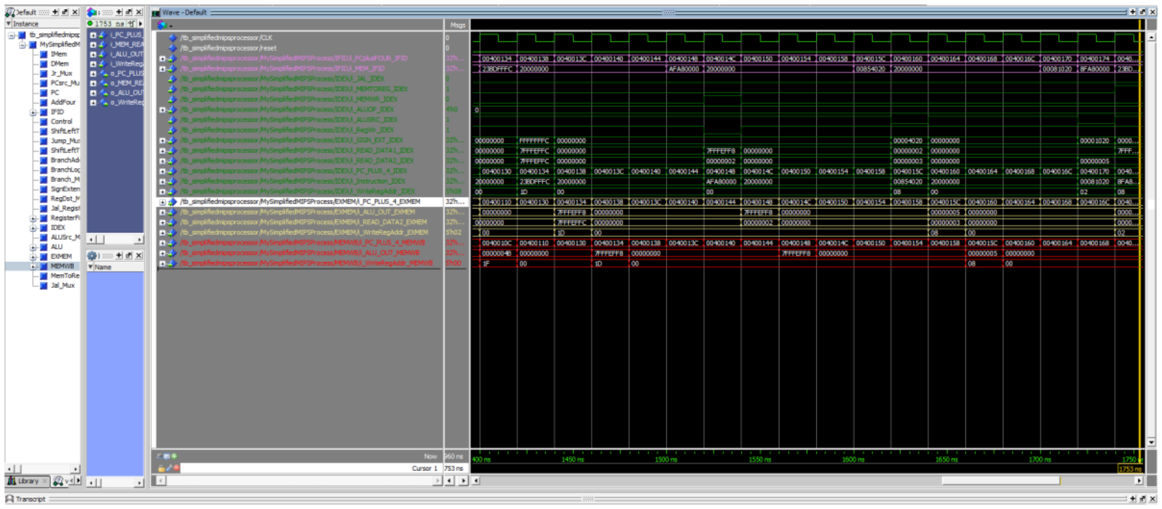
```
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_PC_PLUS_4_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_MEM_READ_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_ALU_OUT_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_WriteRegAddr_MEMWB
```

(1)

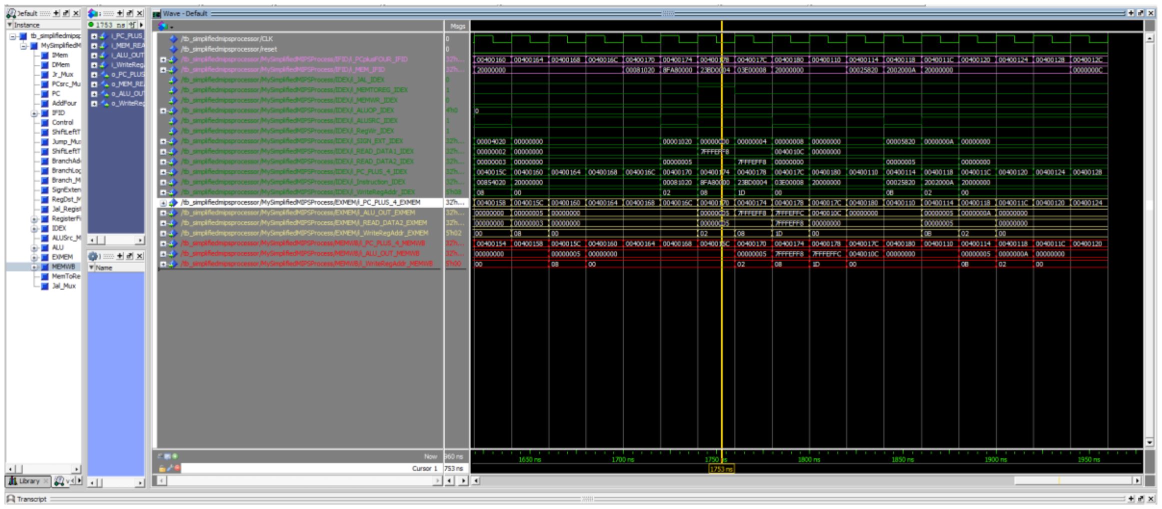

(2)


(3)


(4)



(5)



(6)

Explanation:

These sets of tests in `Proj-C_test1.s` do all of the instructions in Proj-B, but includes all the stalls necessary to show how handling stalls works with software, without any hardware data hazard detection. The first few instructions we needed to stall the instructions due to them being close together, they are supposed to be separated 4 clock cycles apart if we are to do any dependencies that include WAW and RAW.

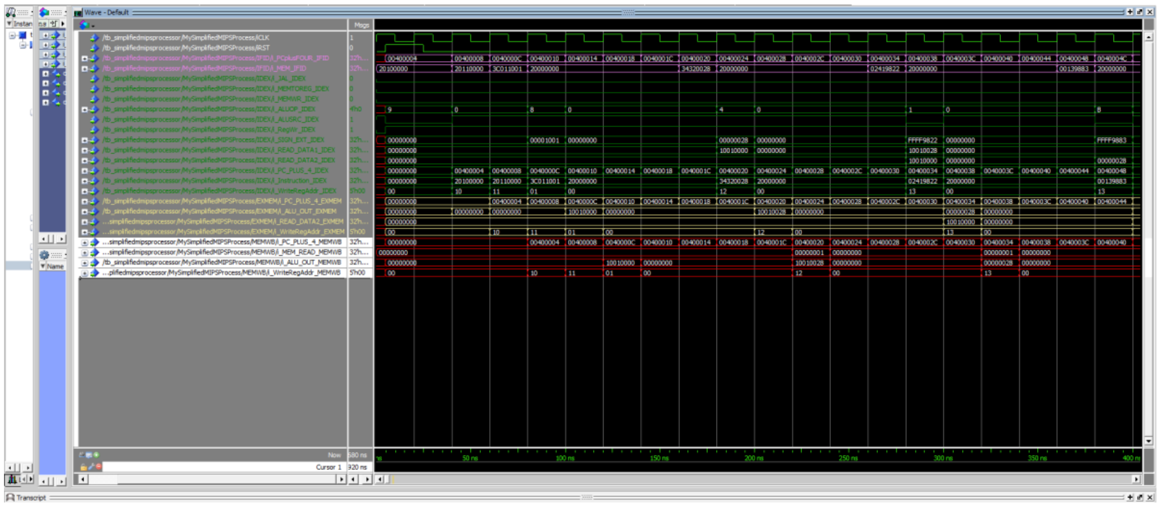
At the very first image, it shows the values being added into 1 and 2 and being set. Immediately after, we do add. This requires two registers values to be loaded in, not immediate. Since we have used registers from before, that weren't fully added in. We had to create NOP(empty instructions to create stalls) to halt the process for as many clock cycles until the values can be used next. It stalls 3 times with `addi $0, $0, 0`(we could have used sll, but this NOP is the same) until its able to do the add instruction.

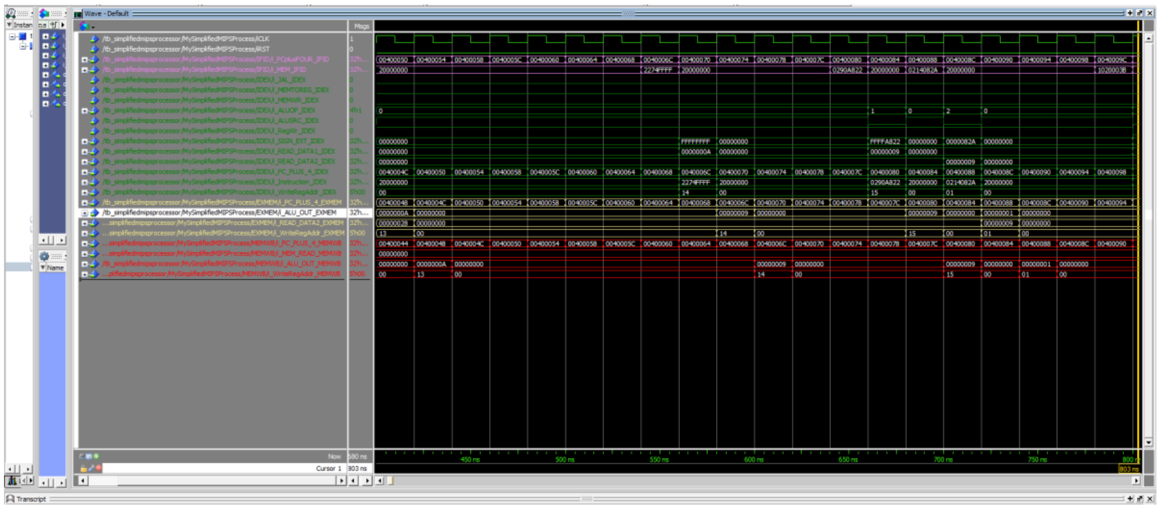
You can follow the i_regWriteAddr to see what values are going in at what times. If you see a 0 at any point, that is where the processor is doing a NOP to do the next instruction. Also, if you look at the PC+4 component, and keep following it to the bottom right. You can see its instructions following the the stages from IF-ID -> ID-EX -> EX-MEM and to MEM-WB.

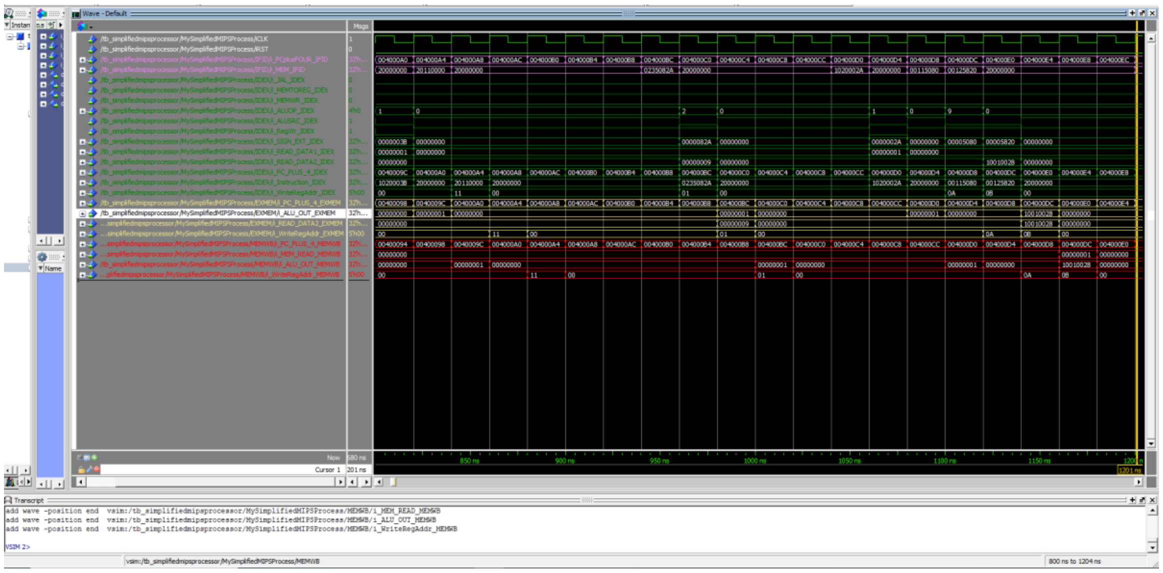
For jumps, and branches, we had instructions be stalled by one clock cycle in order for the register to get to that specific address in the pipelined register. Some examples of jump and branch can be found in images 5-6 up above.

f. [Part 3 (b)] In your writeup, show the ModelSim output for the modified Bubblesort test, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

Waveform:


(1)


(2)

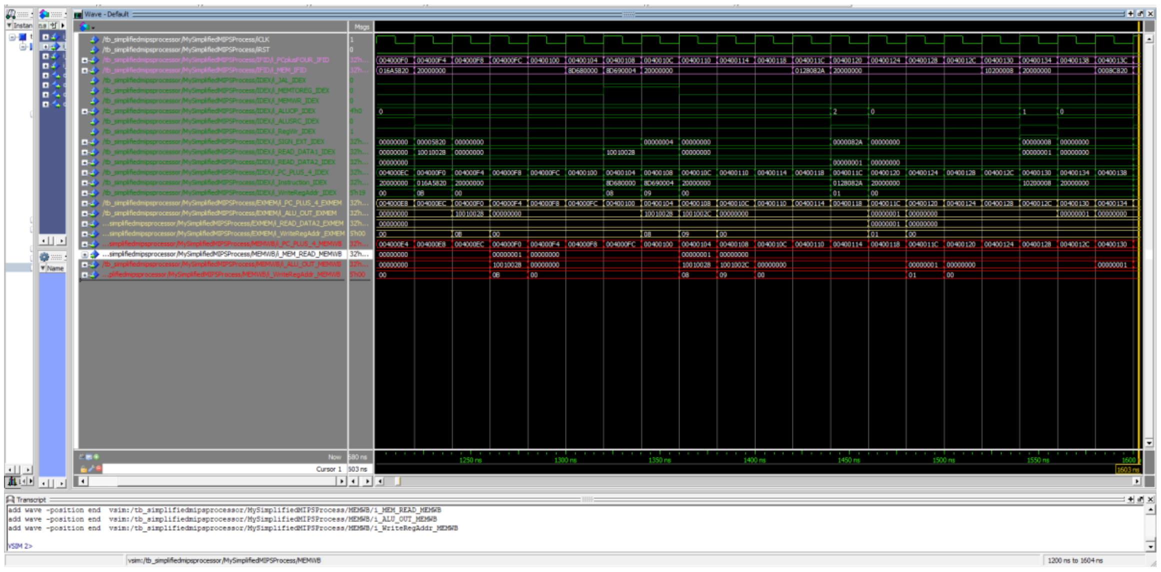


```
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_MEM_READ_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_ALU_OUT_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_WriteRegAddr_MEMWB

VSIM 2>
```

vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB

800 ns to 1204 ns

(3)

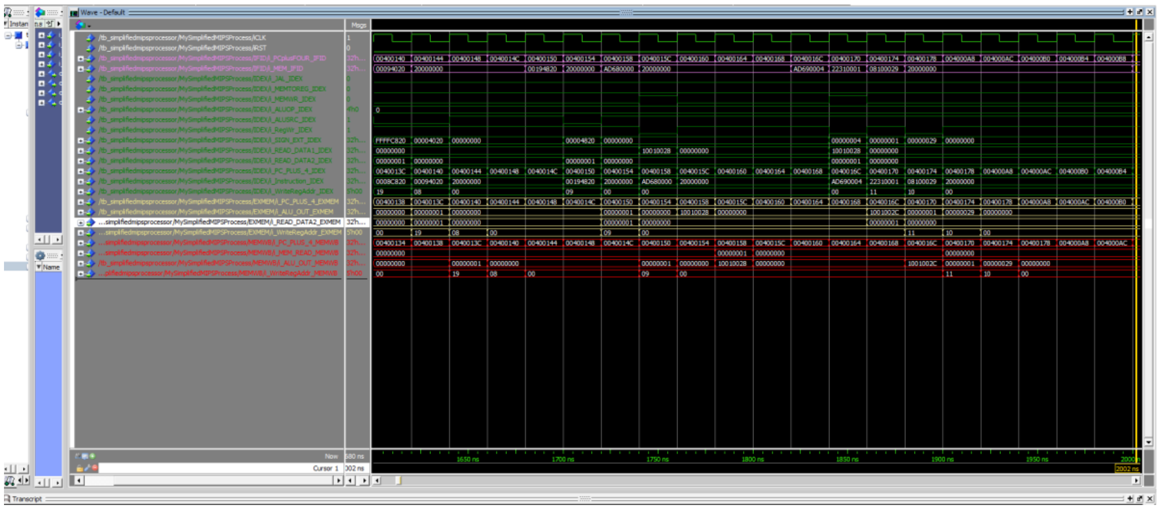


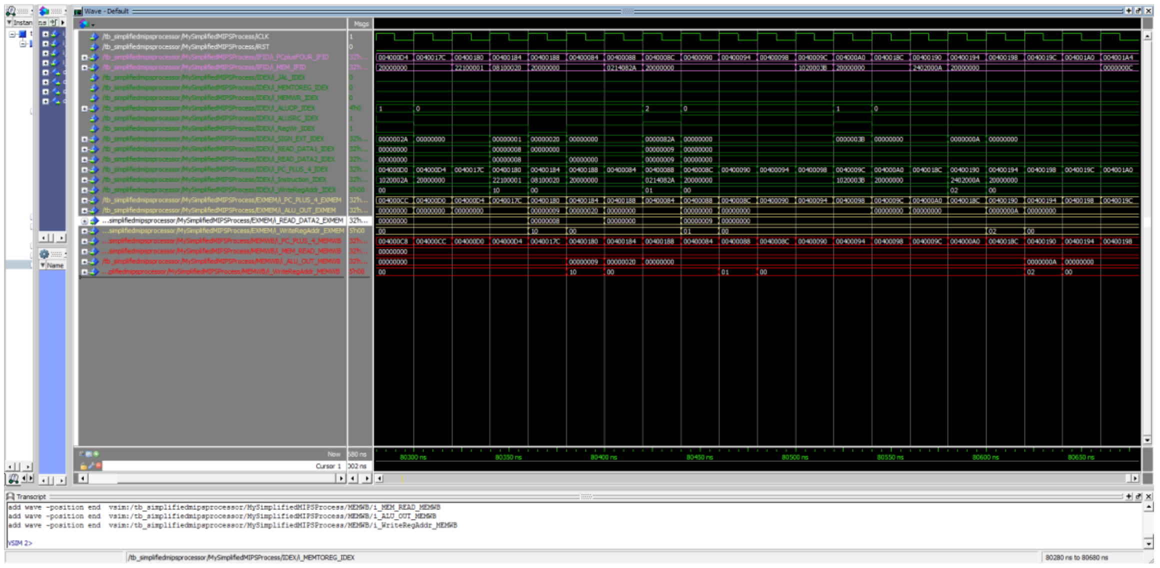
```
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_MEM_READ_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_ALU_OUT_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_WriteRegAddr_MEMWB

VSIM 2>
```

vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB

1200 ns to 1604 ns

(4)


(5)



```
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_MEM_READ_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_ALU_OUT_MEMWB
add wave -position end  vsim:/tb_simplifiedmipsprocessor/MySimplifiedMIPSProcess/MEMWB/i_WriteRegAddr_MEMWB

VSIM 2>
```

(6)

Explanation:

This set test can be located within Proj-C_test2.s. It follows the same guidelines as the first tests. We went through our old Proj-B_test3.s(bubble sort) and added all stalls where the instructions have data dependencies with RAW and WAW. This time, there are very many stalls that need to happen, and there was no way around ours instead of adding NOPs. That's why there is so many clock cycles. The program ran for more than 4,000 cycles, but it still has ordered the numbers we had from before from least to greatest, and greatest to least the other way around.

Similar to the first tests above. You can watch the i_regWriteAddr to see where the instructions are going to be stalled or written into.

Demo of our entire pipelined processor (due to proj-c time constraints):

DEMO:
The core of this pipeline processor is the same as the single cycle processor. Our group successfully completed ProjB and we feel very confident with our work in that project. Therefore, the only new thing that was added to this processor were the pipelines and multiplexors. We started by separating the processor into what we call stages. The group strictly followed the instructions given in class and written on the book to create this processor. Our register is, software scheduled which means that we depend on software to stall and flush our register. The group worried mainly on creating correct and reliable pipelines and implementing them accurately while paying attention to our MIPS code and only placing stalls when needed.
The processor was divided into five stages: IF, ID, EX, MEM and WB. Each stage was separated by a pipeline: IF/ID, ID/EX, EX/MEM and MEM/WB. A pipeline is something very similar to a register file. The group created each pipeline as a collection of register that would store the information for one clock cycle. Each pipeline would have the same amount of register as the amount of inputs. The size of each register also had to be taken into consideration.
The IF stage included a PC, Imem and PC+4 logic. During this phase we generate an instruction and pass it on to the pipeline. The crucial part of this phase is the PC+4 part. The group had to add a mux before the PC register (PCsrc) that would allow the PC+4 to be the new instruction if there were no jumps nor branches.
The ID stage included our Control Logic module, our Register File logic, the Sign Extension, the Jump Logic and the Branch logic. The control logic module gets the instruction from the pipeline and works with little to no modification. The Register File logic also works very similarly to the single cycle processor. The only difference is that the Write Register Address, Write Data and the Write Register Enable all need to be propagated up to the WB (write back) phase and then they will be finally evaluated by our processor. The group decided to write the entirety of the Branch and Jump logic in the ID phase, including the write back to the PC register. This way, all our branch and jump instructions would only have one clock cycle delay. We also added a comparator between both of the Register File outputs for the branch logic.
The EX stage has our ALU. The ALU logic takes both inputs from the pipeline (one of the inputs goes though the ALU Src mux to decide if we are using the Sign Extended value or the Register File data). The output of the ALU is sent to our next pipeline. The Zero output of our ALU is grounded because the group did not need to use it for the Branch Logic anymore.
The MEM stage contains our DMEM. The DMEM takes signals coming from the EX/MEM pipeline representing the ALU output and the second register read of our Register File.
The WB stage uses two multiplexors to decide the destination of our output. We first check if our information from memory is going to the register and then we check if we are doing a jump and link instructions. Those multiplexors are concatenated. It is also during this stage that we write our register Address, Register Data together with enabling Register Write.
It is crucial to point out that there were many control signals being passed from pipeline to pipeline just to be used at the correct stage of the processor. The group made sure that none of the signals were being forwarded by accident.

g. [Part 4] 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).

Max Frequency: 46.35 mhz

The critical path is from:

Starts out at id_ex:IDEX(Register_1Bit) -> IDEX(id_ex_ALUSrc_reg) -> ALUSrc_Mux(mux) -> ALU(g_ALU) -> ALU(g_BarrelShifter0) -> ALU(data_out) -> EXMEM(ex_mem_alu_OUT_IN) -> ex_mem(Register_Nbits)

(if-id, mem-wb don't show up due to the synthesizer getting rid of extra singals)

h. [Part 5 (a)] Of the MIPS instructions supported for Project Part B, list which instructions produce values, and what signals in the pipeline these correspond to.

Instructions that produce values and their corresponding signals in the pipeline:
srl => output of ALU -> EX/MEM pipeline
add => output of ALU -> EX/MEM pipeline
addiu => output of ALU -> EX/MEM pipeline
addu => output of ALU -> EX/MEM pipeline
sw => Register File -> ID/EX pipeline -> ALU output ->EX/MEM pipeline -> Dmem
addi => output of ALU -> EX/MEM pipeline
sub => output of ALU -> EX/MEM pipeline
subu => output of ALU -> EX/MEM pipeline
slti => output of ALU -> EX/MEM pipeline
sltiu => output of ALU -> EX/MEM pipeline
sltu => output of ALU -> EX/MEM pipeline
and => output of ALU -> EX/MEM pipeline
andi => output of ALU -> EX/MEM pipeline
or => output of ALU -> EX/MEM pipeline
ori => output of ALU -> EX/MEM pipeline
xor => output of ALU -> EX/MEM pipeline
xori => output of ALU -> EX/MEM pipeline
srav => output of ALU -> EX/MEM pipeline
nor => output of ALU -> EX/MEM pipeline
lui => output of register file -> output of ALU -> EX/MEM pipeline -> MEM/WB -> Register File
lw => Dmem -> MEM/WB -> Register File
sll => output of ALU -> EX/MEM pipeline
sra => output of ALU -> EX/MEM pipeline
sllv => output of ALU -> EX/MEM pipeline
srlv => output of ALU -> EX/MEM pipeline

i. [Part 5 (b)] List which of these same instructions consume values, and what signals in the pipeline these correspond to.

List of instructions that consume values:
- The add and addu instructions consume two signals in two different registers, or one register used twice by adding them together.
- The addi and addiu instruction consumes one signal from a register by adding it with an immediate.
- The and instruction consumes two signals in two different registers by and-ing them with each other, or one register used twice.
- The andi instruction consumes one signal in a register by and-ing it with an immediate.

- The beq and bne instructions consume two signals by using what is usually a value of a slt operation and another value like $0 to branch whether the signals are equal or not equal, based on whether beq or bne are used, respectively.
- The j, jr, and jal instructions consume one signal by using it as an address to jump to.
- The lui instruction consumes one signal in a register if it overwrites a value in the destination register used in the instruction.
- The lw instruction consumes one signal in a register if it overwrites a value in the destination register used in the instruction.
- The sw instruction consumes one signal in a register from moving the value to memory from the register.
- The nor instruction consumes two signals in two different registers by nor-ing them together, or one signal in one register that is used twice by the instruction.
- The or instruction consumes two signals in two different registers by or-ing them together, or one signal in a register that is used twice by the instruction.
- The ori instruction consumes one signal in a register by or-ing it to an immediate.
- The sll and sllv instructions consume one signal in a register by shifting it to the left by a given number of bits.
- The slt and sltu instructions consume two signals in two registers, or one signal used twice in the instruction, to find the difference in the value(s) to see if one is less than the other.
- The slti and sltiu instructions consume one signal in a register by finding the difference between that signal and an immediate to see if one is less than the other.
- The sra and srav instructions consume one signal in a register by taking that signal and shifting it right by a specified amount of bits arithmatically(and variably for srav).
- The srlv instruction consumes one signal in a register by taking that signal and shifting it right by a specified amount of bits logically.
- The sub and subu instructions consume two signals in two different registers, or one register used twice by subtracting them from each other.
- The xor instruction consumes two signals in two different registers, or one register used twice by xor-ing them together.
- The xori instruction consumes one signal in one register used with an immediate to xor them together.

j. [Part 5 (c)] Come up with a generalized list of potential data dependencies. From this generalized list, select those dependencies that will require forwarding (write down the corresponding pipeline stages that will be forwarding and receiving the data), and those dependencies that will require hazard stalls.

Potential data dependencies:

-Read After Write(RAW) Dependencies: Any Destination register read after it is written to (within 4 or 5 cycles?)

-Write After Read(WAR) Dependencies: Any register being written to in the same instruction that reads from it, or multiple instances of that type of instruction appearing consecutively

-Write After Write(WAW) Dependencies: Consecutive uses of the same destination register being written to or loaded into

-Read After Read(RAR) Dependencies: Consecutive reads from the same register that may be getting written to during those instructions

Pipeline Stages:

-Anything involving the ALU Output like arithmetic operations(add, sub, or, and, slt, xor, nor, addi, ori, andi, etc.) and shifts(sll, srav, sllv, etc.) goes into the EX/MEM pipeline register, so that will need to handle stalls, flushes and forwards.
-ID/EX Register handles the output of the calculation of PC+4, and since thats an arithmetic operation, stalls, flushes and forwards must be handled there.
-ID/EX also handles all RAR and RAW dependencies because the reads are made from the IMEM register.
-MEM/WB has to handle all WAR and WAW depencies because the data that will be written to IMEM is stored there, as well as EX/MEM for any writes going to DMEM.

k. [Part 6] Write a more generalized series of data forwarding and hazard detection logic equations based on the result from part 5).

NOT NEEDED, NO HARDWARE FORWARDING

l. [Part 7] Provide a high-level schematic drawing of the interconnection between components for the MIPS Hardware-scheduled pipelined Processor.

NOT NEEDED, NO HARDWARE FORWARDING

m. [Part 8 (a)] In your writeup, show the ModelSim output for the individual instruction tests and Bubblesort test (the original from Project Part B), and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

NOT NEEDED, NO HARDWARE FORWARDING

n. [Part 8 (b)] In your writeup, show the ModelSim output an application that attempts to exhaustively test the forwarding and detection logic in your pipeline, and provide a discussion of result correctness. It may be helpful to also annotate the waveforms directly.

Forwarding Detection Logic:
NOT NEEDED FOR THIS DEMO

o. [Part 10] Report the maximum frequency your processor can run at and determine what your critical path is. Draw this critical path on top of your top-level schematic. In your writeup, briefly discuss your critical path results. What components would you focus on to improve the frequency?

Maximum frequency:
NOT NEEDED FOR THIS DEMO

p. [Feedback] You must complete this section for your lab to be graded. Please complete each column **separately** for each team member; I expect it to take roughly 10 minutes (do not take more than 20 minutes).

i. How many hours did you spend on this lab?

Task	**During lab time**			**Outside of lab time**		
Team Initials	MS	TR	FC	MS	TR	FC

Reading lab	1	1	1	5	5	5
Pencil/paper design	2	3	3	0	1	5
VHDL design	2	2	3	10	15	20
Assembly coding	1	1	1	20	12	5
Simulation	1	1	1	5	5	5
Debugging	1	1	1	15	18	15
Report writing	0	0	0	2	2	2
Other:						
Total	8	9	10	57	58	57

ii. If you could change one thing about the lab experience, what would it be? Why?
Muhamed:

I would have loved to see a bit more examples of certain components we would use and a bit more explanation about them.

Trevor:

I would love to have in-person labs as I feel like it would be a much better way to work on labs with people, but I think that having something like the discord server did help me a lot more than a traditional lab might. For this lab I would have loved to have that in-person interaction with TA's, so hopefully coronavirus can go away and we can get back to having labs in person again.

Felipe:

If we were to do hardware scheduling, a sample of hazard detection units/forwarding unit as a skeleton would have been very helpful with the shortened schedule.

iii. What was the most interesting part of the lab?

Muhamed:

Making the slow processor from before become almost 4x as fast.

Trevor:

I loved the actual assembly coding of the bubblesort program, as I'm currently in Com S 228 and learning about sorting methods right now, so learning how to make this sorting method in another language was a cool way to apply that knowledge, especially when working in the CPR E 381 specific knowledge of stalls in the program.

Felipe:

Since we succeeded in completing Proj-B. It was great to see our Proj-C continue to work even after all the changes we added to it.