CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: __Camden Fergen__

__Emil Kosic__

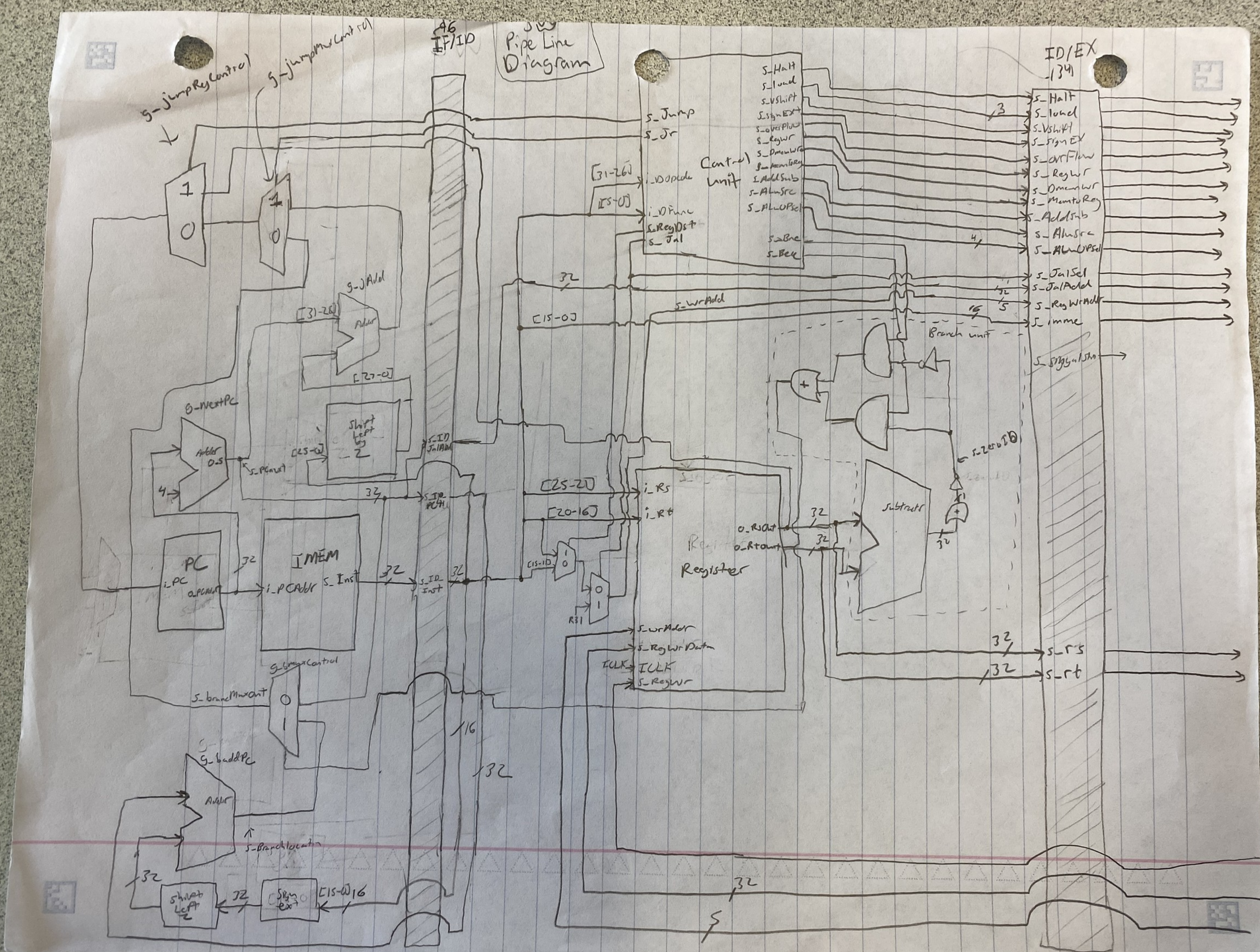
Project Teams Group #:__TermProj1_1_02__

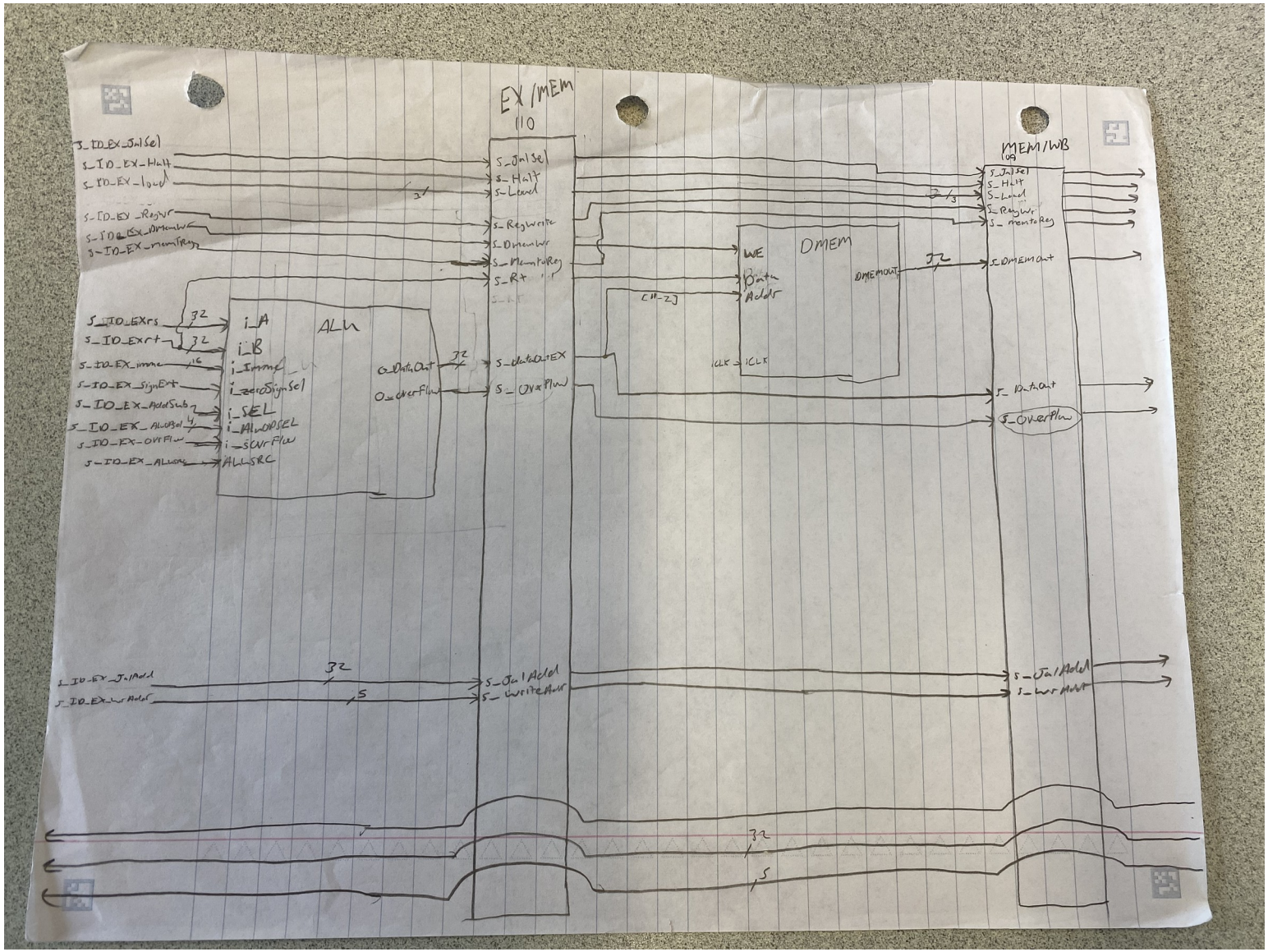
Refer to the highlighted language in the project 1 instruction for the context of the following questions.

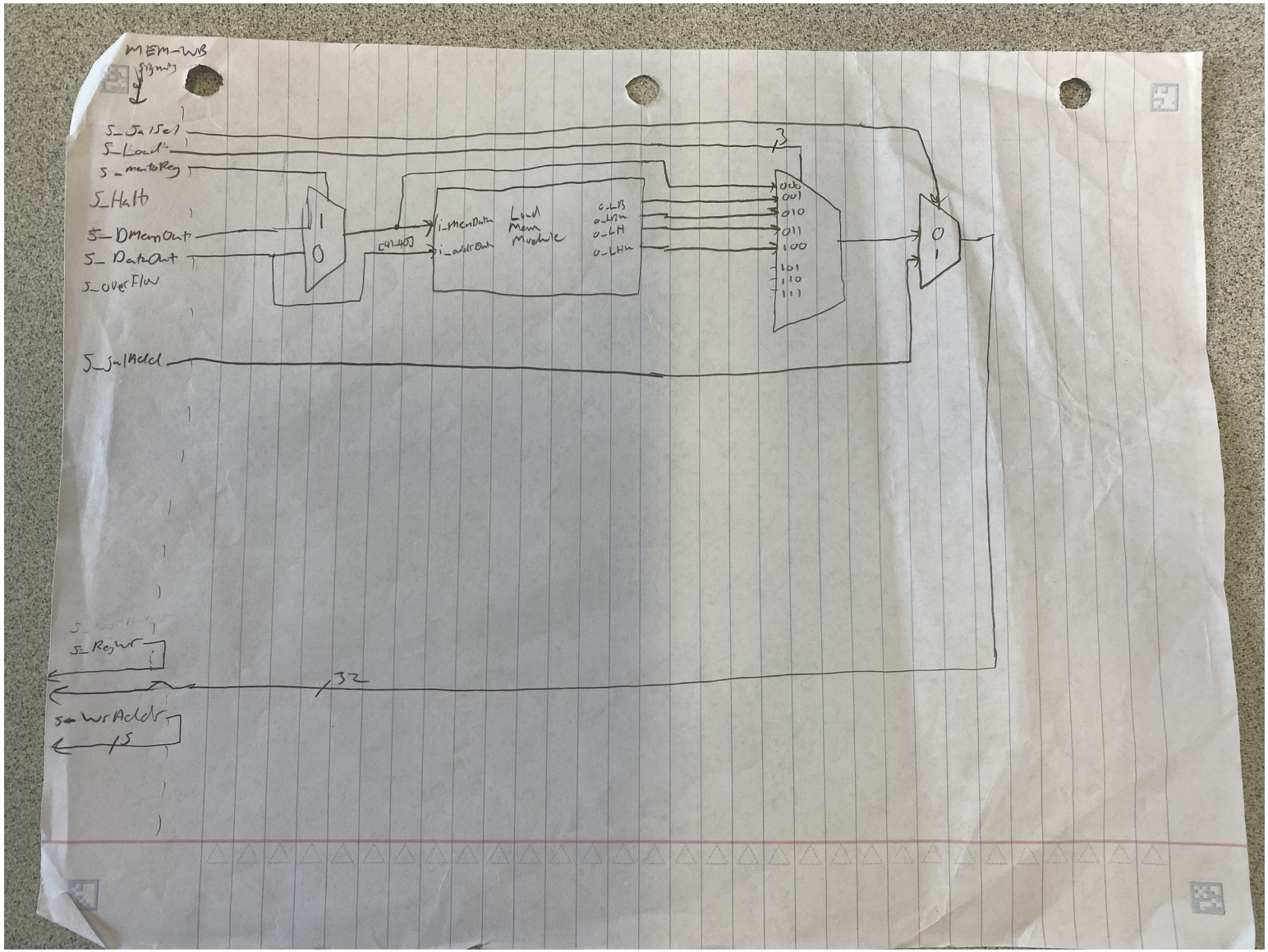
SW Pipeline

[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.
Included inside an excel sheet called “pipelineSignals”.

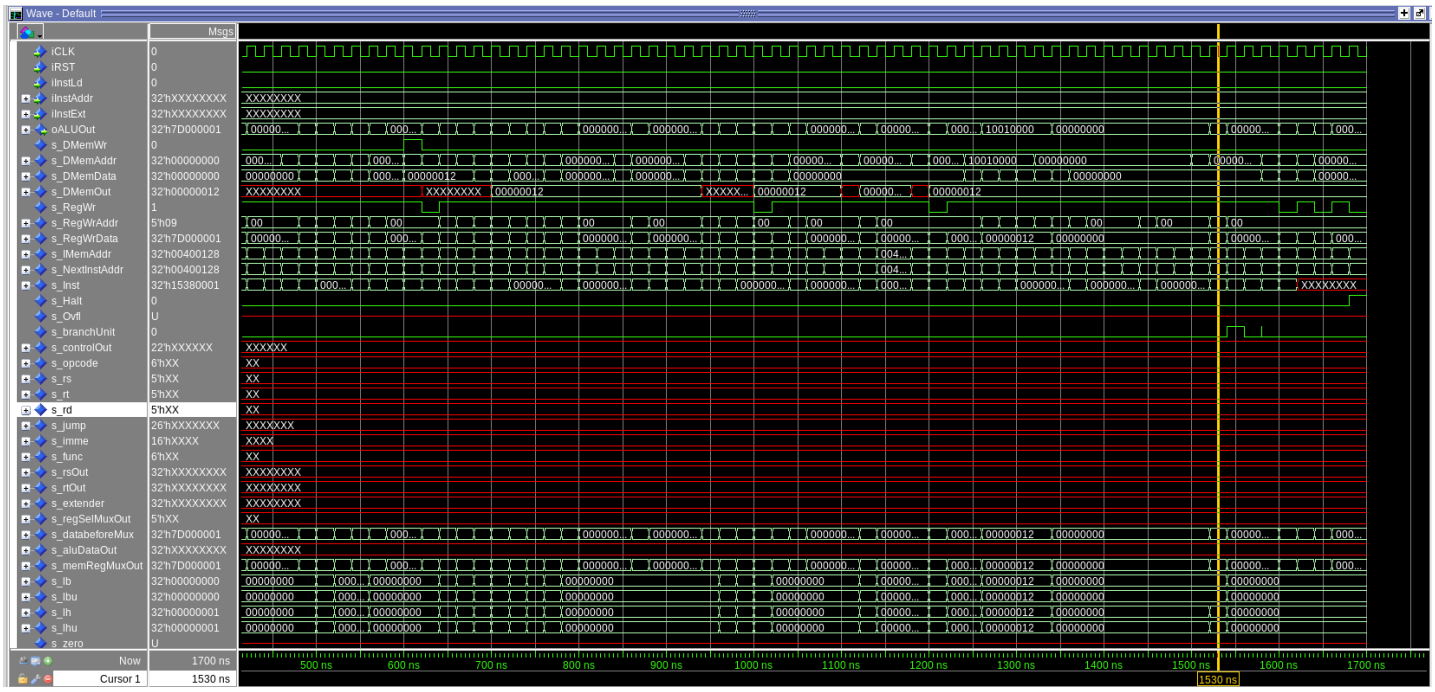
[1.b.ii] high-level schematic drawing of the interconnection between components.


Pipe Line Diagram
IF/ID
ID/EX
Control Unit
IMEM
PC
Register
Branch unit
Subtractor
Adder
Shift Left 2
Sign Ext
s_jumpRegControl
s_jumpMuxControl
s_NextPC
s_JAdd
s_branchMuxCont
s_addPC
s_BranchLocation
s_Jump
s_Jr
i_Opcode
i_Func
s_RegDst
s_Jal
s_Halt
s_load
s_Vshift
s_SignEXT
s_ovrFlow
s_RegWr
s_DmemWr
s_MemtoReg
s_AddSub
s_AluSrc
s_AluOPSel
s_Bne
s_Beq
s_JalSel
s_JalAdd
s_RegWrAddr
s_imme
s_rs
s_rt
i_Rs
i_Rt
o_Rsout
o_Rtout
s_wrAddr
s_RegWrData
iCLK
s_RegWr
s_wrAdd
s_zeroID
i_PC
o_PCout
i_PCAddr
s_Inst
s_ID_Inst
s_ID_PC+4
s_ID_JalAdd
s_PCout
[31-26]
[25-0]
[15-0]
[25-21]
[20-16]
[27-0]
[15-11]
R31


EX/MEM
MEM/WB
ALU
DMEM
WE
Data
Addr
DMEMOUT
iCLK
oCLK
i_A
i_B
i_Imm
i_zeroSignSel
i_SEL
i_ALUOPSEL
i_sOvrFlw
ALUSRC
o_DataOut
o_OverFlw
s_DataOutEX
s_OvrFlw
s_DMEMOut
s_DataOut
s_JalAdd
s_WriteAddr
s_RegWrite
s_DMemWr
s_MemtoReg
s_Halt
s_Load
s_JalSel

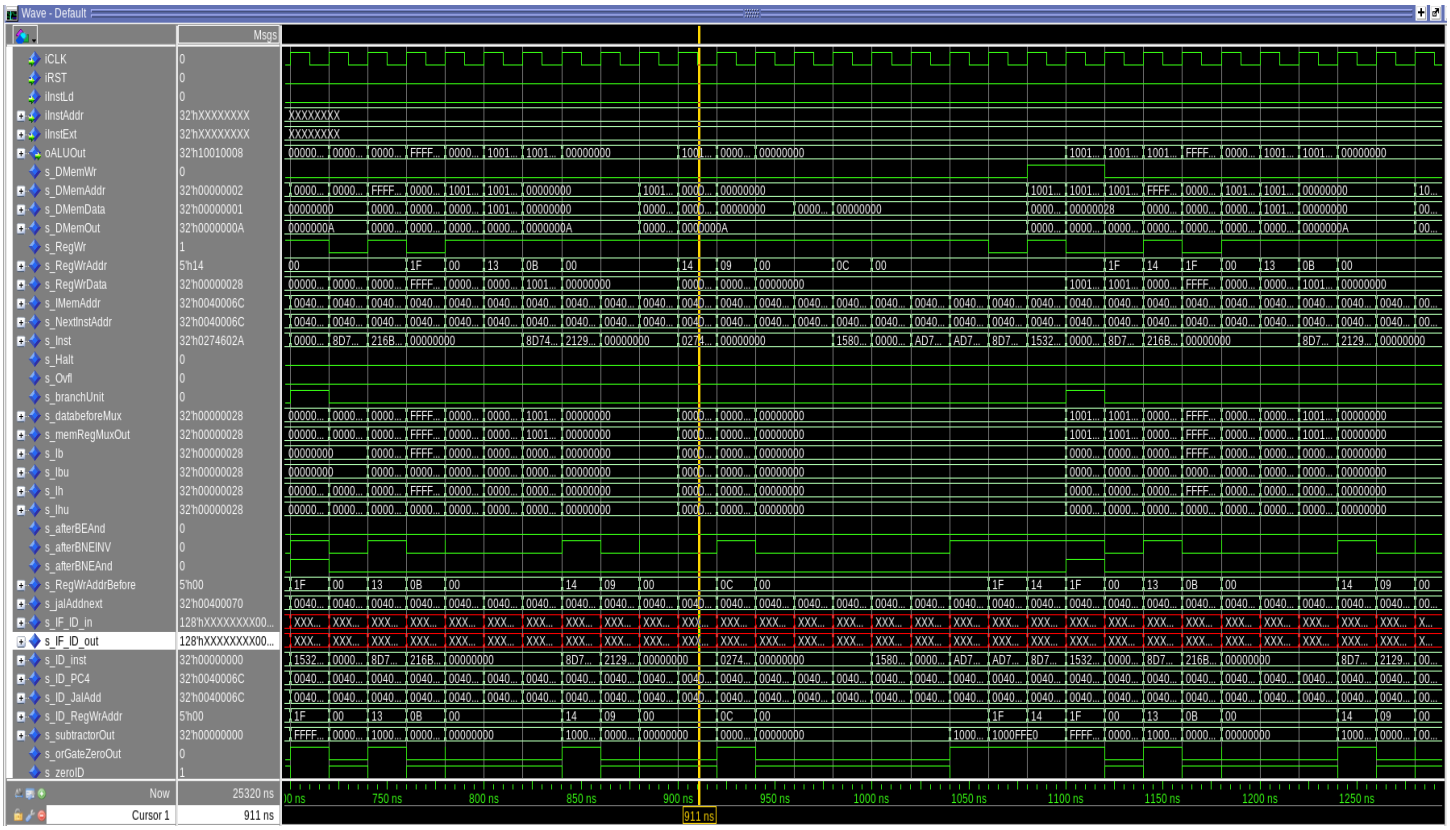

MEM-WB
Signals
S_JalSel
S_LoadU
S_memToReg
S_Halt
S_DMemOut
S_DataOut
S_OverFlw
S_JalAddr
1
0
[4:40]
i_memData
i_addrData
Load
Mem
Module
o_LB
o_LBu
o_LH
o_LHu
000
001
010
011
100
101
110
111
3
0
1
S_RegWr
32
S_WrAddr
5

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

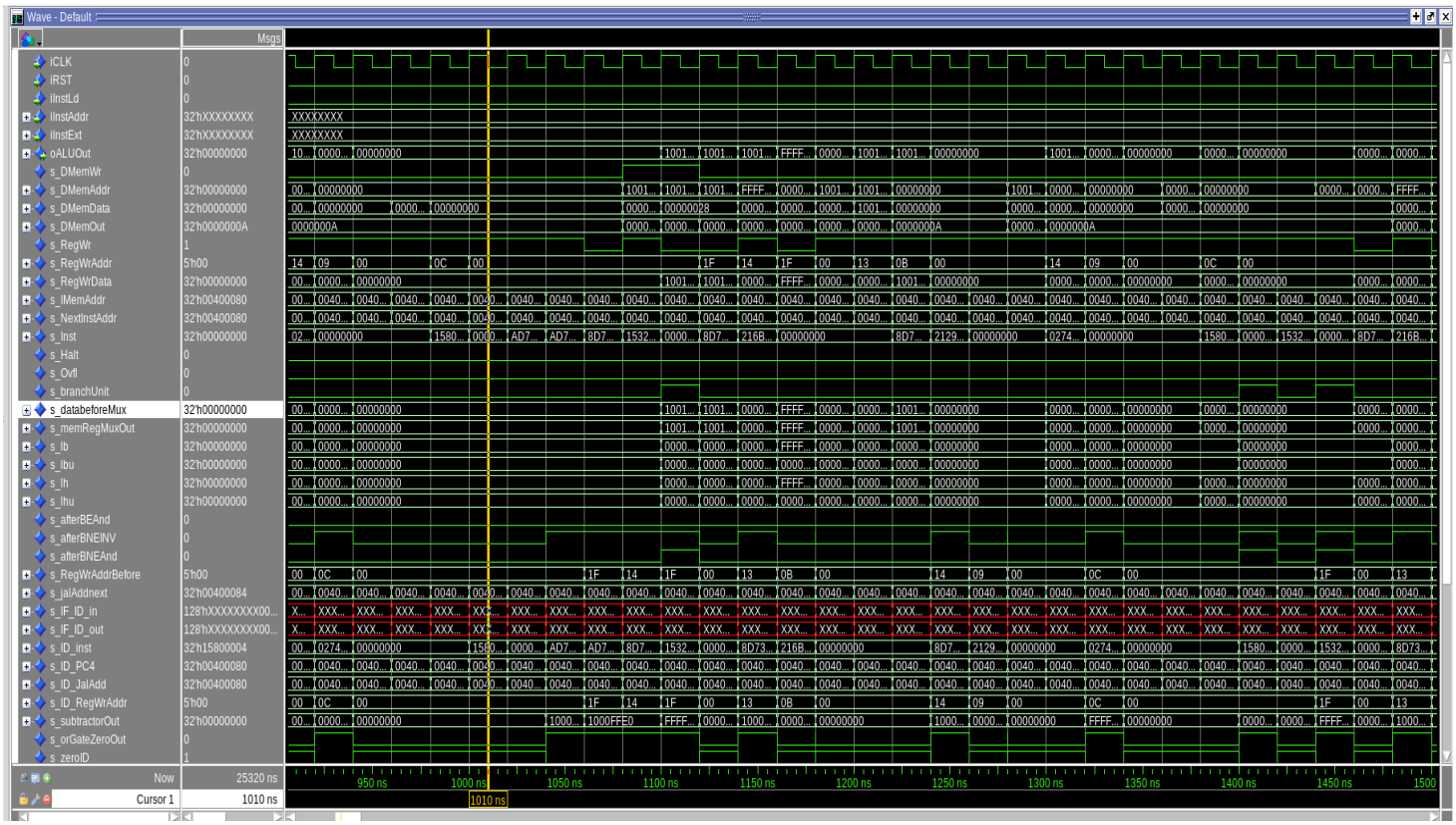


This waveform (very confusing with the amount of signals) provides evidence of the functionality of the sw pipelined processor. At this specific time frame we can see the signal "s_RegWrData" with the value of 0x7D000001. This value, confirmed with the output value on Mars shows the current data value being written into $9. The most important part is that one of the registers used in the addi ($8) was written to in a previous instruction, thus demonstrating the successful use of the sw pipeline that we created. This also occurred towards the end of the program, but also to summaries without going into to much detail we can see that the rest of the tests passed without any data hazards or problems at all.

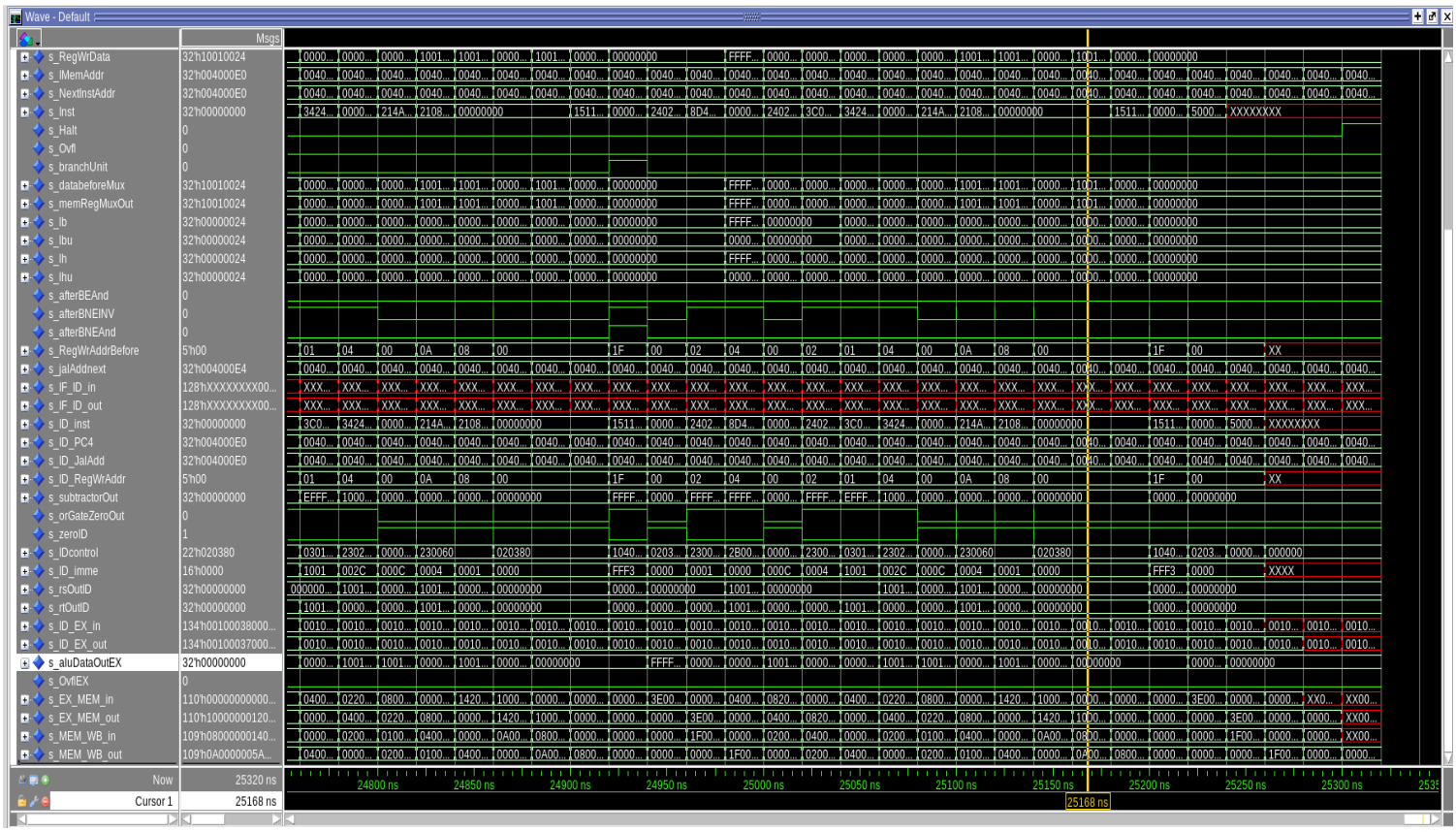
[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did not have to use the maximum number of NOPs.



This waveform is obtained from the output of "NOP_bubbleSort.s" and shows that we can take care of data dependencies without just throwing NOP, but running other instructions that are not dependent instead of doing a NOP. This specific cycle shows the output of a lw into $s4, which is used 4 cycles later. Instead of implementing 3 NOPs we have a an addi instruction that is the increment in the bubble sort, but this instruction does not have any immediate dependencies, so we can do this here.



This second example shows a possible control hazard that is avoided with only 1 NOP. Currently the waveform is on a BNE instruction of the program ("NOP_bubbleSort.s"), and shows that after this cycle NOP is used before the next instruction. Since the branch logic is in the decode stage only 1 NOP is needed to stall the pipelined processor so the correct jump address is received regardless if it is a jump or not.



This final waveform of ("NOP_bubbleSort.s") shows one of the final register writes of the program after many many cycles. Comparing these values with the mars.trace shows that this output is correct, and the only difference is that the clock cycle this occurs is way further than the mar.trace clock cycle, we can explain this because of the amount of NOPs required to run this program without any issues which is dependent on this implementation of the pipelined processor.

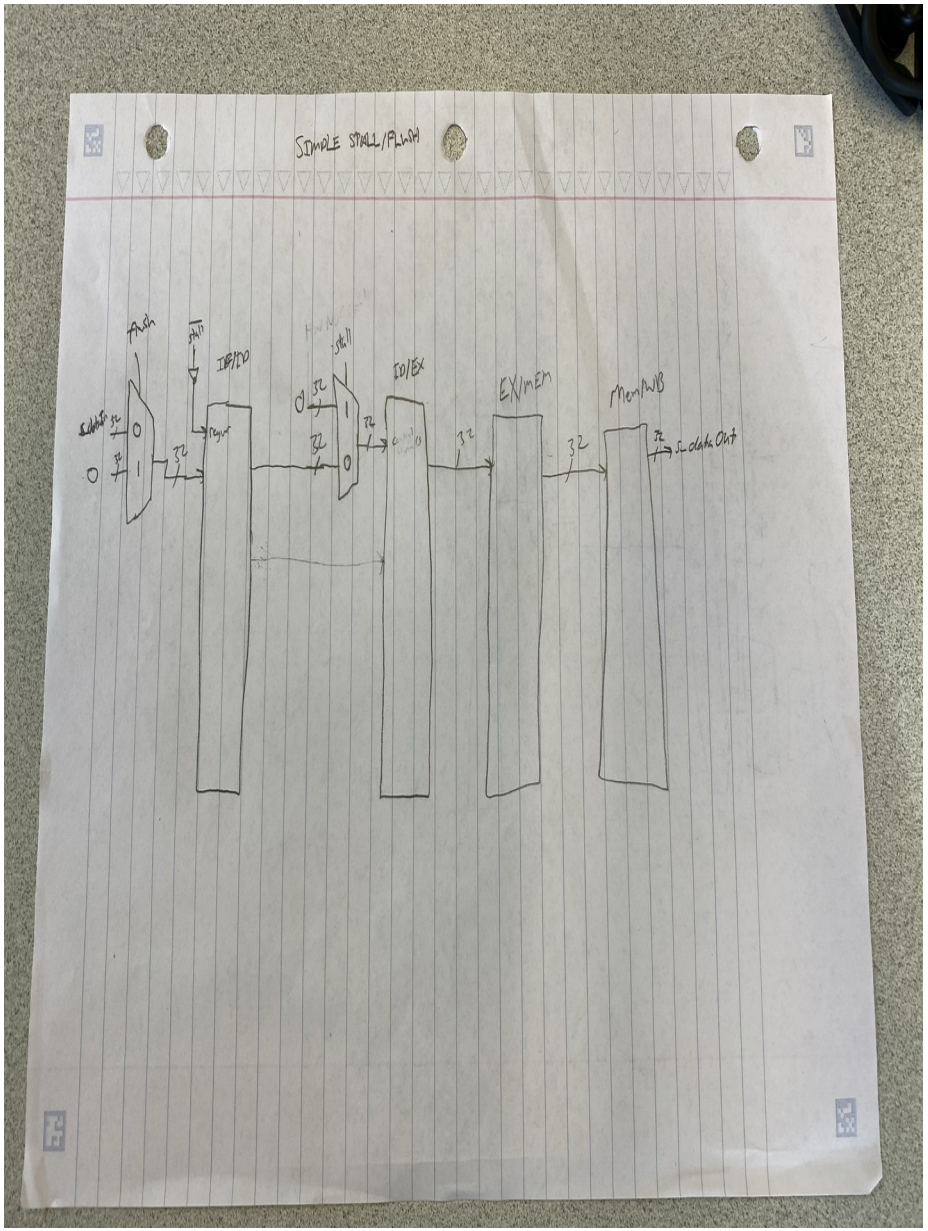
[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

The maximum frequency that the software scheduled pipeline was able to achieve was 47.88mhz, about twice as fast as the non pipelined processor. The critical path of this processor is as follows:

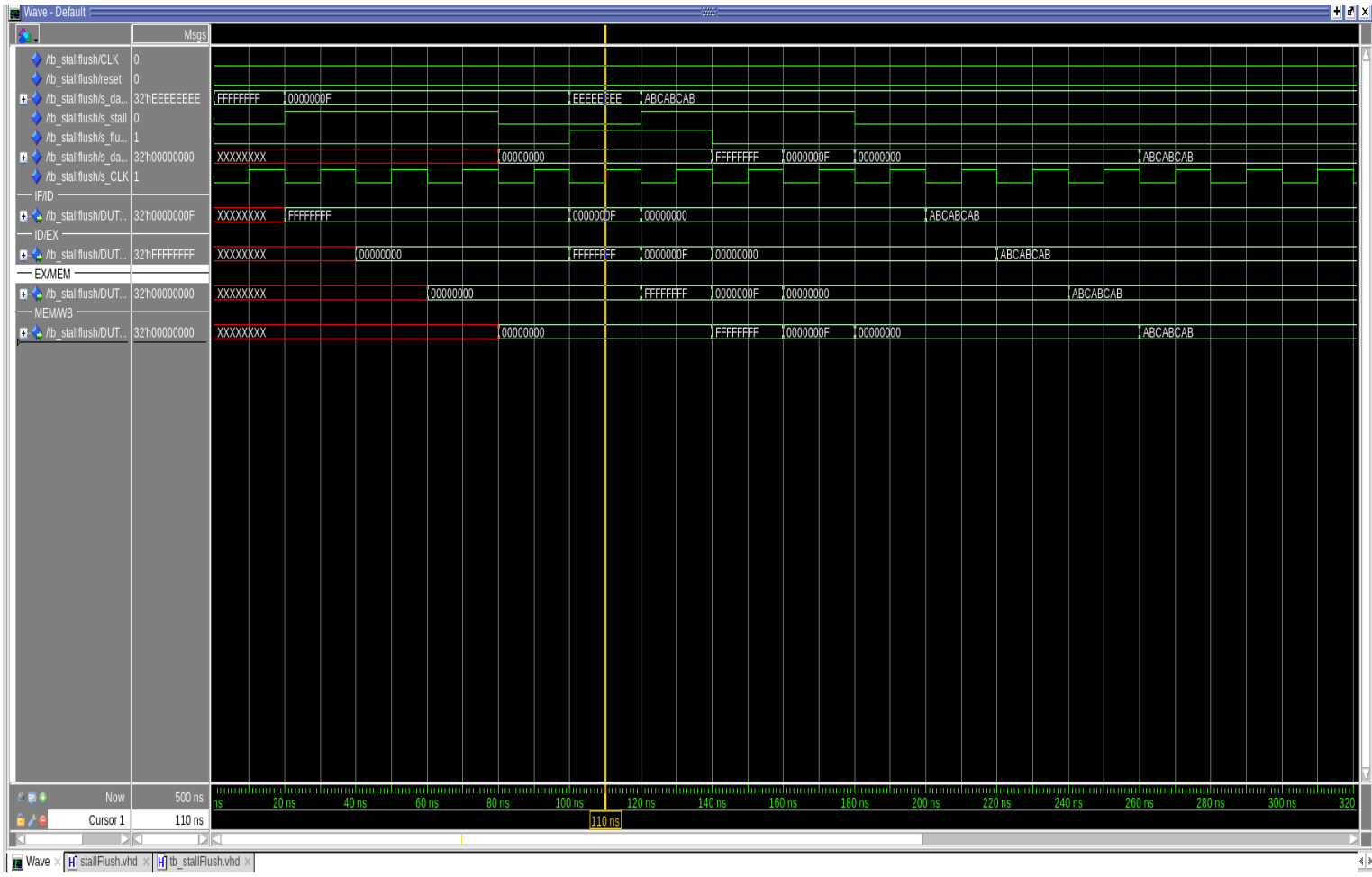
Imem → MipsReg → Zero unit (Just gets zero) → 32Bit Or gate → BranchOr (Determines if branching BE or BNE or neither) → Fetch unit (Jump reg mux) → PCReg

HW Pipeline

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


SIMPLE STALL/FLUSH
flush
stall
IF/ID
ID/EX
EX/MEM
MEM/WB
32
0
1
Register
To Data Out

[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 waveform shows the test bench for “stallFlush” which tests data going over multiple pipelined registers as well as flushing and stalling in combination and

separately. We can confirm that these in fact do work with how our pipelined processor is implemented (branching logic in decode stage). The tests itself are inside of the test bench of what should happen. One example of this test is stalling for 3 cycles with the value of "FFFFFFFF" in the decode stage, we can see this does in fact work as "FFFFFFFF" is inside of the IF/ID register for a total of 4 cycles (1 to load in and 3 stalls after).

[2.b.i] list which instructions produce values, and what signals (i.e., bus names) in the pipeline these correspond to.
Add, addi, addiu, addu, and, andi, lui, lw, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sw, sub, subu, jal, lb, lh, lbu, lhu, sllv, srlv, srav

add,addi,addiu,addu,sub,subu,and,andi,or,ori,xor,xori,nor,slt,slti,sll,srl,sra,sllv,srlv, srav produce values on the ALU Result signal in the processor in the execute stage

lw,lh,lhu,lb,lbu,sw,sh,sb produce/use values on the mem data signal in the processor in the final (memory) stage

beq,bne produce values on the branchLogic signal going into the fetch logic unit

j,jal,jr produce values on the JumpLogic and JregLogic singals going into the fetch logic unit with iJreg being used for jr only

[2.b.ii] List which of these same instructions consume values, and what signals in the pipeline these correspond to.
Add, addi. Addiu, addu, and, andi, nor, xor, lw, xori, or, ori, slt, slti, sll ,srl, sra, sw, sub, subu, lb, lbu, lbhu, sllv, srlv, srav, jr, bne, beq

add,addi,addiu,addu,sub,subu,and,andi,or,ori,xor,xori,nor,sll,srl,sra,sllv,srlv,srav all have signals on the s_rs and s_rt signals, with the I variants also using the imm signal

slt,slti,lw,lh,lhu,lb,lbu all consume signals on the s_rt signals into the Alu and then further into the mem stage for loading

sw,sh,sb all consume signals on s_rs and s_rt signals going into the mem stage

beq,bne both consume signals on the control signals IDControl as well as s_branchUnit

j,jal,jr consume signals on the s_rs and also IDControl signal from the pipeline register

[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.

We can detect a data hazard if the destination register in the MEM stage is matching either the rs or rt register in the ID stage. We can also detect another data hazard if the rs or rt value of the ID stage matches the register destination register in the EX stage. We also check if reg write is 1 and if this occurs while having matching registers AND if the destination register address is not 00000 then we can say there is a data hazard and do a stall. For a possible control hazard we can see if branching or any jump signal is on we will flush. Since our branching logic is in the decode stage we only need to flush in the ID stage.

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

Included in "pipelineSignalsHWschedule"

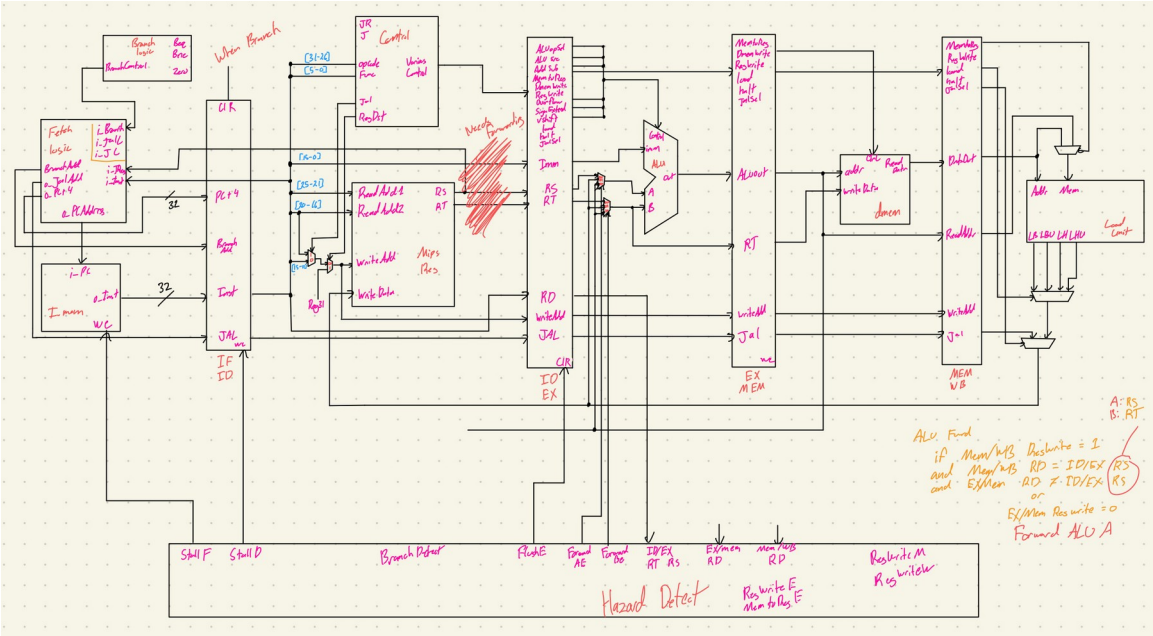
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

BNE, BEQ, J, JAL, JR and this all done during the decode stage of the pipeline.

[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.

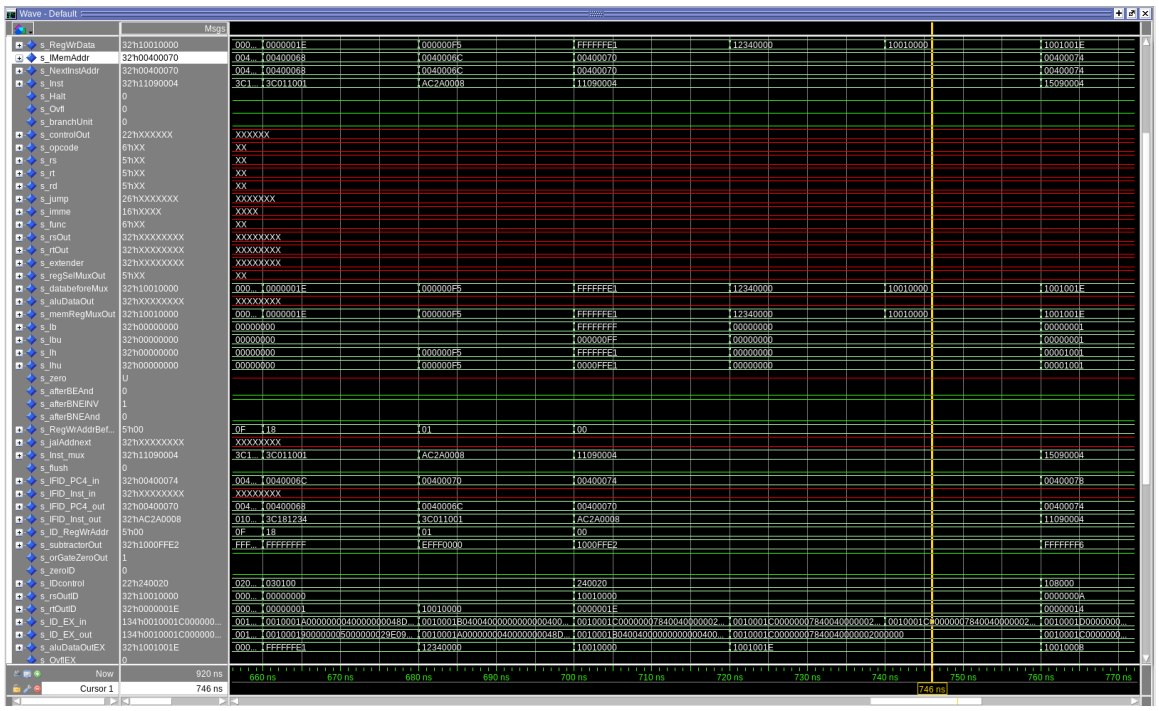
For each of these instructions we stall and flush at the same time for 1 cycle. Flushing occurs inside of the fetch stage and stalling occurs for the IF/ID pipeline register as well as the ID/EX register.

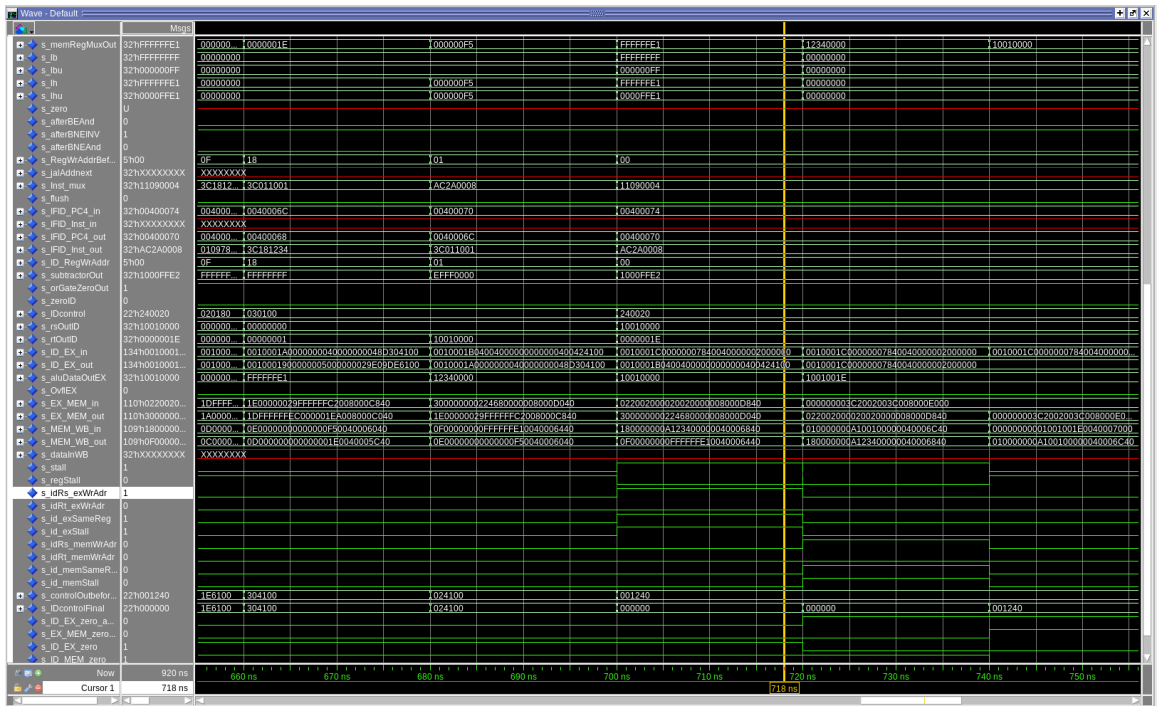
[2.d] implement the hardware-scheduled pipeline using only structural VHDL. As with the previous processors that you have implemented, start with a high-level schematic drawing of the interconnection between components.



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

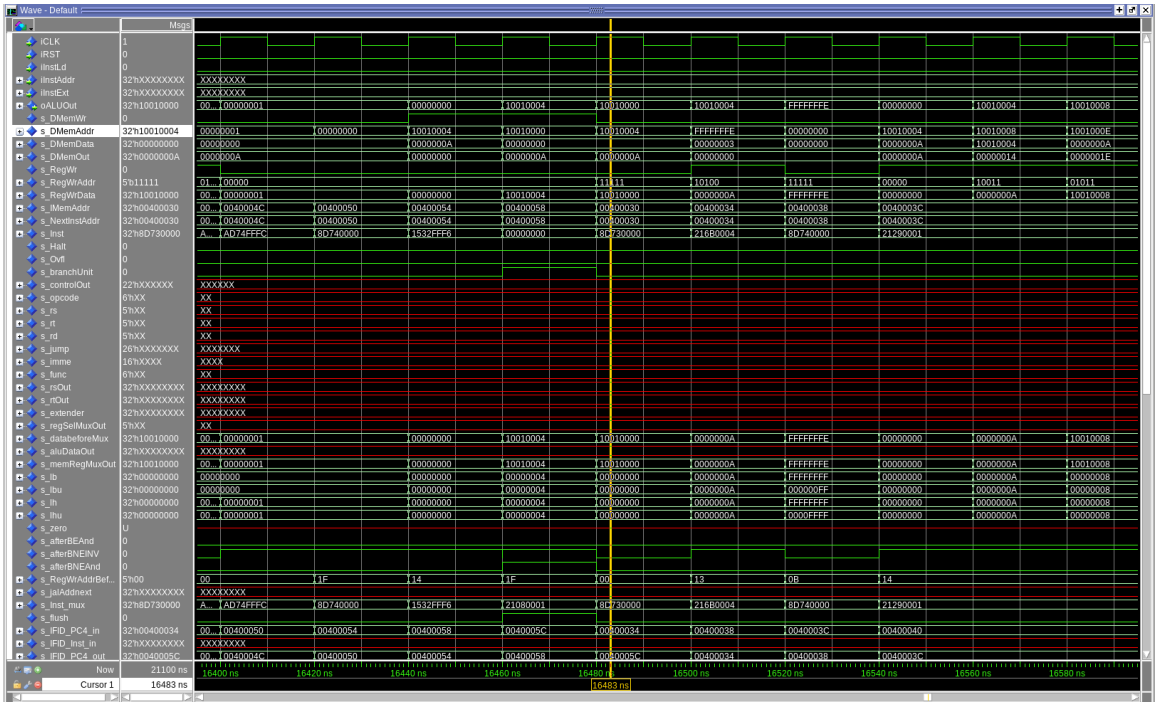
Base Tests:





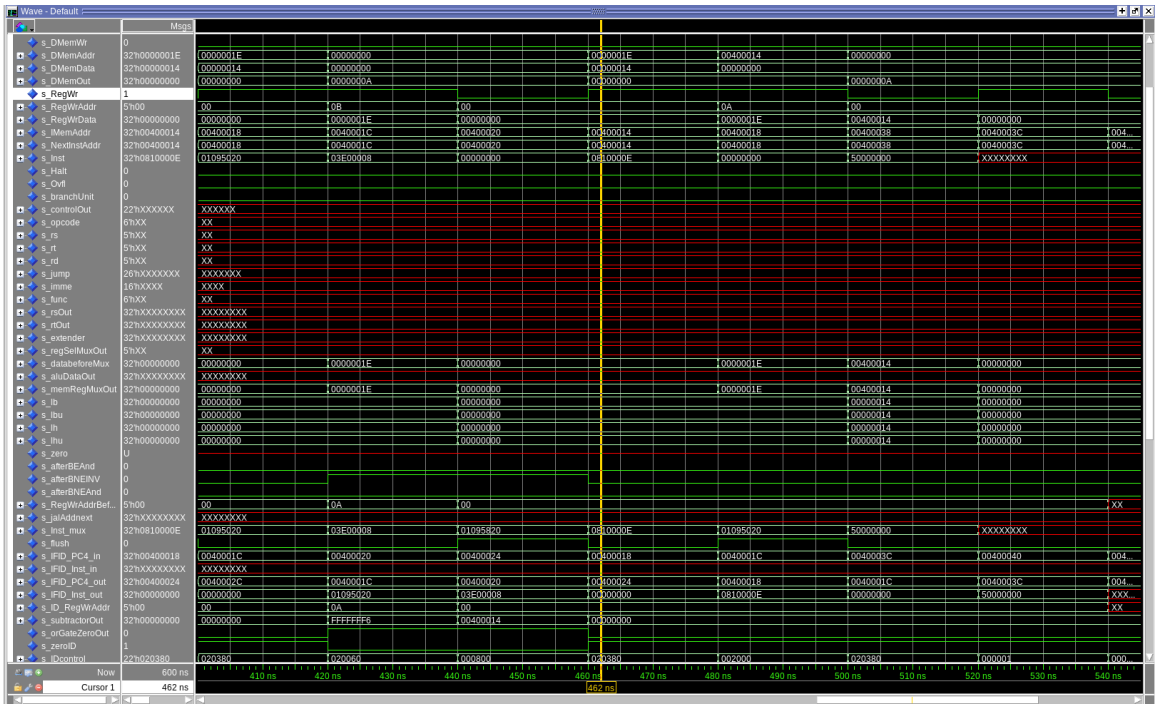
We can see from the waveform that the data correctly lines up with the mars trace writing value 0x10010000 to register 0x01 which you can see at the cursor line. You can also notice from the waveform that the current instruction is stalling due to a data dependency with the next instruction is trying to write to the same RS reg, and then after that is cleared its causing the same stall in the memstage to finish writing back the value

Bubblesort:



From the waveform you see that the data lines up with the expected output from the mars trace of saving word address 1001004 and writing the value 0x0000000A near the 16460ns mark and then writing 0x000000000 to reg 0x10010000 the next cycle. You can also see that the branch unit is working which is correct as right before the save words as indicated by the branch unit signal.

CF Tests:

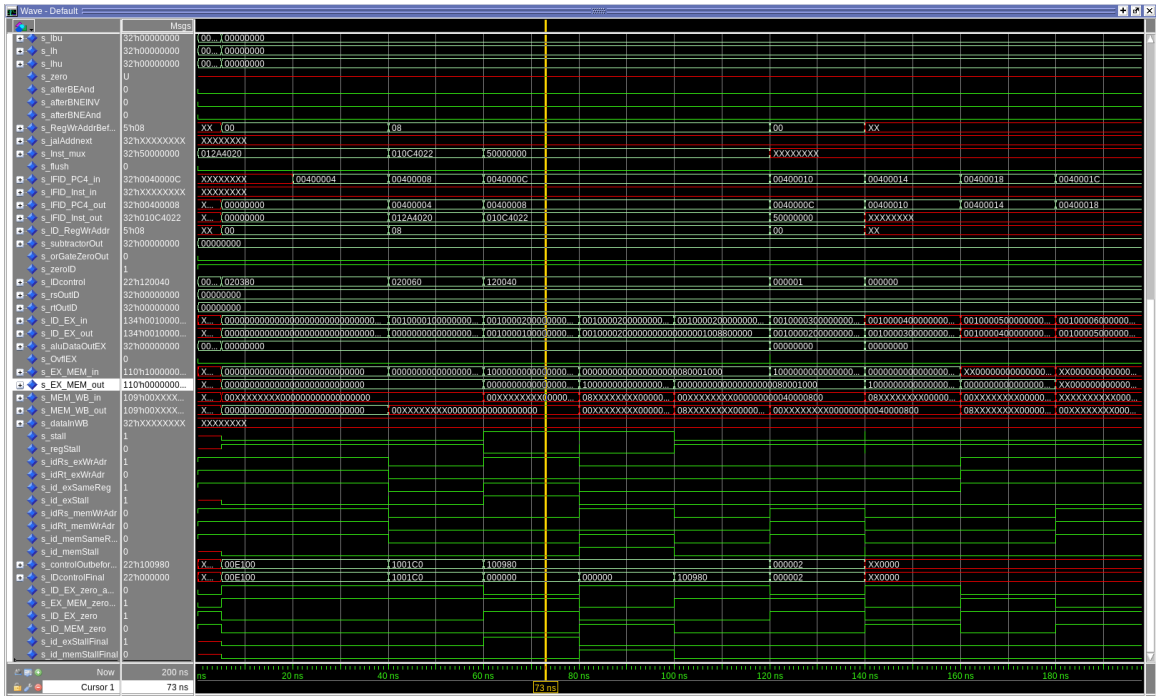


From this waveform we can see that in the next clock cycle starting 480ns it is correctly writing to the 0A reg which can be see from the mars.trace and matches the data being written of 0x0000001E. This proves the correctness of the program as the add instruction relies on the previous load instructions, and since its adding to the correct amount, the program is working as expected.

[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.

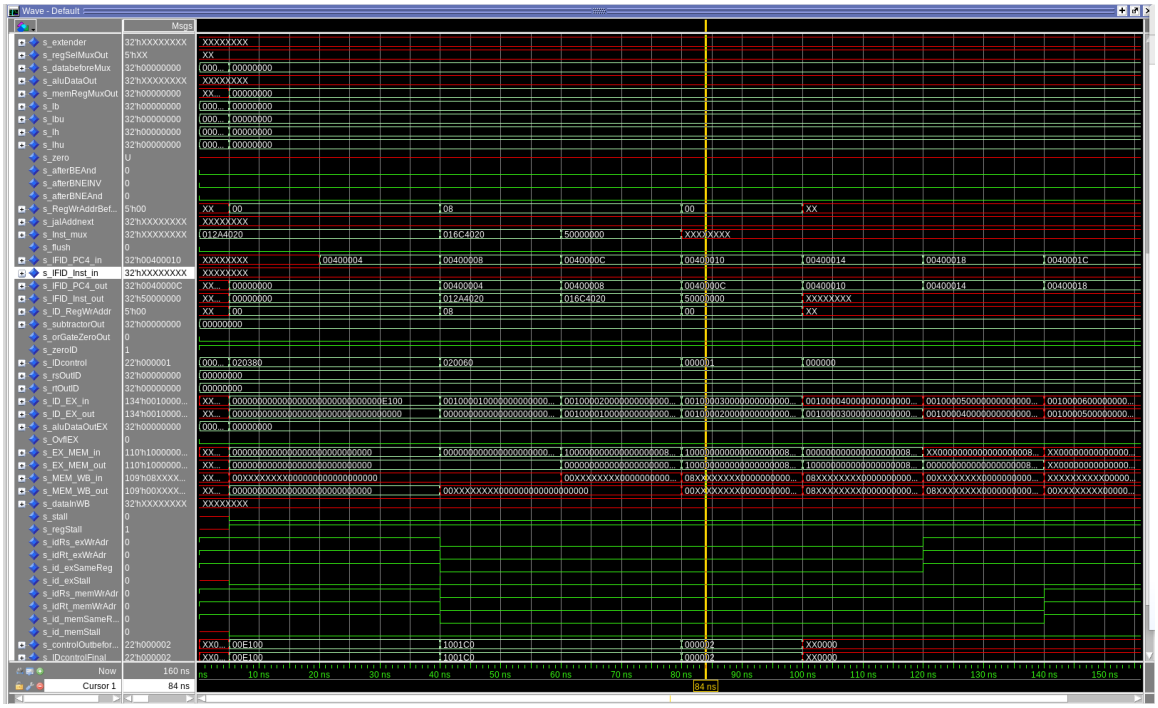
Case	Hazard	Justification
WAR (write after read)	WAR	Need for hazard detection, no forwarding
WAW (write after write)	WAW	Potential for hazards when writing to same register
Combined	WAR, WAW	Tests pipeline to handle WAR, and WAW at the same time

Can be found in WAR.s, WAW.s, and combined.s
WAR

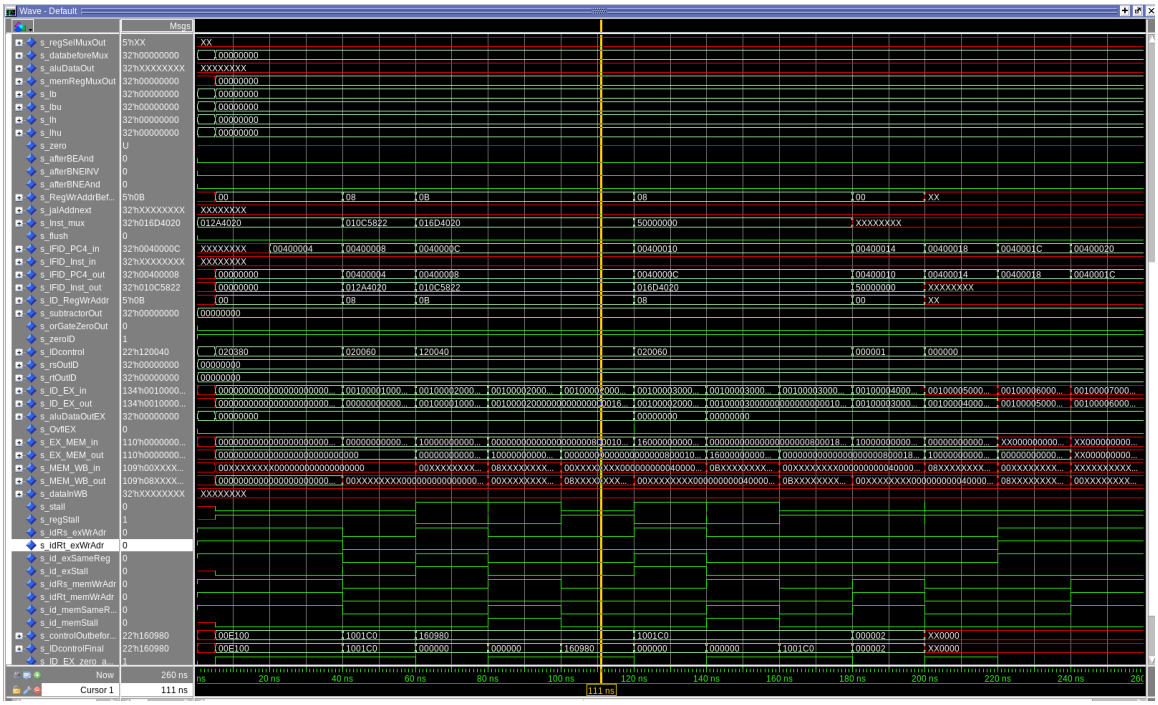


From the wave form you can see in the s_stall signal the processor is correctly stalling when it detects that the sub instruction is trying to read from the $t0 register which is being written to by the add instruction before.

WAW



From the waveform you can see the program is not stalling or causing any delays since the register is being written too right after its also being written too, which would get rid of the previous value. This helps increase the speed of the processor.



Combined

From this waveform you can see the processor is correctly stalling and avoiding the WAW hazards without causing any unneeded stalling

[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.

Num	Instruction	Justification
1	Beq	Test branch hazards
2	Bne	Test branch hazards
3	J	Test jump hazards
4	Jal	Test jal hazards
5	Jr	Test jump hazards
6	Beq,bne	Test sequential branch
7	Beq, jal	Test branch then jump and link

This test cases cover all of the control flow examples that would alter where the PC is and modify the PC. It first starts by testing the individual control flow instructions beq, bne, j, jal, jr. Each one is a simple program that ensures that the instruction is working as expected and isnt skipping instructions. It then goes to test the control flow instructions in sequence to ensure there is no hazard that is presented that causes the program to crash or output unexpected data. Through these cases it will cover all possible control flow hazards that present problems to the processor

[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).

The maximum frequency achieved was 39.52mhz. This speed is slower than the software scheduled pipeline by about 8mhz. The critical path of this processor is as follows: MipsReg → Inside of Reg for MipsReg → Subtractor (for getting 0) → RippleAdder Inside of Subtractor → Nbit mux inside of rippleAddr → OrGate inside of Nbit Mux → 32bit OrGate → Fetch unit jumpRegControl mux → PC reg