CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Kai Heng Gan

Brian McCreary

Project Teams Group #: TermProj 2 04

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

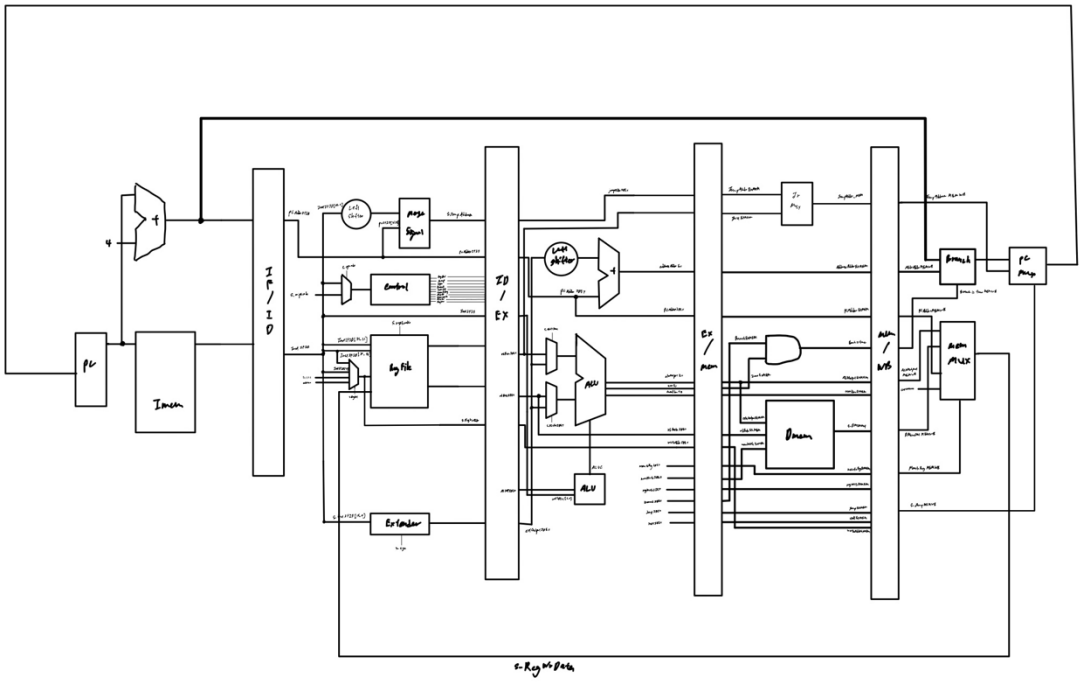
[1.a] Come up with a global list of the datapath values and control signals that are required during each pipeline stage.

```
-- Required register file signals
signal s_RegWr       : std_logic; -- TODO: use this signal as the final active high write enable input to the register file
signal s_RegWrAddr   : std_logic_vector(4 downto 0); -- TODO: use this signal as the final destination register address input
signal s_RegWrData   : std_logic_vector(N-1 downto 0); -- TODO: use this signal as the final data memory data input
signal s_RegRdData0  : std_logic_vector(N-1 downto 0);
signal s_RegRdData1  : std_logic_vector(N-1 downto 0);

-- Required instruction memory signals
signal s_IMemAddr    : std_logic_vector(N-1 downto 0); -- Do not assign this signal, assign to s_NextInstAddr instead
signal s_NextInstAddr : std_logic_vector(N-1 downto 0); -- TODO: use this signal as your intended final instruction memory address input.
signal s_Inst        : std_logic_vector(N-1 downto 0); -- TODO: use this signal as the instruction signal

-- Required halt signal -- for simulation
signal s_Halt        : std_logic;  -- TODO: this signal indicates to the simulation that intended program execution has completed. (Opcode: 01 0100)

-- Required overflow signal -- for overflow exception detection
signal s_Ovfl        : std_logic;  -- TODO: this signal indicates an overflow exception would have been initiated

-- Control signals
signal s_jump,s_finalJump, s_branch, s_memRead, s_aluSrc, s_sign, s_halt_ID : std_logic;
signal s_DMemWr_ID, s_RegWr_ID : std_logic;
signal s_regDst, s_memToReg : std_logic_vector(1 downto 0);
signal aluOp :std_logic_vector(3 downto 0);

-- Sign Extender
signal s_ExtOut : std_logic_vector(31 downto 0);

-- ALU signals
signal s_ALUIn1 : std_logic_vector(31 downto 0);
signal s_Zero : std_logic;
signal s_Cout : std_logic;
signal s_ALUResult : std_logic_vector(31 downto 0);

-- ALU Control signals
signal aluC : std_logic_vector(4 downto 0);

-- PC Adder signals
signal s_pcAdderOutput : std_logic_vector(31 downto 0);
signal s_pcAdderCout : std_logic;

-- firstLeftShifter signals
signal s_firstLeftShifterOuptut : std_logic_vector(27 downto 0);
```

```
-- secondLeftShifter signals
signal s_secondLeftShifterOuptut : std_logic_vector(31 downto 0);

-- g_Merge signals
signal s_jumpAddress : std_logic_vector(31 downto 0);

-- addressAdder signal
signal s_addressAdderOutput : std_logic_vector(31 downto 0);
signal s_addressAdderCout : std_logic;

-- g_and signal
signal s_andOutput : std_logic;

-- BranchMux signal
signal s_branchMuxOutput : std_logic_vector(31 downto 0);

--pcMUX
signal s_pcMuxOutput : std_logic_vector(31 downto 0);

--pcAdder_jal
signal s_pcAdderjalOutput : std_logic_vector(31 downto 0);
signal s_pcAdderjalCout : std_logic;

--MemMux
signal s_MemMuxOutput : std_logic_vector(31 downto 0);

--shifterMux
signal s_shifterMuxOutput : std_logic_vector(31 downto 0);
signal s_shiftInst : std_logic := '0';

--JrMux
signal s_jrMuxOutput : std_logic_vector(31 downto 0);
signal s_jumpReg : std_logic := '0';

--controlLogicMux
signal s_controlLogicMuxOutput : std_logic_vector(5 downto 0);
signal s_opcode : std_logic;

-- bgezal, bltzal, bltz
signal s_mipsOpcode : std_logic_vector(5 downto 0);

--alDataWriteBackMux
signal s_alDataWriteBackMuxOutput : std_logic;
signal s_regEnable : std_logic;
```

```
-- g_PC
signal s_pcEnable : std_logic;

-- PipelineIFID
signal s_pcAdderOutputIFID : std_logic_vector(31 downto 0);
signal s_instIFID : std_logic_vector(31 downto 0);


-- RegFile
signal s_rsData, s_rtData : std_logic_vector(31 downto 0);


-- WMUX
signal s_RegWrAddr_ID : std_logic_vector(4 downto 0);

-- PipelineIDEX
signal s_instIDEX,s_pcAdderOutputOutIDEX, s_jumpAddressOutIDEX, s_rsDataOutIDEX, s_rtDataOutIDEX, s_extOutputOutIDEX : std_logic_vector(31 downto 0);
signal s_wrtieAddrOutIDEX : std_logic_vector(4 downto 0);
signal s_ALUOpOutIDEX : std_logic_vector(3 downto 0);
signal s_regDstOutIDEX, s_memtoRegOutIDEX : std_logic_vector(1 downto 0);
signal s_jumpOutIDEX, s_signOutIDEX, s_branchOutIDEX, s_haltOutIDEX, s_memWriteOutIDEX, s_ALUSrcOutIDEX, s_regWriteOutIDEX : std_logic;

-- JrMux
signal s_jumpAddress_EX : std_logic_vector(31 downto 0);

-- addressAdder
signal s_addressAdderOutput_EX : std_logic_vector(31 downto 0);

-- ALU
signal s_aluOutput_EX : std_logic_vector(31 downto 0);
signal s_aluCout_EX, s_overflow_EX, s_zero_EX : std_logic;

-- PipelineEXMEM

signal s_pcAdderOutputOutEXMEM, s_jumpAddrOutEXMEM, s_branchAddrOutEXMEM, s_rtDataOutEXMEM, s_aluOutputOutEXMEM,s_instEXMEM  : std_logic_vector(31 downto 0);
signal s_writeAddrOutEXMEM, s_RegWrAddr_EX : std_logic_vector(4 downto 0);
signal s_memtoRegOutEXMEM : std_logic_vector(1 downto 0);
signal s_zeroOutEXMEM,s_overflowOutEXMEM,s_memWriteOutEXMEM,s_regWriteOutEXMEM,s_branchOutEXMEM,s_jumpOutEXMEM,s_haltOutEXMEM, s_RegWr_MEM : std_logic;

-- PipelineMEMWB
signal s_pcAdderOutputOutMEMWB, s_jumpAddrOutMEMWB, s_branchAddrOutMEMWB, s_memReadDataOutMEMWB, s_aluOutputOutMEMWB  : std_logic_vector(31 downto 0);
signal s_writeAddrOutMEMWB : std_logic_vector(4 downto 0);
signal s_memtoRegOutMEMWB : std_logic_vector(1 downto 0);
signal s_branchIsTrueOutMEMWB,s_overflowOutMEMWB,s_regWriteOutMEMWB,s_jumpOutMEMWB,s_haltOutMEMWB : std_logic;
```

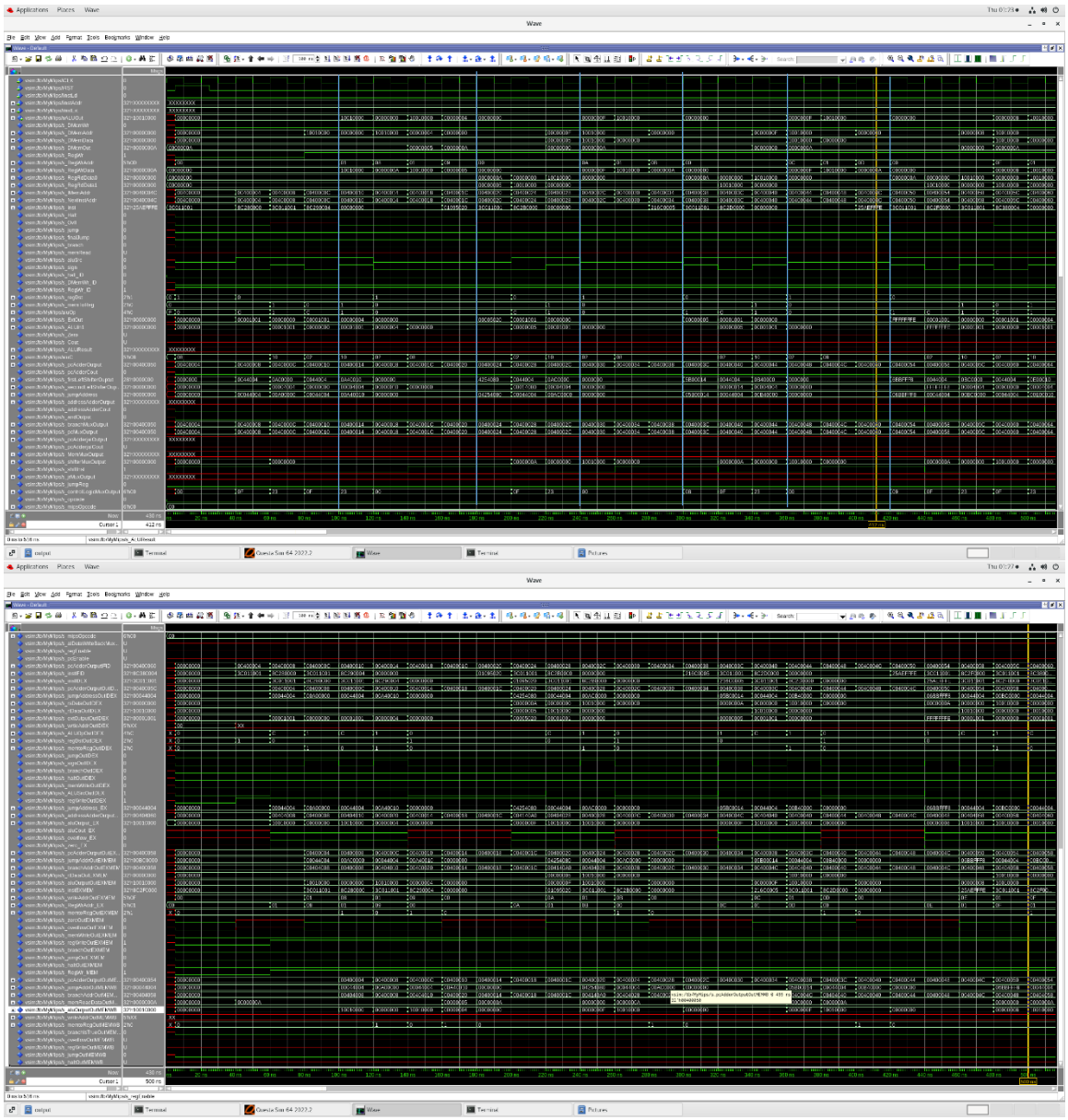
[1.b.ii] high-level schematic drawing of the interconnection between components.

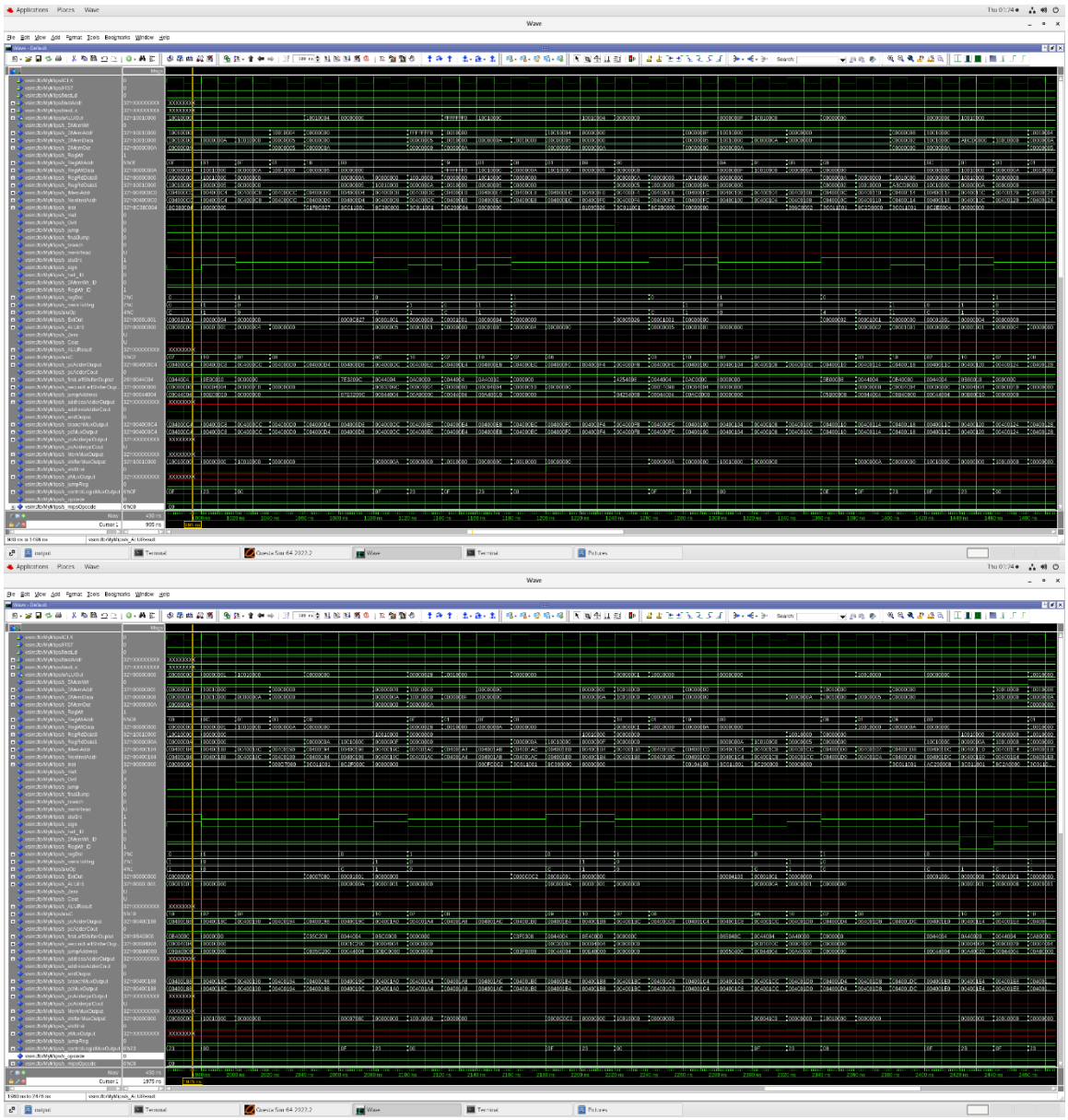


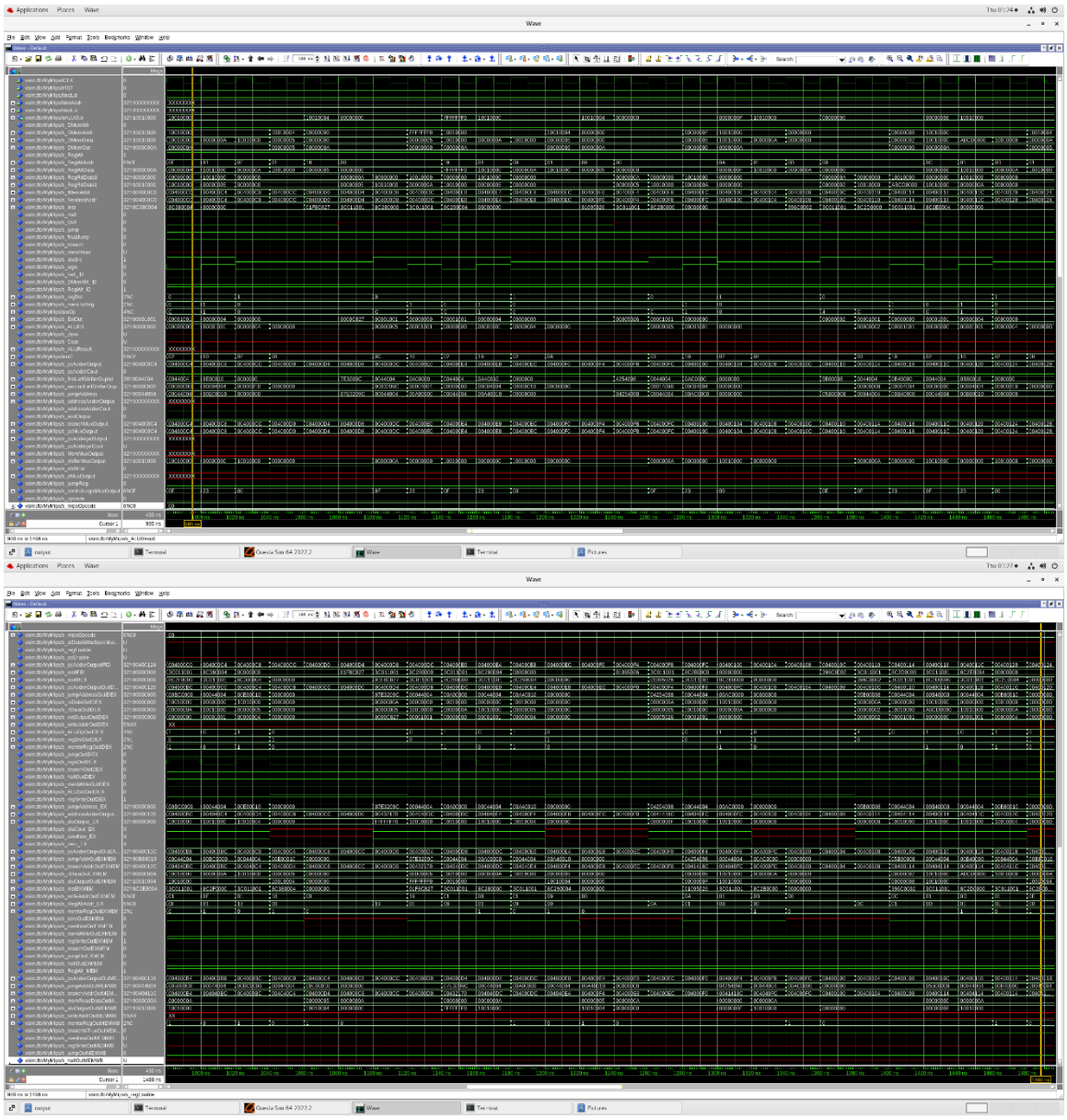
[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

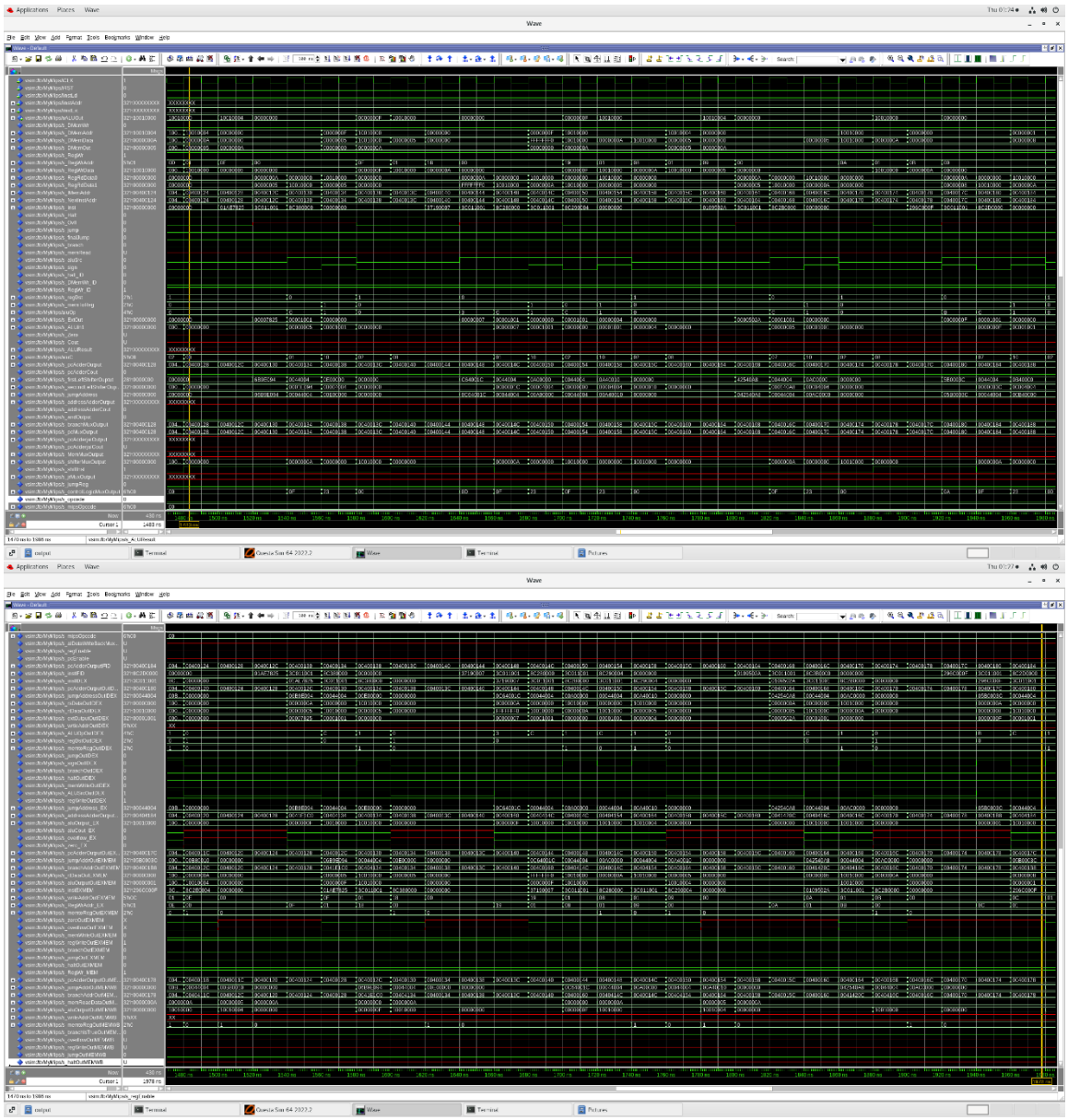
The test case is an edited version of my Proj1_base_test.s by adding NOPs. The waveform is worked as I expected. The values that stored into the register file are correct.

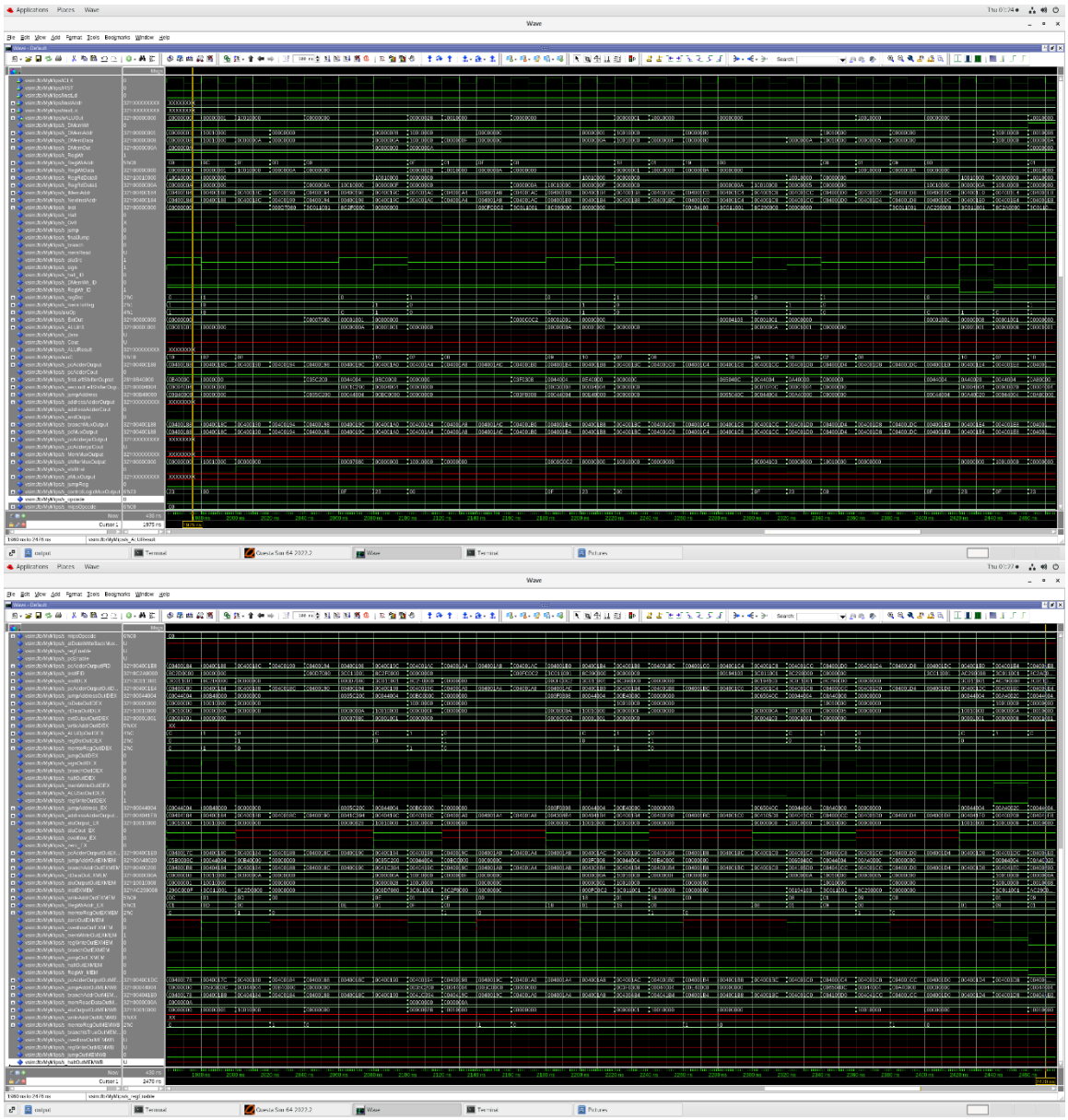
The first instruction was being executed was lw $t0, num1. The final result was written back to register file at cycle 5 due to the pipeline design. The second instruction is lw $t1, num2. Since there are no data hazards between these two instructions. Therefore, the final result of second instruction will store into the register at cycle 7. If these two instructions weren't pseudocode, the first instruction result will store at cycle 4, the second instruction result will store at cycle 6. The third instruction is add $t2, $t0, $t1, which reading $t0 and $t1. The instruction couldn't compute until the first and second instruction data has been written back to register file. Therefore, I add 3 nops to prevent data hazards. Therefore, the third instruction data will be written back to register file at cycle 11.

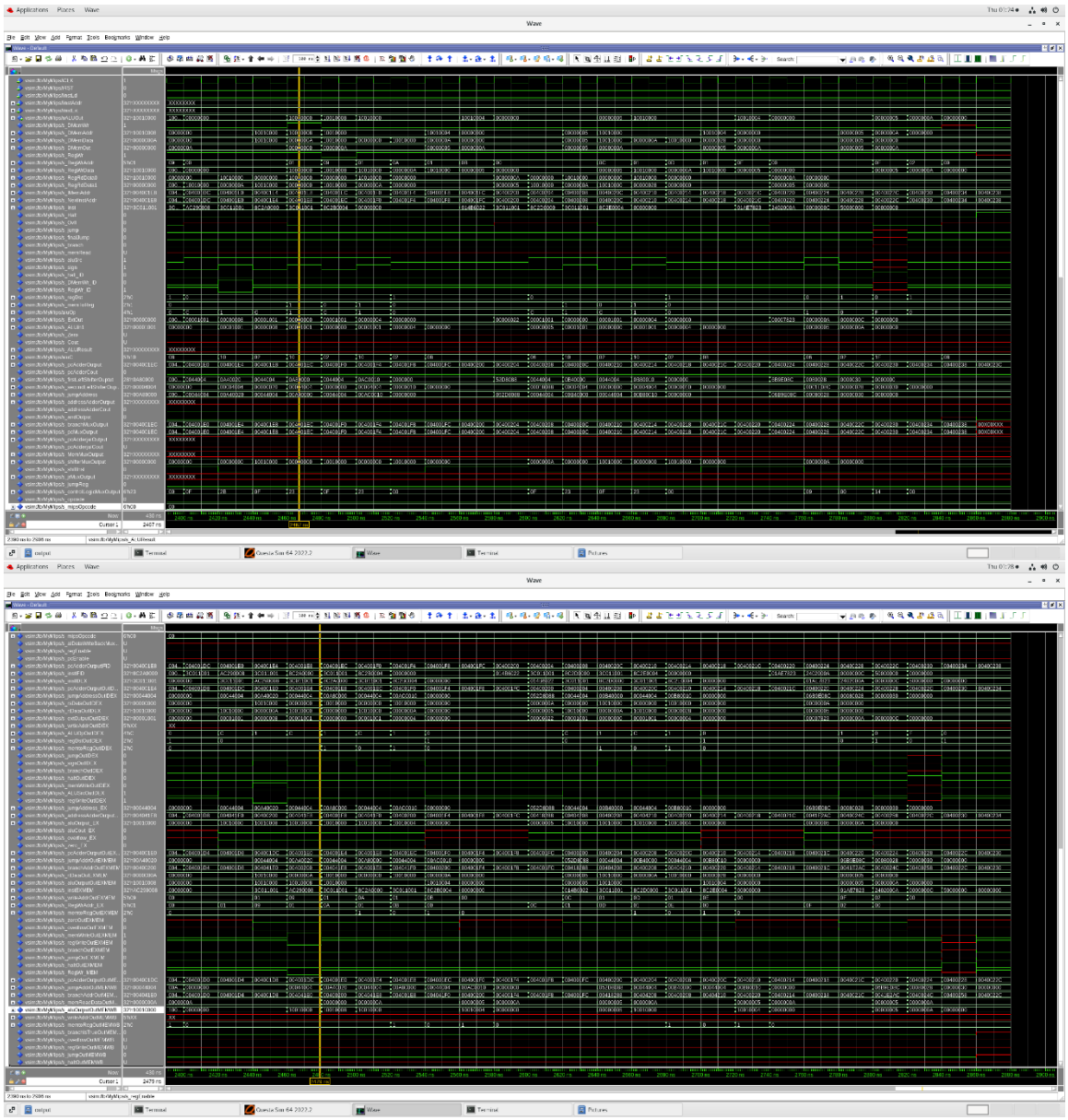












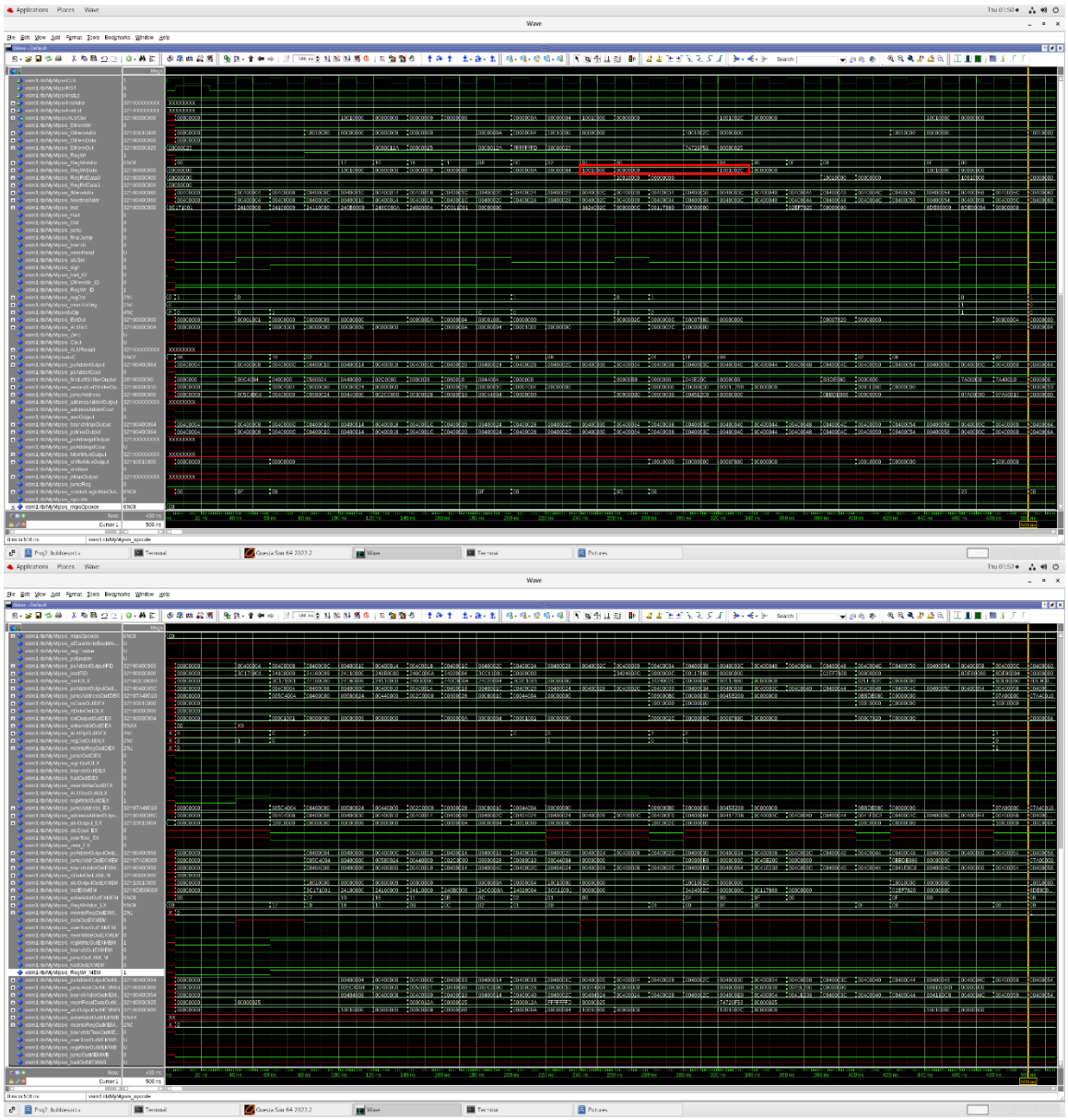
[1.c.ii] Include an annotated waveform in your writeup of two iterations or recursions of these programs executing correctly and provide a short discussion of result correctness. In your waveform and annotation, provide 3 different examples (at least one data-flow and one control-flow) of where you did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

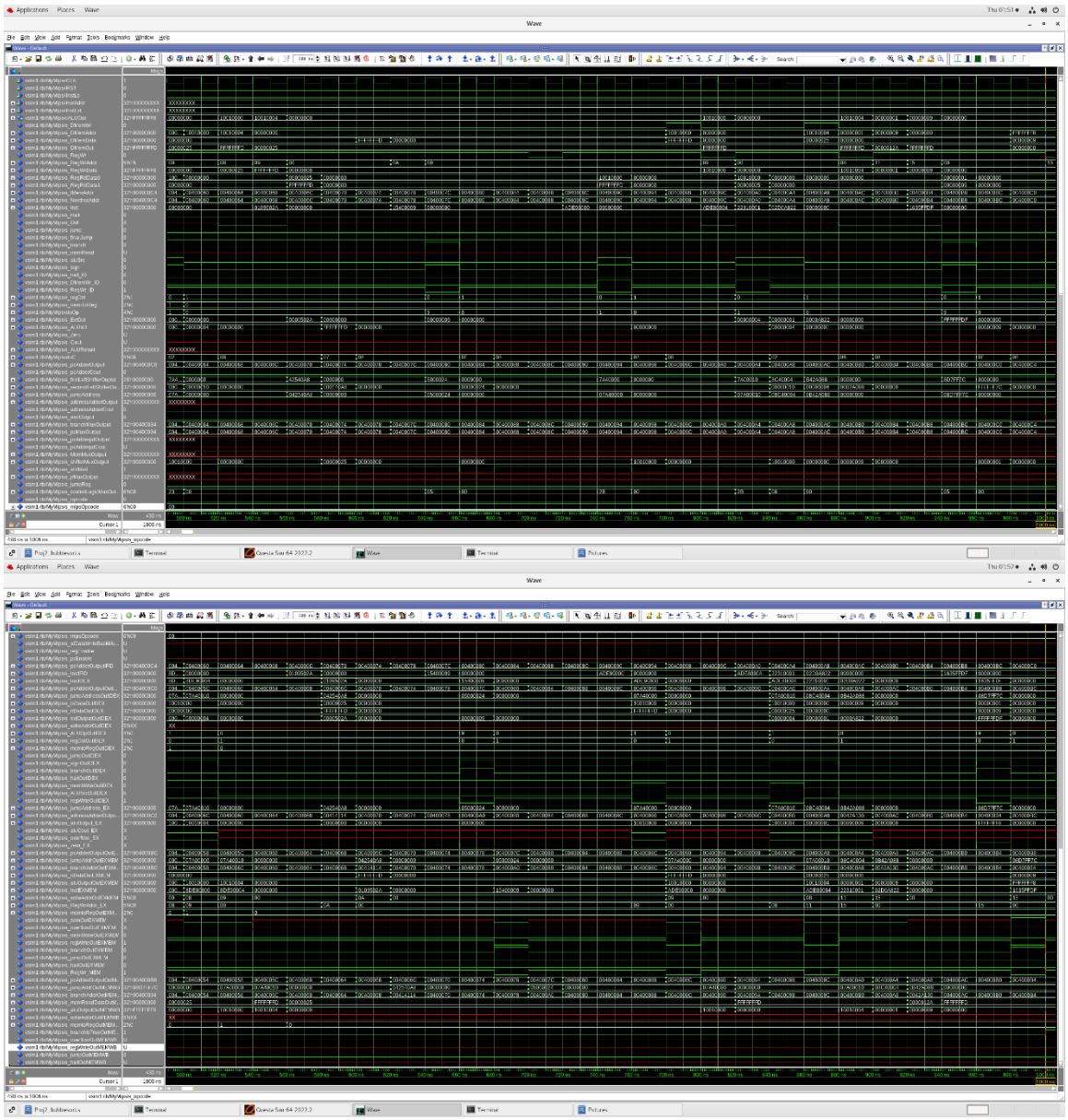
The test case is an edited version of my Proj1_bubblesort.s by adding NOPs. The waveform is worked as I expected. The values that stored into the register file are correct.

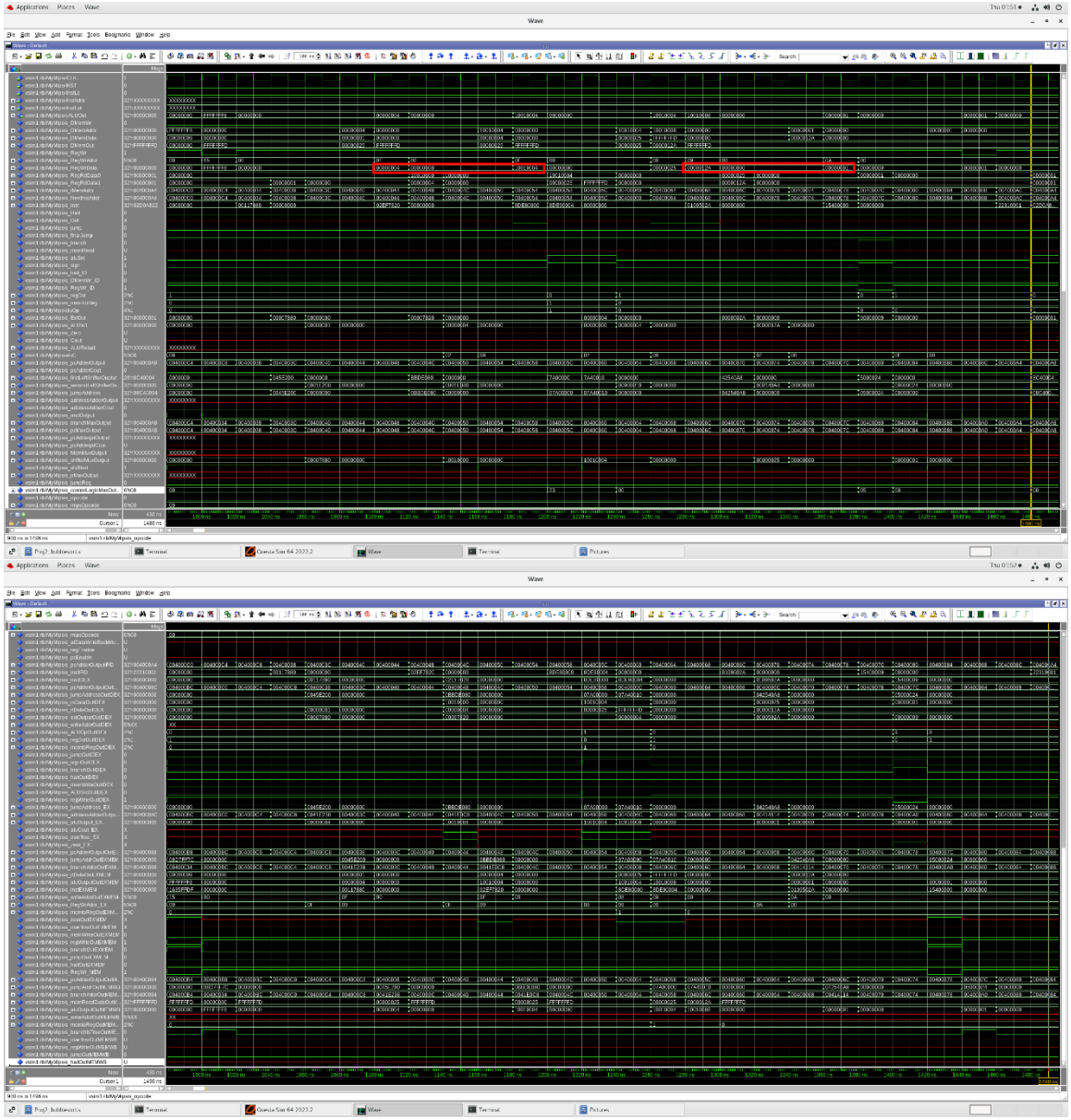
In clock cycle: 3
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 4
Register Write to Reg: 0x01 Val: 0x10010000
In clock cycle: 5
Register Write to Reg: 0x00 Val: 0x00000000

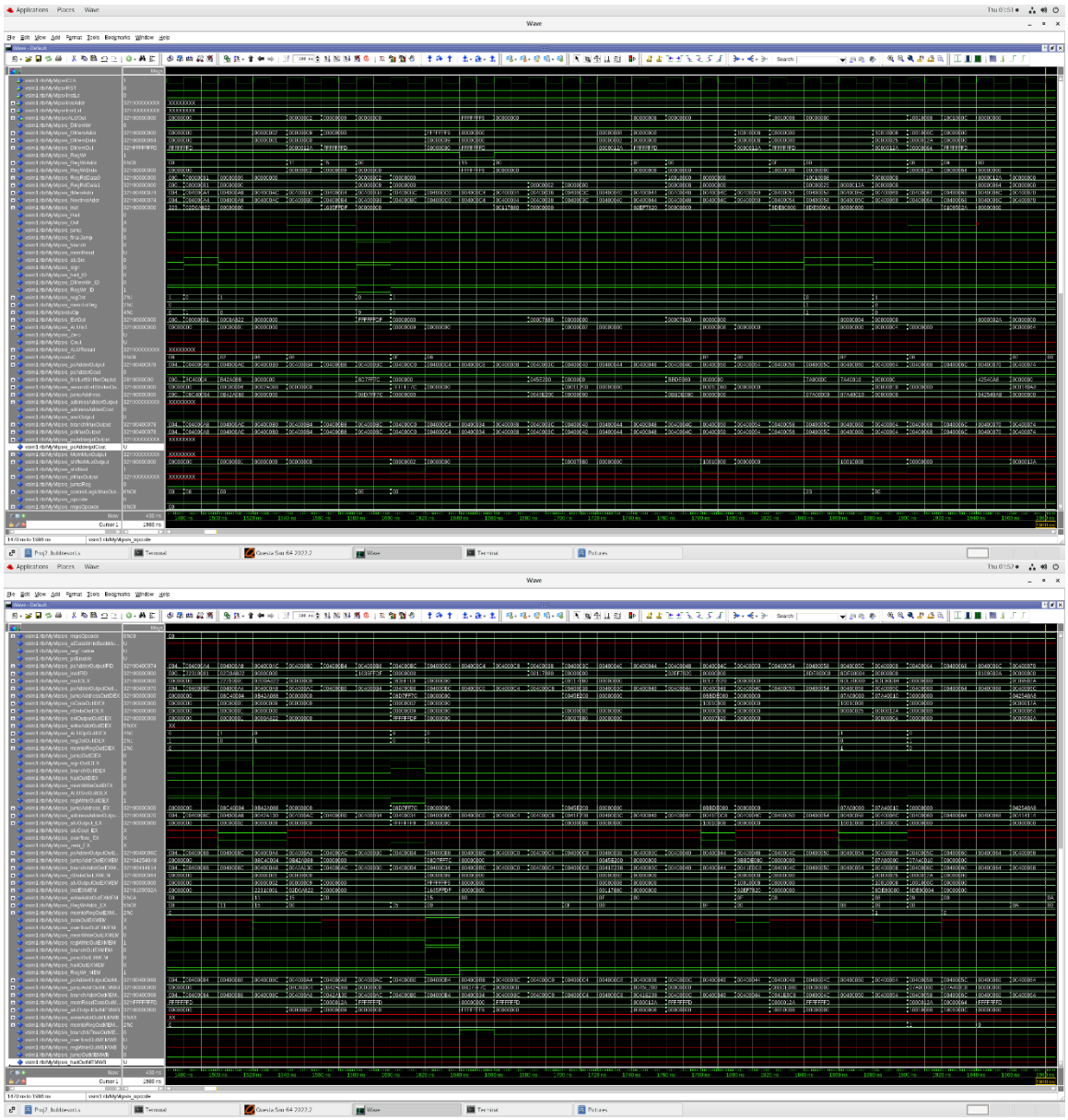
In clock cycle: 6
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 7
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 8
Register Write to Reg: 0x10 Val: 0x10010000
In clock cycle: 9
Register Write to Reg: 0x01 Val: 0x10010000
In clock cycle: 10
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 11
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 12
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 13
Register Write to Reg: 0x15 Val: 0x1001004C
In clock cycle: 14
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 15
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 16
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 17
Register Write to Reg: 0x15 Val: 0x00000013
In clock cycle: 18
Register Write to Reg: 0x12 Val: 0x00000001
In clock cycle: 19
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 20
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 21
Register Write to Reg: 0x00 Val: 0x00000000
In clock cycle: 21
Memory Write to Addr: 0x10010000 Val: 0x00000001
In clock cycle: 22
Memory Write to Addr: 0x10010004 Val: 0x00000001
In clock cycle: 24
Register Write to Reg: 0x11 Val: 0x00000011
In clock cycle: 25
Register Write to Reg: 0x13 Val: 0x00000001
In clock cycle: 26
Register Write to Reg: 0x14 Val: 0x00000001

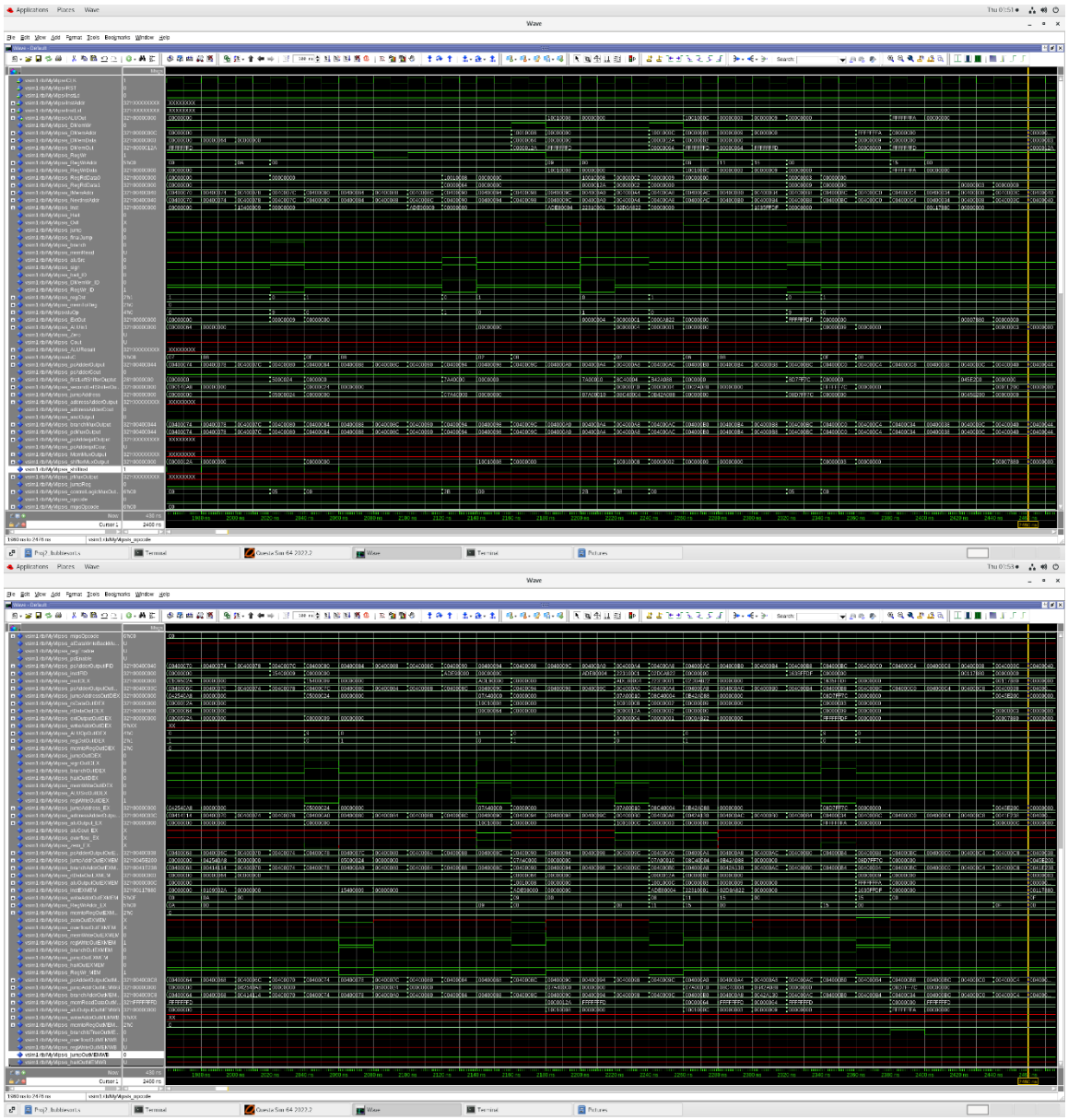
The regular instructions do not need a full flush for data dependencies, and the waveform shows that only 3 nops are needed for those situations, except branch and jump instructions.

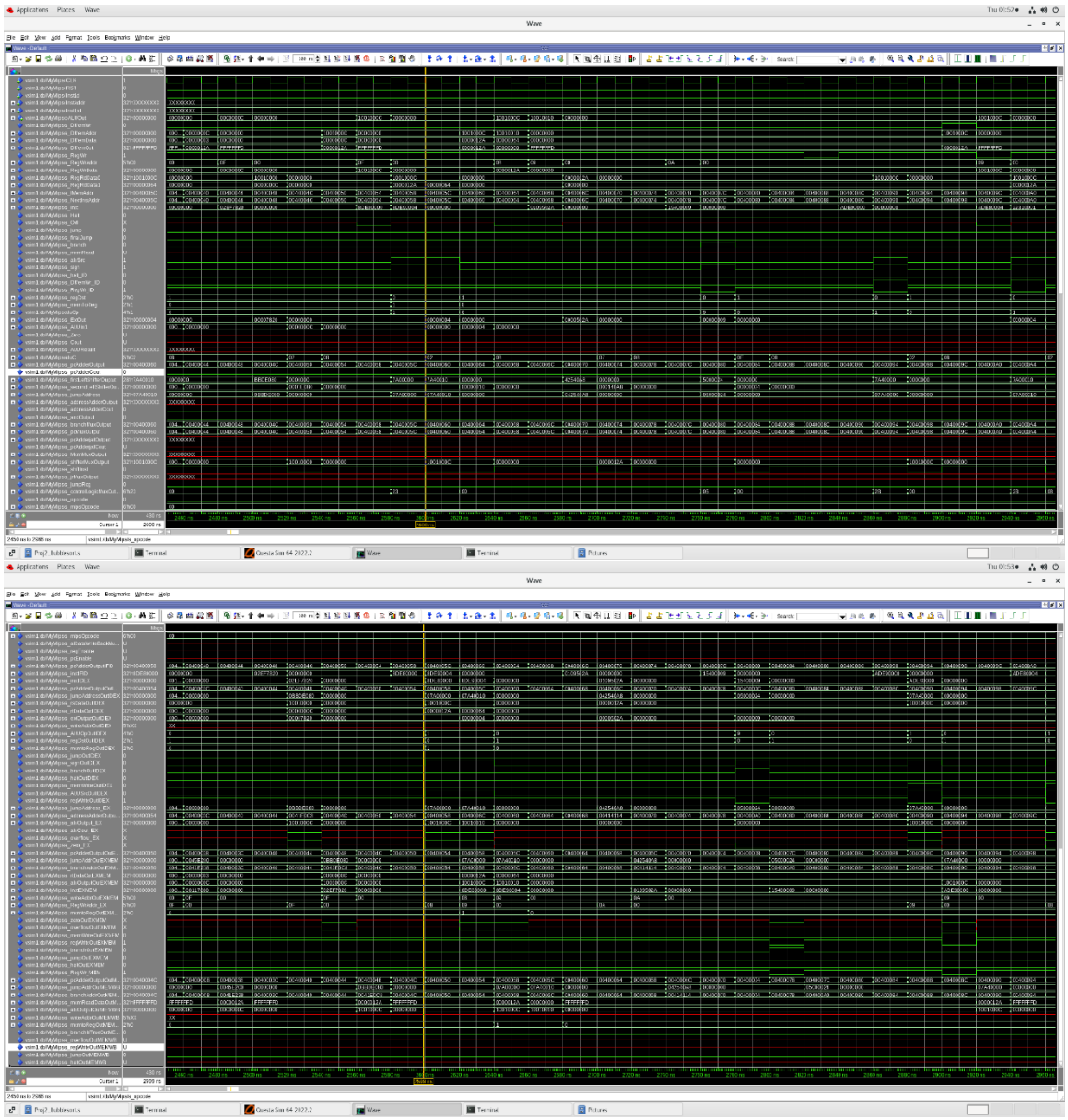








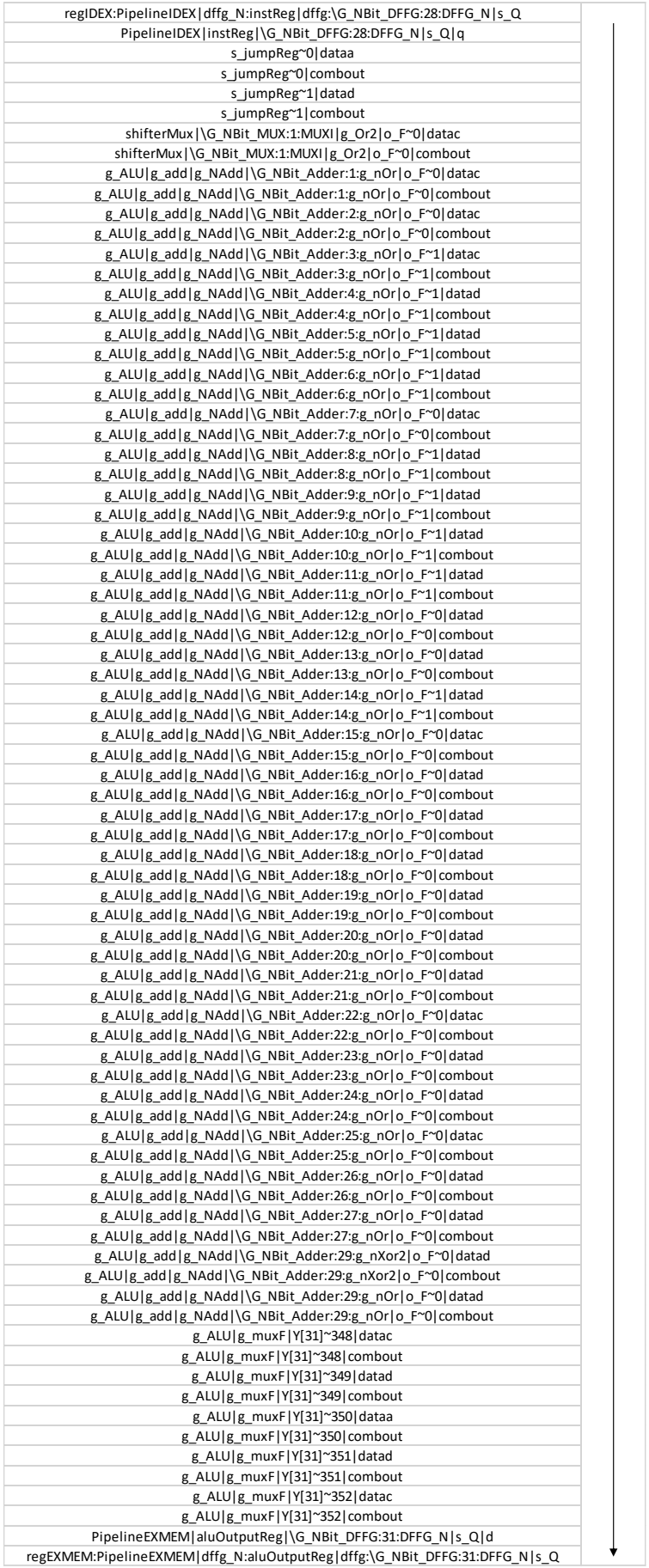




[1.d] report the maximum frequency your software-scheduled pipelined processor can run at and determine what your critical path is (specify each module/entity/component that this path goes through).

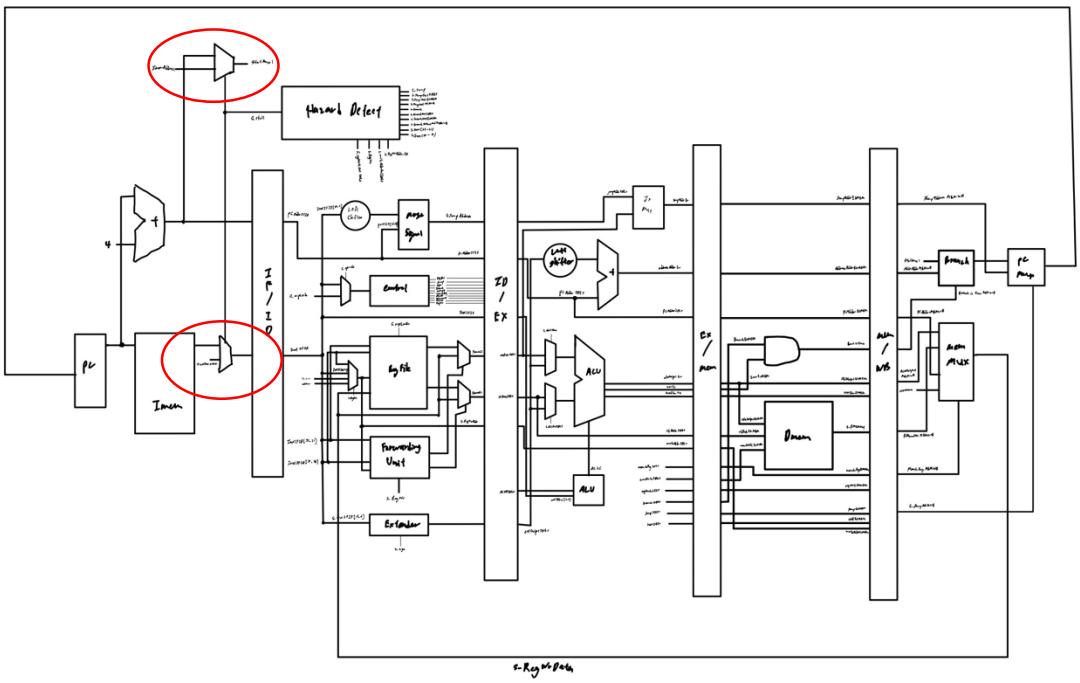
Max frequency: 53.80 mhz

The critical path goes from the ID/EX Pipeline Register, into the ALUSrc Mux, through the main ALU, and ends at the EX/MEM Pipeline Register.


regIDEX:PipelineIDEX|dffg_N:instReg|dffg:\G_NBit_DFFG:28:DFFG_N|s_Q
PipelineIDEX|instReg|\G_NBit_DFFG:28:DFFG_N|s_Q|q
s_jumpReg~0|dataa
s_jumpReg~0|combout
s_jumpReg~1|datad
s_jumpReg~1|combout
shifterMux|\G_NBit_MUX:1:MUXI|g_Or2|o_F~0|datac
shifterMux|\G_NBit_MUX:1:MUXI|g_Or2|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:1:g_nOr|o_F~0|datac
g_ALU|g_add|g_NAdd|\G_NBit_Adder:1:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:2:g_nOr|o_F~0|datac
g_ALU|g_add|g_NAdd|\G_NBit_Adder:2:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:3:g_nOr|o_F~1|datac
g_ALU|g_add|g_NAdd|\G_NBit_Adder:3:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:4:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:4:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:5:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:5:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:6:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:6:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:7:g_nOr|o_F~0|datac
g_ALU|g_add|g_NAdd|\G_NBit_Adder:7:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:8:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:8:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:9:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:9:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:10:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:10:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:11:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:11:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:12:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:12:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:13:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:13:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:14:g_nOr|o_F~1|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:14:g_nOr|o_F~1|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:15:g_nOr|o_F~0|datac
g_ALU|g_add|g_NAdd|\G_NBit_Adder:15:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:16:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:16:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:17:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:17:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:18:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:18:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:19:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:19:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:20:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:20:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:21:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:21:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:22:g_nOr|o_F~0|datac
g_ALU|g_add|g_NAdd|\G_NBit_Adder:22:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:23:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:23:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:24:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:24:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:25:g_nOr|o_F~0|datac
g_ALU|g_add|g_NAdd|\G_NBit_Adder:25:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:26:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:26:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:27:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:27:g_nOr|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:29:g_nXor2|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:29:g_nXor2|o_F~0|combout
g_ALU|g_add|g_NAdd|\G_NBit_Adder:29:g_nOr|o_F~0|datad
g_ALU|g_add|g_NAdd|\G_NBit_Adder:29:g_nOr|o_F~0|combout
g_ALU|g_muxF|Y[31]~348|datac
g_ALU|g_muxF|Y[31]~348|combout
g_ALU|g_muxF|Y[31]~349|datad
g_ALU|g_muxF|Y[31]~349|combout
g_ALU|g_muxF|Y[31]~350|dataa
g_ALU|g_muxF|Y[31]~350|combout
g_ALU|g_muxF|Y[31]~351|datad
g_ALU|g_muxF|Y[31]~351|combout
g_ALU|g_muxF|Y[31]~352|datac
g_ALU|g_muxF|Y[31]~352|combout
PipelineEXMEM|aluOutputReg|\G_NBit_DFFG:31:DFFG_N|s_Q|d
regEXMEM:PipelineEXMEM|dffg_N:aluOutputReg|dffg:\G_NBit_DFFG:31:DFFG_N|s_Q

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

Instead of implementing stalling and flushing function in each pipeline registers, I used two multiplexers to do the stalling and flushing operation. The top multiplexer that showed in the design is used to select between PC+4 and PC. The bottom multiplexer that showed in the design is used to select between s_NextAddrInst and 0x00000000. When s_stall is set to 1, the top multiplexer will output current PC address to branch mux and the bottom multiplexer will output 0x00000000 as s_inst to IF/ID register. Therefore, the new instruction won't be executed until the data hazards has been resolved and s_stall has set to 0.



[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.

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

Instruction	Corresponding Signals Produced	
add, addu, and, nor, xor, or, slt, sll, srl, sra, sub, subu	s_RegWrAddr_ID	o_ALUOut
	s_writeAddrOutIDEX	s_inst
	s_writeAddrOutEXMEM	
addi, addiu, andi, xori, ori, slti, lui	s_RegWr_ID	s_inst
	s_regWriteOutIDEX	
	s_regWriteOutEXMEM	
beq, bne	s_branch	
	s_branchOutIDEX	
	s_branchOutEXMEM	
	s_branchIsTrueOutMEMWB	
j, jr, jal	s_jump	
	s_finalJump	
	s_jumpOutIDEX	
	s_jumpOutEXMEM	
	s_jumpOutMEMWB	

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

Instruction	Corresponding Signals Cosumed	
add, addu, and, nor, xor, or, slt, sll, srl, sra, sub, subu	s_RegWrAddr_ID	s_forwardMux1_ID
	s_writeAddrOutIDEX	s_forwardMux2_ID,
	s_writeAddrOutEXMEM	ALUC
	s_Inst	
	s_InstIDEX	
	s_InstEXMEM	
addi, addiu, andi, xori, ori, slti, lui	s_Inst	s_forwardMux1_ID
	s_InstIDEX	ALUC
	s_InstEXMEM	
	s_RegWr_ID	
	s_regWriteOutIDEX	
	s_regWriteOutEXMEM	
beq, bne	s_branch	
	s_branchOutIDEX	
	s_branchOutEXMEM	
	s_branchIsTrueOutMEMWB	
j, jr, jal	s_jump	
	s_finalJump	
	s_jumpOutIDEX	
	s_jumpOutEXMEM	
	s_jumpOutMEMWB	

[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.

Stall	Forward
s_finalJump	s_RegWrData
s_jumpOutIDEX	
s_jumpOutEXMEM	
s_jumpOutMEMWB	
s_branch	
s_branchOutIDEX	
s_branchOutEXMEM	
s_branchIsTrueOutMEMWB	
s_Inst(25 downto 21)	
s_Inst(20 downto 16)	
s_RegWrAddr_ID	
s_writeAddrOutIDEX	
s_RegWr_ID	
s_regWriteOutIDEX	

s_RegWrData
Receiving: Write Back Stage
Forwarding: Execution Stage

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

```
-- Required register file signals
signal s_RegWr       : std_logic; -- TODO: use this signal as the final active high write enable input to the register file
signal s_RegWrAddr   : std_logic_vector(4 downto 0); -- TODO: use this signal as the final destination register address input
signal s_RegWrData   : std_logic_vector(N-1 downto 0); -- TODO: use this signal as the final data memory data input
signal s_RegRdData0  : std_logic_vector(N-1 downto 0);
signal s_RegRdData1  : std_logic_vector(N-1 downto 0);

-- Required instruction memory signals
signal s_IMemAddr    : std_logic_vector(N-1 downto 0); -- Do not assign this signal, assign to s_NextInstAddr instead
signal s_NextInstAddr : std_logic_vector(N-1 downto 0); -- TODO: use this signal as your intended final instruction memory address input.
signal s_Inst        : std_logic_vector(N-1 downto 0); -- TODO: use this signal as the instruction signal

-- Required halt signal -- for simulation
signal s_Halt        : std_logic;  -- TODO: this signal indicates to the simulation that intended program execution has completed. (Opcode: 01 0100)

-- Required overflow signal -- for overflow exception detection
signal s_Ovfl        : std_logic;  -- TODO: this signal indicates an overflow exception would have been initiated

-- Control signals
signal s_jump,s_finalJump, s_branch, s_memRead, s_aluSrc, s_sign, s_halt_ID : std_logic;
signal s_DMemWr_ID, s_RegWr_ID : std_logic;
signal s_regDst, s_memToReg : std_logic_vector(1 downto 0);
signal aluOp :std_logic_vector(3 downto 0);

-- Sign Extender
signal s_ExtOut : std_logic_vector(31 downto 0);

-- ALU signals
signal s_ALUIn1 : std_logic_vector(31 downto 0);
signal s_Zero : std_logic;
signal s_Cout : std_logic;
signal s_ALUResult : std_logic_vector(31 downto 0);

-- ALU Control signals
signal aluC : std_logic_vector(4 downto 0);

-- PC Adder signals
signal s_pcAdderOutput : std_logic_vector(31 downto 0);
signal s_pcAdderCout : std_logic;

-- firstLeftShifter signals
signal s_firstLeftShifterOuptut : std_logic_vector(27 downto 0);
```

```
-- secondLeftShifter signals
signal s_secondLeftShifterOuptut : std_logic_vector(31 downto 0);

-- g_Merge signals
signal s_jumpAddress : std_logic_vector(31 downto 0);

-- addressAdder signal
signal s_addressAdderOutput : std_logic_vector(31 downto 0);
signal s_addressAdderCout : std_logic;

-- g_and signal
signal s_andOutput : std_logic;

-- BranchMux signal
signal s_branchMuxOutput : std_logic_vector(31 downto 0);

--pcMUX
signal s_pcMuxOutput : std_logic_vector(31 downto 0);

--pcAdder_jal
signal s_pcAdderjalOutput : std_logic_vector(31 downto 0);
signal s_pcAdderjalCout : std_logic;

--MemMux
signal s_MemMuxOutput : std_logic_vector(31 downto 0);

--shifterMux
signal s_shifterMuxOutput : std_logic_vector(31 downto 0);
signal s_shiftInst : std_logic := '0';

--JrMux
signal s_jrMuxOutput : std_logic_vector(31 downto 0);
signal s_jumpReg : std_logic := '0';

--controlLogicMux
signal s_controlLogicMuxOutput : std_logic_vector(5 downto 0);
signal s_opcode : std_logic;

-- bgezal, bltzal, bltz
signal s_mipsOpcode : std_logic_vector(5 downto 0);

--alDataWriteBackMux
signal s_alDataWriteBackMuxOutput : std_logic;
signal s_regEnable : std_logic;
```

```
-- g_PC
signal s_pcEnable : std_logic;

-- PipelineIFID
signal s_pcAdderOutputIFID : std_logic_vector(31 downto 0);
signal s_instIFID : std_logic_vector(31 downto 0);


-- hazardDetectUnit
signal s_stall : std_logic;

-- stallMux1 & stallMux2
signal s_stallMux1Output : std_logic_vector(31 downto 0);
signal s_stallMux2Output : std_logic_vector(31 downto 0);

-- ForwardingUnit
signal s_forwardMux1, s_forwardMux2 : std_logic;

-- RegFile
signal s_rsData, s_rtData : std_logic_vector(31 downto 0);

-- forwardMux1 & forwardMux2
signal s_forwardMux1_ID, s_forwardMux2_ID : std_logic_vector(31 downto 0);

-- WMUX
signal s_RegWrAddr_ID : std_logic_vector(4 downto 0);

-- PipelineIDEX
signal s_instIDEX,s_pcAdderOutputOutIDEX, s_jumpAddressOutIDEX, s_rsDataOutIDEX, s_rtDataOutIDEX, s_extOutputOutIDEX : std_logic_vector(31 downto 0);
signal s_wrtieAddrOutIDEX : std_logic_vector(4 downto 0);
signal s_ALUOpOutIDEX : std_logic_vector(3 downto 0);
signal s_regDstOutIDEX, s_memtoRegOutIDEX : std_logic_vector(1 downto 0);
signal s_jumpOutIDEX, s_signOutIDEX, s_branchOutIDEX, s_haltOutIDEX, s_memWriteOutIDEX, s_ALUSrcOutIDEX, s_regWriteOutIDEX : std_logic;

-- JrMux
signal s_jumpAddress_EX : std_logic_vector(31 downto 0);

-- addressAdder
signal s_addressAdderOutput_EX : std_logic_vector(31 downto 0);

-- ALU
signal s_aluOutput_EX : std_logic_vector(31 downto 0);
signal s_aluCout_EX, s_overflow_EX, s_zero_EX : std_logic;

-- PipelineEXMEM

signal s_pcAdderOutputOutEXMEM, s_jumpAddrOutEXMEM, s_branchAddrOutEXMEM, s_rtDataOutEXMEM, s_aluOutputOutEXMEM,s_instEXMEM  : std_logic_vector(31 downto 0);
signal s_writeAddrOutEXMEM, s_RegWrAddr_EX : std_logic_vector(4 downto 0);
signal s_memtoRegOutEXMEM : std_logic_vector(1 downto 0);
signal s_zeroOutEXMEM,s_overflowOutEXMEM,s_memWriteOutEXMEM,s_regWriteOutEXMEM,s_branchOutEXMEM,s_jumpOutEXMEM,s_haltOutEXMEM, s_RegWr_MEM : std_logic;

-- PipelineMEMWB
signal s_pcAdderOutputOutMEMWB, s_jumpAddrOutMEMWB, s_branchAddrOutMEMWB, s_memReadDataOutMEMWB, s_aluOutputOutMEMWB  : std_logic_vector(31 downto 0);
signal s_writeAddrOutMEMWB : std_logic_vector(4 downto 0);
signal s_memtoRegOutMEMWB : std_logic_vector(1 downto 0);
signal s_branchIsTrueOutMEMWB,s_overflowOutMEMWB,s_regWriteOutMEMWB,s_jumpOutMEMWB,s_haltOutMEMWB : std_logic;
```

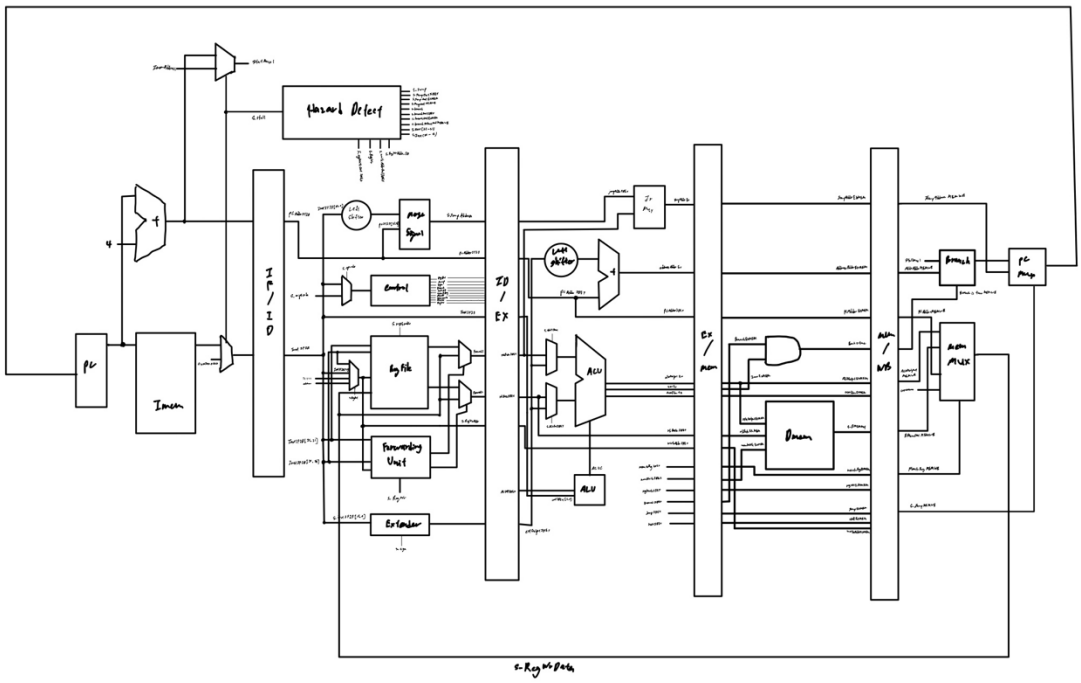
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

Instruction	Pipeline Stage
beq, bne, j, jal	Write Back

[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.

Because we placed our Hazard Detection module inside the Fetch stage of the pipeline, the processor will not load an instruction if there is any branching or jumping instructions in the pipeline already. Therefore, no stages need to be squashed or flushed, and the processor will stall itself and wait for the hazard to resolve itself.
To answer the question, it is possible that stages IF, ID, EX, and MEM may need to wait while the jump instruction is in the WB stage.

[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.

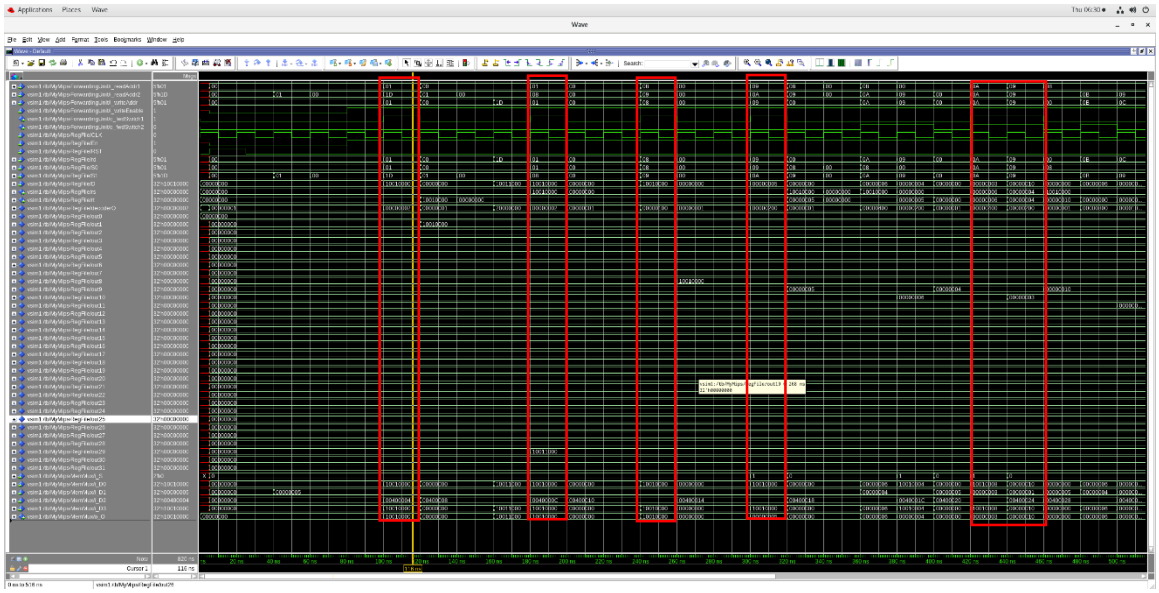
[2.e.i] Create a spreadsheet to track these cases and justify the coverage of your testing approach. Include this spreadsheet in your report as a table.

Type	Description	Example
IF -> ID	Instruction in IF depends on data in ID	addi $t0, $Zero, 0x01
		sw $t0, 4($sp)
IF - EX	Instruction in IF depends on data in EX	lw $t1, 4($sp)
		lw $t2, 8($sp)
		add $t3, $t1, $t1
IF -> MEM	Instruction in IF depends on data in MEM	add $t2, $t1, $t0
		add $t1, $t0, $t0
		sw $t3, 4($sp)
		sw $t2, 8($sp)

The output of the test case was correct and expected. As we can see, when the hazard detector detected a data hazard, it will then set the s_stall to 1. The PC address will be on hold until the hazard has been resolved. Besides that, the forwarder forwarded the data from MemMux (Write Back Stage) to RegisterFile (Decode Stage) when another instruction required to read the value of the register that is in-processing and has been done written to Write back stage.

[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.

Type	Description	Example
Jump	Invariably change PC	j loop
Branch	Conditionally change PC	bne $t0, $zero, example



The output of the test case was correct and expected. The forwarder forwarded the data from MemMux (Write Back Stage) to RegisterFile (Decode Stage) when another instruction required to read the value of the register that is in-processing and has been done written to Write back stage.

[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).

Max frequency: 52.58mhz
The critical path goes from the pcReg register, into the Instruction Memory, through the hazard detection unit, and ends at the IF/ID register.


regIDEX:PipelineIDEX|dffg_N:extOutputReg|dffg:\G_NBit_DFFG:1:DFFG_N|s_Q
PipelineIDEX|extOutputReg|\G_NBit_DFFG:1:DFFG_N|s_Q|q
addressAdder|g_NAdd|\G_NBit_Adder:3:g_nOr|o_F~0|dataa
addressAdder|g_NAdd|\G_NBit_Adder:3:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:4:g_nOr|o_F~1|datad
addressAdder|g_NAdd|\G_NBit_Adder:4:g_nOr|o_F~1|combout
addressAdder|g_NAdd|\G_NBit_Adder:5:g_nOr|o_F~0|dataa
addressAdder|g_NAdd|\G_NBit_Adder:5:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:8:g_nOr|o_F~5|datab
addressAdder|g_NAdd|\G_NBit_Adder:8:g_nOr|o_F~5|combout
addressAdder|g_NAdd|\G_NBit_Adder:8:g_nOr|o_F~2|datad
addressAdder|g_NAdd|\G_NBit_Adder:8:g_nOr|o_F~2|combout
addressAdder|g_NAdd|\G_NBit_Adder:10:g_nOr|o_F~5|datac
addressAdder|g_NAdd|\G_NBit_Adder:10:g_nOr|o_F~5|combout
addressAdder|g_NAdd|\G_NBit_Adder:10:g_nOr|o_F~2|datad
addressAdder|g_NAdd|\G_NBit_Adder:10:g_nOr|o_F~2|combout
addressAdder|g_NAdd|\G_NBit_Adder:12:g_nOr|o_F~5|datad
addressAdder|g_NAdd|\G_NBit_Adder:12:g_nOr|o_F~5|combout
addressAdder|g_NAdd|\G_NBit_Adder:12:g_nOr|o_F~2|datac
addressAdder|g_NAdd|\G_NBit_Adder:12:g_nOr|o_F~2|combout
addressAdder|g_NAdd|\G_NBit_Adder:13:g_nOr|o_F~0|datab
addressAdder|g_NAdd|\G_NBit_Adder:13:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:14:g_nOr|o_F~1|datad
addressAdder|g_NAdd|\G_NBit_Adder:14:g_nOr|o_F~1|combout
addressAdder|g_NAdd|\G_NBit_Adder:15:g_nOr|o_F~0|datab
addressAdder|g_NAdd|\G_NBit_Adder:15:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:16:g_nOr|o_F~1|datad
addressAdder|g_NAdd|\G_NBit_Adder:16:g_nOr|o_F~1|combout
addressAdder|g_NAdd|\G_NBit_Adder:17:g_nOr|o_F~0|dataa
addressAdder|g_NAdd|\G_NBit_Adder:17:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:19:g_nOr|o_F~0|datad
addressAdder|g_NAdd|\G_NBit_Adder:19:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:21:g_nOr|o_F~0|datad
addressAdder|g_NAdd|\G_NBit_Adder:21:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:23:g_nOr|o_F~0|datad
addressAdder|g_NAdd|\G_NBit_Adder:23:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:25:g_nOr|o_F~0|datad
addressAdder|g_NAdd|\G_NBit_Adder:25:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:27:g_nOr|o_F~0|datad
addressAdder|g_NAdd|\G_NBit_Adder:27:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:29:g_nOr|o_F~0|datac
addressAdder|g_NAdd|\G_NBit_Adder:29:g_nOr|o_F~0|combout
addressAdder|g_NAdd|\G_NBit_Adder:31:g_nXor2|o_F|datad
addressAdder|g_NAdd|\G_NBit_Adder:31:g_nXor2|o_F|combout
PipelineEXMEM|branchAddrReg|\G_NBit_DFFG:31:DFFG_N|s_Q|d
regEXMEM:PipelineEXMEM|dffg_N:branchAddrReg|dffg:\G_NBit_DFFG:31:DFFG_N|s_Q