CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Varun Advani

Kalyan Thapaliya

_____

_____

_____

_____

Project Teams Group #: 4-6

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.

IF stage

- s_PCPlusFour_IF: Address of the next instruction to be fetched.
- s_normAddress: Normalized instruction address.
- s_InstOrNOP: The instruction to be executed in this stage or a NOP instruction.
- s_stall: Signal indicating whether the pipeline should be stalled or not.

ID stage

- s_PCPlusFour_ID: Address of the next instruction to be decoded.
- s_Inst_ID: The instruction to be decoded in this stage.
- s_imm16_ID: The 16-bit immediate value contained in the instruction.
- s_imm32_ID: The 32-bit immediate value after sign-extension.
- s_jumpAddress_ID: The target address for a jump instruction.
- s_Ctrl_ID: Control signals for this stage.

EX stage

- s_ALUOp_EX: The ALU operation to be performed.
- s_ALUSrc_EX: Signal indicating whether to use the immediate value or the value from the second register in the ALU operation.
- s_ALU_EX: The ALU.

- s_PCPlusFour_EX: The next address to be fetched after this stage.

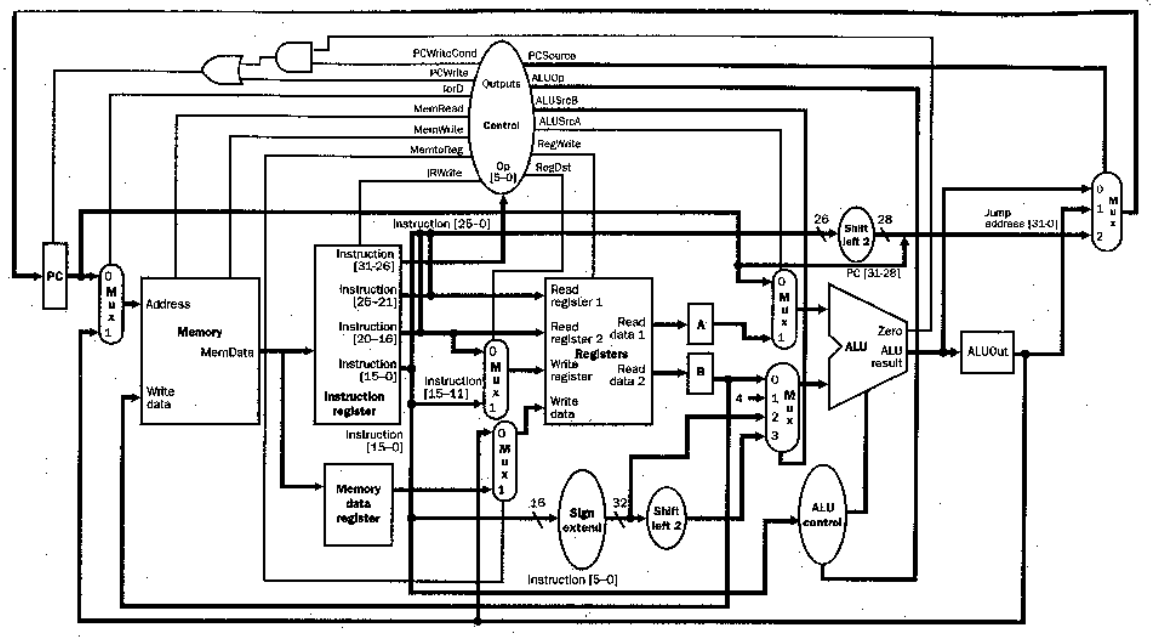
MEM stage

- s_MemRead_MEM: Signal indicating whether to read from memory or not.
- s_MemWrite_MEM: Signal indicating whether to write to memory or not.
- s_MemToReg_MEM: Signal indicating whether to write the memory value to the register file or the ALU output to the register file.
- s_MemAddr_MEM: The memory address to read from or write to.
- s_MemData_MEM: The data to be written to memory.
- s_MemOut_MEM: The data read from memory.

WB stage

- s_RegWrite_WB: Signal indicating whether to write to the register file or not.
- s_MemtoReg_WB: Signal indicating whether to write the memory value to the register file or the ALU output to the register file.
- s_RegAddr_WB: The register address to write to.
- s_RegData_WB: The data to be written to the register file.

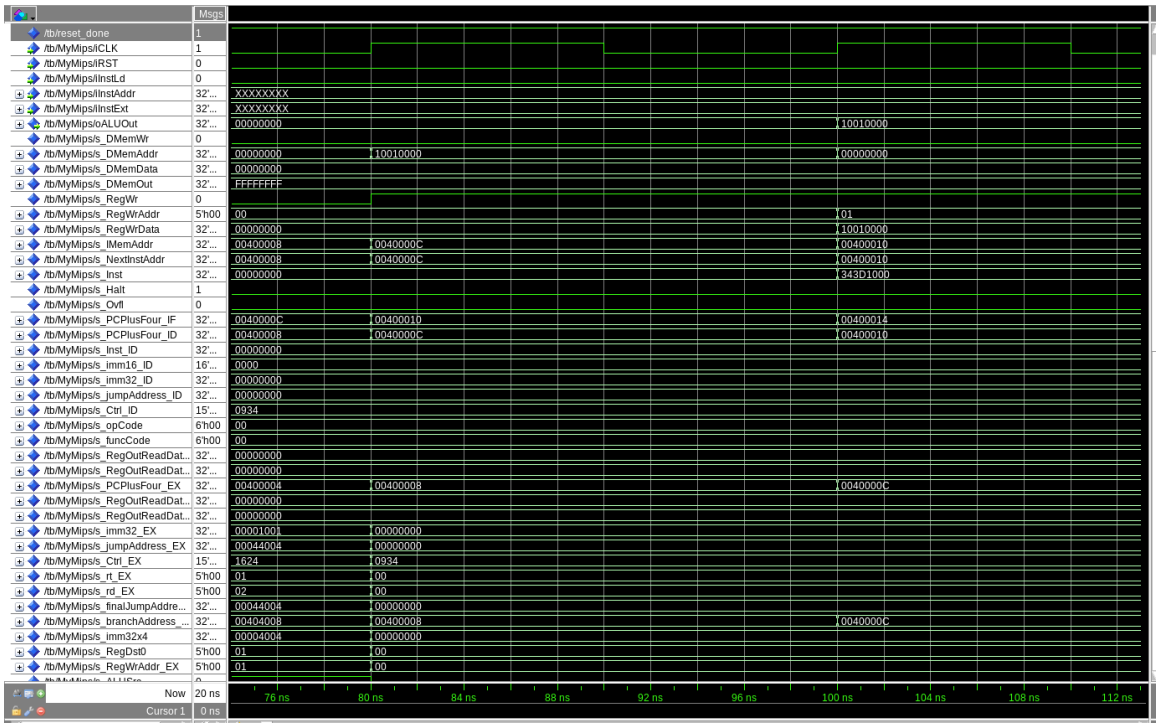
[1.b.ii] high-level schematic drawing of the interconnection between components.



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

The MIPS code in base_tests is a sequence of instructions that perform various operations on registers in the processor. The code begins by loading the immediate value 0x00001001 into register $1 using the lui (load upper immediate) instruction. Then, it uses the ori (OR immediate) instruction to set the value of the stack pointer ($sp) to 0x1000. The code then uses the addi (add immediate) instruction to set the value of register $t0 to 0x5 and register $t1 to 0x10. It then uses the add instruction to add the values in registers $t0 and $t1 and store the result in $t0. This causes the value in $t0 to be updated to 0x15. The code then uses the addiu (add immediate unsigned) instruction to add 0x10 to the value in $t0, updating its value to 0x25. The code then

continues to perform various other operations on registers using various instructions, such as lui, ori, and, andi, lui, sw, lw, nor, xor, xori, or, ori, slt, slti, sll, srl, sra, sub, subu, and finally, jr (jump register). These instructions update the values of various registers in the processor, resulting in the final values of the registers as indicated in the comments. The values in the comments



[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.

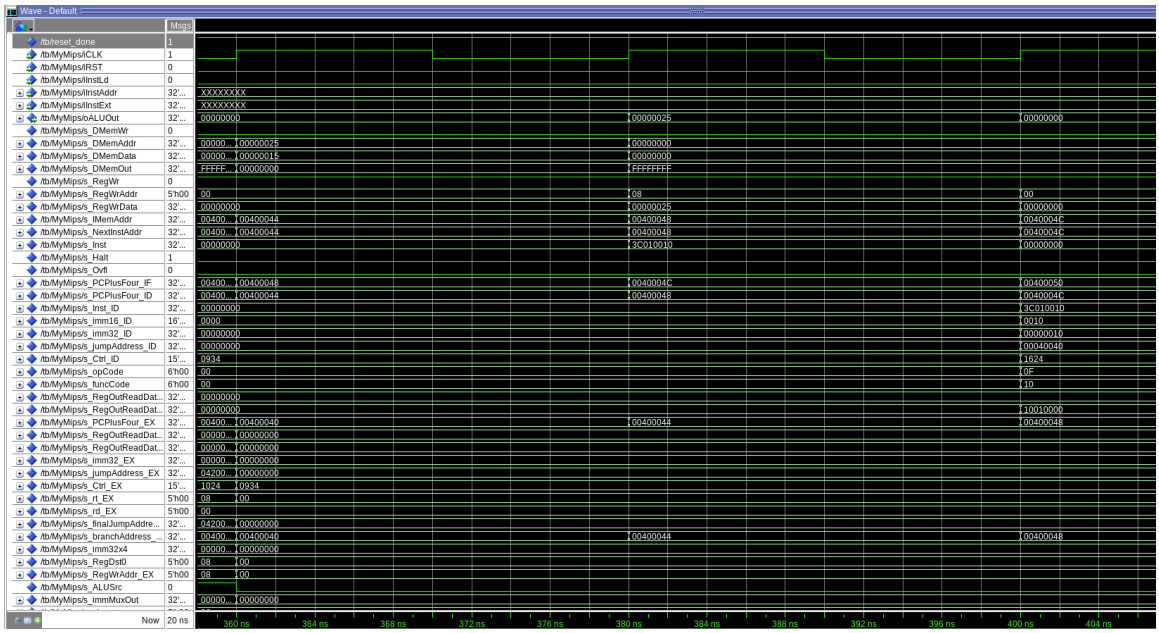
Attached below is the final value of the registers and how the update as can be seen in the waveform
Registers:

- $t0: 0x5, 0x15, 0x25
- $t1: 0x10
- $t2: 0x20, 0x0
- $t3: 0x1001
- $t4: 0x1001
- $t5: 0x01101111, 0x11111111, 0x01101111
- $t6: 0x01101111, 0x11111111
- $t7: 1, 0, 1
- $t8: 0x10, 0x3FFFC48D
- $t9: 0xFFFFC48D, 0xFFFFF448

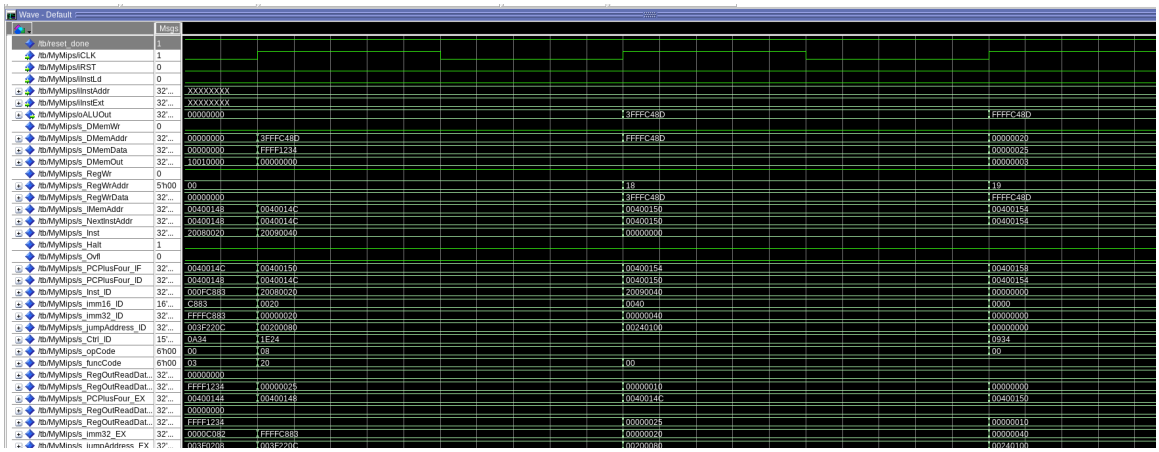
Steps:

1. lui: $1 = 0x00001001

2. ori: $sp = 0x00001001
3. addi: $t0 = 0x5
4. addi: $t1 = 0x10
5. add: $t0 = $t0 + $t1 = 0x15
6. addiu: $t0 = $t0 + 0x10 = 0x25
7. lui: $1 = 0x10
8. ori: $1 = 0x10
9. and: $t2 = $t0 & 0x20 = 0x20
10. andi: $t2 = $t2 & 0x0 = 0x0
11. lui: $t3 = 0x1001
12. sw: store 0x1001 to memory location 0x1000
13. lw: $t4 = memory location 0x1000 = 0x1001
14. nor: $t5 = not ($t4 or $t4) = 0x01101111
15. xor: $t5 = $t5 xor $t4 = 0x11111111
16. lui: $1 = 0x1001
17. xor: $13 = $13 xor $1
18. or: $t6 = $t5 or 0 = 0x01101111
19. lui: $1 = 0x1001
20. or: $t6 = $t6 or $t1 = 0x11111111
21. slt: $t7 = ($t5 < $t6) ? 1 : 0 = 1
22. slti: $t7 = ($t0 < 0x10) ? 1 : 0 = 0
23. addi: $t7 = 0x1
24. sll: $t8 = $t7 << 0x4 = 0x10
25. lui: $1 = 0xFFFF
26. ori: $t7 = 0xFFFF1234
27. srl: $t8 = $t7 >> 0x2 = 0x3FFFC48D
28. sra: $t9 = $t7 >> 0x2 = 0xFFFFC48D
29. addi: $t0 = 0x20
30. addi: $t1 = 0x40
31. sub: $t2 = $t1 - $t0 = 0x20
32. subu: $t2 = $t2 - $t0 = 0x0
33. addi: $t0 = 0x20
34. addi: $t1 = 0x19
35. lui: $t9 = 0xFFFF
36. ori: $t9 = 0xFFFFFFFF



addiu Instruction



SRL instruction

[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 of the software-scheduled pipelined processor can run at is 55.89 MHz.

The critical path goes through the following components:

- ID_EX_reg
- mainALU
- EX_MEM_reg
- MEM_WB_reg

- registerFile
- datamemory
- PC_reg
- s_Halt
- s_Ovfl
- iRST
- iInstLd
- iInstAddr
- iInstExt
- oALUOut

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

I'd probably use a MUX with a stall control signal, that feeds into the N- bit register. One of the MUX lines would be the data that has been fetched.

[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.

testbench is located under tests in the software-scheduling zip file.

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

Instructions	Corresponding Signals (Producing)
add, addu, and, nor	oALUOut
xor, or, slt, sll	oALUOut
srl, sra, sub, subu	oALUOut
addi, addiu, andi	s_RegWrData_ID, s_RegWrData_EX, s_RegWrData_MEM, s_RegWrData_WB, s_RegWrData_ID_J, s_RegWrData_ID_JR
xori, ori, slti, lui	s_RegWrData_ID, s_RegWrData_EX, s_RegWrData_MEM, s_RegWrData_WB, s_RegWrData_ID_J, s_RegWrData_ID_JR
jr	s_JmpDest_ID, s_JmpDest_EX, s_JmpDest_MEM, s_JmpDest_WB, s_JmpDest_ID_J, s_JmpDest_ID_JR
beq, bne	s_Branch_EX, s_Branch_ID_J, s_Branch_ID_JR
j	s_Jump_EX, s_Jump_ID_J, s_Jump_ID_JR, s_Jump_MEM, s_Jump_WB, s_Jump_ID
jal	s_Jump_EX, s_Jump_ID_J, s_Jump_ID_JR, s_Jump_MEM, s_Jump_WB, s_Jump_ID

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

Instructions	Corresponding Signals (Consuming)
add, addu, and, nor	s_ALUOut_ID, s_ALUOut_EX, s_ALUOut_MEM, s_ALUOut_WB
xor, or, slt, sll	s_ALUOut_ID, s_ALUOut_EX, s_ALUOut_MEM, s_ALUOut_WB
srl, sra, sub, subu	s_ALUOut_ID, s_ALUOut_EX, s_ALUOut_MEM, s_ALUOut_WB
addi, addiu, andi	s_ALUOut_ID, s_ALUOut_EX, s_ALUOut_MEM
xori, ori, slti, lui	s_ALUOut_ID, s_ALUOut_EX, s_ALUOut_MEM
jr	s_ALUOut_ID, s_ALUOut_EX, s_ALUOut_MEM, s_ALUOut_WB
beq, bne	s_ALUOut_ID, s_ALUOut_EX, s_ALUOut_MEM
j	s_Inst_ID, s_PCPlusFour_ID, s_NextInstAddr_ID, s_InstOrNOP_ID, s_PCPlusFour_ID_J, s_PCPlusFour_ID_JR, s_InstOrNOP_ID_J, s_InstOrNOP_ID_JR
jal	s_Inst_ID, s_PCPlusFour_ID, s_NextInstAddr_ID, s_InstOrNOP_ID, s_PCPlusFour_ID_J, s_PCPlusFour_ID_JR, s_InstOrNOP_ID_J, s_InstOrNOP_ID_JR

[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.

Producing signal: s_WBData

- Consuming signal: s_RegWrData
- Potential hazard stall: Yes

2. Producing signal: s_ALUOut

- Consuming signal: s_RegWrData
- Potential hazard stall: Yes

3. Producing signal: s_DMemOut

- Consuming signal: s_RegWrData
- Potential hazard stall: Yes

4. Producing signal: s_DMemOut

- Consuming signal: s_ALUInputB
- Potential hazard stall: Yes

5. Producing signal: s_PCPlusFour

- Consuming signal: s_IMemAddr
- Potential hazard stall: No (can be forwarded to the instruction memory stage)

6. Producing signal: s_JumpAddr

- Consuming signal: s_IMemAddr

- Potential hazard stall: No (can be forwarded to the instruction memory stage)

7. Producing signal: s_BranchAddr

- Consuming signal: s_IMemAddr
- Potential hazard stall: No (can be forwarded to the instruction memory stage)

8. Producing signal: s_PCPlusFour_ID

- Consuming signal: s_IMemAddr
- Potential hazard stall: No (can be forwarded to the instruction memory stage)

For dependencies that can be forwarded, the corresponding pipeline stages that will be forwarding and receiving the data are:

1. Producing stage: Execution

- Consuming stage: Instruction Memory

2. Producing stage: Execution

- Consuming stage: Instruction Memory

3. Producing stage: Memory

- Consuming stage: Instruction Memory

For dependencies that will require hazard stalls, the corresponding pipeline stages that will be stalling are:

1. Producing stage: Writeback

- Consuming stage: Register File Write

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

Pipeline Stage	Datapath Values	Control Signals
IF	s_PCPlusFour_IF	
ID	s_PCPlusFour_ID, s_Inst_ID, s_imm16_ID, s_imm32_ID, s_jumpAddress_ID	s_Ctrl_ID, s_opCode, s_funcCode, s_RegRd1Addr, s_RegRd2Addr, s_RegWrAddr, s_ALUSrc, s_ALUOp, s_MemWr, s_MemWrAddr, s_MemWrData, s_Branch, s_Jump, s_Jal, s_Halt
EX	s_ALUOut, s_RegWrData, s_MemWrAddr, s_MemWrData, s_JumpAddress, s_Halt	s_ALUOp, s_MemWr, s_Branch, s_Jump, s_Jal, s_Halt
MEM	s_DMemOut, s_MemWrData	s_MemWr, s_Halt
WB	s_RegWrData	s_RegWr, s_Halt

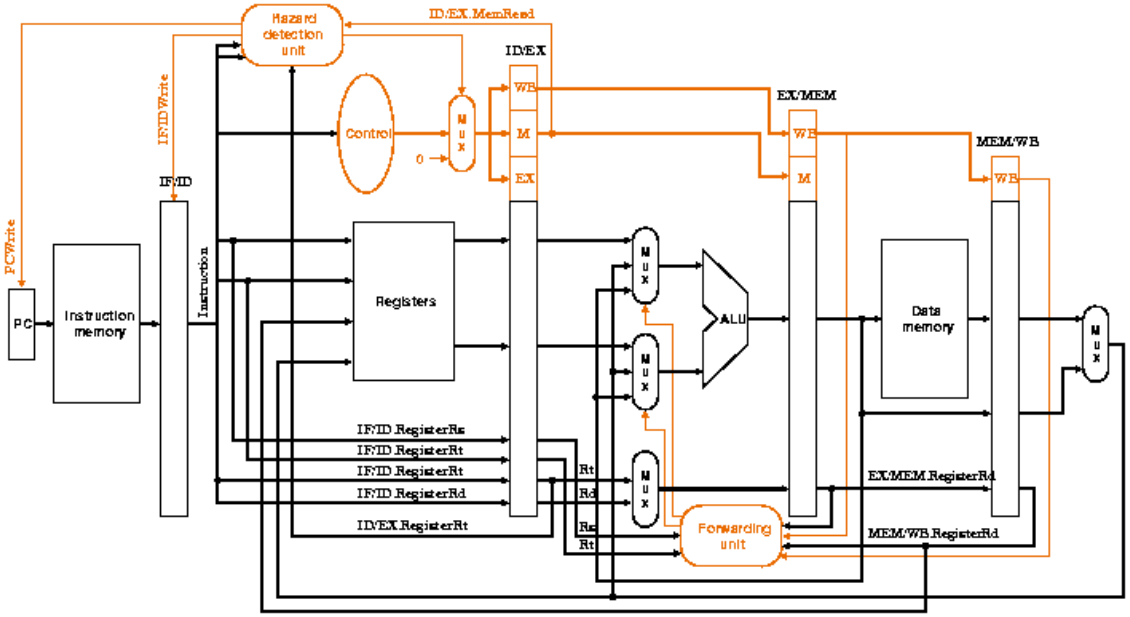
[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.

The instructions that may result in a non-sequential PC update are the jump and branch instructions, such as j, jal, and beq, bne. These instructions will update the PC in the ID pipeline 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.

The j, jal, and beq, bne instructions may result in non-sequential PC updates in the ID stage of the pipeline hence ideally, in order to maintain correct program execution, the pipeline must be stalled at the ID stage for these instructions. In order to correct for potential hazards, the pipeline must be flushed/squashed at the EX stage for these instructions. However for our design, the hazard detection takes place inside the fetch and hence no stall/flush would necessarily need to be implemented

[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.

[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.

Forwarding Type	Test Description	Test Case
IF -> ID	Data forwarding from the IF stage to the ID stage	`lw $t1, 0($t0)`
IF -> EX	Data forwarding from the IF stage to the EX stage	`lw $t1, 4($t0)` `lw $t2, 8($t0)` `sll $t1, $t1, 0x2`
IF -> MEM	Data forwarding from the IF stage to the MEM stage	`add $t3, $t2, $t2` `add $t4, $t1, $t1` `sw $t1, 0($t0)` `sw $t3, 8($t0)`
Structural hazard detection	Detecting conflicts in instruction execution	`lw $t1, 0($t0)` `lw $t2, 4($t0)` `lw $t3, 8($t0)`
WAW hazard detection	Detecting conflicts in register writes	`sw $t2, 4($t0)` `sw $t3, 8($t0)` `sw $t1, 0($t0)`
RAW hazard detection	Detecting conflicts in register reads	`add $t2, $t2, $t1` `add $t3, $t3, $t2` `addi $t1, $t1, 0x1`

The MIPS assembly code tests the data forwarding and hazard detection capabilities of the hardware-scheduled pipeline by testing all three types of data hazards: data dependency (dd), control dependency (cd), and structural hazard (sh).

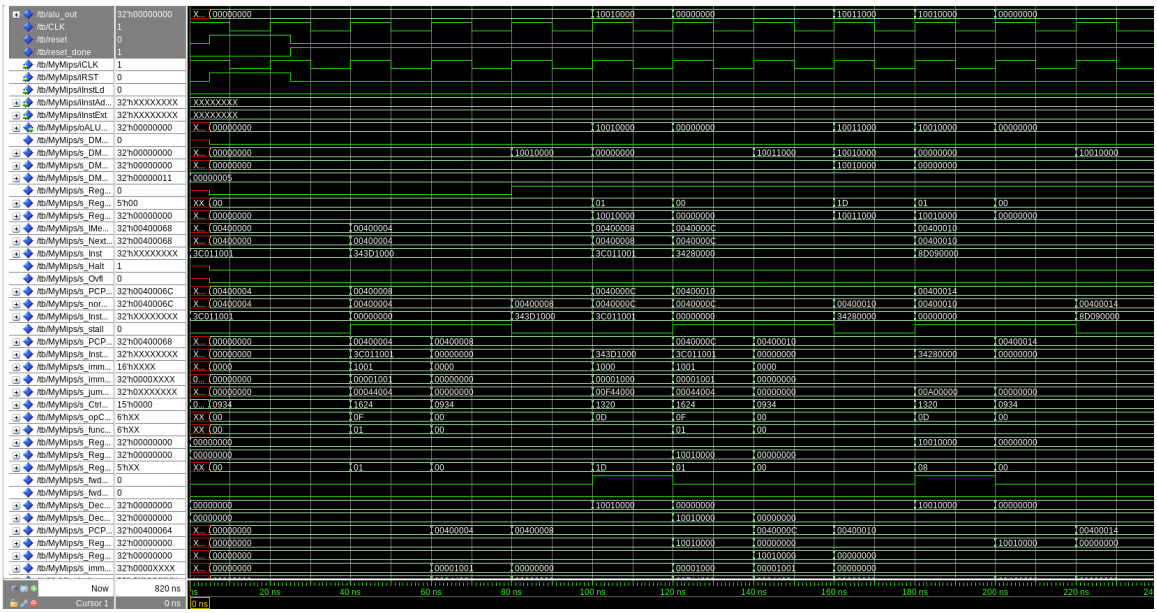
In the first test, the code tests for a dd hazard in the IF -> ID stage by loading a value from the array into a register and performing an ALU operation using the loaded value. This test will show whether the pipeline can correctly forward the data from the ID stage to the IF stage in order to prevent a data hazard.

In the second test, the code tests for a dd hazard in the IF -> EX stage by loading two values from the array into registers and performing an ALU operation using one of the loaded values. This test will show whether the pipeline can correctly forward the data from the EX stage to the IF stage in order to prevent a data hazard.

In the third test, the code tests for a dd hazard in the IF -> MEM stage by performing two ALU operations using the loaded values and storing the results of the ALU operations in the array. This test will show whether the pipeline can correctly forward the data from the MEM stage to the IF stage in order to prevent a data hazard.

In the fourth test, the code tests for all three types of hazards by loading three values from the array into registers and performing multiple ALU operations using the loaded values. This test will show whether the pipeline can correctly detect and handle data, control, and structural hazards in order to prevent any incorrect execution of instructions.

Overall, the MIPS assembly code provides adequate coverage for testing the forwarding and hazard detection capabilities of the hardware-scheduled pipeline.

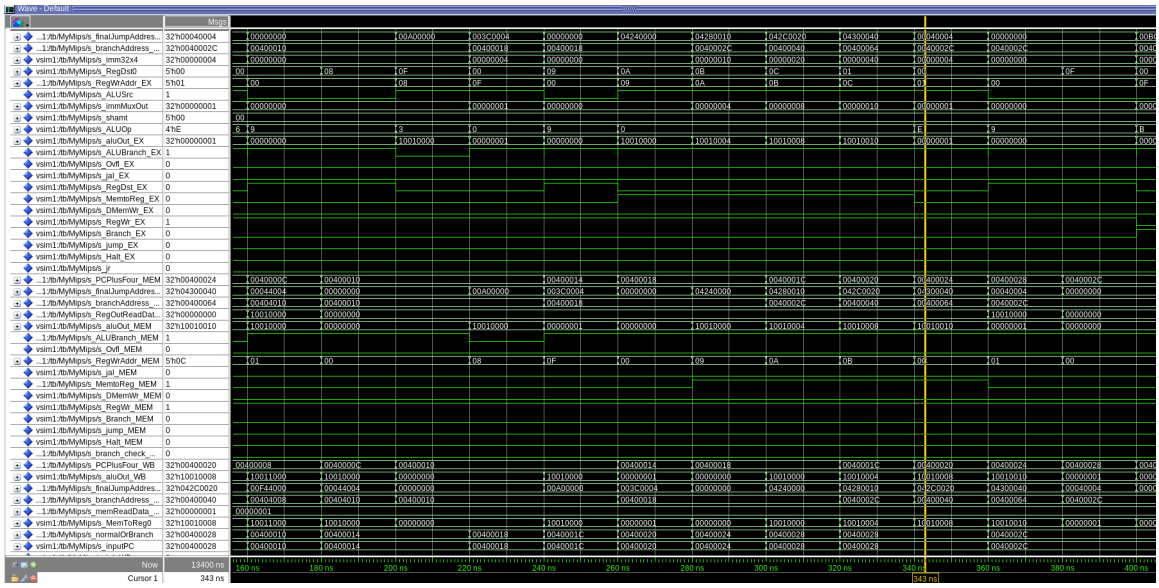


Stalls can be seen if you track PCPlusFour

[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 of Forwarding	Description of Test Case	Test Case
Squash	Basic branch taken	beq $t7, 0x1, odd
Squash	Basic branch not taken	beq $t5, 0x1, start
Stall	Branch taken after branch	beq $t7, 0x1, odd, beq $t5, 0x1, start
Stall	Branch not taken after branch	beq $t5, 0x1, start, beq $t7, 0x1, odd

The MIPS assembly code for control hazard detection tests the pipeline's ability to handle control hazards by using branch instructions and jumping to different parts of the code. The code includes both branch-on-equal and branch-on-not-equal instructions to test the pipeline's handling of different types of branch instructions. Additionally, the code uses a loop with a branch instruction inside to test the pipeline's ability to handle nested branch instructions. Overall, the code covers a range of scenarios involving control hazards, ensuring that the pipeline can handle these types of instructions properly.



[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).

Based on the timing information provided, the maximum frequency at which the hardware-scheduled pipelined processor can run is 52.90 MHz. The critical path goes from the pcReg register, into the Instruction memory, through the hazard detection unit, and ends at the IF/ID register.