CprE 381: Computer Organization and Assembly-Level Programming

Project Part 2 Report

Team Members: Kareem Eljaam

Nihaal Zaheer

Derek Lengemann

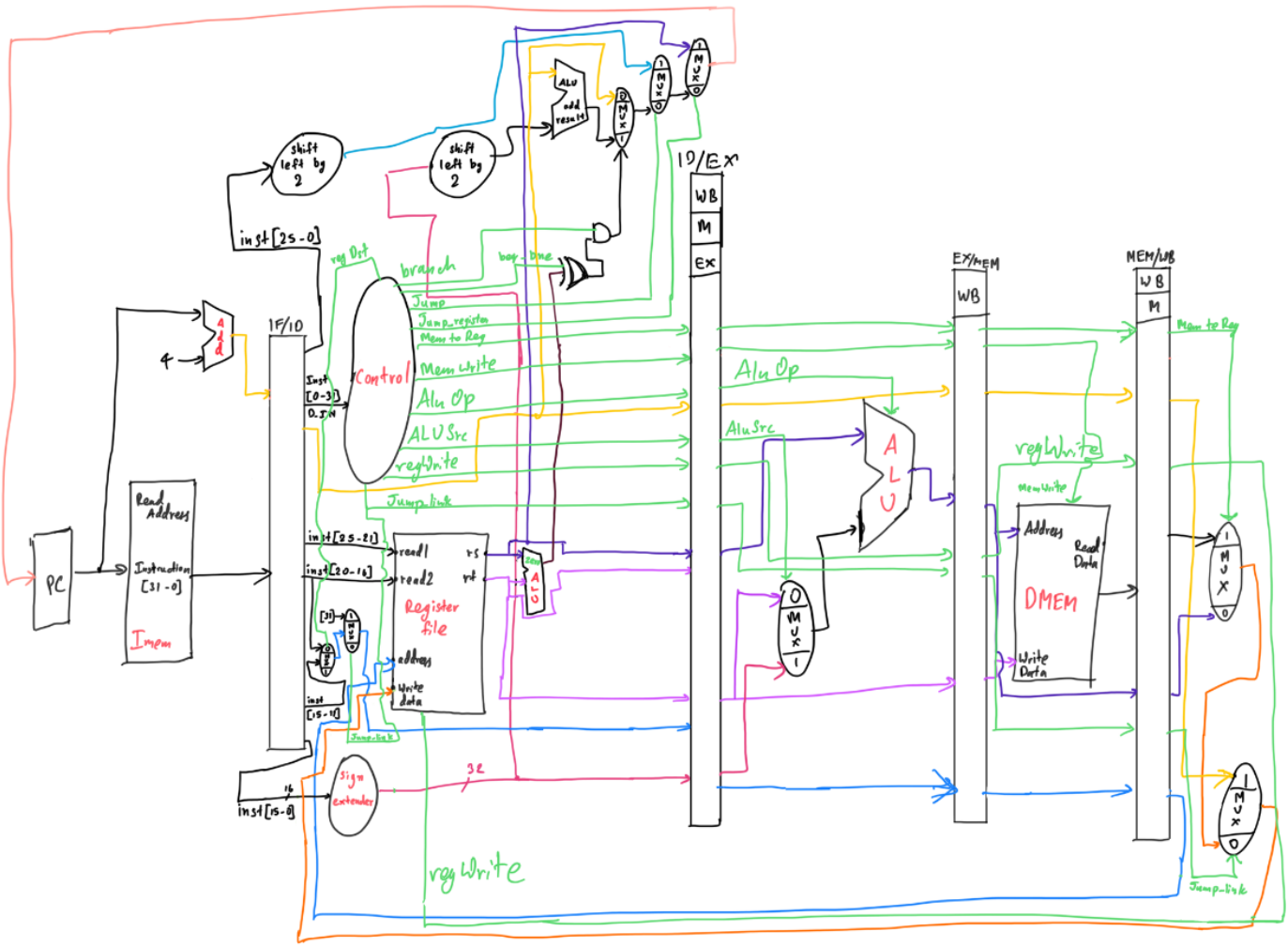
Project Teams Group #: 4-5

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.

Refer signals.odf file.

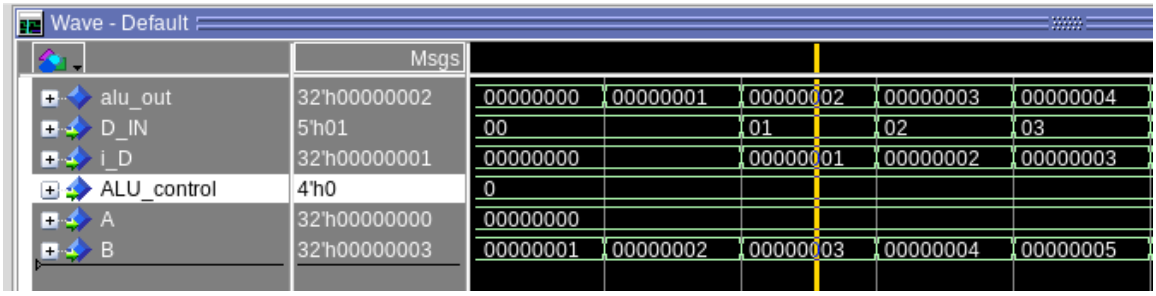
[1.b.ii] high-level schematic drawing of the interconnection between components.

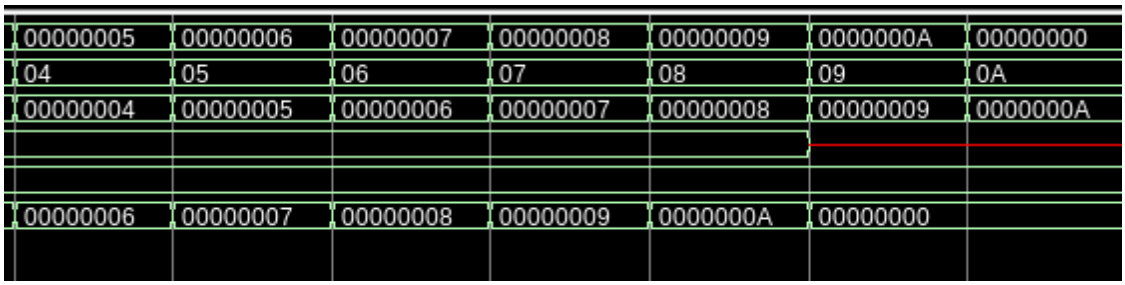


[1.c.i] include an annotated waveform in your writeup and provide a short discussion of result correctness.

Waveform for addiseq.s. This file does not contain any data hazards and has no data dependency, so this is a good resource to test the correctness of the pipeline.

```
#
# First part of the Lab 3 test program
#

# data section
.data

# code/instruction section
.text
addi  $1,  $0,  1              # Place “1?? in $1
addi  $2,  $0,  2              # Place “2?? in $2
addi  $3,  $0,  3              # Place “3?? in $3
addi  $4,  $0,  4              # Place “4?? in $4
addi  $5,  $0,  5              # Place “5?? in $5
addi  $6,  $0,  6              # Place “6?? in $6
addi  $7,  $0,  7              # Place “7?? in $7
addi  $8,  $0,  8              # Place “8?? in $8
addi  $9,  $0,  9              # Place “9?? in $9
addi  $10, $0,  10             # Place “10?? in $10

halt
```





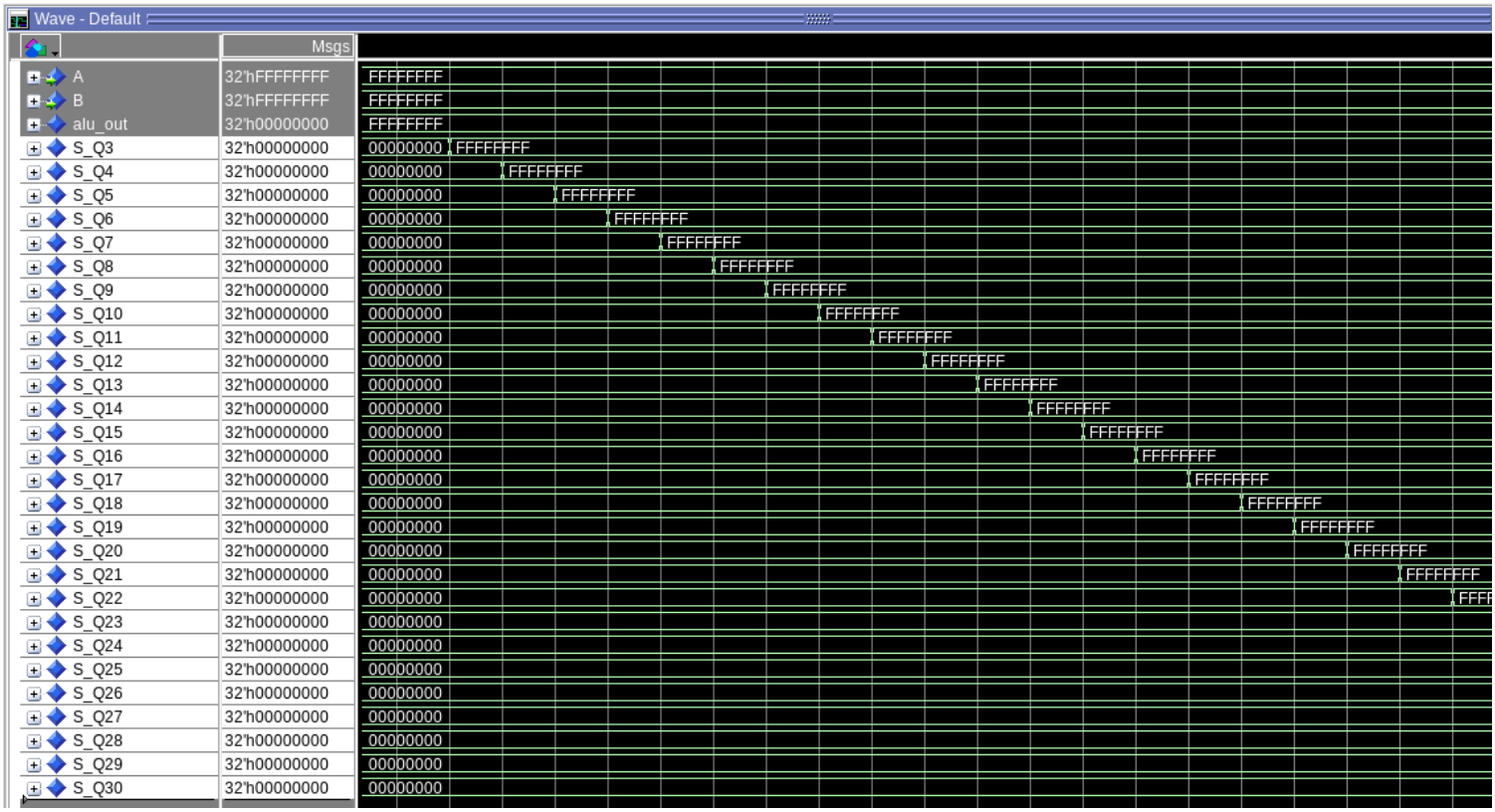
This code adds data and stores it to registers. This can be confirmed from the waveform.

[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.

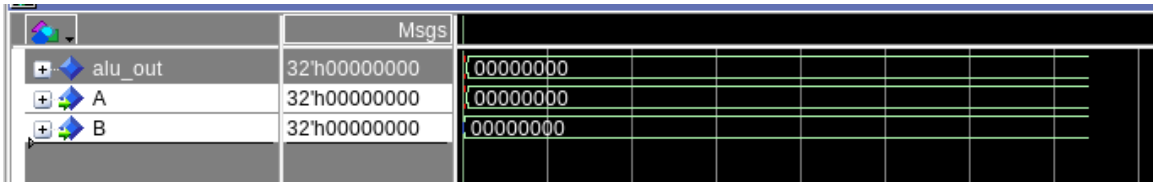
addiseq.s × Proj1_bubblesort.s × andi_0.s × and_0.s

```
.data
.text
.globl main
main:
    # Start Test
    addi $1, $1, 0xFFFFFFFF
    addi $2, $2, 0xFFFFFFFF
    nop
    nop
    nop

    and $3, $1, $2      # verify that anding 1 and 1 results in a 1
    and $4, $1, $2      # verify that anding 1 and 1 results in a 1
    and $5, $1, $2      # verify that anding 1 and 1 results in a 1
    and $6, $1, $2      # verify that anding 1 and 1 results in a 1
    and $7, $1, $2      # verify that anding 1 and 1 results in a 1
    and $8, $1, $2      # verify that anding 1 and 1 results in a 1
    and $9, $1, $2      # verify that anding 1 and 1 results in a 1
    and $10, $1, $2      # verify that anding 1 and 1 results in a 1
    and $11, $1, $2      # verify that anding 1 and 1 results in a 1
    and $12, $1, $2      # verify that anding 1 and 1 results in a 1
    and $13, $1, $2      # verify that anding 1 and 1 results in a 1
    and $14, $1, $2      # verify that anding 1 and 1 results in a 1
    and $15, $1, $2      # verify that anding 1 and 1 results in a 1
    and $16, $1, $2      # verify that anding 1 and 1 results in a 1
    and $17, $1, $2      # verify that anding 1 and 1 results in a 1
    and $18, $1, $2      # verify that anding 1 and 1 results in a 1
    and $19, $1, $2      # verify that anding 1 and 1 results in a 1
    and $20, $1, $2      # verify that anding 1 and 1 results in a 1
    and $21, $1, $2      # verify that anding 1 and 1 results in a 1
    and $22, $1, $2      # verify that anding 1 and 1 results in a 1
    and $23, $1, $2      # verify that anding 1 and 1 results in a 1
    and $24, $1, $2      # verify that anding 1 and 1 results in a 1
    and $25, $1, $2      # verify that anding 1 and 1 results in a 1
    and $26, $1, $2      # verify that anding 1 and 1 results in a 1
    and $27, $1, $2      # verify that anding 1 and 1 results in a 1
    and $28, $1, $2      # verify that anding 1 and 1 results in a 1
    and $29, $1, $2      # verify that anding 1 and 1 results in a 1
    and $30, $1, $2      # verify that anding 1 and 1 results in a 1
    # End Test

    # Exit program
    halt
```

This is the test for and_0.s, we don’t use nops here.
The waveforrm is as shown:


Wave - Default
Msgs
A
B
alu_out
S_Q3
S_Q4
S_Q5
S_Q6
S_Q7
S_Q8
S_Q9
S_Q10
S_Q11
S_Q12
S_Q13
S_Q14
S_Q15
S_Q16
S_Q17
S_Q18
S_Q19
S_Q20
S_Q21
S_Q22
S_Q23
S_Q24
S_Q25
S_Q26
S_Q27
S_Q28
S_Q29
S_Q30
32'hFFFFFFFF
32'h00000000
FFFFFFFF
00000000

```
.data
.text
.globl main
main:

    # Start Test
    sub $1, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $2, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $3, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $4, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $5, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $6, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $7, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $8, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $9, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $10, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $11, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $12, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $13, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $14, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $15, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $16, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $17, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $18, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $19, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $20, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $21, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $22, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $23, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $24, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $25, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $26, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $27, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $28, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $29, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $30, $0, $0     # verify that one can clear registers and 0-0 works in the ALU
    sub $31, $0, $0     # verify that one can clear registers and 0-0 works in the ALU

    # Exit program
    halt
```



This waveform is as expected.Sub_0.sWaveform is as expected, gives all 0s since we sub 0 with 0 produces 0, and that is stored in all registers.

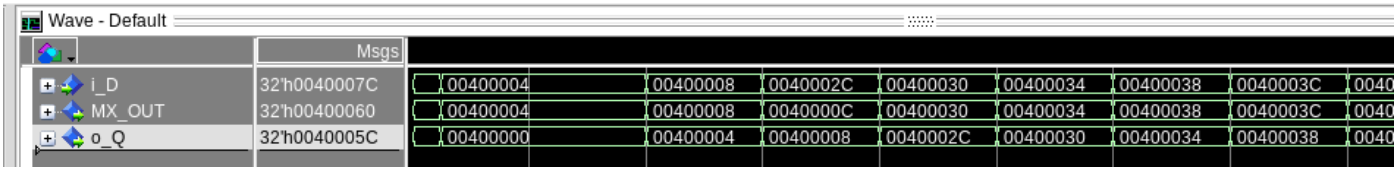
Simplebranch.s

addi_01.s × simplebranch.s ×

```
main:
        ori $s0, $zero 0x1234
        j skip
        Nop
        Nop
        #li $s0 0xffffffff
        lui $s0, 0xffff
        Nop
        Nop
        Nop
        ori $s0, $s0, 0xffff
        Nop
        Nop

skip:
        ori $s1 $zero 0x1234
        Nop
        Nop
        Nop
        beq $s0 $s1 skip2
        Nop
        #li $s0 0xffffffff
        lui $s0, 0xffff
        Nop
        Nop
        Nop
        ori $s0, $s0, 0xffff
skip2:
        jal fun
        Nop
        ori $s3 $zero 0x1234

        beq $s0, $zero exit
        Nop
        ori $s4 $zero 0x1234
        j exit
        Nop
        Nop

fun:
        Nop
        Nop
        ori $s2 $zero 0x1234
        jr $ra
        Nop

exit:
        halt
```

Here we use 1 or 2 Nops in places instead of 3 and the test still passes this shows we did not have to use maximum number of NOPS.

```
proj/mips/simplebranch.s     |  pass  |   pass    |    pass     |  1.47 | output/simplebranch.s
```



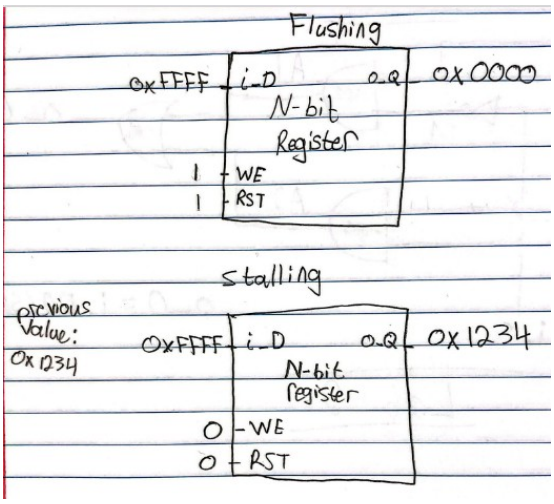
Here i_D is the input of the program counter, MX_out is the output of the branch MUX, and o_Q is the output of the PC, we see that the o_Q is shifted by one cycle, but is the same as the output of the branch MUX, thereby verifying the test.

[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: 42.46MHz.
The critical path should be branch, since it goes through Imem, then register, then ALU_zero which checks if difference between RS and RT are 0, it then goes through the branch and finally to the PC and then to the jump register, so the critical path also could be jump register

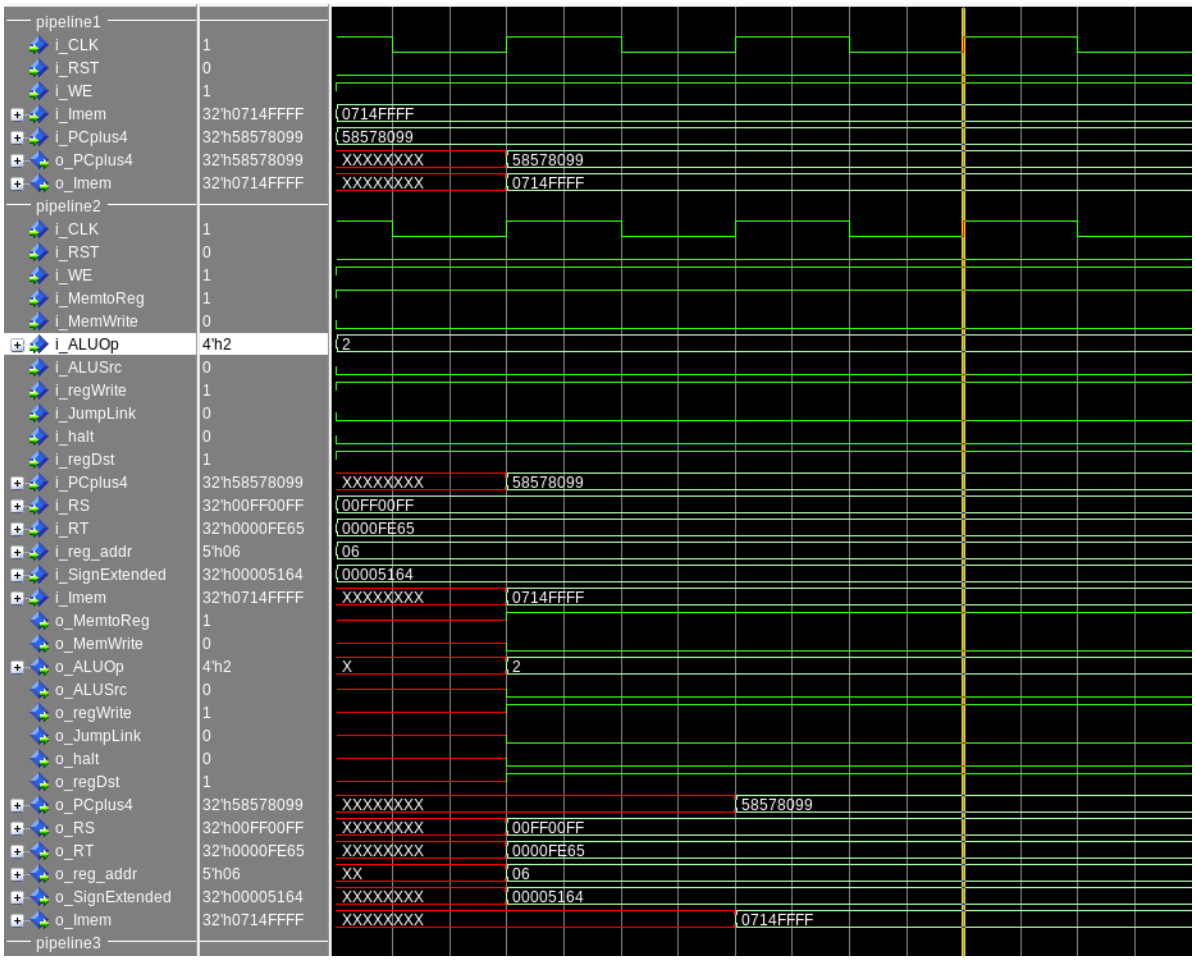
```
FMax: 42.46mhz Clk Constraint: 20.00ns Slack: -3.55ns

The path is given below

 ===================================================================
 From Node    : mem:IMem|altsyncram:ram_rtl_0|altsyncram_eg81:auto_generated|
ram_block1a21~porta_we_reg
 To Node      : pc:PC_0|s_Q[31]
 Launch Clock : iCLK
 Latch Clock  : iCLK
 Data Arrival Path:
 Total (ns)  Incr (ns)     Type  Element
```

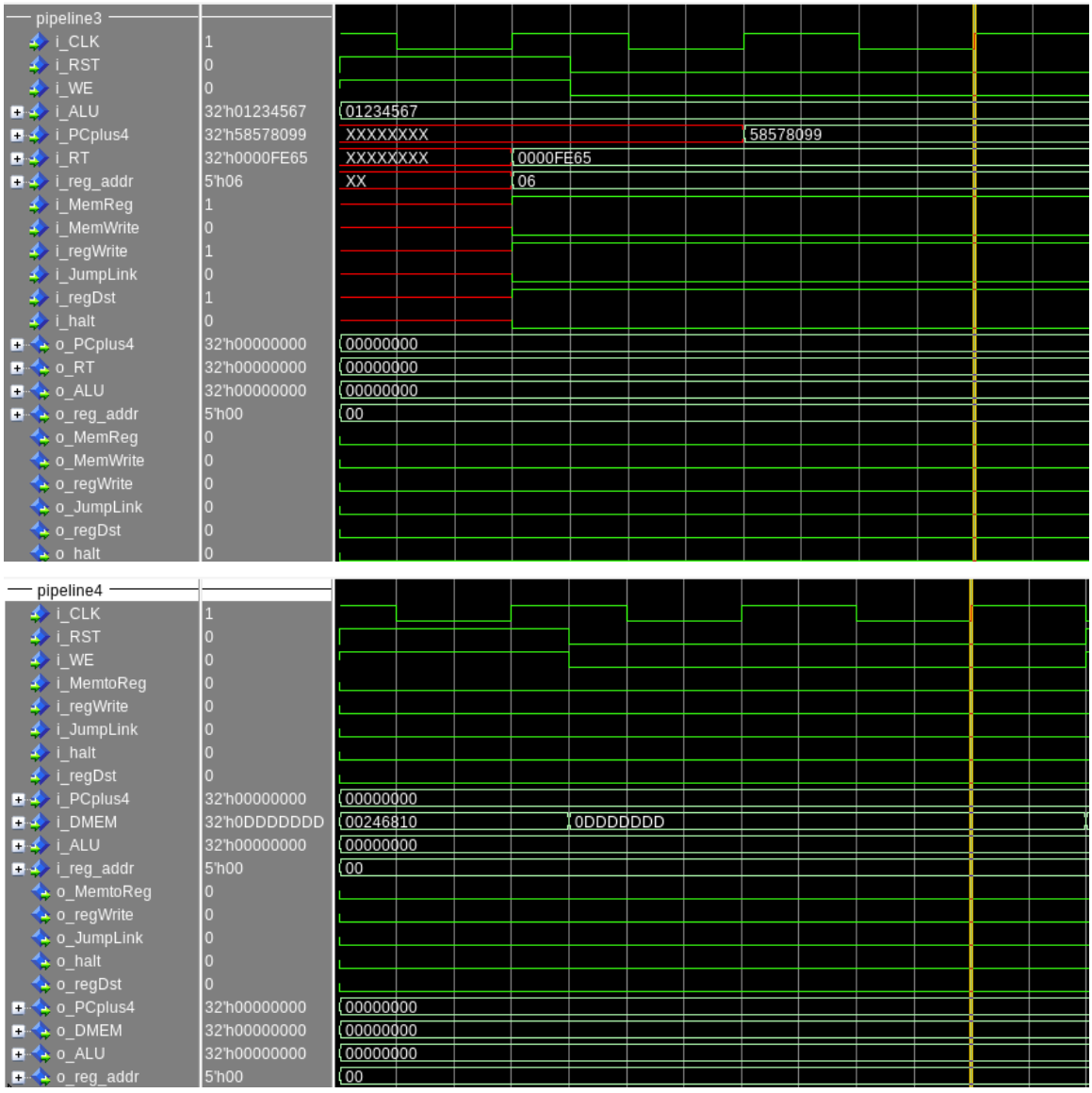
[2.a.ii] Draw a simple schematic showing how you could implement stalling and flushing operations given an ideal N-bit register.



[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 File: tb_pipelineReg.vhd





As we can see in the waveforms, the values of the signals get passed through to the next register. If the RST bit is set to ‘1’ in the register, then the values of the signals would become 0 (flushing). If both the RST and WE bits are set to ‘0’, then the signal values would not update (stalling).

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

Instructions that produce values:
Load instructions: lw, l, lui, etc.
Arithmetic instructions: add, sub, xor, and, or, etc.
Set Instructions: slt, etc.

Bus names that produce signals:
Instruction: carries data from 32 bit instructions
Data bus: carries data signals between components
Address bus: carries address to memory location like Imem and Dmem.
Control Bus: carries control signals to all the components.

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

Consume Values:
Memory Access: These instructions reads or writes data to pr from the data memory, that is the Dmem signal.
RegWrite: Writes the result of the ALU to the reg file.
Arithmetic operations like add, sub, and etc also consume values and is written to the execution stage.
Load instruction: LW,LUI,L, these consume values from the Dmem and is written in the execute stage.
Store instructions: Sw, Consumes values from Dmem and can be written to in the Write back stage.

[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.

generalized list of potential data dependencies in a hardware scheduled pipline.

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.

Generalized list of potential data dependencies:
1. Instruction register to decode register
2. Decode register to arithmetic and logic unit (ALU)
3. Arithmetic and logic unit (ALU) to memory address
4. Memory address register to memory data register

Dependencies that can be forwarded:
1. Instruction register to decode : Forwarding from instruction to decode in the Fetch stage.
2. Decode to arithmetic and logic unit (ALU): Forwarding from decode to ALU in the Decode stage.

3. Arithmetic and logic unit (ALU) to memory address register (MAR): Forwarding from ALU to MAR in the Execute stage.

Dependencies that will require hazard stalls:
1. Memory address to memory data : Stalling in the Memory stage.
2. Dmem to regfile: Stalling in the Writeback stage.

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

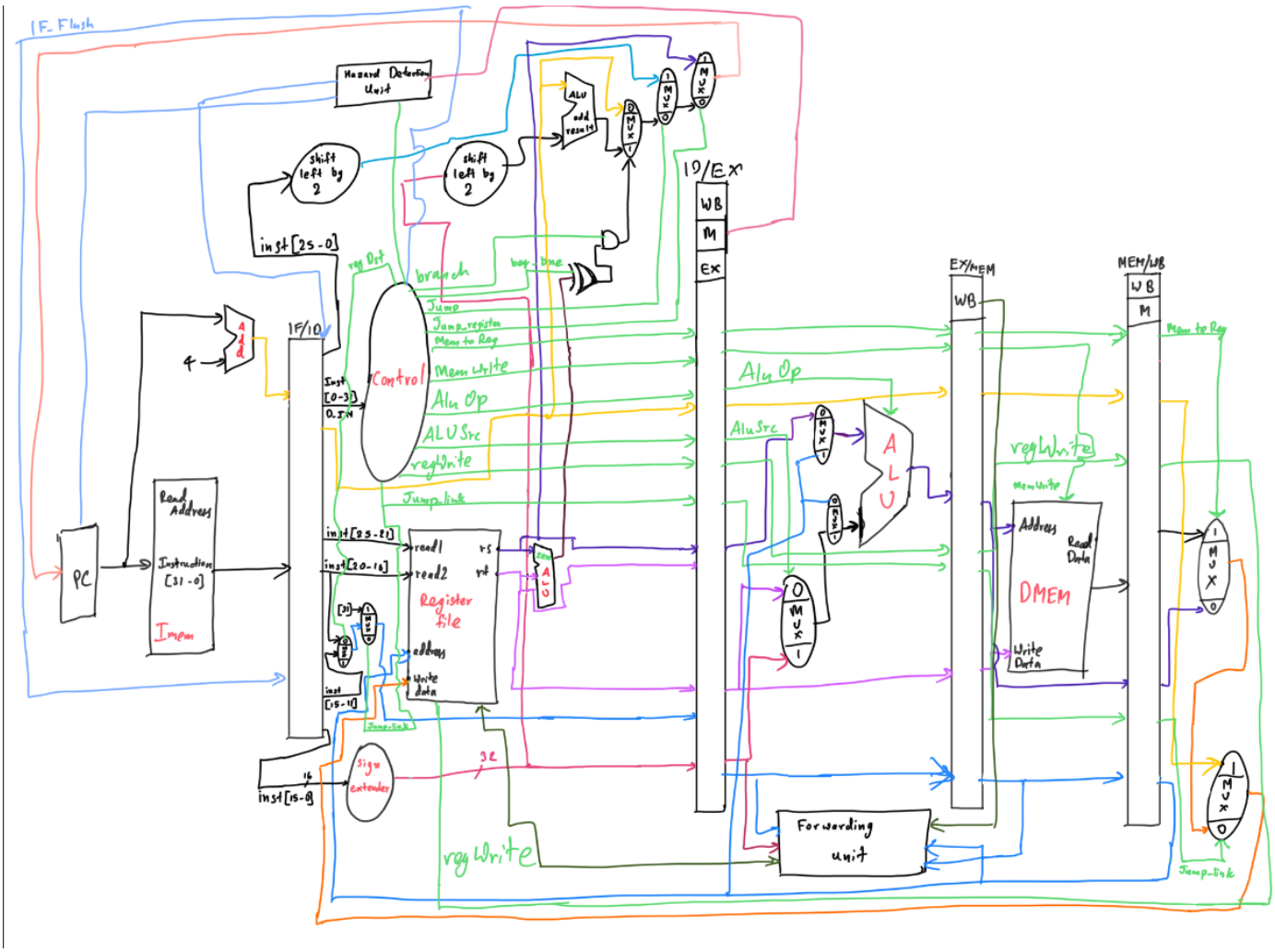
IF/ID			
Datapath Values		Control Signals	
Instruction	32		
PC+4	32		

ID/EX			
Datapath Values		Control Signals	
RS	32	MemToReg	1
RT	32	ALUOp	4
Extended Immediate	32	ALUSrc	1
PC + 4	32	MemWrite	1
Instruction	32	regWrite	1
reg file address	5	JumpLink	1
		regDst	1
		halt	1

EX/MEM			
Datapath Values		Control Signals	
ALU	32	MemToReg	1
RT	32	MemWrite	1
PC+4	32	regWrite	1
reg file address	5	JumpLink	1
		regDst	1
		halt	1

MEM/WB			
Datapath Values		Control Signals	
Data Memory	32	MemToReg	1
ALU	32	regWrite	1
PC + 4	32	JumpLink	1
reg file address	5	regDst	1
		halt	1

[2.c.i] list all instructions that may result in a non-sequential PC update and in which pipeline stage that update occurs.
J EX
jal EX
jr EX
bne MEM
beq MEM
[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 the three j instructions when the instruction enters the Decoding stage the IF/ID register needs to be flushed. No other register needs to be flushed. For beq and bne if a previous instruction needs something that it uses it IF/ID may need to be stalled. When the branch is in Instruction Decode step, IF/ID will need to be flushed for two clock cyles in order for the possible branch to go through the ALU and compute whether there is a branch or not.
[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.

[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.

[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).