1. Consider how far `jal` instructions can jump.

(a) How many instructions can a `jal` instruction jump forward (i.e., to higher addresses)?

`jal` can jump forward $(2^{(21-1)} - 1)/4 = 2^{18} - 1 =$ **262,143** instructions forward.

(b) How many instructions can a jal instruction jump backward (i.e., to lower addresses)?

`jal` can jump backward $-[2^{(21-1)}]/4 = -2^{18} =$ **262,144** instructions backward **.**

2. What is the maximum number of instructions that a branch instruction (like beq) can branch forward (i.e., to higher instruction addresses)?

Since all branches encode the *offset* in their immediate field, the can all always jump forward 1,023 instructions.

Specifically, a 13-bit signed (i.e,, two's complement) number can encode an offset of up to:

$2^{12} - 1$ bytes / (4 bytes/instruction) = [2^{10} instructions – 1instruction]/4.

But we can't have ¼ of an instruction, so it can encode a forward branch offset of up to 2^{10} instructions – 1 instruction = 1,023 instructions.

3. Write a RISC-V assembly language program for swapping the contents of two registers, x6 and x1. You may not use any other registers.

```
xor x6, x6, x1          # x6 = x6 XOR x1
xor x1, x6, x1          # a1 = original a0
xor a0, a0, a1          # a0 = original a1
```

4. Write a RISC-V assembly language program to *reverse the bits* in a register. Use as few instructions as possible.

a0 = register on which to reverse bits

```
reverseBits:
     addi t1, zero, 31          # t1 = 31
     addi t2, zero, 0           # t2 = 0
     addi t3, zero, 0           # t3 = 0
L7:
     beq t1, zero, done3        # if t1 = 0, done
     srl t0, a0, t1             # t0 = a0 >> t1
     andi t0, t0, 1             # isolate lsb of t0
     sll t0, t0, t2             # t0 = t0 << t2
     or t3, t3, t0              # combine bits
     addi t1, t1, -1            # t1--
     addi t2, t2, 1             # t2++
     j L7                       # repeat the loop
done3:
     add a0, t3, zero           # set a0 equal to its reversed self
     jr ra                      # return
```

5. Consider a function that takes a 10-entry array of 32-bit integers stored in little-endian format and converts it to big-endian format. Write this function in RISC-V assembly code. Comment all your code and use a minimum number of instructions.

```
# a0 = base address of arr, t0 = i

swapEndianness:

     addi t0, zero, 0              # i = 0

     addi t1, zero, 10             # t4 = 10 (temp value)

     lui t4, 0xFF0                 # t4 = 0xFF0000

     srli t3, t4, 8                # t3 = 0xFF00

L1:

     bge t0, t1, done              # if i >= 10 return

     slli t5, t0, 2                # t5 = i * 4 (byte offset)

     add t5, t5, a0                # t5 = address of arr[i]

     lw t6, 0(t5)                  # t6 = arr[i]

     slli a1, t6, 24               # a1 = arr[i] << 24

     and a2, t6, t3                # a2 = arr[i] & 0xFF00

     slli a2, a2, 8                # a2 =(arr[i] & 0xFF00) << 8

     and a3, t6, t4                # a3 = arr[i] & 0xFF0000

     srli a3, a3, 8                # a3 =(arr[i] & 0xFF0000) >> 8

     srli a4, t6, 24               # a4 = arr[i] >> 24

     or a0, a1, a2                 # a0 = combine most significant bytes

     or a0, a0, a3                 # a0 = combine 3 most significant bytes

     or a0, a0, a4                 # a0 = combine all bytes

     sw a0, 0(t5)                  # arr[i] = value with other endianness

     addi t0, t0, 1                # i++

     j L1                          # loop

done:
```

6. Each number in the Fibonacci series is the sum of the previous two numbers. Table below lists the first few numbers in the series, *fib*(*n*).

n	1	2	3	4	5	6	7	8	9	10	11	...
fib(n)	1	1	2	3	5	8	13	21	34	55	89	...

Write a function called `fib` in RISCV that *returns the Fibonacci number for any nonnegative value of n*. Use a simulator to test your code on *fib*(9). Clearly comment your code.

```
      addi a0, zero, 9              # n = 9
      jal fib                       # call fib(n), where n = 9
      ...                           # code after function call
fib:
      addi sp, sp, -12              # make room on stack for 3 registers
      sw s0, 8(sp)                  # save s0 on stack
      sw s1, 4(sp)                  # save s1 on stack
      sw s2, 0(sp)                  # save s2 on stack
      addi s0, zero, 0              # current = 0 (fib(i))
      addi s1, zero, 1              # prev = 1 (fib(i-1))
      addi s2, zero, 1              # i = 1
for:
      blt a0, s2, result            # if i > n then loop ends
      add s0, s0, s1                # fib(i) = fib(i - 1) + fib(i - 2)
      sub s1, s0, s1                # fib(i - 1) = fib(i) - fib(i - 2)
      addi s2, s2, 1                # i = i + 1
      j for                         # repeat loop
result:
      add a0, zero, s0              # return fib(n) (put fib(n) in a0)
      lw s0, 8(sp)                  # restore registers from stack
      lw s1, 4(sp)
      lw s2, 0(sp)
      addi sp, sp, 12               # restore stack pointer
      jr ra                         # return
```

7. Suppose that one of the following control signals in the single-cycle RISC-V processor has a stuck-at-0 fault, meaning that the signal is always 0 regardless of its intended value. What instructions would malfunction? Why? Use the extended version of the single-cycle processor shown in Figures below

(a) RegWrite

(b) ALUOp1

(c) ALUOp0

(d) MemWrite

(e) ImmSrc1

(f) ImmSrc0

(g) ResultSrc1

(h) ResultSrc0

(i) PCSrc

(j) ALUSrc

(a) **RegWrite:** lw, addi, jal, and R-type instructions – WE3 will be 0, so no data will be written to the Register File.

(b) **ALUOp1:** R-type instructions except add – These instructions all require a 1 in ALUOp1 for the ALU Decoder to produce the correct FsALUControl signal.

(c) **ALUOp0:** beq – The ALU would incorrectly add the registers rather than subtracting them before checking the Zero flag.

(d) **MemWrite:** sw - WE will not enable the Data Memory to write.

(e) **ImmSrc1:** beq and jal – The incorrect immediates (offsets) are selected.

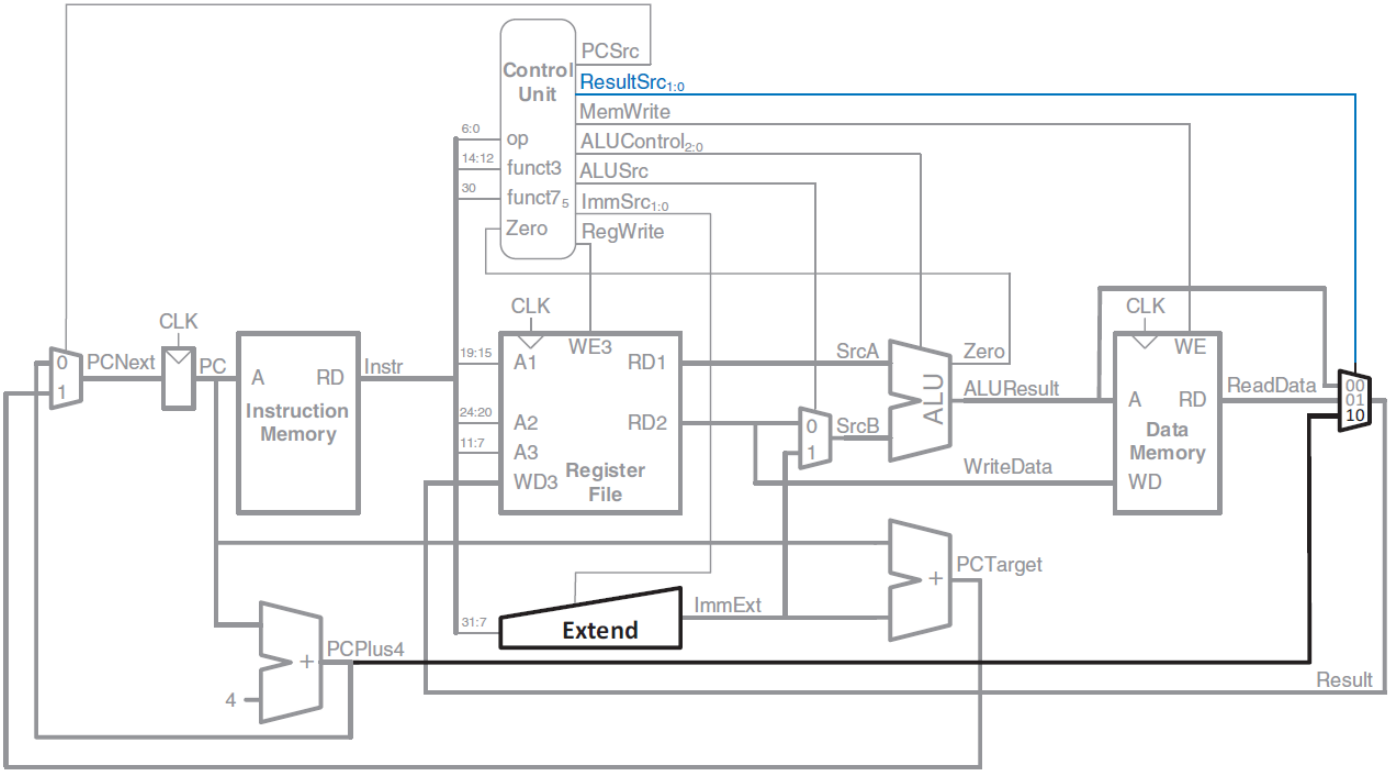
(f) **ImmSrc0:** sw and jal – The incorrect immediates are selected.

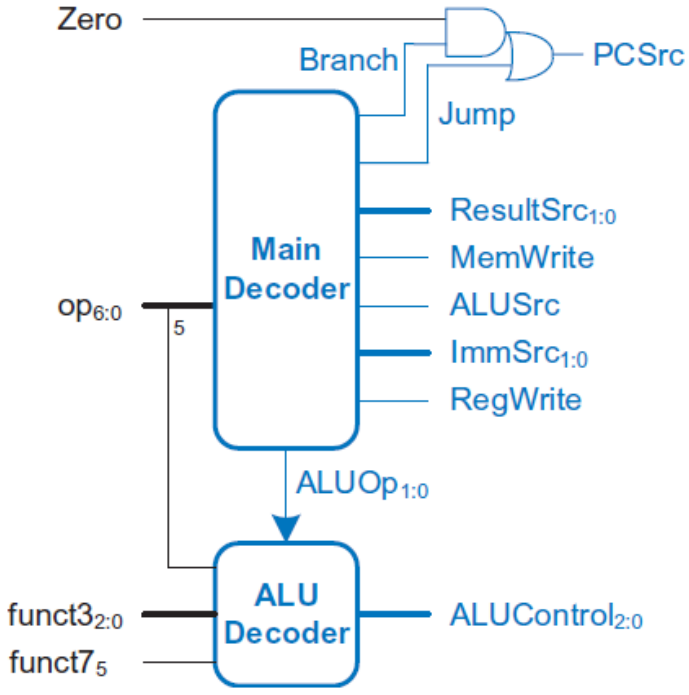
(g) **ResultSrc1:** jal – ALUResult will be selected instead of PCPlus4 as the result to write to the Register File.

(h) **ResultSrc0:** lw – ALUResult will be selected instead of ReadData as the result to write to the Register File.

(i) **PCSrc:** beq and jal – PCPlus4 will always be selected as PCNext instead of the new PCTarget.

(j) **ALUSrc:** lw, sw, and addi (and other I-type ALU instructions) – SrcB for the ALU will incorrectly select RD2 instead of ImmExt.


Control Unit
PCSrc
ResultSrc1:0
MemWrite
ALUControl2:0
ALUSrc
ImmSrc1:0
RegWrite
op
funct3
funct7 5
Zero
6:0
14:12
30
CLK
PCNext
PC
Instruction Memory
A
RD
Instr
19:15
24:20
11:7
A1
A2
A3
WD3
WE3
RD1
RD2
Register File
SrcA
SrcB
ALU
Zero
ALUResult
WriteData
Data Memory
WE
WD
ReadData
00
01
10
PCTarget
ImmExt
31:7
Extend
PCPlus4
4
Result


Zero
Branch
Jump
PCSrc
Main Decoder
op6:0
5
ResultSrc1:0
MemWrite
ALUSrc
ImmSrc1:0
RegWrite
ALUOp1:0
ALU Decoder
funct32:0
funct7 5
ALUControl2:0

8. Redesign one of the units in the single-cycle RISC-V processor to have half the delay. which unit should you work on to obtain the greatest speedup of the overall processor, and what would the cycle time of the improved machine be?

To increase performance most (i.e., decrease cycle time), the crack circuit designer should speed up the Memory Unit – because it has the largest delay component.

The new cycle time of this single cycle datapath would become:

$Tc\_new = tpcq\_PC + 2tmem + tRFread + tALU + tmux + tRFsetup$

= 40 + 2(100) + 100 + 120 + 30 + 60 = **550 *ps*** – a 200ps reduction since the improvements to both Instruction and Data Memory lower the total latency by 200ps

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	30
AND-OR gate	$t_{AND\text{-}OR}$	20
ALU	t_{ALU}	120
Decoder (control unit)	t_{dec}	25
Extend unit	t_{ext}	35
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

9. Suppose the multicycle RISC-V processor has the component delays given in Table of Previous Problem. You design a new register file that has 40% less power but twice as much delay. Should you switch to the slower but lower power register file for your multicycle processor design? Explain why.

You should switch to the slower but lower power register file. By doubling the delay of the register file, it still does not place it on the critical path. This means that power will be saved without affecting the cycle time.

Tmeme_stage = path through memory. (Tcq + Tmem + Tsetup) = 40 + 200+ 50 = 290ps

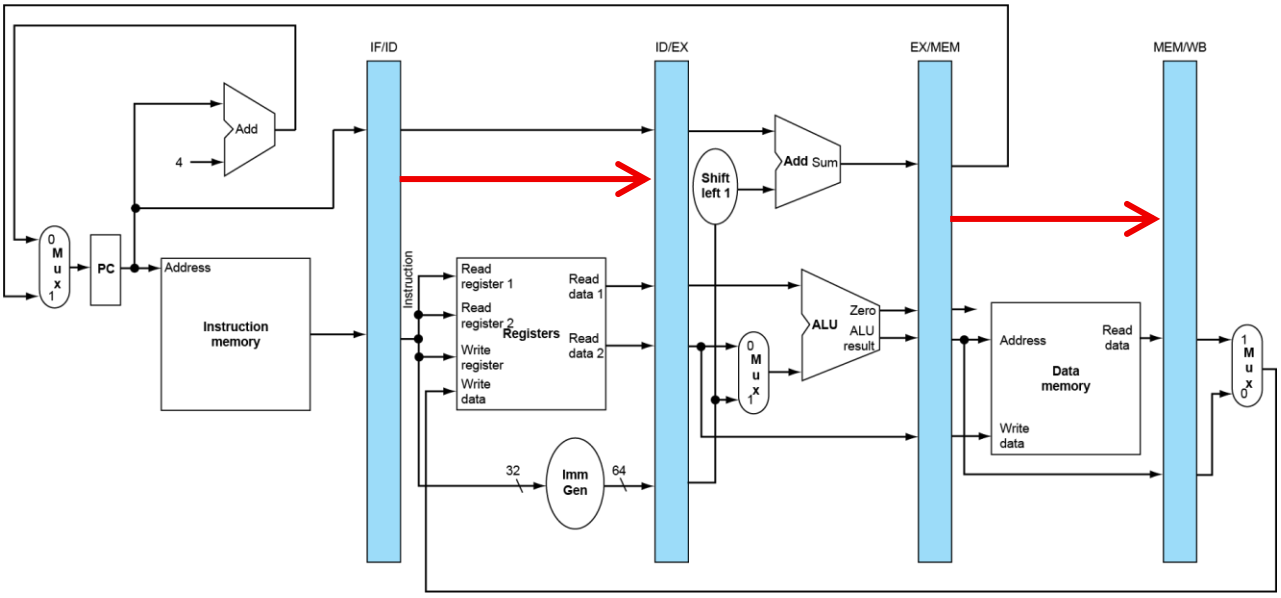
Specifically, the path that includes the register files would require the following constraints:

$Tc\_multi\_RF = Tcq + TRFread + Tsetup$

Tc_multi_RF = (40 + 100 + 50) ps = 190 ps

With twice as much register file (RF) delay, this constraint would be:

Tc_multi_RF = (40 + **2 ∗ 100** + 50) ps = 290 ps, which is still only equal to the 290 ps cycle time required by the path through memory



10. How many cycles are required to run the following program on the multicycle RISC-V processor? What is the CPI of this program? (CPI of `addi` = 4, `bge` = 3, `jal` = 4)

```
     addi x6, zero, 5          # x6 (result) = 5
L1:
     bge zero, x6, Done        # if result <= 0, exit loop
     addi x6, x6, -1           # result = result - 1
     j L1
Done:
```

The program will execute 6 `addi` (4 cycles each), 6 `bge` (3 cycles each), and 5 `jal` (4 cycles each) instructions for a total of (6×4) + (6×3) + (5×4) = **62 clock cycles** for 17 instructions. Thus, the CPI of this program is 62/17 = **3.65 CPI**. (Remember that j `L1` is a pseudoinstruction for `jal x6, L1`.)

11. The pipelined RISC-V processor is running the following code snippet. **Which registers are being written** and **which are being read** on the fifth cycle? Recall that the pipelined RISC-V processor has a Hazard Unit. You may assume a memory system that returns the result within one cycle.

```
xor x1, x2, x3     # x1 = x2 XOR x3

addi x5, x3, -4    # x5 = x3 - 4

lw x3, 16(x7)            # x3 = memory[x7+16]

sw x4, 20(x1)            # memory[x1+20] = x4

or x6, x5, x1            # x6 = x5 | x1
```

Pipeline stages

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	xor				
2	addi	xor			
3	lw	addi	xor		
4	sw	lw	addi	xor	
5	or	**sw**	lw	addi	**xor**

In cycle 5, the `sw` instruction is in the *Decode* stage and `xor` is in the *Writeback* stage. So, `x1` is written (by `xor`) during the first half of cycle 5. `x1` and `x4` are read (by the `sw`) during the second half of cycle 5.

Notice that `x1` is both *written* and *read* in cycle 5. Also note that *no hazards exist* in this code.

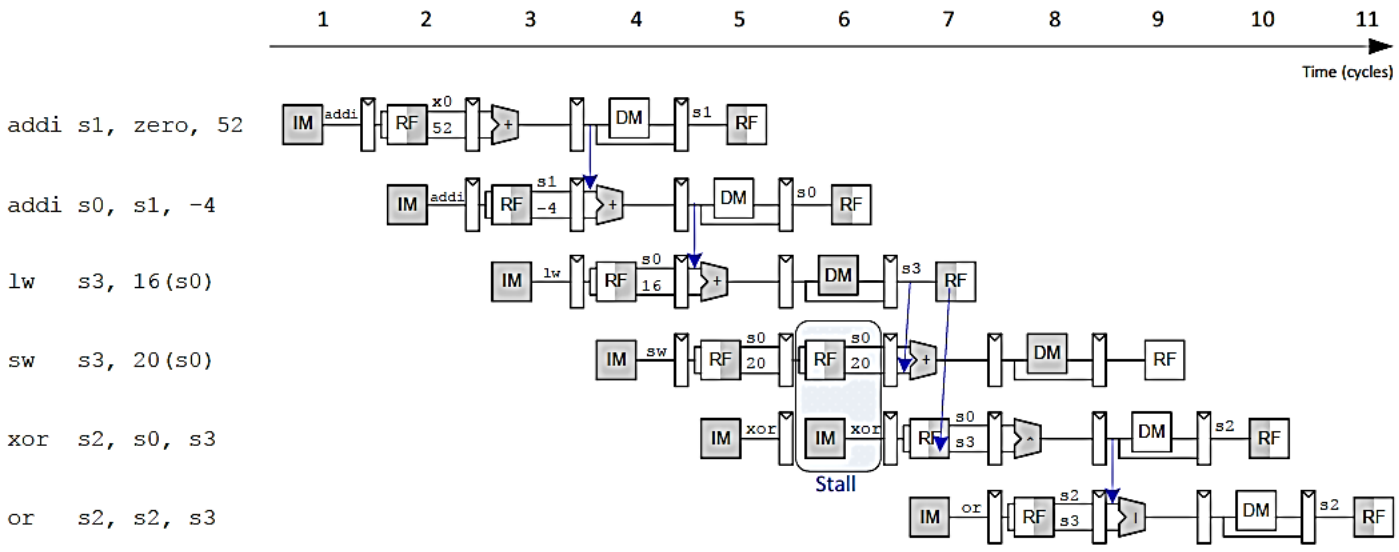
12. The pipelined RISC-V processor is running the following code snippet. **Which registers are being written** and **which are being read** on the fifth cycle? 7.31

```
addi x1, x0, 52      # x1 = 52
addi x6, x1, -4      # x6 = x1 - 4 = 48
lw x3, 16(x6)        # x3 = memory[64]
sw x3, 20(x6)        # memory[68] = x3
xor x2, x6, x3       # x2 = x6 XOR x3
or x2, x2, x3        # x2 = x2 | x3
```

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	addi x1, x0, 52				
2	addi x6, x1, -4	addi x1, x0, 52			
3	lw x3, 16(x6)	addi x6, x1, -4	addi x1, x0, 52		
4	sw x3, 20(x6)	lw x3, 16(x6)	addi x6, x1, -4	addi x1, x0, 52	
5	xor x2, x6, x3	sw **x3**, 20(**x6**)	lw x3, 16(x6)	addi x6, x1, -4	addi **x1**, x0, 52
6	xor x2, x6, x3	sw x3, 20(x6)	**bubble**	lw x3, 16(x6)	addi x6, x1, -4

In cycle 5, **x1** is being written (by addi) in the *Writeback* stage, and **x6** and **x3** are being read by **sw** in the *Decode* stage. Note that in this cycle, **sw** detects that it needs to stall in the next cycle – because a **lw** is in the Execute stage that will produce one of its source registers (x3), and the lw won't have that operand ready until the end of the Memory stage.

How many cycles are required for the pipelined RISC-V processor to issue all of the instructions for the program, What is the CPI of the processor on this program?



It takes **7 clock cycles** to issue all the instructions. # instructions = 6. CPI = 7 clock cycles / 6 instructions = **1.17**

13. Suppose the RISC-V pipelined processor is divided into *10 stages of 400 ps each*, including sequencing overhead. Assume the instruction mix in Table below. Also, assume that

50% of the loads are immediately followed by an instruction that uses the result, requiring *six stalls*, and that

30% of the branches are mis predicted. The target address of a branch instruction is *not computed until the end of the second stage*.

Calculate the average CPI and execution time of processing 100 billion instructions from the SPECINT2000 benchmark for this 10-stage pipelined processor

SPECINT2000 Benchmark	**Load**	**Store**	**Branch**	**Jump**	**R or I type ALU Instructions**
% of Benchmark	25	10	11	2	52

Loads take one clock cycle when there is no dependency and seven clock cycles when there is (load plus 6 stalls), so they have an average CPI of

$(0.5)(1) + (0.5)(7) = 4$

Branches take one clock cycle when they are predicted properly and two when they are not, so their average CPI is

$(0.7)(1) + (0.3)(2) = 1.3$

The remaining instructions have a CPI of 1. Hence, the average CPI for the SPECINT2000 benchmark is:

CPI $= 0.25(4) + 0.10(1) + 0.13(1.3) + 0.52(1) =$ **1.79 CPI.**

Execution time $= (100 \times 10^9$ instructions$)(1.79$ cycles/instruction$)(400 \times 10^{-12}$ s/cycle$)$

$=$ **71.6 seconds**

14. The Hewlett-Packard 2114, 2115, and 2116 used a format with the leftmost 16 bits being the fraction stored in *two's complement format*, followed by another 16-bit field which had the leftmost 8 bits as an extension of the fraction (making the fraction 24 bits long), and the *rightmost 8 bits representing the exponent*. However, in an interesting twist, *the exponent was stored in sign-magnitude format with the sign bit on the far right!*

Write down the bit pattern using the HP format to represent -1.5625×10^{-1}

[assume $-1.5625 \times 10^{-1} = -0.15625 \times 10^{0} = -0.00101 \times 2^{0} = -0.101 \times 2^{-2}$]

No hidden 1 is used (no '1' to the left of the binary point). Comment on how the range and accuracy of this 32-bit pattern compares to the single precision IEEE 754 standard.

Write down the bit pattern to represent -1.5625×10^{-1} assuming this format. *No hidden 1 is used*. Comment on how the range and accuracy of this 32-bit pattern compares to the single precision IEEE 754 standard.

$-1.5625 \times 10^{-1} = -0.15625 \times 10^{0}$
$= -0.00101 \times 2^{0}$
move the binary point two positions to the right
$= -0.101 \times 2^{-2}$

The (absolute value of the) fraction is 0101 (we must write the leading zero to represent 2s complement sign)

0101 0000 0000 0000 0000 0000 [24 bits for fraction field – representing the absolute value of the fraction: 0.101]

Taking the 2s complement – to negate this 24 bit number (since the fraction is negative = -0.101)

Step 1: reverse the bits
1010 1111 1111 1111 1111 1111
Step 2: add 1
1010 1111 1111 1111 1111 1111
+0000 0000 0000 0000 0000 0001

1011 0000 0000 0000 0000 0000

exponent = -2,
using the 8 bits for the exponent field with right most bit corresponding to the sign bit: **0000 0101**

Thus, 32-bit string: 10110000000000000000000000000101

15. Problems in this exercise refer to the following sequence of instructions, and assume that it is executed on a five-stage pipelined datapath:

add x15, x12, x11

ld x13, 4(x15) **EX to 1st RAW Hazard**

ld x12, 0(x2)

or x13, x15, x13 **MEM to 2nd RAW Hazard**

sd x13, 0(x15) **EX to 1st RAW Hazard**

15.1 If there is no forwarding or hazard detection, insert NOPs to ensure correct execution.

add x15, x12, x11

nop

nop **EX to 1st RAW Hazard resolution with 2 NOPs**

ld x13, 4(x15)

ld x12, 0(x2)

nop **MEM to 2nd RAW Hazard resolution with 1 NOP**

or x13, x15, x13

nop

nop **EX to 1st RAW Hazard resolution with 1 NOP**

sd x13, 0(x15)

15.2 Now, change and/or rearrange the code to minimize the number of NOPs needed. You can assume register x17 can be used to hold temporary values in your modified code.

not possible to reduce the number of NOPs.

15.3 If the processor has forwarding, but we forgot to implement the hazard detection unit, what happens when the original code executes?

- ❑ The code executes correctly.
- ❑ Hazard detection relevant only to insert a stall for load-use-data hazards
- ❑ The given instruction sequence does not have **ld** followed by use of register written into

16. Consider the following loop.

```
LOOP: ld   x10, 0(x13)
      ld   x11, 8(x13)
      add  x12, x10, x11
      subi x13, x13, 16
      bnez x12, LOOP
```

Assume that perfect branch prediction is used (no stalls due to control hazards), that there are no delay slots, that the pipeline has full forwarding support, and that branches are resolved in the EX (as opposed to the ID) stage.

16.1 Show a pipeline execution diagram for the first two iterations of this loop

16.2 Mark pipeline stages that do not perform useful work. How often while the pipeline is full do we have a cycle in which all five pipeline stages are doing useful work? (Begin with the cycle during which the subi is in the IF stage. End with the cycle during which the bnez is in the IF stage.)

[1] Since perfect branch prediction is used, we do not lose any cycles due to branch hazards – that is, the branch is always predicted and taken correctly

[2] Availability of full forwarding support is assumed, so we can assume data hazards that can be resolved with forwarding do not stall the pipeline

[3] Load-use-hazards cannot be resolved and are identified in next slide in red boxes - resolved by stalling the pipeline

[4a] pipeline stages unused by any instruction are identified in blue

[4b] any clock cycle that does not have all of the pipeline stages utilized is identified with 'N'