CSM 61C Spring 2020

Pipelining and Hazards
Exam Question Compilation Solutions

This document is a PDF version of old exam questions by topic, ordered from least difficulty to greatest difficulty.

Questions:

- Fall 2019 Final Q7
- Summer 2018 Final Q5
- Summer 2018 Midterm 2 Q4
- Spring 2018 Midterm 2 Q4
- Fall 2017 Midterm 2 Q4

Q7) *RISCV Exam-isim Debug – Pipelined* (18 pts = 3 + 6 + 3 + 6)

After solving your datapath bug, you decide to introduce the traditional five-stage pipeline into your processor. You find that your unit tests with single commands work for all instructions, and write some test patterns with multiple instructions. After running the test suite, the following cases fail. You should assume registers are initialized to 0, the error condition is calculated in the fetch stage, and no forwarding is currently implemented.

Case 1: Assume the address of an array with all different values is stored in `s0`.

```
addi  t0 x0 1
slli  t1 t0 2
add   t1 s0 t1
lw    t2 0(t1)
```

Each time you run this test, there is the same incorrect output for `t2`. All the commands work individually on the single-stage pipeline.
Pro tip: you shouldn't even need to understand what the code does to answer this.

a) What caused the failure? (select ONE)	**b)** How could you fix it? (select all that apply)
○ Control Hazard	[✓ Insert a nop 3 times if you detect this specific error condition]
○ Structural Hazard	☐ Forward execute to write back if you detect this specific error condition
[✓ Data Hazard]	☐ Forward execute to memory if you detect this specific error condition
○ None of the above	[✓ Forward execute to execute if you detect this specific error condition]
	☐ Flush the pipeline if you detect this specific error condition

The issue with the above code is the use of a register (aka we get the value during the decode phase) before we have written back the value of the previous instruction. This is a data hazard as the data which we want is not restored to the regfile. This means that we would have the current instruction in the execute phase while we have the previous in decode. This means that the next cycle, we would have to forward the execute output to the execute input to make sure the value is the correct, updated one. Inserting a nop when you realize this error happens will allow the system to do the write back. The other forwards in this problem are necessary for the given code above. Flushing the pipeline does not work as it means that we will no longer execute the instructions which were flushed. This means we would just drop instructions which would not get the correct value instead of just waiting till they can get the correct value.

Case 2: *After fixing that hazard,* the following case fails:

```
        addi  s0 x0 4
        slli  t1 s0 2
        bge   s0 x0 greater
        xori  t1 t1 -1
        addi  t1 t1 1
greater:
        mul t0 t1 s0
```

When this test case is run, `t0` contains `0xFFFFFFC0`, which is not what it should have been.
Pro tip: you shouldn't even need to understand what the code does to answer this.

c) What caused the failure? (select ONE)	**d)** How could you fix it? (select all that apply)
[✓ Control Hazard]	[✓ Insert a nop 3 times if you detect this specific error condition]
Structural Hazard	Forward execute to write back if you detect this specific error condition
Data Hazard	Forward execute to memory if you detect this specific error condition
None of the above	Forward execute to execute if you detect this specific error condition
	[✓ Flush the pipeline if you detect this specific error condition]

The issue with the code above is we do not clear/flush the instructions if the branch determines it is taken. Remember that we are running on a five stage pipeline CPU which just assumes PC + 4 unless an instruction says otherwise. This means that we will not determine if the branch is taken until the branch is in the execute phase. This means that we will have the next two instructions already in the pipeline (one in instruction fetch, the other in instruction decode). So we have a Control Hazard as we are not executing the correct instructions. Some ways how to fix it: insert nops if you detect a branch instruction in the instruction fetch stage OR flush the pipeline if the branch is in the opposite direction of what was predicted. Forwarding data in this case will not help at all.

Some other notes about this. The [incorrect] value we got was -64. How did we get this? Well if we look at the code, we see that s0 will hold 4 and t1 will hold 16. Due to the control hazard, this means we will execute and write back the following two instructions before we execute the mul. This means that we will flip the bits (xori, note that -1 means all bits are 1 due to twos complement encoding) and add one (addi). This is just twos complement inversion! This is what causes us to get -16 which multiplied by 4 will give us -64. In a correct implementation, we would not have executed the xori and addi thus have gotten 64.

SID: ________________

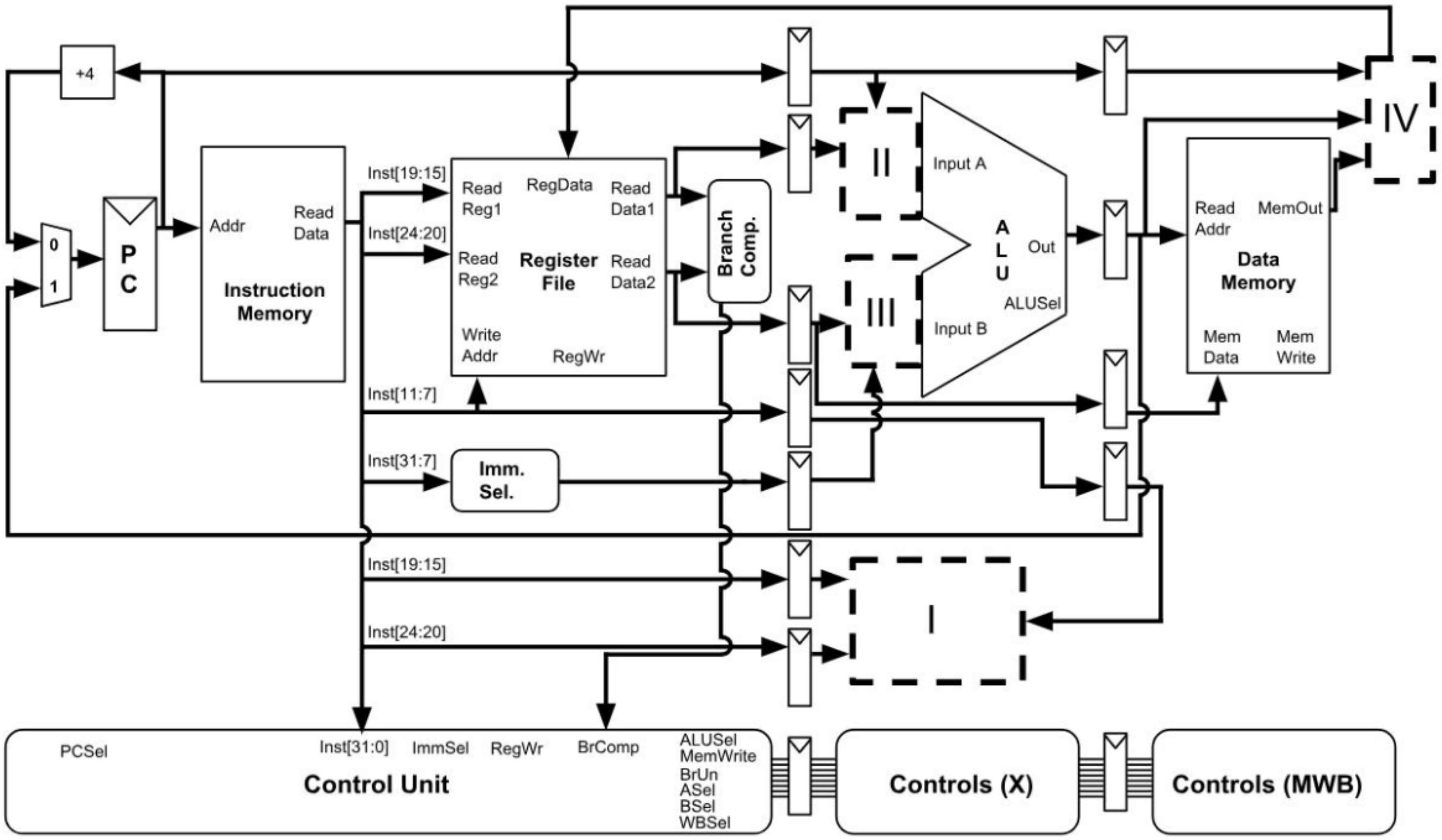
Question 5: DamonPath (8 pts)

Below we have implemented 3-stage CPU with the stages IFD, EX, and MWB. We're interested in implementing forwarding from the output of the MWB stage to the input of the ALU in the EX stage.This datapath should still implement the regular RISC-V instruction set as well as forwarding.

Make sure to read through all parts of the question (I, II, III, IV) as some variables may defined in different parts.

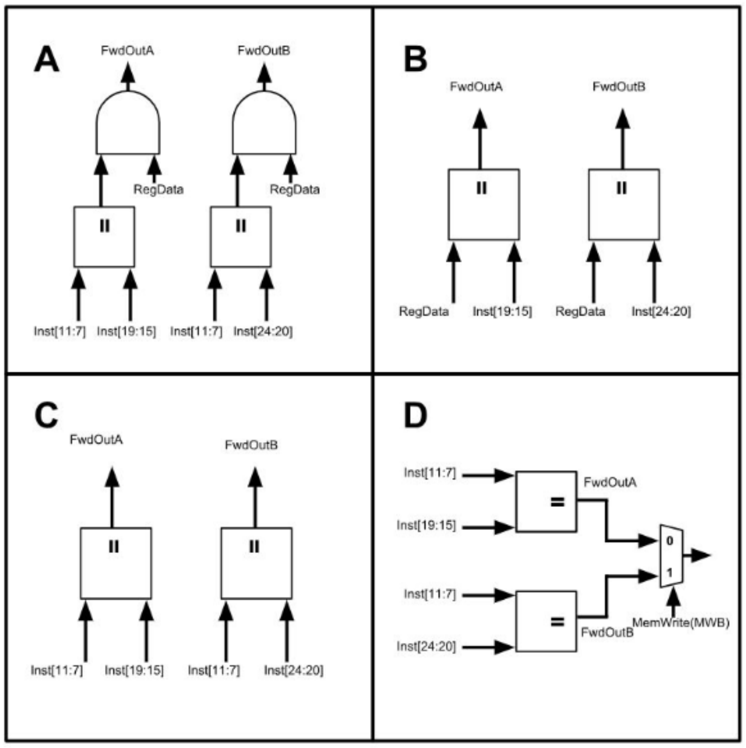
Assume that the control values generated by the control unit are driving their respective control inputs in the datapath.

(Note: In cases that may be ambiguous, we have marked certain values with the stage that they come from. For example, if we write ASel(X) this means the ASel coming from Controls(X) unit.)

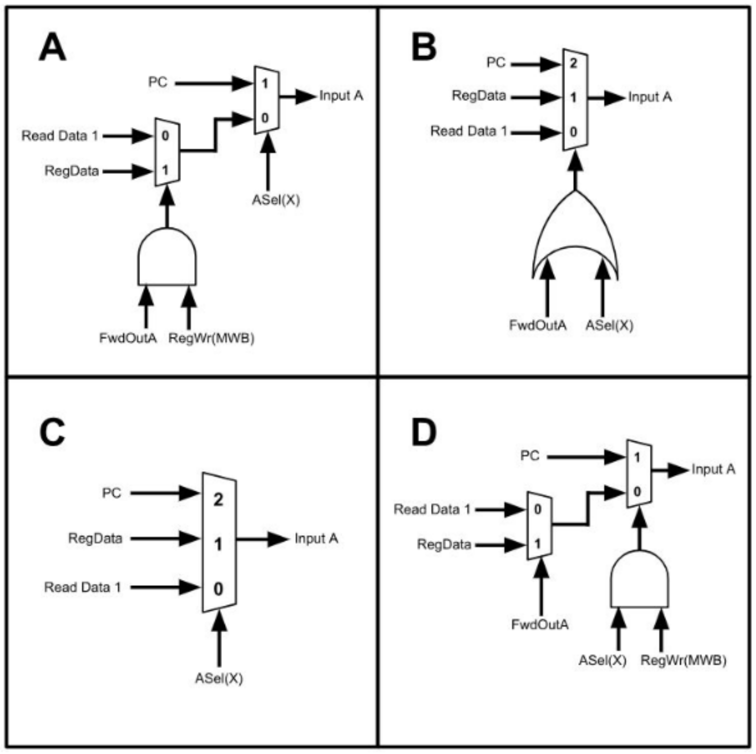


Choose the correct implementation for I - IV from the options below:

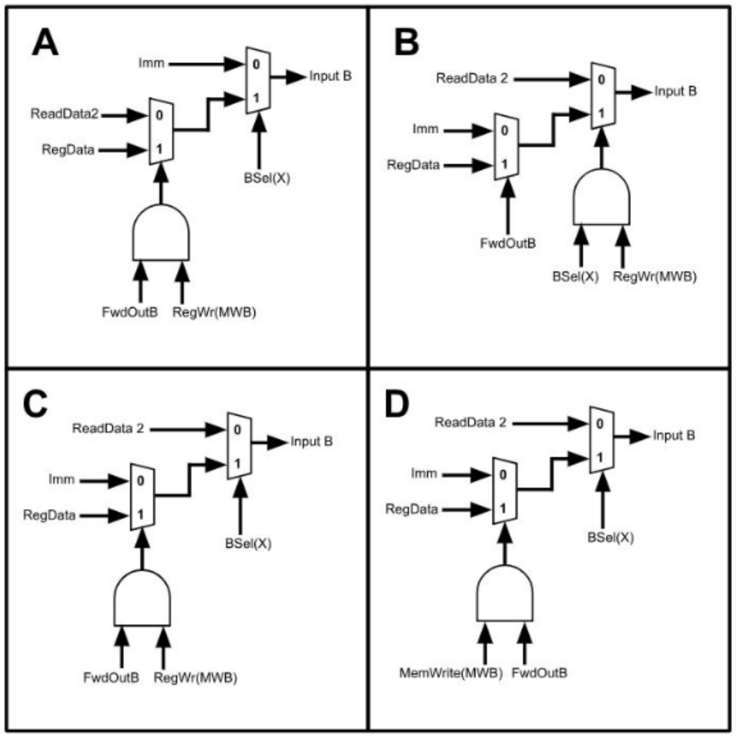
I. C



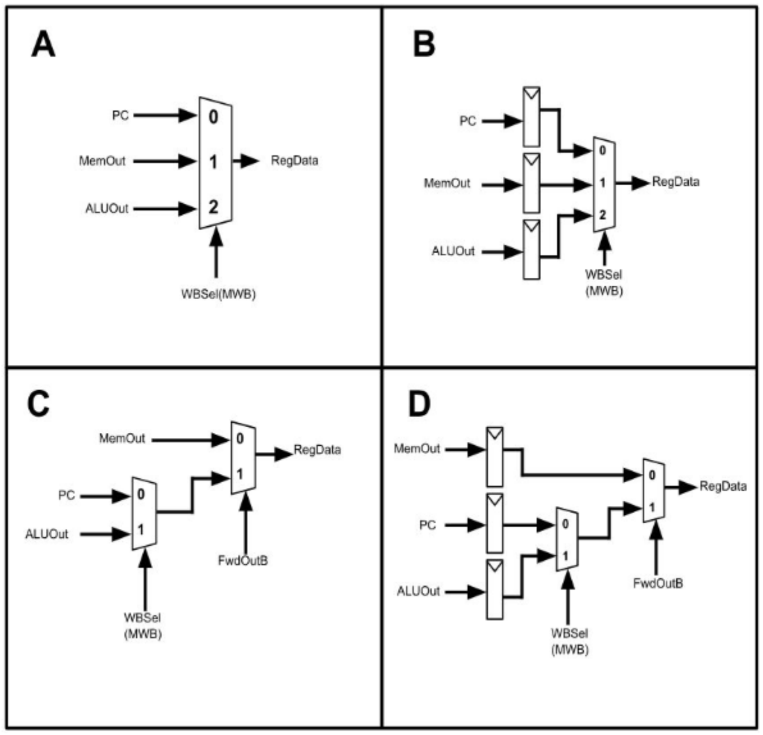
II. A



III. A



IV. A



SID: ________________

Question 4: **RISC IS HAZARDOUS** (22 pts)

Consider a RISC-V pipeline in 3 stages with the following specification:

Stage 1	Stage 2	Stage 3
Instruction Fetch/Instruction Decode (IFD)	Execute (EX)	Memory/Write Back (MWB)

It also has the following details:

- No Forwarding
- No reading and writing to the same register in the same cycle
- Reading and writing to the same register is considered a data hazard

Consider the following piece of code

```
1.         add t1 x0 x0
2.         add t2 x0 x0
3.         addi a0 x0 2
4.         slli a0 a0 2
5.  L2:    bge t1 a0 End
6.         add t3 sp t1
7.         lw t3 0(t3)
8.         add t2 t2 t3
9.         addi t1 t1 4
10.        j L2
      .
      .
      .
   End:
```

1) Assume that instead of branch prediction, the CPU always stalls to resolve a control hazard. How many stall(s) are necessary for each control hazard **each time** it is encountered? You may not need all boxes.

Line Number	# Stalls/Encounter
5	1
10	1

SID: ________________

2)

a) Where are the data hazards that produce stall(s)? You should describe each hazard as a tuple, (A, B), where instruction A is the the instruction that triggered a data hazard in instruction B. Place A's line number in the left box and B's line number in the right box You may not need all boxes.

Line of Instruction that first accesses the data	Line of instruction that must be stalled
3	4
4	5
6	7
7	8
9	5

b) How many **total cycles** will it take to complete the code above? Assume the pipeline is cleared upon reaching End. In addition, assume that there is perfectly accurate branch and jump prediction in the IFD stage. Thus, **ONLY CONSIDER STALLS DUE TO DATA HAZARDS. ASSUME THERE ARE NO STALLS FOR STRUCTURAL OR CONTROL HAZARDS.** Remember to **calculate the total number of cycles for fully executing the code**. A workspace was provided with this exam for you to show your work, which will only be graded if your final answer is not correct.

Total Cycles: 33 (Remember we said there are no stalls for control hazards)

SID: ________________

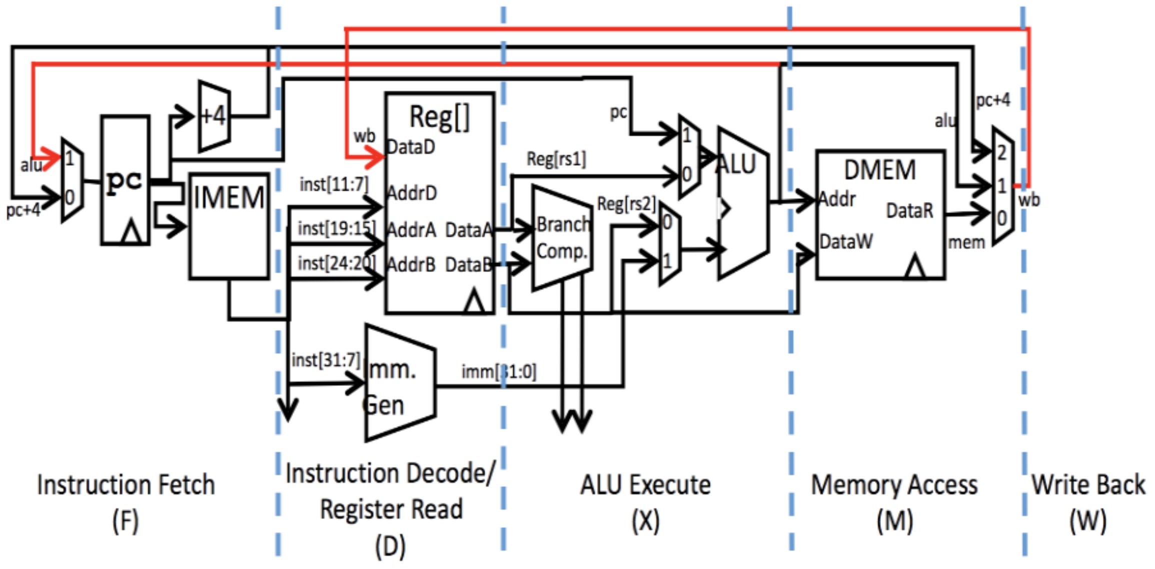
Now assume that our pipeline has full forwarding along with perfect branch/jump prediction done in the IFD stage.

3) What are the data hazards that remain? How many cycles must be stalled **each time** this hazard is encountered? If there are two instructions with the same data hazard and both are resolved by the first stall then only list the first instruction as needing to be stalled. The left box(es) should be a subset of the value(s) you had in the rightmost box(es) in 2a. You may not need all boxes.

Line of instruction that must be stalled	# Stalls/Encounter
8	1

Problem 4 ***More or Less*** **(20 points)**

Consider a typical 5-stage (**F**etch, **D**ecode, **EX**ecute, **M**emory, **W**rite**B**ack) pipeline. Assume pipeline registers exist where the dotted lines are.



For this question, consider the following parameters:

- Forwarding is not implemented
- The branch predictor always predicts the branch is not taken. Flush the pipeline if prediction is wrong.
- We cannot read and write from the same registers or memory address in the same clock cycle.
- No other optimizations are implemented in this datapath.

(a) Fill in the corresponding pipeline stages for the code sequence below for the 5-stage pipeline. The first instruction is done for you. If you need to stall a cycle, write "*" in that cycle.

```
begin:
            ori s1, x0, 0xF
            andi s2, x0, 0
            beq s1, s2, exit
            lw s1, 0xc(s0)
            xor s1, s1, s2
exit:
            lw s1, 0xc(s0)
```

Solution:

Instructions	Cycles															
	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14	c15	c16
ori s1 x0 0xf	F	D	E	M	W											
andi s2 x0 0		F	D	E	M	W										
beq s1 s2 exit			F	D	*	*	*	E	M	W						
lw s1 0xc(s0)				F	*	*	*	D	E	M	W					
xor s1 s1 s2								F	D	*	*	*	E	M	W	
lw s1 0xc(s0)									F	*	*	*	D	E	M	W

The first thing to notice for this question is that the datapath does not implement bypassing, and a register cannot be simultaneously read and written in the same cycle. Recall that instructions read their registers in stage DE, and write registers in WB. These restrictions mean that if instruction B needs a register that instruction A writes, then B cannot start its DE stage until the cycle after A's WB stage (this is a "data hazard"). The other type of hazards to worry about are "structural hazards", this means that no two instructions can be in the same stage at the same time. Now lets go through the answer instruction-by-instruction:

- **andi s2 x0 0:** This doesn't have any data depencies, so we just need to worry about structural hazards. It can start as soon as the F stage is available (c2).
- **beq s1 s2 exit:** This instruction reads register s2 which was written by the previous instruction. Therefore we must wait until the andi has finished its WB stage before running beq's DE stage (c7).
- **lw s1 0xc(s0):** At this point, the result of the branch doesn't matter because it is always predicted to be taken. Also, the branch doesn't write any registers, so we don't have any data dependencies and can start as soon as the stages are available. In this case, the fetch can start on c4, but the decode has to wait until beq is done with it (c8).

- **xor s1 s1 s2:** We now must consider whether or not the branch was predicted correctly. Fortunately it was, so we don't need to take any action. Next we must look for data hazards; s1 is read by xor, but written by lw, so we must wait for lw to finish WB before starting xor's DE (c12).
- **lw s1 0xc(s0):** There are no data hazards (notice that s1 is not read during the lw, only written, so there isn't a hazard). We need only wait for the stages to become available (structural hazards).

(b) Assume the maximum delays through each stage are: `F: 200ns, D: 150ns, EX: 100ns, MEM: 300ns, WB: 250ns`.

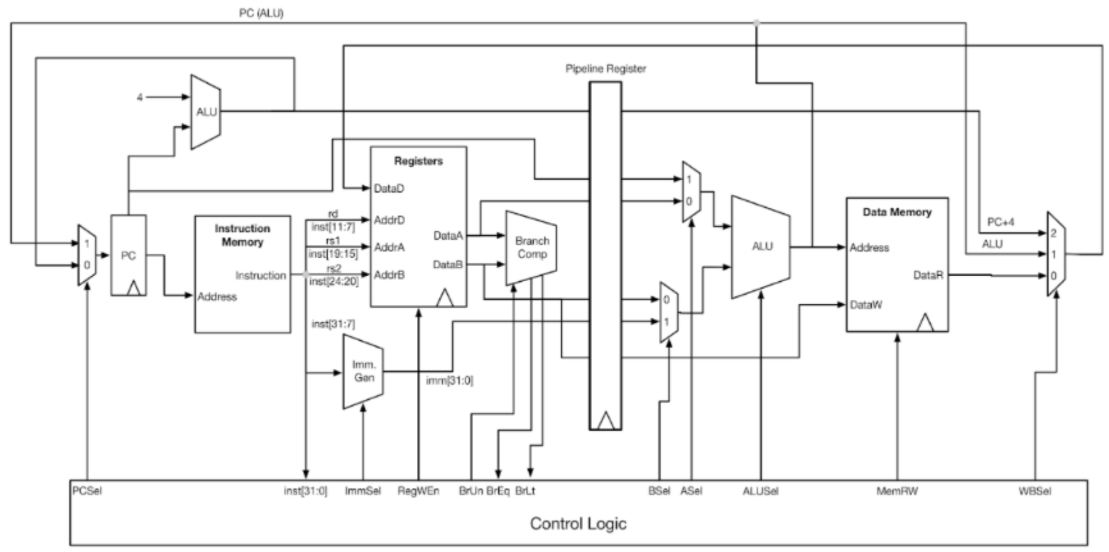
Assume the delays for the pipeline registers are factored into the pipeline stage delays. What is the latency and best case throughput of this 5-stage pipelined CPU? You may leave your answers as fractions. Don't forget the units!

Solution: Latency: 1500 ns Throughput: $\frac{1}{300ns}$

Latency: Latency is the time it takes from when the instruction starts the first stage, to when it exits the last stage. While you may be tempted to sum up the stage latencies, this would not be correct. This is because the CPU must run at a constant clock frequency, and the clock frequency is determined by the slowest stage (MEM in this case). Therefore, the cycle time is 300ns, and it takes 5 cycles to finish all stages, so the latency is $5 * 300 = 1500$ns.

Throughput: Throughtput is rate at which instructions leave the pipeline when there are lots of instructions running (steady-state). Technically, throughput could be affected by data hazards and such, so we ask for the "best-case" throughput (e.g. for an endless series of non-dependent adds). A pipeline completes one instruction every cycle under ideal circumstances, so the throughput of this pipeline is $\frac{1}{cycleTime} = \frac{1}{300ns}$.

You decide to combine certain stages to make a two-stage pipeline by removing certain registers. The two-stage pipelined datapath looks like the following:



Again, consider the following parameters:

- Forwarding is not implemented
- The branch predictor always predicts the branch is not taken. Flush the pipeline if prediction is wrong.
- We cannot read and write from the same registers or memory address in the same clock cycle.
- No other optimization is implemented on this datapath.

(c) Fill in the corresponding pipeline stages for the same code sequence for the 2-stage pipelined CPU. The code sequence is reproduced below. Use "A" to denote stage 1, "B" for stage 2. The first instruction is done for you. If you need to stall a cycle, write "*" in that cycle.

```
begin:
        ori s1, x0, 0xF
        andi s2, x0, 0
        beq s1, s2, exit
        lw s1, 0xc(s0)
        xor s1, s1, s2
exit:
        lw s1, 0xc(s0)
```

Solution:

Instructions	Cycles													
	c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	c11	c12	c13	c14
ori s1 x0 0xf	A	B												
andi s2 x0 0		A	B											
beq s1 s2 exit			A	*	B									
lw s1 0xc(s0)					A	B								
xor s1 s1 s2						A	*	B						
lw s1 0xc(s0)								A	B					

This question is extremely similar to 4.a so we will not cover it in quite as much detail. To do this problem, you must first determine what impact data hazards will have. In this case, the registers are written in stage B and read in stage A, so we just need to make sure that instructions that have data dependencies wait until the previous B stage finishes before starting A. The two data dependencies are on the beq and the xor, so those have stalls to ensure their A stage doesn't run until the previous B has finished. Also note that unlike 4.a, we don't introduce long stalls on later instructions due to structural hazards (the lw's start relatively sooner). This results in the program taking 9 cycles instead of 16. This true in general of pipelining: deeper pipelines are more impacted by stalls than shallower pipelines.

(d) Suppose we want to execute three instructions on this two-stage pipelined CPU. The first instruction begins executing at the start of Cycle C0, the second begins at the start of Cycle C1, and the third, C2.
Fill in the correct control signals on each clock cycle in order to execute these instructions correctly:

- Any signals set by earlier instructions (before the first) should be set to "E".
- Any signals set by later instructions (after the third) should be set to "L".
- Indicate `Enable` or `Disable` for write enable signals.
- ImmSel should be set as `I, S, SB, U` or `UJ`.
- All other signals should be set as 0, 1, an ALU operation, or X (doesn't matter).
- The list of available ALU operations are `ADD, AND, OR, SRL, SRL, SLT`.

You may assume that there are no structural or data hazards.

```
Program:
    srl t1, t2, t3
    sw t0, 4(a0)
    bltu s0, t2, 44
```

Cycle	Signals								
	PCSel	ImmSel	RegWEn	BrUn	BSel	ASel	ALUSel	MemRW	WBSel
C0	0	X	E	X	E	E	E	E	E
C1	0	S	1	X	0	0	SRL	1	01
C2	0	SB	0	1	1	0	000	0	X
C3	X	L	0	L	1	1	000	1	X

PCSel is somewhat ambiguous for C3 so we are allowing any answer for that cell

The first step in solving this problem is to identify which control signals are set in which stage. This is because, as a pipelined datapath, two different instructions will be responsible for setting control signals during any given cycle. The assignments are as follows:

- **A:** ImmSel, BrUn
- **B:** RegWEn, BSel, ASel, ALUSel, MemRW, WBSel
- **PCSel:** Is a bit ambiguous. It's not really set by any particular instruction, but rather defaults to 0, but can be overwritten by earlier branches (if they predicted wrong).

Notice that RegWEn is physically on the left, but is actually set in the second stage (write-back). A common error was setting RegWEn in stage A.

Next we need to figure out what instructions are responsible for which control signals on a cycle-by-cycle basis (you could write this down in the margins or on your scratch paper).

- **C0:** SRL / E
- **C1:** SW / SRL
- **C2:** BLTU / SW
- **C3:** L / BLTU

Now we just need to figure out the control signals for each instruction as if it weren't pipelined, and then fill them into the table in the appropriate locations:

- **SRL:**
  - **ImmSel**: Doesn't use immediates (don't care)
  - **BrUn**: Not a branch (don't care)
  - **RegWen:** R-types write a result to the register, so this must be 1
  - **A/BSel:** R-types use registers for input so 0,0
  - **ALUSel**: ALU needs to do an SRL
  - **MemRW:** Not a memory instruction, so must be "read". It's not X because we mustn't write garbage into memory.
  - **WBSel:** R-types write ALU results back to a register so must be 01.
- **SW:**
  - **ImmSel:** Stores use an immediate for the offset, must ask the imm generator to do an S-type
  - **BrUn:** Not a branch, X
  - **RegWen:** Stores are from reg-¿mem, it doesn't write any registers: 0
  - **A/BSel:** Adds the immediate to a register value so B=1, A=0
  - **ALUSel:** Needs to add the offset, so set ALU to ADD
  - **MemRW:** Writes to memory, 1
  - **WBSel:** Doesn't write to register, X
- **BLTU:**
  - **ImmSel:** Uses immediate for branch target, use SB-type imm
  - **BrUn:** Branch is unsigned: 1

  - **RegWen:** Doesn't write to a register: 0
  - **A/BSel:** Branches in RISC-V are PC-relative, so it needs to add an immediate to the PC (1, 1)
  - **ALUSel:** Needs to add imm with PC: ADD
  - **MemRW:** Doesn't write memory: read
  - **WBSel:** Doesn't write registers: X

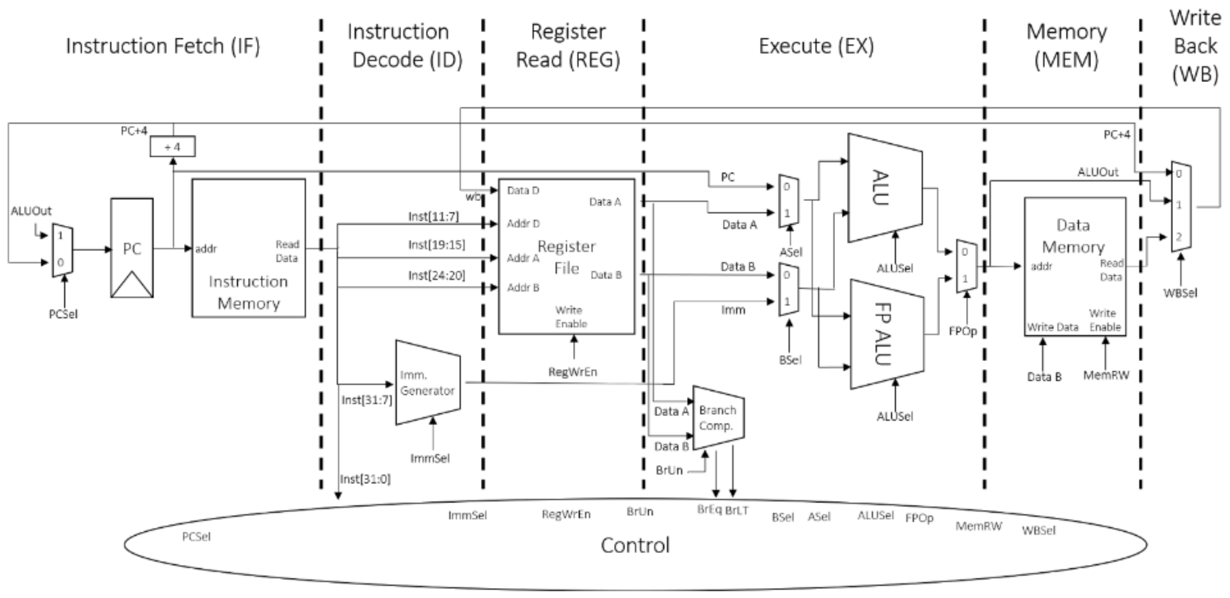
The last thing to mention here is that the PC is a bit ambiguous in this question.

- **C0:** Could be 0 or E. 0 because that's what seemed to happen (the next instruction in memory ran), but it could be E if you think E was a branch.
- **C1:** Must be 0 because there wasn't an earlier branch and stores don't affect the PC
- **C2:** Must be 0 because we said branches are predicted not taken
- **C3:** Could be anything because we didn't say whether the branch was taken or not.

Q4: Danger: Hazardous Material

In this question, you will be working with a modified RISC-V CPU. As opposed to the traditional 5-stage pipeline, this altered CPU has split the second phase into two distinct stages: instruction decode and register read. Furthermore, this CPU has a floating-point ALU (FP ALU) along with a traditional ALU (no floating point Register File is added). The added control signal, FPOp, determines which ALU to use for a given instruction. This floating-point ALU works by interpreting its 32-bit operands in IEEE 754 floating-point format. Unlike in standard RISC-V (and on the green sheet), assume floating point operations use the same registers as normal, non-floating point operations.

A diagram of the modified CPU and its corresponding stages are shown below. To simplify the diagram, any label at the very beginning of a wire acts like a tunnel in Logism. Along with the diagram above, you are given the following delays incurred by the circuit elements in the table below, as well as the delays incurred for some of the datapath stages.



Element	Register CLK-to-Q	Register Setup	MUX	ALU	FP ALU	Mem Read	Mem Write	RegFile Read	RegFile Setup	Imm. Gen.
Delay (ps)	20	25	10	150	215	225	230	100	35	75

a. Consider the floating point instruction, `faddi`, that is similar to `addi` except it considers its operands in floating-point format and executes a floating-point add operation. Assuming that this CPU is **NOT Pipelined** (i.e. it is a single-cycle CPU), what is the shortest clock period possible to execute the instruction `faddi t0, s0, 2.71` corectly? Write your answer as a single, simplified number (no summations or other expressions) on your answer sheet.

20 + 225 + max(75 + 100) + 10 + 215 + 10 + 10 + 35 = 625 ps

b. What is the shortest possible clock period at which we can run this pipelined CPU? Write your answer as a single, simplified number (no summations or other expressions) on your answer sheet.

IF: CLK-to-Q + MemRead + RegSetup = 20 + 225 + 25 = 270ps
ID: CLK-to-Q + ImmGen + RegSetup = 20 + 75 + 25 = 120ps
REG: CLK-to-Q + RegFileRead + RegSetup = 20 + 100 + 25 = 145ps
EX: CLK-to-Q + MUX + FPALU + MUX + RegSetup = 20 + 10 + 215 + 10 + 25 = 280ps
MEM: CLK-to-Q + MemWrite + RegSetup = 20 + 230 + 25 = 275ps
WB: CLK-to-Q + MUX + RegFileSetup = 20 + 10 + 35 = 65ps
The answer is therefore max(270, 120, 145, 280, 275, 65) = 280ps.

c. Keeping the modified pipeline in mind, consider the program below with the following assumptions:

- **There are no pipelining optimizations** (no forwarding, load delay slot, branch prediction, pipeline flushing, etc.)...
- **We cannot read and write from registers in the same clock cycle.**
- **An integer 100 is stored at memory address 0x61C61C61, and that R[a0] = 0x61C61C61**.
- A hazard between two instructions should be counted as only 1 hazard.

```
lw t0, 0(a0)         # R[a0] = 0x61C61C61
srli s0, t0, 4
faddi s1, t0, 1.7
beq a0, s1, Label
add a1, t2, t3
Label ...
```

Write your answers to the questions below on your answer sheet.

```
lw t0, 0(a0)
nop
nop
nop
srli s0, t0, 4
```

```
faddi s1, t0, 1.7
nop
nop
nop
beq a0, s1, Label
nop
nop
nop
add a1, t2, t3
```

i. How many cycles would the program take to execute correctly on the pipelined machine? 19

ii. How many stalls would need to be added for the program to be executed correctly on the pipelined machine? 9

iii. How many data hazards are present in the program? 3

iv. How many control hazards are present in the program? 1

d. For both parts below, reorder the instructions to minimize the number of cycles needed to execute the program, then answer the questions below. Your reordered instructions should produce the same results for all involved registers as the original instructions do.

```
lw t0, 0(a0)
nop
nop
nop
addfi s1, t0, 1.7
srli s0, t0, 4
nop
nop
beq a0, s1, Label
nop
nop
nop
add a1, t2, t3
```

i. How many cycles does the program take to execute with reordered instructions?

18

ii. How many stalls are required?

8