# 6.004 Worksheet Questions
# L20 – Control Hazards in Pipelined Processors

**Problem 1** ★

The loop on the right has been executing for a while on our standard 5-stage pipelined RISC-V processor with branch annulment and full bypassing.

```
    …
L1: addi x10, x10, -4
    slti x11, x10, 10
    beqz x11, L2
    lw x12, 0x200(x10)
    j L3
L2: lw x12, 0x300(x10)
L3: sw x12, 0x400(x0)
    bnez x10, L1
    addi x12, x12, 1
    xor x12, x12, x0
    …
```

| *Cycle #* | *300* | *301* | *302* | *303* | *304* | *305* | *306* | *307* | *308* | *309* |
|---|---|---|---|---|---|---|---|---|---|---|
| **IF** | addi | slti | beqz | lw | j | lw | sw | bnez | bnez | bnez |
| **DEC** | NOP | addi | slti | beqz | lw | NOP | lw | sw | sw | sw |
| **EXE** | NOP | | addi | slti | beqz | NOP | NOP | lw | NOP | NOP |
| **MEM** | bnez | | | addi | slti | beqz | NOP | NOP | lw | NOP |
| **WB** | sw | | | | addi | slti | beqz | NOP | NOP | lw |

(A) **Fill in the pipeline diagram** for cycles 301-309 assuming that at cycle 300 the instruction at L1 is fetched. Also, assume that the branch to L2 is taken, as well as the final branch back to L1. Finally, assume that the value for x10 is available in the register file prior to cycle 300. **Indicate which bypass/forwarding paths are active in each cycle by drawing a vertical arrow in the pipeline diagram** from pipeline stage X in a column to the RF stage in the same column if an operand would be bypassed from stage X back to the RF stage that cycle. Note that there may be more than one vertical arrow in a column.

**Fill in pipeline diagram including bypass arrows in pipeline diagram above**

(B) Assume that the previous iteration of the loop executed the same instructions as the iteration shown in part (A). Please complete the pipeline diagram for cycle 300 by filling in the OPCODEs for the instructions in the DEC, EXE, MEM, and WB stages.

**Fill in OPCODEs for Cycle 300**

Answer filled in above.

(C) Indicate which branches are taken by providing the cycle in which the taken branch instruction enters the IF stage.

**Cycle number(s) or NONE: __302, 307_____**

(D) During which cycle(s), if any, do we have stalled instructions?

**Cycle number(s) or NONE: __307, 308_____**

Now consider a modified processor, P2, which has extra hardware in the decode stage (DEC) to resolve simple branches one cycle earlier: the decode stage includes both a circuit to check whether a register is equal to zero, and an extra adder to compute the branch target for taken branches. This processor can thus compute nextPC for `beqz` and `bnez` in DEC instead of EXE.

(E) Redo part A using processor P2 assuming the same path is taken through the code.

| *Cycle #* | *300* | *301* | *302* | *303* | *304* | *305* | *306* | *307* | *308* | *309* |
|---|---|---|---|---|---|---|---|---|---|---|
| **IF** | addi | slti | beqz | lw | lw | sw | bnez | bnez | bnez | addi |
| **DEC** | NOP | addi | slti | beqz | NOP | lw | sw | sw | sw | bnez |
| **EXE** | bnez | | addi | slti | beqz | NOP | lw | NOP | NOP | sw |
| **MEM** | sw | | | addi | slti | beqz | NOP | lw | NOP | NOP |
| **WB** | NOP | | | | addi | slti | beqz | NOP | lw | NOP |

(F) Compare the number of cycles per loop iteration using the original processor and the modified processor.

**Cycles per loop in original processor: ___12_____**

**Cycles per loop in processor P2: ___10_____**

**Problem 2** ★

You've discovered a secret room in the basement of the Stata center full of discarded 5-stage pipelined RISC-V processors. Unfortunately, many have certain defects. You discover that they fall into four categories:

**C1:** A modified, completely functional 5-stage RISC-V processor with working bypass paths, annulment, and other components, as well as extra hardware support that allows the nextPC to be calculated in the Decode stage.
**C2:** A defective version of **C1** with a bad register file: all data read from the register file is zero.
**C3:** A defective version of **C1** with broken bypass muxes: all source operands come from the register file, even if they should be read from bypassed paths.
**C4:** A defective version of **C1** without annulment of instructions following branches.

To help sort the processors into the above classes, you write the following small test program:

```
. = 0x0
// Start at 0x0, with ZERO
// in all registers…
      addi x10, x0, 4
      jal x12, X
      slli x12, x12, 1
X:    addi x12, x12, -4
      add x13, x12, x10
      jr x13
```

| C1 | C2 | C3 | C4 |
|---|---|---|---|
| x10 = 4 | x10 = 4 | x10 = 4 | x10 = 4 |
| x12 = 8 | x12 = 8 | x12 = 8 | x12 = 8 |
| ---------- | ---------- | ---------- | x12 = 16 |
| x12 = 4 | x12 = 4 | x12 = -4 | x12 = 12 |
| x13 = 8 | x13 = 4 | x13 = 4 | x12 = 16 |
| | want x12 from bypass<br>want x10 from RF | | |

Your plan is to single-step through the program using each processor, carefully noting the address the final `jr` loads into the PC. Your goal is to determine which of the above four classes a chip falls into by this `jr` address.

For each class of RISC-V processor described above, specify the value that will be loaded into the PC by the final `jr` instruction.

Pipeline diagram showing first 7 cycles of test program executing on C1:

| *cycle* | *0* | *1* | *2* | *3* | *4* | *5* | *6* |
|---|---|---|---|---|---|---|---|
| IF | addi | jal | slli | addi | add | jr | |
| DEC | | addi | jal | NOP | addi | add | jr |
| EXE | | | addi | jal | NOP | addi | add |
| MEM | | | | addi | jal | NOP | addi |
| WB | | | | | addi | jal | NOP |

**C1: jr goes to address**: ____8_____

**C2: jr goes to address**: ____4_____

**C3: jr goes to address**: ____0_____

**C4: jr goes to address**: ____16_____

**Problem 3** ★

(A) How many cycles does it take to run each iteration of the following loop (assuming the beqz is always taken) on a standard 5-stage pipelined RISC-V processor?

```
loop: lw x10, 0x100(x0)
      beqz x10, loop
      add x12, x10, x11
      sub x13, x12, x1
```

**Number of cycles per loop iteration**: ____6_____

| *cycle* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | lw | beqz | add | add | add | sub | lw | | | | |
| DEC | | lw | beqz | beqz | beqz | add | NOP | lw | | | |
| EXE | | | lw | NOP | NOP | beqz | NOP | NOP | lw | | |
| MEM | | | | lw | NOP | NOP | beqz | NOP | NOP | lw | |
| WB | | | | | lw | NOP | NOP | beqz | NOP | NOP | lw |


(B) Assuming a defective 5-stage pipelined RISC-V processor where the instructions following a taken branch are not annulled, which of the following statements would be true?

1. The add instruction would be executed each time through the loop.
2. The loop would take 5 cycles to execute
3. The value of the register x10 that is tested by the beqz instruction comes from a bypass path.
4. The value of register x10 that is accessed by the add instruction comes from the register file.

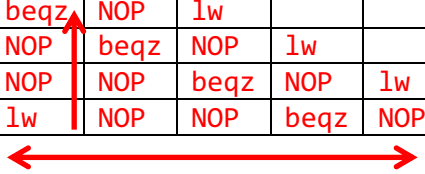
For such a defective processor, the instruction flow diagram looks like this:

| *cycle* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | lw | beqz | add | add | add | sub | lw | | | | |
| DEC | | lw | beqz | beqz | beqz | add | sub | lw | | | |
| EXE | | | lw | NOP | NOP | beqz | add | sub | lw | | |
| MEM | | | | lw | NOP | NOP | beqz | add | sub | lw | |
| WB | | | | | lw | NOP | NOP | beqz | add | add | lw |

(C) Consider a modified processor, P2, which has extra hardware for the special case of checking if a register is equal to zero or not in the decode stage. What would be the number of cycles per loop iteration in this case?

**Number of cycles per loop iteration on processor P2**: ____5_____

| *cycle* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | lw | beqz | add | add | add | lw | | | | | |
| DEC | | lw | beqz | beqz | beqz | NOP | lw | | | | |
| EXE | | | lw | NOP | NOP | beqz | NOP | lw | | | |
| MEM | | | | lw | NOP | NOP | beqz | NOP | lw | | |
| WB | | | | | lw | NOP | NOP | beqz | NOP | lw | |




(D) Now consider a third processor, P3, whose instruction and data memories are pipelined and take 2 clock cycles to respond. Assume that P3 also has the extra hardware for checking if a register is equal to zero or not in the decode stage. What would be the number of cycles per loop iteration using P3?

NOP **Number of cycles per loop iteration on processor P3**: ____7_____

| *cycle* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | lw | beqz | add | sub | sub | sub | sub | lw | | | | | | |
| | | lw | beqz | add | add | add | add | NOP | lw | | | | | |
| DEC | | | lw | beqz | beqz | beqz | beqz | NOP | NOP | lw | | | | |
| EXE | | | | lw | NOP | NOP | NOP | beqz | NOP | NOP | lw | | | |
| MEM | | | | | lw | NOP | NOP | NOP | beqz | NOP | NOP | lw | | |
| | | | | | | lw | NOP | NOP | NOP | beqz | NOP | NOP | lw | |
| WB | | | | | | | lw | NOP | NOP | NOP | beqz | NOP | NOP | lw |

**Problem 4**

You've been given a 5-stage pipelined RISC-V processor. Unfortunately, the processor you've been given is defective: it has no bypass paths, annulment of instructions in branch delay slots, or pipeline stalls.

```
        nop
        nop
        nop
        nop

loop:
        lw x10, 0x0(x10)
AA:
        sll x14, x10, x11
BB:
        bnez x10, loop
CC:
        add x13, x10, x13

        nop
        nop
        nop
        nop
```

You undertake to convert some existing code, designed to run on an unpipelined RISC-V, to run on your defective pipelined processor. The scrap of code on above is a sample of the program to be converted. It doesn't make much sense to you – it doesn't to us either – but you are to add the **minimum** number of **NOP** instructions at the various tagged points in this code to make it give the same results on your defective pipelined RISC-V as it gives on a normal, unpipelined RISC-V.

Note that the code scrap begins and ends with sequences of **NOPs**; thus, you don't need to worry about pipeline hazards involving interactions with instructions outside of the region shown.

(A) Specify the minimal number of **NOP** instructions (*defined as add x0, x0, x0*) to be added at each of the labeled points in the above program.

**NOPs at Loop: __0____**

**NOPs at AA: __3____**

**NOPs at BB: __0____**

**NOPs at CC: __2____**

Below is a diagram of the instruction flow for this program. Note that nextPC is available in the execute stage of the bnez instruction even without bypassing.

| *cycle* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* |
|---|---|---|---|---|---|---|---|---|---|
| IF | lw | NOP | NOP | NOP | sll | bnez | NOP | NOP | lw/add |
| DEC | | lw | NOP | NOP | NOP | sll | bnez | NOP | NOP |
| EXE | | | lw | NOP | NOP | NOP | sll | bnez | NOP |
| MEM | | | | lw | NOP | NOP | NOP | sll | bnez |
| WB | | | | | lw | NOP | NOP | NOP | sll |

(B) On a **fully functional** 5-stage pipeline (with working bypass, annul, and stall logic), the above code will run fine with no added **NOP**s. How many clock cycles of execution time are required by the fully functional 5-stage pipelined RISC-V **for each iteration** through the loop?

**Clocks per loop iteration: __7____**

Below is a diagram of the instruction flow for this program.

| *cycle* | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* |
|---|---|---|---|---|---|---|---|---|
| IF | lw | sll | bnez | bnez | bnez | add | NOP | lw |
| DEC | | lw | sll | sll | sll | bnez | add | NOP |
| EXE | | | lw | NOP | NOP | sll | bnez | NOP |
| MEM | | | | lw | NOP | NOP | NOP | bnez |
| WB | | | | | lw | NOP | NOP | NOP |

**Problem 5** ★

You are designing a four stage RISC-V processor (IF, DEC, EXE, WB). Currently you are trying to decide whether to include bypassing from the write-back stage to the decode stage. As part of this evaluation, you construct two processors:

**Processor A:** No bypassing from WB to DEC.
**Processor B:** Bypassing from WB to DEC.

You are using the following loop of an important program to evaluate the performance of the processor:

```
L1:   lw t0, 0(a0)
      add a1, a1, t0
      addi a0, a0, 4
      blt a0, a2, L1
```

For the following questions, assume this loop has been running for a long time. Assume that both processors will always predict the direction of the `blt` correctly so there are no control hazards.

(A) How many cycles per loop iteration does the decode stage stall due to read after write hazards in the following cases?

**Processor A decode stall cycles per iteration:** _____4_____

**Processor B decode stall cycles per iteration:** _____2_____

(B) How many cycles does this loop take to execute in the following cases?

**Processor A cycles per iteration:** _____8_____

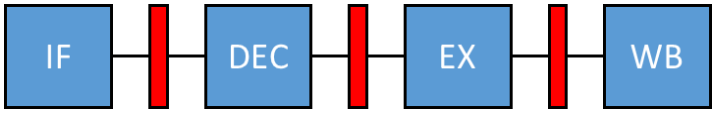
**Processor B cycles per iteration:** _____6_____

| A | IF | lw | add | addi | addi | addi | blt | lw | lw | lw | add | addi |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | DEC | | lw | add | add | add | addi | blt | blt | blt | lw | add |
| | EXE | | | lw | NOP | NOP | add | addi | NOP | NOP | blt | lw |
| | WB | | | | lw | NOP | NOP | add | addi | NOP | NOP | blt |

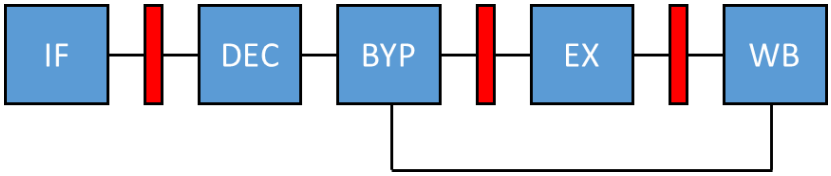
| B | IF | lw | add | addi | addi | blt | lw | lw | add | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | DEC | | lw | add | add | addi | blt | blt | lw | add | add | |
| | EXE | | | lw | NOP | add | addi | NOP | blt | lw | NOP | add |
| | WB | | | | lw | NOP | add | addi | NOP | blt | lw | NOP |

Processor A has the following propagation delays for each of the pipeline stages:

IF: 2 ns
DEC: 3.0 ns
EX: 3.5 ns
WB: 1.0 ns




The logic for the bypassing path of processor B can be viewed as taking the output from the DEC and WB stages of processor A and adding an additional bypass logic (BYP) as shown in the picture below.




Assuming the BYP logic has a propagation delay of 1 ns and that all registers are ideal.

(C) What is the minimum clock period for each processor?

**Clock period for processor A: ____3.5 ns_____**

**Clock period for processor B: ____4 ns_____**

(D) For the loop shown above, what is the average cycles per instruction for the two processors:

**Average cycles per instruction for processor A: ____8/4 = 2_____**

**Average cycles per instruction for processor B: ___6/4 = 3/2_____**

(E) For the loop shown above, what is the average number of instructions per second for the two processors:

**Average number of instructions per second for processor A: ____1/(7ns)_____**

**Average number of instructions per second for processor B: ____1/(6ns)_____**

Instr/sec = 1/(cyc/instr)(sec/cycle)
A: 1/(2*3.5ns) = 1/(7ns)
B: 1/((3/2)*4ns) = 1/(6ns)

**Problem 6**

Ben Bitdiddle and Alyssa P. Hacker are building a five stage RISC-V processor to help grade 6.004 exams. Their pipeline has the standard stages and functionality presented in lecture (IF, DEC, EXE, MEM, and WB). You may assume that instruction and data memories are single-cycle memories with clocked reads and writes.

The following code segment simulates counting the number of correct answers on a question. It loads a student answer, increments the count if its correct, and then loops back to the next student.

```
...
grade_question:
    lw t0, 0(a2)                // load a student's answer
    bne t0, a1, next_student    // check answer; assume bne is NOT TAKEN
    addi t1, t1, 1              // increment num correct
next_student:
    addi a2, a2, 4
    blt a2, a3, grade_question  // next student; assume blt is ALWAYS TAKEN
next_question:
    xor a4, a0, a4
    slli a4, a4, 2
...
```

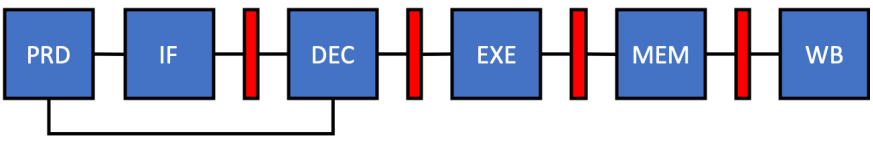
6.004 students are doing well on the exam and are all getting the questions right, so **the bne branch is never taken**. Also, the processor is in the middle of grading exams, so the **blt branch is always taken.**

Ben starts out with a fully functional 5-stage RISC-V processor with **full bypassing and annulment hardware**. Assume branch decisions are made in the EXE stage. Ben's processor always speculates that nextPC will be PC + 4.

Alyssa thinks she can achieve better performance with a smarter branch predictor to better handle loops. Her processor includes additional hardware to speculate that the branch is not taken (i.e. nextPC = PC + 4) on all forward branches (i.e. the branch target address is greater than the current PC) and that the branch is taken on all backwards branches. More formally:

$$nextPC = \begin{cases} PC + immB, & immB \leq 0 \\ PC + 4, & immB > 0 \end{cases}$$

When the branch is detected in the DEC stage, the hardware will make its prediction based on immB and impact the instruction fetched **in that same cycle**. The following schematic shows the position of the prediction logic (PRD). The red bars represent the pipeline registers.



The following pipeline diagrams may be useful. You are not required to fill them out.

Ben's Pipeline Diagram

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | lw | bne | addi | addi | addi | addi | blt | xor | slli | lw | |
| DEC | | lw | bne | bne | bne | addi | addi | blt | xor | NOP | |
| EXE | | | lw | NOP | NOP | bne | addi | addi | blt | NOP | |
| MEM | | | | lw | NOP | NOP | bne | addi | addi | blt | |
| WB | | | | | lw | NOP | NOP | bne | addi | addi | |

Alyssa's Pipeline Diagram

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | lw | bne | addi | addi | addi | addi | blt | lw | | | |
| DEC | | lw | bne | bne | bne | addi | addi | blt | | | |
| EXE | | | lw | NOP | NOP | bne | addi | addi | | | |
| MEM | | | | lw | NOP | NOP | bne | addi | | | |
| WB | | | | | lw | NOP | NOP | bne | | | |

(A) How many cycles does this loop take to execute in each of the processors?

**Ben's processor cycles per iteration: _____9_____**

**Alyssa's processor cycles per iteration: _____7_____**

(B) Within one iteration of this loop, how many instructions need to be annulled due to incorrect speculation?

**Number of annulled instructions in Ben's proc: _____2_____**

**Number of annulled instructions in Alyssa's proc: _____0_____**

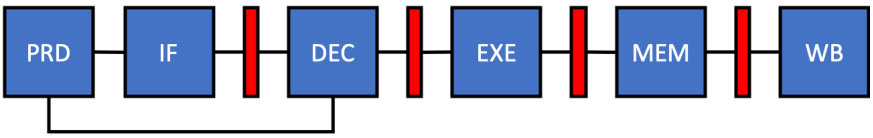
(C) For this loop, what is the average cycles per instruction (CPI) for each of the processors?

**Average CPI for Ben's processor: _____9/5_____**

**Average CPI for Alyssa's processor: _____7/5_____**

The logic for Ben's processor is shown below with the red bars representing pipeline registers.


The logic for the additional hardware that Alyssa added can be viewed as taking values from the DEC stage and adding additional prediction logic (PRD) as shown below (diagram copied from above).




Assume that all registers are ideal ($t_{SETUP} = t_{HOLD} = t_{PD} = t_{CD} = 0\ ns$) and each pipeline stage/piece of combinational logic has the following propagation delays.

| | |
|---|---|
| PRD | 1 ns |
| IF | 3 ns |
| DEC | 4 ns |
| EXE | 7 ns |
| MEM | 5 ns |
| WB | 2 ns |

(D) What is the minimum clock period for each processor?

**Minimum clock period for Ben's processor (ns):** _____7_____

**Minimum clock period for Alyssa's processor (ns):** _____8_____

(E) Which processor takes less time to execute one iteration of this loop, and how much faster is it?

**Processor that executes loop in less time (Ben's or Alyssa's):** _____Alyssa's_____

**Number of nanoseconds difference (ns):** _____7_____

**Problem 7 (From Past Quiz)**

Val wants to have a Thanksgiving party, but there's a nasty virus going around. Instead, she will buy the most expensive vegan turkey, a tomato, and a celery. She is at Whole Foods. She already knows that she can buy a vegan turkey for $10 at Wal-Mart, so she will be looking for something more expensive. Val wants to figure out how much her vegan meal will cost so she writes some C code, before converting it into assembly (shown on the right). Assume the registers are initialized to the values specified in the assembly code comments.

**C Code**

```
int price[6] = {7, 5, 8, 10, 15, 7};
int maximum = 10;
int celery = 3;
int tomato = 5;
for (int i = 0; i < 6; i++) {
    if (price[i] > maximum)
        maximum = price[i];
}
int total_cost = maximum + celery
                 + tomato;
```

**Assembly Code**

```
// x4 = 0x24  - length of price in bytes
// x5 = 0x3   - celery
// x6 = 0x5   - tomato
// x7 = 0xA   - maximum
// x1 = 0x400 - address of price[0]

start:  addi x2, x0, 0
        slli x2, x2, 2
loop:   add  x8, x2, x1
        lw x3, 0(x8)
        bge x7, x3, skip
        mv x7, x3
        ori x7, x7, 0
skip:   addi x2, x2, 4
        blt x2, x4, loop
        add x7, x7, x5
        add x7, x7, x6
```

In the following five-stage pipelined RISC-V processor (IF, DEC, EXE, MEM, WB):

- All branches are predicted not-taken. (Always fetch from PC + 4).
- Branch decisions are made in the EXE stage.
- The pipeline has **full bypassing**.
- The processor annuls instructions following taken branches.
- Assume that in the first iteration of the loop **both branches are taken.**

(A) Her program just started running "start". **Fill in the pipeline diagram for the first 14 cycles. Show all bypass passed used in each cycle.**

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | addi | slli | add | lw | bge | mv | mv | mv | ori | addi | blt | add | add | add |
| DEC | | addi | slli | add | lw | bge | bge | bge | mv | NOP | addi | blt | add | NOP |
| EXE | | | addi | slli | add | lw | NOP | NOP | bge | NOP | NOP | addi | blt | NOP |
| MEM | | | | addi | slli | add | lw | NOP | NOP | bge | NOP | NOP | addi | blt |
| WB | | | | | addi | slli | add | lw | NOP | NOP | bge | NOP | NOP | addi |

(B) How many cycles did it take to execute the first loop iteration on this processor? Make sure not to include the first two instructions at label `start` in your cycle count.

**Cycles to execute first iteration of the loop on this processor: ___11____**

(C) If you could modify your fetch stage to always fetch the correct next instruction instead of predicting all branches not taken, how many cycles will it now take to execute the first iteration of the loop on this modified processor? Explain your answer.

**Removes both branch annulments, so saves 4 cycles, making the cycles per iteration 11-4 = 7.**

(D) Val spent so much money on vegan turkey that she couldn't afford a processor with bypassing. In the cheapo processor she bought, **all data hazards are resolved by stalling.** Also, once again, **all branches are predicted not taken.**

Her program just started running "start". **Fill in the pipeline diagram for the first 14 cycles:**

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | addi | slli | add | add | add | add | lw | lw | lw | lw | bge | bge | bge | bge |
| DEC | | addi | slli | slli | slli | slli | add | add | add | add | lw | lw | lw | Lw |
| EXE | | | addi | NOP | NOP | NOP | slli | NOP | NOP | NOP | add | NOP | NOP | NOP |
| MEM | | | | addi | NOP | NOP | NOP | slli | NOP | NOP | NOP | add | NOP | NOP |
| WB | | | | | addi | NOP | NOP | NOP | slli | NOP | NOP | NOP | add | NOP |

(E) How many cycles did it take to execute the first loop on this processor? Hint: To answer this question use the bypass and stall information you determined in parts (A) and (D) to calculate how many cycles the first iteration of the loop would take on the cheapo processor. **Explain how you arrived at your solution.**

**Cycles per iteration on this processor: ____24_____**

**Explanation: Without bypasses, there is a 3-cycle penalty for each EXE →DEC bypass used in the bypassed pipeline, and a 1-cycle penalty for every WB → DEC bypass used. In part B there were 4 EXE → DEC bypasses and 1 WB → DEC bypass, so cycles = 11 + 4*3 + 1*1 = 24.**

**Extra pipeline diagrams (for parts A and D):**

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **IF** | addi | | | | | | | | | | | | | |
| **DEC** | | | | | | | | | | | | | | |
| **EXE** | | | | | | | | | | | | | | |
| **MEM** | | | | | | | | | | | | | | |
| **WB** | | | | | | | | | | | | | | |

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **IF** | addi | | | | | | | | | | | | | |
| **DEC** | | | | | | | | | | | | | | |
| **EXE** | | | | | | | | | | | | | | |
| **MEM** | | | | | | | | | | | | | | |
| **WB** | | | | | | | | | | | | | | |

**Problem 8 (From Past Quiz)**

Piper Twain is building a RISC-V processor to accelerate machine learning applications in low-power systems. These applications execute mostly ALU operations, so Piper is building the *Twofer*, a new processor that can perform two ALU operations in a single instruction. She is imposing some restrictions on these operations to make the Twofer's implementation very cheap: the processor requires two ALUs but avoids using register files with many ports.

To this end, Piper designs a new RISC-V instruction type, **D-type** (for double-op), that encodes two ALU operations. In assembly, a D-type instruction is written as two operations separated by a semicolon:

```
aop ard, ars1, ars2; bop brd, brs1, brs2
```

For instance, the following two examples are valid D-type instructions:

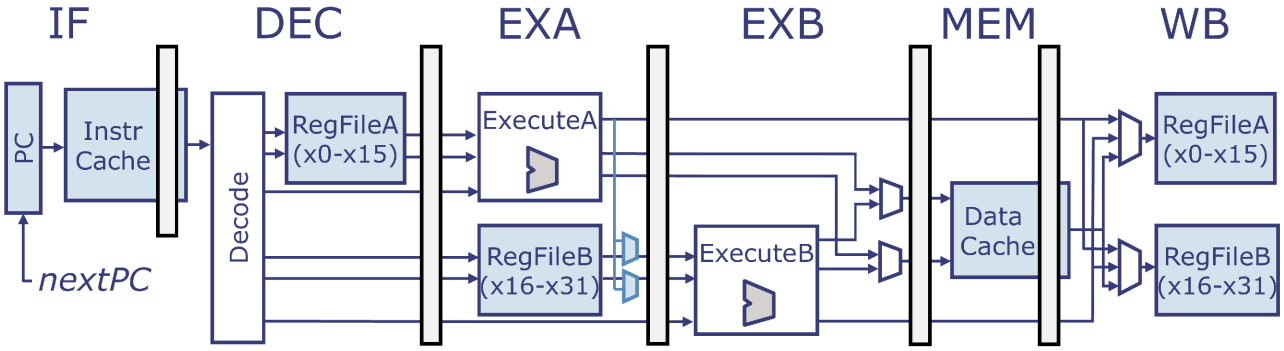
```
(1)   add x3, x2, x1; sub x27, x18, x25
(2)   and x18, x2, x1; xor x7, x18, x25
```

These fields have the following semantics and restrictions:

- The operations (`aop, bop`) can be any of the 10 ALU operations for R-type instructions (`add, sub, sll, srl, sra, slt, sltu, and, or, xor`).
- The source registers for operation A (`ars1,ars2`) must be within the top (lower index) 16 registers (`x0-x15`); and the source registers for operation B (`brs1,brs2`) must be within the bottom (higher index) 16 registers (`x16-x31`).
- The destination registers for operations A and B must be in separate halves of the registers: one operation must write to the top half of the registers (`x0-x15`), and the other must write to the bottom half (`x16-x31`), but `ard` and `brd` must never be in the same half.
- Within the same instruction, both operations have serial semantics. That is, if operation A writes to a register that operation B reads, operation B uses the value written by operation A (this may only happen if A writes to a bottom-half register and B writes to a top-half register). For example, in example (2) above, A (`and`) writes to `x18`, and B (`xor`) reads `x18`. The `xor` operation should use the value of `x18` produced by the `and` operation.

D-type instructions can be encoded in 32 bits and added to a RISC-V processor, but their specific encoding is not needed for this problem.

Piper has designed the following 6-stage pipeline for the Twofer:



This pipeline uses two separate register files, RegFileA and RegFileB, which hold the top and bottom halves of the RISC-V registers. Each of these register files has only 2 read ports and 1 write port, and 16 registers, so together they are as cheap as the standard 32-register RISC-V register file.

The IF and MEM stages are like those in the standard 5-stage pipeline. The DEC stage reads the source registers for operation A (from RegFileA). The EXA stage executes operation A and, in parallel, reads the source registers for operation B (from RegFileB). The EXB stage executes operation B. Both results are sent down to the MEM stage. Finally, the WB stage writes to both RegFileA and RegFileB; because operations A and B cannot both write to the same register file, a single write port suffices for each.

Having operations A and B happen in separate stages makes implementing serial semantics easy: if operation A writes to one of the source registers of operation B, the muxes at the outputs of RegFileB (colored in light blue in the figure) pass this updated value to EXB.

Beyond D-type instructions, this pipeline supports all other RISC-V instruction types. *Other instructions execute on EXA or EXB depending on the registers they use.* Every instruction performs its operation in EXA if its source register(s) are in the top half, and in EXB if in the bottom half. For instructions with two source registers in separate halves (this includes register-register ALU and branch instructions), one of the source registers is read in DEC, the other in EXA, and execution happens in EXB.

You don't need to draw pipeline diagrams to answer the following questions, and we recommend you don't. But if you find them useful, the last page has some blank Twofer pipeline diagrams.

(A) For each of the following code fragments, find out how many cycles are lost to stalls due to data hazards. **Assume full bypassing**, and for each example, list **all** bypass paths are used, if any. Note that bypassing here is more involved than in the 5-stage pipeline: bypassed data is not only routed to DEC. For each bypass path used, fill the given table to list the register whose value is bypassed and the stages it is bypassed from and to.

***Note:*** **Bypassing always happens from a later to an earlier pipeline stage, so the light-blue muxes in EXA used to implement serial semantics on operations A and B are not a bypass.**

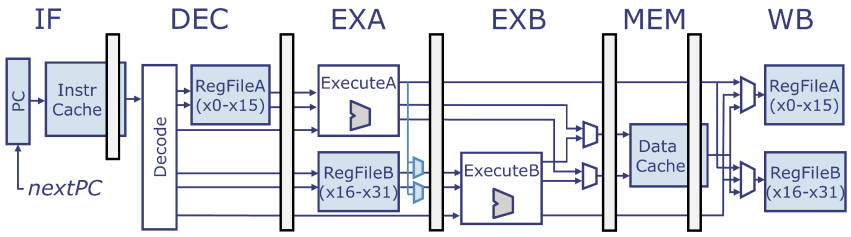
Example:

```
lw x20, 0(x2)
add x16, x20, x22
```

**Cycles lost to stalls: 2**
**Bypass paths:**

| Value of reg | Bypassed from | Bypassed to |
|---|---|---|
| **x20** | **WB** | **EXA** |
| | | |
| | | |

(since `add` runs on EXB)




1. `lw x2, 0(x2)`

   `add x2, x2, x2`

**Cycles lost to stalls: _3_____**

**Bypass paths:**

| Value of reg | Bypassed from | Bypassed to |
|---|---|---|
| x2 | WB | DEC |
| | | |
| | | |

add runs on EXA in this case

2. `add x1, x2, x1; or x17, x18, x17`

   `sub x1, x2, x1; xor x17, x18, x17`

**Cycles lost to stalls: _0_____**

**Bypass paths:**

| Value of reg | Bypassed from | Bypassed to |
|---|---|---|
| x1 | EXA | DEC |
| x17 | EXB | EXA |
| | | |

3. `add x1, x2, x1; or x19, x18, x17`

   `add x17, x1, x19`

**Cycles lost to stalls: _0_____**

**Bypass paths:**

| Value of reg | Bypassed from | Bypassed to |
|---|---|---|
| x1 | EXA | DEC |
| x19 | EXB | EXA |
| | | |

add runs on EXB. Note bypasses happen on different cycles. It's also correct to say x1 value is bypassed from EXB to EXA (then both bypasses would be in the same cycle)

4. `add x18, x2, x1; or x1, x18, x17`

   `add x1, x2, x1`

**Cycles lost to stalls: _1_____**

**Bypass paths:**

| Value of reg | Bypassed from | Bypassed to |
|---|---|---|
| x1 | EXB | DEC |
| | | |
| | | |

add runs on EXA

(B) Assume that the pipeline **always predicts branches not taken**. What is the minimum and maximum number of cycles lost to **taken branches** in this pipeline? Briefly explain what causes this penalty. How would you write code to minimize this penalty?

**Minimum number of cycles lost: _2_____**

**Maximum number of cycles lost: _3_____**

**What causes the penalty, and how would you write code to minimize it?**

**This penalty depends on whether branches are resolves in EXA (2 cycles) or EXB (3 cycles). To minimize this penalty, we should use branches that run on EXA, by operating on the bottom half of registers (x0-x15).**

(C) Consider the following RISC-V code. Assume that the pipeline has **full bypassing** and that the branch at the end of the loop is **predicted taken**, so that in steady state there are no cycles lost to control hazards. Rewrite the four R-type instructions shaded in grey, using D-type instructions to maximize performance. In steady state, how many cycles does each loop iteration take, in both the original code and your code?
Notes:

- You may have to change the registers used by these instructions to maximize performance.
- Registers x4, x5, x6, and x7 are written and read only by these instructions.
- Registers x10, x11, and x12 hold constant values. These constants are replicated in x20, x21, and x22, so you can use x20 in place of x10, etc.
- You may not modify the code (instructions or registers) outside of the shaded block.

```
loop: lw x16, 0(x1)
      addi x1, x1, 4
      srl x4, x16, x10
      sll x5, x7, x11
      or x6, x5, x12
      xor x7, x6, x4
      blt x1, x13, loop
```

**Rewrite shaded instructions:**

```
_sll x5, x7, x11; srl x17, x16, x20_____

_or x18, x5, x12; xor x7, x18, x17_____

_(x17 and x18 can be any bottom-half registers)_____
```

**Cycles per loop iteration for original code: __8_____**

(7 instructions + 1 stall cycle due to lw -> srl dependency, as srl runs on EXB)
**Cycles per loop iteration for your modified code: __6_____**

(5 instructions + 1 stall cycle due to lw -> srl dependency)

(D) Bypasses are expensive, so Piper is considering not implementing them. Are there any RISC-V instruction sequences that would incur some stalls in a conventional 5-stage pipeline without bypasses but that, when modified to use D-type instructions, would incur fewer stalls in the Twofer 6-stage pipeline without bypasses? **If so, give such a sequence; in any case, briefly explain why or why not.**

**Yes.** The following pair of instructions:

```
add x16, x1, x1
add x1, x16, x16
```

suffers a 3-cycle stall on the 5-stage pipeline, while the equivalent D-type instruction:

```
add x16, x1, x1; add x1, x16, x16
```

has no stalls.

**Blank Twofer pipeline diagrams (in case you find them useful)**

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | | | | | | | | | | | | | | |
| DEC | | | | | | | | | | | | | | |
| EXA | | | | | | | | | | | | | | |
| EXB | | | | | | | | | | | | | | |
| MEM | | | | | | | | | | | | | | |
| WB | | | | | | | | | | | | | | |

| Cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IF | | | | | | | | | | | | | | |
| DEC | | | | | | | | | | | | | | |
| EXA | | | | | | | | | | | | | | |
| EXB | | | | | | | | | | | | | | |
| MEM | | | | | | | | | | | | | | |
| WB | | | | | | | | | | | | | | |