University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

CS152/252A
Spring 2024

C. Fletcher
2/27/24

Midterm Exam 1

Name: _____

Student ID number: _____

You have 80 minutes to take the exam.

This is a *closed-book, closed-notes* exam, except for one two-sided cheat sheet. Also no calculators, phones, pads, or laptops allowed.

Each question is marked with its number of points (one point per expected minute of time). Start by answering the easier questions then move on to the more difficult ones. You can use the backs of the pages to work out your answers. Neatly copy your answer to the allocated places. **Neatness counts.** We will deduct points if we need to work hard to understand your answer.

Write the student ID numbers of the person to your left and to your right on this page. If you are sitting on an aisle, just indicate "aisle" for left or right.

Left student ID number: _____

Right student ID number: _____

Before you turn in your exam, write your student ID number on all pages.

Handout 1 and scratch paper are provided at the end of the exam booklet.

March 16, 2024

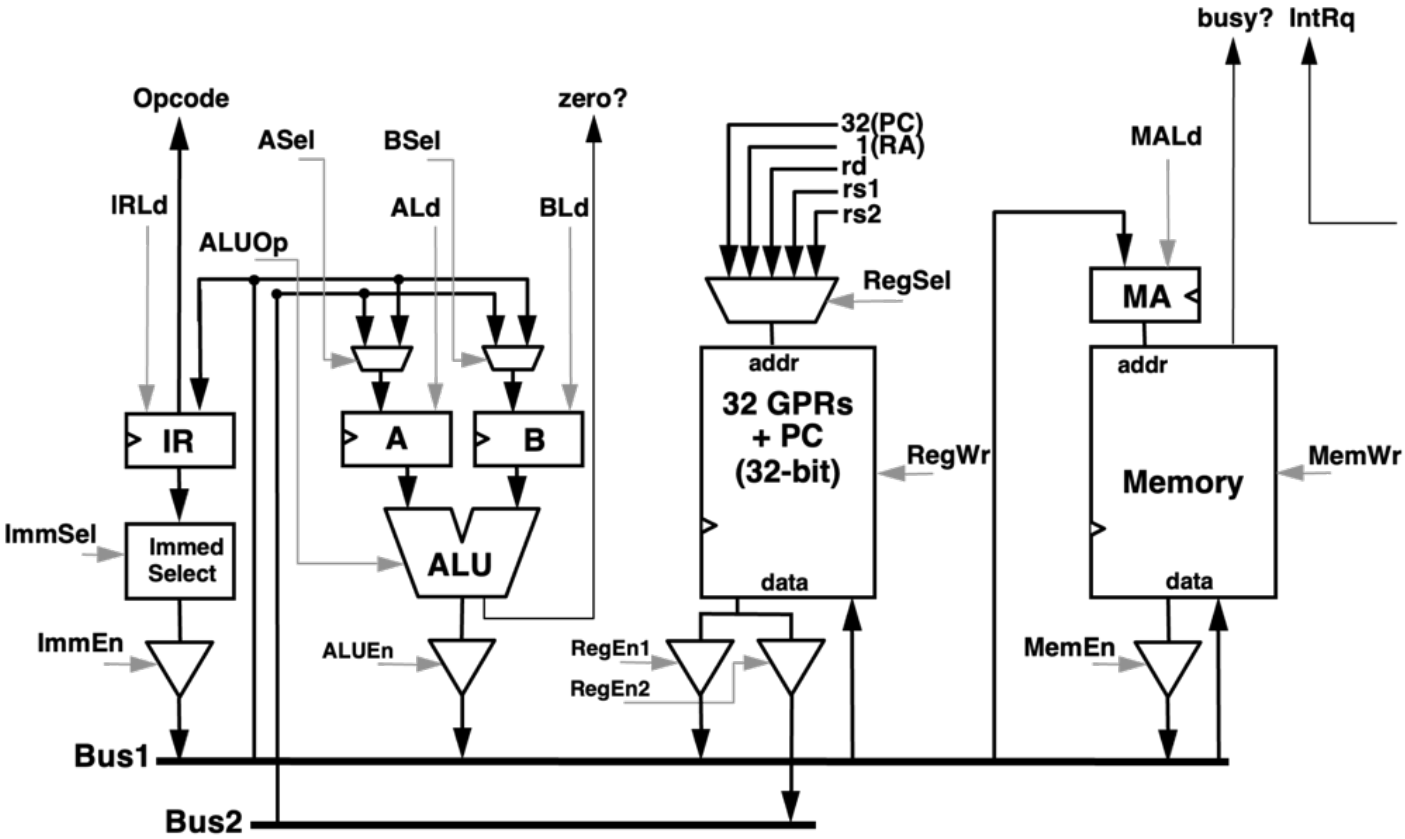
1 [20 points] Microcoding

One of the main issues with the single-bus datapath we've seen in class is that the bus frequently causes a structural hazard. To alleviate this issue, we will add a second bus to this design, called Bus2. However, to cut down on costs, we will only allow the RegFile write to this bus. Additionally, only the A and B registers will be able to read from this bus. Below is a description of the control signals we have to add:

RegEn1 and **RegEn2**: These signals replace RegEn from the original design. RegEn1 enables writing to Bus1, and RegEn2 enables writing to Bus2. These signals may be enabled simultaneously, allowing the RegFile to write the same data to both buses in the same cycle.

ASel and **BSel**: These signals determine which bus the A and B registers will load from during that cycle. They may be set to 1 or 2 to read from Bus1 or Bus2, respectively.

Below is a diagram of this double-bus approach, including the new control signals:



Your task in this question is to write microcode for this new design to implement the SetEqualMem instruction:

```
SetEqualMem rd, rs1, imm(rs2)
```

This instruction checks if the value in register rs1 is equal to the value stored in memory address rs2+imm. If they are equal, rd is set to 1. Otherwise, rd is set to 0. You may assume that reading from rs2+imm will not raise an exception. This instruction uses a new immediate type called X. Fill out the attached microcode table to implement this function. Your microcode should only use the optimal, minimal number of lines possible to implement SetEqualMem on the double-bus CPU. Additionally, your microcode is not allowed to modify either of the source registers (rs1 or rs2).

Some columns of the microcode table are gray. These columns will not be graded and you do not need to fill them out.

Focus first on making your implementation correct. Then, focus on making your correct implementation optimal. The majority of points will be awarded for your pseudocode.

State	Pseudocode	IR Ld	Reg Sel	Reg Wr	Reg En1	Reg En2	A Ld	B Ld	A Sel	B Sel	ALUOp	ALU En	MA Ld	Mem Wr	Mem En	Imm Sel	Imm En	μBr	Next State
FETCH0:	MA ← PC; A ← PC	*	PC	0	1	0	1	*	1	*	*	0	1	0	0	*	0	N	*
	IR ← Mem	1	*	0	0	0	0	*	*	*	*	0	0	0	1	*	0	S	*
	PC ← A+4	0	PC	1	0	0	0	*	*	*	INC_A_4	1	*	0	0	*	0	D	*
. . .																			
NOP0:	microbranch back to FETCH0	*	*	0	0	0	*	*	*	*	*	0	*	0	0	*	0	J	FETCH0
SetEqual Mem:	A<-R[rs2] B<-imm	0	rs2	0	0	1	1	1	2	1	*	0	*	0	0	X	1	N	*
	MA<-A+B A<-R[rs1]	0	rs1	0	0	1	1	*	2	*	ADD	1	1	0	0	*	0	N	*
	B<-Mem	0	*	0	0	*	0	1	*	1	*	0	0	0	1	*	0	S	*
	A,B<-A-B If A!=B, go to NOT_EQ	0	*	0	0	*	1	1	1	1	SUB	1	*	0	0	*	0	NZ	NOT_EQ
	R[rd]<-A+1 J FETCH0	*	rd	1	0	*	*	*	*	*	INC_A_1	1	*	0	0	*	0	J	FETCH0
NOT_EQ	R[rd]<-A-B J FETCH0	*	rd	1	0	*	*	*	*	*	SUB	1	*	0	0	*	0	J	FETCH0

*We also accept 0 instead of * for RegEn2. We can use * since there is never another unit writing to Bus2 simultaneously.

2 [20 points] Iron Law

For this problem, assume that we are working with the fully-bypassed five-stage pipelined processor shown in lecture.

(1) [1 pt each] Suppose that we include a hardware prefetcher. Explain how it would affect each term from the Iron Law.

(a) Instructions / Program:

Instructions per program is unchanged, as this is a purely microarchitectural optimization.

(b) Cycles / Instruction:

Cycles per instruction is likely to decrease assuming the prefetcher is providing timely / useful prefetches, as memory instructions will not be stalled. However, a bad prefetcher could pollute the cache and hurt performance by evicting actually useful data.

(c) Time / Cycle:

Time per cycle is unlikely to change, as the prefetcher should not be on the critical path of the processor.

(2) [1 pt each] Suppose we extended the RISC-V Base ISA with a conditional move instruction which copies `rs1` to `rd` if and only if `rs2` is nonzero. Explain how it would affect each term from the Iron Law.

(a) Instructions / Program:

Instructions per program stays roughly the same or decreases, as short branch sequences could get replaced by a single conditional move.

(b) Cycles / Instruction:

Cycles per instruction can either be impacted very little (if the branch predictor already does a very good job on such sequences) or improves due to the branches it replaces being hard to predict.

(c) Time / Cycle:

Cycle time could either increase due to increased control circuitry to support the new instruction, or it might be overshadowed by a different critical path in the processor.

(3) [1 pt each] Under certain settings, compilers will replace all instances where a function is called with the body of the function (a common optimization known as *inlining*). Explain how it would affect each term from the Iron Law.

(a) Instructions / Program:

Instructions per program decreases, though the binary size could increase (less overhead of call / return instructions, can bypass calling convention and possibly perform more aggressive optimizations)

(b) Cycles / Instruction:

CPI can be hurt due to larger pressure on I$ if the function being inlined is too big, but the aforementioned reasons for lower instruction count could also improve CPI

(c) Time / Cycle:

Time per cycle is totally unchanged, as this has no impact on microarchitecture.

(4) A very common task for computers is computation of dense matrix-matrix products. At the core of these routines is a microkernel, a highly optimized piece of code to compute a small matrix product. For this problem, assume we want to compute $C = AB$, where A, B, and C are all 2×2 matrices. Alice writes the following implementation:

```
int a0, a1;
for (int i = 0; i < 2; i += 1) {
    a0 = A[i][0];
    a1 = A[i][1];
    for (int j = 0; j < 2; j += 1) {
            C[i][j] = a0 * B[0][j] + a1 * B[1][j];
    }
}
```

On the other hand, Bob writes the following implementation:

```
int c00 = 0, c01 = 0;
int c10 = 0, c11 = 0;
int a0, a1, b0, b1;
for (int k = 0; k < 2; k += 1) {
    a0 = A[0][k];
    a1 = A[1][k];
    b0 = B[k][0];
    b1 = B[k][1];
    c00 += a0 * b0;
    c01 += a0 * b1;
    c10 += a1 * b0;
    c11 += a1 * b1;
}
C[0][0] = c00;
C[0][1] = c01;
C[1][0] = c10;
C[1][1] = c11;
```

(a) [6 pts] For this problem, we entirely ignore control flow instructions (jumps, branches) and loop variable increments. Using the RISC-V base ISA, what proportion of instructions are memory instructions? What proportion of instructions are arithmetic instructions? Assume that A, B, and C live in memory while other variables reside in registers (e.g. indexing into A requires a load or store while using `a0` once the value is loaded does not).

Alice's approach requires 4 memory operations per matrix element and 3 arithmetic operations, so 4/7 and 3/7. In particular, notice that there are two multiplies and one add per element, and that we store once for each element (into C) and load a full row of A (2 elements) and a full column of B (2 elements) for each element. However, loading a row of A is amortized over the elements of each row of C.

Bob's approach requires 3 memory operations per matrix element and 4 arithmetic operations, so 3/7 and 4/7. There are four arithmetic operations per element (two multiplies, two adds) and every element of A, B, and C is accessed exactly once.

(b) [1 pt] Assume memory instructions take 5 cycles to execute on average, while arithmetic instructions only take 1 cycle to execute. Whose approach achieves a lower CPI?

Alice: 4/7 * 5 + 3/7 * 1 = 23/7 CPI
Bob: 3/7 * 5 + 4/7 * 1 = 19/7 CPI
Thus, Bob's solution achieves a better CPI. This question is graded based on the answer to the previous part.

(c) [4 pts] Assume that both Alice's microkernel and Bob's microkernel are scaled up to 6×6. Do your results from the previous part still hold? Justify your answer qualitatively. (Hint: how many registers are available to hold temporary values?)

Alice:

```
int a0, a1;
for (int i = 0; i < 6; i++) {
    a0 = A[i][0];
    a1 = A[i][1];
    a2 = A[i][2];
    a3 = A[i][3];
    a4 = A[i][4];
    a5 = A[i][5];
    for (int j = 0; j < 6; j++) {
            C[i][j] = a0 * B[0][j] + a1 * B[1][j] +
                      a2 * B[2][j] + a3 * B[3][j] +
                      a4 * B[4][j] + a5 * B[5][j];
    }
}
```

Bob:

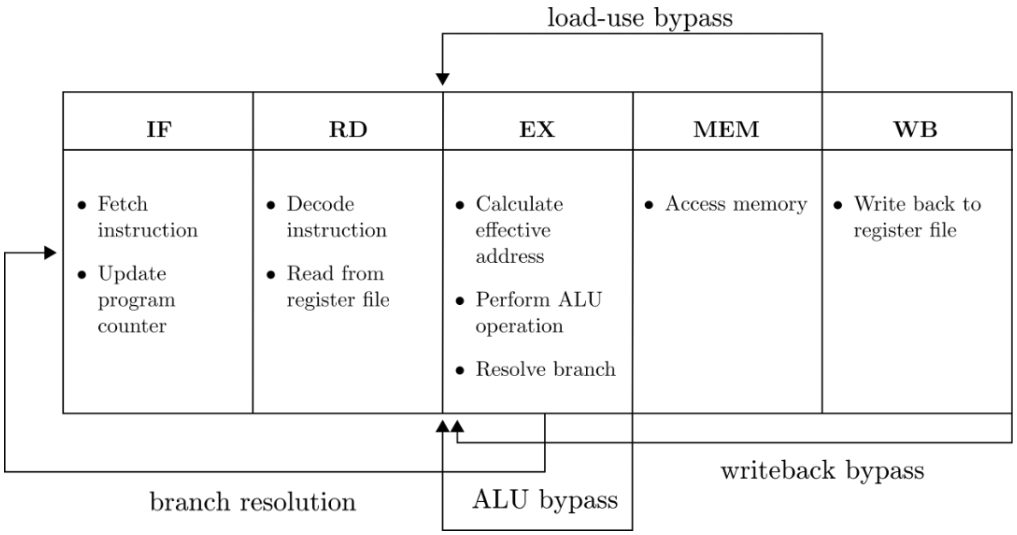
```
int c00 = 0, c01 = 0, c02 = 0, c03 = 0, c04 = 0, c05 = 0;
...
int c50 = 0, c51 = 0, c52 = 0, c53 = 0, c54 = 0, c55 = 0;

int a0, a1, ... a5;
int b0, b1, ... b5;
for (int k = 0; k < 6; k++) {
    a0 = A[0][k];
    a1 = A[1][k];
    ...
    a5 = A[5][k];

    b0 = B[k][0];
    b1 = B[k][1];
    ...
    b5 = B[k][5];

    c00 += a0 * b0;
    c01 += a0 * b1;
    ...
    c55 += a5 * b5;
}
C[0][0] = c00;
C[0][1] = c01;
...
C[5][5] = c55;
```

Using outer products is no longer more effective, since there aren't enough architectural registers (only 32) to do register blocking (at least 36 for just the elements of C). Spilling and restoring values from the stack negates Bob's advantage. In particular, to compute one element of C, Alice's solution needs 1 (amortized loading of 1 row of A) + 6 (loading one column of B) + 1 (storing) memory accesses while Bob's solution needs 2 (only load each element of A, B once) + 11 (save and restore to stack) + 1 (store to C) memory accesses, though this could vary depending on the stack spilling strategy of the compiler.

3 [20 + 3 points] Pipelining Potpourri

For this question, consider the standard fully-bypassed RV32I 5-stage pipeline.

- Branches are resolved at the end of the Execute stage
- Bypass paths bypass into the operand registers before the Execute stage (the bypass select muxes are in the Decode stage)
- There are no illegal opcodes, integer overflows, or illegal memory accesses.



The subsections of this question will consider the execution of the following loop on this pipeline.

```
1  loop:  lw t0, 0(a0)
2         add x1, x1, t0
3         sw x1, 0(a0)
4         lw a0, 4(a0)
5         bne x0, a0, loop
6  end:
```

(1) **[4 pts] Hazards**

(a) What hazards do each of the following microarchitectural optimizations address? (Select all that apply.)

i. [1 pt] Bypass Paths

□ RAW □ WAR □ WAW □ RAR □ Control □ Structural

RAW

ii. [1 pt] Branch Prediction

☐ RAW ☐ WAR ☐ WAW ☐ RAR ☐ Control ☐ Structural

Control

(b) [2 pts] What hazards are unresolvable by bypassing on the above pipeline? (Select all that apply.)

☐
```
lw t0, 0(a0)
add x1, x1, t0
```

☐
```
add x1, x1, t0
sw x1, 0(a0)
```

☐
```
sw x1, 0(a0)
lw a0, 4(a0)
```

☐
```
lw a0, 4(a0)
bne x0, a0, loop
```

lw/add, lw/bne

(2) **[5 pts] Pipeline Diagram** Analyze the above code sequence that runs on the fully-bypassed pipeline. Assume memory accesses take ***one cycle*** and the branch predictor predicts ***not taken***. For this question, fill in the pipeline diagram on the *next page* for the first iteration of this loop and the first instruction of the second iteration. **To reiterate, the pipeline diagram on the next page will be graded.**

(3) **[2 pts] CPI** What is the CPI (cycles per instruction) of 5 iterations of this loop?

CPI:

43/25 = (9*5-2)/(5*5)

(4) **[2 pts] Removing the Load-Use Bypass** Bypassing is expensive. Instead of keeping all of our bypass paths, we decide to remove the load-use bypass (the bypass path from the end of memory to the end of decode).

What is the CPI (cycles per instruction) of 5 iterations of this loop, with the load-use bypass removed? For your convenience, we have provided another pipeline diagram as scratch space, but it will *not be graded*.

CPI:

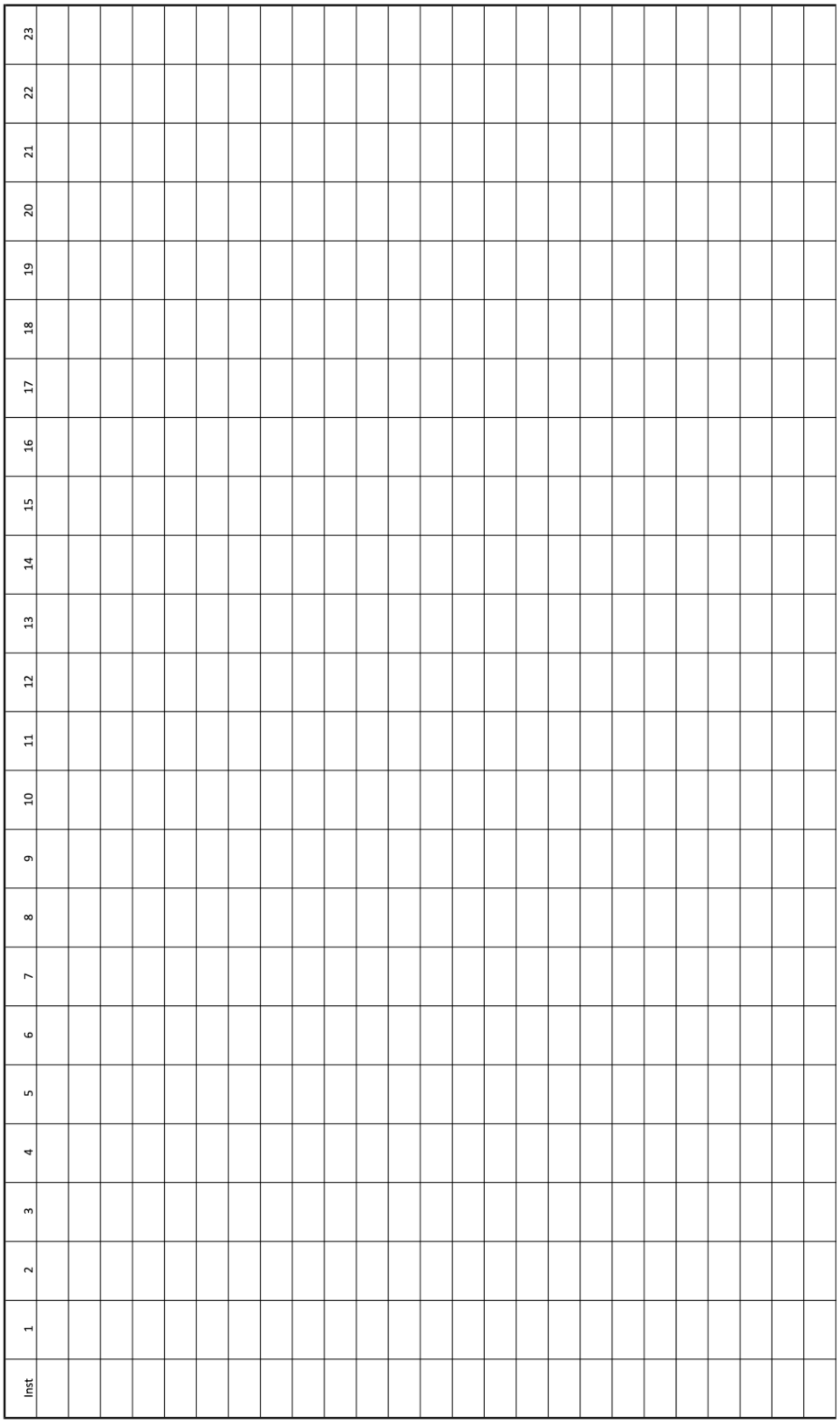
53/25 = (11*5-2)/(5*5)

(5) **[3 pts] Modifications** Which of the following modifications, if any, would improve (decrease) CPI for the above code sequence relative to the original fully bypassed pipeline that has a not-taken branch prediction scheme? (Select all that apply.)

- ☐ Adding a load delay slot
- ☐ Removing bypass paths
- ☐ Adding a branch delay slot
- ☐ Replacing the current branch prediction scheme with perfect branch prediction
- ☐ Splitting the memory stage into more stages, so memory access takes more cycles

Replacing the current branch prediction scheme with perfect branch prediction

[5 pts] Graded Pipeline Diagram: Fully Bypassed Pipeline



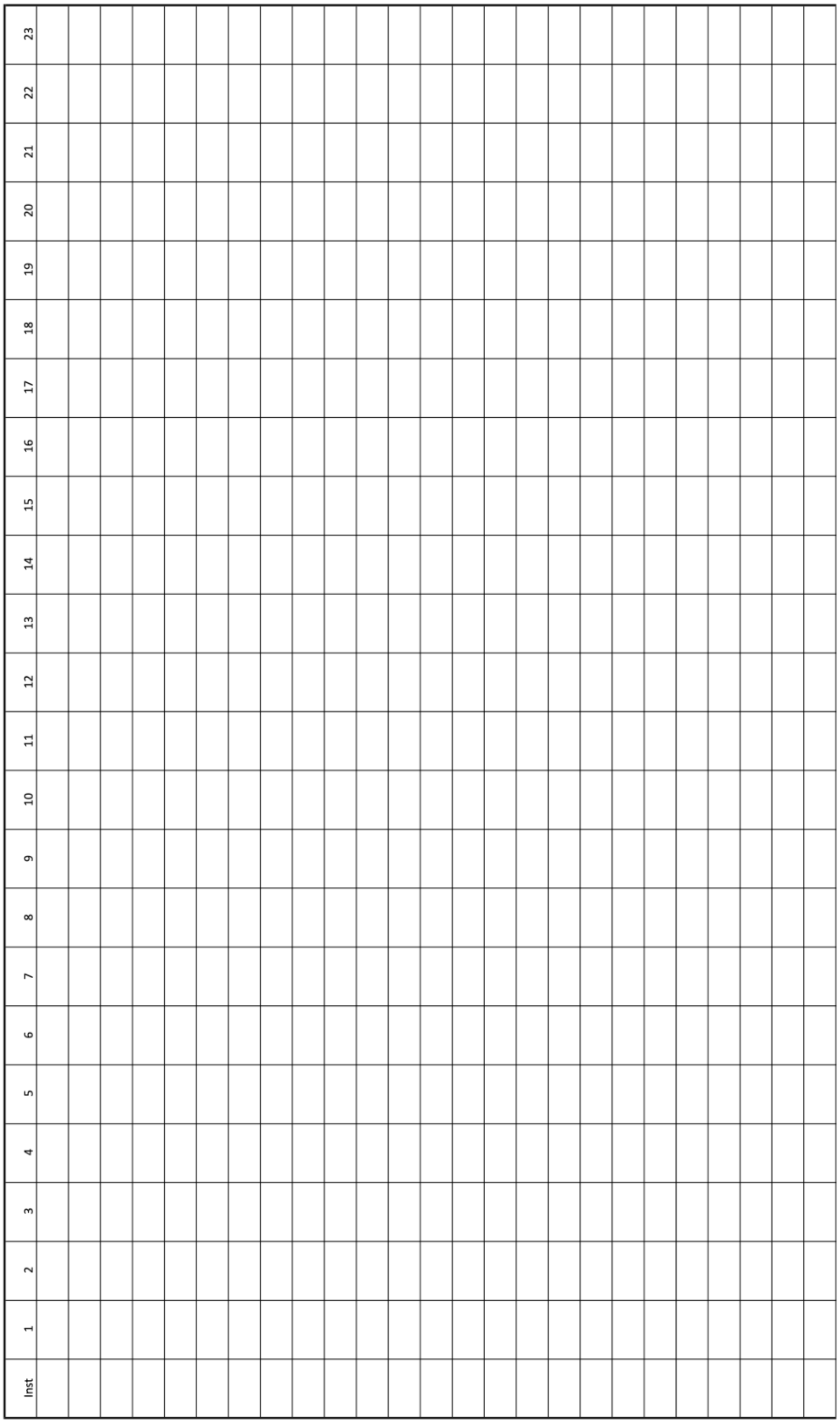
Graded Pipeline Diagram: Fully Bypassed Pipeline

	Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	lw t0, 0(a0)	F	D	X	M	W											
2	add x1, x1, t0		F	D	D	X	M	W									
3	sw x1, 0(a0)			F	F	D	X	M	W								
4	lw a0, 4(a0)					F	D	X	M	W							
5	bne x0, a0, loop						F	D	D	X	M	W					
6	lw t0, 0(a0)										F	D	X	M	W		
7																	

Ungraded Pipeline Diagram: Load-Use Bypass Removed

	Instructions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	lw t0, 0(a0)	F	D	X	M	W											
2	add x1, x1, t0		F	D	D	D	X	M	W								
3	sw x1, 0(a0)			F	F	F	D	X	M	W							
4	lw a0, 4(a0)						F	D	X	M	W						
5	bne x0, a0, loop							F	D	D	D	X	M	W			
6	lw t0, 0(a0)												F	D	X	M	W
7																	

[0pts - Optional] Ungraded Pipeline Diagram: Load-Use Bypass Removed



(6) **[4 pts] Exceptions** Instead of directly computing a sum, we would like to apply a function where we scale the components. We want our RISCV pipeline, detailed at the beginning of this question, to support a new complex integer instruction, so we add a new ISA extension. We add a new instruction, SCALE:

```
SCALE rd, rs1, rs2
R[rd] = (R[rs1] * 5 + 5) / R[rs2]
```

We introduce an unpipelined arithmetic functional unit in parallel with the ALU that has a 3 cycle latency. Because the execute stage is of variable latency, a younger instruction may commit before an older complex integer instruction, assuming no data dependencies. Assume the only exceptions are illegal opcode exceptions and arithmetic exceptions (e.g. divide by zero and integer overflows). Illegal opcode exceptions are detected in the Decode stage and arithmetic exceptions are detected in the Execute stage.

(a) [1 pt] Does the pipeline, as described, support precise exceptions? Select one.

□ Yes □ No

No

(b) [3 pts] If yes, how are precise exceptions supported? If not, how do we add support for precise exceptions? Your answer should be a maximum total of **three sentences**.

We must hold exception flags until the commit point to ensure exceptions are raised in order. Additionally, we must add interlocks accordingly to preserve completion order by stalling younger instructions if a SCALE instruction is executing. If an older SCALE instruction causes an exception, we raise this exception first and kill younger instructions.

(7) **[3 pts - Extra Credit] To Infinity and Beyond** For this part, we will start with a blank slate and not use the pipelines that we defined in the previous parts. Suppose you are given a combinational/single-cycle datapath where every instruction has the same latency of T ns. You are allowed to pipeline this datapath into P stages. Assume that for any P, there are a) no hazards of any type and b) no pipelining overheads due to digital logic-level effects. (In other words, you can assume that given P stages, the clock frequency can be increased by a factor of exactly P.)

Given any terminating program F, what is the Time / Program for F in the limit, as P → infinity, on this pipeline?

T

4 [20 points] Cache

Assume in this question that you have a 4KiB 2-way set associative L1 data cache with 64-byte cache lines, and that you are using 32-bit addresses.

(1) How many bits are required for the tag, index and offset?

$2pts$

index: 5 bit offset: 6 bits tag: 21 bits

(2) Now consider the following program. Assume that each long element is 8 bytes, and that the matrix elements are stored in row-major order. Assume that the first element in the matrix is aligned to the start of a cache line.

```
int sum = 0;
int matrix[64][64];
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        sum += matrix[i][j];
    }
}
```

(a)

$2pts$

What's the number of cache misses?

512

(b)

$2pt each, 8pt total$

What's the number of occurrences each of the following events?

i. Hits:

3584

ii. Compulsory misses:

512

iii. Conflict misses:

0

iv. Capacity misses:

0

(3) Suppose we add software prefetching. Assume that each iteration of the inner loop takes 30 cycles (when there is a L1 miss). Of those 30 cycles, 25 cycles is the L1 miss penalty while 5 cycles is taken by the addition operation. Ignore any overheads from executing other instructions, including the software prefetch instruction. The prefetcher only prefetches when **prefetch_cond** becomes true (which is set by instructions that are not shown, and that you likewise do not need to model).

```
long sum = 0;
long matrix[64][64];
for (int i = 0; i < 64; i++) {
    for (int j = 0; j < 64; j++) {
        if (prefetch_cond) prefetch(&matrix[i][j]+OFFSET);
        sum += matrix[i][j];
    }
}
```

(a)

$2pts$

What should **OFFSET** be set to in order to prefetch the correct value? The address calculation works on the granularity of bytes.

5

(b)

$2pts$

What should **prefetch_cond** be to minimize total number of cycles of execution?

(j+5) is divided by 8

(c)

$2pts$

How many prefetch requests would be issued?

512

(d)

$2pts$

With prefetching, how many cache misses are there?

64

5 [20 points] Virtual Memory

Let us once again consider multiplying two matrices, this time using the classical inner product approach for multiplying 2 NxN square matrices A and B and storing the result in an NxN matrix C. We use the following C language program with N = 512 in this problem.

```
#define N 512
double A[N][N], B[N][N], C[N][N];
...
for (int i = 0; i < N; i += 1) {
    for (int k = 0; k < N; k += 1) {
        for (int j = 0; j < N; j+= 1) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Note:

- `sizeof(<type>)` provides the size of a data type in the C language.
- Arrays are stored contiguously in the C language. `A[x]` refers to row x of A and `A[x][y]` to the element at row x and column y.

(1) [2 pts] Page Usage by Matrix A (Select all that are true.)

- ☐ In the least optimal case, matrix A can use $\frac{N*N*sizeof(double)}{PageSize} + 1$ pages when the start of A is not page aligned.
- ☐ In the most optimal case, matrix A can use $\frac{N*N*sizeof(double)}{PageSize}$ pages when the start of A is page aligned.
- ☐ Matrix A always uses $\frac{N*N*sizeof(double)}{PageSize}$ pages irrespective of page alignment.
- ☐ Matrix A always uses $\frac{N*N}{PageSize}$ pages irrespective of page alignment.

- ■ In the least optimal case, matrix A can use $\frac{N*N*sizeof(double)}{PageSize} + 1$ pages when the start of A is not page aligned.
- ■ In the most optimal case, matrix A can use $\frac{N*N*sizeof(double)}{PageSize}$ pages when the start of A is page aligned.
- ☐ Matrix A always uses $\frac{N*N*sizeof(double)}{PageSize}$ pages irrespective of page alignment.
- ☐ Matrix A always uses $\frac{N*N}{PageSize}$ pages irrespective of page alignment.

Your solution to questions (2) and (3) can use the following:

- Numerical Constants
- Arithmetic Operators: +, −, ∗, /, %
- Parentheses: (...)
- Variables: `sizeof(<type>)`, $PageSize$, $TLBSize$, $PageTableSize$, $CacheSize$, $CacheLineSize$

(2) [4 pts] Page Faults in Matrix Multiply

Given that we enter the matrix multiply loops with arrays A, B, and C entirely in disk, how many page faults does the matrix multiply operation incur? Assume each matrix starts at the beginning of a page and we have enough main memory to fit all 3 matrices.

For each matrix, we encounter a page fault for every $PageSize$ bytes we load. Therefore, the total number of page faults is given by

$$3 * \frac{N * N * sizeof(double)}{PageSize}$$

(3) [4 pts] TLB Usage in Matrix Multiply

What is the minimum number of entries we need in a fully-associative TLB to guarantee all TLB misses in the matrix multiply operation are only caused by page faults (i.e., there are only compulsory misses in the TLB)? Note the order of the three loops. Assume each matrix starts at the beginning of a page and we have enough main memory to fit all 3 matrices.

For each iteration of the outer (i) loop, we traverse the entire B matrix. Similarly, for each iteration of the outer (i) loop, across multiple iterations of the middle (k) loop, we traverse an entire row of the C matrix. However, we always traverse the matrix A element-by-element, never returning to the same element. Therefore,

$$\frac{N * N * sizeof(double)}{PageSize} + \frac{N * sizeof(double)}{PageSize} + 1$$

(4) [2 pts] Comparing Standard Pages and Hugepages

The standard page size provided by the Linux kernel is 4 KiB. The default hugepage size provided by the Linux kernel is 2 MiB. Which of the following is true for our matrix multiply operation, given the descriptions in (2) and (3)? (Select all that apply.)

- ☐ Hugepages incur more page faults than standard pages
- ☐ Standard pages incur more page faults than huge pages

- ☐ Hugepages lead to lesser TLB usage
- ☐ Standard pages lead to lesser TLB usage

- ☐ Hugepages incur more page faults than standard pages
- ■ Standard pages incur more page faults than huge pages
- ■ Hugepages lead to lesser TLB usage
- ☐ Standard pages lead to lesser TLB usage

(5) Broader Comparisons Between Standard Pages and Hugepages

The standard page size provided by the Linux kernel is 4 KiB. The default hugepage size provided by the Linux kernel is 2 MiB.

(a) [3 pts] What are the advantages of using hugepages in general? (Select all that apply.)

- ☐ Use fewer page table entries in general
- ☐ Greater TLB reach
- ☐ Lesser internal fragmentation
- ☐ Greater internal fragmentation
- ☐ Reduced TLB pressure
- ☐ Better performance for applications accessing large contiguous portions of memory

(b) [3 pts] What are the disadvantages of using hugepages in general? (Select all that apply.)

- ☐ Page faults can be much longer
- ☐ Lesser internal fragmentation
- ☐ Greater internal fragmentation
- ☐ Increased TLB pressure
- ☐ Inefficient at enforcing varying protection levels that change at intervals on the order of KiBs
- ☐ Expensive for sharing memory regions of size on the order of KiBs

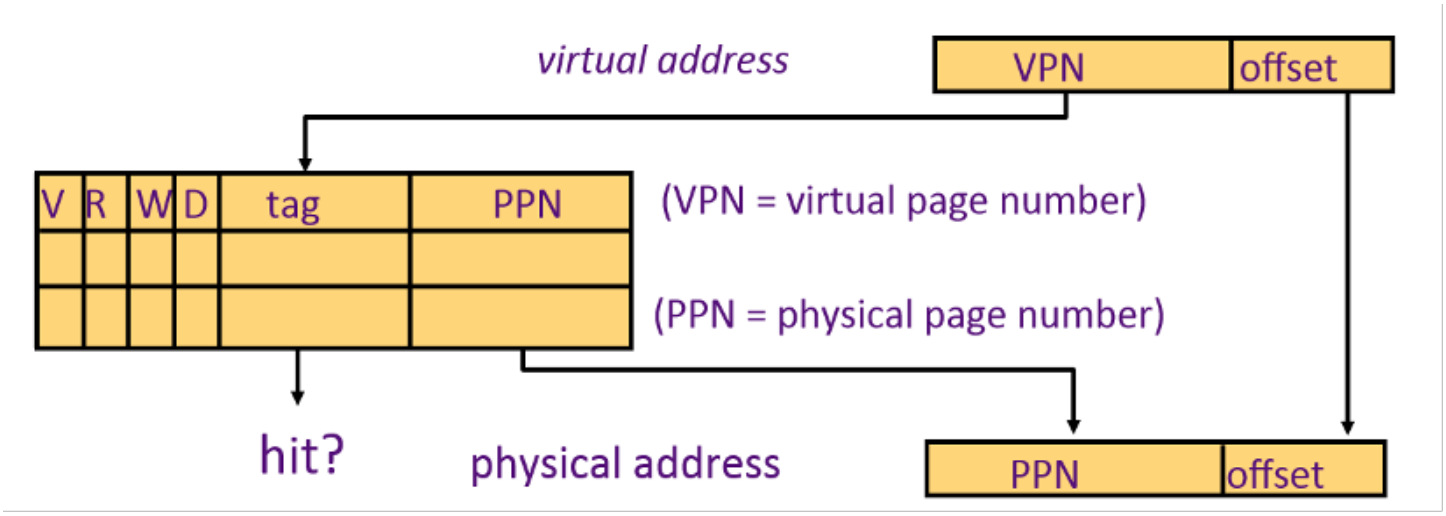
(a) [3 pts] What are the advantages of using hugepages in general? (Select all that apply.)

- ■ Use fewer page table entries in general
- ■ Greater TLB reach
- ☐ Lesser internal fragmentation
- ☐ Greater internal fragmentation
- ■ Reduced TLB pressure
- ■ Better performance for applications accessing large contiguous portions of memory

(b) [3 pts] What are the disadvantages of using hugepages in general? (Select all that apply.)

- ■ Page faults can be much longer
- ☐ Lesser internal fragmentation
- ■ Greater internal fragmentation
- ☐ Increased TLB pressure
- ■ Inefficient at enforcing varying protection levels that change at intervals on the order of KiBs
- ■ Expensive for sharing memory regions of size on the order of KiBs

(6) [2 pts] Does a TLB (as shown below) need any more information to support multiple page sizes at the same time? If so, why? If not, how can the existing TLB support multiple page sizes at once?



Yes, without knowing the page size for a memory access, we cannot identify the Tag, Index, and Offset bits. Without the Tag (and Index) bits, we simply cannot identify which set to look up and what bits of the VA to use for tag comparison.

For more on how to support multiple-page sizes in TLBs, see Q4.6 of the 252a Midterm 1 Exam.

Scratch Paper