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

CS152/252A — C. Fletcher
Spring 2024 — 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.

February 30, 2024 06:50

1 [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:

(b) Cycles / Instruction:

(c) Time / Cycle:

(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:

(b) Cycles / Instruction:

(c) Time / Cycle:

(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:

(b) Cycles / Instruction:

(c) Time / Cycle:

(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).

(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?

(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];
    }
}
```

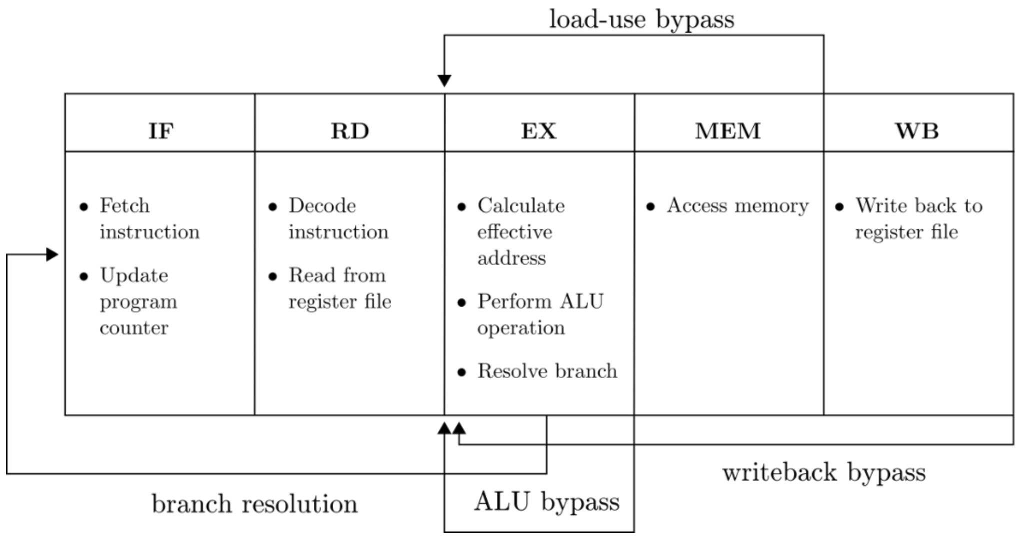
Bob:

```
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;
```

2 [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

ii. [1 pt] Branch Prediction
☐ RAW ☐ WAR ☐ WAW ☐ RAR ☐ Control ☐ Structural

(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
```

(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:

(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:

(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

[5 pts] Graded Pipeline Diagram: Fully Bypassed Pipeline

Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

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

Inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

(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

(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**.

(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?

3 [8 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) 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) [2 pts] What should **OFFSET** be set to in order to prefetch the correct value? The address calculation works on the granularity of bytes.

(b) [2 pts] What should **prefetch_cond** be to minimize total number of cycles of execution?

(c) [2 pts] How many prefetch requests would be issued?

(d) [2 pts] With prefetching, how many cache misses are there?

4 [27 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.

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.

(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.

(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

(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

(6) [9 pts] Which side do I look?

(Source: Hadi Brais'response to "Address translation with multiple pagesize-specific TLBs" on Stack Overflow)

Supporting multiple page sizes in a set-associative TLB introduces significant complexity. Depending on the page size, the page offset, tag, and index bits change. Most commercial processors use one of two designs to deal with this issue.

The first is by using a parallel TLB structure where each **set-associative TLB** is designated for page entries of a particular size only. All TLBs are looked up in parallel. There can either be a single hit or all misses. (We elide the issue of multiple hits here. Handling this can be specified in the hardware and/or constrained by the OS.)

The second is by using a **fully-associative TLB** with a tag-mask field in each entry and the following scheme:

(a) Let T_1 be the largest supported tag size (corresponding to the smallest supported page size) and T_2 be the smallest supported tag size (corresponding to the largest supported page size)

(b) Tags are stored in the TLB with T_1 bits; "overflow" 0s are appended to smaller tags to bring them up to T_1 bits.

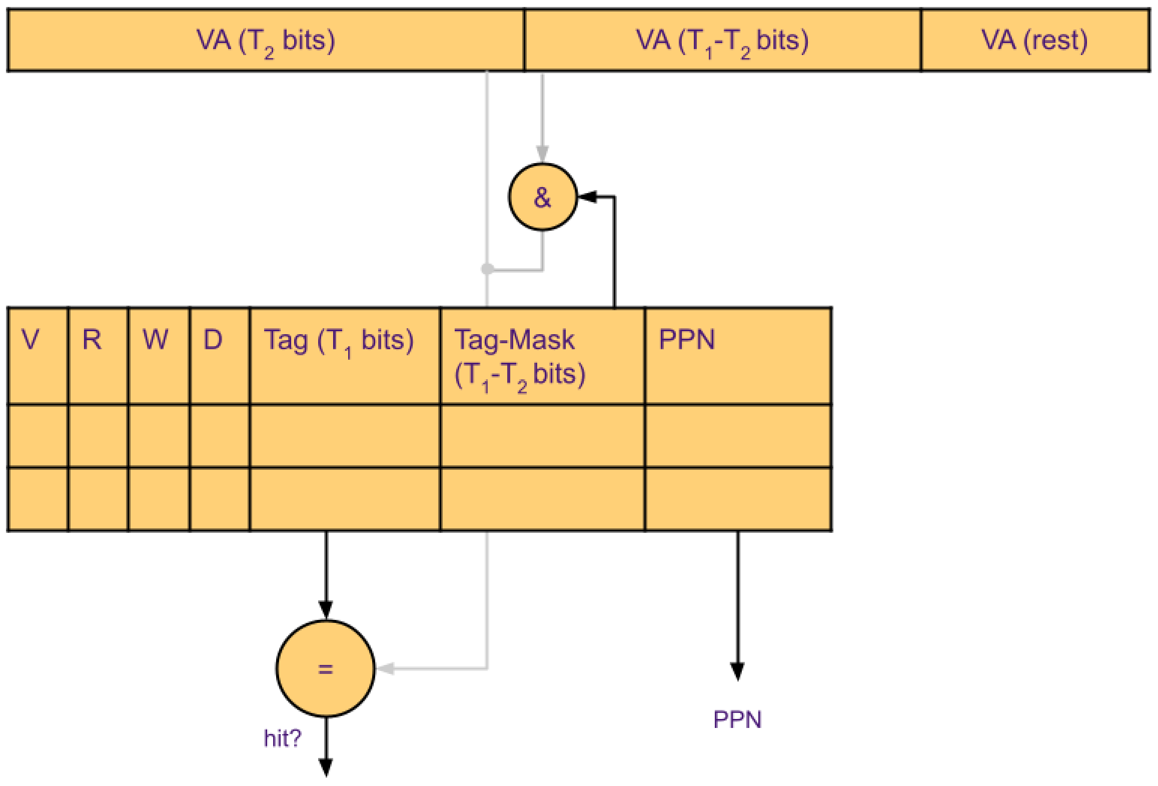
(c) A tag-mask of size $(T_1 - T_2)$ bits has 0s corresponding to the "overflow" 0 bit positions, and 1s otherwise.

(d) Upon a TLB access, the upper T_1 bits of VA are &-ed with the tag-mask. The & will set VA bits in the overflow bit positions to 0s and leave the rest untouched

(e) We compare the now masked upper T_1 bits of the VA to the TLB tags

That is, instead of maintaining separate TLB entries for each page size, the TLB can check for hits by using the additional tag-mask field and account for the page size during comparison using additional circuitry.

Here's a figure illustrating how this works.



Compare and contrast the two design points described above along the following axes.

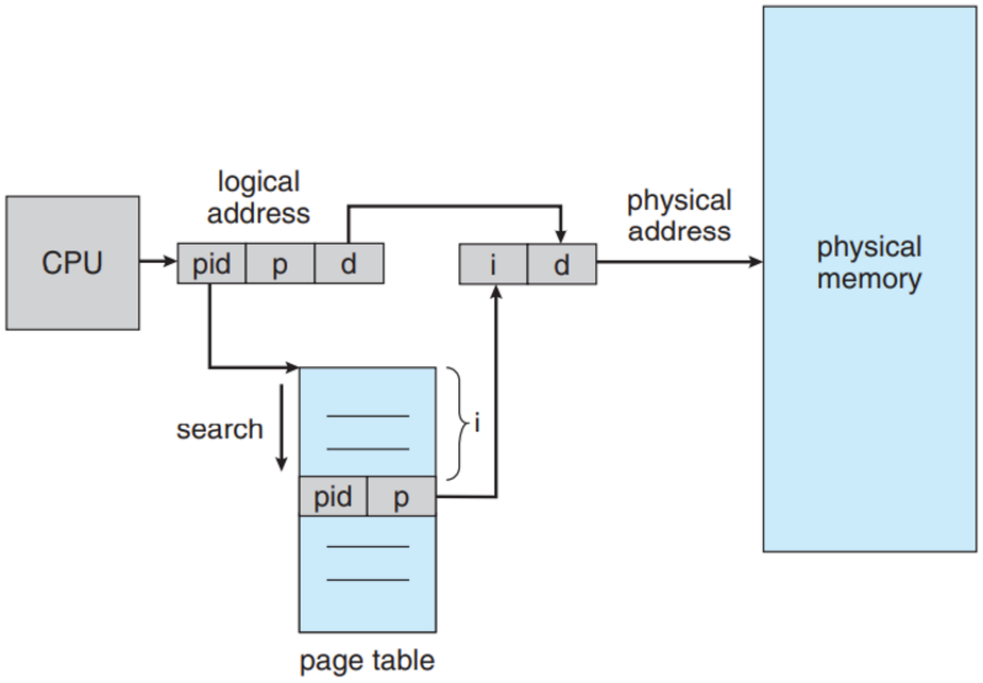
(a) [3 pts] Hit Time - The time required for TLB lookup

(b) [3 pts] TLB Utilization - Utilization of given total SRAM size during program execution

(c) [3 pts] Energy Efficiency - Energy consumed per lookup and during total program execution

5 [15 points] Inverted Page Tables

Each inverted page-table entry is a pair (process-id, page-number) where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of (process-id, page-number), is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry i— then the physical address (i, offset) is generated. If no match is found, then an illegal address access has been attempted.



Assume the following for this question.

1. You are using 32 bit addressing mode.
2. You have 256B page size
3. You have 512MiB of physical memory
4. Assume that the size of the metadata field is 3 bits (valid, read, write)

(1) [3 pts] Assuming that you are using a single level page table, how much memory are you using for the page tables?

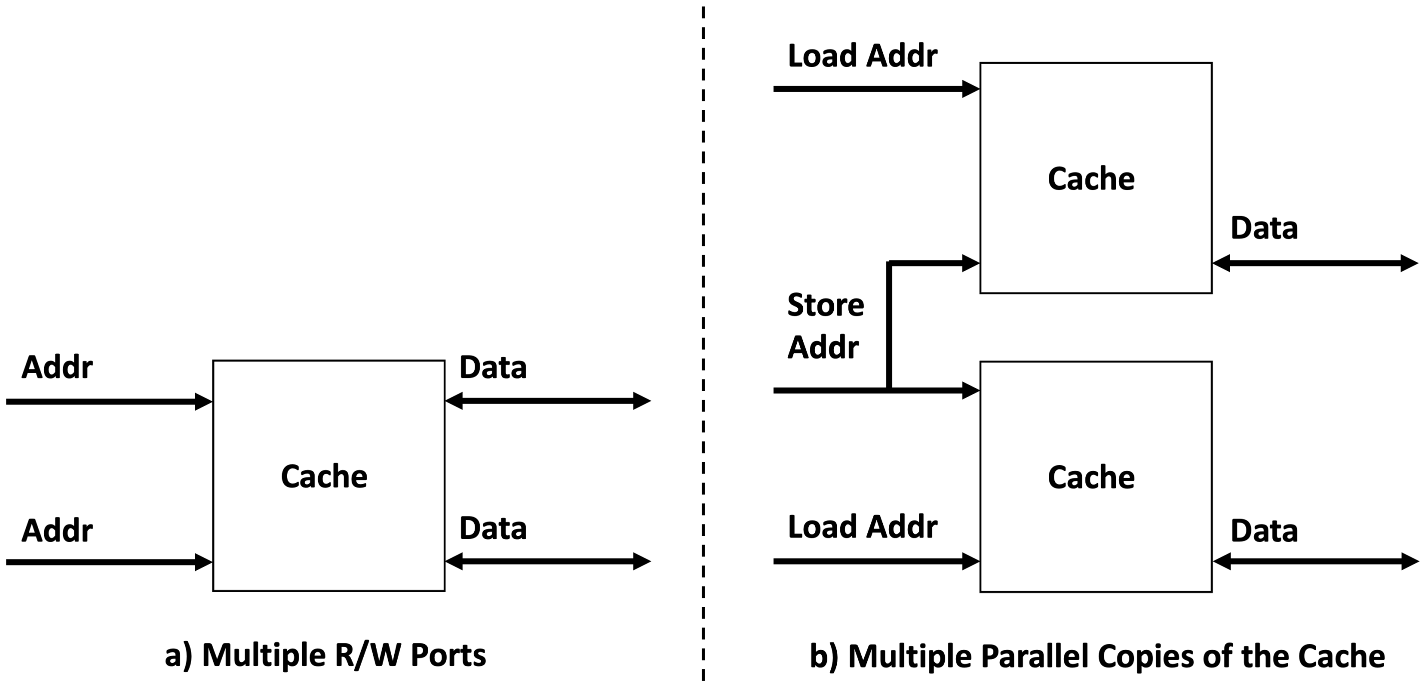
(2) [4 pts] Assuming that you are using an inverted page table, how much memory are you using for the page tables?

(3) [4 pts] Assuming that you are using 65MiB of your physical memory and using a 4 level page table, how much memory are you using for the page tables?

(4) [4 pts] What is the benefit of an inverted page table and when would you want to use it? What are the two downsides of using an inverted page table?

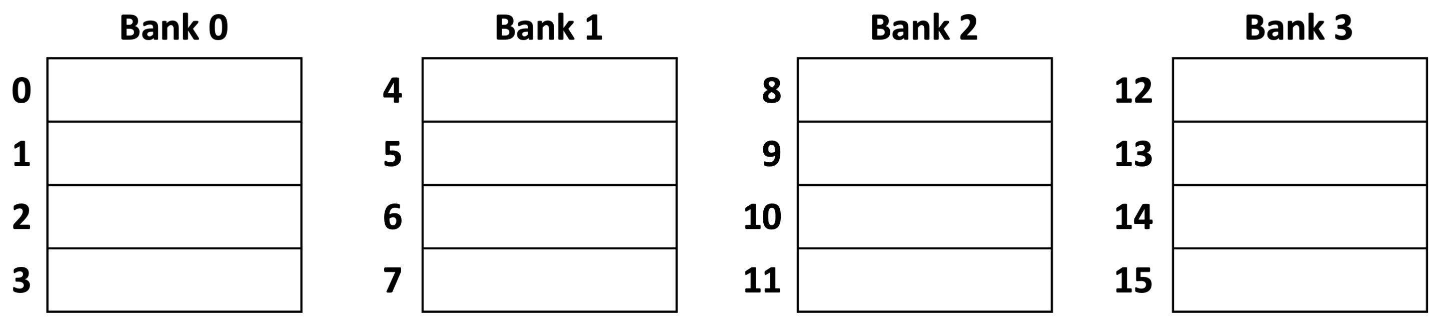
6 [15 points] Optimizing Cache Bandwidth

One challenge with caches is that we may have multiple memory operations that want to access the cache at the same time (leading to a structural hazard). Suppose that we want to optimize a cache to be able to support multiple parallel requests. We will now walk through different design tradeoffs to support higher cache bandwidth.



(1) [5 pts] One option to support higher cache bandwidth is to add more ports to the cache, as shown in (a). Alternatively, we could instead maintain multiple parallel copies of the cache as shown in (b). Describe the tradeoffs between supporting higher cache bandwidth through these approaches.

(2) An alternative way to support higher cache bandwidth is to divide the cache into banks, which are separate physical memories that are each used to store a subset of the data in the cache. Each bank has a single R/W port, so while they can be accessed in parallel, multiple requests to the same bank would still conflict. Assume that the cache is direct-mapped with 16 4-byte cache lines, with 4 cache lines in each bank.



For the following questions, assume that the hit time for each bank is 5 cycles, and that your memory stage is pipelined. Also assume that each bank cannot service any additional memory requests until it returns the previous request.

(a) [4 pts] Describe how you would partition sequential elements in an array of 4-byte integers in the cache in order to minimize the time to read the elements from the array in sequential order.

(b) [3 pts] In order to support this access pattern, which bits from the index in the virtual address should be used to select between banks?

(c) [3 pts] Assuming that you are using the index bits from b) to select between banks, would this still work well if you were iterating through the array with a stride of 4?

Scratch Paper