Computer System Architecture
6.5900 Quiz #1
October 11th, 2024

Name: Solution

This is a closed book, closed notes exam.
80 Minutes
14 Pages (+1 Scratch)

Notes:

- Not all questions are of equal difficulty, so look over the entire exam and budget your time carefully.
- Please carefully state any assumptions you make.
- Show your work to receive full credit.
- Please write your name on every page in the quiz.
- You must not discuss a quiz's contents with other students who have not yet taken the quiz.
- Pages 15 is a scratch page. Use it if you need more space to answer one of the questions, or for rough work.

Part A	_____	35 Points
Part B	_____	30 Points
Part C	_____	35 Points
TOTAL	**_____**	**100 Points**

Name _____

Part A: Caches and Virtual Memory (35 Points)

In this part, we will consider the memory system of the fictional Quadium processor. We will begin with a very simple version of this processor then refine/improve it in various ways across the questions. Across all parts, assume that the system has the following characteristics:

- 32-bit addresses (both physical and virtual)
- all memory accesses are 4-byte aligned
- 4KB page size
- 2 level page table stored in memory
- the 1st level page table base is stored in a special register (cr3)
- a single-level, physically addressed cache with 1-cycle hit latency. A cache miss takes 100 cycles end-to-end
- an MMU that handles page table walks
- page table data can be cached

Question 1 (5 points)

The initial prototype Quadium processor does not feature a TLB. Suppose a program tries to load virtual address 0x1000. What are the minimum and maximum possible access latencies? Ignore all sources of latency outside the memory access. Explain your reasoning.

3 accesses: 1 access L1 page table + 1 access to L2 page table + 1 access to data

Minimum 3 cycles, maximum 300

Name _____

Question 2 (5 points)

That seems slow! To address the long memory access latencies, the designers of the Quadium processor propose adding a single-entry hardware-managed TLB. The TLB is used on every access and shared between instruction and data memory. At the end of each access, hardware updates the TLB to contain the last translation.

There is a problem with design. Please provide both a scenario where this design will work well and a scenario where it will work poorly. Short code snippets are a clear way of illustrating examples.

Works well: instructions that don't touch memory

```
lbl: add x1, x2, x3
     j lbl
```

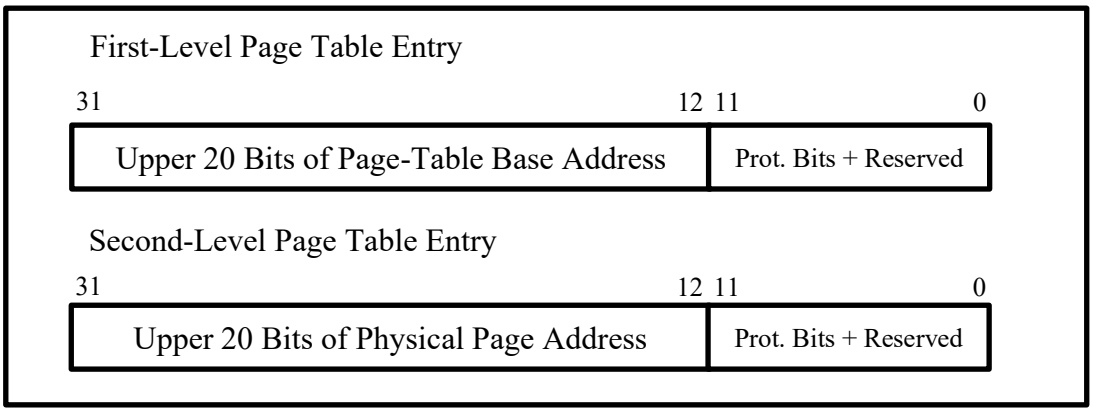
Works poorly: instructions that do touch memory - instruction fetch and data load will thrash each other

```
lbl: lw x1, offset(x2)
     j lbl
```

Name _____

Question 3 (5 points)

Quadium page table entries have exactly the following format (no super pages).



Both the first- and second-level page tables contain 1024 entries.

Suppose that a process wants to map exactly 1GB of *contiguous* virtual memory from address 0x0 to address 0x3fffffff. How many bytes of memory would the requisite page tables consume in memory?

2^30 bytes / 4096 (bytes / page) = 2^18 pages

2^18 pages / (2^10 pages / L2 page table) = 2^8

2^8 L2 page tables + 1 L1 page table = 257 total page tables

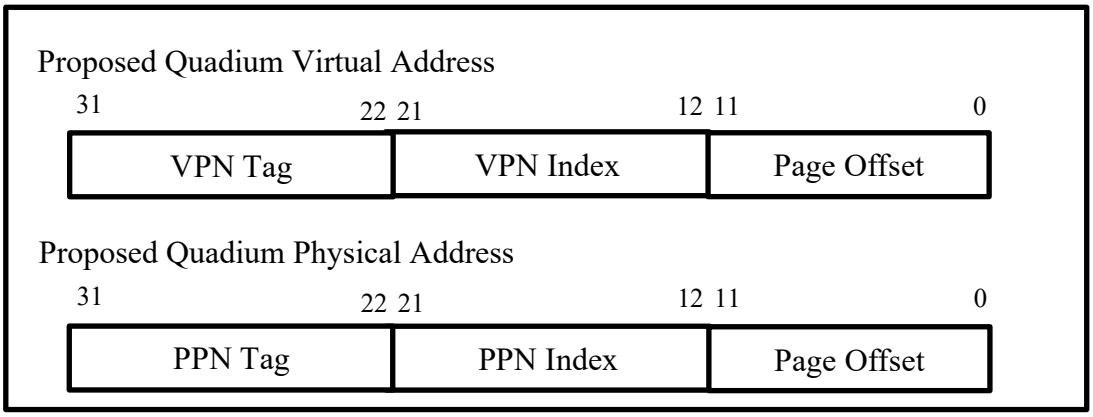
257 * (4096 bytes / page table) = 1052672 bytes

Name _____

Question 4 (20 points)

So far, the page tables we've studied in this course represent a "fully associative" mapping in that any page in virtual memory can be mapped to any page in physical memory. Note how this is *unlike* set-associative caches, where a specific address is mapped only to a *single* cache set.

In the next two questions, we will consider applying similar mapping restrictions to virtual memory. Specifically, we will subdivide the virtual page number (VPN) into two parts: a "VPN Tag" and a "VPN Index" as follows:



Our proposed restriction is that for every valid translation, the virtual address's Virtual Page Index bits *must match* its corresponding physical address's Physical Page Bits. Pages are still 4KB, so the page offset works the same way as it did before.

Question 4.1 (3 points): Please state whether each of the following three virtual address to physical address mappings are valid under this restriction:

- VA: 0xaafdc000 → PA: 0xdcfdc000 valid
- VA: 0xaafdd000 → PA: 0xdcfdc000 invalid
- VA: 0x447dc000 → PA: 0xdcfdc000 valid

Question 4.2 (5 points): In ordinary address translation, the bottom 12 bits (page offset) of both the virtual and physical address must match. In this scheme, the bottom 22 bits must match. However, this scheme *is* different from just having 2^{22} byte pages. Please explain how.

With 2^22 page size, 2^22 byte chunks need to be contiguous in both virtual and physical memory.

With 2^12 pages and the mapping restriction, even though we can't map every virtual page to every physical page, a single 2^22 byte region of virtual memory can be mapped to many discontiguous pages in physical memory.

Name _____

Question 4.3 (7 points): Suppose that instead of a physically addressed cache, we re-engineer the Quadium with a virtually indexed physically tagged (VIPT) cache. How would the proposed virtual to physical address mapping restriction affect the design and implementation of a VIPT cache?

Because VPN Index == PPN Index, we can use any of the lower 22 bits as cache index bits without introducing aliasing problems.

This allows us to have more cache sets than we would without the restriction.

Question 4.4 (5 points): Assume that the operating system uses demand paging. Please describe how this restriction affects the performance of demand paging.

Because the mapping of virtual to physical pages is not fully associative, demand paging is more likely to incur “conflict misses”. Specifically, we could run out of free physical pages that map to a specific virtual address well before we actually run out of physical memory. This would cause additional swapping (poor performance) even though physical memory is not fully exhausted.

Name _____

Part B: Out-of-Order Processors (25 Points)

Question 1 (20 points)

This question uses the out-of-order machine described in the Quiz 1 Handout. We describe events that affect the initial state shown in the handout. Label each event with one of the actions listed in the handout. If you pick a label with a blank (_____), you also have to fill in the blank using the choices (i—vii) listed below. If you pick "R. Illegal action", state why it is an illegal action. If in doubt, state your assumptions.

Example: I6 executes and writes its result to P3. P3 is set as present in I7's and I10's ROB entries.

Answer: B, i

1. I5 commits and returns P0 to the free list.

   Q

2. Assume that all instructions up I7 commit. I7 executes and finds that P5 is equal to zero, triggering an exception.

   N, M, L(i)

3. I8 executes and finds that P1 is not equal to P9, triggering a taken branch that was predicted as taken.

   K(iii)

4. Assume that the rename stage attempts to insert two more instructions into the ROB. The first, I11, successfully enqueues and uses P10 as its value of PRd. The second, I12, can no longer find any free physical registers.

   A(viii)

5. I5 commits and now sets P1 as present in the physical register file.

   R, P1 is set as present on execute not commit

6. Assume I8 executes and finds that P1 *is* equal to P9, resulting in an untaken branch that was predicted as taken.

   L(iii), M

7. Assume I7 triggers a divide-by-zero exception. I8 does not depend on I7, so it commits.

   R, this violates in-order commit

8. Assume xor instructions take 5 cycles. I10 is unable to begin executing due to P3 not yet being present.

   A(i)

9. Assume that all instructions up to and including I7 have committed. I8 finishes execution and commits. It finds that P1 is not equal to P9. It updates the global branch history register.

   Q, J(iii)

10. Assume that I5 and I6 have committed but I7 is still in-flight. I8 commits and returns P9 to the free list.

    R, this violates in-order commit

Name _____

Question 2 (5 points)

When designing this processor, we first simulate it with an *infinite-sized ROB*. We find that across a set of benchmarks, it achieves an average IPC of 2. We also know that instructions take 10 cycles on average from entering the ROB to committing. When implementing the processor (no infinite-sized ROB anymore!), what is the minimum size you would suggest making the ROB?

Little's Law: T = N / L

2 = N / 10

N = 20

So the ROB should be ≥ 20 slots

Name _____

Part C: EDSACjr (35 Points)

For this part, you should consult the EDSACjr handout.

Question 1 (5 points)

Briefly, explain the architectural limitation that forces EDSACjr programs to depend on self-modifying code.

EDSACjr only has a single accumulator register and no index register, so it only supports absolute addressing. Addresses are stored as immediates in instructions. Therefore, for a single instruction to access multiple addresses, it must be modified.

Name _____

Question 2 (10 points)

The following program computes the dot product of a vector with itself. Please write EDSACjr assembly to implement it:

```
int total = 0;
for (int i = 0; i < n; i++) {
     total += A[i] * A[i];
}
```

You should assume the following memory layout and may add temps as needed:

Data Memory

A	A[0]
	A[1]
	…
	A[n-1]
TOTAL	total
N	n
I	0
ONE	1
TOTAL	0

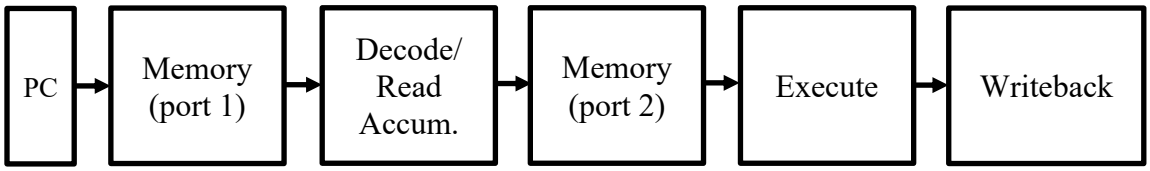
Program

```
loop:   LD I
        SUB N
        BGE done


ld_inst:    LD    A
mul_inst:   MUL   A
            ADD   TOTAL
            ST    TOTAL

            LD    ld_inst
            ADD   ONE
            ST    ld_inst

            LD    mul_inst
            ADD   ONE
            ST    mul_inst



        LD I
        ADD ONE
        ST I
        BGE loop
done:   END
```

Name _____

Question 3 (6 points)

Suppose we implement the EDSACjr ISA on a 5-stage pipeline shown below:



Note that both instructions and data live in a single unified memory that has *two ports*. The memories are both single-cycle SRAMs and the processor is implemented on a modern technology node, not vacuum tubes.

For *just this question*, assume no hazards and single-cycle memory accesses. Is pipelining this processor a good idea? How does pipelining this processor affect each of the following: CPI, frequency, number of instructions, and performance?

Iron Law of performance:

(instructions/program) * (clock cycles/instruction) * (time / clock cycles)

instructions/program: unchanged
clock cycles/instruction: unchanged (still 1 because no hazards)
time/clock cycle: down because shorter critical path

so, better performance

Name _____

Question 4 (7 points)

One potential problem with pipelining the EDSACjr is that working on self-modifying code in a single unified memory introduces *additional hazards*. Specifically, instructions further along in the pipeline (earlier in program order) may modify instructions earlier in the pipeline (later in program order). Appropriately handling this hazard is essential to a correct pipelined EDSACjr implementation.

Please assume that there are *no other* data or control hazards.

In this first question, we will try to solve this hazard *by stalling*. Please describe the following:

- How you would modify the hardware to handle this hazard by stalling. What structures would you add? What information needs to be stored in them?
- How would stall signals be generated?
- Where would they be consumed?

To make your answer concrete, please provide a sequence of up to 3 instructions that would cause a stall and explain how stall signals are generated *for that sequence.*

- Add a register pipeline that contains the addresses of in-flight stores. When a store is decoded in Decode, write its address into the pipeline. In fetch, if PC == any address in the pipeline, stall the fetch stage and place a NOP into fetch's output pipeline register.
- Instruction sequence:

```
            ST inst_addr
inst_addr: ADD 37
```

For this sequence, when the ST is in decode, it generates a stall signal. The write to the fetch pipeline register is gated on this stall signal. Additionally, the PC update is also stalled by this signal.

Name _____

Question 5 (7 points)

Now, instead of handling this self-modifying code hazard by stalling, we will handle it by *bypassing*. Please describe the following:

- How you would modify the hardware to handle this hazard by bypassing.
- What bypass paths need to be added?
- Can bypassing fully avoid stalls? If yes, please justify. If not, please provide an example instruction sequence that would still result in some pipeline bubbles.

Like in the previous part, please provide a sequence of up to 3 instructions to illustrate your thinking.

- Instead, we can bypass instruction stores from Decode to Fetch.
- If the instruction being decoded is a ST whose address field matches the PC in Fetch, then the pipeline register at the end of Fetch is set to the Accumulator
- This will fully avoid stalls:

```
            ST inst_addr
inst_addr: ADD 37
```

When the ST is in decode, the ADD is in fetch. However, the pipeline register at the output of Fetch is just set to the accumulator. In this way, bypassing is effectively able to fully remove this stall.

Name _____

Scratch Space