

# Computer System Architecture

## 6.823 Quiz #1

March 19th, 2021

Name: \_\_\_\_\_

90 Minutes  
17 Pages

Notes:

- Not all questions are equally hard. Look over the whole quiz and budget your time carefully.
- Please state any assumptions you make, and show your work.
- Please write your answers by hand, on paper or a tablet.
- Please email all 17 pages of questions with your answers, including this cover page. Alternatively, you may email scans (or photographs) of separate sheets of paper. Emails should be sent to [6823-staff@csail.mit.edu](mailto:6823-staff@csail.mit.edu)
- Please ensure your name is written on every page you turn in.
- Do not discuss a quiz's contents with students who have not yet taken the quiz.
- Please sign the following statement before starting the quiz. If you are emailing separate sheets of paper, copy the statement onto the first page and sign it.

I certify that I will start and finish the quiz on time, and that  
I will not give or receive unauthorized help on this quiz.

Sign here: \_\_\_\_\_

Part A	_____	20 Points
Part B	_____	40 Points
Part C	_____	40 Points

**TOTAL** \_\_\_\_\_ **100 Points**

## Part A: Caches (20 Points)

Ben Bitdiddle wants to run the following code on his machine:

```
uint32_t A[32], B[32];
uint32_t d;
...

for (uint32_t rounds = 0; rounds < 10; rounds++) {
    for (uint32_t i = 0; i < 32; i++) {
        B[i] = d*A[i];
    }
}
```

His machine has a direct-mapped data cache that has 16 lines with 16 bytes per line. Note that arrays A and B have 4-byte elements. Array A starts at address 0x0000 and B starts at address 0x0100. The values of i, d, and sum are stored in registers.

### *Question 1 (3 points)*

How many compulsory, capacity, and conflict misses will occur when running the above code? (Focus on data accesses only, not on instruction accesses.)

### *Question 2 (5 points)*

Ben changes the configuration of the cache to be 2-way set-associative with 8 sets and 16 bytes per line. The cache uses a Least Recently Used (LRU) replacement policy. With the new cache, how many total misses will occur when running the above code?

**Question 3 (7 points)**

Alyssa P. Hacker comes by and changes the code such that elements of array C are added to the final result:

```
uint32_t A[32], B[32], C[32];
uint32_t d;
...

for (uint32_t rounds = 0; rounds < 10; rounds++) {
    for (uint32_t i = 0; i < 32; i++) {
        B[i] = d*A[i] + C[i];
    }
}
```

Assume that array C starts at address 0x0200.

(a) (3 points) With Ben's 2-way cache configuration from Question 2, how many total misses will occur when running this code?

(b) (4 points) Can a different replacement policy further reduce the number of misses? If so, briefly describe such a policy (do not assume knowledge of the future, i.e., Belady's MIN is not an option). If not, briefly state your reasoning.

***Question 4 (5 points)***

We notice that a program we want to run is mostly accessing consecutive memory locations in sequence. A way to reduce the effect of misses for such an access pattern is to *prefetch* the data before it is actually needed by the processor. For example, given an access stream of  $A, A+1, A+2, \dots$  where  $A$  is the location in memory initially accessed by the program, we can prefetch the data at addresses  $A+3, A+4, \dots$  ahead of the processor, so that they are already in the cache by the time the processor explicitly requires them.

Assume that our processor is issuing a memory request every 4 cycles on average, and the average memory access latency is 100 cycles. Also assume that the prefetcher predicts future accesses perfectly. How many prefetch requests must be in flight on average to prefetch the data required by the processor on time?

## Part B: Caches and Virtual Memory (40 Points)

### *Question 1 (10 points)*

Answer whether the following statements are **True** or **False** (2 points each).

(a) Translation Lookaside Buffers (TLBs) usually have low associativity to enable efficient virtual to physical address translations.

(b) In a multilevel cache hierarchy, using exclusive caches achieves a higher effective capacity than using inclusive caches of same size.

(c) You can only swap out pages containing page table entries that reference physical pages residing in secondary storage.

(d) Doubling the number of cache lines of a direct-mapped cache (while leaving all other parameters unchanged) always decreases conflict misses.

(e) A larger cache, given fixed associativity and block size, will always reduce the average memory access time (AMAT).

**Question 2 (8 points)**

Consider a machine with byte-addressable memory, 24-bit virtual addresses, and 24-bit physical addresses. We want to compare two different page table designs for this machine.

The machine executes the following piece of code:

```
int A[262144]; // A 1MB array
int sum;

while (1) {
    for (int i = 0; i < 256; i++) {
        sum += A[i*1024];
    }
}
```

Array A starts at virtual address `0x800000`. Ignore accesses to the stack and code regions and only focus on accesses to the 1MB array.

Alyssa P. Hacker first wants to use a linear (i.e., single-level) page table with 4KB pages. Each page table entry is 32 bits. Assume there is no TLB in this machine.

(a) (2 points) Derive the number of bits needed for the physical page number and page offset.

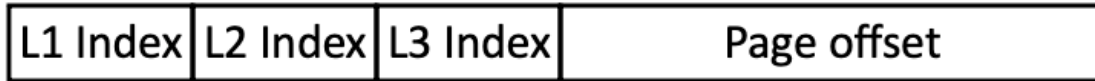
(b) (2 points) What is the maximum size of the page table?

(c) (2 points) What is the size of the page table when executing the above code?

(d) (2 points) How many memory references to the page table are required to execute one iteration of the while loop?

**Question 3 (7 points)**

Alyssa now switches to a 3-level hierarchical page table. The virtual address is divided in the following way:



Here, the L1, L2, and L3 indices evenly divide the bits that exclude the page offset (i.e., the three fields are the same size). Each entry of the L1, L2, and L3 page tables is 32 bits.

(a) (2 points) What is the maximum size of the page table?

(b) (3 points) What is the size of the page table when executing the above code?

(c) (2 points) How many memory references to the page table are required to execute one iteration of the while loop?

***Question 4 (5 points)***

Alyssa wants to add a TLB to speed up address translation. What is the minimum number of entries necessary to have no TLB misses in steady state for the above code?

***Question 5 (5 points)***

Alyssa also decides to implement a virtually-indexed, physically-tagged cache. Her cache is 4-way set-associative with 512 lines and 32 bytes per line. Can her cache have aliasing issues? Briefly explain.



***Question 6 (5 points)***

Alyssa now wants to add support for large pages, i.e., pages that are larger than 4KB.

(a) (2 points) Given her 3-level hierarchical page table, what are the possible large page sizes while using the same page table design?

(b) (3 points) Using large pages, what is the **minimum** number of TLB entries needed to eliminate all misses from the TLB?

## Part C. Pipelining (40 Points)

Ben Bitdiddle wants to run the following C code on his 5-stage pipelined MIPS processor:

```
int A[100];
int B[100];
int difference = 0;
int matches = 0;
...
for (int i = 100; i != 0; i--) {
    int a = A[i];
    int b = B[i];
    if (a == b)
        matches++;
    else
        difference += (a - b);
}
...
```

The equivalent MIPS assembly code segment is shown below. We assume the following at the start of the code segment:

- Registers r1, r2, r3, and r4 hold the values of a, b, matches, and difference, respectively.
- Register r5 holds the value of loop iteration index.
- Registers r6 and r7 hold the base address of arrays A and B, respectively.
- Register r8 is used for intermediate results.

```
...
    ADDI r5, r0, 100
_loop: LW   r1, 0(r6)      ; a = A[i]
        LW   r2, 0(r7)      ; b = B[i]
        ADDI r6, r6, 4
        ADDI r7, r7, 4
        SUB  r8, r1, r2
        BNEZ r8, _else     ; Jump to _else label if a != b
        ADDI r3, r3, 1     ; matches++;
        J    _then
_else:  ADD  r4, r4, r8     ; difference += (a - b);
_then:  ADDI r5, r5, -1    ; i--;
        BNEZ r5, _loop    ; Check loop bound
...
```

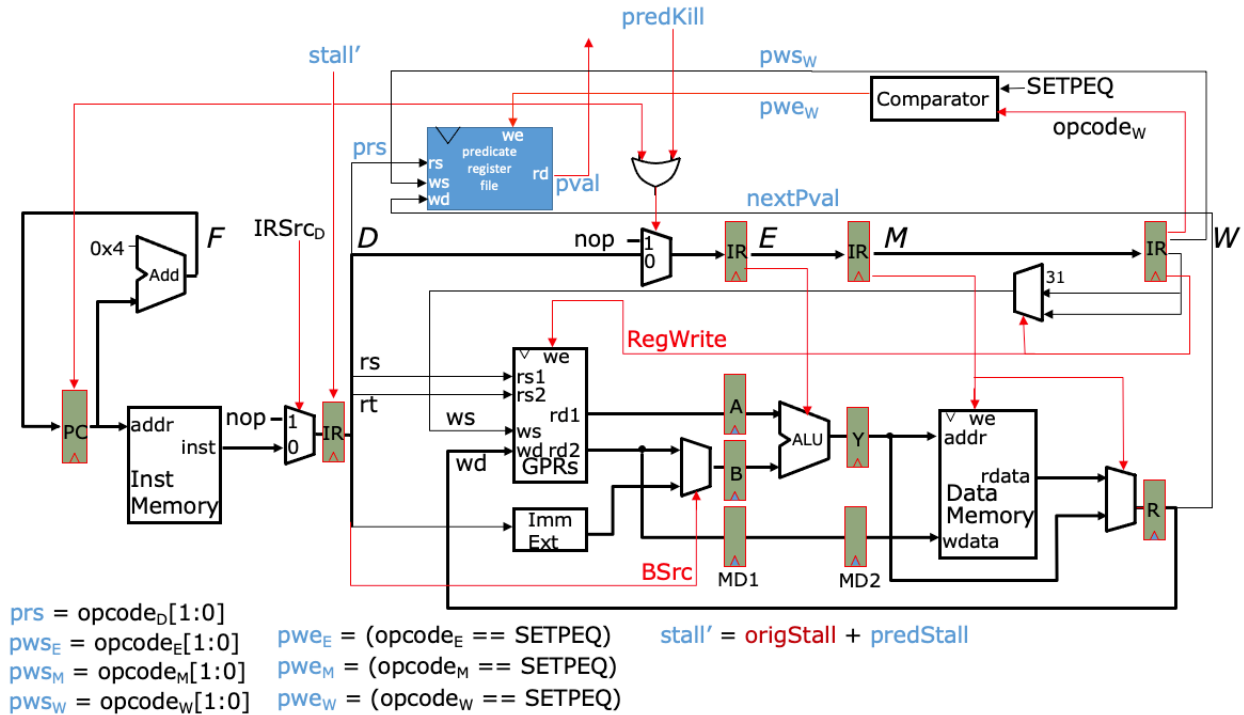
Name \_\_\_\_\_

Ben notices that his simple 5-stage pipelined processor suffers from frequent control hazards on branches. To eliminate branches, Ben decides to implement predicated execution in his MIPS processor. Refer to the accompanying handout for more details on predicated instructions.

***Question 1 (5 points)***

Rewrite the code denoted by the **red box** using predicated instructions instead of conditional branches and jumps. Your code should not have any branches or jumps remaining.

To implement predicated instructions, Ben adds a *predicate register file* to the Decode stage of his pipeline, as shown below (for simplicity, this pipeline diagram does not handle control instructions). New control signals and datapaths are highlighted in blue.



Every predicated instruction and SETPEQ has its source/destination predicate register address encoded in the bottom two bits of its opcode. When a predicated instruction enters the decode stage, it uses  $prs$  to select the predicate register to read. The register file outputs  $pval$ , which is the corresponding predicate register value. Predicated instructions must be treated as no-ops if the predicate register value is 0. To do this, Ben implements a new kill condition  $predKill$ .

The SETPEQ instruction sets the destination predicate register  $pws_W$  and the predicate value  $nextPval$  in the writeback stage. Thus, predicated instructions must stall in the decode stage if the pipeline has a SETPEQ instruction in flight that will write to the same predicate register. This requires Ben to implement a new stall signal  $stall'$ , which adds an additional stall condition  $predStall$  on top of the original stall signal  $origStall$ .

Finally, Ben implements a new control signal  $pwe_W$  that enables writes to the predicate register when SETPEQ reaches the writeback stage.

**Question 2 (6 points)**

Derive the Boolean expression for new control signals *predStall* and *predKill*. You are also provided the following two control signals.

*PredIns* = True if instruction in decode is predicated and reads the predicate register normally (i.e., the instruction is executed normally if predicate register is True / 1).

*InvPredIns* = True if instruction in decode is predicated on the *inverse* of the predicate register. (i.e., the instruction is executed normally if predicate register is False / 0).

*predStall* =

*predKill* =

***Question 3 (4 points)***

Ben now adds **full bypassing** for register values. Ben runs the original code with branches and jumps on his new processor. How many cycles does it take to execute the code in the **red box**? Assume branches are resolved in the Execute stage and jumps are resolved in the Decode stage, and there are no branch delay slots. You only need to consider one iteration of the loop.

***Question 4 (4 points)***

Now Ben runs the predicated code on his new processor with full bypassing for register values. How many cycles does it take to execute the code in the **red box**? You only need to consider one iteration of the loop.

***Question 5 (4 points)***

Ben adds full bypassing for predicate register values as well. How many cycles does it take to execute the code in the **red box**? You only need to consider one iteration of the loop.

***Question 6 (4 points)***

Ben wants to write to the predicate register file in the Execute stage instead of waiting until Writeback. This would allow removing most of the bypass paths for predicate registers. Can this result in incorrect behavior? Justify your reasoning.

**Question 7 (4 points)**

Alyssa notes that predicated instructions need not read the predicate register in the decode stage. As long as the instruction is not changing any architectural state, the predicated instruction can flow normally through the pipeline until it is necessary to read the predicate register. Given this information, where is the latest stage Ben can put the predicate register file? Take exceptions into account.

**Question 8 (4 points)**

In Question 2, we defined the new stall signal as  $stall' = origStall + predStall$ . If you want to minimize the number of stalls, is this the best we can do, or could you change the stall signal to avoid some stalls? Briefly explain why or why not. (You do not need to write the precise stall signal.)

Based on your answer to the question above, if we wanted to maximize performance and, for equal performance, minimize complexity, which would you choose: (1) keeping the predicate register file in Decode, with bypasses; or (2) placing the predicate register file at a later stage as in Question 7, to reduce/avoid bypasses?



***Question 9 (5 points)***

Ben recompiles Linux to target his new ISA and runs it on his new MIPS processor. To his dismay, it crashes immediately. He notices that the OS kernel does not save/restore predicate registers on context switches. Help Ben fix the kernel by writing code that saves the predicate registers upon a context switch. You only need to write code to store one predicate register, say `p1`, into one general-purpose register, say `r1`.