Computer Architecture **Name:** _____
Fall 2019
Final Exam
12/13/2019
Time Limit: 3 hours Computing ID _____

Instructions:

1. This exam contains 22 pages (including this cover page) and 26 questions.
2. You have **three (3) hours** to complete the examination. As a courtesy to your classmates, we ask that you not leave during the last fifteen minutes.
3. Write your answers in this booklet. We scan this into TPEGS, so **please try to avoid writing on the backs of pages**.
4. You may use a calculator. You may not share a calculator with anyone. If you didn't bring a calculator, you may use your phone, **but** you must put it on **flight mode** and clear all visible notifications **before** the examination starts, and you must not open any applications other than the calculator and a timer.
5. Please sign the below Honor Code statement.

I have neither given nor received aid on this exam. Signature: _____

1. For the following questions, consider the following C code:

```
int i = 0;
int j = 0;
do {
  if (i % 4 != 0) goto after;
    j += 1;
after:
    i++
} while (i != 1000000);
```

Assume this compiles to assembly with two conditional jumps: one for the `if` statement that jumps if the condition is true and one for the do-while loop which jumps if the condition is true.

(a) (3 points) Suppose this code is run with a branch direction predictor based on a table of two-bit saturating counters, where each of the two conditional branches use different table entries. Complete the following table with predictions and outcomes for the *`if` statement branch when* i is between 10001 and 10003:

i value	prediction	outcome
10001		taken
10002		not taken
10003		taken

Solution: taken taken taken
(aside: the given outcomes aren't actually right (since 10002 is not a multiple of 4, so they should be all taken), but we don't think this would affect anyone's answer)

(b) (3 points) Suppose this code is run with a branch direction predictor that:

- uses single global history register, which keeps track of whether the last K branches were taken or not taken;
- uses the value of that history register to lookup a counter that determines whether to predict the branch as taken or not taken
- updates the counter used for the prediction based on the actual outcome of the branch

1. If the history register had three entries and indicated a pattern of `TTTN`, then the best prediction for the above code would be?

   √ **Taken**

   ◯ Not-Taken

2. What is the minimum size K of the history register that will allow all branches in the above code to be predicted correctly after the loop runs for several hundred iterations (to give enough time to make sure the branch predictor's counters have appropriate values)?

Solution: (2 points) 7 (take off .5 points if 8; half-credit for 3 (local history))

2. (3 points) Consider a branch predictor which combines the address of each branch with a global branch history to lookup a two-bit saturating counter in a table, then uses this counter to predict branches. Suppose this predictor uses a hash function that takes the last bit of the PC and adds it to the number of taken branches in branch history register that has a length of 2. If the state of the predictor is as shown below, what global history would result in predictions that the branch is taken when the pc is 0xADD? Write your answer using T to represent taken and N for not taken. Assume that adds are preformed with wraparound.

0	11
1	00
2	10
3	01

Solution: TN or NT

3. (2 points) High quality branch predictors are useful because: (Mark all that apply)

   √ **Predicting branches correctly (as opposed to incorrectly) reduces the number of things that a processor need to fetch**

   √ **Predicting branches correctly (as opposed to incorrectly) results in faster programs**

   ◯ Predicting branches correctly (as opposed to incorrectly) reduces the number of instructions stored in main memory

   ◯ High quality branch predictors increase the amount of space available for computation on a chip.

4. (3 points) Consider a system with:

   - 32 bit virtual address
   - 4KB pages

   If the system will also have a 256KB cache with 64 byte blocks, what would the minimum associativity of the cache needed for it to be a virtually-indexed, physically-tagged cache? (That is, one where the cache is based on physical addresses, but the cache lookup and TLB lookup can overlap.)

Solution: 64 (256KB/4KB)

5. (2 points) Which of the following is/are true about entries placed into the TLB (translation lookaside buffer)? (Mark all that apply)

   √ **Only valid page table entries are placed into the TLB on TLB misses.**

   √ **The ISA will generally include an instruction that invalidates TLB entries**

   ◯ The processor will update a TLB entry whenever the corresponding page table entry is written to.

   ◯ The TLB helps translate physical addresses to virtual addresses

Solution: re: 3 — only if the OS explicit invalidates it

6. Consider a system with the following specifications:

   - TLB has a 1 cycle hit time and a 99% hit rate
   - The TLB miss penalty (page table lookup time) is 100 cycles
   - The L1 cache has a hit time of 4 cycles and a hit rate 80%
   - The miss penalty for L1 cache is 50 cycles

   (a) (3 points) Assuming TLB and L1 cache accesses do not overlap, what is the average memory access time for the system described above?

Solution: 16
(1 (TLB hit) + .01 * 100 (TLB miss penalty) + 4 (L1 hit) + 0.2 * 50 (L1 miss penalty))

   (b) (3 points) Suppose the TLB and L1 cache access can overlap, even though the cache is based on physical addresses, and the L1 set index lookup takes 2 cycles, and the rest of the L1 access (tag comparison and extracting the appropriate part of the cache block) takes 2 cycles. What is the average memory access time then?

Solution: 15 or 14.99
(removed TLB hit term from above; or if assuming TLB miss penalty overlaps with L1 set lookup: 2 (L1 set hit) + .01 * 99 (extra time for TLB miss after L1 set lookup finishes) + 2 (L1 rest of hit) + 0.2 * 50 (L1 miss penalty) = 14.99)

7. (8 points) Use the memory layout below to answer the next two questions. The layout below shows the ***partial*** contents of physical memory.

0x00-3	00 00 AA 00
0x04-7	44 55 66 77
0x08-B	00 DA 22 33
0x0C-F	00 11 00 CC
0x10-3	2A 11 22 33
0x14-7	3B 00 00 00
0x18-B	00 AD 21 30
0x1C-F	00 BA AD 31
0x20-3	D0 CA 23 32
0x24-7	00 AC 24 33
0x28-B	00 DC 25 34
0x2C-F	00 BC 26 35
0x30-3	00 00 27 36
0x34-7	00 00 00 00
0x38-B	E7 11 28 33
0x3C-F	00 11 29 33

Assume the following:

- the base table register contains the value 0x0C (which is a physical address that is in the middle of page)
- the page table are 16 bytes
- single level page tables
- physical address are 8 bits
- page table entries are 4 bytes.
- assume the system is running in user mode.
- the most significant 4 bits of first byte represent physical page number.
- the remain bits of the first byte are laid as follows [valid bit — kernel only mode bit— read bit — global bit] where the valid bit is most significant
- the remain bits are unused.

What byte is stored at address virtual address 0x13? If reading this address would cause an exception write "fault" in the answer box. If not enough information is provided write unknown.

Solution: 0x32 (okay if 0x is omitted)
(since PTs are 16 bytes, 4 entries, so 2 bit VPNs. 0x13 has VPN 1, is at 0x10 in table, PPN 2 and valid/readable/not kernel only; access 0x23 is 0x32.)
(count as 3 points, not 8; the 8 was an error)

8. (3 points) Holding all things the same but extending the example to 2-level page tables, where page tables at each level are still 16 bytes. What byte is stored at virtual address 0xA2? If reading this address would cause exception write "fault" in the answer box. If not enough information is provided write unknown.

Solution: fault
(0xA2 is VPN part 1 10, VPN part 2 10. First PTE (at 0x14) yields PPN 0x3, valid/read/global. Second PTE (at 0x38) yields PPN 0xE, but invalid.)

9. (2 points) Which of the following are motivations for switching to multiple level page tables from single level pages? Mark all that apply

- √ **Multiple level paging allows us to reduce the amount of space required to store a process's page table entries.**
- ◯ Multiple level paging allows for faster pagetable entry look ups
- √ **Multiple level paging allows for larger virtual address spaces**
- ◯ Multiple level page tables allow for faster memory reads and writes.

Solution: not faster b/c of more complex lookup

10. (3 points) Consider a system with:

- 6-level page tables
- 1KB page tables at each level
- 4 byte page table entries

How much page table space would be required to allow a process to access two distinct pages, where the virtual addresses only differ in their most significant bit?

Solution: 11KB (1 first-level table, 2 tables at each other level)

11. (2 points) Which of the following conditions would cause the OS to invalidate one or more TLB entries? (Select all that apply.)

- √ **A process' page is evicted from DRAM**
- ◯ A process loads one of its page into an empty DRAM
- √ **The operating system triggers a context switch**
- √ **The program executes a system call that does require a context switch.**

Solution:

12. (2 points) What is/are the advantages of SIMD instructions (compared to non-SIMD instructions)? Mark all that apply

    ◯ They allow the CPU to execute multiple instructions at once

    √ **They allow data to be processed in parallel without adding much extra control logic to a CPU**

    ◯ They allow a CPU to add parallelism to existing programs

    √ **They allow a CPU to take advantage of larger and/or multiple ALUs**

    ◯ They allow a CPU to take advantage of a larger number of registers

Solution: subtract half a point per disagreeing answer
(not 1 b/c it's one instruction that does more things and just pipelining or normal out-of-order techniques with normal instructions was sufficient for this, so using SIMD didn't actually help here)
(not 5 b/c it's about larger registers, not a larger number of them)

13. (3 points) Consider a vector processor built from 3 execution units which can process *one vector element at a time*:

    - a load unit
    - a alu unit
    - a store unit

    Assuming that each unit takes one cycle per operation, but multiple units can be used within the same cycle. How many cycles does it take to execute the following instructions (VR1 represents a vector register and A[0:7] represents elements 0 through 7 inclusive of an array)? (Ignore the time require to fetch/decode/etc. the instructions.)

```
LOAD VR1 <- A[0:7]
ADD VR1 <- VR1, 2
STORE A[0:7] <- VR1
```

Solution: ~~9 (half-credit for off-by-one)~~
10 (8 for the last element to be loaded + add it + store it) — we graded this question with the wrong key; we intend to go through exams and give full credit for students who answered ten sometime during the night of Tuesday 17 December 2019

14. (3 points) Suppose instead we had a processor which had execution units that can process a whole 8-element vector at a time, but where the ALU unit has a latency of 3 cycles? (The load and store units still have a latency of 1 cycle.)

Solution: 5

15. Consider the following C loops where `A`, `B`, and `C` are large arrays of integers that do not overlap and `N` is a large, constant number.

```
/* loop A: */
for (int i = 0; i < N; ++i)
  A[i] = A[i*2]

 /* loop B: */
for (int i = 0; i < N; ++i)
  A[i] = (B[i] + C[i]) * (B[i] - C[i]);

 /* loop C: */
for (int i = 0; i < N; ++i)
  if (B[i] > 10)
    A[i] += B[i];

/* loop D: */
for (int i = 0; i < N; ++i)
  A[i+1] = (A[i] + A[i+1]) / 2;
```

(a) (2 points) Which of these loops would be most straightforward/efficient to implement using SIMD instructions, assuming there is no support for masking or scatter/gather loads and stores? (Write the letter in the box below.)

Solution: B

(b) (2 points) Some processors with SIMD instructions support *masked store* instructions, which act like a vector store instruction, but only store elements specified by 'mask' array. Which of the loops above would benefit from these instructions? (Write the letters for corresponding loops in the box below, separate letters by commas)

Solution: C (-.5 points for each disagreeing answer)

16. (2 points) Some exceptions are triggered deliberately by an application. What is likely to be true about these exceptions? (Select all that apply.)

- ◯ The instruction triggering the exception contains the address of the exception handler.
- ◯ If the application was running in user mode, the exception handler will also run in user mode.
- √ **The processor will save the value of the program counter before running the exception handler.**
- √ **The values of the application's registers can be read by the exception handler.**

17. (2 points) When an operating system performs a context switch from application A to application B; where is the value of the register `%rax` in application A will most likely be saved?

- √ **in the operating system's memory**
- ◯ on application A's stack
- ◯ on application B's stack
- ◯ in application A's page table
- ◯ in the exception table

18. (2 points) An operating system can use page tables and exceptions to perform allocation on demand, where it won't allocate physical memory for a program until the program tries to use it. In an operating system that implements this feature, what happens when a program attempts to access memory that still needs to be allocated?

- ◯ The hardware will notice that the memory is marked as kernel-mode-only in the page table and consequently update the page table and allocate it
- √ **The operating system's page fault handler will allocate the memory, update the page table accordingly, and restart the program starting with the instruction that tried to access the memory**
- ◯ The operating system's page fault handler will allocate the memory, update the page table accordingly, and restart the program from the beginning of the function that tried to access the memory
- ◯ The operating system's page fault handler will allocate the memory, update the page table accordingly, and restart the program from the instruction after the instruction that tried to access the memory
- ◯ the compiler will have inserted a system call before the instruction that attempts to access memory, and this system call will handle allocating the memory before the program actually attempts to use the memory

19. (2 points) When running an infinite loop on a shared server, it will usually be the case that the operating system will still occasionally switch to other programs, even if the server only has one core. How can the operating system do this? (Select all that apply.)

    √ **before starting the infinite loop program, the operating system can setup a timer for the hardware to run operating system code that switches away to another program**

    √ **the hardware will run operating system code when certain I/O devices like keyboard signal the processor, even if they aren't being used the currently active program**

    ◯ the hardware periodically swaps pages from DRAM to memory, so the infinite loop will eventually experience a page fault, even if the operating system is not otherwise run

    ◯ assuming an out-of-order processor, the program will eventually run out of free physical registers, so the hardware will run the operating system to handle this situation

20. Consider a system with:

    - 64-bit physical addresses and caches that use physical addresses
    - an 8-way 64KB L1 cache with a random replacement policy, a write-through, write-no-allocate policy, 64 byte blocks, and a 4 cycle hit time
    - a 4-way 2MB L2 cache with an LRU replacement policy, a write-back, write-allocate policy, 64 byte blocks, and a 16 cycle hit time
    - the miss penalty of the L2 cache is 50 cycles

    (a) (3 points) If the L1 cache has a 90% hit rate and the L2 cache has a 80% hit rate, what is the average memory access time of the L1 cache? (Assume each cache needs their entire hit time to determine whether there is a hit or miss.)

    Solution: 6.6 (4 + 0.1 * (16 + 0.2 * 50))

    (b) (2 points) Give an example of an address whose value would be stored in the same set as address `0x123456`, but not as part of the same cache block in the L1 cache.

Solution: Anything with last set bits match are the same as 0x12 34 56 and different tag

so almost anything with 34 in the middle and a different beginning (plus some variations)

[aside: originally posted version of the key did not specify the part of the tag, but we should have taken that into account when grading]

(c) (2 points) Throughout the entire cache, how much space in the L1 cache is devoted to storing tags and valid bits?

◯ less than 4KB
√ **more than 4KB and not greater than 16KB**
◯ more than 16KB and not greater than 128KB
◯ more than 128KB and not greater than 256KB
◯ none the above

Solution: (64KB/64B per block = 1024 blocks times about 6 to 7 bytes/block for tag)

(d) (2 points) Which of the caches must have a dirty bit to store information about modified blocks?

◯ L1
√ **L2**

Solution: write-back policy; 1 point per disagreeing

21. (2 points) Consider the task of taking an 32-bit unsigned integer `x` and constructing a 16-bit unsigned integer from its most significant byte and least significant byte. For example, given an `x` of `0x12345678`, producing a result of `0x1278`. Which of the following C snippets would do this? (Select all that apply.)

◯ `(x & 0xFF) | ((x >> 16) & 0xFF)`
◯ `(x & 0xFF) | ((x >> 24) & 0xFF)`
◯ `(x / 0x10000) & 0xFF00 + x % 256`
◯ `((x & 0xFF000000) << 8) | ((x & 0xFF) >> 8)`

Solution: `(x / 0x10000) & 0xFF00 + x % 256` parses as `(x / 0x10000) & (0xFF00 + x % 256)` and so is not actually correct. We will give everyone credit for this option (outside of TPEGS) because of this confusion.

22. (2 points) Which of the following are more typical of a RISC-like (reduced instruction set computer) when compared to the CISC-like design philosophy? (Select all that apply.)

◯ Making sure more commonly used instructions have shorter encodings.
◯ Allowing all instructions to access memory rather than varying what kinds of operands different instructions can have.
√ **Using a fixed-length encoding for instructions.**
√ **Making each instruction have at most one output register.**

23. Consider the following assembly snippet:

```
subq %rbx, %rcx          F D E*M W
mrmovq 8(%rcx), %rdx       F D*E M*W
addq %rax, %rdx              F x D*E M*W
xorq %rcx, %r8                   F D E|M W
rmmovq %rcx, 8(%rdx)               F D*E M W
```

(a) (2 points) Consider the assembly snippet executed on the single-cycle processor we discussed in lecture. During the cycle in which the `mrmovq` instruction is fetched, the address input to the data memory is equal to

- ◯ one of the register file outputs
- ◯ part of the output of the instruction memory
- ◯ the output of the program counter
- √ **the output of the ALU**
- ◯ the register number for `%rdx`
- ◯ the register number for `%rcx`
- ◯ the value 8
- ◯ irrelevant; the data memory's address input does not matter for this instruction

(b) (2 points) Consider the assembly snippet executed on the single-cycle processor we discussed in lecture. After the `mrmovq` instruction is fetched, the value of `%rdx` changes

- √ **at the following rising edge of the clock signal**
- ◯ sometime before the following rising edge of the clock signal (depending on how fast components of the processor are)
- ◯ after two rising edges of the clock signal
- ◯ after the `addq` instruction is fetched

(c) (3 points) Consider the assembly snippet executed on the 5 stage pipelined processor we discussed in lecture with forwarding and branch prediction. If the `subq` instruction is fetched during cycle 0, during what cycle is the writeback stage of `rmmovq` performed?

Solution: 9 (1-cycle stall for hazard between mrmovq and addq; subq finishes during cycle 4, so 4 cycles + 1 stall later = 9)

(d) (2 points) Consider the assembly snippet executed on the pipelined processor we discussed in lecture with forwarding and branch prediction. Which of the following forwarding operations would be useful? (Mark all that apply.)

- √ **forwarding `%rcx` from `subq` to `mrmovq`**
- ◯ forwarding `%rcx` from `subq` to `xorq`
- ◯ forwarding `%rcx` from `subq` to `rmmovq`
- ◯ forwarding `%rdx` from `mrmovq` to `rmmovq`
- √ **forwarding `%rdx` from `mrmovq` to `addq`**
- √ **forwarding `%rdx` from `addq` to `rmmovq`**

Solution: (subtract .5 per disagreement to minimum of 0)

(e) (2 points) Consider executing the above assembly snippet on an out-of-order processor. Assuming the processor has sufficiently many functional units (also known as execution units) and capacities in instruction queues and other buffers, while the processor is fetching a value from memory for `mrmovq`, it could also: (Mark all that apply)

- ◯ perform the subtraction for the `subq`
- ◯ perform the addition for the `addq`
- √ **perform the xor computation for the `xorq`**
- ◯ perform the store for the `rmmovq`

24. Suppose an eight-stage pipelined processor uses branch prediction and forwarding to resolve hazards. When a branch is fetched during cycle X, the processor identifies the actual target of the branch and whether or not it is taken near the end of cycle $X + 3$.

When running a benchmark program on this processor, 10% of the instructions are branches that require prediction and they are predicted correctly 80% of the time. In the benchmark program, there are no other causes of squashing or stalling.

(a) (3 points) What is the average cycles per instruction of this processor on the benchmark program?

Solution: 1.06 (1 cycle + 10% * 20% * 3 extra cycles); half-credit for 1.24 (used prediction rate instead of misprediction rate)
since this is a pipelined processor and multiple instructions execute in the same cycle, the overall number of stages doesn't really affect the long-run cycles-per-instruction

(b) (3 points) Suppose this processor did not implement branch prediction. What would its average cycles per instruction be on the benchmark program?

Solution: 1.3 (1 cycle + 10% * 3 extra cycles)

25. Use the code snippet below to answer the following questions:

```
  irmovq $2, %rsi
  irmovq $1, %rdi
foo:
  subq %rdi, %rsi
  jle foo
  halt
```

(a) (3 points) what is the final value of `%rsi` when the above assembly snippet is executed?

Solution: 1

(b) (3 points) Suppose the above assembly snippet is executed on the five-stage pipelined processor we discussed in lecture with branch prediction that predicts branches as always taken as well as forwarding. If the first `irmovq` instruction is fetched during cycle 0 then the halt instruction will run its writeback stage during what cycle?

Solution: 10

(c) (3 points) By how many cycles (if any) would performance from the previous question improve if the branch predictor never mispredicted?

Solution: 2

26. Reference the code below to answer the answer the following questions:

```
int foo(int *a, int *b, int *c, int n) {
  for (int i = 0; i < n; ++i) {
    c[i] = 0;
    for (int j = 0; j < n; ++j) {
      c[i] += a[i] * b[j];
    }
  }
}
```

(a) (2 points) Which of the following transformations would the compiler not be able to perform without adding additional checks that pointers do not point to overlapping memory regions? (Select all that apply)

- ◯ unrolling the loop with index `j`
- √ **using multiple accumulators to compute each `c[i]`**
- √ **reordering the `i` and `j` loops**
- √ **computing each `c[i]` in a register and only storing it after the loop with index `j`**

(b) (2 points) Suppose we try using loop unrolling and multiple accumulators for the computation of `c[i]` above, but determine that it does not improve performance. Both before and after attempting to use multiple accumulators, the inner loop has a throughput of about 1 cycle per iteration. Which of the following are plausible reasons for this? (Mark all that apply.)

- ◯ The processor's multipliers have a latency that is more than twice the latency of additions.
- √ **The processor can only start one multiply per cycle.**
- ◯ Although the processor can start multiple adds per cycle, the latency of the adders is too high
- √ **The processor can only load two values per cycle.**