Name: _____

Write your name and computing ID above. Write your computing ID at the top of each page in case pages get separated. Sign the honor pledge below.

Generally, we will not be answer questions about the exam during the exam time. If you think a question is unclear and requires additional information to answer, please explain how in your answer. For multiple choice questions, write a * next to the relevant option(s) along with your explanation.

On my honor as a student I have neither given nor received aid on this exam.

_____

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

```
for (int i = 0; i < 1000; i += 1) {
    int sum = B[i];
    for (int j = 0; j < 3; j += 1) {
        sum += A[i+j];
    }
    B[i] = sum;
}
```

(a) (4 points) Suppose the code above is adapted to use vector instructions. Which of the following are plausible ways the code's performance could be improved with vector instructions? **Select all that apply.**

- ☐ one instruction could perform multiple `sum += A[i+j]` operations for a single `i` value but multiple `j` values
- ■ one instruction could perform multiple `sum += A[i+j]` operations for a single `j` value but multiple `i` values
- ■ one instruction could store several values to the array `B` at once
- ■ one instruction could load several values from the array `A` at once

(b) The inner loop (the one with the index `j`) can be unrolled and transformed to use multiple accumulators, making the above code look like:

```
for (int i = 0; i < 1000; i += 1) {
    int sum1 = B[i];
    sum1 += A[i];           /* line X */
    int sum2 = A[i+1];
    sum2 += A[i+2];         /* line Y */
    B[i] = sum1 + sum2;     /* line Z */
}
```

i. (4 points) On an out-of-order processor, the original code cannot perform the additions to `B[i]` for a single `i` in parallel. After this optimization, some additions can be performed in parallel. Identify which ones. (You may use the "line X" through "line Z" comments to specify which ones.)

line X and line Y

ii. (4 points) Suppose it is found that this attempt at optimization does **not** increase performance on an out-of-order processor that can perform many additions in parallel. Which of the following are plausible reasons that this could happen? **Select all that apply.**

- ■ additions from multiple iterations of the loop over `i` were already being performed in parallel, so the processor was already doing as many additions per cycle as it could
- ☐ the transformed code performs more addition operations than the original code, which outweighed the benefits of having more additions to perform in parallel
- ☐ unrolling the loop over j increased the number of comparisons needed due to the small number of iterations
- ■ the processor could only perform one load per cycle even though it could perform more additions per cycle, so the adders ended up being idle most of the time

2. Suppose a single-core system is running two processes A and B.

   Process A is running the following assembly function:

```
addValues:
    movq $0x10000, %rax     /* instruction A */
    movq $0x20000, %rbx     /* instruction B */
    movq (%rax), %r8        /* instruction C */
    movq (%rbx), %r9        /* instruction D */
    addq %r8, %r9           /* instruction E */
    movq %r9, (%rbx)        /* instruction F */
    ret                     /* instruction G */
```

(a) (3 points) Suppose when this snippet is running, memory at address `0x20000` is inaccessible (that is, the corresponding virtual page is marked as invalid in process A's page table), so process A crashes. An exception will occur that triggers this crash as a result of the processor's attempt to execute instruction D (fill in one of the letters A through F marking instructions above).

(b) (4 points) Suppose the operating system set a timer before process A started running the snippet above. The timer triggers an exception which causes the operating system to run after instructions E and before instruction F above. The operating system responds to this exception by letting process B run for a while, then returns to running process A.

Which of the following are true about this scenario? **Select all that apply.**

- ☐ the location to which instruction F stores a value depends on what process B does with %rbx
- ☐ when the operating system resumes running process A, it will run instruction A again
- ☐ process B will run primarily in kernel mode because its execution occurred in the middle of process A's execution
- ■ when process B is running, the page table base register is likely to have a different value than when process A was running

3. Suppose a system has:
   - 65536 *was written 16384 on exam in error* byte (2^{16} byte) pages (and therefore 16-bit page offsets)
   - 2-level page tables with 8-byte page table entries and 8192 *was written 2048 on exam in error* (2^{13}) page table entries for page tables at each level
   - page table entries that contain
     - valid (also known as a present), user-mode accessible, writeable, and executable bits,
     - 20 unused bits, and
     - a 40-bit physical page number

   error above: 16384 should have been 65536; 2048 should have been 8192

(a) (3 points) What is the size of the virtual addresses on this system? You may leave your answer as an unsimplified arithmetic expression.

Solution: $16 + 13 \cdot 2 = 42$ (2^{16}) OR variations due to typos above, e.g. $13 + 11 \cdot 2 = 35$ OR $16 + 11 \cdot 2 = 38$

(b) While looking up the virtual address `0x123451234`, the processor finds a first-level page table entry which contains:
   - a valid bit, user-mode accessible bit, writeable bit, and executable bit all set to 1
   - physical page number `0x1000`

   and then finds a second-level page table entry which contains:
   - a valid bit, user-mode accessible bit, writeable bit all set to 1
   - an executable bit set to 0
   - physical page number `0x2000`

   i. (4 points) The **second-level** page table entry was retrieved from what physical address? You may leave your answer as an unsimplified arithmetic expression. If not enough information is supplied write "unknown" and explain briefly.

   Solution: $0x1000\ 0000 + 0x345 \cdot 8 = 0x1000\ 1a28$ OR variations due to typos above, e.g. $0x1000 \cdot 16384 + 0x345 \cdot 8$

   ii. (4 points) If the program is writing a value to virtual address `0x123451234`, then what physical address will it write to? You may leave your answer as an unsimplified arithmetic expression. If not enough information is supplied write "unknown" and explain briefly. If an exception would occur instead write "fault" and explain briefly.

   Solution: 0x20001234 OR variations due to typos above, e.g. $0x2000 \cdot 16384 + 0x1234$

4. (10 points) Below there is an incomplete C function, where `______` represents an omitted integer.

   When complete, the function will take a four-byte integer `A` and return a copy of A where:

   - the second least significant byte is replaced with a A's least significant byte;
   - the fourth least significant byte is replaced with a A's third least significant byte;
   - and the other bytes are unchanged

   (for example, `0x12345678` becomes `0x34347878`.)

```
unsigned int f(unsigned int A) {

    unsigned int A_masked = A & __ 0xFF00FF __ ;

    return (A_masked << __ 8 __) | A_masked;
}
```

   Fill in the integers above to complete the function. (If you think you cannot complete the function by merely filling in the blanks above, explain briefly.)

5. Consider the following x86-64 Linux assembly snippet:

```
.global foo
foo:
    movq (%rsi), %rsi
    addq (%rdi), %rsi
    movq %rsi, (%rcx)
    movq global_variable, %rax
    addq %rsi, %rax
    ret
```

(a) (8 points) Suppose the function `foo` was declared so it could be called by C code. Complete the declaration in a way that's as consistent as possible with the assembly above by adding in a return type and one or more argument types. (We do not care about the names of the arguments. The calling convention is summarized on the reference sheet.)

long foo(long*, long*, ALMOST-ANYTHING, long*);*or* `char* foo(char**,char**, ALMOST-ANYTHING, char**)` *or variations with added 'unsigned' or similar*

(b) (6 points) Object files generally contain *relocations* which indicate places where, before an executable is produced, an address must be filled in by the linker within some machine code. Identify what relocations will most likely be produced when the above assembly snippet is converted into an object file. For each relocation, specify what instruction it will be produced for.

Solution: for global_variable in movq

6. (7 points) Consider the following Y86-64 program running on the single-cycle processor we showed in lecture and which is shown on the reference sheet:

```
0x000: 30f80200000000000000 | irmovq $0x2, %r8
0x00a: 6088                 | addq %r8, %r8
0x00c: 00                   | halt
```

This program should place `0x4` in %r8 when it halts after three cycles.

Normally the PC register is set using a MUX that most commonly takes input from a circuit that adds the current instruction's length to the PC register. Suppose instead of setting this directly, a buggy implementation placed the MUX's output into a new register, then fed this new register to the PC.

As a result of this bug, we'd expect the program above to halt after 5 cycles and place 0x8 in %r8. (Fill in the blanks, showing any work below.)

7. Consider adding a new instruction `pop2q rA, rB` to Y86-64, which would pop two 64-bit values from the stack. The first value would be stored in the register specified by rA, the second in rB.

For the following questions, suppose this instruction was added to the single-cycle processor (without making any instructions take additional cycles). To support this instruction, the data memory is modified to support 128-bit reads from *was written to in original* an address in addition to 64-bit reads from an address.

(a) (5 points) Besides the changes to the data memory, the processor would also need to increase the number and/or size of which of the following to support this instruction? *-1 point per wrong* **Select all that apply.**

- ☐ ALU inputs
- ☐ ALU outputs
- ☐ instruction memory inputs
- ☐ instruction memory outputs
- ■ register file inputs
- ☐ register file outputs

(b) (6 points) What calculations, if any, will the ALU perform during this instruction?

Solution: `%rsp + 16`

8. For the following questions, consider a system with a 3-way set associative data cache with 16 sets, 8 byte blocks, an LRU replacement policy, a write-back policy, and a write-allocate policy.

(a) (12 points) Consider the following C code that accesses arrays `A` and `B`, each of which are arrays of 4-byte integers which start on an address that is a multiple of 2^{10} (1024). (Note that each cache block can hold two integers.)

```
for (int ii = 0; ii < 1024; ii += 16) {
    for (int j = 0; j < 1024; j += 1) {
        for (int i = ii; i < ii + 16; i += 1) {
            result += A[i] * B[j];
        }
    }
}
```

How many data cache misses will occur when the above code is run? Assume only accesses to `A` and `B` use the data cache and that the cache is initially empty. You may leave your answer as an unsimplified arithmetic expression.

Solution: for each ii loop: 16 A[i] accesses, yielding 8 misses; for each j loop 1/2 of a B[j] miss; $(1024 \div 16) \cdot (8 + 1024 \cdot \frac{1}{2})$

(b) Suppose the cache is initially empty, and then the following accesses occur:

1. a write of 4 bytes to an address with tag 0x1000, set 0, and offset 0
2. a write of 4 bytes to an address with tag 0x1001, set 0, and offset 4
3. a read of 4 bytes from an address with tag 0x1002, set 0, and offset 0
4. a read of 4 bytes from an address with tag 0x1001, set 0, and offset 4
5. a read of 4 bytes from an address with tag 0x1000, set 0, and offset 4
6. a write of 4 bytes to an address with tag 0x1003, set 0, and offset 0

i. (6 points) For which of the accesses identified above will the cache need to read some value from the next level cache or main memory? (Identify the accesses by their numbers above.) *-1 point per disagreeing access*

1, 2, 3, 6

ii. (4 points) Access number 6 above will be a cache miss. What, if anything, will be evicted from the cache as a result?

the block with tag 0x1002, set 0

9. For the following questions, consdier a pipelined Y86-64 processor like the design we discussed in lecture except:

   - it has the following 7 pipeline stages:
     - fetch, decode 1, decode 2, execute 1, execute 2, memory, writeback
   - register values are read during the first decode stage but not available until the second decode stage (so if a reigster is updated during the decode 1 stage, the updated value will not be read)
   - the result of a ALU operation is not available until near the end of the execute 2 stage (after the inputs were obtained near the beginning of the execute 1 stage)
   - conditional jumps are predicted as taken; if there is a misprediction, the corrected instruction is fetched during the conditional jump's **memory** stage
   - `ret` instructions cause the processor to stall until the cycle after the return address is read from memory (the `ret` instruction's writeback stage)

(a) Consider the following assembly code which starts execution at the `irmovq` and finishes at the `halt`:

```
                      1 2  3  4  5  6  7  8
irmovq $10, %rdi      F D1 D2 E1 E2 M  W
call bar                F  D1 D2 E1 E2 M  W
pushq %rax
halt

bar:                                        9  10 11 12 13
    addq %rdi, %rdi           F  D1 D2 E1 E2 M  W
    mrmovq 500(%rdi), %rax       F  D1 x  D2 E1 E2 M  W
        /* was written: rmmovq 500(%rdi), %rax on exam */
    ret                             F     D1 D2 E1 E2 M  W
                                                     F  D1 D2 E1 E2 M  W     push
                                                        F                    halt
```

i. (10 points) On the processor above, if the `irmovq $10, %rdi` instruction is fetched during cycle 1, then the `halt` instruction will be fetched during what cycle? Show your work below or above.

Solution: 13

ii. (8 points) Which of the following forwarding will occur when the assembly code from the previous page is run? *-1 point per wrong* **Select all that apply.**

- [x] `%rdi` from `irmovq` to `addq`
- [ ] `%rdi` from `irmovq` to `mrmovq` *was written rmmovq (and below)*
- [x] `%rdi` from `addq` to `mrmovq`
- [ ] `%rsp` from `call` to `pushq`
- [x] `%rsp` from `call` to `ret`
- [ ] `%rsp` from `pushq` to `ret`
- [ ] `%rsp` from `ret` to `pushq`
- [ ] `%rax` from `mrmovq` to `pushq`
- [ ] `%rax` from `ret` to `pushq`

(b) (4 points) Suppose the pipelined processor is implemented using pipeline stage implementations that excluding pipeline registers require:

- 100 ps for the fetch stage
- 150 ps for the decode 1 stage
- 150 ps for the decode 2 stage
- 125 ps for the execute 1 stage
- 125 ps for the execute 2 stage
- 150 ps for the memory stage
- 100 ps for the writeback stage

and the pipeline registers have a register delay of 10 ps. Assume other components of the processor require negligible time.

Suppose the processor runs a conditional jump instruction which is not taken. This conditional jump is mispredicted, so the processor takes extra time to fetch the correct next instruction to run.

How long **in ps** after the processor starts the fetch stage of the conditional jump will it start the fetch stage of the correct next instruction? You may leave your answer as an unsimplified arithmetic expression.

160ps * 5 = 800ps

10. Consider the C function:

```
unsigned int DoComputation() {
    int N = 1024 * 1024 * 1024;
    unsigned int sum = 0;
    for (int i = 0; i < N; i += 1) {
        unsigned int a_value = A[i]; // Line X
        unsigned int b_value = B[i]; // Line Y
        sum += a_value - b_value;
        A[i] = a_value * sum;
    }
    return sum;
}
```

(a) (5 points) When running this code with a 2^{16}-byte L1 data cache, we discover that the lines labeled 'Line X' and 'Line Y' are responsible for a bulk of the cache misses, having about a 8% cache miss rate. Which of the following changes are likely to substantially improve the L1 data cache hit rate? **Select all that apply.**

- [x] increasing the cache's block size (but keeping the cache size the same)
- [ ] increasing the cache's associativity (but keeping the cache size the same)
- [x] when the program accesses a cache block, starting to fetch the cache block containing the sequentially following address (if it's not already in the cache)
- [ ] changing the cache use a write-allocate policy instead of a write-no-allocate policy
- [ ] unrolling the for loop

(b) (4 points) Suppose after improving the cache hit rate, we discovered that the performance of the function above on an out-of-order processor was determined by the speed at which the processor could perform multiplications. Suppose the processor had a single, pipelined multiplier that accepted a pair of numbers to multiply and produced a result 5 cycles later. We would expect the loop to take approximately how many cycles per iteration?

1