Fill out the bottom of this page with your computing ID.
Write your computing ID at the top of each page in case pages get separated.

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

1. For the following question, suppose memory initially contains the following bytes:

memory address	byte value	memory address	byte value	memory address	byte value
…	…				
0x1000	0x44	0x1008	0x40	0x1010	0x99
0x1001	0x54	0x1009	0x50	0x1011	0x98
0x1002	0x64	0x100a	0x66	0x1012	0x97
0x1003	0x74	0x100b	0x78	0x1013	0xA0
0x1004	0x84	0x100c	0x9A	0x1014	0xB0
0x1005	0x00	0x100d	0xBC	0x1015	0xC0
0x1006	0x10	0x100e	0x10	0x1016	0xD0
0x1007	0x20	0x100f	0x20	0x1017	0xE0
				…	…

(a) (4 points) When reading a 4-byte value from address `0x1008` and interpreting them as little-endian, what value is read? Write your answer in hexadecimal; if not enough information is given write "unknown" and explain briefly.

0x78665040

(b) Suppose `%rax` contains `0x1000` and `%rcx` contains `0x12345678`, and the Y86-64 instruction **`rmmovq`** `%rcx, 0x10(%rax)` is run. After this happens, write down what the values at each of the following addresses will be.

i. (2 points) `0x100f`

0x20

ii. (2 points) `0x1011`

0x56

iii. (2 points) `0x1017`

0x00

(c) (4 points) Suppose `%rbx` contains `0x800` and `%rcx` contains `0x12345678`, After running the x86-64 instruction `leaq 0x5(%rbx,%rbx), %rcx`, the value of `\%rcx` will be? Write your answer as a hexadecimal number. (`leaq` is the 64-bit version of the load effective address instruction.) If not enough information is given, write "unknown" and explain briefly.

0x1005

2. For the following questions, consider the following assembly function "foo" which takes one integer argument and has one integer return value:

```
foo:
    movq %rdi, %rax
    shlq $4, %rdi
    andq $0xF0, %rdi
    andq $0xF, %rax
    orq %rdi, %rax
    ret
```

(shlq is the shift left instruction; andq is bitwise and instruction; orq is the bitwise or instruction)

(a) (4 points) When this function is called with a argument of `0x21`, what will its return value be?

0x11

(b) When this function is called with an argument of `0x1`, what will the value of the condition codes SF and ZF be after its first `andq` instruction executes?

i. (2 points) SF

0

ii. (2 points) ZF

0

(c) (5 points) If this function is executed on an out-of-order, multiple issue processor, the **`shlq`** instruction could execute at the same time as which of the following instructions? **Select all that apply.**

- √ **`andq $0xF, %rax`**
- ◯ **`orq %rdi, %rax`**
- √ **`movq %rdi, %rax`**
- ◯ **`andq $0xF, %rax`**
- ◯ **`andq $0xF0, %rdi`**

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

```
int array[4] = {0,1,2,3};
int *p = &array[0];
p += 2;
*p = 8;
p[1] = 9;
```

(a) (4 points) After the above code runs, what is the value of the first four elements of `array`?

{ 0 , 1 , 8 , 9 ,…}

(b) (4 points) After the above code runs, if the address of `array[0]` is `0x1234` and `int`s take up 4 bytes, what is the final value of the pointer `p`?

0x123c

4. (4 points) Consider the following C function:

```
unsigned int mystery(unsigned int argument) {
    int value1 = ((argument & 0xFF) << 16);
    int value2 = (argument & 0xFF00);
    return value1 | value2;
}
```

Which of the following describes what this function returns?

- ◯ its argument with the least significant byte of its argument moved into the third least significant byte and all other bytes cleared
- ◯ its argument with the least significant and second least significant byte swapped, while keeping the other bytes the same
- √ its argument with the least significant byte of its argument moved into the third least significant byte and all other bytes except the second least significant byte cleared
- ◯ a number which is all zeroes except for its second least significant byte, and that byte is constructed by bitwise or'ing the least significant byte of the argument with the second least significant significant byte of the argument
- ◯ its argument with the least significant and second least significant byte swapped, while clearing the other bytes
- ◯ none of the above: _____

5. For these questions, consider the **single-cycle** Y86-64 processor we discussed in lecture (and implemented in the HCL homeworks). A diagram of this processor is included in the reference material at the end of the exam. Suppose we want to add a new instruction to this processsor `immovq I, D(rA)` which would move the 8-byte constant `I` into memory at address `D+R[rA]` (where `R[rA]` represents the value of the register with index rA).

(a) (5 points) Which of the following changes would we need to make to our single-cycle processor to support this insturction? **Select all that apply.**

- ◯ increasing the width of one or more of the ALU inputs
- √ increasing the size of the output of the instruction memory
- ◯ increasing the size of the output of the data memory
- ◯ increasing the size of the input to the data memory
- ◯ adding a new operation to the ALU

(b) (4 points) When this instruction is executing, the data input to the data memory will be equal to

- ◯ one of the outputs of the register file
- ◯ the output of the program counter (PC) register plus a fixed constant
- ◯ the output of the ALU
- ◯ part of the output of the instruction memory
- ◯ the output of the program counter (PC) register
- ◯ none of the above, explain briefly:

(c) (4 points) Which encoding for this new instruction would likely minimize the amount of additional logic or circuitry (outside of any changes to the register file, memories, or register file) required to implement it? (*rB* represents the register index for rB, *D* and *I* represent the 8-byte constants D and I, respectively (in little-endian). When two values are listed for a byte they represent the most significant 4 bits of that byte, followed by the least significant 4 bits of that byte.)

- ◯ byte 0: `0xC`, *rB*; byte 1–8: *D*; byte 9–16: *I*
- ◯ byte 0: `0xC0`; byte 1: `0xF`, *rB*; byte 10–17: *I*; byte 2–9: *D*
- √ byte 0: `0xC0`; byte 1: `0xF`, *rB*; byte 2–9: *D*; byte 10–17: *I*
- ◯ byte 0: `0xC0`; byte 1: *rB*, `0xF`; byte 10–17: *I*; byte 2–9: *D*
- ◯ byte 0: `0xC0`; byte 1: *rB*, `0xF`; byte 10–17: *D*; byte 2–9: *I*
- ◯ byte 0: `0xC0`; byte 1–8: *D*; byte 9–16: *I*; byte 17: `0xF`, *rB*

6. For following questions, consider a seven-stage pipelined processor with the following pipeline stages:

   - fetch
   - decode
   - execute 1
   - execute 2
   - memory 1
   - memory 2
   - writeback

   In the case of stages divided into two compared to the five-stage pipelined processor discussed in lecture, the result of the stage (the ALU result for execute, the value read from the data memory for memory) is not available until near the end of the second stage.

   This processor uses branch prediction which assumes all branches (e.g. conditional jumps) are taken. The result of a conditional jump is determined near the end of the execute 2 stage of the conditional jump instruction, so the correct instruction can be fetched during the conditional jump's memory 1 stage.

(a) The following questions ask about how the processor would execute the assembly function `example`, shown below, assuming the conditional jump is **not taken**. Each instruction is marked with a number that is used in the questions below:

```
example:
    addq %r8, %r9           // 1
    jle label               // 2
    subq %r10, %r11         // 3
    andq %r12, %r13         // 4
    rmmovq %r9, 0(%r11)     // 5
    rmmovq %r13, 0(%r15)    // 6
    ret                     // 7

label:
    xorq %r10, %r11         // 8
    pushq %r12              // 9
    mrmovq 0(%r14), %r8     // 10
    mrmovq 8(%r14), %r9     // 11
    mrmovq 16(%r14), %r10   // 12
    ret
```

i. (4 points) As a result of branch misprediction, the processor will fetch some instructions which should not be executed. Identify these instructions by their numbers above.

8-10

ii. (4 points) When executing instructions 3–7, how many cycles of stalling will be required? (Count each time a new instruction cannot be fetched as a cycle of stalling.)

0

iii. (8 points) When executing instructions 3–7, the values of some registers must be retrieved by forwarding (instead of using a value from the register file). Identify which register values in which instructions below. (More blanks are provided than is required.)

For instruction 3: _____

For instruction 4: _____

For instruction 5: _____

For instruction 6: _____

For instruction 7: _____

7. Suppose we build a processor using the following components, which require the following amounts of time to perform their operations (reading values, writing values, performing computations, etc.):

   - registers for program counter and/or pipeline registers: 10 ps register delay per register
   - instruction cache: 200 ps
   - predicted program counter value computation (given opcode and current program counter): 50 ps
   - register file read: 150 ps
   - register file write: 150 ps
   - ALU operation: 225 ps
   - data cache read/write: 175 ps

   Assume other components require negligible time.

(a) (4 points) If we build a **single-cycle** (non-pipelined) processor using this design, what would we expect the cycle time to be?

200 ps (fetch; predicted PC happens in parallel) + 150 ps (reg read) + 225 ps (ALU) + 175 ps (data cache access)

(b) (4 points) If we build a five-stage pipelined processor using these components, what would we expect its cycle time to be?

235 ps (partial credit for 225 ps)

8. For the following questions, consider a two-way, write-back, wire-allocate set associative cache with a least recently used (LRU) replacement policy with the following contents:

	way 0				way 1				
index	valid	tag	data bytes	dirty	valid	tag	data bytes	dirty	LRU way
0	1	0x12	33 44 66 99	0	1	0x23	55 66 77 99	0	
1	1	0x12	44 55 66 00	0	1	0x03	AA BB CC DD	1	
2	1	0x12	00 01 02 03	0	1	0x04	00 01 02 03	1	
3	1	0x12	08 09 0A 0B	0	1	0x06	04 05 06 07	0	

The data bytes shown are a sequence of hexadecimal numbers, starting from the data byte at the lowest address.

Each question is **independent** and starts from the cache in the state shown above.

(a) (4 points) Based on how the above cache splits up addresses, write the address with tag `0x01`, index `0x1`, and offset `0x3`. (The value at this address is not currently stored in the cache.)

0b10111 = 0x17

(b) For each of the following accesses *evaluated independently*, identify wheter it would be a hit and whether it would require updating the LRU way or dirty bit information. Assume the cache starts in the state shown above **before each access**.

i. (6 points) reading a byte from an address with tag `0x12`, index `0x0`, offset `0x1` **Select all that apply.**

√ hit, returning 0x66 ◯ updates a dirty bit √ updates LRU way ◯ causes a value to be written to the next level of cache

ii. (5 points) writing a byte from an address with tag `0x12`, index `0x2`, offset `0x1` **Select all that apply.**

√ hit √ updates a dirty bit ◯ updates LRU way ◯ causes a value to be written to the next level of cache

iii. (6 points) reading a byte from an address with tag `0x06`, index `0x3`, offset `0x0` **Select all that apply.**

√ hit, returning 0x04 √ updates a dirty bit ◯ updates LRU way ◯ causes a value to be written to the next level of cache

9. For the following questions, consider a system with the following caches:

- an L1 unified (data and instruction) cache:
  - write-back, write-allocate policy
  - 16KB total size
  - 4-way set associative
  - 64B block size
  - LRU replacement policy
  - 3 cycle hit time
- an L2 unified (data and instruction) cache:
  - write-back, write-allocate policy
  - a 512KB cache size
  - 8-way set associative
  - 64B block size
  - LRU replacement policy
  - 10 cycle hit time

Assuming that when there is a miss in the L1 cache, it takes 3 cycles (the hit time) to detect this, and then an L2 access starts, and that the access time for main memory is 200 cycles.

(a) (6 points) Divide the address `0xF1234` into tag, index, and offset for the L1 cache above. (Write your answers in either hexadecimal or binary.)

Tag: 0xF1 = 0b11110001 ; Index: 0b001000 = 0x8 ; Offset: 0b111000 = 0x34

(b) (4 points) If a program has a 90% hit rate in the L1 cache and a 50% hit rate in the L2 cache, would be the average memory access time for an access to the L1 cache (in cycles)? You may write your answer as an unsimplified arithmetic expression.

$3 + 0.1 \cdot (10 + 0.5 \cdot 100) = 9$

(c) (6 points) When a value is written to the L1 cache, which of the following **could** happen? **Select all that apply.**

- ◯ another value which shares the same tag bits (but has different index bits) will be written to the L2 cache
- √ another value which shares the same index bits (but has different tag bits) will be written to main memory
- ◯ that value is also written to the L2 cache
- √ another value which shares the same index bits (but has different tag bits) will be written to the L2 cache
- ◯ that value is also written to main memory
- ◯ another value which shares the same tag bits (but has different index bits) will be written to main memory

10. (10 points) Consider the following C code:

```
char array[24];
void foo() {
    for (int i = 0; i < 5; ++i) {
        for (int j = 0; i < 24; j += 6) {
            array[j] += 1;
        }
    }
}
```

Assuing that:

- foo() only accesses the data cache to access `array`
- the data cache is initially empty (all invalid) when foo() starts
- the address of `array[0]` is a multiple of 2^{20}
- the code is run using a 16-byte direct-mapped cache with 4-byte blocks

How many data cache misses will occur when foo() is run?

Solution: 13; first iteration (5 misses): miss 0 (0-3; set 0); miss 6 (4-7; set 1); miss 12 (12-15; set 3); miss 18 (16-19; set 0); subsequence iterations: miss 0 and 18 (8 total)

11. For the following questions, consider the following function:

```
void example(int N, int *A, int *B, int *C) {
    for (int i = 0; i < N; i += 1) {  // loop 1
        for (int j = 0; j < i; j += 1) { // loop 2
            C[i * N + j] += A[i * N + j] * B[j * N + i];
        }
        for (int j = i; j < N; j += 1) { // loop 3
            C[i * N + j] += B[i * N + j] * A[j * N + i];
        }
    }
}
```

(a) (5 points) Which of the following transformations **can** be performed without adding some check for A, B, and C pointing to overlapping arrays? **Select all that apply.**

- ◯ using vector instructions to perform the additions to 'C'
- √ unrolling the loop labelled 'loop 2'
- √ unrolling the loop labelled 'loop 3'
- ◯ performing cache blocking
- ◯ transforming the above code so the inner loop(s) iterates through 'i' instead of 'j'

(b) (4 points) Which of the following transformations are likely to improve the cache locality of the the above code? **Select all that apply.**

- ◯ transforming the above code so the inner loop(s) iterates through 'i' instead of 'j'
- ◯ unrolling the loop labelled 'loop 2'
- √ performing cache blocking
- ◯ unrolling the loop labelled 'loop 3'

(c) (4 points) Which of the following changes to (the non-vectorized code in) `example()` would result in a function that is simpler to vectorize? (Each of these changes makes a new function that calculates something different.) **Select all that apply.**

- √ accessing `B[i * N + j]` in loop 2 (instead of `B[j * N + i]`)
- ◯ changing loop 3 to increment `j` by 8 instead of 1
- ◯ accessing `C[i]` in loop 3 (instead of `C[i * N + j]`)
- √ accessing `A[j + i]` in loop 3 (instead of `A[j * N + i]`)

12. For these quetions, consider a system which has:

- 33-bit virtual addresses
- 8192 byte pages (2^{13} byte pages)
- 8 byte page table entries
- 2-level page tables, with 1024 entries in tables at each level
- a 16-entry, 8-way TLB with a random replacement policy
- a 32 KB direct-mapped L1 cache (shared between data and instructions)

For the questions below, the *first-level* page table is the one pointed to by the page table base register.

(a) (3 points) If the page table base register contains `0x100000`, then what would the physical address of the first-level page table entry for virtual page 4 be?

0x100000

(b) (3 points) If the second-level page table for virtual page 4 starts at physical address `0x200000`, then what is the physical address of the second-level page table entry for virtual page 4?

0x200020

(c) (4 points) The page table lookup for the virtual address would use the same **first-level** page table entry as the lookup for _____. **Select all that apply.**

- √ `0x012345001`
- ◯ `0x022345678`
- √ `0x012346678`
- ◯ `0x112345678`

(d) (4 points) The TLB lookup for the virtual address would access the same TLB set as the lookup for _____. **Select all that apply.**

- √ `0x012345001`
- √ `0x022345678`
- ◯ `0x012346678`
- √ `0x112345678`

(e) (4 points) If a memory access to virtual address `0x012345678` has a TLB hit during its virtual to physical adress translation, then _____. **Select all that apply.**

- ◯ the memory access would also be an L1 cache hit
- ◯ an access to `0x012345679` would also cause a TLB hit
- ◯ there was a prior access to this virtual address
- ◯ the access would not result in an exception *could if read-only/kernel-mode-only/etc.*

(f) (4 points) How much storage (in bits) does the TLB have for tags? (You may leave your answer as an unsimplified arithmetic expression.)

$16 \times (33 - 14) = 16 \times 19 = 304$ bits

13. Suppose a single-core system x86-64 has three active processes A, B, and C, and

- process A is waiting for keyboard input before it can run again
- process B and C are running infinite loops performing some computation

(a) (4 points) Based on the scenario described above, the value of the processor register `%rax` could be _____. **Select all that apply.**

- ◯ process A's value for %rax
- √ process B's value for %rax
- √ process C's value for %rax
- √ a value for %rax used by the OS

(b) (5 points) Order these events that might occur after keyboard input occurs:

#	event
2	an operating system function starts running
1	the program counter for process B or C is saved
3	the registers for process A are restored from a saved copy
4	a return from exception instrution is executed
5	code for process A starts running