Name: ____________________________________

3. Address Translation (25 pts)

Imagine we have a machine with 16-bit virtual addresses, 12-bit physical addresses, and:

- Page size of 256 bytes.
- Translation lookaside buffer (TLB) with 8 ways and 4 sets.
- One-level cache with capacity of 256 bytes, 16-byte cache block size, and 2-way associativity.

(a) For the virtual address below, label the bits used for each component, either by labelling boxes or with arrows to indicate ranges of bits. *Hint:* there may be more than one label for some bits.

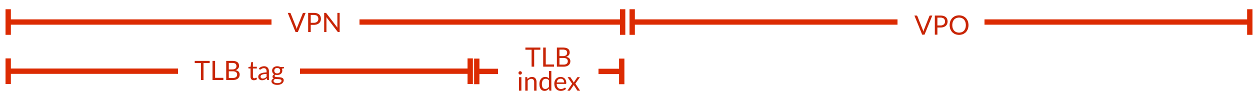
(i) Virtual page offset ("VPO")

(ii) Virtual page number ("VPN")

(iii) TLB index ("index")

(iv) TLB tag ("tag")

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	1	0	0	1	1	0	1	1	0	0	0



(b) How many total page-table-entries are there per process in this system?

256-byte pages = 2^8, 2^16 / 2^8 = 2^8 VPNs — <u>256 page table entries per process</u>

(c) How many ***sets*** are there in the cache?

256 bytes total / 16 bytes/block = 16 cache blocks total
16 blocks / 2 ways = <u>8 sets</u>

(d) Assume that the virtual address above has been translated to a physical address in memory.
Fill in the known bits of the physical address below, and **label the bits for each component** as you did in part (a) — again, some bits may have more than one label.

(i) Physical page number ("PPN")

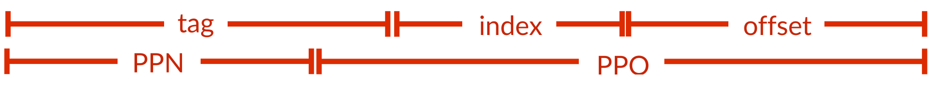
(ii) Physical page offset ("PPO")

(iii) Cache index ("index")

(iv) Cache tag ("tag")

(v) Cache offset ("offset")

11	10	9	8	7	6	5	4	3	2	1	0
				1	1	0	1	1	0	0	0



Question 5: The Stack [12 pts]

The recursive factorial function `fact()` and its x86-64 disassembly is shown below:

```
int fact(int n) {
     if(n==0 || n==1)
          return 1;
     return n*fact(n-1);
```

```
000000000040052d <fact>:
  40052d:  83 ff 00        cmpl   $0, %edi
  400530:  74 05           je     400537 <fact+0xa>
  400532:  83 ff 01        cmpl   $1, %edi
  400535:  75 07           jne    40053e <fact+0x11>
  400537:  b8 01 00 00 00  movl   $1, %eax
  40053c:  eb 0d           jmp    40054b <fact+0x1e>
  40053e:  57              pushq  %rdi
  40053f:  83 ef 01        subl   $1, %edi
  400542:  e8 e6 ff ff ff  call   40052d <fact>
  400547:  5f              popq   %rdi
  400548:  0f af c7        imull  %edi, %eax
  40054b:  f3 c3           rep ret
```

(A) Circle one: [1 pt] `fact()` is saving `%rdi` to the Stack as a [✓ **Caller**] // **Callee**

(B) How much space (in bytes) does this function take up in our final executable? [2 pt]

Count all bytes (middle columns) or subtract address of next instruction (0x40054d) from 0x40052d.

32 B

(C) **Stack overflow** is when the stack exceeds its limits (i.e. runs into the Heap). Provide an argument to `fact(n)` here that will cause stack overflow. [2 pt]

Any negative int

We did mention in the lecture slides that the Stack has 8 MiB limit in x86-64, so since 16B per stack frame, credit for anything between 512 and TMax (2^{31}-1).

(D) If we use the `main` function shown below, answer the following for the execution of the entire program: [4 pt]

```
void main() {
     printf("result = %d\n",fact(3));
}
```

Total frames created: **5**	Maximum stack frame depth: **4**

main → fact(3) → fact(2) → fact(1)
→ printf

(E) In the situation described above where `main()` calls `fact(3)`, we find that the word `0x2` is stored on the Stack at address `0x7fffdc7ba888`. At what address on the Stack can we find the return address to `main`()? [3 pt]

`0x7fffdc7ba8a0`

Only `%rdi` (current `n`) and return address get pushed onto Stack during `fact()`.

Address	**Contents**
	<Rest of Stack>
`0x7fffdc7ba8a0`	Return addr to `main()`
`0x7fffdc7ba898`	Old `%rdi` (n=3)
`0x7fffdc7ba890`	Return addr to `fact()`
`0x7fffdc7ba888`	Old `%rdi` (n=2)
`0x7fffdc7ba880`	Return addr to `fact()`

Name: ________________________________

5. Pointers and Memory (15 pts)

For this section, refer to this 8-byte aligned diagram of memory, with addresses increasing top-to-bottom and left-to-right (address 0x00 at the top left). When answering the questions below, don't forget that x86-64 machines are little-endian. If you don't remember exactly how endianness works, you should still be able to get significant partial credit.

Memory Address	+0	+1	+2	+3	+4	+5	+6	+7
0x00	aa	bb	cc	dd	ee	ff	00	11
0x08	00	00	00	00	00	00	00	00
0x10	ab	01	51	f0	07	06	05	04
0x18	de	ad	be	ef	10	00	00	00
0x20	ba	ca	ff	ff	1a	2b	3c	4d
0x28	a0	b0	c0	d0	a1	b1	c1	d1

```
int*  x = 0x10;
long* y = 0x20;
char* s = 0x00;
```

(a) Fill in the type and value for each of the following C expressions:
(.5pts for each type, 2pts for each value)

Expression (in C)	Type	Value (in hex)
*x	int	0xf05101ab
x+1	int*	0x14
*(y-1)	long	0x00000010efbeadde
s[4]	char	0xEE

(b) Assume that all registers start with the value 0, except `%rax` which is set to 8. Determine what the final values of each of these registers will be *after* executing the following instructions:

Instruction	Points	Register	Value
	(1 pt)	%rax	8
movb %al, %bl	(2 pts)	%bl	8 or 0x8
leal 2(%rax), %ecx	(2 pts)	%ecx	10 or 0xa
movsbw (,%rax,4), %dx		%dx	65466 or 0xffba

Question 9: Virtual Memory (8 pts)

This election season, the US will computerize the voting system. There were approximately 2^{27} voters in 2012. There are four candidates in the running and so each voter will submit letter A, B, C, or D. The votes are stored in the `char votes[]` array.

The following loop will count the votes to determine the winner. We are given a 1 MiB byte-addressed machine with 4 MiB of VM and 128 KiB pages. Assume that `votes[]` and `candidates[]` are page-aligned and `i` is stored in a register.

```
#define NUM_VOTERS 134217728                    // 2^27
int candidates[] = {0,0,0,0};                   // initialize to 0s
for (int i = 0; i < NUM_VOTERS; i++) {          // Loop 1
     if (votes[i] == 'A')  candidates[0]++;
     if (votes[i] == 'B')  candidates[1]++;
     if (votes[i] == 'C')  candidates[2]++;
     if (votes[i] == 'D')  candidates[3]++;
}
```

a) How many bits wide are the following? [2 pt]

VPN	__5__	Page Offset	__17__
PPN	__3__	Page Table Base Register	__20__

b) We are given a fully-associative TLB with 4 entries and LRU replacement policy. One entry is reserved for the Code. In the *best case scenario*, how many votes will be counted before a TLB miss occurs? [2 pt]

$\boxed{2^{18}}$

Best case: TLB already has code page, candidate page, and 2 votes pages loaded. One page is 2^{17}B. votes is a character array so each page holds 2^{17} votes. $2 * 2^{17} = 2^{18}$ votes

We want to improve our machine by expanding the TLB to hold 8 entries instead of 4. We also revised our `for` loop, which replaces `Loop 1`. Assume `i` and `vote` are stored in registers.

```
for (int i = 0; i < NUM_VOTERS; i++) {     // Loop 2
     char vote = votes[i];
     if (vote == 'A')     candidates[0]++;
     if (vote == 'B')     candidates[1]++;
     if (vote == 'C')     candidates[2]++;
     if (vote == 'D')     candidates[3]++;
}
```

c) Now how many votes can be counted before a TLB miss in the *best case scenario*? [2 pt]

$\boxed{6 * 2^{17}}$

Even though we have 1 access per vote instead of 4 with the new for loop, this does not change the fact that in the best case, we will only miss in the TLB if we go through all our pre-loaded pages. Since our TLB can now hold 6 pages for the votes pages, we can get $6 * 2^{17}$ votes before a miss.

Question 8: Caches (10 pts)

We are using a 20-bit byte addressed machine. We have two options for caches: **Cache A** is fully associative and **Cache B** is 4-way set associative. Both caches have a capacity of 16 KiB and 16 B blocks.

a) Calculate the TIO address breakdown for **Cache A**: [1 pt]

Tag	Index	Offset
16	0	4

b) Below is the initial state of **one set** (four ~~slots~~ lines) in **Cache B**. Each slot holds 2 LRU bits, with `0b00` being the most recently used and `0b11` being the least recently used. Circle ONE option below for two memory accesses that result in the final LRU bits shown and ***only one block replacement***. [2 pt]

Index	~~Slot~~ Line	Tag	Initial LRU bits		Final LRU bits
1001 1110	0	0110 1010	00		10
	1	0000 0001	10	→	00
	2	0101 0101	01		11
	3	1010 1100	11		01

(1) 0x019D0, 0xAD9D0 [✓ (2) 0xAC9E0, 0x129E0]

(3) 0xAD9D0, 0x019D0 (4) 0x129E0, 0xAC9E0

c) For the code given below, calculate the hit rate for **Cache B** assuming that it starts cold. [3 pt]

```
#define ARRAY_SIZE 8192
int int_arr[ARRAY_SIZE];                        // &int_arr = 0x80000
for (int i = 0; i < ARRAY_SIZE / 2; i++) {
      int_arr[i] *= int_arr[i + ARRAY_SIZE / 2];
}
```

Access pattern is R `i`, R `i+ARRAY_SIZE/2`, W `i`. Array index jump is 4096*4 = 2^{14} B away, so maps into same set same set (`I+O=12<14`). N=4, so both blocks can fit in cache at once. Indices are not revisited and each block holds 16 B / 4 B = 4 indices, so first index is MMH, other 3 are HHH, so HR = 10/12 = 5/6.

$\boxed{5/6}$

d) For each of the proposed changes below, write **U** for "increase", **N** for "no change", or **D** for "decrease" to indicate the effect on the hit rate of **Cache B** for the loop shown in part (c): [2 pt]

Direct-mapped	__D__	Increase cache size	__N__
Double ARRAY_SIZE	__N__	Random block replacement	__D__

e) Calculate the AMAT for ~~a multi-level cache given~~ the following values. Don't forget units! [2 pt]

HT = Hit Time, MR = Miss Rate, ~~GMR = Global Miss Rate~~

L1\$ HT	L1\$ MR	~~L2\$ HT~~	~~GMR~~	MEM HT
4 ns	20%	~~25 ns~~	~~5%~~	500 ns

~~$HT_1 + MR_1*HT_2 + GMR*HT_{MEM} = 4 + 5 + 25$~~ AMAT = 4 + 0.2*500

104 ns ~~34 ns~~

Name:____

8. *Memory Allocation* (**11** points) Consider the (tiny) heap below using an implicit free-list allocator with coalescing, where each rectangle represents 4 bytes and a header or boundary tag of the form `x|y` indicates a block of size `x` (in base-10) where `y` being `1` means allocated. Addresses (in base-10) are written below the rectangles. Unlike your Lab 5 allocator, the allocator for this heap uses boundary tags for allocated blocks.

16|1	12	27	16|1	20|0	9	10	15	20|0	12|1	42	12|1	12|1	43	12|1	12|0	42	12|0
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124	128	132

(a) Suppose the next call to the allocator is `free(104)`. Show the state of the heap after this call completes by indicating in the picture of the heap above what addresses have different contents and what the new contents are.

(b) Suppose the next allocator call (after part (a)) is `malloc(4)`. Under a *first-fit policy*, what address would the call to `malloc` return?

(c) Suppose the next allocator call (after part (a)) is `malloc(4)`. Under a *best-fit policy*, what address would the call to `malloc` return?

(d) Given your answer to part (a) (i.e., after doing this free), what is the smallest `z` such that `malloc(z)` would not succeed unless the allocator expanded the size of the heap?

Solution:

(a)

16|1	12	27	16|1	32|0	9	10	15	20|0	12|1	42	32|0	12|1	43	12|1	12|0	42	12|0
64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	124	128	132

(b) 84

(c) 128

(d) 25 (partial credit for 28)

Question 5: Floating Point (10 pts)

Assume integers and IEEE 754 single precision floating point are **32 bits wide**.

a) Convert from IEEE 754 to decimal: **0xC0900000** [3 pts]

S = 1, E = 0b1000 0001, M = 0010…0; $-1.001_2 \times 2^2 = -100.1_2$

$\boxed{-4.5}$

b) What is the smallest positive integer that is a power of 2 that can be represented in IEEE 754 but not as a signed int? You may leave your answer as a power of 2. [2 pts]

Largest 32-bit signed int is $2^{31} - 1$.

$\boxed{2^{31}}$

c) What is the *smallest positive* integer x such that `x + 0.25` can't be represented? You may leave your answer as a power of 2. [3 pts]

Need 2^{-2} digit to run off end of mantissa, so
$10000000000000000000000.01_2 = 1.000000000000000000000001 \times 2^{22}$

$\boxed{2^{22}}$

d) We have the following word of data: **0xFFC00000**. Circle the number representation below that results in the *most negative number*. [1 pt]

Unsigned Integer	[✓ Two's Complement]	Floating Point
(positive number)	(negative number)	(NaN)

e) If we decide to stray away from IEEE 754 format by making our Exponent field 10 bits wide and our Mantissa field 21 bits wide. This gives us (circle one): [1 pt]

MORE PRECISION // [✓ LESS PRECISION]

Fewer mantissa bits means less precision.

Question 1: Number Representation (8 pts)

a) Convert `0x1A` into base 6. Don't forget to indicate what base your answer is in! [1 pt]

0x1A = 0b1 1010 = 16 + 8 + 2 = 26 = $4 \times 6^1 + 2 \times 6^0$

$\boxed{42_6}$

b) In IEEE 754 floating point, how many numbers can we represent in the interval [10,16)? You may leave your answer in powers of 2. [3 pts]

$\boxed{2^{22} + 2^{21} = 3 \times 2^{21}}$

10 = $0b1010 = 1.01 \times 2^3$ and 16 = $0b10000 = 1.0 \times 2^4$
Count all numbers with Exponent of 2^3 and Mantissa bits of the form { 1b'0, 1b'1, 21{1b'X} } and { 1b'1, 22{1b'X} }, for a total of $2^{21} + 2^{22}$ numbers.

c) If we use 7 Exponent bits, a denorm exponent of -62, and 24 Mantissa bits in floating point, what is the largest positive power of 2 that we can multiply with 1 to get *underflow*? [2 pts]

$\boxed{2^{-87}}$

Smallest denorm is $2^{-62} \times 0.0000\ 0000\ 0000\ 0000\ 0000\ 0001 = 2^{-86}$, which is representable. So the next smaller power of 2 is unrepresentable and causes underflow.

Name: ________________________________

2. C to Assembly (25 pts)

Imagine we're designing a new, super low-power computing device that will be powered by ambient radio waves (that part is actually a real research project). Our imaginary device's CPU supports the x86-64 ISA, but its general-purpose integer multiply instruction (`imul`) is very bad and consumes lots of power. Luckily, we have learned several other ways to do multiplication in x86-64 in certain situations. To take advantage of these, we are designing a custom multiply function, `spmult`, that checks for specific arguments where we can use other instructions to do the multiplication. But we need your help to finish the implementation.

Fill in the blanks with the correct instructions or operands. It is okay to leave off size suffixes.
Hint: there are reference sheets with x86-64 registers and instructions at the end of the exam.

```
long spmult(long x, long y) {
  if (y == 0)       return 0;
  else if (y == 1)  return x;
  else if (y == 4)  return x * 4;
  else if (y == 5)  return x * 5;
  else if (y == 16) return x * 16;
  else              return x * y;
}
```

```
spmult(long, long):
        testq  %rsi, %rsi
        je      .L3
        cmpq    $1, %rsi
        je      .L4
        cmpq    $4, %rsi
    jne   .L1
.case4:
        leaq    0(,%rdi,4), %rax
        ret
.L1:
        cmpq    $5, %rsi
        jne     .L2
        leaq   (%rdi,%rdi,4), %rax
        ret
.L2:
        cmpq    $16, %rsi
        jne     .else
        movq    %rdi, %rax
        salq  $4, %rax
        ret
.L3:
        movq    $0, %rax
        ret
.L4:
        movq    %rdi, %rax
        ret
.else:  # fall back to multiply
        movq    %rsi, %rax
        imulq   %rdi, %rax
        ret
```

Name:____

4. *Processes* (**12** points) In this problem, assume Linux.

   (a) Can the same program be executing in more than one process simultaneously?

   (b) Can a single process change what program it is executing?

   (c) When the operating system performs a context switch, what information does *NOT* need to be saved/maintained in order to resume the process being stopped later (circle all that apply):

   - The page-table base register
   - The value of the stack pointer
   - The time of day (i.e., value of the clock)
   - The contents of the TLB
   - The process-id
   - The values of the process' global variables

   (d) Give an example of an exception (asynchronous control flow) in which it makes sense to later re-execute the instruction that caused the exception.

   (e) Give an example of an exception (asynchronous control flow) in which it makes sense to abort the process.

Solution:

(a) Yes (the question is ambiguous as to what "simultaneous" means. We clarified during the exam, "Assume it *is* the case that multiple processes execute simultaneously. Then the question is whether more than one of these processes can be executing the same program." Under this interpretation, only "yes" is plausibly correct.)

(b) Yes

(c) The time of day and the contents of the TLB

(d) Page fault for memory on disk (other answers possible; full credit given just for page-fault even though that's ambiguous)

(e) Division by zero (other answers possible)

Name:____

10. *C vs. Java* (**11** points) Consider this Java code (left) and somewhat similar C code (right) running on x86-64:

```
public class Foo {
  private int[] x;
  private int y;
  private int z;
  private Bar b;
  public Foo() {
     x = null;
     b = null;
  }
}
```

```
struct Foo {
  int x[6];
  int y;
  int z;
  struct Bar * b;
};

struct Foo * make_foo() {
  struct Foo * f = (struct Foo *)malloc(sizeof(struct Foo));
  f->x = NULL;
  f->b = NULL;
  return f;
}
```

(a) In Java, `new Foo()` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `Foo`'s fields? (Do *not* include space for any header information, vtable pointers, or allocator data.)

(b) In C, `malloc(sizeof(struct Foo))` allocates a new object on the heap. How many bytes would you expect this object to contain for holding `struct Foo`'s fields? (Do *not* include space for any header information or allocator data.)

(c) The function `make_foo` attempts to be a C variant of the `Foo` constructor in Java. One line fails to compile. Which one and why?

(d) What, if anything, do we know about the values of the `y` and `z` fields after Java creates an instance of `Foo`?

(e) What, if anything, do we know about the values of the `y` and `z` fields in the object returned by `make_foo`?

Solution:

(a) 24

(b) 40

(c) `f->x = NULL` does not compile. In C, the field declaration `int x[6]` creates an inline array, not a pointer, so it does not make any sense to "assign NULL to the array" — the struct itself has slots for six array elements.

(d) We know both fields hold 0.

(e) We know nothing. (We know something abou their size, but not their contents – it could be any bit-pattern.)