Worksheet 9

Name: _____

UID: _____

Q3.

Which of the following is the best justification for using the middle bits of an address as the set index into a cache rather than the most significant bits?

(a) Indexing with the most significant bits would necessitate a smaller cache than is possible with middle-bit indexing, resulting in generally worse cache performance.

(b) It is impossible to design a system that uses the most significant bits of an address as the set index.

(c) The process of determining whether a cache access will result in a hit or a miss is faster using middle-bit indexing.

(d) A program with good spatial locality is likely to make more efficient use of the cache with middle-bit indexing than with high-bit indexing.

Q4. When you print the address of a variable from C, what kind of address is that?
(a) Physical Address
(b) Virtual Address
(c) Depends on the context
(d) None of the above

Q5. For a floating point number, what would be an effect of allocating more bits to the exponent part by taking them from the fraction part?
(a) You could represent fewer numbers, but they could be much larger.
(b) You could represent the same numbers, but with more decimal places.
(c) You could represent both larger and smaller numbers, but with less precision.
(d) Some previously representable numbers would now round to infinity

2. Integer Puzzles

True or false? If false, given an explanation as to why.

```
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

if x < 0, then x * x > 0	
ux > -1	
If x >= 0, then -x <= 0	
If x <= 0, then -x >= 0	
x >> 3 == x/8	
if x & 7 = 7, then (x << 30) < 0	

3. Bit manipulation

Implement the following using only integer constants 0 to 255, asn unary and binary integer operations (~ ! & | ^ + - << >>). Assume two's complement, 32 bit integers and arithmetic right shifts.

a.
```
// using | and ~ only
// returns the result of the and
int bitAnd (int x, int y) {

}
```

b.
```
// return 1 if x > 0, 0 otherwise
int isPositive (int x) {

}
```

c.
```
// given n and m where 0 <= n, m <= 3
// swap the nth and mth bytes
int byteSwap (int x, int n, int m) {

}
```

4. Caches

- ➔ Main memory is 256 KB (2^{18} bytes), byte-addressable
- ➔ Virtual address is 14 bits long
- ➔ A page of memory is 32 B (2^5 bytes)
- ➔ 8-way set associative TLB with 16 entries
- ➔ 4-way set associative cache with 8 lines and cache line of 4 bytes

TLB

Index	Tag	Valid	PPN
0	56	1	1185
0	9F	1	0DDE
0	FF	0	12C5
0	00	0	018A
0	448	1	0890
0	80A	1	1014
0	719	0	0E32
0	FDE	0	1FBC
1	F9	0	0E6D
1	1E6	1	03CD
1	AAE	1	155D
1	4CF	1	099F
1	B2D	0	165B
1	759	1	0EB3
1	E4B	0	1C97
1	BB	0	1427

Page Table

VPN	Valid	PPN	VPN	Valid	PPN
00	0	018A	141	0	07B7
0F	0	18E0	143	1	0E32
23	1	03CD	147	1	1FBC
44	0	1F77	15D	1	165B
5C	0	0ABF	15F	0	1687
5F	1	155D	161	0	1889
68	0	0AAE	166	0	1BCA
69	0	0866	191	0	17DF
75	1	099F	1C3	0	1E85
8B	1	0890	1E4	0	188A
A1	0	1E3D	1E5	0	069E
AC	1	1185	1E6	1	0EB3
C1	0	12F0	1E8	1	1D50
E8	0	1064	1EB	1	1C97
104	1	1014	1F3	1	0E6D
13E	1	0DDE	1FE	1	12C5

Cache

Index	Tag	Valid	Data [0:3]
0	39B7	1	FD CC DF 90
0	3779	0	D4 D8 8F 1D
0	4B17	0	53 27 4F C2
0	008B	0	EB 02 A4 57
1	4617	1	90 12 AA 67
1	7541	1	4C 63 B2 4B
1	7EF2	1	80 D1 0B 79
1	2240	1	1C C3 66 4E

Given the above information about the system memory system and states of the TLB, Page Table and Cache. Answer the following questions:

Fill out the following for an access to virtual address: `0x27CB`

VPN:
PPO/VPO:
TLB Index:
TLB Tag:
TLB Hit?:
Page Fault?:
PPN:
Physical Address:
Cache Offset (CO/BO):
Cache Index:
Cache Tag:
Cache Hit?:
Cache Data:

Fill out the following for an access to virtual address: `0x3E78`

VPN:
PPO/VPO:
TLB Index:
TLB Tag:
TLB Hit?:
Page Fault?:
PPN:
Physical Address:
Cache Offset (CO/BO):
Cache Index:
Cache Tag:
Cache Hit?:
Cache Data:

5.

Can the following program deadlock? Why or why not?

Initially: a = 1, b = 1, c = 1.

Thread 1:	Thread 2:
P(a);	P(c);
P(b);	P(b);
V(b);	V(b);
P(c);	V(c);
V(c);	
V(a);	

6. Assembly

When running the main function, what input would we need to provide in order to call "phase_defused"?

```
00000000004004b0 <main>:
  4004b0:      48 83 ec 18          sub    $0x18,%rsp
  4004b4:      bf 03 07 40 00       mov    $0x400703,%edi
  4004b9:      31 c0                xor    %eax,%eax
  4004bb:      48 8d 54 24 0c        lea    0xc(%rsp),%rdx
  4004c0:      48 8d 74 24 08        lea    0x8(%rsp),%rsi
  4004c5:      e8 d6 ff ff ff        callq  4004a0 <__isoc99_scanf@plt>|
  4004ca:      8b 74 24 0c           mov    0xc(%rsp),%esi
  4004ce:      8b 7c 24 08           mov    0x8(%rsp),%edi
  4004d2:      31 c0                xor    %eax,%eax
  4004d4:      e8 4e 01 00 00       callq  400627 <func2>
  4004d9:      83 f8 1d             cmp    $0x1d,%eax
  4004dc:      b0 00                mov    $0x0,%al
  4004de:      75 07                jne    4004e7 <main+0x37>
  4004e0:      e8 52 01 00 00       callq  400637 <phase_defused>
  4004e5:      eb 05                jmp    4004ec <main+0x3c>
  4004e7:      e8 27 01 00 00       callq  400613 <explode_bomb>
  4004ec:      48 83 c4 18          add    $0x18,%rsp
  4004f0:      c3                   retq
```

```
00000000004005dd <func1>:
  4005dd:       41 54                   push   %r12
  4005df:       41 89 fc                mov    %edi,%r12d
  4005e2:       55                      push   %rbp
  4005e3:       31 ed                   xor    %ebp,%ebp
  4005e5:       53                      push   %rbx
  4005e6:       89 f3                   mov    %esi,%ebx
  4005e8:       85 db                   test   %ebx,%ebx
  4005ea:       74 17                   je     400603 <func1+0x26>
  4005ec:       83 fb 01                cmp    $0x1,%ebx
  4005ef:       74 18                   je     400609 <func1+0x2c>
  4005f1:       8d 73 ff                lea    -0x1(%rbx),%esi
  4005f4:       44 89 e7                mov    %r12d,%edi
  4005f7:       83 eb 02                sub    $0x2,%ebx
  4005fa:       e8 de ff ff ff          callq  4005dd <func1>
  4005ff:       01 c5                   add    %eax,%ebp
  400601:       eb e5                   jmp    4005e8 <func1+0xb>
  400603:       41 bc 02 00 00 00       mov    $0x2,%r12d
  400609:       5b                      pop    %rbx
  40060a:       42 8d 44 25 00          lea    0x0(%rbp,%r12,1),%eax
  40060f:       5d                      pop    %rbp
  400610:       41 5c                   pop    %r12
  400612:       c3                      retq


0000000000400627 <func2>:
  400627:       31 c0                   xor    %eax,%eax
  400629:       83 ff 08                cmp    $0x8,%edi
  40062c:       7e 02                   jle    400630 <func2+0x9>
  40062e:       eb e3                   jmp    400613 <explode_bomb>
  400630:       83 fe 02                cmp    $0x2,%esi
  400633:       7e f9                   jle    40062e <func2+0x7>
  400635:       eb a6                   jmp    4005dd <func1>
```

Helpful gdb, run in the beginning of the program:

```
(gdb) x/25c 0x400703
0x400703:       37 '%'  100 'd' 32 ' '  37 '%'  100 'd' 0 '\000'        0 '\000'        0 '\000'
0x40070b:       0 '\000'        1 '\001'        27 '\033'       3 '\003'        59 ';'  80 'P'  0 '\000'        0 '\000'
0x400713:       0 '\000'        9 '\t'  0 '\000'        0 '\000'        0 '\000'        84 'T'  -3 '\375'       -1 '\377'
0x40071b:       -1 '\377'
```

7. Stack Overflow

a) Alright, you're a hacker now and you happen to have an inside source at a company that can provide you with assembly for the company's administrative code. The code is typically air tight but your source tells you a rookie programmer, Alex, was just hired and that their code has vulnerabilities and no OS protections (Alex is not the best at their job). Specifically, your source provides you with this snippet of assembly that is in charge of taking in a user password attempt as well as checking whether the user has an existing password or not (why these two tasks are in one function is beyond me, but Alex is a rookie).

```
00000000086012b4 <get_attempt_and_check_null_password>:    #1st argument is user's pw :)
  86012b4:       48 83 ec 38            sub    $0x3c,%rsp
  86012b8:       48 89 fd               mov    %rdi,%rbp
  86012bb:       48 89 e7               mov    %rsp,%rdi
  86012be:       e8 38 02 00 00         callq  501e2b <Gets>     #looks familiar
  86012c3:       48 89 ef               mov    %rbp,%rdi
  86012c6:       e8 38 ff ff ff         callq  702ee3 <Null_Pw_Check>  #Not important
  86012cb:       b8 01 00 00 00         mov    $0x1,%eax
  86012d1:       48 83 c4 38            add    $0x3c,%rsp
  4017fb:        c3                     retq
```

Your inside source also informs you of the existence of the following function

```
0000000004b023e4 <Print_String_Exit>:  #Prints whatever string is passed in
        ...                            #as 2nd arg and exits with value of
        ...                            #1st arg
```

Lastly, your inside source informs you that rsp will be set to ff ff ff ff 57 4e 3b 52 when entering the snippet in the first image (this is a really good inside source)

Assuming you acquired an individual's username information, what string will allow you to view their password(Encodings below for your reference and answer in the same form as lab 3)?

b) Sadly, before this code was able to be deployed Alex's supervisor saw it and angrily told Alex to fix it and explained to them that somebody could inject **executable code** onto the stack and cause problems, as well as judging Alex for creating such an impractical function. Alex, narrowing in on the term **executable code** (and not really listening to anything else that was said) simply turned on an OS feature that made the stack non-executable. Luckily for you, your

inside source is a good one and has provided you with the following farm in order to circumnavigate this issue.

```
00000000a576f3e2 <good_function>:
  ...
  a576f420:     b8 48 89 fc 90     some instr
  a576f425:     c3                 retq

0000000042013122 <bad_function>:
  ...
  42013122:     c6 48 89 d6 20 c0  some instr
  42013128:     c3                 retq

00000000c462a204 <that_function>:
  ...
  c462a24c:     c7 07 48 89 e6 90  some instr
  c462a212:     c3                 retq

00000000e2e2e2e2 <some_function>:
  ...
  e2e2e2ff:     c7 09 07 48 89 ca  some instr
  e2e2e305:     c3                 retq

00000000052ea100 <this_function>:
  ...
  52ea116:      8d 87 48 89 f9 90  some instr
  52ea11c:      c3                 retq

00000000ffffffff <f_function>:
  ...
  100000f0:     b8 48 89 fe 20 d9  some instr
  100000f6:     c3                 retq
```

What string will still allow you to view the user's password?

c) Alright at this point you are almost hit by a car and, having narrowly avoided death, decide that being a hacker is not the right thing to do. You turn a new leaf and want to help Alex before they are ultimately fired. **What methods can you offer to Alex in order to prevent people from abusing the code (although you are not allowed to tell Alex to get rid of the impractical combination of getting input and checking password existence cause that would hurt Alex's feelings)?**

(Might want to check attack lab specs for reminder on instructions for gadgets)

8. Code Optimization/Performance

Suppose we wish to write a function to evaluate a polynomial, where a polynomial of degree n is defined to have a set of coefficients a0, a1, a2, . . . , an. For a value x, we evaluate the polynomial by computing

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

This evaluation can be implemented by the following function, having as arguments an array of coefficients a, a value x, and the polynomial degree degree (the value n in Equation 5.2). In this function, we compute both the successive terms of the equation and the successive powers of x within a single loop:

```
1  double poly(double a[], double x, long degree)
2  {
3      long i;
4      double result = a[0];
5      double xpwr = x;  /* Equals x^i at start of loop */
6      for (i = 1; i <= degree; i++) {
7          result += a[i] * xpwr;
8          xpwr = x * xpwr;
9      }
10     return result;
11 }
```

A. For degree n, how many additions and how many multiplications does this code perform?

B. On our reference machine, with arithmetic operations having the latencies shown in the figure below, we measure the CPE for this function to be 5.00. Explain how this CPE arises based on the data dependencies formed between iterations due to the operations implementing lines 7–8 of the function (code inside the for loop).

	Integer			Floating point		
Operation	Latency	Issue	Capacity	Latency	Issue	Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3–30	3–30	1	3–15	3–15	1

9.

Question 1. Linking (4 pts)

Suppose main.c and lib.c are compiled and linked separately. Determine if the following combinations of source files would cause warnings or errors. If the code runs, what would get printed? If an answer is undefined, simply write “undefined” in the result box.

main.c	**lib.c**	**Result? (“compile error”, “linker error” or describe output)**

`int i;`
`void func();`
`int main() {`
`  printf("%d ",i);`
`  func();`
`  printf("%d",i);`
`}`
	`int i=2;`
	`void func() {`
	`  i=3;`
	`}`	

`int i;`
`void func();`
`int main() {`
`  printf("%d ",i);`
`  func();`
`  printf("%d",i);`
`}`
	`static int i=2;`
	`void func() {`
	`  i=3;`
	`}`	

`extern int i;`
`void func();`
`int main() {`
`  printf("%d ",i);`
`  func();`
`  printf("%d",i);`
`}`
	`int i=2;`
	`void func() {`
	`  i=3;`
	`}`	

`extern int i;`
`void func();`
`int main() {`
`  printf("%d ",i);`
`  func();`
`  printf("%d",i);`
`}`
	`static int i=2;`
	`void func() {`
	`  int i=3;`
	`}`	

```
extern int i=1;
void func();
int main() {
  printf("%d ",i);
  func();
  printf("%d",i);
}
```

```
static int i=2;
void func() {
  int i=3;
}
```