Worksheet 5

Name: ________________

UID: ________________

1. What do I need to do if I want to access a function through buffer overflow (what needs to be done to the stack)? The function's address is 0x500142.

The function Gets is similar to the standard library function gets—it reads a string from standard input (terminated by '\n' or end-of-file) and stores it (along with a null terminator) at the specified destination (such as a char array previously declared). Functions Gets() and gets() have no way to determine whether their destination buffers are large enough to store the string they read.

```
Dump of assembler code for function getbuf:
=> 0x0000000000601748 <+0>:       push   %rax
   0x000000000060174c <+4>:       sub    $0x40,%rsp
   0x000000000060174f <+7>:       mov    %rsp,%rdi
   0x0000000000601754 <+12>:      callq  0x40198a <Gets>
   0x0000000000601759 <+17>:      add    $0x40,%rsp
   0x000000000060175d <+21>:      pop    %rax
   0x000000000060175f <+23>:      retq
```

What do I need to do if I want some instructions to be executed before the function is accessed? (Assume the value of %rsp right before getbuf is called is 0xabcd0000)

2. What are some optimizations that can be made to the following function?

```
void cs33fun(char* Midterm, char* Grade, int* Final, int n) {

      for (int i = 0; i < (strlen(Midterm)); i++) {
            strcat(Grade, Midterm);

            for (int j = 0; j < n; j++)
                  for (int k = 0; k < i; k++)
                        Final[j] += strlen(Grade);
      }
}
```

3. Suppose we wish to write a procedure that computes the inner product of two vectors *u* and *v*. An abstract version of the function has a CPE of 14-18 with x86-64 for different types of integer and floating-point data. By doing the same sort of transformations we did to transform the abstract program *combine1* into the more efficient *combine4*, we get the following code:

```
/* Inner product. Accumulare in temporary */
void inner4(vec_ptr u, vec_ptr v, data_t* dest) {
      long i;
      long length = vec_length(u);
      data_t *udata = get_vec_start(u);
      data_t *vdata = get_vec_start(v);
      data_t sum = (data_t) 0;

      for (i = 0; i < length; i++) {
      sum = sum + udata[i] * vdata[i];
      }
      *dest = sum;
}
```

Our measurements show that this function has CPEs of 1.50 for integer data and 3.00 for floating-point data. For data type double, the x86-64 assembly code for the inner loop is as follows:

```
.L15:                                      ;loop:
  vmovsd  0(%rbp, %rcx, 8), %xmm1          ; Get udata[i]
  vmulsd  (%rax, %rcx, 8), %xmm1, %xmm1    ; Multiply by vdata[i]
  vaddsd  %xmm1, %xmm0, %xmm0              ; Add to sum
  addq      $1, %rcx                        ; Increment i
  cmpq      %rbx, %rcx                      ; Compare i:limit
  jne       .L15                            ; If !=, goto loop
```

Latency Table	Integer	Double
Arithmetic (except mul)	1	3
Multiply	2	4
Load Store	1	1

2. Suppose our memory can store 64 bytes of data, addressed between 6'b000000 and 6'b111111, and that we have stored a 4 x 4 array of ints in memory. This could be declared by:

```
int arr[4][4] = {
      {0,1,2,3},
      {4,5,6,7},
      {8,9,10,11},
      {12,13,14,15}
};
```

In addition, suppose we have a **fully associative cache** (only one set) with a single line of **16 bytes**.

Then, our addresses can be partitioned into **2 tag bits** along with **4 block offset bits**. For example, 6'b000000 would be formatted as:

tag: 00	block offset: 0000

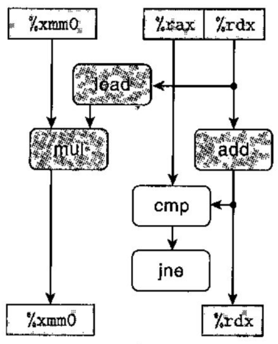
A. Now, say we accessed the array using the following loop, and our cache is cold (empty) to start. How many cache hits/misses would there be? (Think about how the array is stored in memory, and how data will be cached).

```
let sum = 0;
for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
            sum += arr[i][j];
      }
}
```

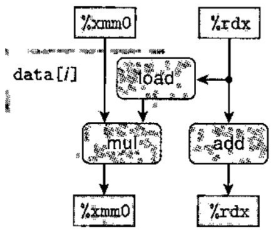
B. Again, assume our cache is empty to start. How many cache hits/misses would there be if we used this loop instead?

```
let sum = 0;
for (int j = 0; j < 4; j++) {
      for (int i = 0; i < 4; i++) {
            sum += arr[i][j];
      }
}
```

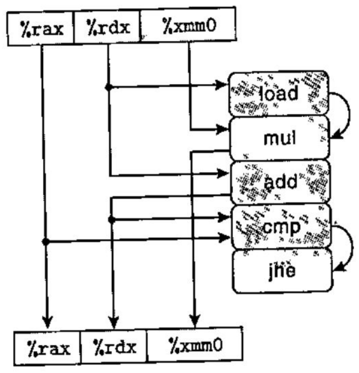
a. Diagram how this instruction sequence would be decoded into operations and show how the data dependencies between them would create a critical path of operations, in the style of the textbook's figures shown below



(a)



(b)



b. For data type double, what lower bound on the CPE is determined by the critical path?

c. Assuming similar instruction sequences for the integer code as well, what lower bound on the CPE is determined by the critical path for integer data?

d. Explain how the two floating-point versions can have CPEs of 3.00, even though the multiplication operation requires 5 clock cycles.