15-213: S20 Midterm Review Session

Kashish, Jeremy, Di
Agenda

- Review midterm problems
  - Cache
  - Assembly
  - Stack
  - Floats, Arrays, Structs (time permitting)
- Q&A for general midterm problems
Reminders

■ ONLY Conceptual office hours until Wednesday. If you need any help with midterm questions after today, please make a public Piazza post (and specify exactly which question!)

■ Cheat sheet: ONE 8½ x 11 in. sheet, both sides. Please use only English! Make a copy prior to midterm. No practice problems!

■ Lecture is still happening this week! Go learn things!
Problem 1: Cache

- Things to remember/put on a cheat sheet because please don’t try to memorize all of this:
  - Direct mapped vs. n-way associative vs. fully associative
  - Tag/Set/Block offset bits, how do they map depending on cache size?
  - LRU policies
Problem 1: Cache

A. Assume you have a cache of the following structure:
   a. 32-byte blocks
   b. 2 sets
   c. Direct-mapped
   d. 8-bit address space
   e. The cache is cold prior to access

B. What does the address decomposition look like?

   0 0 0 0 0 0 0 0
Problem 1: Cache

A. Assume you have a cache of the following structure:
   a. 32-byte blocks
   b. 2 sets
   c. Direct-mapped
   d. 8-bit address space
   e. The cache is cold prior to access

B. What does the address decomposition look like?

   0 0 0 0 0 0 0 0
## Problem 1: Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Set</th>
<th>Tag</th>
<th>H/M</th>
<th>Evict? Y/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x56</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x6D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x49</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x3A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Problem 1: Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Set</th>
<th>Tag</th>
<th>H/M</th>
<th>Evict? Y/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101 0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110 1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Problem 1: Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Set</th>
<th>Tag</th>
<th>H/M</th>
<th>Evict? Y/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101 0110</td>
<td>0</td>
<td>01</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>0110 1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Problem 1: Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Set</th>
<th>Tag</th>
<th>H/M</th>
<th>Evict? Y/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101 0110</td>
<td>0</td>
<td>01</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>0110 1101</td>
<td>1</td>
<td>01</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>0100 1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Problem 1: Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Set</th>
<th>Tag</th>
<th>H/M</th>
<th>Evict? Y/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101 0110</td>
<td>0</td>
<td>01</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>0110 1101</td>
<td>1</td>
<td>01</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>0100 1001</td>
<td>0</td>
<td>01</td>
<td>H</td>
<td>N</td>
</tr>
<tr>
<td>0011 1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Problem 1: Cache

<table>
<thead>
<tr>
<th>Address</th>
<th>Set</th>
<th>Tag</th>
<th>H/M</th>
<th>Evict? Y/N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101 0110</td>
<td>0</td>
<td>01</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>0110 1101</td>
<td>1</td>
<td>01</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>0100 1001</td>
<td>0</td>
<td>01</td>
<td>H</td>
<td>N</td>
</tr>
<tr>
<td>0011 1010</td>
<td>1</td>
<td>00</td>
<td>M</td>
<td>Y</td>
</tr>
</tbody>
</table>
Problem 1: Cache

A. Assume you have a cache of the following structure:
   a. 2-way associative
   b. 4 sets, 64-byte blocks

B. What does the address decomposition look like?

   ... 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Problem 1: Cache

A. Assume you have a cache of the following structure:
   a. 2-way associative
   b. 4 sets, 64-byte blocks

B. What does the address decomposition look like?

\[
\ldots 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
\]
Problem 1: Cache

B. Assume A and B are 128 ints and cache-aligned.
   a. What is the miss rate of pass 1?
   b. What is the miss rate of pass 2?

```c
int get_prod_and_copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```
Problem 1: Cache

B. Pass 1: Only going through 64 ints with step size 4. Each miss loads 16 ints into a cache line, giving us 3 more hits before loading into a new line.

```c
int get_prod_and_copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```
Problem 1: Cache

B. Pass 1: 25% miss

```c
int get_prod_and_copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```
Problem 1: Cache

B. Pass 2: Our cache is the same size as our working set! Due to cache alignment, we won’t evict anything from A, but still get a 1:3 miss:hit ratio for B.

```c
int get_prod_and_copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```
Problem 1: Cache

B. Pass 2: For every 4 loop iterations, we get all hits for accessing A and 1 miss for accessing B, which gives us $\frac{1}{8}$ miss.

```c
int get_prod_and_copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```
B. Pass 2: 12.5% miss

```c
int get_prod_and_copy(int *A, int *B) {
    int length = 64;
    int prod = 1;
    // pass 1
    for (int i = 0; i < length; i+=4) {
        prod*=A[i];
    }
    // pass 2
    for (int j = length-1; j > 0; j-=4) {
        A[j] = B[j];
    }
    return prod;
}
```
Problem 2: Assembly

- Typical questions asked
  - Given a function, look at assembly to fill in missing portions
  - Given assembly of a function, intuit the behavior of the program
  - (More rare) Compare different chunks of assembly, which one implements the function given?

- Important things to remember/put on your cheat sheet:
  - Memory Access formula: $D(R_b, R_i, S)$
  - Distinguish between mov/lea instructions
  - Callee/Caller save regs
  - Condition codes and corresponding eflags
Problem 2: Assembly

Consider the following x86-64 code (Recall that %cl is the low-order byte of %rcx):

```assembly
# On entry:
#   %rdi = x
#   %rsi = y
#   %rdx = z

4004f0 <mysterious>:
  4004f0:  mov   $0x0,%eax
  4004f5:  lea   -0x1(%rsi),%r9d
  4004f9:  jmp   400510 <mysterious+0x20>
  4004fb:  lea   0x2(%rdx),%r8d
  4004ff:  mov   %esi,%ecx
  400501:  shl   %cl,%r8d
  400504:  mov   %r9d,%ecx
  400507:  sar   %cl,%r8d
  40050a:  add   %r8d,%eax
  40050d:  add   $0x1,%edx
  400510:  cmp   %edx,%edi
  400512:  ja    4004fb <mysterious+0xb>
  400514:  retq
```
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = Z ; ; ){
        e = i + 2;
        e =
        e =
        d =
    }
    return
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z

4004f0 <mysterious>:
4004f0:    mov $0x0,%eax
4004f5:    lea -0x1(%rsi),%rdx
4004f9:    jmp 400510 <mysterious+0xb>
4004fb:    lea 0x2(%rdx),%rdx
4004ff:    mov %esi,%ecx
400501:    shl %cl,%rdx
400504:    mov %rdx,%ecx
400507:    sar %cl,%rdx
40050a:    add %rdx,%eax
40050d:    add $0x1,%edx
400510:    cmp %edx,%edi
400512:    ja 4004fb <mysterious+0xb>
400514:    retq
```
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = z; ; ; e = i + 2; e = ; ; d = ; }
    return ;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z

4004f0 <mysterious>:
    4004f0:  mov  $0x0,%eax
    4004f5:  lea  -0x1(%rsi),%r9d
    4004f9:  jmp  400510 <mysterious+0x20>
    4004fb:  lea  0x2(%rdx),%r8d
    4004ff:  mov  %esi,%ecx
    400501:  shl  %cl,%r8d
    400504:  mov  %r9d,%ecx
    400507:  sar  %cl,%r8d
    40050a:  add  %r8d,%eax
    40050d:  add  $0x1,%edx
    400510:  cmp  %edx,%edi
    400512:  ja   4004fb <mysterious+0xb>
    400514:  retq
```

e = %r8d
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z) {
    unsigned i;
    int d = 0;
    int e;
    for (i = __; __; __) {
        e = i + 2;
        e = __;
        e = __;
        d = __;
    }
    return __;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z

4004f0 <mysterious>:  
4004f0:    mov    $0x0,%eax
4004f5:    lea    -0x1(%rsi),%rdx
4004f9:    jmp    400510 <mysterious+0x20>
4004fb:    lea    0x2(%rdx),%rdx
4004ff:    mov    %esi,%ecx
400501:    shl    %cl,%rdx
400504:    mov    %rdx,%ecx
400507:    sar    %cl,%rdx
40050a:    add    %rdx,%eax
40050d:    add    $0x1,%edx
400510:    cmp    %edx,%edi
400512:    ja     4004fb <mysterious+0xb>
400514:    retq
```

Loop end: add 1, compare, iterate
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = 0; x > i; i++) {
        e = i + 2;
        e = 0;
        d = 0;
    }
    return 0;
}
```

```
# On entry:
#  %rdi = x
#  %rsi = y
#  %rdx = z

4004f0 <mysterious>:  
4004f0:   mov    $0x0,%eax         
4004f5:   lea    -0x1(%rsi),%rdx   
4004f9:   jmp    400510 <mysterious+0x20> 
4004fb:   lea    0x2(%rdx),%rdx    
4004ff:   mov    %esi,%ecx        
400501:   shl    %cl,%rdx        
400504:   mov    %rdx,%ecx        
400507:   sar    %cl,%rdx        
40050a:   add    %rdx,%eax       
40050d:   add    $0x1,%edx       
400510:   cmp    %edx,%edi       
400512:   ja     4004fb <mysterious+0x1b> 
400514:   retq                
```

cmp %edx, %edi  =>  (edi - edx > 0), same as x > i
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = Z; x > i; i++){
        e = i + 2;
        e = [___];
        e = [___];
        d = [___];
    }
    return [___];
}
```

```
# On entry:
#  %rdi = x
#  %rsi = y
#  %rdx = z

4004f0 <mysterious>:
4004f0:  mov   $0x0,%eax
4004f5:  lea   -0x1(%rsi),%r9d
4004f9:  jmp   400510 <mysterious+0x20>
4004fb:  lea   0x2(%rdx),%r8d
4004ff:  mov   %esi,%ecx
400501:  shl   %cl,%r8d
400504:  mov   %rd9,%ecx
400507:  sar   %cl,%r8d
40050a:  add   %r8d,%eax
40050d:  add   $0x1,%edx
400510:  cmp   %edx,%edi
400512:  ja    4004fb <mysterious+0xb>
400514:  retq
```

We know that e = %r8d...
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z) {
    unsigned i;
    int d = 0;
    int e;
    for(i = z; x > i; i++) {
        e = i + 2;
        e = e << y;
        e = e;
        d = e;
    }
    return e;
}
```

Where did %cl come from?
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = z; x > i; i++)
        e = i + 2;
    e = e << y;
    e = 0;
    d = 0;
}
return 0;
}
```

Again, e = %r8d...
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = z; x > i; i++) {
        e = i + 2;
        e = e << y;
        e = e >> (y - 1);
        d =
    }
    return
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z

4004f0 <mysterious>:
4004f0: mov 0x00, %eax
4004f5: lea -0x1(%rsi), %rdx
4004f9: jmp 400510 <mysterious+0x20>
4004fb: lea 0x2(%rdx), %rdx
4004ff: mov %esi, %ecx
400501: shl %cl, %rdx
400504: mov %rdx, %ecx
400507: sar %cl, %rdx
40050a: add %rdx, %eax
40050d: add $0x1, %edx
400510: cmp %edx, %edi
400512: ja 4004fb <mysterious+0xb>
400514: retq
```
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = 0; x > i; i++) {
        e = i + 2;
        e = e << y;
        e = e >> (y - 1);
        d = 0;
    }
    return d;
}
```

What’s left?
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = z; x > i; i++)
        e = i + 2;
    e = e << y;
    e = e >> (y - 1);
    d = e + d;
}
return
```
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = __; x > i; i++)
        e = __;
    e = __;
    e = __;
    d = __
    return __;
}
```

```assembly
# On entry:
#  %rdi = x
#  %rsi = y
#  %rdx = z

4004f0 <mysterious>:
4004f0:   mov    $0x0,%eax
4004f5:   lea    -0x1(%rsi),%r9d
4004f9:   jmp    400510 <mysterious+0x20>
4004fb:   lea    0x2(%rdx),%r8d
4004ff:   mov    %esi,%ecx
400501:   shl    %cl,%r8d
400504:   mov    %r9d,%ecx
400507:   sar    %cl,%r8d
40050a:   add    %r8d,%eax
40050d:   add    $0x1,%edx
400510:   cmp    %edx,%edi
400512:   ja     4004fb <mysterious+0xb>
400514:   retq
```
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z){
    unsigned i;
    int d = 0;
    int e;
    for(i = z; x > i; i++) {
        e = i + 2;
        e = e << y;
        e = e >> (y - 1);
        d = e + d;
    }
    return d;
}
```

```
# On entry:
# %rdi = x
# %rsi = y
# %rdx = z

4004f0 <mysterious>:
4004f0: mov $0x0,%eax
4004f5: lea -0x1(%rsi),%rdx
4004f9: jmp 400510 <mysterious+0x20>
4004fb: lea 0x2(%rdx),%rdx
4004ff: mov %esi,%ecx
400501: shl %cl,%rdx
400504: mov %rdx,%ecx
400507: sar %cl,%rdx
40050a: add %rdx,%eax
40050d: add $0x1,%edx
400510: cmp %edx,%edi
400512: ja 4004fb <mysterious+0xb>
400514: retq
```
Problem 2: Assembly

1) Please fill in the corresponding blanks below to make the C source equivalent to the assembly.

```c
int mysterious(int x, int y, int z) {
    unsigned i;
    int d = 0;
    int e;
    for (i = z; x > i; i++) {
        e = i + 2;
        e = e << y;
        e = e >> (y - 1);
        d = e + d;
    }
    return d;
}
```

```
# On entry:
#   %rdi = x
#   %rsi = y
#   %rdx = z

4004f0 <mysterious>:
    4004f0:   mov   $0x0,%eax
    4004f5:   lea   -0x1(%rsi),%rdx
    4004f9:   jmp   400510 <mysterious+0x20>
    4004fb:   lea   0x2(%rdx),%r8d
    4004ff:   mov   %esi,%ecx
    400501:   shl   %cl,%rdx
    400504:   mov   %rdx,%ecx
    400507:   sar   %cl,%rdx
    40050a:   add   %r8d,%eax
    40050d:   add   $0x1,%edx
    400510:   cmp   %edx,%edi
    400512:   ja    4004fb <mysterious+0xb>
    400514:   retq
```
Problem 3: Stack

- Important things to remember:
  - Stack grows towards lower addresses
  - \%rsp = stack pointer, always point to “top” of stack
  - **Push and pop, call and ret**
  - Stack frames: how they are allocated and freed
  - Which registers used for arguments? Return values?
  - Little endianness

- ALWAYS helpful to draw a stack diagram!!
- Stack questions are like Assembly questions on steroids
Problem 3: Stack

Consider the following code:

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

Hints:
- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating `\0` character) to address `dst`.
- Keep endianness in mind!
- Table of hex values of characters in “midtermexam”

Assumptions:
- `%rsp = 0x800100` just before `caller()` calls `foo()`
- `.LC0` is at address 0x400300
Problem 3: Stack

Consider the following code:

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```assembly
foo:
    subq $24, %rsp
    cmpl $0xdeadbeef, %esi
    je .L2
    movl $0xdeadbeef, %esi
    call foo
.L2:
    movq %rdi, %rsi
    movq %rsp, %rdi
    call strcpy
.L1:
    addq $24, %rsp
    ret

caller:
    subq $8, %rsp
    movl $86547, %esi
    movl $.LC0, %edi
    call foo
    addq $8, %rsp
    ret
```

Hints:
- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '0' character) to address `dst`.
- Keep endianness in mind!
- Table of hex values of characters in "midtermexam"

Assumptions:
- `%rsp = 0x800100` just before `caller()` calls `foo()`.
- `.LC0` is at address `0x400300`. 
Problem 3: Stack

Question 1: What is the hex value of %rsp just before strcpy() is called for the first time in foo()?

Hints:
- Step through the program instruction by instruction from start to end
- Draw a stack diagram!!!
- Keep track of registers too
Problem 3: Stack

Question 1: What is the hex value of %rsp just before strcpy() is called for the first time in foo()?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
void caller() {
    foo("midtermexam", 0x15213);
}
```

### Solution:

<table>
<thead>
<tr>
<th></th>
<th>%rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x800100</td>
</tr>
<tr>
<td>%rdi</td>
<td>.LC0</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x15213</td>
</tr>
</tbody>
</table>

**foo:**

```
subq $24, %rsp
compl $0xdeadbeef, %esi
ej .L2
movl $0xdeadbeef, %esi
call foo
jmp .L1
.L2:
movq %rdi, %rsi
movq %rsi, %rdi
End
call strcpy
```

**caller:**

```
subq $8, %rsp
movl $86547, %esi
movl $.LC0, %edi
call foo
addq $8, %rsp
ret
```

**Arrow is instruction that will execute NEXT:**

0x800100
0x8000f8
0x8000f0
0x8000e8
0x8000e0
0x8000d8
0x8000d0
0x8000c8
0x8000c0
0x8000b8
Problem 3: Stack

Question 1: What is the hex value of %rsp just before strcpy() is called for the first time in foo()? 

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

```
<table>
<thead>
<tr>
<th></th>
<th>%rsp</th>
<th>0x8000f8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>.LC0</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>0x15213</td>
<td></td>
</tr>
</tbody>
</table>
```

Subq $24, %rsp
Cmpl $0xdeadbeef, %esi
J e .L2
Movl $0xdeadbeef, %esi
Call foo
J mp .L1
.L2:
Movq %rdi, %rsi
Movq %rsi, %edi
End
Call strcpy
Addq $24, %rsp
Ret

.LC0 = 0x400300
.String "midtermexam"
Question 1: What is the hex value of \texttt{%rsp} just \textbf{before} \texttt{strcpy()} is called for the first time in \texttt{foo()}?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<table>
<thead>
<tr>
<th>\textbf{Foo}</th>
<th>\textbf{Caller}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{foo}</td>
<td>\textbf{caller}</td>
</tr>
<tr>
<td>\textbf{L1:}</td>
<td>addq $24, %rsp</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
<tr>
<td>\textbf{L2:}</td>
<td>cmpl $0xdeadbeef, %esi</td>
</tr>
<tr>
<td></td>
<td>je .L2</td>
</tr>
<tr>
<td></td>
<td>movl $0xdeadbeef, %esi</td>
</tr>
<tr>
<td></td>
<td>call foo</td>
</tr>
<tr>
<td></td>
<td>jmp .L1</td>
</tr>
<tr>
<td>\textbf{.L2:}</td>
<td>movq %rdi, %rsi</td>
</tr>
<tr>
<td></td>
<td>movq %rsp, %rdi</td>
</tr>
<tr>
<td></td>
<td>call strcpy</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\textbf{RSP}</th>
<th>\textbf{RDI}</th>
<th>\textbf{RSI}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800100</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret address for \texttt{foo()}</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e8</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0x8000d8</td>
<td>0x8000d0</td>
<td></td>
</tr>
<tr>
<td>0x8000c8</td>
<td>0x8000c0</td>
<td></td>
</tr>
<tr>
<td>0x8000b8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Hint:** \$24 \text{ in decimal } = \texttt{0x18}
**Problem 3: Stack**

**Question 1:** What is the hex value of `%rsp` just **before** `strcpy()` is called for the first time in `foo()`?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```assembly
foo:  subq  $24, %rsp  ; subq  $8, %rsp
        cmpl  $0xdeadbeef, %esi  ; movl  $86547, %esi
        je    .L2  ; movl  $.LC0, %edi
        movl  $0xdeadbeef, %esi  ; call  foo
        call  foo
        jmp   .L1  ; addq  $8, %rsp
        .L1:    ret
.
.L2:  movq  %rdi, %rsi
        movq  %rsp, %rdi
        .End  call  strcpy
.
.LC0:  .section .rodata
        .string "midtermexam"
        .string "midtermexam"
```

<table>
<thead>
<tr>
<th>%rsp</th>
<th>0x8000e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>.LC0</td>
</tr>
<tr>
<td>%rsi</td>
<td>0xdeadbeef</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x800100</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000f8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret address for foo()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000f0</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000e8</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000d8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000e0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000c8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000c0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000b8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000b8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000e0</td>
</tr>
</tbody>
</table>
Problem 3: Stack

Question 1: What is the hex value of `%rsp` just **before** `strcpy()` is called for the first time in `foo()`?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

<table>
<thead>
<tr>
<th>%rsp</th>
<th>0x8000d8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>.LC0</td>
</tr>
<tr>
<td>%rsi</td>
<td>0xdeadbeef</td>
</tr>
</tbody>
</table>

### Assembly code

**foo:**
- `subq $24, %rsp`
- `cmpl $0xdeadbeef, %esi`
- `je .L2`
- `movl $0xdeadbeef, %esi`
- `call foo`
- `jmp .L1`
- `.L2:
  - `movq %rdi, %rsi`
  - `movq %rsp, %rdi`
- `End` `call strcpy`
- `addq $24, %rsp`
- `ret`

**caller:**
- `subq $8, %rsp`
- `movl $86547, %esi`
- `movl $.LC0, %edi`
- `call foo`
- `addq $8, %rsp`
- `ret`
Problem 3: Stack

Question 1: What is the hex value of %rsp just before strcpy() is called for the first time in foo()?

void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}

void caller() {
    foo("midtermexam", 0x15213);
}

<table>
<thead>
<tr>
<th>%rsp</th>
<th>0x8000c0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>$.LC0</td>
</tr>
<tr>
<td>%rsi</td>
<td>0xdeadbeef</td>
</tr>
</tbody>
</table>

| 0x800100 | ? |
| 0x8000f8 | ret address for foo() |
| 0x8000f0 | ? |
| 0x8000e8 | ? |
| 0x8000e0 | ? |
| 0x8000d8 | ret address for foo() |
| 0x8000d0 | ? |
| 0x8000c8 | ? |
| 0x8000c0 | ? |
| 0x8000b8 | |
Problem 3: Stack

Question 1: What is the hex value of `%rsp` just **before** `strcpy()` is called for the first time in `foo()`?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```assembly
foo:
    subq $24, %rsp
    cmpl $0xdeadbeef, %esi
    je .L2
    movl $%edx, %esi
    call foo
    jmp .L1
.L2:
    movq %rdi, %rsi
    movq %rsp, %rdi
    call strcpy
    .L1:
    addq $24, %rsp
    ret
```

```assembly
caller:
    subq $8, %rsp
    movl $86547, %esi
    movl $.LC0, %edi
    call foo
    addq $8, %rsp
    ret
```

---

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800100</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret address for foo()</td>
<td></td>
</tr>
<tr>
<td>0x8000f0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000e8</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000e0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000d8</td>
<td>ret address for foo()</td>
<td></td>
</tr>
<tr>
<td>0x8000d0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000c8</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000c0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000b8</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
Problem 3: Stack

Question 1: What is the hex value of %rsp just before strcpy() is called for the first time in foo()?

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

Answer!

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800100</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret address for foo()</td>
</tr>
<tr>
<td>0x8000f0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e8</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000d8</td>
<td>ret address for foo()</td>
</tr>
<tr>
<td>0x8000d0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000c8</td>
<td>?</td>
</tr>
<tr>
<td>0x8000c0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000b8</td>
<td>?</td>
</tr>
</tbody>
</table>

Table:

<table>
<thead>
<tr>
<th>%rsp</th>
<th>0x8000c0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>0x8000c0</td>
</tr>
<tr>
<td>%rsi</td>
<td>.LCO</td>
</tr>
</tbody>
</table>
Problem 3: Stack

Question 2: What is the hex value of `buf[0]` when `strcpy()` returns?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy(char* buf, str);
}
```

- `%rsp` 0x8000c0
- `%rdi` 0x8000c0
- `%rsi` .LC0

```
foo:
    subq $24, %rsp
    cmpl $0xdeadbeef, %esi
    je .L2
    movl $0xdeadbeef, %esi
    call foo
    jmp .L1
.L2:
    movq %rdi, %rsi
    movq %rsi, %rdi
    call strcpy
    .section .rodata.str1.1,"aMS",@progbits,1
    .LC0 = 0x4003C0
    .string "midtermexam"
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000100</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret addr for foo()</td>
<td></td>
</tr>
<tr>
<td>0x8000f0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000e8</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000e0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000d8</td>
<td>ret addr for foo()</td>
<td></td>
</tr>
<tr>
<td>0x8000d0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000c8</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000c0</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>0x8000b8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3: Stack

Question 2: What is the hex value of \texttt{buf[0]} when \texttt{strcpy()} returns?

void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800100</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret address for \texttt{foo()}</td>
</tr>
<tr>
<td>0x8000f0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e8</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000d8</td>
<td>ret address for \texttt{foo()}</td>
</tr>
<tr>
<td>0x8000d0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000c8</td>
<td></td>
</tr>
<tr>
<td>0x8000c0</td>
<td></td>
</tr>
</tbody>
</table>

- \texttt{\%rsp} = 0x8000c0
- \texttt{\%rdi} = 0x8000c0
- \texttt{\%rsi} = \texttt{.LC0}

```
foo:
    subq $24, \%rsp
    cmpl $0xdeadbeef, \%esi
    je .L2
    movl $0xdeadbeef, \%esi
    call foo
    jmp .L1

.L2:
    movq \%rdi, \%rsi
    movq \%rsp, \%rdi
    call strcpy

.L1:
    addq $24, \%rsp
    ret
```

\texttt{\textcolor{red}{\%rdi}} = 0x8000c0
\texttt{\textcolor{red}{\%rsi}} = \texttt{.LC0}

```
c7
  c2
c1
c0
```

\texttt{.LC0: = 0x400300}
\texttt{.string "midtermexam"}

\texttt{.section .rodata.s}
Problem 3: Stack

Question 2: What is the hex value of buf[0] when strcpy() returns?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    memcpy((char*) buf, str, a);
}
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800100</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret address for foo()</td>
</tr>
<tr>
<td>0x8000f0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e8</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000d8</td>
<td>ret address for foo()</td>
</tr>
<tr>
<td>0x8000d0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000c8</td>
<td>?</td>
</tr>
<tr>
<td>0x8000b8</td>
<td>\0 \m \a \x</td>
</tr>
</tbody>
</table>

```
.L2:
    movq %rdi, %rsi
call strlen
jmph .L1
```
Problem 3: Stack

Question 2: What is the hex value of `buf[0]` when `strcpy()` returns?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
    }
    strcpy((char*) buf, str);
}
```

The relevant instructions are:
- `%rsp` at 0x8000c0
- `%rdi` at 0x8000c0
- `%rsi` at `.LC0`
- `%rdx` at 0x8000f0
- `%rsi` at 0x8000f8
- `%rsi` at 0x8000e8
- `%rsi` at 0x8000e0
- `%rsi` at 0x8000d8
- `%rsi` at 0x8000d0
- `%rsi` at 0x8000c8
- `%rsi` at 0x8000c0

The stack frame for `foo()` is as follows:
- `%rdi` at 0x8000c0
- `%rsi` at `.LC0`
- `%rsi` at 0x8000f0
- `%rsi` at 0x8000f8
- `%rsi` at 0x8000e8
- `%rsi` at 0x8000e0
- `%rsi` at 0x8000d8
- `%rsi` at 0x8000d0
- `%rsi` at 0x8000c8
- `%rsi` at 0x8000c0

The hex value of `buf[0]` is `0x8000c0`.
Problem 3: Stack

\[ \text{buf[0]} = \text{‘t’ ‘d’ ‘i’ ‘m’} = 74 \ 64 \ 69 \ 6d \]

(as int) = \text{0x7464696d}
Problem 3: Stack

Question 3: What is the hex value of buf[1] when strcpy() returns?

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy(char*) buf, str ;
}
```

```
subq $24, %rsp
 cmpl $0xdeadbeef, %esi
  je .L2
 movl $0xdeadbeef, %esi
 call foo
 jmp .L1
.L2:
 movq %rdi, %rsi
 movq %rsp, %rdi
 call strcpy
 .L1:
 addq $24, %rsp
 ret
```

```
void caller() {
    foo("midtermexam", 0x15213);
}
```

```
subq $8, %rsp
 movl $86547, %esi
 movl $.LC0, %edi
 call foo
 addq $8, %rsp
 ret
```

```
%rdi 0x8000c0
%rsi .LC0
```

```
0x800100
0x8000f8
0x8000f0
0x8000e8
0x8000e0
0x8000d8
0x8000d0
0x8000c8
0x8000c0
0x8000b8
```

<table>
<thead>
<tr>
<th>Ret address for foo()</th>
<th>0x8000f8</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ret address for foo()</th>
<th>0x8000e8</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ret address for foo()</th>
<th>0x8000d8</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ret address for foo()</th>
<th>0x8000d0</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ret address for foo()</th>
<th>0x8000c8</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000c0</th>
<th>'e'</th>
<th>'m'</th>
<th>'r'</th>
<th>'e'</th>
<th>'t'</th>
<th>'d'</th>
<th>'i'</th>
<th>'m'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000b8</td>
<td>buf[1]</td>
<td>c4</td>
<td></td>
<td>buf[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 3: Stack

\[ \text{buf}[1] = \text{'e'} \text{'m'} \text{'r'} \text{'e'} \]

\[ = 65 \text{ } 6d \text{ } 72 \text{ } 65 \]

(as int) = 0x656d7265

<table>
<thead>
<tr>
<th>Char</th>
<th>Hex</th>
<th>Char</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>61</td>
<td>m</td>
<td>6d</td>
</tr>
<tr>
<td>d</td>
<td>64</td>
<td>r</td>
<td>72</td>
</tr>
<tr>
<td>e</td>
<td>65</td>
<td>t</td>
<td>74</td>
</tr>
<tr>
<td>i</td>
<td>69</td>
<td>x</td>
<td>78</td>
</tr>
</tbody>
</table>

\begin{array}{|c|c|c|c|}
\hline
0x800100 & ? & \\
\hline
0x8000f8 & ret address for \text{foo()} & \\
\hline
0x8000f0 & ? & \\
\hline
0x8000e8 & ? & \\
\hline
0x8000e0 & ? & \\
\hline
0x8000d8 & ret address for \text{foo()} & \\
\hline
0x8000d0 & ? & \\
\hline
0x8000c8 & ? & ? & ? & ? & 0' & 'm' & 'a' & 'x' & \\
\hline
0x8000c0 & \text{'e'} \text{'m'} \text{'r'} \text{'e'} & t & 'd' & 'i' & 'm' & \\
\hline
0x8000b8 & \text{buf}[1] & \\
\hline
\end{array}
Problem 3: Stack

Question 4: What is the hex value of %rdi at the point where foo() is called recursively in the successful arm of the if statement?

This is before the recursive call to foo()
Problem 3: Stack

Question 4: What is the hex value of %rdi at the point where foo() is called recursively in the successful arm of the if statement?

- This is before the recursive call to foo()
- Going backwards, %rdi was loaded in caller()
- %rdi = $.LC0 = 0x400300 (based on hint)

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```

```
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xdeadbeef) {
        foo(str, 0xdeadbeef);
        return;
    }
    strcpy((char*) buf, str);
}

void caller() {
    foo("midtermexam", 0x15213);
}
```
Question 5: What part(s) of the stack will be corrupted by invoking `caller()`? Check all that apply.

- return address from `foo()` to `caller()`
- return address from the recursive call to `foo()`
- `strcpy()`’s return address
- there will be no corruption
Problem 3: Stack

Question 5: What part(s) of the stack will be corrupted by invoking `caller()`? Check all that apply.

- return address from `foo()` to `caller()`
- return address from the recursive call to `foo()`
- `strcpy()`’s return address
- there will be no corruption

The `strcpy` didn’t overwite any return addresses, so there was no corruption!
Bonus Coverage: Float

- Things to remember/ put on your cheat sheet:
  - Floating point representation \((-1)^s M 2^E\)
  - Values of M in normalized vs denormalized
  - Difference between normalized, denormalized and special floating point numbers
  - Rounding
  - Bit values of smallest and largest normalized and denormalized numbers
Bonus Coverage: Float

A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) 31/8
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) \( \frac{31}{8} \)

Step 1: Convert the fraction into the form \((-1)^s M 2^E\)
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) $31/8$

Step 1: Convert the fraction into the form $(-1)^s M 2^E$

$s = 0$

$M = 31/16$ (M should be in the range $[1.0, 2.0)$ for normalised numbers)

$E = 1$
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) $31/8$

Step 2: Convert $M$ into binary and find value of exp $s = 0$

$M = 31/16$ (M should be in the range $[1.0, 2.0)$ for normalised numbers)

$E = 1$
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) \(\frac{31}{8}\)

Step 2: Convert \(M\) into binary and find value of exponent:

\[
M = \frac{31}{16} = 1.1111
\]

\[
bias = 2^{k-1} - 1 \quad (k \text{ is the number of exponent bits}) = 1
\]

\[
E = 1 \Rightarrow \text{exponent} = 1 + \text{bias} = 2
\]
Bonus Coverage: Float

A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) \( \frac{31}{8} \)

Step 3: Find the fraction bits and exponent bits

\( s = 0 \)

\[ M = 1.1111 \Rightarrow \text{fraction bits are 1111} \]

exponent bits are 10
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) 31/8

Step 4: Take care of rounding issues
Current number is 0 10 111 1 <= excess bit
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) $31/8$

Step 4: Take care of rounding issues
Current number is 0 10 111 $1 \leq$ excess bit

Guard bit = 1
Round bit = 1

Round up! (add 1 to the fraction bits)
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) $31/8$

Step 4: Take care of rounding issues
Current number is $0 10 111$ $1 <= excess$ bit

Adding 1 overflows the floating bits, so we increment the exponent bits by 1 and set the fraction bits to 0.
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) $31/8$

Step 4: Take care of rounding issues

Result is $0\ 11\ 000 \leq \text{Infinity!}$
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

b) \(-7/8\)
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

b) $-\frac{7}{8}$

Step 1: Convert the fraction into the form $(-1)^s M 2^E$

$s = 1$

$M = \frac{7}{4}$

$E = -1$
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

b) -7/8

Step 2: Convert M into binary and find value of exp

\[ M = 7/4 \Rightarrow 1.11 \]

Bias = \(2^{k-1} - 1\) (\(k\) is the number of exponent bits) = 1

\[ E = -1 \Rightarrow \text{exponent} = -1 + \text{bias} = 0 \]
Bonus Coverage: Float

A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

b) \(-\frac{7}{8}\)

Step 2: Convert M into binary and find value of exp
s = 1

\[
M = \frac{7}{4} \Rightarrow 1.11 <= \text{(We assumed M was in the range \([1.0, 2.0]\). Need to update the value of M)}
\]

bias = \(2^{k-1} - 1\) (k is the number of exponent bits) = 1

E = -1 \Rightarrow \text{exponent} = -1 + \text{bias} = 0 <= \text{denormalized}
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

b) \(-7/8\)

Step 2: Convert \(M\) into binary and find value of \(exp\)

\[ M = 7/8 \Rightarrow 0.111 \leq M \text{ should be in the range } [0.0, 1.0) \text{ for denormalized numbers so we divide it by } 2 \]

\[ exp = 0 \]
A. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

b) \(-7/8\)

Step 3: Find the fraction bits and exponent bits

\[ s = 1 \]

\[ M = 0.111 \Rightarrow \text{Fraction bits} = 111 \]

\[ \text{exp bits} = 00 \]

Result = 1 00 111
Bonus Coverage: Float

B. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

b) 0 10 101
**Bonus Coverage: Float**

B. Consider a floating point representation with 1 sign bit, 2 exponent bits and 3 fraction bits. Convert the following numbers into their floating point representation.

a) \( 0 \ 10 \ 101 \)

\( s = 0 \)

\( \text{exp} = 2 \Rightarrow E = \text{exp} - \text{bias} = 1 \) (normalized)

\( M = 1.101 \) (between 1 and 2 since it is normalised)

Result = \( 2 \times 1.101 = 2 \times (13/8) = 13/4 \)
Bonuc Coverage: Arrays

IMPORTANT POINTS + TIPS:

- Remember your indexing rules! They’ll take you 95% of the way there.
- Be careful about addressing (&) vs. dereferencing (*)
- You may be asked to look at assembly!
- Feel free to put lecture/recitation/textbook examples in your cheatsheet.
**Bonus Coverage: Arrays**

**Good toy examples** (for your cheatsheet and/or big brain):

```c
int val[5];
```

- A can be used as the pointer to the first array element: `A[0]`

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val</code></td>
<td></td>
</tr>
<tr>
<td><code>val[2]</code></td>
<td></td>
</tr>
<tr>
<td><code>*(val + 2)</code></td>
<td></td>
</tr>
<tr>
<td><code>&amp;val[2]</code></td>
<td></td>
</tr>
<tr>
<td><code>val + 2</code></td>
<td></td>
</tr>
<tr>
<td><code>val + i</code></td>
<td></td>
</tr>
</tbody>
</table>
Bonus Coverage: Arrays

**Good toy examples (for your cheatsheet and/or big brain):**

```c
int val[5];
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>int *</td>
</tr>
<tr>
<td>val[2]</td>
<td>int</td>
</tr>
<tr>
<td>*(val + 2)</td>
<td>int</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
</tr>
<tr>
<td>val + 2</td>
<td>int *</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
</tr>
</tbody>
</table>

● A can be used as the pointer to the first array element: `A[0]`
### Bonus Coverage: Arrays

**Good toy examples (for your cheatsheet and/or big brain):**

```c
int val[5];
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val</code></td>
<td><code>int *</code></td>
</tr>
<tr>
<td><code>val[2]</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>*(val + 2)</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>&amp;val[2]</code></td>
<td><code>int *</code></td>
</tr>
<tr>
<td><code>val + 2</code></td>
<td><code>int *</code></td>
</tr>
<tr>
<td><code>val + i</code></td>
<td><code>int *</code></td>
</tr>
</tbody>
</table>

- `A` can be used as the pointer to the first array element: `A[0]`

---

Accessing methods:
- `val[index]`
- `*(val + index)`
Bonus Coverage: Arrays

Good toy examples (for your cheatsheet and/or big brain):

- A can be used as the pointer to the first array element: A[0]

<table>
<thead>
<tr>
<th></th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val[2]</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>*(val + 2)</td>
<td>int</td>
<td>2</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>x + 8</td>
</tr>
<tr>
<td>val + 2</td>
<td>int *</td>
<td>x + 8</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x + (4 * i)</td>
</tr>
</tbody>
</table>

Addressing methods:
- &val[index]
- val + index

Accessing methods:
- val[index]
- *(val + index)
Bonus Coverage: Arrays

**Nested indexing rules** (for your cheatsheet and/or big brain):

- Declared: `T A[R][C]`
- Contiguous chunk of space (think of multiple arrays lined up next to each other)

```c
int A[R][C];
```

![Diagram](image)
Nested indexing rules (for your cheatsheet and/or big brain):

- Arranged in ROW-MAJOR ORDER - think of row vectors
- A[i] is an array of C elements ("columns") of type T

```c
int A[R][C];
```

A+(i*C*4) + (j*4)
Bonus Coverage: Arrays

Nested indexing rules (for your cheatsheet and/or big brain):

\[ A[i][j] \text{ is element of type } T, \text{ which requires } K \text{ bytes} \]

Address \[ A + i \times (C \times K) + j \times K \]

\[ = A + (i \times C + j) \times K \]
Bonus Coverage: Arrays

Consider accessing elements of A....

<table>
<thead>
<tr>
<th></th>
<th>Compiles</th>
<th>Bad Deref?</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int A1[3][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A5[3]) [5]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bonus Coverage: Arrays

Consider accessing elements of $A$....

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>N</td>
<td>3\cdot5\cdot4 = 60</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int *A2[3][5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int (*A3)[3][5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int *(A4[3][5])</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int (*A5[3])[5]</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
</tr>
</tbody>
</table>
### Bonus Coverage: Arrays

Consider accessing elements of \textbf{A}....

<table>
<thead>
<tr>
<th></th>
<th>Compiles</th>
<th>Bad Deref?</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int A1[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3\times5\times(4) = 60</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3\times5\times(8) = 120</td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A5[3])[5]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Bonus Coverage: Arrays

Consider accessing elements of \( A \)....

<table>
<thead>
<tr>
<th>Access Pattern</th>
<th>Compiles</th>
<th>Bad Deref?</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int A1[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(4) = 60</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(8) = 120</td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td>Y</td>
<td>N</td>
<td>1*8 = 8</td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A5[3])[5]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Bonus Coverage: Arrays

Consider accessing elements of A....

<table>
<thead>
<tr>
<th>Description</th>
<th>Compiles</th>
<th>Bad Deref?</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int A1[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(4) = 60</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(8) = 120</td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td>Y</td>
<td>N</td>
<td>1*8 = 8</td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(8) = 120</td>
</tr>
<tr>
<td>int (*A5[3]) [5]</td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

A4 is a pointer to a 3x5 (int *) element array.
## Bonus Coverage: Arrays

Consider accessing elements of A....

<table>
<thead>
<tr>
<th>Type</th>
<th>Compiles</th>
<th>Bad Deref?</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int A1[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(4) = 60</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(8) = 120</td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td>Y</td>
<td>N</td>
<td>1*8 = 8</td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td>Y</td>
<td>N</td>
<td>3<em>5</em>(8) = 120</td>
</tr>
<tr>
<td>int (*A5[3]) [5]</td>
<td>Y</td>
<td>N</td>
<td>3*8 = 24</td>
</tr>
</tbody>
</table>

A5 is an array of 3 elements of type (int *)
# Bonus Coverage: Arrays

<table>
<thead>
<tr>
<th>Decl</th>
<th>An</th>
<th>*An</th>
<th>**An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</td>
<td>Bad</td>
<td>Size</td>
</tr>
<tr>
<td>int A1[3][5]</td>
<td>Y</td>
<td>N</td>
<td>60</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
<td>N</td>
<td>120</td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td>Y</td>
<td>N</td>
<td>8</td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td>Y</td>
<td>N</td>
<td>120</td>
</tr>
<tr>
<td>int *(*A5[3])[5]</td>
<td>Y</td>
<td>N</td>
<td>24</td>
</tr>
</tbody>
</table>

ex., A3: pointer to a 3x5 int array  
*A3: 3x5 int array (3 * 5 elements * each 4 bytes = 60)  
**A3: BAD, but means stepping inside one of 3 “rows” c
## Bonus Coverage: Arrays

<table>
<thead>
<tr>
<th>Decl</th>
<th>An</th>
<th>*An</th>
<th>**An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</td>
<td>Bad</td>
<td>Size</td>
</tr>
<tr>
<td>int A1[3][5]</td>
<td>Y</td>
<td>N</td>
<td>60</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
<td>N</td>
<td>120</td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td>Y</td>
<td>N</td>
<td>8</td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td>Y</td>
<td>N</td>
<td>120</td>
</tr>
<tr>
<td>int (*A5[3])[5]</td>
<td>Y</td>
<td>N</td>
<td>24</td>
</tr>
</tbody>
</table>

ex., A5: array of 3 (int *) pointers  
*A5: 1 (int *) pointer, points to an array of 5 ints  
**A5: BAD, means accessing 5 individual ints of the pointer (stepping inside “row”)
Sample assembly-type questions

```c
int *get_pgh_zip(int index) {
    return pgh[index];
}
```

```assembly
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq pgh(,%rax,4),%rax # pgh + (20 * index)
```
Bonus Coverage: Arrays

Nested Array Row Access Code

```
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

- Row Vector
  - \( pgh[\text{index}] \) is array of 5 int’s
  - Starting address \( pgh + 20 \times \text{index} \)

- Machine Code
  - Computes and returns address
  - Compute as \( pgh + 4 \times (\text{index} + 4 \times \text{index}) \)
Bonus Coverage: Arrays

Nested Array Element Access Code

```
int get_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

```
leaq (%rdi,%rdi,4), %rax       # 5*index
addl %rax, %rsi               # 5*index+dig
movl pgh(,%rsi,4), %eax      # M[pgh + 4*(5*index+dig)]
```

- **Array Elements**
  - `pgh[index][dig]` is int
  - Address: `pgh + 20*index + 4*dig`
    - `= pgh + 4*(5*index + dig)`
Bonus! Another Cache problem

- Consider you have the following cache:
  - 64-byte capacity
  - Directly mapped
  - You have an 8-bit address space
A. How many tag bits are there in the cache?
   - Do we know how many set bits there are? What about offset bits? $2^6 = 64$
   - If we have a 64-byte direct-mapped cache, we know the number of $s + b$ bits there are total!
   - Then $t + s + b = 8 \rightarrow t = 8 - (s + b)$
   - Thus, we have 2 tag bits!
Bonus!

B. Fill in the following table, indicating the set number based on the hit/miss pattern.
   a. By the power of guess and check tracing through, identify which partition of $s + b$ bits matches the H/M pattern.

<table>
<thead>
<tr>
<th>Load</th>
<th>Binary Address</th>
<th>Set</th>
<th>H/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1011 0011</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1010 0111</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1101 1001</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1011 1100</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1011 1001</td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>
Bonus!

B. Fill in the following table, indicating the set number based on the hit/miss pattern.

a. By the power of guess-and-check tracing through, identify which partition of $s + b$ bits matches the H/M pattern.

<table>
<thead>
<tr>
<th>Load</th>
<th>Binary Address</th>
<th>Set</th>
<th>H/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1011 0011</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>1010 0111</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>1101 1001</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>1011 1100</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>5</td>
<td>1011 1001</td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>
B. Fill in the following table, indicating the set number based on the hit/miss pattern.
   a. By the power of guess-and-check tracing through, identify which partition of \( s + b \) bits matches the H/M pattern.

<table>
<thead>
<tr>
<th>Load</th>
<th>Binary Address</th>
<th>Set</th>
<th>H/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1011 0011</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>2</td>
<td>1010 0111</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td>1101 1001</td>
<td></td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>1011 1100</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>5</td>
<td>1011 1001</td>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>
**Bonus!**

B. Fill in the following table, indicating the set number based on the hit/miss pattern.

a. By the power of guess and check tracing through, identify which partition of \( s + b \) bits matches the H/M pattern.

<table>
<thead>
<tr>
<th>Load</th>
<th>Binary Address</th>
<th>Set</th>
<th>H/M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( 1011 \ 0011 )</td>
<td>3</td>
<td>( M )</td>
</tr>
<tr>
<td>2</td>
<td>( 1010 \ 0111 )</td>
<td>2</td>
<td>( M )</td>
</tr>
<tr>
<td>3</td>
<td>( 1101 \ 1001 )</td>
<td>1</td>
<td>( M )</td>
</tr>
<tr>
<td>4</td>
<td>( 1011 \ 1100 )</td>
<td>3</td>
<td>( H )</td>
</tr>
<tr>
<td>5</td>
<td>( 1011 \ 1001 )</td>
<td>3</td>
<td>( H )</td>
</tr>
</tbody>
</table>
Bonus!

C. How many sets are there? 2 bits $\rightarrow$ 4 sets
   How big is each cache line? 4 bits $\rightarrow$ 16 bytes
In summary...

- Read the write-up textbook!
- Also read the write-up lecture slides!
- Midterm covers CS:APP Ch. 1-3, 6
- Ask questions on Piazza! For the midterm, make them public and specific if from the practice server!
- G~O~O~D~L~U~C~K (also go Knicks)