15-213: M20 Midterm Review Session

Anja and Josh Z

Agenda

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

Reminders

- Exam This Friday

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

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

Address	Set	Tag	H/M	Evict? Y/N
0x56				
0x6D				
0x49				
0x3A				

Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110				
0110 1101				
0100 1001				
0011 1010				

Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101				
0100 1001				
0011 1010				

Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101	1	01	M	N
0100 1001				
0011 1010				

Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101	1	01	M	N
0100 1001	0	01	H	N
0011 1010				

Problem 1: Cache

Address	Set	Tag	H/M	Evict? Y/N
0101 0110	0	01	M	N
0110 1101	1	01	M	N
0100 1001	0	01	H	N
0011 1010	1	00	M	Y

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

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

Problem 1: Cache

B. Assume A and B are 128 ints and cache-aligned. Assume the cache dimensions given on the previous slide (4 sets, 64-byte blocks, 2 way assoc.).
   a. What is the miss rate of pass 1?
   b. What is the miss rate of pass 2?

```
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.

```
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**

```
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.**

```
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 ⅛ miss.

```
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: 12.5% miss**

```
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(Rb,Ri,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`):

```
# 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.

```
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),%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.

```
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),%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
```

Let's trace through the code!

Problem 2: Assembly

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

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

%eax is d

```
# 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.

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

Keep this in mind...

```
# 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.

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

```
# 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
```

cmp %edx (z), %edi (x)

ja is "Jump if (dest) above (src)"

i.e. jump if x > z

Don't want to memorize jump variants? Hmmmm…...

Problem 2: Assembly

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

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

```
# 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
```

No other "+ 2" in assembly

e is %r8d, i = z in %rdx

Problem 2: Assembly

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

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

```
# 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
```

y is in %rsi, e is in %r8d

Where did %cl come from?

%ecx	%cx	%ch	%cl

Problem 2: Assembly

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

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

```
# 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
```

What's in %r9d? %rsi - 1 i.e. y - 1

Problem 2: Assembly

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

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

d is in %eax, e is in %r8d

```
# 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.

```
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 = [ d + e ];
  }
  return [ d ];
}
```

What's left?

Increment loop counter

```
# 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.

```
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 = [ d + e ];
  }
  return [ d ];
}
```

```
# 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 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:

```
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);
}
```

```
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
```

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "midtermexam"
```

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:

```
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);
}
```

```
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
```

```
caller:
        subq    $8, %rsp
        movl    $86547, %esi
        movl    $.LC0, %edi
   →    call    foo                 %rsp = 0x800100
        addq    $8, %rsp
        ret

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:= 0x400300
        .string "midtermexam"
```

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()`?

```
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);
}
```

Hints:

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

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

```
caller:
        subq    $8, %rsp
        movl    $86547, %esi
        movl    $.LC0, %edi
 Start  call    foo              %rsp = 0x800100
        addq    $8, %rsp
        ret

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

Problem 3: Stack

Arrow is instruction that will execute NEXT

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);
}
```

%rsp	0x800100
%rdi	.LC0
%rsi	0x15213

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

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

```
caller:
        subq    $8, %rsp
        movl    $86547, %esi
        movl    $.LC0, %edi
  ->    call    foo              %rsp = 0x800100
        addq    $8, %rsp
        ret

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

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);
}
```

%rsp	0x8000f8
%rdi	.LC0
%rsi	0x15213

0x800100	?
0x8000f8	ret address for `foo()`
0x8000f0	
0x8000e8	
0x8000e0	
0x8000d8	
0x8000d0	
0x8000c8	
0x8000c0	
0x8000b8	

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

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

Problem 3: Stack

Hint: `$24` in decimal = `0x18`

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);
}
```

%rsp	0x8000e0
%rdi	.LC0
%rsi	0x15213

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

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

0x800100	?
0x8000f8	ret address for foo()
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);
}
```

%rsp	0x8000e0
%rdi	.LC0
%rsi	0xdeadbeef

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

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

0x800100	?
0x8000f8	ret address for `foo()`
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);
}
```

%rsp	0x8000d8
%rdi	.LC0
%rsi	0xdeadbeef

0x800100	?
0x8000f8	ret address for `foo()`
0x8000f0	?
0x8000e8	?
0x8000e0	?
0x8000d8	ret address for `foo()`
0x8000d0	
0x8000c8	
0x8000c0	
0x8000b8	

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

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

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);
}
```

%rsp	0x8000c0
%rdi	.LC0
%rsi	0xdeadbeef

0x800100	?
0x8000f8	ret address for `foo()`
0x8000f0	?
0x8000e8	?
0x8000e0	?
0x8000d8	ret address for `foo()`
0x8000d0	?
0x8000c8	?
0x8000c0	?
0x8000b8	

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

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

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);
}
```

%rsp	0x8000c0
%rdi	.LC0
%rsi	0xdeadbeef

0x800100	?
0x8000f8	ret address for foo()
0x8000f0	?
0x8000e8	?
0x8000e0	?
0x8000d8	ret address for foo()
0x8000d0	?
0x8000c8	?
0x8000c0	?
0x8000b8	

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

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

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!

%rsp	0x8000c0
%rdi	0x8000c0
%rsi	.LCO

0x800100	?
0x8000f8	ret address for `foo()`
0x8000f0	?
0x8000e8	?
0x8000e0	?
0x8000d8	ret address for `foo()`
0x8000d0	?
0x8000c8	?
0x8000c0	?
0x8000b8	

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

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

Problem 3: Stack

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

```
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);
}
```

%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    %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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

0x800100	?
0x8000f8	ret address for `foo()`
0x8000f0	?
0x8000e8	?
0x8000e0	?
0x8000d8	ret address for `foo()`
0x8000d0	?
0x8000c8	?
0x8000c0	?
0x8000b8	

Problem 3: Stack

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

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

```
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);
}
```

```
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
```

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

        .section        .rodata.s
.LC0: = 0x400300
        .string "midtermexam"
```

0x800100	?							
0x8000f8	ret address for `foo()`							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for `foo()`							
0x8000d0	?							
0x8000c8								
0x8000c0					←	'd'	'i'	'm'
	c7					c2	c1	c0
0x8000b8								

Problem 3: Stack

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

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

```
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);
}
```

```
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
```

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

        .section        .rodata.s
.LC0: = 0x400300
        .string "midtermexam"
```

0x800100	?							
0x8000f8	ret address for `foo()`							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for `foo()`							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8	c7					c2	c1	c0

Problem 3: Stack

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

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

```
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);
}
```

```
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
```

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

        .section        .rodata.s
.LC0: = 0x400300
        .string "midtermexam"
```

0x800100	?							
0x8000f8	ret address for `foo()`							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for `foo()`							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8					c3	buf[0]		c0

Problem 3: Stack

`buf[0]` = | 't' | 'd' | 'i' | 'm' |

't'	'd'	'i'	'm'

=

74	64	69	6d

(as int) = **0x7464696d**

Char	Hex	Char	Hex
a	61	m	6d
d	64	r	72
e	65	t	74
i	69	x	78

0x800100	?							
0x8000f8	ret address for `foo()`							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for `foo()`							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8					**`buf[0]`**			

Problem 3: Stack

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

Question 3: What is the hex value of **`buf[1]`** when `strcpy()` **<u>returns</u>**?

```
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);
}
```

```
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
```

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

        .section        .rodata.s
.LC0: = 0x400300
        .string "midtermexam"
```

0x800100	?							
0x8000f8	ret address for `foo()`							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for `foo()`							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8	c7	buf[1]		c4		buf[0]		

Problem 3: Stack

`buf[1]` = | 'e' | 'm' | 'r' | 'e' |

= | 65 | 6d | 72 | 65 |

`(as int)` = **0x656d7265**

Char	Hex	Char	Hex
a	61	m	6d
d	64	r	72
e	65	t	74
i	69	x	78

0x800100	?							
0x8000f8	ret address for foo()							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for foo()							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8	**buf[1]**							

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()`

```
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);
}
```

```
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
```

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

        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

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?

```
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);
}
```

```
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
```

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

        .section    .rodata.str1.1,"aMS",@progbits,1
.LC0: = 0x400300
        .string "midtermexam"
```

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

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

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 overwrite any return addresses, so there was no corruption!

0x800100	?							
0x8000f8	ret address for `foo()`							
0x8000f0	?							
0x8000e8	?							
0x8000e0	?							
0x8000d8	ret address for `foo()`							
0x8000d0	?							
0x8000c8	?	?	?	?	'\0'	'm'	'a'	'x'
0x8000c0	'e'	'm'	'r'	'e'	't'	'd'	'i'	'm'
0x8000b8								

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

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

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

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

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

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

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

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

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

s = 0

M = 31/16 => 1.1111

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

E = 1 => exponent = 1 + 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) 31/8

Step 3: Find the fraction bits and exponent bits

s = 0

M = 1.1111 => fraction bits are 1111

exponent bits are 10

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

Step 4: Take care of rounding issues

Current number is 0 10 111 1 <= excess bit

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

Step 4: Take care of rounding issues

Current number is 0 10 111 1 <= excess bit

Guard bit (last bit kept) = 1

Round bit (first bit removed) = 1

Sticky bit (OR of other removed bits) = 0

I.e. same distance between 111 and “000”, so round up to even (add 1 to the fraction bits).

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

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

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

Step 4: Take care of rounding issues

Result is 0 11 000 <= Infinity!

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) -7/8

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) -7/8

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

s = 1

M = 7/4

E = -1

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) -7/8

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

s = 1

M = 7/4 => 1.11

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

E = -1 => exponent = -1 + 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) -7/8

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

s = 1

M = 7/4 => 1.11 <= (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 => exponent = -1 + bias = 0 <= denormalized!

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) -7/8

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

s = 1

M = 7/8 => 0.111 <= M should be in the range [0.0, 1.0) for denormalized numbers so we divide it by 2

exp = 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) -7/8

Step 3: Find the fraction bits and exponent bits

s = 1

M = 0.111 => Fraction bits = 111

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

exp = 2 => E = exp - bias = 1 (normalized)

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

Result = 2*1.101 = 2*(13/8) = 13/4

Bonus Coverage: Arrays

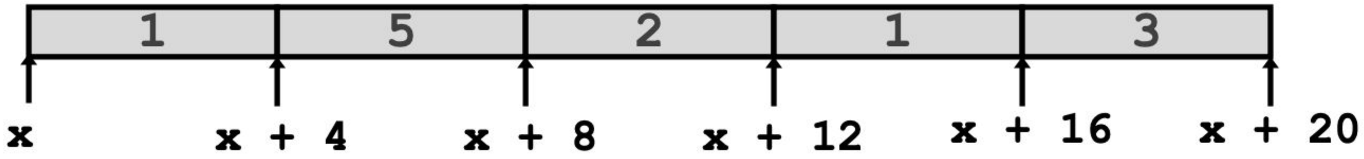
<u>IMPORTANT POINTS + TIPS:</u>

- ***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):

`int val[5];`



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

	Type	Value
`val`		
`val[2]`		
`*(val + 2)`		
`&val[2]`		
`val + 2`		
`val + i`		

Bonus Coverage: Arrays

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

`int val[5];`

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

	Type	Value
`val`	`int *`	`x`
`val[2]`	`int`	`2`
`*(val + 2)`	`int`	`2`
`&val[2]`	`int *`	`x + 8`
`val + 2`	`int *`	`x + 8`
`val + i`	`int *`	`x + (4 * i)`

Bonus Coverage: Arrays

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

`int val[5];`

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

	Type	Value
`val`	`int *`	`x`
`val[2]`	`int`	`2`
`*(val + 2)`	`int`	`2`
`&val[2]`	`int *`	`x + 8`
`val + 2`	`int *`	`x + 8`
`val + i`	`int *`	`x + (4 * i)`

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

Bonus Coverage: Arrays

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

`int val[5];`

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

	Type	Value
`val`	`int *`	`x`
`val[2]`	`int`	`2`
`*(val + 2)`	`int`	`2`
`&val[2]`	`int *`	`x + 8`
`val + 2`	`int *`	`x + 8`
`val + i`	`int *`	`x + (4 * i)`

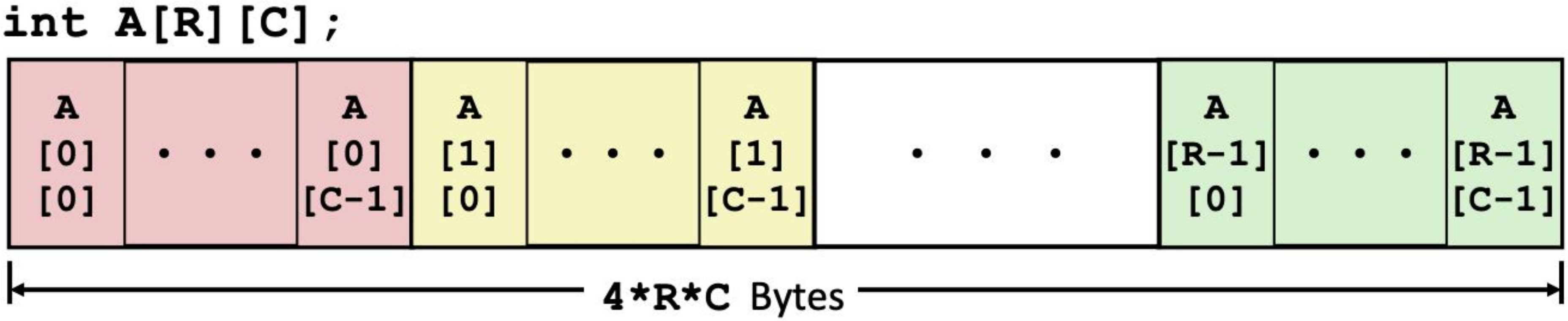
Accessing methods:
- *val[index]*
- **(val + index)*

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

Bonus Coverage: Arrays

<u>Nested indexing rules</u> (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)

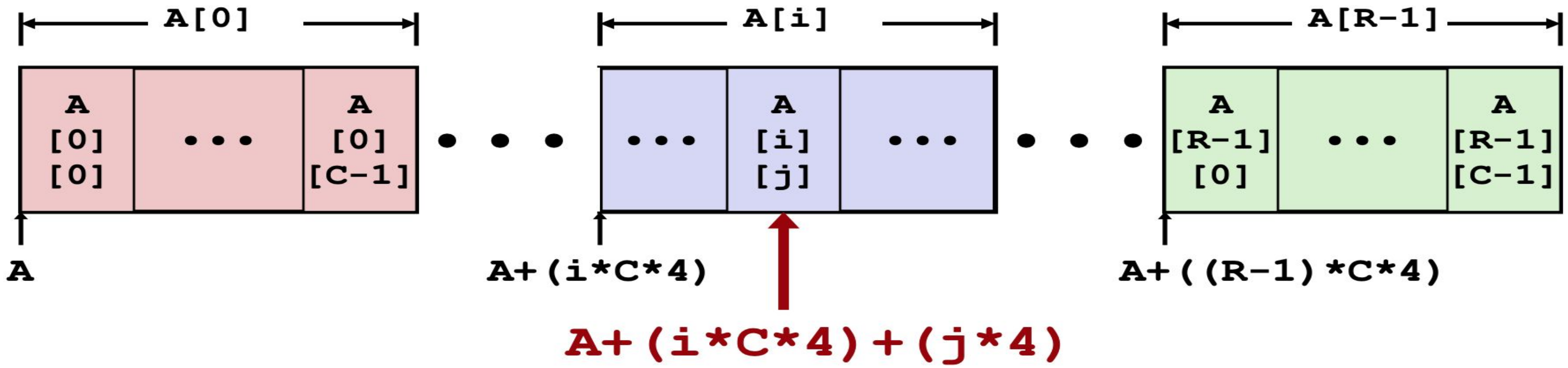


Bonus Coverage: Arrays

<u>Nested indexing rules</u> (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

`int A[R][C];`



Bonus Coverage: Arrays

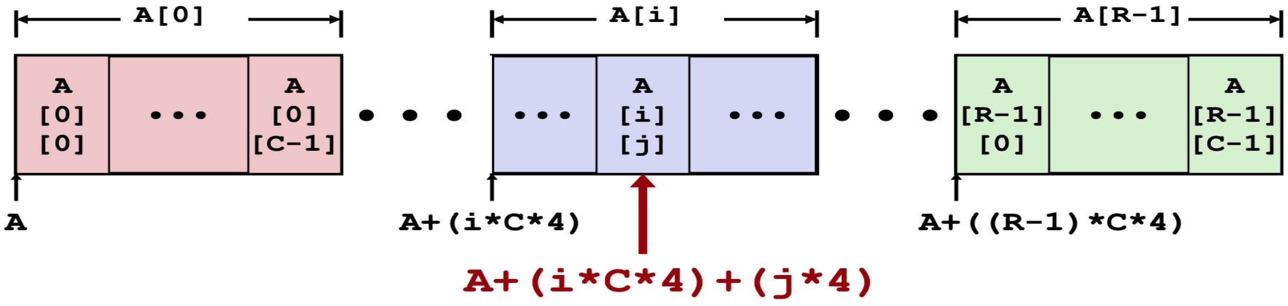
Nested indexing rules (for your cheatsheet and/or big brain):

`A[i][j]` is element of type *T*, which requires *K* bytes

Address `A + i * (C * K) + j * K`

`= A + (i * C + j) * K`

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



Bonus Coverage: Arrays

Consider accessing elements of **A**....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Bonus Coverage: Arrays

Consider accessing elements of A....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*4 = 60
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Bonus Coverage: Arrays

Consider accessing elements of A....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*(4) = 60
`int *A2[3][5]`	Y	N	3*5*(8) = 120
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Bonus Coverage: Arrays

Consider accessing elements of A….

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*(4) = 60
`int *A2[3][5]`	Y	N	3*5*(8) = 120
`int (*A3)[3][5]`	Y	N	1*8 = 8
`int *(A4[3][5])`			
`int (*A5[3])[5]`			

Bonus Coverage: Arrays

Consider accessing elements of A….

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*(4) = 60
`int *A2[3][5]`	Y	N	3*5*(8) = 120
`int (*A3)[3][5]`	Y	N	1*8 = 8
`int *(A4[3][5])`	Y	N	3*5*(8) = 120
`int (*A5[3])[5]`			

A4 is a pointer to a 3x5 (int *) element array

Bonus Coverage: Arrays

Consider accessing elements of A....

	Compiles	Bad Deref?	Size (bytes)
`int A1[3][5]`	Y	N	3*5*(4) = 60
`int *A2[3][5]`	Y	N	3*5*(8) = 120
`int (*A3)[3][5]`	Y	N	1*8 = 8
`int *(A4[3][5])`	Y	N	3*5*(8) = 120
`int (*A5[3])[5]`	Y	N	3*8 = 24

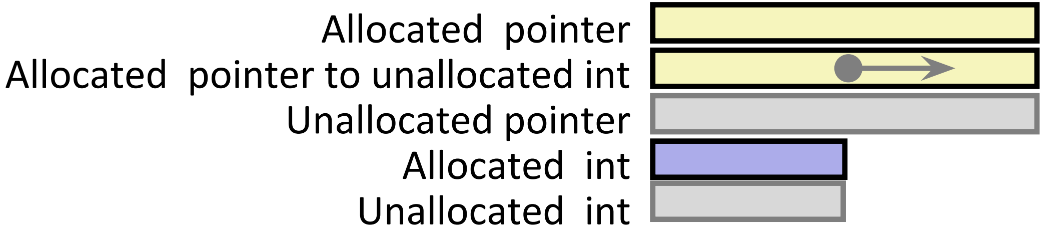
A5 is an array of 3 elements of type (int *)

Understanding Pointers & Arrays #3

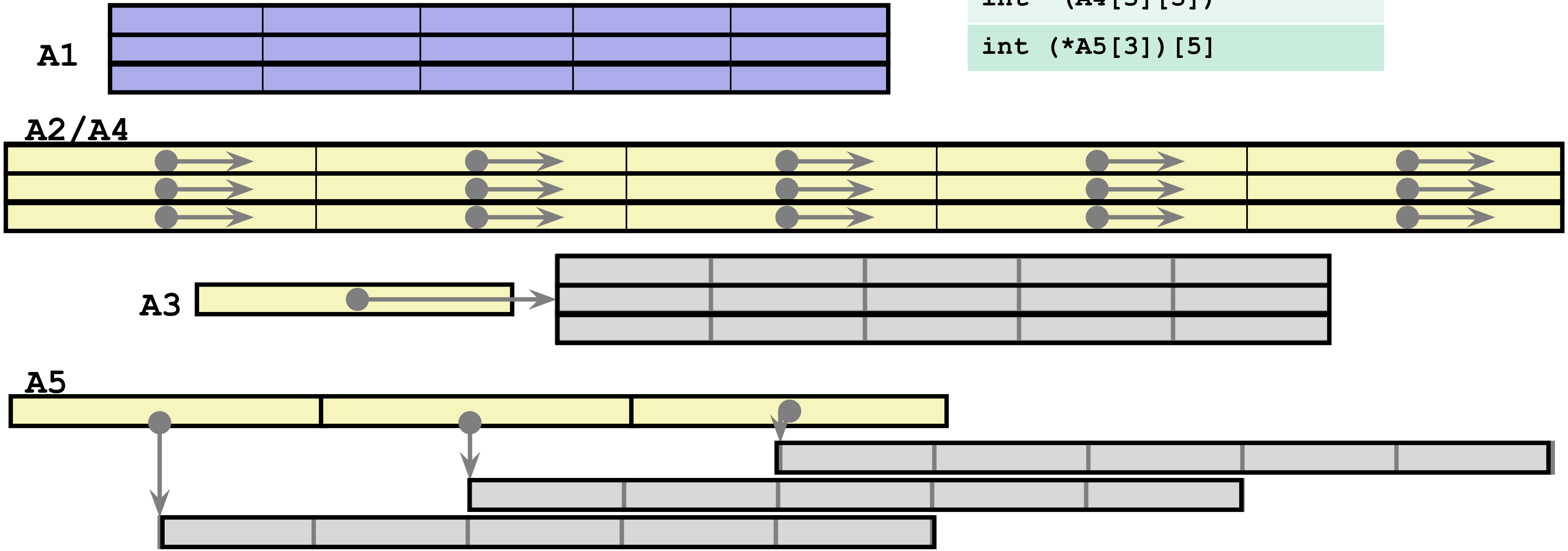
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`									
`int *A2[3][5]`									
`int (*A3)[3][5]`									
`int *(A4[3][5])`									
`int (*A5[3])[5]`									

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`			
`int *A2[3][5]`			
`int (*A3)[3][5]`			
`int *(A4[3][5])`			
`int (*A5[3])[5]`			



Declaration
`int A1[3][5]`
`int *A2[3][5]`
`int (*A3)[3][5]`
`int *(A4[3][5])`
`int (*A5[3])[5]`



Understanding Pointers & Arrays #3

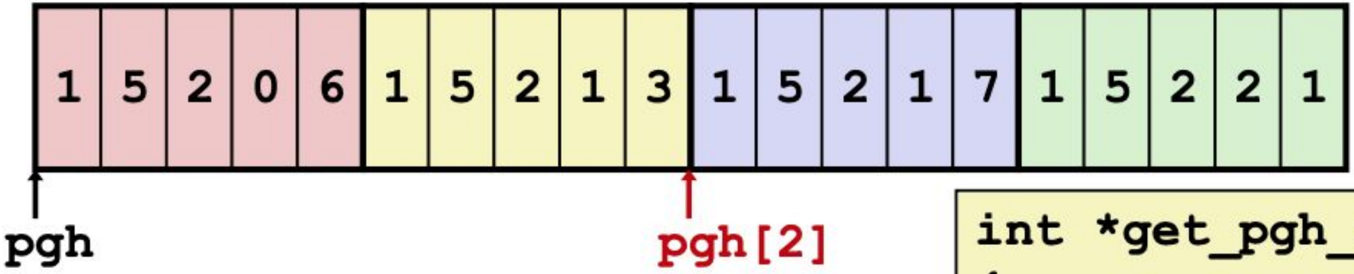
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3][5]`	Y	N	60	Y	N	20	Y	N	4
`int *A2[3][5]`	Y	N	120	Y	N	40	Y	N	8
`int (*A3)[3][5]`	Y	N	8	Y	Y	60	Y	Y	20
`int *(A4[3][5])`	Y	N	120	Y	N	40	Y	N	8
`int (*A5[3])[5]`	Y	N	24	Y	N	8	Y	Y	20

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Decl	***An		
	Cmp	Bad	Size
`int A1[3][5]`	N	-	-
`int *A2[3][5]`	Y	Y	4
`int (*A3)[3][5]`	Y	Y	4
`int *(A4[3][5])`	Y	Y	4
`int (*A5[3])[5]`	Y	Y	4

Bonus Coverage: Arrays

Sample assembly-type questions

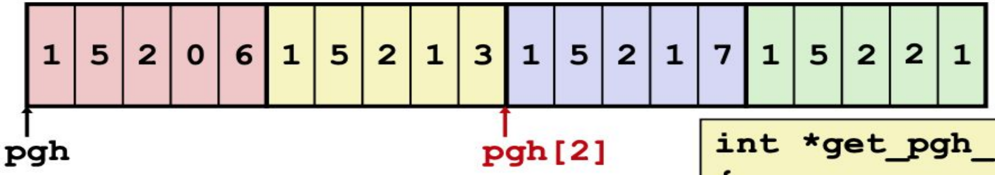



```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
# %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];
}
```

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax  # 5 * index
leaq pgh(,%rax,4),%rax   # pgh + (20 * index)
```

- **Row Vector**
  - `pgh[index]` is array of 5 `int`'s
  - Starting address `pgh+20*index`
- **Machine Code**
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`

Bonus Coverage: Arrays

Nested Array Element Access Code

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

`pgh` (points to the first element) `pgh[1][1]` (points to the seventh element)

```
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

Bonus!

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 <u>2 tag bits!</u>

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.

Load	Binary Address	Set	H/M
1	1011 0011		M
2	1010 0111		M
3	1101 1001		M
4	1011 1100		H
5	1011 1001		H

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.

Load	Binary Address	Set	H/M
1	1011 0011		M
2	1010 0111		M
3	1101 1001		M
4	1011 1100		H
5	1011 1001		H

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.

Load	Binary Address	Set	H/M
1	1011 0011		M
2	1010 0111		M
3	1101 1001		M
4	1011 1100		H
5	1011 1001		H

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.

Load	Binary Address	Set	H/M
1	10<u>11</u> 0011	3	M
2	10<u>10</u> 0111	2	M
3	11<u>01</u> 1001	1	M
4	10<u>11</u> 1100	3	H
5	10<u>11</u> 1001	3	H

Bonus!

C. How many sets are there? 2 bits → 4 sets
How big is each cache line? 4 bits → 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