15-213: S19 Midterm Review Session

Big Brain Cyrus, Eugene, Niko and Emma
3 Mar 2019
Agenda

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

- There will be office hours this week, but with fewer TAs than usual! 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!

- Lecture is still happening! 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 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
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
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
#include <stdio.h>

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;
}
```
Problem 1: Cache

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: 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) $31/8$
Problem 2: 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 \((-1)^s \times M \times 2^E\)
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)
   - \(E = 1\)
Problem 2: 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 2: Convert M into binary and find value of exp.

\( E = 0 \)

\( M = \frac{31}{16} \) (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 $exp$

$s = 0$

$M = 31/16 => 1.1111$

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

$E = 1 => \text{exponent} = 1 + \text{bias} = 2$
Problem 2: 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

\( s = 0 \)

\( M = 1.1111 \Rightarrow \) fraction bits are 1111

exponent bits are 10
Problem 2: 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 4: Take care of rounding issues

Current number is 0 10 111 \( 1 \) \( \leq \) excess bit
Problem 2: 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 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)
Problem 2: 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 4: Take care of rounding issues
Current number is 0 10 111 \( 1 \leq \text{excess bit} \)

Adding 1 overflows the floating bits, so we increment the exponent bits by 1 and set the fraction bits to 0
Problem 2: 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 <= \textit{Infinity!}
Problem 2: 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) \(-7/8\)
Problem 2: 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 \((-1)^s \cdot M \cdot 2^E\)

\(s = 1\)

\(M = 7/4\)

\(E = -1\)
Problem 2: 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 exponent
\[ s = 1 \]
\[ M = 7/4 \Rightarrow 1.11 \]

bias = \(2^{k-1} - 1\) (k is the number of exponent bits) = 1
E = -1 \Rightarrow exponent = -1 + bias = 0
Problem 2: 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

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

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

E = -1 \Rightarrow \text{exponent} = -1 + \text{bias} = 0 \leq \text{denormalized}
Problem 2: 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

\[ s = 1 \]

\[ M = \frac{7}{8} \Rightarrow 0.111 \]

<= M should be in the range \([0.0, 1.0)\) for denormalized numbers so we divide it by 2

\[ \text{exp} = 0 \]
Problem 2: 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 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
Problem 2: 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
Problem 2: 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) \[\begin{array}{c}
0 \\
10 \\
101 \\
\end{array}\]

\[\begin{align*}
\text{s} &= 0 \\
\text{exp} &= 2 \Rightarrow E = \text{exp} - \text{bias} = 1 \text{ (normalized)} \\
M &= 1.101 \text{ (between 1 and 2 since it is normalised)} \\
\text{Result} &= 2 \times 1.101 = 2 \times (13/8) = 13/4
\end{align*}\]
Problem 2: 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
Problem 3: 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.
Problem 3: 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>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>
Problem 3: Arrays

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

- A can be used as the pointer to the first array element: \( \text{A}[0] \)

```
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>

```
\[ \text{Type} \quad \text{Value} \]
\[ \text{int} \quad 2 \]
\[ \text{int} \quad 2 \]
\[ \text{int} \quad x + 8 \]
\[ \text{int} \quad x + 8 \]
\[ \text{int} \quad x + (4 \times i) \]
Problem 3: Arrays

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

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

```plaintext
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>
<tr>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>( 2 )</td>
<td></td>
</tr>
<tr>
<td>( x + 8 )</td>
<td></td>
</tr>
<tr>
<td>( x + 8 )</td>
<td></td>
</tr>
<tr>
<td>( x + (4 \times i) )</td>
<td></td>
</tr>
</tbody>
</table>

Accessing methods:
- \( val[index] \)
- \( *(val + index) \)
Problem 3: 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>Variable</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val</code></td>
<td>int *</td>
<td><code>x</code></td>
</tr>
<tr>
<td><code>val[2]</code></td>
<td>int</td>
<td><code>2</code></td>
</tr>
<tr>
<td><code>*(val + 2)</code></td>
<td>int</td>
<td><code>2</code></td>
</tr>
<tr>
<td><code>&amp;val[2]</code></td>
<td>int *</td>
<td><code>x + 8</code></td>
</tr>
<tr>
<td><code>val + 2</code></td>
<td>int *</td>
<td><code>x + 8</code></td>
</tr>
<tr>
<td><code>val + i</code></td>
<td>int *</td>
<td><code>x + (4 * i)</code></td>
</tr>
</tbody>
</table>
Problem 3: 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 of array indexing](image)
Problem 3: Arrays

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[0][0]      A[i][j]      [A[R-1][0]  
[A[0][C-1]]  ...          [A[R-1][C-1]]
A              A+ (i*C*4)   A+ ((R-1)*C*4)
```

$A+ (i*C*4) + (j*4)$
Problem 3: Arrays

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

\[ A[i][j] \] is element of type \( T \), which requires \( K \) bytes.

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

\[ = A + (i \times C + j) \times K \]
Problem 3: Arrays

Consider accessing elements of A....

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A5[3])[5]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Problem 3: Arrays

Consider accessing elements of $A$.

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

Consider accessing elements of $A$.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Compiles</th>
<th>Bad Deref?</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int A1[3][5]</code></td>
<td>Y</td>
<td>N</td>
<td>$3 \times 5 \times (4) = 60$</td>
</tr>
<tr>
<td><code>int *A2[3][5]</code></td>
<td>Y</td>
<td>N</td>
<td>$3 \times 5 \times (8) = 120$</td>
</tr>
<tr>
<td><code>int (**A3)[3][5]</code></td>
<td>Y</td>
<td>N</td>
<td>$1 \times 8 = 8$</td>
</tr>
<tr>
<td><code>int *(A4[3][5])</code></td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td><code>int (**A5[3])[5]</code></td>
<td>Y</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
Problem 3: Arrays

Consider accessing elements of $A$.

<table>
<thead>
<tr>
<th>Data 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>$3 \times 5 \times 4 = 60$</td>
</tr>
<tr>
<td>int $*A2[3][5]$</td>
<td>Y</td>
<td>N</td>
<td>$3 \times 5 \times 8 = 120$</td>
</tr>
<tr>
<td>int $(*A3)[3][5]$</td>
<td>Y</td>
<td>N</td>
<td>$1 \times 8 = 8$</td>
</tr>
<tr>
<td>int $*(A4[3][5])$</td>
<td>Y</td>
<td>N</td>
<td>$3 \times 5 \times 8 = 120$</td>
</tr>
<tr>
<td>int $(*A5[3])[5]$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Type 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 \times 5 \times 4 = 60 )</td>
</tr>
<tr>
<td>int ( *A2[3][5] )</td>
<td>Y</td>
<td>N</td>
<td>( 3 \times 5 \times 8 = 120 )</td>
</tr>
<tr>
<td>int ( (*A3)[3][5] )</td>
<td>Y</td>
<td>N</td>
<td>( 1 \times 8 = 8 )</td>
</tr>
<tr>
<td>int ( *(A4[3][5]) )</td>
<td>Y</td>
<td>N</td>
<td>( 3 \times 5 \times 8 = 120 )</td>
</tr>
<tr>
<td>int ( (*A5[3])[5] )</td>
<td>Y</td>
<td>N</td>
<td>( 3 \times 8 = 24 )</td>
</tr>
</tbody>
</table>

A5 is an array of 3 elements of type (int *)
### Problem 3: 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
**Problem 3: 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”)
### Problem 3: Arrays

#### 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)
```
Problem 3: 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)`
Problem 3: 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)`
Problem 4: Stack

- Important things to remember:
  - Stack grows **DOWN**!
  - %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 4: 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);
}

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

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
Consider the following code:

```c
void foo(char *str, int a) {
    int buf[2];
    if (a != 0xd6ad5beef) {
        foo(str, 0xd6ad5beef);
        return;
    }
    strcpy((char*) buf, str);
}
```

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

**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 4: 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);
}
```

Hints:
- Step through the program instruction by instruction from start to end
- Draw a stack diagram!!!
- Keep track of registers too
Problem 4: 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);
}
```

Arrow is instruction that will execute NEXT

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

<table>
<thead>
<tr>
<th>foo:</th>
<th>caller:</th>
</tr>
</thead>
<tbody>
<tr>
<td>subq $24, %rsp</td>
<td>subq $8, %rsp</td>
</tr>
<tr>
<td>cmpl $0xdeadbeef, %esi</td>
<td>movl $86547, %esi</td>
</tr>
<tr>
<td>je .L2</td>
<td>movl $.LC0, %edi</td>
</tr>
<tr>
<td>movl $0xdeadbeef, %esi</td>
<td>call foo</td>
</tr>
<tr>
<td>call foo</td>
<td>addq $8, %rsp</td>
</tr>
<tr>
<td>jmp .L1</td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>.L2:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>movq %rdi, %rsi</td>
<td></td>
</tr>
<tr>
<td>movq %rsp, %rdi</td>
<td></td>
</tr>
<tr>
<td>End</td>
<td></td>
</tr>
<tr>
<td>call strcpy</td>
<td></td>
</tr>
</tbody>
</table>

Arrow is instruction that will execute NEXT

<table>
<thead>
<tr>
<th>.L1:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addq $24, %rsp</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
Problem 4: 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);
}
```

```asm
value: 0x8000f8 = ret address for foo()
value: 0x8000f0
value: 0x8000e8
value: 0x8000e0
value: 0x8000d8
value: 0x8000d0
value: 0x8000c8
value: 0x8000c0
value: 0x8000b8
Problem 4: 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>0x8000e0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>.LC0</td>
</tr>
<tr>
<td>%rsi</td>
<td>0x15213</td>
</tr>
</tbody>
</table>

**Hint:** $24$ in decimal = $0x18$
Problem 4: 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);  
}
```

```asm
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
    jmp .L1

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

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

- `%rsp` = 0x800e0
- `%rdi` = `.LC0`
- `%rsi` = 0xdeadbeef

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800100</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret address for <code>foo()</code></td>
</tr>
<tr>
<td>0x8000e0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000e8</td>
<td>?</td>
</tr>
<tr>
<td>0x8000d8</td>
<td>?</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>
Problem 4: 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>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000d8</td>
<td></td>
<td>ret address for <code>foo()</code></td>
</tr>
<tr>
<td>0x8000f8</td>
<td></td>
<td>ret address for <code>foo()</code></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>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 4: 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><code>%rsp</code></th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x800100</td>
<td>?</td>
</tr>
<tr>
<td>0x8000f8</td>
<td>ret address for <code>foo()</code></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 <code>foo()</code></td>
</tr>
<tr>
<td>0x8000d0</td>
<td>?</td>
</tr>
<tr>
<td>0x8000c8</td>
<td>?</td>
</tr>
<tr>
<td><strong>0x8000c0</strong></td>
<td>?</td>
</tr>
<tr>
<td>0x8000b8</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>%rdi</code></th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LC0</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>%rsi</code></th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xdeadbeef</td>
<td>?</td>
</tr>
</tbody>
</table>
### Problem 4: 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);
}
```

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

---

#### Stack Frames

<table>
<thead>
<tr>
<th>Function</th>
<th>%rsp</th>
<th>Address</th>
<th>%rdi</th>
<th>Address</th>
<th>%rsi</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller</td>
<td>0x8000c0</td>
<td>0x8000100</td>
<td>.LC0</td>
<td>0x8000f8</td>
<td>0x8000c0</td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>0x8000f0</td>
<td>0x8000f8</td>
<td>.LC0</td>
<td>0x8000e0</td>
<td>0x8000f0</td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>0x8000e0</td>
<td>0x8000e0</td>
<td>.LC0</td>
<td>0x8000d0</td>
<td>0x8000e0</td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>0x8000d0</td>
<td>0x8000d0</td>
<td>.LC0</td>
<td>0x8000c0</td>
<td>0x8000d0</td>
<td></td>
</tr>
<tr>
<td>foo</td>
<td>0x8000c0</td>
<td>0x8000c0</td>
<td>.LC0</td>
<td>0x8000b0</td>
<td>0x8000c0</td>
<td></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`
- `call strcpy`

**End**
- `call strcpy`
- `.section .rodata.str1.l"aMS",@progbits,1`
- `.rodata.str1.l":midtermexam"
- `.string "midtermexam"`

Problem 4: Stack

Question 1: What is the hex value of $\text{rsp}$ just before `strcpy()` is called for the first time in `foo()`?

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

- $\text{rsp}$: 0x8000c0
- $\text{rdi}$: 0x8000c0
- $\text{rsi}$: .LCO

---

<table>
<thead>
<tr>
<th>0x8000100</th>
<th>0x800100</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret address for <code>foo()</code></td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000f8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000f0</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000e8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000e0</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000d8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000d0</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000c8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000c0</td>
</tr>
<tr>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x8000b8</th>
</tr>
</thead>
</table>

---

The answer is: 0x8000c0.
Problem 4: 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);
}
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000c0</td>
<td>ret address for foo()</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></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>
Problem 4: Stack

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

```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
    jmp .L1
.L2:
    movq %rdi, %rsi
    movq %rsp, %rdi
    call strcpy
    addq $24, %rsp
    ret

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</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>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>\texttt{d} 'i' 'm'</td>
</tr>
<tr>
<td>0x8000b8</td>
<td></td>
</tr>
</tbody>
</table>
**Problem 4: 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);
}
```

| Address   | | Description |
|-----------|----------------|
| 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 4: 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 ;
}
```

```
subq $24, %rsp      subq $8, %rsp
cmpl $0xdeadbeef, %esi movl $86547, %esi
je   .L2             movl $.LC0, %edi
call foo            call foo
jmp   .L1            addq $8, %rsp
.L2:                   ret
movq   %rdi, %rsi    .section .rodata
movq   %rsp, %rdi     .string "midtermexam"
call   strcpy
.LC0:  = 0x400300
addq   $24, %rsp     buffer[0]
ret
```

```
<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>‘e’ ‘m’ ‘r’ ‘e’ ‘t’ ‘d’ ‘i’ ‘m’</td>
</tr>
<tr>
<td>0x8000b8</td>
<td></td>
</tr>
</tbody>
</table>
```
Problem 4: Stack

buf[0] = ‘t’ ‘d’ ‘i’ ‘m’

= 74 64 69 6d

(as int) = 0x7464696d
Problem 4: 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;
}
```

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

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

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

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

```
0x8000c0  'e' 'm' 'r' 'e'
0x8000b8  buf[1]  c4  buf[0]
```

0x8000f8: ret address for `foo()`
0x8000d8: ret address for `foo()`
0x8000c0: `e' `m' `r' `e'
Problem 4: Stack

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

\[ = 65 \ 6d \ 72 \ 65 \]

\(\text{(as int)} = 0x656d7265\)
Problem 4: 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?

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

This is before the FIRST time we call foo():

```assembly
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
```
Problem 4: 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);
}

This is before the FIRST time we call foo()

Going backwards, %rdi was loaded in caller()

%rdi = $.LC0 = 0x400300 (based on hint)

loaded %rdi
Problem 4: 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 4: 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!
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></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></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>
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 → 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 (also go Knicks)