15-213

"The Class That Gives CMU Its Zip!"

Introduction to Computer Systems

Taerim Kim

March 5, 2012

Topics:

- Assembly
- Stack discipline
- Structs/alignment
- Caching

Midterm

What?

- Everything through caching

Where?

- UC McConomy

When?

- 1:30pm – 2:50pm, Tuesday, March 6

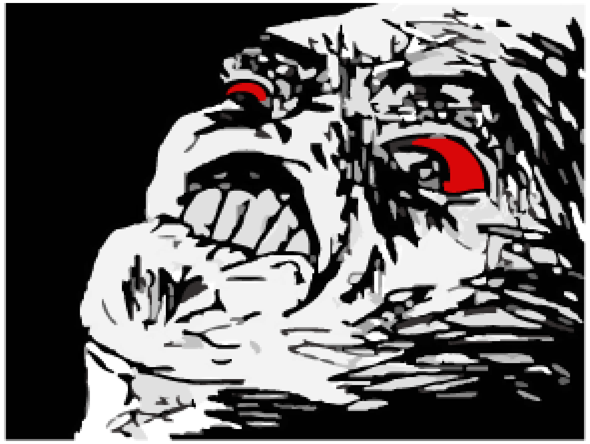
Who?

- *You*

Why? D:

- 20 percent of your final grade

Relax—you get a cheat sheet



Brief overview of exam topics

Data representation

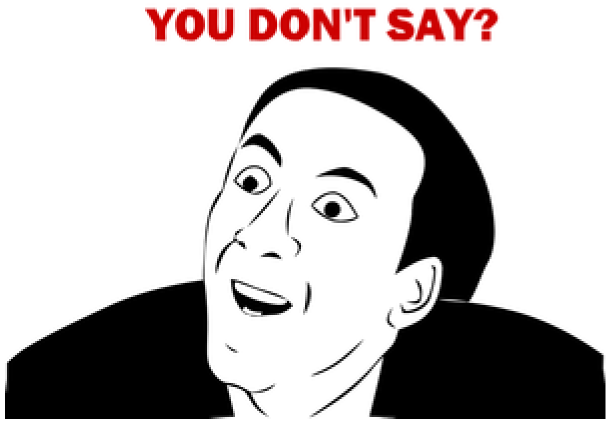
- Integers
- Floating point
- Arrays
- Structs

Assembly

- Registers
- Memory addressing
- Control flow
- Stack discipline

Caching

- Locality
- Dimensions
- Tag, set index, block offset
- Eviction policy
- Blocking



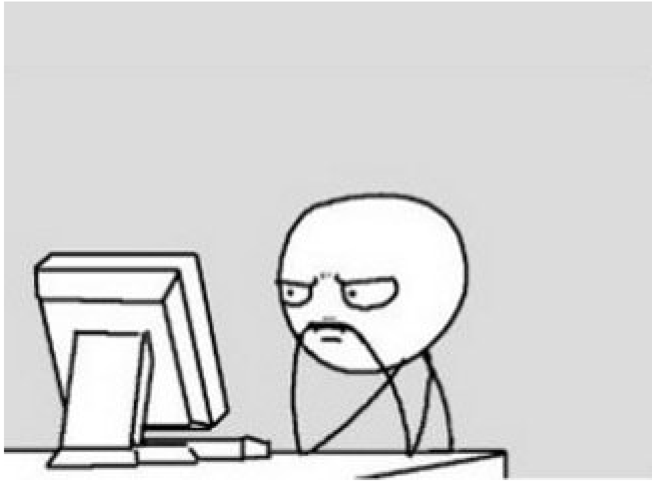
By request

Floating point
- Representation
- Conversion

Assembly
- Stack discipline
- Translation to C

`struct`s
- Alignment/padding
- Assembly

Caching
- Blocking
- Miss rate analysis



Floating point

Representation

- Value: (-1)^s * M * 2^E
- Bias: 2^(k - 1) - 1
- Denormalized: E = 1 - bias
- Normalized: E = exp - bias
- Special values: infinities, NaN

Conversion examples

- 1 sign bit, 3 exponent bits, 3 fraction bits
  - Convert `0 101 101` to decimal fraction
  - Convert -43/32 to floating point

Floating point (2)

Conversion example solutions

- `0 101 101` → 13/2
- -43/32 → `1 011 011`

Food for thought

- What happens when the number of exponent bits increases?
- What happens when the number of fraction bits increases?
- Why can't every real be represented in floating point?
- What happens to resolution as absolute value increases?
- If a number is greater than 1, is it normalized? Converse?
- Why not use fixed point instead?

Floating point

Questions?

Assembly

Special registers

- Stack pointer
  - %esp, %rsp
- Frame pointer
  - %ebp, sometimes %rbp
- Program counter
  - %eip, %rip
- Return value
  - %eax, %rax
- Arguments (x86-64)
  - %rdi, %rsi, etc.

Instructions

- Addressing
  - lea, mov
- Arithmetic
  - add, sub, imul, idiv
- Stack manipulation
  - push, pop, leave
- Local jumps
  - cmp, test
  - jmp, je, jg, jle, etc.
- Procedure calls
  - call, ret

Assembly (2)

What is the difference between lea and mov?

- mov can access memory
  - mov 8(%rsp), %rax → %rax = *(void **)(%rsp + 8)
- lea is arithmetic
  - lea 8(%rsp), %rax → %rax = %rsp + 8

What do push and pop do?

- Inverse operations
- Both manipulate the stack
- Both are analogous to two instructions
  - push %rax → sub $8, %rsp; mov %rax, (%rsp)
  - pop %rax → mov (%rsp), %rax; add $8, %rsp

Assembly (3)

What does leave do?

- Unallocates stack frame
- Akin to two instructions
  - leave → mov %ebp, %esp; pop %ebp
  - Draw a stack diagram

What do call and ret do?

- Procedure calls
- Inverse operations
- Both manipulate the stack
  - call 0xcafebabe → push %eip; jmp 0xcafebabe
  - ret → pop %eip

Assembly (4)

Assembly control flow cookbook

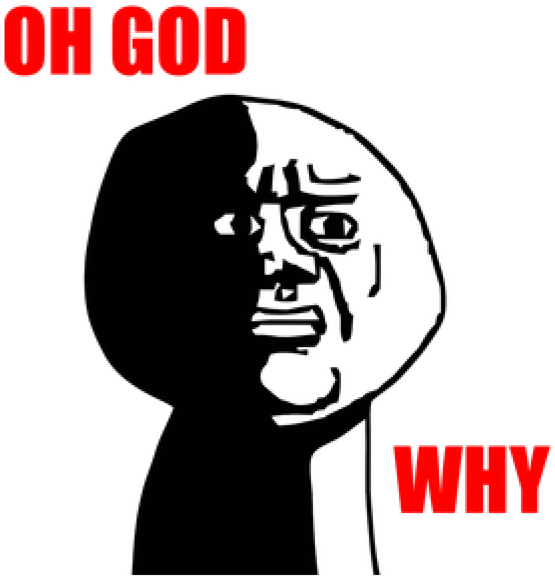
- Assume x is a C variable whose value is in %eax
- To test if x is equal to zero
  - test %eax, %eax
  - Use with je (sometimes jz)
- To test if x (signed) is greater than 15213
  - cmp $15213, %eax
  - Use with jg
- To test if x (unsigned) is greater than 15213
  - cmp $15213, %eax
  - Use with ja
- In general
  - test is like and—only sets condition codes
  - cmp is like sub—only sets condition codes

Assembly (5)

```
int lol(int a, int b)
{
    switch(a)
    {
        case 210:
            b *= 13;
            ________
        case 213:
            b = 18243;
            ________
        case 214:
            b *= b;
            ________
        case 216:
        case 218:
            b -= a;
            ________
        case 219:
            b += 13;
            ________
        default:
            b -= 9;
    }

    return b;
}
```

```
40045c <lol>:
40045c: lea    -0xd2(%rdi),%eax
400462: cmp    $0x9,%eax
400465: ja     40048a <lol+0x2e>
400467: mov    %eax,%eax
400469: jmpq   *0x400590(,%rax,8)
400470: lea    (%rsi,%rsi,2),%eax
400473: lea    (%rsi,%rax,4),%eax
400476: retq
400477: mov    $0x4743,%esi
40047c: mov    %esi,%eax
40047e: imul   %esi,%eax
400481: retq
400482: mov    %esi,%eax
400484: sub    %edi,%eax
400486: retq
400487: add    $0xd,%esi
40048a: lea    -0x9(%rsi),%eax
40048d: retq
```



Hint: `0xd2 = 210` and `0x4743 = 18243`.

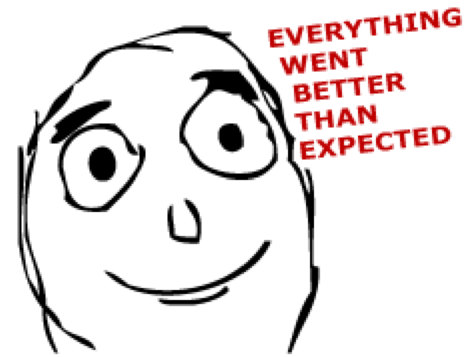
`0x400590:`	________________	`0x400598:`	________________
`0x4005a0:`	________________	`0x4005a8:`	________________
`0x4005b0:`	________________	`0x4005b8:`	________________
`0x4005c0:`	________________	`0x4005c8:`	________________
`0x4005d0:`	________________	`0x4005d8:`	________________

Assembly (6)

```
int lol(int a, int b)
{
    switch(a)
    {
        case 210:
            b *= 13;
            break;
        case 213:
            b = 18243;
            ________
        case 214:
            b *= b;
            break;
        case 216:
        case 218:
            b -= a;
            break;
        case 219:
            b += 13;
            ________
        default:
            b -= 9;
    }

    return b;
}
```

```
40045c <lol>:
40045c: lea     -0xd2(%rdi),%eax
400462: cmp     $0x9,%eax
400465: ja      40048a <lol+0x2e>
400467: mov     %eax,%eax
400469: jmpq    *0x400590(,%rax,8)
400470: lea     (%rsi,%rsi,2),%eax
400473: lea     (%rsi,%rax,4),%eax
400476: retq
400477: mov     $0x4743,%esi
40047c: mov     %esi,%eax
40047e: imul    %esi,%eax
400481: retq
400482: mov     %esi,%eax
400484: sub     %edi,%eax
400486: retq
400487: add     $0xd,%esi
40048a: lea     -0x9(%rsi),%eax
40048d: retq
```



Hint: `0xd2 = 210` and `0x4743 = 18243`.

0x400590:	0x400470	0x400598:	0x40048a
0x4005a0:	0x40048a	0x4005a8:	0x400477
0x4005b0:	0x40047c	0x4005b8:	0x40048a
0x4005c0:	0x400482	0x4005c8:	0x40048a
0x4005d0:	0x400482	0x4005d8:	0x400487

Assembly

Questions?

`struct`s

Data type size v. alignment

- These are not the same!
- For example, on 32-bit x86 Linux, a `double` is eight bytes wide but has four-byte alignment

x86 v. x86-64

- Obviously, pointer width is different
- Some other primitives change widths

Windows v. Linux

- Linux alignment rules are byzantine; refer to the cheat sheet
- Windows rule of thumb: *k*-byte primitives are *k*-byte aligned

`structs` (2)

Aggregate types

- On any system, the alignment requirement of an aggregate type is equal to the longest alignment requirement of its member primitives
- `structs` are not primitives
- Arrays are not primitives

On 32-bit x86 Linux

- `sizeof(struct foo)`: 24
- `sizeof(struct bar)`: 48

```
struct foo
{
    char a;
    int b;
    double c;
    char d[5];
};

struct bar
{
    int a;
    double b;
    long double c;
    struct foo d;
};
```

`structs` (3)

Assembly

- Assume `x` is a C variable whose value is in %eax
- Assume `f` is an instance of `struct foo` whose address is in %edi
- `x = f.d;`
  - lea 16(%edi), %eax
- `x = f.d[0];`
  - mov 16(%edi), %al
- `x = f.d[3];`
  - mov 19(%edi), %al

```
struct foo
{
    char a;
    int b;
    double c;
    char d[5];
};
```

`struct`s

Questions?

Caching

Dimensions: S, E, B

- S: Number of sets
- E: Associativity—number of lines per set
- B: Block size—number of bytes per block (1 block per line)

Dissecting a memory address

- s: log_2(S)
- b: log_2(B)
- t: [number of bits in address] - (s + b)

Caching (2)

Given a 32-bit Linux system that has a 2-way associative cache of size 128 bytes with 32 bytes per block. Long longs are 8 bytes. For all parts, assume that `table` starts at address 0x0.

```
int i;
int j;
long long table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

A. This problem refers to code sample 1. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0								
1								
2								
3								

What is the miss rate of this code sample?

Caching (3)

Given a 32-bit Linux system that has a 2-way associative cache of size 128 bytes with 32 bytes per block. Long longs are 8 bytes. For all parts, assume that `table` starts at address 0x0.

```
int i;
int j;
long long table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

A. This problem refers to code sample 1. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0	m	m	m	m	m	m	m	m
1	m	m	m	m	m	m	m	m
2	m	m	m	m	m	m	m	m
3	m	m	m	m	m	m	m	m

What is the miss rate of this code sample?

1

Caching (4)

```
int i;
int j;
int table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

B. This problem refers to code sample above. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0								
1								
2								
3								

What is the miss rate of this code sample?

Caching (5)

```
int i;
int j;
int table[4][8];
for (j = 0; j < 8; j++) {
  for (i = 0; i < 4; i++) {
    table[i][j] = i + j;
  }
}
```

B. This problem refers to code sample above. In the table below write down in each space whether that element's access will be a hit or a miss. Indicate hits with a 'H' and misses with a 'M'

	0	1	2	3	4	5	6	7
0	m	h	h	h	h	h	h	h
1	m	h	h	h	h	h	h	h
2	m	h	h	h	h	h	h	h
3	m	h	h	h	h	h	h	h

What is the miss rate of this code sample?

1/8

Caching (6)

Food for thought

- Why do caches exist? Why do they help?
- Why does the tag go in the front? Why not the set index?
- Why not have tons of lines per set?
- Why have main memory at all? Why not have 4+ GiB of cache if it is so fast?
- Why is LRU so popular? What does it approximate?
- True or false: A single memory dereference can result in at most one cache miss.
- True or false: A memory address can only ever be mapped to one particular line of a set.

Caching

Questions?

Life

Questions?

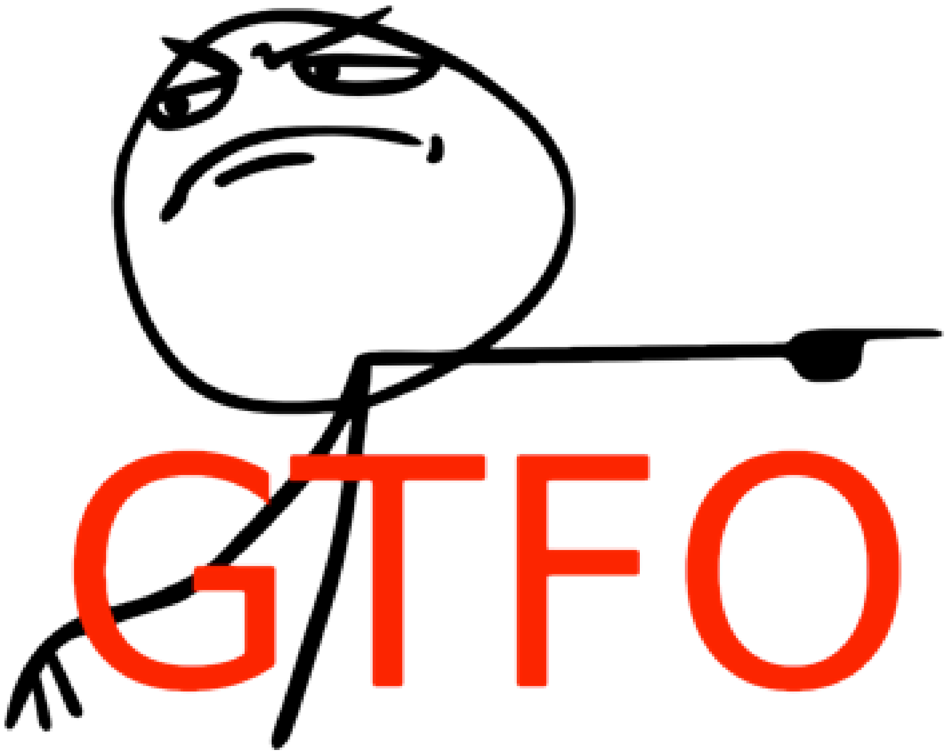
Announcements

Exam

- Grading party Tuesday night
- Scores should be out soon after exam

Office hours

- Canceled Tuesday through Thursday
- Capacity to be doubled during assignment weeks


GTFO