Midterm Review

15-213: Introduction to Computer Systems
October 15, 2012

Instructor:

Agenda

- Midterm tomorrow!
  - Cheat sheet: One 8.5 x 11, front and back
- Review
  - Everything up to caching
- Questions

Brief Overview of Topics

- Labs!
  - We try to reward people who did them well.

- Data Representation
  - Primitive types
  - Floating Point
  - Arrays
  - Structs

Brief Overview of Topics

- Assembly
  - Registers
  - Memory addressing
  - Control flow
  - Stack discipline
  - Translation to C
- Caching
  - Locality
  - The cache itself
  - Miss / hit analysis
  - Blocking

Primitive Types

- Recall datalab – bit operators
- Sizes of primitive types
- Casting
  - Sign extension, truncation
- Signed vs. Unsigned
  - If unsigned and signed are mixed in a single expression, signed values are implicitly cast to unsigned
- Two's Complement

Floating Point

- Sign, Exponent, Mantissa
  - $(-1)^s \times M \times 2^E$
- Bias $(2^{k-1} - 1)$
- Denormalized (exp = 000…000, M = 0.FFF…FFF, E = 1 – bias)
  - Small values close to zero.
- Special Values (exp = 111…111)
  - +/-inf, NaN
- Normalized (M = 1.FFF…FFF, E = exp – bias)
  - Everything else
- Rounding
  - When to round to even

Floating Point - Example

Floating point encoding. Consider the following 5-bit floating point representation based on the IEEE floating point format. This format does not have a sign bit – it can only represent nonnegative numbers.

- There are $k = 3$ exponent bits. The exponent bias is 3.
- There are $n = 2$ fraction bits.

Recall that numeric values are encoded as a value of the form $V = M \times 2^E$, where E is the exponent after biasing, and M is the significand value. The fraction bits encode the significand value M using either a denormalized (exponent field 0) or a normalized representation (exponent field nonzero). The exponent E is given by $E = 1 - \mathit{Bias}$ for denormalized values and $E = e - \mathit{Bias}$ for normalized values, where e is the value of the exponent field `exp` interpreted as an unsigned number.

Below, you are given some decimal values, and your task it to encode them in floating point format. In addition, you should give the rounded value of the encoded floating point number. To get credit, you must give these as whole numbers (e.g., 17) or as fractions in reduced form (e.g., $3/4$). Any rounding of the significand is based on ***round-to-even***, which rounds an unrepresentable value that lies halfway between two representable values to the nearest even representable value.

Value	Floating Point Bits	Rounded value
$9/32$	`001 00`	1/4
1		
12		
11		
$1/8$		
$7/32$		

Structs

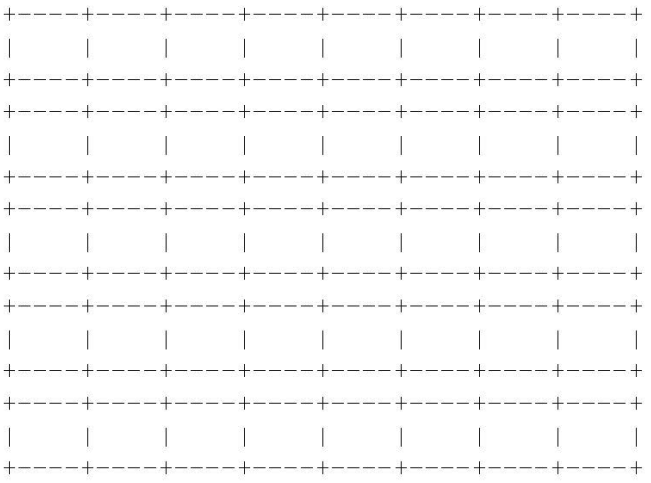
- Padding and alignment
- Data type size vs. Alignment
  - On 32-bit x86 Linux, a double is eight bytes wide by has four-byte alignment
- x86 vs. x86-64
  - Pointer width
  - Some other primitives
- Alignment rules (Windows vs. Linux)
- What accessing looks like in assembly

Struct - Example

Take the struct below compiled on Linux 32-bit:

```
struct my_struct {
    short b;
    int x;
    short s;
    long z;
    char c[5];
    long long a;
    char q;
}
```

1. Please lay out the struct in memory below (each cell is 1 byte). Please shade in boxes used for padding.



Assembly

- Registers
  - Stack pointer: %esp
  - Frame pointer: %ebp
  - Program counter: %eip
  - Return value: %eax
  - General purpose
    - Caller vs. Callee saved
- Equivalent registers in x86-64
- Differences between x86 and x86-64

Assembly

- Instructions
  - Addressing: lea, mov
  - Arithmetic: add, sub, imul, idiv
  - Stack manipulation: push, pop, leave
  - Conditionals: cmp, test
  - Local jumps: jmp, je, jg, jle, etc.
  - Procedure calls: call, ret
- What do they do?
- How do certain calls change the stack?

Assembly - Example

Use the x86_64 assembly to fill in the C function below

```
0x0000000000400498 <mystery+0>:    push   %r13
0x000000000040049a <mystery+2>:    push   %r12
0x000000000040049c <mystery+4>:    push   %rbp
0x000000000040049d <mystery+5>:    push   %rbx
0x000000000040049e <mystery+6>:    sub    $0x8,%rsp
0x00000000004004a2 <mystery+10>:   mov    %rdi,%r13
0x00000000004004a5 <mystery+13>:   mov    %edx,%r12d
0x00000000004004a8 <mystery+16>:   test   %edx,%edx
0x00000000004004aa <mystery+18>:   jle    0x4004c7 <mystery+47>
0x00000000004004ac <mystery+20>:   mov    %rsi,%rbx
0x00000000004004af <mystery+23>:   mov    $0x0,%ebp
0x00000000004004b4 <mystery+28>:   mov    (%rbx),%edi
0x00000000004004b6 <mystery+30>:   callq  *%r13
0x00000000004004b9 <mystery+33>:   mov    %eax,(%rbx)
0x00000000004004bb <mystery+35>:   add    $0x1,%ebp
0x00000000004004be <mystery+38>:   add    $0x4,%rbx
0x00000000004004c2 <mystery+42>:   cmp    %r12d,%ebp
0x00000000004004c5 <mystery+45>:   jne    0x4004b4 <mystery+28>
0x00000000004004c7 <mystery+47>:   add    $0x8,%rsp
0x00000000004004cb <mystery+51>:   pop    %rbx
0x00000000004004cc <mystery+52>:   pop    %rbp
0x00000000004004cd <mystery+53>:   pop    %r12
0x00000000004004cf <mystery+55>:   pop    %r13
0x00000000004004d1 <mystery+57>:   retq
```

```
void mystery(int (*funcP)(int), int a[], int n) {



}
```

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)
- Why do caches exist? When do they help?
- Why have main memory at all, if cache is so fast?
- Why use LRU for eviction?
- Recall arrays are accessed in row-major order…

Caching - Example

Cache operation. Assume a cache memory with the following properties:

- The cache size (C) is 512 bytes (contains 512 data bytes)
- The cache uses an LRU (least-recently used) policy for eviction.
- The cache is initially empty.

Suppose that for the following sequence of addresses sent to the cache, **0, 2, 4, 8, 16, 32**, the hit rate is **0.33**. Then what is the block size (B) of the cache?

A. $B = 4$ bytes

B. $B = 8$ bytes

C. $B = 16$ bytes

D. None of the above.

Questions/Advice

- Relax!
- Work past exams.
- Email us (15-213-staff@cs.cmu.edu)