CS356: Discussion #3

Floating-Point Operations

Marco Paolieri (paolieri@usc.edu)

USC University of Southern California

Schedule: Exams and Assignments

- Week 1: Binary Representation **HW0**
- Week 2: Integer Operations
- **Week 3: Floating-Point Operations** **Data Lab 1**
- Week 4: Assembly
- Week 5: Assembly **Data Lab 2**
- Week 6: Assembly **Bomb Lab**
- Week 7: **Exam I** (Oct. 2) and Security Vulnerabilities
- Week 8: Memory Organization
- Week 9: Caching **Attack Lab**
- Week 10: Virtual Memory
- Week 11: Dynamic Memory Allocation and Linking
- Week 12: Processor Organization and **Exam II** (Nov. 8) **Cache Lab**
- Week 13: Processor Organization
- Week 14: Code Optimization and **Thanksgiving**
- Week 15: Cache Coherency and Review **Allocation Lab**
- Week 16: Study Days and **Final** (Dec. 6)

Data Lab 2

- **Deadline**: Monday Sep. 17th, 2018 at 11:59pm PDT
- **Steps**
  - Read the instructions at http://bytes.usc.edu/cs356/assignments/datalab-2.pdf
  - You already cloned your class repository inside the VM
    `$ git clone git@github.com:usc-csci356-fall2018/hw-`**`username`**`.git`
  - Now, you need to pull the new assignment
    `$ cd hw-username;` **`git pull`**`; cd proj-2`
  - Inside the file **`bits.c`**, complete the body of the functions **`byteSwap`**, **`ezThreeFourths`**, **`float_abs`**, **`float_half`**, **`float_f2i`**
  - Check violations (**`./dlc bits.c`**), correctness (**`make; ./btest`**) and your final score (**`./driver.pl`**)
  - Commit, push, submit full commit hash at http://bytes.usc.edu/cs356/assignments

Data Lab 2: What to implement

Integer Problems: Only 1-byte constants (`0xFA`), no loops (`for`, `while`), no conditionals (`if`), no macros (`INT_MAX`), no comparisons (`x==y`, `x>y`), no `unsigned int`, no operators `- && ||`, only `! ~ & | ^ + << >>`

- `int byteSwap(int x, int n, int m)`: swap bytes `n` and `m`
- `int ezThreeFourths(int x)`: return `x*3/4` (beware of rounding)

Floating-point Problems: 4-byte constants (`0x12345678`), loops (`for`, `while`), conditionals (`if`), comparisons (`x==y`, `x>y`), operators `- && ||`, but no macros (`INT_MAX`), no `float` types or operations.

The `unsigned` input and output are the **bit-level equivalent** of 32-bit `float`s

- `unsigned float_abs(unsigned x)`: return `abs(f)` (**NaNs unchanged**)
- `unsigned float_half(unsigned x)`: return `f/2` (**NaNs unchanged**)
- `int float_f2i(unsigned x)`: return `(int)f`
    - For x out of range (including NaN and infinity), return `0x80000000`

Exercise: Reset Bytes

Write a function `reset_bytes(int x, int n, int m)` that resets bytes of **x** at positions **n** and **m** (possible input positions: 0, 1, 2, 3) using only `<<`, `~`, `&`

```
#include <stdio.h>

int reset_bytes(int x, int n, int m) {
    int reset_n = ~(0xFF << (n << 3));  // shift 0xFF by n*8 bits
    int reset_m = ~(0xFF << (m << 3));  // shift 0xFF by m*8 bits
    return x & reset_n & reset_m;
}

int main() {
    printf("%08X [DD0000AA]\n", reset_bytes(0xDDCCBBAA,1,2));
    printf("%08X [00CCBBAA]\n", reset_bytes(0xDDCCBBAA,3,3));
    printf("%08X [DD00BB00]\n", reset_bytes(0xDDCCBBAA,2,0));
}
```

Exercise: Multiply using shifts

Write a function `void mult(int x)` that multiplies x

- by 6, using 2 shifts and 1 add/sub;
- by 31, using 1 shifts and 1 add/sub;
- by -6, using 2 shifts and 1 add/sub;
- by 55, using 2 shifts and 2 add/sub.

```c
#include <stdio.h>
static void mult(int x) {  printf("\nx = %d\n", x);
    printf(" 6 * x =     (8-2) * x = %d\n", (x << 3) - (x << 1));
    printf("31 * x =    (32-1) * x = %d\n", (x << 5) - x);
    printf("-6 * x =     (2-8) * x = %d\n", (x << 1) - (x << 3));
    printf("55 * x = (64-8-1) * x = %d\n", (x << 6)-(x << 3)-x);
}
int main() {
    mult(0); mult(1); mult(-1); mult(10); mult(-100); mult(7);
}
```

Dividing Two's-Complement by Powers of 2

- **x / 2^k** when x ≥ 0: **`x >> k`**
- **x / 2^k** when x < 0: **`(x + (1 << k) - 1) >> k`**
  - Consider `(-3)/2` with `signed char` (1 byte)
  - `0xFD >> 1` gives `0xFE` which is -2 (instead, **`-3/2` gives -1 in C**)
  - `x >> k` rounds toward -∞ for negative x, not toward 0 (unlike `x/y` in C)
  - In other words, it computes $\lfloor x / 2^k \rfloor$ instead of $\lceil x / 2^k \rceil$ for x < 0
  - But, it is always true that $\lfloor (x + (y-1)) / y \rfloor = \lceil x / y \rceil$
  - **Biasing**: add **2^k - 1** before the shift when x < 0

k	Bias	−12,340 + Bias (Binary)	>> k (Binary)	Decimal	$-12340/2^k$
0	0	1100111111001100	1100111111001100	−12340	−12340.0
1	1	1100111111001101	1110011111100110	−6170	−6170.0
4	15	1100111111011011	1111110011111101	−771	−771.25
8	255	1101000011001011	1111111111010000	−48	−48.203125

Figure 2.29 **Dividing two's-complement numbers by powers of 2.** By adding a bias before the right shift, the result is rounded toward zero.

Exercise: Divide by Eight

Write a function `divide_by_8(int x)` that returns `x/8` using only `>>`, `+`, `&`

```c
#include <stdio.h>

int divide_by_8(int x) {
    int bias = (x >> 31) & 7;
    return (x + bias) >> 3;
}
int main() {
    int minOdd = 0x80000001;
    printf("%d [0]\n", divide_by_8(0));
    printf("%d [0]\n", divide_by_8(7));
    printf("%d [0]\n", divide_by_8(-7));
    printf("%08X [%08X]\n", divide_by_8(minOdd), minOdd/8);
}
```

Fixed Point vs Floating Point

Fixed-point format: a fixed number of bits is reserved for the fractional part.

- Example: use unsigned chars (1 byte) and reserve 2 bits for fractional part.

8				7			
1	0	0	0	0	**1**	**1**	**1**
32	16	8	4	2	1	0.5	0.25

0x87 represents **33.75**

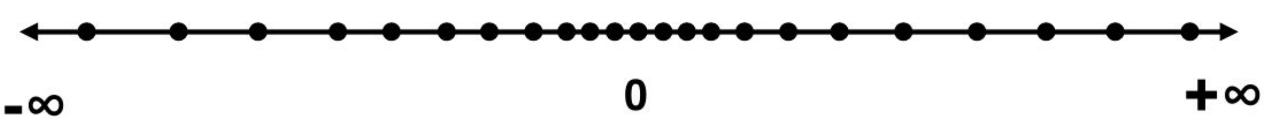
The range for unsigned chars was 0 to 255.

By reserving 2 bits for the fractions part:

- The range is now [0, 63.75] (0x00 to 0xFF)
- We can represent fractional values with increments of 0.25

Floating-point format: the position of the binary point can change.

- Flexible trade-off between **range** and **precision**



IEEE 754 Standard: 32-bit

Binary32 Format (`float`)

sign	exponent	fraction
1 bit	8 bits	23 bits

- **Decimal value**: $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent} - 127}$
- **Decimal range**: (7 significant decimal digits) $\times 10^{\pm 38}$
- **Exponent** encodes values [-126, 127] as unsigned integers with bias
- Exponent of all 0's reserved for:
  - Zeros: `0x00000000` (0.0), `0x80000000` (-0.0)
  - Denormalized values: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{-126}$ (nonzero fraction)
- Exponent of all 1's reserved for:
  - Infinity: `0x7F800000` (∞), `0xFF800000` ($-\infty$)
  - NaN: with any nonzero fraction

IEEE 754 Standard: 64-bit

Binary64 Format (`double`)

sign	exponent	fraction
1 bit	11 bits	52 bits

- **Decimal value**: $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - 1023}}$
- **Decimal range**: ($\approx$ 16 significant decimal digits) $\times 10^{\pm 308}$
- **Exponent** encodes values [-1022, 1023] as unsigned integers with bias
- Exponent of all 0's reserved for:
  - Zeros: `0x0000000000000000` (0.0), `0x8000000000000000` (-0.0)
  - Denormalized values: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{-1022}$ (nonzero fraction)
- Exponent of all 1's reserved for:
  - Infinity: `0x7FF0000000000000` (∞), `0xFFF0000000000000` ($-\infty$)
  - NaN: any nonzero fraction

Other formats, same patterns (from CS:APP)

Description	Bit representation	Exponent e	E	2^E	Fraction f	M	Value $2^E \times M$	V	Decimal
Zero	0 0000 000	0	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
Smallest pos.	0 0000 001	0	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	−6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
	0 0000 011	0	−6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
Largest denorm.	0 0000 111	0	−6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
Smallest norm.	0 0001 000	1	−6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	−6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
	0 0110 110	6	−1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	−1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
One	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
Largest norm.	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	0 1111 000	—	—	—	—	—	—	∞	—

Figure 2.34 **Example nonnegative values for 8-bit floating-point format.** There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.

Bias: 2^{k-1}-1 (0111...1)

Same bit patterns for

- Zero
- Smallest denormalized
- Largest denormalized
- Smallest normalized
- One
- Largest normalized
- Infinity

To **negate**, just flip the sign bit (except NaN)

Rounding and Casting in C

The IEEE 754 standard defines four **rounding modes**:

- **Round to nearest, ties to even**: default rounding in C for `float/double` ops
- **Round towards zero** (truncation): used to cast `float/double` to `int`
- **Round up** (ceiling): go towards +∞ (gives an upper bound)
- **Round down** (floor): go towards -∞ (gives a lower bound)

Floating point operations

- Addition and subtraction are **not associative**
  - Add small-magnitude numbers before large-magnitude ones
- Multiplication and division are **not associative** (**nor distributive**)
  - Control magnitude with divisions (if possible)

    `(big1 * big2) / (big3 * big4)` overflows on first multiplication

    `1/big3 * 1/big4 * big1 * big2` underflows on first multiplication

    `(big1 / big3) * (big2 / big4)` is likely better
- Comparison should use `fabs(x-y) < epsilon` instead of `x==y`
- **Instead:** 2's complement is associative (even after overflow), can use `x==y`

Exercise: Return 1

Write a function `unsigned one()` that returns the bit-level value of `1.0f`

```c
#include <stdio.h>

unsigned int one() { return 0x3f800000; }

// union used to print the bit-level encoding of a float
union converter { float f; unsigned int i; };
unsigned int f2b(float x) {
    union converter c; c.f = x;
    return c.i;
}
int main() {
    printf("1.0: %08X [%08X]\n", one(), f2b(1.0f));
}
```

Exercise: Return 2

Write a function `unsigned two()` that returns the bit-level value of `2.0f`

```c
#include <stdio.h>

unsigned int two() { return 0x40000000; }

// union used to print the bit-level encoding of a float
union converter { float f; unsigned int i; };
unsigned int f2b(float x) {
    union converter c; c.f = x;
    return c.i;
}
int main() {
    printf("2.0: %08X [%08X]\n", two(), f2b(2.0f));
}
```

Variations

- What about the bit-level value of `-1.0f` and `-2.0f` ?
- What about the bit-level value of `4.0f`? And `0.1f`?

These bit-level values will be the unsigned input of your functions.

Note that the assignment directory includes the **`fshow`** command:

```
$ ./fshow 2.0

Floating point value 2
Bit Representation 0x40000000,
  sign = 0, exponent = 0x80, fraction = 0x000000

Normalized.  +1.0000000000 X 2^(1)
```

Exercise: Floating-point Sign

Write a function `int sign(unsigned int x)` that returns the sign of `x` as 1/-1

```
int sign(unsigned int x) {
    return (x & 0x80000000) ? -1 : 1;
}

int main() {
    printf(" Sign of       2.0: %2d [ 1]\n", sign(f2b( 2.0f)));
    printf(" Sign of      -1.0: %2d [-1]\n", sign(f2b(-1.0f)));
    printf(" Sign of       0.0: %2d [ 1]\n", sign(f2b( 0.0f)));
    printf(" Sign of      -0.0: %2d [-1]\n", sign(f2b(-0.0f)));
    printf(" Sign of  1.0/0.0: %2d [ 1]\n", sign(f2b(1.0f/0.0f)));
    printf(" Sign of  1.0/-.0: %2d [-1]\n", sign(f2b(1.0f/-.0f)));
}
```

Exercise: Extract Exponent

Write a function `int exponent(unsigned int x)` that returns the exponent of `x` (as is, including the bias).

```
int exponent(unsigned int x) {
    return (x >> 23) & 0xFF;
}

int main() {
    printf("    2.0: %3d [128]\n", exponent(f2b( 2.0f)));
    printf("   -1.0: %3d [127]\n", exponent(f2b(-1.0f)));
    printf("    0.0: %3d [0]\n",   exponent(f2b( 0.0f)));
    printf("   -0.0: %3d [0]\n",   exponent(f2b(-0.0f)));
    printf("1.0/0.0: %3d [255]\n", exponent(f2b(1.0f/0.0f)));
    printf("1.0/-.0: %3d [255]\n", exponent(f2b(1.0f/-.0f)));
}
```

Exercise: Extract Fraction

Write a function `int fraction(unsigned int x)` returning the fraction of `x`, including the implicit leading bit equal to 1 (ignore denormalized numbers).

```
int fraction(unsigned int x) {
    return (x & 0x007FFFFF) | 0x00800000;
}

int main() {
    printf("    2.0: %08X [0x00800000]\n", fraction(f2b( 2.0f)));
    printf("   -1.0: %08X [0x00800000]\n", fraction(f2b(-1.0f)));
    printf("    2.5: %08X [0x00A00000]\n", fraction(f2b( 2.5f)));
}
```

Exercise: Detect Floating-point Zero

Write a function `int is_zero(unsigned int x)` returning 1 if x is 0.0 or -0.0, and 0 otherwise. (Trivial solution under relaxed assignment rules!)

```
int is_zero(unsigned int x) {
    return (x == 0x00000000 || x == 0x80000000) ? 1 : 0;
}


int main() {
    printf("    0.0: %d [1]\n", is_zero(f2b( 0.0f)));
    printf("   -0.0: %d [1]\n", is_zero(f2b(-0.0f)));
    printf("    1.0: %d [0]\n", is_zero(f2b( 1.0f)));
    printf("   -1.0: %d [0]\n", is_zero(f2b(-1.0f)));
    unsigned int denormalized = f2b(1.4e-45f);
    printf("1.4e-45: %d [0]\n", is_zero(denormalized));
    printf("1.4e-45 is %08X [0x00000001]\n", denormalized);
}
```

Exercise: Detect Denormalized Numbers

Write a function `int denorm(unsigned int x)` that returns 1 if `x` is denormalized, and 0 otherwise.

```
int denorm(unsigned int x) {
    return !((x >> 23) & 0xFF) && (x & 0x007FFFFF);
}

int main() {
    printf("    0.0: %d [0]\n", denorm(f2b( 0.0f)));
    printf("   -0.0: %d [0]\n", denorm(f2b(-0.0f)));
    printf("    1.0: %d [0]\n", denorm(f2b( 1.0f)));
    printf("   -1.0: %d [0]\n", denorm(f2b(-1.0f)));
    unsigned int denormalized = f2b(1.4e-45f);
    printf("1.4e-45: %d [1]\n", denorm(denormalized));
    printf("1.4e-45 is %08X\n", denormalized);
}
```

Assignment: Divide `float` by 2

Function prototype: `unsigned float_half(unsigned uf)`

sign	exponent	fraction
1 bit	8 bits	23 bits

Float Value

- Normalized: $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - 127}}$
- Denormalized: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{-126}$

Exponent of all 0's reserved for zeros and denormalized values.
Exponent of all 1's reserved for: `0x7F800000` (+∞), `0xFF800000` (-∞), NaN.

What happens after division by 2?

- Nothing for +0.0, -0.0, +∞, -∞, NaN
- Can we decrease the exponent by 1?
  - What if the exponent becomes `0x00`? 1.(fraction) vs 0.(fraction)
- For denormalized numbers, how do we divide by 2?
- Do we need a round-up term? When? ...00? ...01? ...10? ...11?

Assignment: Cast `float` to `int`

Function prototype: `int float_f2i(unsigned uf)`

sign	exponent	fraction
1 bit	8 bits	23 bits

Float Value

- Normalized: $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - 127}}$
- Denormalized: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{-126}$

What happens after cast to `int`?

- For NaN, +∞, -∞ and out-of-range values, return `0x80000000`
  - Which values of the exponent make the **`float`** out-of-range for **`int`**?
  - Maximum **`int`** $(2^{31} - 1)$ is < $1.(\text{fraction}) \times 2^{\text{exponent - 127}}$ for exponent > ???
- Denormalized values are < 2^{-126}. What happens to them?
- What happens to other numbers with exponent < 127?
- For normalized values in range, how to compute $1.(\text{fraction}) \times 2^{\text{exponent - 127}}$?
- How should we handle negative floats? (Using `(-x)` is allowed.)