Data Lab: Manipulating Bits

CSE251, Spring 2019

Recitation 1: Wed, March 13th, 2019

Changmin Yi

ulistar93@unist.ac.kr

Reference : CMU 15-213: Intro to Computer Systems Fall 2015
Recitation 3 - Dhruven Shah, Ben Spinelli

Welcome to Recitation

- Lab info
  - Assigned: Mar 11 (Mon), Due: Mar 17, 11:59PM

- TA's
  - Changmin Yi (ulistar93@unist.ac.kr, Wed 17:30~18:30 @106-605)
  - Anvar Alisheri (alisher@unist.ac.kr, Thu 19:30~20:30 @106-709)

- We'll cover:
  - Some notices
  - Briefly recap of contents from class
  - Look around the Data Lab description and some hints

Notices

- Typo in pdf, Table 1,
  bitOr(x,y) means “x | y using only ~ and &”
  not “x & y using only ~ and &”

- In 3.3 Floating-Point Operations,
  it says return a NaN value as 0x7FC00000.
  But DO NOT handle the case artificially.
  Even though, you should consider the case with those variables as an input.
  -> more detail in later

Agenda

- How do I Data Lab?
- Integers
  - Encoding Byte Values
  - Endianness
- Floating point
  - Binary fractions
  - IEEE standard
  - Example problem

Encoding Byte Values

- Byte = 8 bits
  - Binary 00000000_2 to 11111111_2
  - Decimal: 0_{10} to 255_{10}
  - Hexadecimal 00_{16} to FF_{16}
    - Base 16 number representation
    - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
    - Write $FA1D37B_{16}$ in C as
      - 0xFA1D37B
      - 0xfa1d37b

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

$$15213(10): \underbrace{0011}_{3}\ \underbrace{1011}_{B}\ \underbrace{0110}_{6}\ \underbrace{1101}_{D}\ (2)$$

Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8
`unsigned int`	4		

Bit-Level Operations in C

- Operations
  - & Intersection
  - | Union
  - ^ Symmetric difference
  - ~ Complement

In a one bit level, ~ and ! work same thing

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

! or ~	
0	1
1	0

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  01000001      01111101      00111100      10101010
```

But it's different when it has multi bits

~0x41 → 0xBE
!0x41 → 0x00

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior
  - Shift amount < 0 or ≥ word size

Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

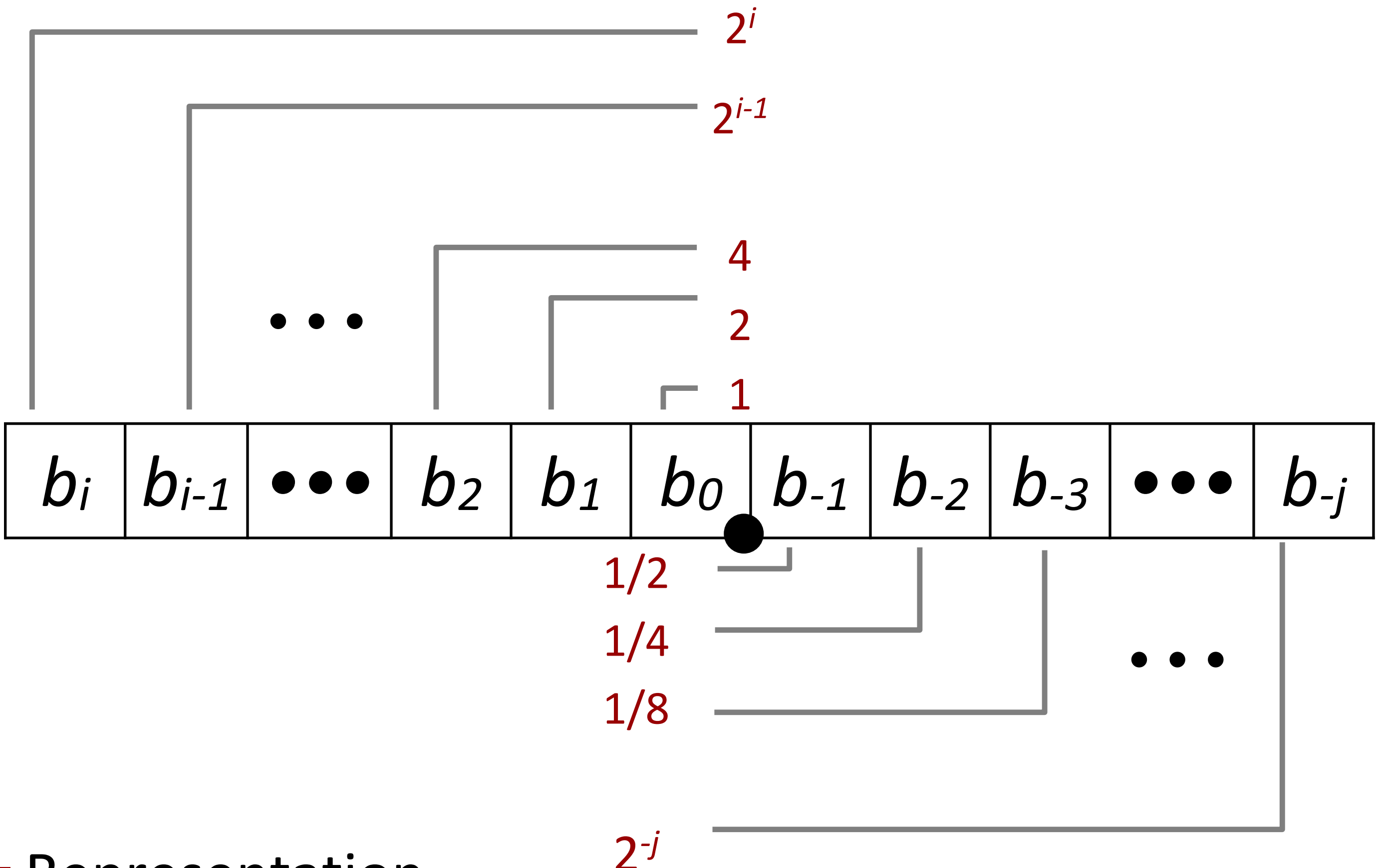
Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Unsigned & Signed Numeric Values

X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **Expression containing signed and unsigned int:**
  `int` is cast to `unsigned`

Floating Point – Fractions in Binary



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i} b_k \times 2^k$

Floating Point Representation

Example:
$15213_{10} = (-1)^0 \times 1.1101101101101_2 \times 2^{13}$

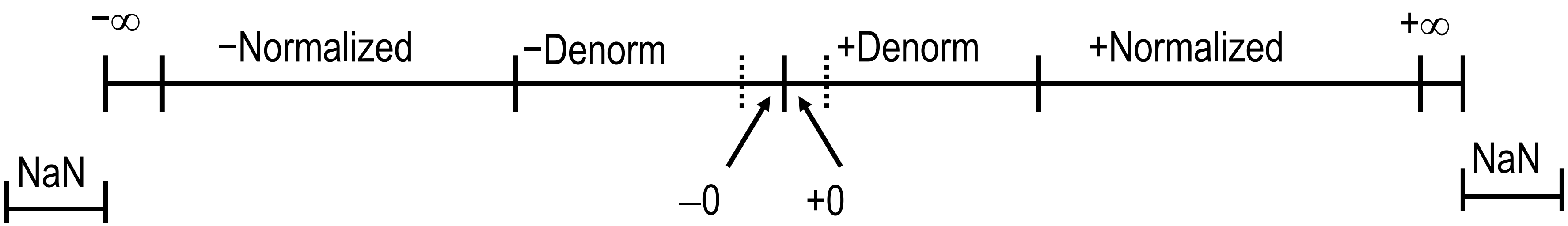
- Numerical Form:

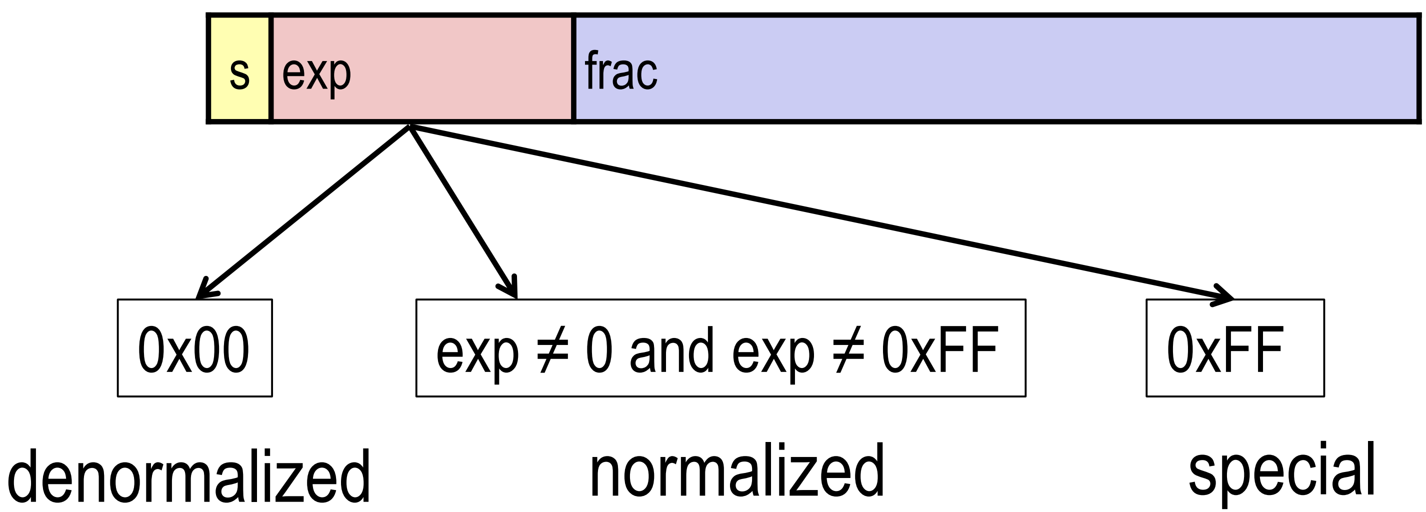
$$(-1)^s \, \boldsymbol{M} \; 2^{\boldsymbol{E}}$$

  - **Sign bit *s*** determines whether number is negative or positive
  - **Significand *M*** normally a fractional value in range [1.0,2.0).
  - **Exponent *E*** weights value by power of two
- Encoding
  - MSB s is sign bit ***s***
  - exp field encodes ***E*** (but is not equal to E)
  - frac field encodes ***M*** (but is not equal to M)
- Single precision: 32 bits (IEEE Standard)

s	exp	frac
1	8-bits	23-bits

Visualization: Floating Point Encodings





Normalized Encoding Example

$v = (-1)^s M\, 2^E$
$E = \mathbf{exp} - Bias$

- Value: `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

- Significand

M = $1.\underline{1101101101101}_2$
frac= $\underline{1101101101101}0000000000_2$

- Exponent

E = 13
$Bias$ = 127
exp = 140 = 10001100_2

- Result:

s	exp	frac
0	10001100	11011011011010000000000

Normalized Decoding Example

$v = (-1)^s M\, 2^E$
$E = \texttt{exp} - Bias$

$Bias = 2^{k-1} - 1 = 127$

float: **0xC0A00000**

binary: 1100 0000 1010 0000 0000 0000 0000 0000

1	1000 0001	010 0000 0000 0000 0000 0000
1	8-bits	23-bits

E = **exp** – Bias = 129 – 127 = 2 (decimal)

S = 1 -> negative number

M = 1.010 0000 0000 0000 0000 0000

= 1 + 1/4 = 1.25

$v = (-1)^s M\, 2^E = (-1)^1 * 1.25 * 2^2 = -5$

Denormalized Values

$v = (-1)^s M\, 2^E$
$E = 1 - Bias$

- Condition: exp = 000…0

- Exponent value: $E = 1 - \text{Bias}$ (instead of **`exp`** – *Bias*) (why?)
- Significand coded with implied leading 0: $M = 0.\text{xxx…x}_2$
  - **`xxx…x`**: bits of **`frac`**
- Cases
  - **`exp`** = 000…0, **`frac`** = 000…0
    - Represents **zero** value
    - Note distinct values: +0 and –0 (why?)
  - **`exp`** = 000…0, **`frac`** ≠ 000…0
    - Numbers closest to 0.0
    - Equispaced

Special Values

- Condition: **`exp`** = **`111…1`**

- Case: **`exp`** = **`111…1`**, **`frac`** = **`000…0`**
  - **Represents value ∞ (infinity)**
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- Case: **`exp`** = **`111…1`**, **`frac`** ≠ **`000…0`**
  - **Not-a-Number (NaN)**
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty \times 0$

Round to even rule

- Round to even
  - Why? Avoid statistical bias of rounding up or down on half.
  - How? Like this:

1.0100_2	truncate	1.01_2
1.0101_2	below half; round down	1.01_2
1.0110_2	interesting case; round to even	1.10_2
1.0111_2	above half; round up	1.10_2
1.1000_2	truncate	1.10_2
1.1001_2	below half; round down	1.10_2
1.1010_2	Interesting case; round to even	1.10_2
1.1011_2	above half; round up	1.11_2
1.1100_2	truncate	1.11_2

Rounding

1.BBGRXXX

Guard bit: LSB of result

Sticky bit: OR of remaining bits

Round bit: 1st bit removed

- Round up conditions
  - Round = 1, Sticky = 1 → > 0.5
  - Guard = 1, Round = 1, Sticky = 0 → Round to even

Value	*Exp*	*Fraction*	*GRS*	*Incr?*	*Rounded*	*Value*
8	3	1.0000000	000	N	1.000	8
13	3	1.1010000	100	N	1.101	13
8.5	3	1.0001000	010	N	1.000	8
9.5	3	1.0011000	110	Y	1.010	10
8.625	3	1.0001010	011	Y	1.001	9
15.75	3	1.1111100	111	Y	10.000	16

Floating Point – Example 32bit IEEE Std

- In this data lab, ./fshow will be help you

- Value = 1
- Bit Representation
  0x3f800000 = 0011 1111 1000 0000 ... 0000
  23bit
  sign = 0, exponent = 0x7f, fraction = 000...0

- Value = ??? = 8.816207631e-39 = +0.7500000000 X 2^(-126)
- Bit Representation
  0x00600000 = 0000 0000 0110 0000 ... 0000
  sign = 0, exponent = 0x00, fraction = 110...0

Data Lab.

- Step 1. Bit Manipulations

Name	Description	Rating	Max Ops
bitOr(x,y)	x & y using only ~ and &	1	8
getByte(x,n)	Get byte n from x.	2	6
logicalShift(x,n)	Shift right logical.	3	20
bitCount(x)	Count the number of 1's in x.	4	40
bang(x)	Compute !n without using ! operator.	4	12

Table 1: Bit-Level Manipulation Functions.

- Step 2. Two's Complement Arithmetic

Name	Description	Rating	Max Ops
tmin()	Most negative two's complement integer	1	4
fitsBits(x,n)	Does x fit in n bits?	2	15
negate(x)	-x without negation	2	5
isPositive(x)	x > 0?	2	8
isLess(x,y)	x < y?	3	24
isPower2(x)	is x a power of 2?	4	20
sign(x)	is x positive? Negative? Or zero?	2	10

Table 2: Arithmetic Functions

- Step 3. Floating-Point Operations

Name	Description	Rating	Max Ops
floatNegate(uf)	Compute -f	2	10
floatInt2Float(x)	Compute (float) x	4	30
floatIsLess(uf,ug)	Compute uf < ug	3	30

Table 3: Floating-Point Functions. Value f is the floating-point number having the same bit representation as the unsigned integer uf.

Be careful !

- Write C like it's 1989
  - Declare variable at top of function
  - Make sure closing brace ("}") is in 1st column
  - We won't be using the dlc compiler for later labs
- Be careful of operator precedence
  - Do you know what order `~a+1+b*c<<3*2` will execute in?
  - Neither do I. Use parentheses: `(~a)+1+(b*(c<<3)*2)`
- Any declaration must appear in a block before any statement that is not a declaration
- Integer constants should be in 0 through 255 (0xFF)
- **PLEASE READ THE COMMENT IN THE CODE CAREFULLY**

Data Lab.

- ./btest

```
[cs        @uni06 datalab]$ ./btest
Score    Rating  Errors  Function
 1        1       0       bitOr
 1        1       0       tmin
 2        2       0       negate
 2        2       0       getByte
 4        4       0       bitCount
 3        3       0       logicalShift
 2        2       0       isPositive
 3        3       0       isLess
 4        4       0       bang
 4        4       0       isPower2
 2        2       0       fitsBits
 4        4       0       floatInt2Float
 2        2       0       floatNegate
 3        3       0       floatIsLess
 2        2       0       sign
Total points: 39/39
```

Data Lab.

- ./dlc bits.c

```
[cs[redacted]@uni06 datalab]$ ./dlc -e bits.c
dlc:bits.c:189:bitOr: 4 operators
dlc:bits.c:200:tmin: 1 operators
dlc:bits.c:216:negate: 2 operators
dlc:bits.c:232:getByte: 5 operators
dlc:bits.c:300:bitCount: 39 operators
dlc:bits.c:316:logicalShift: 7 operators
dlc:bits.c:329:isPositive: 5 operators
dlc:bits.c:352:isLess: 23 operators
dlc:bits.c:370:bang: 12 operators
dlc:bits.c:395:isPower2: 11 operators
dlc:bits.c:412:fitsBits: 6 operators
dlc:bits.c:487:floatInt2Float: Warning: 39 operators exceeds max of 30
dlc:bits.c:517:floatNegate: 9 operators
dlc:bits.c:567:floatIsLess: 30 operators
dlc:bits.c:585:sign: 9 operators
dlc:bits.c:602:twosComp2SignMag: 10 operators
Total points: 39/39
```

Data Lab.

- ./driver.pl

```
5. Running './dlc -e' to get operator count of each function.

Correctness Results     Perf Results
Points  Rating  Errors  Points  Ops     Puzzle
1       1       0       2       4       bitOr
1       1       0       2       1       tmin
2       2       0       2       2       negate
2       2       0       2       5       getByte
4       4       0       2       39      bitCount
3       3       0       2       7       logicalShift
2       2       0       2       5       isPositive
3       3       0       2       23      isLess
4       4       0       2       12      bang
4       4       0       2       11      isPower2
2       2       0       2       6       fitsBits
4       4       0       0       39      floatInt2Float
2       2       0       2       9       floatNegate
3       3       0       2       30      floatIsLess
2       2       0       2       9       sign

Score = 67/69 [39/39 Corr + 28/30 Perf] (212 total operators)
```

Data Lab.

- ./fshow

```
[cs          @uni06 datalab]$ ./fshow 0x3fc00000

Floating point value 1.5
Bit Representation 0x3fc00000, sign = 0, exponent = 0x7f, fraction = 0x400000
Normalized.  +1.5000000000 X 2^(0)
[cs          @uni06 datalab]$ ./fshow 0x3f800000

Floating point value 1
Bit Representation 0x3f800000, sign = 0, exponent = 0x7f, fraction = 0x000000
Normalized.  +1.0000000000 X 2^(0)
[cs          @uni06 datalab]$ ./fshow 0x00600000

Floating point value 8.816207631e-39
Bit Representation 0x00600000, sign = 0, exponent = 0x00, fraction = 0x600000
Denormalized.  +0.7500000000 X 2^(-126)
```

Questions?