Bitwise Operations

Instructor: Jeeho Ryoo

Bitwise operations

Logical NOT: !

A	!A
T	F
F	T

Bitwise operations

Logical NOT: !

A	!A
non-zero	0
zero	1

Bitwise operations

Bitwise NOT: ~

A	~A
00	11
01	10
10	01
11	00

Bitwise operations

Logical AND: &&

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

Bitwise operations

Logical AND: &&

A	B	A && B
non-zero	non-zero	1
non-zero	zero	0
zero	non-zero	0
zero	zero	0

Bitwise operations

Bitwise AND: &

A	B	A & B
1	1	1
1	0	0
0	1	0
0	0	0

Example: 1100 & 0101 = 0100

Bitwise operations

Logical OR: ||

A	B	A || B
T	T	T
T	F	T
F	T	T
F	F	F

Bitwise operations

Logical OR: ||

A	B	A || B
non-zero	non-zero	1
non-zero	zero	1
zero	non-zero	1
zero	zero	0

Bitwise operations

Bitwise OR: |

A	B	A | B
1	1	1
1	0	1
0	1	1
0	0	0

Example: 1100 | 0101 = 1101

Bitwise operations

Logical XOR:

A	B	A XOR B
T	T	F
T	F	T
F	T	T
F	F	F

Bitwise operations

Bitwise XOR: ^

A	B	A ^ B
1	1	0
1	0	1
0	1	1
0	0	0

Example: 1100 ^ 0101 = 1001

Exercise:
Can you write a function that checks whether the third bit (from right) of a number is 1 or 0? Signature should be:

```
int thirdBitFromRight(int n);
```

Bitwise operations

Example:

```
#include <stdio.h>
int thirdBitFromRight(int n) {
    int mask = 4;
    return (n & mask) == 4;
}
void runTest(int n) {
    printf("n = %d, thirdBitFromRight = %d\n", n,
thirdBitFromRight(n));
}
int main() {
    runTest(4);
    runTest(15);
    runTest(0);
    runTest(11);
    runTest(-1);
    return 0;
}
```

Bitwise operations

Bit Masking:

A bit mask is an integer whose binary representation is intended to combine with another value using &, | or ^ to extract or set a particular bit or set of bits.

For example mask = 4 in the code from previous slide.

Bitwise operations

Another exercise:

Write a function that turns “on” the third bit (from right):

Practice Problems

MEDIUM – Write a function that takes a number and turns ***on*** the first and third binary digits (from right) for this number. Here are some examples:

$8 = (1000)_2 \rightarrow 13 = (1101)_2$
$0 = (0)_2 \rightarrow 5 = (101)_2$
$17 = (10001)_2 \rightarrow 21 = (10101)_2$
$29 = (11101)_2 \rightarrow 29 = (11101)_2$

Encoding

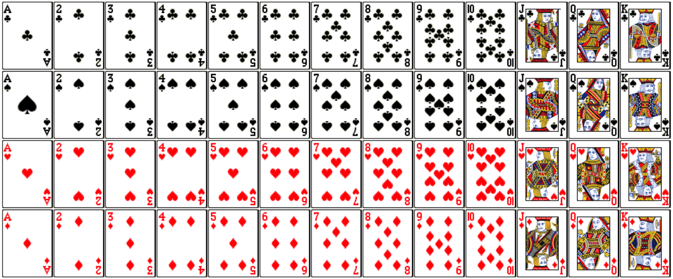
Encode a standard deck of playing cards

52 cards in 4 suits

- How do we encode suits, face cards?

What operations do we want to make easy to implement?

- Which is the higher value card?
- Are they the same suit?



Boolean Algebra

Developed by George Boole in 19th Century

- Algebraic representation of logic (True → 1, False → 0)
- AND: `A&B=1` when both A is 1 and B is 1
- OR: `A|B=1` when either A is 1 or B is 1
- XOR: `A^B=1` when either A is 1 or B is 1, but not both
- NOT: `~A=1` when A is 0 and vice-versa
- DeMorgan's Law: `~(A|B) = ~A & ~B`
  `~(A&B) = ~A | ~B`

AND

&	0	1
0	**0**	**0**
1	**0**	**1**

OR

|	0	1
0	**0**	**1**
1	**1**	**1**

XOR

^	0	1
0	**0**	**1**
1	**1**	**0**

NOT

~	
0	**1**
1	**0**

General Boolean Algebras

Operate on bit vectors

- Operations applied bitwise
- All of the properties of Boolean algebra apply

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
```

Examples of useful operations:

$$x \mathbin{\char`^} x = 0$$

```
  01010101
^ 01010101
  00000000
```

$$x \mid 1 = 1, \qquad x \mid 0 = x$$

```
  01010101
| 11110000
  11110101
```

Two possible representations

1) 1 bit per card (52): bit corresponding to card set to 1

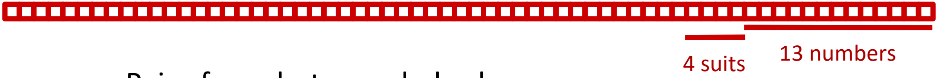
"One-hot" encoding (similar to set notation)

Drawbacks:

Hard to compare values and suits

Large number of bits required

2) 1 bit per suit (4), 1 bit per number (13): 2 bits set



Pair of one-hot encoded values

Easier to compare suits and values, but still lots of bits used

Two better representations

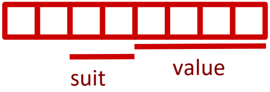
3) Binary encoding of all 52 cards – only 6 bits needed

$$2^6 = 64 \geq 52$$

Fits in one byte (smaller than one-hot encodings)

How can we make value and suit comparisons easier?

4) Separate binary encodings of suit (2 bits) and value (4 bits)



♣	00
♦	01
♥	10
♠	11

Also fits in one byte, and easy to do comparisons

K	Q	J	. . .	3	2	A
1101	1100	1011	. . .	0011	0010	0001

Compare Card Suits

```c
char hand[5];        // represents a 5-card hand
char card1, card2;   // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( isSameSuit(card1, card2) ) { ... }
```

```c
#define SUIT_MASK  0x30

int isSameSuit(char card1, char card2) {
  return (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

SUIT_MASK = 0x30 =

0	0	1	1	0	0	0	0

suit value

Compare Card Values

```
char hand[5];        // represents a 5-card hand
char card1, card2;   // two cards to compare
card1 = hand[0];
card2 = hand[1];
...
if ( greaterValue(card1, card2) ) { ... }
```

```
#define VALUE_MASK  0x0F

int greaterValue(char card1, char card2) {
  return ((unsigned char)(card1 & VALUE_MASK) >
          (unsigned char)(card2 & VALUE_MASK));
}
```

VALUE_MASK = 0x0F =

0	0	0	0	1	1	1	1

suit

value

Encoding Integers

The hardware (and C) supports two flavors of integers

unsigned – only the non-negatives

signed – both negatives and non-negatives

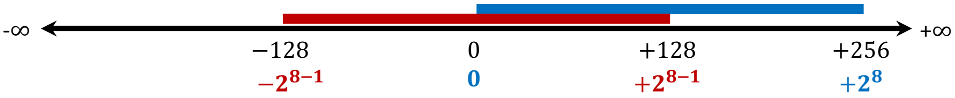
Cannot represent all integers with w bits

Only 2^w distinct bit patterns

Unsigned values: $0 \ldots 2^w - 1$

Signed values: $-2^{(w-1)} \ldots 0 \ldots 2^{(w-1)} - 1$

Example: 8-bit integers (*e.g.* `char`)



Unsigned Integers

Unsigned values follow the standard base 2 system

$$b_7b_6b_5b_4b_3b_2b_1b_0$$
$$= b_7 2^7 + b_6 2^6 + \cdots + b_1 2^1 + b_0 2^0$$

Add and subtract using the normal “carry” and “borrow” rules, just in binary

```
  63
+  8
----
  71
```

```
 00111111
+00001000
---------
 01000111
```

Useful formula: N ones in a row = $2^{\mathrm{N}} - 1$

How would you make *signed* integers?

Sign and Magnitude

Designate the high-order bit (MSB) as the “sign bit”

`sign=0`: positive numbers; `sign=1`: negative numbers

Benefits:

Using MSB as sign bit matches positive numbers with unsigned

All zeros encoding is still = 0

Examples (8 bits):

0x00 = 00000000_2 is non-negative, because the sign bit is 0

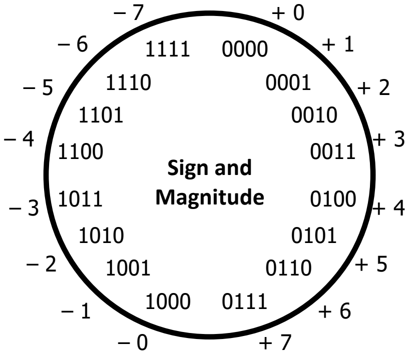
0x7F = 01111111_2 is non-negative ($+127_{10}$)

0x85 = 10000101_2 is negative (-5_{10})

0x80 = 10000000_2 is negative...

Sign and Magnitude

MSB is the sign bit, rest of the bits are magnitude
Drawbacks?

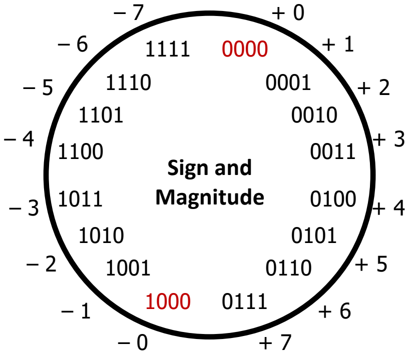


Sign and Magnitude

MSB is the sign bit, rest of the bits are magnitude

Drawbacks:

Two representations of 0 (bad for checking equality)



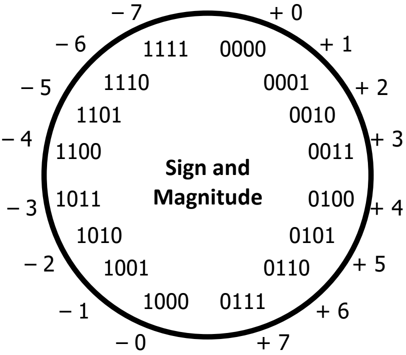
Sign and Magnitude

MSB is the sign bit, rest of the bits are magnitude

Drawbacks:

- Two representations of 0 (bad for checking equality)
- Arithmetic is cumbersome
  - Example: `4-3 != 4+(-3)`

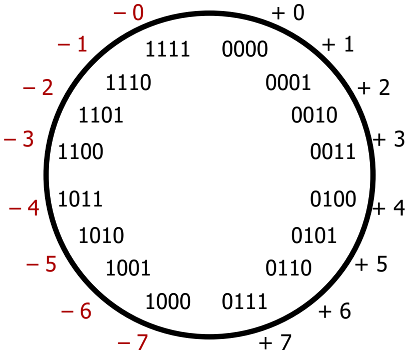
4	0100
- 3	- 0011
1	0001



Two’s Complement

Let’s fix these problems:

1) “Flip” negative encodings so incrementing works



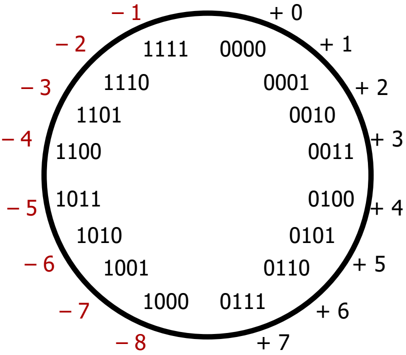
Two's Complement

Let's fix these problems:

1) "Flip" negative encodings so incrementing works
2) "Shift" negative numbers to eliminate –0

MSB *still* indicates sign!

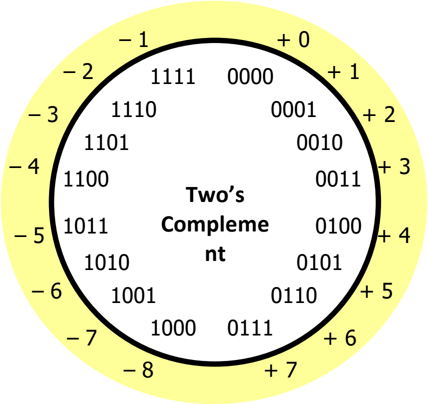
This is why we represent one more negative than positive number (-2^{N-1} to $2^{N-1}-1$)



Why Two's Complement is So Great

Roughly same number of (+) and (–) numbers
Positive number encodings match unsigned
Simple arithmetic (x + -y = x – y)
Single zero
All zeros encoding = 0

Simple negation procedure:
Get negative representation of any integer by taking bitwise complement and then adding one!
`( ~x + 1 == -x )`



Two's Complement Arithmetic

The same addition procedure works for both unsigned and two's complement integers

Simplifies hardware: only one algorithm for addition

Algorithm: simple addition, discard the highest carry bit

Called modular addition: result is sum *modulo* 2^w

4-bit Examples:

4 0100 +3 +0011	-4 1100 +3 +0011	4 0100 -3 +1101
=7	=-1	=1

Why Does Two's Complement Work?

For all integers x, we want:

$$\begin{array}{r} \textit{bit representation of } x \\ + \textit{ bit representation of } -x \\ \hline 0 \end{array}$$

(ignoring the carry-out bit)

Start with an observation: what's `x + ~x`?

```
  00000001        00000010        11000011
+ 11111110      + 11111101      + 00111100
----------      ----------      ----------
  11111111        11111111        11111111
```

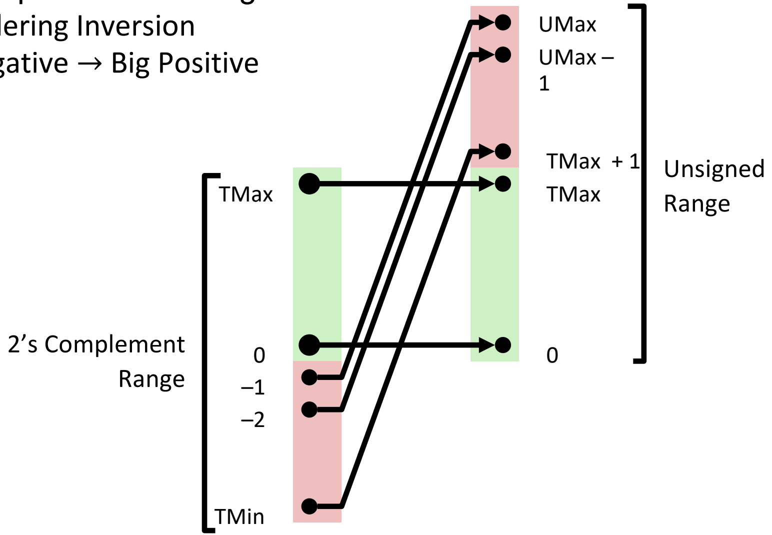
So what's `x + ~x + 1`

Signed/Unsigned Conversion Visualized

Two's Complement → Unsigned

Ordering Inversion

Negative → Big Positive



Values To Remember

Unsigned Values

UMin= `0b00…0`

= 0

UMax =

`0b11…1`

= $2^w - 1$

Two's Complement Values

TMin = `0b10…0`

= $-2^{(w-1)}$

TMax = `0b01…1`

= $2^{(w-1)} - 1$

-1 = `0b11…1`

Example: Values for $w = 64$

	Decimal	Hex
UMax	18,446,744,073,709,551,615	FF FF FF FF FF FF FF FF
TMax	9,223,372,036,854,775,807	7F FF FF FF FF FF FF FF
TMin	-9,223,372,036,854,775,808	80 00 00 00 00 00 00 00
-1	-1	FF FF FF FF FF FF FF FF
0	0	00 00 00 00 00 00 00 00

Arithmetic Overflow

Bits	Unsigned	Signed
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

When a calculation produces a result that can't be represented in the current encoding scheme

- Integer range limited by fixed width
- Can occur in both the positive and negative directions

C and Java ignore overflow exceptions

- You end up with a bad value in your program and no warning/indication... oops!

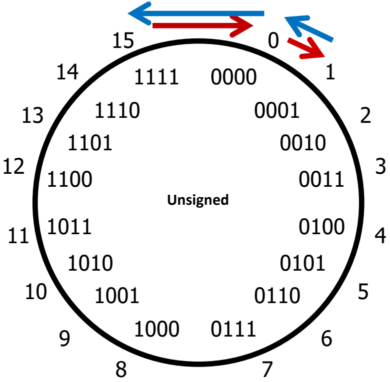
Overflow: Unsigned

Addition: drop carry bit (-2^N)

$$\begin{array}{r} 15 \\ +\ \ 2 \\ \hline \cancel{17} \\ 1 \end{array} \qquad \begin{array}{r} 1111 \\ +\ \ 0010 \\ \hline \cancel{1}0001 \end{array}$$

Subtraction: borrow $(+2^N)$

$$\begin{array}{r} 1 \\ -\ \ 2 \\ \hline \cancel{-1} \\ 15 \end{array} \qquad \begin{array}{r} 10001 \\ -\ \ 0010 \\ \hline 1111 \end{array}$$



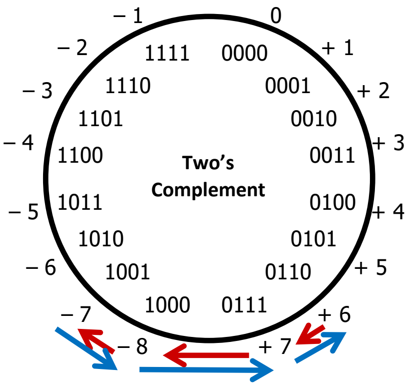
Overflow: Two's Complement

Addition: (+) + (+) = (–) result?

$$\begin{array}{r} 6 \\ +\ \ 3 \\ \hline \cancel{9} \\ -7 \end{array} \qquad \begin{array}{r} 0110 \\ +\ \ 0011 \\ \hline 1001 \end{array}$$

Subtraction: (–) + (–) = (+)?

$$\begin{array}{r} -7 \\ -\ \ 3 \\ \hline -\cancel{10} \\ 6 \end{array} \qquad \begin{array}{r} 1001 \\ -\ \ 0011 \\ \hline 0110 \end{array}$$



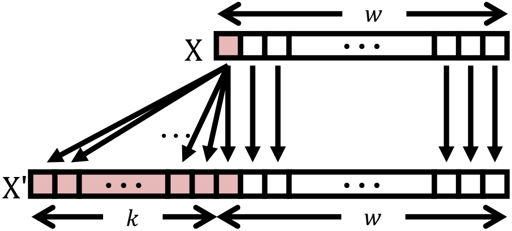
Sign Extension

Task: Given a w-bit signed integer X, convert it to w+k-bit signed integer X' *with the same value*

Rule: Add k copies of sign bit

Let x_i be the i-th digit of X in binary

$$\mathrm{X}' = \underbrace{x_{w-1}, \dots, x_{w-1}}_{k \text{ copies of MSB}}, \underbrace{x_{w-1}, x_{w-2}, \dots, x_1, x_0}_{\text{original X}}$$



Sign Extension Example

Convert from smaller to larger integral data types
C (and Java) automatically performs sign extension when converting to larger types

```
short int x = 
12345;
int       ix = (int)
x;
short int y =  -
12345;
int       iy = (int)
y;
```

Var	Decimal	Hex	Binary
x	12345	30 39	00110000 00111001
ix	12345	00 00 30 39	00000000 00000000 00110000 00111001
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

Shift Operations

Left shift (`x<<n`) bit vector `x` by `n` positions

- Throw away (drop) extra bits on left
- Fill with `0`s on right

Right shift (`x>>n`) bit-vector `x` by `n` positions

- Throw away (drop) extra bits on right
- Logical shift (for unsigned values)
  - Fill with `0`s on left
- Arithmetic shift (for signed values)
  - Replicate most significant bit on left
  - Maintains sign of `x`

Shift Operations

	x	0010 0010
	x<<3	0001 0000
logical:	x>>2	0000 1000
arithmetic:	x>>2	0000 1000

	x	1010 0010
	x<<3	0001 0000
logical:	x>>2	0010 1000
arithmetic:	x>>2	1110 1000

- Left shift (`x<<n`)
  - Fill with `0`s on right
- Right shift (`x>>n`)
  - Logical shift (for unsigned values)
    - Fill with 0s on left
  - Arithmetic shift (for signed values)
    - Replicate most significant bit on left
- Notes:
  - Shifts by `n<0` or `n≥w` (bit width of x) are *undefined*
  - **In C:** behavior of `>>` is determined by compiler
    - depends on data type of `x` (signed/unsigned)

Shifting Arithmetic?

What are the following computing?

`x>>n`

`0b 0100 >> 1 = 0b 0010`

`0b 0100 >> 2 = 0b 0001`

Divide by 2^n

`x<<n`

`0b 0001 << 1 = 0b 0010`

`0b 0001 << 2 = 0b 0100`

Multiply by 2^n

Shifting is faster than general multiply and divide operations

Left Shifting Arithmetic 8-bit Example

No difference in left shift operation for unsigned and signed numbers (just manipulates bits)

Difference comes during interpretation: $x*2^n$?

			Signed	Unsigned
`x = 25;`	`00011001`	=	25	25
`L1=x<<2;`	`0001100100`	=	100	100
`L2=x<<3;`	`00011001000`	=	-56	200
`L3=x<<4;`	`000110010000`	=	-112	144

Right Shifting Arithmetic 8-bit Examples

Reminder: C operator `>>` does *logical* shift on unsigned values and *arithmetic* shift on signed values

Logical Shift: `x/2`^{n}?

```
xu = 240u;    11110000        = 240
R1u=xu>>3;    00011110000     =  30
R2u=xu>>5;    0000011110000   =   7
```

Right Shifting Arithmetic 8-bit Examples

Reminder: C operator >> does *logical* shift on unsigned values and *arithmetic* shift on signed values

Arithmetic Shift: $x/2^n$?

```
xs = -16;      11110000           = -16
R1s=xu>>3;     11111110000        =  -2
R2s=xu>>5;     1111111110000      =  -1
```