# Data Representation

# Representation of data in a computer

Two conditions:

1. Presence of a voltage – “1”
2. Absence of a voltage – “0”

# Kinds of data

- Numbers – signed, unsigned, integers, floating point, complex, rational, irrational, …
- Text – characters, strings, …
- Images – pixels, colors, …
- Sound
- Instructions
- …

# Hardware & Software Data Types

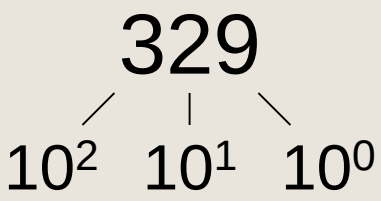
Some data types are supported directly by the instruction set architecture.

Other data types are supported by interpreting values in the software that we write.

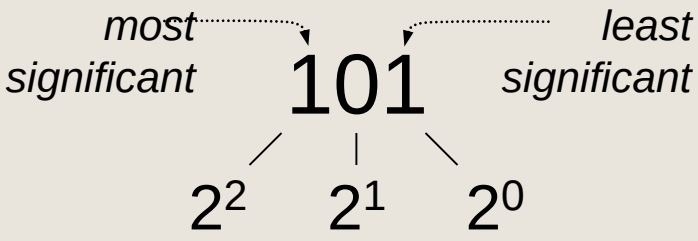
The compiler takes care of this.

# Unsigned Integers

# Unsigned Integers




3x100 + 2x10 + 9x1 = 329




1x4 + 0x2 + 1x1 = 5

# Unsigned Integers

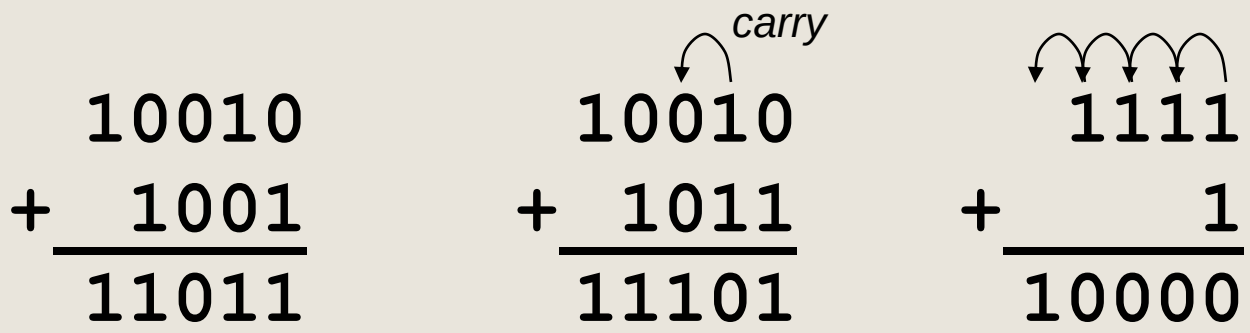
An *n*-bit unsigned integer represents $2^n$ values: from 0 to $2^n$-1.

| $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 1 | 0 | 0 | 4 |
| 1 | 0 | 1 | 5 |
| 1 | 1 | 0 | 6 |
| 1 | 1 | 1 | 7 |

# Unsigned Binary Arithmetic

Base-2 addition – just like base-10!

- add from right to left, propagating carry



# Converting Positive Decimal to Binary

## First Method: *Division*

1. Divide by two – remainder is least significant bit.
2. Keep dividing by two until answer is zero, writing remainders from right to left.
3. Bits that haven't been filled – fill with zeros

$X = 104_{ten}$

| | | | |
|---|---|---|---|
| 104/2 | = | 52 r0 | *bit 0* |
| 52/2 | = | 26 r0 | *bit 1* |
| 26/2 | = | 13 r0 | *bit 2* |
| 13/2 | = | 6 r1 | *bit 3* |
| 6/2 | = | 3 r0 | *bit 4* |
| 3/2 | = | 1 r1 | *bit 5* |
| 1/2 | = | 0 r1 | *bit 6* |

$X = 01101000_{two}$

# Converting Positive Decimal to Binary

| $n$ | $2^n$ |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |
| 5 | 32 |
| 6 | 64 |
| 7 | 128 |
| 8 | 256 |
| 9 | 512 |
| 10 | 1024 |

Second Method: *Subtract Powers of Two*

1. Subtract largest power of two less than or equal to number.
2. Put a one in the corresponding bit position.
3. Keep subtracting until result is zero.
4. Bits that haven't been filled – fill with zeros

$X = 104_{ten}$

| | | | |
|---|---|---|---|
| 104 - 64 | = | 40 | *bit 6* |
| 40 - 32 | = | 8 | *bit 5* |
| 8 - 8 | = | 0 | *bit 3* |

$X = 01101000_{two}$

# Hexadecimal Notation

We can use other bases, for example,

hexadecimal (base-16) numbers are often used for memory addresses

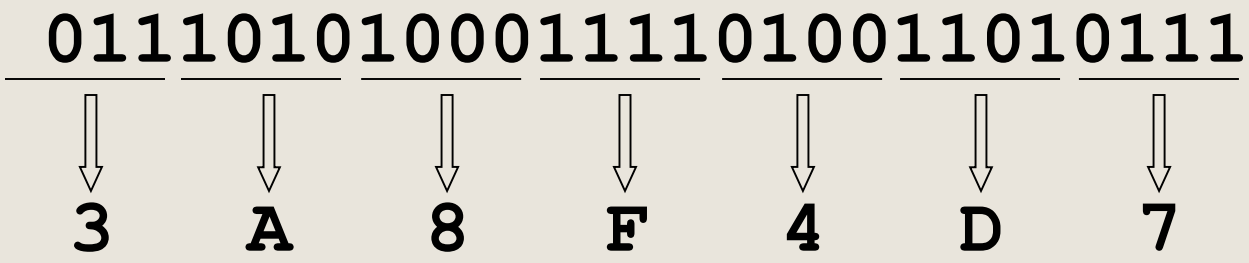
| Binary | Hex | Decimal |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |

| Binary | Hex | Decimal |
|---|---|---|
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | A | 10 |
| 1011 | B | 11 |
| 1100 | C | 12 |
| 1101 | D | 13 |
| 1110 | E | 14 |
| 1111 | F | 15 |

# Converting from Binary to Hexadecimal

Every four bits is a hex digit.

- start grouping from right-hand side



# Signed Integers

# Signed Integers

With n bits, we have $2^n$ distinct values.

- assign about half to positive integers (1 through $2^{n-1}$-1) and about half to negative (- $2^{n-1}$-1 through -1)
- that leaves two values: one for 0, and one extra

# Signed Integers

Positive integers

- just like unsigned – zero in *most significant* (MS) bit
  00101 = 5

Negative integers

- sign-magnitude – set MS bit to show negative, other bits are the same as unsigned
  10101 = -5
- one's complement – flip every bit to represent negative
  11010 = -5

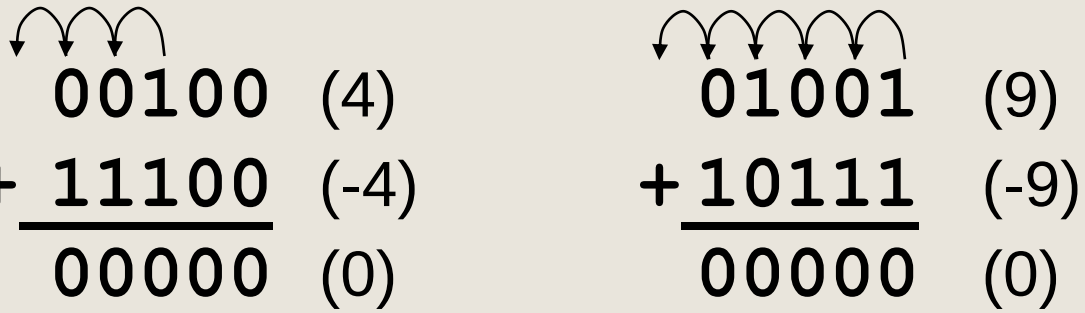
Problems with sign-magnitude and 1's complement

- two representations of zero (+0 and –0)
- arithmetic circuits are complex

# Two's Complement Representation

*Two's complement* representation developed to make circuits easy for arithmetic.

For each positive number (X),
assign value to its negative (-X),
such that X + (-X) = 0 with "normal" addition,
ignoring carry out



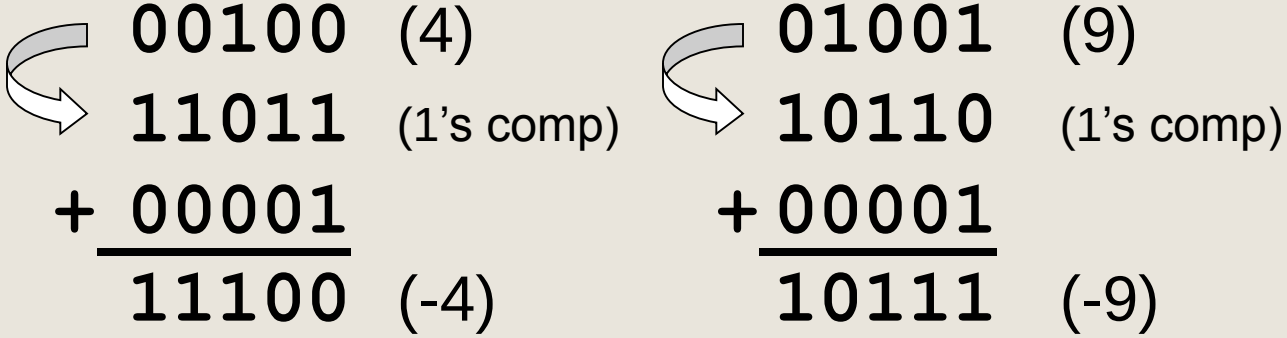
# Two's Complement Conversion

If number is positive or zero,

- normal binary representation, zeroes in upper bit(s)

If number is negative,

- start with positive number
- flip every bit (i.e., take the one's complement)
- then add one



# Two's Complement Signed Integers

MS bit is sign bit with weight $-2^{n-1}$.

Range of an n-bit number: $-2^{n-1}$ through $2^{n-1} - 1$.

- The most negative number ($-2^{n-1}$) has no positive counterpart.

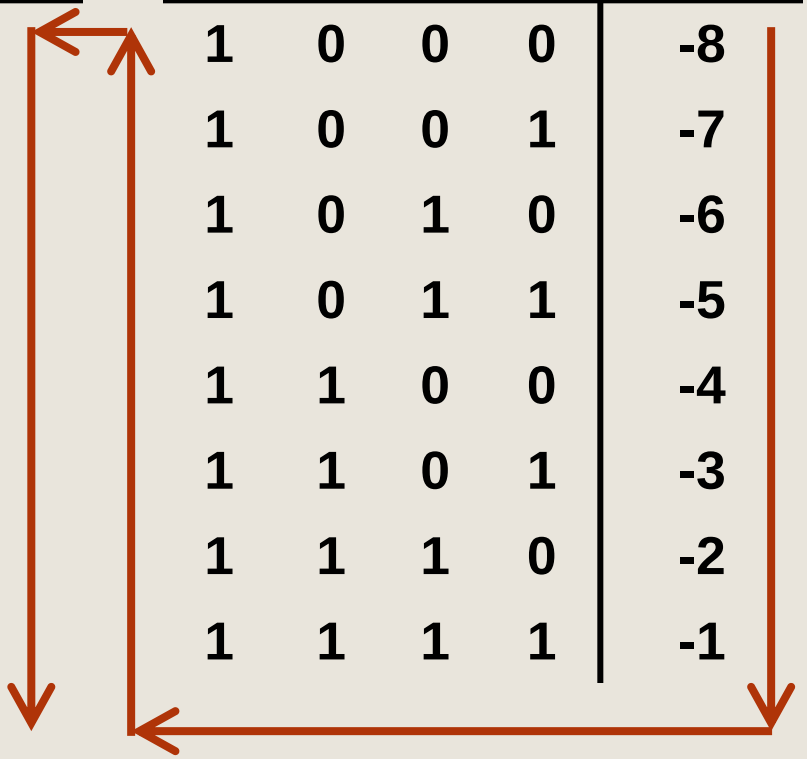
| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | -8 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | 1 | -1 |

# Cyclic structure allows simple addition

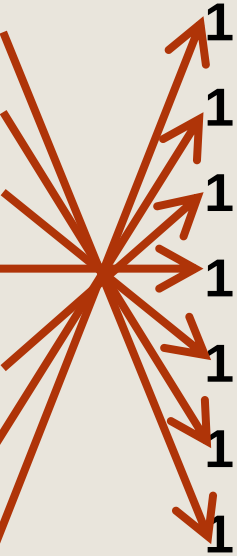
| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | -8 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | 1 | -1 |



# Cyclic structure was achieved using Two's Complement: negation for reverse ordering +1 for starting from +1

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |

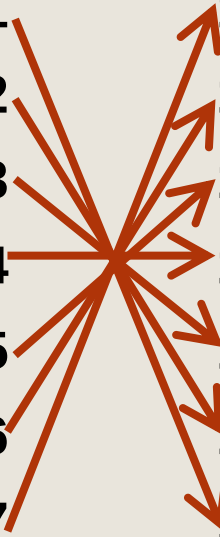


| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | -8 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | 1 | -1 |

# Cyclic structure was achieved using Two's Complement:
# negation for reverse ordering
# +1 for starting from +1

Why -8 and not +8, +8 also cyclic ?

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |



| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | -8 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | 1 | -1 |

# Cyclic structure was achieved using Two's Complement:
# negation for reverse ordering
# +1 for starting from +1

Why -8 and not +8, +8 also cyclic ?

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |

| $-2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | -8 |
| 1 | 0 | 0 | 1 | -7 |
| 1 | 0 | 1 | 0 | -6 |
| 1 | 0 | 1 | 1 | -5 |
| 1 | 1 | 0 | 0 | -4 |
| 1 | 1 | 0 | 1 | -3 |
| 1 | 1 | 1 | 0 | -2 |
| 1 | 1 | 1 | 1 | -1 |

Easy to check if negative

# Arithmetic Operations

# Arithmetic Operations

a data type includes *representation* and *operations*.

We now have a good representation for signed integers, so let's look at some arithmetic operations:

- Addition
- Negation
- Subtraction
- Sign Extension
- Shifts

We'll also look at overflow conditions for addition.

Multiplication, division, etc., can be built from these basic operations.

# Addition

2’s comp. addition is just binary addition.

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that sum fits in n-bit 2’s comp. representation

```
  01101000  (104)
+ 11110000  (-16)
----------
  01011000  (88)
```

*Assuming 8-bit 2’s complement numbers.*

# Negation

2’s comp negation is just taking the 2’s comp…

# Subtraction

Negate subtrahend (2nd no.) and add.

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that difference fits in n-bit 2's comp. representation

```
  01101000  (104)
- 00010000  (16)
----------
  01101000  (104)
+ 11110000  (-16)
----------
  01011000  (88)
```

*Assuming 8-bit 2's complement numbers.*

# Sign Extension

To add two numbers, we must represent them with the same number of bits.

If we just pad with zeroes on the left:

| **4-bit** | | **8-bit** | |
|---|---|---|---|
| `0100` | (4) | `00000100` | (still 4) |
| `1100` | (-4) | `00001100` | (12, not -4) |

Instead, replicate the MS bit -- the sign bit:

| **4-bit** | | **8-bit** | |
|---|---|---|---|
| `0100` | (4) | `00000100` | (still 4) |
| `1100` | (-4) | `11111100` | (still -4) |

# Sign Extension

| **4-bit** | | **8-bit** | |
|---|---|---|---|
| `0100` | (4) | `00000100` | (still 4) |
| `1100` | (-4) | `11111100` | (still -4) |

Let $n$ be the small number of bits.

Let $N$ be the large number of bits.

Instead of adding $-2^{n-1}$ due to the last bit we added:

$$-2^{N-1} + \sum_{i=n-1}^{N-2} 2^i = -2^{N-1} + \sum_{i=0}^{N-2} 2^i - \sum_{i=0}^{n-2} 2^i =$$

$$-2^{N-1} + 2^{N-1} - 1 - 2^{n-1} + 1 = -2^{n-1}$$

Geometric sum and negative geometric sum

# Sign Extension

| **4-bit** | | **8-bit** | |
|---|---|---|---|
| `0100` | (4) | `00000100` | (still 4) |
| `1100` | (-4) | `11111100` | (still -4) |

Let $n$ be the small number of bits.

Let $N$ be the large number of bits.

Instead of adding $-2^{n-1}$ due to the last bit we added:

$$-2^{N-1} + \sum_{i=n-1}^{N-2} 2^i = -2^{N-1} + \sum_{i=0}^{N-2} 2^i - \sum_{i=0}^{n-2} 2^i =$$

$$-2^{N-1} + 2^{N-1} - 1 - 2^{n-1} + 1 = -2^{n-1}$$

Geometric sum and negative geometric sum

# Overflow

If operands are too big, then sum cannot be represented as an *n*-bit 2's comp number.

```
  01000  (8)          11000  (-8)
+ 01001  (9)        + 10111  (-9)
-------              -------
  10001  (-15)         01111  (+15)
```

We have overflow if:

- signs of both operands are the same, and
- sign of sum is different.

Another test -- easy for hardware:

- carry into MS bit does not equal carry out

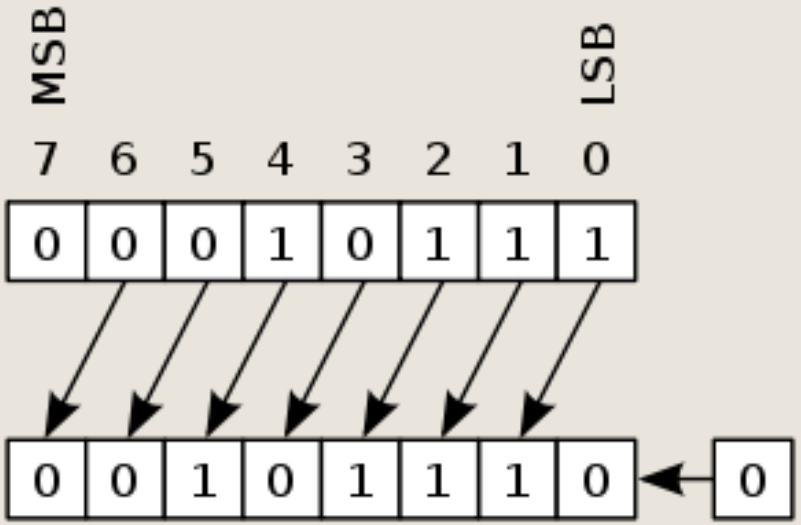
# Overflow

Can also happen with negation,

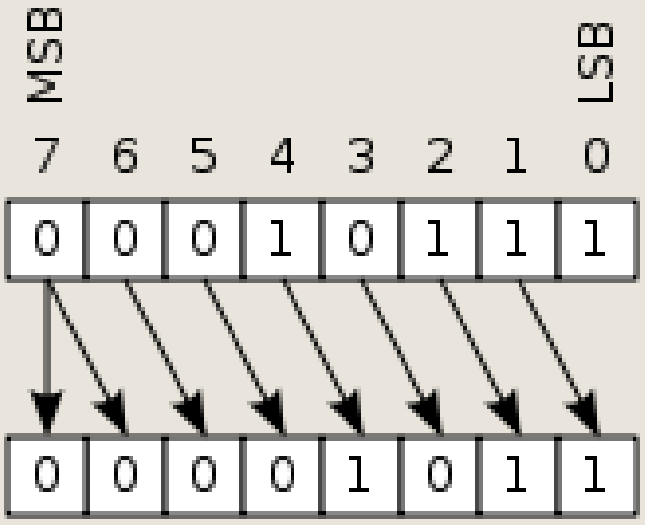
the negation of -4 with 3 bits is:

```
  100    (-4)
  011 (1's comp)
+ 001
  ---
  100    (-4)
```

# Shifts Operations




Left Logical Shift




Right Arithmetic Shift

# Shifts Operations

<< Shift left

- Similar to multiply by 2

>> Shift right

- Similar to divide by 2

Similar but not equivalent since there might be differences due to rounding strategies.

Also, according to C standard right shift of signed numbers is implementation defined. In practice for signed numbers usually arithmetic shift is done.

**a**<<**n** == fast multiply the variable **a** by $2^{n}$

# Logical Operations

# Bitwise Operations in Integers

Operations on logical TRUE or FALSE

- two states -- takes one bit to represent: TRUE=1, FALSE=0

| A | B | A AND B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| A | B | A OR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| A | NOT A |
|---|---|
| 0 | 1 |
| 1 | 0 |

View *n*-bit number as a collection of *n* logical values

- operation applied to each bit independently

# Bitwise Operations in Integers

AND

- useful for clearing bits
  - AND with zero = 0
  - AND with one = no change

```
  11000101
AND 00001111
------------
  00000101
```

OR

- useful for setting bits
  - OR with zero = no change
  - OR with one = 1

```
  11000101
OR 00001111
------------
  11001111
```

NOT

- unary operation -- one argument
- flips every bit

```
NOT 11000101
------------
  00111010
```

# Bitwise Operations in Integers

Operations on logical TRUE or FALSE

- two states -- takes one bit to represent: TRUE=1, FALSE=0

| A | B | A NAND B |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| A | B | A XOR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

View *n*-bit number as a collection of *n* logical values

- operation applied to each bit independently

# Bitwise Operations in Integers

NAND

- Any logical operand can be created using NAND operands (we can construct a computer using NAND operands and memory, see NAND2TETRIS book and course)

```
  11000101
NAND 00001111
-------------
  11111010
```

XOR

- Useful for checking odd or even number of 1 bits

```
  11000101
XOR 00001111
-------------
  11001010
```

# Bitwise Operations in Integers

- `&` AND
  - Result is `1` if both operand bits are `1`
- `|` OR
  - Result is `1` if either operand bit is `1`
- `~` Complement
  - Each bit is reversed
- `^` Exclusive OR
  - Result is `1` if operand bits are different

Apply to all kinds of *integer* types:–
signed and unsigned
char, short, int, long, long long

# Bitwise & Regular Operations are Not Equivalent!

```
  00001111
& 11110000
----------
  00000000
```

**A zero value that evaluates to false**

```
   00001111
&& 11110000
-----------
```

**value which is not zero and evaluates to true**

# C Literatls

# Literals: a notation for representing a fixed value in source code

- 42 → int
- ‘c’ → char (in C the type is actually an int, sizeof(‘c’)==sizeof(int), in C++ it is a char)
- “hello” → c-string, array of chars that ends with a ‘\0’
- “hel””lo” → same as the c-string above
- 42.0 → double
- 1e+2 → double in scientific notation (1x10^+2==100)
- 1e-2 → double in scientific notation (1x10^-2==0.01)
- 42f → float

# Integer Literals Rules

- An integer literal can have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.
- Bases:
  - Empty prefix: decimal (10)
  - 0 prefix: octal (8)
  - 0x prefix: hexadecimal (16)

Examples:

- 42 → decimal int
- **0**42**u** → **octal** (34 in decimal) **unsigned** int
- **0x**42**L** → **hexadecimal** (66 in decimal) **long** int

# Why to use octal or hexa bases?

- compressed representations of binary

# Char Literals Rules

- sizeof(char)==1
- We can write **‘a’** and it will be represented using the binary representation of the decimal value of 97 (‘a’ is an int literal in C).
- We can also write it in octal (8) base with 3 digits:

**‘\141’** or in hexadecimal (16) base with 2 digits: **‘\x61’**

,these are called “bit patterns”


**Octal bit patterns MS bit should be <=3**

# Binary literals: C++14, gcc supports it in C

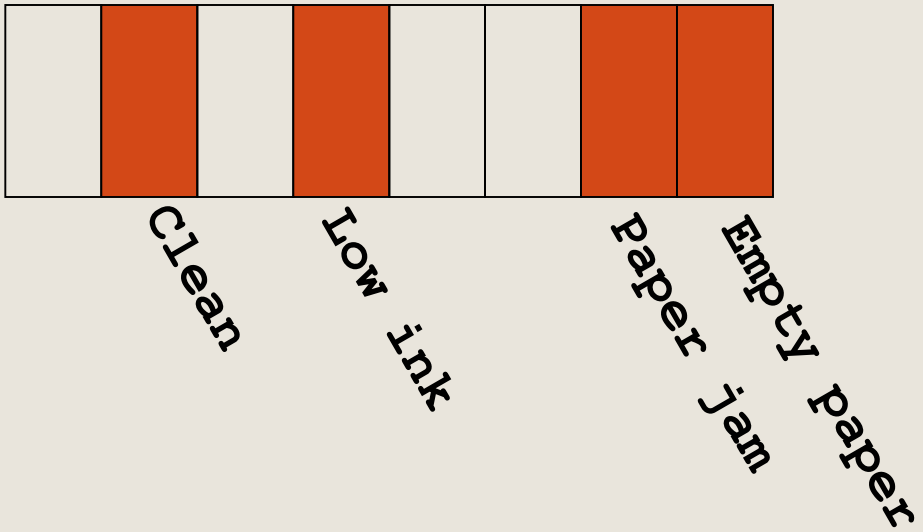
```
int val= 0b0101;

Or

int val= 0B0101;
```

# Working with Bits

# Two Approaches

- Use #define and a lot of bitwise operations
- Use bit fields

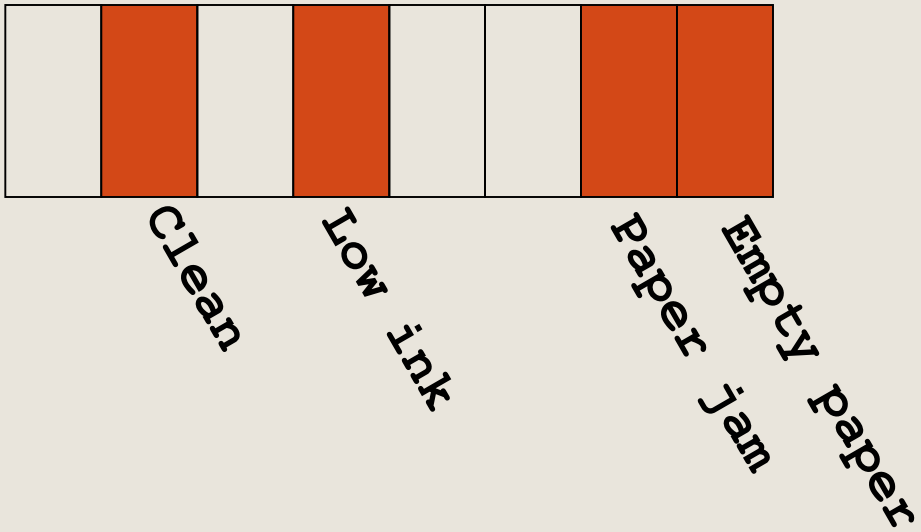
# Example – Printer Status Register




Definition of bit masks

```
#define EMPTY    1
#define JAM      2
#define LOW_INK  16
#define CLEAN    64
```

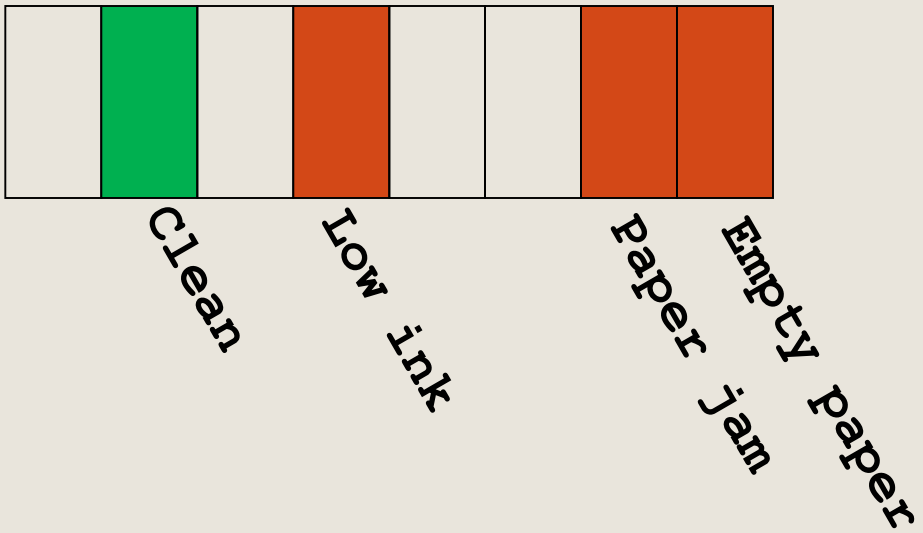
# Example – Printer Status Register




Definition of bit masks in octal

```
#define EMPTY    01
#define JAM      02
#define LOW_INK  020
#define CLEAN    0100
```

# Example – Printer Status Register

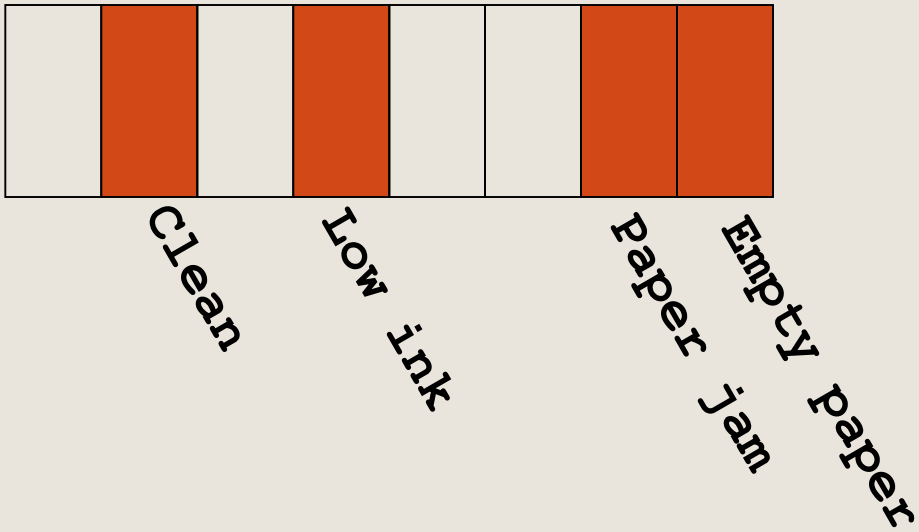



Definition of bit masks in octal

```
#define EMPTY    01
#define JAM      02
#define LOW_INK  020
#define CLEAN    0100
```

**First two triplets are 000 and only the first bit of the third bit is 001**

# Example – Printer Status Register




```
char status;
...
while (!(status & LOW_INK)) ...;

status|= CLEAN;    /* turns on CLEAN bit */
status&= ~JAM;     /* turns off JAM bit */
```

# #define and bitwise operations

Used very widely in *C*

- Including a *lot* of existing code

No checking

- You are on your own to be sure the right bits are set

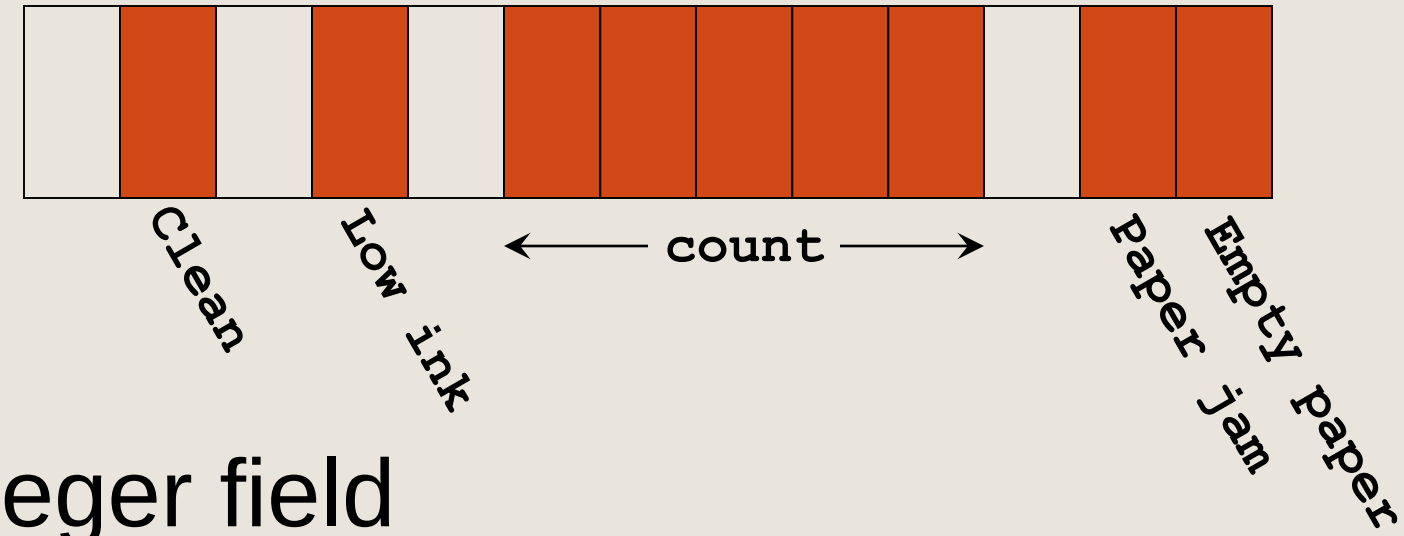
Machine dependent

- Need to know *bit order* in bytes, *byte order* in words

# Integer Fields Within a Register

- Need to **`AND`** and shift to extract
- Need to shift and **`OR`** to insert

# Example – Printer Status Register

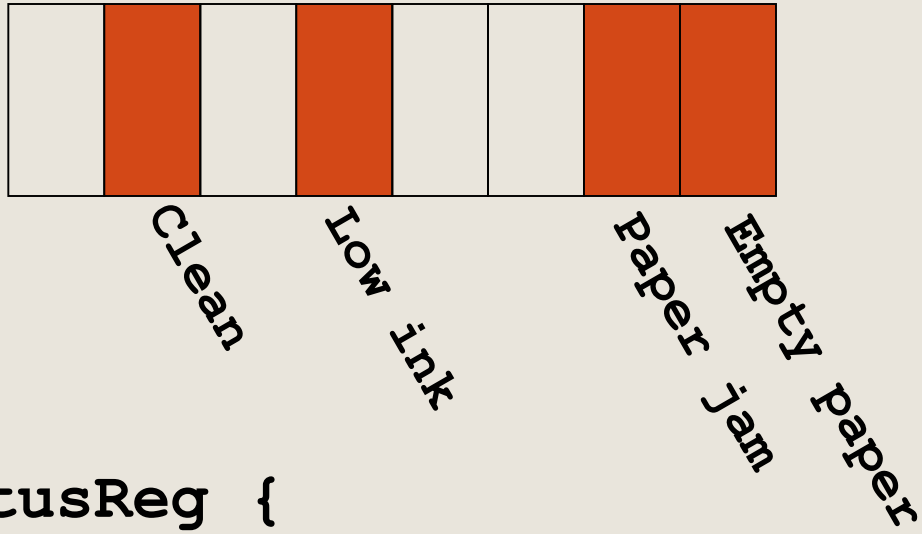



An integer field

```
#define COUNT (8|16|32|64|128)
// extract to c
unsigned int c = (status & COUNT) >> 3;
// insert v
status = (v << 3) | (status & ~COUNT);
```

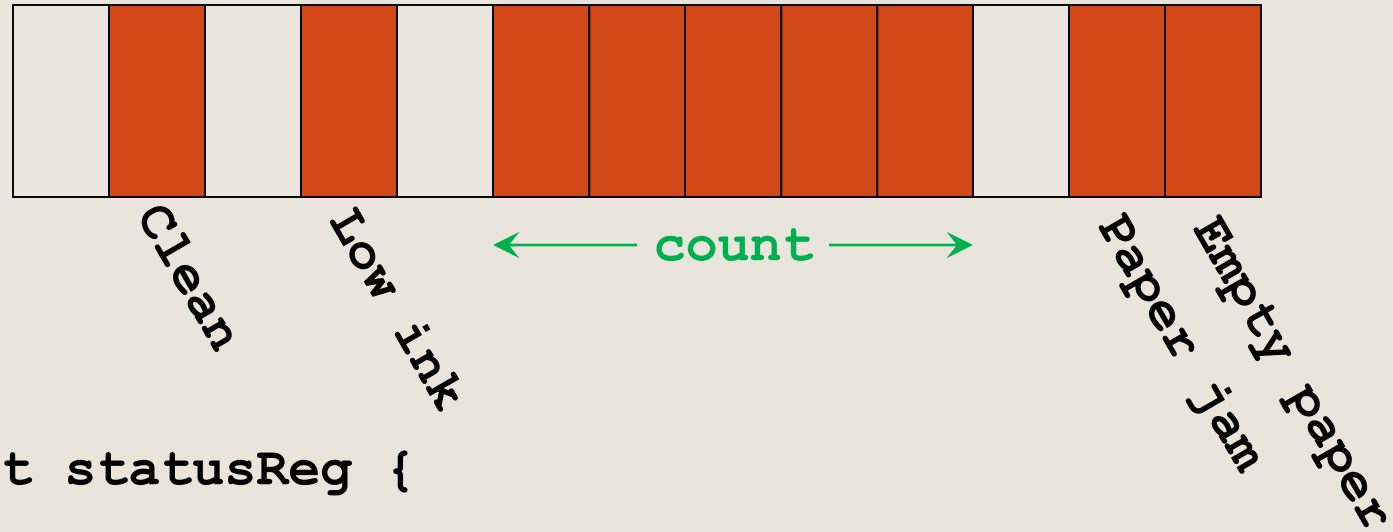
# Bit Fields

# Bit-Fields




```
struct statusReg {

  unsigned int emptyPaperTray :1;
  unsigned int paperJam       :1;
                              :2;
  unsigned int lowInk         :1;
                              :1;
  unsigned int needsCleaning  :1;
                              :1;

};
```

# Bit-Fields




```
struct statusReg {

  unsigned int emptyPaperTray :1;
  unsigned int paperJam       :1;
                              :1;
  unsigned int count          :5;
                              :1;
  unsigned int lowInk         :1;
                              :1;
  unsigned int needsCleaning  :1;
                              :1;
};
```

# Bit-Fields

```
struct statusReg s;

if (s.emptyPaperTray || s.paperJam) ...;
while(!s.lowInk) ...;

s.needsCleaning = 1;
s.paperJam = 0;

int c = s.count;
s.count = 2;
```

# Bit-Fields

Like a **`struct`**, except:

- Fields are bit-fields within a *word*
- Fields can be signed or unsigned
- Accessed like members of a **`struct`**
- Fields may be named or unnamed
- In some compilers (e.g., MSVS) fields can be any integer type, size is actually the number of bits, but type should be bigger than size

Machine-dependent:

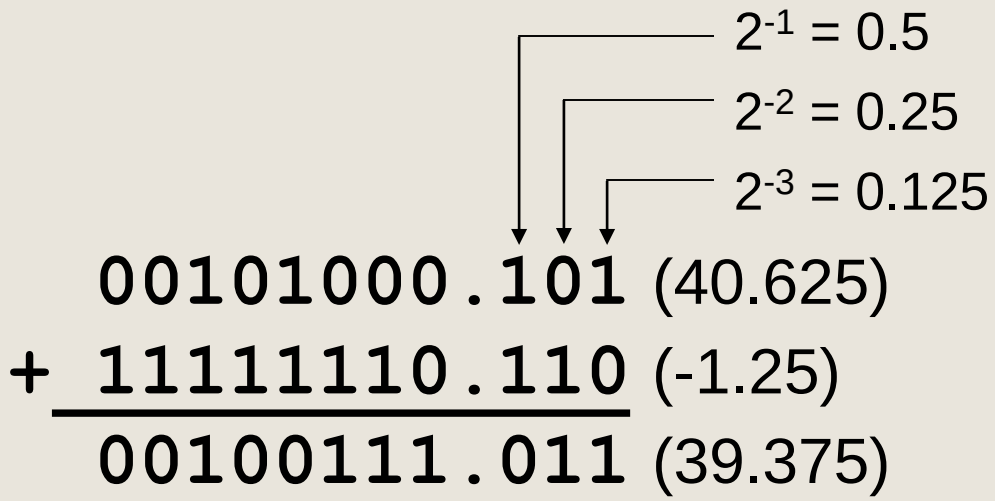
- Everything about the actual allocation details of bit fields within the class object. For example, order of bits in word
- Without the keyword signed or unsigned they might be signed or unsigned (unlike regular variables where the default is signed). Changed in c++-14

# Non-Integers

# Fractions: Fixed-Point

How can we represent fractions?

- Use a “binary point” to separate positive from negative powers of two -- just like “decimal point.”
- 2’s comp addition and subtraction still work.
  - if binary points are aligned




*No new operations -- same as integer arithmetic.*

# Floating-Point

Use equivalent of “scientific notation”: F x $2^E$

Need to represent F (*fraction*), E (*exponent*), and sign.

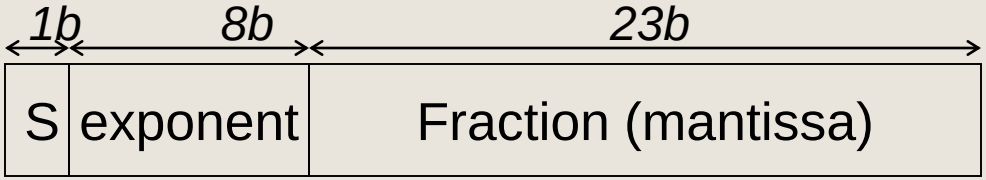
**Achieves greater range**

# Floating-Point

$$F \times 2^E$$

Need to represent F (*fraction*), E (*exponent*), and sign.

IEEE 754 Floating-Point Standard (32-bits):




$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127},\ 1 \le \text{exponent} \le 254$$

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126},\ \text{exponent} = 0$$

exponent = 255 used for special values:
If fraction is non-zero, NaN (not a number).
If fraction is zero and sign is 0, positive infinity.
If fraction is zero and sign is 1, negative infinity.

# Important Take Home Messages

If you need to know the representation – check, don't guess

Beware of overflows and rounding errors (numeric stability)

→ Beware of comparison with ==

We won't necessarily be able to represent all integers of a type A with a floating point number of type B even if A has less bits that B (we will be able up to 2^{mantissa bits+1}+1)

Remember the special values, and beware of unexpected behaviors! (e.g., NaN!=NaN,

IEEE 754 Floating-Point have signed zeros: -0 and +0)