University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Good Representations and Modular Arithmetic

What About Negative Numbers?

Last time, we developed

- the **N-bit unsigned representation**
- for integers in the range **$[0, 2^N - 1]$**

Now, let's think about negative numbers.

- How should we represent them?
- Can we use a minus sign?

$$-\mathbf{11000} = -24_{10}\ ?$$

There's no "–" in a bit!

One Option: The Signed-Magnitude Representation

But we can use another bit for a sign:

0 $\rightarrow$ **+**, and **1** $\rightarrow$ **–**

Doing so gives the
N-bit signed-magnitude representation:

sign	(N-1)-bit magnitude

This representation can represent numbers in the range **$[-2^{N-1} - 1, 2^{N-1} - 1]$**.

What Happened to the Last Bit Pattern?

Signed-magnitude was used in some early computers (such as the IBM 704 in 1954).

A question for you:

- The range represented is **$[-2^{N-1} - 1, 2^{N-1} - 1]$**.
- That gives **$2^N - 1$ different numbers**.
- **What's the last pattern being used to represent?**

Signed-Magnitude Has Two Patterns for Zero

There are two bit patterns for 0!

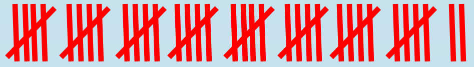
0	00000…00000	+0
1	00000…00000	-0

This aspect made some hardware more complex than is necessary.

Modern machines do not use signed-magnitude.

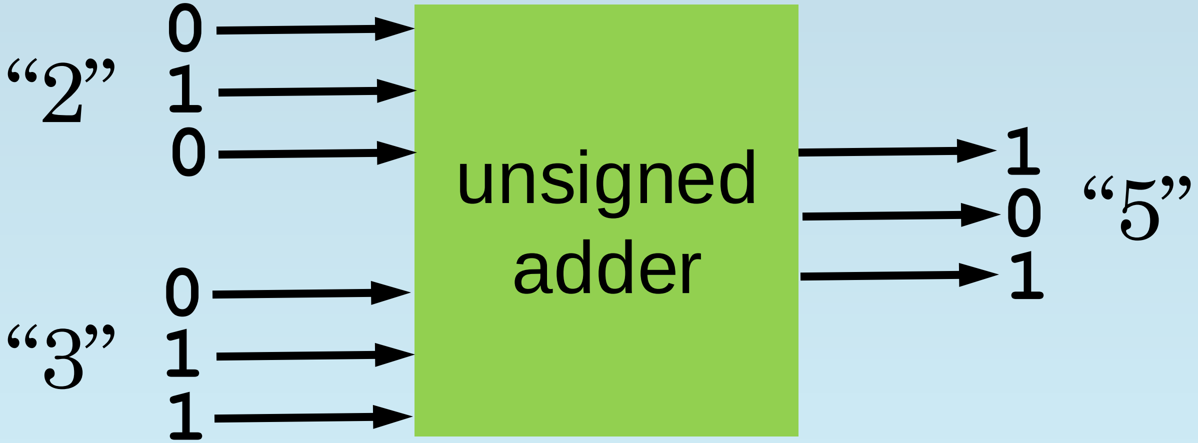
How Do We Choose Among Representations?

What makes a representation good?

- **efficient**: most bit patterns represent some item uniquely (so, not unary! )
- **easy/fast implementation of common operations**: such as arithmetic for numbers
- **shared implementation with other representations**: in this case, implementation is "free" in some sense

Representations Can be Chosen to Share Hardware

Imagine a device that performs addition on two bit patterns of an **unsigned** representation.



Can we use the same “adder” device for signed numbers? **Yes! If we choose the right representations.**

Add Unsigned Bit Patterns Using Base 2 Addition

Recall that the unsigned representation is drawn from base 2.

We use base 2 addition for unsigned patterns.

- Like base 10, we **add digit by digit**.
- Unlike base 10, the single-digit table of sums is quite small…
- What is `1 + 1 + 1`? `11`

A	B	Sum
`0`	`0`	`0`
`0`	`1`	`1`
`1`	`0`	`1`
`1`	`1`	`10`

Example: Addition of Unsigned Bit Patterns

Let's do an example with **5-bit unsigned**

```
   11
   01110  (14)
+  00100  (4)
---------
   10010  (18)
```

Good, we got the right answer!

Overflow Can Occur with Unsigned Addition

The unsigned representation is **fixed width**.

- If we start with **N** bits,
- we must end with **N** bits.

What is the condition under which the sum cannot be represented?

- The sum should have a 1 in the 2^N place.
- Only occurs when the most significant bits of the addends generate a carry.

We call this condition an **overflow**.

Example: Overflow of Unsigned Bit Patterns

Let's do an another example, again with **5-bit unsigned**

We have no space for that bit!

$$\begin{array}{rrl} & \cancel{1}11 & \\ & 01110 & (14) \\ + & 10101 & (21) \\ \hline & 00011 & (3) \end{array}$$

Oops! (The carry out indicates an overflow for unsigned addition.)

Unsigned Addition is Modular Arithmetic

Modular arithmetic is related to the idea of the "remainder" of a division.

Given integers **A**, **B**, and **M**,
- **A** and **B** are said to be **equal mod M** iff*
- **A = B + kM** for some integer **k**.

Note that **k** can be negative or zero, too.

We write: **(A = B) mod M**.

* "iff" means "if and only if," an implication in both directions, and is often used for mathematical definitions

Unsigned Addition is Always Correct Mod 2^N

Let **$SUM_N(A,B)$** be the number represented by the sum of two **N-bit unsigned** bit patterns.

If no overflow occurs (**$A + B < 2^N$**), we have **$SUM_N(A,B) = A + B$**.

For sums that produce an overflow, the bit pattern of the sum is missing the **2^N** bit, so **$SUM_N(A,B) = A + B - 2^N$**

In both cases,

$$(SUM_N(A,B) = A + B) \bmod 2^N.$$

Modular Arithmetic Key to Good Integer Representations

Modular arithmetic is the key.

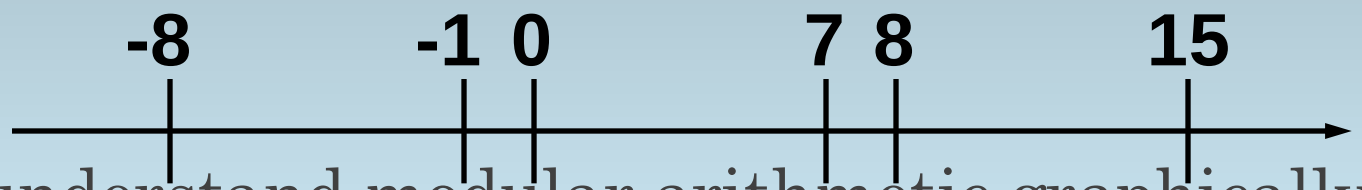
It allows us to define

- a **representation for signed integers**
- that **uses the same devices**
- as are **needed for unsigned arithmetic**.

The representation is called **2's complement**.

Details soon…

Modular Arithmetic on the Number Line



To understand modular arithmetic graphically, imagine breaking the number line into groups of **M** numbers, as shown above for **M=8**.

Two numbers are equal mod **M** if they occupy the same position in their respective groups.

For example, 0 is equal to an infinite number of other numbers (…, -24, -16, -8, 8, 16, 24, …).

We usually name sets of numbers that are equal mod **M** using the number in the range **[0, M-1]**.

University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Signed Integers and 2's Complement

Strategy: Use Common Hardware for Two Representations

Recall:

- addition of bit patterns in **N-bit unsigned** representations
- corresponds to arithmetic mod $\mathbf{2^N}$.

Using this arithmetic, we develop the **2's complement representation** for signed integers.

The same hardware can then perform arithmetic for both representations.

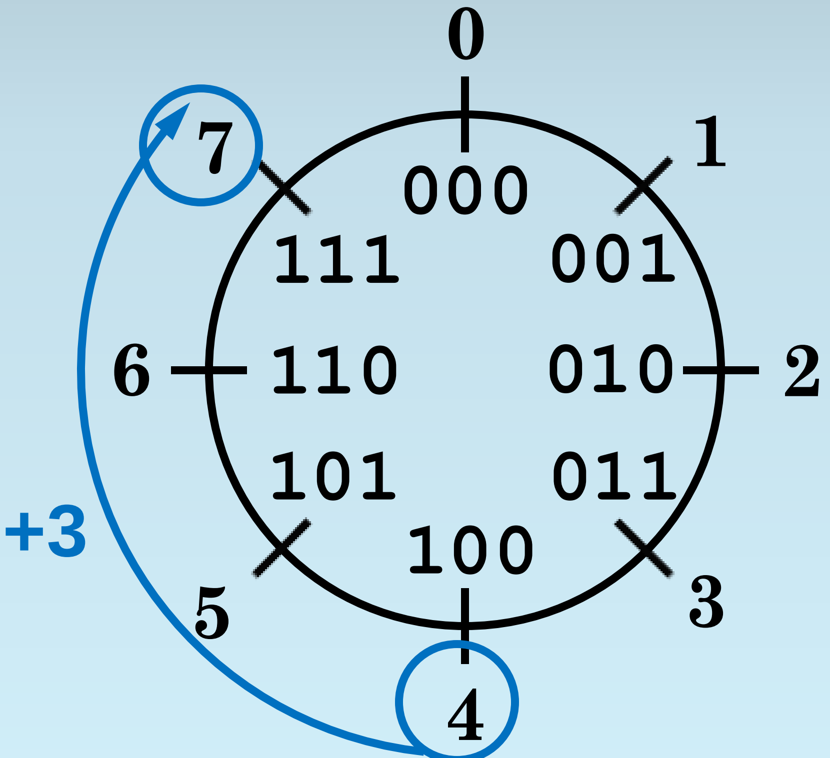
What about the name? Later.

Graphical Illustration of Modular Arithmetic

The circle illustrates **3-bit unsigned**.

Adding a number corresponds to counting clockwise.

The answer is always correct mod 8.



(For subtraction, count the other way.)

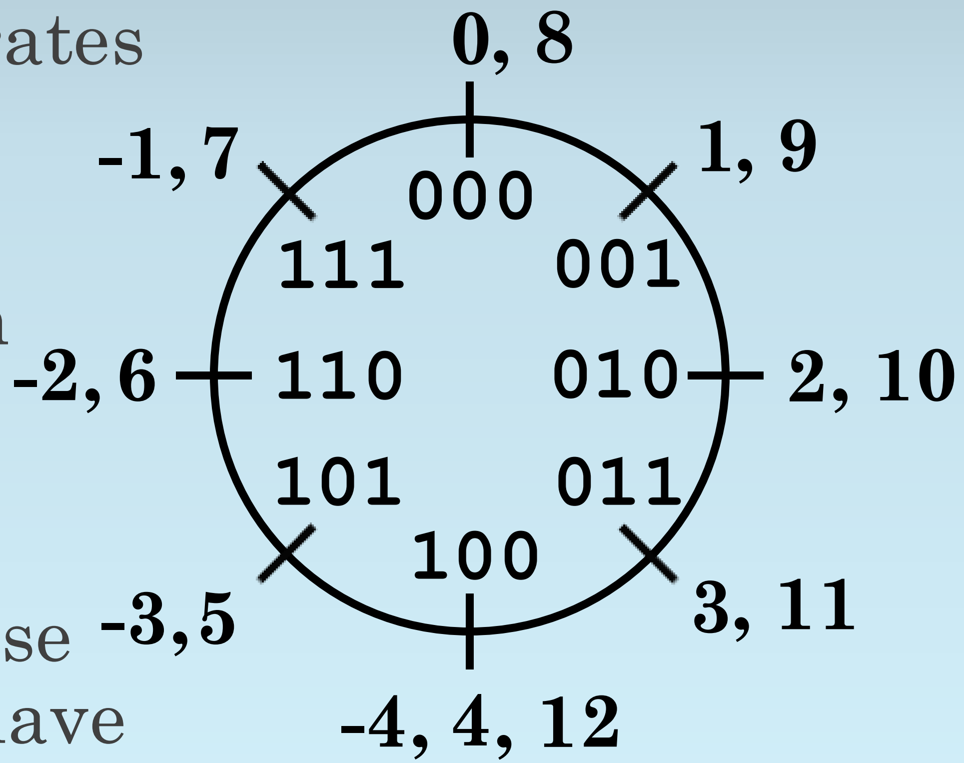
Representations Must be Unambiguous

The same circle illustrates **equality mod 8**.

For example, we can extend the numbers in a clockwise direction.

Or the other way.

Overflow occurs because a representation can have only **one value per bit pattern**.



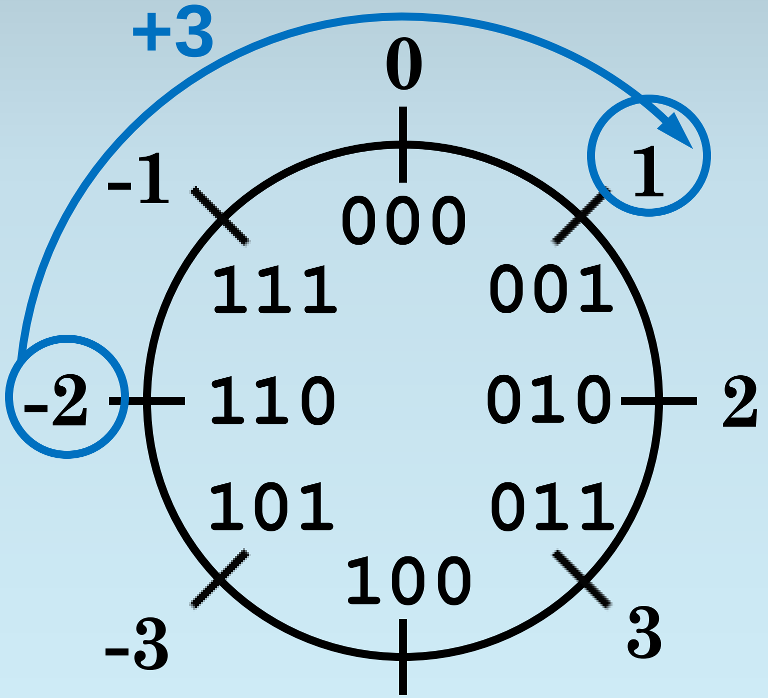
We Can Choose Any Meaning for a Bit Pattern

But what if we pick a different set of labels?

The arithmetic doesn't change.

Let's include both positive and negative numbers!

And try some addition.



That's One Way to Define 2's Complement

Draw a circle for **N** bits (**2^N** points).

Starting at 0 at the top.

Write unsigned bit patterns clockwise around the circle.

Starting again from 0,

- find bit patterns for negative numbers
- by moving counter-clockwise.

What about the name? Later.

2's Complement Can Also be Derived Algebraically

We can also define **N-bit 2's complement** algebraically.

An adder for **N-bit unsigned** gives

$$\mathbf{SUM_N(A,B) = A + B \bmod 2^N}$$

N-bit 2's complement includes positive numbers in the range $[1, 2^{N-1} - 1]$. These bit patterns all start with a "0" bit.

We need to find bit patterns for negative numbers.

Properties Needed for Negative Number Bit Patterns

For each number **K**, $0 < K < 2^{N-1}$,

- we want to find an **N**-bit pattern P_K, $0 \leq P_K < 2^N$,
- such that **for any integer M,**

$$(-K + M = P_K + M) \bmod 2^N$$

The bit pattern P_K then produces the same results as **-K** when used with unsigned arithmetic.

Also, P_K **must not be used by a number** ≥ 0.

Do Algebra to Define Negative Patterns

Starting with our property,

$$(-K + M = P_K + M) \bmod 2^N,$$

subtract M from both sides to obtain

$$(-K = P_K) \bmod 2^N.$$

Next, note that

$$(2^N = 0) \bmod 2^N.$$

Now add the last two equations to obtain

$$(2^N - K = P_K) \bmod 2^N.$$

Final Answer: -K is Represented by $2^N – K$

One easy solution to $\mathbf{(2^N – K = P_K) \bmod 2^N}$ is
$\mathbf{P_K = 2^N – K}$.

Since $\mathbf{0 < K < 2^{N-1}}$, this solution
gives $\mathbf{2^{N-1} < P_K < 2^N}$.

But these are all unused bit patterns—the
patterns starting with "1!"

So we're done:
-K is represented by the pattern $\mathbf{2^N – K}$.

What about the name? Are you really ready?

Negating Twice Gives an Identity Operation

Let's do a sanity check.

What is the bit pattern for **- (-K)**?

We know that **-K** is $\mathbf{2^N - K}$.

Substituting once, we obtain $\mathbf{-\ (2^N - K)}$.

Substituting again, we obtain $\mathbf{2^N - (2^N - K)}$.

But that's just **K**, as we expect.

What name? Oh, "2's complement?"

Is There an Easy Way to Find -K?

How do we calculate $\mathbf{2^N - K}$?

We can subtract (for example, with **N=5**)…

```
  100000  (2^N)
-  ?????  (K)
--------
```

But that seems painful.

Instead, notice that $\mathbf{2^N = (2^N - 1) + 1}$.

So we can calculate $\mathbf{(2^N - 1) - K + 1}$.

2's Complement is 1's Complement Plus One!

Again for **N=5**:

```
  11111   (2^N – 1)
– ?????   (K)
-------
(answer)
+     1
-------
```

The first step is trivial: replace 0 with 1, and 1 with 0. The result (**(2^N – 1) – K**) is called the **1's complement of K**.

Adding 1 more gives the **2's complement**.

Distinguish 2's Complement from Negation

Here or elsewhere, you will hear the phrase "take the 2's complement."

We will **try not to use "2's complement" in that way**.

Students get confused between the **2's complement representation** for signed integers and the operation of **negation** on a bit pattern for a number represented with 2's complement.

For clarity, we suggest that you do the same.

Example: Negating a Number in 2's Complement

Let's do an example of negation with **8-bit 2's complement**.

As you know, I like 42.

As you may remember, 42_{10} = **`00101010`**.

So what's -42?

First, complement the bits: **`11010101`**.

Then add 1: **`11010110`** = -42_{10}!

2's Complement Conversion Can Be Same as Unsigned

For **non-negative numbers** (bit patterns starting with 0),

conversion between decimal value and **2's complement** bit pattern

is **identical to conversion for the unsigned representation**.

Use Two Negations to Convert Negative Numbers

To convert decimal **D < 0** to **2's complement**,

- first **convert –D** (as **unsigned**),
- then **negate the resulting bit pattern**.

To convert a negative **2's complement** bit pattern (a bit pattern starting with 1) to decimal,

- first **negate the bit pattern**,
- then **convert to decimal D** (as unsigned).
- The **answer is -D**.

Alternate Method for Calculating 2's Complement Values

If we have a negative number **$-K$**, we can use the base 2 polynomial to calculate **$2^N - K$**:

$$2^N - K = a_{N-1}2^{N-1} + a_{N-2}2^{N-2} + \ldots + a_0 2^0$$

We know that **$a_{N-1} = 1$** for a negative number. Substituting and subtracting **2^N** gives:

$$-K = (2^{N-1} - 2^N) + a_{N-2}2^{N-2} + \ldots + a_0 2^0$$

$$-K = -a_{N-1}2^{N-1} + a_{N-2}2^{N-2} + \ldots + a_0 2^0$$

This polynomial also works when **$a_{N-1} = 0$**.

What about the Last Bit Pattern?

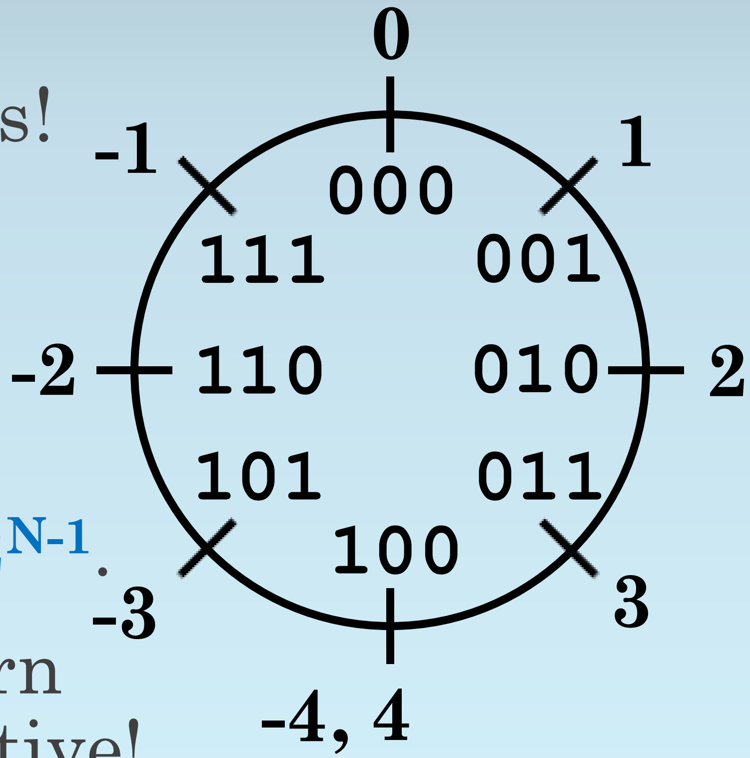
We didn't define a value for one of the bit patterns!

What should it be?

2^{N-1}? -2^{N-1}? Undefined?

In **2's complement**, the pattern always means -2^{N-1}.

Why? So that any pattern starting with a 1 is negative!



Extend Unsigned Bit Patterns by …

In some cases, we need

- to convert a bit pattern
- from a smaller representation (fewer bits)
- to a larger one (more bits)

How do we convert **N-bit unsigned** to **(N+k)-bit unsigned** (for **k** > 0)?

Hint: We already had to solve a similar problem when a number does not require **N** bits in base 2.

Add k more leading 0s (called **zero extension**).

What about 2's Complement?

How do we convert **N-bit 2's complement** to **(N+k)-bit 2's complement** (for **k** > 0)?

For **non-negative values**,

- **2's complement** is the same as **unsigned** (with an extra 0 for the sign)
- So **add k more leading 0s**.

What about negative values?

Extend 2's Complement Bit Patterns by …

In **5-bit 2's complement**,

-5_{10} has bit pattern 11011

-10_{10} has bit pattern 10110

(spaces added to help us humans)

And in **8-bit 2's complement**?

-5_{10} has bit pattern 111 11011

-10_{10} has bit pattern 111 10110

So how do we convert **N-bit 2's complement** to **(N+k)-bit 2's complement** (for **k** > 0)?

Add k copies of the sign bit
(called **sign extension**).