Binary Arithmetic, Bitwise Operations

CS 2130: Computer Systems and Organization 1
January 21, 2026

Announcements

- My Office Hours
  - Tuesdays 11:00am - 12:00pm
  - Wednesdays 2:00pm - 3:30pm
- TA Office Hours on website (location coming soon)
- Homework 1 available later this week

Multi-bit Values

- So far, only talking about 2 things
- Numbers, strings, objects, ...

Numbers

From our oldest cultures, how do we mark numbers?

- **unary** representation: make marks, one per ”thing”
  - Awkward for large numbers, ex: CS 2130?
  - Hard to tell how many marks there are
- Update: group them!
- Romans used new symbols: V, X, L, C, M

Numbers

From our oldest cultures, how do we mark numbers?

- Arabic numerals
  - Positional numbering system

Numbers

From our oldest cultures, how do we mark numbers?

- Arabic numerals
  - Positional numbering system
  - The 10 is significant:
    - 10 symbols, using 10 as base of exponent

Numbers

From our oldest cultures, how do we mark numbers?

- Arabic numerals
  - Positional numbering system
  - The 10 is significant:
    - 10 symbols, using 10 as base of exponent
  - The 10 is *arbitrary*
  - We can use other bases! π, 2130, 2, ...

Base-8 Example

Try to turn 134_8 into base-10:

Bases

We will discuss a few in this class

- Base-10 (decimal) - talking to humans
- Base-8 (octal) - shows up occasionally
- Base-2 (binary) - most important! (we've been discussing 2 things!)
- Base-16 (hexadecimal) - nice grouping of bits

Binary

2 digits: 0, 1

Try to turn 1100101_2 into base-10:

Binary

Any downsides to binary?

Turn 2130_{10} into base-2:
hint: find largest power of 2 and subtract

Long Numbers

How do we deal with numbers too long to read?

Long Numbers

How do we deal with numbers too long to read?

- Group them by 3 (right to left)

Long Numbers

How do we deal with numbers too long to read?

- Group them by 3 (right to left)
- In decimal, use commas: ,
- Numbers between commas: 000 - 999

Long Numbers

How do we deal with numbers too long to read?

- Group them by 3 (right to left)
- In decimal, use commas: ,
- Numbers between commas: 000 - 999
- Effectively base-1000

Long Numbers in Binary - Readability

- Typical to group by 3 or 4 bits
- No need for commas *Why?*

100001010010

Long Numbers in Binary - Readability

- Typical to group by 3 or 4 bits
- No need for commas *Why?*
- We can use a separate symbol per group
- How many do we need for groups of 3?

100001010010

Long Numbers in Binary - Readability

- Typical to group by 3 or 4 bits
- No need for commas *Why?*
- We can use a separate symbol per group
- How many do we need for groups of 3?
- Turn each group into decimal representation

100001010010

Long Numbers in Binary - Readability

- Typical to group by 3 or 4 bits
- No need for commas *Why?*
- We can use a separate symbol per group
- How many do we need for groups of 3?
- Turn each group into decimal representation
- Converts binary to **octal**

100001010010

Long Numbers in Binary - Readability

- Groups of 4 more common
- How many symbols do we need for groups of 4?

100001010010

Long Numbers in Binary - Readability

- Groups of 4 more common
- How many symbols do we need for groups of 4?
- Converts binary to **hexadecimal**
- Base-16 is very common in computing

100001010010

Hexadecimal

Need more than 10 digits. What next?

1110

Hexadecimal Exercise

Consider the following hexadecimal number:

852dab1e

Is it even or odd?

Finally, Numbers!

Storing Integers

- Use binary representation of decimal numbers
- Usually have a limited number of bits (ex: 32, 64)
  - Depending on language
  - Depending on hardware

Using Different Bases in Code

	Old Languages	New Languages
binary		
octal		
decimal		
hexadecimal		

Finally, Numbers!

Storing Integers

- Use binary representation of decimal numbers
- Usually have a limited number of bits (ex: 32, 64)
  - Depending on language
  - Depending on hardware
- Is there something missing?

Negative Integers

Representing negative integers

Negative Integers

Representing negative integers

- Can we use the minus sign?

Negative Integers

Representing negative integers

- Can we use the minus sign?
- In binary we only have 2 symbols, must do something else!
- Almost all hardware uses the following observation:

Representing Negative Integers

Computers store numbers in fixed number of wires

- Ex: consider 4-digit decimal numbers

Representing Negative Integers

Computers store numbers in fixed number of wires

- Ex: consider 4-digit decimal numbers
- Throw away the last borrow:
  - 0000 - 0001 = 9999 == -1
  - 9999 - 0001 = 9998 == -2
  - Normal subtraction/addition still works
  - Ex: -2 + 3

Representing Negative Integers

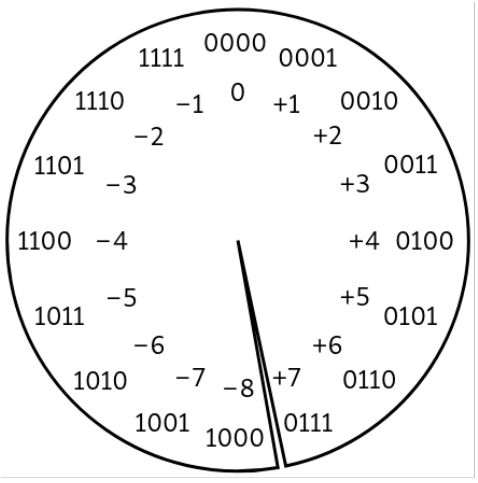
Computers store numbers in fixed number of wires

- Ex: consider 4-digit decimal numbers
- Throw away the last borrow:
  - 0000 - 0001 = 9999 == -1
  - 9999 - 0001 = 9998 == -2
  - Normal subtraction/addition still works
  - Ex: -2 + 3
- This works the same in binary

Two's Complement

This scheme is called **Two's Complement**

- More generically, a *signed* integer
- There is a break as far away from 0 as possible
- First bit acts vaguely like a minus sign
- Works as long as we do not pass number too large to represent



Two’s Complement

Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

11010011

What is its value in decimal?

Values of Two's Complement Numbers

Consider the following 8-bit binary number in Two's Complement:

11010011

What is its value in decimal?

1. Flip all bits
2. Add 1

Addition

```
  01001010
+ 01111100
----------
```

Subtraction

```
  01001010
- 01111100
----------
```

Operations

So far, we have discussed:

- Addition: $x + y$
  - Can get multiplication
- Subtraction: $x - y$
  - Can get division, but more difficult
- Unary minus (negative): $-x$
  - Flip the bits and add 1

Operations (on Integers)

Bit vector: fixed-length sequence of bits (ex: bits in an integer)

- Manipulated by bitwise operations

Bitwise operations: operate over the bits in a bit vector

- Bitwise not: `~x` - flips all bits (unary)
- Bitwise and: `x & y` - set bit to 1 if x, y have 1 in same bit
- Bitwise or: `x | y` - set bit to 1 if either x or y have 1
- Bitwise xor: `x ^ y` - set bit to 1 if x, y bit differs

Example: Bitwise AND

```
  11001010
& 01111100
----------
```

Example: Bitwise OR

```
  11001010
| 01111100
----------
```

Example: Bitwise XOR

```
  11001010
^ 01111100
----------
```

Your Turn!

What is: `0x1a ^ 0x72`

Operations (on Integers)

- Logical not: $!x$
  - $!0 = 1$ and $!x = 0, \forall x \neq 0$
  - Useful in C, no booleans
  - Some languages name this one differently

Operations (on Integers)

- Left shift: $x << y$ - move bits to the left
  - Effectively multiply by powers of 2
- Right shift: $x >> y$ - move bits to the right
  - Effectively divide by powers of 2
  - Signed (extend sign bit) vs unsigned (extend 0)