

Lecture 05

Data Operations, Fractions in Binary, and Binary Operations

Comp_Sci 213

Introduction to Computer Systems

Monday, Oct. 7, 2026

Kate Smith – Fall 2026

Slides adapted from:

Ghena, St-Amour, Hardavellas, Bustamante (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Encoding strings (The C way)

```
char S[6] = "18243";
```

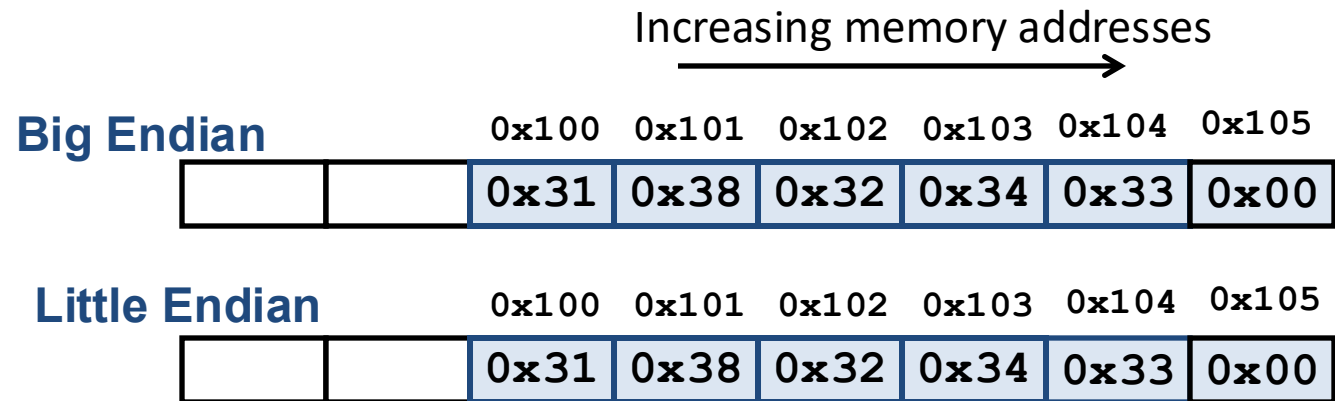
Dec	Hx	Oct	Html	Chr
48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9

Let's represent a number as a string! This is represented by array of characters

- Each character encoded in ASCII format
- NULL character (code 0) to mark the end
- The string has five characters, and the sixth byte holds the terminating NULL character ('\0', value 0, 0x00)
 - If `char S[] = "18243";` with no size, the compiler makes the array 6 bytes and includes the null automatically

Compatibility

1. Byte ordering not an issue (data all single-byte!) since endianness is for multi-byte values. Array elements always go in increasing address order
2. ASCII text files generally platform independent, except for different conventions of line termination character(s)!



Encoding strings, continued

- What happens if we store as an int rather than a character string?
 - $18243 = 0x4743$
- Endianness would be apparent!
 - Little endian: 43 47
 - Big endian: 47 43
- Integer encoding depends on byte order, but text form doesn't!

Topics

- Integer Operations
- Fractional binary numbers
- Intro Binary Operations – Boolean Algebra

Topics

- **Integer Operations**
- Fractional binary numbers
- Intro Binary Operations – Boolean Algebra

C versus the hardware

- Operations you can perform on types have edge conditions
 - Usually going above or below the bounds for the given size/encoding
- If we say what happens in that scenario, it'll be what “the hardware” (i.e., a computer) does
 - In today's examples, pretty much every computer does the same thing
- That is not the same as what C does
 - Unclear choices are left as **undefined behavior**
 - **Undefined Behavior:** C standard places no requirement on what happens if a program does that thing
 - The compiler can make any choice it wants
 - Program might crash, produce garbage, appear to work, behave differently based on optimization level, etc.

Unsigned Addition

- Like grade-school addition, but in base 2, and ignores final carry
 - If you want, can do addition in base 10 and convert to base 2. Same result! But here we're going to understand what the hardware is doing.
- **Example: Adding two 4-bit numbers**

$$\begin{array}{r} \overset{1}{0}\overset{1}{1}\overset{1}{0}1 \\ + \quad 0011 \\ \hline 1000 \end{array}$$

- $5_{10} + 3_{10} = 8_{10}$ ✓

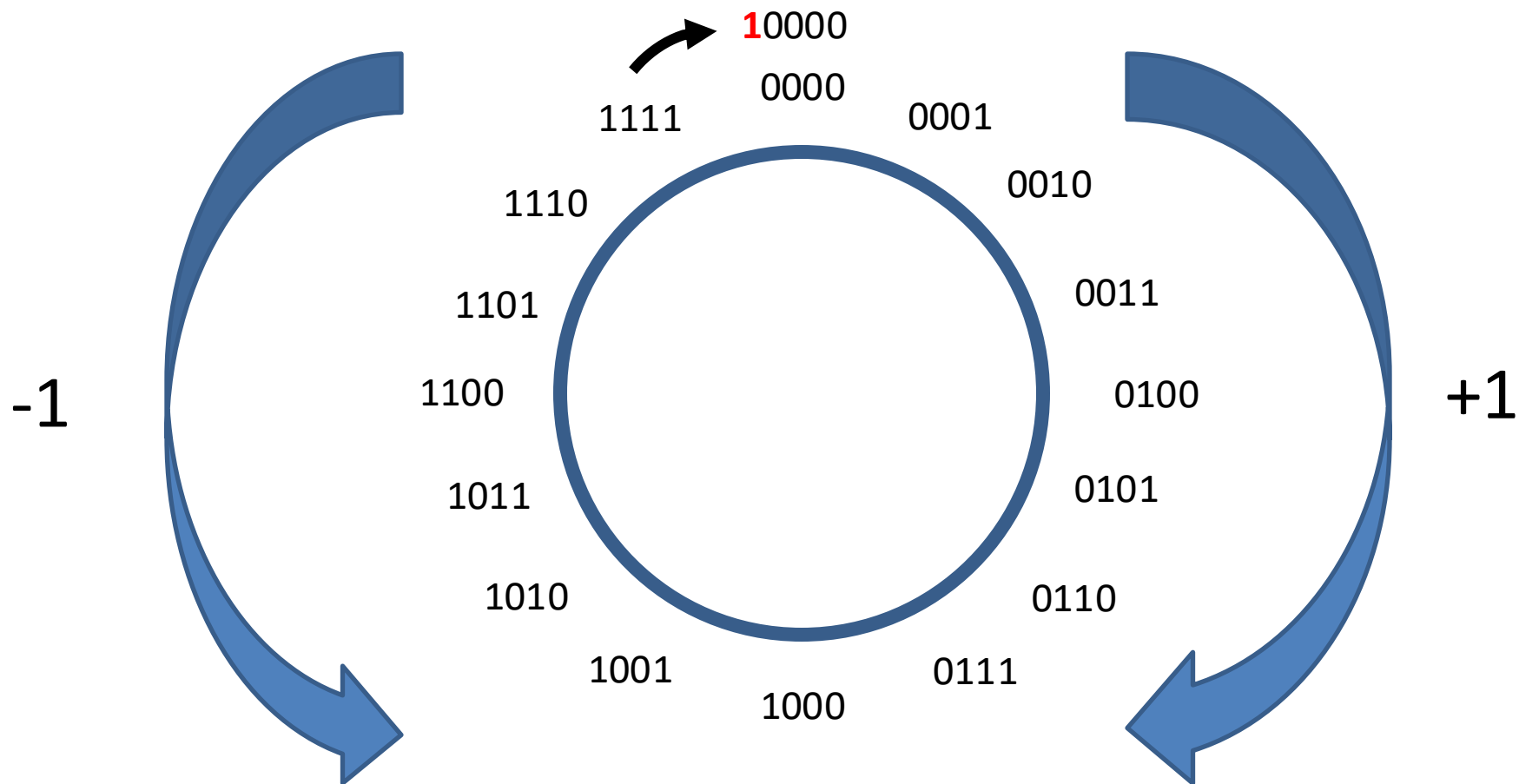
Unsigned Addition and Overflow

- What happens if the numbers get too big?
- **Example: Adding two 4-bit numbers**

$$\begin{array}{r} 1\ 1\ 1\ 1 \\ 1\ 1\ 0\ 1 \\ +\ 0\ 0\ 1\ 1 \\ \hline 1\ 0\ 0\ 0\ 0 \end{array}$$

- $13_{10} + 3_{10} = 16_{10}$
 - Too large for 4 bits!
 - Truncate most-significant bits that do not fit
 - Gives us modular (= modulo) behavior: $16 \text{ modulo } 2^4 = 0$
 - Final result is the 4 least significant bits (all we can fit): so 0_{10}

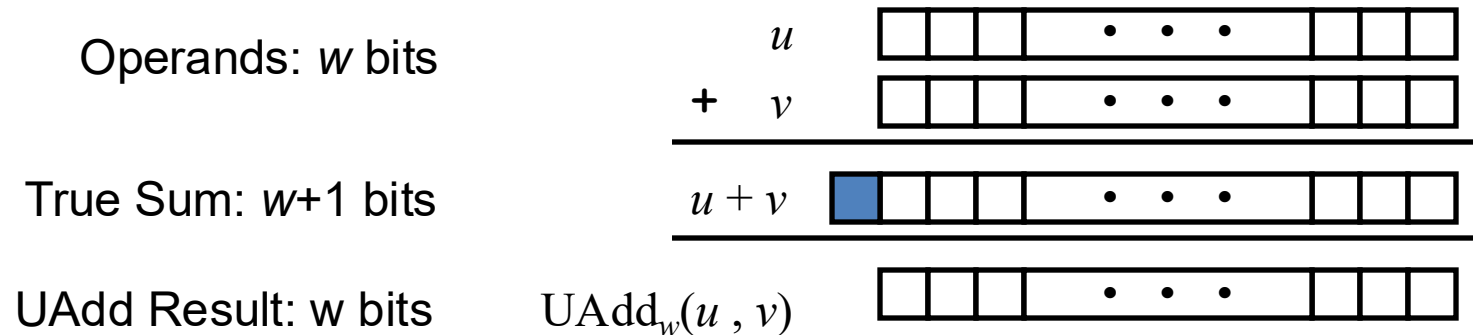
Modulo behavior in binary numbers



****Note:**
 $N \bmod r$ ($N \% r$) will be
some value $[0, r-1]$

Unsigned addition is modular

- Implements modular arithmetic
 - $\text{UAdd}_w(u, v) = (u + v) \bmod 2^w$
- Need to drop carry-out bit, otherwise results will keep getting bigger
 - Example in base 10: $80_{10} + 40_{10} = 120_{10}$ (2-digit inputs become a 3-digit output!)



- Warning: C does not tell you that the result was truncated!
 - **Unsigned** addition in C silently truncates most-significant bits beyond the limit

Signed (2's Complement) Addition

- Works exactly the same as unsigned addition!
 - Just add the numbers in binary, and the result will work out
- Signed and unsigned sum have the exact same bit-level representation
 - Computers use the same machine instruction and the same hardware!
 - That's a big reason 2's complement is so nice! Shares operations with unsigned

Signed addition example

- Same addition method as unsigned
- **Example: Adding two 4-bit signed numbers**

$$\begin{array}{rcl} & \overset{1}{1} & \overset{1}{1} \\ & 10\bar{1}1 & (-5) \\ + & 0011 & (+3) \\ \hline & 1110 & (-2) \end{array}$$

○ $-5_{10} + 3_{10} = -2_{10} \checkmark$

Combining negative and positive numbers

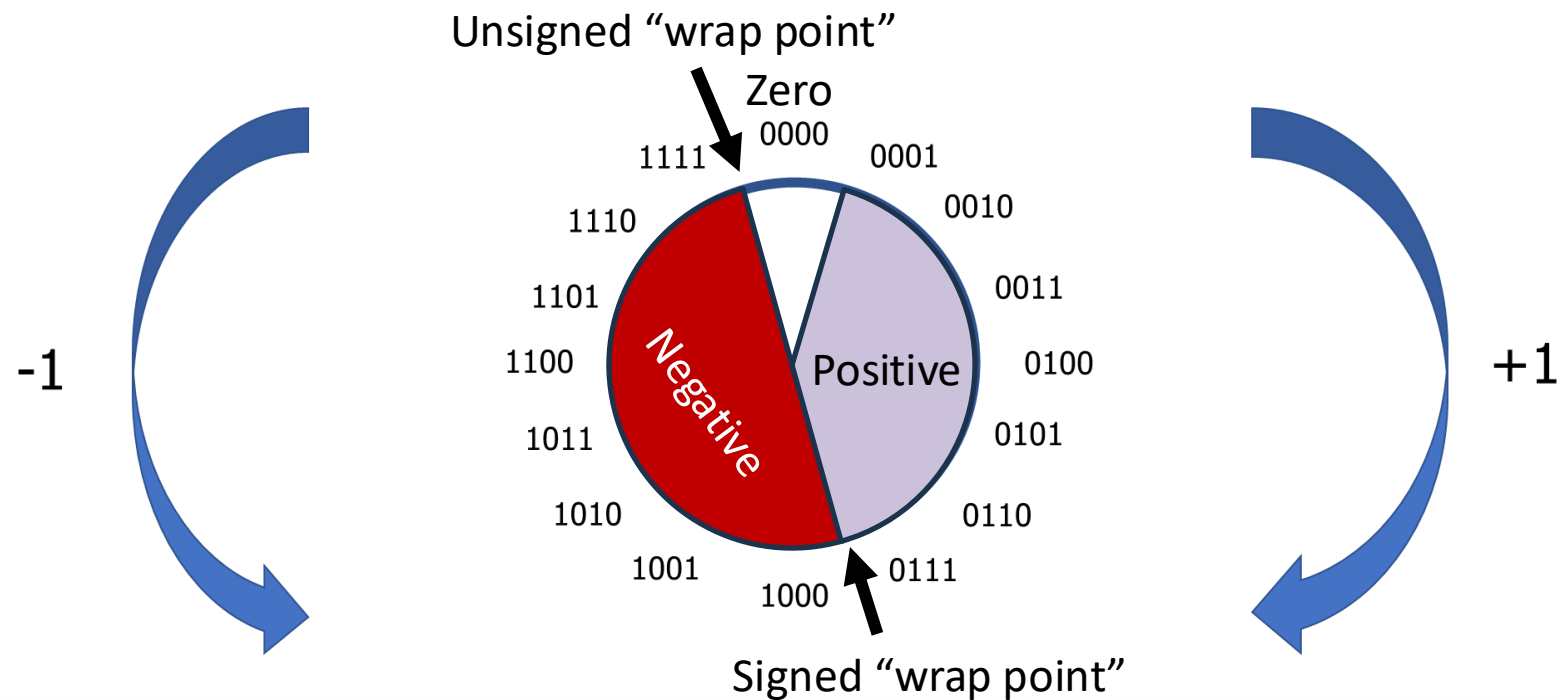
- Truncation makes signed addition work!
- **Example: Adding two 4-bit signed numbers**

$$\begin{array}{r} \overset{1}{1} \overset{1}{1} \overset{1}{1} \overset{1}{1} \\ 1101 \quad (-3) \\ + 0011 \quad (+3) \\ \hline 10000 \end{array}$$

- $-3_{10} + 3_{10} = 0_{10}$
 - Too large for 4 bits! Drop the carry bit
 - Result is what we expect as long as we truncate

Overflow of signed values

- Addition can lead to **overflow**
 - Unsigned: Result wraps around from big to small or vice versa
 - Signed: Both operands are positive and result is negative
 - Signed: Both operands are negative and result is positive



Signed addition and overflow

- Overflow can still happen in signed addition though
- **Example: Adding two 4-bit signed numbers**

$$\begin{array}{r} \overset{1}{0}\overset{1}{1}\overset{1}{0}1 \\ + \quad 0011 \\ \hline 1000 \end{array}$$

****For a fixed width w :**

- *Smallest value, most negative is 1 followed by all 0s bit pattern ($100\dots0 = -2^{w-1}$)*
- *Largest value, most positive is 0 followed by all 1s bit pattern ($01\dots1 = 2^{w-1} - 1$)*

- $5_{10} + 3_{10} = -8_{10}$ (+8 is too big to fit in 4 bits using a signed representation)
- Remember, this was also unsigned $5_{10} + 3_{10} = 8_{10}$

Signed addition and negative overflow

- Overflow also happens in the negative direction
- **Example: Adding two 4-bit signed numbers**

$$\begin{array}{r} \overset{1}{1} \quad \overset{1}{1} \overset{1}{1} \\ 1011 \\ + 1011 \\ \hline 10110 \end{array}$$

****For a fixed width w:**

- *Smallest value, most negative is 1 followed by all 0s bit pattern ($100\dots0 = -2^{w-1}$)*
- *Largest value, most positive is 0 followed by all 1s bit pattern ($01\dots1 = 2^{w-1} - 1$)*

- $-5_{10} + -5_{10} = +6_{10}$ (-10 was too small to fit in 4 bits using a signed representation)

Overflow: hardware vs C standard

- Hardware implementations for unsigned and signed addition are the same
 - Both implement truncation of overflowing bits, leads to modular arithmetic
- Unsigned overflow in C is defined as modular arithmetic
 - Safe to use. Big number wraps around to a small number
- Signed overflow in C is **undefined behavior**
 - Compiler *probably* does modular result
 - But there are no promises about this, and it can make *assumptions*
 - So don't rely on it
 - Generally: overflow is bad and is to be avoided!

Negating a number

- In C:
 - $x = -y;$
- Operation
 - Determine the negative, signed version of the number (two's complement)
 - Hardware method: flip bits and add one
- Complement operator (~)
 - This is a bit-level operator
 - Flips all bits: zeros become a one and ones become a zero
 - $\sim 0b1011 \rightarrow 0b0100$

Negating via Complement & Increment

- Claim: The following is true for 2's complement

- $\sim x + 1 == -x$

$$\begin{array}{r}
 \mathbf{x} \quad \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \\
 + \quad \sim \mathbf{x} \quad \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \\
 \hline
 -1 \quad \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1} \boxed{1}
 \end{array}$$

- Complement

- Observation: $\sim x + x == 1111...11_2 == -1$

- Increment

- $\sim x + 1 == \sim x + \mathbf{x} - \mathbf{x} + 1 == \cancel{-1} - \mathbf{x} + \mathbf{1} == -x$

- Example, 4 bits: $6_{10} = 0110_2$

- Complement: $1001_2 \rightarrow \text{Increment} = 1010_2 = -8 + 2 = -6_{10}$

****Note: $x - x = 0$**

Subtraction in two's complement

- Subtraction becomes addition of the negative number
 - $5 - 3 = 5 + -3 = 2$
- Both unsigned and signed subtraction
 - Convert subtrahend to its two's complement negative form
 - i.e., negate it
 - Then do addition
 - Treat result as original encoding

$$\begin{array}{r} \overset{1}{0} \overset{1}{1} \overset{1}{0} 1 \quad (+5) \\ + \quad 1 1 0 1 \quad (-3) \\ \hline \textcolor{red}{1} 0 0 1 0 \end{array}$$

C rules vs hardware rules

- Exact same overflow rules apply
- Unsigned subtraction overflow can wrap below zero to make a large number
 - Modular arithmetic
- Signed subtraction overflow is **UNDEFINED BEHAVIOR**
 - And therefore should not be trusted

Practice

- Adding two 8-bit binary numbers:
 - Also determine the decimal version of the result
 - Omitting the encoding is intentional

$$\begin{array}{r} 00010101 \\ + 10110001 \\ \hline \end{array}$$

Practice

- Adding two 8-bit binary numbers:
 - Also determine the decimal version of the result
 - Omitting the encoding is intentional

	Unsigned encoding		Signed encoding
$\begin{array}{r} \overset{1}{0} \overset{1}{0} \overset{1}{0} 1 0 1 \\ + \underline{1 0 1 1 0 0 0 1} \\ 1 1 0 0 0 1 1 0 \end{array}$	$\begin{array}{l} 16+4+1 = 21 \\ 128+32+16+1 = 177 \\ 128+64+4+2 = 198 \end{array}$	OR	$\begin{array}{l} 16+4+1 = 21 \\ -128+32+16+1 = -79 \\ -128+64+4+2 = -58 \end{array}$

Practice

- Adding two 8-bit binary numbers:
 - Also determine the decimal version of the result
 - Omitting the encoding is intentional

	Unsigned encoding		Signed encoding
$\begin{array}{r} \overset{1}{0}\overset{1}{0}\overset{1}{0}\overset{1}{0}\overset{1}{0} \\ 00010101 \\ + \underline{10110001} \\ \hline 11000110 \end{array}$	$\begin{array}{l} 16+4+1 = 21 \\ 128+32+16+1 = 177 \\ 128+64+4+2 = 198 \end{array}$	OR	$\begin{array}{l} 16+4+1 = 21 \\ -128+32+16+1 = -79 \\ -128+64+4+2 = -58 \end{array}$

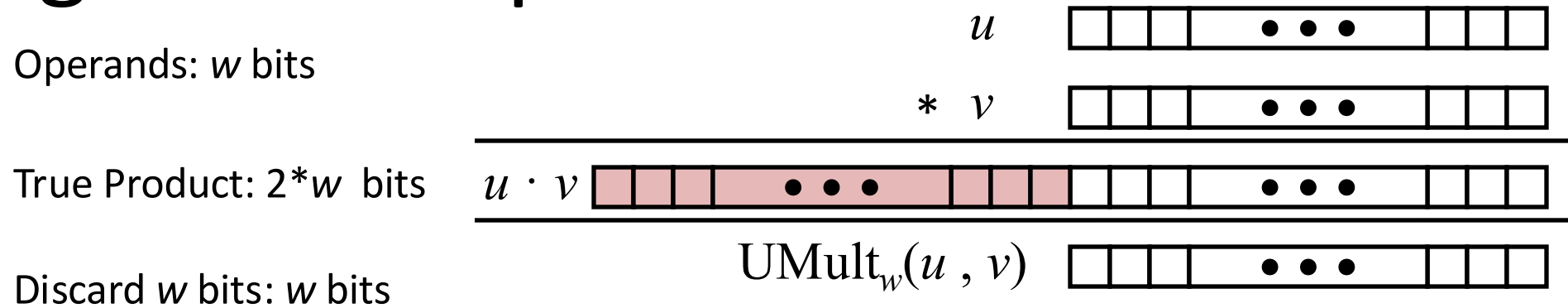
What about unsigned subtraction 21-79?

That would treat the result as unsigned, with the value 198. Modular arithmetic in action!!

Multiplication

- Goal: Compute the Product of two w -bit numbers x, y
 - Either signed or unsigned
- But, exact results can be bigger than w bits
 - Double the size ($2w$), in fact!
 - Example in base 10: $50_{10} * 20_{10} = 1000_{10}$
 - (2-digit inputs become a 4-digit output!)
- As with addition, result is truncated to fit in w bits
 - Because computers are finite, results can't grow indefinitely

Unsigned Multiplication



- **Standard Multiplication Function**
 - Equivalent to grade-school multiplication
 - But ignores most significant w bits of the result
 - As a person, we can do base 10 multiplication, convert to base 2, then truncate
- Implements modular arithmetic like addition does

$$\text{UMult}_w(u, v) = (u \cdot v) \bmod 2^w$$

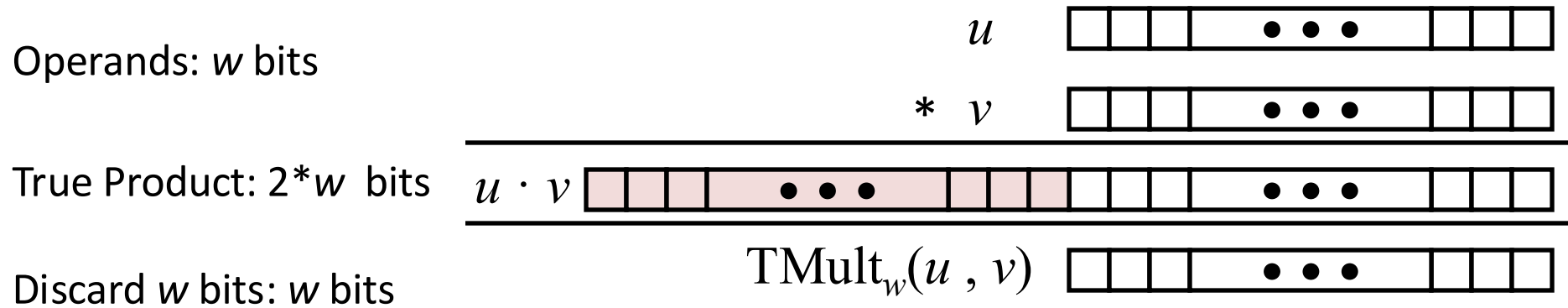
Unsigned multiplication

- Example: Multiplying two 4-bit numbers

$$\begin{array}{r} 0010 \\ \times 0101 \\ \hline 0010 \\ 00000 \\ 001000 \\ + 0000000 \\ \hline \cancel{000}1010 \end{array}$$

$$2_{10} * 5_{10} = 10_{10} \checkmark$$

Signed (2's Complement) Multiplication



- **Standard Multiplication Function** (same as unsigned)
 - Ignores most significant w bits
 - Lower bits still give the correct result
 - So we can use same machine instruction for both!
 - Again, that's one reason why 2's complement is so nice
- **In C, signed overflow is undefined**
 - ...but probably you'll see the two's complement behavior

Signed multiplication

- Example: Multiplying two's complement 5-bit numbers

$$\begin{array}{r} 11110 \quad -2 \\ \times 00011 \quad 3 \\ \hline 11110 \\ + 111100 \\ \hline \textcolor{red}{1}011010 \end{array}$$

What are these two 5-bit numbers?

What is the result of this addition?

$$-2_{10} * 3_{10} = -6_{10} \checkmark$$

What about divide?

- Annoying operation, not going to discuss in this class
 - Similar to long division process
 - Tedious and complicated to get right
- There are some computers that don't have hardware support for division at all!!
- Important thing to remember is that integers don't have fractional parts
 - In C: $1/2 == 0$
 - We'll need a different encoding for fractional numbers: floating point

Topics

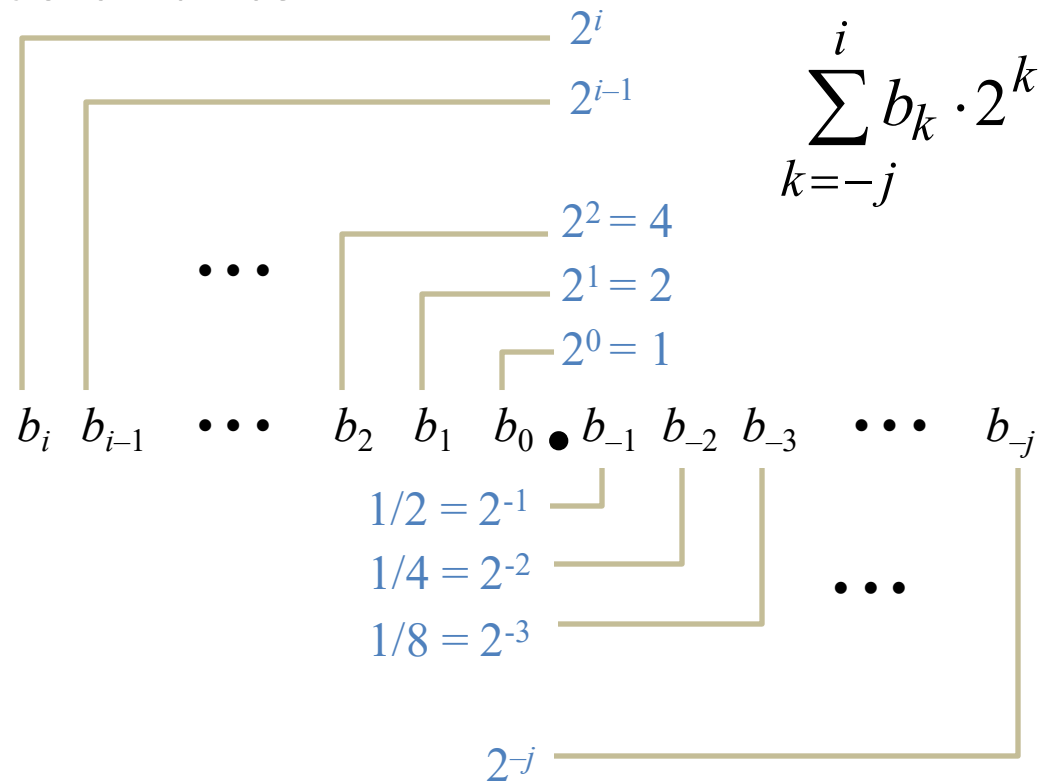
- Integer Operations
- **Fractional binary numbers**
- Intro Binary Operations – Boolean Algebra

Floating point numbers

- In decimal:
 - 123450_{10}
 - 123.450_{10}
 - 1.23450_{10}
- We can use this same system in binary as well:
 - 1010110_2 (86_{10})
 - 1010.110_2 ($10.75_{10} = \frac{86}{2^3}$)
 - 1.010110_2 ($1.34375_{10} = \frac{86}{2^6}$)

Fractional Binary Numbers

- Representation
 - Bits to right of “binary point” represent fractional powers of 2
 - Represents rational number:



Example binary conversion

1010.110

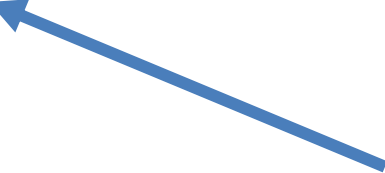
Before the binary point:

$$1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 1*2^3 + 1*2^1 = 8+2 = 10$$

After the binary point:

$$1*2^{-1} + 1*2^{-2} + 0*2^{-3} = 1*2^{-1} + 1*2^{-2} = \frac{1}{2} + \frac{1}{4} = \frac{3}{4} = 0.75$$

Fractional Binary Number Examples

- $5 + 3/4$ = 0b101.11


Note:
This the number 3!
- $2 + 7/8$ = 0b10.111


This is the number 7!
- $63/64$ = 0b0.111111


This is the number 63!

Scientific Notation Binary

- Scientific notation works in binary just like in decimal
 - Decimal: $134 = 1.34 * 10^2$
 - Binary: $101 = 1.01 * 2^2$
- Positive and negative exponents change the direction the point moves
 - Positive powers get bigger: $1.00101 * 2^4 \rightarrow 10010.1$
 - Negative powers get smaller: $1.00101 * 2^{-3} \rightarrow 0.00100101$

Binary point is part of the solution, but not an entire encoding

- Some problems remain:
 1. Computers are finite, but real numbers are not
 - Need to choose how many bits to use
 - Many decimal numbers would take infinite binary bits to represent perfectly
 - $3.14_{10} = 11.0010001111010111_2$ (we could keep going)
 2. We also need to represent where the “binary point” is located
 - We’ll use scientific notation and some of our bits to do so
 3. Should include signed numbers while we’re at it

Topics

- Integer Operations
- Fractional binary numbers
- Intro Binary Operations – Boolean Algebra

Boolean algebra

- You've programmed with **and** and **or** in earlier classes
 - C and C++ logical operations: `&&` (and), `||` (or)
- **Boolean algebra is a generalization of that**
 - A mathematical system to represent logic (propositional logic)
 - 2 truth values: true = **1**, false = **0**
 - Operations: and **&**, or **|**, not (or complement) **~**
 - And others

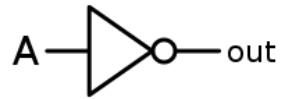
Aside: Propositions, logic, and mathematics

- Propositions affirm or denies a “predicate” of a “subject”
 - “All humans (subject) are mortal (predicate) -> True
 - “London (s) is a capitol city (p) AND London (s) is the capitol of Germany (p)” -> False
- Long ago, Aristotle and others studied logic in terms of rhetoric with philosophy
- In the 18th century, attempts were made to treat formal logic in a symbolic or algebraic way by philosophical mathematicians
 - Leibniz, Lambert
- In the 19th century, Boole and De Morgan developed the mathematical treatment of logic
 - Development of Boolean Algebra and important relationships within it
- In the 20th century, Claude Shannon applied Boolean logic to model the behavior of switching circuits
 - Shannon’s master’s thesis would establish the theory behind digital circuit-based computation!

Truth tables for Boolean algebra

- For each possible value of each input, what is the output
 - Column for each input
 - Column for the output operation

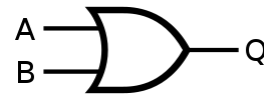
Complement (NOT)



$\sim A$

A	$\sim A$
0	1
1	0

OR



A | B

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

AND



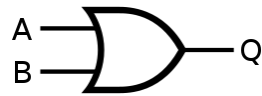
A & B

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

Performing Boolean algebra

- **Follow the rules for each operation to compute results**

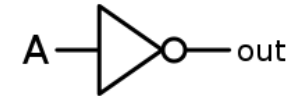
- Rules are like those you know from programming



OR: |



AND: &



NOT: ~

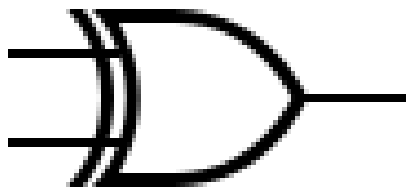
1: True 0: False

$$(1 \mid 0) \& 0 \longrightarrow 1 \& 0 \longrightarrow 0$$

$$(1 \& 1) \& \sim(0 \mid 0) \longrightarrow 1 \& \sim(0) \longrightarrow 1 \& 1 \longrightarrow 1$$

Exclusive Or (xor)

$A \wedge B$		
A	B	$A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0



- An operation you likely haven't used before:
 - Xor - either A or B, but not both
 - $\wedge$ symbol in C
- We can build Xor out of $\&$, $|$, and $\sim$
 - $A \wedge B = (\sim A \& B) | (A \& \sim B)$
 - (exactly one of A and B is true)
 - $A \wedge B = (A | B) \& \sim(A \& B)$
 - (either is true but not both are true)
 - The two definitions are equivalent
 - Produce the same Truth Table

Practice problem

		(A & B) B	
A	B	(A&B) B	
0	0		
0	1		
1	0		
1	1		

Practice problem

(A & B) B		
A	B	(A&B) B
0	0	0
0	1	1
1	0	0
1	1	1

Practice problem

(A & B) B		
A	B	(A&B) B
0	0	0
0	1	1
1	0	0
1	1	1

This is equivalent to B
(A has no influence on the solution)

De Morgan's Law

- We can express Boolean operators in terms of the others
- De Morgan's laws: allow swapping & with |
 - $A \& B = \sim(\sim A \mid \sim B)$ $\rightarrow$ $\sim(A \& B) = \sim A \mid \sim B$
 - (neither A nor B is false)
 - $A \mid B = \sim(\sim A \& \sim B)$ $\rightarrow$ $\sim(A \mid B) = \sim A \& \sim B$
 - (A and B are not both false)
 - Useful for simplifying logical statements

Note: Distributive property in Boolean algebra

- Boolean operation $\&$ distributes over $|$
 - $A \& (B | C) = (A \& B) | (A \& C)$
- Boolean operation $|$ also distributes over $\&$
 - $A | (B \& C) = (A | B) \& (A | C)$

This is different from what we typically assume because we cannot say that $A + (B * C) = (A+B) * (A+C)$ holds for all integers!