CS 250: Foundation of Computer Systems [Binary Representations & Operations]

Powering up with Binary
All you have is a 0 and 1
But the fun's just begun
Simpler math operations
Multiplications simple as additions
Representing numbers on both sides of zero
Using two's complement, our unsung hero

Shrideep Pallickara
Computer Science
Colorado State University

Computer Science Department
Colorado State University

1

Frequently asked questions from the previous class survey

- Hexadecimal?
- Conventions: 0b or base-2, 0x or base-16, or oct/octal for base-8
- CPU cycles and the ALU:
- How does the CPU perform arithmetic operations?
- Does increasing the size of the memory (RAM) impact the speed of a system?
- Where is the Program Counter?
- What is the physical mechanism that allows re-writes on HDDs?
- What makes infinite loops useful? Do we code systems with infinite loops?
- Why not go to a 128-bit system?
- Memory concepts: registers, caches, main memory (RAM)

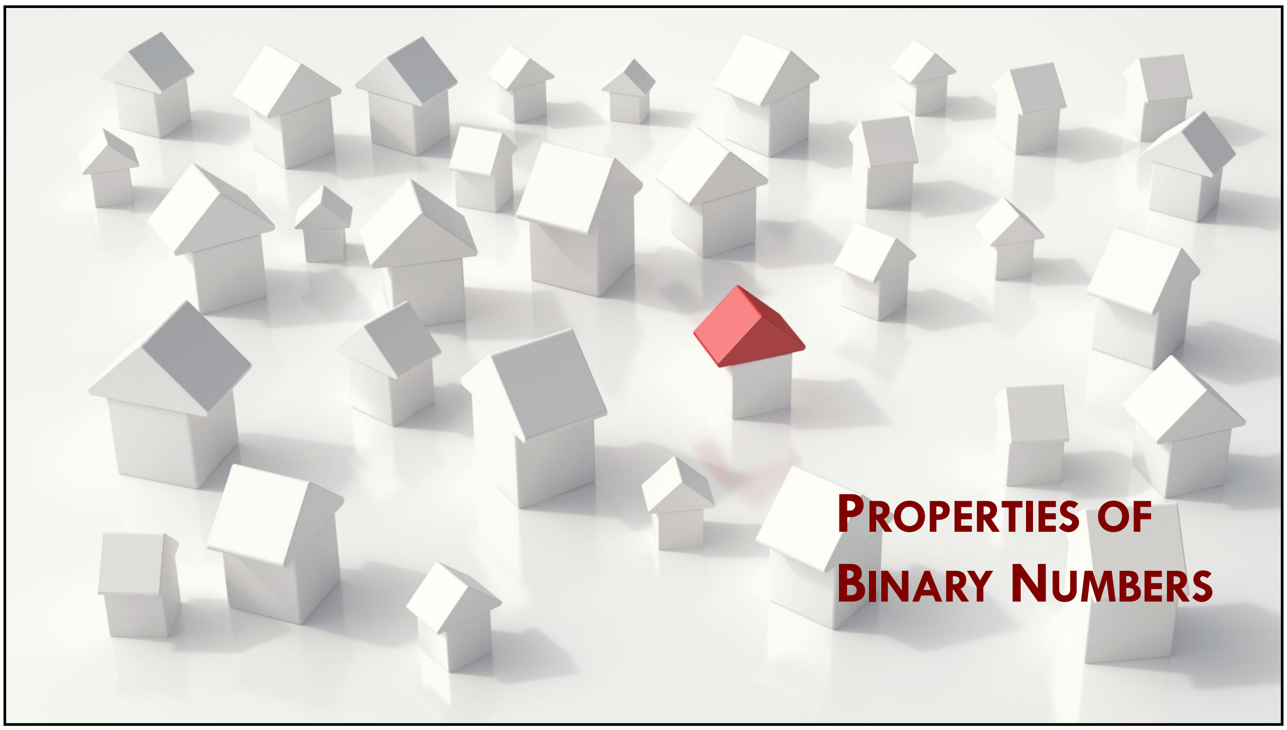
Colorado State University | Professor: Shrideep Pallickara, Computer Science Department | Binary Representations & Operations | L3.2

2

Topics covered in this lecture

- Binary Representations
  - Properties of binary numbers
  - Operations on binary numbers
- Decimal to Binary
- Integers and word size implications
- Signed numbers

3



4

Properties of binary numbers [4/8]

- Shifting all the bits in a number **to the left** by one position **multiplies** the binary value by 2
  - E.g.: 0b00000111 (value = 7)
    - Shift to the left (<<): 0b000001110 (value = $2^3 + 2^2 + 2^1 + 0 = 14$)
  - E.g.: 0b01010111 (value = 87)
    - Shift to the left (<<): 0b010101110
      - Value = $2^7 + 0 + 2^5 + 0 + 2^3 + 2^2 + 2^1 + 0 = 128 + 32 + 8 + 4 + 2 = 174$

Properties of binary numbers [5/8]

- Shifting all the bits of an unsigned binary number **to the right** by one position effectively **divides** that number by 2
  - This does not apply to signed integer values
  - Odd numbers are rounded down
- E.g.: 0b01010110 (value = 86)
  - Shift to the right (>>): 0b0101011~~0~~ = 0b0101011 = 43
- E.g.: 0b01010111 (value = 87)
  - Shift to the right (>>): 0b0101011~~1~~ = 0b0101011 = 43

Properties of binary numbers [6/8]

- Multiplying two *n*-bit binary values together may require *as many as* $2 \times n$ bits to hold the result
- Adding or subtracting two *n*-bit binary values never requires more than $n + 1$ bits to hold the result

Properties of binary numbers [7/8]

- *Incrementing* (adding 1 to) the largest unsigned binary value for a given number of bits always produces a value of 0
  - 0b11111111 = 0b00000000 (the last carryover of 1 **overflows**)
- *Decrementing* (subtracting 1 from) 0 always produces the largest unsigned binary value for a given number of bits

Number of unique combinations in a byte?

- Another way to think about this question is how many unique combinations of 0s and 1s can we make with our 8 bits?
- Let's first illustrate this with 4-bits

16 unique combinations of 0s and 1s in a 4-bit number, ranging in decimal value from 0 to 15

- We could determine the largest possible number that 4 bits can represent by setting all the bits to one, giving us 0b1111
  - That is 15 in decimal
- If we add 1 to account for representing 0, then we come to our total of 16

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Properties of binary numbers [8/8]

- In general, for n bits
  - The total number of unique combinations: 2^n
  - The largest possible number is $2^n - 1$

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT BINARY REPRESENTATIONS & OPERATIONS L3.11

11

DECIMAL TO BINARY

COLORADO STATE UNIVERSITY

12

Conversion from decimal to binary

- The binary number is constructed from **right to left**
- Step 1: Divide the decimal number by 2 and note down the remainder
- Step 2: Divide the obtained quotient by 2, and note remainder again
- Step 3: Repeat the above steps until you get **0** as the quotient
  - **Stopping criteria**

Example: Decimal number 23

- $23 \div 2 =$ **11** (Remainder 1)
- **11** $\div 2 =$ **5** (Remainder 1)
- **5** $\div 2 =$ **2** (Remainder 1)
- **2** $\div 2 =$ **1** (Remainder 0)
- **1** $\div 2 =$ **0** (Remainder 1)

Top to Bottom
is
Right to Left
{LSB to MSB}

- Binary Representation: **10111**
  - $2^4 + 2^2 + 2^1 + 2^0 = 16 + 4 + 2 + 1 = \mathbf{23}$

Example: Decimal number 99

- 99 ÷ 2 = **49** (Remainder 1)
- **49** ÷ 2 = **24** (Remainder 1)
- **24** ÷ 2 = **12** (Remainder 0)
- **12** ÷ 2 = **6** (Remainder 0)
- **6** ÷ 2 = **3** (Remainder 0)
- **3** ÷ 2 = **1** (Remainder 1)
- **1** ÷ 2 = **0** (Remainder 1)

Top to Bottom
is
Right to Left
{LSB to MSB}

- Binary Representation: **1100011**
  - $2^6 + 2^5 + 2^1 + 2^0$ = 64 + 32 + 2 + 1 **= 99**

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT BINARY REPRESENTATIONS & OPERATIONS L3.15

15

PREFIXES FOR LARGE COLLECTIONS

COMPUTER SCIENCE DEPARTMENT COLORADO STATE UNIVERSITY

16

Prefixes

- To more easily communicate the size of data, we use **prefixes** like giga- and mega-
- The International System of Units (SI), also known as the metric system, defines a set of standard prefixes
- These prefixes are used to describe anything that can be quantified, *not just bits*

Common SI Prefixes

Prefix name	Prefix symbol	Base-10 value	English word
Peta	P	10^{15}	quadrillion
Tera	T	10^{12}	trillion
Giga	G	10^{9}	billion
Mega	M	10^{6}	million
Kilo	K	10^{3}	thousand
centi	c	10^{-2}	hundredth
milli	m	10^{-3}	thousandth
micro	µ	10^{-6}	millionth
nano	n	10^{-9}	billionth
pico	p	10^{-12}	trillionth

These prefixes are used to describe anything that can be quantified, not just bits.

When dealing with bytes, most software is actually working in base 2, not base 10

- If your computer tells you that a file is 1MB in size, it is actually 1,048,576 bytes!
  - That is approximately one million, but not quite

Prefix name	Prefix symbol	Value	Base 2
Peta	P	1,125,899,906,842,624	2^{50}
Tera	T	1,099,511,627,776	2^{40}
Giga	G	1,073,741,824	2^{30}
Mega	M	1,048,576	2^{20}
Kilo	K	1024	2^{10}

19

BINARY ARITHMETIC OPERATIONS

20

Addition

- A pair of binary numbers can be added bitwise **from right to left**
  - Using the same decimal addition algorithm learned in elementary school
- First, we add the two rightmost bits, also called the least significant bits (LSB) of the two binary numbers
  - Next, add the resulting **carry** bit to the sum of the next pair of bits
- Continue this lockstep process ***until*** the two left most significant bits (MSB) are added

Adding binary numbers: **Rules**

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0 with **carry**
- Carry + 0 + 0 = 1
- Carry + 0 + 1 = 0 with carry
- Carry + 1 + 0 = 0 with carry
- Carry + 1 + 1 = 1 with carry

Examples: Binary Addition

	0	0	0	1	
# 9		1	0	0	1
# 5		0	1	0	1
	0	1	1	1	0
	No Overflow				

CARRY	1	1	1	1		
		1	0	1	1	# 11
		0	1	1	1	# 7
	1	0	0	1	0	
	Overflow					

Addition of 9 and 5 = 8 + 4 + 2 = 14

Addition of 11 and 7 = 10010_2 = 18 (no truncation)
Addition of 11 and 7 = 0010_2 = 2 (with truncation to 4-bits)

Subtracting binary numbers: **Rules**

- 0 – 0 = 0
- 0 – 1 = 1 with a borrow
- 1 – 0 = 1
- 1 – 1 = 0

Subtraction: An example

		0	b		
# 5		0	1	0	1
# 3		0	0	1	1
		0	0	1	0

Borrow

Multiplication of binary numbers: Rules

- 0 × 0 = 0
- 0 × 1 = 0
- 1 × 0 = 0
- 1 × 1 = 1

Multiplication: An example

			1	0	1	0	#10
			0	1	0	1	# 5
			1	0	1	0	⟵ 1 x 1 0 1 0
							⟵ 0 x 1 0 1 0
	1	0	1	0			⟵ 1 x 1 0 1 0
	1	1	0	0	1	0	

Decimal representations:
10 x 5

0b110010 =
32 + 16+ 2 = 50

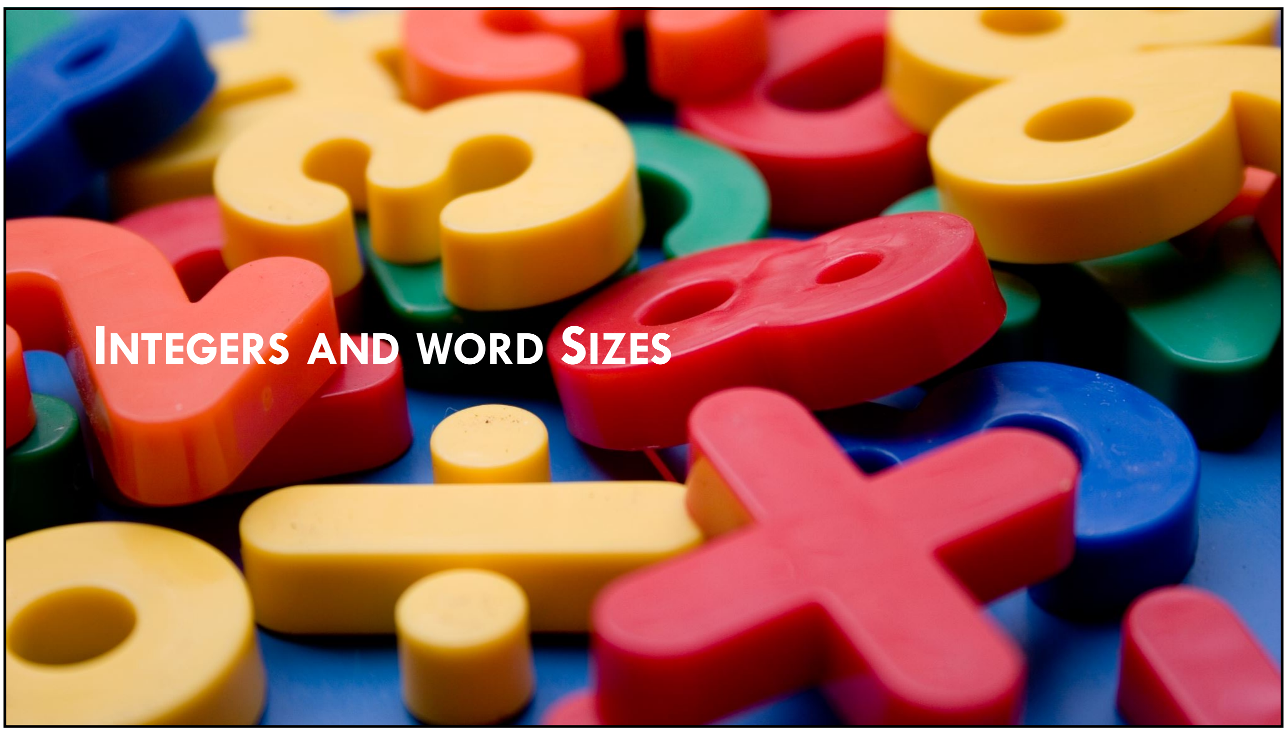
Division: An example

- 3456 ÷ 12 = 288
  - How? Let's take a look at a long-division example

Division in binary

- The basic algorithm is easier in binary because
  - At each step you don't have to guess how many times 12 goes into the remainder or multiply 12 by your guess to obtain the amount to subtract
  - At each step in the binary algorithm, **the divisor goes into the remainder exactly zero or one times**
- Let's take a look at: 0b11011 ÷ 0b11 = 0b1001
  - 27 ÷ 3 = 9

29



30

Integer numbers are, of course, unbounded

- For any given number x:
  - There are integers that are less than x and integers that are greater than x
- Yet computers are *finite machines* that use a **fixed word size** for representing numbers
- **Word size** is a common hardware term used for specifying number of bits that computers use
  - For representing a basic chunk of information — in this case, integer values
  - Typically, 8-, 16-, 32-, or 64-bit registers are used for representing integers

Word size implications

- The **fixed word size** implies that there is a **limit** on the number of values that these registers can represent
- For example, suppose we use 8-bit registers for representing integers
  - This representation can code $2^8 = 256$ different things
  - If we wish to represent only nonnegative integers?
    - We can assign 00000000 for representing 0,
    - 00000001 for representing 1,
    - 00000010 for representing 2, 00000011 for representing 3,
    - All the way up to assigning 11111111 for representing 255

Word size implications

- In general, using n bits we can represent all the nonnegative integers ranging from 0 to $2^n - 1$
  - What about representing negative numbers using binary codes? Soon …

Representing numbers that are > maximal, or < minimal, values permitted by the fixed register size

- Every high-level language provides abstractions for handling numbers that are as large or as small as we can practically want
  - For e.g., java.math.BigInteger ($-2^{\text{Integer.MAX\_VALUE}}$ … $+2^{\text{Integer.MAX\_VALUE}}$)
  - Integer.MAX_VALUE = $2^{31}-1$ = 2,147,483,647
- These abstractions are typically implemented by **lashing together** as many n-bit registers as necessary for representing the numbers
- Since executing arithmetic and logical operations on multi-word numbers is a **slow affair**
  - It is recommended to use this practice only when the application requires processing extremely large or extremely small numbers

SIGNED NUMBERS

There's a sign on the wall, but she wants to be sure
'Cause you know sometimes words have two meanings
Stairway to Heaven, Jimmy Page / Robert Plant; Led Zeppelin

35

Signed Numbers

- An n-bit binary system can code 2^n different things
- If we have to represent **signed** (+ and –) numbers in binary code, a solution is to split the available space into two subsets
  - One for representing nonnegative (+) numbers, and
  - The other for representing negative (–) numbers

36

Desirable properties of signed number representations

- Ideally, the coding scheme should be chosen such that the introduction of signed numbers
  - Complicates the hardware implementation of arithmetic operations **as little as possible**
- This challenge has led to the development of several coding schemes for representing signed numbers in binary code

The solution used today in almost all computers?

- **Two's complement** method, also known as *radix complement*
- Example:
  - Consider a binary system that uses a word size of n bits
  - The two's complement binary code that represents negative x is taken to be the code that represents $2^n - x$
  - Representation of $-x$ ➔ $2^n - x$

Two's complement representation of 4-bit numbers

- Recall, in a n-bit number system
- $-x \rightarrow 2^n - x$
- For example, in a 4-bit binary system:
  - –7 is represented using the binary code associated with $2^4 - 7 = 9$
    - Which happens to be 1001

4-bit binary	Base-10	Derivation
0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	– 8	$2^4 - 8 = 16 - 8 = 8$
1001	– 7	$2^4 - 7 = 16 - 7 = 9$
1010	– 6	$2^4 - 6 = 16 - 6 = 10$
1011	– 5	$2^4 - 5 = 16 - 5 = 11$
1100	– 4	$2^4 - 4 = 16 - 4 = 12$
1101	– 3	$2^4 - 3 = 16 - 3 = 13$
1110	– 2	$2^4 - 2 = 16 - 2 = 14$
1111	– 1	$2^4 - 1 = 16 - 1 = 15$

Let's take a closer look at +7 and –7

- +7 = 0111
- –7 = 1001
- +7 –7 = 0000 (ignoring the overflow bit)

An n-bit binary system with two's complement representation has attractive properties

- The system codes 2^n signed numbers, ranging from -2^{n-1} to $(2^{n-1} - 1)$
- The code of any *nonnegative* number begins with a **0**
- The code of any *negative* number begins with a **1**
- To obtain the binary code of $-x$ from the binary code of x?
  - **Flip all the bits of x and add 1 to the result**

0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	−8	(16−8)
1001	−7	(16−7)
1010	−6	(16−6)
1011	−5	(16−5)
1100	−4	(16−4)
1101	−3	(16−3)
1110	−2	(16−2)
1111	−1	(16−1)

41

Two's complement: binary code of $-x$ from the binary code of x?

- Flip all the bits of x and add 1 to the result

4-bit binary	Base-10	Flip the bits	Add 1	Base-10
0001	1	1110	1111	– 1
0010	2	1101	1110	– 2
0011	3	1100	1101	– 3
0100	4	1011	1100	– 4
0101	5	1010	1011	– 5
0110	6	1001	1010	– 6
0111	7	1000	1001	– 7
1000	8	0111	1000	– 8

0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	−8	(16−8)
1001	−7	(16−7)
1010	−6	(16−6)
1011	−5	(16−5)
1100	−4	(16−4)
1101	−3	(16−3)
1110	−2	(16−2)
1111	−1	(16−1)

42

Another attractive feature of two's complement

- Subtraction is handled as a special case of addition
- To illustrate, consider 5–7 in our 4-bit binary number system
  - This 5 + (–7)
  - 0101 + 1001
  - = 1110
  - Which is –2

Implications of the two's complement method? [1/2]

- The two's complement method enables adding and subtracting signed numbers
  - Using nothing more than the hardware required for adding positive numbers!
- Every arithmetic operation, from multiplication to division to square root, can be implemented reductively using binary addition

Implications of the two's complement method? [2/2]

- A huge range of computer capabilities rides on top of binary addition
- The two's complement method *obviates the need for special hardware* for adding and subtracting signed numbers
- The two's complement method is one of the most **remarkable and unsung** heroes of applied computer science

ONE'S COMPLEMENT

Another useful concept to know … one's complement

- A not so **successful** attempt to represent signed numbers
- Get to negative numbers by taking positive numbers and flipping all the bits (i.e., 1 becomes 0 and 0 becomes 1)

One's complement

- Flipping each bit of 0111 (+7) yields 1000 (–7)
- Has **two** different representations for zero

Sign	2^2	2^1	2^0	Decimal
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	1	1	0	–1
1	1	0	1	–2
1	1	0	0	–3
1	0	1	1	–4
1	0	1	0	–5
1	0	0	1	–6
1	0	0	0	–7

One's complement: Addition is a little more complicated

1

0	0	1	0	+2
1	1	1	0	−1
0	0	0	0	0
				1
				1

End-around carry

The contents of this slide-set are based on the following references

- Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. 2nd Edition. ISBN-10/ ISBN-13: 0262539802 / 978-0262539807. MIT Press. [Preface, Chapter 1-2]
- Jonathan E. Steinhart. *The Secret Life of Programs: Understand Computers -- Craft Better Code*. ISBN-10/ ISBN-13 : 1593279701/ 978-1593279707. No Starch Press. [Chapter 1]
- Randall Hyde. Write Great Code, Volume 1, 2nd Edition: Understanding the Machine 2nd Edition. ASIN: B07VSC1K8Z. No Starch Press. 2020. [Chapter 2]
- Matthew Justice. *How Computers Really Work: A Hands-On Guide to the Inner Workings of the Machine*. ISBN-10/ISBN-13 : 1718500661/ 978-1718500662. No Starch Press. 2020. [Chapter 1]