

CSE 30: Computer Organization and Systems Programming

Lecture 2: Number Representation

Diba Mirza
University of California, San Diego

Reading Quiz!

Note the intimidating red border!

Computers use _____ for their underlying representation of numbers

A. Binary

B. Decimal

C. Hexadecimal

Computers use binary for their underlying representation of numbers because:

- A. Binary is easier to understand by humans than other bases
- B. Logic circuits operate on binary values.
- C. Less memory is needed to store information

Hex (base 16) representation is useful because

- A. It is easy to convert from binary to hex and reverse
- B. It allows more compact notation of large numbers compared to decimal
- C. Both A & B

10 in Binary is __ in Decimal

A. 1

B. 2

C. 3

D. 10

Digits: In Base 6, the set of digits that will be used is

- A. 0,1,2,3,4,5
- B. 0,1,2,3,4,5,6
- C. 0,1,2,3,4,5,6,7,8,9
- D. 1,2,3,4,5,6

External vs. Internal Representation

- External representation:
 - Convenient for programmer – usually decimal or hex
- Internal representation:
 - Actual representation of data in the computer's memory and registers: Always binary (1's and 0's)
- Important to be able to convert between the two

Numbers: Positional Notation

- The meaning of a digit depends on its position in a number.
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ in base b represents the value

$$d_n * b^n + d_{n-1} * b^{n-1} + \dots + d_2 * b^2 + d_1 * b^1 + d_0 * b^0$$

Representation in different bases

- We can represent numbers in any base
- Most widely used are:
 - Base 2 (binary)
 - Base 8 (Octal)
 - Base 10 (decimal)
 - Base 16 (hex)

Which base do we use?

- **Decimal (base 10)**: great for humans, especially when doing arithmetic
- **Hex (base 16)**: if a human is looking at long strings of binary numbers, its much easier to convert to hex and look 4 bits/symbol
 - Terrible for arithmetic on paper
- **Binary (base 2)**: what computers use; you will learn how computers do $+$, $-$, $*$, $/$
 - To a computer, numbers always binary
 - Regardless of how number is written:
 - $32_{\text{ten}} == 32_{10} == 0x20 == 100000_2 == 0b100000$
 - Use subscripts “ten” , “hex” , “two”

Digits: In Base 6, the set of digits that will be used is

A. 0,1,2,3,4,5

B. 0,1,2,3,4,5,6

C. 0,1,2,3,4,5,6,7,8,9

D. 1,2,3,4,5,6

Consider 101

- In base 10, it represents the number 101 (one hundred one) = $1 * 10^2 + 0 * 10^1 + 1 * 10^0$
= 101
- In base 2, $101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
= 4 + 1 = 5
- In base 16, $101_{16} = 1 * 16^2 + 0 * 16^1 + 1 * 16^0$
= 256 + 1 = 257

$$101_5 = ?$$

A. 26

B. 51

C. 126

D. 130

Binary: Base 2

- Used by computers
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0,1\}$, represents the value

$$d_n * 2^n + d_{n-1} * 2^{n-1} + \dots + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0$$

Computers Use Binary Because

- A. Decimal takes too much space
- B. It's easier to do math with binary
- C. It is easy to represent two states (on/off) with electricity

Decimal: Base 10

- Used by humans
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0,1,2,3,4,5,6,7,8,9\}$, represents the value

$$d_n * 10^n + d_{n-1} * 10^{n-1} + \dots + d_2 * 10^2 + d_1 * 10^1 + d_0 * 10^0$$

Octal: Base 8

- Allows use of 7 segment displays, nixie tubes, etc
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0,1,2,3,4,5,6,7\}$, represents the value

$$d_n * 8^n + d_{n-1} * 8^{n-1} + \dots + d_2 * 8^2 + d_1 * 8^1 + d_0 * 8^0$$

How many bits are needed to
represent any hex digit

A. 6

B. 5

C. 4

D. 3

Hexadecimal: Base 16

- Like binary, but shorter!
- Each digit is a “nibble”, or half a byte
- Indicated by prefacing number with 0x
- A number, written as the sequence of digits $d_n d_{n-1} \dots d_2 d_1 d_0$ where d is in $\{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$, represents the value
$$d_n * 16^n + d_{n-1} * 16^{n-1} + \dots + d_2 * 16^2 + d_1 * 16^1 + d_0 * 16^0$$
- Table 1.1, page 13 (ARM) shows the decimal, hex and binary equivalents of all hex digits

Hexadecimal: Base 16

- Hex is frequently used in programs
- In many problems e.g. generating the header of a data communication packet: you would like to be able to visualize the data at a bit level, which is much easier with hex
- Compact notation but allows bit level visualization

Base Conversions

- You should be able to convert between any two bases
 - Decimal to Binary
 - Decimal to hex
 - Hex to binary

Binary to Decimal

$10110_2 = ?$ In decimal

Binary to Decimal

- Use polynomial expansion

$$10110_2 = 1*16 + 1*4 + 1*2 = 22$$

Decimal to binary: $34_{10}=?_2$

A. 10001

B. 10010

C. 100010

D. 111110

Decimal to Binary

- Repeatedly divide by 2, recording the remainders.
- The remainders form the binary digits of the number.
- Converting 39 to binary

	<u>0</u>	1
2	<u>1</u>	0
2	<u>2</u>	0
2	<u>4</u>	1
2	<u>9</u>	1
2	<u>19</u>	1
2	39	

$\Rightarrow 39 = 100111_2$

Hex to binary: $23C_{16} = ?_2$

A. 10001

B. 10111110

C. 1000111100

D. 1000011011000

Hexadecimal to binary

- Each hex digit corresponds directly to four binary digits
- $35AE_{16} = 0011010110101110_2$ (3 = 0011, 5 = 0101, A = 1010, E = 1110)

Every hex digit corresponds to 4 binary digits, how many binary digits does an octal digit correspond to?

A. 2

B. 3

C. 4

D. 5

Hexadecimal to decimal

$$25B_{16} = ? \text{ Decimal}$$

Hexadecimal to decimal

- Use polynomial expansion
- $25B_{16} = 2*256 + 5*16 + 11*1 = 512 + 80 + 11 = 603$

Decimal to hex: $26_{10}=?_{16}$

A. 19

B. 1A

C. 1B

Decimal to hexadecimal

- Use repeated division by 16.
- Example: Convert 701 to hexadecimal.

$$\begin{array}{r} \underline{ 0} \\ 16 \mid \underline{ 2} \\ 16 \mid \underline{43} \\ 16 \mid 701 \end{array}$$

$\Rightarrow 701 = 2BD_{16}$

BIG IDEA: Bits can represent anything!!

- Numbers:
 - integers (signed, unsigned)
 - floats
- Characters:
 - 26 letters using 5 bits ($2^5 = 32$)
 - upper/lower case + punctuation: 128 characters using 7 bits (“ASCII”)
 - standard code to cover all the world’s languages 8,16,32 bits (“Unicode”) www.unicode.com
- Logical values:
 - 0: False, 1:True
- Colors: Example  *Red (00)*  *Green (01)*  *Blue (11)*
- locations / addresses, instructions
- **MEMORIZE:** N bits can be used to represent at most 2^N things

How We Store Numbers

- Binary numbers in memory are stored using a finite, fixed number of bits typically:
 - 8 bits (byte)
 - 16 bits (half word)
 - 32 bits (word)
 - 64 bits (double word or quad)
- If positive pad extra digits with leading 0s
- A byte representing $4_{10} = 00000100$

Integer representation

- You have to think about representation and storage together
- The number of bits available (finite and fixed) determines:
 - the exact representation of numbers in memory
 - the range of values that can be correctly represented

Integer representation

We will look at representation for:

- Unsigned numbers
 - Only positive values
 - All the bits determine the magnitude of the positive number
- Signed numbers
 - These can be positive or negative
 - We cannot use all the bits to just represent the magnitude

A byte can store unsigned values up to

A. 127

B. 128

C. 255

D. 256

Unsigned Numbers

- Some types of numbers, such as memory addresses, will never be negative
- Some programming languages reflect this with types such as “unsigned int”, which only hold positive numbers
- In an unsigned byte, values will range from 0 to 255

How to Represent Signed Numbers?

- So far, unsigned numbers
- Obvious solution: define leftmost bit to be sign!
- Representation called sign and magnitude
 - Have a separate bit for sign (Most significant Bit MSB)
 - Set it to 0 for positive, and 1 for negative

A byte representing -112_{10} in Sign and Magnitude (with leftmost bit for sign) is

A. 01110000

B. 01110001

C. 11110000

D. 10000111

In sign and magnitude, we will be able to represent _____ in one byte

- A. -63 to 63
- B. -127 to 127
- C. -128 to 127
- D. -255 to 255
- E. 0 to 255

Number Representation

- Just what we do with numbers!

- Add them

- Subtract them

- Multiply them

- Divide them

- Compare them

$$\begin{array}{rcccc} & \textcolor{red}{1} & \textcolor{red}{1} & & \\ & 1 & 0 & 1 & 0 \\ + & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

- Example: $10 + 7 = 17$

$$\begin{array}{rccccc} 1 & 0 & 0 & 0 & 1 \end{array}$$

- ...so simple to add in binary that we can build circuits to do it!

- subtraction just as you would in decimal

- Comparison: How do you tell if $X > Y$?

Which is NOT a drawback of Sign and Magnitude?

- A. There are two zeros
- B. Need a separate step to assign sign in addition
- C. Difficult to convert numbers to negative representation

Shortcomings of sign and magnitude?

- Arithmetic circuit complicated
 - Special steps depending whether signs are the same or not
- Also, two zeros
 - $0x00000000 = +0_{\text{ten}}$
 - $0x80000000 = -0_{\text{ten}}$
- Therefore sign and magnitude abandoned !

One's Complement

- To make a number negative, just flip all its bits!
- For n bits, the unsigned version of $-x = 2^n - x - 1$
- Still have two zeros (11111111 and 00000000)
- Instead of padding with leading 0s, now extend the sign bits

What is the range of numbers we can represent in One's complement format using N bits

A. -2^{N-1} to 2^{N-1}

B. -2^{N-1} to $2^{N-1} - 1$

C. $-(2^{N-1}-1)$ to $2^{N-1}-1$

D. -2^N to 2^N

A byte representing -112_{10} in One's Complement is

A. 00001111

B. 01110000

C. 10001111

D. 11110000

What is $-112_{10} + 112_{10}$ using a byte wide one's complement format for each number?

A. 11111111_2

B. 00000000_2

C. Either A or B

Short comings of One's Complement

- Still have two zeros (11111111 and 00000000)
- Need an extra step in addition if there is a carry out – see example 1.6 in ARM book

Two's Complement

- Flip all the bits and add 1
- For n bits, in unsigned version $-x = 2^n - x$
- Can represent -128 to 127 in a byte
- Only one zero (00000000)
- Used in modern computers

-112 in Two's Complement

A. 01110001

B. 10001110

C. 10010000

D. 10001111

Two's Complement: $1101_2 = ?_{10}$

A. -2

B. -3

C. -4

D. -5

Two's Complement

- Each two's complement number is now
$$-2^{n-1} * d_{n-1} + 2^{n-2} * d_{n-2} + \dots + 2^1 * d_1 + 2^0 * d_0$$

Negating Two's Complement

- To negate any number, flip all the bits and add 1

The negation of 11110001_2 is _____₂

A. 00001110

B. 00001111

C. 00011110

D. 01110001

Negating in Hex

- Obtain the hex representation.
- Replace each digit by $(15 - \text{digit})$.
- Add one.

The negation of $00A6_{16}$ is _____₁₆

A. 115A

B. 1154

C. FF5A

D. FF58

Addition and Subtraction

- Positive and negative numbers are handled in the same way.
- The carry out from the most significant bit is ignored.
- To perform the subtraction $A - B$, compute $A +$ (two's complement of B)

$$00000200_{16} + \text{FFFFFFFF}_{16} = ?_{16}$$

A. 000001FF

B. 000002FF

C. FFFFFFFDFF

D. FFFFFFF1FF

Overflow

- Overflow occurs when an addition or subtraction results in a value which cannot be represented using the number of bits available.
- In that case, the algorithms we have been using produce incorrect results.

Is overflow a problem in modern programs?

- A. We have totally solved this business!
- B. Yep, still a problem.

Handling Overflow

- Hardware can detect when overflow occurs
- Software may or may not check for it
 - C and Java don't!

Detecting Overflow in hardware

- In the ARM architecture, there is a special register: the Current Program Status Register that has 4 bits (flags) that are set after an arithmetic operation is executed based on what 'happened'. See ARM Chapter 7, section 7.2



- N flag: 1 if result is negative
- Z flag: 1 if result is zero
- V flag: 1 if oVerflow (more next slide)
- C flag: 1 if there is a Carry out of MSB

How To Detect Overflow

- Overflow occurs if adding two negative numbers produces a positive result or if adding two positive numbers produces a negative result.
- On an addition, an overflow occurs if and only if the carry into the sign bit differs from the carry out from the sign bit.
- Overflow detected using V flag which is set as:
 $C_{in} \text{ XOR } C_{out}$

Will $0111111_2 + 00000101_2$ result in overflow?

A. Yes

B. No

C. It depends

Subtract using 2's complement 16 bit
representation: $204_{10} - 46_{10}$.
what will the NZVC flags be?

A. $N=0, Z=0, V=0, C=0$

B. $N=0, Z=0, V=0, C=1$

C. $N=0, Z=0, V=1, C=1$

D. $N=0, Z=0, V=1, C=0$

Add using 8 bit 2's complement
representation: $72_{10} - 64_{10}$.
what will the NZVC flags be?

A. $N=0, Z=0, V=0, C=1$

B. $N=1, Z=0, V=0, C=0$

C. $N=0, Z=0, V=1, C=1$