

# Background/Review on Numbers and Computers

ICS312 - Spring 2009  
Machine-Level and  
Systems Programming

Henri Casanova (henric@hawaii.edu)

## Numbers and Computers

- Throughout this course we will
  - use **binary** and **hexadecimal** representations of numbers
  - need to be aware of the ways in which the computer stores numbers
- So let us go through a simple review before we start learning how to write assembly code
  - Numbers in different bases
  - Number representation in computers and basic arithmetic

## Numbers and bases

- We are used to thinking of numbers as written in decimal, that is, in base 10
 
$$25 = 2 \cdot 10^1 + 5 \cdot 10^0$$

$$136 = 1 \cdot 10^2 + 3 \cdot 10^1 + 6 \cdot 10^0$$
  - Each number can be decomposed into a sum of terms
  - Each term is the product of two factors
    - A digit (from 0 to 9)
    - The base (10) raised to a power corresponding to the digit's position in the number
- $$136 = \dots + 0 \cdot 10^4 + 0 \cdot 10^3 + 1 \cdot 10^2 + 3 \cdot 10^1 + 6 \cdot 10^0$$
- $$= \dots 00000136$$

## Numbers and Bases

- Any number can be written in base b, using b digits
  - If b = 10 we have "**decimal**" with 10 digits [0-9]
  - If b = 2 we have "**binary**" with 2 digits [0-1], which are also called **bits**
  - If b = 8 we have "**octal**" with 8 digits [0-7]
  - If b = 16 we have "**hexadecimal**" with 16 digits [0-9,A,B,C,D,E,F]
- Computers use binary internally
  - Since Shannon made a case for it and because it is easy to associate two states to a current
    - Low voltage = 0, high voltage = 1
    - Associating 16 states to a current is more complicated and error-prone
- However, binary is cumbersome
  - The lower the base the longer the numbers!
  - It's really difficult for a human to remember binary
- Therefore we, as humans, like to use higher bases
- **Bases that are powers of 2 make for easy translation to binary**, and thus are particularly useful, and in particular hexadecimal

## Binary Numbers

- Counting in binary:
 

$0_2$	$0_{10}$
$1_2$	$1_{10}$
$10_2$	$2_{10}$
$11_2$	$3_{10}$
$100_2$	$4_{10}$
$101_2$	$5_{10}$
$110_2$	$6_{10}$
$111_2$	$7_{10}$
$1000_2$	$8_{10}$
...	
- A binary number with d bits corresponds to integer values between 0 and  $2^d - 1$ 

$$\sum_{k=0}^{d-1} 2^k = 2^d - 1$$
- Example:
  - An integer stored in 8 bits has values between 0 and 255
  - $128+64+32+16+8+4+2+1 = 255$

## Converting from Binary to Decimal

- We denote by  $XXXX_2$  a binary representation of a number and by  $XXXX_{10}$  a decimal representation
- Converting from binary to decimal is straightforward:
 
$$10010110_2 = 1 \cdot 2^7 + 1 \cdot 2^4 + 1 \cdot 2^2 + 1 \cdot 2^1$$

$$= 1 \cdot 128 + 1 \cdot 16 + 1 \cdot 4 + 1 \cdot 2$$

$$= 150_{10}$$
- The rightmost bit of a binary number is called the **least significant bit**
- The leftmost non-zero bit of a binary number is called the **most significant bit**
- **If the least significant bit is 0, then the number is even, otherwise it's odd**

## Converting from Decimal to Binary

- The conversion here is a little bit more complicated
- It proceeds by a series of **integer divisions** by 2, and by recording the remainder of the division
  - Integer division  $a/b$ :  $a = b \cdot q + \text{remainder}$ , where all are integers
- Example: converting  $37_{10}$  into binary
  - Divide 37 by 2:  $37 = 2 \cdot 18 + 1$
  - Divide 18 by 2:  $18 = 2 \cdot 9 + 0$
  - Divide 9 by 2:  $9 = 2 \cdot 4 + 1$
  - Divide 4 by 2:  $4 = 2 \cdot 2 + 0$
  - Divide 2 by 2:  $2 = 2 \cdot 1 + 0$
  - Divide 1 by 2:  $1 = 2 \cdot 0 + 1$
  - Result:  $100101_2$
- The **least significant bit is computed first**
- The **most significant bit is computed last**
- Note that if we continue dividing, we get extraneous leading 0s
  - $\dots 00000100101_2$

## Binary Arithmetic

- Adding a 0 to the right of a binary number multiplies it by 2
  - $10101_2 = 16_{10} + 4_{10} + 1_{10} = 21_{10}$
  - $101010_2 = 32_{10} + 8_{10} + 2_{10} = 42_{10}$
- Adding two binary numbers is just like adding decimal numbers: using a carry

With no previous carry				With a previous carry			
0	0	1	1	0	0	1	1
+ 0	+ 1	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1
= 0	= 1	= 1	= 0	= 1	= 0	= 0	= 1
			<b>C</b>		<b>C</b>	<b>C</b>	<b>C</b>

## Binary Addition

$$\begin{array}{r}
 \text{c c c c} \\
 1001 \\
 + 1111 \\
 \hline
 = 11000
 \end{array}
 \qquad
 \begin{array}{r}
 9_{10} \\
 + 15_{10} \\
 \hline
 = 24_{10}
 \end{array}$$
  

$$\begin{array}{r}
 \text{c} \qquad \text{c c} \\
 10100110 \\
 + 11000011 \\
 \hline
 = 101101001
 \end{array}
 \qquad
 \begin{array}{r}
 166_{10} \\
 + 195_{10} \\
 \hline
 = 361_{10}
 \end{array}$$

## Counting in Hexadecimal

$0_{16} = 0_{10}$	$A_{16} = 10_{10}$	$14_{16} = 20_{10}$	$1E_{16} = 30_{10}$
$1_{16} = 1_{10}$	$B_{16} = 11_{10}$	$15_{16} = 21_{10}$	$1F_{16} = 31_{10}$
$2_{16} = 2_{10}$	$C_{16} = 12_{10}$	$16_{16} = 22_{10}$	$20_{16} = 32_{10}$
$3_{16} = 3_{10}$	$D_{16} = 13_{10}$	$17_{16} = 23_{10}$	$21_{16} = 33_{10}$
$4_{16} = 4_{10}$	$E_{16} = 14_{10}$	$18_{16} = 24_{10}$	$22_{16} = 34_{10}$
$5_{16} = 5_{10}$	$F_{16} = 15_{10}$	$19_{16} = 25_{10}$	$23_{16} = 35_{10}$
$6_{16} = 6_{10}$	$10_{16} = 16_{10}$	$1A_{16} = 26_{10}$	$24_{16} = 36_{10}$
$7_{16} = 7_{10}$	$11_{16} = 17_{10}$	$1B_{16} = 27_{10}$	$25_{16} = 37_{10}$
$8_{16} = 8_{10}$	$12_{16} = 18_{10}$	$1C_{16} = 28_{10}$	$26_{16} = 38_{10}$
$9_{16} = 9_{10}$	$13_{16} = 19_{10}$	$1D_{16} = 29_{10}$	$27_{16} = 39_{10}$

## Converting from hex to decimal

- This is again straightforward
- $$\begin{aligned}
 A203DE_{16} &= 10 \cdot 16^5 + 2 \cdot 16^4 + \\
 &\quad 3 \cdot 16^2 + 13 \cdot 16^1 + 14 \cdot 16^0 \\
 &= 10,617,822_{10}
 \end{aligned}$$

## Converting from decimal to hex

- Use the same idea as for binary
- Example: convert  $1237_{10}$ 
  - $1237 = 77 \cdot 16 + 5$
  - $77 = 4 \cdot 16 + 13$
  - $4 = 0 \cdot 16 + 4$
  - Result:  $4D5_{16}$

## Hexadecimal addition

$$\begin{array}{r} \text{c} \\ \text{A 2 3 F} \\ + \text{3 D 1 3} \\ \hline = \text{D F 5 2} \end{array} \quad \begin{array}{r} 41535_{10} \\ + 15635_{10} \\ \hline = 57170_{10} \end{array}$$

$$\begin{array}{r} \text{c} \quad \text{c} \quad \text{c} \\ \text{D 1 F F} \\ + \text{A 4 D F} \\ \hline = \text{1 7 6 D E} \end{array} \quad \begin{array}{r} 53759_{10} \\ + 42207_{10} \\ \hline = 95965_{10} \end{array}$$

## Why is hexadecimal useful?

- We need to think in binary because computers operate on binary quantities
- But binary is cumbersome
- However, hexadecimal makes it possible to represent binary quantities in a **compact form**
- Conversions back and forth from binary to hex are straightforward
  - Just convert hex digits into 4-bit numbers
  - Just convert 4-bit binary numbers into hex digits

## Converting from hex to binary

- Consider  $A43FE2_{16}$
- We convert each hex digit into a 4-bit binary number:
  - $A_{16}$ :  $1010_2$
  - $4_{16}$ :  $0100_2$
  - $3_{16}$ :  $0011_2$
  - $F_{16}$ :  $1111_2$
  - $E_{16}$ :  $1110_2$
  - $2_{16}$ :  $0010_2$
- We “glue” them all together:
  - $A43FE2_{16} = 10100100001111111100010_2$
- Note that:
  - You must have the leading 0's for the 4-bit numbers, which is what a computer would store anyway
  - It all works because  $F_{16} = 15_{10}$ , and a 4-bit number has maximum value of  $2^4 - 1 = 15_{10}$

## Converting from binary to hex

- Let's convert  $1001010101111_2$  into hex
- We split it in 4-bit numbers, which we convert separately
- First we add leading 0's to have a number of bits that's a multiple of 4:  
 $0001\ 0010\ 1010\ 1111$
- Then we convert
  - $0001_2$ :  $1_{16}$
  - $0010_2$ :  $2_{16}$
  - $1010_2$ :  $A_{16}$
  - $1111_2$ :  $F_{16}$
- And the result:  $1001010101111_2 = 12AF_{16}$

## Integer representation

- A computer needs to store integers
- These numbers are stored using different numbers of bytes (1 byte = 8 bits):
  - 1-byte: “byte”
  - 2-byte: “half word” (or “word”)
  - 4-byte: “word” (or “double word”)
  - 8-byte: “double word” (or “paragraph”, or “quadword”)
  - Different computers have used different word sizes, so it's always a bit confusing to just talk about a “word” without any context
- Regardless of the number of bytes, these integers are stored in binary
- Integers come in two flavors:
  - **Unsigned**: values from 0 to  $2^b - 1$
  - **Signed**: negatives values, with about the same number of negative values as the number of positive values
- You can actually declare variables as signed or unsigned in some high-level programming languages, like C

## Sign-Magnitude

- Storing unsigned integers is easy: just store the bits of the integer's binary representation
- Storing signed integer raises a question: how to store the sign?
- One approach is called **sign-magnitude**: reserve the leftmost bit to represent the sign
  - $00100101$  denotes  $+0100101_2$
  - $10100101$  denotes  $-0100101_2$
- It's very easy to negate a number: just flip the leftmost bit
- Unfortunately, sign-magnitude complicates the logic of the CPU (i.e., ICS331-type stuff)
  - There are two representations for zero:  $10000000$  and  $00000000$
  - Some operations are thus more complicated to implement in hardware

## One's complement

- Another idea to store a negative number is to take the complement (i.e., flip all bits) of its positive counterpart
- Example: I want to store integer -87
  - $87_{10} = 01010111_2$
  - $-87_{10} = 10101000$
- Simple, but still two representations for 0: 00000000 and 11111111
- It turns out that computer logic to deal with 1's complement arithmetic is complicated
- Note: it's easy to compute the 1's complement of a number represented in hexadecimal
  - let's consider:  $57_{16}$
  - Subtract each hex digit from F:  $F-5=A$ ,  $F-7=8 \rightarrow A8$
  - the 1's complement of  $57_{16}$  is  $A8_{16}$

## Two's complement

- While sign-magnitude and 1's complement were used in older computers, **nowadays all computers use 2's complement**
- Computing the 2's complement is in 2 steps:
  - Compute the 1's complement of the positive number
  - Add 1 to the result
  - The result is the representation of the negative number
- Example: Let's represent  $-87_{10}$ 
  - $87_{10} = 01010111_2$  or  $57_{16}$
  - 1's complement:  $10101000_2$  or  $A8_{16}$
  - Add one:  $10101001_2$  or  $A9_{16}$
- Let's invert again
  - We start with  $A9_{16}$
  - Invert:  $56_{16}$
  - Add one:  $57_{16}$

## Two's complement

- **Note that when adding 1 in the second step a carry may be generated but is ignored!**
  - Difference between **arithmetic** and **computer arithmetic**
  - When adding two X-bit quantities in a computer one always obtain another X-bit quantity ( $X=8, 16, 32, \dots$ )
- Example: Computing 2's complement of 00000000
  - Take the invert: 11111111
  - Add one: 00000000 with a carry generated!
    - Should be a 9-bit quantity: 100000000
- The above example shows that 0 has only one representation: a signed byte can store values from -128 to +127 ( $128 < 0$  values, and  $128 \geq 0$  values)
- It turns out that 2's complement makes for very simple arithmetic logic when building ALUs
- **Important:** The leftmost bit still indicates the sign of the number (0: positive, 1: negative)
  - In hex, if the left-most "digit" is 8, 9, A, B, C, D, E, or F, then the number is negative, otherwise it is positive

## In-class Exercise

- Conversions:
  - What is  $123_{10}$  in binary?
  - What is  $F3EA_{16}$  in binary?
  - What is  $111_{10}$  in hexadecimal?
  - What is  $100110_2$  in hexadecimal?

## In-class Exercise

- Conversions:
  - What is  $123_{10}$  in binary?  
 $1111011$  ( $127 - 4$ )
  - What is  $F3EA_{16}$  in binary?  
 $1111001111101010$
  - What is  $111_{10}$  in hexadecimal?  
 $6F$  ( $112 - 1$ )
  - What is  $100110_2$  in hexadecimal?  
 $26$

## In-class Exercise

- Arithmetic
  - What is:  $10101101 + 11001011$  ?
  - What is:  $A5F + E32$  ?

## In-class Exercise

### ■ Arithmetic

- What is:  $10101101 + 11001011$  ?

```
10101101
+ 11001011
= 101111000
```

- What is:  $A5F + E32$  ?

```
A5F
+ E32
= 1891
```

## In-class Exercises

### ■ Two's complement:

- What is the 2's complement 2-byte representation of  $-153_{10}$  in hexadecimal?
- What is the decimal value of  $FF4A_{16}$ , a 2-byte numbers stored in 2's complement fashion?

## In-class Exercises

### ■ Two's complement:

- What is the 2's complement representation of  $-153_{10}$  in hexadecimal?

```
15310 = 009916
complement: FF6616
add 1: FF6716
```

- What is the decimal value of  $FF4A_{16}$ , a 2-byte numbers stored in 2's complement fashion?

```
FF4A16 = 1...2
Therefore it represents a negative number, let's invert it
Invert: 00B5
Add 1: 00B6 = B6
B616 = 11*16 + 6 = 176 + 6 = 18210
Therefore, in 2's complement notation, FF4A16 is -18210
```

## Ranges of Numbers

### ■ For 1-byte values

- Unsigned
  - Smallest value: 00 (0d)
  - Largest value: FF (255d)
- Signed
  - Smallest value: 80 (-128d)
  - Largest value: 7F (+127d)
  - FF: -1d

### ■ For 2-byte values

- Unsigned
  - Smallest value: 0000 (0d)
  - Largest value: FFFF (65,535d)
- Signed
  - Smallest value: 8000 (-32,768)
  - Largest value: 7FFF (+32,767)
  - FFFF: -1d

### ■ etc.

## Conclusion

- When writing assembly code we will need to manipulate data
  - Make sure you can do all conversions quickly (**HW #1**)
- An important point is that a computer simply stores data as bits
- The computer internally has no idea what the data means
- We, as programmers have precise interpretations of what bits mean
  - "I store a 4-byte integer", "I store a 1-byte integer which is really the ASCII code for a character", etc.
- When using a high-level language like C or Java, the language helps us tell the computer what data means
  - "I declare x as an int and y as a char"
- But when writing assembly code, we don't have this notion of "data types"
- We have to constantly keep in mind what data is and which instructions to use on it depending on our interpretation of the data
- The Instruction Set Architecture simply provides us with many instructions that operate on all types of data
- It's our role to use the instructions that correspond to how we want to treat the data (using the wrong instruction on the data often will just produce bogus results without generating an error)
- This is one of the difficulties of assembly programming

## Homework #1

- Homework #1 will be assigned shortly