
ELEC2041
Microprocessors and Interfacing

Lecture 6: Number Systems – I

<http://webct.edtec.unsw.edu.au/>

March 2006

Saeid Nooshabadi
saeid@unsw.edu.au

ELEC2041 lec06-numbers-I.1

Saeid Nooshabadi

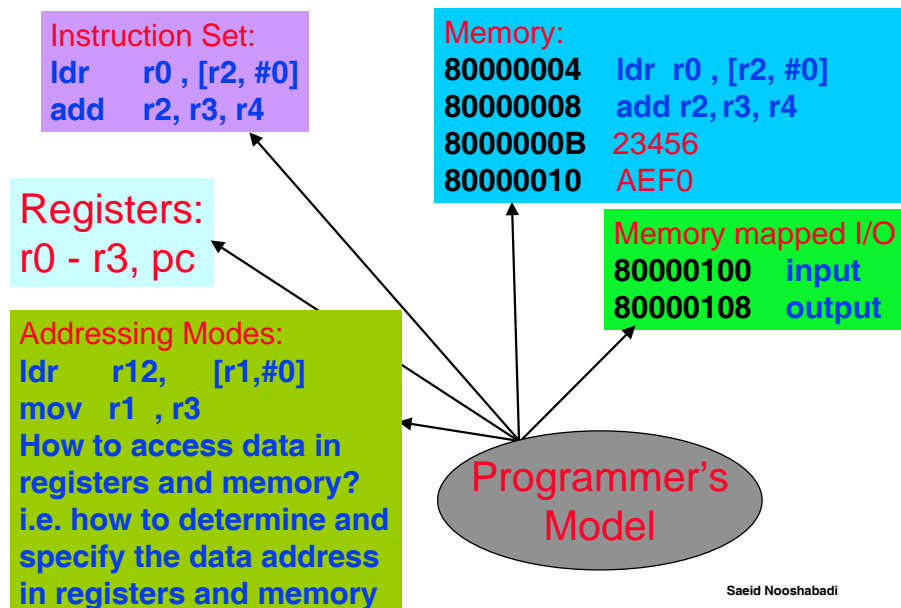
Overview

- Computer representation of “things”
- Unsigned Numbers
- Signed Numbers: search for a good representation
- Shortcuts
- In Conclusion

ELEC2041 lec06-numbers-I.2

Saeid Nooshabadi

Review: The Programmer’s Model of a Microcomputer



Saeid Nooshabadi

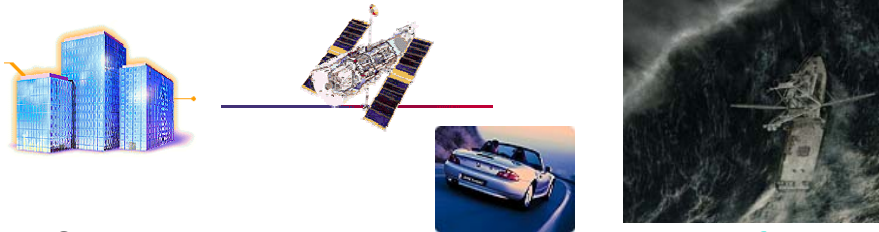
Review: Compilation

- How to turn notation programmers prefer into notation computer understands?
- Program to translate C statements into Assembly Language instructions; called a **compiler**
- Example: compile by hand this C code:
`a = b + c;`
`d = a - e;`
- Easy: `add r1, r2, r3`
`sub r4, r1, r6`
- **Big Idea:** compiler translates notation from 1 level of abstraction to lower level

ELEC2041 lec06-numbers-I.4

Saeid Nooshabadi

What do computers



- Computers **manipulate representations of things!**
- What can you represent with N bits?
 - 2^N things!
- Which things?
 - Numbers! Characters! Pixels! Dollars! Position! Instructions! ...
 - Depends on what operations you do on them

Decimal Numbers: Base 10

- Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Example:
 $3271 = (3 \times 10^3) + (2 \times 10^2) + (7 \times 10^1) + (1 \times 10^0)$

Numbers: positional notation

- Number Base B => B symbols per digit:
 - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
 - Base 2 (Binary): 0, 1
- Number representation:
 - $d_{31}d_{30} \dots d_2d_1d_0$ is a 32 digit number
 - value = $d_{31} \times B^{31} + d_{30} \times B^{30} + \dots + d_2 \times B^2 + d_1 \times B^1 + d_0 \times B^0$
- Binary: 0,1
 - $1011010 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 \times 1 = 64 + 16 + 8 + 2 = 90$
 - Notice that 7 digit binary number turns into a 2 digit decimal number
 - A base that converts to binary easily?

Hexadecimal Numbers: Base 16 (#1/2)

- Digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Normal digits have expected values
- In addition:
 - A → 10
 - B → 11
 - C → 12
 - D → 13
 - E → 14
 - F → 15

Hexadecimal Numbers: Base 16 (#2/2)

° Example (convert hex to decimal):

$$\begin{aligned} B28F0DD &= (B \times 16^6) + (2 \times 16^5) + (8 \times 16^4) + (F \times 16^3) + (0 \times 16^2) + \\ &\quad (D \times 16^1) + (D \times 16^0) \\ &= (11 \times 16^6) + (2 \times 16^5) + (8 \times 16^4) + (15 \times 16^3) + \\ &\quad (0 \times 16^2) + (13 \times 16^1) + (13 \times 16^0) \\ &= 187232477 \text{ decimal} \end{aligned}$$

° Notice that a 7 digit hex number turns out to be a 9 digit decimal number

Decimal vs. Hexadecimal vs. Binary

° Examples:

° 1010 1100 0101 (binary)
= ? (hex)

° 10111 (binary)
= 0001 0111 (binary)
= ? (hex)

° 3F9(hex)
= ? (binary)

00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Hex to Binary Conversion

- ° HEX is a more compact representation of Binary!
- ° Each hex digit represents 16 decimal values.
- ° Four binary digits represent 16 decimal values.
- ° Therefore, each hex digit can replace four binary digits.
- ° Example:

0011 1011 1001 1010 1100 1010 0000 0000_{two}
3 b 9 a c a 0 0_{hex}
C uses notation 0x3b9aca00

Which Base Should We Use?

- ° Decimal: Great for humans; most arithmetic is done with these.
- ° Binary: This is what computers use, so get used to them. Become familiar with how to do basic arithmetic with them (+, -, *, /).
- ° Hex: Terrible for arithmetic; **but if we are looking at long strings of binary numbers, it's much easier to convert them to hex in order to look at four bits at a time.**

Limits of Computer Numbers

- Bits can represent anything!
- Characters?
 - 26 letter => 5 bits
 - upper/lower case + punctuation => 7 bits (in 8) (ASCII)
 - rest of the world's languages => 16 bits (unicode)
- Logical values?
 - 0 -> False, 1 => True
- colors ?
- locations / addresses? commands?
- but N bits => only 2^N things

What if too big?

- Binary bit patterns above are simply **representatives** of numbers
- Numbers really have an infinite number of digits
 - with almost all being zero except for a few of the rightmost digits: e.g: **0000000 ... 000098 == 98**
 - Just don't normally show leading zeros
- Computers have fixed number of digits
 - In general, adding two n-bit numbers can produce an (n+1)-bit result.
 - Since computers use fixed, 32-bit integers, this is a problem.
 - If result of add (or any other arithmetic operation), cannot be represented by these rightmost hardware bits, **overflow** is said to have occurred



Overflow Example

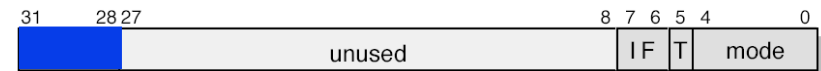
- Example (using 4-bit numbers):

+15	1111
<u>+3</u>	<u>0011</u>
+18	10010

 - But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, which is wrong.

How avoid overflow, allow it sometimes?

- Some languages detect overflow (Ada), some don't (C and JAVA)
- ARM has N, Z, C and V flags to keep track overflow
 - Refer Book!
 - Will cover details later



Comparison

- How do you tell if $X > Y$?
- See if $X - Y > 0$
- We need representation for both +ve and -ve numbers

How to Represent Negative Numbers?

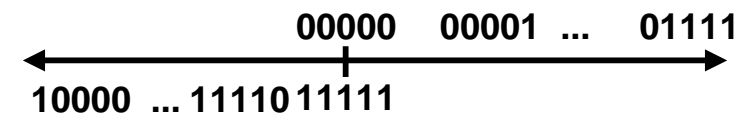
- So far, **unsigned numbers**
- Obvious solution: define leftmost bit to be sign!
 - $0 \Rightarrow +$, $1 \Rightarrow -$
 - Rest of bits can be numerical value of number
- Representation called **sign and magnitude**
- ARM uses 32-bit integers. $+1_{\text{ten}}$ would be:
0000 0000 0000 0000 0000 0000 0000 0001
- And -1_{ten} in sign and magnitude would be:
1000 0000 0000 0000 0000 0000 0000 0001

Shortcomings of sign and magnitude?

- Arithmetic circuit more complicated
 - Special steps depending whether signs are the same or not
- Also, Two zeros
 - $0x00000000 = +0_{\text{ten}}$
 - $0x80000000 = -0_{\text{ten}}$
 - What would it mean for programming?
- Sign and magnitude abandoned because another solution was better

Another try: complement the bits

- Example: $7_{10} = 00111_2$ $-7_{10} = 11000_2$
- Called **one's Complement**
- Note: positive numbers have leading 0s, negative numbers have leading 1s.



- What is -00000 ?
- How many positive numbers in N bits?
- How many negative ones?

Shortcomings of ones complement?

- Arithmetic not too hard
- Still two zeros
 - $0x00000000 = +0_{ten}$
 - $0xFFFFFFFF = -0_{ten}$
 - What would it mean for programming?
- One's complement eventually abandoned because another solution was better

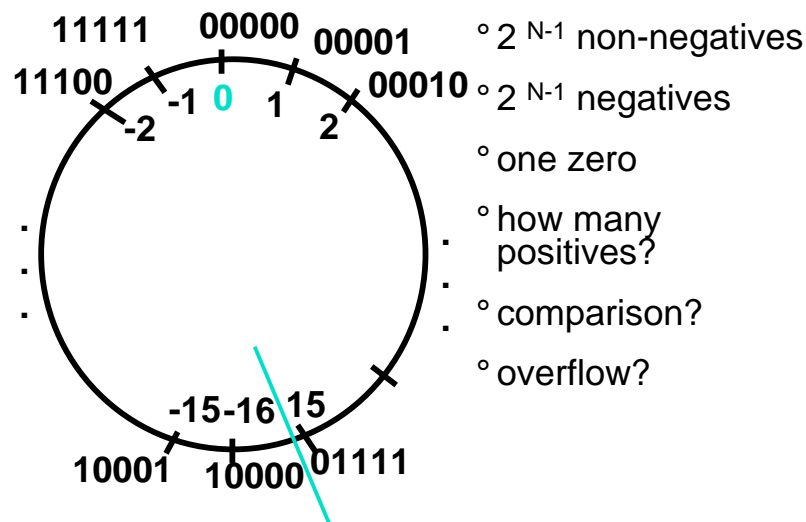
Search for Negative Number Representation

- Obvious solution didn't work, find another
 - What is result for unsigned numbers if tried to subtract large number from a small one?
 - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
- $3 - 7 = -4$

```

111
00011
-00011
-----
111100
111
000011
-000100
-----
111111
                    
```
- With no obvious better alternative, pick representation that made the hardware simple: leading 0s => positive, leading 1s => negative
 - $000000...xxx$ is ≥ 0 , $111111...xxx$ is < 0
 - This representation called **two's complement**

2's Complement Number line



Two's Complement

$0000 \dots 0000$	0000	0000	0000	0000	0000	$two =$	0_{ten}
$0000 \dots 0000$	0000	0000	0000	0000	0001	$two =$	1_{ten}
$0000 \dots 0000$	0000	0000	0000	0000	0010	$two =$	2_{ten}
$0111 \dots 1111$	1111	1111	1111	1111	1101	$two =$	$2,147,483,645_{ten}$
$0111 \dots 1111$	1111	1111	1111	1111	1110	$two =$	$2,147,483,646_{ten}$
$0111 \dots 1111$	1111	1111	1111	1111	1111	$two =$	$2,147,483,647_{ten}$
$1000 \dots 0000$	0000	0000	0000	0000	0000	$two =$	$-2,147,483,648_{ten}$
$1000 \dots 0000$	0000	0000	0000	0000	0001	$two =$	$-2,147,483,647_{ten}$
$1000 \dots 0000$	0000	0000	0000	0000	0010	$two =$	$-2,147,483,646_{ten}$
$1111 \dots 1111$	1111	1111	1111	1111	1101	$two =$	-3_{ten}
$1111 \dots 1111$	1111	1111	1111	1111	1110	$two =$	-2_{ten}
$1111 \dots 1111$	1111	1111	1111	1111	1111	$two =$	-1_{ten}

- One zero, 31st bit $\rightarrow \geq 0$ or < 0 , called **sign bit**
 - but one negative with no positive $-2,147,483,648_{ten}$

