Digital Number Systems

EE2222 Computer Interfacing and Microprocessors

Partially based on
Microprocessors and Embedded Systems by Hui Wu, UNSW
Number Systems by Dr. Paul Beame, University of Washington
Number Systems: Negative Numbers by Dr. Chung-Kuan Cheng, UC San Diego

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
- In general, given a base b, and number of digits p, the number is written as

$$d_{p-1} d_{p-2} \ldots d_2 d_1 d_0$$

$$value = d_{p-1} \times b^{p-1} + d_{m-2} \times b_{p-2} + \ldots d_2 \times b_2 + d_1 \times b + d_0$$

- The leftmost, d_{p-1}, is the **most significant digit**, d_0 , the rightmost is the **least significant digit**.
  - 1011010 = $1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 \times 1 = 64 + 16 + 8 + 2 = 90$

Digital numbers

- Digital = discrete
  - Binary codes *(example: Binary Coded Decimal)*
- Binary codes
  - Represent symbols using binary digits (bits)
- Digital computers:
  - I/O is digital
    - ASCII, decimal, etc.
  - Internal representation is binary
    - Process information in bits

Decimal Symbols	**BCD Code**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Digital numbers

- Binary numbers
  - Computers work with 0's and 1's; binary is like the alphabets of a language
- Base conversion
  - For convenience, people use other bases (like octal, decimal, hexadecimal)
  - Need to know how to convert from one to another
- Number systems
  - There are more than one way to express a number in binary.
  - So 1010 could be 10, -2, -5 or -6 and need to know which one.
- A/D and D/A conversion
  - Real world signals come in continuous/analog format
  - It is good to know how they become 0's and 1's (and vice versa)

Binary Numbers: Base 2

- Bases we will use
  - Binary: Base 2
    - 0,1
  - Octal: Base 8
    - 0,1,2,3,4,5,6,7
  - Decimal: Base 10
    - 0,1,2,3,4,5,6,7,8,9
  - Hexadecimal: Base 16
    - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

- Positional number system
  - $101_2 = 1\times2^2 + 0\times2^1 + 1\times2^0$
  - $63_8 = 6\times8^1 + 3\times8^0$
  - $A1_{16} = 10\times16^1 + 1\times16^0$
- Addition and subtraction

$$\begin{array}{r} 1011 \\ +\ 1010 \\ \hline 10101 \end{array} \qquad \begin{array}{r} 1011 \\ -\ 0110 \\ \hline 0101 \end{array}$$

Hexadecimal Numbers: Base 16

- 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

- Example (convert hex to decimal):

  B28F0DD = (Bx166) + (2x165) + (8x164) + (Fx163) + (0x162) + (Dx161) + (Dx160)

  = (11x166) + (2x165) + (8x164) + (15x163) + (0x162) + (13x161) + (13x160)

  = 187232477 decimal

- 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

Binary → hex/decimal/octal conversion

- Conversion from binary to octal/hex
  - Binary : 10011110001
  - Octal : 10 | 011 | 110 | 001=2361_8
  - Hex : 100 | 1111 | 0001=$4F1_{16}$
- Conversion from binary to decimal
  - $101_2 = 1\times2^2 + 0\times2^1 + 1\times2^0 = 5_{10}$
  - $63.4_8 = 6\times8^1 + 3\times8^0 + 4\times8^{-1} = 51.5_{10}$
  - $A1_{16} = 10\times16^1 + 1\times16^0 = 161_{10}$

Decimal → binary/octal/hex conversion

Binary

	Quotient	Remainder
56÷2=	28	0
28÷2=	14	0
14÷2=	7	0
7÷2=	3	1
3÷2=	1	1
1÷2=	0	1

Octal

	Quotient	Remainder
56÷8=	7	0
7÷8=	0	7

$56_{10}=111000_2$

$56_{10}=70_8$

- Why does this work?
  - $N=56_{10}=111000_2$
  - Q=N/2=56/2=111000/2=11100 remainder 0
- Each successive divide liberates an LSB (least significant bit)

Hex → 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.*

How Do We Tell the Difference?

- In general, append a subscript at the end of a number stating the base:
  - 10_{10} is in decimal
  - 10_2 is binary (= 2_{10})
  - 10_{16} is hex (= 16_{10})
- When dealing with AVR microcontrollers:
  - Hex numbers are preceded with "$" or "0x"
    - $10 == 0x10 == 10_{16} == 16_{10}
  - Binary numbers are preceded with "0b"
  - Octal numbers are preceded with "0" (zero)
  - Everything else by default is Decimal

Inside the Computer

- To a computer, numbers are always in binary; all that matters is how they are printed out: binary, decimal, hex, etc.
- As a result, it doesn't matter what base a number in C is in...
  - 32_{10} == 0x20 == 100000_2
- Only the value of the number matters.

Bits Can Represent Everything

- 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?

- 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
  - Adding two n-bit numbers may produce an (n+1)-bit result.
  - Since registers’ length (8 bits on AVR) is fixed, this is a problem.
  - If the result of add (or any other arithmetic operation), cannot be represented by a register, overflow is said to have occurred

An Overflow Example

- Example (using 4-bit numbers):

+15	1111
+3	0011
+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)
- Some processors have overflow flags
  - AVR has N, Z, C and V flags to keep track of overflow

How to Represent Negative Numbers?

- So far, unsigned numbers
- Historically: 3 approaches
  - Sign-and-magnitude
  - Ones-complement
  - Twos-complement
- For all 3, the most-significant bit (MSB) is the sign digit
  - 0 ≡ positive
  - 1 ≡ negative
- Twos-complement is the important one
  - Simplifies arithmetic
  - Used almost universally

Sign-and-magnitude

- Obvious solution: define leftmost bit to be sign!
- The most-significant bit (MSB) is the sign digit
  - 0 ≡ positive; 1 ≡ negative
- The remaining bits are the number's magnitude
- $+1_{10}$ would be: 0000 0001
- -1_{10} in sign and magnitude would be: 1000 0001

Add		Subtract			Compare and subtract		
4	0100	4	0100	0100	− 4	1100	1100
+ 3	+ 0011	− 3	+ 1011	− 0011	+ 3	+ 0011	− 0011
= 7	= 0111	= 1	≠ 1111	= 0001	− 1	≠ 1111	= 1001

Shortcomings of sign-and-magnitude

- Problem 1:
  - Two representations for zero
  - 0 = 0000 and also -0 = 1000
- Problem 2:
  - Arithmetic circuit is more complicated
  - Special steps depending whether signs are the same or not
- Sign and magnitude abandoned because another solution was better

Ones-complement

- Invert the 0s & 1s of a equivalent binary number provides the 1s complement.
- For positive integer x, represent -x:
  - Formula: $2^n - 1 - x$
  - i.e. n=4, $2^4 - 1 - x = 15 - x$
  - In binary: $(1\ 1\ 1\ 1) - (b_3\ b_2\ b_1\ b_0)$
  - Just flip all the bits.

Ones-complement

- Examples:
  - $7_{10} = 00000111_2$
  - $-7_{10} = 11111000_2$
- Questions:
  - What is -00000000_2?
  - How many positive numbers in N bits?
  - How many negative numbers in N bits?

Ones-complement

- Negative number: Bitwise complement positive number
  - 0111 ≡ 7_{10}
  - 1000 ≡ $–7_{10}$
- Solves the arithmetic problem

Add		Invert, add, add carry		Invert and add	
4	0100	4	0100	– 4	1011
+ 3	+ 0011	– 3	+ 1100	+ 3	+ 0011
= 7	= 0111	= 1	1 0000	– 1	1110
		add carry:	+1		
			= 0001		

- Remaining problem: Two representations for zero
  - 0 = 0000 and also –0 = 1111

Why ones-complement works?

- The ones-complement of an 8-bit positive y is 11111111_2 – y
- What is 11111111_2 ?
  - 1 less than 1 $00000000_2 \equiv 2^8 \equiv 256_{10}$
  - So in ones-complement –y is represented by $(2^8 - 1) - y$
- Adding representations of x and –y where x, y are positive: we get $(2^8 - 1) + x - y$
  - If $x < y$ then $x - y < 0$ there is no carry and get –ve number
    - Just add the representations if no carry
  - If $x > y$ then $x - y > 0$ there is a carry and get +ve number
    - Need to add 1 and ignore the 2^8, i.e. “add the carry”
  - If $x = y$ then answer should be 0, get $2^8 - 1 = 11111111_2$

Arithmetic Operations: 1’s Complement

Input: two positive integers x & y,

1. We represent the operands in one’s complement.
2. We sum up the two operands.
3. We delete 2^n-1 if there is carry out at left.
4. The result is the solution in one’s complement.

Arithmetic	1’s complement
x + y	x + y
x - y	$x + (2^n - 1 - y) = 2^n - 1 + (x - y)$
-x + y	$(2^n - 1 - x) + y = 2^n - 1 + (-x + y)$
-x - y	$(2^n - 1 - x) + (2^n - 1 - y) = 2^n - 1 + (2^n - 1 - x - y)$

Arithmetic Operations: Example: 4 – 3 = 1

$4_{10} = 0100_2$

$3_{10} = 0011_2$ $-3_{10} \rightarrow 1100_2$ in one's complement

0100 (4 in decimal)
+ 1100 (12 in decimal or 15-3)
1,0000 (16 in decimal or 15+1)
0001(after deleting 2^n-1)

We discard the extra 1 at the left which is 2^n and add one at the first bit.

Arithmetic Operations: Example: -4 +3 = -1

$4_{10} = 0100_2$ -4_{10} → Using one's comp.→ 1011_2
(Invert bits)

$3_{10} = 0011_2$

1011 (11 in decimal or 15-4)
+ 0011 (3 in decimal)
1110 (14 in decimal or 15-1)

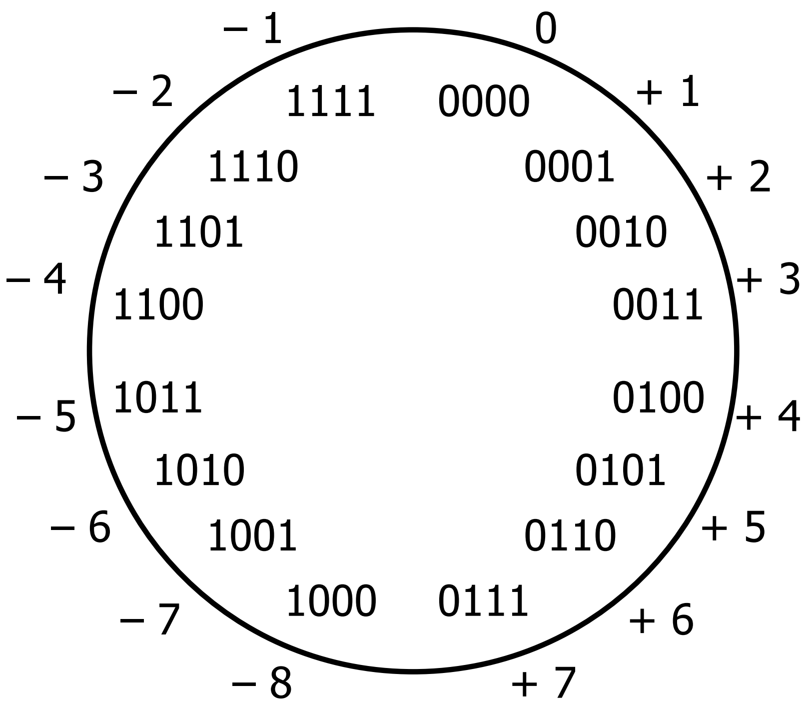
If the left-most bit is 1, it means that we have a negative number.

Two’s complement

- Add 1 to the one's complement provides the two's complement.
- For positive integer x, represent -x:
  - Formula: $2^n – x$
  - i.e. n=4, $2^4 – x = 16 – x$
  - In binary: (1 0 0 0 0) – (0 b_3 b_2 b_1 b_0)
  - Just flip all the bits and add 1.

Twos-complement

- Negative number: Bitwise complement plus one
  - 0111 ≡ 7_{10}
  - 1001 ≡ -7_{10}
- Number wheel
  - Only one zero!
  - MSB is the sign digit
  - 0 ≡ positive
  - 1 ≡ negative



Twos-complement

- Complementing a complement the original number
- Arithmetic is easy
  - Subtraction = negation and addition
  - Easy to implement in hardware

Add		Invert and add		Invert and add	
4	0100	4	0100	– 4	1100
+ 3	+ 0011	– 3	+ 1101	+ 3	+ 0011
= 7	= 0111	= 1	1 0001	– 1	1111
		drop carry	= 0001		

Why twos-complement works better

- Recall:
  - The ones-complement of a b-bit positive y is $(2^b-1) - y$
- Adding 1 to get the twos-complement represents $-y$ by $2^b - y$
  - So $-y$ and $2^b - y$ are equal mod 2^b
    (leave the same remainder when divided by 2^b)
  - Ignoring carries is equivalent to doing arithmetic mod 2^b
- Adding representations of x and $-y$ yields $2^b + x - y$
  - If there is a carry then that means $x \geq y$ and dropping the carry yields $x-y$
  - If there is no carry then $x < y$ and then we can think of it as $2^b - (y-x)$

Arithmetic Operations: 2's Complement

Input: two positive integers x & y,

1. We represent the operands in two's complement.
2. We sum up the two operands and ignore bit n.
3. The result is the solution in two's complement.

Arithmetic	2's complement
$x + y$	$x + y$
$x - y$	$x + (2^n - y) = 2^n+(x-y)$
$-x + y$	$(2^n - x) + y =2^n+(-x+y)$
$-x - y$	$(2^n - x) + (2^n - y) = 2^n+2^n-x-y$

Arithmetic Operations: Example: 4 – 3 = 1

$4_{10} = 0100_2$

$3_{10} = 0011_2$ $-3_{10} \rightarrow 1101_2$

$$\begin{array}{r} 0100 \\ +\ 1101 \\ \hline 10001 \end{array}$$

10001 → 1 (after discarding extra bit)

We discard the extra 1 at the left which is 2^n from 2's complement of -3. Note that bit b_{n-1} is 0. Thus, the result is positive.

Arithmetic Operations: Example: -4 +3 = -1

$4_{10} = 0100_2$ $-4_{10} \rightarrow$ Using two's comp. $\rightarrow$ $1011 + 1 = 1100_2$

(Invert bits)

$3_{10} = 0011_2$

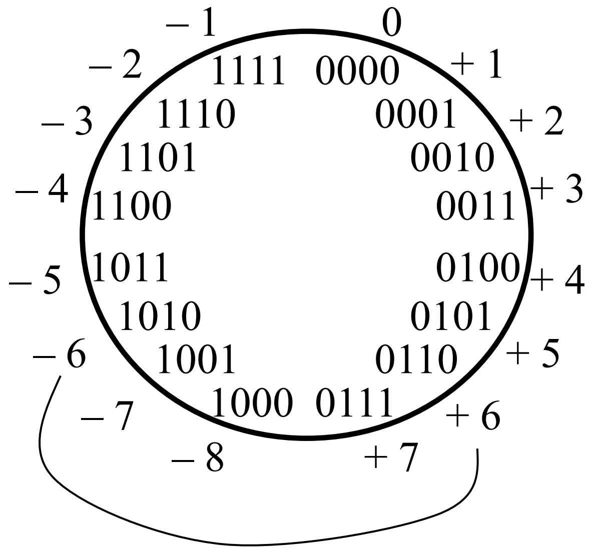
```
  1100
+ 0011
  1111
```

1111 → Using two's comp. → 0000 + 1 = 1, so our answer is -1

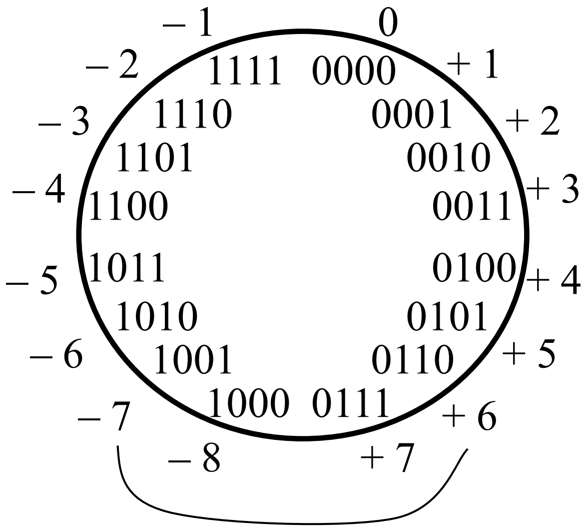
If left-most bit is 1, it means that we have a negative number.

Twos-complement overflow

- The rules for detecting overflow in a two's complement sum are simple:
  - Summing two positive numbers can give negative result
  - Summing two negative numbers can give a positive result



$6 + 4 \Rightarrow -6$



$-7 - 3 \Rightarrow +6$

Representing fractional numbers

- To represent fractional numbers, we simply extend the positional system to include digits corresponding to negative powers.
- A number which includes a *q* bit fractional part, for a total of *p*+*q* bits:

$$d_{p-1}d_{p-2}\dots d_2 d_1 d_0.d_{-1}d_{-2}\dots d_{-q+1}d_{-q}$$

- and its value is:

$$value = d_{p-1}\times b^{p-1} + d_{m-2}\times b_{p-2} + \dots d_2\times b_2 + d_1\times b + d_0 + d_{-1}\times b_{-1} + d_{-2}\times b_{-2} + \dots + d_{-q+1}\times b_{-q+1} + d_{-q}\times b_{-q}$$

- Two's complement for fractional numbers:
  - $1.6875_{10} = 01.1011_2$
  - $-1.6875_{10} = 10.0101_2$

Sign extension

- increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value
- done by appending digits to the most significant side of the number
- Example:
  - Write +6 and –6 as 2’s complement
    - 0110 and 1010
  - Sign extend to 8-bit bytes
    - 00000110 and 11111010

Still…

- Can't infer a representation from a number
  - 11001 is 25 (unsigned)
  - 11001 is –9 (sign magnitude)
  - 11001 is –6 (ones complement)
  - 11001 is –7 (twos complement)

Summary

- Positional notation
- Binary/Octal/Decimal/Hexadecimal numbers
- Negative numbers
  - Sign-and-magnitude
  - Ones-complement
  - Twos-complement
- Arithmetic operations
- Representing fractional numbers