Encoding of Counting Numbers (0, 1, 2, 3¼)

Based Number Systems:

- **base**: set of atomic **symbols** + **interpretation** of each symbol as a counting "value" (quantity)
- **number**: string of symbols (digits)

  PLUS interpretation rule

  - symbols (digits) are numbered right to left, start at 0
    - e.g. 4-digit number: $d_3d_2d_1d_0$
  - "value" of a symbol: depends on **value** of the individual symbol and **position** in string
    - value of symbol_i = (value of symbol_i) $\times$ base^i

"value" of number is sum of values of symbols

Base-10 Example: (done this before, but …)

- symbols = { 0, 1, 2, 3, … , 9} – usual meaning
- 4-digit example: $d_3d_2d_1d_0 = 1436$

$$= d_3 \times 10^3 + d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$$

$$= 1 \times 10^3 + 4 \times 10^2 + 3 \times 10^1 + 6 \times 10^0$$

Binary Number System:

- set of symbols = **{ 0 , 1 }** – usual meaning
- 4-digit number example: 1011_2

$$= \mathbf{1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0}$$

$$= 8 + 2 + 1 = 11$$

Examples

Write each of the following binary numbers as decimal numbers

11111 101010 1001011001

Write each of the following decimal numbers as binary numbers

7 0 10 15 33

What about the **range** of representation?

- **n-bit values** can represent (at most) $\mathbf{2^n}$ different counting numbers
- if start representation at **0**, then largest value represented is $\mathbf{2^n - 1}$
- recall 4-bit example: ($2^4 = 16$)
  - range = 0 . . 15

Some Observations:

- binary representation often requires lots of bits
  - example: $601_{10} = 1001011001_2$ (10 bits!)
- converting binary ←→ decimal is not trivial

binary can be awkward and error prone for use by people!

Hexadecimal (Base-16) Number System:

- 16 symbols = { **0 . . 9, A, B, C, D, E, F**}
- interpret: 0 . . 9 – usual meaning

A	10	C	12	E	14
B	11	D	13	F	15

Why use hexadecimal (**hex**) notation?

- **shorthand** for binary notation
- **converting** binary ←→ hexadecimal is easy

  4 binary digits required to encode the value of one hexadecimal digit

Converting Binary → Hex

- group bits in 4's starting from **least significant**
- **replace** each group by corresponding hex digit

Examples:

16-bit: 1000101001101110_2 (= $35{,}468_{10}$)

group in 4's → 1000 1010 0110 1110

replace with hex → 8 **A** 6 E $_{16}$

10-bit: 1001011001_2 (= 601_{10})

group in 4's → 0010 0101 1001

replace with hex → 2 **5** 9 $_{16}$

Converting Hex → Binary

replace each hex digit by 4-digit binary rep.

(already know this from 267?)

MEMORIZE THESE!

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

Examples

Write each of the following binary numbers as hexadecimal numbers

1001011001 101010 111111

Write each of the following decimal number as hexadecimal numbers

7 0 10 15 33 98 167

Write each of the following hexadecimal number as decimal numbers

C 1E 21 148

Binary Math (intro)

Addition and Subtraction proceeds the same as for decimal numbers, except that you carry and borrow 1 (not 10)

111 + 10 =

101 – 10 =

1010 + 111 =

1001 – 111 =

binary addition:
$0 + 0 = 0_2$
$0 + 1 = 1_2$
$1 + 1 = 10_2$
(0 carry 1)

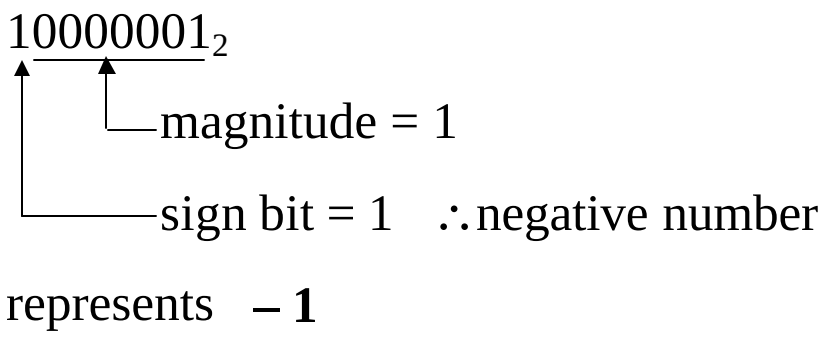
binary substraction:
$1 - 0 = 1_2$
$0 - 0 = 0_2$
$1 - 1 = 0_2$
$0 - 1 = (1)\ 1_2$
(1 borrow 1)

Encoding of Integers

- positive and negative values
- must encode **sign & magnitude**

1) <u>N-Bit Signed-Magnitude Encoding</u>:

- use **most significant bit** to encode **sign**
  - 0 = positive, 1 = negative
- use remaining **n – 1 bits** to encode **magnitude** using binary number system (as for counting #s)
- example: 8-bit signed-magnitude



represents **– 1**

Problems with Signed-Magnitude Encoding:

- **two representations for 0** ☹

  positive 0 (sign bit = 0)

  negative 0 (sign bit = 1)

  [Is 0 positive or negative?]

- performing arithmetic operations with signed-magnitude values can be awkward ☹
- signed-magnitude not often used in practice

2) N-Bit 2's Complement Encoding: ☺

- complement of one bit: $\overline{b} = 1 - b$

  complement of 0 = 1 complement of 1 = 0

- complement of an **n-bit** value: complement **each bit**
- 2's complement of an n-bit value:

  complement **each** bit, and then **add 1**

ignore any carry out of most significant bit

Examples:

original value =	100111010010	
complement each bit	011000101101	
add 1	+ 1	
2's complement =	**011000101110**	

binary addition:
$0 + 0 = 0_2$
$0 + 1 = 1_2$
$1 + 1 = 10_2$
(0 carry 1)

To encode an *integer* in an n-bit value:

- half of binary values for **negative** values (i.e. 2^{n-1} values), rest for positive (and 0)
- encode positive values (and 0) the same way as **counting** numbers
  - range $0 \ldots 2^{n-1} - 1$
  - uses all values that start with 0

- use 2's complement as **negation operation**!
  - to find encoding of **– x**

    form **2's complement of + x**

8-bit example:

$+1_{10}$ ⟶ 0000 0001

complement	1111 1110	
add 1	+ 1	
-1_{10} ⟶	1111 1111	

2's complement operation

Negating a negated number should give the original number **– (– x) = x**

Does this work for negating (2's complementing) negative values ?

-1_{10} ⟶ 1111 1111

complement	0000 0000	
add 1	+ 1	

2's complement operation

$+1_{10}$ ⟶ 0000 0001 works! ☺

How many representations for 0?

0_{10} ⟶ 0000 0000

complement	1111 1111	
add 1	+ 1	

ignore carry out of most significant bit

0_{10} ⟶ 0000 0000 ☺

- range for negative values

$$-2^{n-1} \ldots -1$$

all negative value encodings start with 1

Consider 8-bit 2's complement encodings:

- $2^8 = 256$, $2^7 = 128$
- range: **-128 . . $+127$**

  (-2^{n-1}) $(2^{n-1} - 1)$

$+127_{10} \longrightarrow$ 0111 1111

complement	1000 0000	
add 1	+ 1	
		2's complement operation

$-127_{10} \longrightarrow$ 1000 0001

How to get encoding for – 128 ?

– 127 – 1 = – 128

binary subtraction:

$0 - 0 = 0_2$

$1 - 0 = 1_2$

$1 - 1 = 0_2$

$0 - 1 =$ 1 borrow 1

$-127_{10} \longrightarrow$ 1000 0001

subtract 1 – 1

$-128_{10} \longrightarrow$ 1000 0000

What if we negate – 128 ? **[special case !!]**

$-128_{10} \longrightarrow$ 1000 0000

complement 0111 1111

add 1 + 1

$-128_{10} \longrightarrow$ 1000 0000 ??? huh ???

Is this expected? Why don't we get +128 ???

(What is $1000\ 0000_2$ as a counting number?)

Examples

Write each of the following decimal numbers

- 8-bit signed integers
- 8-bit two’s complement integers

65 -65 42 -42

Encoding of Characters

- want to represent “displayable” characters:
  - **characters**: { A, B , . . . , Y, Z }
    - $26 \times 2 = 52$ (upper and lower case)
  - **decimal (10) digits**: {0, 1, . . . , 8, 9 }
  - **punctuation**: ! ″ ′ , . – ? / : ;
  - **math symbols**: + * = (– and / above)
  - **brackets**: ( ) [ ] { } < >
  - **others**: @ # $ % ^ & \ | ~
  - **blank space**: “ ”
  - 90^{+} symbols (??)
- various encoding schemes have been used
- ASCII encoding – international standard

American Standard Code for Information Interchange

7-Bit ASCII Encoding

- 7 bits to encode each character (128 codes)
- often extended to 8-bit (byte) values by making ***m***ost ***s***ignificant ***b***it (msb) = 0

[in following: all codes are given as hex values]

00 – 1F **non-displayable control char's**

- 00 NULL
- 07 BELL
- 08 backspace
- 09 tab
- 0A line feed
- 0C form feed
- 0D carriage return

others – often serve special purposes in communication applications

30 – 39 decimal digit char's

- 30 "0"
- 39 "9"

character "0" ≠ number 0

41 – 5A Upper Case Letter char's

- 41 "A"
- 5A "Z"

61 – 7A Lower Case Letter char's

- 61 "a"
- 7A "z"

20 blank space

- example: “3006 is FUN!”

  encoding: 33 30 30 36 20 69 73 20 42 4F 52 49 4E 47 21

  shorthand 00110000 01110011

  for binary!

Other Character Encoding Schemes:

- IBM standardized an 8-bit scheme (256 char’s) as defacto standard (PC’s !)
  - see inside back cover of text – overlaps with 7-bit ASCII for displayable char’s
- Java: unicode – 16-bit scheme (65,536 char’s)
  - multi-lingual character sets

Important Concept !!!!!!

- **fixed-width binary values** are used to represent information in computers
- the computer works with the **binary representations** (NOT the information!!)
- the **same** binary value can be used to represent **different** information !

8-bit example: $1111\ 0000_2$

unsigned: 240_{10}

signed: -16_{10}

8-bit ASCII: “ ≡ ”

other?

Programmers Must Be Aware!