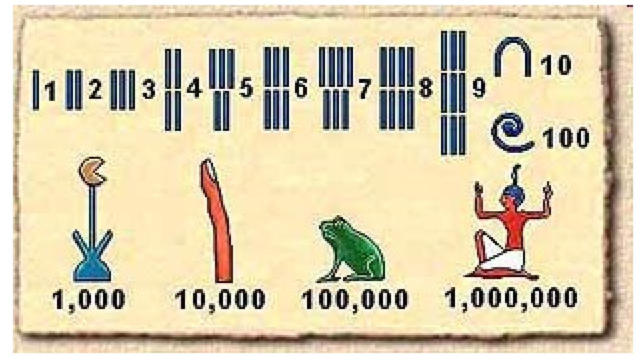
CS 211: Computer Architecture

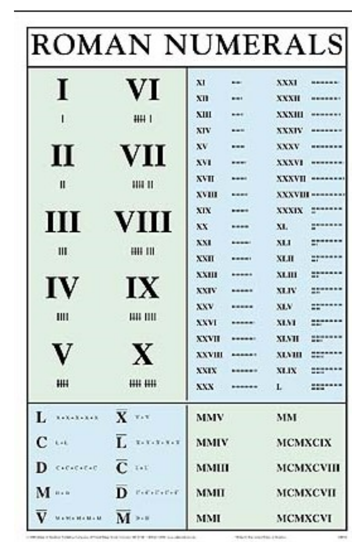
Data representation

What Do Computers Do?

- Manipulate stored information
- Information is data
  - How is it represented?
- Basic information: numbers
- Human beings have represented numbers throughout history
  - Egyptian number system
  - Roman numeral
- Typically decimal
  - Natural for humans



Discoveregypt.com



Number System

Comprises of

- Set of numbers or elements
- Operations on them
- Rules that define properties of operations

Need to assign value to numbers

Let us take decimal

- Base 10
- Numbers are written as $d_n \ldots d_2 d_1 d_0$
- Each digit is in [0-9]
- Value of a number is interpreted as $\sum_{i=0}^{n} d_i \times 10^i$

Binary Numbers

Base 2 = each digit is 0 or 1

Numbers are written as $d_n \ldots d_2 d_1 d_0$

Value of number is computed as $\sum_{i=0}^{n} d_i \times 2^i$

Binary representation is used in computers

- Easy to represent by switches (on/off)
- Manipulation by digital logic in hardware

Written form is long and inconvenient to deal with

Hexadecimal Numbers

Base 16

Each digit can be one of 16 different values

- Symbols = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F}

First 10 symbols (0 through 9) are same as decimal

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

Numbers are written as $d_n \ldots d_2 d_1 d_0$

Value = $\sum_{i=0}^{n} d_i \times 16^i$

Octal Numbers

Base 8 $\Rightarrow$ each digit is in [0-7]

Numbers are written as $d_n \ldots d_2 d_1 d_0$

Value of number is computed as $\sum_{i=0}^{n} d_i \times 8^i$

Converting Hex to Binary

Each hexadecimal digit can be represented by 4 binary digits

- Why?

0x2A8C (hex) = 0b0010101010001100 (binary)

- 0xC = 12 x 16^0 = 8 + 4 = (1 x 2^3) + (1 x 2^2) = 0b1100
- 0x80 = 8 x 16^1 = 2^3 x 2^4 = 2^7 = 0b 10000000
- And so on …

So, to convert hex to binary, just convert each digit and concatenate

What about octal to binary?

Converting Binary to Hex

- Do the reverse
  - Group each set of 4 digits and change to corresponding digit in hex
  - Go from right to left
- Example 1011011110011100
  - 0b1011011110011100 = 0xB79C
- What about binary to octal?

Decimal to Binary

What's the largest p, q, r … such that

- $n = 2^p + r_1$, where $r_1 < 2^p$
- $n - 2^p = 2^q + r_2$, where $r_2 < 2^q$
- $n – (2^p + 2^q)= 2^r + r_3$, where $r_3 < 2^r$
- …

The above means that

- $n = (1 \times 2^p) + (1 \times 2^q) + (1 \times 2^r) + \ldots + r_m$, where $r_m = n \% 2$
- Can you see why this now allows n to be easily written in binary form?

Example: convert 21 to binary

- $21 = 2^4 + 5$, $5 = 2^2 + 1 \Rightarrow 21 = 0b10101$

Decimal to Binary and Back

How to do the conversion algorithmically?

What about binary to decimal?

What about decimal to hex? Hex to decimal?

Decimal to octal? Octal to decimal?

Hex to octal? Octal to Hex?

Decimal and Binary fractions

In decimal, digits to the right of radix point have value $1/10^i$ for each digit in the i^{th} place

- 0.25 is 2/10 + 5/100

Similarly, in binary, digits to the right of radix point have value $1/2^i$ for each i^{th} place

- Just the base is different

8.625 is 1000.101

- .625 = 6/10 + 2/100 + 5/1000 = 1/2 + 1/8

How to convert?

Decimal to Binary Example

```
Algorithm

Number = decimalFraction
while (number > 0) {
    number = number*2
    if (number >=1) {
        Output 1;
        number = number-1
    }
    else {
        Output 0
    }
}
```

Why does it work?

Example: 0.625 to binary

- ANS: 0.101
  - 0.625*2 = 1.25
  - output 1
  - 0.25*2 =0.5
  - output 0
  - 0.5*2 = 1
  - output 1
  - Exit

Representing integers

How do we represent negative numbers in computers?

- Use a bit … after all, that’s how we store information, right?

Signed Magnitude:

S	Magnitude

- 0100 = 4, 1100 = -4
- 0011 = 3, 1011 = -3

What is 1000?

- Have two zeros +0 (0000) and -0 (1000)
- As we shall see, inconvenient for arithmetic computations

Signed magnitude

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

111	110	101	100	000	001	010	011
-3	-2	-1	-0	0	1	2	3

Signed magnitude

- What is -1 + 1?
- 101 + 1 = ?
- 100?

111	110	101	100	000	001	010	011
-3	-2	-1	-0	0	1	2	3

One's Complement

Represent negative numbers by complementing positive numbers

Still have two zeros but arithmetic computation becomes easier

000	001	010	011	100	101	110	111
0	1	2	3	-3	-2	-1	-0

One’s complement

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

100	101	110	111	000	001	010	011
-3	-2	-1	-0	0	1	2	3

One’s complement

- What is -1 + 1?
- 110 + 1 = 111

100	101	110	111	000	001	010	011
-3	-2	-1	-0	0	1	2	3

Two's Complement

000	001	010	011	100	101	110	11 1
0	1	2	3	-4	-3	-2	-1

One's complement plus one

- Most significant bit still gives the "sign"
- Trick: copy all '0' bits from LSB till first '1' bit. Copy '1' bit, then flip all remaining bits till MSB.

Advantages:

- Only 1 zero
- Most convenient for arithmetic computations

Used in almost all computers today

Two's complement

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7

100	101	110	111	000	001	010	011
-4	-3	-2	-1	0	1	2	3

Two's complement

- What is -1 + 1?
- 111 + 1 = (1) 000

100	101	110	111	000	001	010	011
-4	-3	-2	-1	0	1	2	3

Two’s complement

- To go from 3 to -3, flip all bits and add 1
- (or change rightmost 0 to 1, and all 1’s to its right to 0’s)
- How to go from -3 to 3?

100	101	110	111	000	001	010	011
-4	-3	-2	-1	0	1	2	3

Numerical Value of Two's Complement

Given a two's complement number of length n, written as $d_{n-1} \ldots d_1 d_0$

- It's value is interpreted as $-d_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} d_i 2^i$

The range of values is then $[-2^{n-1}, 2^{n-1} - 1]$

- More negative numbers than positive (if we do not count 0)
- 101 = ?
- 0101 != 101

ASCII

A character is stored as 1 byte according to the ASCII standard

Originally used only 128 values (7 bits)

- One bit could be used for error detection (will discuss later)

Subsequently extended to use all 256 values

ASCII table

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Unicode and UTF-8

What about characters for other languages?

Unicode is a standard that defines more than 107,000 characters across 90 scripts (and more …)

Unicode can be implemented by different character encodings

Most common: UTF-8

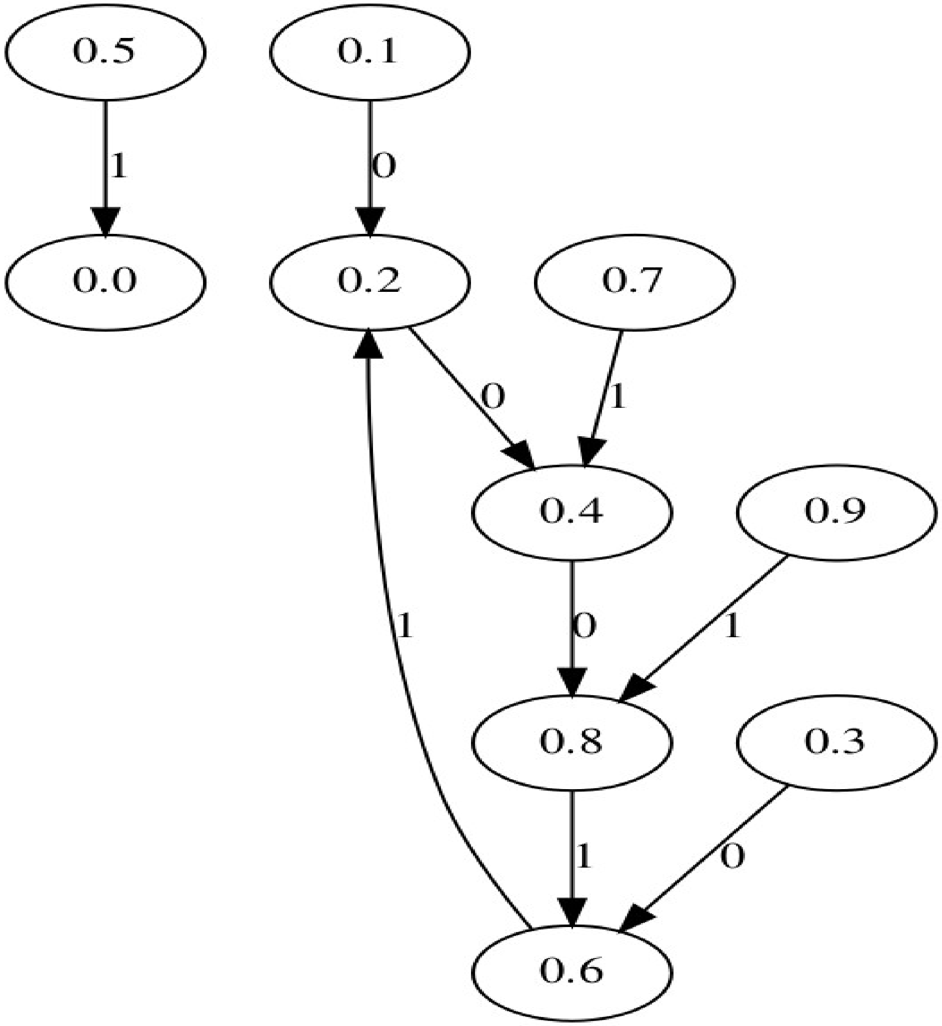
- Variable length encoding of Unicode: 1-4 bytes for each character
- 1-byte form is reserved for ASCII for backward compatibility

Floating point

- Recall that 0.1 can’t be represented exactly in binary:
- 0.1 * 2 = 0.2 (0)
- 0.2 * 2 = 0.4 (0)
- 0.4 * 2 = 0.8 (0)
- 0.8 * 2 = 1.6 (1)
- 0.6 * 2 = 1.2 (1)
- 0.2 * 2 = 0.4 (0) …

- What about 0.2, 0.3, 0.4, …?

Floating point



Only 0.0 and 0.5 are exact! (for single digit fractional values)

But others multi-digit fractionals may also be, e.g., 0.125.

Floating point comparison

- float f1 = …;
- float f2 = …;
- if (f1 == f2) … // dangerous!

- We actually want to ask if they're really similar:

  $| f1 - f2 | < \varepsilon$

- if (fabs(f1 – f2) < 1e-6) …

Floating point

Integers typically written in ordinary decimal form

- E.g., 1, 10, 100, 1000, 10000, 12456897, etc.

But, can also be written in scientific notation

- E.g., $1x10^4$, $1.2456897x10^7$

What about binary numbers?

- Works the same way
- $0b100 = 0b1x2^2$

Scientific notation gives a natural way for thinking about floating point numbers

- $0.25 = 2.5x10^{-1} = 0b1x2^{-2}$

How to represent in computers?

IEEE floating point standard

Most computers follow IEEE 754 standard

Single precision (32 bits)

Double precision (64 bits)

Extended precision (80 bits)

S	Exponent	Fraction

Floating point in C

32 bits single precision (type float)

- 1 bit for sign, 8 bits for exponent, 23 bits for mantissa
  - Sign bit: 1 = negative numbers, 0 = positive numbers
  - Exponent is power of 2
- Have 2 zero's
- Range is approximately -10^{38} to 10^{38}

64 bits double precision (type double)

- 1 bit for sign, 11 bits for exponent, 52 bits for mantissa
- Majority of new bits for mantissa → higher precision
- Range is -10^{308} to $+10^{308}$

Numerical Values

Three different cases:

- Normalized values
  - exponent field ≠ 0 and exponent field ≠ 2^k-1 (all 1's)
  - exponent = binary value – Bias
    - » Bias = 2^{k-1}-1 (e.g., 127 for float)
  - mantissa = 1.(mantissa field)
  - Ex: (sign: 0, exp: 1, mantissa: 1) would give 0b1.1x2^{-126}
- Denormalized values
  - exponent field = 0
  - exponent = 1 – Bias (e.g., -126 for float)
  - Mantissa = mantissa field (no leading 1)
  - Ex: (sign: 0, exp: 0, mantissa: 10) would give 0b10x2^{-126}
- Special values: represent +∞, -∞, and NaN

Decimal to IEEE Floating Point

5.625

In binary

101.101 = 1.01101 x 2^2

Exponent field has value 2

- add 127 to get 129

Exponent is 10000001

Mantissa is 01101

Sign bit is 0

0 10000001 01101000000000000000000

One more example

Convert 12.375 to floating point representation

Binary is 1100.011

1.100011×2^3

Exponent = 127 + 3 = 130 = 0b10000010

Mantissa = 100011

Sign = 0

Extended precision

80 bits used to represent a real number

1 sign bit, 15 bit exponent, 64 bit mantissa

20 decimal digits of accuracy

10^{-4932} to 10^{4932}

Not supported in C