Number systems💚

Signpost

- Computers operate on binary values
- A computer has two modes (on or off)
- Binary digits represents off and on (zero means off and 1 is on)
- When we say zero the transistor is off and vice-versa

Bit - single value
Bit are bundled together into 8-bit collations called bytes

Number representations

Notation: Nr for number N using radix r
Radix refers to the base - the number of possible symbols for each digit

Decimal Codes

Radix = 10 .

- Possible values for digits 0-9

Binary Codes

Radix = 2
Possible values for digits: 0 and 1

- n-bit **binary number** can represent numbers from 010 to (2n-1)10
- Largest **8-bit** (unsigned) number: 111111112 = 25510(not positive or negative)largest number you can have is all ones

Binary Addition
Adding binary numbers
1+0 = 0+1 = 1
0 +0 = 0
1 + 1 = 0 carry 1
Possibility of an overflow

We have only 8 bits to store answer… so it's zero!

Signed numbers

Representing positive and negative numbers without a '-' sign

- We don’t just store positive umbers we need to store negative numbers as well
- Can use left most bit to code sign and magnitude(know whether positive or negative

This is wasteful, we need a better system for working with signed numbers
0 represents positive and 1 represents negative

Negative Numbers

- To get the negative representation for a positive number
- Sign/magnitude:invert just the sign bit(if its - its a 1 and 0 is positive)
- 1’s complement: invert all bits(if its a negative number we invert all the bits and store as negative number)
- 2’s complement:invert all bits and add one to the results

<u>1’s complement</u>

Negative numbers obtained by flipping signs
Positive numbers don’t change(we just use the bits to store it but foe a negative we have to flip it)
Addition is easier

<u>2’s complement</u>
Now when we add, discard carry

- 1’s complement still has two zeros
- An alternative to access is the 2’s complement
- Complement then add 1

- It doesn’t mean that if it start with a zero its a positive number can a a negative number in a 2’s form(positive)
- What do we do with the carry?,we just discard it

Floating-point Numbers

- Fixed point numbers have very limited range(determine by bit length)
- Floating point(scientific notation)

eg9.76*10^-14
Contists of two parts:mantissa & exponent
*Mantissa: the number multiplying the base(**9.76)***
Exponent:the power

*Significand**(76)***

Floating Point range

- [x] *Range of numbers is very large, but accuracy is limited by significant*

***Normalising -** converting a number to the scientific notation*

- [ ] *It only stores 8 bits, so whatever is in front of the 8 will be left out (not accurate) - truncation error*

How do we normalise a floating point

- [ ] *Shift point until only one non-zero digit is left*
- [ ] *If left shift - add 1*
- [ ] *Right shift - subtract 1*

Floating Point Formats

IEEE 745 formart
***single(32-bit)** and **double(64-bit)** formats;mjch needed standard*
Also extended precision- 80 bits(long double)

When storing this **single precision number** represented internally as

- [ ] *Sign bit(0 positive and 1 is neg)*
- [ ] *Followed by exponent(8-bits)*
- [ ] *Then the fraction part of the normalised number*

Don't store the full mantissa but the significant part of the number

*The leading 1 is implied **not sorted***

Double precision
Has 11 bit exponent and 52-bit significant

Floating point Exponents

The exponent is **'biased'**: no explicit negative number
Single precision:127, Double precision 1023

So for single precision:

- [ ] *Stored value of 255 indicated exponent of 255-127 = 128, and if value stored is 0 subtract - 127 = -127(cant' be symmetric, because of zero)*
- [ ]

More formats and characters

IEEE 754 has special codes for zero, error conditions(0/0 etc)

Zero: exponent and significant are zero

Infinity: exp = 1111…1111, significand = 0(if exponent is all ones we cant represent that particular number)

NaN (not a number):0/0'exponent = 1111…1111, significand != 0(ERROR)

<u>Concept of underflow and overflow</u>
Too large or too small of a value that you want to store

Expressible numbers : numbers that you CAN represent

- Negative Overflow(negative numbers slower than your lower limit), Expressible negative numbers, negative underflow(lower than your lower limit)

- Positive underflow , Expressible Positive Numbers, Positive Overflow(greater than your upper limit)

Underflow is a less serious problem because it just denotes a loss of precision

<u>FP Operations and Errors</u>

Trick - always normalise and make sure that exponents are the same

- **Addition/subtraction:**normalise, match to larger exponent then add, normalise

<u>Error conditions:</u>
Exponent overflow(Exponent bigger than max permissible size, may be set to infinity)

Exponent underflow(negative exponent smaller than minimum size; may be set to zero)

Significant underflow(alignment may cause loss of significant digits)

Significant overflow(alignment may cause carry overflow; realign significands)

Character represented using ‘character set’.

Unicode is 16 bits
Using ASCII or unicode we can store characters on our computer systems

Boolean Algebra

- **Modern Computer devices are digital rather than analog**
- **Use two discrete states to represent all entities:0 and 1**
- **Call these two logical states True and False**

The computer system store information and does computation(calculations, additions etc)

In a computer chip we have transistors and they aren’t isolated

These transistors have two states ON and OFF (1 and 0)
All operations are on these values

***** George Boole formalised such a logic algebra ‘ Boolean algebra’**

Three basic logic operations:AND, OR, NOT

A.B -> A and B
AB -> A and B
A + B -> A or B
not A - > opposite

False = 0

Truth tables

- Truth Tables show the value each operator(or combinations)
- Not is a unary operator: inverts truth value

NAND is False only if both args are true
[NOT (A AND B)]

NOR is TRUE only if both args are False
[NOT (A OR B)]

XOR is TRUE if either input is True, but not BOTH

Logic gates

- **The Boolean operators have symbolic representations: 'logic gates'**

These are the building blocks for all computer circuits

- To work out which logic gates are required for an operation, specify the function,F, using a truth table ,then derive the Boolean expression.Then simplify

Commutative: A.B = B.A and A + B = B+A

Distributive:
A.(B+C) = (A.B) + (A.C)
A+(B.C) =(A+B) .(A+C)

Associative:
A.(B.C) = (A.B).C and A+(B+C) = (A+B)+C

De Morgans's Laws:

NotA.B = Not A + Not B and
NotA + NotB = NotA.B