Chapter 2: The Language of Bits

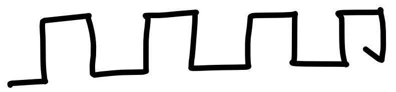
Basic Computer Architecture

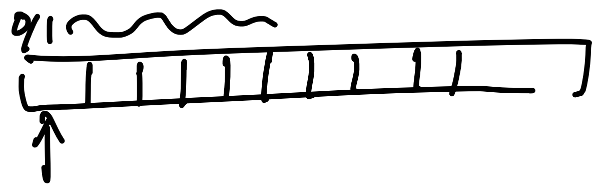
Outline

- Boolean Algebra
- Positive Integers
- Negative Integers
- Floating-Point Numbers
- Strings

What does a Computer Understand ?

* Computers do not understand natural human languages, nor programming languages
* They only understand the language of bits





Bit	0 or 1
✓ Byte	8 bits
Word	4 bytes
kiloByte	1024 bytes $\Rightarrow 2^{10} \quad 2^{20}$
megaByte	10^6 bytes $\Rightarrow 10^3 \times 10^3 = 10^6$

$2^{10} \times 2^{10}$

$1\ KG = 10^3$ gm

$1\ KB = 2^{10}$ bite

$= 10^3$ bite

$1024 \downarrow 1000$

Review of Logical Operations

- A + B (A or B)

A	B	A + B
0	0	0
1	0	1
0	1	1
1	1	1

0/1

= Truth Table

OR

A OR B

~~A~~ || B

A | B

- A.B (A and B)

A	B	A.B
0	0	0
1	0	0
0	1	0
1	1	1

AND

A B

NOT

Review of Logical Operations - II

A	B	A NAND B
0	0	1
1	0	1
0	1	1
1	1	0

A	B	A NOR B
0	0	1
1	0	0
0	1	0
1	1	0

$\bar{A}$

$\sim A$

$A!$

* NAND and NOR operations
* These are universal operations. They can be used to implement any Boolean function.

Review of Logical Operations

* XOR Operation : $(A \oplus B)$

A	B	A XOR B
0 ✓	0 ✓	0
1	0	1
0	1	1
1	1	0

How many truth tables we can build?

Review of Logical Operations

- NOT operator
  - Definition: $\overline{0} = 1$, and $\overline{1} = 0$
  - Double negation: $\overline{\overline{A}} = A$, NOT of (NOT of A) is equal to A itself
- OR and AND operators
  - Identity: A + 0 = A, and A.1 = A
  - Annulment: A + 1 = 1, A.0 = 0

- Idempotence: $A + A = A$, $A.A = A$, The result of computing the OR and AND of A with itself is A.
- Complementarity: $A + \overline{A} = 1$, $A.\overline{A} = 0$
- Commutativity: $A + B = B + A$, $A.B = B.A$, the order of Boolean variables does not matter
- Associativity: $A+(B+C) = (A+B)+C$, $A.(B.C) = (A.B).C$, similar to addition and multiplication.
- Distributivity: $A.(B + C) = A.B + A.C$, $A+ (B.C) = (A+B). (A+C)$ → Use this law to open up parantheses and simplify expressions

$$A + B$$

$$A + (B + C)$$
$$\shortparallel$$
$$(A+B) + C$$

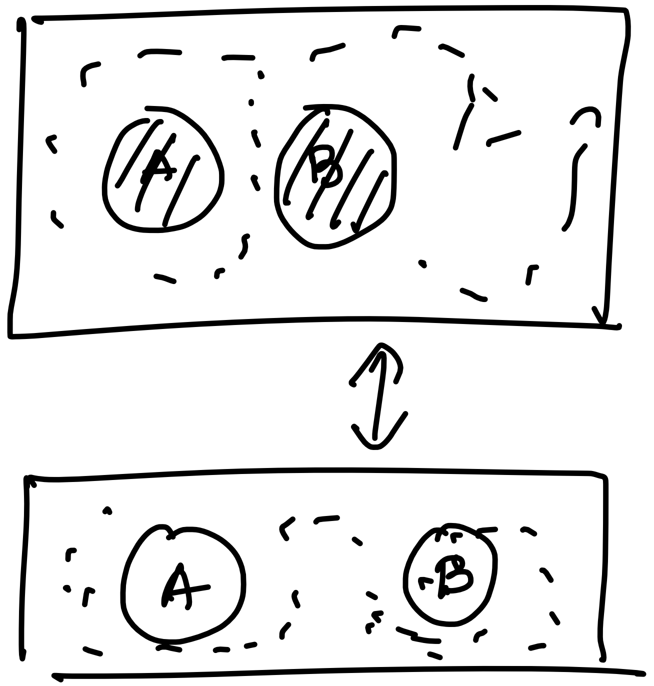
$$x(y+z) = xy + xz$$

De Morgan's Laws

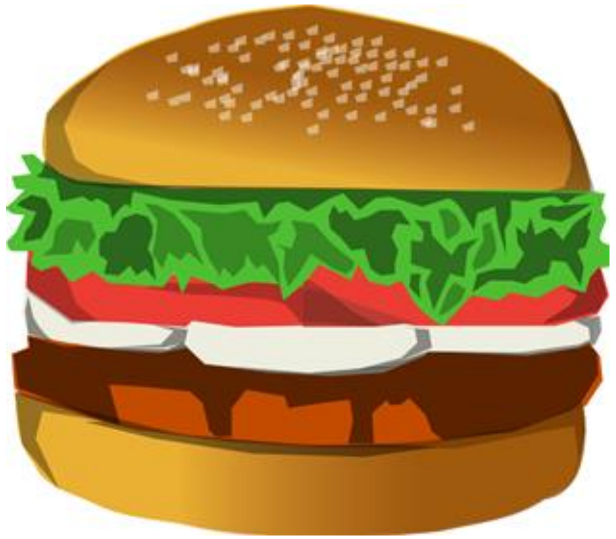
* Two very useful rules

$$\overline{A + B} = \underline{\underline{\overline{A}.\overline{B}}}$$

$$\overline{A.B} = \overline{A} + \overline{B}$$



Consensus Theorem



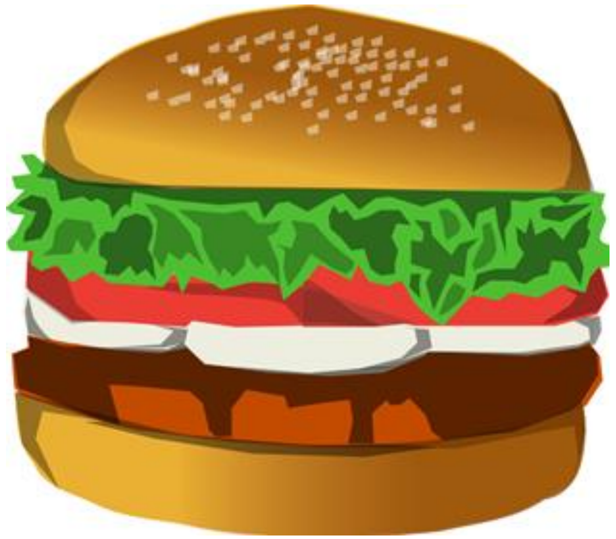
- Prove :
  - $\boxed{X.Y + \bar{X}.Z + Y.Z = X.Y + \bar{X}.Z}$

$X.Y + \bar{X}Z + YZ(X + \bar{X})$

$XY + \bar{X}Z + XYZ + \bar{X}YZ$

$XY(1+Z) + \bar{X}Z(1+\cancel{Y}) = XY + \bar{X}Z$

Consensus Theorem



- Prove :
  - $X.Y + \overline{X}.Z + Y.Z = X.Y + \overline{X}.Z$

Outline

- Boolean Algebra
- Positive Integers
- Negative Integers
- Floating Point Numbers
- Strings

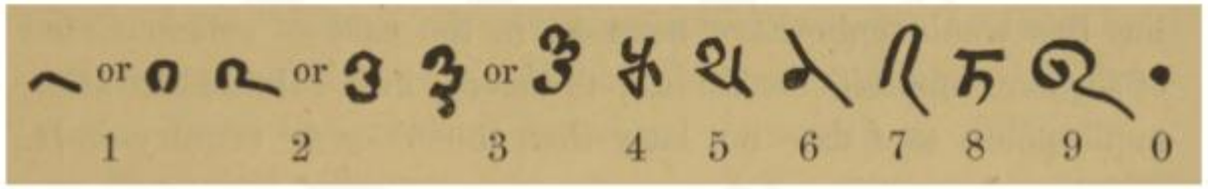
Representing Positive Integers

* Ancient Roman System

Symbol	I	V	X	L	C	D	M
Value	1	5	10	50	100	500	1000

* Issues :
  * There was no notion of 0
  * Very difficult to represent large numbers
  * Addition, and subtraction (very difficult)

Indian System (place –value system)



Bakshali numerals, 7th century AD

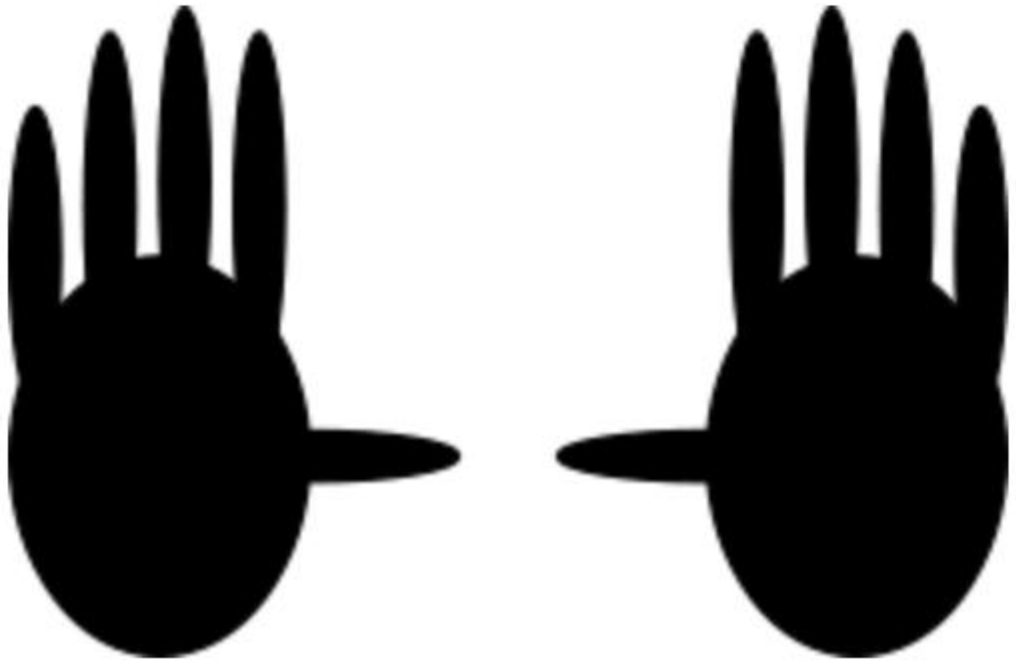
* Uses the place value system

$5301 = 5 * 10^3 + 3 * 10^2 + 0 * 10^1 + 1*10^0$

Example in base 10

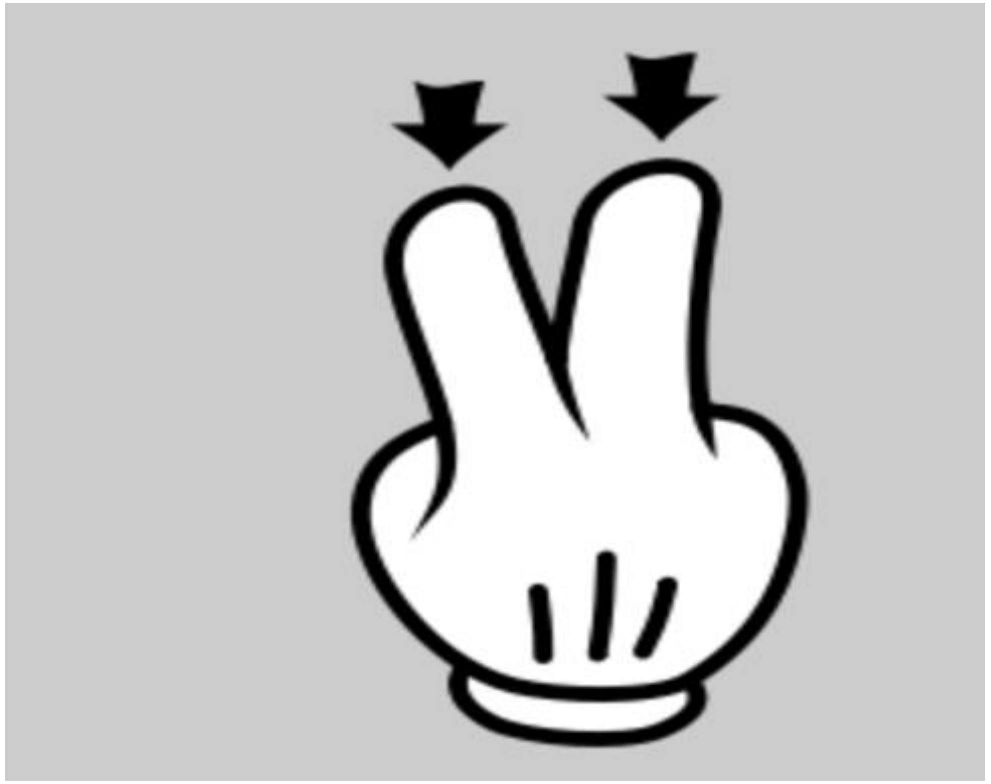
Number Systems in Other Bases

- Why do we use base 10 ?
  - because ...



What if we had a world in which ...

- People had only two fingers.



Binary Number System

- They would use a number system with base 2.

Number in decimal	Number in binary
5	101
100	1100100
500	111110100
1024	10000000000

MSB and LSB

- MSB (Most Significant Bit) → The leftmost bit of a binary number. E.g., MSB of 1110 is 1
- LSB (Least Significant Bit) → The rightmost bit of a binary number. E.g., LSB of 1110 is 0

Hexadecimal and Octal Numbers

- Hexadecimal numbers
  - Base 16 numbers – 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - Start with 0x
- Octal Numbers
  - Base 8 numbers – 0,1,2,3,4,5,6,7
  - Start with 0

Examples

Convert 110010111 to the octal format : $\underbrace{110}\,\underbrace{010}\,\underbrace{111}$ = 0627

Convert 111000101111 to the hex format : $\underbrace{1110}\,\underbrace{0010}\,\underbrace{1111}$ = 0xE2F

Examples

Convert 110010111 to the octal format : $\underbrace{110}\,\underbrace{010}\,\underbrace{111}$ = 0627

Convert 111000101111 to the hex format : $\underbrace{1110}\,\underbrace{0010}\,\underbrace{1111}$ = 0xE2F

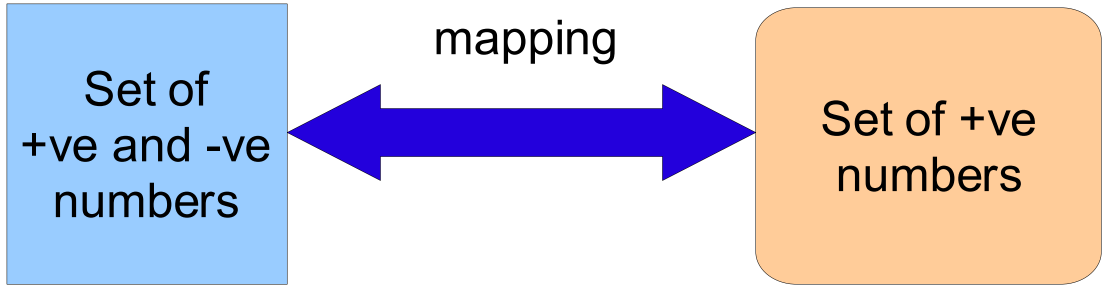
Outline

- Boolean Algebra
- Positive Integers
- Negative Integers
- Floating Point Numbers
- Strings

Representing Negative Integers

- Problem
  - Assign a binary representation to a negative integer
  - Consider a negative integer, S
  - Let its binary representation be : $x_n x_{n-1} \dots x_2 x_1$ (x_i=0/1)
  - We can also expand it to represent an unsigned, +ve, number, N
  - If we interpret the binary sequence as :
    - An unsigned number, we get N
    - A signed number, we get S

* We need a mapping :
  * F : S → N (mapping function)
  * S → set of numbers (both positive and negative – signed)
  * N → set of positive numbers (unsigned)



Properties of the Mapping Function

- Preferably, needs to be a one to one mapping
- All the entries in the set, S, need to be mapped
- It should be easy to perform addition and subtraction operations on the representation of signed numbers
- Assume an n bit number system

$$\mathrm{SgnBit}(u) = \begin{cases} 1, & u < 0 \\ 0, & u >= 0 \end{cases}$$

Sign-Magnitude Base Representation

$$F(u) = SgnBit(u) * 2^{n-1} + |u|$$

sign bit	|u|

- Examples :
  - -5 in a 4 bit number system : 1101
  - 5 in a 4 bit number system : 0101
  - -3 in a 4 bit number system : 1011

Problems

- There are two representations for 0
  - 000000
  - 100000
- Addition and subtraction are difficult
- The most important takeaway point :
  - Notion of the sign bit

1's Complement Representation

$$F(u) = \begin{cases} u, u \geq 0 \\ \sim (|u|) or (2^n - 1 - |u|), u < 0 \end{cases}$$

- Examples in a 4 bit number system
  - 3 → 0011
  - -3 → 1100
  - 5 → 0101
  - -5 → 1010

Notion of sign bit also exists

Problems

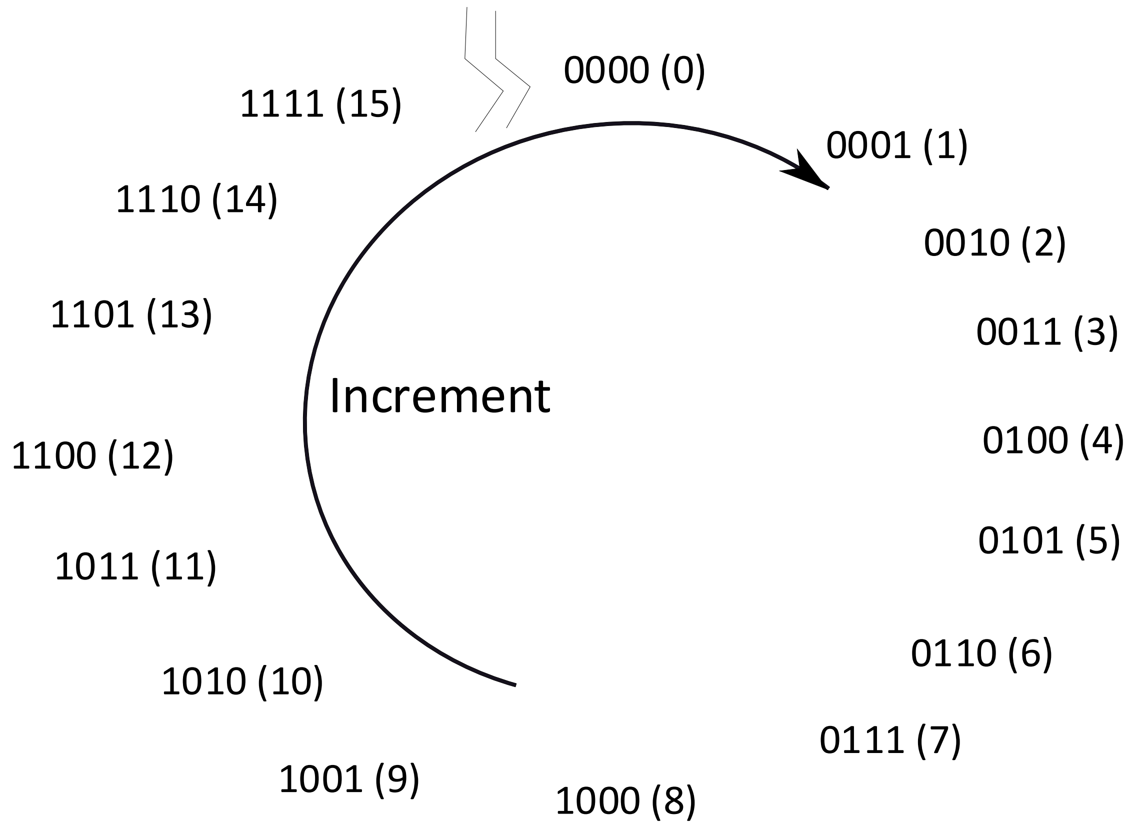
- Two representations for 0
  - 0000000
  - 1111111
- Easy to add +ve numbers
- Hard to add -ve numbers
- Point to note :
  - The idea of a complement

Bias Based Approach

$$F(u) = u + bias$$

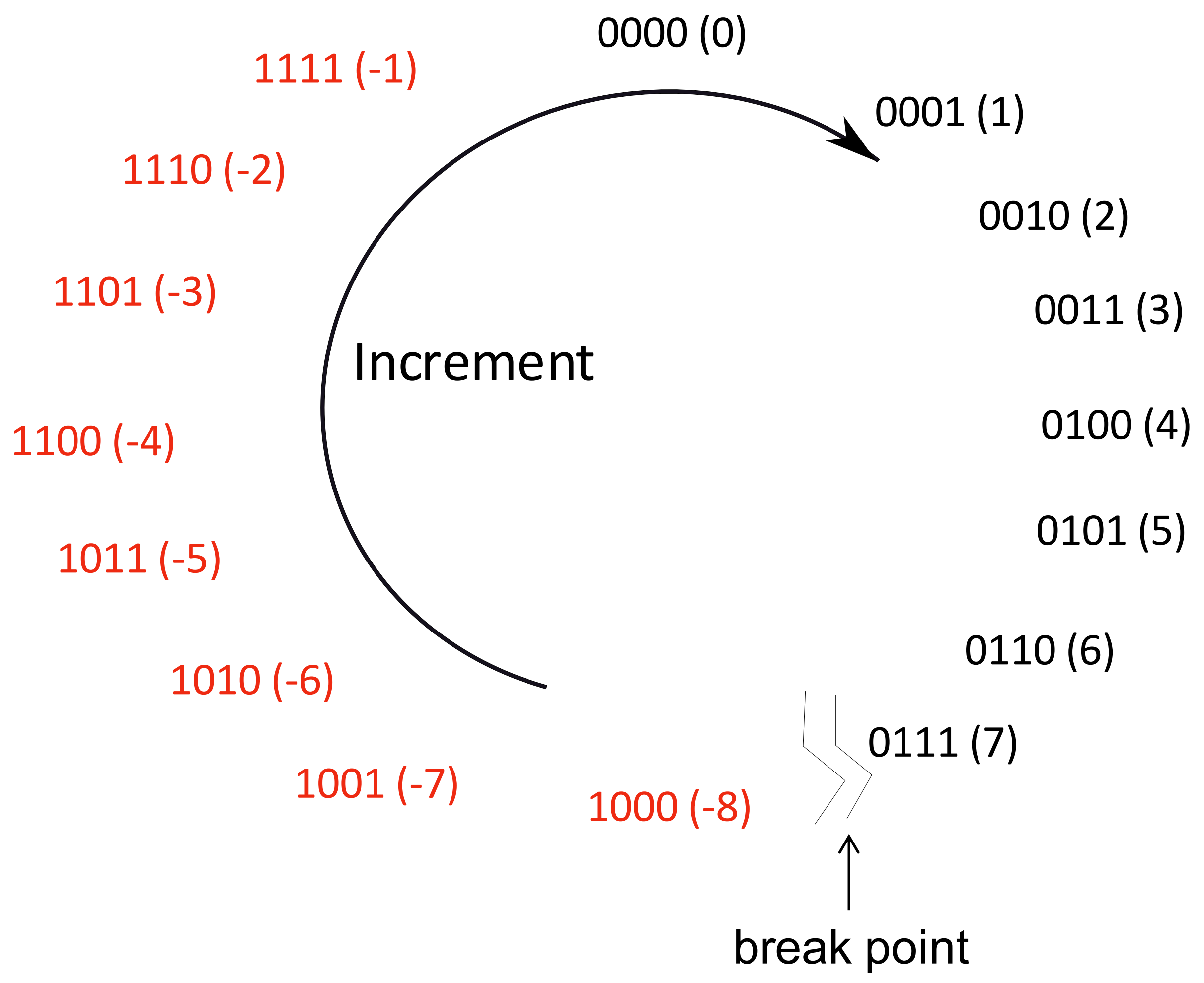
- Consider a 4 bit number system with bias equal to 7
  - -3 → 0100
  - 3 → 1010
- F(u+v) = F(u) + F(v) – bias
- Add and Sub are also easy
- Multiplication is difficult

The Number Circle



Clockwise: increment
Anti-clockwise: decrement

Number Circle with Negative Numbers



Using the Number Circle

- To add M to a number, N
  - locate N on the number circle
  - If M is +ve
    - Move M steps clockwise
  - If M is -ve
    - Move M steps anti-clockwise, or 2^n – M steps clockwise
  - If we cross the break-point
    - We have an overflow
    - The number is too large/ too small to be represented

2's Complement Notation

$$F(u) = \begin{cases} u, 0 \le u \le 2^{n-1} - 1 \\ 2^n - |u|, -2^{n-1} \le u < 0 \end{cases}$$

* F(u) is the index of a point on the number circle. It varies from 0 to 2^n - 1
* Examples
  * 4 → 0100
  * -4 → 1100
  * 5 → 0101
  * -3 → 1101

Properties of the 2's Complement Notation

- Range of the number system :
  - $-2^{(n-1)}$ to $2^{n-1} - 1$
- There is a unique representation for 0 → 000000
- msb of F(u) is equal to SgnBit(u)
  - Refer to the number circle
  - For a +ve number, $F(u) < 2^{(n-1)}$. MSB = 0
  - For a -ve number, $F(u) >= 2^{(n-1)}$. MSB = 1

Properties - II

- Every number in the range $[-2^{(n-1)}, 2^{(n-1)} - 1]$
  - Has a unique mapping
  - Unique point in the number circle
- $a \equiv b \rightarrow (a = b \bmod 2^n)$
- $\equiv$ means same point on the number circle
- $F(-u) \equiv 2^n - F(u)$
  - Moving F(u) steps counter clock wise is the same as moving $2^n - F(u)$ steps clockwise from 0

Prove : $F(u+v) \equiv F(u) + F(v)$

- Start at point u
  - Its index is $F(u)$
  - If v is +ve,
    - move v points clockwise. We arrive at $F(u+v)$.
    - Its index is equal to $(F(u) + v) \bmod 2^n$.
    - Since $v = F(v)$, we have $F(u+v) = (F(u) + F(v)) \bmod 2^n$

Prove : F(u+v) $\equiv$ F(u) + F(v)

- If v is -ve,
  - move |v| points anti-clockwise.
  - Same as moving $2^n - |v|$ points clockwise.
  - We arrive at F(u+v).
  - $F(v) = 2^n - |v|$
  - The index – F(u+v) – is equal to:
    - $(F(u) + 2^n - |v|) \bmod 2^n = (F(u) + F(v)) \bmod 2^n$

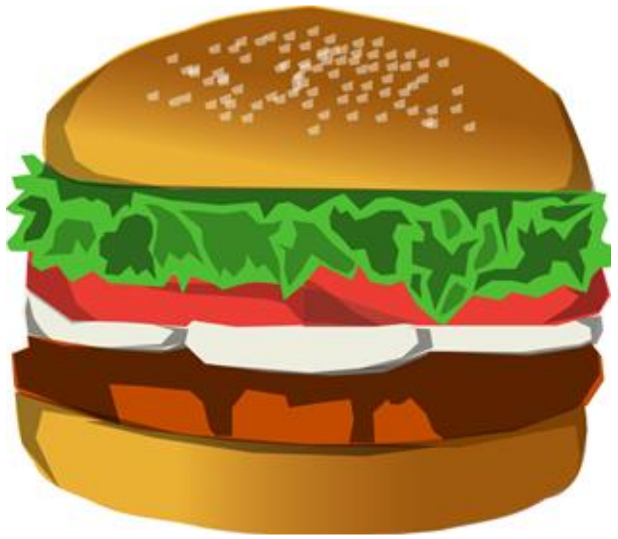
Subtraction

* $F(u-v) \equiv F(u) + F(-v)$

  $\equiv F(u) + 2^n - F(v)$

* Subtraction is the same as addition
* Compute the 2's complement of $F(v)$

Prove that :



- Prove that :

$$F(u*v) \equiv \mathbf{F(u) * F(v)}$$

Computing the 2's Complement

- $2^n - u$

  $= 2^n - 1 - u + 1$

  $= \sim u + 1$

  - ~u (1's complement)

- 1's complement of 0100 | 2's complement of 0100

$$\begin{array}{r} 1111 \\ -\,0100 \\ \hline 1011 \end{array} \qquad \begin{array}{r} 1011 \\ +\,0001 \\ \hline 1100 \end{array}$$

Sign Extension

- Convert a n bit number to a m bit 2's complement number (m > n)
- +ve
  - Add (m-n) 0s in the msb positions
  - Example, convert 0100 to 8 bits → 0000 0100
- -ve
  - $F(u) = 2^n – |u|$ (n bit number) system
  - Need to calculate $F'(u) = 2^m - |u|$

Sign Extension - II

- $2^m - u - (2^n - u)$

  $= 2^m - 2^n$

  $= 2^n + 2^{(n+1)} + \ldots + 2^{(m-1)}$

  $$= \underbrace{1111}_{m-n}\underbrace{0000}_{n}$$

$$\boxed{F'(u) = F(u) + 2^m - 2^n}$$

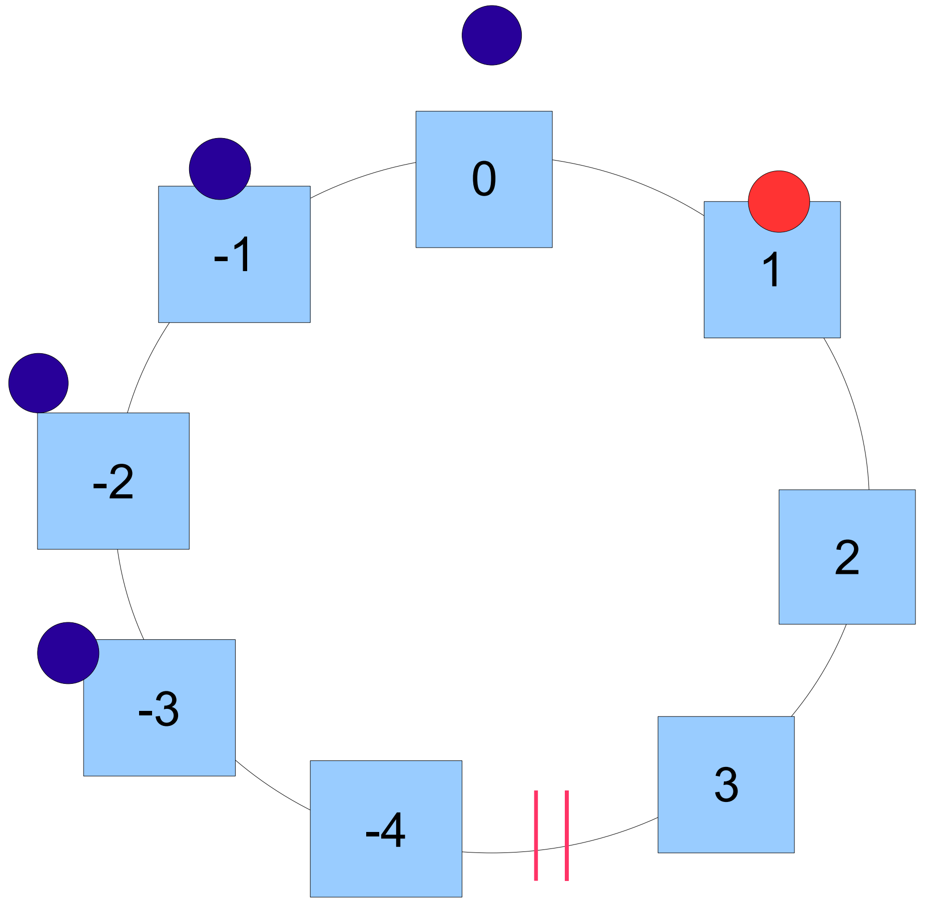
Sign Extension - III

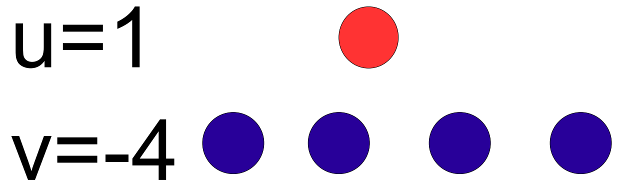
- To convert a negative number :
  - Add (m-n) 1s in the msb positions
- In both cases, extend the sign bit by :
  - (m-n) positions

The Overflow Theorem

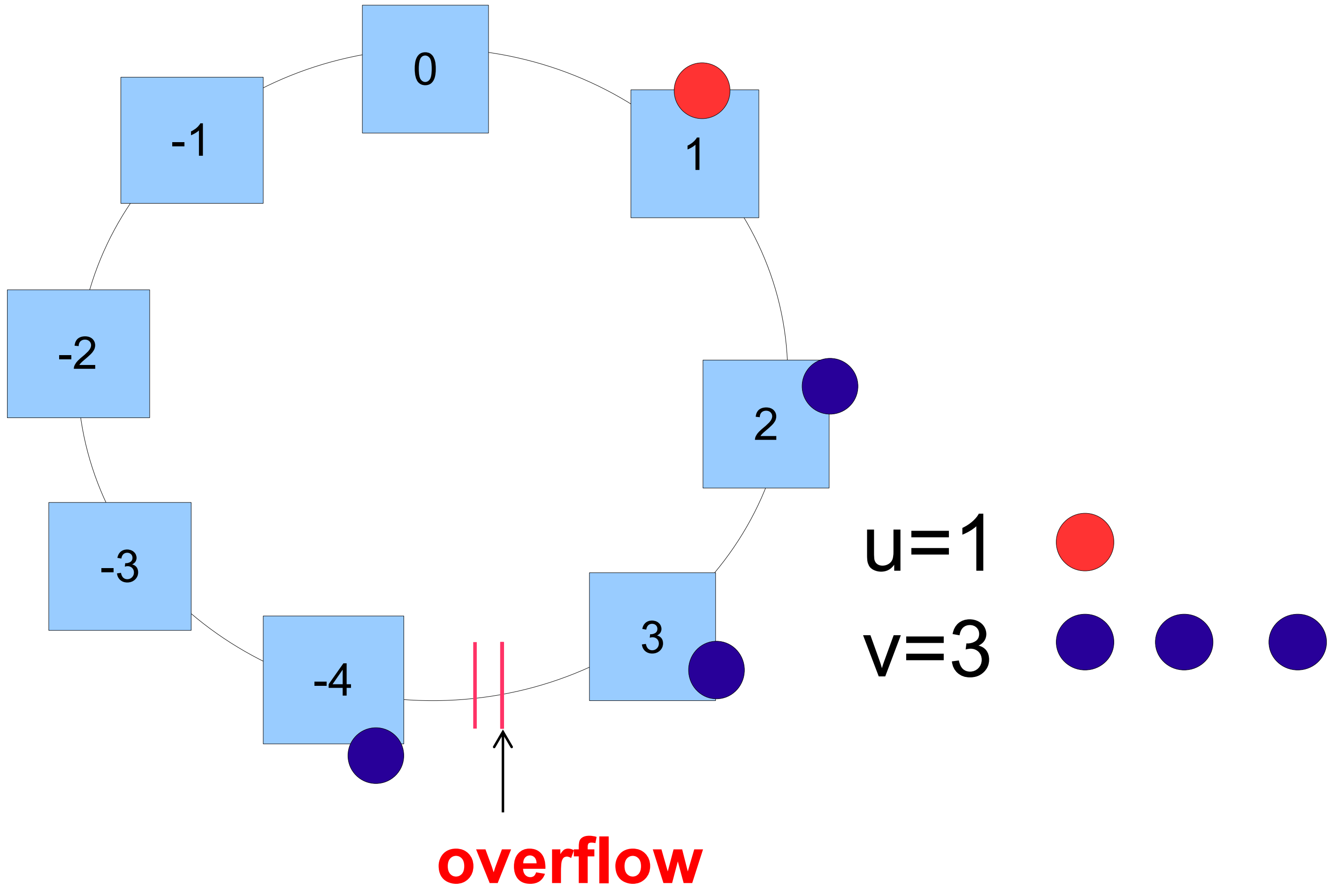
- Add : u + v
- If uv < 0, there will never be an overflow
- Let us go back to the number circle
  - There is an overflow only when we cross the break-point
- If uv = 0, one of the numbers is 0 (no overflow)
- If uv > 0, an overflow is possible

Number Circle: uv < 0





Number Circle: uv > 0



Conditions for an Overflow

- uv <= 0
  - Never
- uv > 0 (u and v have the same sign)
  - The sign of the result is different from the sign of u

Outline

- Boolean Algebra
- Positive Integers
- Negative Integers
- Floating-Point Numbers
- Strings

Floating-Point Numbers

- What is a floating-point number ?
  - 2.356
  - 1.3e-10
  - -2.3e+5
- What is a fixed-point number ?
  - Number of digits after the decimal point is fixed
  - 3.29, -1.83

Generic Form for Positive Numbers

- Generic form of a number in base 10

$$A = \sum_{i=-n}^{n} x_i 10^i$$

- Example :
  - $3.29 = 3 * 10^0 + 2*10^{-1} + 9*10^{-2}$

Generic Form in Base 2

- Generic form of a number in base 2

$$A = \square_{i=-n}^{n} x_i 2^i$$

Number	Expansion
0.375	$2^{-2} + 2^{-3}$
1	2^{0}
1.5	$2^{0} + 2^{-1}$
2.75	$2^{1} + 2^{-1} + 2^{-2}$
17.625	$2^{4} + 2^{0} + 2^{-1} + 2^{-3}$

Binary Representation

- Take the base 2 representation of a floating-point (FP) number
- Each coefficient is a binary digit

Number	Expansion	BinaryRepresentation
0.375	$2^{-2} + 2^{-3}$	0.011
1	2^{0}	1.0
1.5	$2^{0} + 2^{-1}$	1.1
2.75	$2^{1} + 2^{-1} + 2^{-2}$	10.11
17.625	$2^{4} + 2^{0} + 2^{-1} + 2^{-3}$	10001.101

Normalized Form

- Let us create a standard form of all floating point numbers

$$A = (-1)^S * P * 2^X, (P = 1 + M, 0 \le M < 1, X \in Z)$$

- S → sign bit, P → significand
- M → mantissa, X → exponent, **Z** → set of integers

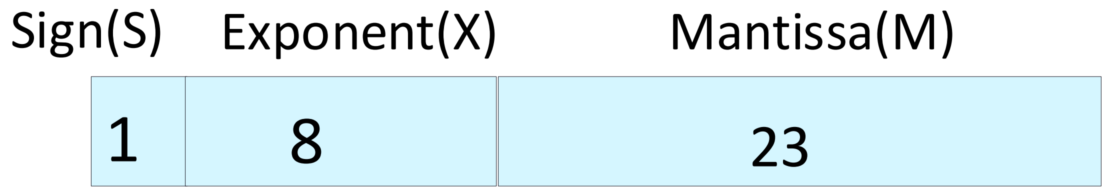
Examples (in decimal)

- 1.3827 * 1e-23
  - Significand (P) = 1.3827
  - Mantissa (M) = 0.3827
  - Exponent (X) = -23
  - Sign (S) = 0
- -1.2*1e+5
  - P = 1.2 , M = 0.2
  - S = 1, X = 5

IEEE 754 Format

- General Principles
  - The significand is of the form : 1.xxxxx
  - No need to waste 1 bit representing (1.) in the significand
  - We can just save the mantissa bits
  - Need to also store the sign bit (S), exponent (X)

IEEE 754 Format - II



- sign bit – 0 (+ve), 1 (-ve)
- exponent, 8 bits
- mantissa, 23 bits

Representation of the Exponent

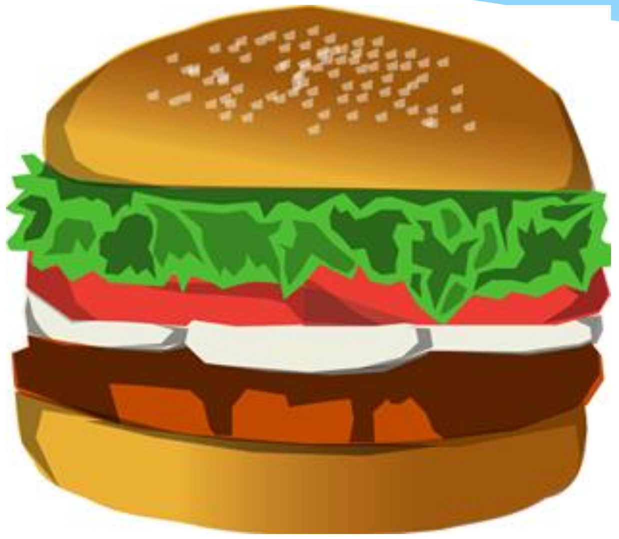
- Biased representation
  - bias = 127
  - E = X + bias
- Range of the exponent
  - 0 – 255 ⟷ -127 to +128
- Examples :
  - X = 0, E = 127
  - X = -23, E = 104
  - X = 30 , E = 157

Normal FP Numbers

- Have an exponent between -126 and +127
- Let us leave the exponents : -127, and +128 for special purposes.

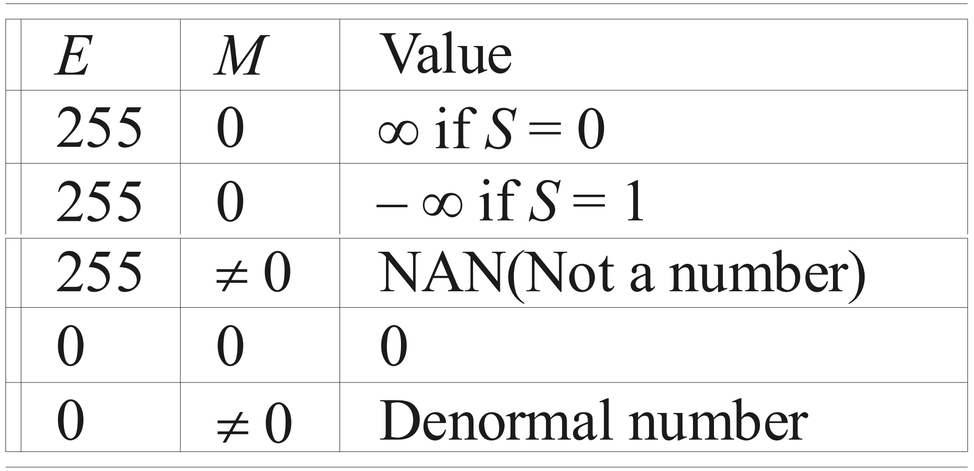
$$A = (-1)^S * P * 2^{E-bias}$$

$$(P = 1 + M, 0 \le M < 1, X \square Z, 1 \le E \le 254)$$



* What is the largest +ve normal FP number ?
* What is the smallest –ve normal FP number ?

Special Floating Point Numbers



E	M	Value
255	0	∞ if $S = 0$
255	0	$-\infty$ if $S = 1$
255	$\neq 0$	NAN(Not a number)
0	0	0
0	$\neq 0$	Denormal number

- NAN + x= NAN $\qquad 1/0 = \infty$
- 0/0 = NAN $\qquad -1/0 = -\infty$
- $\sin^{-1}(5)$ = NAN

Denormal Numbers

```
f = 2^(-126);
g = f/2;
if (g == 0)
  print ("error");
```

- Should this code print "error" ?
- How to stop this behaviour ?

Denormal Numbers - II

$$A = (-1)^S * P * 2^{-126}$$

$$(P = 0 + M, 0 \le M < 1)$$

- Significand is of the form : 0.xxxx
- E = 0, X = -126 (why not -127?)
- Smallest +ve normal number : 2^{-126}
- Largest denormal number :
  - 0.11…11 * 2^{-126} = $(1 - 2^{-23}) * 2^{-126}$
    - $= 2^{-126} - 2^{-149}$

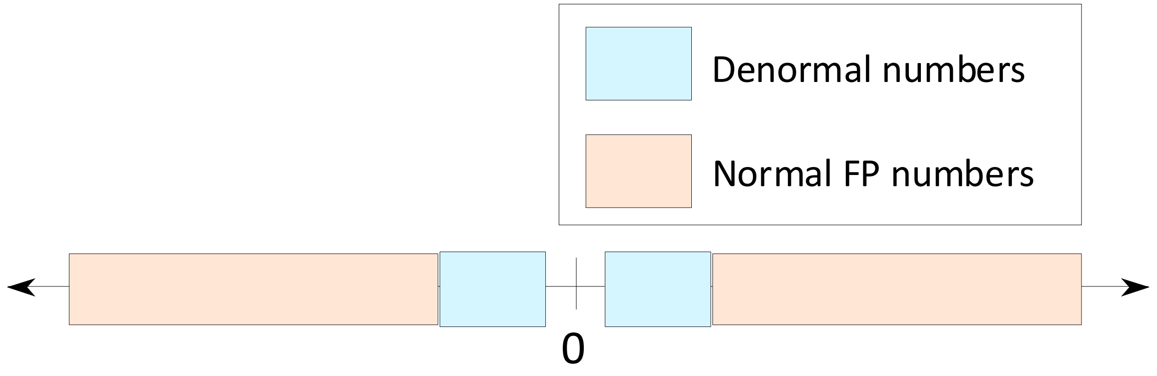
Example

Find the ranges of denormal numbers.

Answer

- For positive denormal numbers, the range is $[2^{-149}, 2^{-126} - 2^{-149}]$
- For negative denormal numbers, the range is $[-2^{-149}, -2^{-126} + 2^{-149}]$

Denormal Numbers in the Number Line



Extend the range of normal floating point numbers.

Double Precision Numbers

Field	Size(bits)
S	1
E	11
M	52

- Approximate range of doubles
  - $\pm 2^{1023} = \pm 10^{308}$
  - This is a lot !!!

Floating Point Mathematics

```
A = 2^(50);
B = 2^(10);
C = (B+A)- A;
```

- C will be computed to be 0
  - There is no way of representing A+B in the IEEE 754 format
- A smart compiler can reorder the operations to increase precision
- Floating point math is approximate

Outline

- Boolean Algebra
- Positive Integers
- Negative Integers
- Floating Point Numbers
- Strings

ASCII Character Set

- ASCII – American Standard Code for Information Interchange
- It has 128 characters
- First 32 characters (control operations)
  - backspace (8)
  - line feed (10)
  - escape (27)
- Each character is encoded using 7 bits

ASCII Character Set

Character	Code	Character	Code	Character	Code
a	97	A	65	0	48
b	98	B	66	1	49
c	99	C	67	2	50
d	100	D	68	3	51
e	101	E	69	4	52
f	102	F	70	5	53
g	103	G	71	6	54
h	104	H	72	7	55
i	105	I	73	8	56
j	106	J	74	9	57
k	107	K	75	!	33
l	108	L	76	#	35
m	109	M	77	$	36
n	110	N	78	%	37
o	111	O	79	&	38
p	112	P	80	(	40
q	113	Q	81	)	41
r	114	R	82	*	42
s	115	S	83	+	43
t	116	T	84	,	44
u	117	U	85	.	46
v	118	V	86	;	59
w	119	W	87	=	61
x	120	X	88	?	63
y	121	Y	89	@	64
z	122	Z	90	^	94

Unicode Format

* UTF-8 (Universal character set Transformation Format)
  * UTF-8 encodes 1,112,064 characters defined in the Unicode character set. It uses 1-6 bytes for this purpose. E.g.अ आ क ख, ௹□ஐஶ
  * UTF-8 is compatible with ASCII. The first 128 characters in UTF-8 correspond to the ASCII characters. When using ASCII characters, UTF-8 requires just one byte. It has a leading 0.
  * Most of the languages that use variants of the Roman script such as French, German, and Spanish require 2 bytes in UTF-8. Greek, Russian (Cyrillic), Hebrew, and Arabic, also require 2 bytes.

UTF-16 and 32

- Unicode is a standard across all browsers and operating systems
- UTF-8 has been superseded by UTF-16, and UTF-32
- UTF-16 uses 2 byte or 4 byte encodings (Java and Windows)
- UTF-32 uses 4 bytes for every character (rarely used)

THE END