Data Representation

Computers process binary patterns (0s and 1s)

To store data on a computer, we must convert it to binary

Decimal to Binary

Divide number by 2, creating quotient and remainder

Repeat previous step with the new quotient until a zero quotient is gained

Answer is found by reading remainders bottom to top

e.g. 74

37 r 0

18 r 1

9 r 0

4 r 1

2 r 0

1 r 0

0 r 1

1001010

Check = 64 + 8 + 2 = 74 = Correct!

Octal

Used to be used as a more convenient base to store long binary values

Convert from Binary to Octal:

Start with the LSB

convert each group of 3 binary digits to one octal digit

e.g. 10101

010101

010 101

25

Convert from Octal to Binary:

Convert each octal digit into 3 binary digits

e.g. 357

3 5 7

011 101 111

011101111

11101111

Hexadecimal

Now used as a more convenient base to store long binary values

Convert from Binary to Hexadecimal:

Start with the LSB

convert each group of 4 binary digits to one hexadecimal digit

e.g. 10010100

1001 0100

9 4

94

Convert from Hexadecimal to Binary:

Convert each hexadecimal digit into 4 binary digits

e.g. 86

8 6

1000 0110

10000110

1 byte = 8 bits

4 bits = 1 nibble = 1 hexadecimal digit

1 word = 2 bytes

1 long word = 4 bytes

In any representation of signed integers, desirable properties are:

one bit-pattern per value

Equal number of positive and negative values

Maximum range of values

No gaps in the range

Fast, efficient hardware implementation

Minimal number of transistors and fast arithmetic

Sign and Magnitude:

MSB = sign

remaining bits = magnitude

for n bits, -(2n-1 – 1) <= Range <= 2n-1 – 1

Pros: Simplest for humans to understand

Cons: Two representations for 0 (+0, -0), Expensive (compare signs and implement subtractors)

One’s Complement:

Get negative of an integer: Invert each bit

e.g. -3 -> +3 is 0011 -> 1100

for n bits, -(2n-1 – 1) <= Range <= 2n-1 – 1

Pros: Cheaper to implement

Cons: Less intuitive for humans to use than S&M, Two representations for 0 (+0, -0)

Two’s Complement (the best one):

Get negative of an integer: Invert each bit and add 1

e.g. -3 -> +3 is 0011 -> 1100 -> 1101

for n bits, -2n-1 <= Range <= 2n-1 – 1

Pros: X – Y = X + (-Y); no need for separate subtractor (needed in S&M) or carry-out adjustments (1’s C); one bit pattern for 0

Con: Asymmetric: there is one more negative value than positive values

Sorting in 2’s C is hard (assuming you could compare numbers, it would always say negative numbers are greater).

Excess Representation:

bitstring with N 0’s = smallest value

bitstring with N 1’s = largest value

Allows you to represent negative numbers while maintaining the same ordering (000, 001, 010, 011 … 111)

e.g. N = 3

8 values in total

Start at -4 (-(0.5 * (2)N))

-4, -3, …. 2, 3

Smallest value = -4, so we shift by 4

Each value stored is +4 of actual value -> Excess-4

Stored Value	Actual Value
000	-4
001	-3
010	-2

etc	

Binary Coded Decimal:

Each decimal digit is represented by a fixed number of bits, usually 4 or 8

Pros: Easy for humans to understand

Cons: Takes up much more space

e.g. 9876510 with 4 bits:

9 8 7 6 5 1 0

1001 1000 0111 0110 0101 0001 0000

100110000111011001010001 0000 (24 bits)

ASCII:

7 bits

Most modern computers extend this to 8 bits (Extended ASCII)

26 lowercase, 26 uppercase, 10 digits, 32 punctuation marks

Remaining 34 bit-patterns: Whitespace (space, return, tab etc) and special control characters

String = Sequence of Characters

Unicode:

Attempts to provide a number for every character, no matter the language

Over 120,000 characters already defined

First 65,536 (16 bit) characters cover the world’s major alphabets

First 127 characters correspond to ASCII characters