

Encoding of Information

- Digital computers only have binary digits available to represent information internally so we must develop binary encodings of data for their use.
- Form of encoding depends upon:
 - the amount of data
 - the operations to be performed on the data
 - the properties of hardware used
- Simplest case – purely symbolic information, i.e., no arithmetic
- b bits available implies 2^b patterns and therefore any assignment of those patterns to represent up to 2^b symbols is a legitimate encoding.
- ASCII is most prevalent symbol or character encoding
 - 7 bits to encode 128 standard characters
 - typically stored in a byte (8 bits) with the extra bit used for error detection
- Others: EBCDIC and Baudot

Numerical Codes

- Assume that we are encoding the **value** of positive integers
- Single-radix positional number systems are defined by
 - a base or radix, r – decimal system $\rightarrow r = 10$
 - number of digits stored for each encoding, m

$$(7392)_{10} = 7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

or more generally, an m -digit decimal number D is the result of the encoding

$$D = \sum_{i=0}^{m-1} d_i \times r^i$$

where $0 \leq d_i < r$

- Such a system can represent all integers $0 \leq j < r^m - 1$
- conversion to decimal is easily done via the definition
- the process can be reversed for conversion from decimal

Base r to decimal: Expand, multiply, add (all arithmetic in decimal)

$$\begin{aligned}(4021)_5 &= 4 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 1 \times 5^0 \\ &= 4 \times 125 + 0 \times 25 + 2 \times 5 + 1 \times 1 \\ &= 500 + 0 + 10 + 1 \\ &= (511)_{10}\end{aligned}$$

Explicit evaluation of powers needed above. This can be made into a recursion.

$$\begin{aligned}(d_3d_2d_1d_0)_r &= d_3r^3 + d_2r^2 + d_1r^1 + d_0r^0 \\ &= (d_3r^2 + d_2r^1 + d_1r^0)r + d_0 \\ &= ((d_3r^1 + d_2r^0)r + d_1r^0)r + d_0 \\ &= (((d_3)r^1 + d_2)r + d_1)r + d_0\end{aligned}$$

This is actually Horner's rule for the evaluation of polynomials.

Base r to decimal: Horner's rule (all arithmetic in decimal)

```
 $S = 0$   
do  $i = m - 1, \dots, 0$   
     $S \leftarrow Sr + d_i$   
end do  
 $S$  is decimal equivalent
```

Decimal to base r: reverse the process (all arithmetic in decimal)

Note any integer N can be expressed as $N = QD + R$ where Q, D, R are all integers. Q is the quotient and $R < D$ is the remainder. Write this as $N \div D = Q \text{ rem } R$.

$$\begin{aligned} D &= (((d_3)r^1 + d_2)r + d_1)r + d_0 \\ Q_0 &= D \div r \\ &= ((d_3)r^1 + d_2)r + d_1 \text{ rem } d_0 \end{aligned}$$

So d_0 can be found by one decimal integer division and Q_0 has the same form as D with one fewer digit in base r . Repeat to get d_1, \dots, d_3 :

$$\begin{aligned} Q_1 &= Q_0 \div r \\ &= (d_3)r^1 + d_2 \text{ rem } d_1 \\ Q_2 &= Q_1 \div r \\ &= (d_3) \text{ rem } d_2 \\ Q_3 &= Q_2 \div r \\ &= 0 \text{ rem } d_3 \end{aligned}$$

Decimal to base r: all arithmetic in decimal

$$Q = D$$

do $i = 0, \dots, m - 1$

$$(S, R) \leftarrow Q \div r$$

$$d_i \leftarrow R$$

$$Q \leftarrow S$$

end do

$(d_{m-1} \dots d_0)_r$ is base r equivalent

- For digital computers r is usually taken to be a power of 2
- binary $r = 2$, octal $r = 8$ and hexadecimal $r = 16$
- hexadecimal is often used as a convenient display form of binary information. The characters A through F are used to extend the set of digits beyond 9.

Decimal	binary	octal	hexadecimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Conversion between binary, octal and hexadecimal is a simple regrouping of bits.

Binary to octal: group 3 bits together starting from right end.

$$\begin{aligned}(10110001101011)_2 &= (26153)_8 \\ (10110001101011)_2 &= (2C6B)_{16}\end{aligned}$$

$$\begin{array}{ccccccccc} & & & 6 & & & 5 & & \\ & & & \frown & & & \frown & & \\ \underbrace{10} & & 110 & & 001 & & 101 & & 011 \\ & 2 & & & 1 & & & & 3 \end{array}$$

$$\begin{array}{ccccccc} & & & C & & & B \\ & & & \frown & & & \frown \\ \underbrace{10} & & 1100 & & 0110 & & 1011 \\ & 2 & & & 6 & & \end{array}$$

- So far all of the representations have assumed positive integers.
- Fractions are easily incorporated in to radix systems
- Assume a fixed number of digits and a fixed position of the radix point.

6 digits with 2 fractional places

$$\begin{aligned}
 D &= d_3r^3 + d_2r^2 + d_1r^1 + d_0r^0 \\
 &\quad + d_{-1}r^{-1} + d_{-2}r^{-2} \\
 (4021.20)_5 &= 4 \times 5^3 + 0 \times 5^2 + 2 \times 5^1 + 1 \times 5^0 \\
 &\quad + 2 \times 5^{-1} + 0 \times 5^{-2}
 \end{aligned}$$

In general,

$$D = \sum_{i=-n}^{m-1} d_i \times r^i$$

where $0 \leq d_i < r$.

- There are $m + n$ digits
- n fractional positions
- note the radix point position is fixed for all computations

- Integers are also fixed point (with no fractional positions)
- Integers are used extensively in actual machines and computations
- Fixed point used in many special-purpose devices
- Algorithms that used fixed point must be carefully designed to deal with small range of numbers
- Note that it is possible to define codes that use more than one radix. That is, the powers of the form r^3 are replaced by $r_1r_2r_3$.
- Such systems are called mixed radix
- Mixed radix tend to have special properties with respect to speed of certain operations and fault tolerance. e.g., redundant residue arithmetic.

Addition and Subtraction of Positive Fixed Point Numbers

- binary addition and subtraction algorithms are simple adaptations of their decimal counter parts
- main issue is handling carries and borrows
- algorithms can be altered to affect hardware cost and evaluation time

Addition: Assume X and Y are two integers with binary representations $(x_k \dots x_1 x_0)$ and $(y_k \dots y_1 y_0)$ respectively. The representation of the sum $S = X + Y$ is denoted $(s_{k+1} s_k \dots s_1 s_0)$. Note the extra bit. c_i will denote the binary carry from the production of s_i that must be added to the sum defining s_{i-1} . ($c_0 = 0$ by definition.)

- $(c_{i+1}, s_i) = x_i + y_i + c_i$ (where the addition is binary)
- this implies a sequential evaluations of the s_i due to the requirement of the availability of c_i .
- overcoming this is one of the design problems we face.

$$\begin{array}{rcl}
X & : & - 1 1 0 1 1 \\
Y & : & - 1 0 0 1 1 \\
C & : & 1 0 0 1 1 0 \\
S & : & 1 1 1 1 1 0
\end{array}$$

Note if 6 bits were available for S then the answer has been produced and can be stored. If however all representation is based on 5 bits then we have an overflow. This is detected by looking at c_{k+1}

Addition and Subtraction of Positive Fixed Point Numbers

Subtraction: Assume X and Y are two integers with binary representations $(x_k \dots x_1 x_0)$ and $(y_k \dots y_1 y_0)$ respectively. The representation of the difference $D = X - Y$ is denoted $(d_k \dots d_1 d_0)$. b_i will denote the binary borrow needed to compute d_{i-1} . It must be subtracted when producing d_i . ($d_0 = 0$ by definition.)

- $(b_{i+1}, d_i) = x_i - y_i - b_i$ (where the subtraction is binary)
- this implies a sequential evaluations of the d_i due to the requirement of the availability of b_i .
- overcoming this is one of the design problems we face.

$$\begin{array}{rcl}
X & : & - 1 1 0 0 1 \\
Y & : & - 0 0 0 1 0 \\
B & : & 0 0 1 1 0 0 \\
D & : & - 1 0 1 1 1
\end{array}$$

Note $b_5 = 0$ so the process can stop. This implies that $X \geq Y$.

$$\begin{array}{rcl}
X & : & - 0 0 0 1 0 \\
Y & : & - 0 1 1 0 1 \\
B & : & 1 1 1 0 1 0 \\
D & : & ? 1 0 1 0 1
\end{array}$$

Note $b_5 = 1$ so there is borrowing to do from places that do not exist. This implies that $X < Y$ and underflow has occurred, i.e., we no longer have only positive integers involved.

Positive and Negative Integers

- In order to represent positive and negative integers given a fixed number of patterns in the set of codewords, we must divide the patterns roughly in half.
- Positive or negative is a binary decision (with the exception of 0) so it requires at least one bit to represent the result.
- Simplest approach is to use the sign/magnitude encoding.
 - $9 \rightarrow 01001$ and $-9 \rightarrow 11001$
 - non-unique representation of 0: 00000 or 10000

- Arithmetic follows normal rules.
- Addition:
 - compare signs
 - if same then add magnitudes and give result same sign
 - if different then compare magnitudes and subtract smaller from larger and give result sign of larger magnitude
- Subtraction: reverse sign of operand and add
- requires comparators, adder, subtractor

Example:

$$(+25) + (-37) = -(37 - 25) = (-12)$$

- It is possible to represent negative fixed point numbers in a way that simplifies the arithmetic and as a result sign/magnitude is used only selectively.
- We want:
 - unique 0
 - approximately half positive and half negative codewords
 - simpler addition/subtraction (and therefore multiplication and division)
- We will try and associated with each integer M another integer M' that can act as the representation of $-M$
 - $(M')' = M$
 - $M + M' = 0$

Complement Representations

- Two types of M' : diminished radix complement and radix complement
- Works for any radix
- Satisfies requirements of improvement over sign/mag.

Diminished Radix Complement:

- Assume radix r and n digits.
- Given an integer M the diminished radix or $(r - 1)$ complement is

$$M' = (r^n - 1) - M$$

- Examples:
 - $(r, n) = (10, 2)$: $M = 32$ and $M' = 99 - 32 = 67$
 - $(r, n) = (2, 4)$: $M = 0101$ and $M' = 1111 - 0101 = 1010$
- These are called 9's complement and 1's complement respectively
- Note that $r^n - 1$ has an encoding where all digits are $r - 1$.
- So the digits of M' are all trivially produced (simultaneously if necessary) by subtracting the digit of M from $r - 1$. This must yield a digit m'_i such that $0 \leq m'_i < r - 1$

- Since the corresponding digits of M and M' sum to $r - 1$ it is clear that $(M')' = M$
- For 1's complement: $m'_i = 1 - m_i$ has only two forms.

$$\begin{aligned} m'_i &= 1 - m_i \\ 1 &= 1 - 0 \\ 0 &= 1 - 1 \end{aligned}$$

- So replacing $1 \rightarrow 0$ and $0 \rightarrow 1$ gives complement.
- Unfortunately, $M' + M \neq 0$ by definition.
- $M = 32$ and $M' = 99 - 32 = 67$ so $M + M' = 99$
- We need to add a one into M' at some point.

Radix Complement:

- Assume radix r and n digits.
- Given an integer $M \neq 0$ the diminished radix or r complement is

$$M' = [(r^n - 1) - M] + 1$$

- This is simply adding 1 to the $(r - 1)$ complement.
- If $M = 0$ then $M' = 0$.

$$\begin{aligned}(M')' &= (r^n - 1) - M' + 1 \\ &= (r^n - 1) - [(r^n - 1) - M + 1] + 1 \\ &= (r^n - 1) - (r^n - 1) + M - 1 + 1 \\ &= M\end{aligned}$$

- Examples:
 - $(r, n) = (10, 2)$: $M = 32$ and $M' = (99 - 32) + 1 = 68$
 - $(r, n) = (2, 4)$: $M = 0101$ and $M' = (1111 - 0101) + 1 = 1011$
- These are called 10's complement and 2's complement respectively
- Note that after producing the $(r - 1)$ complement an addition is needed, potentially increasing the cost of its use to unacceptable levels

$$\begin{aligned}
M' + M &= [(r^n - 1) - M + 1] + M \\
&= r^n - 1 - M + 1 + M \\
&= r^n
\end{aligned}$$

But, r^n is encoded by a 1 followed by n 0's. So it is truncated we get 0 as desired.

- $M = 32$ and $M' = 68$: $M + M' = 100 \rightarrow 00$
- $M = 0101$ and $M' = 1011$: $M + M' = 10000 \rightarrow 0000$

Pattern	s/m decoding	2's complement
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-0	-8
1001	-1	-7
1010	-2	-6
1011	-3	-5
1100	-4	-4
1101	-5	-3
1110	-6	-2
1111	-7	-1

Note sign bit still leftmost.

Pattern	0	1	2	3	4	5	6	7	8	9
10's comp. decode	0	1	2	3	4	-5	-4	-3	-2	-1

Signs can now be ignored during addition

$$\begin{array}{rcl} 7 + (-3) & = & 4 \\ 0111 + 1101 & = & (1) 0100 \end{array}$$

$$\begin{array}{rcl} (-3) + (-4) & = & (-7) \\ 1101 + 1100 & = & (1) 1001 \end{array}$$

The carry-out bit value of 1 is ignored. Overflow detection is discussed in the text.

Subtraction uses 2's complement definition and ignores signs of operands.

To compute $X - Y$:

- form Y' the 2's complement of Y
- add $X + Y'$
- need to evaluate Y' efficiently
- get Z the 1's complement of Y
- set carry-in bit $c_0 = 1$

$X = 7 = 0111$, $Y = 3 = 0011$

X	:	-	0	1	1	1
Z	:	-	1	1	0	0
C	:		1	1	1	1
S	:	-	0	1	0	0

Binary Codes for Decimals

- There is a compromise approach used occasionally on computers. The numerical codes encode the value of the number not its digits in the original base (typically base 10)
- Binary codes for decimals encode the values of the each of the decimal digits rather than the value of the integer.
- The binary encoding of the value of $(12)_{10}$ yields 1100.
- Binary codes for decimals would encode the value for the units digit 2 and the value for the tens digit 1 separately.
- nine digits requires a 4 bit encoding for each
- BCD uses the standard binary representation for each digit of the decimal number so $(12)_{10} = (0001\ 0010)_{BCD}$.
- Arithmetic can be defined using basic 4 bit binary arithmetic on each digit and BCD carry generation.