Data Representation in Digital Computers

The material presented herein is excerpted from a series of lecture slides originally prepared by David Lowther and Peet Silvester for their textbook Computer Engineering. The material has been adapted by Frank Ferrie to fit the current implementation of course 304-221.

Binary Numbers

The common representation in most digital computers. Only 2 symbols are required! Consider the following example:

$$1\ 1\ 0 \quad 1\ 0\ 1 \quad 1\ 1\ 1 \quad 0\ 0\ 1$$

$110101111001_2 =$

$$\begin{array}{cccccc} = & 1 \times 2^{11} & + & 1 \times 2^{10} & + & 0 \times 2^{9} \\ + & 1 \times 2^{8} & + & 0 \times 2^{7} & + & 1 \times 2^{6} \\ + & 1 \times 2^{5} & + & 1 \times 2^{4} & + & 1 \times 2^{3} \\ + & 0 \times 2^{2} & + & 0 \times 2^{1} & + & 1 \times 2^{0} \end{array}$$

$$= \quad 3449_{10}$$

Binary Numbers: cont.

While binary numbers are fine for computers, humans prefer more compact representations. This can be accomplished by extending our repetoire of symbols. Converting to *octal*, i.e., Base-8 is easy:

1 1 0	1 0 1	1 1 1	0 0 1
6	5	7	1

$$
\begin{aligned}
&= (1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0)\ \times 2^9 = 6 \times (8^3) \\
&+ (1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)\ \times 2^6 = 5 \times (8^2) \\
&+ (1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)\ \times 2^3 = 7 \times (8^1) \\
&+ (0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)\ \times 2^0 = 1 \times (8^0)
\end{aligned}
$$

$$= 6571_8$$

Notice how the above factoring is equivalent seperating binary digits into groups of 3, i.e., 2^8.

Binary Numbers: cont.

Conversion to hexadecimal notation is accomplished by factoring the numbers into powers of 16. This can be accomplished be seperating binary digits into groups of 4 as follows.

1 1 0 1	0 1 1 1	1 0 0 1
D	7	9

$$
\begin{aligned}
&= (1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)\ \times 2^8 \\
&+ (0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0)\ \times 2^4 \\
&+ (1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)\ \times 2^0
\end{aligned}
$$

$$
\begin{aligned}
&= D \times (16^2) \\
&+ 7 \times (16^1) \\
&+ 9 \times (16^0)
\end{aligned}
$$

($A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, $F = 15$)

Base Conversion

Decimal to binary, i.e., $N_{10} \to N_2$, for some number N.

Write N as an even number plus 0 or 1 as appropriate:

$$N = Q^{(1)} * 2 + R^{(1)}$$

Observe that the first term above is *even* and the second is *odd*. Notice that the second is also in N_2.

Similarly, write

$$Q^{(1)} = Q^{(2)} * 2 + R^{(2)}$$

$$N = (Q^{(2)} * 2 + R^{(2)}) * 2 + R^{(1)}$$

Base Conversion: cont.

We can continue expanding N recursively:

$$\begin{aligned} N &= (Q^{(2)} * 2 + R^{(2)}) * 2 + R^{(1)} \\ &= ((Q^{(3)} * 2 + R^{(3)}) * 2 + R^{(2)}) * 2 + R^{(1)} \\ &= (((Q^{(4)} * 2 + R^{(4)}) * 2 + R^{(3)}) * 2 + R^{(2)}) \\ &\quad *2 + R^{(1)} \end{aligned}$$

Re-writing back to front . . .

$$\begin{aligned} N &= R^{(1)} + R^{(2)} * 2 + R^{(3)} * 2 * 2 \\ &\quad + R^{(4)} * 2 * 2 * 2 + \ldots \\ &= R^{(1)} * 2^0 + R^{(2)} * 2^1 + R^{(3)} * 2^2 + R^{(4)} * 2^3 \\ &= + \ldots + R^{(n)} * 2^{n-1} \end{aligned}$$

Hence the R^i are exactly the coefficients we're looking for.

Example

$$
\begin{aligned}
9_{10} &= (4_{10} * 2) + 1 \\
&= ((2_{10} * 2) + 0) * 2 + 1 \\
&= (((\mathbf{1}_{10} * 2) + \mathbf{0}) * 2 + \mathbf{0}) * 2 + \mathbf{1} \\
&= 1001_2
\end{aligned}
$$

Formal Procedure base $a \to b$

1. Set $Q^{(0)} = N_a$
2. Compute

$$
\begin{aligned}
Q^{(j)} &= \text{integer } \left[\frac{Q^{(j-1)}}{b}\right] \\
R^{(j)} &= \text{remainder } \left[\frac{Q^{(j-1)}}{b}\right]
\end{aligned}
$$

   Until $Q^{(j)} = 0$

3. $N_b = R^n * b^{n-1} + R^{n-1} * b^{n-2} + \ldots + R^1 * b^0$

Another Example

$179_{10} \to N_2$

$$
\begin{array}{lll}
Q_0 = & 179 & \\
Q_1 = & \frac{179}{2} = 89 & R_1 = 1 \\
Q_2 = & \frac{89}{2} = 44 & R_2 = 1 \\
Q_3 = & \frac{44}{2} = 22 & R_3 = 0 \\
Q_4 = & \frac{22}{2} = 11 & R_4 = 0 \\
Q_5 = & \frac{11}{2} = 5 & R_5 = 1 \\
Q_6 = & \frac{5}{2} = 2 & R_6 = 1 \\
Q_7 = & \frac{2}{2} = 1 & R_7 = 0 \\
Q_8 = & \frac{1}{2} = 0 & R_8 = 1
\end{array}
$$

$179_{10} \to 10110011_2$

What About Other Bases?

$179_{10} \rightarrow N_{16}$

$$\begin{aligned} Q_0 = & \ 179 & \\ Q_1 = & \ \frac{179}{16} = 11 & R_1 = 3 \\ Q_2 = & \ \frac{11}{16} = 0 & R_2 = 11 \end{aligned}$$

For digits greater than 9, we continue with A, B, C, ...

Hence $179_{16} \rightarrow B3_{16}$.

Fractions

Handled as an extension of the integers.

$$\begin{aligned} 27 &= 2 \times 10^1 + 7 \times 10^0 \\ 27.51 &= 2 \times 10^1 + 7 \times 10^0 + 5 \times 10^{-1} + 1 \times 10^{-2} \end{aligned}$$

The decimal point seperates *negative* exponents from nonnegative.

Exactly the same technique applies in binary (or other) notations.

$$\begin{aligned} 10 &= 1 \times 2^1 + 0 \times 2^0 \\ 10.11 &= 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \end{aligned}$$

The *binary point* seperates negative from nonnegative exponents.

The point is called	binary point,
	decimal point,
	octal point, etc.

according to the notation used.

Conversion of Fractions

A fraction, e.g., 0.625_{10} can be converted to binary notation by *multiplying and dividing by 2* as follows:

$$
\begin{aligned}
0.625_{10} &= 0.625 \times \frac{2}{2} \\
&= \frac{0.625 \times 2}{2} \\
&= 1.250 \times 2^{-1} \\
&= 1 \times 2^{-1} + (0.250) \times 2^{-1} \\
&= 1 \times 2^{-1} + 0.5 \times 2^{-2} \\
&= 1 \times 2^{-1} + 0 \times 2^{-2} + (0.5) \times 2^{-2} \\
&= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\
&= 0.101_2
\end{aligned}
$$

Recipe: Multiply by target base, seperate integer and fractional parts. Repeat. Integer parts taken in order are the fraction digits!

Another Example

$$
\begin{aligned}
0.7_{10} &= (0.7 \times 2) \times 2^{-1} \\
&= 1 \times 2^{-1} + (0.4) \times 2^{-1} \\
&= 1 \times 2^{-1} + 0 \times 2^{-2} + (0.8) \times 2^{-2} \\
&= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\
&\quad +(0.6) \times 2^{-3} \\
&= 0.101_2 + (0.6) \times 2^{-3} \\
&= 0.101_2 + 1 \times 2^{-4} + (0.2) \times 2^{-4} \\
&= 0.101_2 + 1 \times 2^{-4} + 0 \times 2^{-5} \\
&\quad +(0.4) \times 2^{-5} \\
&= 0.10110_2 + (0.4) \times 2^{-5}
\end{aligned}
$$

But note:

$$
\begin{aligned}
0.7 &= 0.1_2 + (0.4) \times 2^{-1} \\
&= 0.10110_2 + (0.4) \times 2^{-5} \\
&= 0.1\ 0110\ 0110\ 0110\ 0110\ 0110\ \ldots
\end{aligned}
$$

Fractions exactly representable in one notation are not always so in another.

Arithmetic

Addition is much the same in any number system. Digits are added one at a time. Carries are propagated as a third set of digits. Example:

0000	carry
4562	addend
1719	augend
6281	sum

Details

			0			1	.		0	.	.	1	.	.	.
.	.	.	2	.	.	6	.	.	5	.	.	4	.	.	.
.	.	.	9	.	.	1	.	.	7	.	.	1	.	.	.
			1			8	1		2	8	1	6	2	8	1

Arithmetic cont.

At each step only three digits are dealt with:

carry digit,	range	$[0 \ldots 1]$
addend digit,	range	$[0 \ldots 9]$
augend digit,	range	$[0 \ldots 9]$
sum,	possible range	$[0 \ldots 19]$

Required knowledge:

All combinations $(a + b)$, where a and b are in the range $[0 \ldots 9]$.

Binary Addition

Done like its decimal counterpart.

$$0101 + 0110 = ?$$

in detail:

$$\begin{array}{|cccc|cccc|cccc|cccc|}
 & & & 0 & & & 0 & . & & 0 & . & . & 1 & . & . & . \\
. & . & . & 1 & . & . & 0 & . & . & 1 & . & . & 0 & . & . & . \\
. & . & . & 0 & . & . & 1 & . & . & 1 & . & . & 0 & . & . & . \\
\hline
 & & & 1 & & & 1 & 1 & & 0 & 1 & 1 & 1 & 0 & 0 & 1
\end{array}$$

The binary number combination rules are much simpler than decimal:

$$\begin{aligned}
0 + 0 &= 0 \\
1 + 0 &= 1 \\
1 + 1 &= 10
\end{aligned}$$

The multiplication table is simple too:

$$\begin{aligned}
0 \times 0 &= 0 \\
0 \times 1 &= 0 \\
1 \times 1 &= 1
\end{aligned}$$

Finite Length Registers

Real computers have registers of fixed length n, so all numbers must have exactly n digits.

Suppose $n = 8$. The range of natural numbers that can be represented is

00000000	to	11111111	(binary)
000	to	377	(octal)
000	to	255	(decimal)
00	to	FF	(hexadecimal)

Addition may possibly overflow:

```
 10111001   or    B9
 11000100   or    C4
 ————————         ——
 01111101   or    7D
not
101111101   or   17D
```

because there is no place to store a ninth digit!

Finite Length Registers cont.

Similarly, counting (repeated addition of 1) runs to 11111111 or FF, then *wraps around* to 00000000, like an automobile odometer.

More Finite-Length Addition:

```
 01001000
 10111000
 ————————
 00000000
```

The answer is zero, *modulo 256*, or *to 8 bits*.

It would be different in a 9-bit machine (modulo 512):

```
001001000
010111000
—————————-
100000000
```

Proposition:

To every string S1 if N bits there corresponds exactly one other string S2 of the same length such that the sum of the two strings (evaluated to N bits) is zero.

S2 is said to be the *twos complement of S1.*

Twos Complement

This proposition can be proved by construction. Invert every bit in S1 to form another string S0. Their sum is necessarily a string 11111 . . . 11111. For example,

$$\begin{array}{c} 01001000 \\ 10110111 \\ \hline 11111111 \end{array}$$

S0 is unique: no other string will yield 111 . . . 1 as a sum.

Proof: rightmost digits must be inverses, otherwise 1 cannor result. If there are inverses, there can be no carry. If there is no carry, the same argument applies to the next digit and so on.

Twos Complement cont.

If 00000001 is added to 11111111, zero results:

$$\begin{array}{r} 11111111 \\ 00000001 \\ \hline 00000000 \end{array}$$

Therefore, S2 can always be constructed by adding 00000001 to S0.

S0 is unique, so S2 is unique.

Twos Complement cont.

A rule of ordinary arithmetic:

$$S1 + S2 = 0 \quad \text{implies} \quad S2 = -S1.$$

Adopt the same rule for arithmetic *modulo K*:

$$S1 + S2 = 0 \quad \text{modulo K}$$

implies

$$S2 = -S1 \qquad \text{modulo K.}$$

Then

01001000	means	+72	(modulo 256)
10111000	”	-72	”
————————		———	
00000000		0	

Twos complementation:

1. Form logical complement by inverting each bit.

2. Add 1.

Complemented Arithmetic

Complemented arithmetic works in any base including decimal, if word length is fixed.

Example: 3-digit word length.

To 3 digits,

	499	
	<u>626</u>	
	125	
so the	symbol	626
stands for the	value	-374

Tens complementation works just like twos complementation:

the number	374
has the 9's complement	625
add 1	1
tens complement of number	626

Using complemented notations, all arithmetic can be done without ever inventing the minus sign!

Twos complement numbers

Two different interpretations of any binary symbol are available:

Binary Symbol	Decimal natural	Decimal twos cp.
00000000	000	000
00000001	001	001
00000010	002	002
...	...	...
01111110	126	126
01111111	127	127
10000000	128	-128
10000001	129	-127
10000010	130	-126
...	...	...
11111110	254	-002
11111111	255	-001

Twos complement numbers cont.

The *symbol* does not change, only its *interpretation* does.

1. There is only one unique zero.

2. Leftmost bit gives away the sign.

3. Range is slightly asymmetric, because zero looks positive.

4. Numbers wrap around, smallest always follows largest.

Binary Subtraction

Totally unnecessary as a seperate operation. (Form negative of number, then add). Example:

$$010110 - 001001 = ?$$

Subtrahend:	001001	
Its complement:	110110	(1)
Add 1:	<u>000001</u>	
Negative of subtrahend:	110111	(2)
Minuend:	010110	
Negative of subtrahend:	<u>110111</u>	
(add!) Difference:	001101	(3)

To prove the answer correct, add difference to subtrahend:

$$\begin{array}{r} 001001 \\ \underline{001101} \\ 010110 \end{array}$$

Carries and Overflows

Twos complement arithmetic works *because* it is length-limited (not in spite of this limitation)!

A carry out of the high-order bit does not mean the answer is wrong:

$$\begin{array}{rr} 0001 & 1 \\ \underline{0010} & \underline{2} \\ 0011 & 3 \end{array} \qquad \begin{array}{rr} 1111 & -1 \\ \underline{1110} & \underline{-2} \\ 1\ 1101 & -3 \end{array}$$

Carry from bit 3 on right; none on left. Both are correct.

$$\begin{array}{rr} 0110 & 6 \\ \underline{0100} & \underline{4} \\ 1010 & -6 \end{array} \qquad \begin{array}{rr} 1010 & -6 \\ \underline{1100} & \underline{-4} \\ 1\ 0110 & +6 \end{array}$$

Carry from bit 3 on right; none on left. Both are wrong.

Carries and Overflows cont.

Answers are wrong if the available number range is overflowed. Carries have nothing to do with it!

Suppose the addition

$$A + B = C$$

overflows the admissible number range.

Can A and B have opposite signs?

If they do,

$$|C| < max(|A|, |B|),$$

but if A and B are both within range, then C must be also!

Carries and Overflows cont.

Overflow can only occur if A and B have like signs.

Overflow always produces the wrong sign.

Overflow

An overflow is known to have occurred if

1. both operands have the same sign,

2. and the result has a different sign.

Multiplication

Multiplication is similar in all number bases:

1. Write down the multiplier and the multiplicand,

2. multiply entire multiplicand by each multiplier digit in turn,

3. add the partial results.

```
 000 000 010 111   *      000 000 011 001
------------------------------------------
                          000 000 011 001
                        0 000 000 110 01
                       00 000 001 100 1
                    0 000 000 110 01
------------------------------------------
                            1 000 111 111
```

Multiplication cont.

Computers cannot handle *blanks*; fill in trailing zeros where necessary:

$$
\begin{array}{lcr}
000\ 000\ 010\ 111 & * & 000\ 000\ 011\ 001 \\
\hline
 & & 000\ 000\ 011\ 001 \\
 & & 0\ 000\ 000\ 110\ 01\underline{0} \\
 & & 00\ 000\ 001\ 100\ 1\underline{00} \\
 & & 0\ 000\ 000\ 110\ 01\underline{0\ 000} \\
\hline
 & & 1\ 000\ 111\ 111
\end{array}
$$

Multiplication Technique

Every digit in a binary number is either 1 or 0, so every step in multiplication requires *either* adding *or* not adding.

To multiply M times N,

```
1.     Set I = M;  set P = 0.
2.     For j = 0, 1, 2, ..., n-1 do the following:
     3.     If digit j of N is 1 then
                   set P = P + I;
            Else do nothing;
     4.     Shift I left one place.
```

```
This process requires the ability to
1. add,
2. shift left.
```

In a *left shift*, digits migrate left;

```
     most significant bit is lost,
     least significant bit is set to 0.
```

Negative Numbers

Multiplication is really repeated addition.

Addition of negative twos complement numbers works, so multiplication works too.

00011 * 11011	+3 * -5
11011	-15
1 1011	
10001	works!

Two negative numbers:

1101 * 1110	-3 * -2
1110	+6
11 10	
111 0	works!
0110	

Division is done similarly – it requires *subtraction* and *shifting right*.

Binary Division

$$Dividend = Quotient \times Divisor + \text{Re}mainder$$

$$= \left(\sum_{i=0}^{n-1} q_i 2^i \times Divisor \right) + \text{Re}mainder$$

$$= q_{n-1} 2^{n-1} \times Divisor + ... + q_1 2 \times Divisor + q_0 Divisor + \text{Re}mainder$$

Algorithm

```
Remainder = Dividend

For i = n-1 to 0
    if  Remainder – 2^i x Divisor  ≥ 0
        q_i = 1
        Remainder = Remainder – 2^i x Divisor  ≥ 0
    else
        q_i = 0
```

A slightly more efficient method

```
Remainder = Dividend
D = 2^(n-1) x Divisor              ← left shifting

For i = n-1 to 0
    if  Remainder – D  ≥ 0
        q_i = 1
        Remainder = Remainder – D
    else
        q_i = 0
    D = D/2                        ← right shift
```

Conclusion:

Binary division can be implemented using only 3 operators (-, <<, >>).

Practical Implementation

Binary division operates symmetrically to multiplication, the difference being that one SUBTRACTS instead of adds and shifts RIGHT to divide by 2 instead of multiplying by 2.

The algorithm is best illustrated by example:

Divide 01101101 by 00010101

Assume a fixed length register of 8-bits and 2’s complement number representation. The divisor and dividend are both assumed to be positive integers.

```
void div8(char dividend, char divisor,
          char *quotient, char *rmdr) {
  int shifts, i;
```

Step 0: Initialization

```
  *rmdr = dividend;
  *quotient = 0;
   shifts = 0;
```

Step 1: Normalization

Shift the divisor to the left until the leftmost 1 is just to the right of the sign bit. Count the number of shifts.

```
     while ((divisor & 0x40) != 0x40) {
        divisor = divisor << 1;
        shifts++;
     }
```

For this example, the number of shifts would be 2, meaning that we would need to perform 3 subtractions to complete the process.

n.b. shifting left is equivalent to *multiplying* by 2; shifting right is equivalent to *dividing* by 2.

For our example:

01101101	dividend	(109)
00010101	divisor	(21)
00101010	first shift	
01010100	second shift	

Step 2: Subtract and Shift Loop:

```
for (i=0; i<=shifts; i++) {
   if (*rmdr-divisor >= 0) {
     *rmdr-= divisor;
     *quotient+=1;
   }
  divisor=divisor>>1;
  if (i != shifts) *quotient=*quotient<<1;
}
```

Observations

- If the subtraction is positive, the quotient has a 1 in the current bit position.
- The quotient string is built incrementally by adding the result for the current but position to the and of the quotient, and shifting left.

For our example:

`01101101`	divisor < dividend
`01010100`	subtract
`--------`	
`00011001`	quotient = 1

shift divisor and quotient for next iteration:

`00101010`	divisor
`00000010`	quotient

end of iteration 1

`00011001`	divisor > dividend
`00101010`	do not subtract
`--------`	
`00011001`	quotient = 10

shift divisor and quotient for next iteration:

```
00010101        divisor
100             quotient
```

end of iteration 2

final iteration:

```
00011001        divisor < dividend
00010101        subtract
--------
00000100        quotient = 101
```

shift divisor but not quotient on last iteration:

```
00001010        divisor
00000101        quotient
00000100        remainder
```

sanity check: quotient x initial divisor + remainder

5 x 21 + 4 = 109 → correct result.

Obviously this is not a general implementation, but it does outline the salient points.

Shifting Left and Right

Shifting left is equivalent to multiplication by the base – by 2 in binary notation, by 10 in decimal.

Shifting right is equivalent to division by the base.

	00101	(decimal	5)
left:	01010	(decimal	10)
right:	00101	(decimal	5)

	11010	(decimal	-6)
left:	10100	(decimal	-12)
rigit:	01010	(decimal	+10)

Shifting right destroys sign. Adopt the convention that *right shifts leave most significant bit unaltered*. Then

	11010	(decimal	-6)
left:	10100	(decimal	-12)
rigit:	11010	(decimal	-6)
rigit:	11101	(decimal	-3)

Shifting Left and Right cont.

Arithmetic shift right (ASR) leaves the high-order bit unchanged.

Logical shift right (SHR) places 0 in the high order bit.

Thus far, to support the operations of Addition, Subtraction, Multiplication, and Division, we require the following repitoire if basic operations:

ADD	addition modulo N
INV	complement (invert)
SHL	shift left
SHR	shift right
ASR	arithmetic shift right

Sign and Magnitude

An alternative approach to signed numbers; use one bit for sign, remaining bits for magnitude.

Example: 8-bits

+/-	1	0	1	1	0	0	1

Magnitude is now limited to half the natural number range, which makes the signed number range

-1111111	to	+1111111	(binary)
-127	to	+127	(decimal)

This is called *sign* and *magnitude* representation. It is more complicated than twos complement notation:

1. There are two distinct zeros, +0 and -0.
2. Subtraction (or addition with different signs) is a distinct operation.

Real Number Representation

Real numbers are encoded using IEEE 754 format. The number is first normalized to 1 before the binary point as shown below. This works for all numbers except 0, which must be handled as a special case.

$$+1.0110011 \times 2^1$$

0	10000000	01100110000000000000000
31	$< 30 - 23 >$	$< 22 - 0 >$

The IEEE encoding is shown above. Observe that there are 3 distinct fields:

1. The sign (bit-31) is set to 0 for positive numbers and 1 for negative.

2. The exponent (bits 23-30) is stored in excess-127 notation: as an unsigned integer with 127 added. This value is often referred to as a *bias*.

Real Number Representation cont.

3. The mantissa is stored in bits 0-22. It is not necessary to explicitly store the leading 1.

We will refer to this interpretation of a binary string S as *ieee(S)*. Some observations:

- With 24 bits available for the mantissa, the maximum precision is 7-8 significant figures, base 10.

- The largest exponent is 2^{127} (1.70×10^{38}) and the smallest 2^{-126} (1.17×10^{-38}).

- The addition of a bias shifts the representation of the exponent from $[-126, 127]$ to $[1, 254]$.

- Exponent field values 0 and 255 signal special cases.

Real Number Representation cont.

Special Cases (signaling)

	Mant=0	Mant=0x7FFFFF
Exp = 0	0	Denornalized
Exp = 255	∞	NAN

These are the principal cases. For more detail see the reference document on the 221 Home Page.

Floating Point Example

$$
\begin{aligned}
-265.73_{10} &= -100001001.1011101_2 \\
&= -1.000010011011101_2 \times 2^8
\end{aligned}
$$

The floating point representation is determined as follows:

$$
\begin{array}{ll}
\text{Sign} & 1 \\
\text{Exponent} & \;\;10000111 \\
\text{Mantissa} & \underline{\hphantom{110000111}000010011011101} \\
 & 11000011100001001101110\,1
\end{array}
$$

Sign: 1 negative, 0 positive

Exponent: to 8 bits,

$$
\begin{aligned}
8_{10} &= 00001000 \\
127_{10} &= \underline{01111111} \\
&\ 10000111
\end{aligned}
$$

Mantissa: first 23 bits from binary point. If $<$ 23 bits, pad with zeros.

ieee(S) ranges

Symbol S	Mantissa	Exp.	Real Number
7FFFFFFF	7FFFFF	80	NaN
7F7FFFFF	7FFFFF	7F	3.4028235 E+38
7F7FFFFE	7FFFFE	7F	3.4028233 E+38
7F600000	600000	7F	2.9774707 E+38
41C00000	400000	04	2.4000000 E+01
41400000	400000	03	1.2000000 E+01
40C00003	400003	02	6.0000014 E+00
40C00002	400002	02	6.0000010 E+00
40C00001	400001	02	6.0000005 E+00
40C00000	400000	02	6.0000000 E+00
00800001	000001	-7E	1.1754945 E-38
00800000	000000	-7E	1.1754944 E-38
00000000	000000	00	0.0000000 E+00

ieee(S) ranges cont.

Symbol S	Mantissa	Exp.	Real Number
80000000	000000	00	0.0000000 E+00
80800000	000000	-7E	-1.1754944 E-38
80800001	000001	-7E	-1.1754945 E-38
C0C00000	400000	02	-6.0000000 E+00
C0C00001	400001	02	-6.0000005 E+00
C0C00002	400002	02	-6.0000010 E+00
C0C00003	400003	02	-6.0000014 E+00
C1400000	400000	03	-1.2000000 E+01
C1C00000	400000	04	-2.4000000 E+01
FF600000	600000	7F	-2.9774707 E+38
FF7FFFFE	7FFFFE	7F	-3.4028233 E+38
FF7FFFFF	7FFFFF	7F	-3.4028235 E+38
FFFFFFFF	7FFFFF	80	NaN

Double Precision

An extension to 64-bits using two 32-bit words as follows.

63	62———52	51——————————32
sign	11-bit exp.	upper 20 mant.

31—————————————————————0
lower 32-bits of mantissa

Exponent bias = 1023. Values 0 and 2047 reserved for signaling.

Same conventions as single precision with respect to zero and other special cases.

IEEE 754 Summary

	Formula	Single	Double
P	$\log(2^{1+N_m})$	7 digit	15 digits
R	$\pm 2^{2^{N_e-1}}$	$\pm 3.4 \times 10^{38}$	$\pm 1.7 \times 10^{308}$
S	$\pm 2^{-(2^{N_e-1}-2)}$	$\pm 1.18 \times 10^{-38}$	$\pm 2.2 \times 10^{-308}$
B	$2^{N_e-1} - 1$	127	1023

where P - precision, R - range, S - smallest normalized value greater or less than 0, B - bias, N_e - number of bits in exponent field, and N_m - number of bits in mantissa field.

Smallest (0) and largest (2^{N_e-1}) values of exponent field are reserved for signaling.

Isomorphism

This form of encoding makes the real ("floating-point") positive numbers *isomorphic* to the integers.

Any given 32-bit word W has at least 3 possible interpretations (so far):

unsigned integer	int(W)
twos-complement integer	int2(W)
floating-point number	ieee(W)

These are defined so that

$$\begin{array}{rl} \text{IF} & int(W_1) > int(W_2) \\ \text{THEN} & ieee(W_1) > ieee(W_2) \end{array}$$

and conversely, if ieee(W_k) $\geq$ 0.0

Precision

34.671 is given to 5 significant digits; fractional part is given to 3.

To represent 0.671 to the same precision requires *10* bits in base-2.

Why?

$$671_{10} = 1010011111_2$$

In General:

$$\begin{aligned} \text{Let } n_{10} &= \text{given number in base-10.} \\ d_b &= \text{\# significant digits in base-b.} \end{aligned}$$

Then it follows that

$$\begin{aligned} n_{10} &= b^{d_b} \\ \log(n_{10}) &= d_b \log(b) \\ d_b &= \frac{\log(n_{10})}{\log(b)} \end{aligned}$$

Precision cont.

Converting from some arbitrary base a to some other base b is slightly less convenient since a base-a log function is not readily available. We can still obtain a reasonable approximation as follows:

$$\begin{aligned} \text{Let } d_a &= \#\text{ significant digits in base-a.} \\ d_b &= \#\text{ significant digits in base-b.} \end{aligned}$$

Then it follows that

$$\begin{aligned} a^{d_a} &= b^{d_b} \\ d_a \log(a) &= d_b \log(b) \\ d_b &= d_a \frac{\log(a)}{\log(b)} \end{aligned}$$

Precision cont.

Example 34.671

$0.671_{10} \rightarrow 3$ significant digits in base-10. How many in base-2?

$$
\begin{aligned}
d_b &= \frac{\log(671)}{\log(b)} = \frac{\log(671)}{\log(2)} \\
&= \frac{2.827}{0.3011} = 9.39
\end{aligned}
$$

or 10 when rounded to the next highest integer.

n.b. we use 671 for a in the above expression, not 0.671.

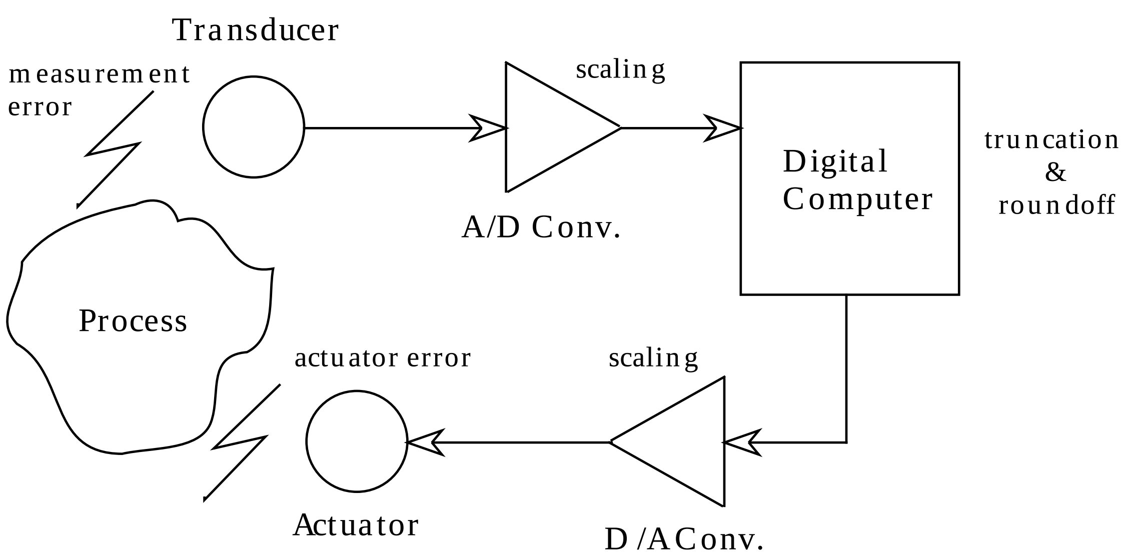
Representational Error

3 Sources

-**Measurement**

-**Scaling**

-**Truncation and Roundoff**

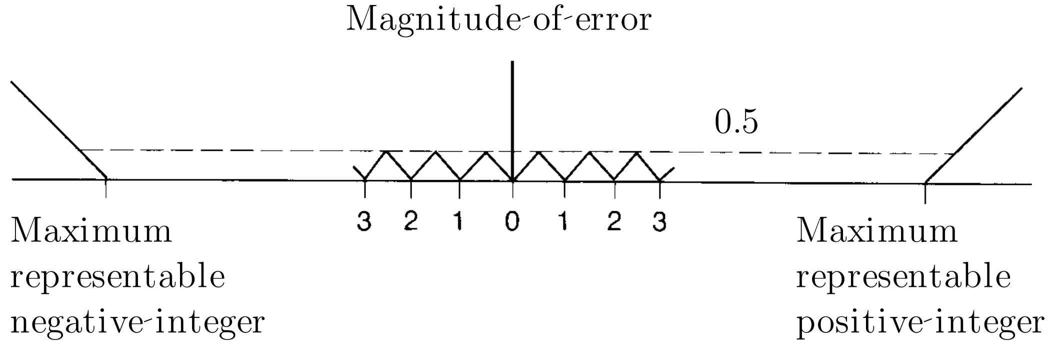


$$\text{Resolution} = \frac{\text{Dynamic Range of Signal}}{2^n}$$

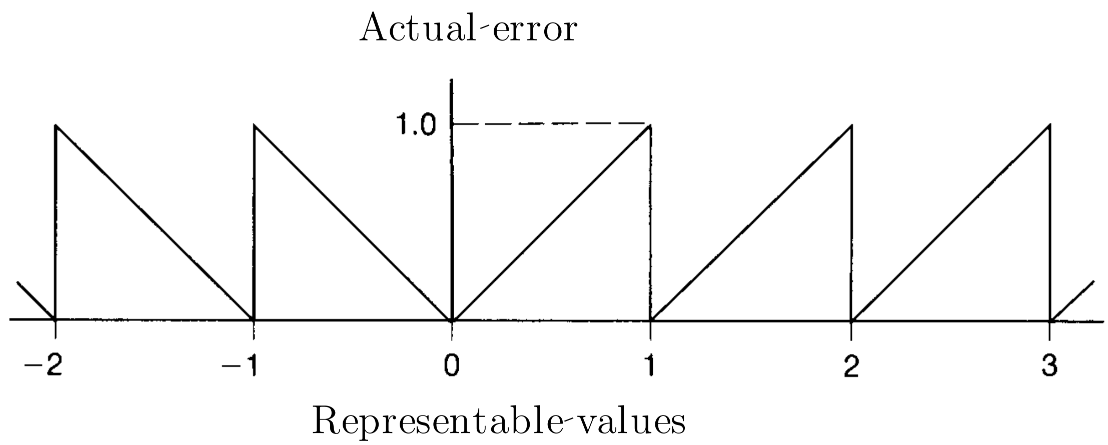
Representational Error - Integers

For numbers within the representational range, i.e. $[-2^{n-1}, 2^{n-1}-1]$, the error characteristic is constant across the range.

Case 1: Rounding to the nearest digit

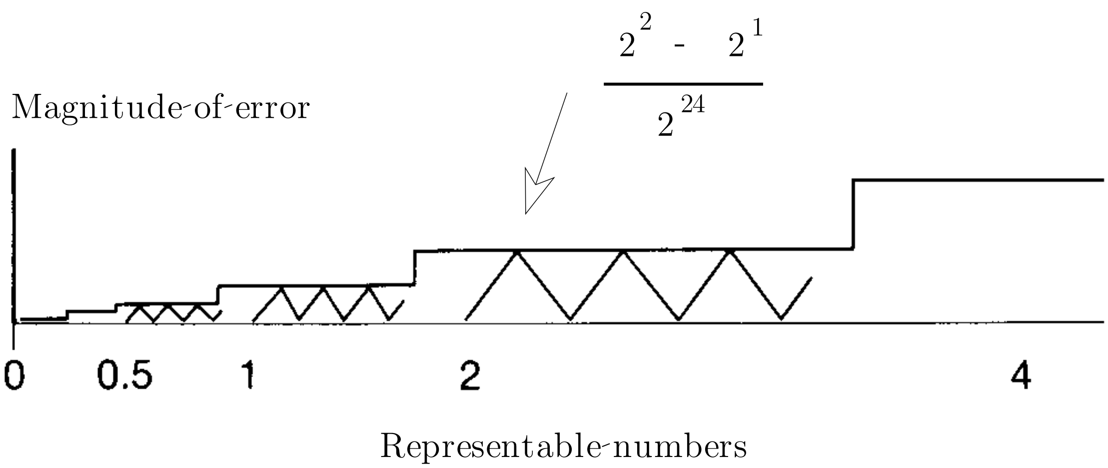


Case 2: Truncation



Representational Error - Floats

Floating point representations sample the real line in intervals determined by the exponent.



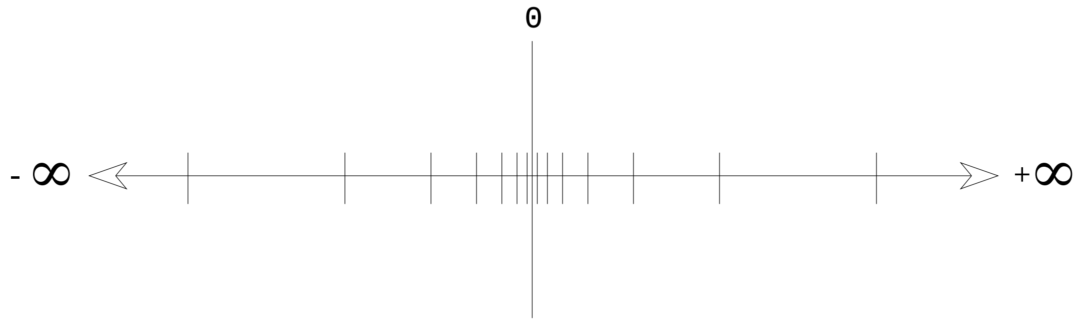
The magnitude of this error is given by:

$$\text{Rep. Error} = \frac{\text{length of interval}}{\text{quantization}}$$

$$= \frac{\text{radix}^n - \text{radix}^{n-1}}{\text{radix}^{(\#digits\ in\ mantissa)}}$$

Rounding

Necessary because of quantization.



$$\pm\infty = \pm\text{radix}^{\text{radix}^{(\#\text{exponent digits}-1)}}$$

Example: radix=2, # exponent bits = 8.

$$\pm\infty = \pm 2^{2^7-1} = \pm 2^{127} = \pm 1.7 \times 10^{38}$$

Consider $10110.101001 = 0.10110101001 \times 2^5$.

If the mantissa bas 5 bits then we can approximate by:

```
-taking the first 5 digits to       TRUNCATION
 the right of ., i.e., 0.10110

-rounding to the nearest digit,     ROUNDING
 i.e., 0.10111.
```

Rounding cont.

$$\text{With base-10} \quad \begin{array}{rcl} 192634 & . & 51834 \\ & . & 5 \\ \hline 192635 & . & 01835 \end{array}$$

This is refered has *half adjusting*, i.e., rounding to the nearest integer.

Half-adjusting is fine most of the time, but is problematic at the center, i.e., 0.5, introducing a bias.

Statistical Rounding gets around this. If the digit is at the center of range, then 50% of the time round up, else round down.

Since the distribution of ODD & EVEN numbers is about equal, then rounding to make the result ODD or EVEN will achieve the desired effect.

Statistical rounding is more "accurate".

Rounding cont.

What about rounding binary numbers?

$$10110.101001$$

Half adjusting rule says if digit to be rounded is 1, then round UP else if 0 round DOWN.

Statistical rounding rule says 50% of the time round UP and the other 50% round down (i.e. round to make the result ODD or EVEN depending on the convention chosen).

Rounding cont.

Observations:

- Half adjusting and truncation introduce about the same error for binary numbers.

- Statistical rounding is a better choice, but is more complex to implement (parity circuit).

- Better (simpler) to add additional bits and truncate.

Binary Coded Decimal

Another representation, often used in commercial data processing.

The bit string is chopped into groups of 4 bits, one for each decimal digit.

Example: 3728

0011	0111	0010	1000
3	7	2	8

Good:	in exact accord with manual calculations.
Bad:	arithmetic rules are complicated (slow!).

Non-Numeric Data (Text)

The basic entities are printable characters. Requirements:

Capital letters ABC...	26	
Small letters abcd...	26	
Numerals 0123456789	10	
Punctuation marks	15	(or more)
Space	1	
Total	78	(or more)

There are	64	different 6-bit symbols,
	128	different 7-bit symbols,
	256	different 8-bit symbols.

Non-Numeric Data (Text) cont.

At least 7 bits must be used to make up a useful set of characters.

Generally people reserve a word of 8-bits, use the lease significant 7 for the character.

Bit 7 is reset, or used as an error check (parity) bit.

A 6-bit character set is sometimes used -- it has capital letters only.

ASCII encoding

The most common coding scheme is the ASCII (*American standard for computer information interchange*) character set. In recent years ASCII has been superceeded by ISO standards, but it is still widely used.

ASCII Character Set

Binary	Octal	Hex	Char
0100000	040	20	space
0100001	041	21	!
0100010	042	22	”
	...		
0101111	057	2F	/
0110000	060	30	0
0110001	061	31	1
	...		
0111001	071	39	9
0111010	071	3A	:
0111011	072	3B	;

ASCII encoding cont.

ASCII Character Set

Binary	Octal	Hex	Char
1000000	100	40	@
1000001	101	41	A
1000010	102	42	B
	...		
1100001	141	61	a
1100010	142	62	b
1100011	143	63	b
	...		
1111010	172	7A	z
1111011	173	7B	{
	...		

Control Characters

ASCII characters 0000000 to 0011111 are non-printable. Most of them control communication or printing.

Binary	Octal	Hex	Char
0000000	000	00	NUL
	...		
0000100	004	04	EOT
	...		
0000110	006	06	ACK
0000111	007	07	BEL
	...		
0001011	011	09	HT
0001010	012	0A	NL
0001011	013	0B	VT
0001100	014	0C	FF
0001101	015	0D	CR
	...		

HT (hor. tab), NL (line feed), VT (ver. tab), FF (form feed), CR (carriage return).

Text Encoding

Text encoding is done on a character by character basis:

F	r	e	d
01000110	01110010	01100101	01100100
46	72	65	64

It is often useful to *pack* several ASCII characters in a single machine register. Consider a machine with a 64-bit register length.

Upper 32 bits

A	S	C	I
41	53	43	49
01000001	01010011	01000011	01001001

Lower 32 bits

I	NUL	NUL	NUL
49	00	00	00
01001001	00000000	000000000	00000000

NUL (00) used to pad text.

Alphabetic Ordering

Let the ASCII bit strings corresponding to characters be interpreted as numbers, e.g., "ASCII" $= 4153434949_{16}$. Let int(X) = integer value of bit string X, and ascii(X) = character value. Then

$$\begin{array}{rl} \text{if} & int(S_1) > int(S_2) \\ \text{then} & ascii(S_1) > ascii(S_2) \end{array}$$

Ascending order of numerical values corresponds to ascending alphabetic order (isomorphism). Hence *alphabetic sorting is equivalent to numeric sorting*.

Alphabetic Sorting

Register length has a direct impact on the speed of textual sorting as can be seen in the following example.

Sorting with register length = 8

Initial			Pass 1			Pass 2		
IN	49	4E	AT	41	54	AT	41	54
AT	41	54	IN	49	4E	IF	49	46
IF	49	46	IF	49	46	IN	49	4E
NO	4E	4F	NO	4E	4F	NO	4E	4F

Sorting with register length = 16

Initial		Pass 1	
IN	494E	AT	4154
AT	4154	IF	4946
IF	4946	IN	494E
NO	4E4F	NO	4E4F

Observe how larger registers reduce the number of sorting passes required.

Case Changes

In the ASCII character set

01000001	A	01100001	a
01000010	B	01100010	b
01000011	C	01100011	c
	...		
01011010	Z	01111010	z

Altering case is accomplished by changing bit 5 (single character operation). We need a mechanism to unconditionally *set* or *clear* specified bits.

This can be done arithmetically, but a comparison would be required to determine whether to add or subtract a constant.

Case Changes cont.

If shift operations are augmented by a *rotate* operator, then it becomes possible to set and clear any register bit, e.g., convert “a” to “A”.

01100001	a
11000010	rotate left
10000101	rotate left
00001010	shift left
01000001	rotate right 3 times

Logical Operators

A more efficient approach to bit manipulation is to include logical AND and OR operators. Recall that

X	Y	$X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	$X + Y$
0	0	0
0	1	1
1	0	1
1	1	1

Bit 5 can be unconditionally cleared by ANDing with the constant $5F$, and unconditionally set by ORing with 20_{16}..

Example, a = 01100001, A = 01000001

$$\begin{array}{cccc} 01100001 & \cdot & 01011111 & 01000001 \\ 01000001 & + & 00100000 & 01100001 \end{array}$$

Quasi-Numeric Data

Some textual data looks numeric, but isn't. Example – a personnel record:

Name	Payroll number	Soc. Insc. number	Birth date
BLOGGS, J.Q.	72536	525367021	022546
not numeric data			numeric or text

Such data are usually stored as characters, not as numeric values, because any manipulation to be done will be of a text-editing rather than arithmetic type.

Isomorphism with Integers

In the previous example, the date field of the personnel record was stored as day-month-year. This is chronologically bad for *ordering*. A better scheme is

year-month-day,

because historical sequences of dates are now isomorphic with integers.

If S_n is some bit string,

$$int(S_1) > int(S_2)$$

if and only if

$$date(S_1) > date(S_2)$$

In other words, it is advantageous to design data structures so that they are isomorphic with integers.

Text Masking

Logical AND and OR were used earlier to set or clear bit 5 in an ASCII representation in order to change case.

These operations can also be used to screen out portions of a data structure for the purpose of *pattern matching*.

Consider the following 8 byte representation for a date record (ASCII):

1	7	5	6	J	A	2	7
31	37	35	36	4A	41	32	37

Suppose you had to write a program to find all date records containing JA. Solution: extract the month field in each record and compare against JA.

Text Masking cont.

Using a logical product (AND)

$$\begin{aligned}&31\ 37\ 35\ 36\ 4A\ 41\ 32\ 37 \cdot 00\ 00\ 00\ 00\ FF\ FF\ 00\ 00\\ &= 00\ 00\ 00\ 00\ 4A\ 41\ 00\ 00\end{aligned}$$

Using a logical sum (OR)

$$\begin{aligned}&31\ 37\ 35\ 36\ 4A\ 41\ 32\ 37 + FF\ FF\ FF\ FF\ 00\ 00\ FF\ FF\\ &= FF\ FF\ FF\ FF\ 4A\ 41\ FF\ FF\end{aligned}$$

Text Masking cont.

Procedure to detect records containing JA:

1. AND with mask 00000000FFFF0000, or OR with mask FFFFFFFF0000FFFF.

2. Subtract 4A410000 from the result if AND or FFFFFFFF4A41FFFF if OR.

3. Result = 0 in both cases if record contains JA.

Operator Summary

To perform useful computation, a general purpose computer will usually contain at least the following operations on data (Modulo-N):

Operator	Operation
ADD	binary addition
CMP	bitwise complement
SHL	shift left
SHR	shift right
ASR	arith. shift right
ROL	rotate left
ROR	rotate right
AND	bitwise and
OR	bitwise or