Section B: Digital Information

Representing numbers (cont.)

Adding 1-bit numbers

p	q	$p+q$
1	1	1 0
1	0	0 1
0	1	0 1
0	0	0 0

Adding 1-bit numbers

p	q	$p+q$	
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p+q$	
		LB	RB
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Adding 1-bit numbers

p	q	$p+q$	
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p+q$	
		LB	RB
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

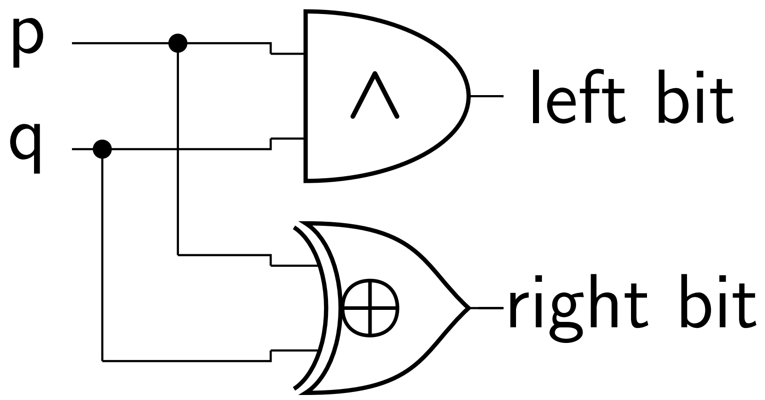
p	q	$p+q$	
		$p \wedge q$	$p \oplus q$
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Adding 1-bit numbers

p	q	$p+q$	
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p+q$	
		LB	RB
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

p	q	$p+q$	
		$p \wedge q$	$p \oplus q$
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0



This circuit is called a **half adder** (2 bits in, 2 bits out)

Adding 1-bit numbers

p	q	r	$p+q+r$	
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

Adding 1-bit numbers

p	q	r	$p+q+r$	
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

p	q	r	$p+q+r$	
			LB	RB
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

CHALLENGE: Construct a circuit diagram for a circuit which implements the above (3 bits in, 2 bits out). We shall call such a circuit a **full adder**.

Meaningful renaming of bits in a full adder

p	q	carry	carry	sum
		in	out	out
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

4-bit addition via a cascade of full adders

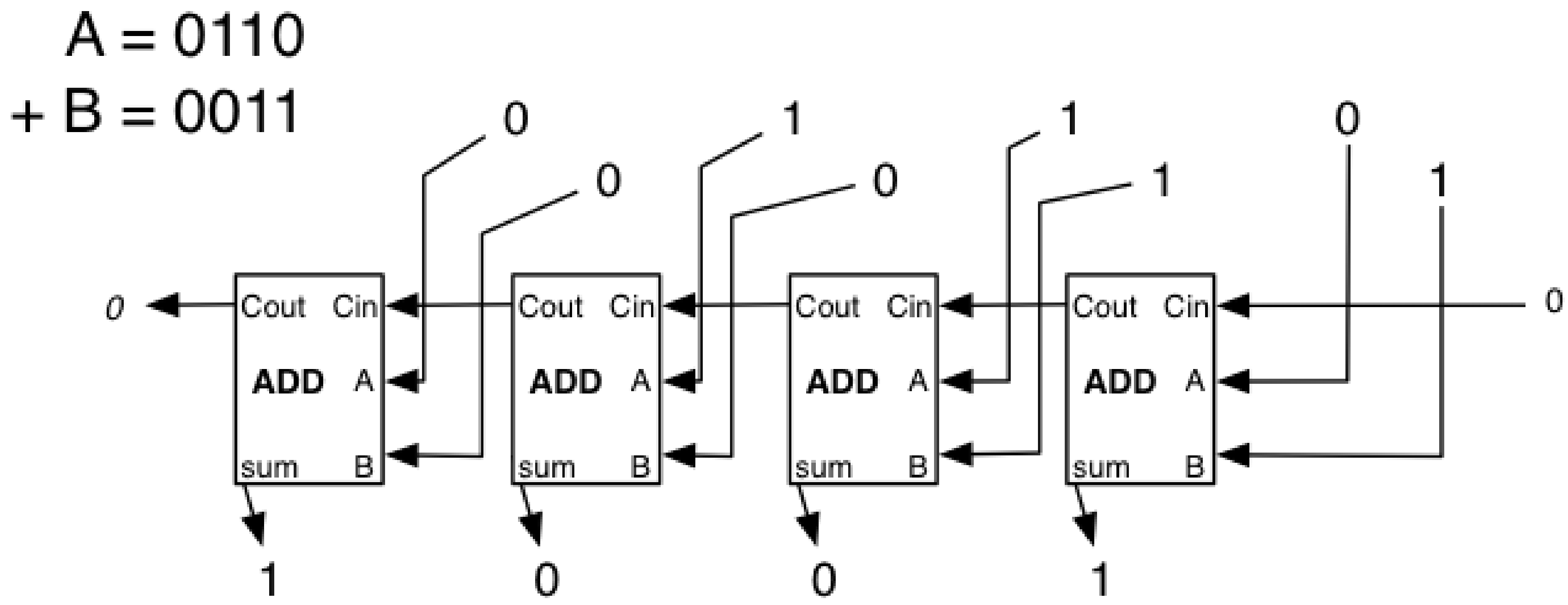


Diagram extracted from "Concepts in Computing" course notes at Dartmouth College, Hanover, New Hampshire, USA, 2009.

http://www.cs.dartmouth.edu/~cbk/classes/4/notes/19.php

Circuits for integer subtraction

Since $x - y$ is the same as $x + (-y)$, we can accomplish integer subtraction provided we can convert y to $-y$ (because we already know how to add).

Our clever way of representing negative numbers means

- the integer representation of an integer y can be converted to the integer representation of $-y$ by toggling all bits and adding one.
- A NOT gate will toggle a single bit, and we already know how to build a circuit that adds, so we can build a circuit that adds one.

Something amazingly neat

Our clever way of representing negative numbers means

- the usual algorithm (the step-by-step process) that works for adding two non-negative integers, works for adding two negative integers and for adding a non-negative integer and a negative integer
- the carry bit from the most significant place in an addition can be ignored.
- this will work provided that the sum is in the range of integers that can be represented by the number of bits we have chosen.

Examples

4-bit examples:

$$\begin{array}{r} 4 \\ -6 \\ \hline -2 \end{array} \qquad \begin{array}{r} 0\,1\,0\,0 \\ +\,1\,0\,1\,0 \\ \hline 1\,1\,1\,0 \end{array} \qquad\qquad \begin{array}{r} 7 \\ -3 \\ \hline 4 \end{array} \qquad \begin{array}{r} 0\,1\,1\,1 \\ +\,1\,1\,0\,1 \\ \hline (1)\,0\,1\,0\,0 \end{array}$$

The carry bit in () is ignored.

8-bit example:

$$\begin{array}{r} 81 \\ -98 \\ \hline -17 \end{array} \qquad \begin{array}{r} 0\,1\,0\,1\,0\,0\,0\,1 \\ +\,1\,0\,0\,1\,1\,1\,1\,0 \\ \hline 1\,1\,1\,0\,1\,1\,1\,1 \end{array} \qquad \left[\begin{array}{l} \quad 81 = \ \ 64+16+1 \ \ \to 0101\,0001 \\ \quad 98 = \ \ 64+32+2 \ \ \to 0110\,0010 \\ -98 \to 1001\,1101 + 1 \to 1001\,1110 \\ \hline 1110\,1111 \to -(0001\,0000+1) \\ \qquad\qquad \to -0001\,0001 \to -17 \end{array}\right]$$

Multiplication

Example:

$$
\begin{array}{cccccccc}
 & & & & 1 & 0 & 1 & 1 \\
 & & & \times & 1 & 1 & 0 & 0 \\
\cline{5-8}
 & & & & 0 & 0 & 0 & 0 \\
 & & + & 0 & 0 & 0 & 0 & 0 \\
 & + & 1 & 0 & 1 & 1 & 0 & 0 \\
+ & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
{\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & &
\end{array}
$$

Multiplication

Example:

$$
\begin{array}{cccccccc}
 & & & & 1 & 0 & 1 & 1 \\
 & & & \times & 1 & 1 & 0 & 0 \\
\cline{5-8}
 & & & & 0 & 0 & 0 & 0 \\
 & & + & 0 & 0 & 0 & 0 & 0 \\
 & + & 1 & 0 & 1 & 1 & 0 & 0 \\
+ & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
{\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & &
\end{array}
$$

Notice that no ‘multiplication tables’ are required since the only multiplications used are ‘times 0’ which results in all zeroes, and ‘times 1’ which has no effect.

Multiplication

Example:

$$\begin{array}{ccccccccc}
 & & & & & 1 & 0 & 1 & 1 \\
 & & & & \times & 1 & 1 & 0 & 0 \\
\cline{6-9}
 & & & & & 0 & 0 & 0 & 0 \\
 & & & + & 0 & 0 & 0 & 0 & 0 \\
 & & + & 1 & 0 & 1 & 1 & 0 & 0 \\
+ & & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & &
\end{array}$$

Notice that no ‘multiplication tables’ are required since the only multiplications used are ‘times 0’ which results in all zeroes, and ‘times 1’ which has no effect.

This is one of the bonuses of using binary in computers.

Multiplication by shifts and additions

For computer implementation of multiplication (within $\mathbb{N}$) we only need a ‘shift’ circuit (to move bits left and append a zero on the right) and addition circuits.

But we also need an *algorithm* to control the process.

An **algorithm** is a sequence of operations on an input. The result is called an output.

Examples: Addition, subtraction, multiplication, division.

Example

Here is very simple list processing algorithm, set out using a standard format for displaying algorithms:

Input: L: a list of students.
Output: O: list of students who have submitted their assignment.
Method:

```
Set $j = 1$, $O =$ empty list. (Initialization phase)
Loop:   If $j =$ length$(L) + 1$, stop.
        If $L[j]$ (the j-th element in $L$) has
            submitted the assignment, append $L[j]$ to $O$.
        Replace $j$ by $j + 1$.
Repeat loop.
```

Algorithm for binary addition

This algorithm formalizes the method described earlier, which is implemented with a cascade of full adders.

Contrary to notation used in some earlier examples, in this context it is usual to *number the bits starting from the rightmost bit, which is called the 0-th bit*.
So the leftmost bit of an n-bit number is its $(n-1)$-th bit.

Example: For the number 01101001 we have

0	1	1	0	1	0	0	1
$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$	$\uparrow$
7-th bit	6-th bit	5-th bit	4-th bit	3-th bit	2-th bit	1-th bit	0-th bit

Algorithm for binary addition

Input: Two numbers p, q in base 2 with n bits.
Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.
Method:

```
Set $j = 0$, $c = 0$, and the $n$-th bits of $p$ and $q$ to $0$.
Loop:   If $j = n + 1$ stop.
        Add the j-th bits of $p$ and $q$ plus $c$.
        Store the result part as the j-th bit of $s$.
        Store the carry part as $c$ (replacing the old value of $c$).
        Replace $j$ by $j + 1$.
Repeat loop
```

Example: Add $p = 0110$ and $q = 1111$ with $n = 4$.

Input: Two numbers p, q in base 2 with n bits.
Output: The sum $s = p + q$ in base 2 with $n + 1$ bits.
Method:

```
Set $j = 0$, $c = 0$, and the $n$-th bits of $p$ and $q$ to $0$.
Loop:  If $j = n + 1$ stop.
       Add the j-th bits of $p$ and $q$ plus $c$.
       Store the result part as the j-th bit of $s$.
       Store the carry part as $c$ (replacing the old value of $c$).
       Replace $j$ by $j + 1$.
Repeat loop
```

Algorithm for binary multiplication

This algorithm formalises the method described earlier, except that addition is done progressively rather than at the end.

The additions may be done by the algorithm of the previous slide.

This algorithm also assumes the existence of a shift algorithm, which shifts all the bits of a number one place to the left and then fills the vacant rightmost place with a 0.

Algorithm for binary multiplication

Input: Two numbers x, y in base 2 with n bits.
Output: The product $p = x \times y$ in base 2 with $2n$ bits.
Method:

```
Set $j = 0$ and $p = 0$.
Loop:   If $j = n$ stop.
        If the $j$-th bit of $y$ is 1 then replace $p$ by $p + x$.
        Shift $x$.
        Replace $j$ by $j + 1$.
Repeat loop.
```

Example: Multiply $x = 1111$ by $y = 0110$ with $n = 4$.

Input: Two numbers x, y in base 2 with n bits.
Output: The product $p = x \times y$ in base 2 with $2n$ bits.
Method:

```
Set $j = 0$ and $p = 0$.
Loop:   If $j = n$ stop.
        If the $j$-th bit of $y$ is 1 then replace $p$ by $p + x$.
        Shift $x$.
        Replace $j$ by $j + 1$.
Repeat loop.
```

Rational numbers

What is a rational number?

Recall that $\mathbb{Q}$ is the set of **rational numbers**. A rational number is a number that can be represented as the ratio of two integers.

EXAMPLE $\frac{2}{3}$ is a rational number.

Please note that every integer is a rational number as, for example $6 = \frac{6}{1}$.

What is a rational number?

Recall that $\mathbb{Q}$ is the set of **rational numbers**. A rational number is a number that can be represented as the ratio of two integers.

EXAMPLE $\frac{2}{3}$ is a rational number.

Please note that every integer is a rational number as, for example $6 = \frac{6}{1}$.

Are you happy with this definition?

What is a rational number?

Let $Q = \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

What does an element of Q look like?

What is a rational number?

Let $Q = \mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

What does an element of Q look like?

The set Q may be partitioned so that any elements (n_1, d_1) and (n_2, d_2) of Q are in the same partition set if and only if $n_1 d_2 = n_2 d_1$.

So, for example, $\{(2,3), (-2,-3), (4,6), (-4,-6), (6,9), (-6,-9), \dots\}$ is one of the sets in the partition

The sets in the partition may themselves be considered rational numbers. We usually write $\frac{2}{3}$ instead of
$\{(2,3), (-2,-3), (4,6), (-4,-6), (6,9), (-6,-9), \dots\}$.

Representing rationals in a computer

For computer storage of any **non-zero** rational number q we need to express it using **scientific notation**. For any base b this is

$$\boxed{q = (-1)^s \times m \times b^n}$$

where

- $q \in \mathbb{Q}$, $q \neq 0$;
- $b \in \mathbb{Z}$, $b \geq 2$;
- $s \in \{0, 1\}$, (s is the **sign bit**)
- $m \in \mathbb{Q}$, $1 \leq m < b$, (m is the '**mantissa**') and
- $n \in \mathbb{Z}$ (n is the **exponent**).

Given q and b, the values of of s, m and n are uniquely determined by these conditions.

Example in base 10: $13.5 = (-1)^0 \times 1.35 \times 10^1$.

Examples

Example in base 10: $23.5 = (-1)^0 \times 2.35 \times 10^1$.

To save space, we store 235 instead of 2.35 because 2.35 is the only number between 1 and 10 with digits $1, 3, 5$.

Examples

Example in base 10: $23.5 = (-1)^0 \times 2.35 \times 10^1$.

To save space, we store 235 instead of 2.35 because 2.35 is the only number between 1 and 10 with digits $1, 3, 5$.

Examples in base 10:

- $-154 = (-1)^1 \times 1.54 \times 10^2$. Store m as 154.
- $0.031 = (-1)^0 \times 3.1 \times 10^{-2}$. Store m as 31.

The number of digits used to store m is called the **precision**.

IEEE half-precision

The shortest IEEE standard for representing rational numbers is called *half-precision floating point*. It uses a 16-bit word partitioned as in the diagram at right.

To store a rational number x it is first represented as $(-1)^s \times m \times 2^n$ with $1 \leq m < 2$. The sign bit s is stored as the left-most bit (bit 15), the mantissa m (called “significand” in IEEE parlance) is stored in the right-most 10 bits (bits 9 to 0), and the exponent n is stored in the 5 bits in between (bits 14 to 10). However...

IEEE half-precision

- *Only the fractional part of m is stored.* Because $1 \leq m < 2$, the binary representation of m **always** has the form $1 \cdot \star\star\star\ldots$ where the stars stand for binary digits representing the fractional part of m. Hence there is no need to store the $1\cdot$ part.
- *the exponent n is stored with an "offset".* In order to allow for both positive and negative exponents, but to avoid another sign bit, the value stored is $n + 15$. In principle this means that the five exponent bits can store exponents in the range $-15 \leq n \leq 16$, but 00000 and 11111 are reserved for special purposes so in fact n is restricted to the range $-14 \leq n \leq 15$.

An example

A rational number x is stored in half-precision floating point as the word 5555_{16}. Which number is being represented?

An example

A rational number x is stored in half-precision floating point as the word 5555_{16}. Which number is being represented?

$$(5555)_{16} = \underbrace{(0101)}_{(5)}\underbrace{(0101)}_{(5)}\underbrace{(0101)}_{(5)}\underbrace{(0101)}_{(5)} =$$

$$\begin{array}{r|c|l} (0) & (10101) & (0101010101) \\ s & n+15 & \text{frac.part of } m \\ & = 21 & \end{array}$$

$$\begin{aligned} \longrightarrow\ (-1)^0 \times 1.0101010101 \times 2^{21-15} & \\ &= 1.0101010101 \times 2^6 \\ &= 1010101.0101 \qquad \text{(moving the binary point 6 places right)} \\ &= 64+16+4+1+\frac{1}{4}+\frac{1}{16} = \boxed{85\tfrac{5}{16} = 85.3125_{10}}. \end{aligned}$$

A warning

WARNING: With limits on precision and exponent size, some rational numbers can only be stored inaccurately, if at all.

Of course, the same sort of thing is true for integers. But with integers we can represent ALL of the integers close enough to 0, so it is easier to understand which integers we can and cannot represent.

If you have a reason to need to represent rational numbers with an accuracy beyond the accuracy provided by some sort of standard set up, you can write dedicated software to represent numbers with greater precision.