MATH1005/MATH6005: Discrete Mathematical Models

Adam Piggott

Semester 1, 2021

Section B: Digital Information

Representing numbers (cont.)

Addition and subtraction in decimal

In base 10:

$123 + 678$:

$$\begin{array}{rccc} & 1 & 2 & 3 \\ + & 6 & 7 & 8 \\ \hline & 8 & 0 & 1 \\ \hline \end{array}$$

$123 - 78$:

$$\begin{array}{rccc} & 1 & 2 & 3 \\ - & & 7 & 8 \\ \hline & & 4 & 5 \\ \hline \end{array}$$

Addition and subtraction in binary

In base 2:

$$111_2 + 10_2 : \quad \begin{array}{cccc} & 1 & 1 & 1 \\ + & & 1 & 0 \\ \hline 1 & 0 & 0 & 1 \end{array}$$

$$1010_2 - 111_2 : \quad \begin{array}{cccc} & 1 & 0 & 1 & 0 \\ - & & 1 & 1 & 1 \\ \hline & 0 & 0 & 1 & 1 \end{array}$$

Addition and subtraction in hexadecimal

In base 16:

$$456_{16} + ABC_{16}: \quad \begin{array}{rccc} & 4 & 5 & 6 \\ + & A & B & C \\ \hline & F & 1 & 2 \\ \hline \end{array}$$

$$F12_{16} - ABC_{16}: \quad \begin{array}{rccc} & F & 1 & 2 \\ - & A & B & C \\ \hline & 4 & 5 & 6 \\ \hline \end{array}$$

Representing positives, negatives and zero

When writing positive and negative integers in binary, we may use a - or + in front of the integer to represent its sign.

We may write $+(1101)_2$ for $(13)_{10}$
and $-(1101)_2$ for $-(13)_{10}$.

How can we, in a computer, represent integers in a range that includes negative integers?

Representing positives, negatives and zero in a computer

How can we, in a computer, represent integers in to a range that includes negative integers?

FIRST IDEA: Use an extra **sign bit**. Fix a number of bits to use for each integer, and the left-most bit is 0 if the number is positive, and 1 is the number is negative.

In such a scheme with say, a byte to represent an integer:
$(00001101)_2$ would represent $(13)_{10}$
$(10001101)_2$ would represent $(-13)_{10}$
and $(00000000)_2$ and $(10000000)_2$ would both represent zero.

This is **NOT** how integers are usually represented.

Representing positives, negatives and zero in a computer

How can we, in a computer, represent integers in to a range that includes negative integers?

BETTER IDEA: Use an extra **sign bit**, but the binary representation of a negative is chosen according to a scheme called two's complement...

Representing positives, negatives and zero in a computer

We fix a number of bits, say t, to use for each integer. A string of t bits $d_1 d_2 \ldots d_t$ is interpreted as a number in one of two ways, depending on the value of d_1:

- if $d_1 = 0$, then the bit string $(0 d_2 d_3 \ldots d_t)$ represents $(d_2 d_3 \ldots d_t)_2$
- if $d_1 = 1$, then the bit string $(1 d_2 d_3 \ldots d_t)$ represents $(d_2 d_3 \ldots d_t)_2 - 2^{t-1}$, which is equivalent to $-[(e_2 e_3 \ldots e_t)_2 + 001_2]$ where

$$e_i = \begin{cases} 1, \text{if } d_i = 0 \\ 0, \text{if } d_i = 1. \end{cases}$$

The process of evaluation $(1 d_2 d_3 \ldots d_t)_2$ is often described as: 'toggle' all bits, add one, then negate.

Example: 4-bit signed integers

When we use nibbles to represent integers, then $t = 4$ and we have:

nibble	decimal value	nibble	decimal value
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

In this case, we say we are using **4-bit signed integers**.

Example: 8-bit signed integers

When we use bytes to represent integers, then $t = 8$ and we have:

nibble	decimal value	nibble	decimal value
00000000	0	10000000	-128
00000001	1	10000001	-127
00000010	2	10000010	-126
⋮	⋮	⋮	⋮
01111110	126	11111110	-2
01111111	127	11111111	-1

In this case, we say we are using **8-bit signed integers**.

NOTE: To go from the bit string representing an integer y to the bit string representing $-y$, simply toggle all bits, add one (and throw away any carry digit from the most significant place after the addition).

Adding 1-bit numbers p and q

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 and q

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 and q

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

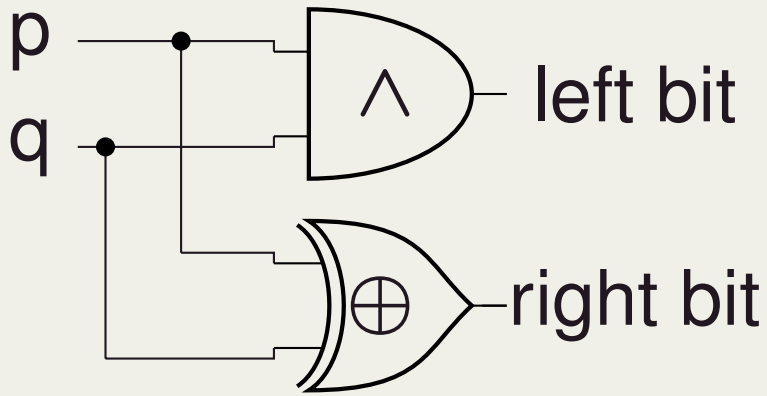
p	q	$p+q$	
		$p \wedge q$	$p \oplus q$
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

Adding 1-bit numbers p and q

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 and r

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 and r

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 in	carry out	sum 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

Example: 4-bit addition via a cascade of full adders

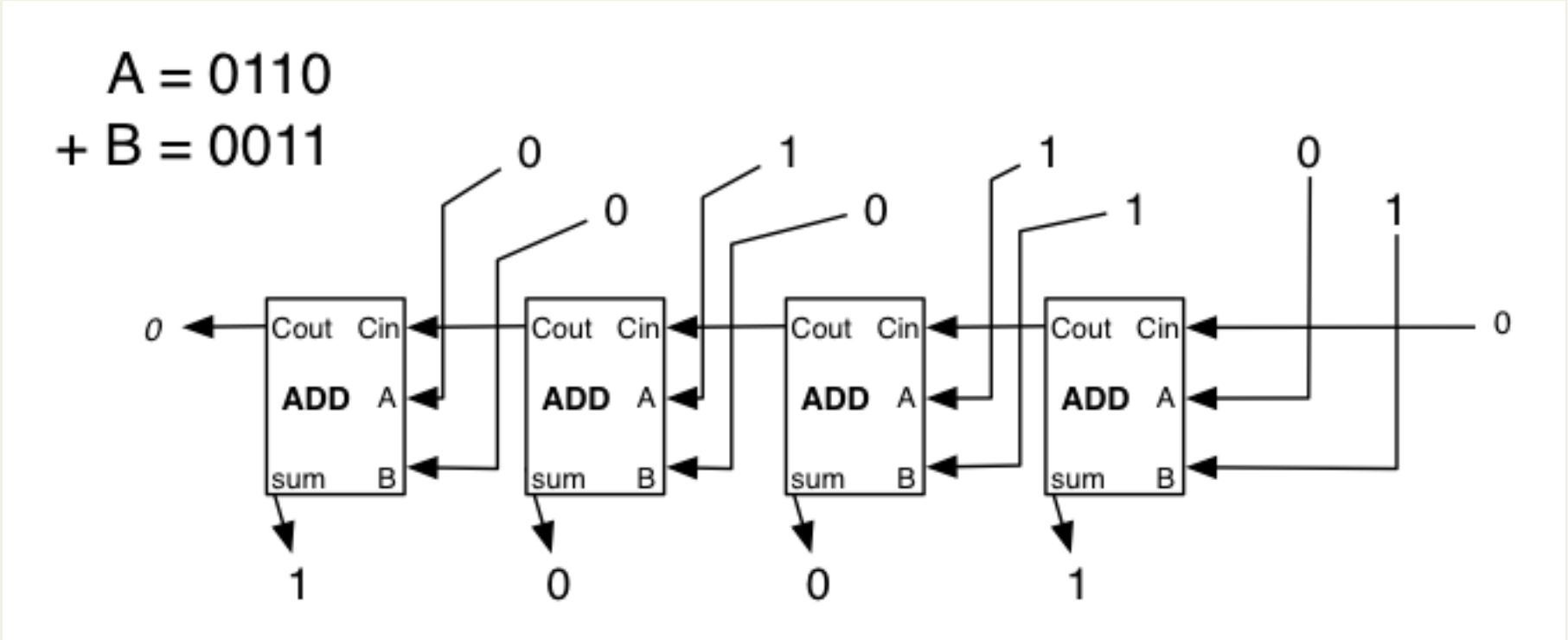


Diagram extracted from “Concepts in Computing” course notes at Dartmouth College, Hanover, New Hampshire, USA, 2009.

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:

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

The carry bit in () is ignored.

8-bit example:

$$\begin{array}{rr} 81 & 0\,1\,0\,1\,0\,0\,0\,1 \\ -98 & +\,1\,0\,0\,1\,1\,1\,1\,0 \\ \hline -17 & 1\,1\,1\,0\,1\,1\,1\,1 \end{array}$$

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

Multiplication

Example:

$$
\begin{array}{cccccccc}
 & & & & & 1 & 0 & 1 & 1 \\
 & & & & \times & 1 & 1 & 0 & 0 \\
\hline
 & & & & & 0 & 0 & 0 & 0 \\
 & & & + & 0 & 0 & 0 & 0 & 0 \\
 & & + & 1 & 0 & 1 & 1 & 0 & 0 \\
 & + & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \\
\hline
1 & 1 & 1 & 1 & & & & &
\end{array}
$$

Multiplication

Example:

$$
\begin{array}{cccccccc}
 & & & & 1 & 0 & 1 & 1 \\
 & & & \times & 1 & 1 & 0 & 0 \\
\hline
 & & & & 0 & 0 & 0 & 0 \\
 & & & + & 0 & 0 & 0 & 0 & 0 \\
 & & + & 1 & 0 & 1 & 1 & 0 & 0 \\
 & + & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
1 & 1 & 1 & 1 & & & & &
\end{array}
$$

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:

$$
\begin{array}{cccccccc}
 & & & & 1 & 0 & 1 & 1 \\
 & & & \times & 1 & 1 & 0 & 0 \\
\hline
 & & & & 0 & 0 & 0 & 0 \\
 & & & + & 0 & 0 & 0 & 0 & 0 \\
 & & + & 1 & 0 & 1 & 1 & 0 & 0 \\
 & + & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hline
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
1 & 1 & 1 & 1 & & & & &
\end{array}
$$

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 is accomplished 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.

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
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.