Computer Arithmetic (II)

- Combinatorial Multiplier
- Signed Multiplication
- Booth Multiplier

Combinatorial Multiplication

Consider the example of two, 2-digit numbers, A and B.

A has two digits, a_1 and a_0, hence $A = \{a_1a_0\}$; similarly $B = \{b_1b_0\}$.

Let us consider $A \times B$.

Now the notation $A = \{a_1a_0\}$ is really shorthand for:
$A = (a_1 0 + a_0)$ and $B = (b_1 0 + b_0)$

Therefore:

$$
\begin{aligned}
A \times B &= (a_1 0 + a_0) \times (b_1 0 + b_0) \\
&= (a_1 \times b_1)00 + (a_1 \times b_0 + a_0 \times b_1)0 + (a_0 \times b_0)
\end{aligned}
$$

left shift two places

left shift one place

$$A \times B = (a_1 \times b_1)00 + (a_1 \times b_0 + a_0 \times b_1)0 + (a_0 \times b_0)$$

(a1 × b1)00 the possible values can be 0(00) or 1(00) or 10(00) if there is carry bit to be propagated from the previous term .

possible carry

whole term brackets will produce either 0, 1 or 10 (a1×b0+a0×b1)0 will contribute either: 00, 10 or 100 to the overall product.

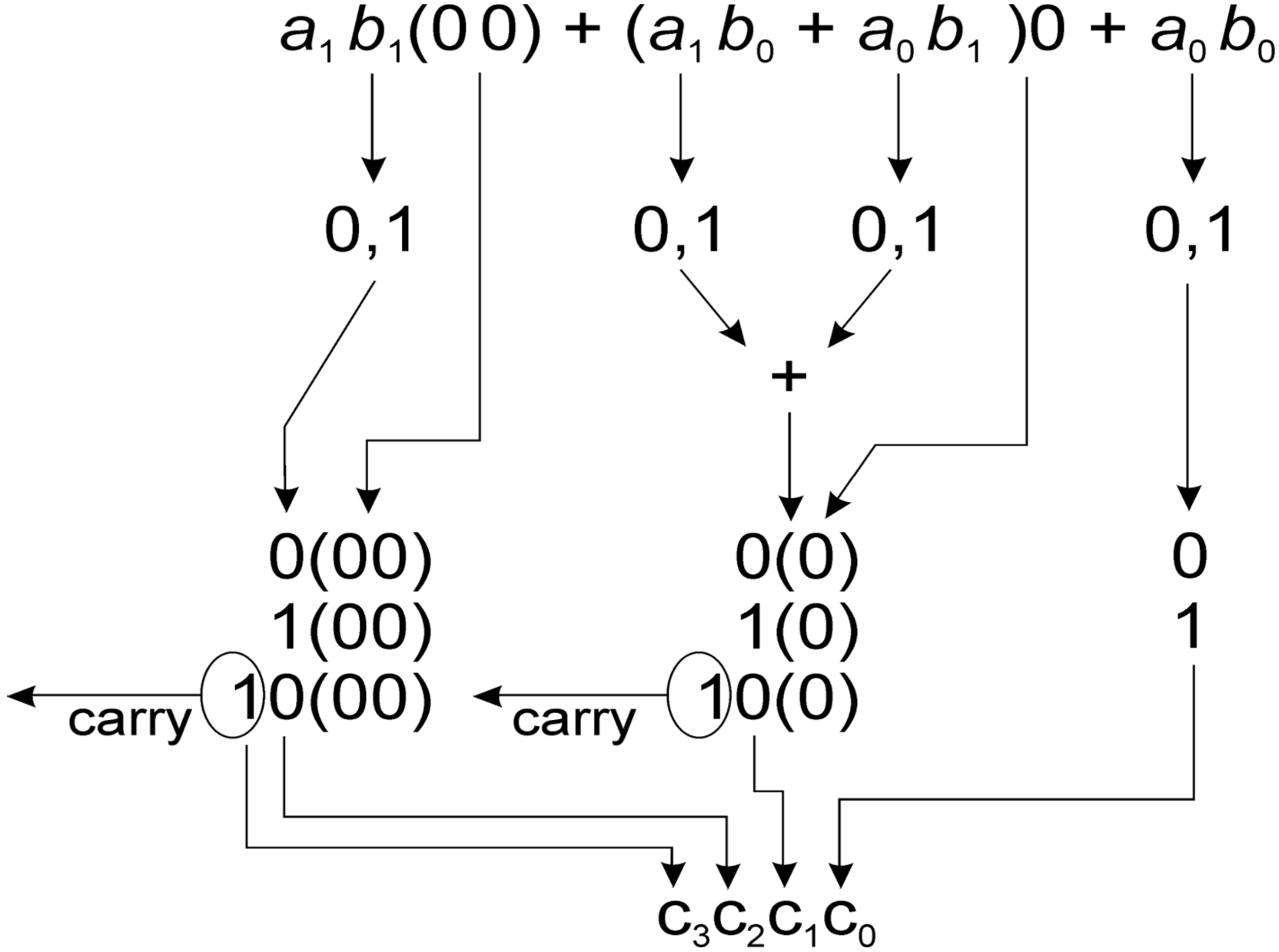
possible carry

LSB : unaffected by any of the other terms. product could be either 0 or 1

multiplication of 2 bits is achieved using an AND gate.

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

combinatorial multiplication for two, 2-bit numbers

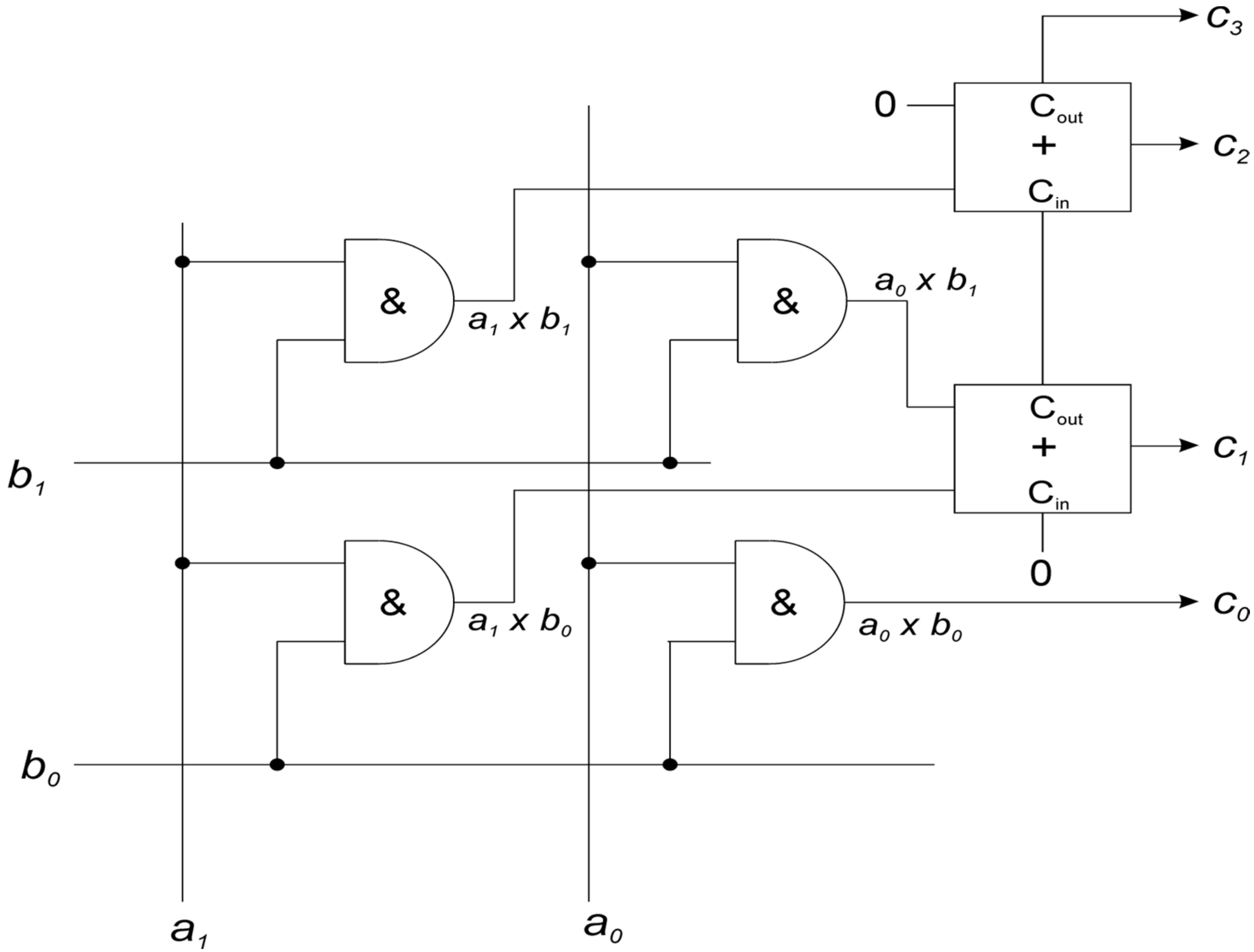


Multiplication Table

*	0	1
0	0	0
1	0	1

This is the same as the truth table for the logical AND function.
Remember, the AND function is also known as a product.

Combinatorial Multiplier



this can be extended for bigger numbers

For a 2's complement number, the sign bit can be taken as a negative value in the power series expansion.

$5 = 0101$

$-5 = 1011$

$-8 \quad +2 \quad +1$

Two's complement properties hold for fractional numbers as well:

$$(01.011)_{2\text{'s-compl}} = (-0\times2^{1}) + (1\times2^{0}) + (0\times2^{-1}) + (1\times2^{-2}) + (1\times2^{-3}) = +1.375$$

$$(11.011)_{2\text{'s-compl}} = (-1\times2^{1}) + (1\times2^{0}) + (0\times2^{-1}) + (1\times2^{-2}) + (1\times2^{-3}) = -0.625$$

Signed Multiplication

Multiplication needs adapting to cope with 2's complement numbers.

- The multiplicand is left-shifted and added
- This is consistent with two's complement *as long as the sign of the multiplicand is maintained*
- This condition can be met by sign-extending the multiplicand to the length of the product.

e.g. -7 x 3 = 1001 x 0011

Firstly, sign-extend -7 to 8 bits (the length of the product) i.e. 11111001 and then perform the multiplication

It does not work without sign extension to the width of the product because the weight of the most significant bit of a two's complement number is -2^{N-1} and not 2^{N-1} as in an unsigned number.

This can be demonstrated by looking at some examples (using N=8):

Weight	**-128**	**64**	**32**	**16**	**8**	**4**	**2**	**1**	**Value**	
	1	0	0	0	0	0	0	0	-128 + 0 =	-128
	1	1	1	1	1	1	1	1	-128 + 127 =	-1

$$
\begin{array}{rr}
 & 11111001 \\
\times & \underline{00000011} \\
 & 11111001 \\
 & 11110010 \\
 & 00000000 \\
 & \underline{00000000} \\
 & 11101011
\end{array}
$$

11101011= - (00010101) = -21 = -7 x 3

The multiplier is less straightforward.

Repeat the example with the multiplier and multiplicand swapped then the result should be the same:

```
          00000011
x             1001
          --------
          00000011
          00000000
          00000000
          00011000
          --------
          00011011
```

which is clearly the wrong answer

Repeat the example with the multiplier extended:

```
         00000011
x        11111001
         --------
         00000011
         00000000
         00000000
         00011000
         00110000
         01100000
         11000000
         10000000
         --------
         11101011
```

which is the right answer but requires more work to calculate

Booth's Algorithm

Multiplication trick

- When multiplying by 9:
  - Multiply by 10 (by shifting digits left one place)
  - Subtract once
- e.g.
  - 123454 x 9 = 123454 x (10 – 1) = 1234540 – 123454
  - Converts addition of several partial products to one shift and one subtraction

Booth's algorithm applies same principle

- Instead of decimal digits, in binary, just '1' and '0'

Booth noticed the following equality: For a string of 1s

$$2^{j} + 2^{j-1} + 2^{j-2} + \ldots + 2^{k} = 2^{j+1} - 2^{k}$$

e.g.
0111 = 1000 – 0001

0111111111111111 = 1000000000000000 - 0000000000000001

- This can be exploited to create a faster multiplier
- Sequence of N 1s in the multiplier yields sequence of N additions
- Replace with one addition and one subtraction
- Works for signed numbers

This equates to a set of rules for encoding the multiplier:

- **00**: middle of a run of 0s, do nothing
- **10**: beginning of a run of 1s, subtract multiplicand
- **11**: middle of a run of 1s, do nothing
- **01**: end of a run of 1s, add multiplicand

```
     43  =  00000101011
*    12  =  00000001100
-----------------------
      0  =  00000000000     // multiplier bits 0_ (implicit 0)
+     0  =  00000000000     // multiplier bits 00
-   172  =  11101010100     // multiplier bits 10
+     0  =  00000000000     // multiplier bits 11
+   688  =  01010110000     // multiplier bits 01
-----------------------
    516  =  01000000100
```

```
    43 = 00000101011
*   12 = 00000001100(0)
---------------------
     0 = 00000000000
+    0 = 00000000000    x2 shift
- 172 = 11101010100     x4 shift
+    0 = 00000000000    x8 shift
+  688 = 01010110000    x16 shift
---------------------
   516 = 01000000100
```

$$43 \times 12 = 43 \times (16 - 4)$$
$$= 688 - 172$$

$$43 \times 4 = 172$$

$$43 \times 16 = 688$$

43 = 00000101011
172 = 00010101100

-43 = 11111010101
-172 = 11101010100

- The multiplicand is exactly as in the unsigned operation.
- It is only the multiplier that is recoded
- The sub-product is added, subtracted, or ignored depending on the recoded bit of the multiplier.

Consider the case of 3 x -7 1001 recoded is -0+-

		00000011
		1001
00000011	-	11111101
00000110	+	00000110
00001100	0	00000000
00011000	-	11101000
		11101011

subtract

add

do nothing

12 = 0000001100

Consider the number 0001001100 as a multiplier (64 + 12 = 76)

Multiplying by 76 is the same as multiplying by 64 + 12

The 12 will be dealt with as previously by Booth. The algorithm works fine for other additions to the multiplier but what actually happens to this additional single '1' ?

0001001100

Add 128 x multiplicand

Subtract 64 x multiplicand

64 = 128 – 64

Booth Efficiency

Good for three or more consecutive 1's
    Replaces three or more adds with one add and one subtract

For two 1's there is no gain
    Replaces two adds with one add and one subtract

Worse for single 1's
    Replaces one add with one add and one subtract

Modified Booth solves this problem by examining multiplier bits in groups of three and performing a single add for the case 010.