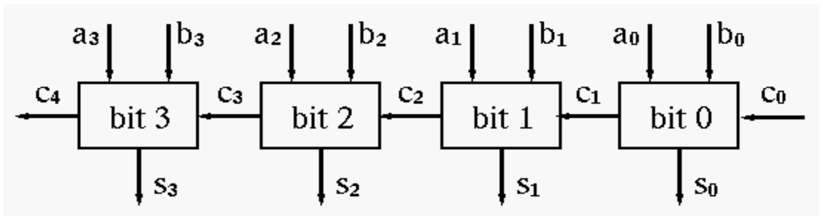
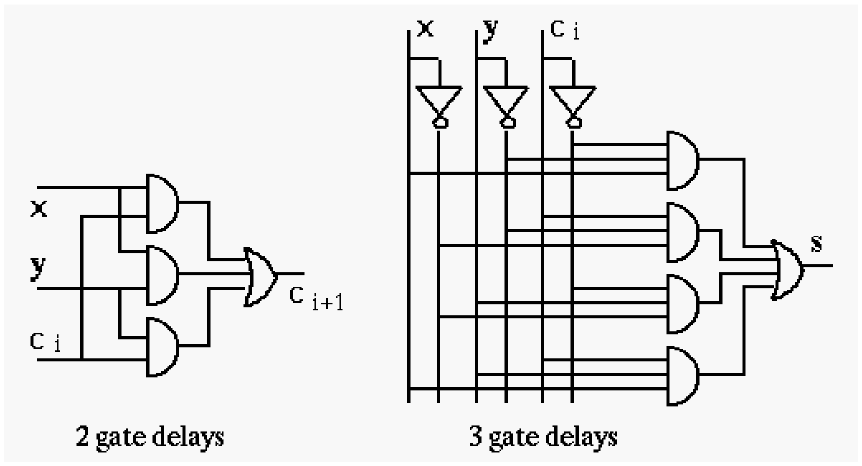
Fast Addition: Carry Lookahead

Ripple-carry Adder

The *n*-bit adder below is called a ripple-carry adder, as the carry c_i needs to be passed on through all lower bits to compute the sums for the higher bits.



Recall the logic operations in the i^{th} full adder:



$$\begin{cases} c_{i+1} = x_i y_i + x_i c_i + y_i c_i & 2\ \textit{gate delays} \\ s_i = x_i' y_i' c_i + x_i' y_i c_i' + x_i y_i' c_i' + x_i y_i c_i & 3\ \textit{gate delays} \end{cases}$$

The total time for computing the final *n*-bit sum from *X*, *Y* and c_0 is $2(n-1)+3$ gate delays. When n = 64, there will be 129 gate delays. How can we speed up this process?

Carry-lookahead Adder

The bottle neck for ripple carry addition is the calculation of c_i, which takes linear time proportional to *n*, the number of bits in the adder. To improve this, we define

- *Generate* function: $g_i = x_i \cdot y_i$

  If $g_i = 1$, the i^{th} bit generates a carry, $c_i = 1$.

- *Propagate* function: $p_i = x_i + y_i$

  If $p_i = 1$, the i^{th} bit propagates a carry c_i from the $(i-1)^{\text{th}}$ bit to the $(i+1)^{\text{th}}$ bit.

Both g_i and p_i can be generated for all *n* bits in constant time (1 gate delay).

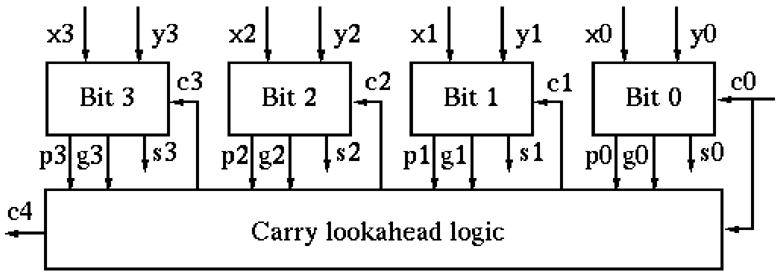
$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i = x_i y_i + c_i(x_i + y_i) = g_i + p_i c_i$$

Interpretation: c_{i+1} is either generated in the i^{th} bit ($g_i = 1$), or propagated from the $(i-1)^{\text{th}}$ bit ($c_i = 1$ and $p_i = 1$), or maybe both.

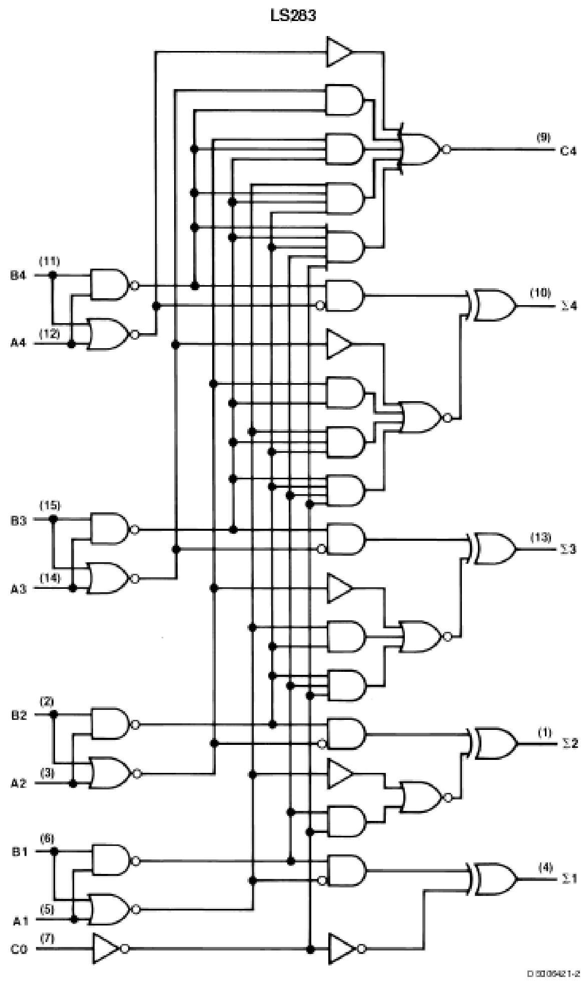
$$\begin{aligned}
c_{i+1} &= g_i + p_i c_i \\
&= g_i + p_i(g_{i-1} + p_{i-1}c_{i-1}) \\
&= g_i + p_i g_{i-1} + p_i p_{i-1}(g_{i-2} + p_{i-2}c_{i-2}) \\
&= \ldots\ldots\ldots \\
&= g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + p_i p_{i-1} p_{i-2} g_{i-3} + \ldots + p_i p_{i-1} \cdots p_1 p_0 c_0
\end{aligned}$$

Now all c_i, $i = 0, \ldots, n-1$ can be generated in a constant time (independent of n) of two more gate delays after g_i and p_i are available.

Interpretation: $c_{i+1} = 1$ if (a) at least one of the previous bits can generate a carry ($g_j = 1$, $j = 1, \ldots, i$ or $g_0 = c_0 = 1$), and (b) this carry can be propagated through all the bits to reach the $(i+1)^{\text{th}}$ bit ($p_i p_{i-1} \ldots g_i = 1$).



This is the logic diagram of the MSI chip 74x283 for a 4-bit adder:

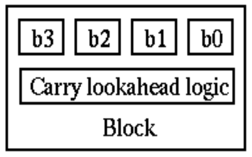


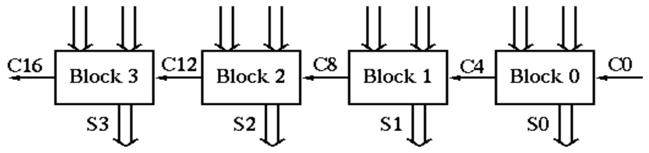
Operations	Number of Gate Delays
Generate g_i and p_i	1
Generate c_i	2
Generate s_i	3
Total	6

All carries c_n, ..., c_0 can be generated by the carry-lookahead logic in two gate delays after g_i and p_i are available, and all sum bits s_{n-1}, ..., s_0 can be made available in constant time of six gate delays, independent of the number of bits in the adder.

Two-level Carry Lookahead

The carry lookahead adder requires AND and OR gates with as many inputs as $n + 1$ (for c_n), which is impractical to realize in hardware. To compromise, we pack $n = 4$ bits as a block with carry lookahead, and still use ripple carry between the blocks (c_4, c_8, c_{12} and c_{16}).





There are $n/4$ blocks in an n-bit adder, and the total gate delays can be found as:

Operations	Number of Gate Delays
Generate g_i and p_i	1
Generate c_i (i=1,2,3,4) in block 1	2
Generate c_i (i=5,6,7,8) in block 2	2
......	......
Generate s_i	3
Total	$1+2(n/4)+3$

When n = 64, the number of gate delays is 36.
To improve the speed further using the same idea, define second-level generate and propagate functions:

- $P_i = p_{4i+3}p_{4i+2}p_{4i+1}p_{4i}$

  If all four bits in a block propagate, the block propagates a carry.

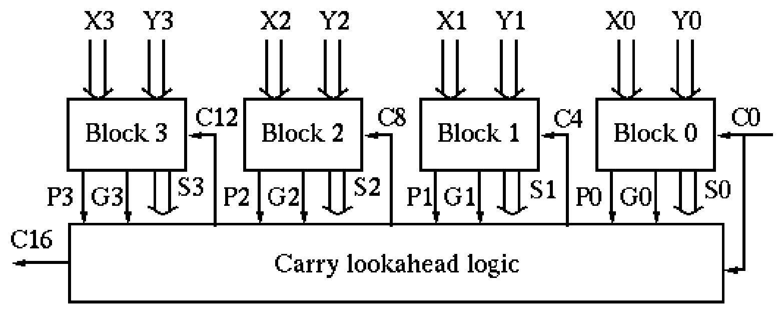
- $G_i = g_{4i+3} + p_{4i+3}g_{4i+2} + p_{4i+3}p_{4i+2}g_{4i+1} + p_{4i+3}p_{4i+2}p_{4i+1}g_{4i}$

  If at least one of the four bits generates carry and it can be propagated to the MSB, the block generates a carry.

Now c_4 can be generated in constant time (independent of n):

$$c_4 = (g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0) + (p_3p_2p_1p_0)c_0 = G_0 + P_0c_0$$

Similarly, $c_8 = G_1 + P_1c_4$, $c_{12} = G_2 + P_2c_8$ and $c_{16} = G_3 + P_3c_{12}$ can be generated in constant time.
Combining four blocks of 4-bit carry-lookahead adder as a super block, we get a 16-bit adder with two levels of carry-lookahead logic.



There are $n/16$ super blocks in an n-bit adder, and the total gate delays can be found as:

Operations	Number of Gate Delays
Generate all g_i and p_i	1
Generate all G_i and P_i $(i = 0, 1, 2, \cdots)$	2
Generate c_i $(i = 4, 8, 12, 16)$ in Block 0	2
Generate c_i $(i = 20, 24, 28, 32)$ in Block 1	2
......	......
Generate s_i	3
Total	$1+2+2(n/16)+3$

When n = 64, the number of gate delays is 14.

The very same idea can be carried out to a third level, with the carries c_{16}, c_{32}, c_{48} and c_{64} generated simultaneously by the third-level carry-look ahead logic:

$$\begin{aligned} c_{16} &= G_0 + P_0 c_0 \\ P_0 &= P_3 P_2 P_1 P_0 \\ G_0 &= G_4 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 \end{aligned}$$

Operations	Number of Gate Delays
Generate all g_i and p_i	1
Generate all G_i and P_i $(i = 0, 1, 2, \cdots)$	2
Generate all $\mathbf{G}_i$ and $\mathbf{P}_i$ $(i = 0, 1, 2, \cdots)$	2
Generate c_i $(i = 16, 32, 48, 64)$ in super block 0	2
……	……
Generate s_i	3
Total	$1 + 2 + 2 + 2(n/64) + 3$

When $n = 64$, the number of gate delays is 10.

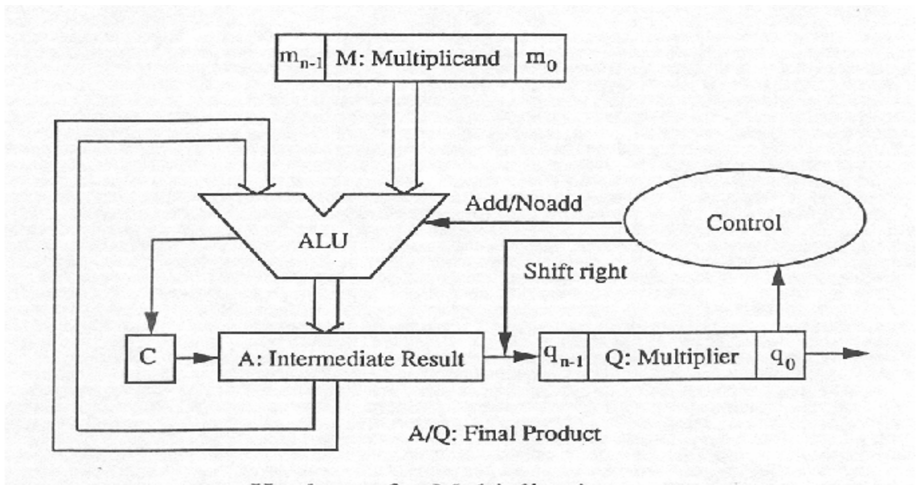
Multiplication

Example for multiplication: 13 × 11 = 143. The paper-and-pencil method:

```
              1  1  0  1
           x  1  0  1  1
--------------------------
              1  1  0  1
           1  1  0  1
        0  0  0  0
+    1  1  0  1
--------------------------
  1  0  0  0  1  1  1  1
```

Improve the method so that only two numbers are added each time:

```
              1  1  0  1
           x  1  0  1  1
--------------------------
              1  1  0  1
        +  1  1  0  1
--------------------------
        1  0  0  1  1  1
        0  0  0  0
--------------------------
        1  0  0  1  1  1
+    1  1  0  1
--------------------------
  1  0  0  0  1  1  1  1
```



Hardware for Multiplication

Algorithm for hardware multiplication

```
Do n times:
{
       if (Q0=0) then A?A+M;
       right shift A and Q by 1 bit
}
```

Note: When *A* and *Q* are right shifted, the MSB of *A* is filled with 0 and the LSB of *A* becomes the MSB of *Q*, and the LSB of *Q* is lost.

Continuing the example 13 × 11 = 143: Always use three registers *M*, *A*, and *Q*. Initially, the multiplicand is 13 = 1101_2 in *M*, the multiplier is 11 = 1011_2 in *Q* and *A* is zero.

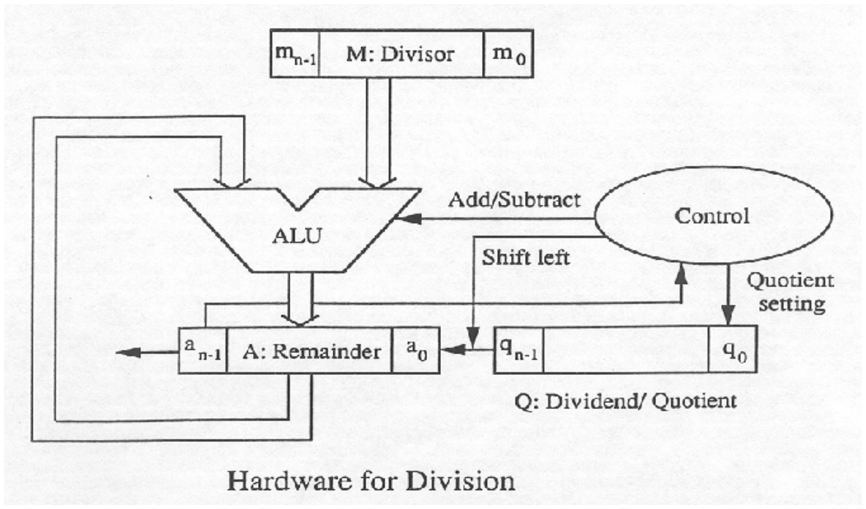
	[M]	1101		
	[A]	0000	[Q]	1011
q0=1, add A=[A]+[M]	+	1101		
	0	1101		1011
right shift A/Q	0	0110		1101
q0=1, add A=[A]+[M]	+	1101		
	1	0011		1101
right shift A/Q	0	1001		1110
q0=0, right shift A/Q	0	0100		1111
q0=1, add A=[A]+[M]	+	1101		
	1	0001		1111
right shift A/Q	0	1000		1111

The upper half of the product 010001111_2 is in register *A*, while the lower half is in register *Q*.

Division

Example for division: $27 \div 13 = 21\frac{1}{13}$

```
                  000010101   Quotient
Divisor   1101 )100010010    Dividend
                -1101
                  10000
                  -1101
                      1110
                     -1101
                         1    Remainder
```



Algorithm for hardware division (restoring):

```
Do n times:
{
        left shift A and Q by 1 bit;
        $A \leftarrow A-M$;
        if $A<0$ ($a_{n-1}=1$) then $q_0 \leftarrow 0$, $A \leftarrow A+M$ (restore)
        else $q_0 \leftarrow 1$;
}
```

Note: When *A* and *Q* are left shifted, the MSB of *Q* becomes the LSB of *A* and the MSB of *A* is lost. The LSB of *Q* is made available for the next quotient bit.

Example: $8 \div 3 = 2\frac{2}{3}$

Initially, the divisor $3 = 0011_2$ is in register *M*, the dividend $8 = 1000_2$ is in register *Q* and register *A* is zero.

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		$000\underline{.}$
$A = [A] - [M]$	+	1101		
$A < 0$		1110		$000\underline{0}$
$A = [A] + [M]$	+	0011		
		0001		$000\underline{0}$
left shift A/Q		0010		$00\underline{0.}$
$A = [A] - [M]$	+	1101		
$A < 0$		1111		$00\underline{00}$
$A = [A] + [M]$	+	0011		
		0010		$00\underline{00}$
left shift A/Q		0100		$0\underline{00.}$
$A = [A] - [M]$	+	1101		
$A > 0$		0001		$0\underline{001}$
left shift A/Q		0010		$\underline{001.}$
$A = [A] - [M]$	+	1101		
$A < 0$		1111		$\underline{0010}$
$A = [A] + [M]$	+	0011		
		0010		$\underline{0010}$

Note that subtraction by 3 is implemented by adding its 2s-complement $-3 = 1101_2$. The quotient $2 = 0010_2$ is in register *Q* and the reminder $2 = 0010_2$ is in register *A*.

Non-restoring Division

In the algorithm above, if the subtraction produces a non-negative result ($A \geq 0$), registers *A* and *Q* are left shifted and the next subtraction is carried out. But if the subtraction produces a negative result ($A < 0$), the dividend need be first restored by adding the divisor back before left shift *A* and *Q* and the next subtraction:

- If $A \geq 0$, then $2A - M$ (left shift and subtract);
- If $A < 0$, then $2(A + M)$ (restore, left shift and subtract).

When $A < 0$, the restoration is avoided by combining the two steps. This leads to a faster non-restoring division algorithm:

Algorithm for hardware division (non-restoring):

```
Do n times:
{
        left shift A and Q by 1 bit;
        if (previous A >= 0) then A←A-M
        else A←A+M;
        if (current A >= 0) then q0←1
        else q0←0;
}
if (A<0) then A←A+M (remainder must be positive);
```

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		$000\underline{.}$
A=[A]-[M]	+	1101		
$A < 0$		1110		$000\underline{0}$
left shift A/Q		1100		$00\underline{0.}$
$A = [A] + [M]$	+	0011		
$A < 0$		1111		$00\underline{00}$
left shift A/Q		1110		$0\underline{00.}$
$A = [A] + [M]$	+	0011		
$A > 0$		0001		$0\underline{001}$
left shift A/Q		0010		$\underline{001.}$
$A = [A] - [M]$	+	1101		
$A < 0$		1111		$\underline{0010}$
$A = [A] + [M]$	+	0011		
		0010		$\underline{0010}$

The quotient 2 = 0010_2 is in register *Q* and the remainder 2 = 0010_2 is in register *A*. The restoring division requires two operations (subtraction followed by an addition to restore) for each zero in the quotient. But non-restoring division only requires one operation (either addition or subtraction) for each bit in the quotient.

Signed Multiplication

The following example shows that signed 2s-complement representation can be used to represent negative operands as well as positive ones in multiplication.

Example: (–5) × (–4) = 20

4 : `000100`; –4 : `111110`; 5 : `000101`; –5 : `111011`

Use *n* = 6 bits to represent the product.

We first represent both operands in signed 2s-complement, and then carry out the normal multiplication:

					1	1	1	0	1	1
				x	1	1	1	1	0	0
1	1	1	1	1	0	0	1	1		
1	1	1	1	0	0	1	1			
1	1	1	0	0	1	1				
1	1	0	0	1	1					
.	.	.	.	.	0	1	0	1	0	0

The last six bits of the result are 010100_2, representing a positive product (20_{10}). Note:

- Both positive and negative operands should be properly sign extended whenever needed.
- When the multiplier is negative represented in signed 2s-complement, each `1` added in front due to the sign extension requires an addition of the multiplicand to the partial product. However, you only need to consider enough bits to guarantee *n* bits required in the result.

As shown in a homework problem, this method works in all cases of both positive and negative multiplicands and multipliers.

Fast Multiplication: Booth’s Algorithm

Booth’s algorithm serves two purposes:

- Fast multiplication (when there are consecutive 0s or 1s in the multiplier)
- Signed multiplication

First, consider two decimal multiplications: 98765 × 10001 and 98765 × 9999. It is obvious that if straight forward multiplication is used, the first one is easier than the second, as only two single-digit multiplications are needed for the first, while four are needed for the second. However, as we also realize that:

98765 × 10001 = 98765 × (10000 + 1) = 98765 × 10000 + 98765

and

98765 × 9999 = 98765 × (10000 – 1) = 98765 × 10000 – 98765

the two multiplications should be equally easy.

Example 1

If there is a sequence of `0`s in the multiplier, the multiplication is easy, as all `0`s can be skipped.

$$
\begin{array}{cccc||ccccccccccc}
 & & 2 & 2 & & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
 & \mathrm{x} & 1 & 7 & & & & & & \mathrm{x} & 0 & 1 & 0 & 0 & 0 & 1 \\
\hline
 & 1 & 5 & 4 & & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
+ & 2 & 2 & & + & 0 & 1 & 0 & 1 & 1 & 0 & & & & & \\
\hline
 & 3 & 7 & 4 & & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
$$

Example 2

However, it does not help if there is a sequence of `1`s in the multiplier. We have to go through each one of them:

$$
\begin{array}{cccc||ccccccccccc}
 & & 2 & 2 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
 & \mathrm{x} & 1 & 4 & & & & & \mathrm{x} & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
 & & 8 & 8 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & \\
+ & 2 & 2 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & \\
 & & & & + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & & & \\
\hline
 & 3 & 0 & 8 & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
$$

How can we enjoy a sequence of `1`s as well as a sequence of `0`s? We first realize that

`001110 = 010000 - 000010`

or, in general, a string of `1`s in the multiplier *A* can be written as:

$$
\begin{array}{cccccccccccc}
 & \mathrm{d} & \mathrm{d} & \underline{0} & \underline{1} & 1 & \cdots & 1 & \underline{1} & \underline{0} & \mathrm{d} & \mathrm{d} \\
= & \mathrm{d} & \mathrm{d} & \underline{1} & \underline{0} & 0 & \cdots & 0 & 0 & 0 & \mathrm{d} & \mathrm{d} \\
- & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & \underline{1} & \underline{0} & 0 & 0
\end{array}
$$

where d is “don’t care” (either `0` or `1`). If we define the first part above as $A_{\mathrm{leftend}} = dd10...00dd$ and the second part as $A_{\mathrm{rightend}} = 0000...1000$, then the multiplication becomes:

$$B \times A = B \times (A_{\mathrm{leftend}} - A_{\mathrm{rightend}}) = B \times A_{\mathrm{leftend}} - B \times A_{\mathrm{rightend}}$$

In other words, only the two ends of a string of 1s in the multiplier need to be taken care of. At the left end, the multiplicand is added to the partial product, while at the right end, the multiplicand is subtracted from the partial product. The above multiplication can therefore be written as:

$$\begin{array}{ccccccccccc||ccccccccccc}
 & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 \\
 & & & & \mathrm{x} & 0 & \underline{0} & \underline{1} & 1 & \underline{1} & \underline{0} & & & & & \mathrm{x} & 0 & \underline{0} & \underline{1} & 1 & \underline{1} & \underline{0} \\
\hline
 & 0 & 1 & 0 & 1 & 1 & 0 & & & & & & 0 & 1 & 0 & 1 & 1 & 0 & & & & \\
- & & & & 0 & 1 & 0 & 1 & 1 & 0 & & + & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & \\
\hline
 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}$$

On the right side above, the subtraction is carried out by adding 2s-complement. We observe that if there is a sequence of 1s in the multiplier, only the two ends need to be taken care of, while all 1s in between do not require any operation.

Booth's algorithm for multiplication is based on this observation. To do a multiplication $B \times A$, where $B = b_{n-1}b_{n-2}\dots b_1 b_0$ is the multiplicand and $A = a_{n-1}a_{n-2}\dots a_1 a_0$ is the multiplier, we check every two consecutive bits in A at a time:

a_i	a_{i-1}	$a_{i-1} - a_i$	Operations
0	0	0	in middle of string of 0. No operation.
1	0	-1	beginning of string of 1. Subtract B from partial product
1	1	0	in middle of string of 1. No operation.
0	1	1	end of string of 1. Add B to partial product

where $i = 0, 1, ..., n-1$, and when $i = 0$, $a_{i-1} = a_{-1} = 0$.

Why does it work? What we did can be summarized as the following:

$$\begin{aligned}
Product &= (a_{-1} - a_0) \times B \times 2^0 + (a_0 - a_1) \times B \times 2^1 + \dots + (a_{n-2} - a_{n-1}) \times 2^{n-1} \\
&= B \times \left(-a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a^i \times 2^i \right) \\
&= B \times Val(A)
\end{aligned}$$

Recall that the value of a signed 2s-complement number (either positive or negative) can be found by

$$Val(A = a_{n-1} \dots a_0) = -a_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} a^i \times 2^i$$

Another Example

Assume $n = 7$ bits available. Multiply $B = 22 = 0010110_2$ by $A = -34 = 1011110_2$. First represent both operands and their negation in signed 2s-complement, then carry out the multiplication in the hardware:

		[M]	0010110			
q_iq_{i-1}	Action	[A]	0000000	[Q]	1011110	0
00	right shift		0000000		0101111	0
10	-B	+	1101010			
			1101010		0101111	0
	right shift		1110101		0010111	1
11	right shift		1111010		1001011	1
11	right shift		1111101		0100101	1
11	right shift		1111110		1010010	1
01	+B	+	0010110			
			0010100		1010010	1
	right shift		0001010		0101001	0
10	-B	+	1101010			
			1110100		0101001	0
	right shift		1111010		0010100	1

The upper half of the final result 1111010 0010100 is in register [A] while the lower half is in register [Q]. The product is given in signed 2s-complement and its actual value is negative of the 2s-complement:

$$B \times A = -\overline{1111010\ 0010100} = -00001011101100 = -748_{10}$$

Also note that:

- As the operands are in signed 2s-complement form, the arithmetic shift is used for the right shifts above, i.e., the MSB bit (sign bit) is always repeated, while all other bits are shifted to the right. This guarantees proper sign extension for both positive and negative values represented in signed 2s-complement.
- When the multiplicand is negative, represented by signed 2s-complement, it needs to be complemented again for subtraction (when the LSB of the multiplier is 1 and the extra bit is 0, i.e., the beginning of a string of 1s).