Part 3
Advanced Arithmetic Hardware

- Adder
- Multiplier

Adder

Rounding Adder, Half Adder

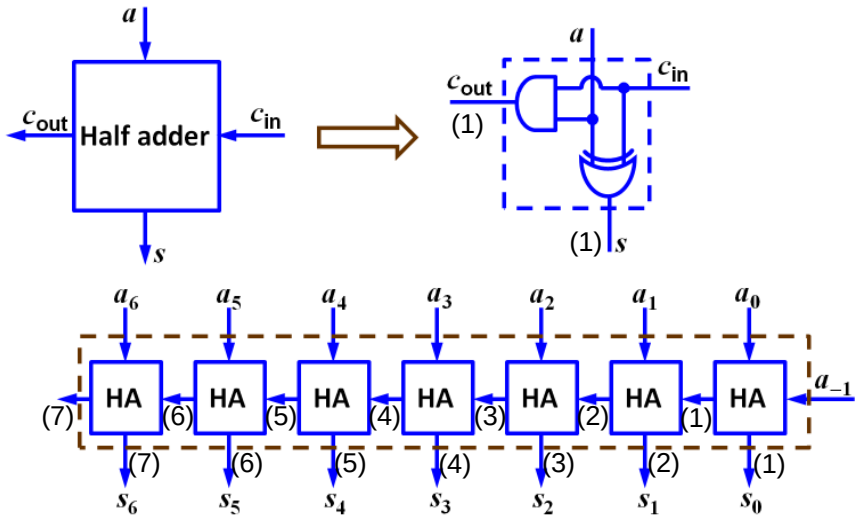
Consider rounding a binary number consisting of integer part and fractional part into an integer.

```
  1 0 1 1 1 1 1 1 1 . 1 0 1 0 1
+ 0 0 0 0 0 0 0 0 1 |
  x x x x x x x x x x |
```

This rounding process requires an adder with as many bits as there are bits in the integer part. This is because all the integer bits may be "1".

The adder used for rounding purpose need not be constructed from full adders since the basic adder block only need to add two bits together.

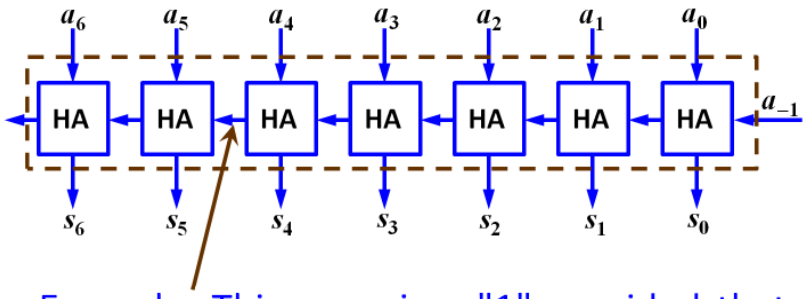
Rounding adder



Each half adder needs 1 gate delay. The *N*-bit rounding adder needs *N* gate delays to complete. This is because the carry bit propagates from stage to stage.

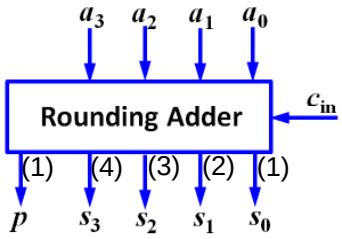
Carry Look-Ahead in Rounding Adder

In carry look-ahead, the carry bit does not propagate from stage to stage but is computed separately.



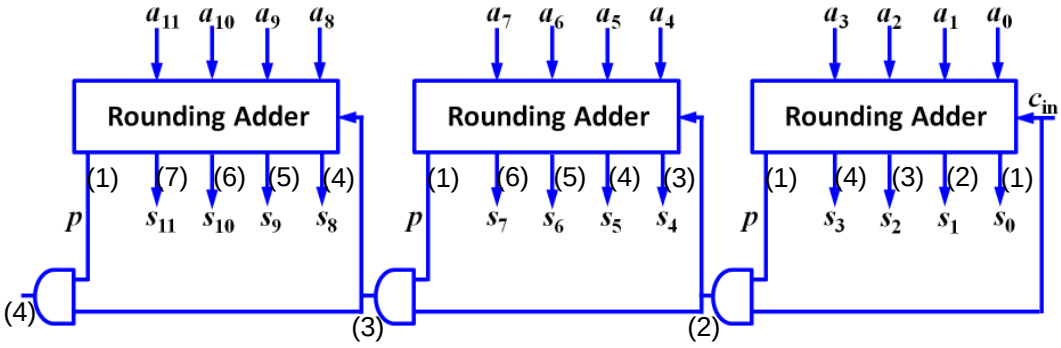
Example: This carry is a "1" provided that a_4, a_3, a_2, a_1, a_0, and a_{-1} are "1".
Thus, carry = $(a_4 \bullet a_3 \bullet a_2 \bullet a_1 \bullet a_0) \bullet a_{-1}$.
$(a_4 \bullet a_3 \bullet a_2 \bullet a_1 \bullet a_0)$ can be computed separately.

Let $p = a_3 \bullet a_2 \bullet a_1 \bullet a_0$.
p is computed independent of c_{in}.
$c_{out} = p \bullet c_{in}$



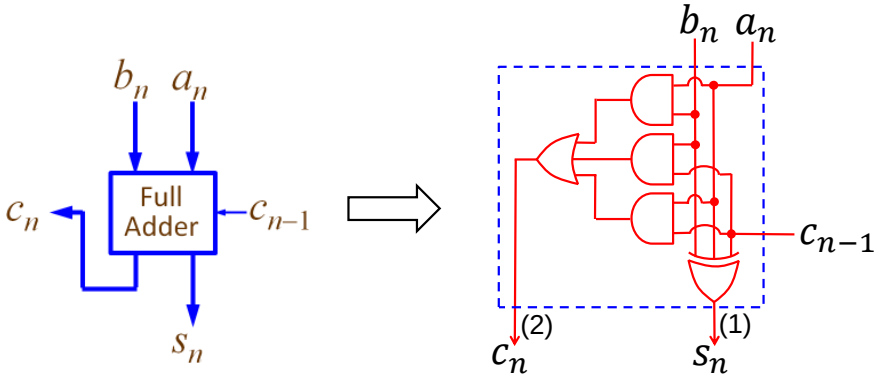
If each half adder, or each gate, requires 1 gate delay, then this block consisting of 4 (actually 3½) half adders requires 4 gate delays. However, *p* requires only 1 gate delay.

Therefore, cascading of such blocks is possible.
Carry-out takes 1 (to compute first *p*) + 3 (propagation through 3 gates) = 4 gate delays.



Full Adder

Bitwise addition of two 2's complement numbers requires adding not only two input bits but also the carry bit.

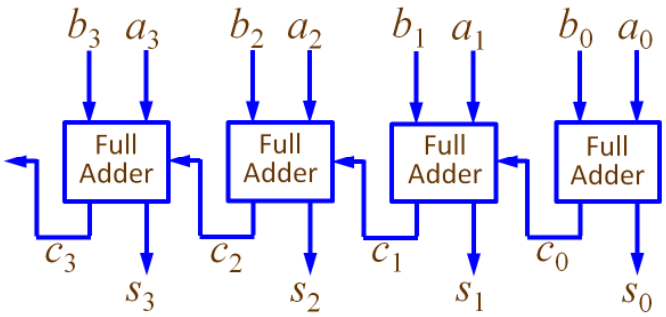


$$s_n = a_n \oplus b_n \oplus c_{n-1}$$

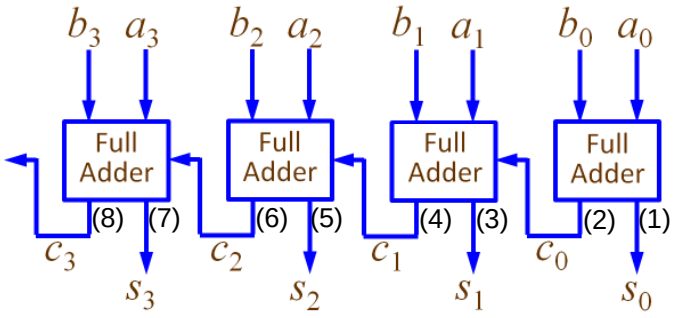
$$c_n = a_n \cdot b_n + b_n \cdot c_{n-1} + a_n \cdot c_{n-1}$$

Ripple Carry Adder

$$\begin{array}{cccccccc} & a_{N-1} & a_{N-2} & x & x & x & a_1 & a_0 \\ + & b_{N-1} & b_{N-2} & x & x & x & b_1 & b_0 \\ \hline s_N & s_{N-1} & s_{N-2} & x & x & x & s_1 & s_0 \end{array}$$



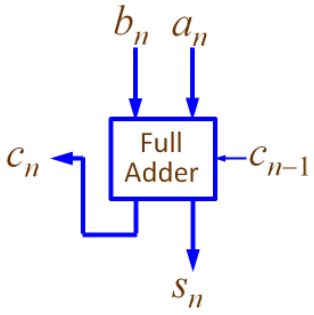
c_3 can be computed only after c_2 has been computed.
c_2 can be computed only after c_1 has been computed.
c_1 can be computed only after c_0 has been computed.



Computing each carry from valid inputs needs 2 gate delays. Therefore, computing c_3 needs 8 gate delays.

In general, an N stage ripple carry adder needs $2N$ gate delays to compute c_{N-1}.

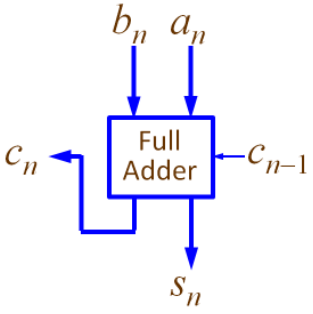
Carry Look-Ahead Adder



If a_n = b_n = 1, the full adder generates a carry, i.e. c_n = 1 regardless of c_{n-1}.

Let g_n denotes the state when the n^{th} full adder generates a carry. Hence,

$$g_n = a_n \bullet b_n \quad (a_n \text{ AND } b_n)$$

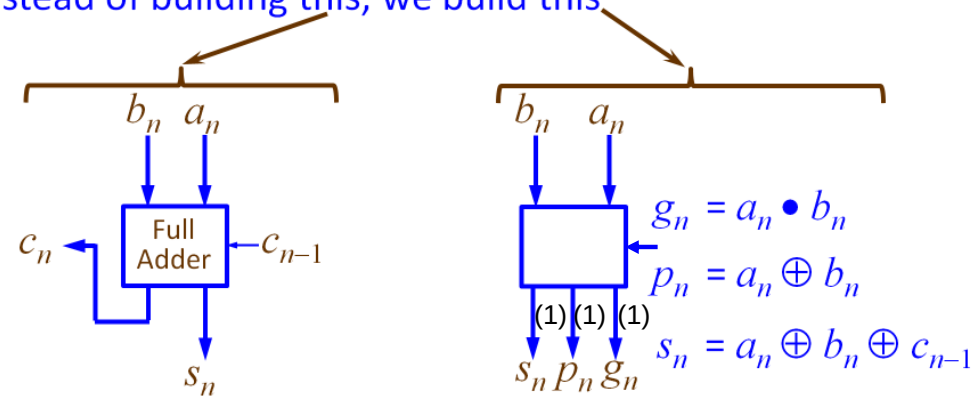


If, either a_n OR b_n is a 1, c_n = 1 if c_{n-1} = 1 and c_n = 0 if c_{n-1} = 0, i.e. the full adder propagates carry-in to carry-out.

Let p_n denotes the state when the n^{th} full adder propagates a carry. Hence,

$$p_n = a_n \oplus b_n \quad (a_n \text{ XOR } b_n)$$

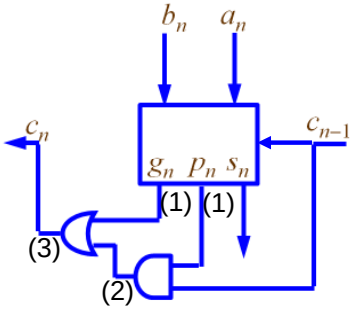
Instead of building this, we build this



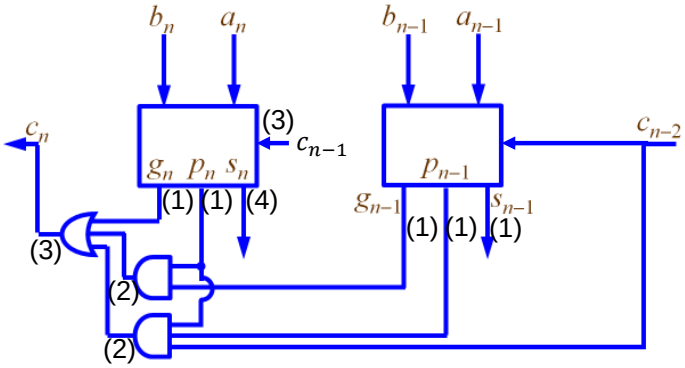
Carry generation for 1 bit:

$$c_n = g_n + p_n \bullet c_{n-1}$$

c_n needs 1 gate delay to compute g_n and p_n, plus 2 gate delays = 3 gate delays.



Carry generation for 2 bits:



$$c_n = g_n + p_n \bullet c_{n-1}$$

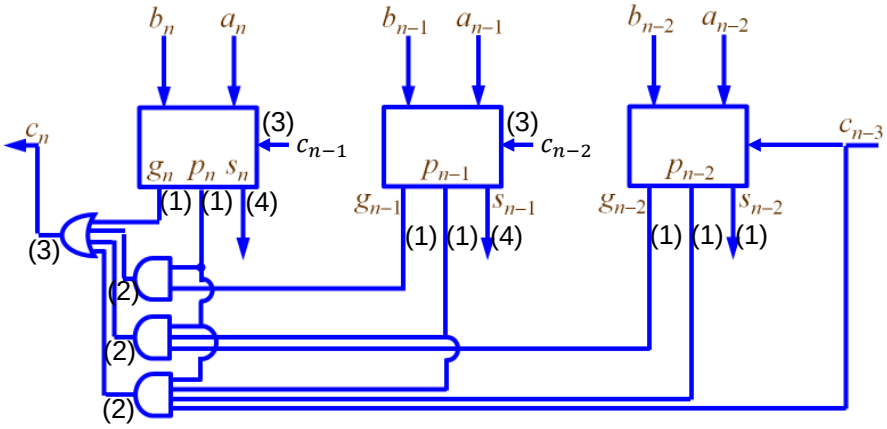
$$c_{n-1} = g_{n-1} + p_{n-1} \bullet c_{n-2}$$

$$c_n = g_n + p_n \bullet \{g_{n-1} + p_{n-1} \bullet c_{n-2}\}$$

$$= g_n + p_n \bullet g_{n-1} + p_n \bullet p_{n-1} \bullet c_{n-2}$$

c_n still takes 3 gate delays.

Carry generation for 3 bits:

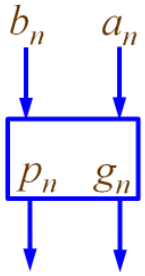


$$c_n = g_n + p_n \bullet g_{n-1} + p_n \bullet p_{n-1} \bullet g_{n-2} + p_n \bullet p_{n-1} \bullet p_{n-2} \bullet c_{n-3}$$

c_n still takes 3 gate delays.

Note that the gate delays will remain 3, 1, 1, 4 for c_n, g_n, p_n, and s_n for any number of bits n.

Carry generation in general:



$$c_0 = g_0 + p_0\, c_{in} = g_0 \text{ OR } (p_0 \text{ AND } c_{in})$$

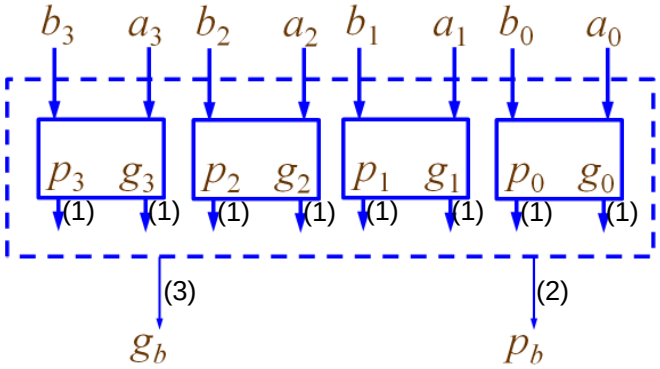
$$c_1 = g_1 + p_1\, c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{in}$$

$$c_2 = g_2 + p_2\, c_1 = g_2 + p_2 (g_1 + p_1 (g_0 + p_0\, c_{in}))$$

$$c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_{in}$$

$$c_n = g_n + p_n g_{n-1} + p_n p_{n-1} g_{n-2} + \ldots + p_n p_{n-1} \ldots p_0\, c_{in}$$

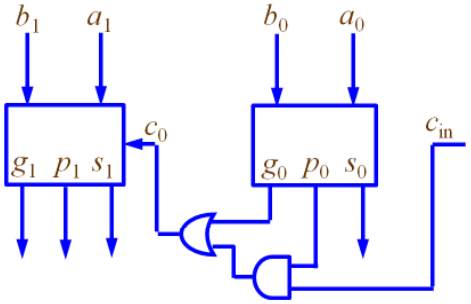
Block state generation:

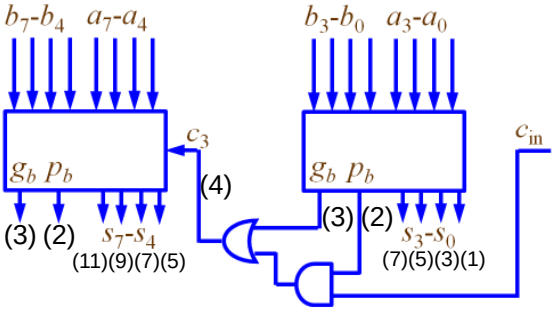


Block generate: $g_b = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$

Block propagate: $p_b = p_3 p_2 p_1 p_0$

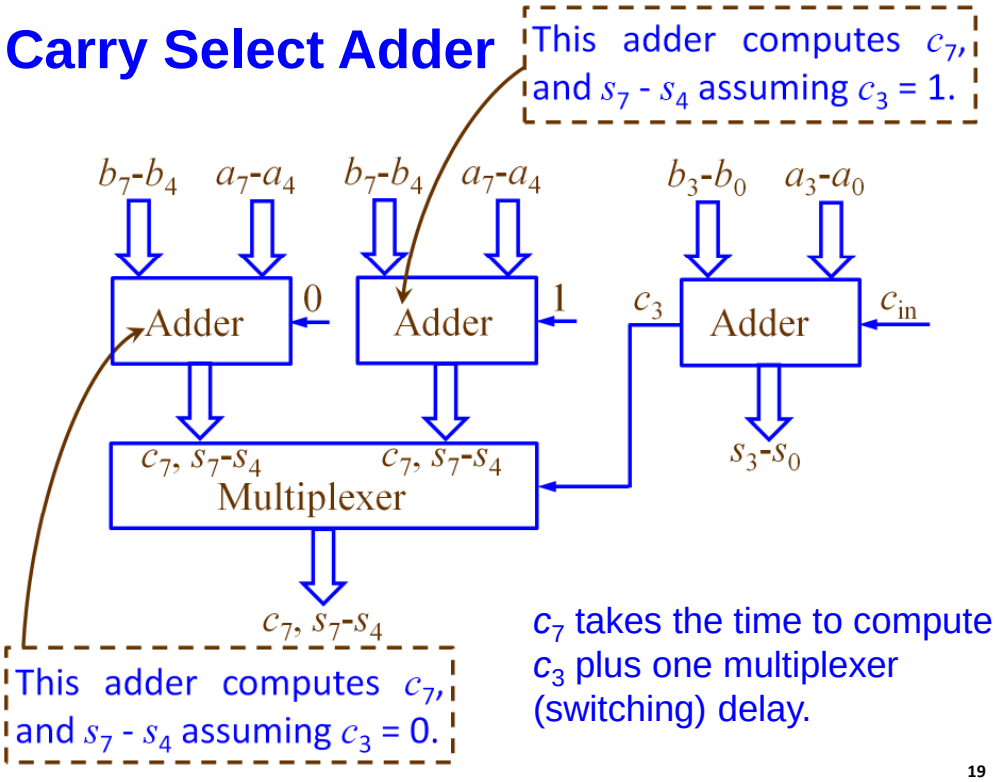
Carry generation for block carry look-ahead adder is the same as the bit-wise carry look ahead adder.





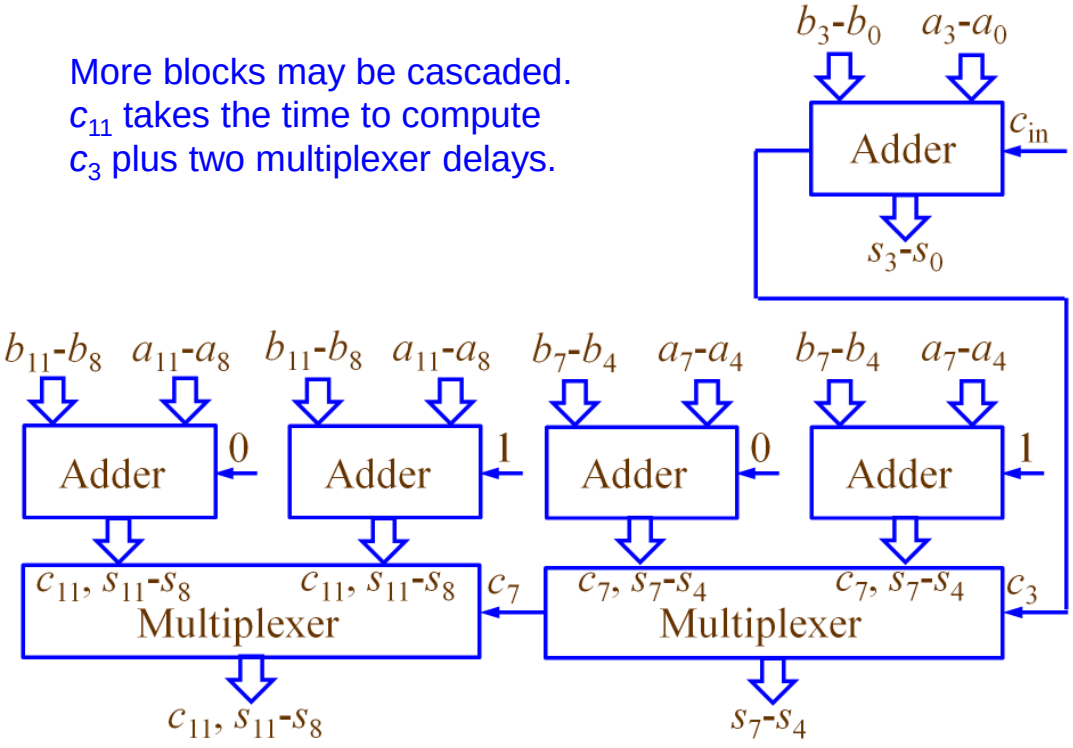
Note that the gate delays will remain 3, 2, 11, 9, 7, 5 for g_b , p_b , s_{n-1} to s_{n-4} for any number of blocks.

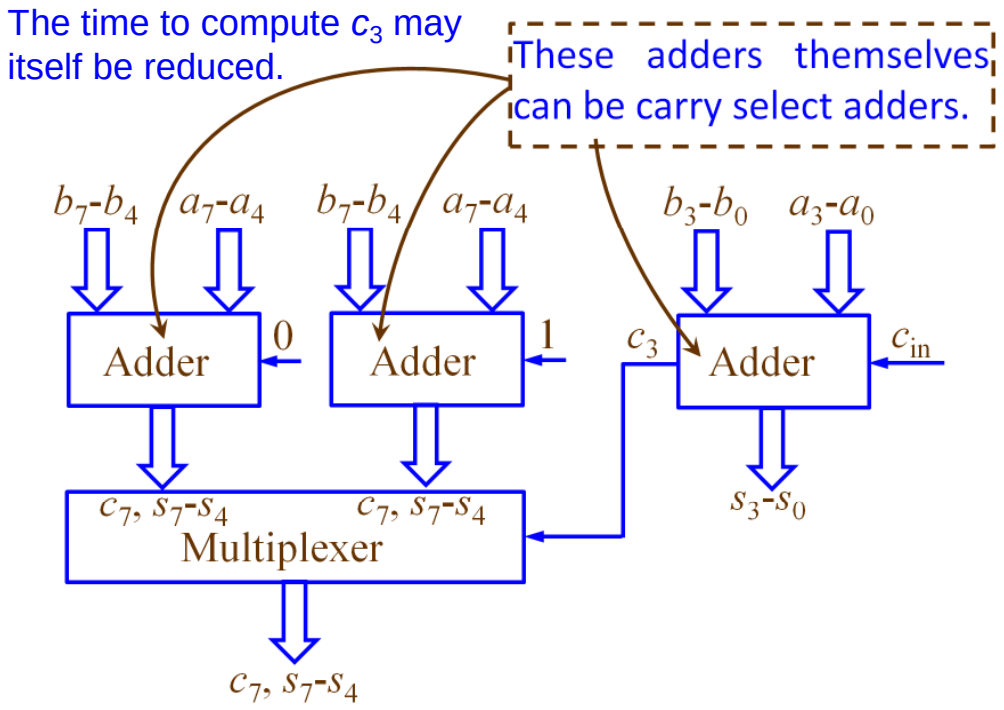
Carry Select Adder



c_7 takes the time to compute c_3 plus one multiplexer (switching) delay.

More blocks may be cascaded.
c_{11} takes the time to compute c_3 plus two multiplexer delays.



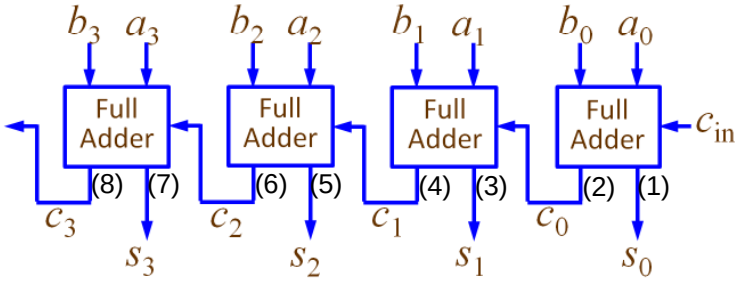


Carry Save Adder

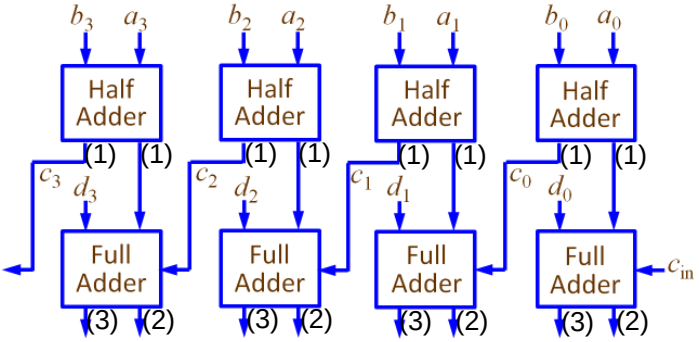
Consider the addition of 2 numbers A and B where

$A = [a_{N-1}, \ldots, a_n, \ldots, a_1, a_0]$
$B = [b_{N-1}, \ldots, b_n, \ldots, b_1, b_0]$

Suppose that the addition of a_0, b_0 and c_{in} produces sum s_0 and carry c_0. In general, the addition of a_n, b_n and c_{n-1} produces sum s_n and carry c_n. c_n is added to a_{n+1} and b_{n+1} to produce c_{n+1}. In a ripple carry adder, the carry-bit ripples "horizontally".

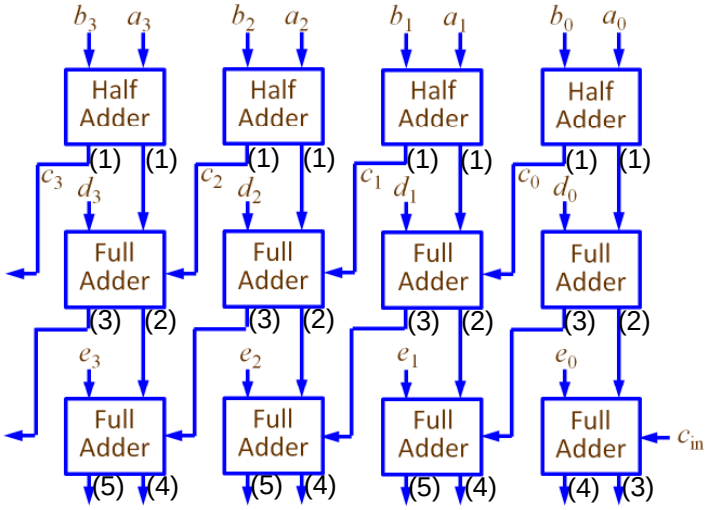


Now consider the addition of 3 numbers A, B, and D where $D = [d_{N-1}, \ldots, d_n, \ldots, d_1, d_0]$. The addition can be done in the following way.



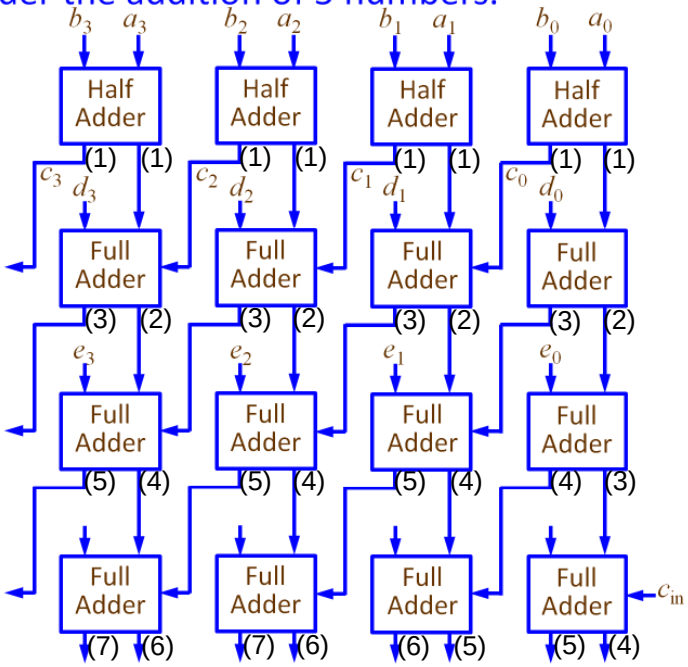
In the above scheme, c_n is not added to a_{n+1} and b_{n+1} but is added in the full adder that adds d_{n+1}. The scheme avoided "horizontal" addition of carry-bit in the addition of A and B. Nevertheless, the carry bits at the output of the full adders must still be added "horizontally".

Now consider the addition of 4 numbers A, B, D, and E where $E = [e_{N-1}, \ldots, e_n, \ldots, e_1, e_0]$.



The carry bits for the last row of full adders must still be added "horizontally".

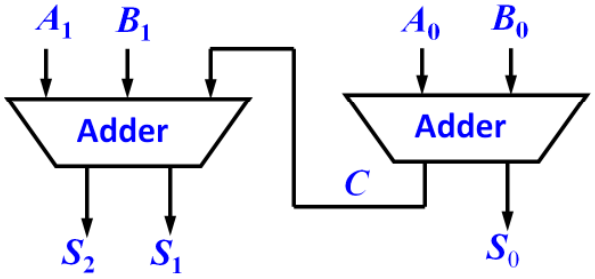
Now consider the addition of 5 numbers.



Only the carry bits for the last row ripple "horizontally".

Pipelined Adder

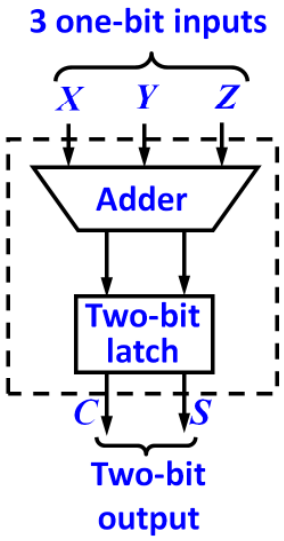
$$\begin{array}{r} A_1\ A_0 \\ +\ B_1\ B_0 \\ \hline S_2\ S_1\ S_0 \end{array}$$



Suppose that the time required by an adder is x seconds.

The time required to compute the sum will be $2x$ seconds.

A pipelined full adder

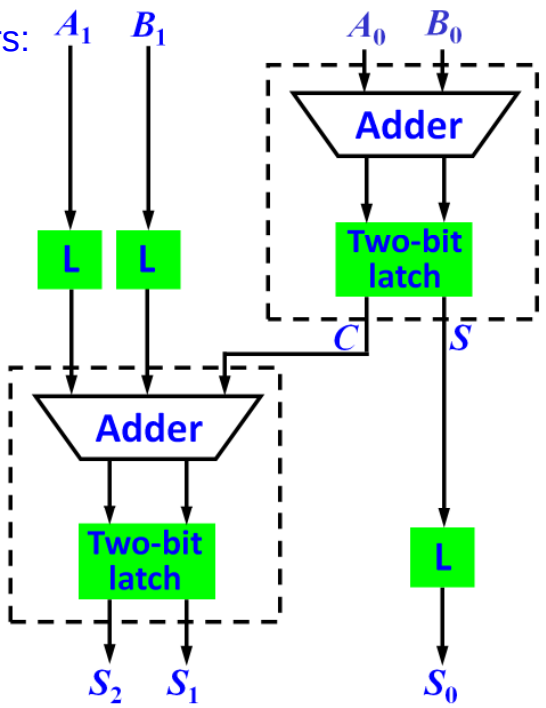


X	Y	Z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

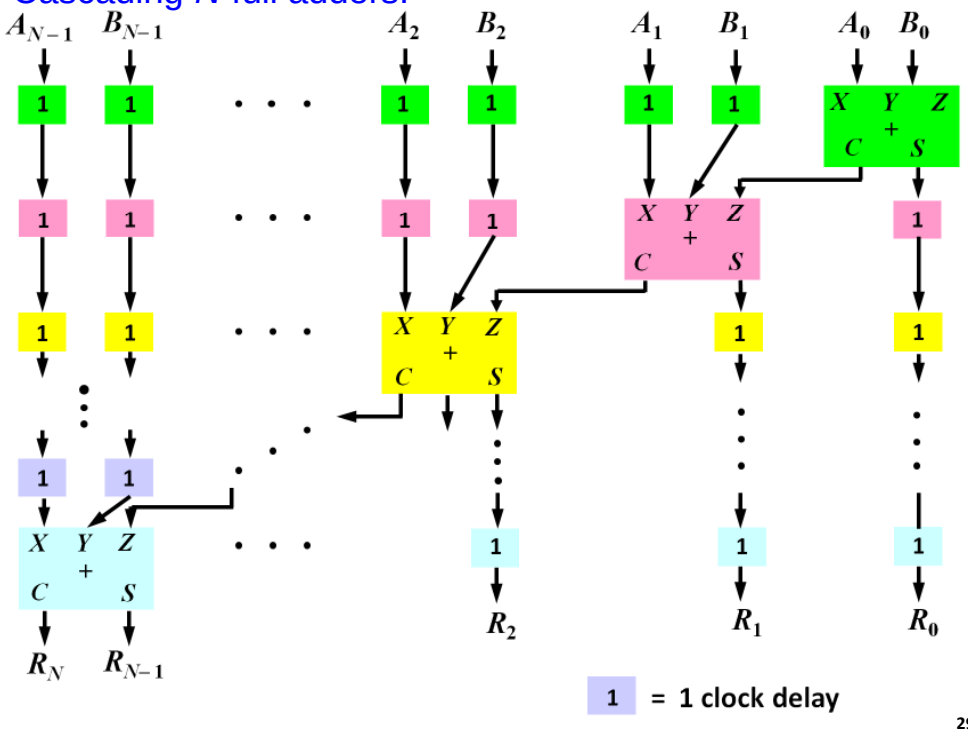
Cascading 2 full adders:

$$\begin{array}{r} A_1\ A_0 \\ +\ B_1\ B_0 \\ \hline S_2\ S_1\ S_0 \end{array}$$

L = latch



Cascading *N* full adders:



Floating-Point Adder

$0.9878 \times 10^5 + 0.8934 \times 10^4$
$= 0.9878 \times 10^5 + 0.08934 \times 10^5$
$= 1.07714 \times 10^5$
$= 0.107714 \times 10^6$

Operands: A, B

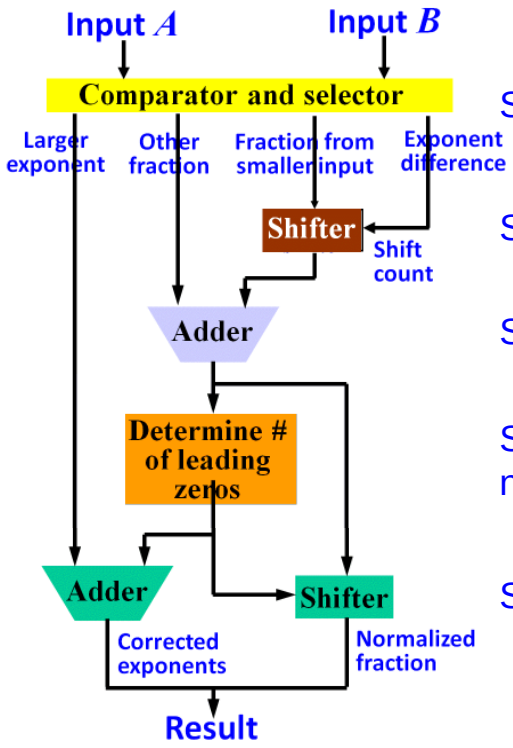
Step 1: Compare A and B

Step 2: Data alignment

Step 3: Add

Step 4: Determine number of leading zeros

Step 5: Normalization



Step 1: Compare *A* and *B*

Step 2: Data alignment

Step 3: Add

Step 4: Determine number of leading zeros

Step 5: Normalization

Multiplier

Integer-Power-of-Two Multiplier

$(101000)_{2's} = -24$

$(110100)_{2's} = -12$

$(111010)_{2's} = -6$

$(111101)_{2's} = -3$

$(010100)_{2's} = 20$

$(001010)_{2's} = 10$

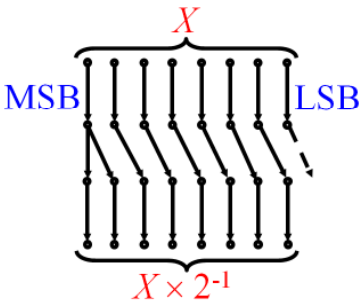
$(000101)_{2's} = 5$

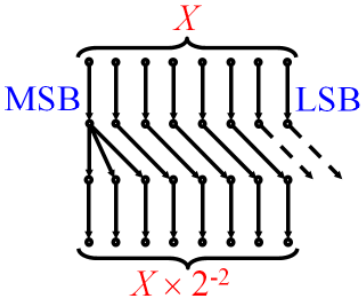
Shifting a two's complement number to the right (with sign extension) by 1 place has the effect of dividing the number by two.

Shifting a two's complement number to the right (with sign extension) by K places has the effect of dividing the number by 2^K.

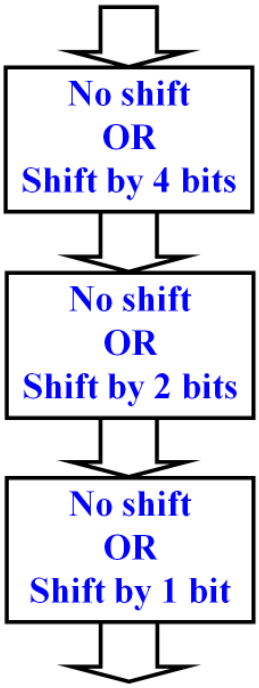
Thus, multiplying a number by an integer-power-of-two is a very simple process.

Multiplying by 2^{-K}



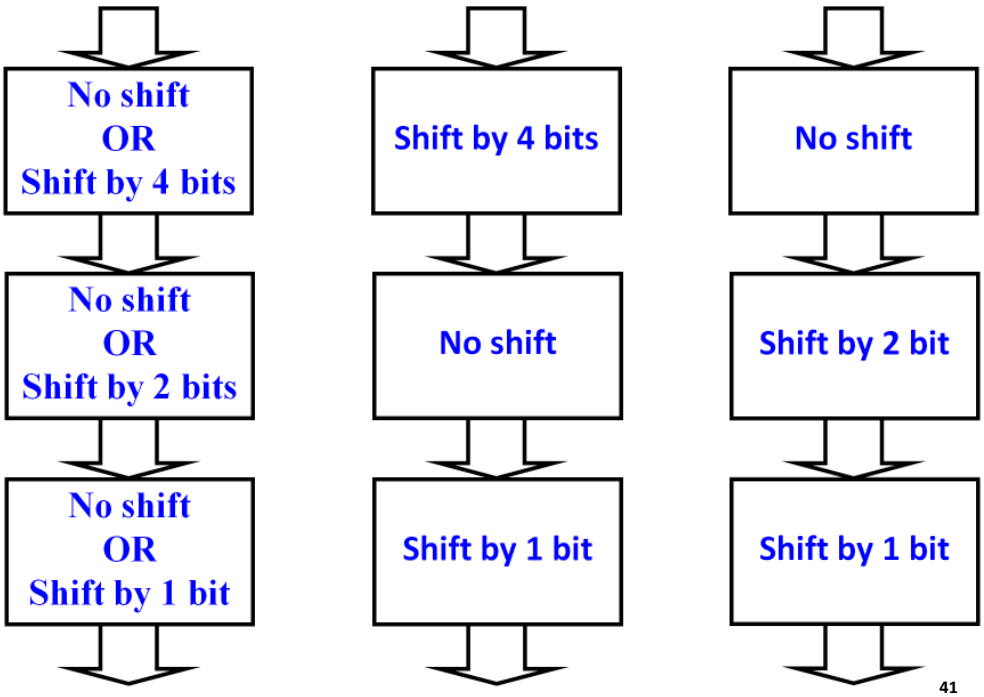


Variable shifter: shift by K bits where $0 \leq K \leq 7$



Shift by 5 bits example

Shift by 3 bits example



Booth Multiplier

Multiply $(6375)_{10}$ x $(99)_{10}$

$(6375)_{10}$ x $(99)_{10}$ = $(6375)_{10}$ x $(100 - 1)_{10}$
= $(637500)_{10} - (6375)_{10}$
= $(631125)_{10}$

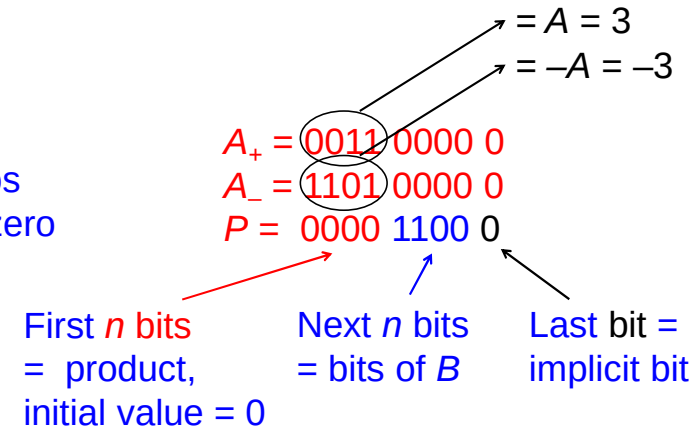
Multiply A x $(00011110)_{2's}$ = A x $(001000\text{-}10)_{2's}$

= A x $(00100000)_{2's} - A$ x $(00000010)_{2's}$

Algorithm:
2 n-bit numbers A, B

Precompute:
$A_+ = A$, n +1 zeros
$A_- = -A$, n +1 zeros
$P = n$ zeros, B, 1 zero

Example:
$n = 4$
$A = (0011)_{2's} = 3$
$B = (1100)_{2's} = -4$



Loop n times:

If 2 LSB's of P = 01, $P \leftarrow P + A_+$

$A \times (00011110)_{2's} = A \times (001000\text{-}10)_{2's}$

$\Rightarrow$ Add $2^5 A$ to the product

If 2 LSB's of P = 10, $P \leftarrow P + A_-$

$A \times (00011110)_{2's} = A \times (001000\text{-}10)_{2's}$

$\Rightarrow$ Subtract $2^1 A$ from the product

If 2 LSB's of P = 00 or 11, do nothing

$A \times (00011110)_{2's} = A \times (001000\text{-}10)_{2's}$

$\Rightarrow$ Do nothing

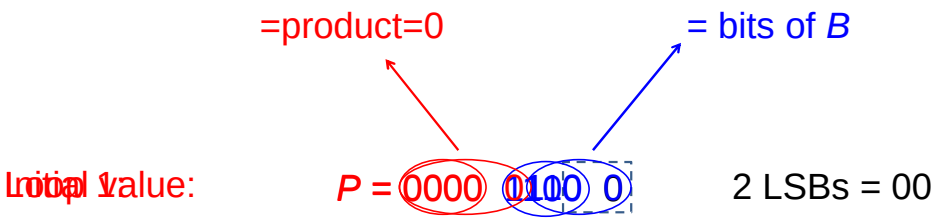
Shift P right by 1 bit

Brings next bit of B

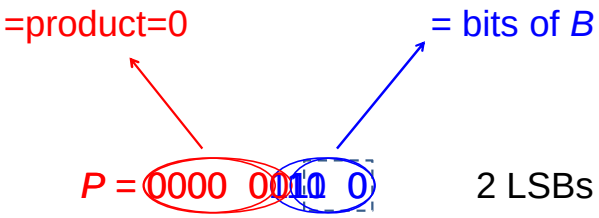
Effectively shifts A_+, A_- to the left, or multiplies $2^i A$ by 2

Final product = P without the implicit bit

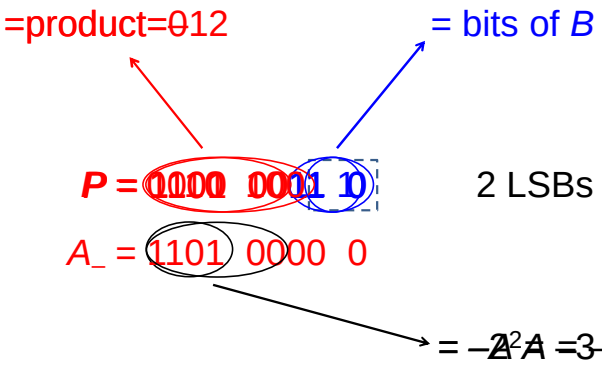
Example: A = 0011 = 3, B = 1100 = –4



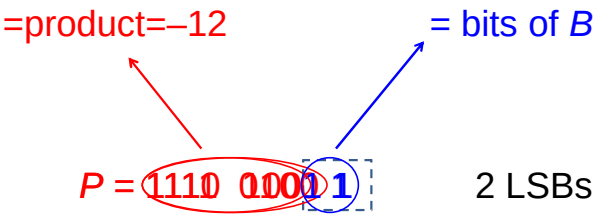
Example: A = 0011 = 3, B = 1100 = –4



Example: A = 0011 = 3, B = 1100 = –4



Example: A = 0011 = 3, B = 1100 = –4



Final product P = 1111 0100 = –12

Wallace Tree Multiplier

2's complement multiplication: multiply 7 and 3

Step 1: Bitwise multiply (AND) to partial products

```
    0111
  x 0011
  -------
  1 1 1   ←carry
    0111
   0111
  0000
 0000
 --------
 00010101  =  21
```

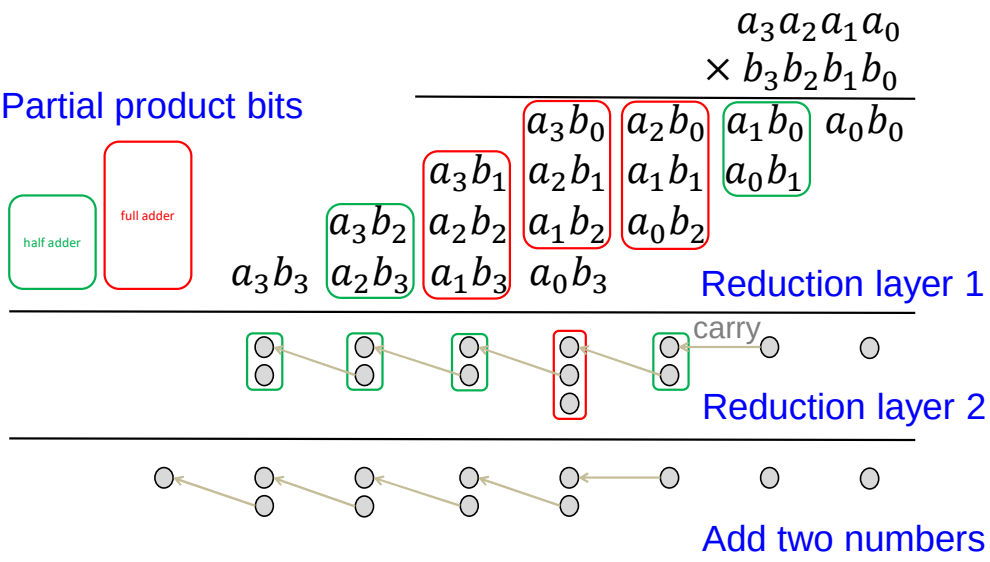
Step 2: Add partial products for each position

Adding partial product bits = counting number of 1's

Wallace tree uses layers of half/full adders to add these bits until only 2 bits are left in each position.

Adding these two numbers give the result.

Multiply $a_3a_2a_1a_0$ with $b_3b_2b_1b_0$:



Multiplication of two *n*-bit numbers gives m_i partial product bits in the *i*-th position, where $m_i \leq n$.

In a reduction layer, m_i partial product bits are reduced to $\lceil \frac{m_i}{3} \rceil$ bits.

Since the (*i*–1)th position generates $\text{round}\left(\frac{m_{i-1}}{3}\right)$ carries, number of bits in the *i*-th position is reduced to:

$$\lceil \frac{m_i}{3} \rceil + \text{round}\left(\frac{m_{i-1}}{3}\right)$$

n=4 example: Initial $m_i = 1,2,3,4,3,2,1$ for $i = 0$ to 6.

After reduction layer 1, $m_4 = 3$ reduces to:

$$\lceil \frac{m_4}{3} \rceil + \text{round}\left(\frac{m_3}{3}\right) = \lceil \frac{3}{3} \rceil + \text{round}\left(\frac{4}{3}\right) = 1 + 1 = 2$$

Wallace tree multiplier is faster since it takes log(*n*) time.