Advanced Parallel Architecture

Lesson 7

Annalisa Massini - 2016/2017

Computer arithmetic

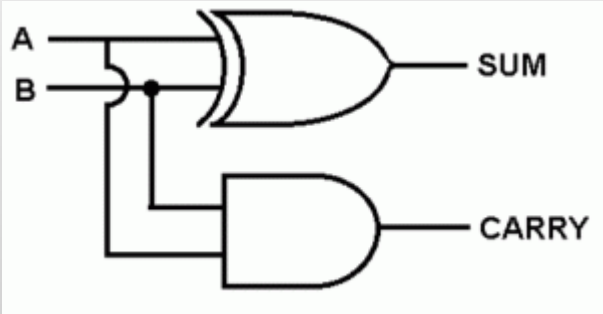
Hennessy, Patterson

Computer architecture A quantitive approach

Appendix J

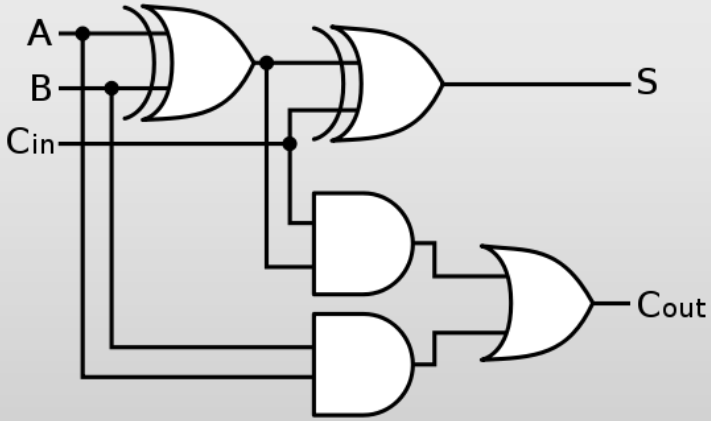
Half adder and Full adder

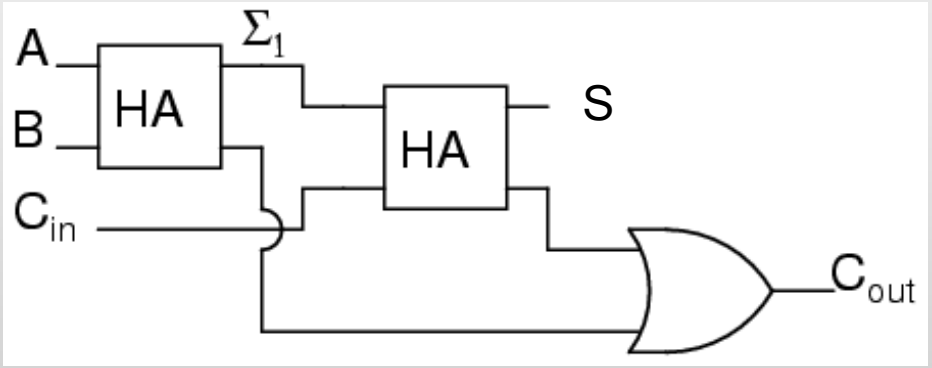
- Adders are usually implemented by combining multiple copies of simple components
- The natural components for addition are ***half adders*** and ***full adders***
- The half adder takes two bits *a* and *b* as input and produces a sum bit *s* and a carry bit c_{out} as output
- As logic equations, $s = a\bar{b} + \bar{a}b$ and $c_{out} = ab$



Half adder and Full adder

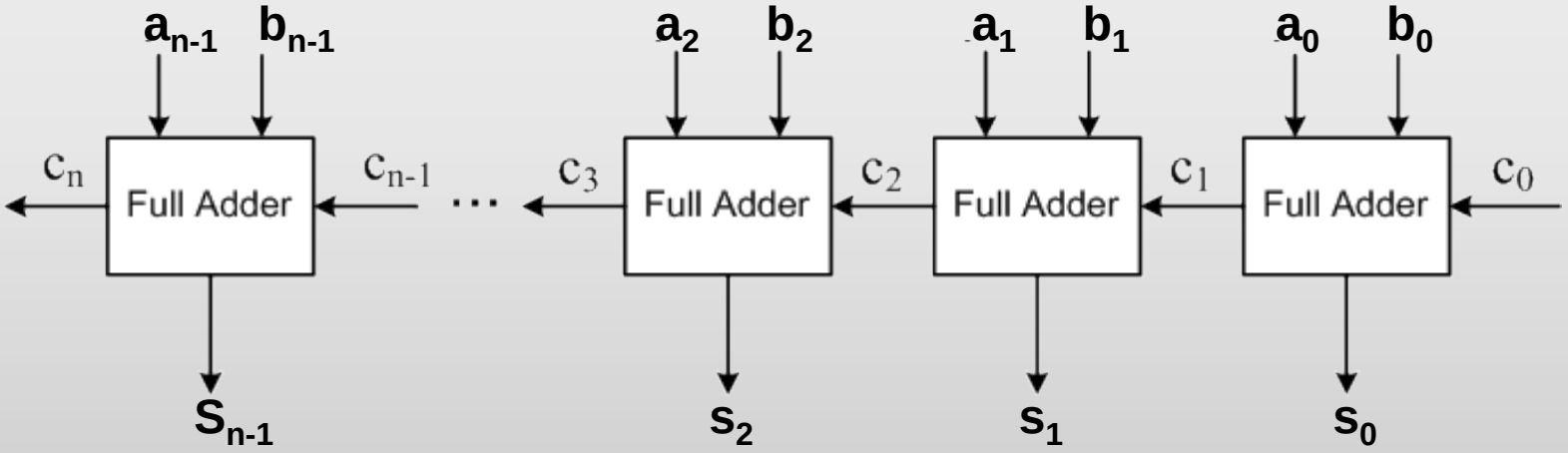
- The full adder takes three bits *a, b* and *c* as input and produces a sum bit *s* and a carry bit c_{out} as output
- As logic equations, $s = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc = (a \oplus b) \oplus c$ and $c_{out} = (a \oplus b)c + ab$
- The half adder is a (2,2) adder, since it takes two inputs and produces two outputs. The full adder is a (3,2) adder, since it takes three inputs and produces two outputs





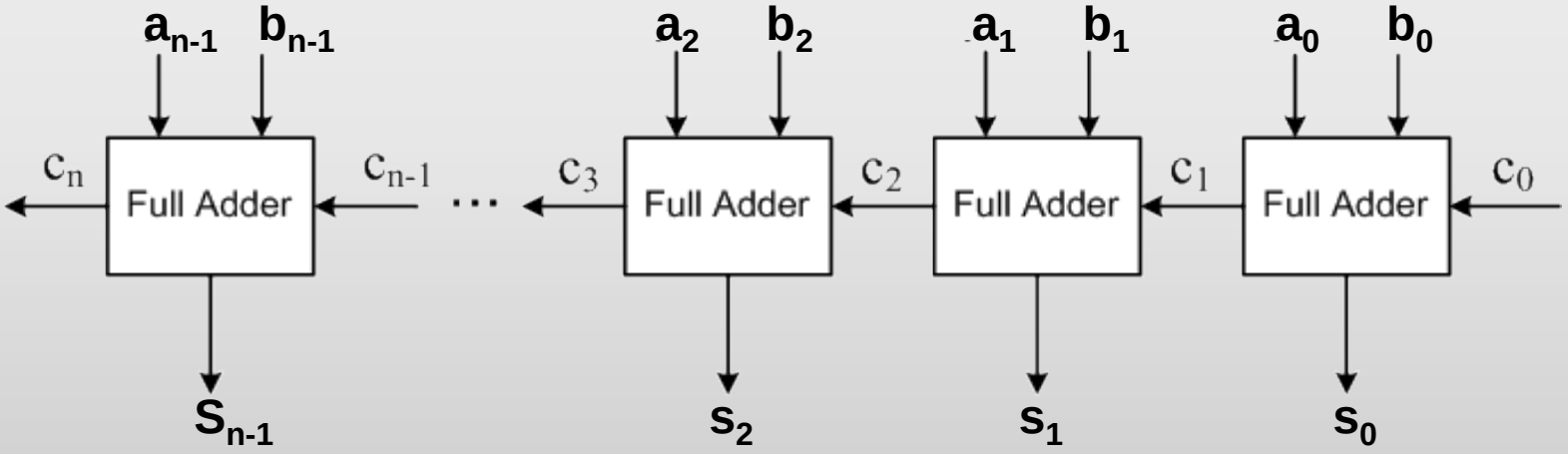
Ripple-Carry Addition

- The principal problem in constructing an adder for *n*-bit numbers out of smaller pieces is propagating the carries from one piece to the next
- The most obvious way to solve this is with a *ripple-carry adder*, consisting of *n* full adders



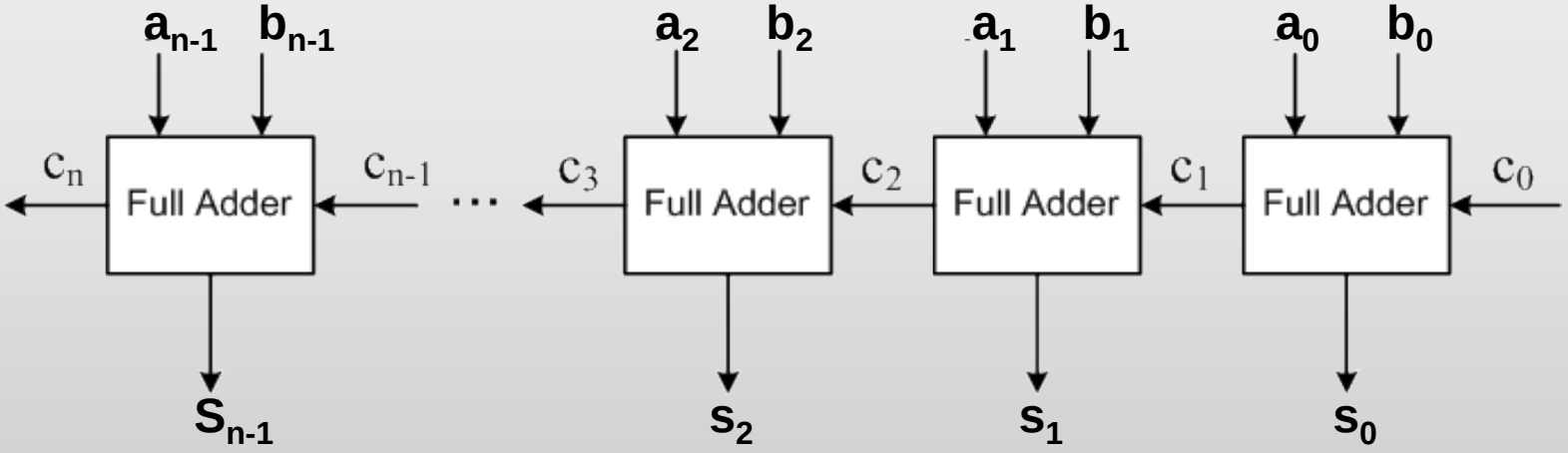
Ripple-Carry Addition

- The time a circuit takes to produce an output is proportional to the maximum number of logic levels through which a signal travels
- Determining the exact relationship between logic levels and timings is highly technology dependent



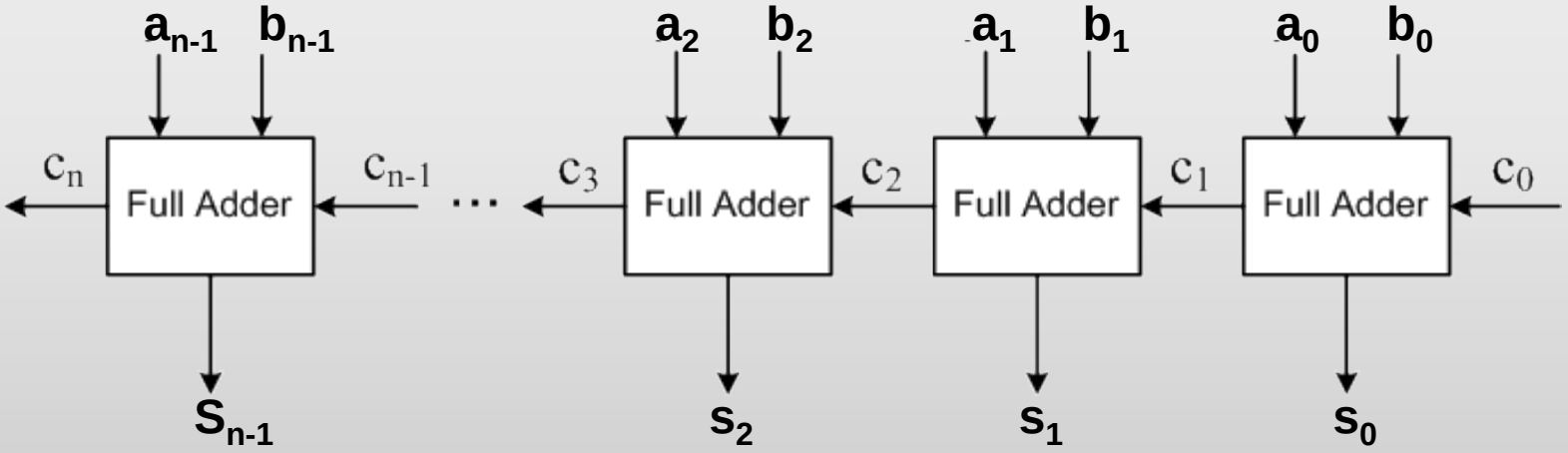
Ripple-Carry Addition

- When comparing adders we will simply compare the number of logic levels in each one
- A ripple-carry adder takes two levels to compute *c*1 from *a*0 and *b*0. Then it takes two more levels to compute *c*2 from *c*1, *a*1, *b*1, and so on, up to *cn*
- So, there are a total of 2*n* levels



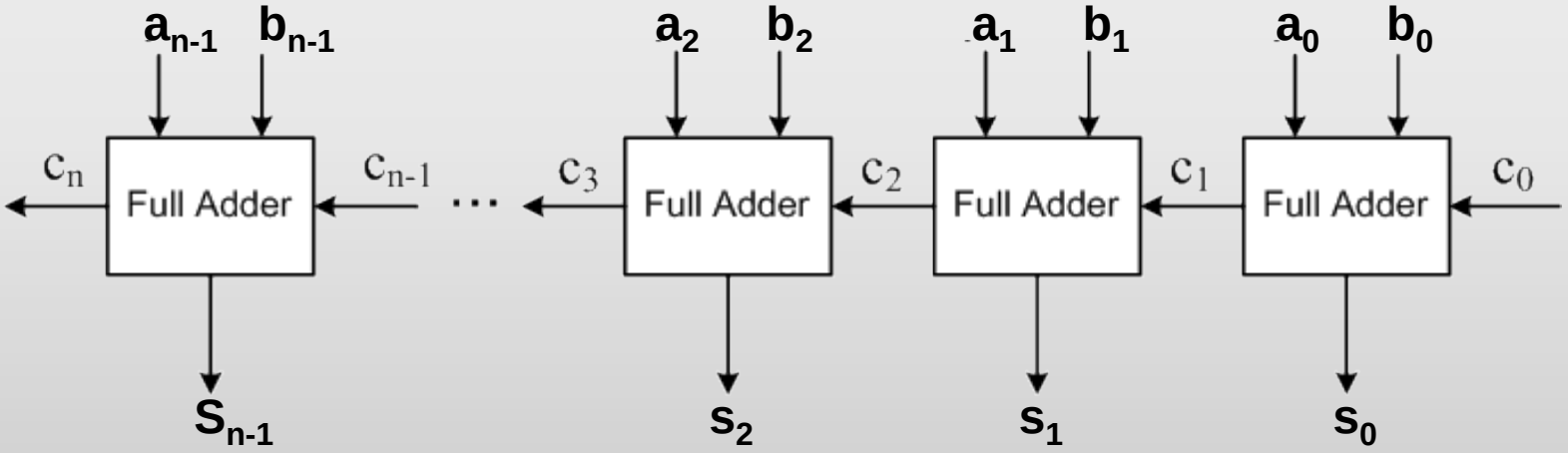
Ripple-Carry Addition

- Typical values of *n* are 32 for integer arithmetic and 53 for double-precision floating point
- The ripple-carry adder is the slowest adder, but also the cheapest
- It can be built with only *n* simple cells, connected in a simple, regular way



Ripple-Carry Addition

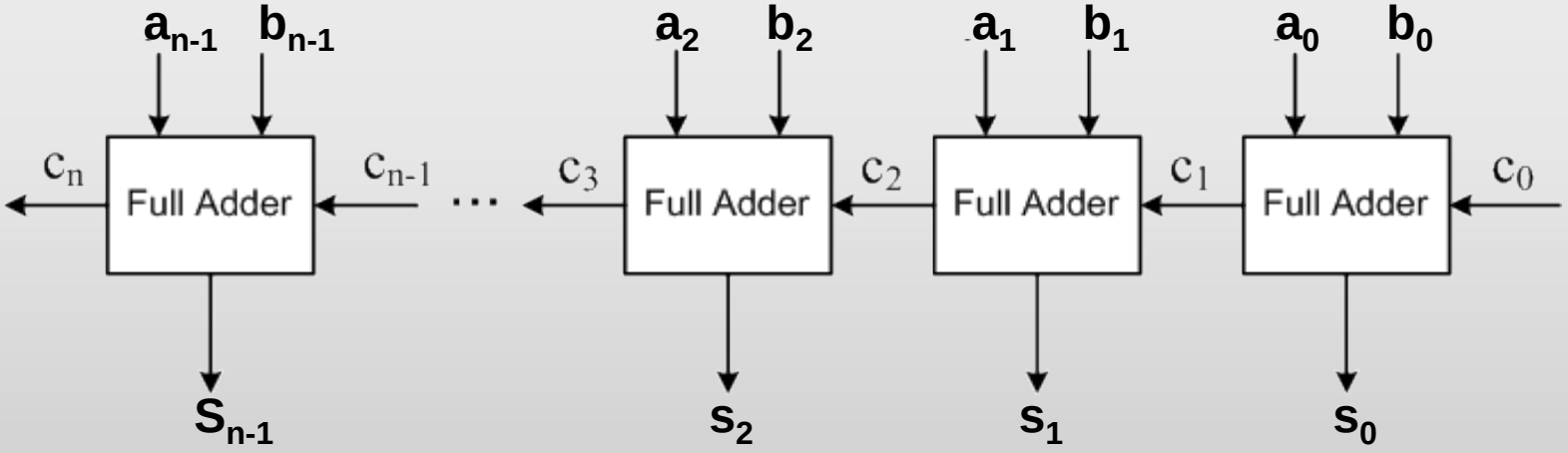
- The ripple-carry adder is relatively slow → it takes time O(*n*)
- But it is used because in technologies like CMOS, the constant factor is very small
- Short ripple adders are often used as building blocks in larger adders



Ripple-Carry Addition for Signed Numbers

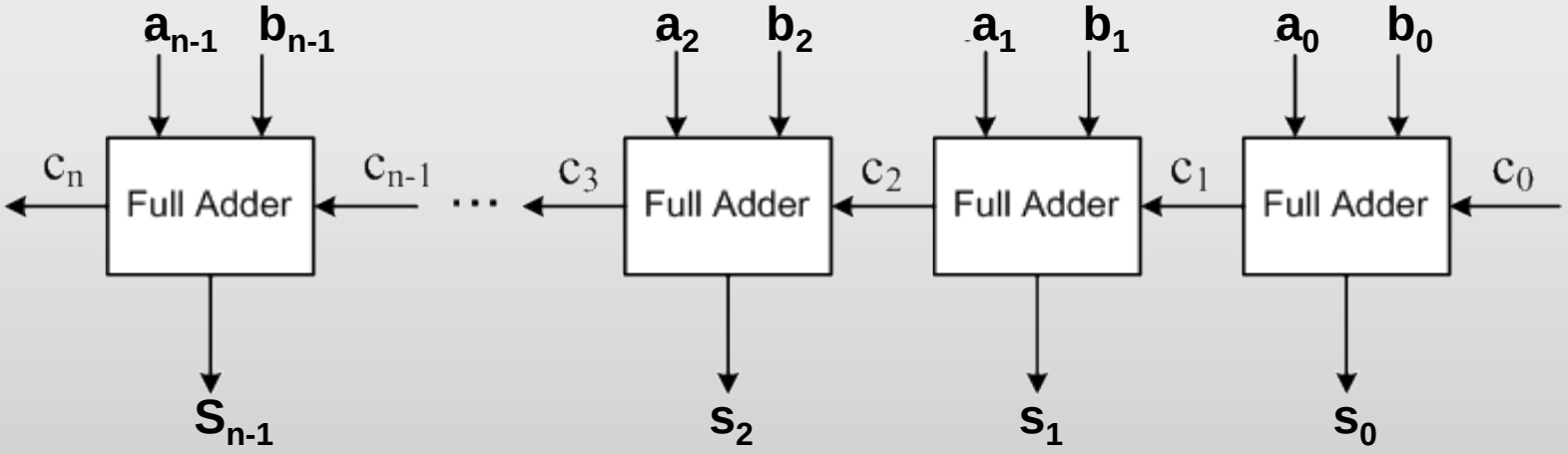
- The most widely used system for representing integers is the two's complement, where the MSB is considered associated with a negative weight
- The value of a two's complement number $a_{n-1}a_{n-2}\cdots a_1a_0$ is:

$$-a_{n-1}2^{n-1}+a_{n-2}2^{n-2}+\cdots+a_1 2^1+a_0 2^0$$



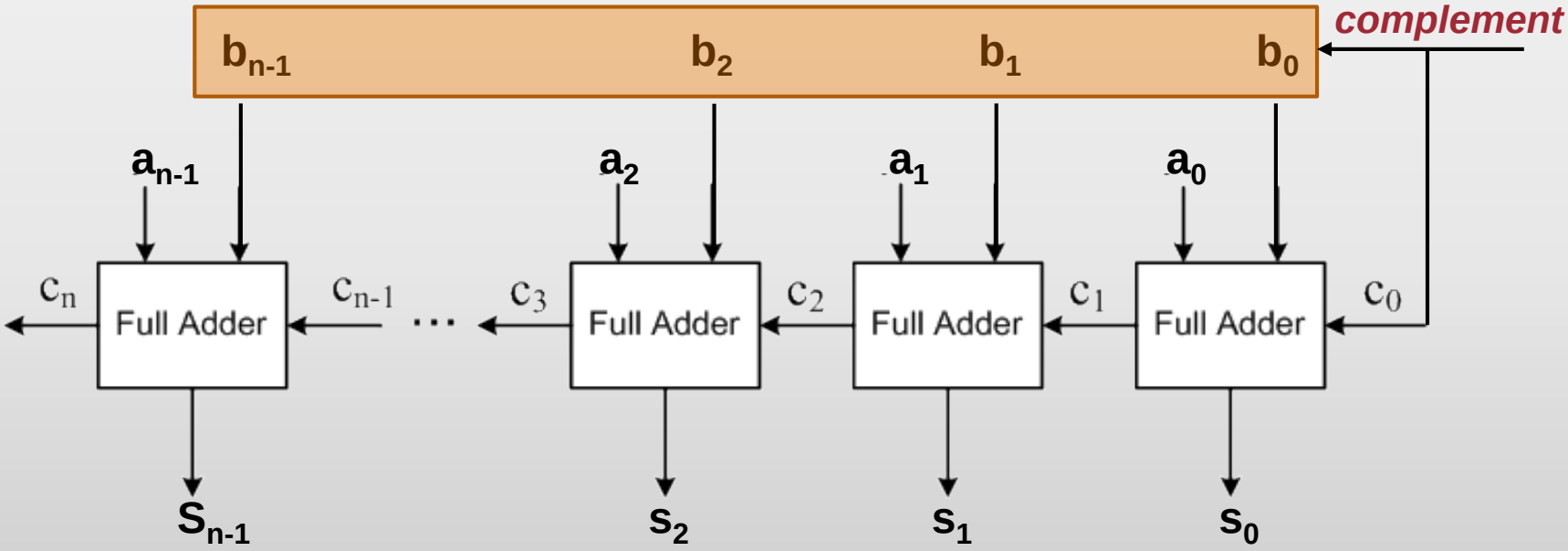
Ripple-Carry Addition for Signed Numbers

- One reason for the popularity of two's complement is that it makes signed addition easy → Simply discard the carry-out from the high order bit
- Subtraction is executed as an addition: A-B = A+(-B), recalling that– $X = \overline{X} + 1$



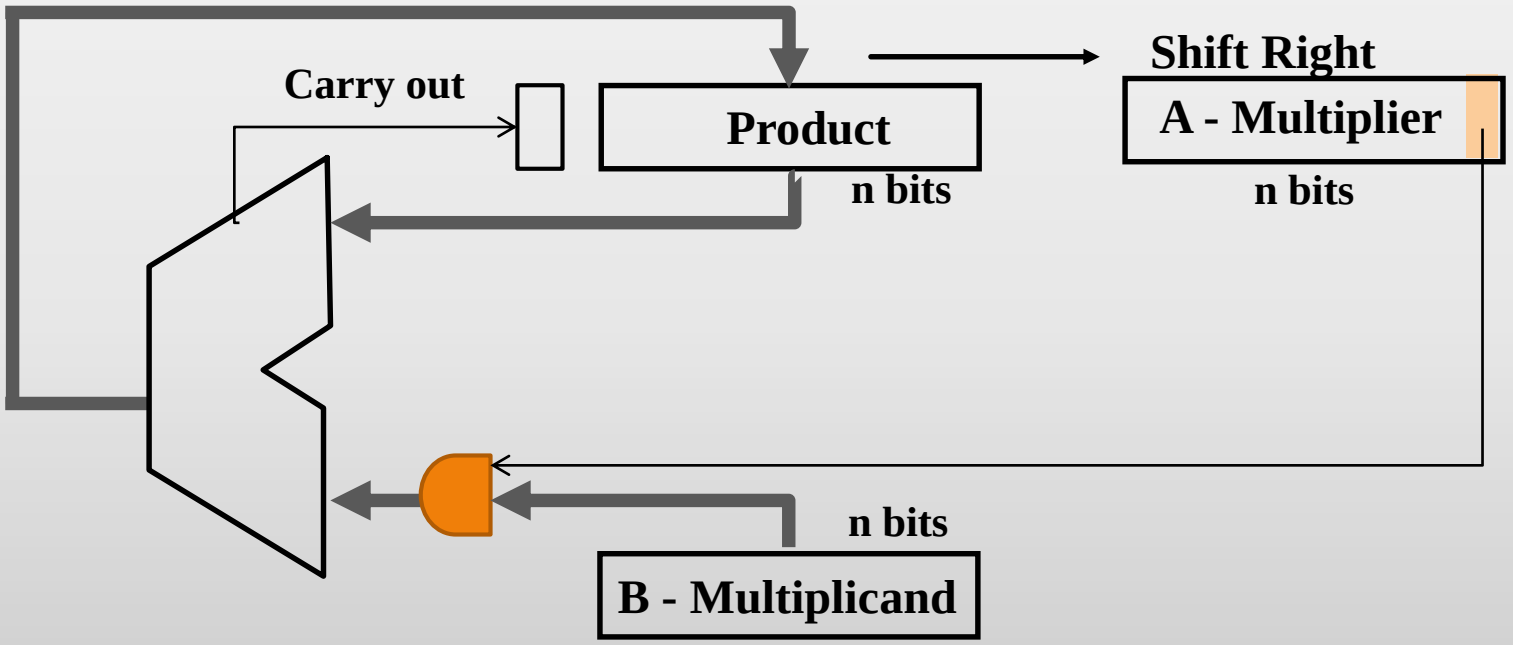
Ripple-Carry Addition for Signed Numbers

- The Ripple-Carry adder can be used also for subtraction acting on second operand B and on C_0
- If line ***complement*** is 1 then operand B is bit wise complemented and C_0=1



Unsigned Multiplication

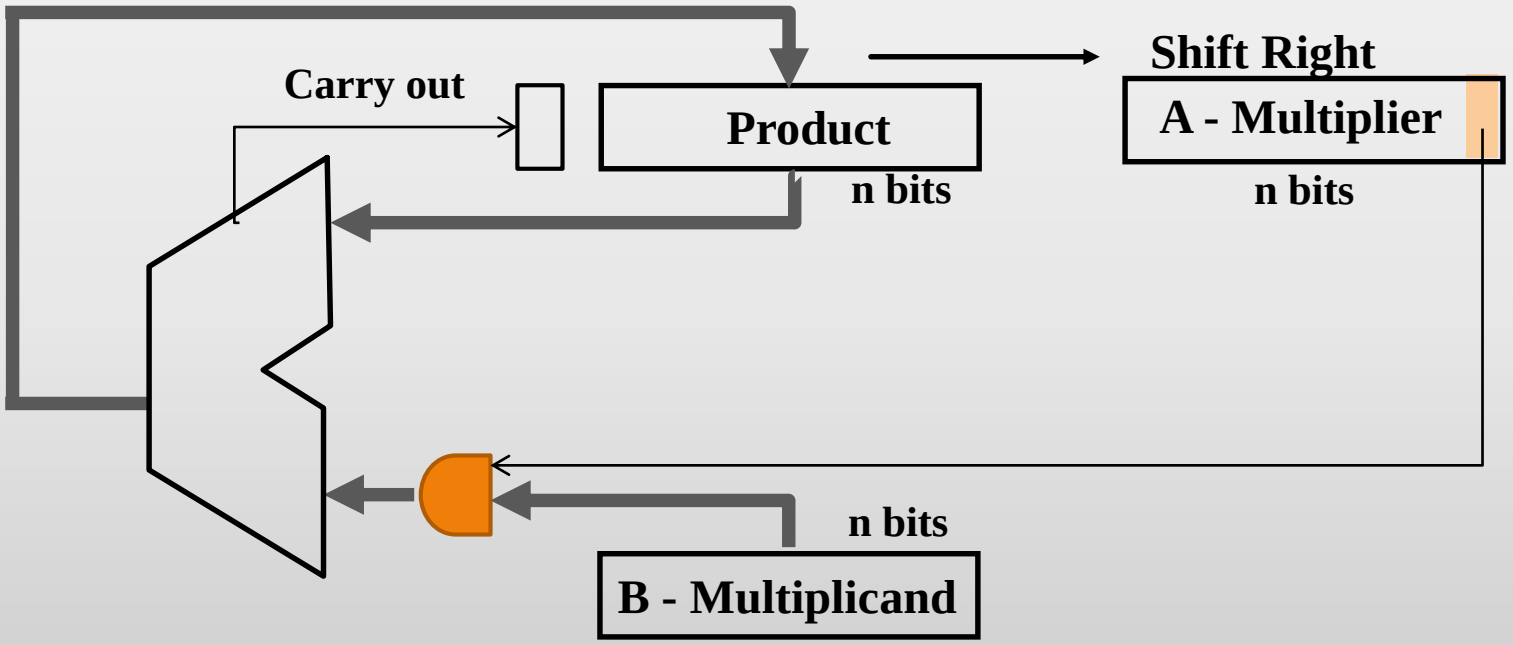
- The simplest multiplier computes the product of two unsigned numbers, $a_{n-1}a_{n-2} \cdots a_0$ and $b_{n-1}b_{n-2} \cdots b_0$, one bit at a time
- Register ***Product*** is initially 0



Unsigned Multiplication

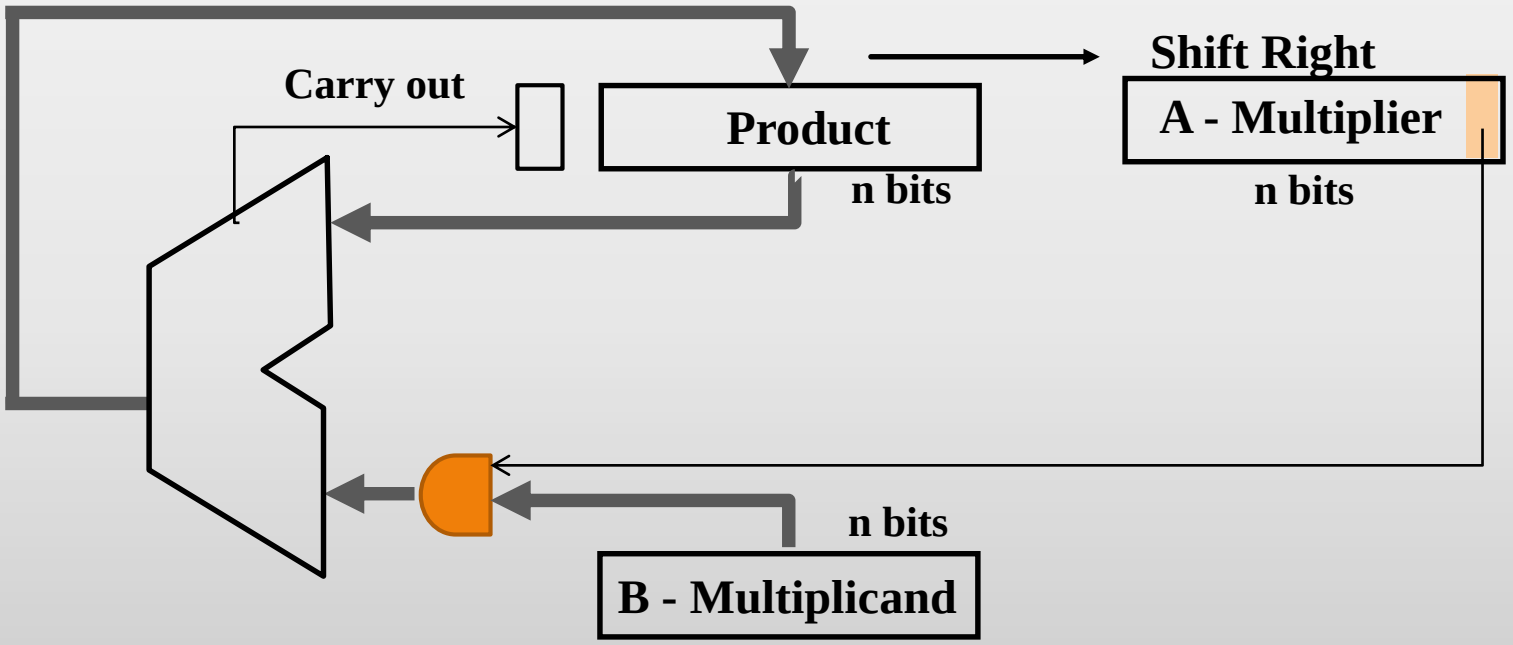
- Each multiply step has two parts:

  (i) If the least-significant bit of A is 1, then register B, containing $b_{n-1}b_{n-2} \cdots b_0$, is added to P; otherwise, $0 \cdots 00$ is added to P. The sum is placed back into P



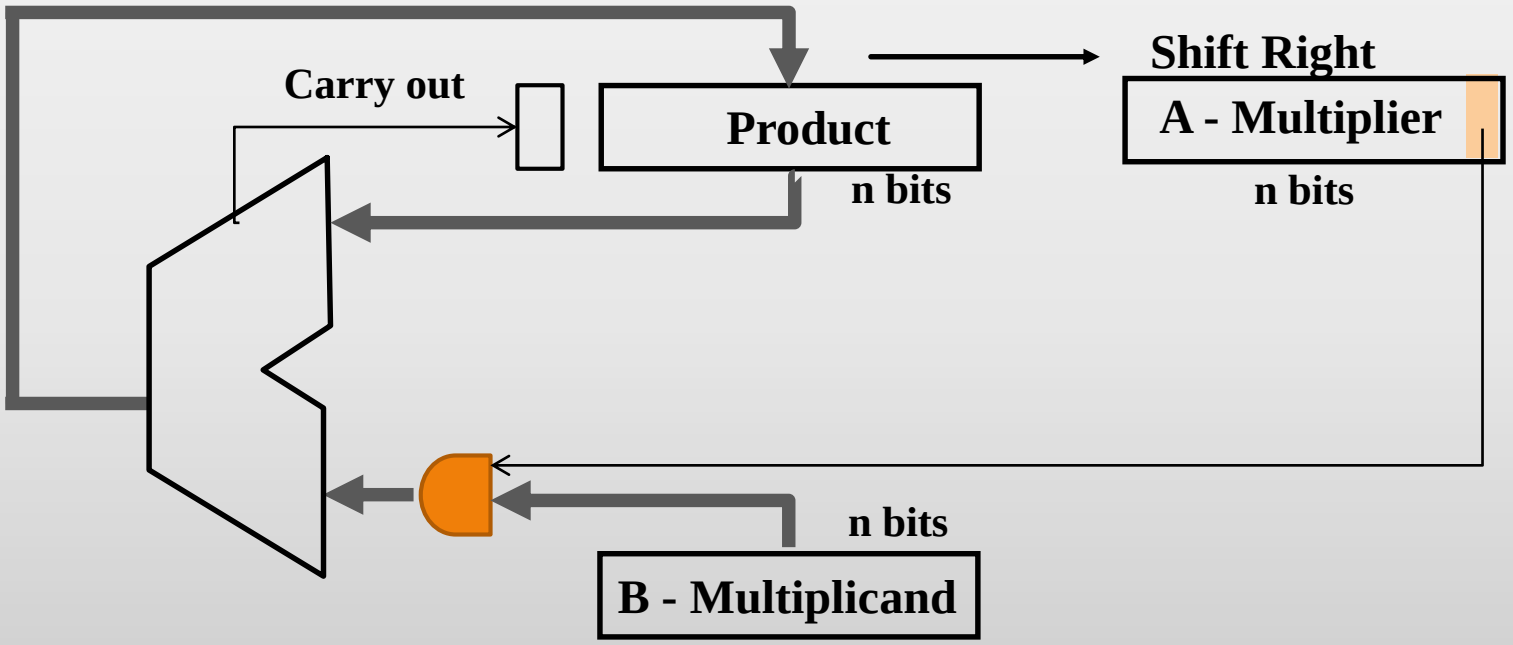
Unsigned Multiplication

(ii) Registers P and A are shifted right, with the carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved into register A, and the rightmost bit of A (not used in the rest of the algorithm) being shifted out



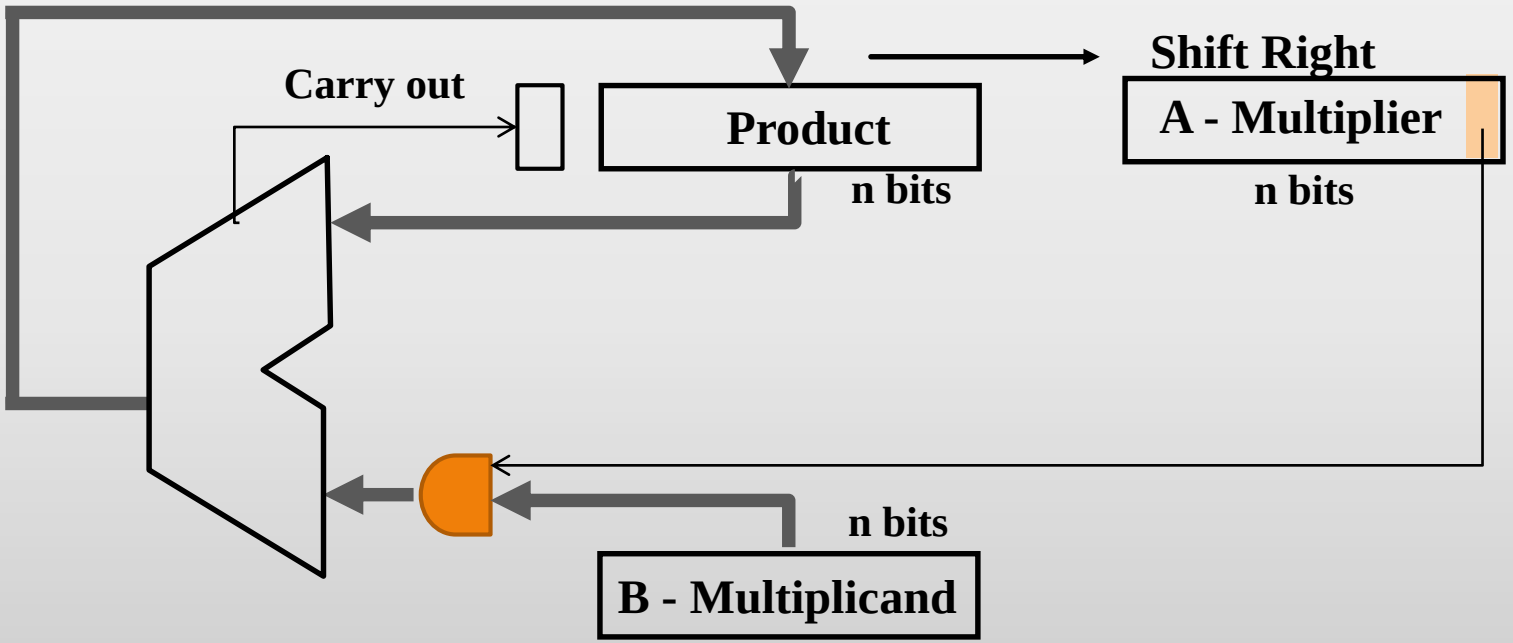
Unsigned Multiplication

- Hence, we add the contents of P to either B or 0 (depending on the low-order bit of A), replace P with the sum, and then shift both P and A one bit right
- After ***n* steps**, the product appears in registers P and A, with A holding the lower-order bits



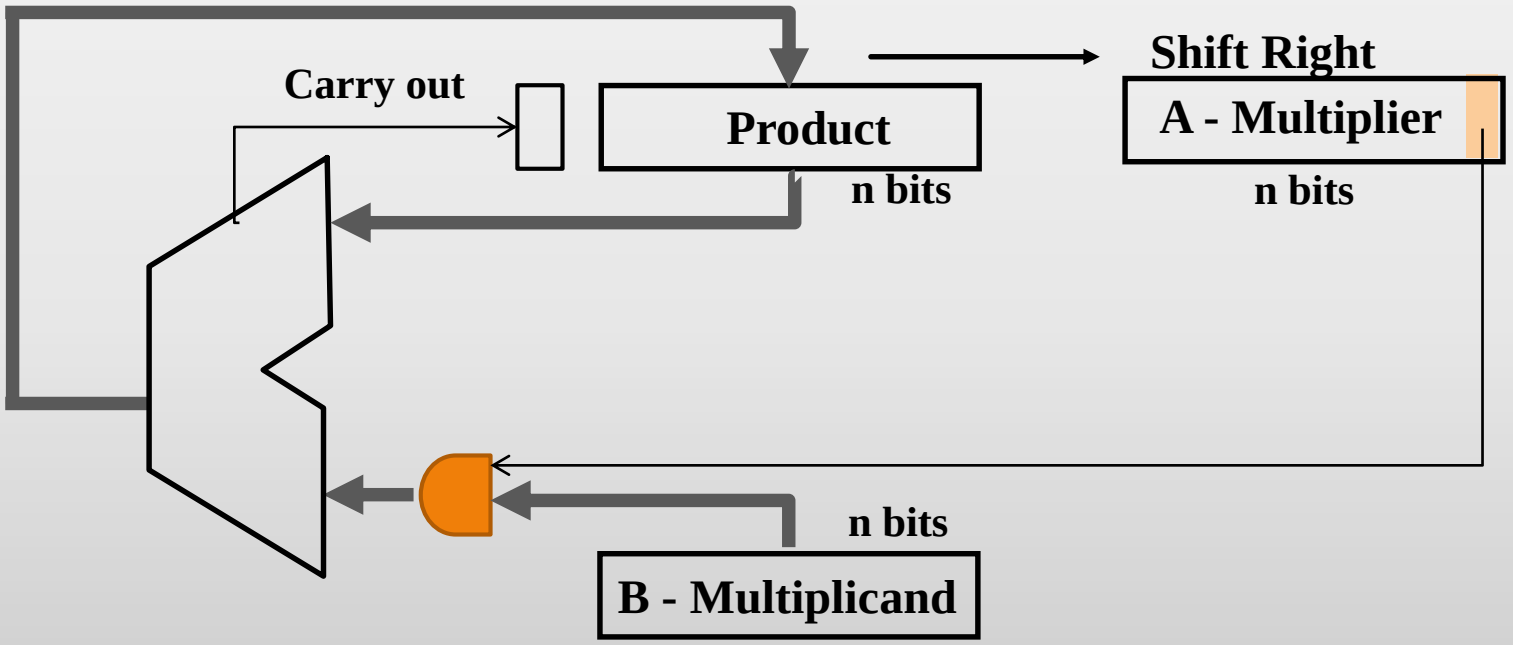
Signed Multiplication

- To multiply two's complement numbers, the obvious approach is to convert operands to be nonnegative, do an unsigned multiplication, and then (if the original operands were of opposite signs) negate the result
- This requires **extra time and hardware**



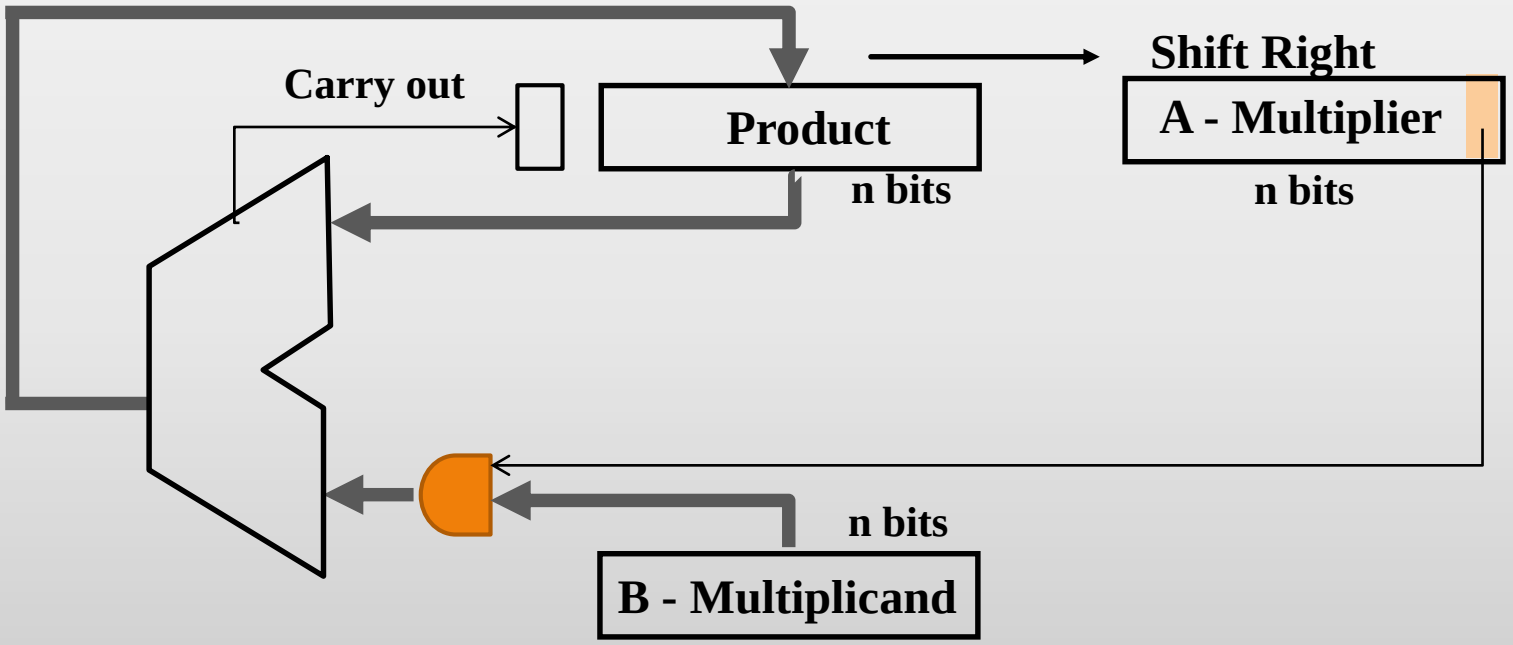
Signed Multiplication

- A better approach to multiply *A* and *B* using the hardware below:
  - If **B is potentially negative** but **A is nonnegative**, to convert the unsigned multiplication algorithm into a two's complement one we need that when P is shifted, it is shifted arithmetically



Signed Multiplication

- A better approach to multiply *A* and *B* using the hardware below:
  - If **A is negative,** the method is *Booth recoding* that is based on the fact that **any sequence of 1s in a binary number can be written as 011…11 = 100..00 - 1**



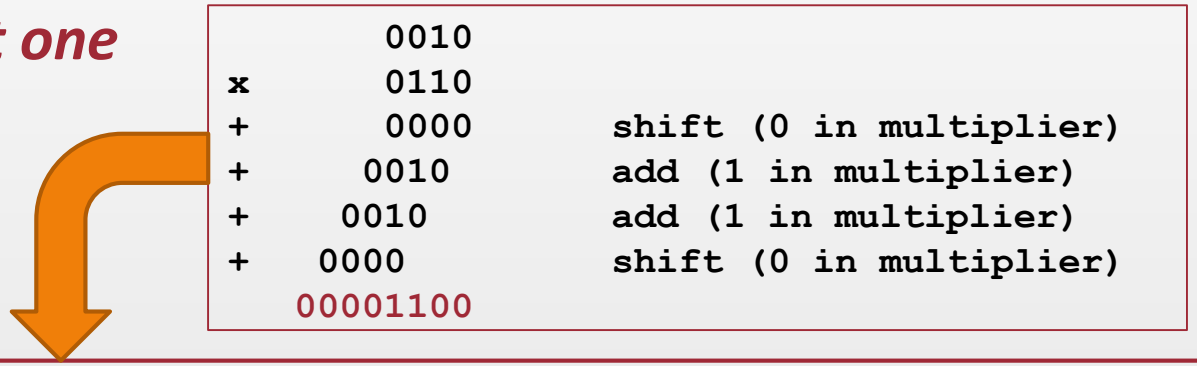
Signed Multiplication

- Then, we ***replace a string of 1s in multiplier*** with an initial ***subtract*** when we first see a one and then later ***add for the bit after the last one***

```
        0010
x       0110
-------------------------------------
+       0000    shift (0 in multiplier)
+      0010     add (1 in multiplier)
+     0010      add (1 in multiplier)
+    0000       shift (0 in multiplier)
-------------------------------------
  00001100
```

Signed Multiplication

- Then, we ***replace a string of 1s in multiplier*** with an initial ***subtract*** when we first see a one and then later ***add for the bit after the last one***




```
        0010
x       0110
-------------
+       0000    shift (0 in multiplier)
-       0010    sub(first 1 in multpl)
+      0000     shift(mid string of 1s)
+     0010       add(prior step had last 1)
-------------
    00001100
```

Signed Multiplication

- Hence, to deal with **negative values of A**, all that is required is to sometimes subtract B from P, instead of adding either B or 0 to P

- Rules: If the initial content of A is $a_{n-1} \cdots a_0$, then step (i) in the multiplication algorithm becomes:
  - If $ai = 0$ and $ai-1 = 0$, then add 0 to P
  - If $ai = 0$ and $ai-1 = 1$, then add B to P
  - If $ai = 1$ and $ai-1 = 0$, then subtract B from P
  - If $ai = 1$ and $ai-1 = 1$, then add 0 to P
  - For the first step, when $i = 0$, take $ai-1$ to be 0

Speeding Up Integer Addition

- Integer addition is the simplest operation and the most important
- Even for programs that don't do explicit arithmetic, addition must be performed to increment the program counter and to calculate addresses
- The delay of an N-bit ripple-carry adder is:

$$t_{ripple} = Nt_{FA}$$

where t_{FA} is the delay of a full adder

- There are different techniques to increase the speed of integer operations (that lead to faster floating point)

Carry-Lookahead Adder

- A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits

$$c_{i+1} = a_i b_i + (a_i \oplus b_i) c_i$$

We define:

Carry Generate $g_i = a_i b_i$

Carry propagate $p_i = a_i \oplus b_i$

Then the expression of the carry is: $c_{i+1} = g_i + p_i c_i$

And the expression of the sum is:

$$s_i = a_i \overline{b}_i \overline{c}_i + \overline{a}_i b_i \overline{c}_i + \overline{a}_i \overline{b}_i c_i + a_i b_i c_i = (a_i \oplus b_i) \oplus c_i = p_i \oplus c_i$$

Carry-Lookahead Adder

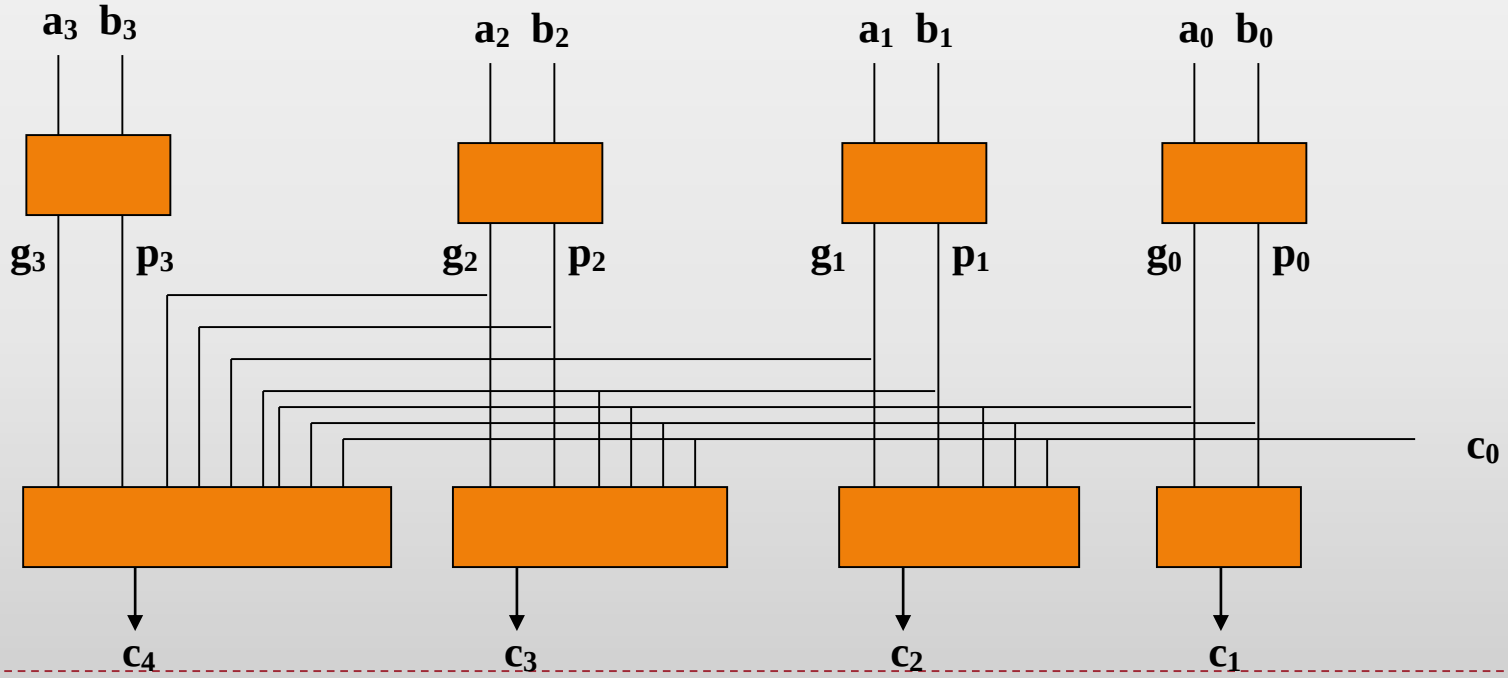
If we consider 4 bits, we have that c_1, c_2, c_3, c_4, depend only on c_0:

$c_1 = a_0b_0 + (a_0+b_0)c_0 = g_0 + p_0c_0$

$c_2 = a_1b_1 + (a_1+b_1)c_1 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$

$c_3 = a_2b_2 + (a_2+b_2)c_2 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

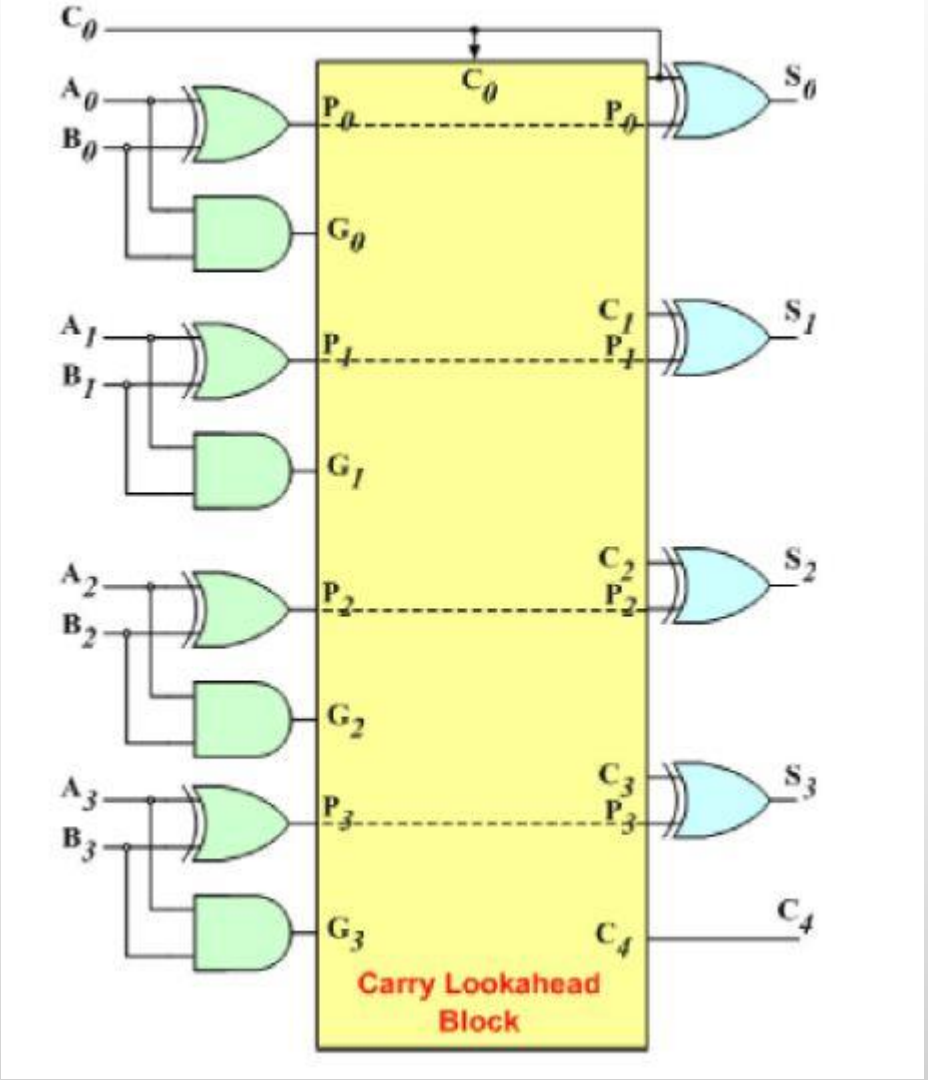
$c_4 = a_3b_3 + (a_3+b_3)c_3 = g_3 + p_3c_3 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$



Carry-Lookahead Addition

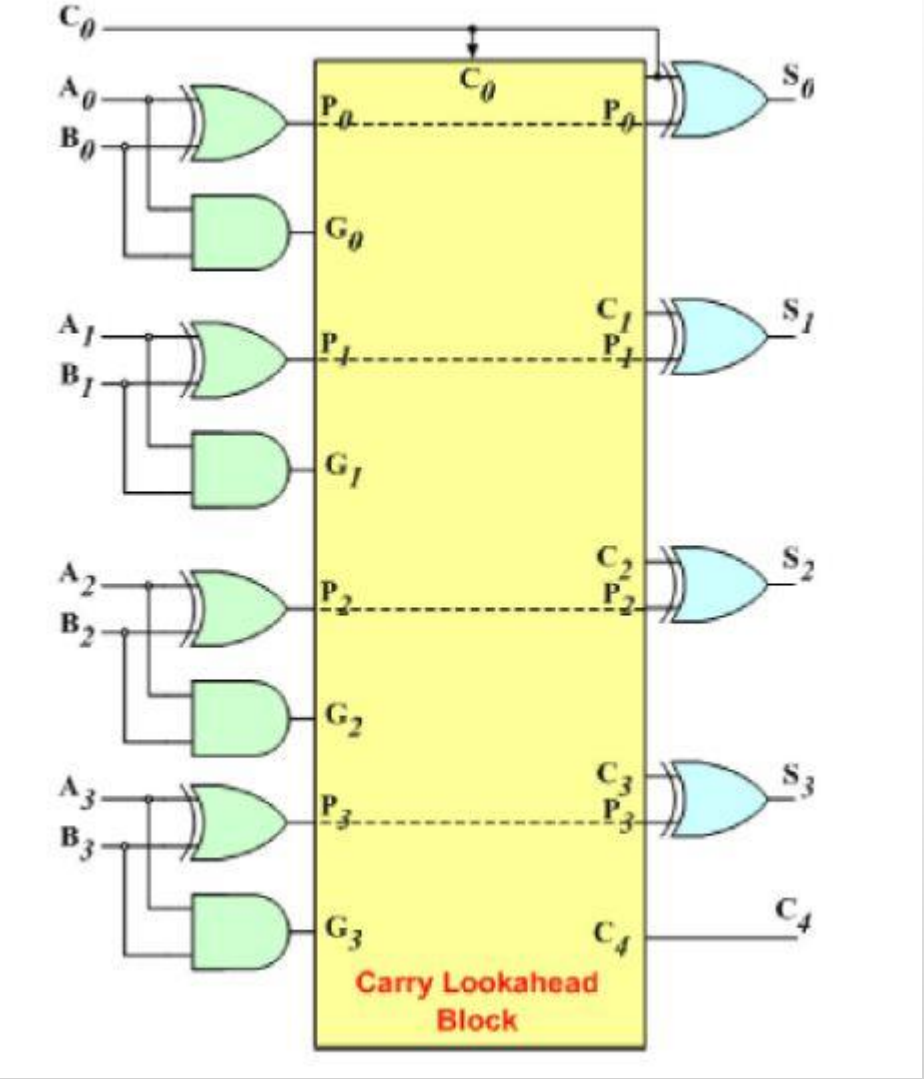
- Structure of a 4 bit CLA

- A CLA requires one logic level to form *p* and *g*, two levels to form the carries, and two for the sum, for total of **five logic levels** → improvement over the 2*n* levels required for the ripple-carry adder



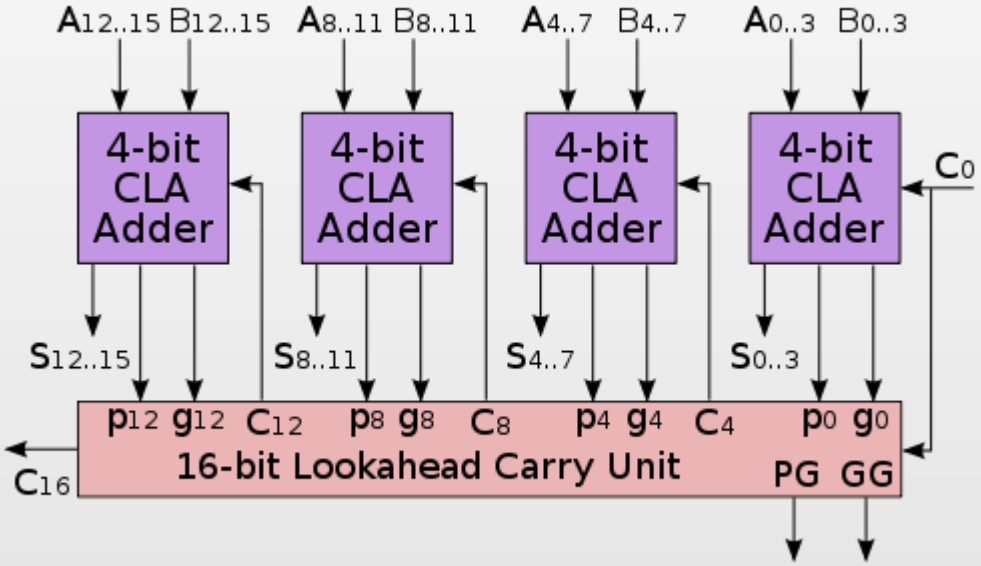
Carry-Lookahead Addition

- Unfortunately, as is evident, a carry-lookahead adder on *n* bits requires a fan-in of *n* + 1 at the OR gate as well as at the rightmost AND gate
- The irregular structure and long wires make it impractical to build a full carry-lookahead adder when *n* is large



Carry-Lookahead Addition

- A 16-bit adder can be built from four 4-bit adders, and a 4-bit carry look-ahead unit at the second level



- A 64-bit adder can be built from sixteen 4-bit adders, four 4-bit carry look-ahead units at the second level, and a single 4-bit carry look-ahead unit at the third level

Speeding Up Integer Multiplication

- Methods that increase the speed of multiplication can be divided into two classes:
  - single adder
  - multiple adders

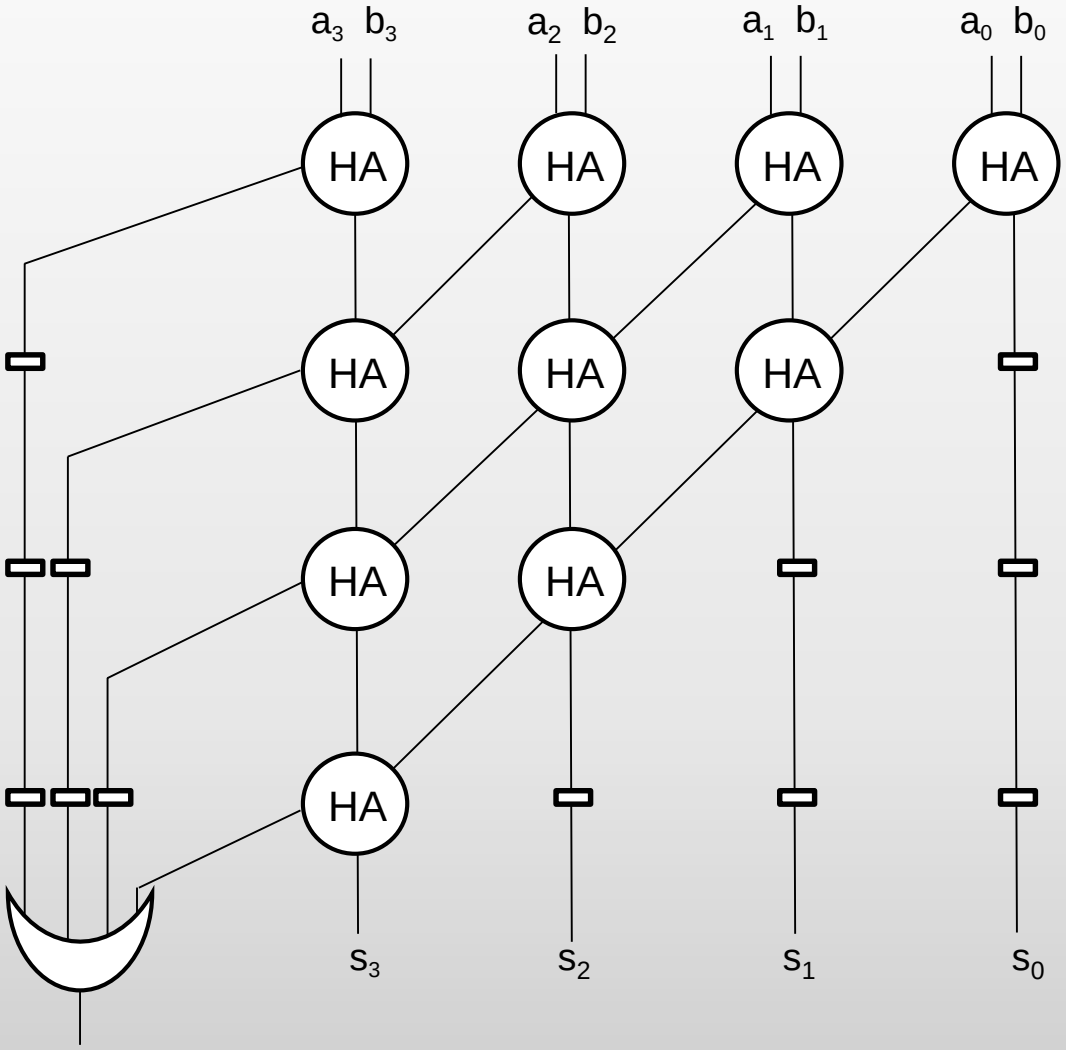
- In the simple multiplier we described, each multiplication step passes through the single adder
- The amount of computation in each step depends on the used adder
- If the space for many adders is available, then multiplication speed can be improved

Pipelined arithmetic

- Consider the instruction pipelining already described
- The processor goes through a repetitive cycle of fetching and processing instructions
- In the absence of hazards, the processor is continuously fetching instructions from sequential locations → the pipeline is kept full and a savings in time is achieved
- Similarly, a **pipelined ALU** will save time if it is fed a **stream of data** from sequential locations
- A single, isolated operation is not speeded up by pipeline
- The speedup is achieved when a vector of operands is presented to the units in the ALU

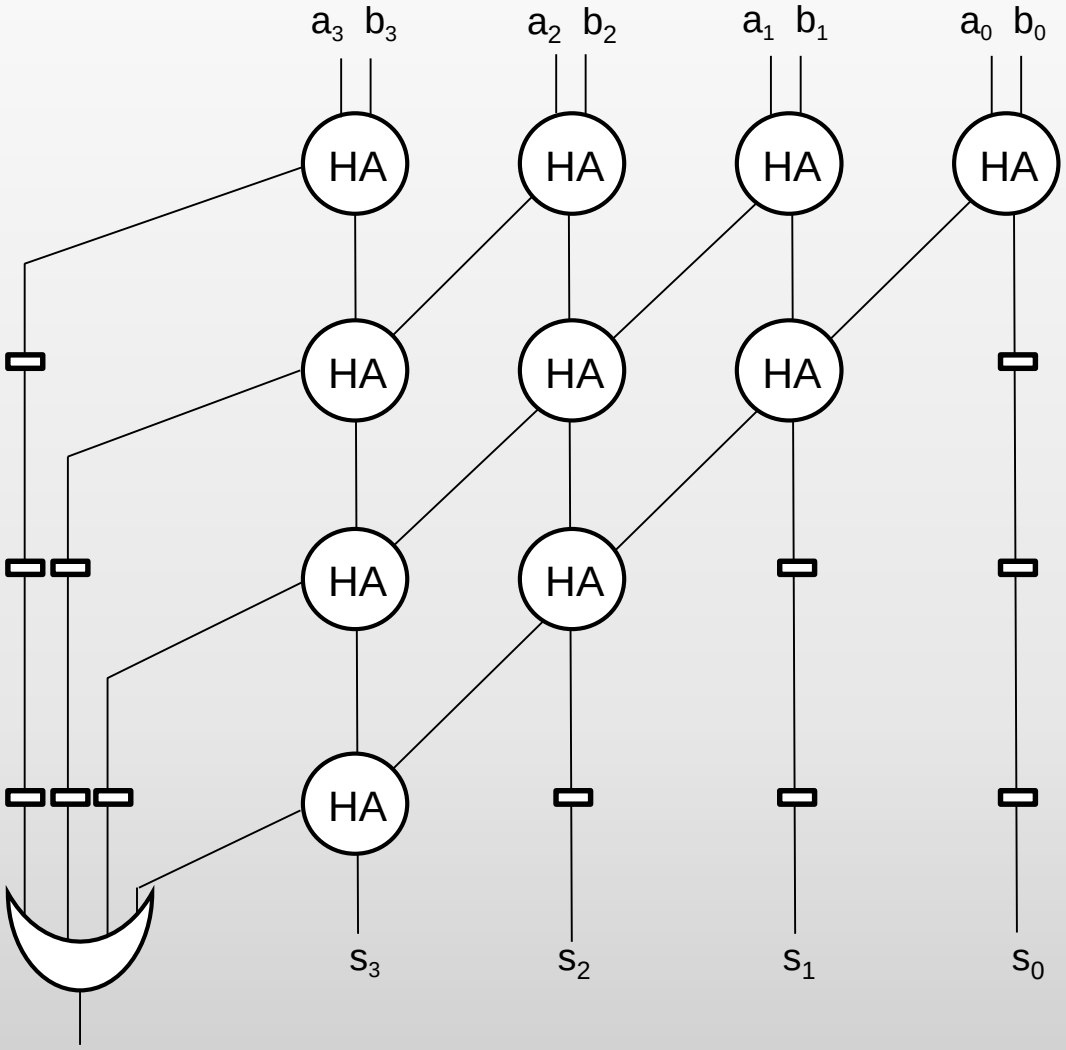
Pipelined Addition

- For n bits operands, a **pipeline adder** consists of n stages of half adders
- Registers are inserted at each stage to synchronize the computation
- At each clock cycle a new pair of operands is applied to the inputs of the adder



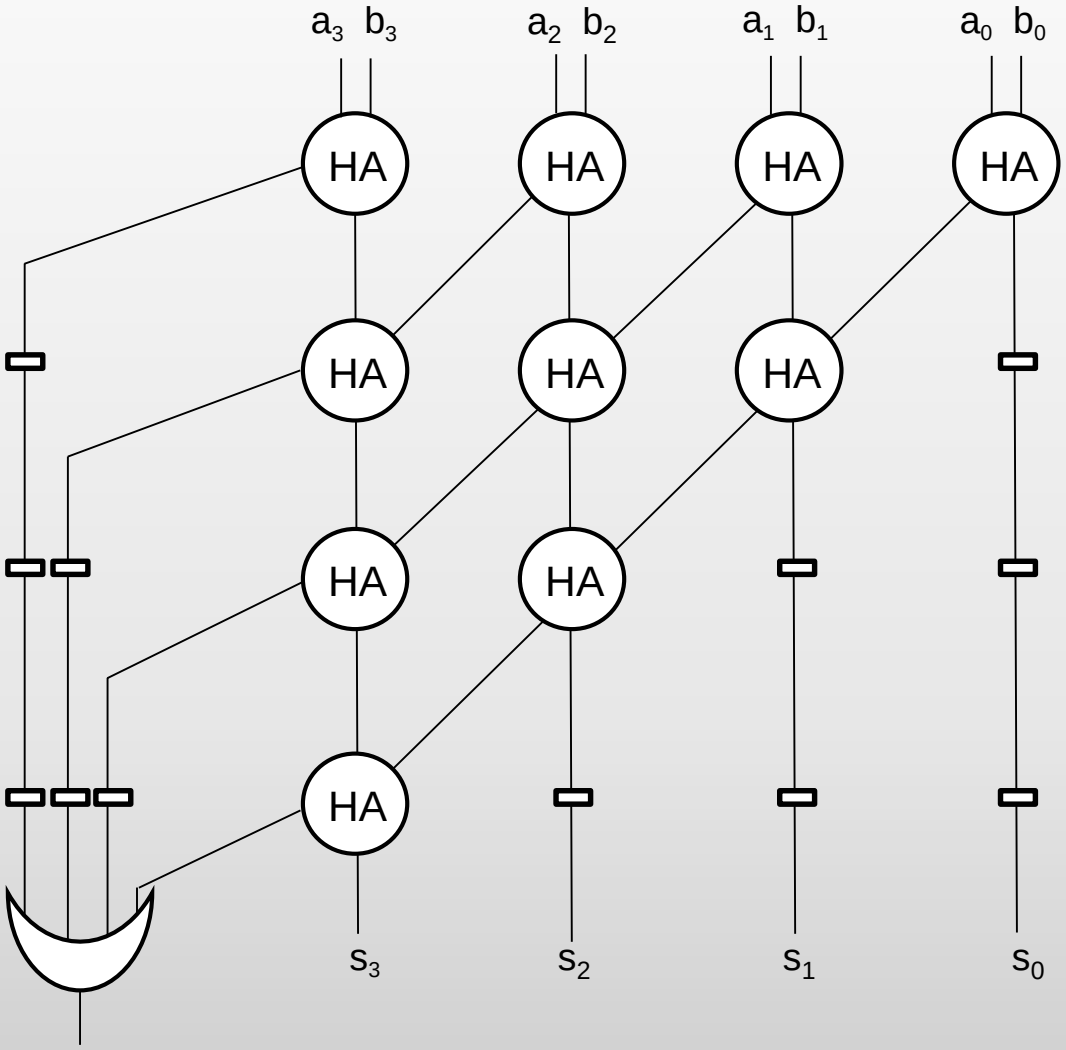
Pipelined Addition

- After ***n* clock cycles**, the sum of the first pair of operands is obtained
- The computing time for a single sum is the same of the carry-ripple adder
- A new sum is obtained at each clock cycle starting from the (*n*+1)-th clock cycle



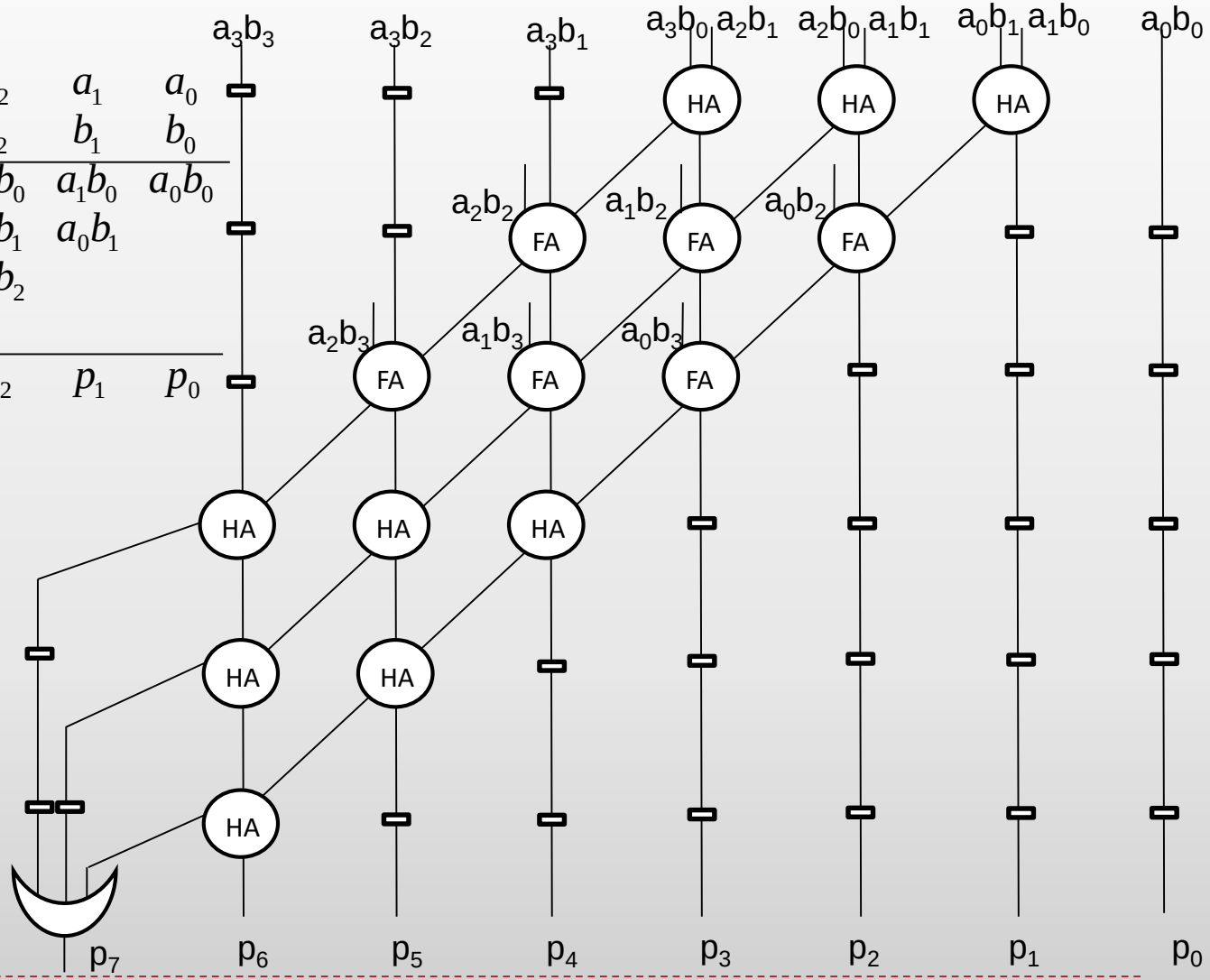
Pipelined Addition

- The number of **HA** is **O(n^2)**, whereas the circuit complexity of the carry-ripple adder is O(n)
- The added circuit complexity pays off if long sequences of numbers are being added



Pipelined Unsigned Multiplication

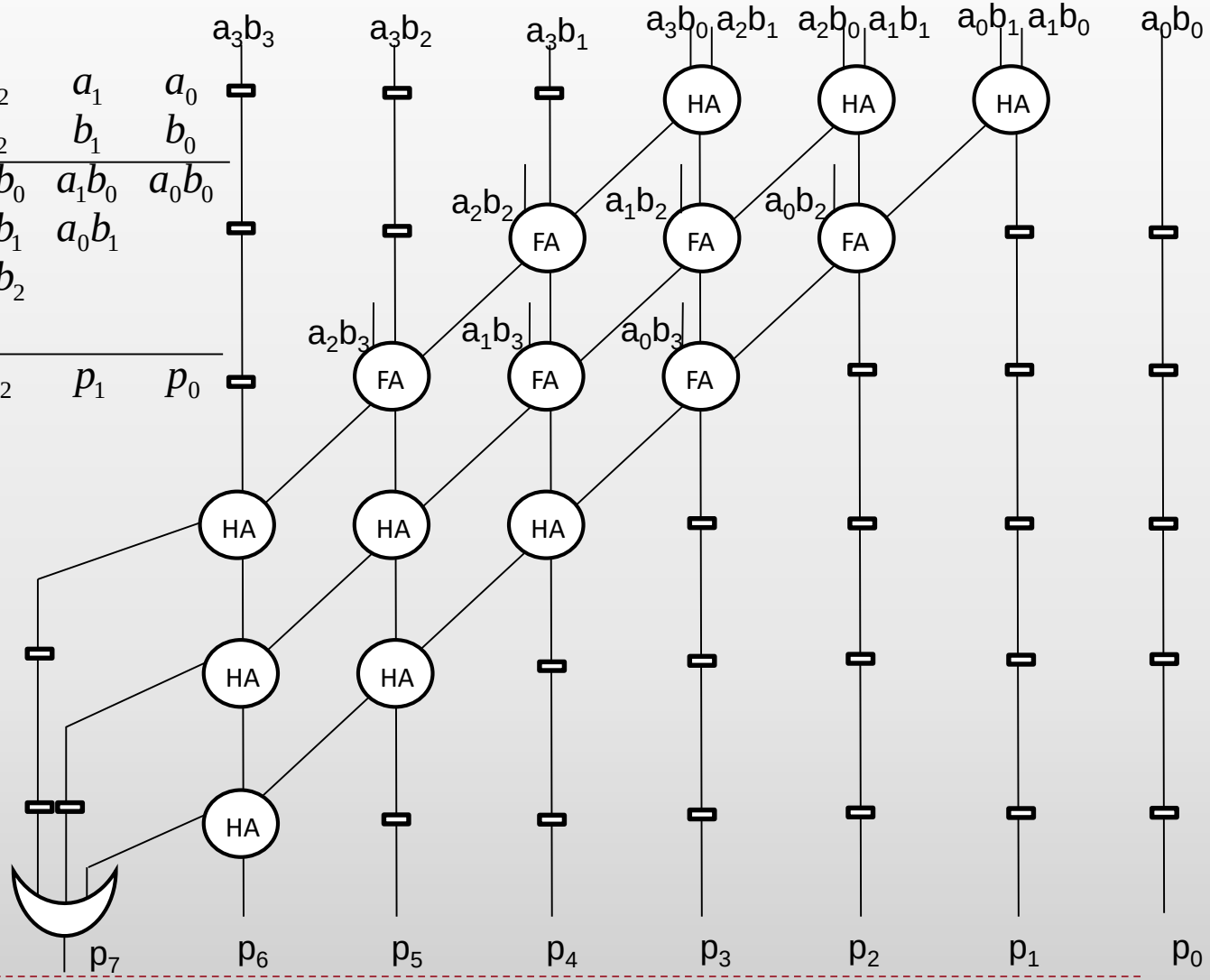
$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$



- The product of two *n* bit operands has length 2*n*
- Result is obtained by executing n-1 sums

Pipelined Unsigned Multiplication

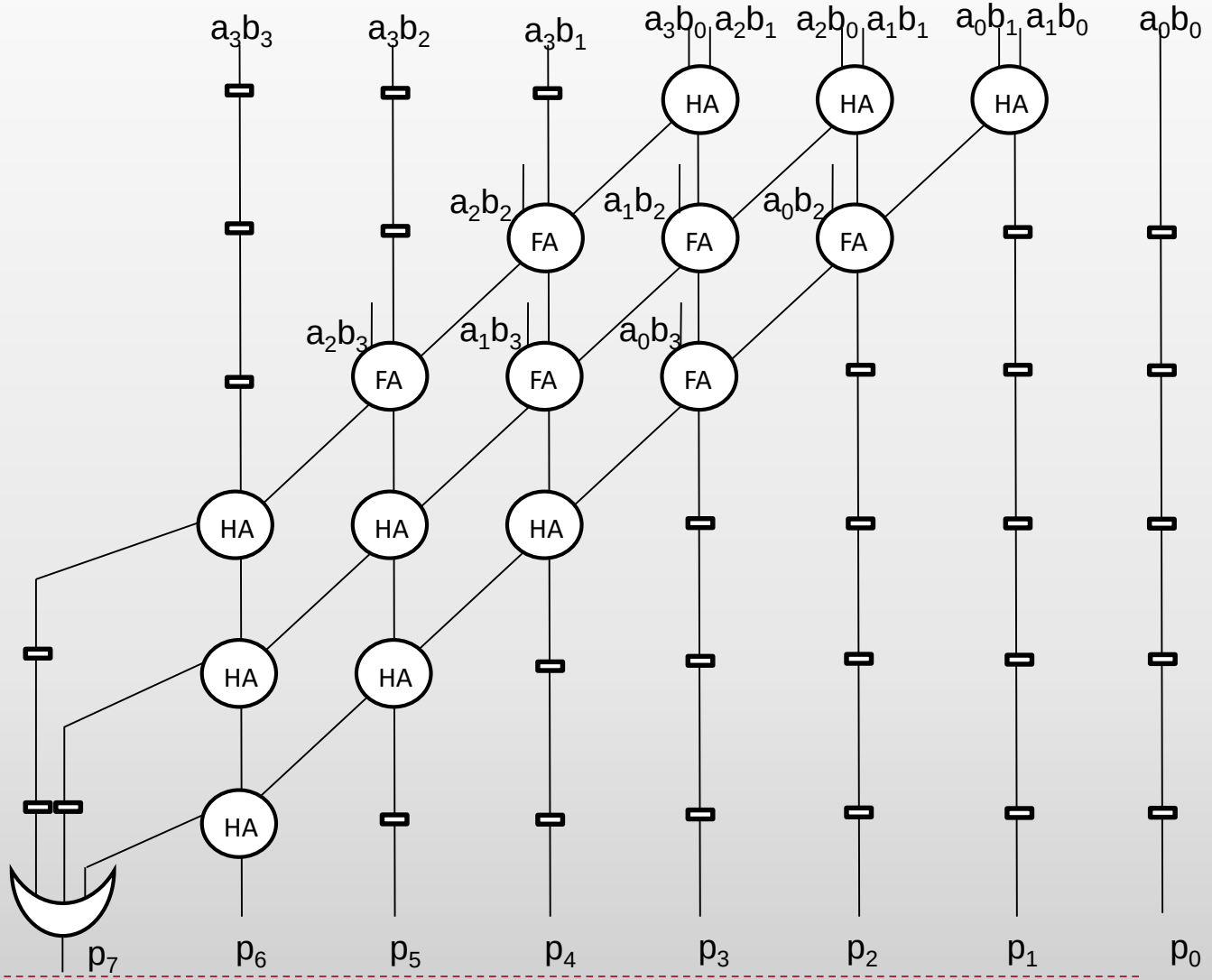
$$
\begin{array}{cccccccc}
 & & & & a_3 & a_2 & a_1 & a_0 \\
 & & & & b_3 & b_2 & b_1 & b_0 \\
\hline
 & & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
 & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\
\hline
p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
$$



- Inputs to the multiplier are logical AND among pairs of bits
- There are 2(n-1) stages of FA or HA

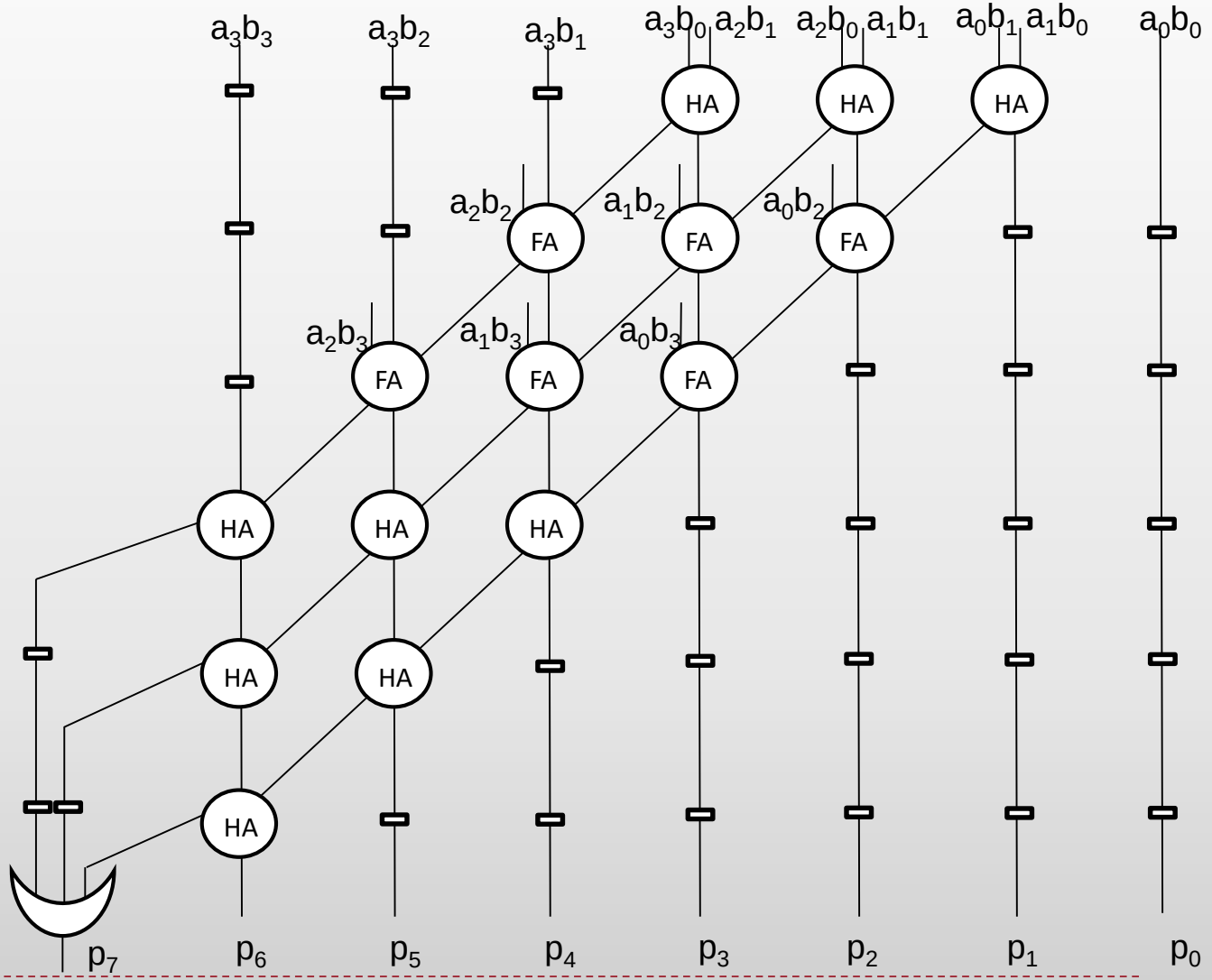
Pipelined Unsigned Multiplication

- After stage (n-1) all bit products (AND) are added
- Last (n-1) stages represent a pipelined adder
- Bit p_{2n-1} of the result is obtained as OR among the carries generated by the most left HA of each stage



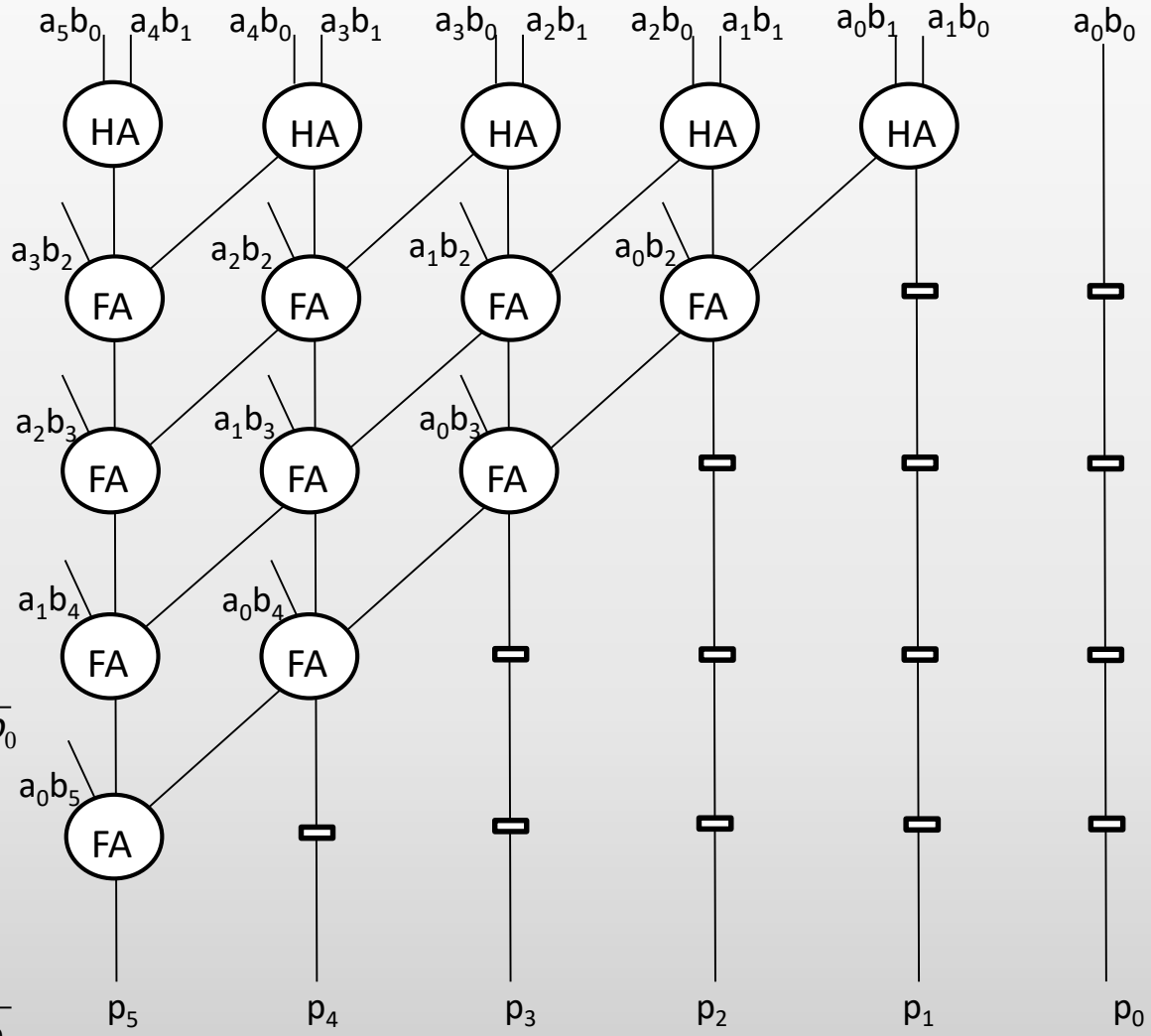
Pipelined Unsigned Multiplication

- After 2(*n*-1) clock cycles, the product of the first pair of operands is obtained
- A new result is obtained at each clock cycle starting from the **(2*n*-1)-th** clock cycle



Pipelined Signed Multiplication

- Signed numbers are extended to the length $2n$ of the product and used as operands



		a_5	a_4	a_3	a_2	a_1	a_0
		b_5	b_4	b_3	b_2	b_1	b_0
		a_5b_0	a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
	a_5b_1	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
a_5b_2	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
a_3b_4	a_2b_4	a_1b_4	a_0b_4				
a_2b_5	a_1b_5	a_0b_5					
		p_5	p_4	p_3	p_2	p_1	p_0

Pipelined Signed Multiplication

- Partial products of length $2n$ are considered (the remaining part is ignored)
- All stages but the first consists of FAs

		a_5	a_4	a_3	a_2	a_1	a_0
		b_5	b_4	b_3	b_2	b_1	b_0
		a_5b_0	a_4b_0	a_3b_0	a_2b_0	a_1b_0	a_0b_0
	a_5b_1	a_4b_1	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
a_5b_2	a_4b_2	a_3b_2	a_2b_2	a_1b_2	a_0b_2		
a_4b_3	a_3b_3	a_2b_3	a_1b_3	a_0b_3			
a_3b_4	a_2b_4	a_1b_4	a_0b_4				
a_2b_5	a_1b_5	a_0b_5					
		p_5	p_4	p_3	p_2	p_1	p_0