

# COMPUTER ARITHMETIC

---

Intensive Computation

**Annalisa Massini**

2022-2023

*Lecture 3-4*

# References

Computer Architecture - A Quantitative Approach

Hennessy Patterson – *Fifth Edition*

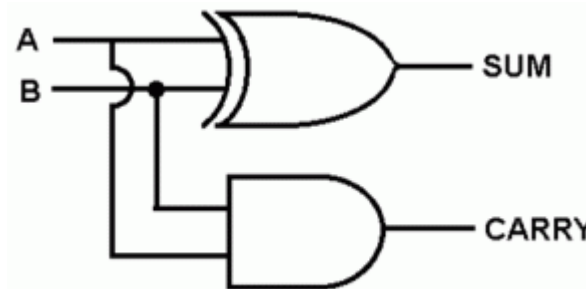
**Appendix J – *Computer arithmetic* - David Goldberg**

# ADDITION AND ADDERS

---

# 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
- Logic equations:  $s = a\bar{b} + \bar{a}b = a \oplus b$  and  $c_{out} = ab$



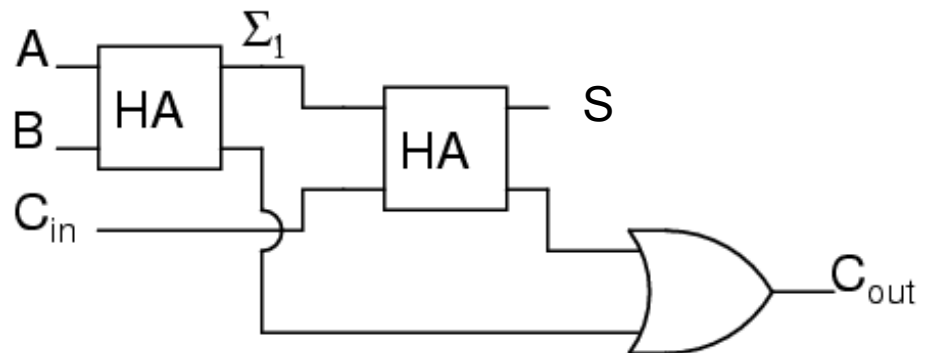
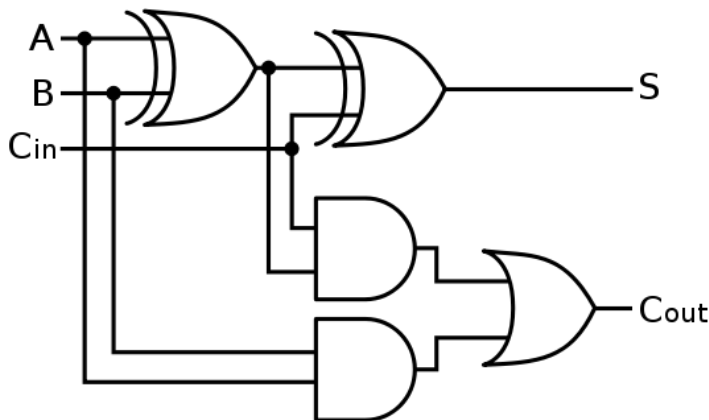
# Half adder and Full adder

- 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

- Logic equations:

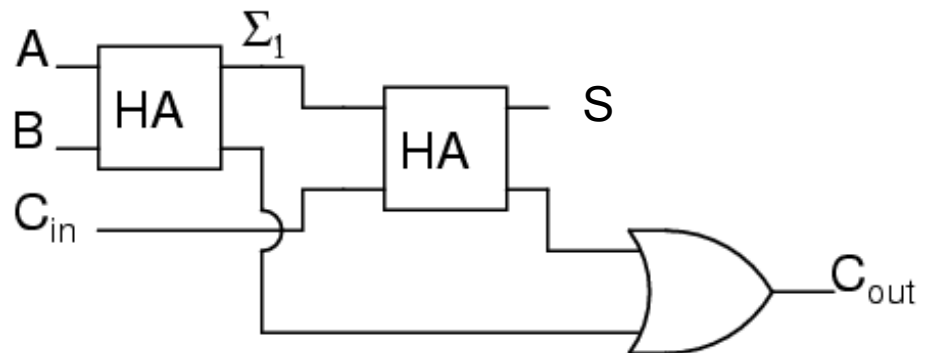
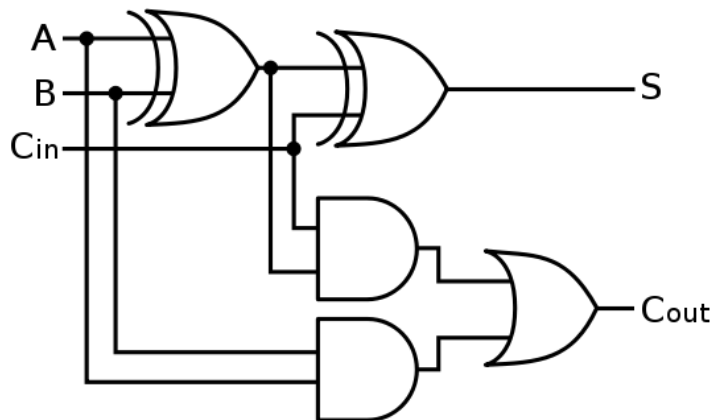
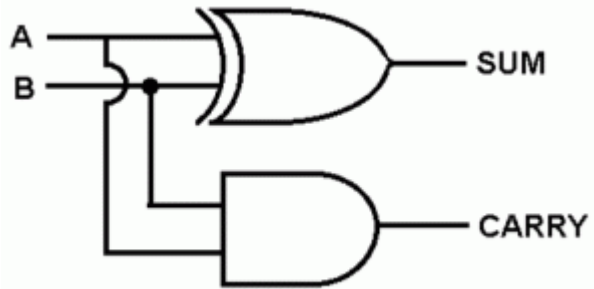
$$s = \overline{a}\overline{b}c + \overline{a}b\overline{c} + a\overline{b}\overline{c} + abc = (a \oplus b) \oplus c$$

$$\text{and } c_{out} = (a \oplus b)c + ab$$



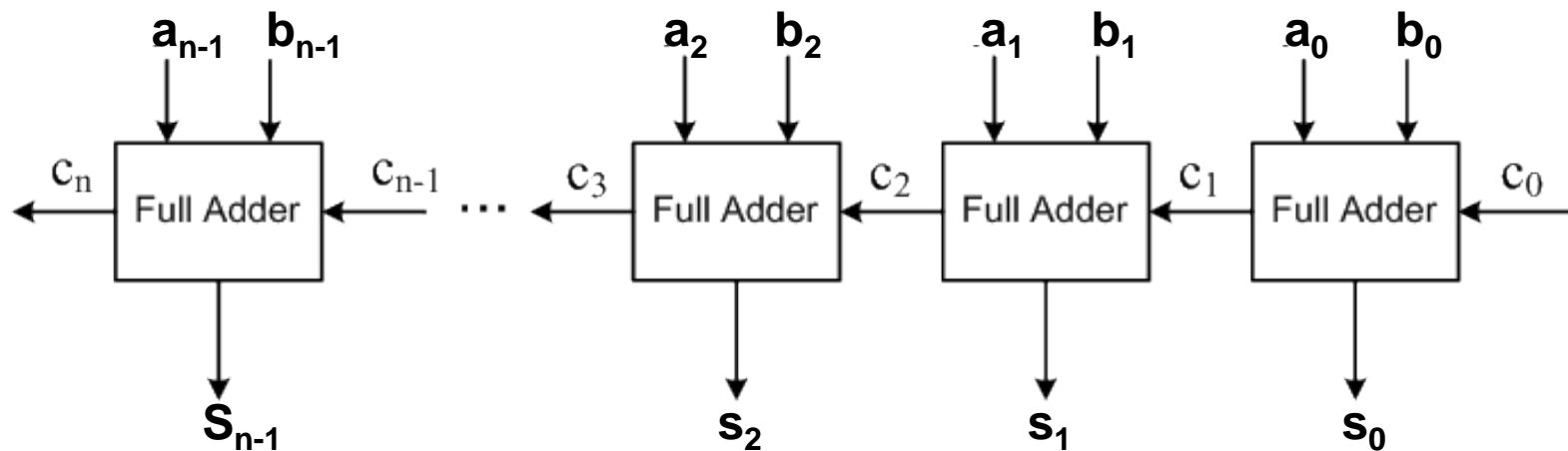
# Half adder and Full adder

- The **half adder** is a (2,2) adder:
  - it takes **two inputs** and produces **two outputs**
- The **full adder** is a (3,2) adder:
  - it takes **three inputs** and produces **two outputs**



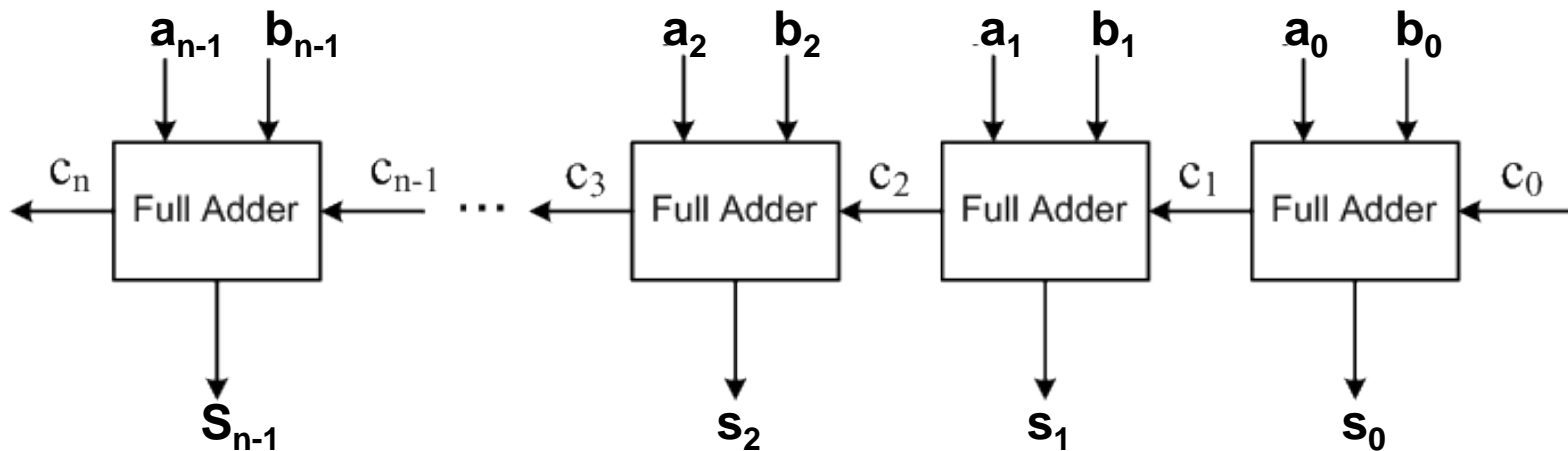
# Ripple-Carry Addition

- 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

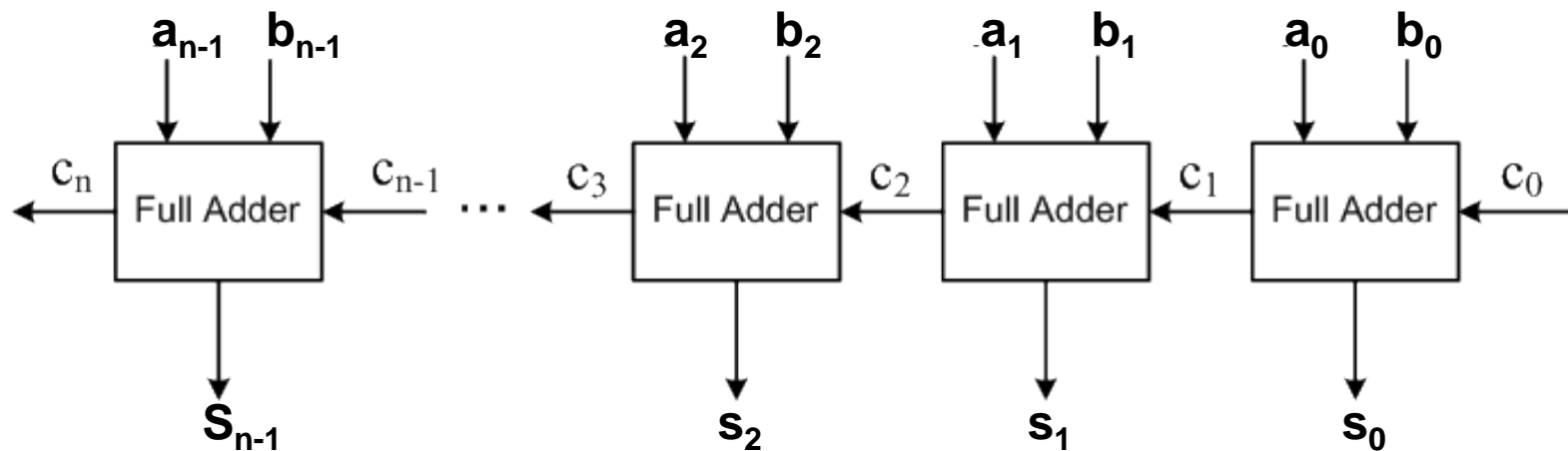
- Note that the low-order carry-in could be wired to 0, hence the **low-order adder** could be a half adder
- However, setting the low-order carry-in bit to 1 is useful for performing **subtraction**





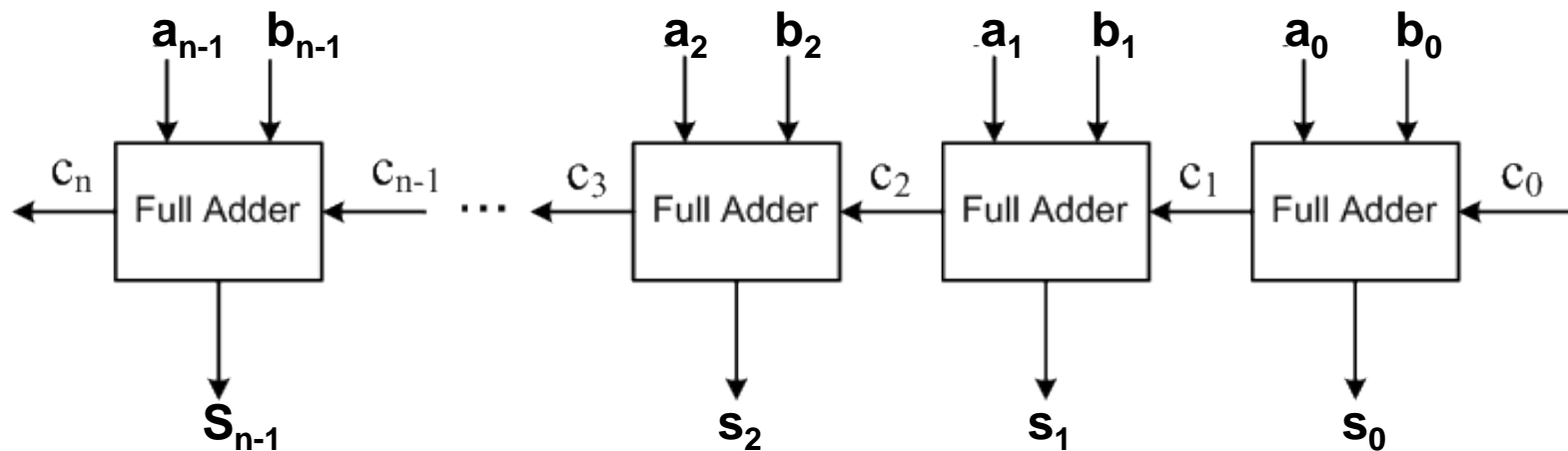
# Ripple-Carry Addition

- 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



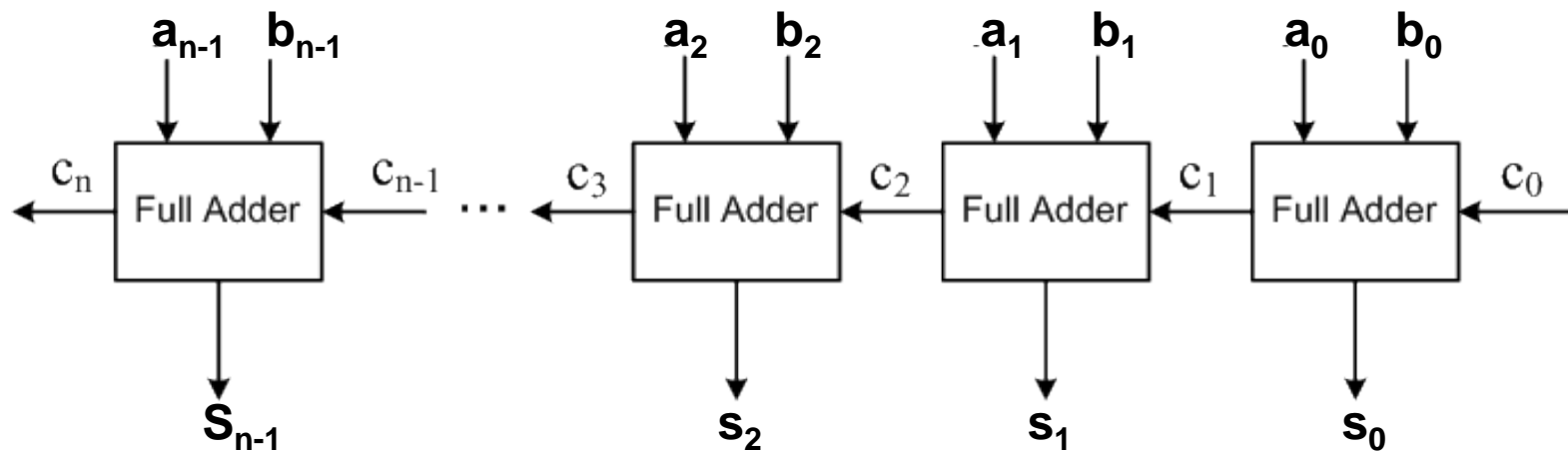
# Ripple-Carry Addition

- When **comparing adders** we 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  $b_0$
  - two more levels to compute  $c_2$  from  $c_1$ ,  $a_1$ ,  $b_1$  - *and so on*, up to  $c_n$
- So, there are a total of  $2n$  levels



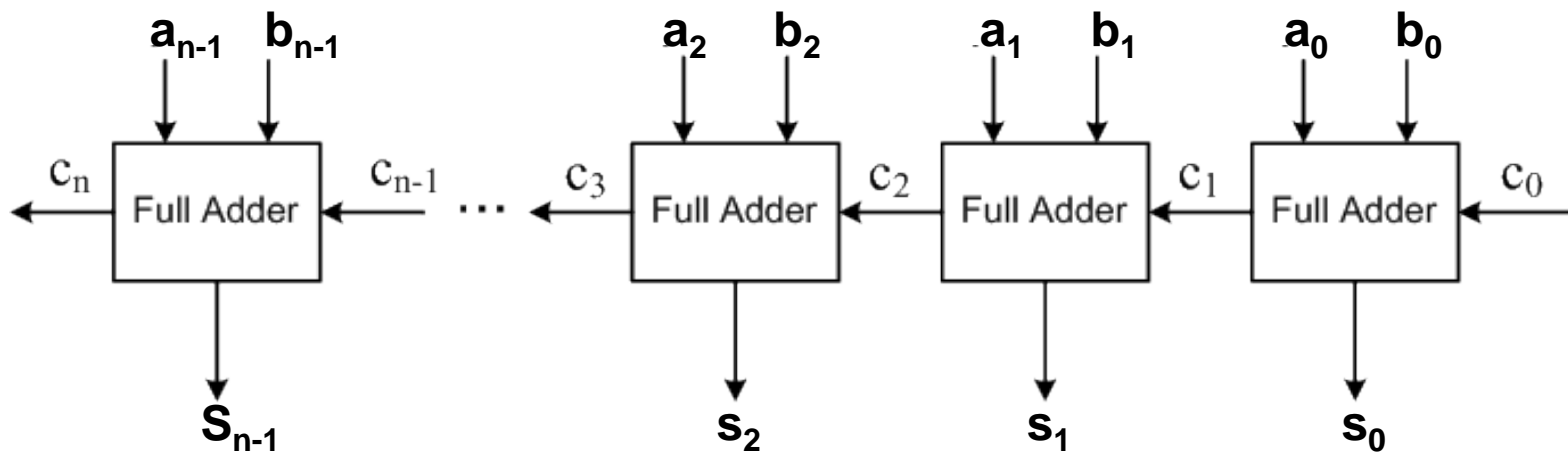
# Ripple-Carry Addition

- 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

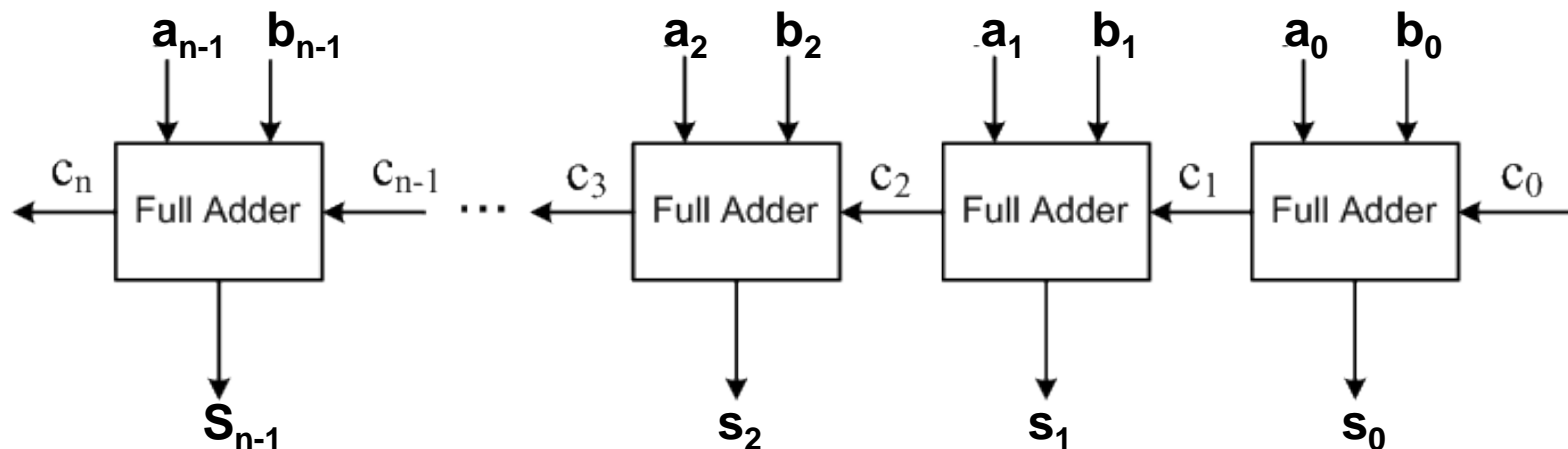
- The ripple-carry adder is relatively slow  $\rightarrow$  it takes time  $O(n)$
- But it is used because in technologies like CMOS, the constant factor is very small
- **Short ripple adders** are often used as **building blocks** in larger adders



# Ripple-Carry Addition for Signed Numbers

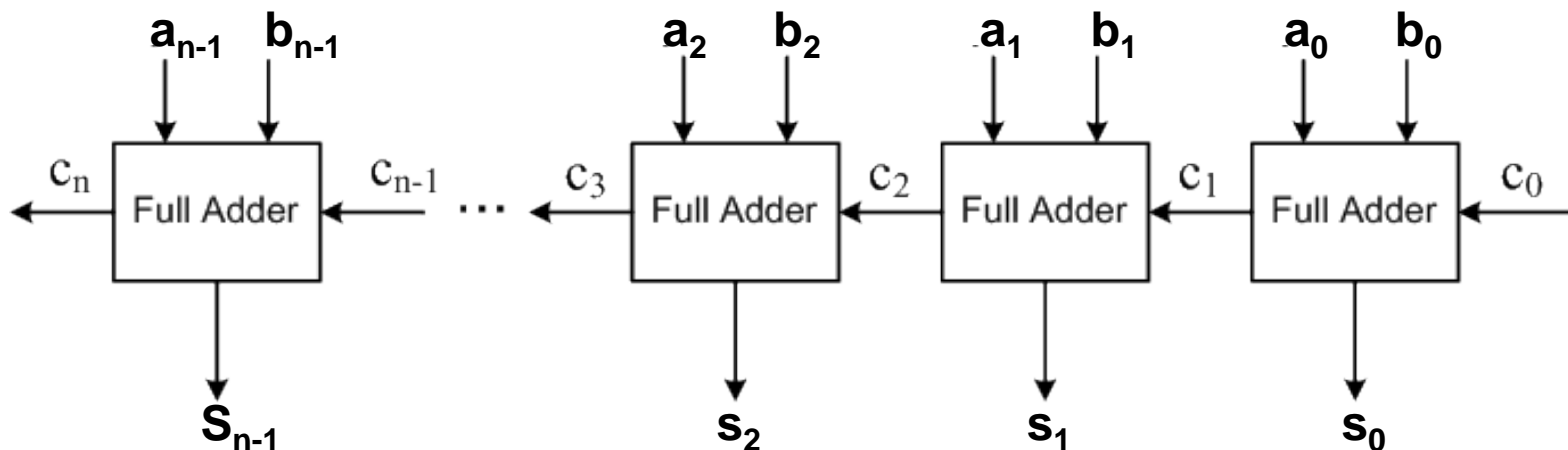
- The most widely used system for representing integers is the **two's complement**, where the **MSB** is considered associated with a **negative weight**
- The value of a two's complement number  $a_{n-1}a_{n-2} \cdots a_1a_0$  is:

$$-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_12^1 + a_02^0$$



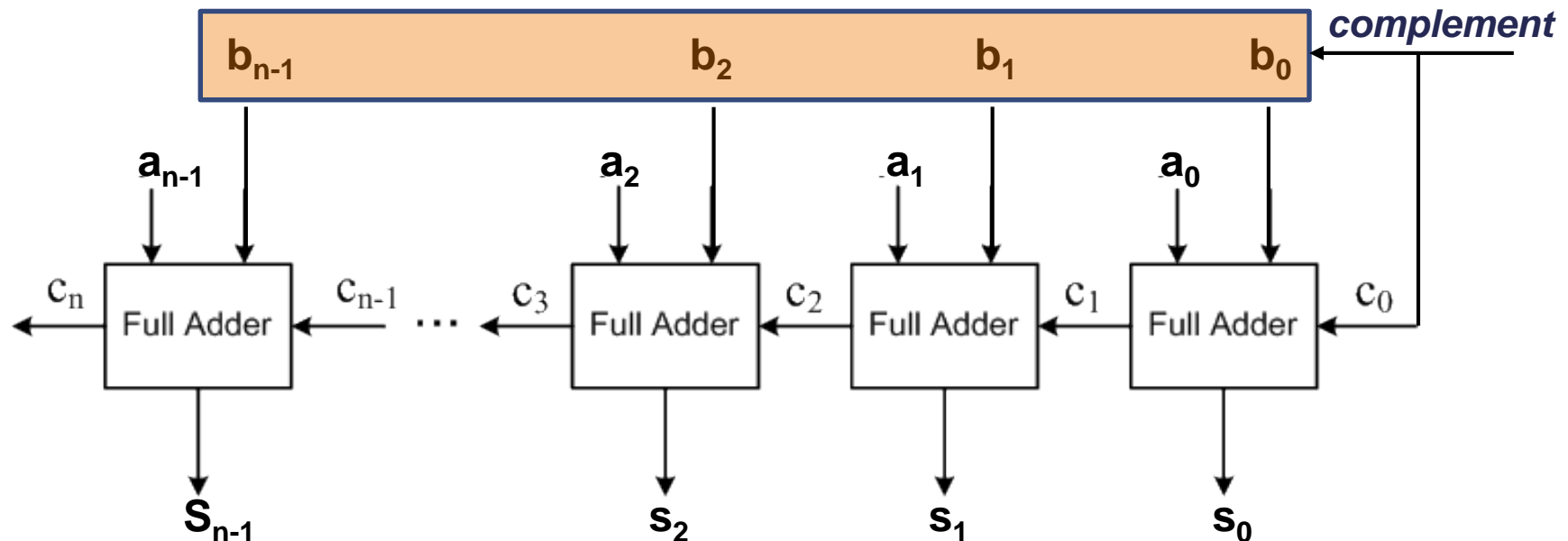
# Ripple-Carry Addition for Signed Numbers

- The reasons for the popularity of two's complement are:
  - 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  $-X = \bar{X} + 1$



# Ripple-Carry Addition for Signed Numbers

- The Ripple-Carry adder is used for subtraction acting on second operand B and on  $c_0$
- If line **complement** is 1 then operand B is complemented bit wise and  $c_0=1$



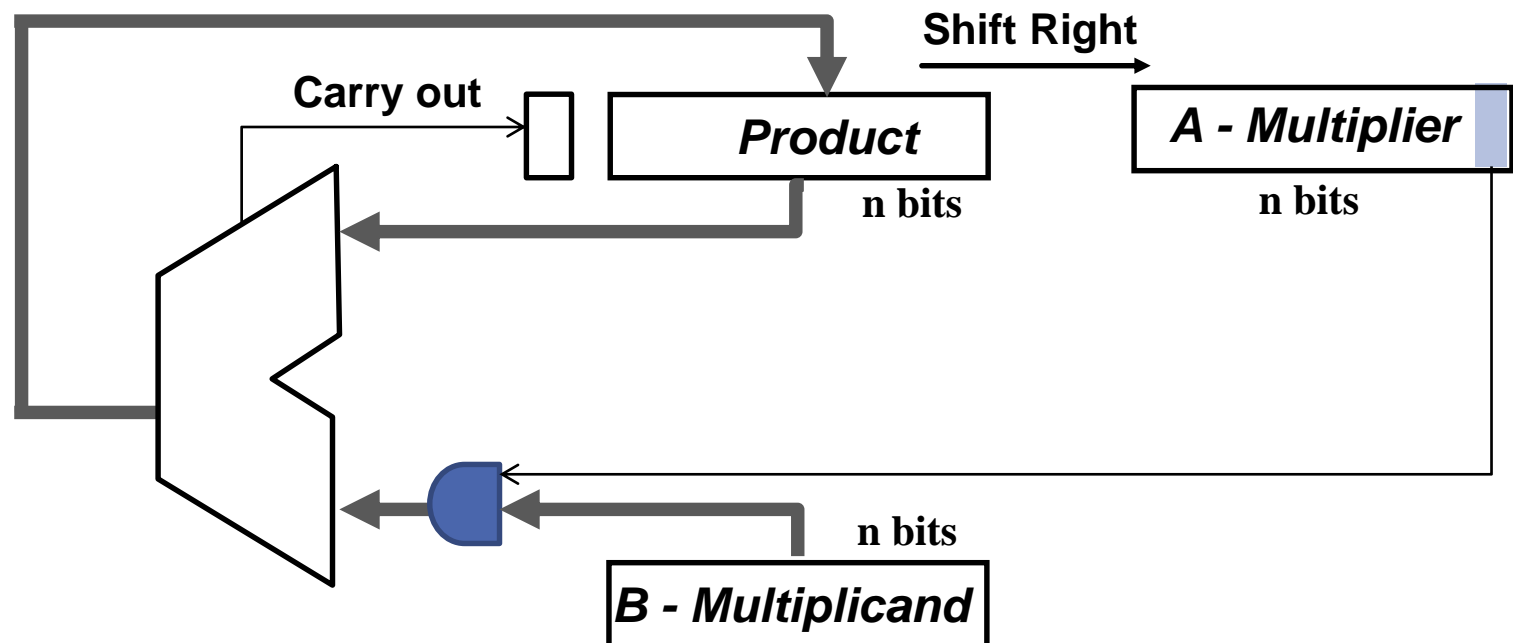
# MULTIPLICATION AND MULTIPLIERS

---



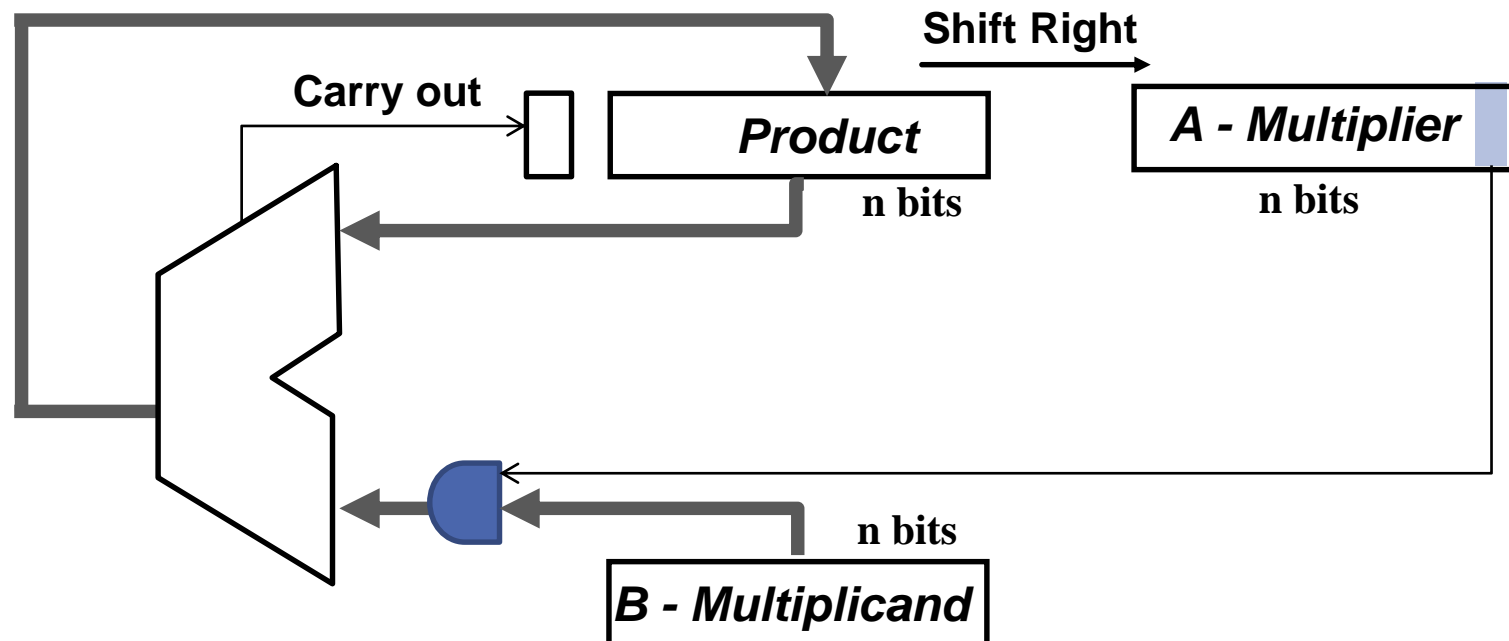
# Unsigned Multiplication

- 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

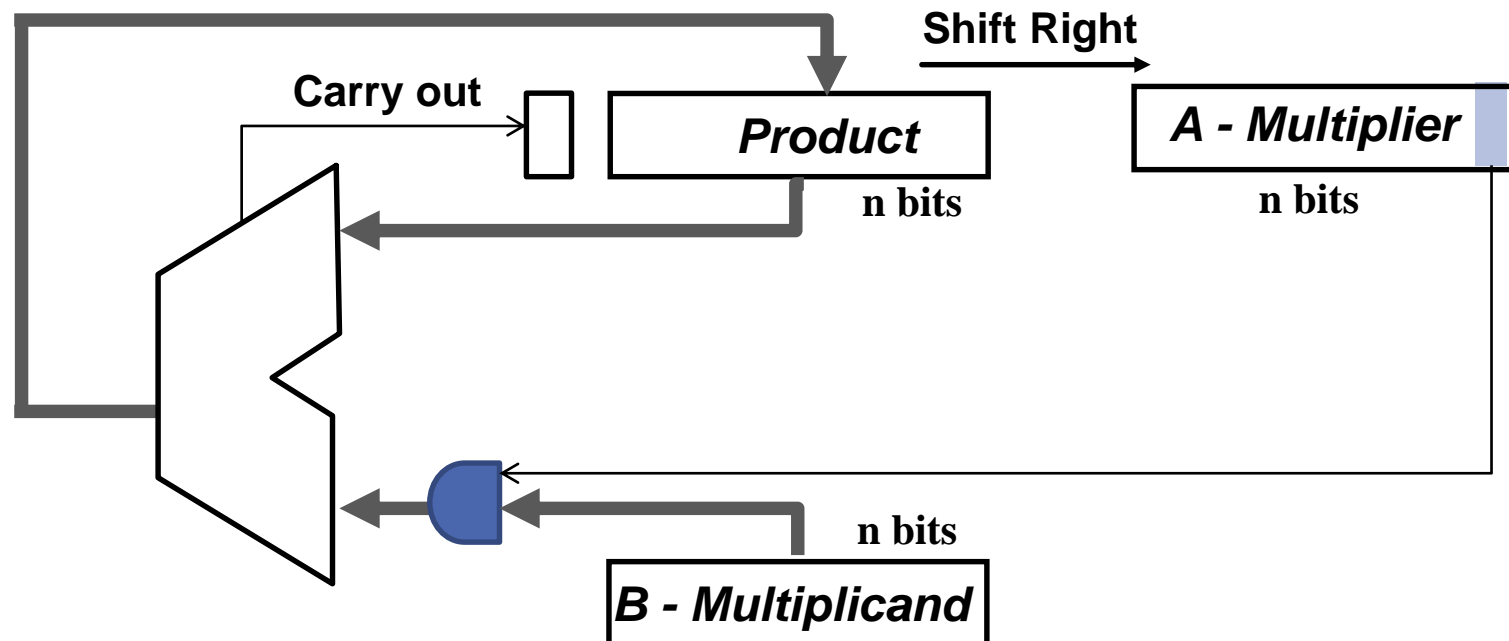
- Each multiply step has two parts:
  - (i) Partial product and accumulation:
    - If the lsb of A is 1, *then* register B ( $b_{n-1}b_{n-2} \cdots b_0$ ) is added to P;  
*else*  $0 \cdots 00$  is added to P
    - The **sum** is placed back **into P**



# Unsigned Multiplication

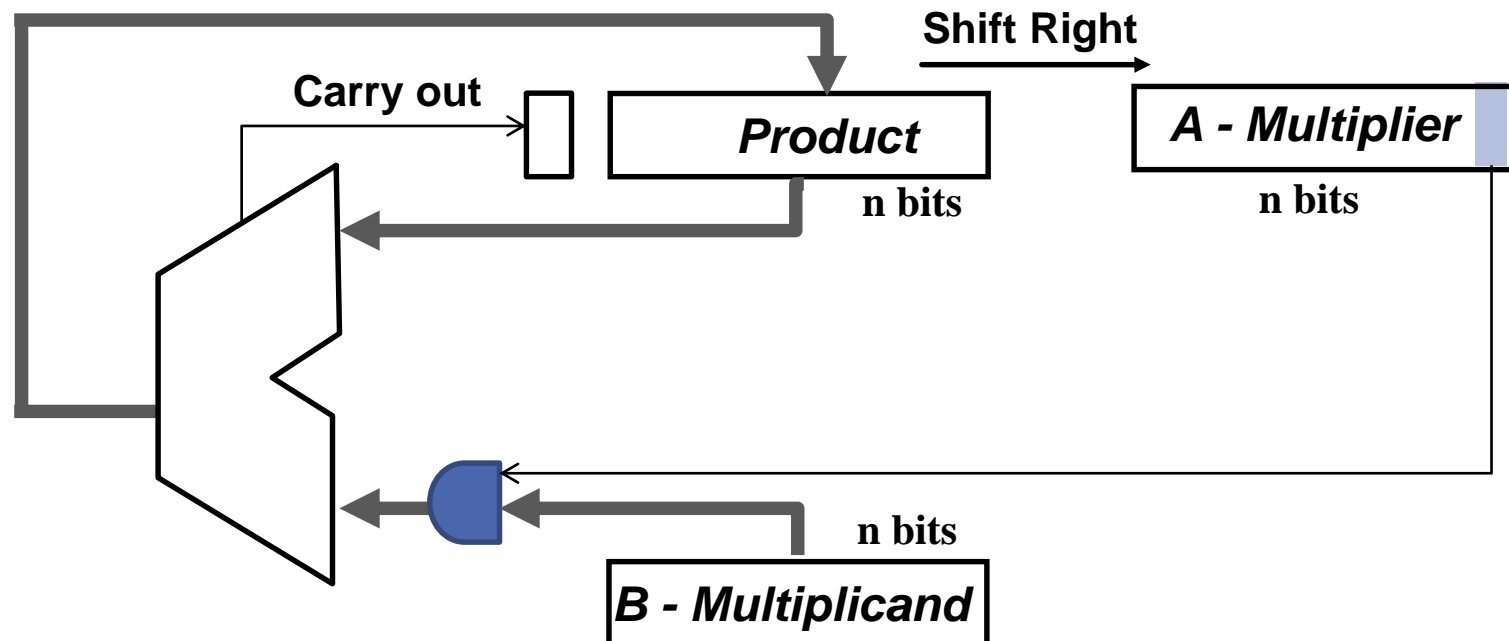
(ii) Registers **P** and **A** are **shifted right**:

- the **carry-out** of the sum is moved into the **high-order bit** of **P**
- the **low-order bit** of **P** is moved into **register A**,
- the **rightmost bit** of **A** (not used any more) is **shifted out**



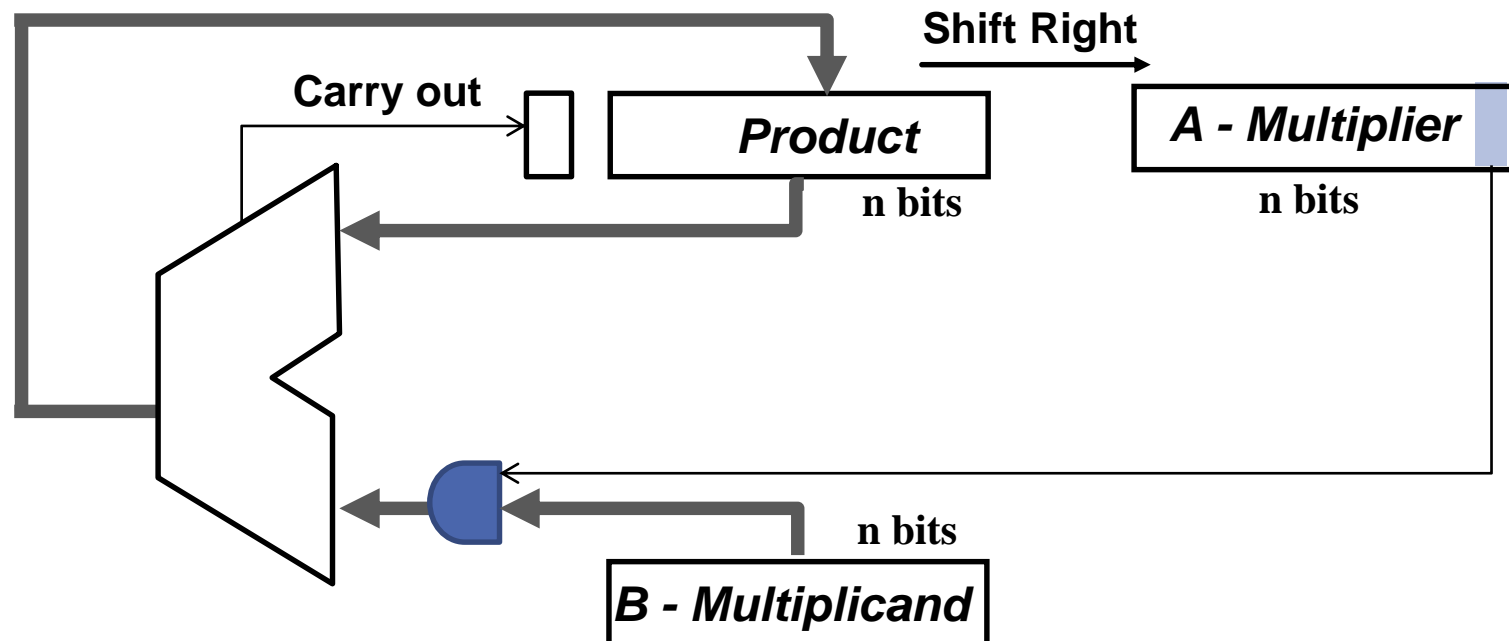
# Unsigned Multiplication

- In summary, each multiplication step consists of: **adding** the contents of P to **either B or 0** (depending on the low-order bit of A), **replace** P with the sum, 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



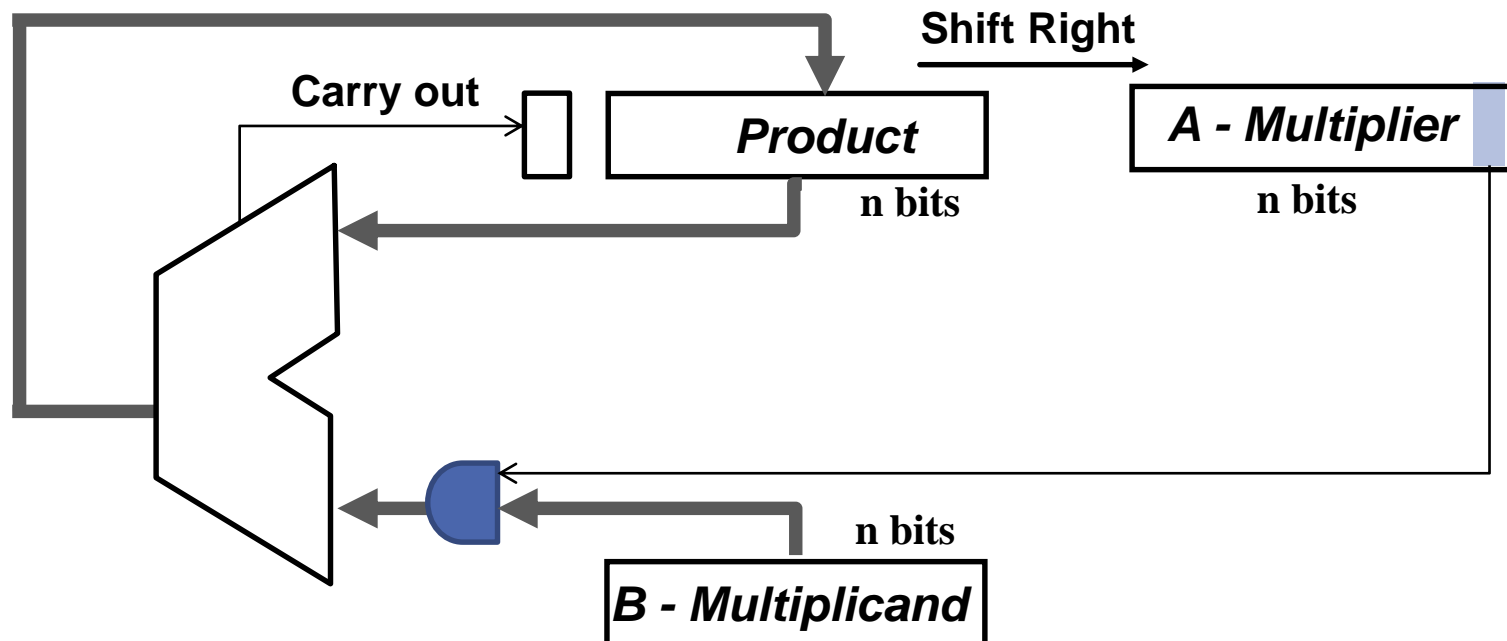
# Signed Multiplication

- 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  **$A$  is nonnegative** and  **$B$  is potentially negative**, to convert the unsigned multiplication algorithm into a two's complement one we need that when  $P$  is shifted, it is **shifted arithmetically**
  - Our adder will now be **adding  $n$ -bit two's complement numbers** between  $-2^{n-1}$  and  $2^{n-1} - 1$



# Signed Multiplication

- If **A is negative**, the method of *Booth recoding* is used
- It is based on the fact that **any sequence of 1s** in a **binary number** can be written as:  **$011\dots11 = 100\dots00 - 000\dots01$**
- **Example** If  $a = 7 = 0111_2$ , then step (i) will successively
  - add B, add B, add B, and add 0
  - Booth recoding “recodes” the number 7 as  $8 - 1 = 1000_2 - 0001_2$

		Multiplication	
	0010		
x	0111		
+	0010	shift	(0 in multiplier)
+	0010	add	(1 in multiplier)
+	0010	add	(1 in multiplier)
+	0000	shift	(0 in multiplier)
	00001110		

# Signed Multiplication

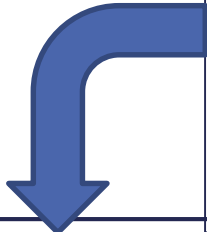
- The idea is that:
  - we **subtract** when we first see a 1 to **replace a string of 1s in multiplier**
  - then later we **add** for the bit **after the last one**

	0010	
<b>x</b>	0111	
<hr/>		
+	0010	shift (0 in multiplier)
+	0010	add (1 in multiplier)
+	0010	add (1 in multiplier)
+	0000	shift (0 in multiplier)
<hr/>		
	00001110	



# Signed Multiplication

- The idea is that:
  - we **subtract** when we first see a 1 to **replace a string of 1s in multiplier**
  - then later we **add** for the bit **after the last one**



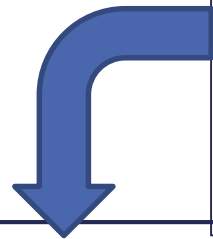
	0010	
x	0111	
+	0010	shift (0 in multiplier)
+	0010	add (1 in multiplier)
+	0010	add (1 in multiplier)
+	0000	shift (0 in multiplier)
	00001110	

	0010	
x	0111	
<hr/>		
-	0010	sub(first 1 in multpl)
+	0000	shift (0 in multiplier)
+	0000	shift (0 in multiplier)
+	0010	add(prior step had last 1)
<hr/>		
	00001110	

# Signed Multiplication

- The idea is that:
  - we **subtract** when we first see a 1 to **replace a string of 1s in multiplier**
  - then later we **add** for the bit **after the last one**



	0010	
x	0111	
+	0010	shift (0 in multiplier)
+	0010	add (1 in multiplier)
+	0010	add (1 in multiplier)
+	0000	shift (0 in multiplier)
	00001110	

	0010	
x	0111	
+	11111110	sub(first 1 in multpl)
+	0000	shift (0 in multiplier)
+	0000	shift (0 in multiplier)
+	0010	add(prior step had last 1)
	00001110	

# 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  $a_{n-1} \cdot \dots \cdot a_0$ , then step (i) in the multiplication algorithm becomes:
  - If  $a_i = 0$  and  $a_{i-1} = 0$ , then **add 0** to P
  - If  $a_i = 0$  and  $a_{i-1} = 1$ , then **add B** to P
  - If  $a_i = 1$  and  $a_{i-1} = 0$ , then **subtract B** from P
  - If  $a_i = 1$  and  $a_{i-1} = 1$ , then **add 0** to P

For the first step, when  $i = 0$ , take  $a_{i-1}$  to be 0

# SPEEDING UP OPERATIONS

---

# Speeding Up Operations

- ▶ Integer **addition** is the simplest operation and the most important
- ▶ Even for programs that do not 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:

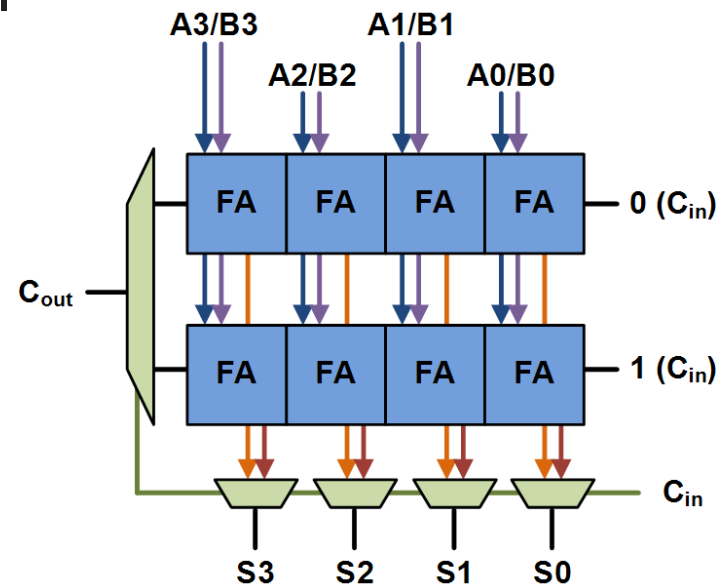
$$t_{\text{ripple}} = Nt_{FA}$$

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), as the Carry Look-ahead Adder (CLA)

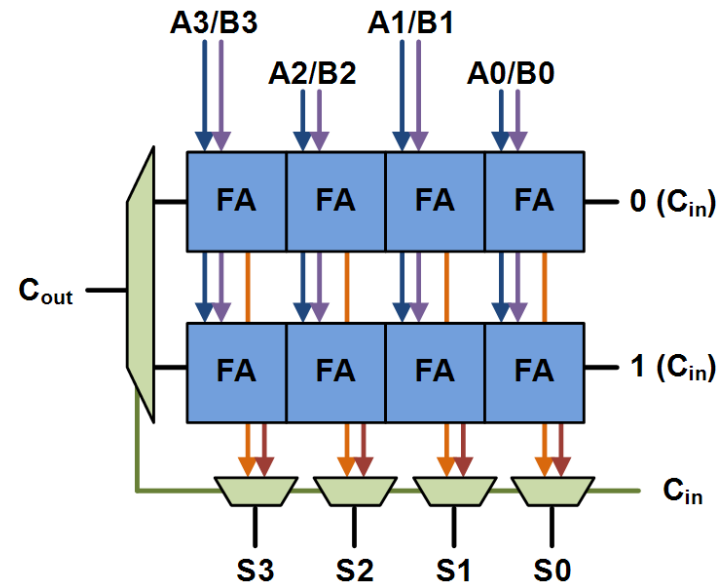
# Carry-Select Adder

- The **carry-select adder** improves speed dividing operands bits in blocks
- A **carry-select adder** consists of:
  - short ripple carry adders acting on blocks of bits
  - multiplexers
- The two blocks of bits is added with **two ripple-carry adders**, one with the carry-in equal to 0 and the other with the carry-in equal to 1



# Carry-Select Adder

- In the figure below **two** 4-bit **ripple-carry adders** are **multiplexed** together
- The resulting carry and sum bits are selected by the carry-in
- The correct result is selected by the actual carry-in which selects which adder had the correct assumption



# Carry-Select Adder

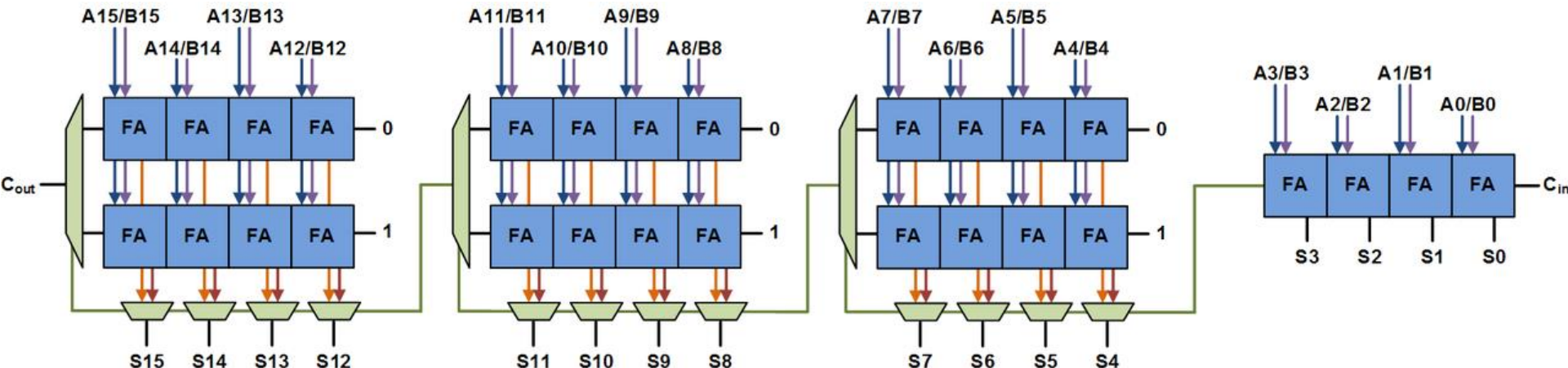
## Uniform-sized adder

A 16-bit carry-select adder with a uniform block size of 4 has:

- three of these blocks
- a 4-bit ripple-carry adder

A carry select block is **not needed** for the four **LSBs**

The **delay** of this adder will be **four full adder delays plus three 2-to-1 MUX delays**



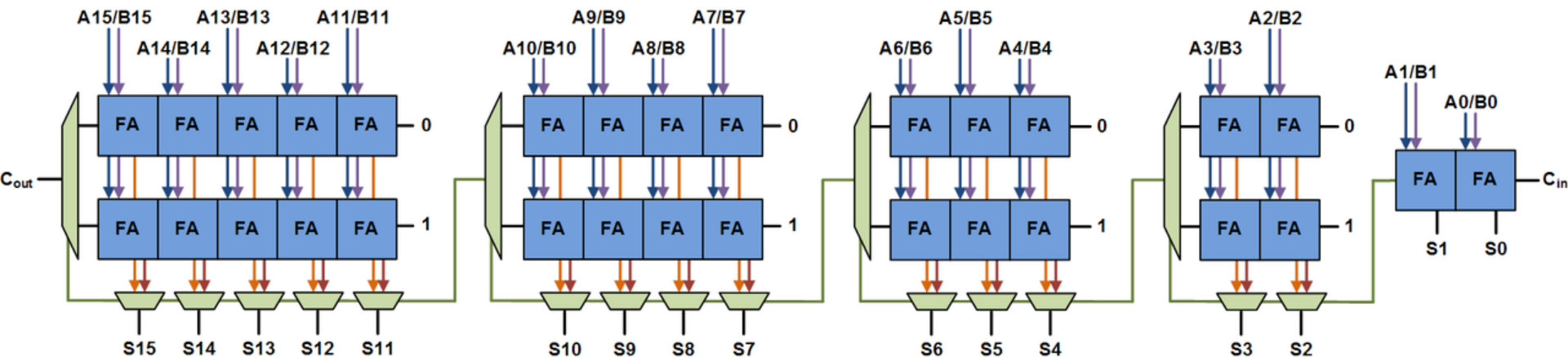


# Carry-Select Adder

## Variable-sized adder

A 16-bit carry-select adder with variable size can be created using block sizes of 2-2-3-4-5

- This break-up is ideal when **the full-adder delay is equal to the MUX delay**
- The total **delay** is **two full adder delays plus four mux delays**



# Carry-Lookahead Adder

- A **carry-lookahead adder** improves speed by reducing the amount of time required to **determine carry bits**
- The carry-lookahead adder calculates **one or more carry bits** before the sum, which reduces the wait time to calculate the result of the larger-value bits of the adder
- Remember that:

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

$$s_i = \overline{a_i} \overline{b_i} c_i + \overline{a_i} b_i \overline{c_i} + \overline{a_i} \overline{b_i} c_i + a_i b_i c_i = (a_i \oplus b_i) \oplus c_i$$

# Carry-Lookahead Adder

We define:

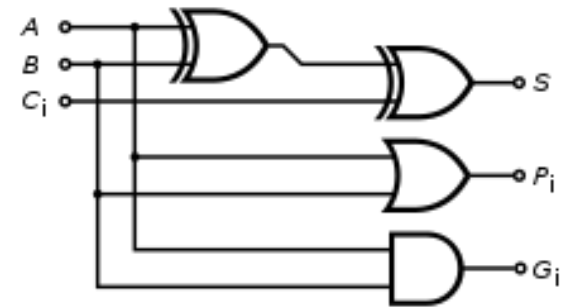
- **Carry Generate**  $g_i = a_i b_i$
- **Carry propagate**  $p_i = a_i \oplus b_i$  or  $p_i = a_i + b_i$

Then the **expression of the carry** is:

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

and the **expression of the sum** is:

$$s_i = (a_i \oplus b_i) \oplus c_i = p_i \oplus c_i$$



# Carry-Lookahead Adder

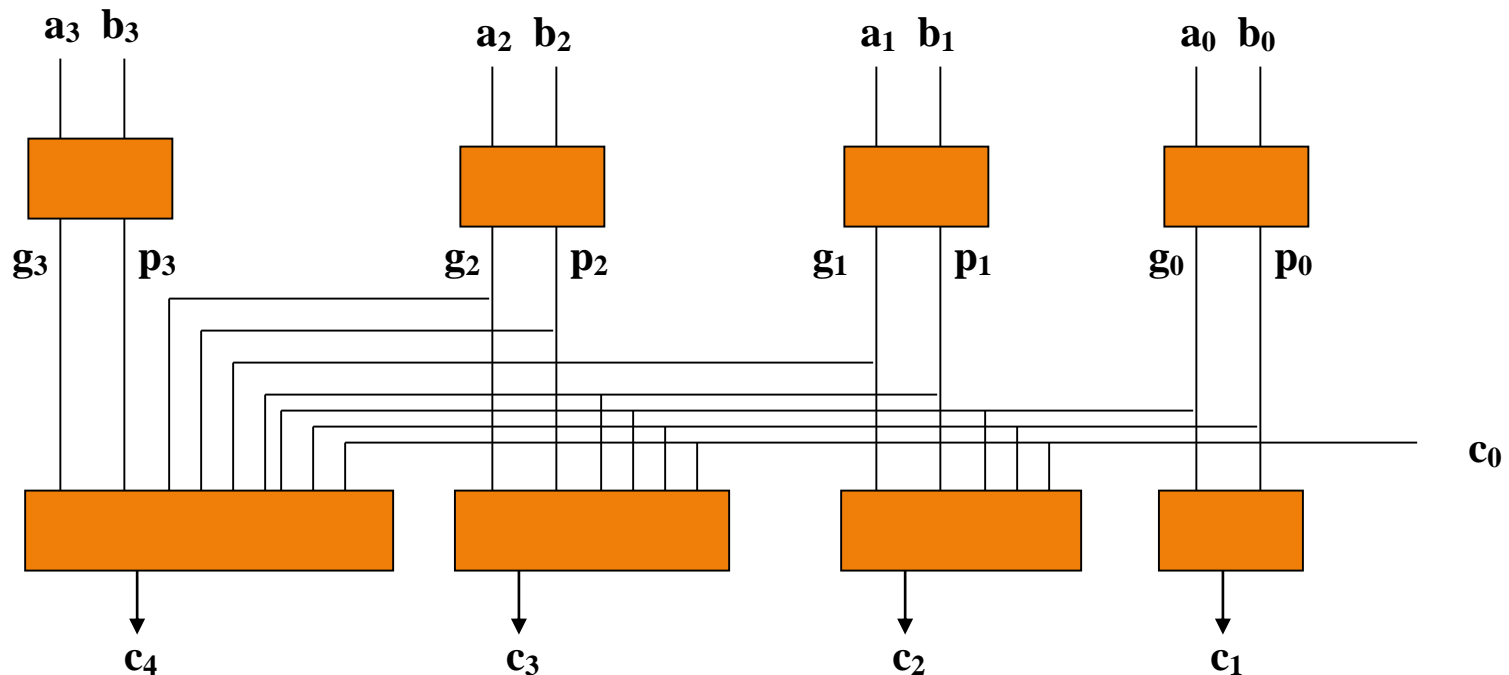
If we consider 4 bits, we have  $c_1, c_2, c_3, c_4$ , depend only on  $c_0$ :

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

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

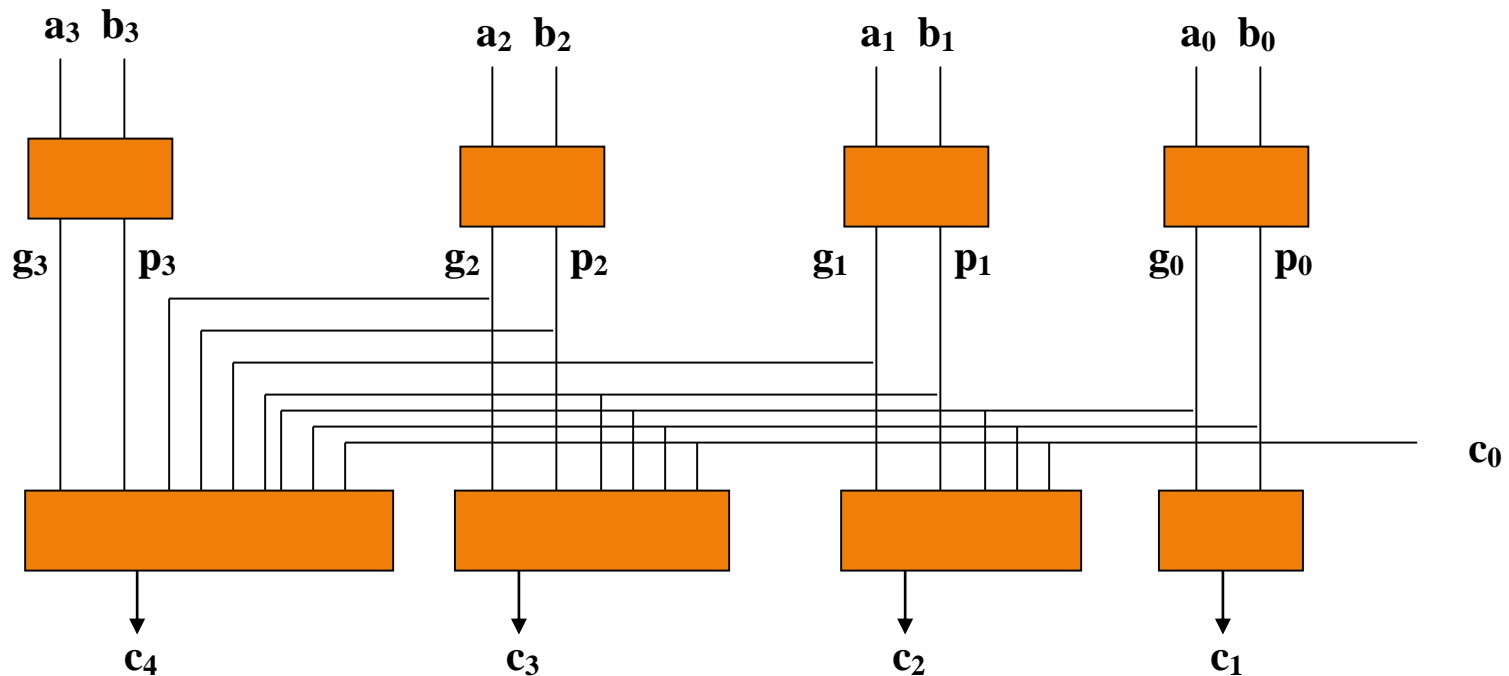
$$c_3 = a_2b_2 + (a_2+b_2)c_2 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$$

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



# Carry-Lookahead Adder

- So, 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

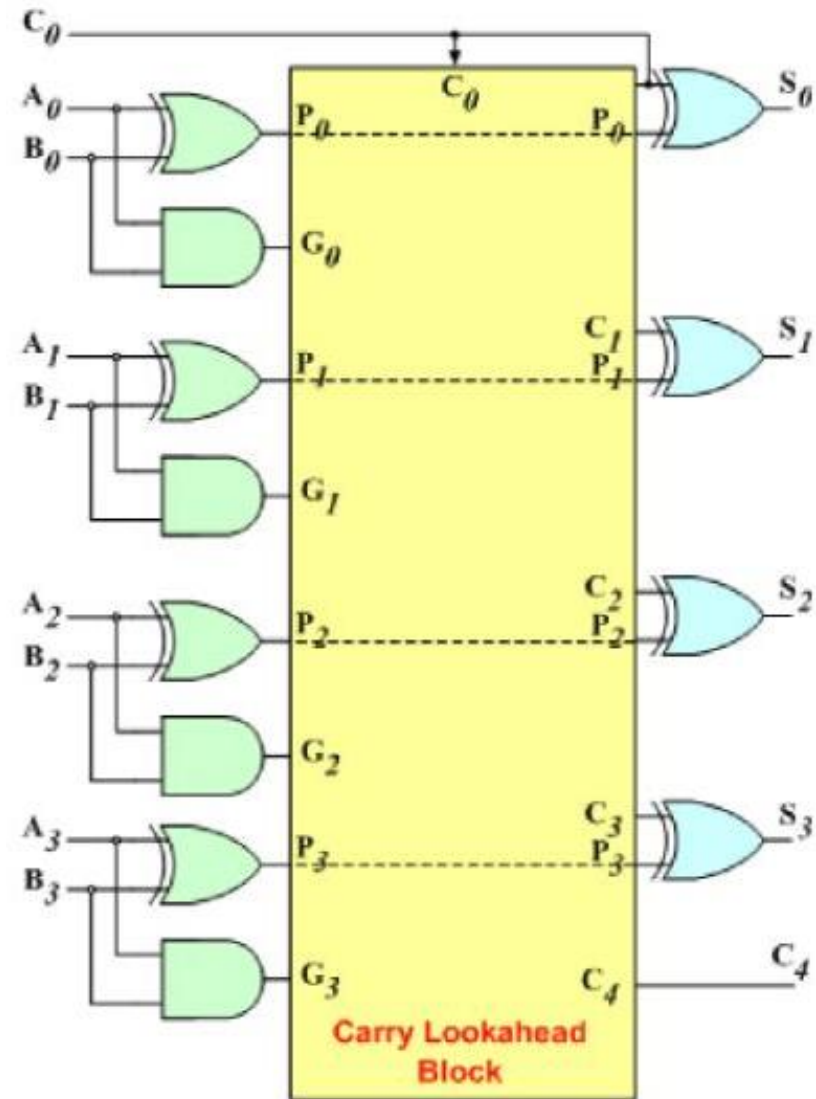


# Carry-Lookahead Addition

- Structure of a **4-bit CLA**

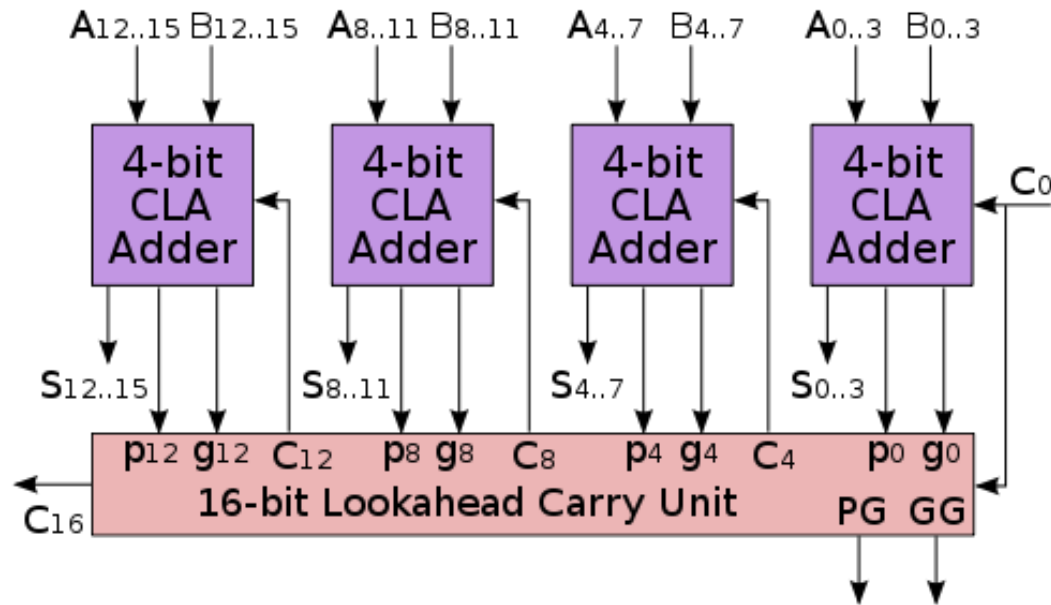
$$s_i = (a_i \oplus b_i) \oplus c_i = p_i \oplus c_i$$

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



# Carry-Lookahead Addition

- A **16-bit adder** can be built from four 4-bit CLAs and a 4-bit Look-ahead Carry Unit (LCU) at the second level



- A **64-bit adder** can be built from four **16-bit adder** shown above, and an additional LCU that accepts bits from each LCU above and generates carry bits fed back to
- [https://en.wikipedia.org/wiki/Lookahead\\_carry\\_unit](https://en.wikipedia.org/wiki/Lookahead_carry_unit)

# Carry-Lookahead Addition

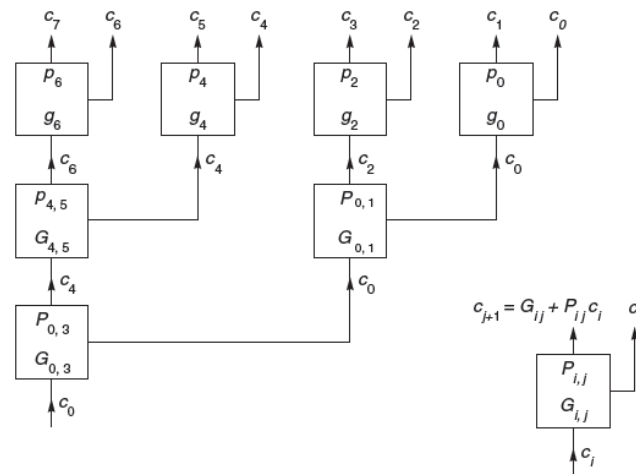
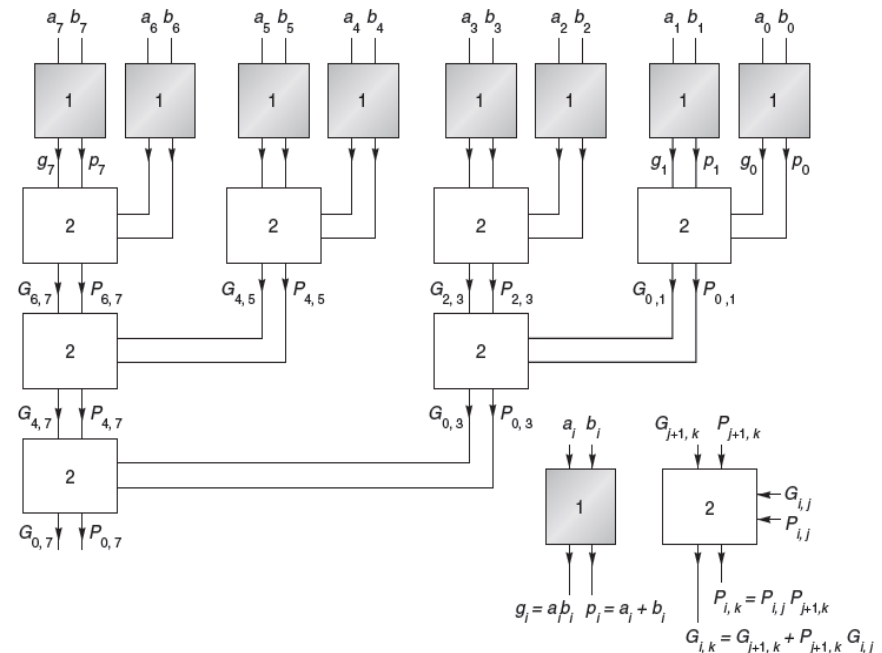
We can use the carry-lookahead idea to build an adder that has about  $\log_2 n$  logic levels

Starting from

- $G_{01} = g_1 + p_1 g_0$
- $P_{01} = p_1 p_0$

In general, for any  $j$  with  $i < j$  and  $j + 1 < k$ , we have the **recursive relations**:

- $c_{k+1} = G_{ik} + P_{ik} c_i$
- $G_{ik} = G_{j+1,k} + P_{j+1,k} G_{ij}$
- $P_{ik} = P_{ij} P_{j+1,k}$





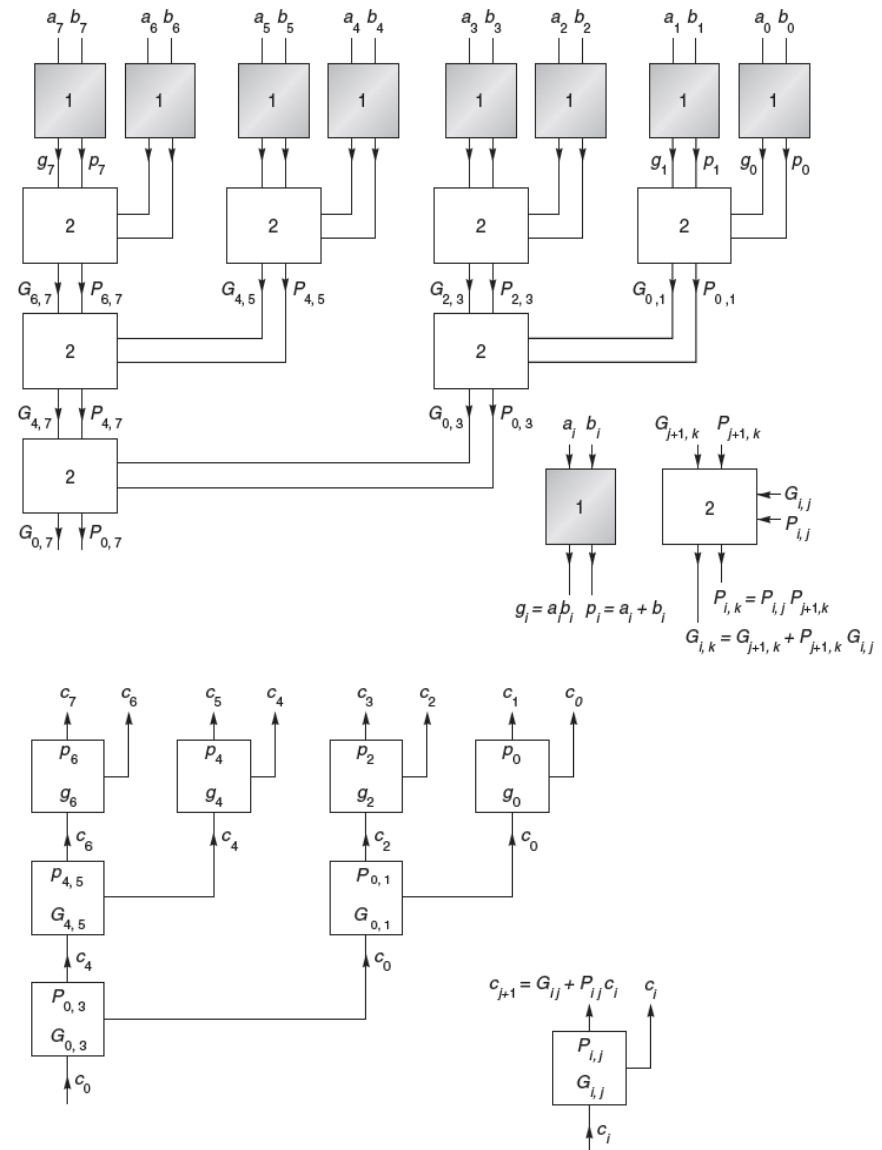
# Carry-Lookahead Addition

**First part** of carry-lookahead tree:

- signals flow from the top to the bottom,
- various values of P and G are computed

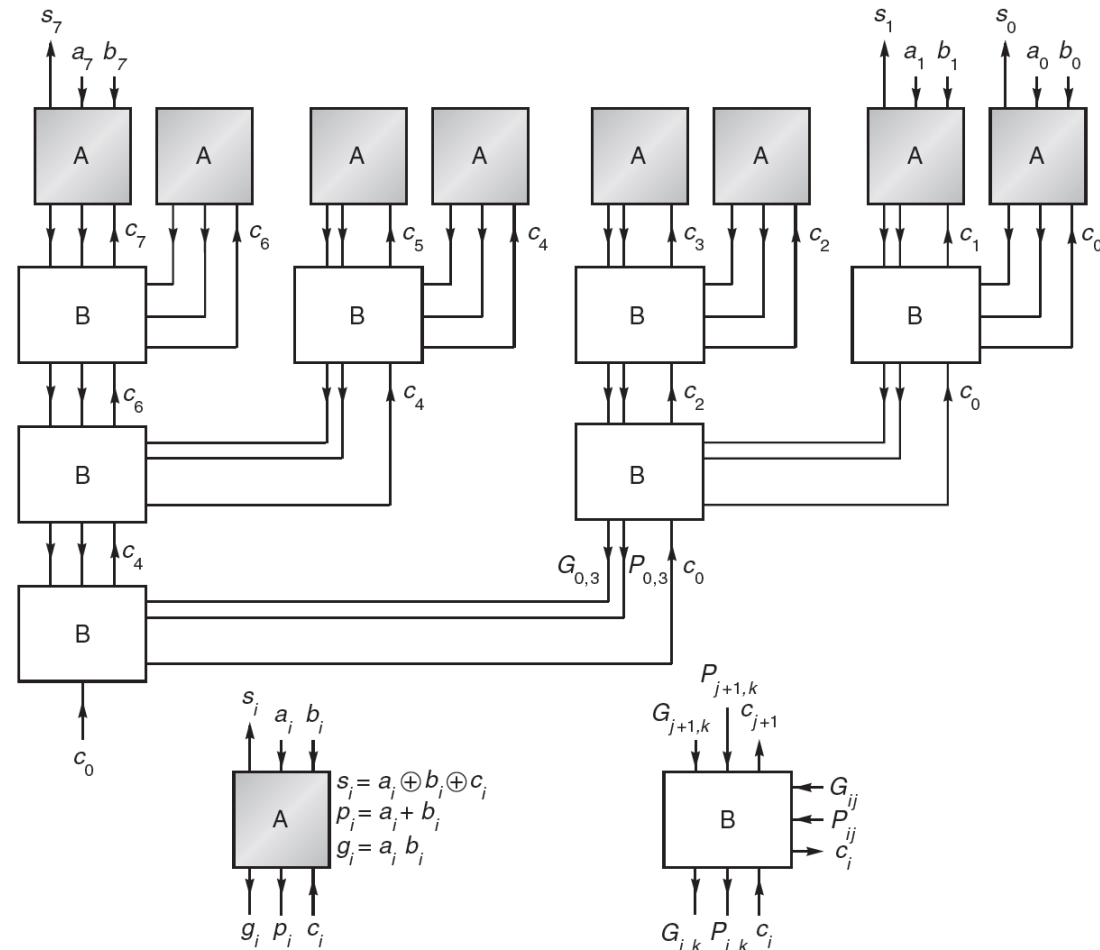
**Second part** of carry-lookahead tree:

- signals flow from the bottom to the top, combining with P and G to form the carries



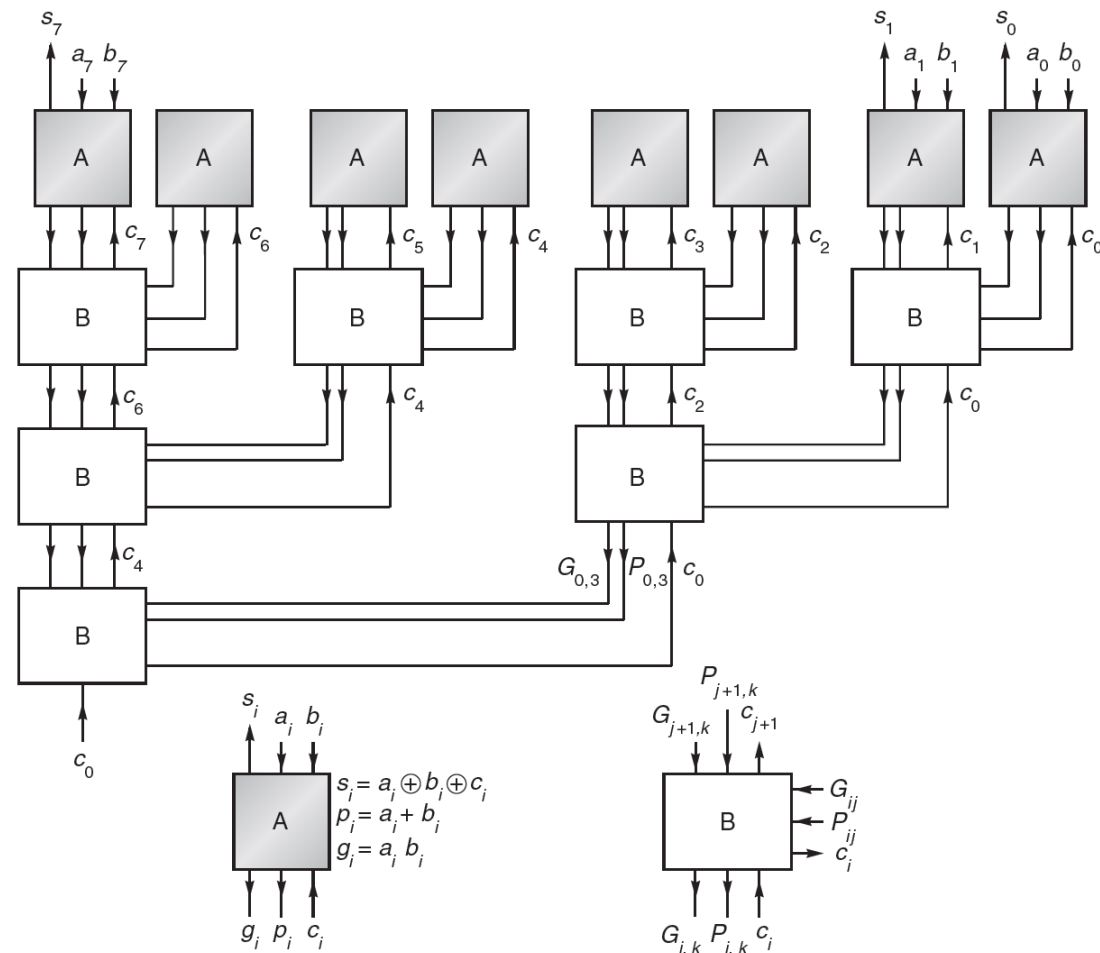
# Carry-Lookahead Addition

- Using the **recursive** relations, we can design practical CLAs combining cells in a **binary tree structure**
- Complete tree adder**  
The numbers to be added flow into the **top** and **downward** through the tree, combining with **c0** at the **bottom** and **flowing back up** the tree to form the carries



# Carry-Lookahead Addition

- The bits in a CLA pass through about  $\log_2 n$  logic levels, compared with  $2n$  for a ripple-carry adder
- But** the ripple-carry adder has  $n$  cells and the CLA has  $2n$  cells, even if they will take  $n \log n$  space
- Speed improvement** especially for a large  $n$



# Speeding Up Multiplications

- 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 (consider the difference between an RCA and a CLA)
- If the space for **many adders** is available, then multiplication speed can be increased thanks to the replication of resources

# PIPELINED ARITHMETIC OPERATIONS

---

# Pipelined arithmetic

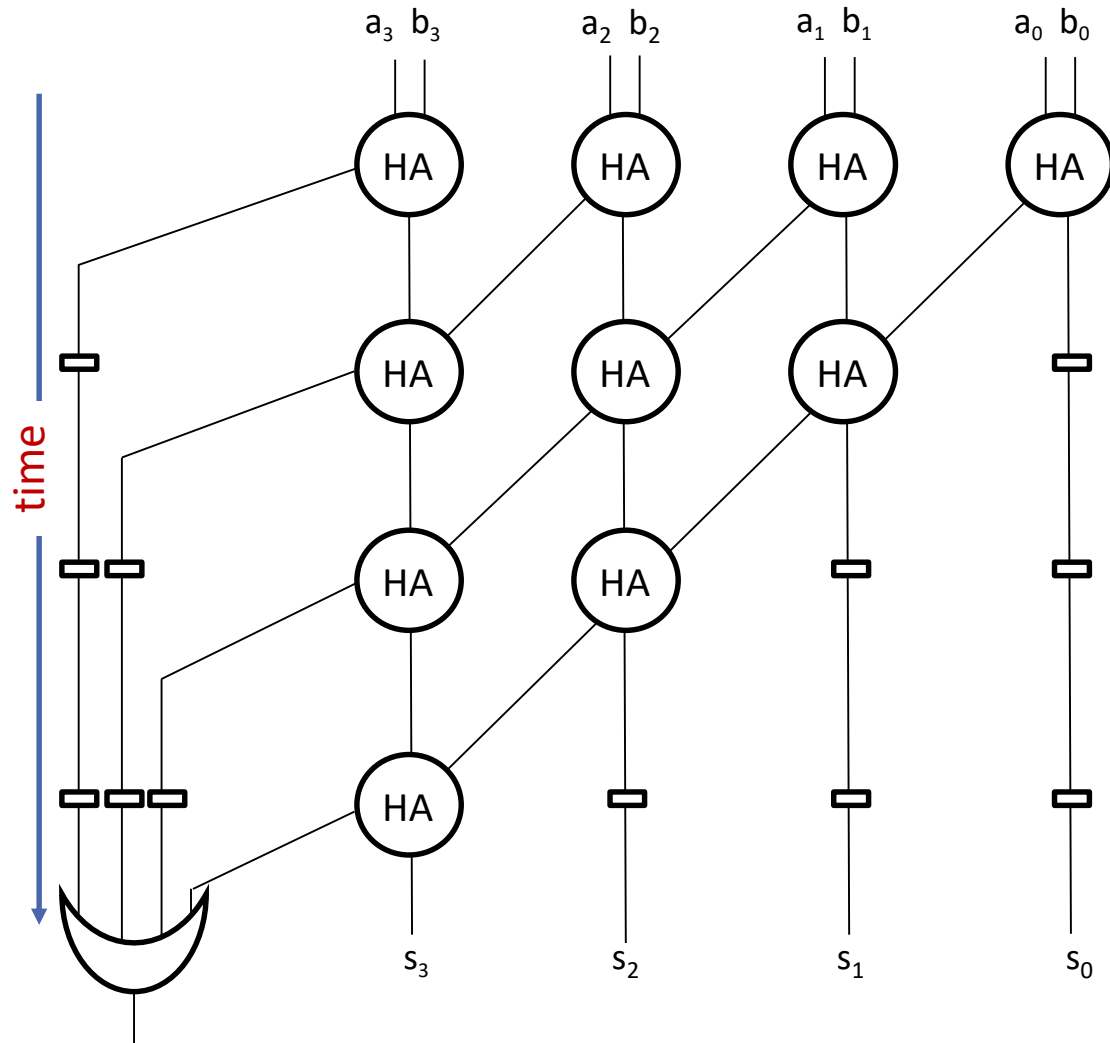
- Consider the instruction pipelining:
  - The processor goes through a repetitive cycle of fetching and processing instructions
  - In the absence of hazards:
    - the processor is continuously fetching instructions from their locations
    - the pipeline is kept full
    - a savings in time is achieved
- 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 arithmetic

- The relative simplicity of two-operand adders usually does not justify their implementation as pipelines
- In special-purpose design, when **many successive additions** are needed, such implementations are justifiable
- Some adders can be implemented as pipeline, such as the conditional-sum adder or the *carry-save adder* (for multiple operands)
- But, for example, some design of the *carry look-ahead adder* cannot be pipelined because some carry signals propagate backward
- There are very simple schemes for the **pipelined adders** and **multipliers** along the lines of the ripple-carry adder

# Pipelined Addition

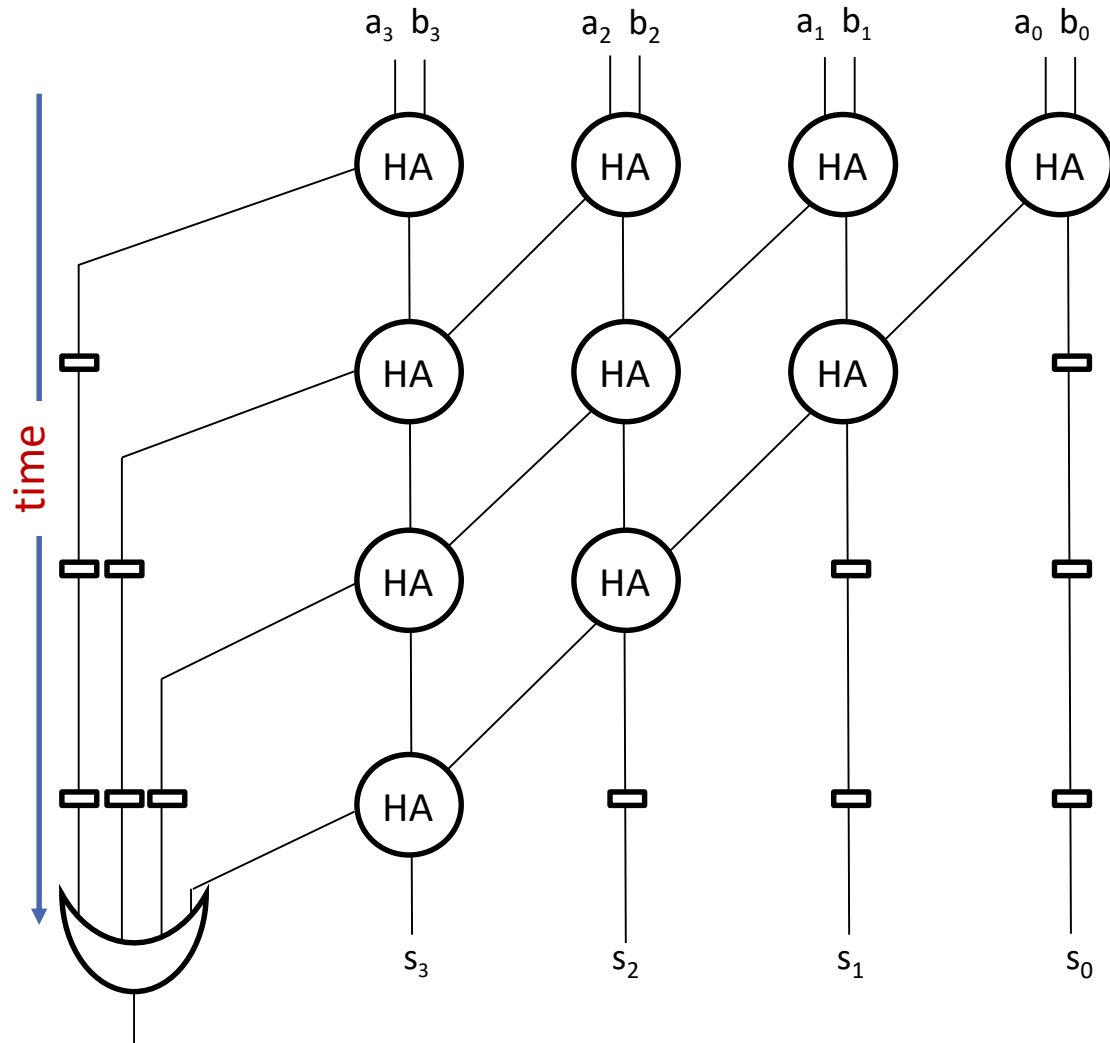
- For  $n$  bits operands, a **pipeline adder** consists of  $n$  stages of half adders
- Registers (FF D) are inserted at each stage to synchronize the computation
- At each clock cycle a new pair of operands is applied to the inputs of the adder





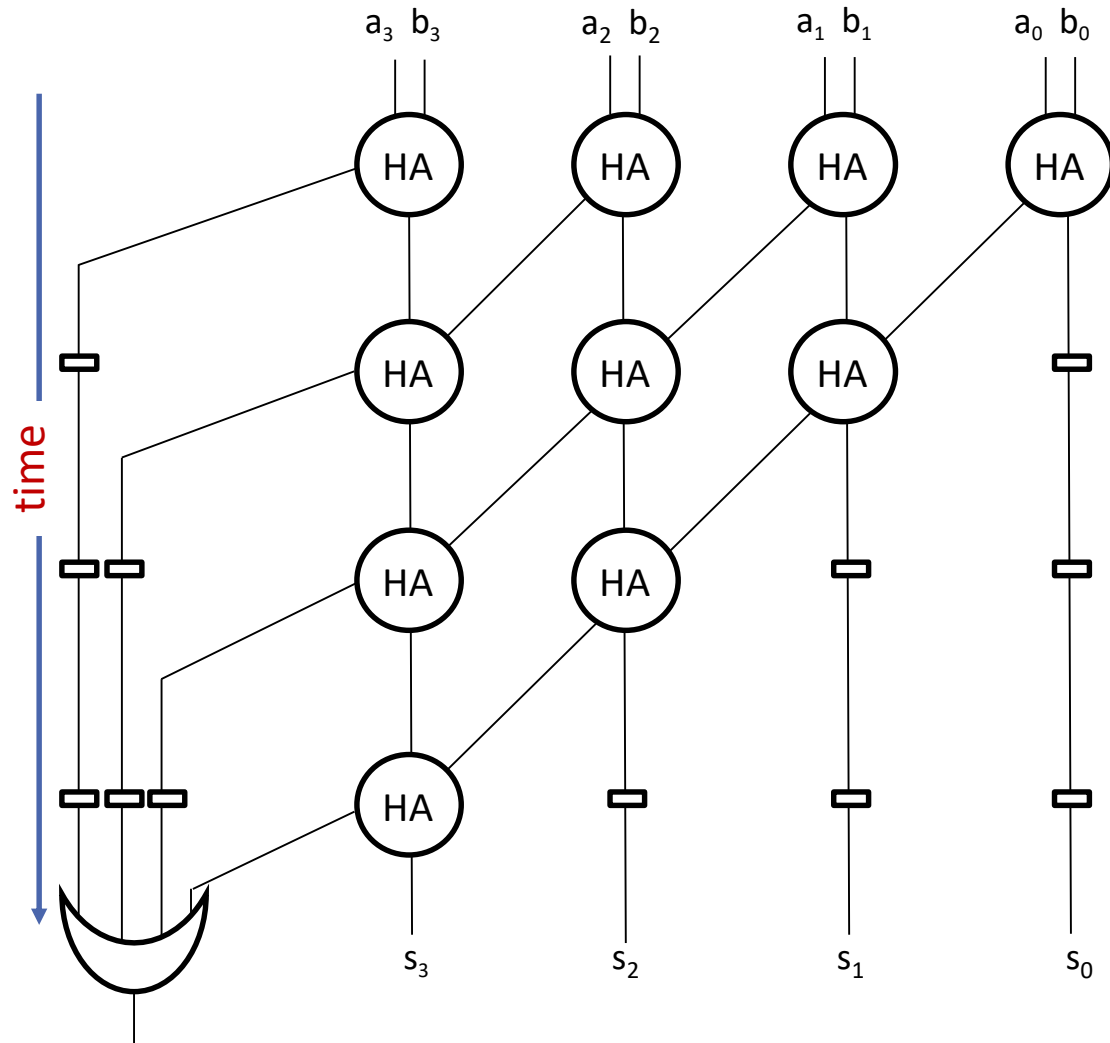
# Pipelined Addition

- 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**

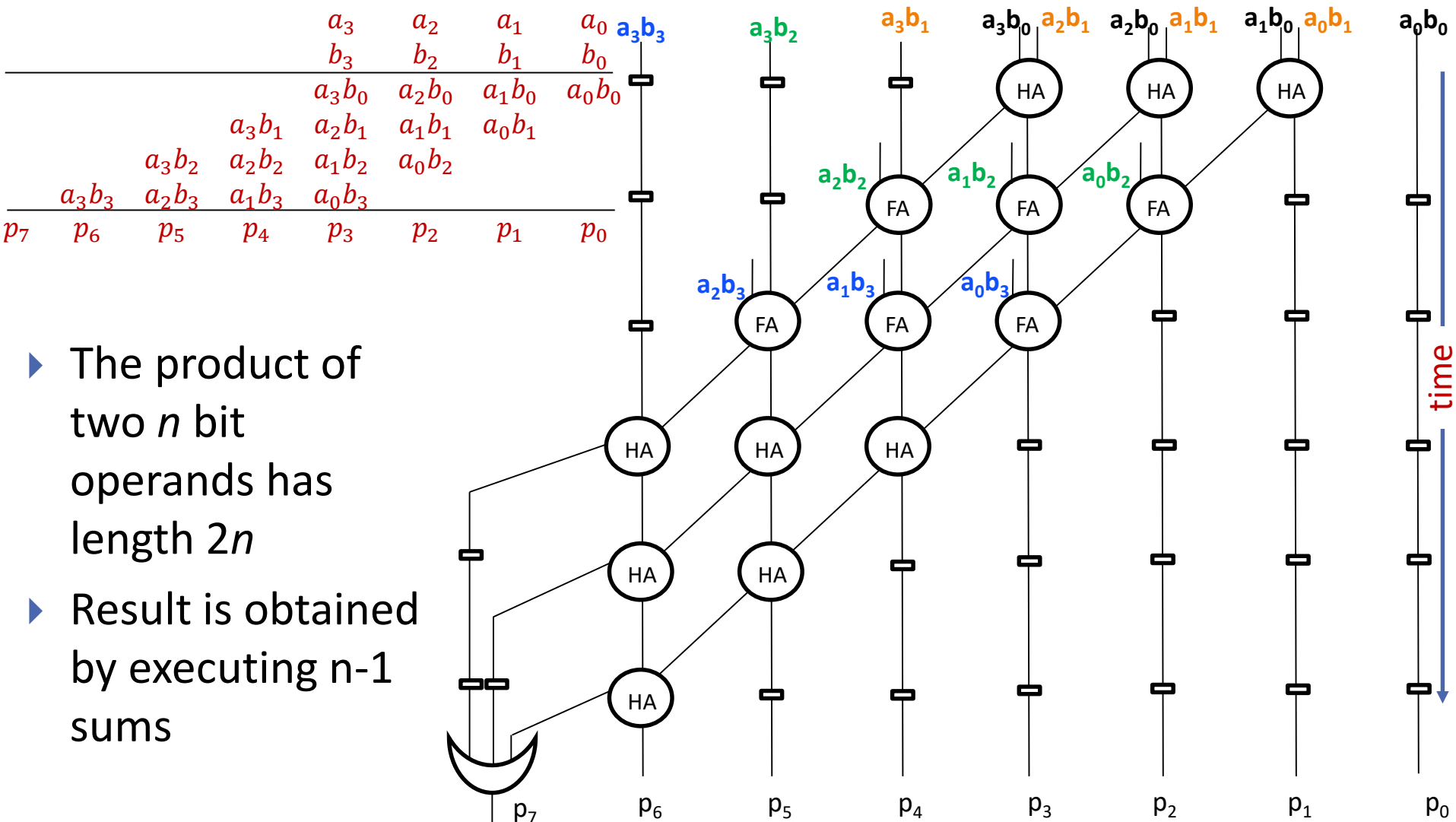


# Pipelined Addition

- 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

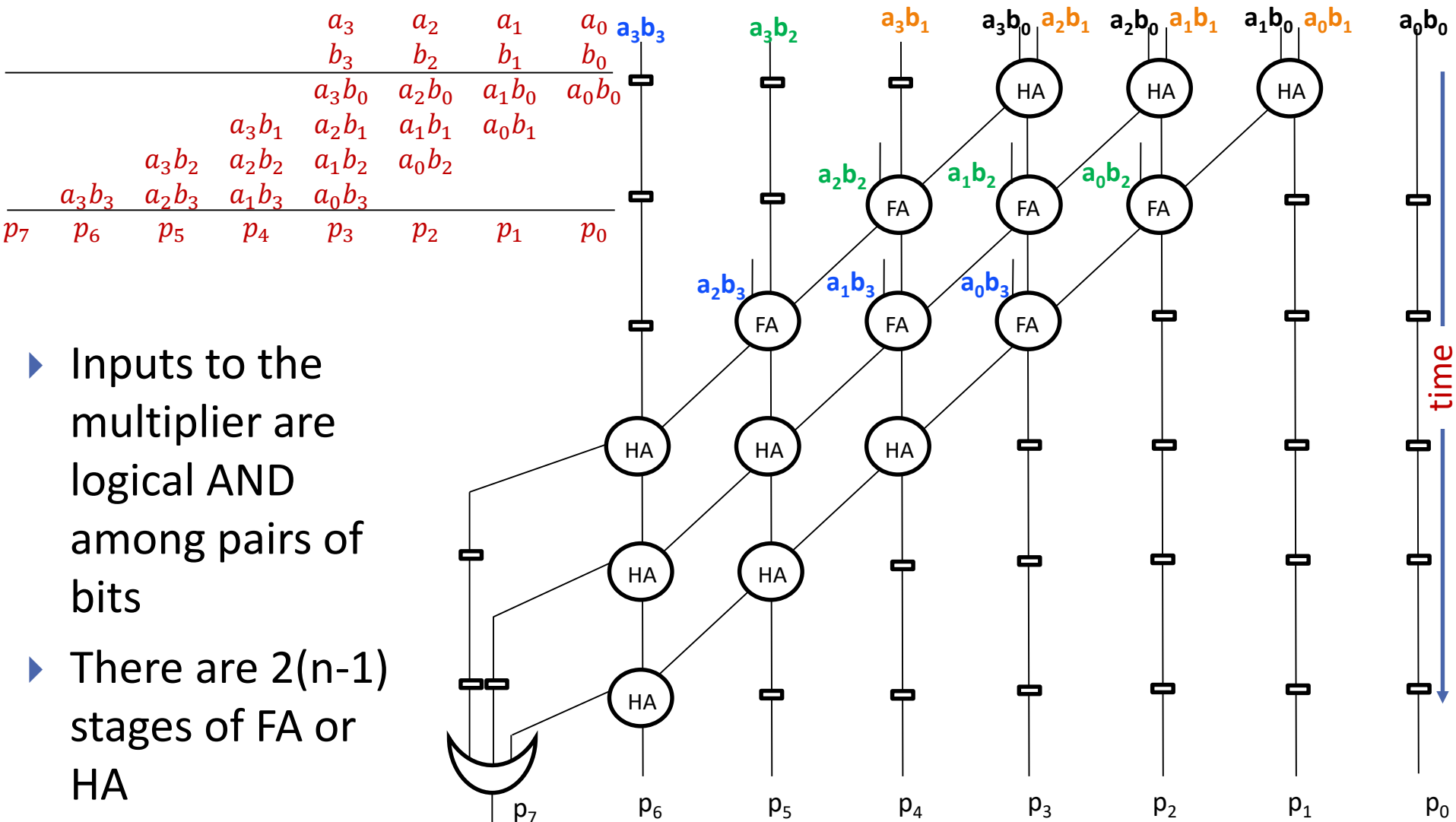


# Pipelined Unsigned Multiplication



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

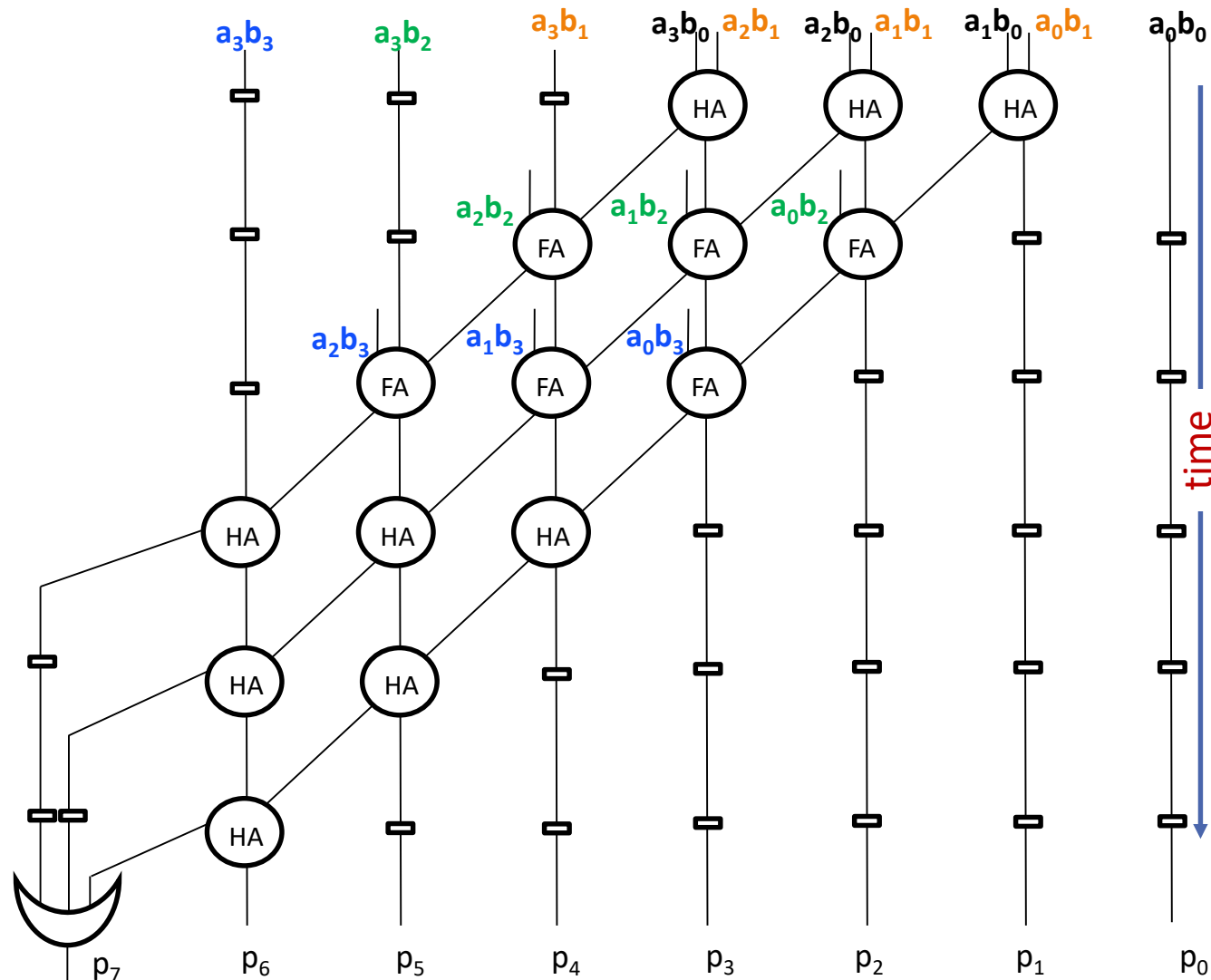
# Pipelined Unsigned Multiplication



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

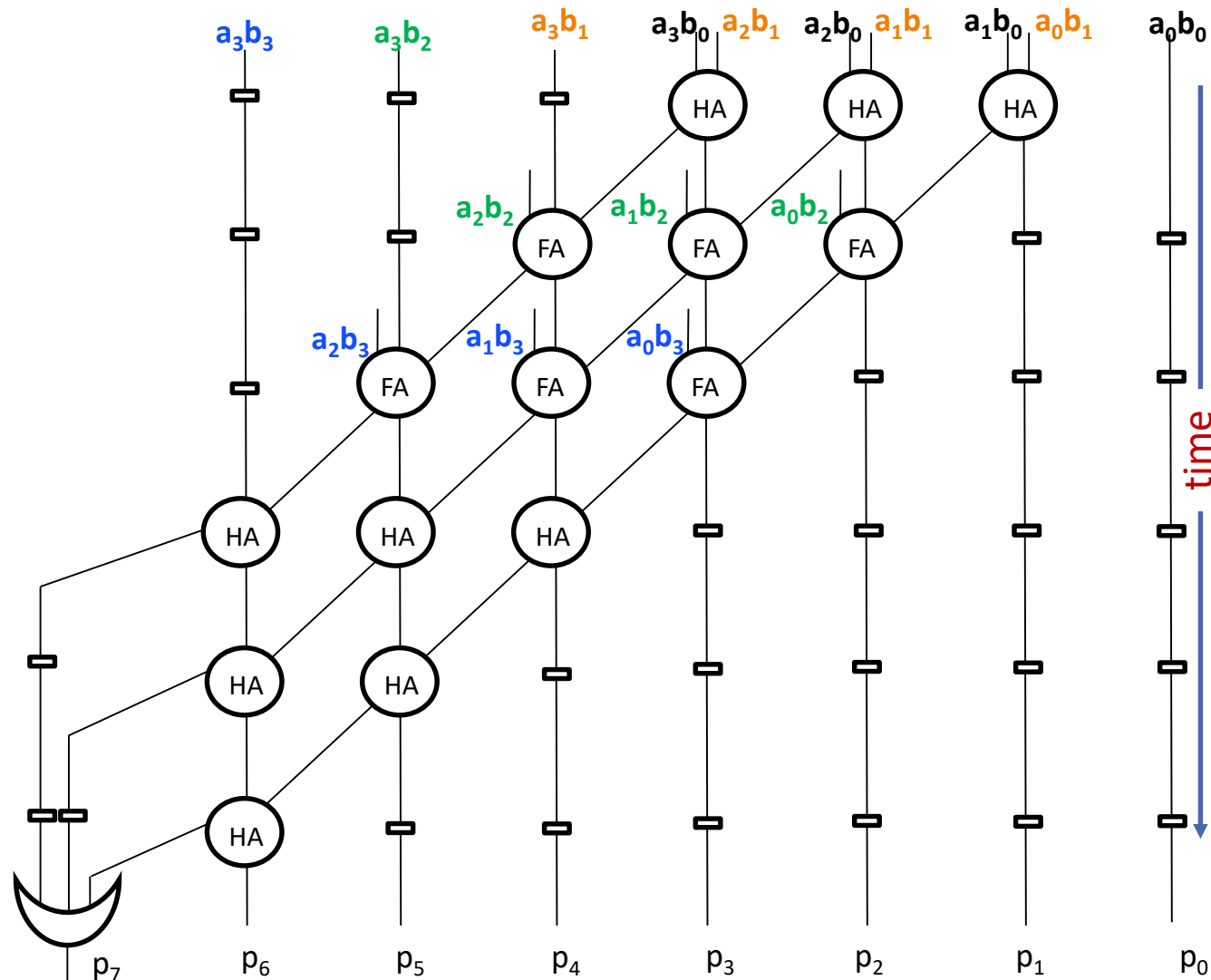
# Pipelined Unsigned Multiplication

- ▶ 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  $(2n-1)$ -th clock cycle

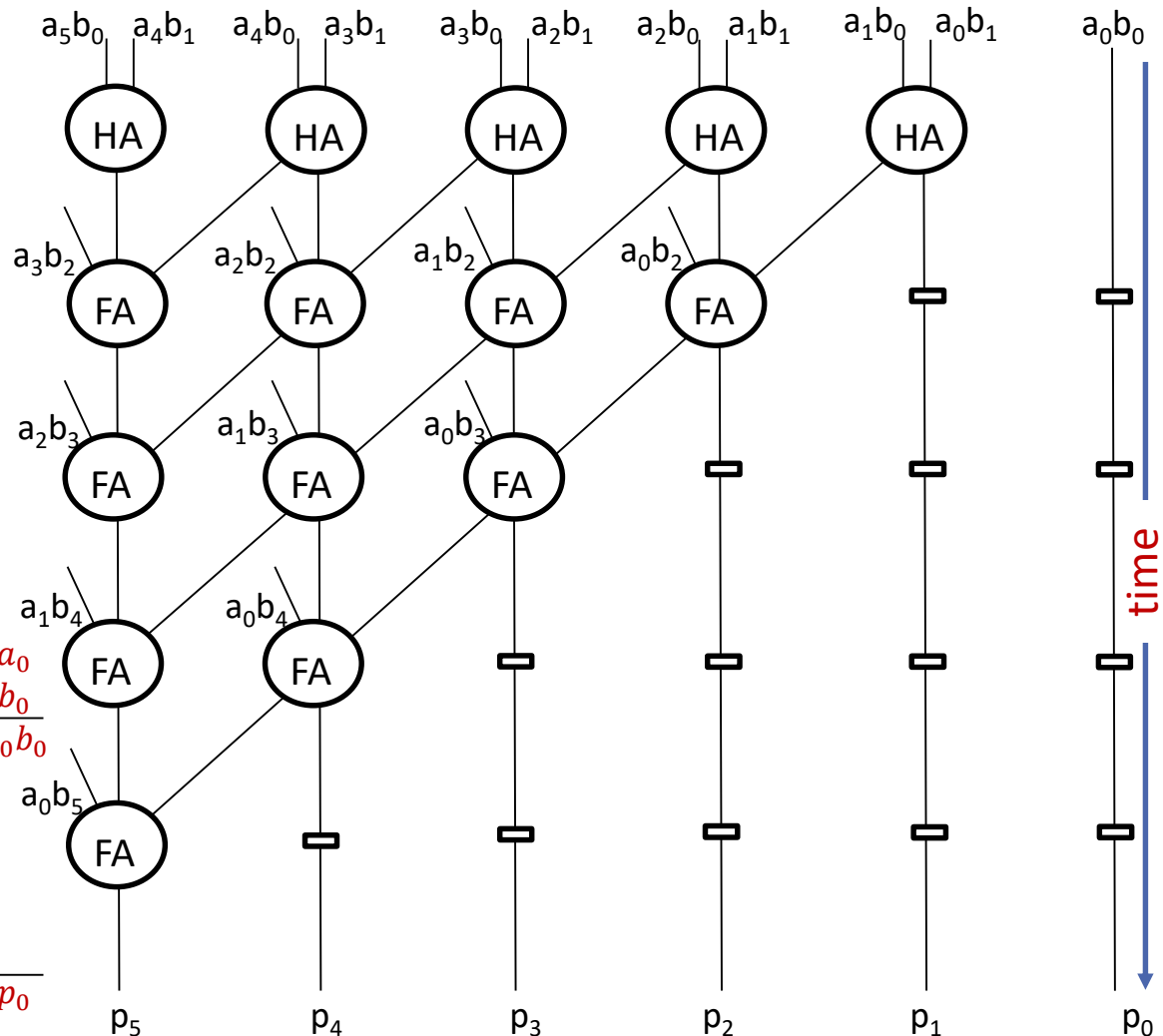


# Pipelined Signed Multiplication

- Signed numbers** are arithmetically extended to the **length  $2n$**  of the product

- Example with 3-bit operands**

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

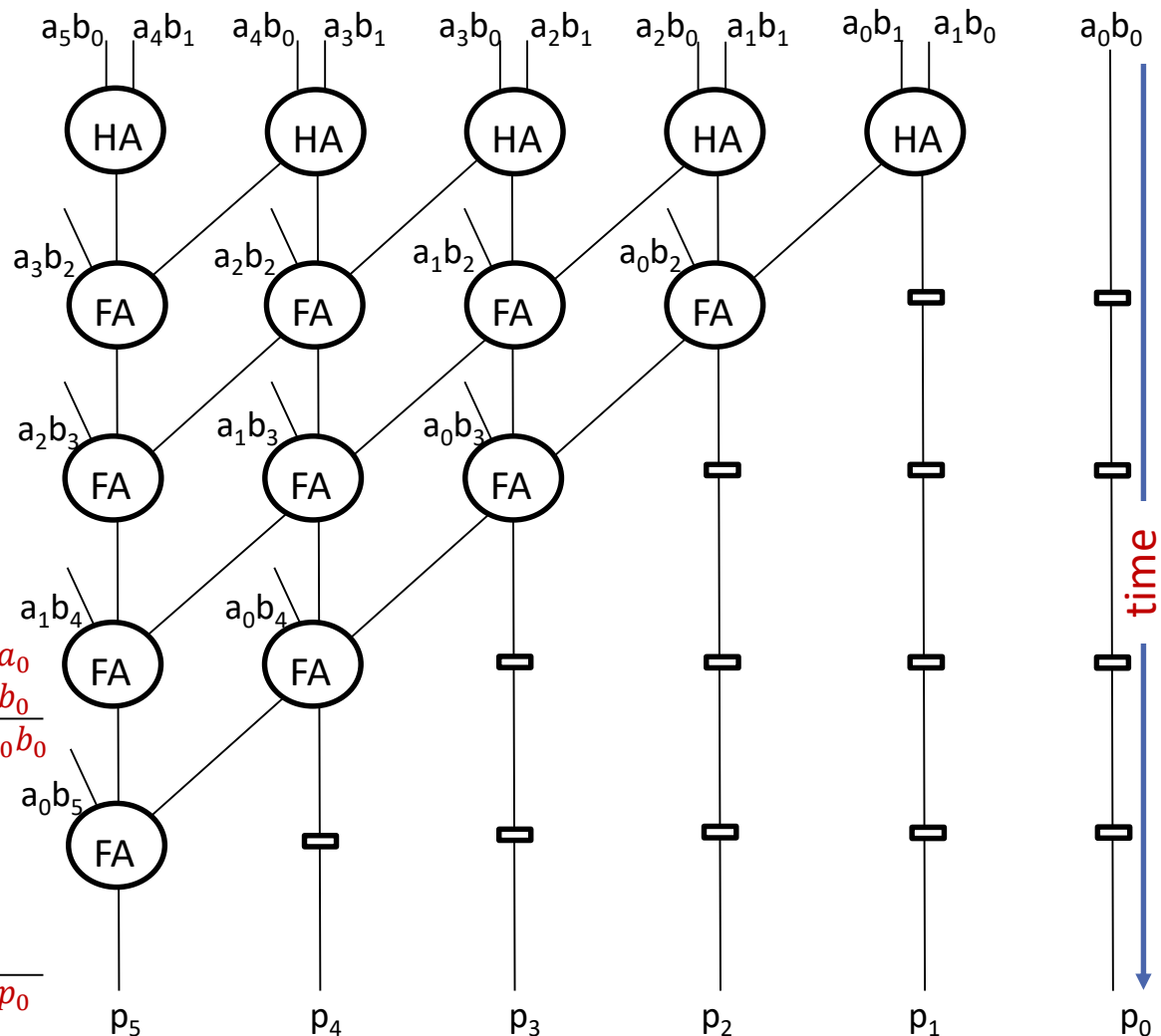


# Pipelined Signed Multiplication

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

- Example with 3-bit operands**

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





# CIRCUIT AREA AND TIME EVALUATION

---

# Circuit area and time

- To discuss about the time and area, it is useful the analytical model called **unit-gate model** presented in
  - A. Tyagi, *A reduced-area scheme for carry-select adders*, IEEE Trans. Comput., 1993is commonly used
- They propose a simplistic model for **gate-count** and **gate-delay**:
  - Each gate except **EX-OR** counts as **one elementary gate**
  - An **EX-OR gate** is counted as **two elementary gates**, because in static CMOS, an **EX-OR** gate is implemented as two elementary gates (**NAND**)
  - The **delay** through an elementary gate is counted as **one gate-delay unit**, but an **EX-OR gate** is **two gate-delay units**

# Circuit area and time

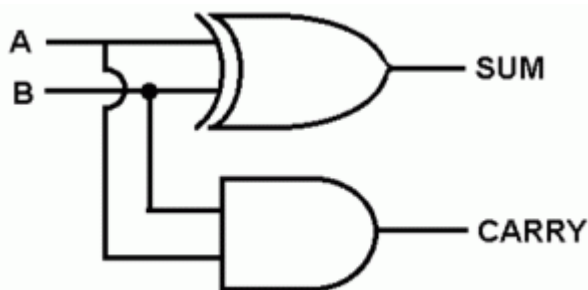
- If the ***fan-in*** and ***fan-out*** of the gates are ignored, unfair comparisons are produced for circuits containing gates with a large difference in *fan-in* or *fan-out*
  - For example, gates in the **CLA adder** have different *fan-in*
  - A **carry-ripple adder** has no gates with *fan-in* and *fan-out* greater than 2
- The **gate-count** and **gate-delay** comparisons may not always be consistent with the area-time comparisons if the *fan-in* of gates is not taken into account
- The best comparison for a VLSI implementation is **actual** area and time

# Circuit area and time

- In summary, we consider:
  - **Any gate** (but the EX-OR) counts as **one gate** for both area and delay  $\rightarrow A_{\text{gate}}$  and  $T_{\text{gate}}$
  - An **exclusive-OR gate** counts as **two elementary gates** for both area and delay  $\rightarrow A_{\text{EX-OR}} = 2A_{\text{gate}}$  and  $T_{\text{EX-OR}} = 2T_{\text{gate}}$
- To **take into account the fan-in and fan-out**, we consider that an **m-input gate** counts as:
  - **m - 1 gates** for area  $\rightarrow A_{\text{m-gate}} = (m-1)A_{\text{gate}}$
  - **$\log_2 m$  gates** for delay  $\rightarrow T_{\text{m-gate}} = \log_2 m T_{\text{gate}}$

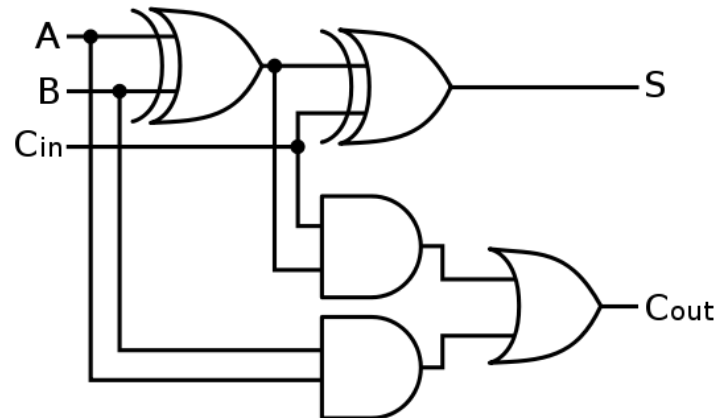
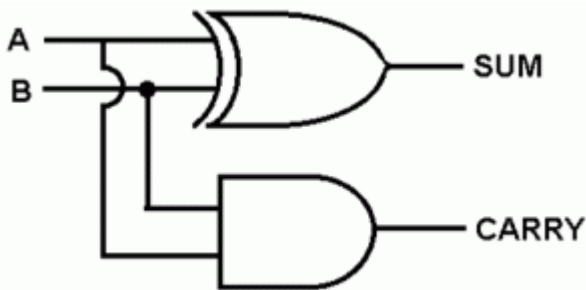
# Circuit area and time

- A half adder (HA) has:
  - **Delay:** 2 unit gates –  $T_{HA} = 2 T_{gate}$
  - **Area:** 3 unit gates –  $A_{HA} = 3 A_{gate}$



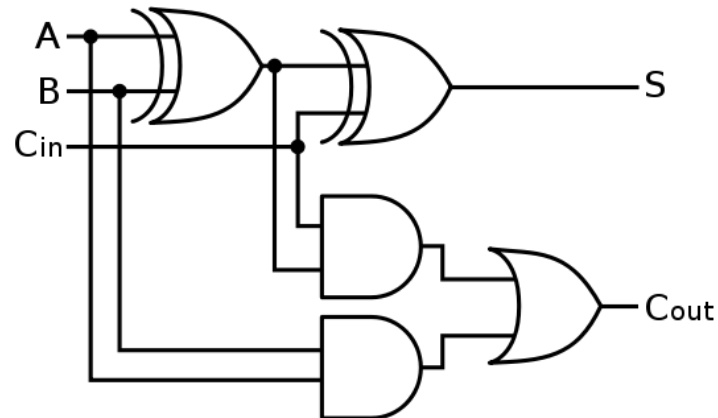
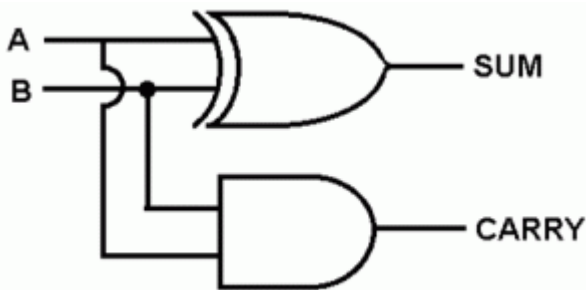
# Circuit area and time

- A **half adder** (HA) has:
  - **Delay:** 2 unit gates –  $T_{HA} = 2 T_{gate}$
  - **Area:** 3 unit gates –  $A_{HA} = 3 A_{gate}$
- A **full adder** (FA) has:
  - **Delay:** 4 unit gates –  $T_{FA} = 4 T_{gate}$
  - **Area:** 7 unit gates –  $A_{FA} = 7 A_{gate}$



# Circuit area and time

- A **half adder** (HA) has:
  - **Delay:** 2 unit gates –  $T_{HA} = 2 T_{gate}$
  - **Area:** 3 unit gates –  $A_{HA} = 3 A_{gate}$
- A **full adder** (FA) has:
  - **Delay:** 4 unit gates –  $T_{FA} = 4 T_{gate} = 2 T_{HA}$
  - **Area:** 7 unit gates –  $A_{FA} = 7 A_{gate} = 2 A_{HA} + A_{gate}$

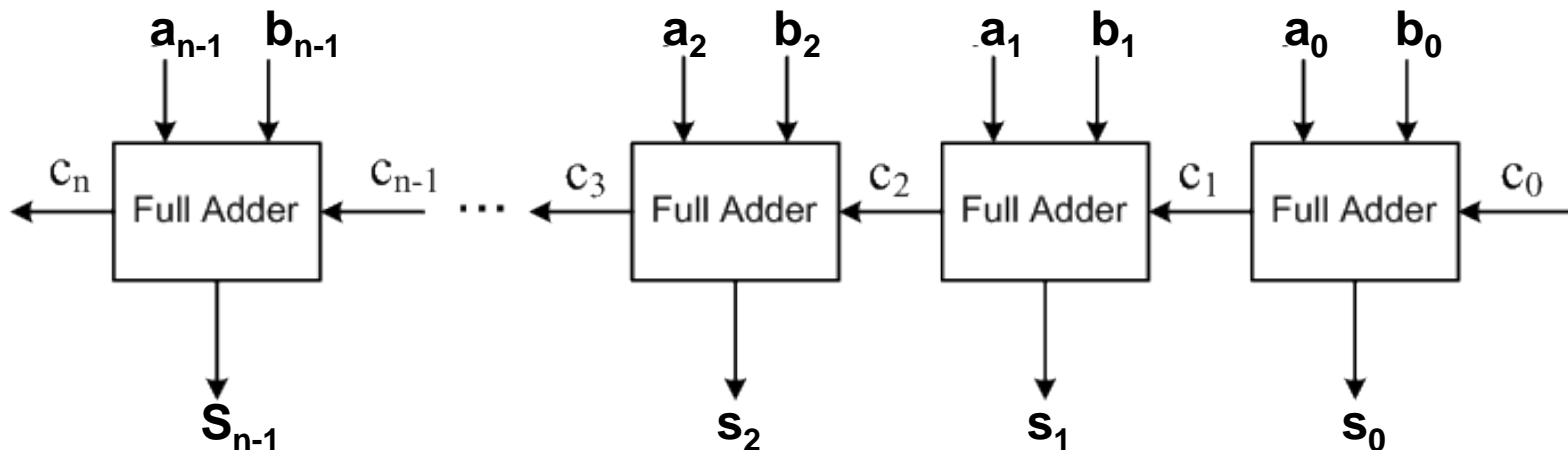


# Circuit area and time

- A ripple-carry adder for n-bits operands has:

- Delay:**  $T_{\text{RC-adder}} \rightarrow T_{\text{RC-adder}} = n T_{\text{FA}} = 2n T_{\text{HA}} = 4n T_{\text{gate}}$

- Area:**  $A_{\text{RC-adder}} \rightarrow A_{\text{RC-adder}} = n A_{\text{FA}} = 2n A_{\text{HA}} + n A_{\text{gate}} = 7n A_{\text{gate}}$



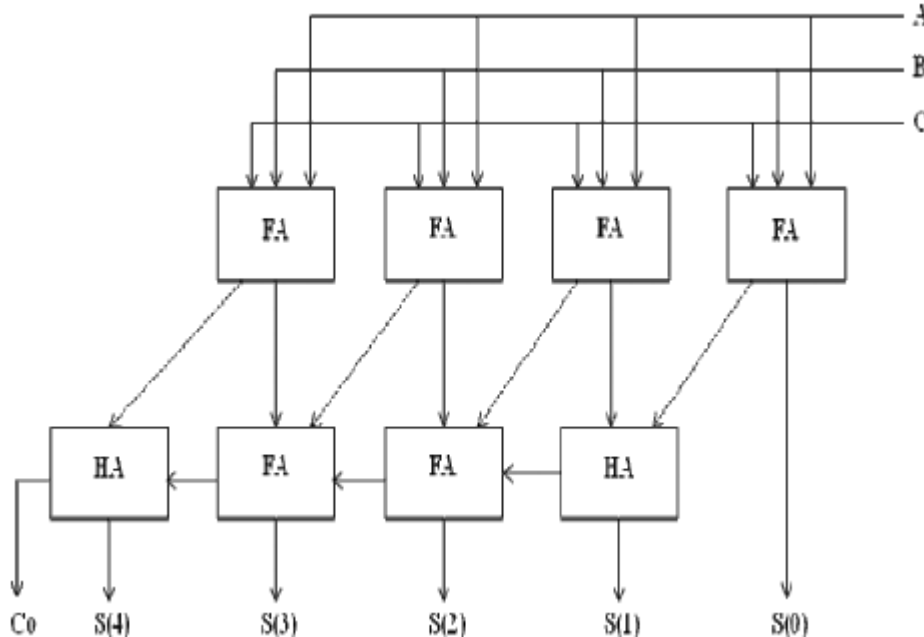


# Circuit area and time

## Exercise

Compute the time (propagation delay) and area required by the 4-bits Carry-Save-Adder, that is an adder for three values A, B and C, shown here below.

Compute the speedup of 4-bits Carry-Save-Adder with respect to the standard binary ripple-carry adder.



Delay and Area for the Ripple-carry adder

- $T_{\text{RC-adder}} = n T_{\text{FA}} = 2n T_{\text{HA}} = 4n T_{\text{gate}}$
- $A_{\text{RC-adder}} = n A_{\text{FA}} = 2n A_{\text{HA}} + n A_{\text{gate}} = 7n A_{\text{gate}}$