Introduction to Digital Systems

Part II

2020/2021

Combinational Logic Blocks

Arnaldo Oliveira, Augusto Silva, Iouliia Skliarova

Universidade de Aveiro

Lecture contents

- Block oriented combinational logic design
- Arithmetic Circuits

Addition

- Addition is a very common arithmetic operation in digital systems
- Let´s recall some concepts …

Addition of Binary Numbers

- Addition and subtraction of non-decimal numbers by hand uses the same technique that you know from school for decimal numbers.
- The only catch is that the addition and subtraction tables are different.
- To add two **unsigned binary numbers** *X* and *Y*, we add together the least significant bits with an initial carry (c_{in}) of 0, producing carry (c_{out}) and sum (s) bits according to the table. We continue processing bits from right to left, adding the carry out of each column into the next column's sum.

Example:

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

c_{in}	x	y	c_{out}	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

Universidade de Aveiro

Subtraction of Binary Numbers

- Binary subtraction is performed similarly, using borrows (b_{in} and b_{out}) instead of carries between steps, and producing a difference bit *d*.

b_{in}	x	y	b_{out}	d
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

Examples:

```
   0 1 1 1 1 0 0
   1 1 1 0 0 0 0 1
 - 1 0 1 0 1 1 0 1
 -----------------
   0 0 1 1 0 1 0 0
```

```
   1 1 1
   1 0 0 0
 - 0 0 1 1
 ---------
   0 1 0 1
```

Overflow

- With n bits it is possible to represent **unsigned integer numbers** ranging from 0 to 2^n-1.
- If an arithmetic operation produces a result that exceeds the range of the number system, **overflow** is said to occur.
- Overflows can easily be detected by analyzing **a carry or borrow from the most significant bit**.
  - the carry bit c_{out} or the borrow bit b_{out} out of the MSB = 1

Examples:

n=8: [0..255]

$173_{10} + 97_{10} = 270_{10}$

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

overflow

n=4: [0..15]

$4_{10} - 11_{10} = -7_{10}$

```
 1 0 1 1
   0 1 0 0
 - 1 0 1 1
 ---------
   1 0 0 1
```

overflow

Universidade de Aveiro

Representation of Negative Numbers

- There are many ways to represent negative numbers.
- In everyday business we use the **signed-magnitude system** (i.e. reserve a special symbol to indicate whether a number is negative).
- However, most computers use **two's-complement representation**:
  - The **most significant bit** (**MSB**) of a number in this system serves as the sign bit; a number is negative if and only if its MSB is 1.
  - The weight of the MSB is negative: for an *n*-bit number the weight is -2^{n-1}.
  - The decimal equivalent for a two's-complement binary number is computed the same way as for an unsigned number, except that the weight of the MSB is negative:
    - D= $d_{n-1}d_{n-2} \dots d_1 d_0 = -2^{n-1} + \sum_{i=0}^{n-2} d_i \times 2^i$

Examples:

$1010_2 = ???_{10}$ $\quad$ $1010_2 = -2^3 + 2^1 = -8 + 2 = -6_{10}$

$1111_2 = ???_{10}$ $\quad$ $1111_2 = -2^3 + 2^2 + 2^1 + 2^0 = -8 + 4 + 2 + 1 = -1_{10}$

$0111_2 = ???_{10}$ $\quad$ $0111_2 = 2^2 + 2^1 + 2^0 = 4 + 2 + 1 = 7_{10}$

Universidade de Aveiro

Two's Complement Representation

- For *n* bits, the range of representable numbers is $[-2^{n-1}, 2^{n-1}-1]$.
- For *n*=4, the range is [-8, 7]:

0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
-8	1	0	0	0
-7	1	0	0	1
-6	1	0	1	0
-5	1	0	1	1
-4	1	1	0	0
-3	1	1	0	1
-2	1	1	1	0
-1	1	1	1	1

Universidade de Aveiro

Towards implementation

- The immediate approach
  - Digit-wise addition and carry propagation
  - Iterative hardware
  - Building blocks
    - Half-Adder
    - Full-Adder

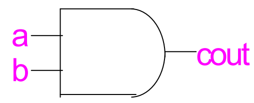
Universidade de Aveiro

The Half-Adder (HA)

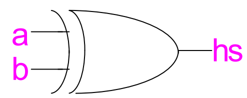
- Inputs: 2 single bit operands (a, b)
- Outputs
  - A 2 bit result:
    - The Half-Sum (hs)
    - The Carry-out (Cout)
    - Note that: $0 \leq (\text{Cout}, \text{hs})_{10} \leq 2$

a	b	c_{out}	hs
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$c_{out} = a \cdot b$$



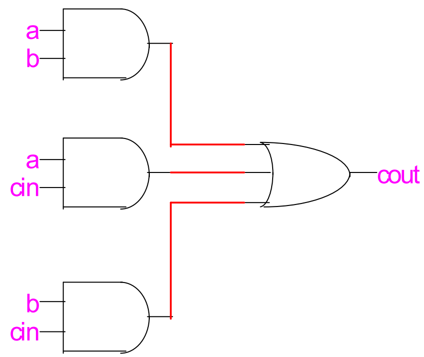
$$hs = a \oplus b$$



The Full-Adder (FA)

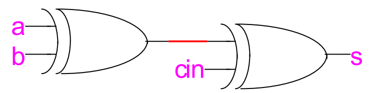
- Inputs: 2 single bit operands (a, b) and a Carry-in bit (C_{in})
- Outputs
  - A 2 bit result:
    - The Sum (S)
    - The Carry-out (C_{out})
    - Note that: $0 \leq (C_{out}, S)_{10} \leq 3$

c_{in}	a	b	c_{out}	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



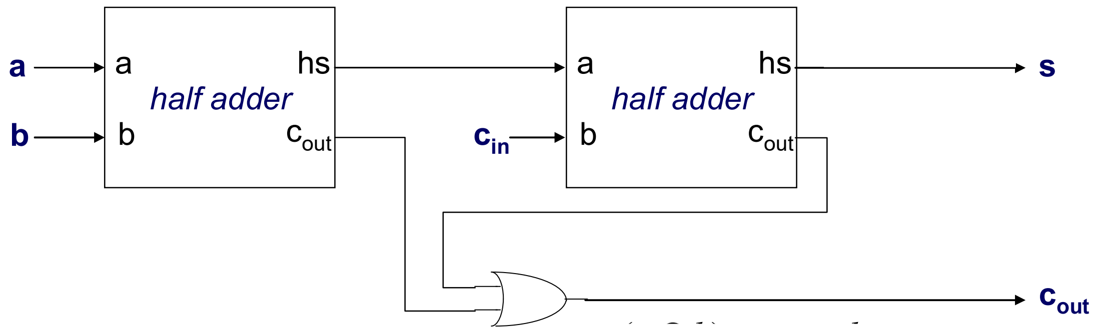
$$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$$

$$s = a \oplus b \oplus c_{in}$$



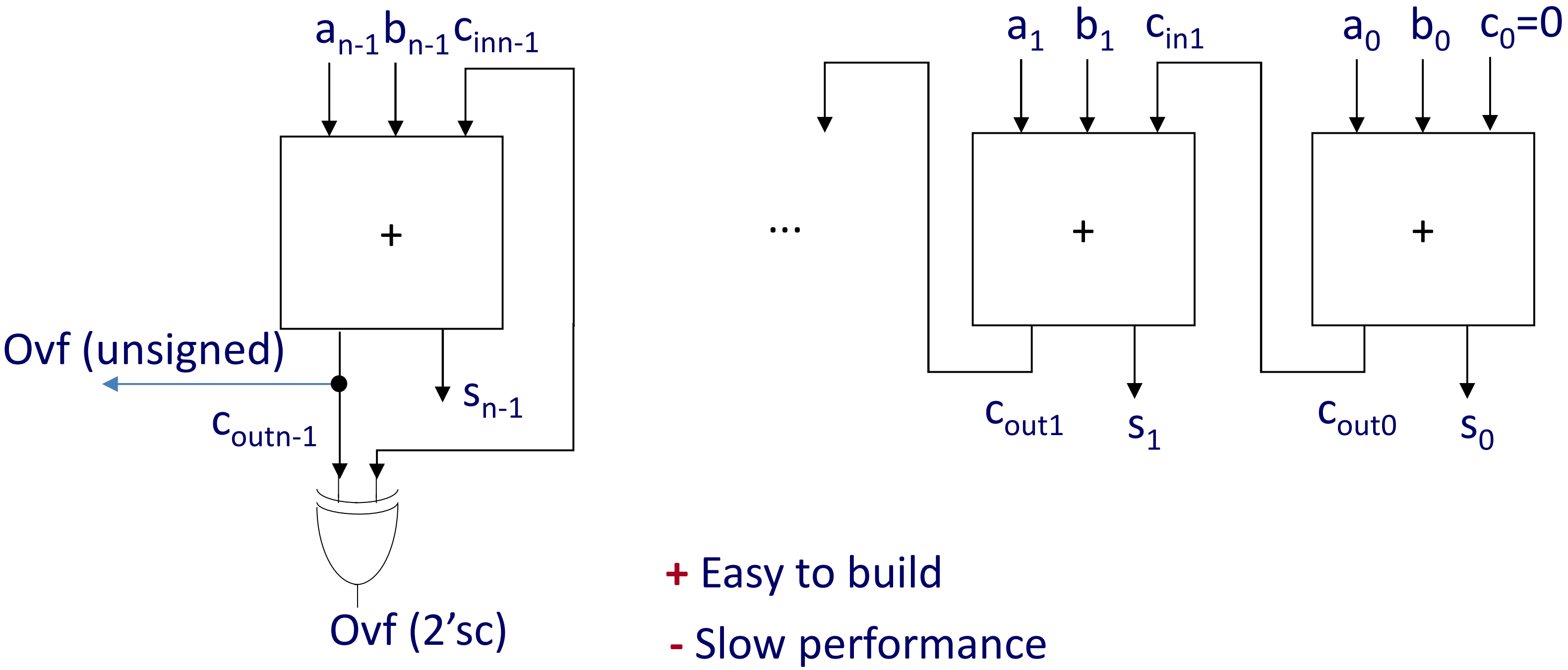
Exercise

- Show that the following circuit is Full-Adder



Ripple Adders

- The bit-wise addition and carry-propagation is implemented by a cascade of Full-Adders
- An iterative circuit paradigm



\+ Easy to build

\- Slow performance

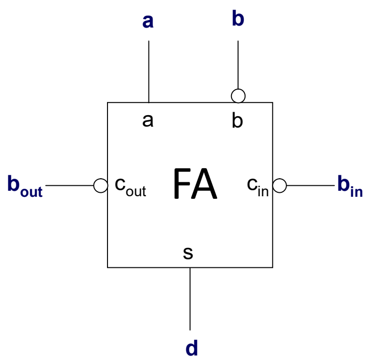
Full Subtractor

- Inputs: 2 single bit operands (a, b) and a Borrow-in bit (b_{in})
- Outputs
  - A 2 bit result:
    - The difference (d)
    - The Borrow-out (b_{out})
- Exercise

b_{in}	a	b	b_{out}	d
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

$$b_{out} = \overline{a} \cdot b + \overline{a} \cdot b_{in} + b \cdot b_{in}$$

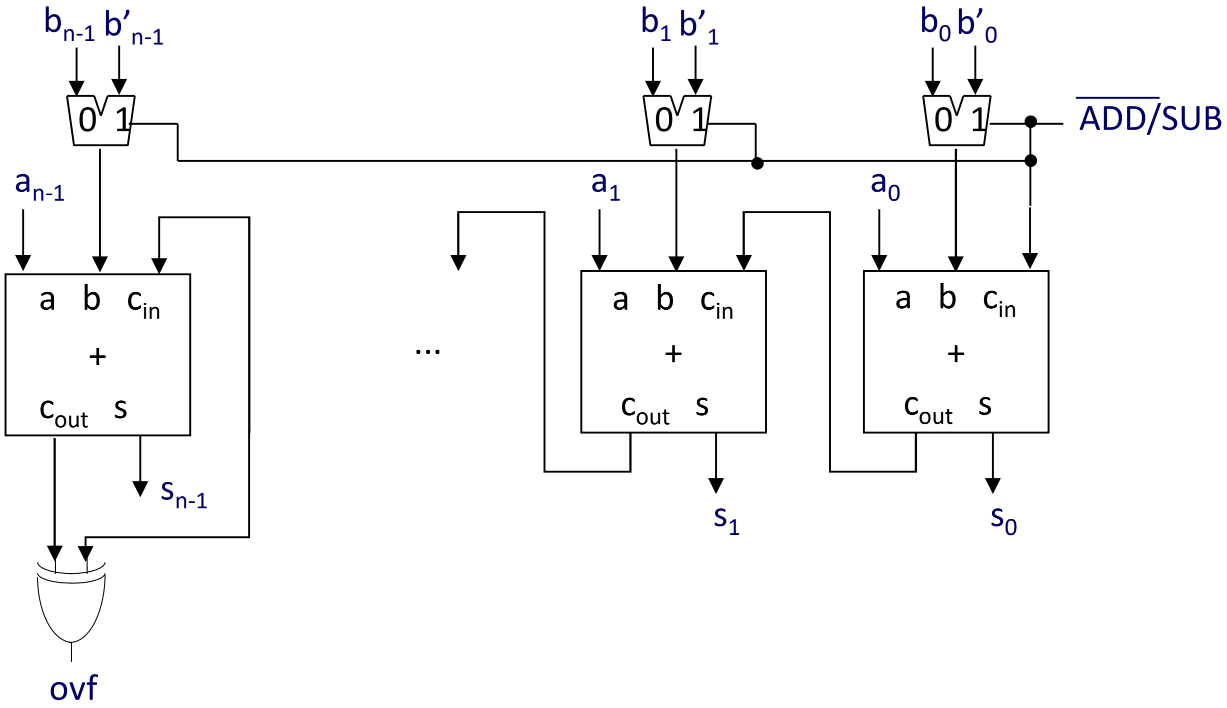
$$d = a \oplus b \oplus b_{in}$$



Write the output equations and show that the Full-Adder with modified inputs is a subtractor

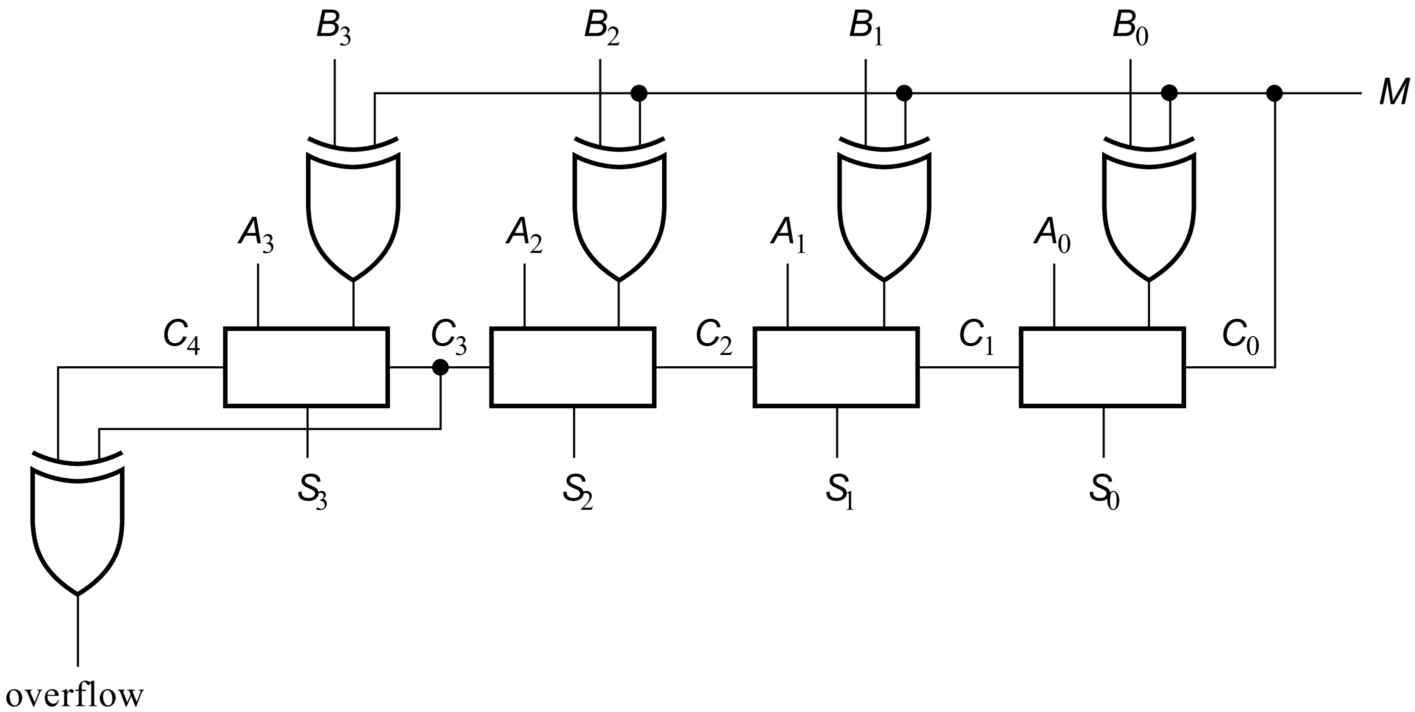
Ripple Adder/Subtractor

- Using 2's complement representation means the same hardware for addition and subtraction
  - "Muxed" b and b' inputs.
  - Initial carry = 1 if subtraction

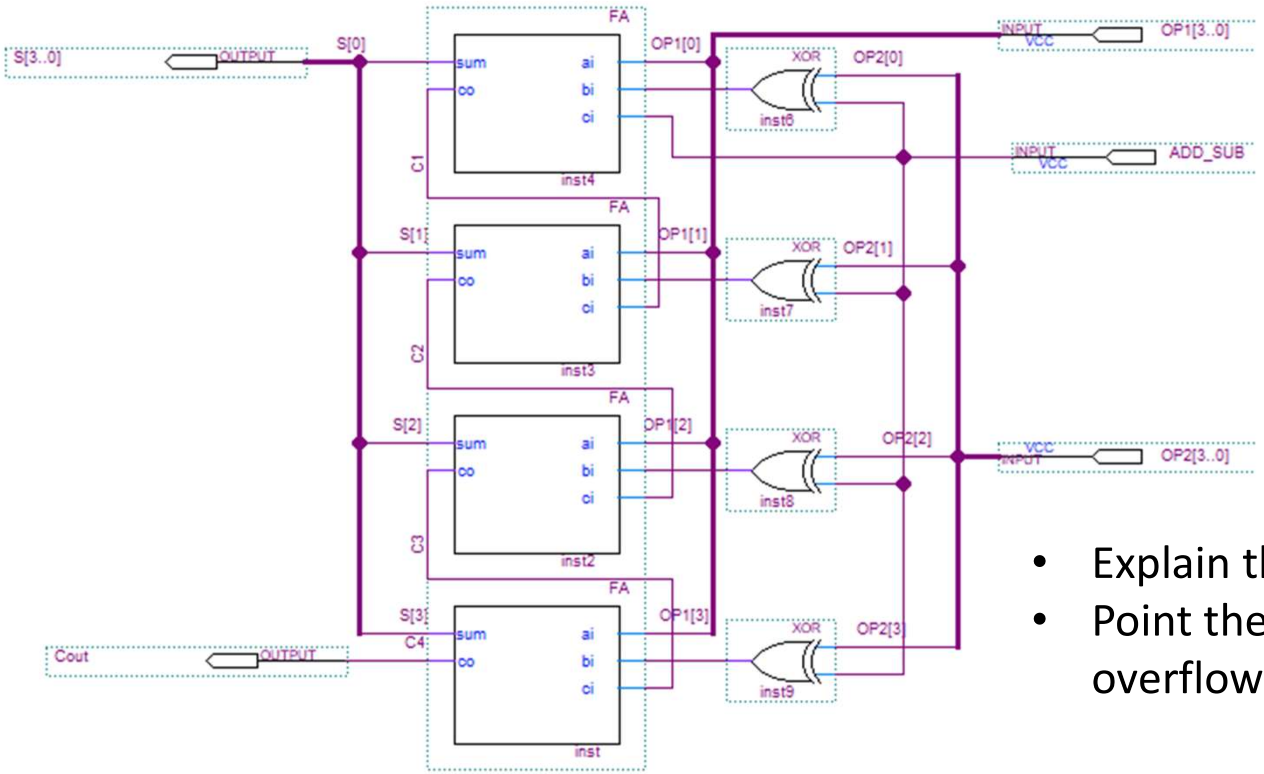


Exercise

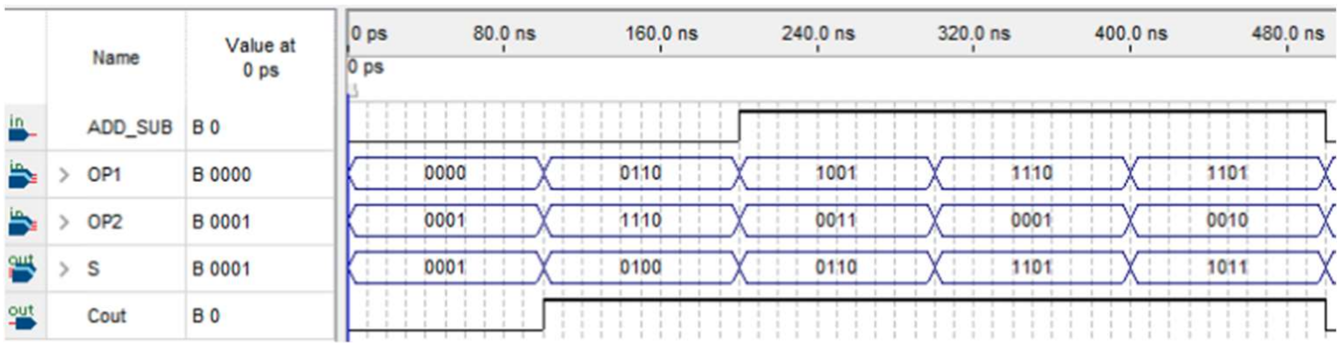
- Verify that the following implementation is equivalent to the adder/subtractor circuit of the previous slide



Exercise



- Explain the timing diagram.
- Point the time intervals where overflow occurs



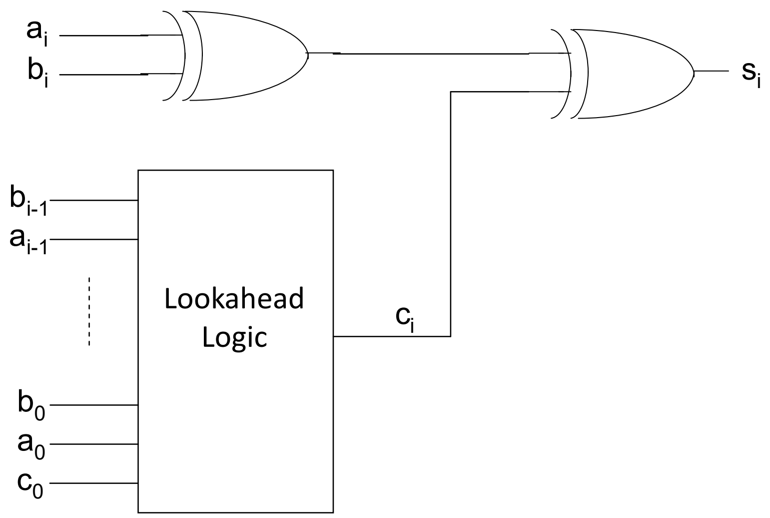
Universidade de Aveiro

Carry-Lookahead Adders (CLA)

- The idea: Compute the *ith* carry non-iteratively
- The starting points:
  - The usual equations

$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$



  - Intermediate functions
    - Carry Generation
    - Carry Propagation

$$g_i = a_i \cdot b_i$$

$$p_i = a_i + b_i$$

Remarks about the Carry

- At any stage, we necessarily have a carry generation $C_{i+1} = 1$ whenever $a_i = b_i = 1$. So $g_i = a_i.b_i$
- If $a_i \neq b_i$ but $a_i+b_i = 1$ the carry propagates since $C_{i+1} = C_i$ so $p_i = a_i + b_i$
- Finally the Carry equation becomes

$$c_{i+1} = g_i + p_i \cdot c_i$$

Universidade de Aveiro

The 4 bit CLA

- Carry equations

$$c_1 = g_0 + p_0 \cdot c_0$$

$$c_2 = g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0$$

$$c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) =$$
$$= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

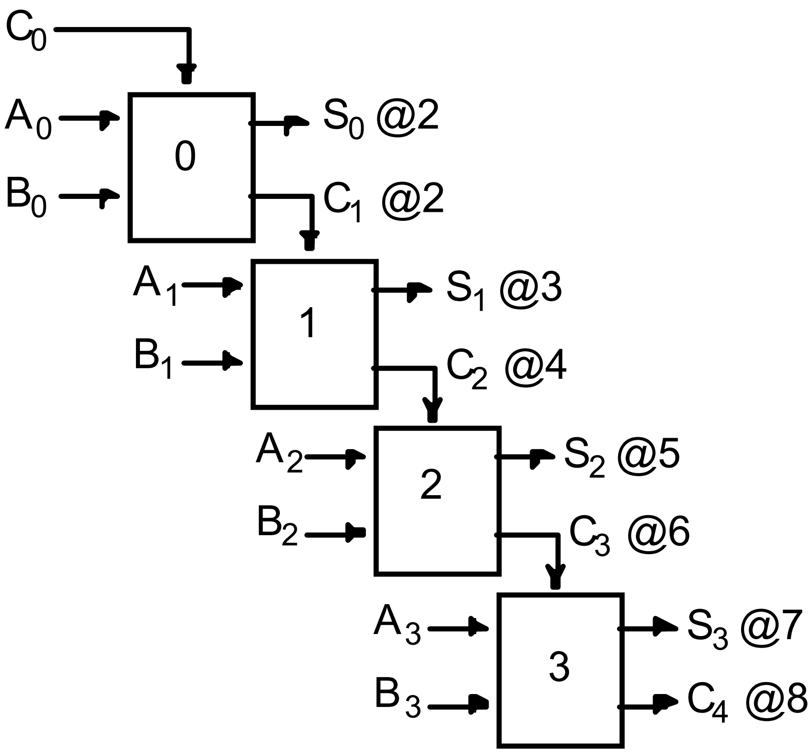
$$c_4 = g_3 + p_3 \cdot c_3 = g_3 + p_3 \cdot (g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0) =$$
$$= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0$$

- Notice that any Carry is determined after 3 delay levels

Universidade de Aveiro

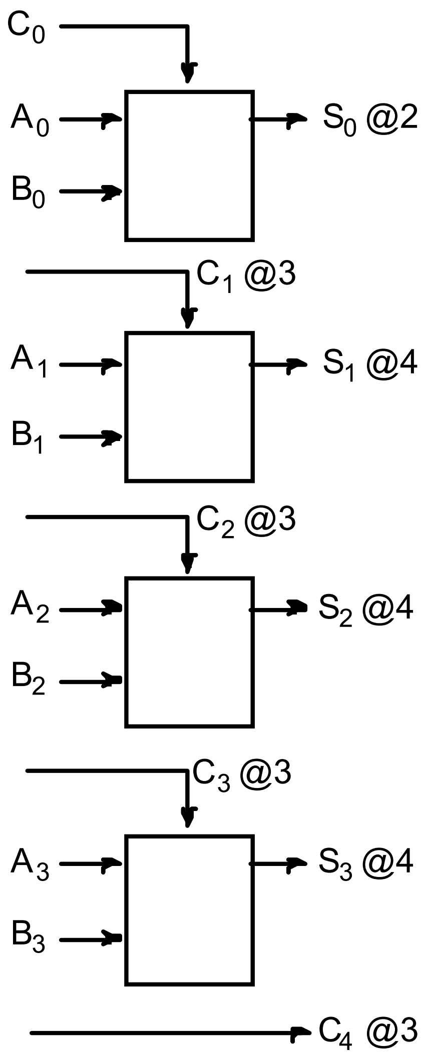
Ripple Adder versus CLA

- 4 bit ripple adder



Final result always after 2x4 = 8 delays

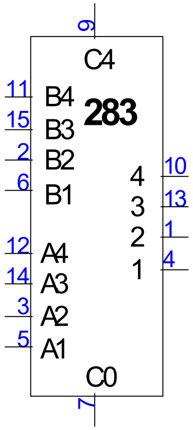
- 4 bit CLA

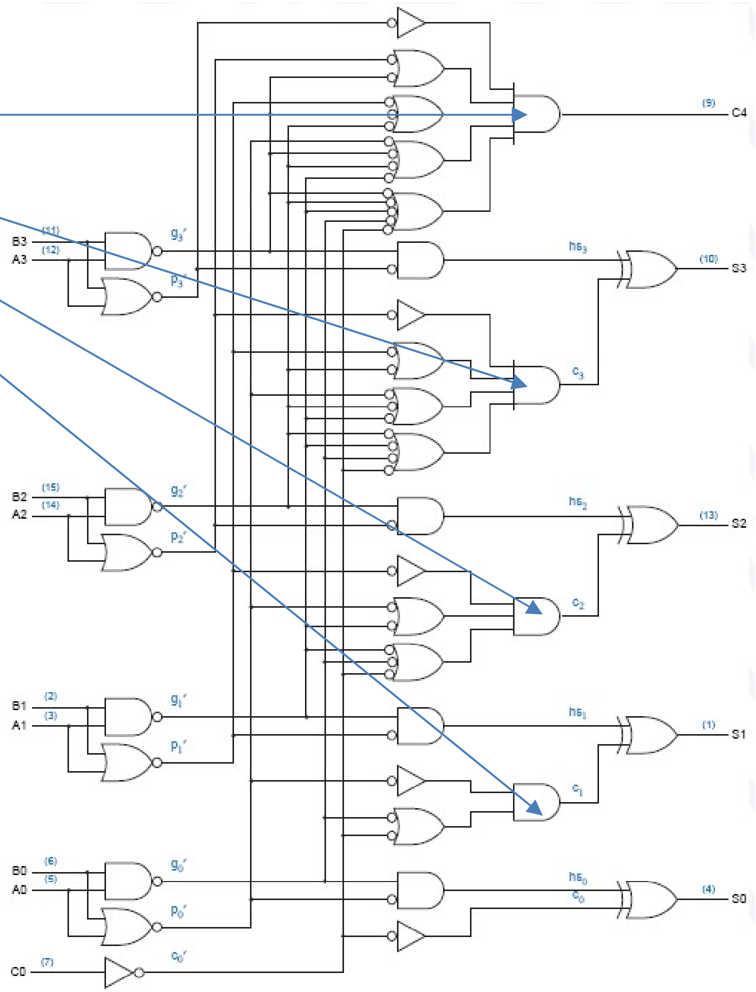


Final result always after 4 delays

The 74283 model

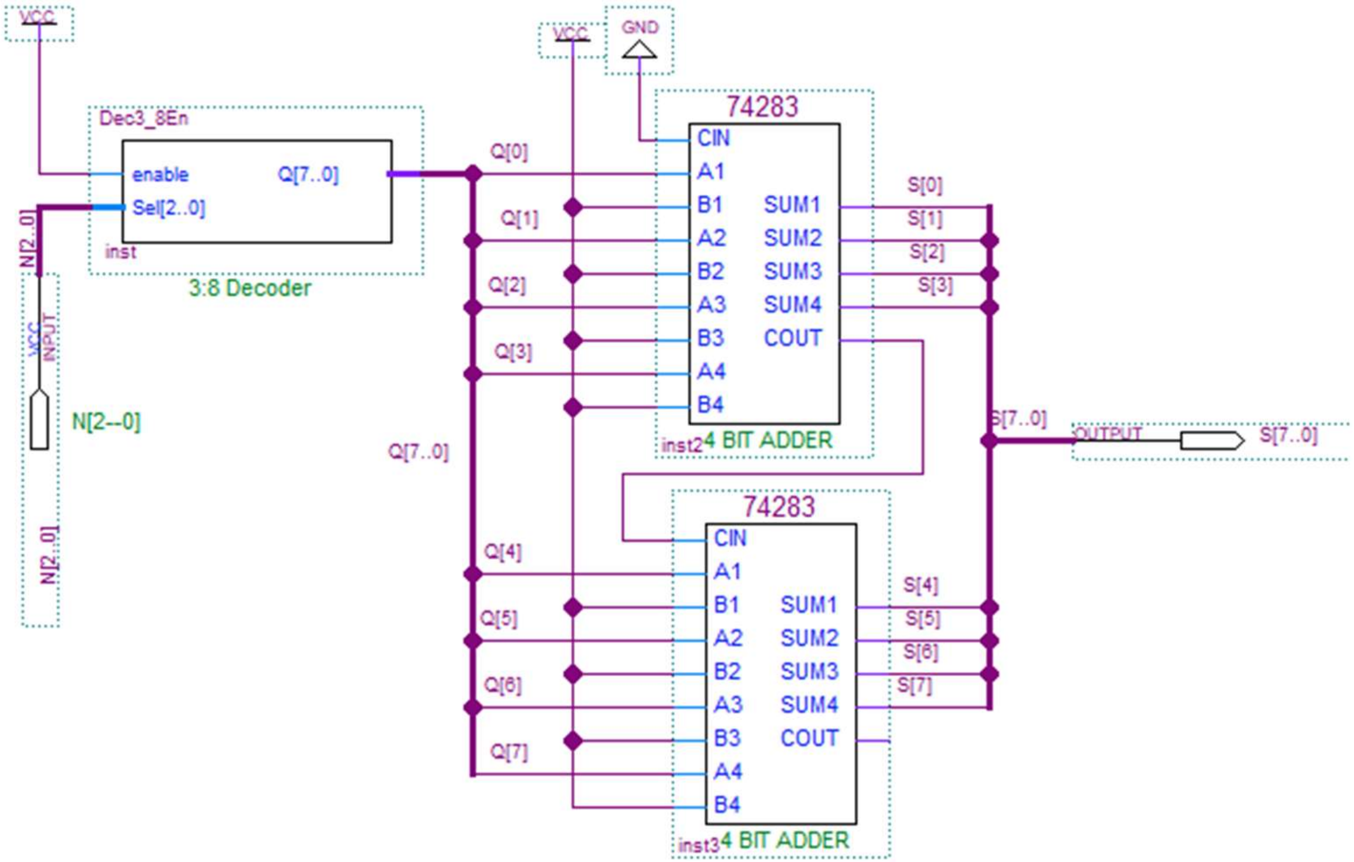
- CLA logic





Exercise

- In a 2'sc complement representation, what's the decimal value of the result S with N = 6_{10}



Universidade de Aveiro

BCD Addition

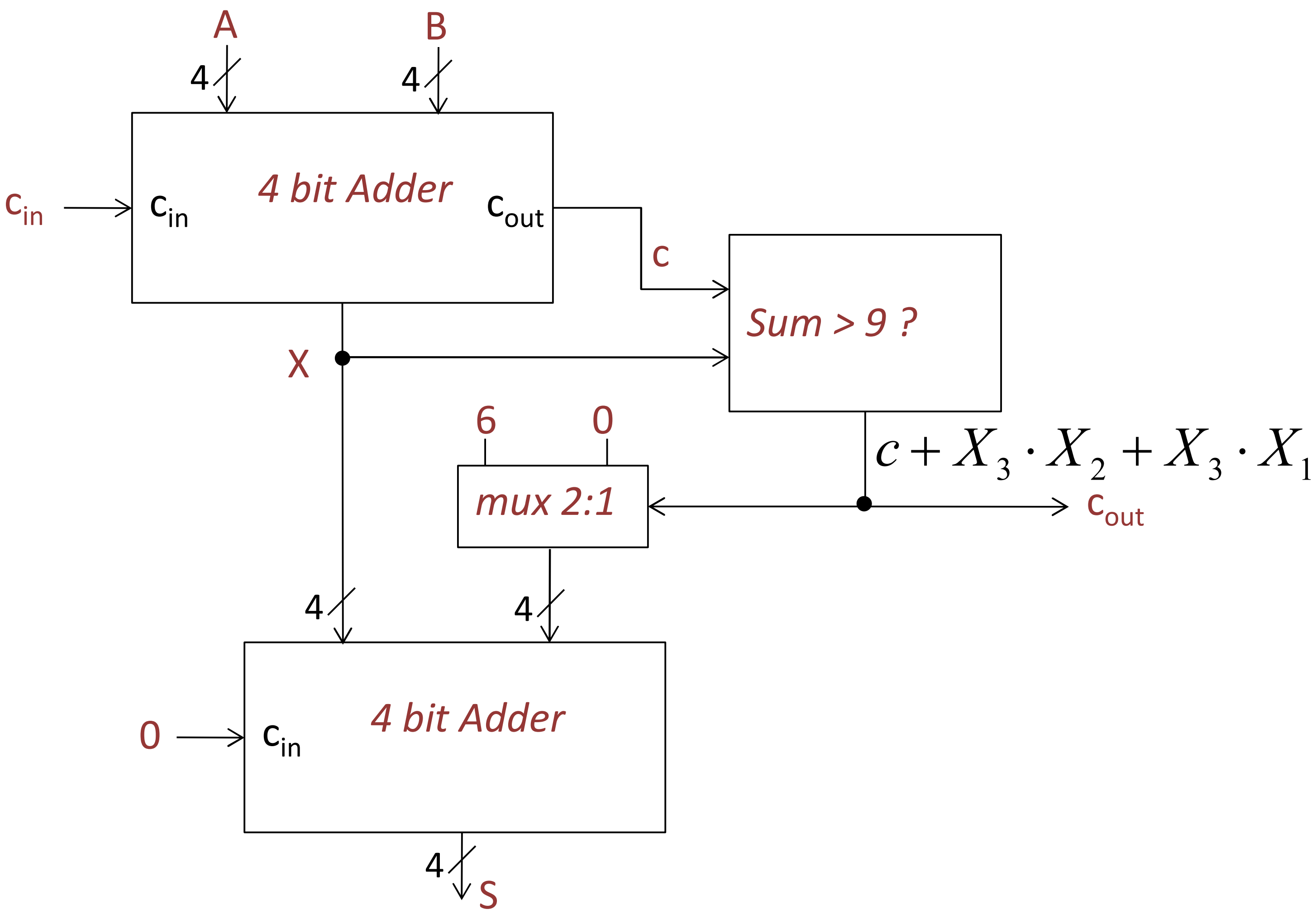
- Possible results for 2 digit BCD addition with carry

soma	binário		BCD	
	carry out	soma	*carry out*	soma
0	0	0000	0	0000
1	0	0001	0	0001
2	0	0010	0	0010
3	0	0011	0	0011
4	0	0100	0	0100
5	0	0101	0	0101
6	0	0110	0	0110
7	0	0111	0	0111
8	0	1000	0	1000
9	0	1001	0	1001
10	0	1010	1	0000
11	0	1011	1	0001
12	0	1100	1	0010
13	0	1101	1	0011
14	0	1110	1	0100
15	0	1111	1	0101
16	1	0000	1	0110
17	1	0001	1	0111
18	1	0010	1	1000
19	1	0011	1	1001

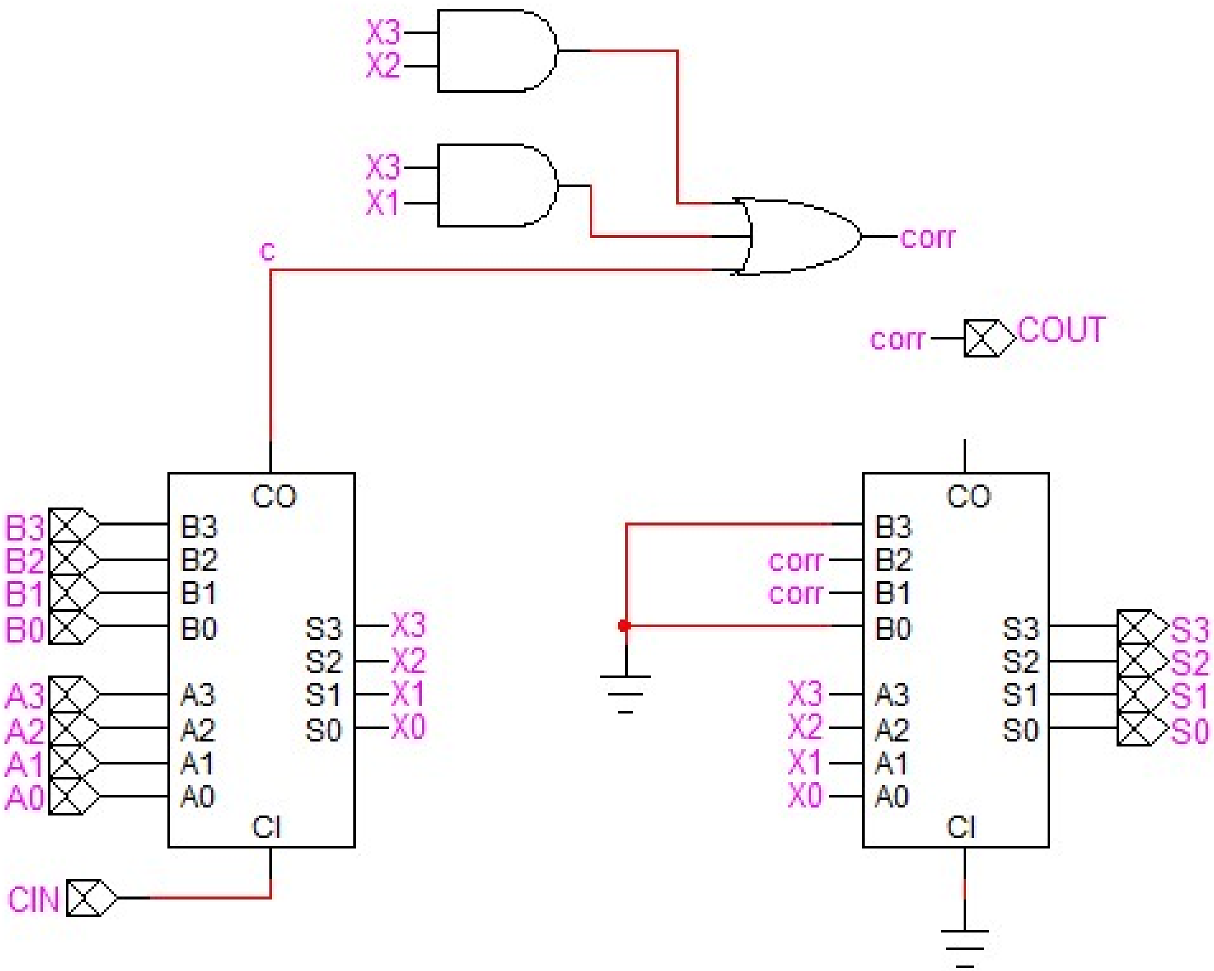
Offset correction required:
add 6 to the binary result

Universidade de Aveiro

BCD addition algorithm



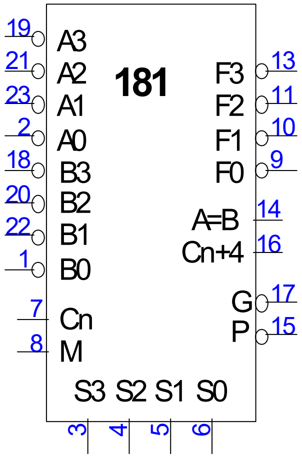
Possible implementation



ALU

- An arithmetic logic unit is a combinational block that executes any logical or arithmetic operation over a pair of *b* bits operands.
- There is a mode input that chooses between the logical or arithmetic behavior
- There is a op-code set of inputs that choose a particular operation from a limited "operation" set.

The 74181 ALU



S3	S2	S1	S0	M=0 (op. aritm.)	M=1 (op. lógica)
0	0	0	0	F = A – 1 + CIN	F = $\overline{A}$
0	0	0	1	F = A *AND* B – 1 + CIN	F = $\overline{A}$ *OR* $\overline{B}$
0	0	1	0	F = A *AND* $\overline{B}$ – 1 + CIN	F = $\overline{A}$ *OR* B
0	0	1	1	F = 1111 + CIN	F = 1111
0	1	0	0	F = A + (A *OR* $\overline{B}$) + CIN	F = $\overline{A}$ *AND* $\overline{B}$
0	1	0	1	F = A *AND* B + (A *OR* $\overline{B}$) + CIN	F = $\overline{B}$
0	1	1	0	F = A – B – 1 + CIN	F = A *XOR* $\overline{B}$
0	1	1	1	F = A *OR* $\overline{B}$ + CIN	F = A *OR* $\overline{B}$
1	0	0	0	F = A + (A *OR* B) + CIN	F = $\overline{A}$ *AND* B
1	0	0	1	F = A + B + CIN	F = A *XOR* B
1	0	1	0	F = A *AND* $\overline{B}$ + (A *OR* B) + CIN	F = B
1	0	1	1	F = A *OR* B + CIN	F = A *OR* B
1	1	0	0	F = A + A + CIN	F = 0000
1	1	0	1	F = A *AND* B + A + CIN	F = A *AND* $\overline{B}$
1	1	1	0	F = A *AND* $\overline{B}$ + A + CIN	F = A *AND* B
1	1	1	1	F = A + CIN	F = A

Unsigned Multiplication

- We follow the same rules of the decimal system

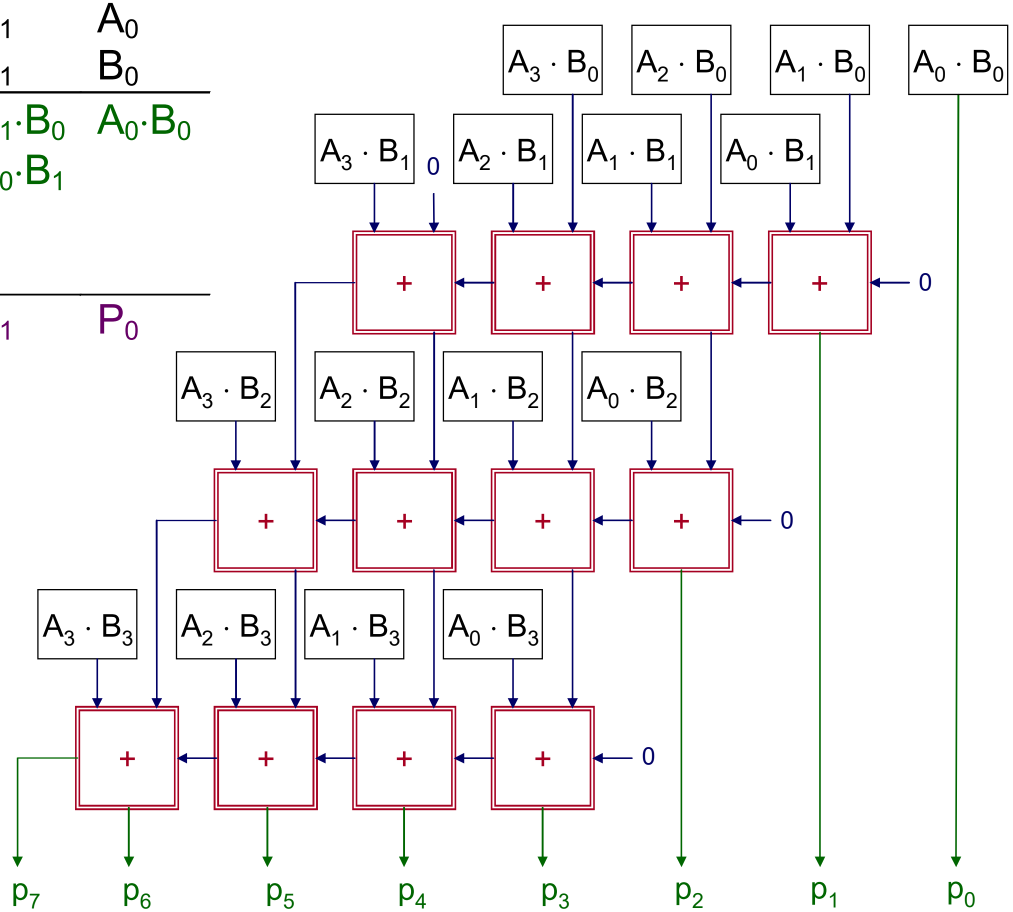
$12 \times 13 = 156$

					1	1	0	0	← multiplicand
				×	1	1	0	1	← multiplier
					1	1	0	0	Partial products
				0	0	0	0		
			1	1	0	0			
		1	1	0	0				
	1	0	0	1	1	1	0	0	Final product

Combinational Multipliers

				A_3	A_2	A_1	A_0
			$\times$	B_3	B_2	B_1	B_0
				$A_3 \cdot B_0$	$A_2 \cdot B_0$	$A_1 \cdot B_0$	$A_0 \cdot B_0$
			$A_3 \cdot B_1$	$A_2 \cdot B_1$	$A_1 \cdot B_1$	$A_0 \cdot B_1$	
		$A_3 \cdot B_2$	$A_2 \cdot B_2$	$A_1 \cdot B_2$	$A_0 \cdot B_2$		
	$A_3 \cdot B_3$	$A_2 \cdot B_3$	$A_1 \cdot B_3$	$A_0 \cdot B_3$			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

					1	1	0	0
				$\times$	1	1	0	1
					1	1	0	0
				0	0	0	0	
			1	1	0	0		
		1	1	0	0			
	1	0	0	1	1	1	0	0



16 $\times$ AND-2
12 adders

Final Remarks

- Always recall
  - The block symbol
  - The types of inputs (operands) and outputs
  - Distinguish between iterative and non-iterative solutions
- Design with encapsulated logic requires mastering all the functional details of each block

Universidade de Aveiro