

# UNIT 3: INTEGER ARITHMETIC

---

Estructura de Computadores (Computer Organization)

Course 2018/2019

ETS Ingeniería Informática

Universitat Politècnica de València

# Unit Goals

- To map high-level numeric types (C, Java...) to processor native types
- To translate high-level arithmetic expressions to assembly
- To obtain response times and productivity of both sequential and combinational operators
- To implement arithmetic operations directly and using bit-wise operations
- To understand the basic response mechanism of a processor in the presence of arithmetic overflow errors

# Unit contents

- 1. Introduction
  - Data types
  - Operations and operators
  - Logical operations and conditional branches
- 2. Integer addition and subtraction
  - Fundamentals
  - Addition and subtraction in MIPS R2000
  - Addition operators
  - Operators for subtraction and comparison
- 3. Integer multiplication and division
  - Fundamentals
  - Multiplication and division in MIPS R2000
  - Shift operators
  - Unsigned multiplication operators
  - Signed multiplication operators

# Bibliography

- D. Patterson, J. Hennessy. *Computer organization and design. The hardware/software interface*. 4<sup>th</sup> edition. 2009. Elsevier
  - Chapter 3, Appendix C
- W. Stallings. *Computer Organization and Architecture. Designing for Performance*. 7<sup>th</sup> edition. 2006. Prentice Hall
- D. Goldberg: Computer Arithmetic
  - Appendix H of J. Hennessy, D. Patterson: *Computer Architecture: A Quantitative Approach*. 4<sup>th</sup> Edition. Morgan-Kaufmann, 2002
    - (PDF) <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.3375&rep=rep1&type=pdf>
    - (PDF) <http://www.cs.clemson.edu/~mark/464/appH.pdf>

# 1. Introduction

- Data types
- Operations and operators
- Logical operations and conditional branches

# High-level types in low-level

	bits	Java	C/C++ (32 bits)	x86 (16)	MIPS32
Character	8	—	char	byte	byte
	16	char	wchar_t	word	halfword
Unsigned integer	8	—	unsigned byte	byte	byte
	16	—	unsigned short	word	halfword
	32	—	unsigned long	dword	word
Signed integer	8	byte	byte	byte	byte
	16	short	short	word	halfword
	32	int	long	dword	word
	64	long	—	qword	—
Floating point	32	float	float	float	float
	64	double	double	double	double

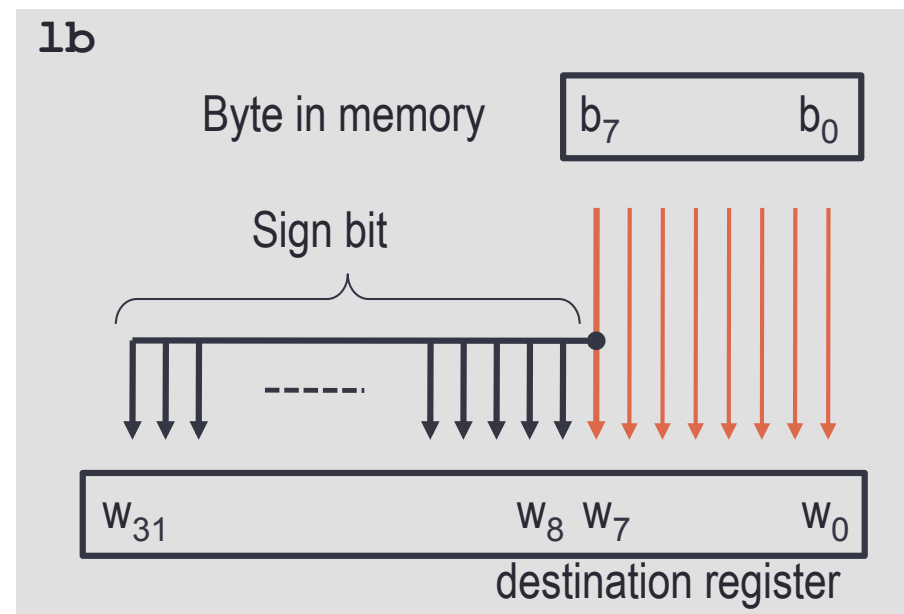
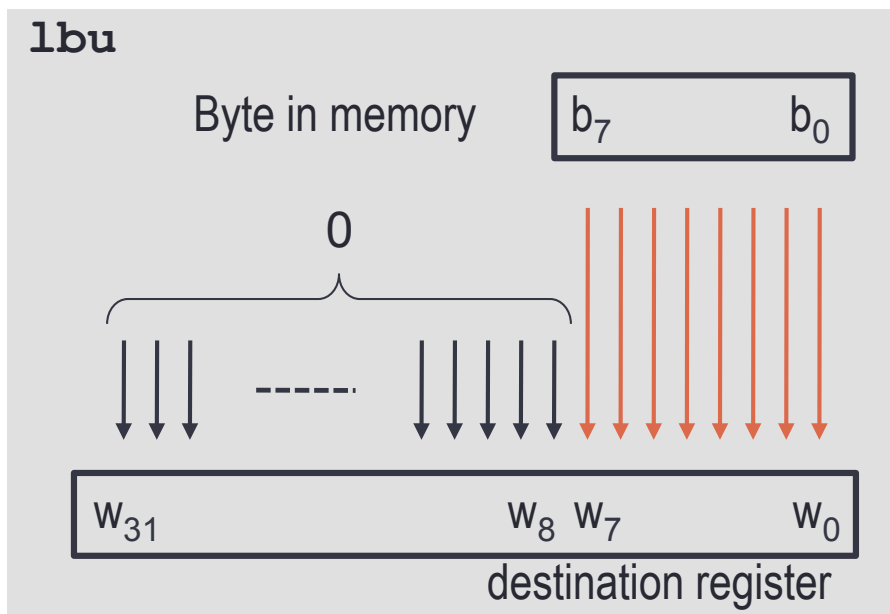
# Integer data types

- The N and Z sets are unbounded, but they have to be represented using a bounded number of bits. With  $n$  bits, only up to  $2^n$  values can be represented
  - Integer representation is *exact*, but *bounded*
- Ranges with  $n$  bits
  - N (naturals): Natural Binary Code (NBC)  $\rightarrow [0 \dots +2^n-1]$
  - Z (integers): Two's complement (2'sC)  $\rightarrow [-2^{n-1} \dots +2^{n-1}-1]$

n	Unsigned (NBC)	Signed (2'sC)
8	0 ... 255	-128 ... +127
16	0 ... 65.535	-32.768 ... +32.767
32	0 ... 4.294.967.295	-2.147.483.648 ... +2.147.483.647
64	0 ... $1.84 \times 10^{19}$ (approx)	$-9.2 \times 10^{18}$ ... $+9.2 \times 10^{18}$ (approx)

# Integer type conversions in R2000

- The *native* integer type is 32-bit wide (ALU and registers)
- Load/Store instructions convert to shorter formats if needed
  - **lbu** and **lhu** add zeroes to the left (valid for NBC)
  - **lb** and **lh** extend the sign bit (valid for 2'sC)
  - **sb** and **sh** simply discard the 24 or 16 bits to the left



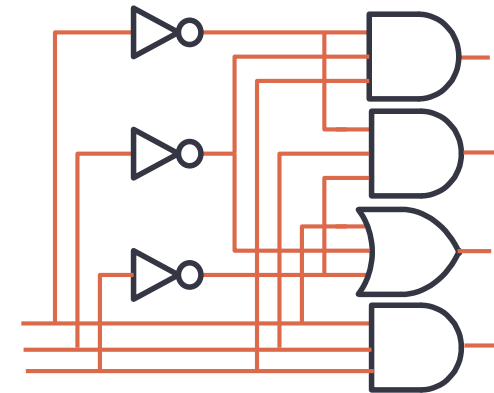


# Operations and operators

- High-level logical and arithmetic operations are translated by the compiler into assembly code and data
- During the instruction cycle, the CPU uses the proper **operators**

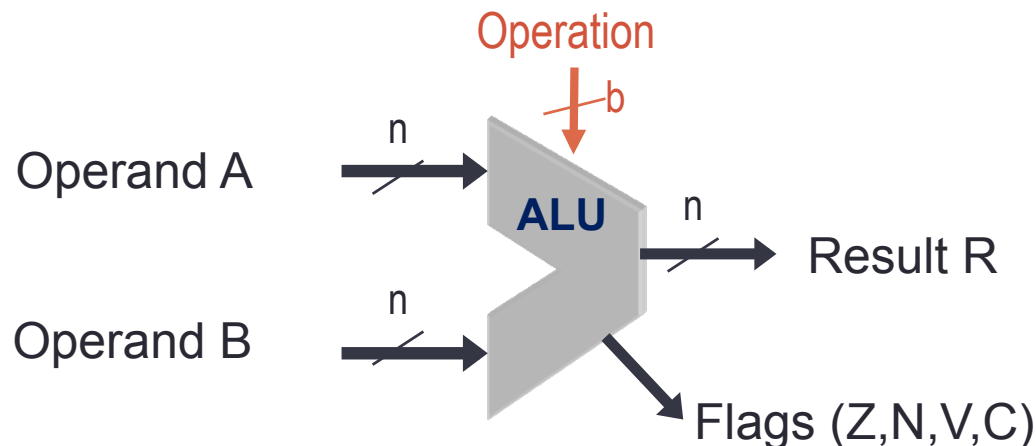
```
int[] j;  
short a;  
float x,y;  
  
x = 5*j[a];  
if (a>x){  
    y=x*1.3e5;  
    j=(int)exp(x);  
}
```

lbu \$t0,0xFF0(\$0)
lw \$t1,0x1020(\$t2)
add \$t0,\$t0,\$t1
mtc1 \$t0,\$f1
...



# Arithmetic Logic Unit

- The **Arithmetic-Logic Unit** (ALU) is the functional unit in the CPU containing digital operators to support the operations provided by the instruction set
- Each operator in the ALU implements one or more operations
- The CPU's **Control Unit** selects the operation to perform, depending on the instruction, routes the operands to its inputs and the result to the destination register
- The ALU may also activate indicators (flags) that give information about the result (typically, **Z**ero, **N**egative, **oV**erflow, and **C**arry)



# Arithmetic Logic Unit

- ALU parameters
  - Functional
    - Implemented operations
      - Type conversion
      - Bitwise operations (and, or, xor...), shifts, rotations
      - Arithmetic operations: add, sub, multiply, divide
      - Comparison ( $=$ ,  $\neq$ ,  $<$ ,  $\leq$ ,  $\geq$ ,  $>$ )
    - Types of operands
  - Performance (temporal cost)
    - Speed of the ALU
    - Number of operations per time unit
  - Complexity (spatial cost)
    - How many physical resources
    - How much area

# Performance and complexity

- Expressing performance
  - Response time
    - Time to execute an operation. The shorter the better
    - Measured in time units (ns, gate time,...)
  - Productivity or throughput
    - Number of operations per time unit. The larger the better
    - Measured as Operations Per Second (OPS, KOPS, MOPS...)
      - In floating point operators, FLOPS, KFLOPS, MFLOPS, ...
- Expressing complexity
  - Number of logical gates
  - Number of transistors
  - Chip area ( $\mu\text{m}^2$ ,  $\text{nm}^2$ )

# Logical operations

- Java and C equivalents to **&** (and) **|** (or) **^** (xor) **~** (not)

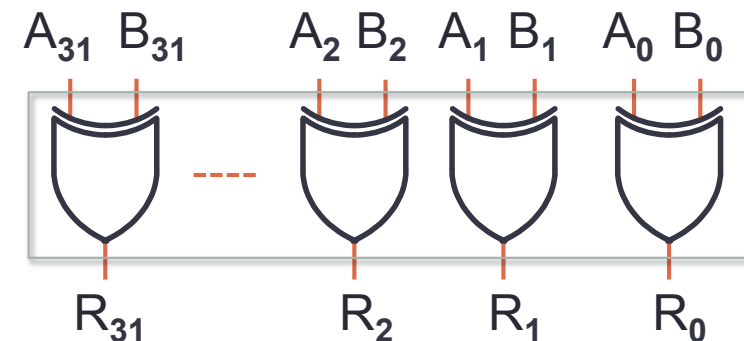
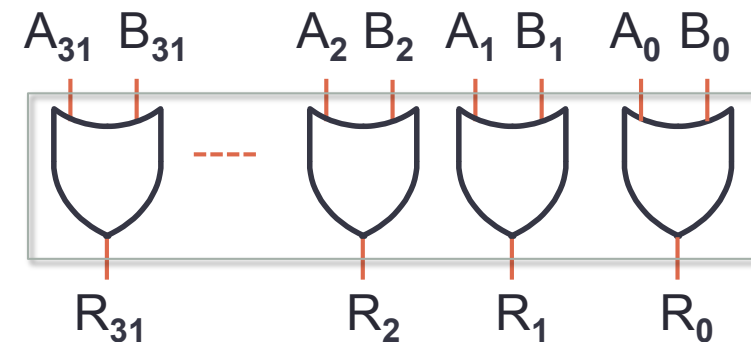
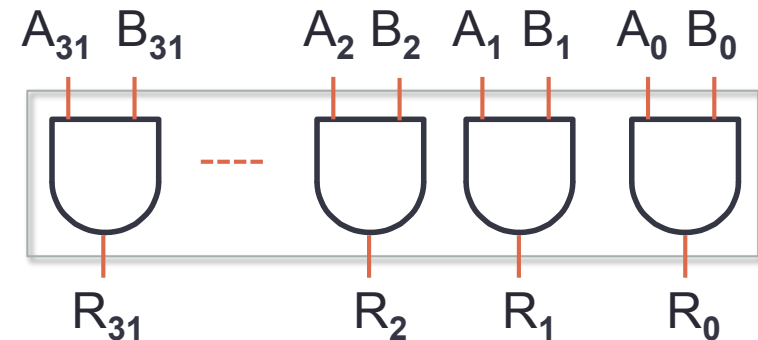
```
int a = 0xA;  
int b = 0x3;  
int c;  
...  
c = a & ~b;  
...  
System.out.println  
    (a + " ^ ~" + b + " = " + c);
```

lw \$t0,a
lw \$t1,b
xori \$t1,\$t1,-1
and \$t0,\$t0,\$t1
sw \$t0,c

$$10 \wedge \sim 3 = 8$$

# Logical operations

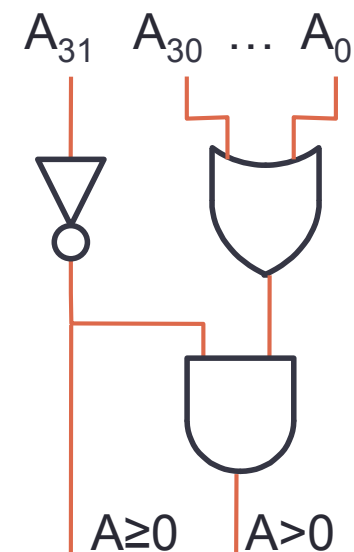
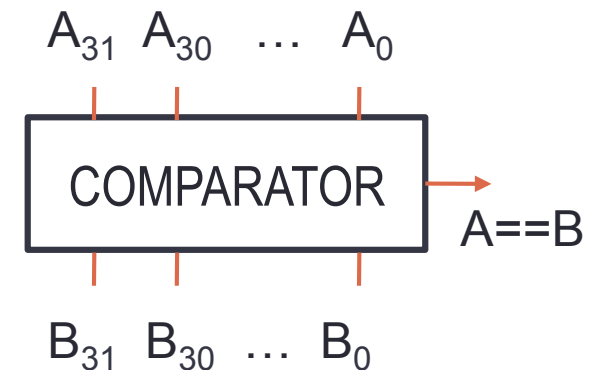
- Logical operations:
  - R format: **or**, **and**, **xor**, **nor**
  - I format: **ori**, **andi**, **xori**
  - We can easily derive **not** (one's complement):
    - **nor** rdst, rsrc, \$0
    - **xori** rdst, rsrc, -1
- Productivity =  $1/t_{\text{gate}}$ 
  - E.g.:  $t_{\text{gate}} = 100 \text{ ps} \rightarrow P = 10 \text{ GOPS}$



# Conditions and branches

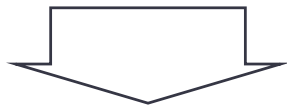
- Conditional branches in MIPS
  - Equality Instructions
    - **beq**: branch if equal
    - **bne**: branch if not equal
  - Equality pseudoinstructions
    - **beqz** and **bnez** derive from **beq** and **bne**
  - Tests for inequalities (one operand against zero)
    - **bgez**  $\rightarrow$  condition:  $A_{31} = 0$
    - **bgtz**  $\rightarrow$  condition:  $A_{31} = 0$  and  $A_{30}..A_0 \neq 0$
    - **bltz**  $\rightarrow$  condition:  $A_{31} = 1$
    - **blez**  $\rightarrow$  condition:  $A_{31} = 1$  or  $A_{31}..A_0 = 0$

## Some condition operators



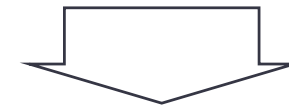
# Condition examples

```
int i;  
...  
do {  
    ...  
} while (i>=0);
```



```
lbl_do: ...  
    ...  
    lw $t0,i  
    bgez $t0, lbl_do
```

```
int i,j;  
...  
if ((i==j)&&(j>0))  
    ...  
else  
    ...
```



```
lw $t0,i  
lw $t1,j  
bne $t0,$t1, lbl_else  
blez $t1, lbl_else  
...  
lbl_else:  
    ...
```

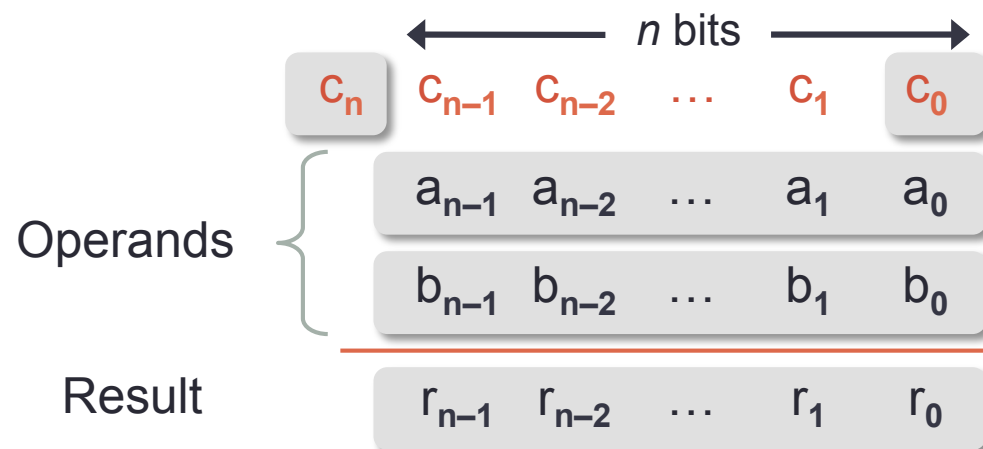


## 2. Integer addition and subtraction

- Fundamentals
- Addition and subtraction in MIPS R2000
- Addition operators
- Subtraction and comparison operators

# Anatomy of addition

- The general addition procedure calculates  $R = A + B + c_0$



Carry bits:

- $c_0$  (input carry): usually  $c_0 = 0$
- $c_{n-1} \dots c_1$ : part of the calculation
- $c_n$  (output carry): may be useful

- Same procedure both for signed (2'sC) and unsigned (NBC) integers

← 4 bits →	NBC	2'sC
0 1 0 1	5	+5
+ 1 0 0 1	+ 9	+ -7
<hr/>	<hr/>	<hr/>
1 1 1 0	14	-2

# Detecting overflow

- In **unsigned NBC**, overflow = ( $c_n = 1$ )
- In **signed 2'sC**, overflow can only occur when both operands have the same sign. Two ways to detect it:
  - The sign of the result is contrary to the operands':
    - $a_{n-1} = b_{n-1} \neq r_{n-1}$  or  $(a_{n-1} \odot b_{n-1}) \cdot (b_{n-1} \oplus r_{n-1}) = 1$
  - Or the two last carry bits of the result are different
    - $c_n \neq c_{n-1}$  or  $(c_n \oplus c_{n-1}) = 1$
- Examples with  $n = 4$  bits

NBC:

$c_n \rightarrow$	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	
		0	1	1	0
	+	1	1	1	0
		<hr/>			
		0	1	0	0

6  
14  
~~4~~

2'sC:

$\rightarrow$	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	
		0	1	1	0
	+	0	1	0	1
		<hr/>			
$\rightarrow$	<b>1</b>	0	1	1	

+6  
+5  
~~-5~~

# Subtraction

- Subtraction can be transformed into addition
  - $R = A + (-B)$
  - To obtain  $(-B)$  we calculate the 2's complement of B
    - $R = A + 2'sC(B) = A + \text{not}(B) + 1$

$R = A - B$						$R = A + (-B)$					
<b>0</b>	<b>0</b>	<b>0</b>	<b>1</b>		NBC		<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>
	1	1	1	0				1	1	1	0
-	0	1	0	1	-	5	+	1	0	1	0
<hr/>							<hr/>				
	1	0	0	1		9		1	0	0	1


$-5 = \text{not}(5) + 1$

- Carry in addition is the complement to borrow in subtraction
- Overflow detection in NBC:  $(c_n = 0)$
- Overflow detection in 2'sC:  $(c_n \neq c_{n-1})$

# Subtraction (used for comparison)

- The result of a comparison is a single bit indicating whether the inequality holds or not
- A comparison's result such as  $A < B$  follows from subtracting ( $R = A - B$ ) and analysing carry and sign of  $R$ 
  - The actual value of  $R$  would be irrelevant in this case
- Considering all cases in NBC and 2'sC:

Comparison	NBC	2'sC
$A == B$	$R == 0$	$R == 0$
$A \geq B$	$c_n = 1$ ( $R$ is representable)	$R$ is not negative
$A < B$	$c_n = 0$ ( $R$ is not representable)	$R$ is negative

# Additive instructions in MIPS R2000

- 32-bit operands
- Register-register and register-immediate versions

Operation	Format	Signed	Unsigned
Addition	R	<b>add</b>	<b>addu</b>
Addition	I	<b>addi</b>	<b>addiu</b>
Subtraction	R	<b>sub</b>	<b>subu</b>
Comparison	R	<b>slt</b>	<b>sltu</b>
Comparison	I	<b>slti</b>	<b>sltiu</b>

- **add/addu**: same operation, but different reaction to overflow
- All constants in I format instructions are sign-extended
- No immediate subtraction → use negative constants
  - eg. **addi \$t0, \$t0, -1**

# Semantics of additive instructions

- **add rd,rs1,rs2**  
**addi rd,rs,imm**
  - Cause an exception if (signed) overflow; **rd** not modified in that case
- **addu rd,rs1,rs2**  
**addiu rd,rs,imm**
  - Don't check overflow; never cause exception (modular addition)
- **sub rd,rs1,rs2**
  - Cause an exception if (signed) overflow; **rd** not modified in that case
- **subu rd,rs1,rs2**
  - Don't check overflow; never cause exception

# Overflow detection in high-level

- Integer arithmetic in languages such as C and Java ignores overflow. Java example:

```
int a,b,c;  
a = 2000000000; // 0x77359400  
b = 1000000000; // 0x3B9ACA00  
c = a + b;  
System.out.println  
    (a + " + " + b + " = " + c);
```

2000000000 + 1000000000 = -1294967296



- The compiler will use **addu** instead of **add** for  $c = a + b$ ;
- Overflow must be detected by user's code

```
if ((a^b)>=0 && ((b^c)<0))  
    System.out.println("Overflow!");
```

```
if ((a^b)>=0 && ((b^c)<0))  
    throw new ArithmeticOverflowException;
```



# Overflow detection in low-level

- Detection by sign analysis:

```
int a,b,c;  
c = a + b;
```

```
lw    $t0,a  
lw    $t1,b  
addu  $t2,$t0,$t1  
xor    $t3,$t0,$t1  
bltz   $t3,OK  
xor    $t3,$t0,$t2  
bltz   $t3,Arithmetic_Overflow  
OK: sw  $t2,c
```

Add a + b

Compare for equal signs  
of operands

Compare for equal signs  
of result and one operand

Store result if no overflow

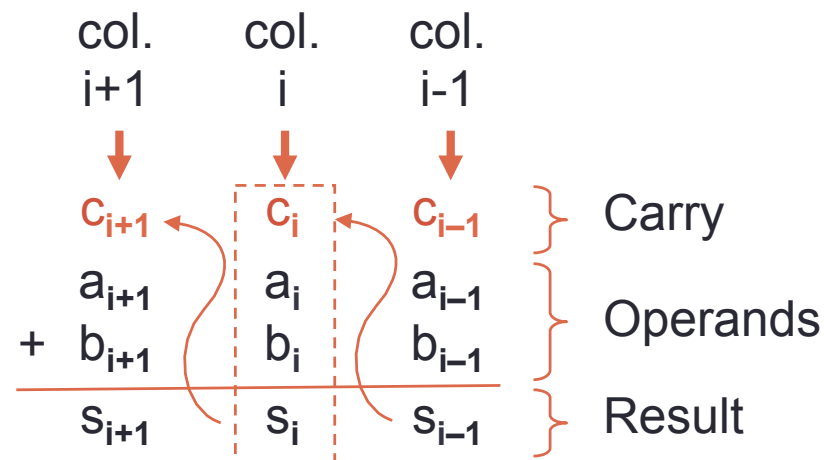
Branch to overflow handler

# Semantics of comparison instructions

- `slt rd,rs1,rs2` (*set on less than*)  
`slti rd,rs,imm`
  - Take two signed 2'sC operands and compare them for **strict lowness** ( $rs1 < rs2$  or  $rs < imm$ )
  - If the condition holds, then  $rd=1$ ; else  $rd=0$
  - Never cause an exception
- `sltu rd,rs1,rs2`  
`sltiu rd,rs,imm`
  - Exactly the same behaviour, but operands are regarded as unsigned NBC values

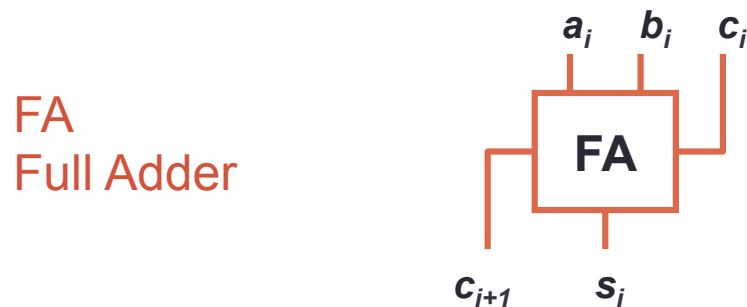
# Addition operators

- Serial addition emulates the *human* procedure for addition
- Addition proceeds bit by bit, from LSB to MSB
- For each column  $i$ , we add  $a_i$ ,  $b_i$ , and  $c_i$ , the carry input coming from column  $i-1$ ; we obtain  $s_i$  and  $c_{i+1}$
- Global input carry is  $c_0$

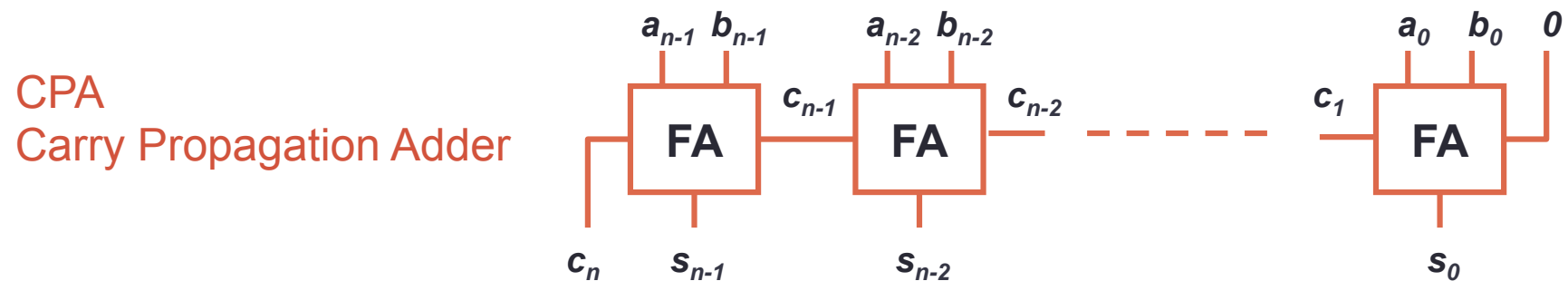


# Addition operators

We will first design the Full Adder (FA) circuit for each stage



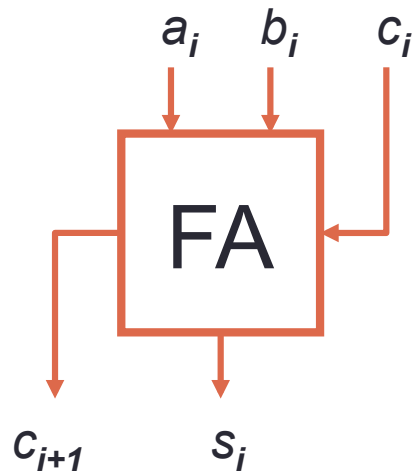
We will then combine n FAs for building an n-bit serial adder



We will improve performance by means of the *Carry Select Adder*

# Full adder – design

- Calculates one stage of a serial addition
- Takes three one-bit inputs; produces two outputs



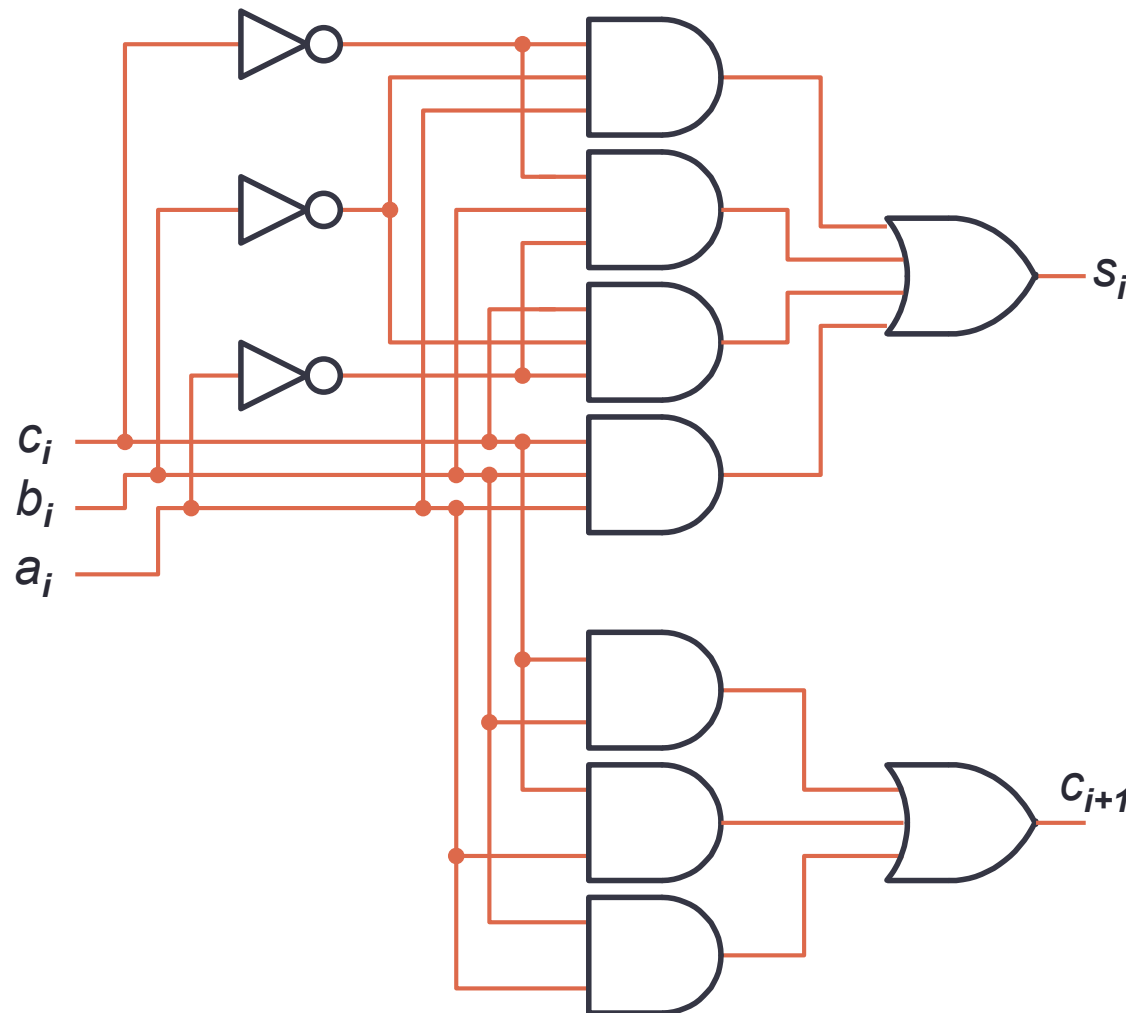
$a_i$	$b_i$	$c_i$	$c_{i+1}$	$s_i$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$s_i = \overline{a_i} \cdot \overline{b_i} \cdot c_i + \overline{a_i} \cdot b_i \cdot \overline{c_i} + a_i \cdot \overline{b_i} \cdot \overline{c_i} + a_i \cdot b_i \cdot c_i$$

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

# Full adder – Implementation

- Derived from logical functions  $s_i$  and  $c_{i+1}$



# Full adder – performance

- Response time

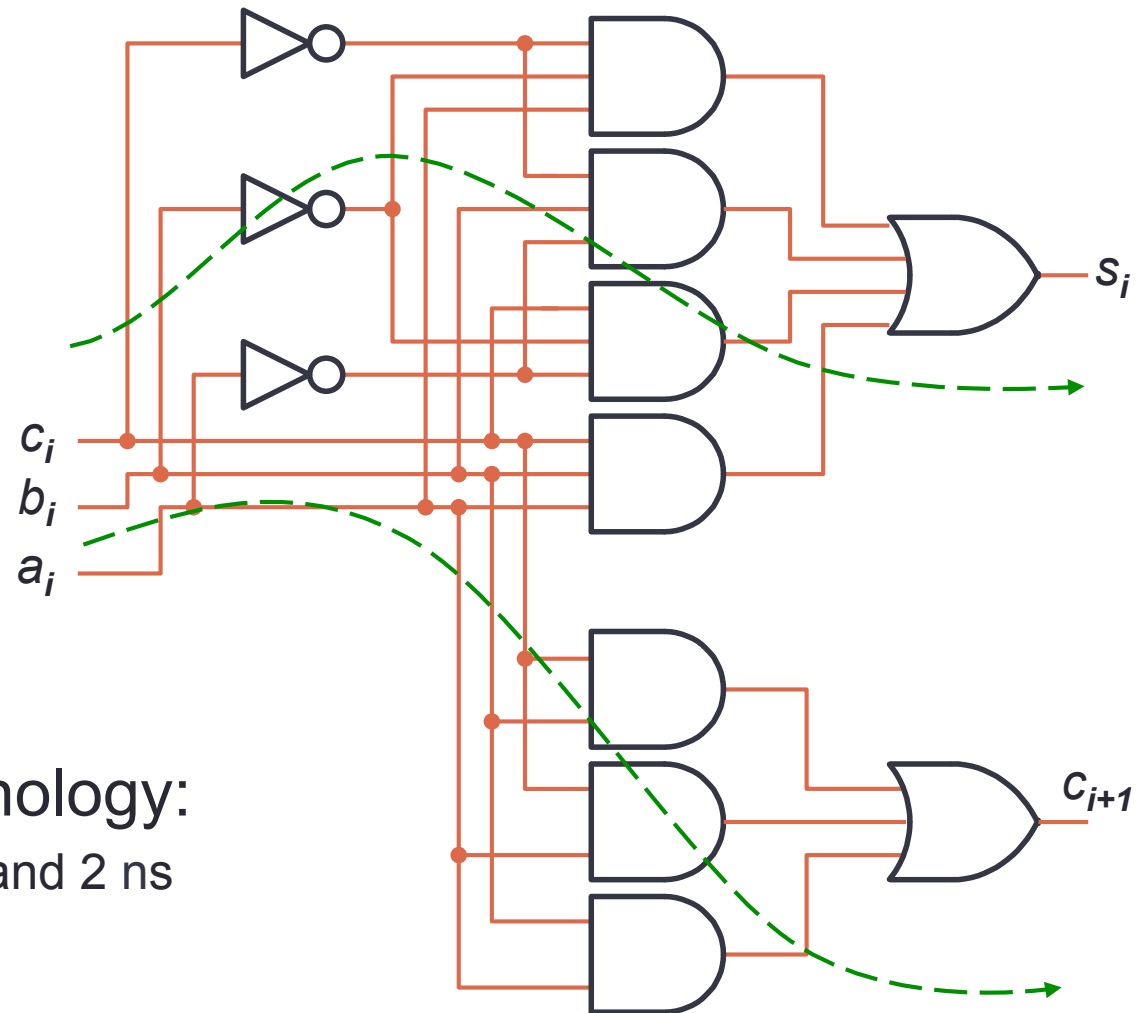
$$t_S = t_{NOT} + t_{AND} + t_{OR}$$

$$t_C = t_{AND} + t_{OR}$$

- Complexity: 12 gates

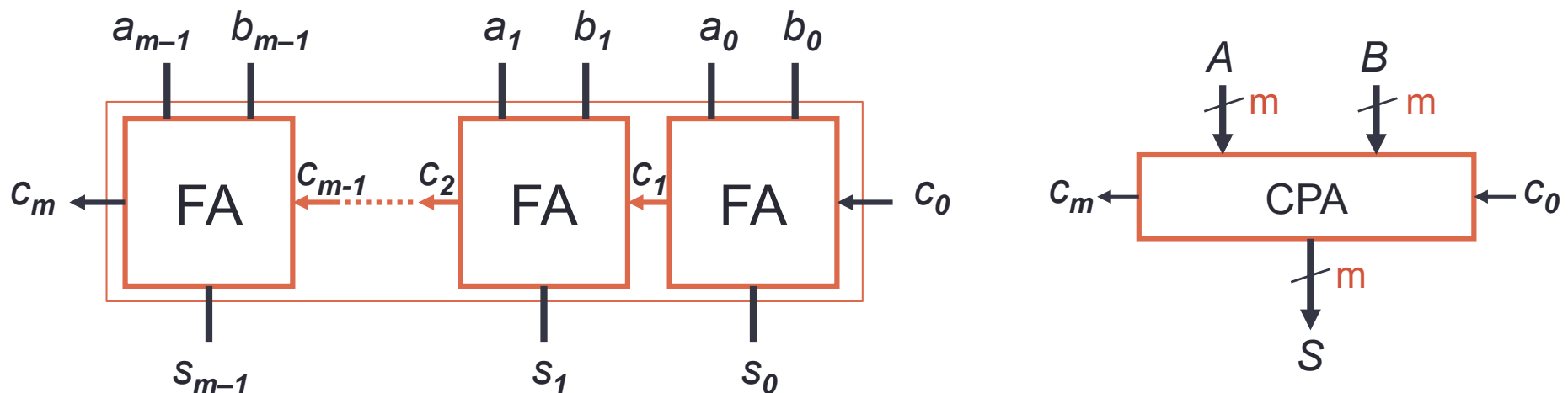
- With 0.5  $\mu\text{m}$  CMOS technology:

- Response time: between 1 and 2 ns
- Area: 1000  $\mu\text{m}^2$



# Carry propagation adder (CPA)

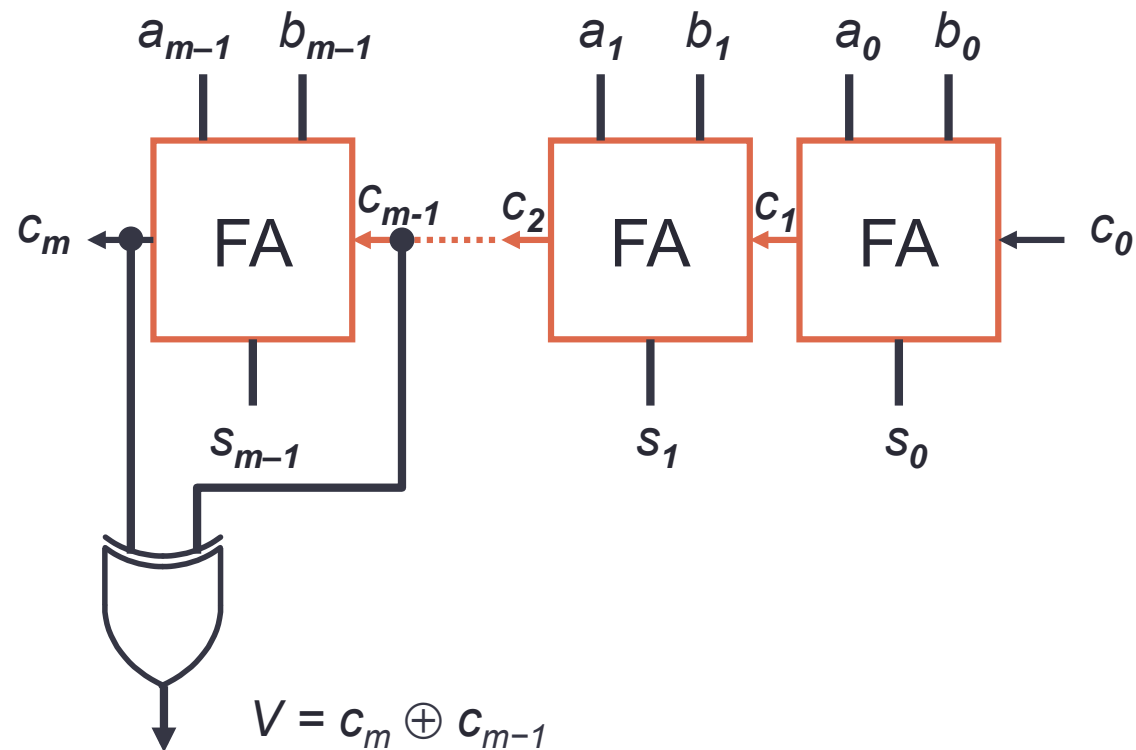
- For  $m$ -bit operands, cascade  $m$  one-bit FAs
  - Connect  $i$ -th output carry to carry input  $i+1$
  - The CPA has one global input carry and one global output carry
    - This enables further cascading of CPAs





# Carry propagation adder (CPA)

- Overflow detection (signed addition)



## CPA – Complexity

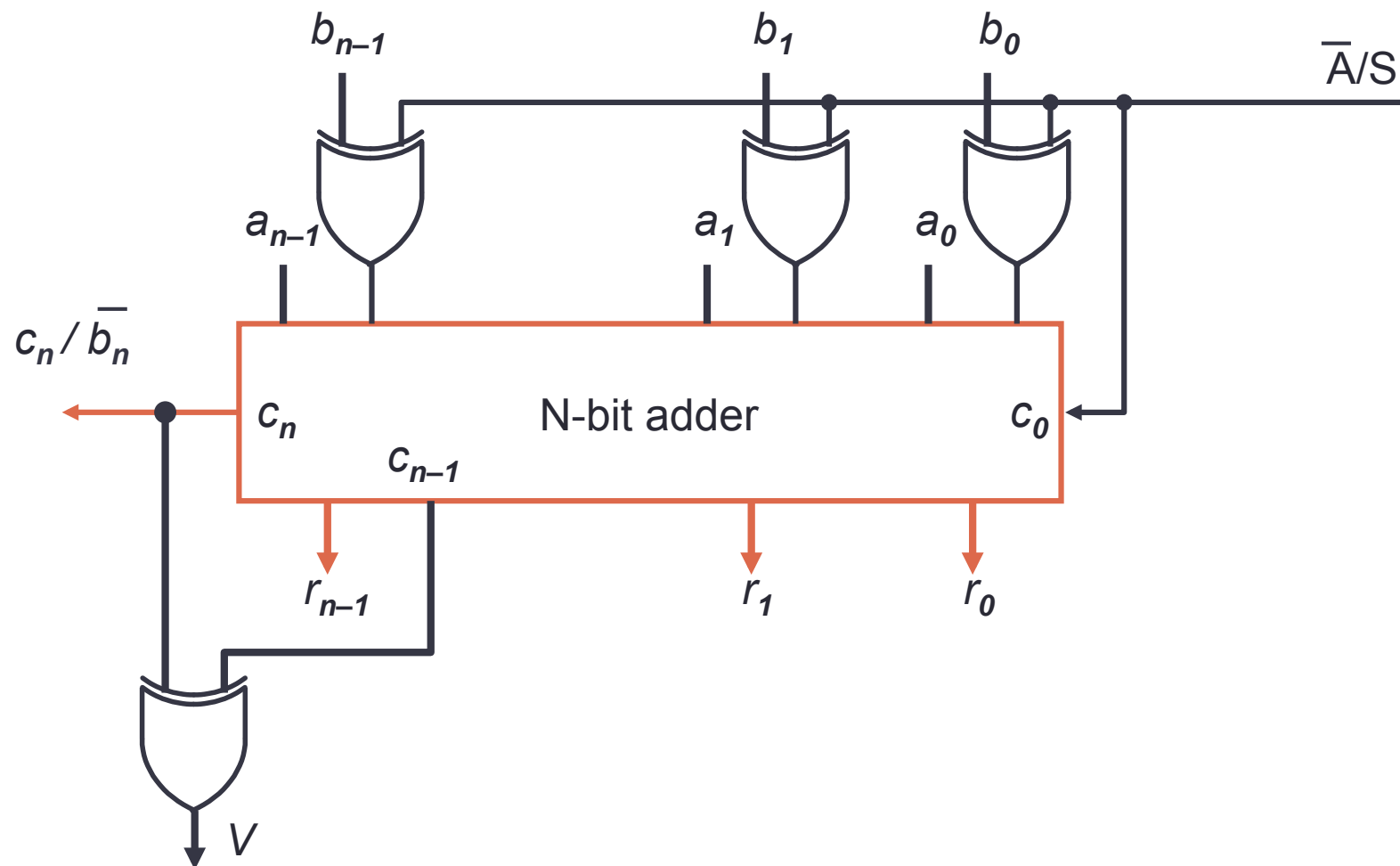
- Temporal complexity of this design is tied to the CPA size

$$t(n) = (n-1) \cdot t_c + \max\{t_c, t_s\}$$

- Without considering overflow detection. How would you include overflow detection in the above expression?
- Asymptotically,  $t(n) = O(n)$
- Spatial cost is also  $O(n)$
- Serial addition with CPA performs poorly for typical processor nowadays ( $n=32$ ,  $n=64$  bits)

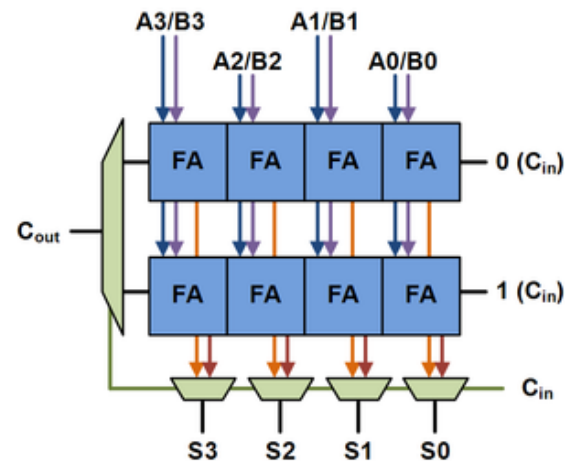
# Subtraction operator

- Classical design of an adder/subtractor



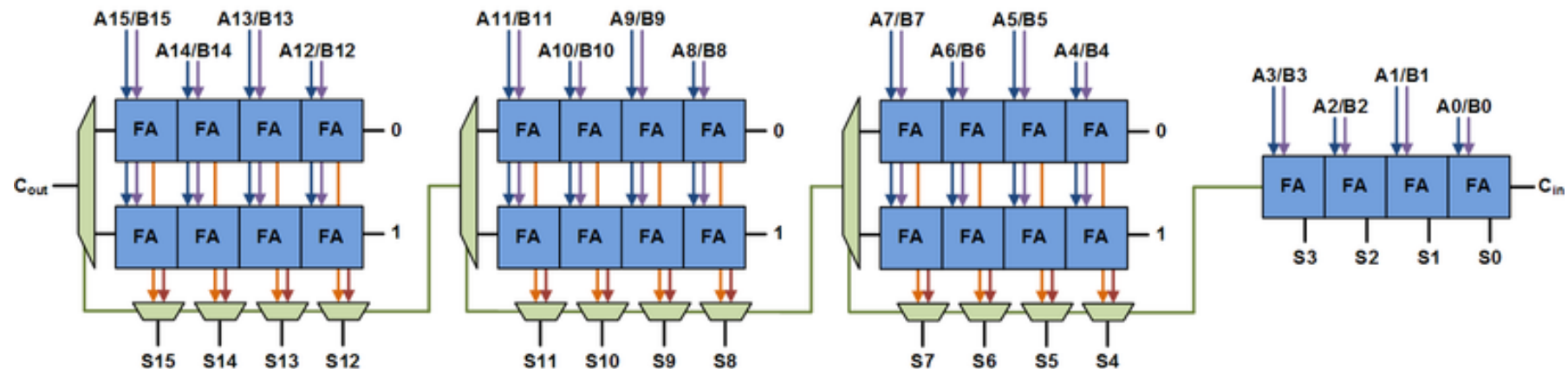
# Faster adders: CSA (Carry Select Adder)

- Accelerates the addition by replicating hardware



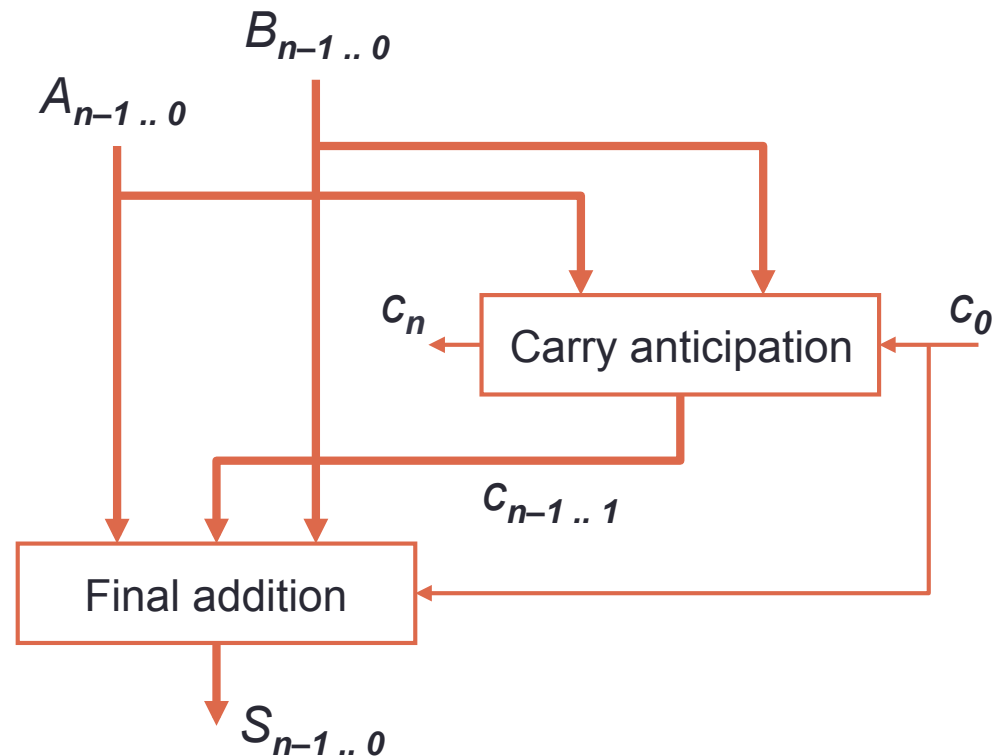
Building block

16-bit CSA



# Fast adders: CLA (Carry Lookahead Adder)

- Calculates all carry outputs beforehand ( $3 \times$  gate delay) and then adds them to the operands with parallel, decoupled FAs
- The cost is  $O(\log(n))$



# 3. Integer multiplication and division

- Fundamentals
- Multiplication and division in MIPS R2000
- Shift operators
- Unsigned multiplication operators
- Signed multiplication operators

# Arithmetic effects of shifts on integers

- An  $n$ -bit **left shift** is equivalent to **multiplication** by  $2^n$ 
  - The least-significant  $n$  bits (bits to the right) are filled with zeros
  - This property holds both for signed and unsigned integers

$$\begin{array}{cccccc} 0 & 0 & 0 & 1 & 1 & 0 \\ & \swarrow & \searrow & \swarrow & \searrow & \\ & & \ll 2 & & & \\ & \swarrow & \searrow & \swarrow & \searrow & \\ 0 & 1 & 1 & 0 & 0 & 0 \end{array} \quad \begin{array}{l} 6 \\ \times 2^2 \\ 24 \end{array}$$

$$\begin{array}{cccccc} 1 & 1 & 1 & 0 & 1 & 0 \\ & \swarrow & \searrow & \swarrow & \searrow & \\ & & \ll 2 & & & \\ & \swarrow & \searrow & \swarrow & \searrow & \\ 1 & 0 & 1 & 0 & 0 & 0 \end{array} \quad \begin{array}{l} -6 \\ \times 2^2 \\ -24 \end{array}$$


```
int a = -12;
int b = a << 3;
System.out.println(a + " * 8 = " + b);
```

Java

`-12 * 8 = -96`

# Arithmetic effects of shifts on integers

- Conversely, n-bit **right shift** is equivalent to **division** by  $2^n$ 
  - Unsigned**: n leftmost bits are filled with zeros – **logical shift**
  - Signed**: n leftmost bits replicate the sign bit – **arithmetic shift**

Unsigned, logical shift		Signed, arithmetic shift	
0 1 1 0 0 1	25	0 1 1 0 0 1	+25
	$\div 2^2$		$\div 2^2$
0 0 0 1 1 0	6	0 0 0 1 1 0	6
1 0 1 0 0 0	40	1 0 1 0 0 0	-24
	$\div 2^2$		$\div 2^2$
0 0 1 0 1 0	10	1 1 1 0 1 0	-6



# Compiling multiplications

- Due to higher temporal cost of multiplication, compilers tend to avoid `mult` instructions

## Multiplication by **constants**

```
int a,b,c,d;  
a = a*2;    // 2=21  
b = b*8;    // 8=23  
c = c*1024; // 1024=210  
d = d * 5;  // 5=22+1
```



```
lw $s0,a  
lw $s1,b  
lw $s2,c  
lw $s3,d  
add $s0,$s0,$s0    # a = a*2  
sll $s1,$s1,3      # b = b*8  
sll $s2,$s2,10     # c = c*1024  
sll $t0,$s3,2  
add $s3,$t0,$s3    # d = d * 5;  
sw $s0,a  
sw $s1,b  
sw $s2,c  
sw $s3,d
```

# Anatomy of unsigned multiplication

- In general,  $2 \times n$  bits are required for the result of multiplying two  $n$ -bit operands
- The usual *human* procedure uses shifts and addition

				1	1	0	1	(13 <sub>10</sub> )
			×	1	0	0	1	(9 <sub>10</sub> )
				1	1	0	1	
				0	0	0	0	
		0	0	0	0	0	0	
	1	1	0	1	0	0	0	
0	1	1	1	0	1	0	1	
2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	
Weights								

## Notation

$M$  = Multiplicand     $m_i$  =  $i$ -th bit  
 $Q$  = Multiplier     $q_i$  =  $i$ -th bit  
 $P$  = Product     $p_i$  =  $i$ -th bit

$$P = M \times Q = \sum_{i=0}^{n-1} Mq_i 2^i$$

$$(117_{10}) = 1101_2 \times (1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)$$

# Signed multiplication

- The unsigned procedure doesn't work...
- We need to sign-extend partial products and consider that the weight of the sign bit of the multiplier is  $-2^{n-1}$  (not  $2^{n-1}$ )
  - Later in this unit, we will study alternative homogeneous encodings for both signed and unsigned integers

1	1	1	1	1	1	0	1	$(-3_{10})$
			×	1	0	0	1	$(-7_{10})$
1	1	1	1	1	1	0	1	
			0	0	0	0	0	
		0	0	0	0	0	0	
0	0	0	1	1	0	0	0	
0	0	0	1	0	1	0	1	
2 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	

$$P = M \times Q = \sum_{i=0}^{n-2} Mq_i 2^i - Mq_{n-1} 2^{n-1}$$

$$(+21_{10}) = 11111101_2 \times (-1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)$$

# Integer division

- Division of a dividend by a divisor produces two results: quotient and a remainder
- The *human* procedure uses shifts and subtraction
  - When no fit (✗):  $\text{quotient}_i \leftarrow 0$
  - When fit (✓): subtract divisor and set  $\text{quotient}_i \leftarrow 1$
- Unsigned example

	1 1 0 1	0 1 0 1	
✗ -	0 1 0 1	0 0 1 0	
✗ -	0 1 0 1		
✓ -	0 1 0 1		
	-----		
	0 0 1 1		
✗ -	0 1 0 1		
	0 0 1 1		

Final quotient

Final remainder

# Multiplication & division in high level

- Remainder or modulus


```
int x,y,z,t;
x = 13;
y = 5;
z = x/y;
t = x%y;
System.out.println(x + " = " + y + "*" + z + " + " + t);
```

Java

13 = 5\*2 + 3

- Division by zero

```
int x,y,z; Java,C
x = 0;
y = 1;
z = y/x;
```



Exception in thread "main"  
java.lang.ArithmeticException: / by zero at ...

# Multiplication & division in MIPS

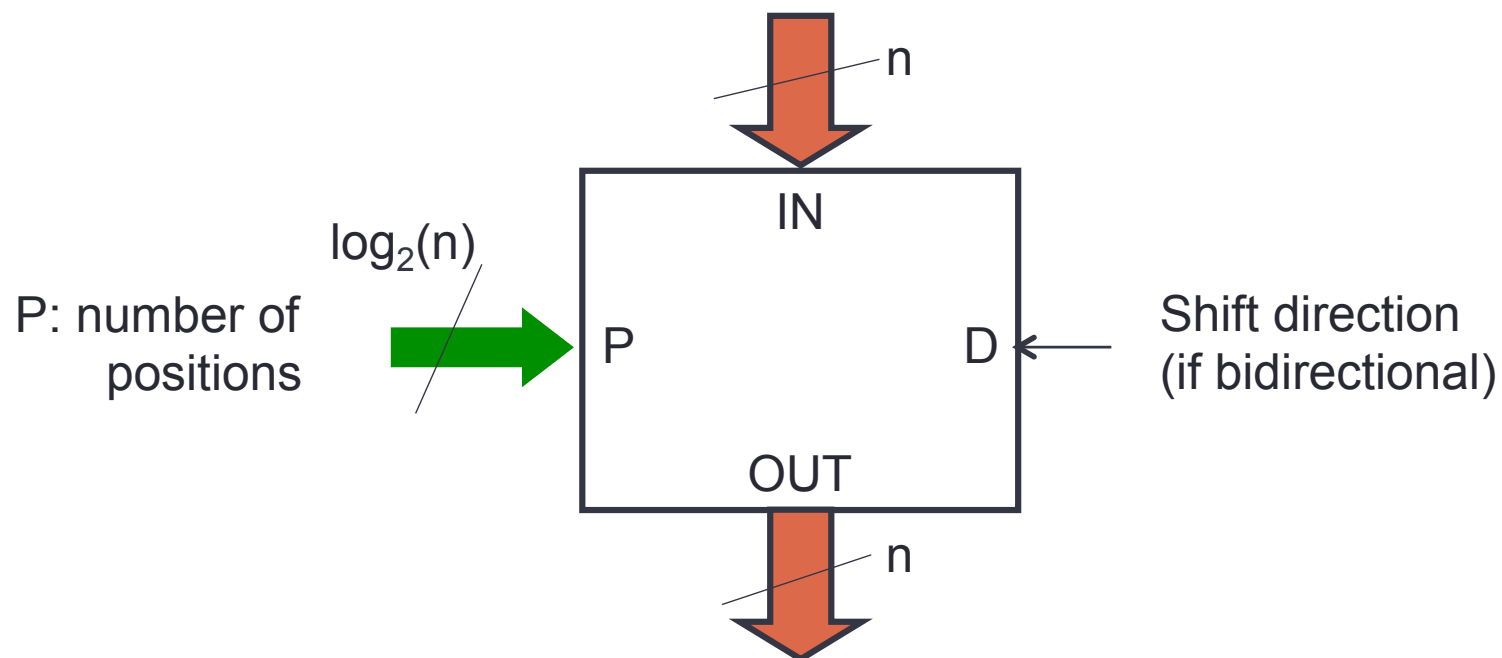
- Shift instructions

- They have the form *shift rd,rs,long* where *long* can be an immediate constant or a variable value stored in a register
- Maximum shift = 31 positions – only the 5 LSbs count

Type of shift	reg-imm form	reg-reg form
Left	<code>sll rt,rs,imm</code>	<code>sllv rd,rs,rt</code>
Right (logical)	<code>srl rt,rs,imm</code>	<code>srlv rd,rs,rt</code>
Right (arithmetic)	<code>sra rt,rs,imm</code>	<code>srav rd,rs,rt</code>

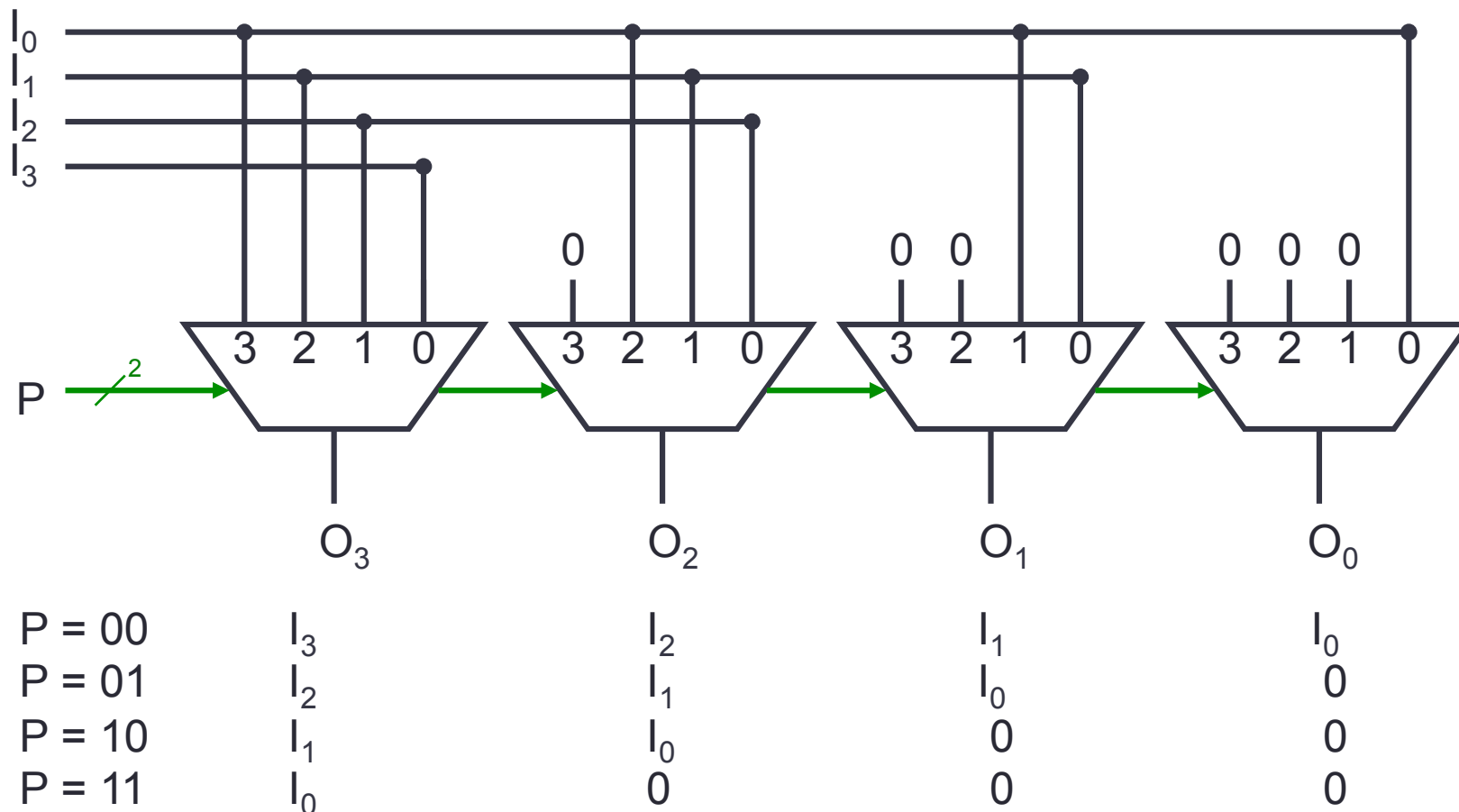
# Shift operators: the Barrel Shifter

- A barrel shifter implements variable shifts
- Inputs:
  - Operand to be shifted ( $n$  bits)
  - Number of positions to shift ( $\log_2(n)$  bits)
  - Optionally, direction of shift (1 bit)



# A barrel shifter implementation

- This barrel shifter implements shift left logical (**sll**) of one 4-bit operand





# Multiplication & division in MIPS

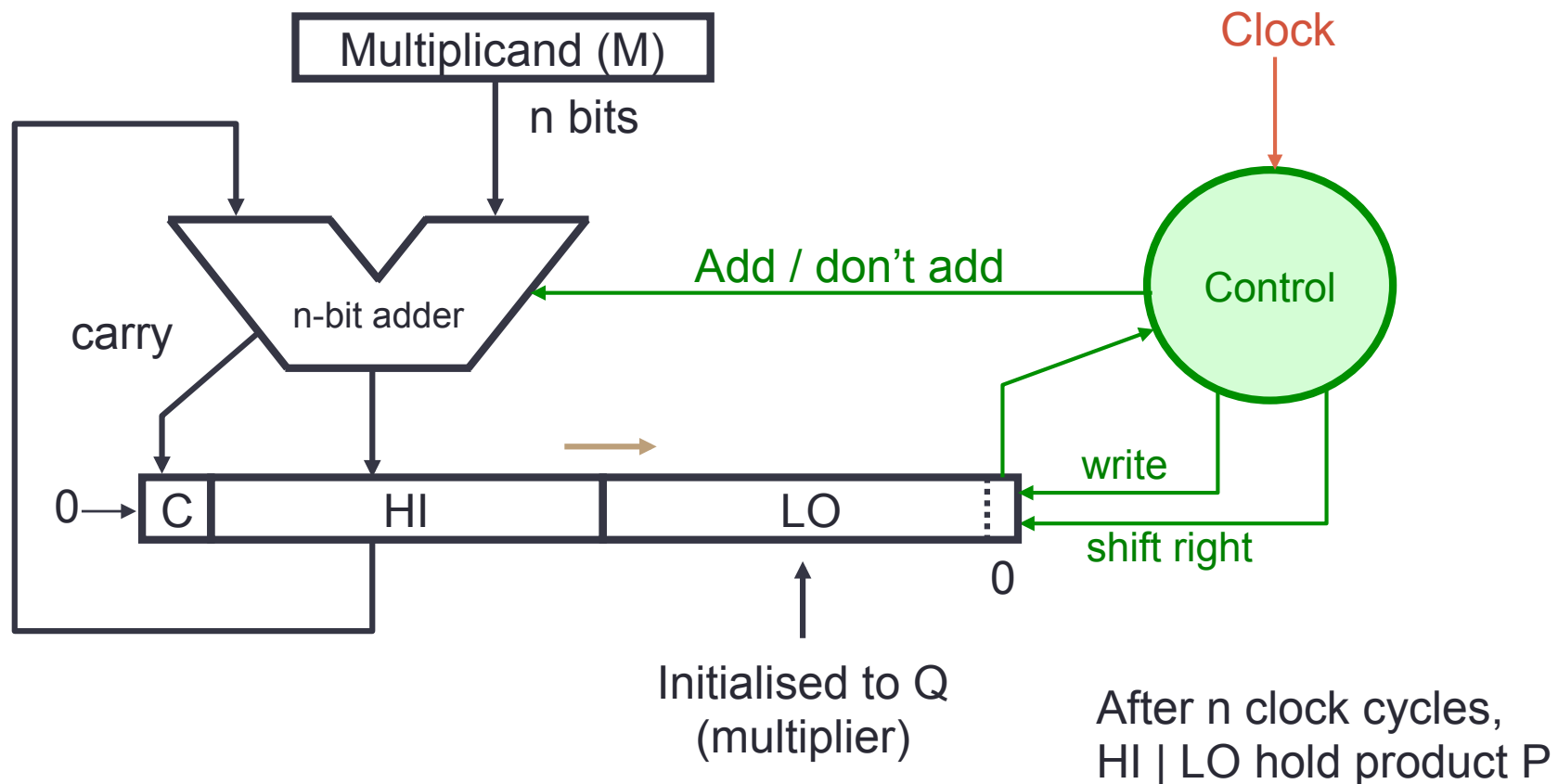
- Multiplication and division instructions
  - Two special 32-bit registers store the 64-bit result: HI and LO
  - Operations
    - `mult $2, $3:` HI-LO  $\leftarrow$   $\$2 * \$3$ ; Signed operands
    - `multu $2, $3:` HI-LO  $\leftarrow$   $\$2 * \$3$ ; Unsigned operands
    - `div $2, $3:` LO  $\leftarrow$   $\$2 / \$3$ ; HI  $\leftarrow$   $\$2 \bmod \$3$ ; Signed
    - `divu $2, $3:` LO  $\leftarrow$   $\$2 / \$3$ ; HI  $\leftarrow$   $\$2 \bmod \$3$ ; Unsigned
  - Transferring results
    - `mfhi $2:`  $\$2 \leftarrow$  HI
    - `mflo $2:`  $\$2 \leftarrow$  LO
  - There are pseudoinstructions that store the result in a general-purpose register and multiply by constants
  - None of these instructions check for overflow or division by zero: those checks need be done by software

# Unsigned multiplication operators

- We'll focus on sequential, synchronous operators
  - These operators perform the specified operation within a given number of clock cycles
  - The clock cycle must be large enough for the circuits to operate
  - For an operator of  $n$  cycles of  $t$  seconds:
    - The circuit delay is  $T = n \times t$
    - The productivity is  $P = f / n$ , where  $f = 1 / t$
- Notation
  - $M$  = multiplicand;  $m_i = i$ -th bit of multiplicand
  - $Q$  = multiplier;  $q_i = i$ -th bit of multiplier
  - $P$  = product;  $p_i = i$ -th bit of product
  - $n$  = size of  $M$  and  $Q$  (in bits, from 0 to  $n-1$ )

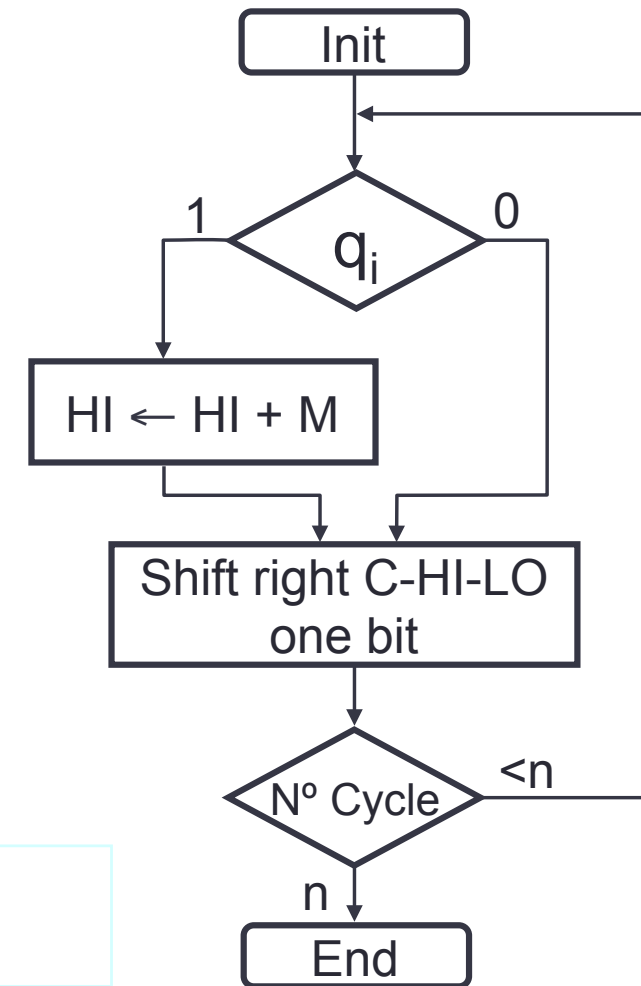
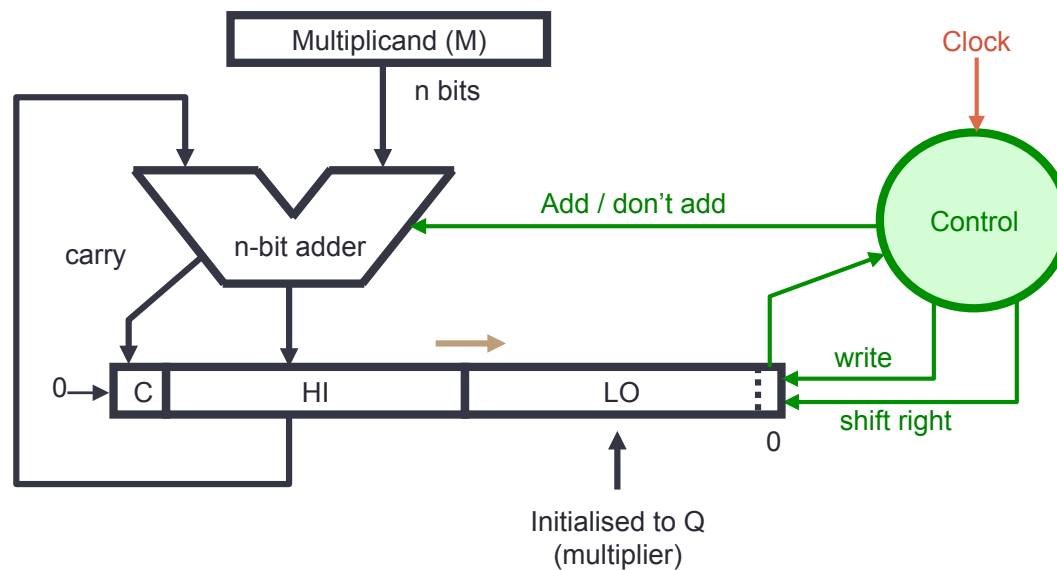
# Unsigned multiplication operator

- Operator for the shift-add algorithm
  - M and Q of n bits; P of 2n bits



# Unsigned multiplication operator

- Operator's algorithm



- The algorithm requires  $n$  cycles
- Up to one add and one shift per cycle

# Example

- $n = 4$ ;  $M = 1011_2$ ;  $Q = 0101_2$ ; (Base 10:  $11 \times 5 = 55$ )

Cycle	Action	C-HI-LO
0	Initial values	0 0000 <u>0101</u>
1	$HI \leftarrow HI + M$	0 1011 0101
	Shift right C-HI-LO 1 bit	0 0101 <u>1010</u>
2	Don't add	0 0101 <u>1010</u>
	Shift right C-HI-LO 1 bit	0 0010 <u>1101</u>
3	$HI \leftarrow HI + M$	0 1101 <u>1101</u>
	Shift right C-HI-LO 1 bit	0 0110 <u>1110</u>
4	Don't add	0 0110 <u>1110</u>
	Shift right C-HI-LO 1 bit	0 <u>0011</u> 0111

# Exercise

- $n = 4$ ;  $M = 1101_2$ ;  $Q = 1011_2$ ; (Base 10:  $13 \times 11 = 143$ )

Cycle	Action	C-HI-LO
0	Initial values	0 0000 <u>1011</u>
1		
2		
3		
4		

Solution: 1000 1111

# Signed multiplication

- Dealing with sign separately (option 1)
  - One option to reuse the previous multiplier is to consider sign separately. Assume  $\text{Sign}(X)$  is the sign bit of  $X$ :

```
Prod_Sign ← Sign(M) XOR Sign(Q);  
if M < 0 then M ← -M; end if;  
if Q < 0 then Q ← -Q; end if;  
P ← M × Q;  
if Prod_Sign = 1 then P ← -P; end if;
```

- Algorithm drawbacks
  - Requires additional hardware for signed numbers
  - Other methods enable symmetric algorithms for signed and unsigned numbers (eg., Booth's algorithm – later)

# Signed multiplication

- Sign-extended shift-add algorithm (option 2)
  - The shift/add algorithm works with signed numbers only when  $Q \geq 0$
  - We only need to sign-extend partial products
  - Example with  $n = 4$ ;  $M = -3$ ; Both represented in 2'sC

$$\begin{array}{r}
 \phantom{0000} 1\ 1\ 0\ 1\ (-3_{10}) \\
 \times \phantom{000} 0\ 1\ 1\ 0\ (6_{10}) \\
 \hline
 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
 1\ 1\ 1\ 1\ 1\ 0\ 1\phantom{0} \\
 1\ 1\ 1\ 1\ 0\ 1\phantom{00} \\
 0\ 0\ 0\ 0\ 0\phantom{000} \\
 \hline
 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ (-18_{10})
 \end{array}$$

To adapt to the general case, signs of M and Q can be pre-processed beforehand:

```

if Q < 0 then
    Q ← -Q;
    M ← -M;
end if;
P ← M × Q;
  
```

Although simpler than previous algorithm, it also requires separate sign processing and to complement M and Q when  $Q < 0$



# Booth's algorithm

- The multiplier is recoded as a sum of **positive and negative powers** of the base
  - Booth's code uses digits 0, 1 and -1
  - Example
    - $30_{10}$  can be represented as  $32 - 2$
    - $30_{10} = 0011110_2 = 0 + 1\ 0\ 0\ 0 - 1\ 0_{\text{Booth}} = (+1) \times 2^5 + (-1) \times 2^1$

								0	1	0	1	1	0	1	(45 <sub>10</sub> )	
								×	0	+1	0	0	0	-1	0	(30 <sub>Booth</sub> )
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		
1	1	1	1	1	1	1	1	0	1	0	0	1	1		←	Copy 2'sC of M (subtract M)
0	0	0	0	0	0	0	0	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0	0	0					
0	0	0	0	0	0	0	0	0	0	0						
0	0	0	1	0	1	1	0	1							←	Copy M (add M)
0	0	0	0	0	0	0	0									
0	0	0	1	0	1	0	1	0	0	0	1	1	0			(1350 <sub>10</sub> )

# Booth's algorithm

- Works both with signed and unsigned integers:
  - Unsigned: assume an implicit sign bit = 0 for M
  - Signed: assume an implicit one-bit sign extension for M
  - Example: signed and unsigned product  $1101_2 \times 0+10-1_{\text{Booth}}$

## Unsigned

Implicit positive sign  $\rightarrow$

0	1	1	0	1					
$\times$	0	+1	0	-1					
1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0			
0	0	1	1	0	1				
0	0	0	0	0					
0	0	1	0	0	1	1	1		

(13<sub>10</sub>)  
(3<sub>Booth</sub>)  
(39<sub>10</sub>)

## Signed

Implicit sign extension  $\rightarrow$

1	1	1	0	1					
$\times$	0	+1	0	-1					
0	0	0	0	0	0	1	1		
0	0	0	0	0	0	0			
1	1	1	1	0	1				
0	0	0	0	0					
1	1	1	1	0	1	1	1		

(-3<sub>10</sub>)  
(3<sub>Booth</sub>)  
(-9<sub>10</sub>)

# Booth's recoding

- Booth's recoding of the multiplier:
  - Always proceed considering two correlative digits, right to left
  - Assume an implicit 0 bit to the right of the LSB
  - Apply the following conversion:

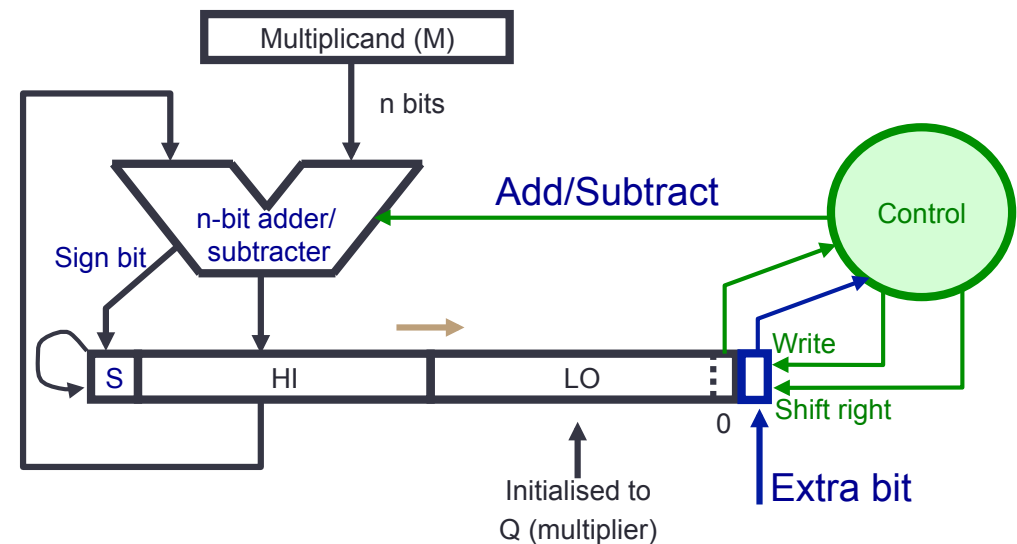
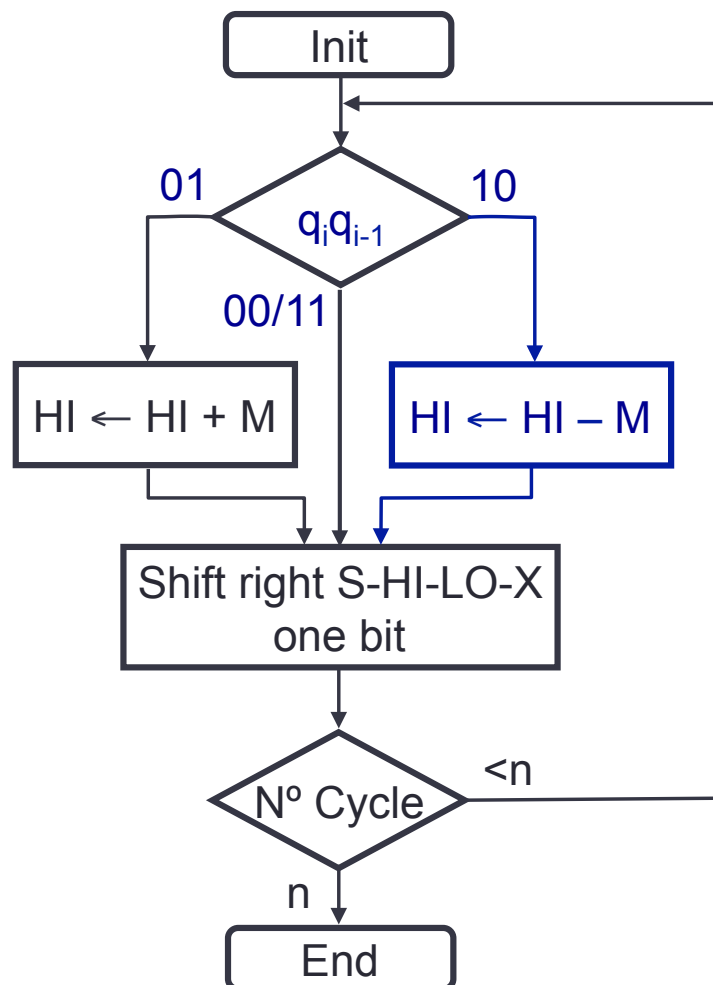
$q_i$	$q_{i-1}$	Booth digit
0	0	0
0	1	+1
1	0	-1
1	1	0

Note the rule:  
Booth digit =  $q_{i-1} - q_i$

- Example: Recode 1110 0111 0011 ( $-397_{10}$ )
  - Solution: 0 0 -1 0 +1 0 0 -1 0 +1 0 -1 =  
 $-1 \times 2^0 + 1 \times 2^2 - 1 \times 2^4 + 1 \times 2^7 - 1 \times 2^9 =$   
 $-1 + 4 - 16 + 128 - 512 = -397$

# Booth-based operator and algorithm

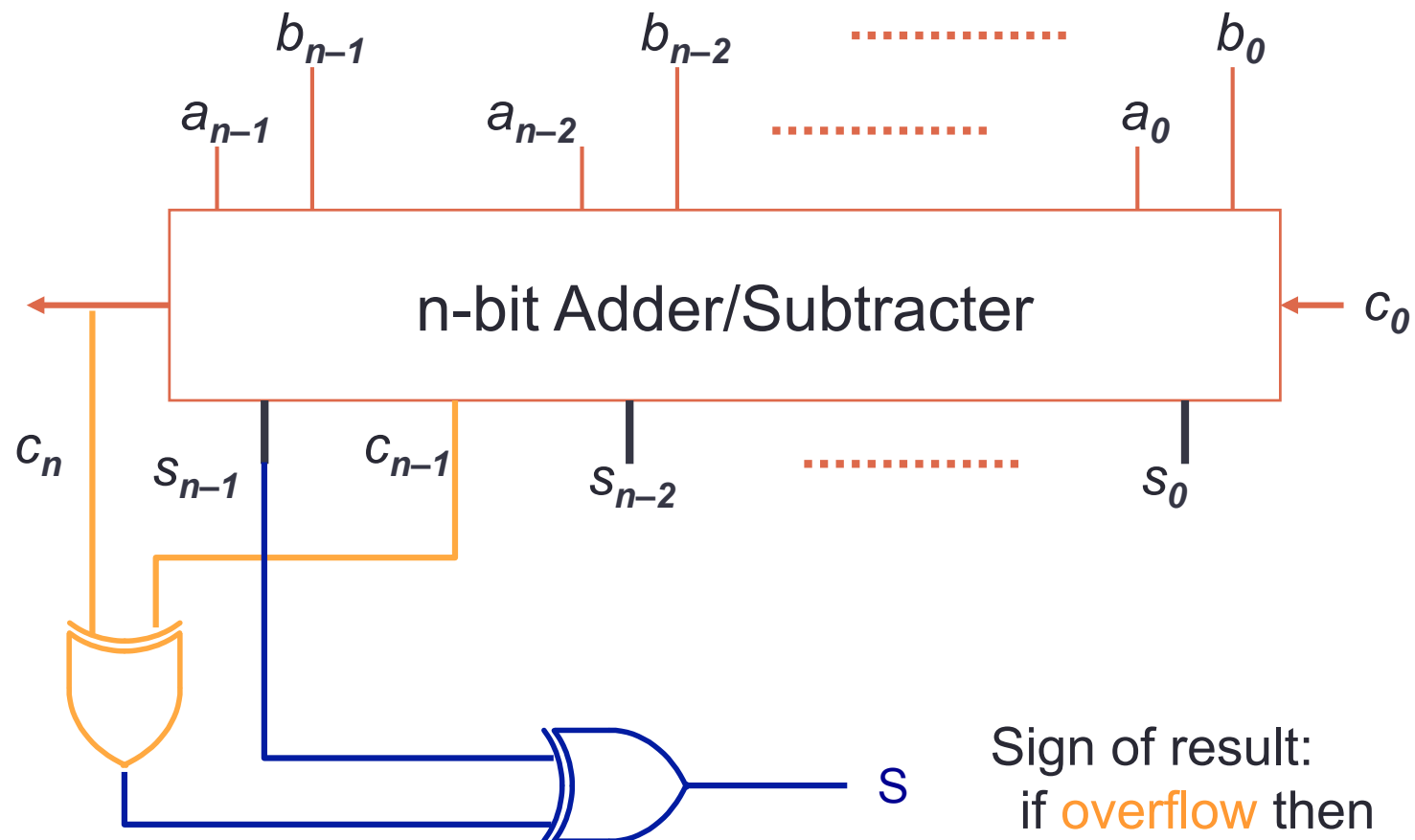
- Modifications to the shift-add operator and algorithm



- Requires  $n$  cycles
- One add or sub plus one shift per cycle

# Booth operator

- Detail for obtaining the sign bit



Sign of result:  
if **overflow** then  
complement MSB

# Booth example

- Signed;  $n = 4$ ;  $M = 0010_2$ ;  $Q = 1001_2$ ; (Base 10:  $2 \times (-7) = -14$ )

Cycle	Action	S-HI-LO-X	Extra bit
0	Initial values	0 0000 1001 0	
1	Case 10: $HI \leftarrow HI - M$	1 1110 1001 0	
	Shift right S-HI-LO 1 bit	1 1111 0100 1	
2	Case 01: $HI \leftarrow HI + M$	0 0001 0100 1	
	Shift right S-HI-LO 1 bit	0 0000 1010 0	
3	Do nothing	0 0000 1010 0	
	Shift right S-HI-LO 1 bit	0 0000 0101 0	
4	Case 10: $HI \leftarrow HI - M$	1 1110 0101 0	
	Shift right S-HI-LO 1 bit	1 1111 0010 1	

# Exercise

- $n = 4$ ;  $M = 1101_2$ ;  $Q = 0110_2$ ; (Base 10:  $-3 \times 6 = -18$ )

Cycle	Action	S-HI-LO-X
0	Initial values	0 0000 <u>0110</u> 0
1		
2		
3		
4		

Solution: 1110 1110

## Further improvement: bit-pair recoding

- Bit-pair recoding halves the needed number of cycles
  - A single digit contains the information provided by two bits of Q
  - Digits used:  $-2$ ,  $-1$ ,  $0$ ,  $1$ , and  $2$

			Booth		Bit-pair	Action
$q_{i+1}$	$q_i$	$q_{i-1}$	$q'_{i+1}$	$q'_i$	$q''_i$	
0	0	0	0	0	0	Do nothing
0	0	1	0	1	1	Add M
0	1	0	1	-1	1	Add M
0	1	1	1	0	2	Add $2 \times M$
1	0	0	-1	0	-2	Subtract $2 \times M$
1	0	1	-1	1	-1	Subtract M
1	1	0	0	-1	-1	Subtract M
1	1	1	0	0	0	Do nothing

S-HI-LO is shifted right two bits each cycle



# Bit-pair recoding applied

- $n = 5$ ;  $M = 01101_2$ ;  $Q = 11010_2$ ; (Base 10:  $13 \times (-6) = -78$ )

Sign extension (need 2k bits)  $\rightarrow$  1 1 1 0 1 0 0  $\leftarrow$  Implicit extra bit

Booth

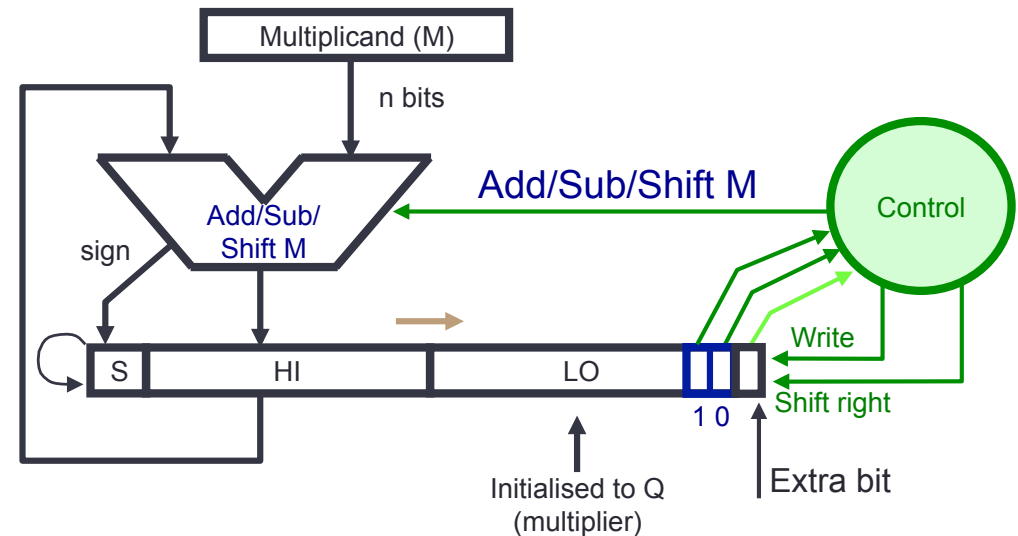
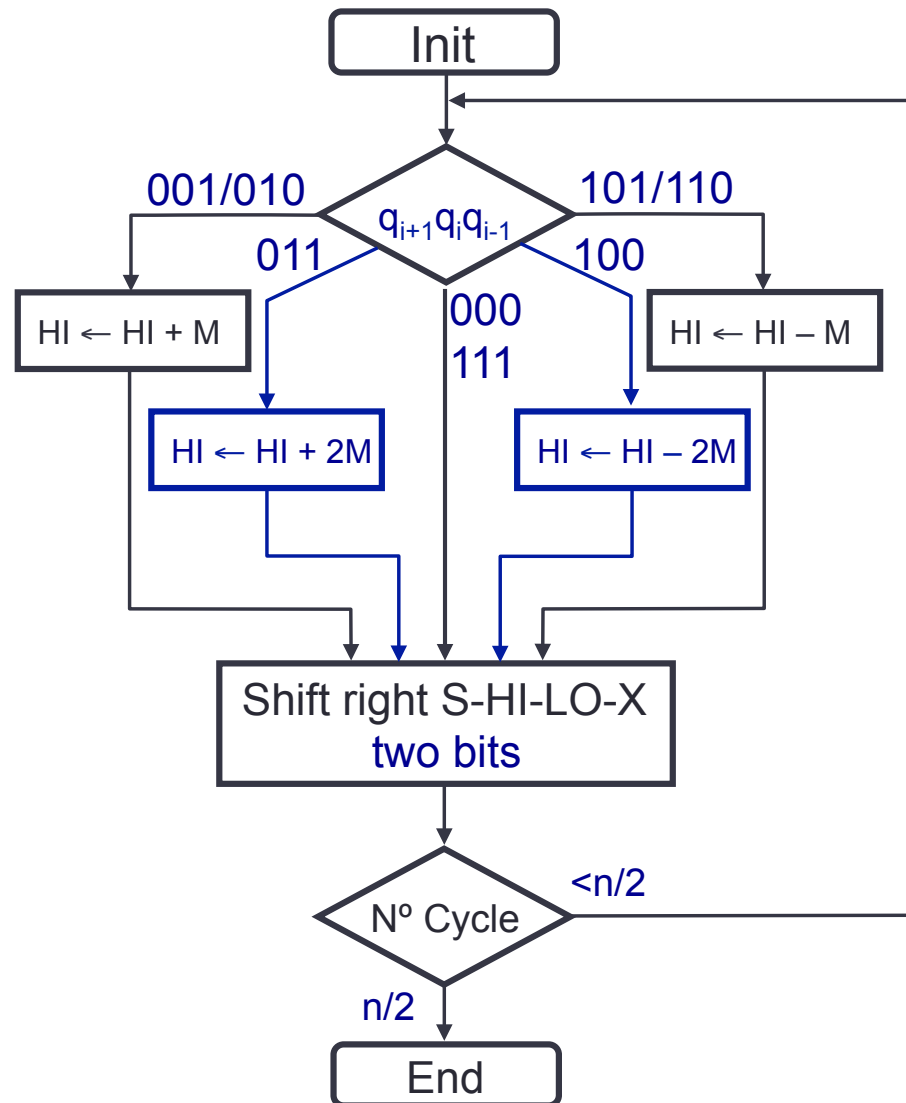
Bit-pair recoding  $\rightarrow$

	0	0	-1	1	-1	0
	└──┘		└──┘		└──┘	
	↓		↓		↓	
	0		-1		-2	

						0	1	1	0	1
					x	0		-1		-2
						<hr/>				
Subtract $2 \times M$	1	1	1	1	1	0	0	1	1	0
Subtract $M$	1	1	1	1	0	0	1	1		
Do nothing	0	0	0	0	0	0				
	<hr/>									
	1	1	1	0	1	1	0	0	1	0

( $-78_{10}$ )

# Bit-pair recoding operator



Both HI and LO have an even number of bits

- Requires  $n/2$  cycles
- On each cycle, up to one add or sub plus one 2-bit shift

# Bit-pair example

- $n = 6$ ;  $M = 001101_2$ ;  $Q = 111010_2$ ; (Base 10:  $13 \times (-6) = -78$ )

Cycle	Action	S-HI-LO-X
0	Initial values	0 000000 111010 0
1	Case 100: $HI \leftarrow HI - 2M$	1 100110 111010 0
	Shift right S-HI-LO 2 bits	1 111001 101110 1
2	Case 101: $HI \leftarrow HI - M$	1 101100 101110 1
	Shift right S-HI-LO 2 bits	1 111011 001011 1
3	Case 111: Do nothing	1 111011 001011 1
	Shift right S-HI-LO 2 bits	1 111110 110010 1

# Exercise

- $n = 6$ ;  $M = 101001_2$ ;  $Q = 001001_2$ ; (Base 10:  $(-23) \times 9 = -207$ )

Cycle	Action	S-HI-LO-X
0	Initial values	0 000000 001001 0
1		
2		
3		

Solution: 111100 110001

# Signed sequential multiplication: summary

- About the operator
  - Multiplication can be implemented with adds and shifts. The hardware requires only:
    - Shift registers
    - Adder or adder/subtractor
    - A sequential control circuit
- Booth's method
  - Enables uniform handling of signed and unsigned operations
- Bit-pair recoding
  - Enables faster multiplications by reducing the number of cycles
  - More elaborated recoding techniques enable faster operators
    - Higher radix recoding (eg., bit-quad recoding...)