Electronics Systems (938II)

Lecture 2.2

Building Blocks of Electronic Systems – Arithmetic operations

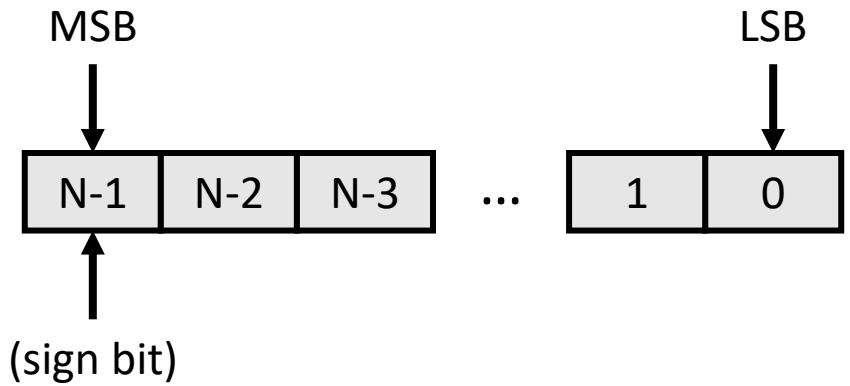
Combinational circuits for arithmetic operations

- Let's assume arithmetic operations on **integers** (**with sign**)
  - Addition
  - Subtraction
  - Multiplication
  - Division

- For this purpose, integers are represented using **two's complement** (**C2**)

Two's complement representation (of integers with sign)

- Assume N bits
  - Most significant bit (MSB), the leftmost one, is the sign bit



Two's complement representation (of integers with sign)

- Positive numbers
  - Natural binary representation
  - Sign bit = 0

- Example: 1 → 000…01

Two's complement representation (of integers with sign)

- Negative numbers
  1. Get absolute binary representation (i.e. corresponding positive value)
  2. Invert (or complement) all bits
  3. Add 1 (ignoring overflow)

Two's complement representation (of integers with sign)

- Negative numbers
  1. Get absolute binary representation (i.e. corresponding positive value)
  2. Invert (or complement) all bits
  3. Add 1 (ignoring overflow)

  - Example: –1

Two's complement representation (of integers with sign)

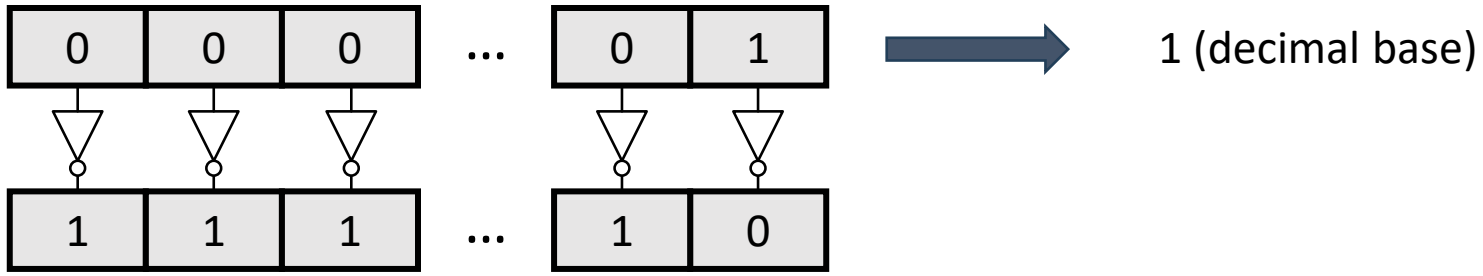
- Negative numbers
  1. **Get absolute binary representation (i.e. corresponding positive value)**
  2. Invert (or complement) all bits
  3. Add 1 (ignoring overflow)

- Example: –1

Two's complement representation (of integers with sign)

- Negative numbers
  1. Get absolute binary representation (i.e. corresponding positive value)
  2. **Invert (or complement) all bits**
  3. Add 1 (ignoring overflow)

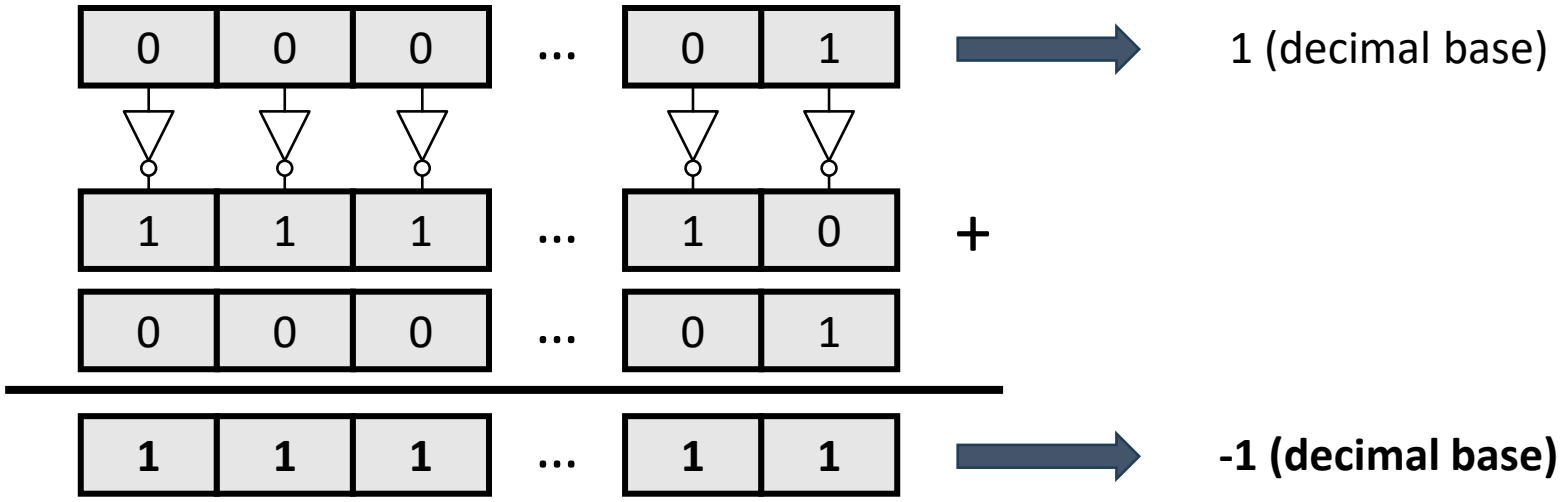
- Example: –1



Two's complement representation (of integers with sign)

- Negative numbers
  1. Get absolute binary representation (i.e. corresponding positive value)
  2. Invert (or complement) all bits
  3. **Add 1 (ignoring overflow)**

- Example: –1



Two's complement representation (of integers with sign)

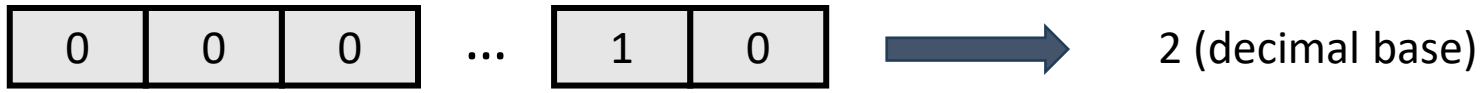
- Negative numbers
  1. Get absolute binary representation (i.e. corresponding positive value)
  2. Invert (or complement) all bits
  3. Add 1 (ignoring overflow)

  - Example: –2

Two's complement representation (of integers with sign)

- Negative numbers
  1. **Get absolute binary representation (i.e. corresponding positive value)**
  2. Invert (or complement) all bits
  3. Add 1 (ignoring overflow)

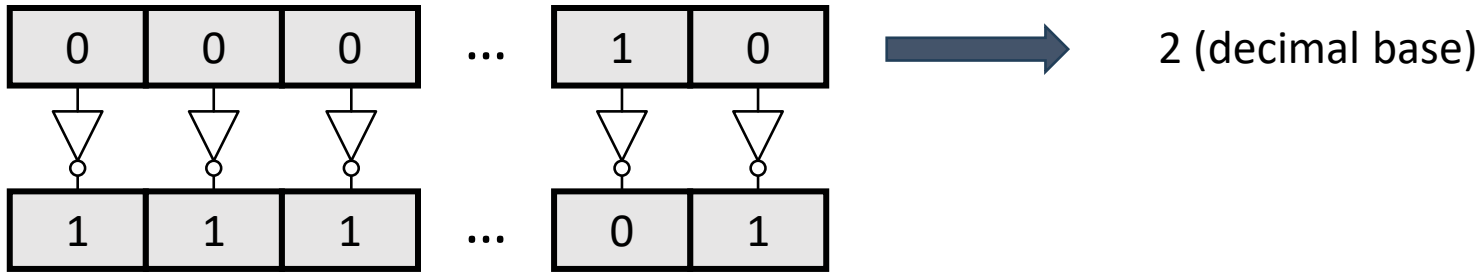
- Example: –2



Two's complement representation (of integers with sign)

- Negative numbers
  1. Get absolute binary representation (i.e. corresponding positive value)
  2. **Invert (or complement) all bits**
  3. Add 1 (ignoring overflow)

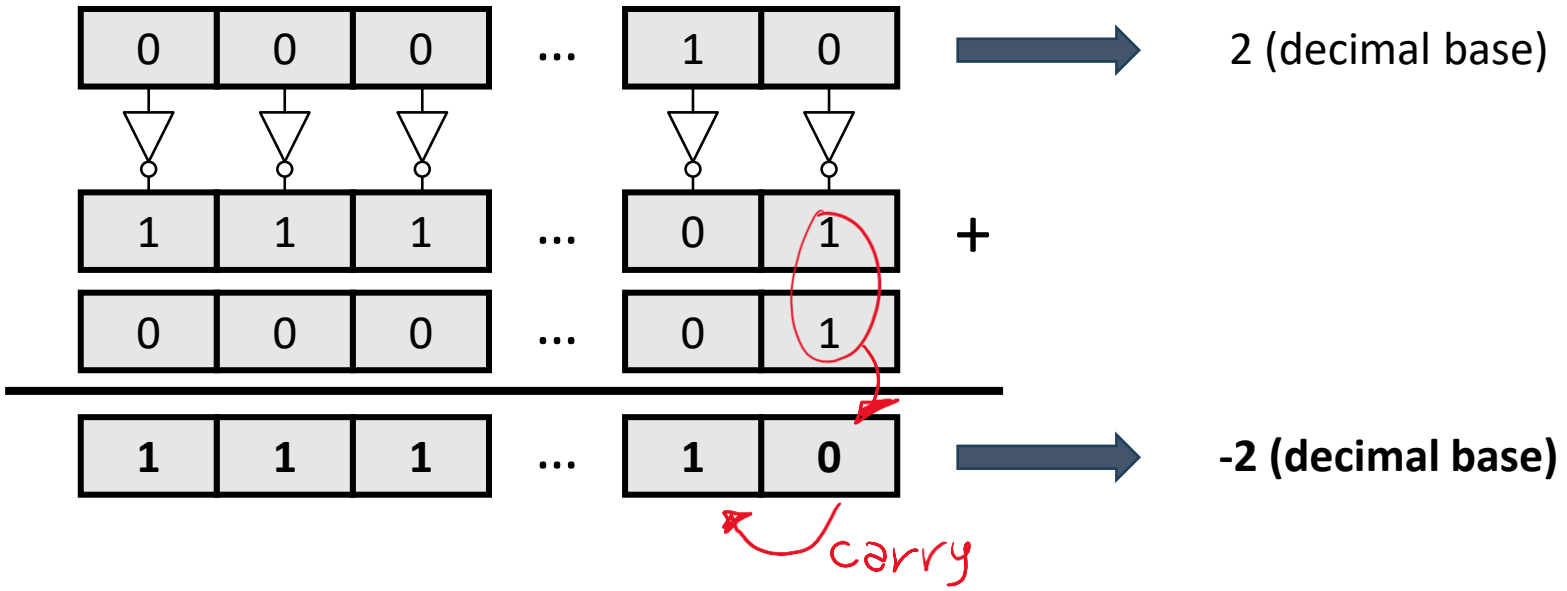
- Example: –2



Two's complement representation (of integers with sign)

- Negative numbers
  1. Get absolute binary representation (i.e. corresponding positive value)
  2. Invert (or complement) all bits
  3. **Add 1 (ignoring overflow)**

- Example: –2



Two's complement representation (of integers with sign)

- Range of signed integer numbers (in decimal base) = $[-2^{N-1} \div 2^{N-1} - 1]$

Two's complement representation (of integers with sign)

- Range of signed integer numbers (in decimal base) = $[-2^{N-1} \div 2^{N-1} - 1]$

C2 representation
011…11
…
000…01
000…00
111…11
111…01
…
100…00

Two's complement representation (of integers with sign)

- Range of signed integer numbers (in decimal base) = $[-2^{N-1} \div 2^{N-1} - 1]$
  - Example: $N = 3$
    - $2^{N-1} - 1 = 2^2 - 1 = 4 - 1 = 3$
    - $-2^{N-1} = -2^2 = -4$

Two's complement representation (of integers with sign)

- Range of signed integer numbers (in decimal base) = $[-2^{N-1} \div 2^{N-1} - 1]$
  - Example: $N = 3$
    - $2^{N-1} - 1 = 2^2 - 1 = 4 - 1 = 3$
    - $-2^{N-1} = -2^2 = -4$

C2 representation	Decimal base
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

Two's complement representation (of integers with sign)

- Why this?
  - Because it allows to perform arithmetic operations without particular (or dedicated) resources to handle the sign
    - Lower costs
    - Lower delays
    - **Subtraction** can be implemented as the **addition with the opposite number**; for examples …
      - 10 – 18 = 10 + (-18)
      - 12 – (-7) = 12 + 7
      - …
- Let's see some examples
  - Assuming N = 3 bits

Two's complement representation (of integers with sign)

1 + 1

(1)	0	0	1	+
(1)	0	0	1	
(2)	0	1	0	

1 + (-1) → 1 – 1

(1)	0	0	1	+
(-1)	1	1	1	
(0)	0	0	0	

1 + (-2) → 1 – 2

(1)	0	0	1	+
(-2)	1	1	0	
(-1)	1	1	1	

-1 + (-1) → -1 – 1

(-1)	1	1	1	+
(-1)	1	1	1	
(-2)	1	1	0	

-2 + (-2) → -2 – 2

(-2)	1	1	0	+
(-2)	1	1	0	
(-4)	1	0	0	

-2 + 3 → 3 – 2

(-2)	1	1	0	+
(3)	0	1	1	
(1)	0	0	1	

Adder

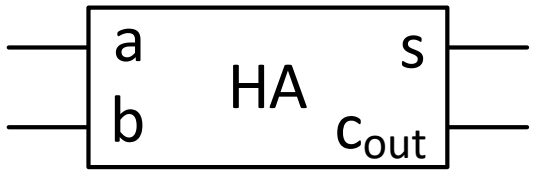
- The sum of 2 N-bit numbers requires N+1 bits to be represented
  - N = 1 → 2 bits to represent the sum

- The simplest adder is the **Half Adder** (**HA**)
  - Just the sum of the two 1-bit inputs (a, b)
  - Output must be 2-bit

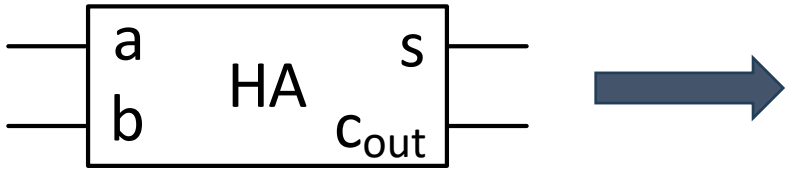
Adder

- The sum of 2 N-bit numbers requires N+1 bits to be represented
  - N = 1 → 2 bits to represent the sum

- The simplest adder is the **Half Adder** (**HA**)
  - Just the sum of the two 1-bit inputs (a, b)
  - Output must be 2-bit
    - s
    - c_{out} = **carry** (**output**)



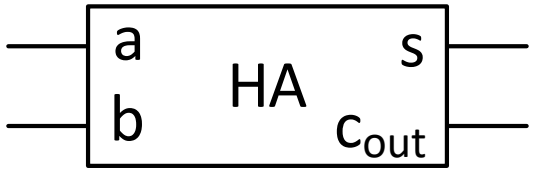
Half Adder (HA)



HA truth table

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

Half Adder (HA)



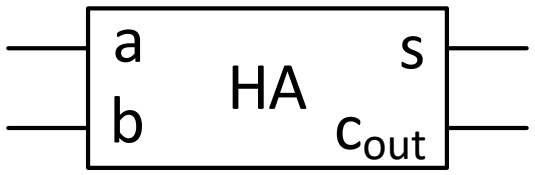
HA truth table

a	b	s	c_{out}
0	0	0	0
0	**1**	1	0
1	**0**	1	0
1	1	0	1

$s = 1$, when $a \neq b$

Half Adder (HA)

Does this remind you a logic operator we have seen before?

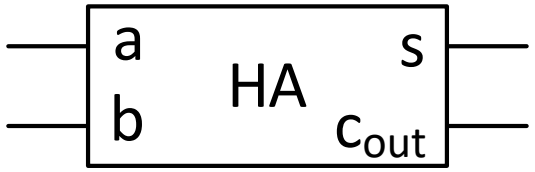


HA truth table

a	b	s	c_{out}
0	0	0	0
0	**1**	1	0
1	**0**	1	0
1	1	0	1

$s = 1$, when $a \neq b$

Half Adder (HA)



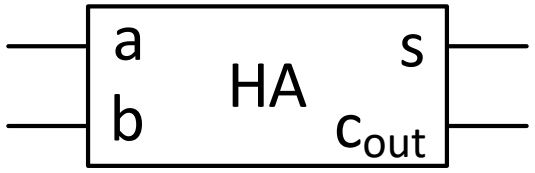
HA truth table

a	b	s	c_{out}
0	0	0	0
0	**1**	1	0
1	**0**	1	0
1	1	0	1

$s = 1$, when $a \neq b$

$$s = a \oplus b$$

Half Adder (HA)



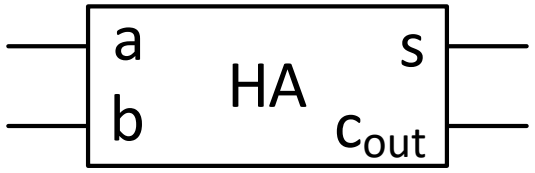
HA truth table

a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	**1**	0	1

$s = 1$, when $a \neq b$ → $s = a \oplus b$

$c_{out} = 1$, when $a = 1$ **and** $b = 1$

Half Adder (HA)



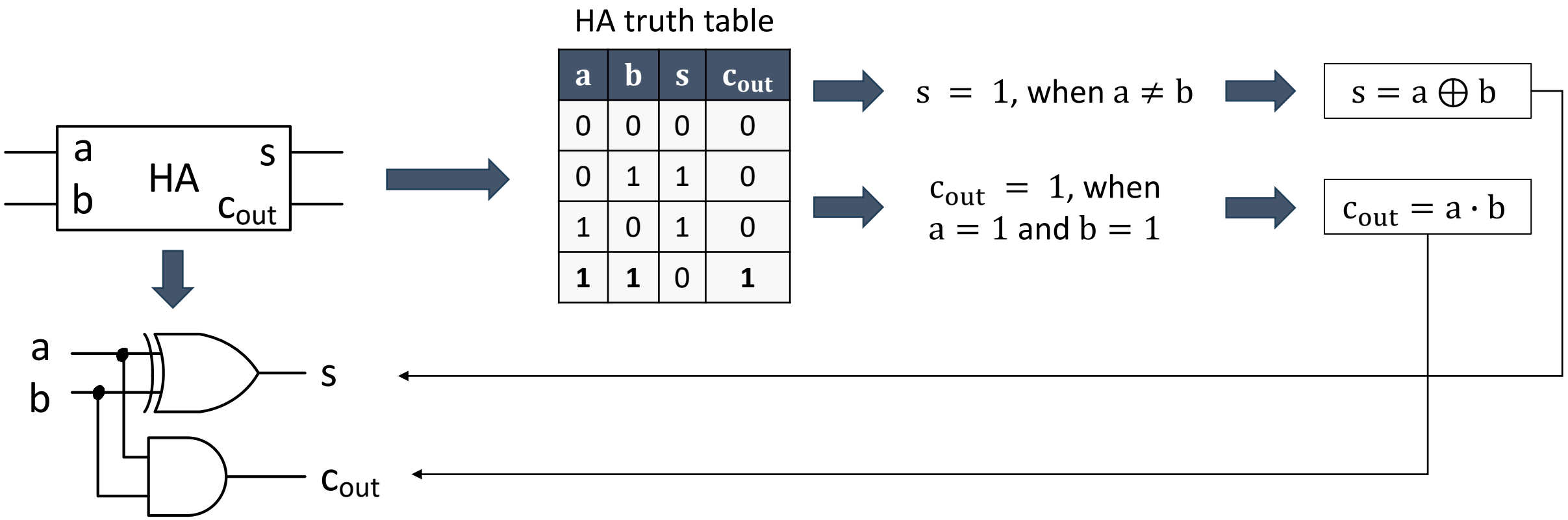
HA truth table

a	b	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	**1**	0	1

$s = 1$, when $a \neq b$ → $s = a \oplus b$

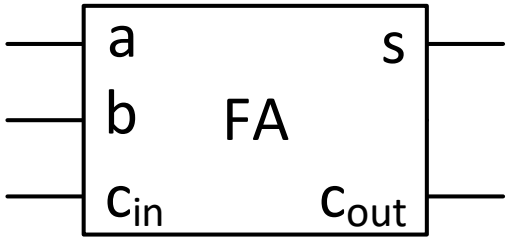
$c_{out} = 1$, when $a = 1$ **and** $b = 1$ → $c_{out} = a \cdot b$

Half Adder (HA)

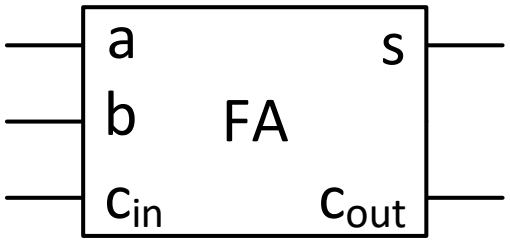


Adder

- But, as we saw earlier, when summing two multi-bit vectors, it may happen that the carry is generated by the sum of the previous bits
  - To fully support the addition, also a carry input must be considered
    - c_{in} = **carry** (**input**)
  - The corresponding (1-bit) adder is called **Full Adder** (**FA**)



Full Adder (FA)

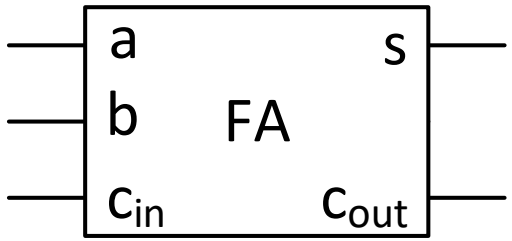


FA truth table

c_{in}	a	b	s	c_{out}
0	0	0	0	0
0	**0**	**1**	1	0
0	**1**	**0**	1	0
0	1	1	0	1
1	**0**	**0**	1	0
1	0	1	0	1
1	1	0	0	1
1	**1**	**1**	1	1

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

Full Adder (FA)



FA truth table

c_{in}	a	b	s	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	**1**	**1**	0	1
1	0	0	1	0
1	**0**	**1**	0	1
1	**1**	**0**	0	1
1	**1**	**1**	1	1

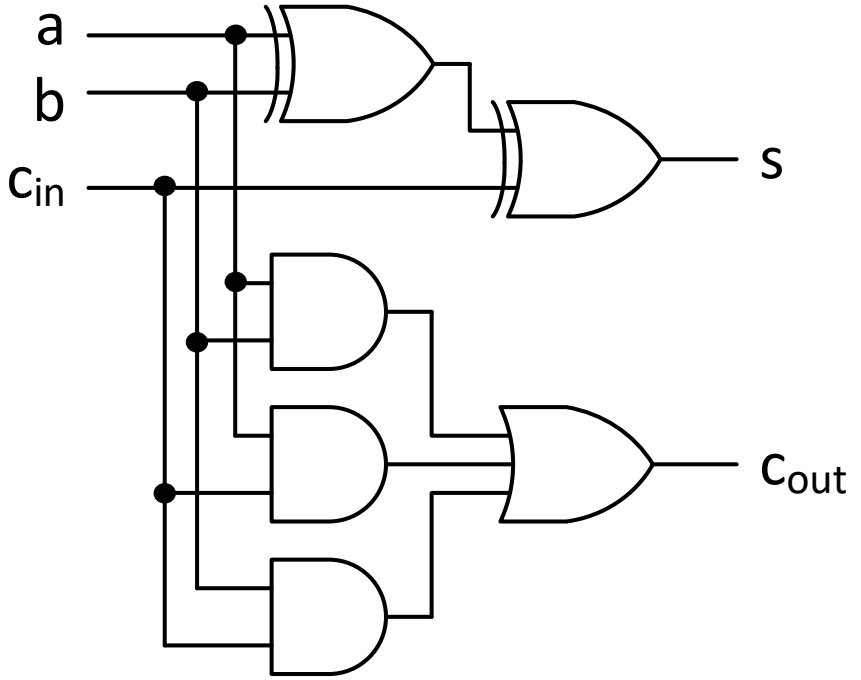
$$s = a \oplus b \oplus c_{in}$$

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

Full Adder (FA)

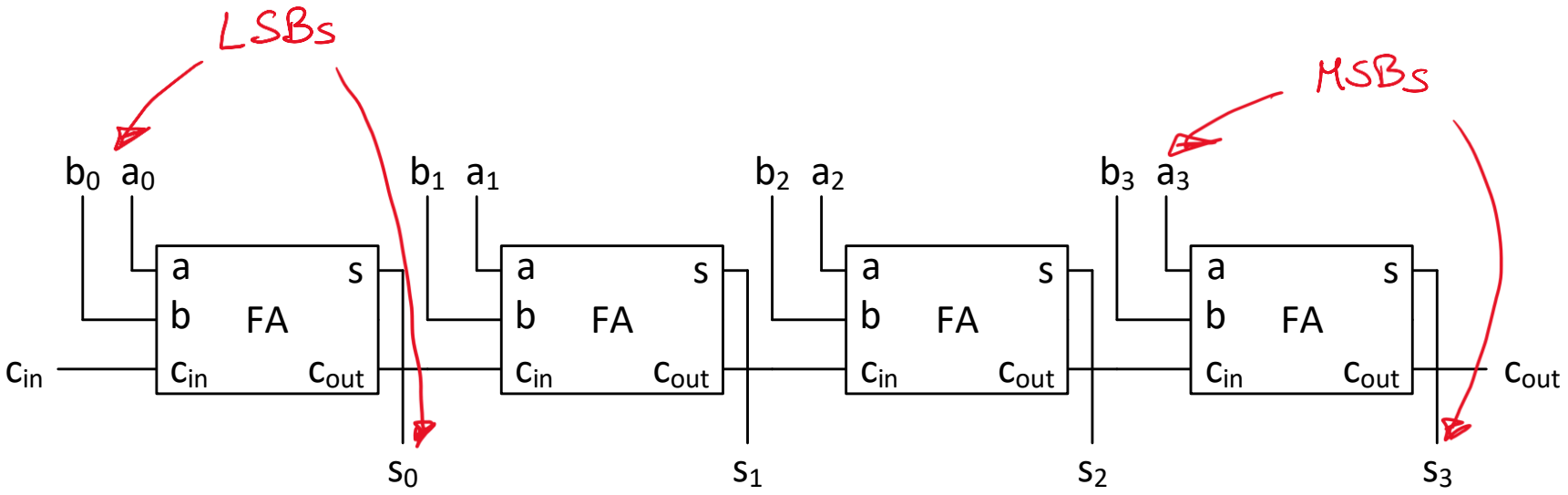
$$s = a \oplus b \oplus c_{in}$$

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



Adder (for multi-bit operands)

- A generic adder (for N-bit operands) can be built by cascading N FAs
  - Example: 4-bit adders
    - Inputs
      - $a = \{a_3\ a_2\ a_1\ a_0\}$
      - $b = \{b_3\ b_2\ b_1\ b_0\}$
      - c_{in}
    - Outputs
      - $s = \{s_3\ s_2\ s_1\ s_0\}$
      - c_{out}



Adder (for multi-bit operands)

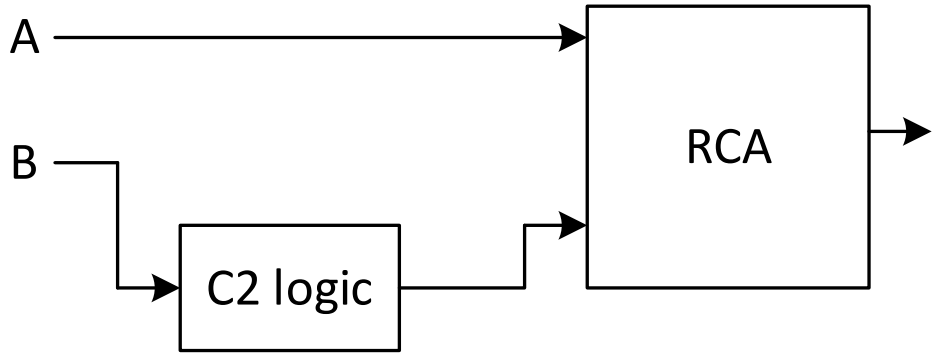
- The one saw before is called also Ripple Carry Adder (RCA)
  - The carry path propagate from the input to the output
    - Path is proportional to the bit width of the input operands
    - The delay path may be very high
  - Other solutions (and architectures/circuits) for adders exist to solve this problem
    - Carry Save Adder (CSA)
    - Carry Bypass Adder (CBA), or Carry Skip Adder
    - Carry Lookahead Adder (CLA)

Adder (for multi-bit operands)

- CSA, CBA, CLA, …: just for the sake of completeness
  - We are not going to analyze them
  - However, remember that each solution is a trade-off
    - They can be advantageous in terms of delay with respect to the RCA
    - But they can also present costs in other terms (e.g., resources)

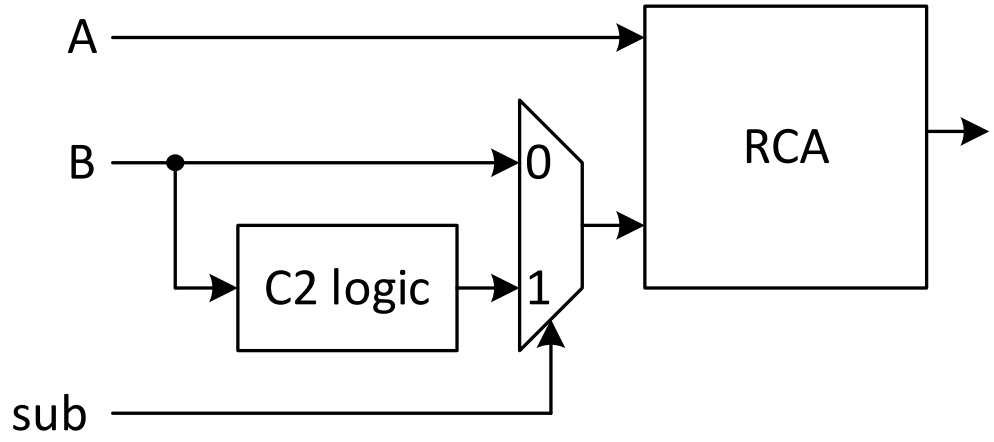
Subtractor (for multi-bit operands)

- If you remember the properties of two's complement representation…
  - A – B = A + (-B)



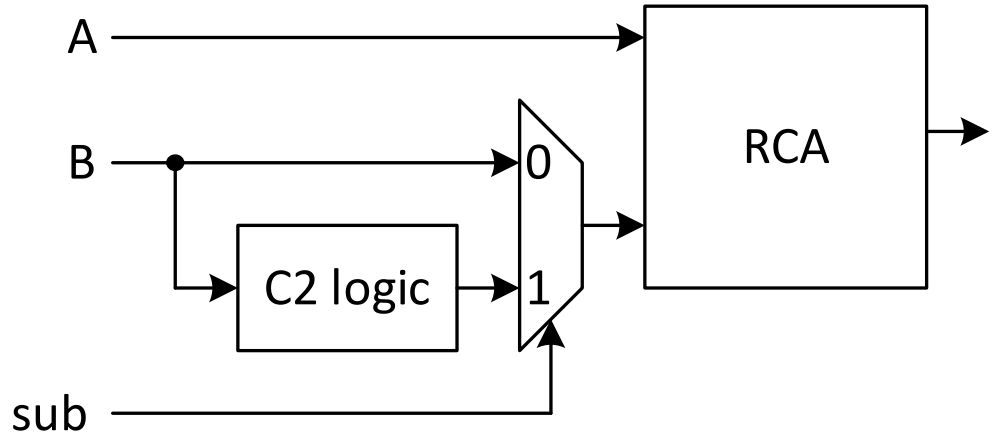
Adder/Subtractor (for multi-bit operands)

- For a full-case solution, i.e. adder/subtractor
  - Inputs : A, B
  - Control : **sub**
    - If **sub** = 0 → A + B
    - If **sub** = 1 → A – B = A + (-B)



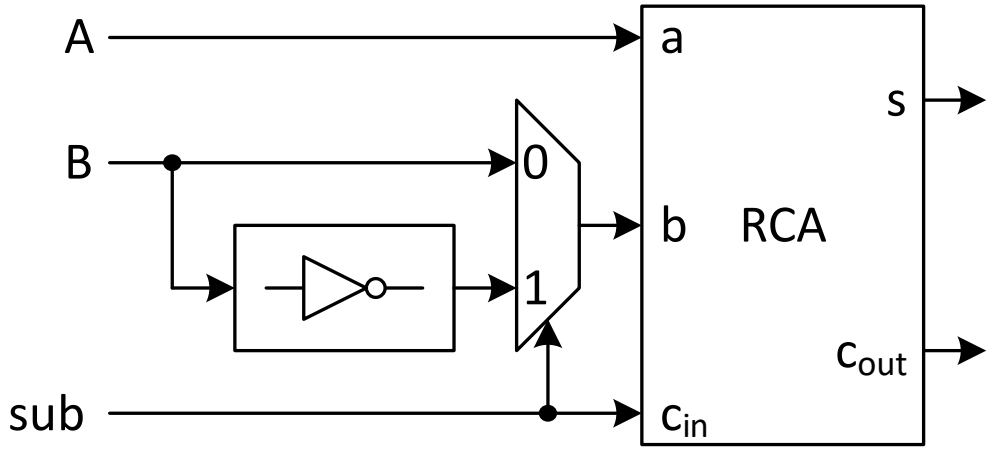
Adder/Subtractor (for multi-bit operands)

- For a full-case solution, i.e. adder/subtractor
  - If you remember, to get -B from B:
    - Complement all bits of B
    - Add 1
      - This can be done by exploiting the c_{in} input of RCA



Adder/Subtractor (for multi-bit operands)

- For a full-case solution, i.e. adder/subtractor
  - If you remember, to get -B from B:
    - Complement all bits of B
    - Add 1
      - This can be done by exploiting the c_{in} input of RCA



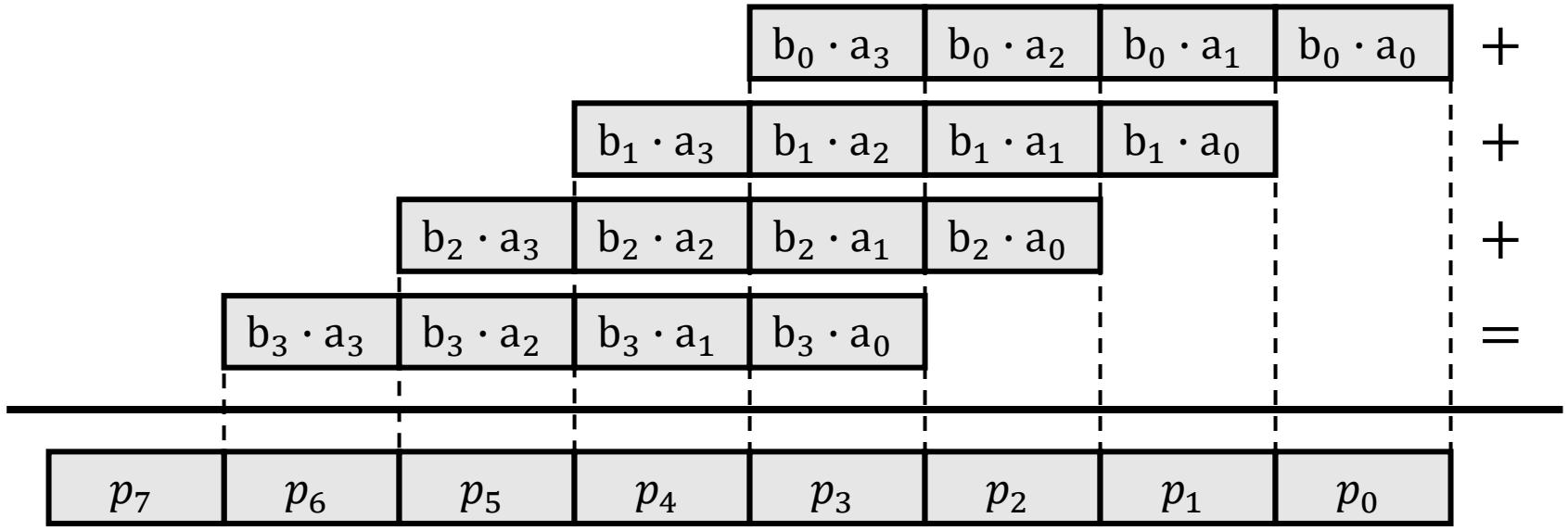
Multiplier

- The multiplication of a N-bit number by an M-bit number requires N+M bits to be represented
  - Typically, N = M → 2N bits to represent the product

- The simplest way: shift-and-add algorithm
  - Similar to the paper-and-pencil method for decimal multiplication

Multiplier

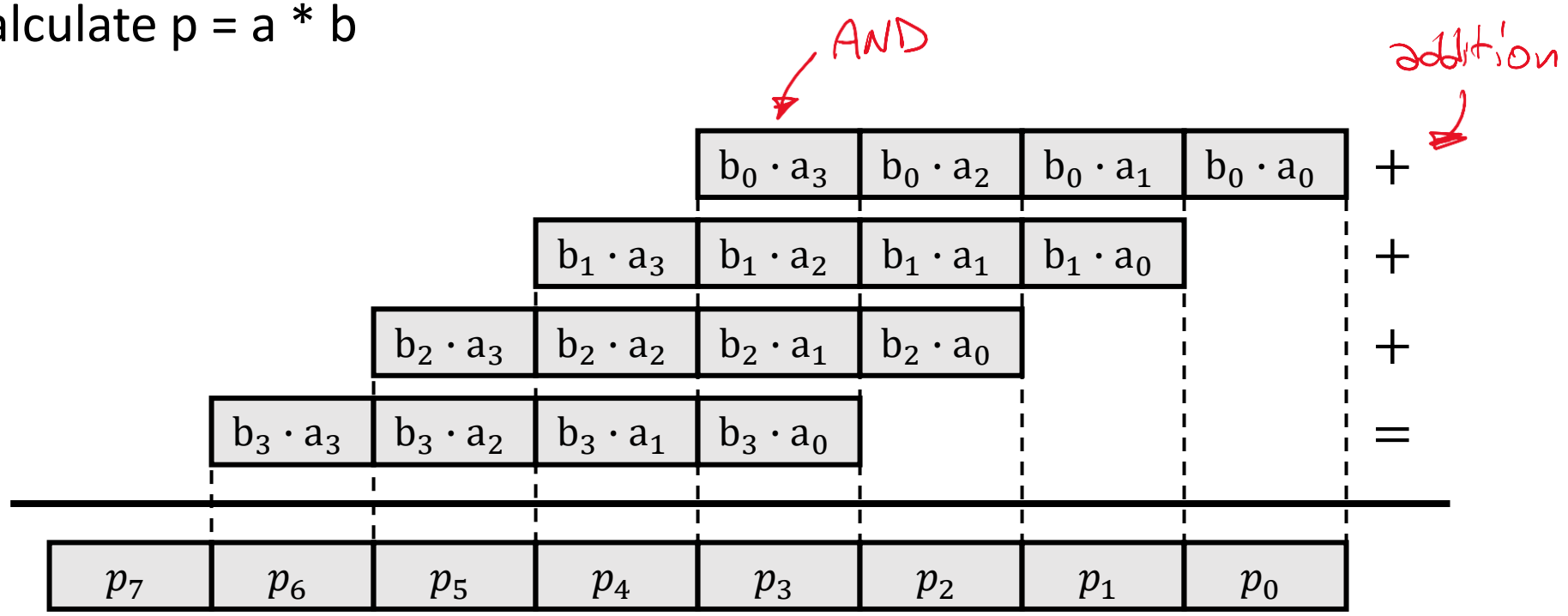
- Assume N = 4 and calculate p = a * b
  - a = {a_3 a_2 a_1 a_0}
  - b = {b_3 b_2 b_1 b_0}



- p

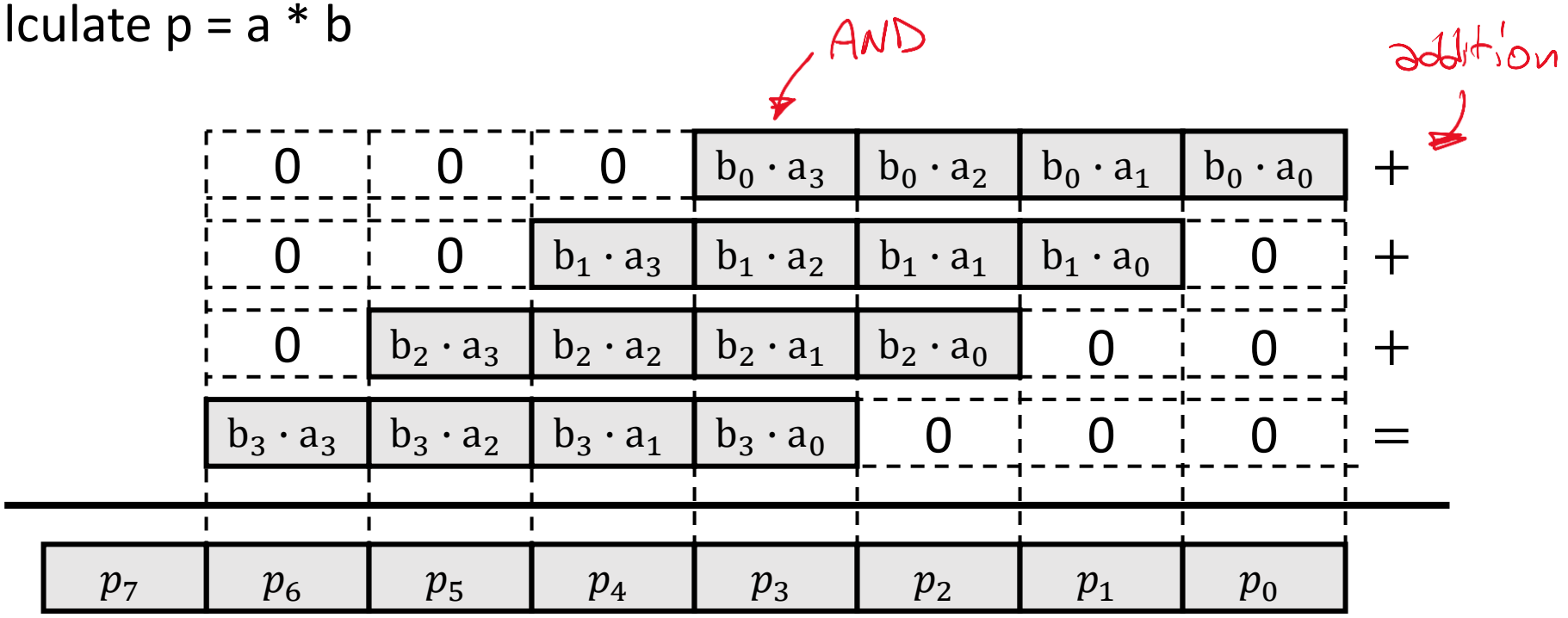
Multiplier

- Assume N = 4 and calculate p = a * b
  - a = {a_3 a_2 a_1 a_0}
  - b = {b_3 b_2 b_1 b_0}
  - p



Multiplier

- Assume N = 4 and calculate p = a * b
  - a = {a_3 a_2 a_1 a_0}
  - b = {b_3 b_2 b_1 b_0}



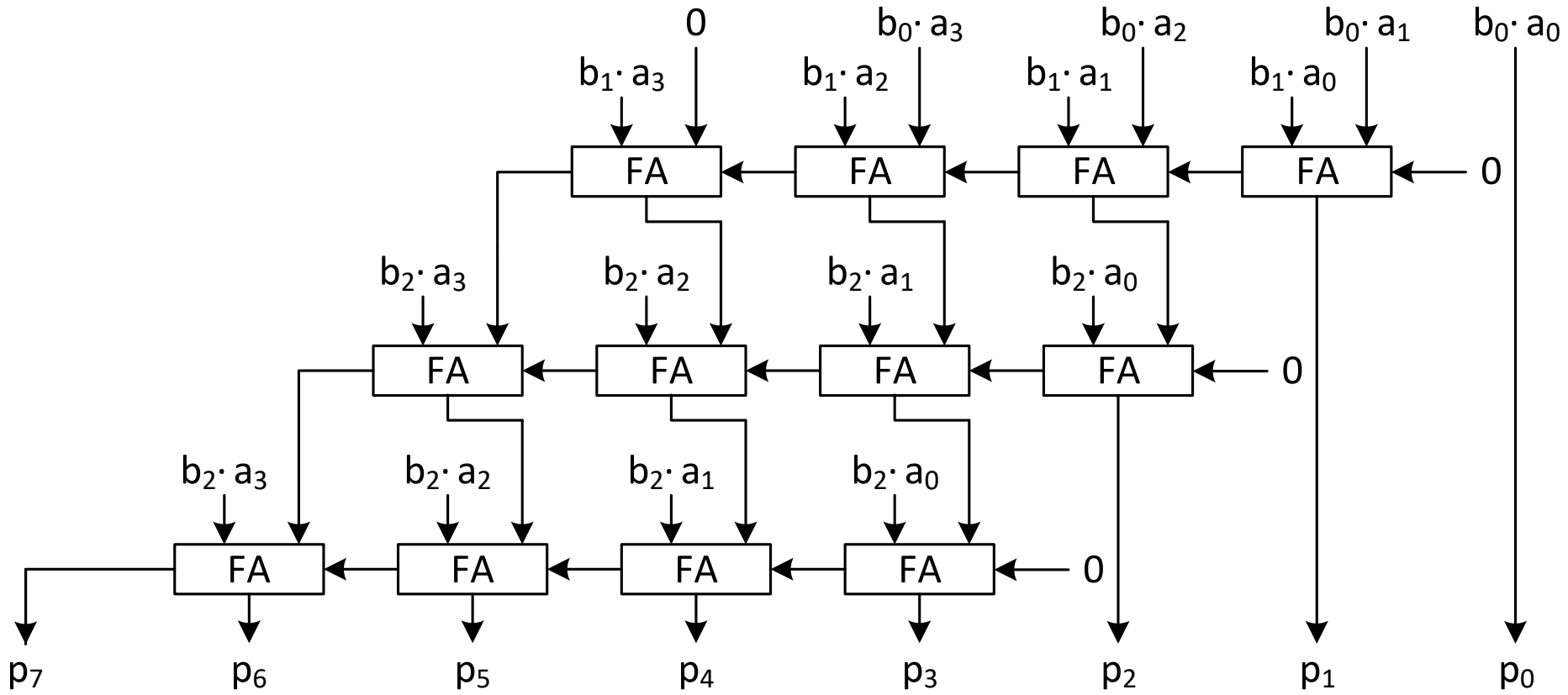
- **N – 1 additions of numbers with a bit width of 2N – 1 → Sequential multiplier**

Sequential multiplier

- Logic for calculation of AND products
- One adder
  - The RCA we saw before
- A register to store partial products
- Requiring **N – 1 steps/cycles** to perform the multiplication

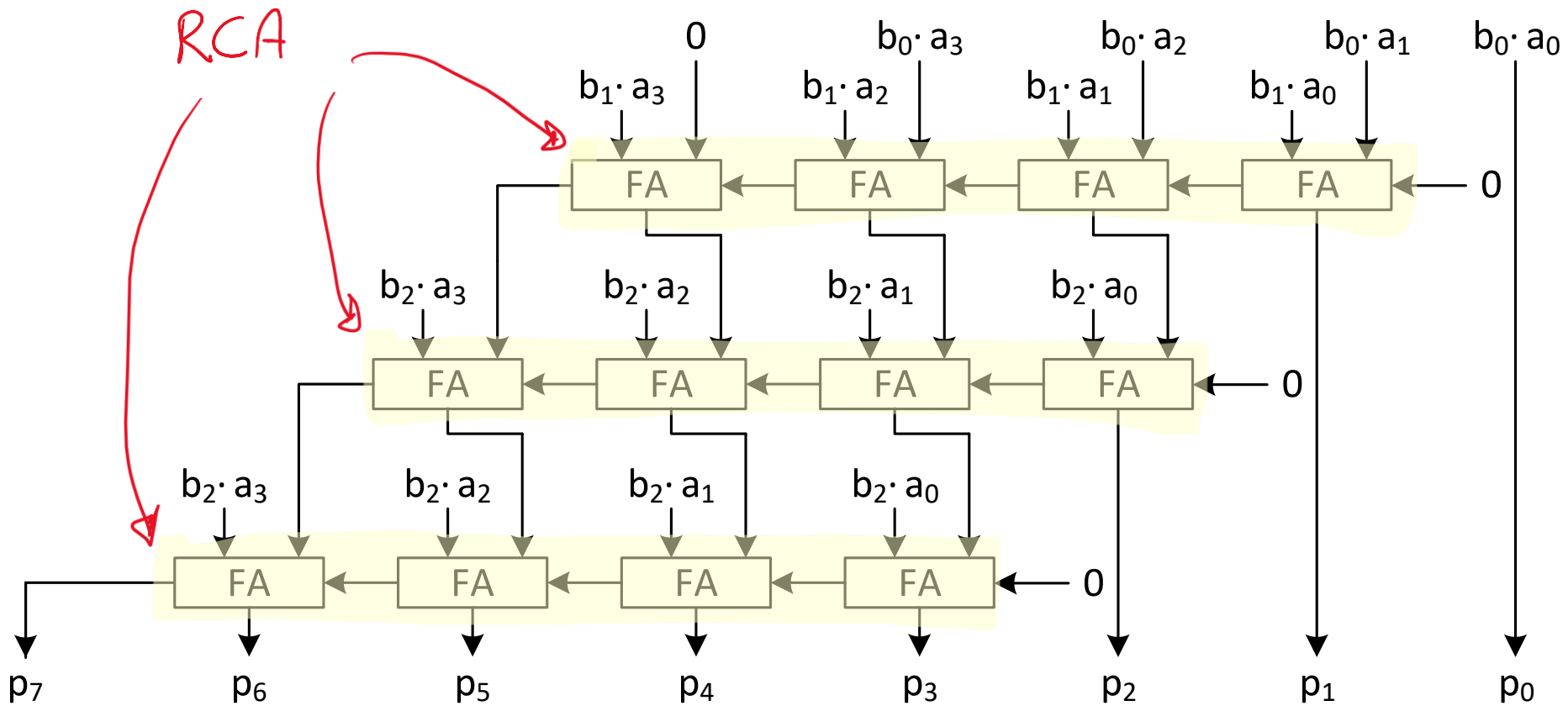
Combinational multiplier

- Example: N = 4



Combinational multiplier

- Example: N = 4
  - 3x 4-bit RCAs
- In general
  - N – 1x RCAs
  - Each of N bits
- Plus logic for AND!!!



Divisor

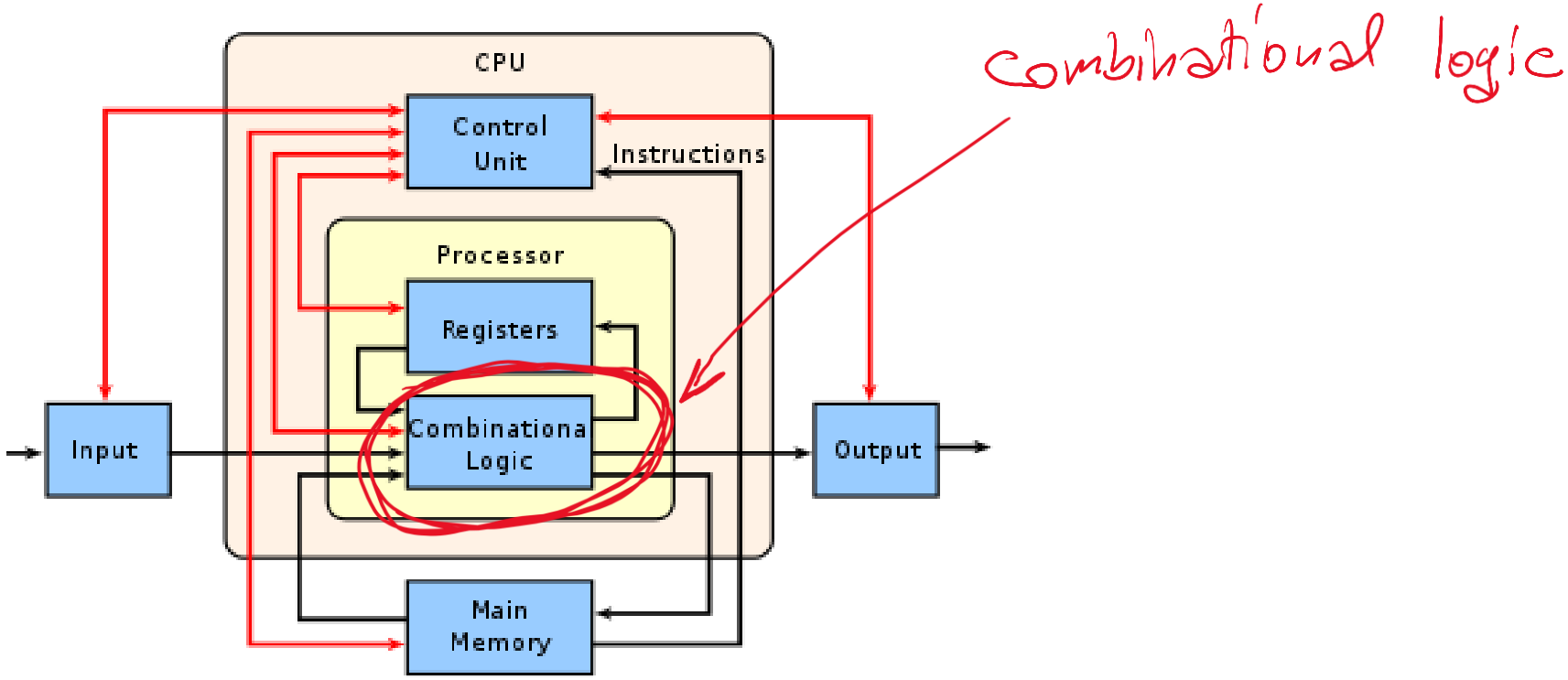
- Similarly to multiplier, the divisor can be implemented exploiting again an adder (RCA) supporting two's complement addition
  - Division can be split in a sequence of subtractions and comparison
  - Example: 63 / 12
    - Quotient = 0
    - 63 – 12 = 51 → Quotient += 1 (1), 51 > 12 ? Yes
    - 51 – 12 = 39 → Quotient += 1 (2), 39 > 12 ? Yes
    - 39 – 12 = 27 → Quotient += 1 (3), 13 > 12 ? Yes
    - 27 – 12 = 15 → Quotient += 1 (4), 15 > 12 ? Yes
    - 15 – 12 = 3 → Quotient += 1 (5), 3 > 12 ? No → Reminder = 3
  - Or other combinational circuits can be implemented

Multiplicator/Divisor

- We are not going to see the circuit of a combinational divisor, however …
- … remember that multipliers and divisors are expansive
  - Resources and/or time (steps/cycles)

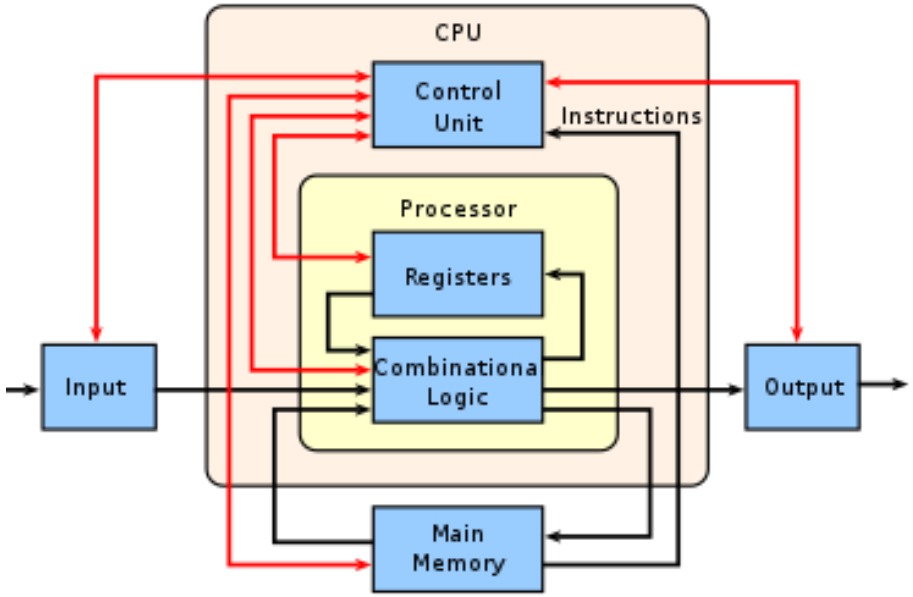
Combinational logic in modern processors

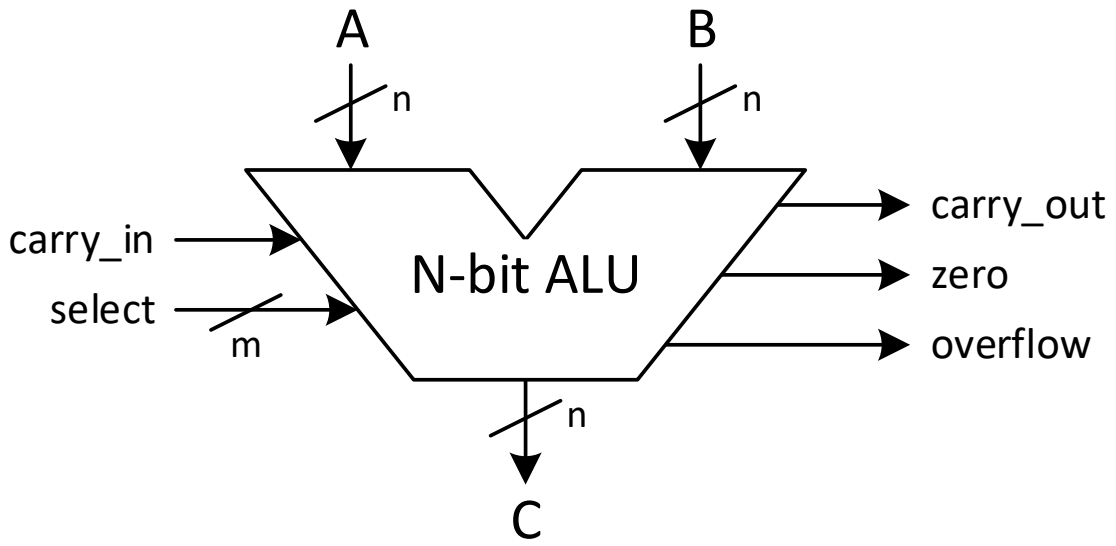
- If you remember the overall outline of a modern processor



Combinational logic in modern processors

- The main component of the Combinational logic block is called **ALU**
  - **ALU** = **A**rithmetic **L**ogic **U**nit

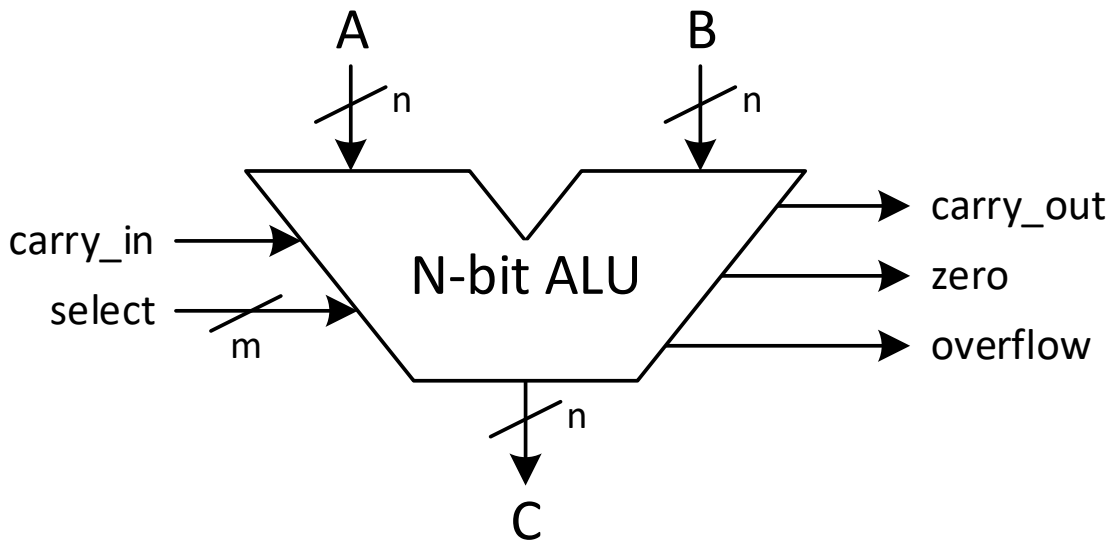




ALU

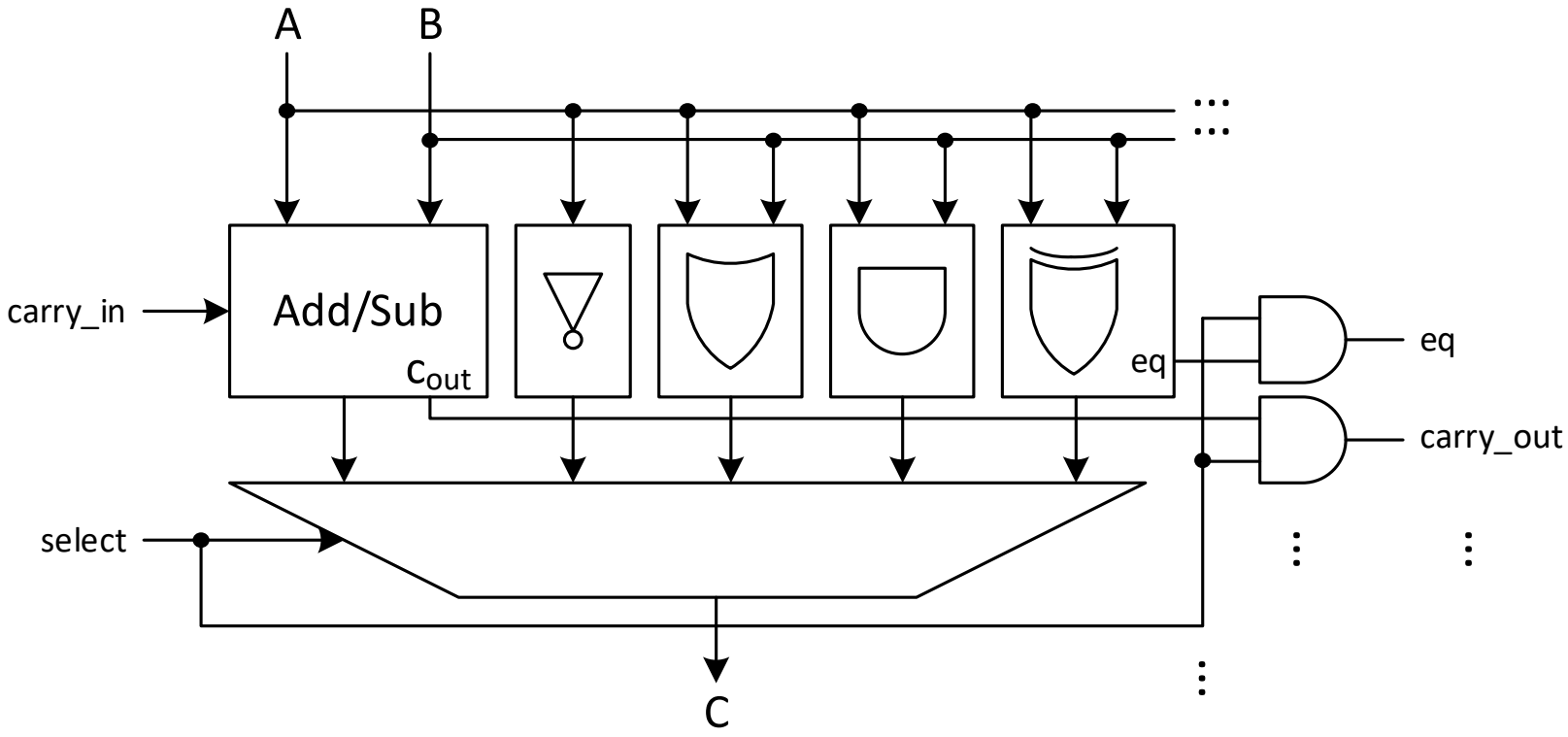
- Main operations (but not all) supported by an ALU

OPCODE (select)		Meaning
ADD	00001	A + B
SUB	00010	A – B
MULT	01000	A * B
NOT	10000	Complement A
AND	10001	Bit-wise AND
OR	10010	Bit-wise OR
XOR	10011	Bit-wise XOR
EQL	01011	Check A == B
…	…	…



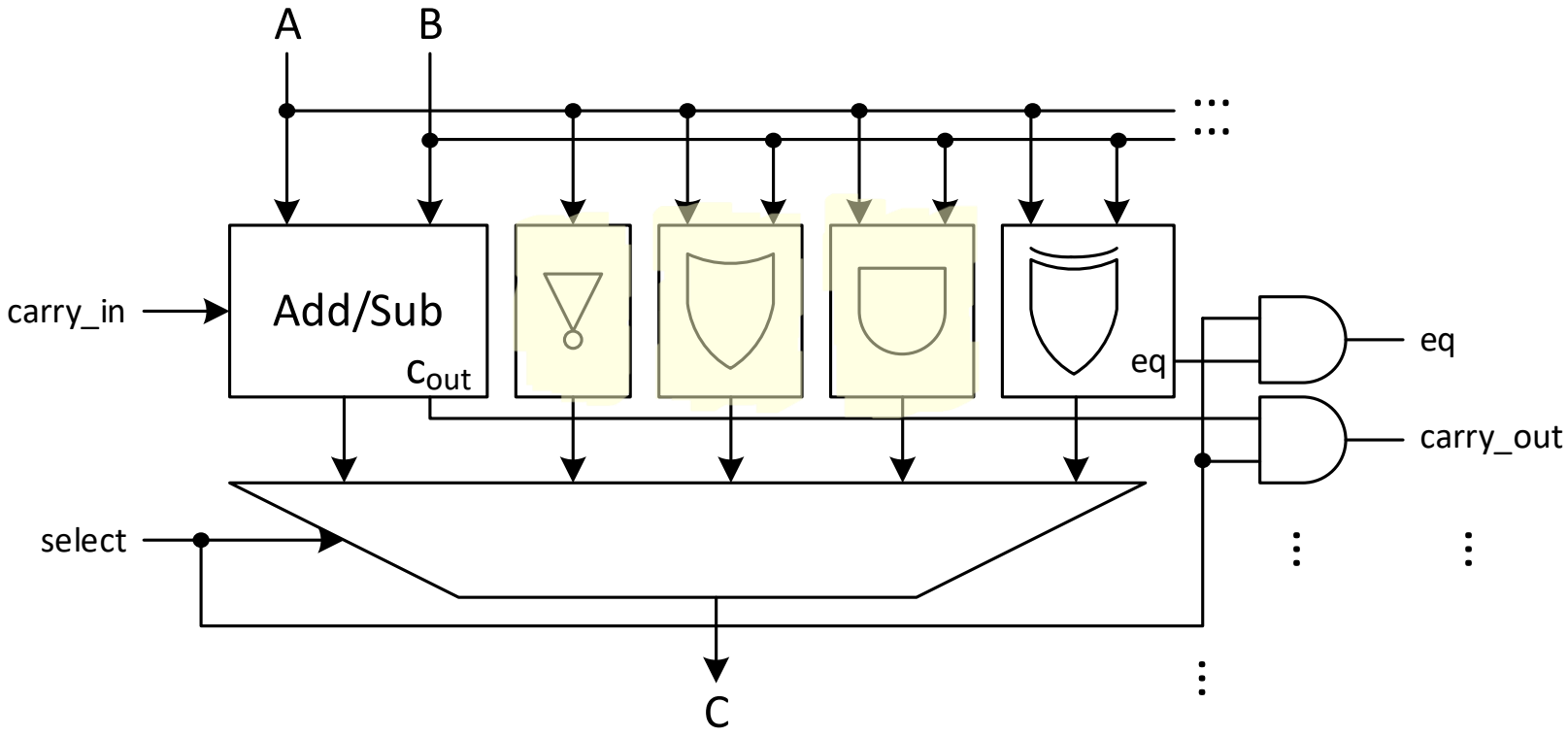
ALU

- Therefore, a possible architecture for an ALU can be the following one



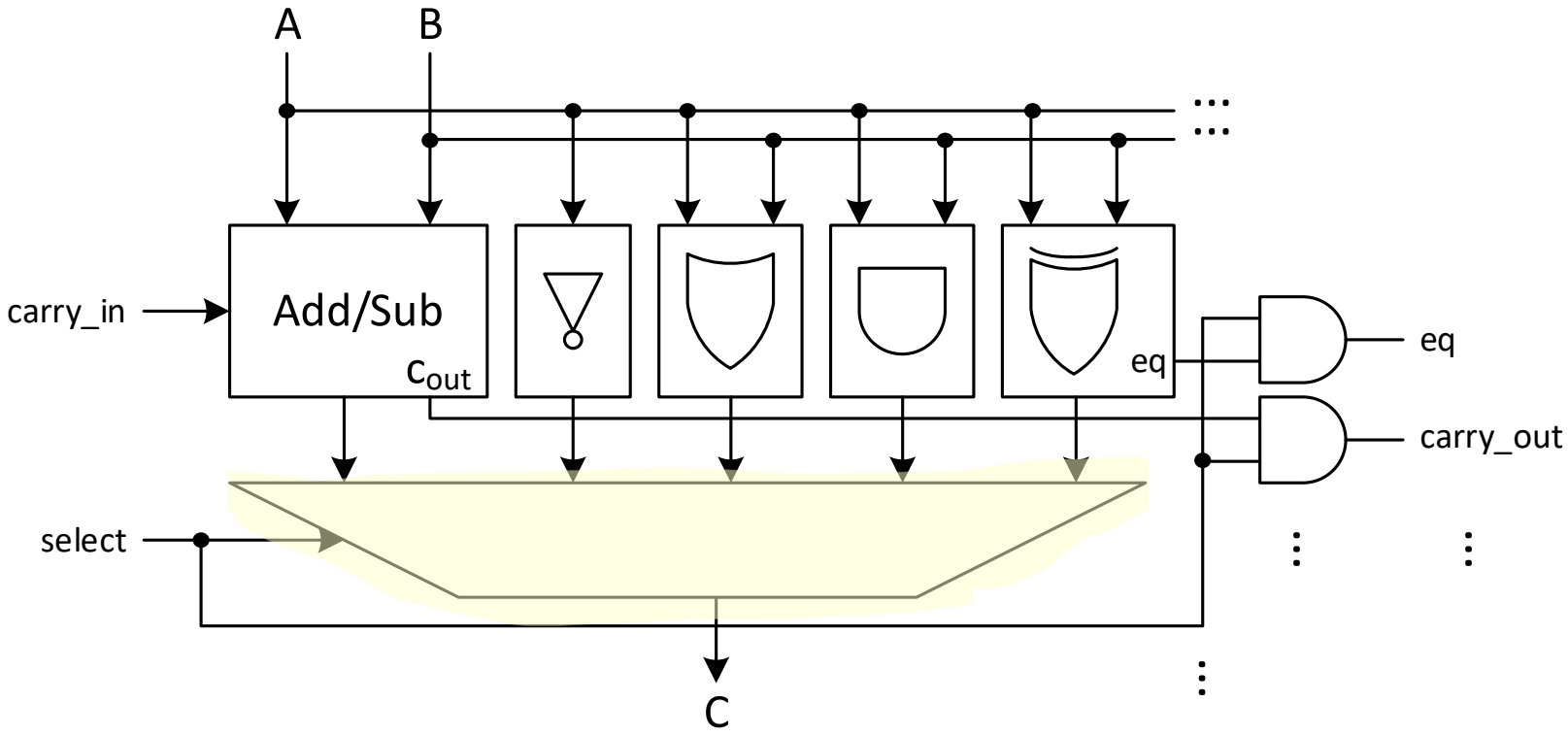
ALU

- Therefore, a possible architecture for an ALU can be the following one
  - Basic logic gates



ALU

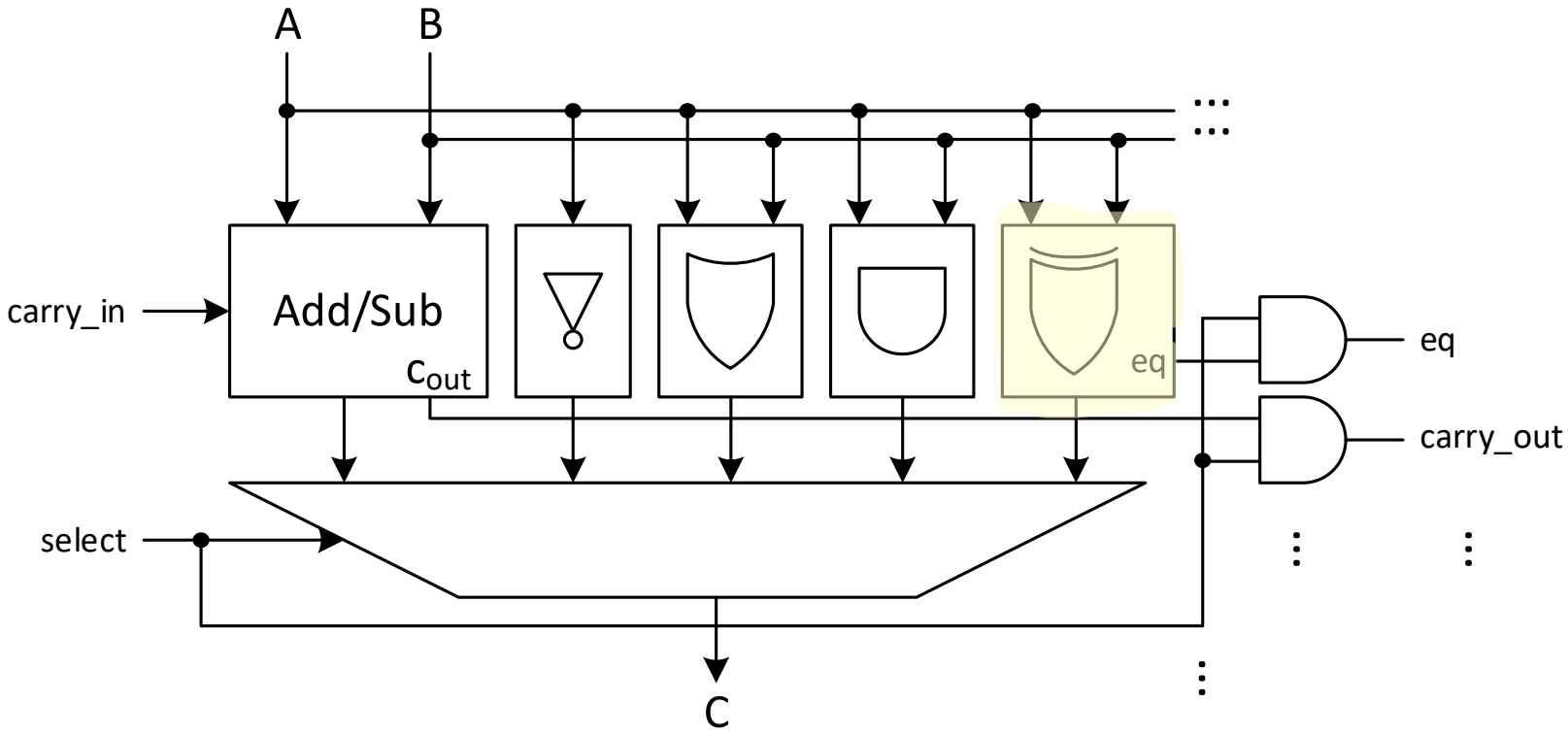
- Therefore, a possible architecture for an ALU can be the following one
  - Basic logic gates
  - Multiplexers



ALU

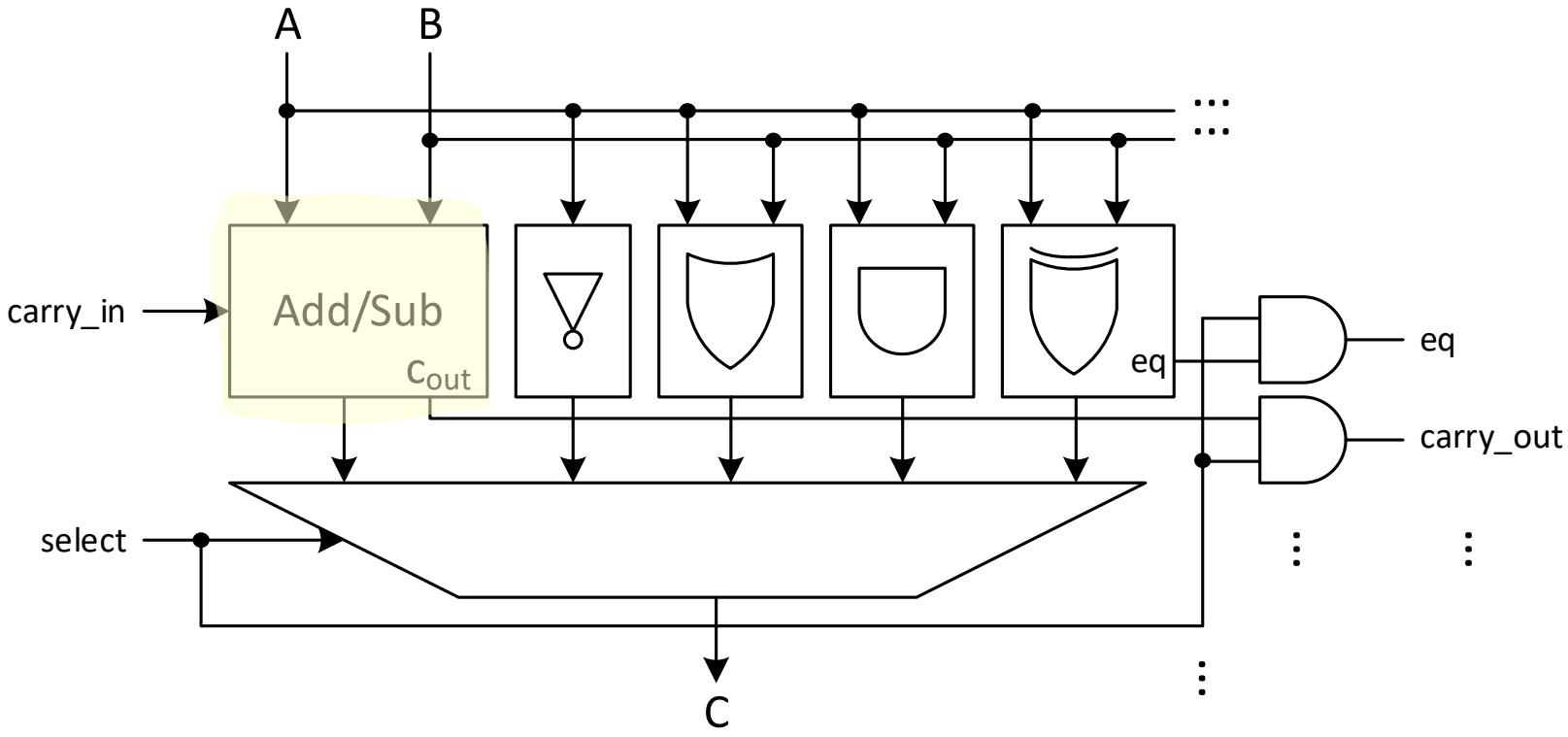
- Therefore, a possible architecture for an ALU can be the following one
  - Basic logic gates
  - Multiplexers
  - XOR/Comparator

Comparator can be implemented cascading XOR gates



ALU

- Therefore, a possible architecture for an ALU can be the following one
  - Basic logic gates
  - Multiplexers
  - XOR/Comparator
  - Adder/Subtractor



Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

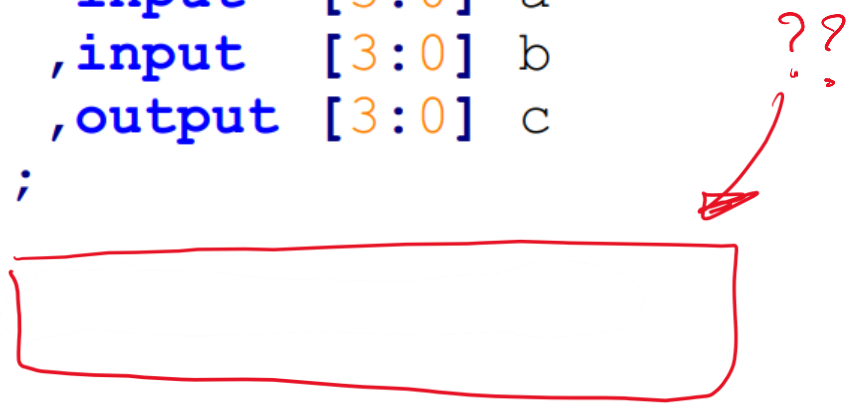
```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

endmodule
```

4-bit signals

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

endmodule
```



Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

   assign c = a + b;

endmodule
```

- Continuous assignment (again)
  - assign <signal> = <expression>;
  - But using arithmetic operator (+) in <expression>

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

  assign c = a + b;

endmodule
```

- **Some operators in SV**
  - Arithmetic
    - They can be use in any expression/assignment (also blocking or non-blocking)

    - Addition +
    - Subtraction –
    - Multiplication *
    - Division /
    - Power ** (e.g., a**b = a^b)
    - Mod % (e.g., a % b = a mod b)

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

  assign c = a + b;

endmodule
```

- **Some operators in SV**
  - Bit-wise
    - They can be use in any expression/assignment (also blocking or non-blocking)
    - AND &
    - OR |
    - NOT ~

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

  assign c = a + b;

endmodule
```

- **Some operators in SV**
  - Logical
    - They can be used in any control condition (if–else, ternary operator)
    - AND &&
    - OR ||
    - NOT !

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim

```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

  assign c = a + b;

endmodule
```

- **Some operators in SV**
  - Relational
    - They can be used in any control condition (if–else, ternary operator)

    - Equality ==
    - Inequality !=
    - Greater than >
    - Greater than or equal >=
    - Lower than <
    - Lower than or equal <=

Exercise with SystemVerilog

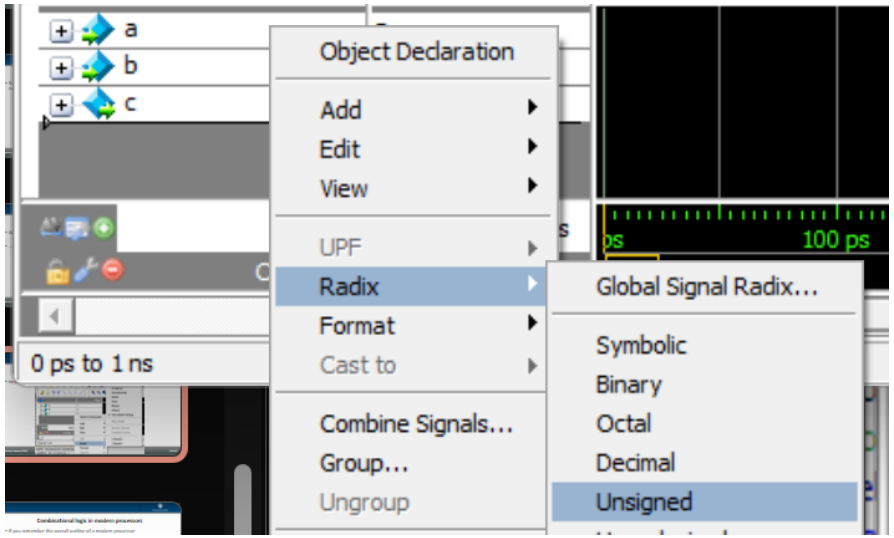
- Implementation of a 4-bit Adder and simulation with Modelsim

```
module adder_4_bit (
    input  [3:0] a
   ,input  [3:0] b
   ,output [3:0] c
);

  assign c = a + b;

endmodule
```

- Try to simulate it on Modelsim with the **force** and **run** command
  - To assign a multi-bit value with the force command use the syntax for constants in SV
    - force a 4'd0
    - force b 4'd3
    - …

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim



- Try to simulate it on Modelsim with the **force** and **run** command
  - A suggestion: to plot the waveform, use the Unsigned radix (for a, b, and c)

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim
  - Signed version: supporting two's complement operands

```
module adder_c2_4_bit (
    input  signed [3:0] a
   ,input  signed [3:0] b
   ,output signed [3:0] c
);

  assign c = a + b;

endmodule
```

- Signals must be declared as signed
  - By default, all signals are unsigned

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim
  - Signed version: supporting two's complement operands

```
module adder_c2_4_bit (
    input  signed [3:0] a
   ,input  signed [3:0] b
   ,output signed [3:0] c
);

  assign c = a + b;

endmodule
```

- Signals must be declared as **signed**
  - By default, all signals are unsigned
  - It applies to both ports and internal signals

```
// ...
wire signed [3:0] d;
reg signed  [7:0] e;
// ...
```

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim
  - Signed version: supporting two's complement operands

```
module adder_c2_4_bit (
    input  signed [3:0] a
   ,input  signed [3:0] b
   ,output signed [3:0] c
);

  assign c = a + b;

endmodule
```

- Signals must be declared as **signed**
  - By default, all signals are unsigned
  - It applies to both ports and internal signals

```
// ...
wire signed [3:0] d;
reg signed  [7:0] e;
// ...
```

  - It must be specified <u>after the type</u>

Exercise with SystemVerilog

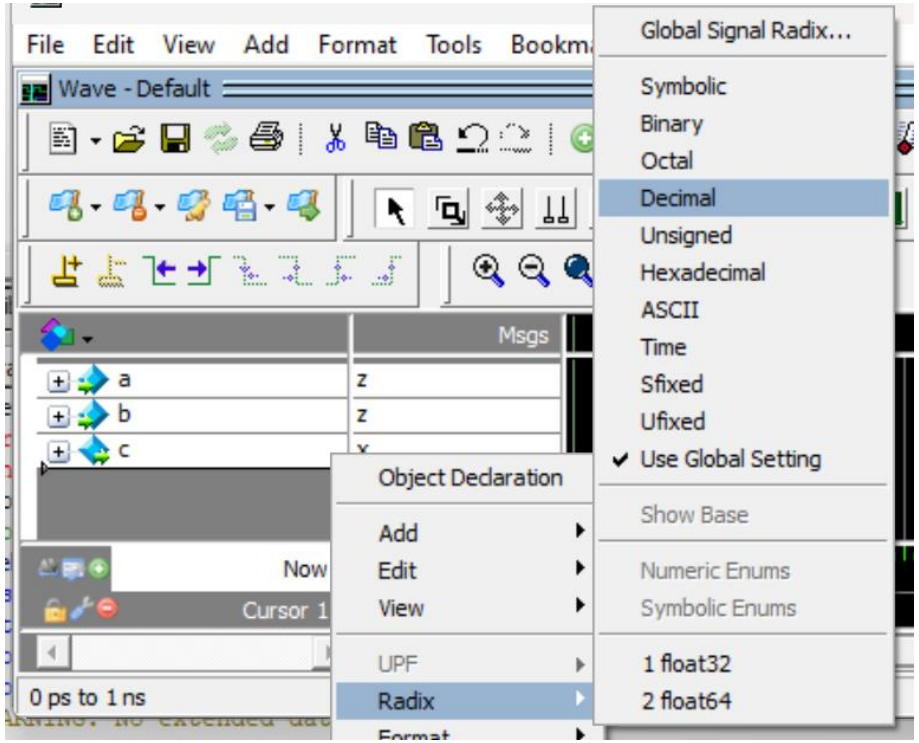
- Implementation of a 4-bit Adder and simulation with Modelsim
  - Signed version: supporting two's complement operands

```
module adder_c2_4_bit (
    input  signed [3:0] a
   ,input  signed [3:0] b
   ,output signed [3:0] c
);

  assign c = a + b;

endmodule
```

- Try to simulate it on Modelsim with the **force** and **run** command
  - To assign negative value with the force command use …
    - force a –4'd1
    - force b –4'd3
    - …
    - Remember of representable range!!!

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim
  - Signed version: supporting two's complement operands



- Try to simulate it on Modelsim with the **force** and **run** command
  - A suggestion: to plot the waveform, use the Decimal radix (for a, b, and c)

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim
  - You can find all the files about this exercise in the dedicated folder on the Team of the course
    - File > Electronics Systems module > Crocetti > Exercises > 2.2
  - Try to simulate on your own both unsigned and signed version of the 4-bit adder
    - Unsigned version → ‘adder_4_bit’
    - Signed version → ‘adder_c2_4_bit’

Exercise with SystemVerilog

- Implementation of a 4-bit Adder and simulation with Modelsim
  - For the signed version, you can find ‘adder_c2_4_bit’ two auxiliary .do files
    - ‘wave_adder_c2.do’
    - ‘sim_adder_c2.do’
    - Refer to README.txt file
    - However, you can run them using again the Transcript Tab and the **do** command
      - After having launched the simulation

```
VSIM 24> do wave_adder_c2.do
VSIM 25> do sim_adder_c2.do
```

Thank you for your attention

Luca Crocetti
(luca.crocetti@unipi.it)