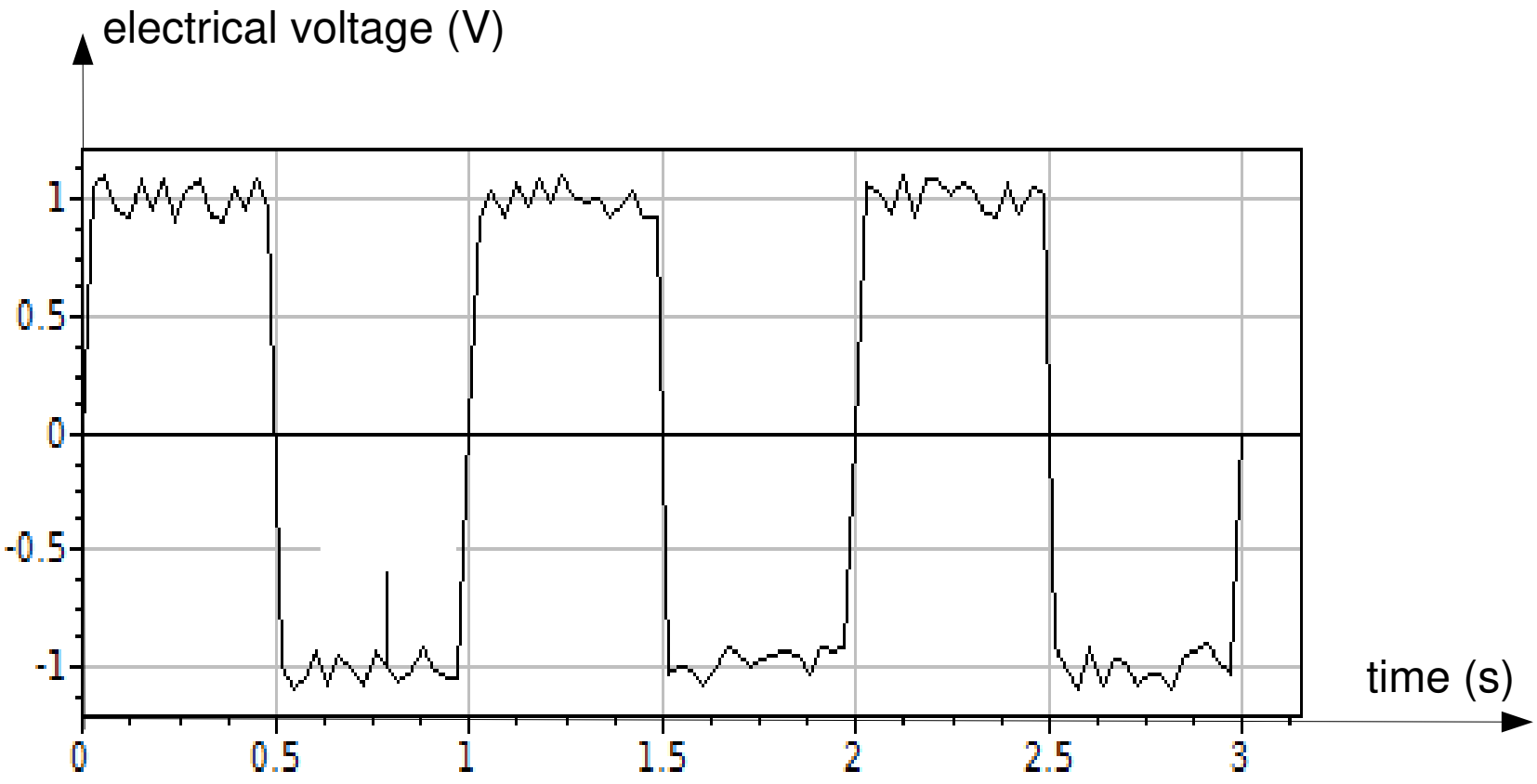
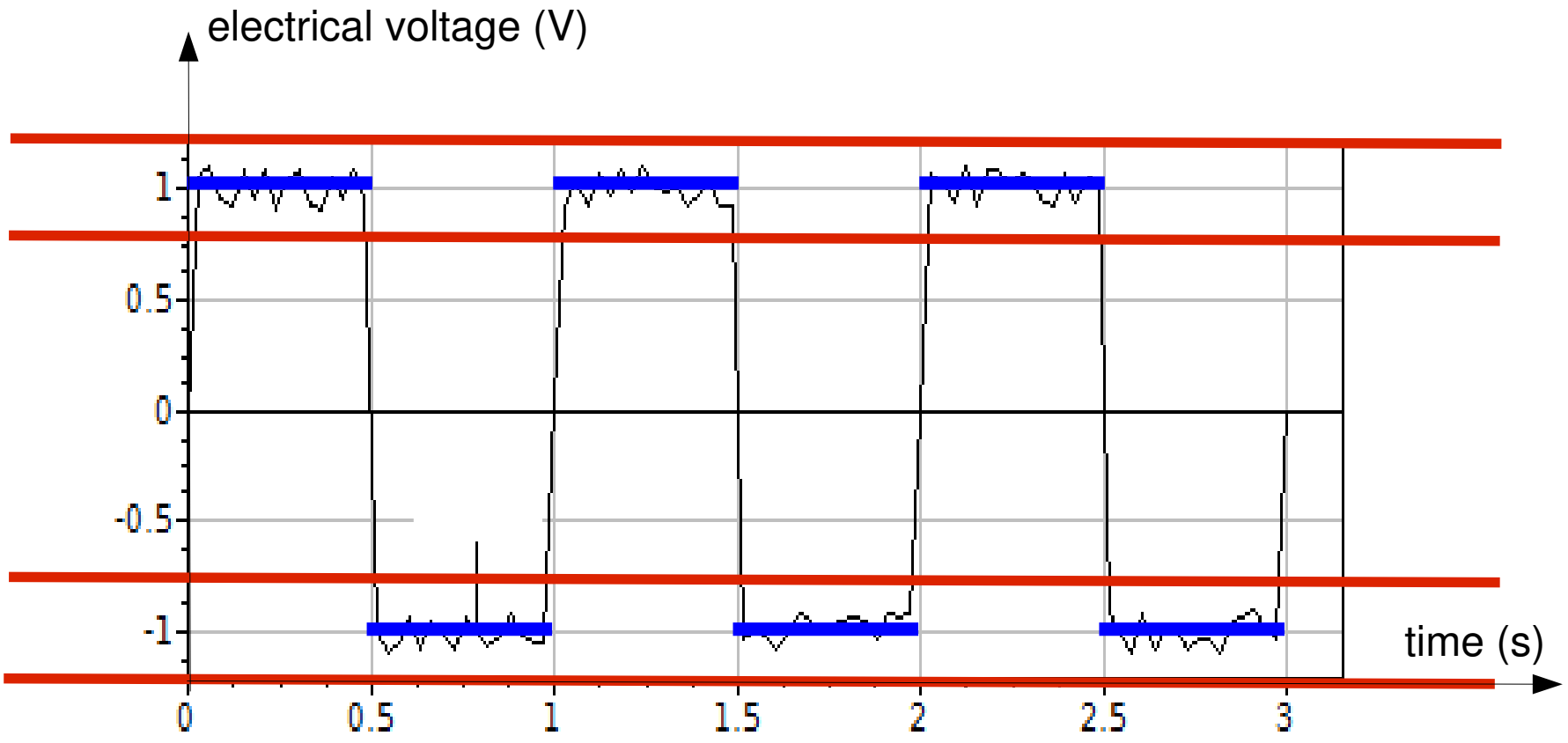
Logic Design: Implementation of functionality by means of “Logic Gates”

(Appendix B of P&H textbook)

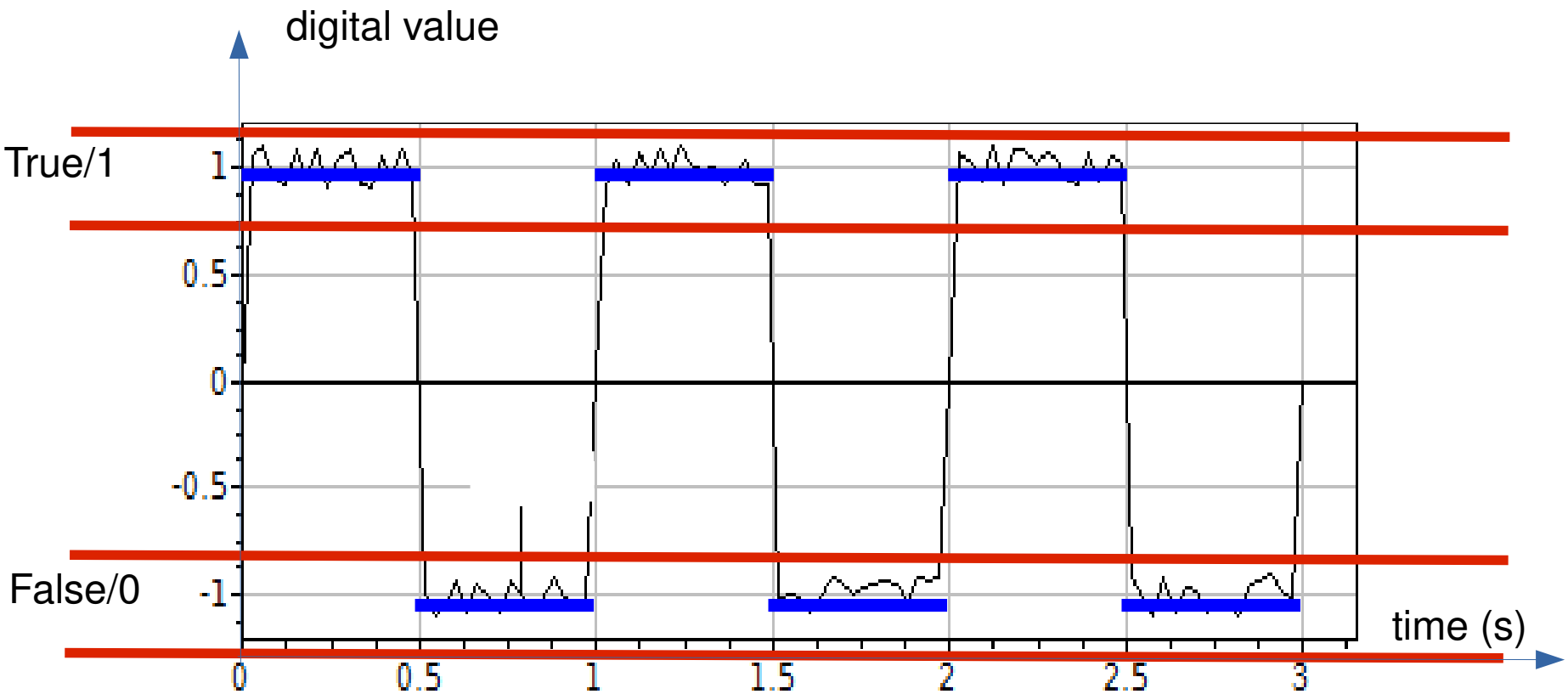
from Analog ...



… to Digital

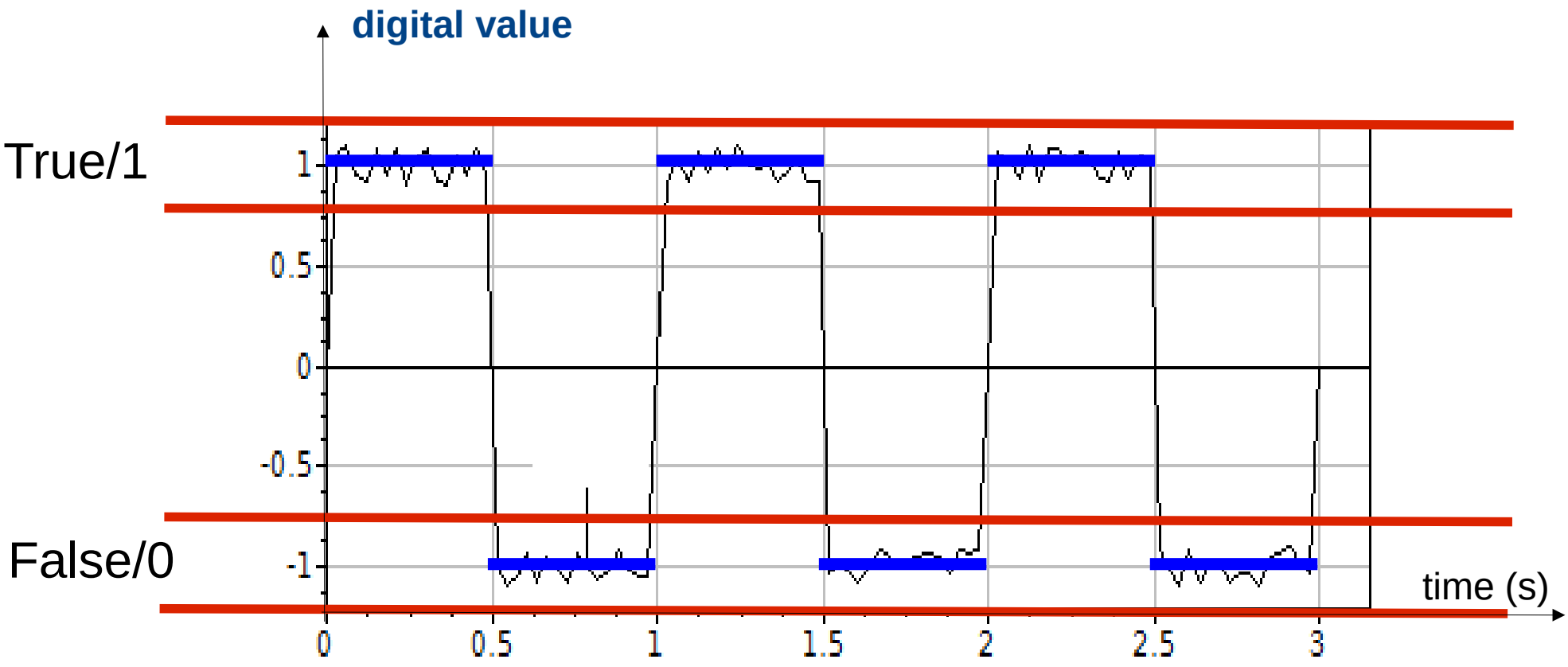


… to Digital



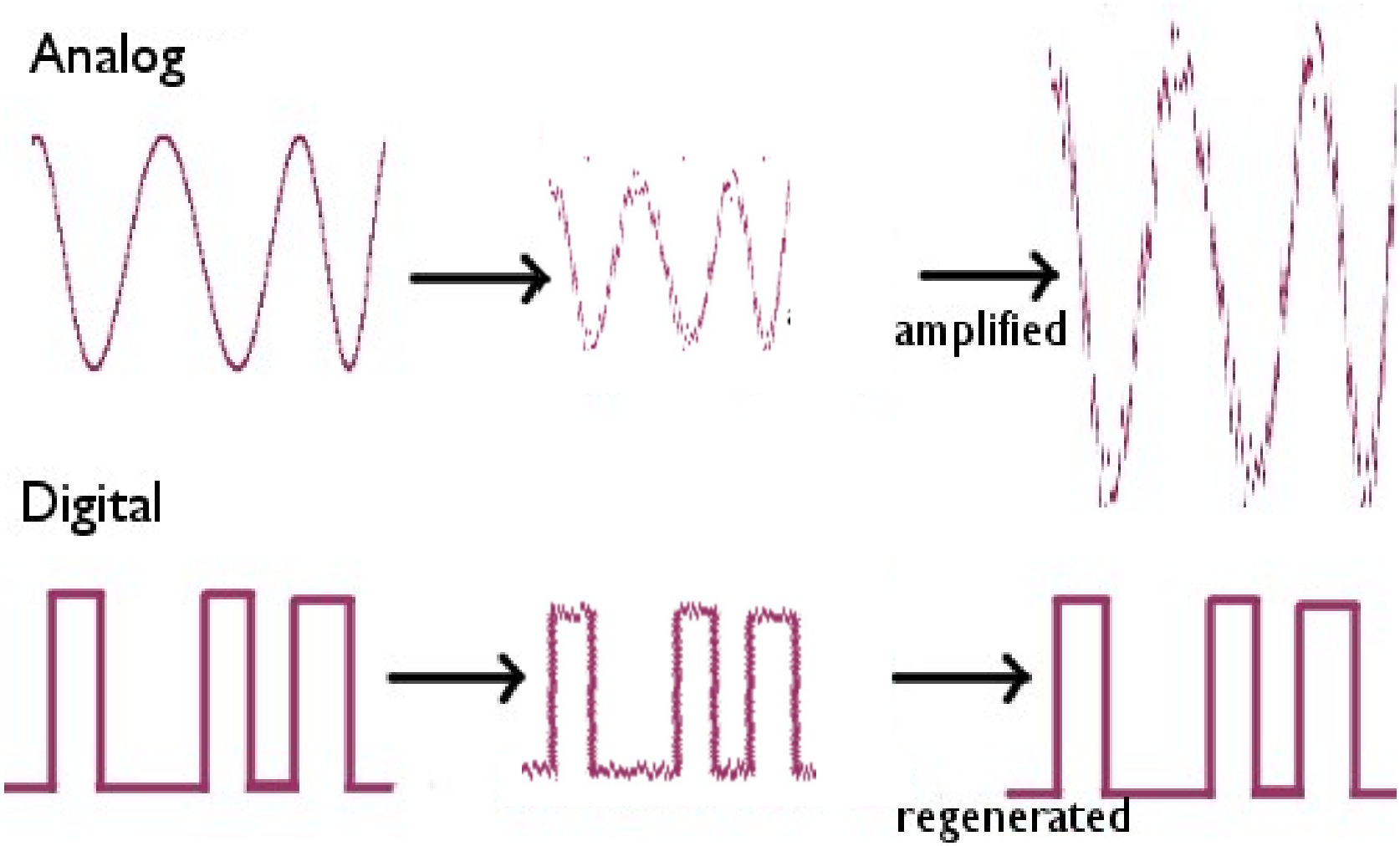
logical 0/1 – Boolean True/False – asserted/de-asserted – high/low

Digital Signals are based on Analog Signals
→ Digital (logic) operations process Analog Signals



logical 0/1 – Boolean True/False – asserted/de-asserted – high/low

Attenuation, Noise, … : Analog vs. Digital

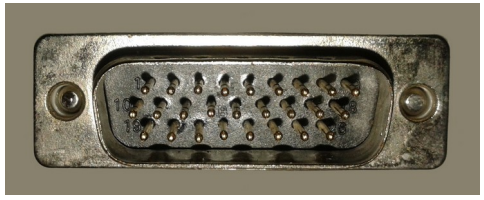


RS232 (serial communication data transmission)

The **RS-232 standard** defines the **voltage levels** that correspond to **logical one** and **logical zero** levels for the **data transmission** and the **control signal lines**.

Valid signals are either in the range of **+3 to +15 volts** or the range **−3 to −15 volts** with respect to the "Common Ground" (**GND**) pin; consequently, the range between **−3 and +3 volts** is not a valid RS-232 level.

For **data transmission lines** (TxD, RxD, and their secondary channel equivalents), **logic one** is represented as a **negative voltage** and the signal condition is called "mark". **Logic zero** is signaled with a **positive voltage** and the signal condition is termed "space".
Control signals have the opposite polarity: the asserted or active state is positive voltage and the de-asserted or inactive state is negative voltage. Examples of control lines include request to send (RTS), clear to send (CTS), data terminal ready (DTR), and data set ready (DSR).

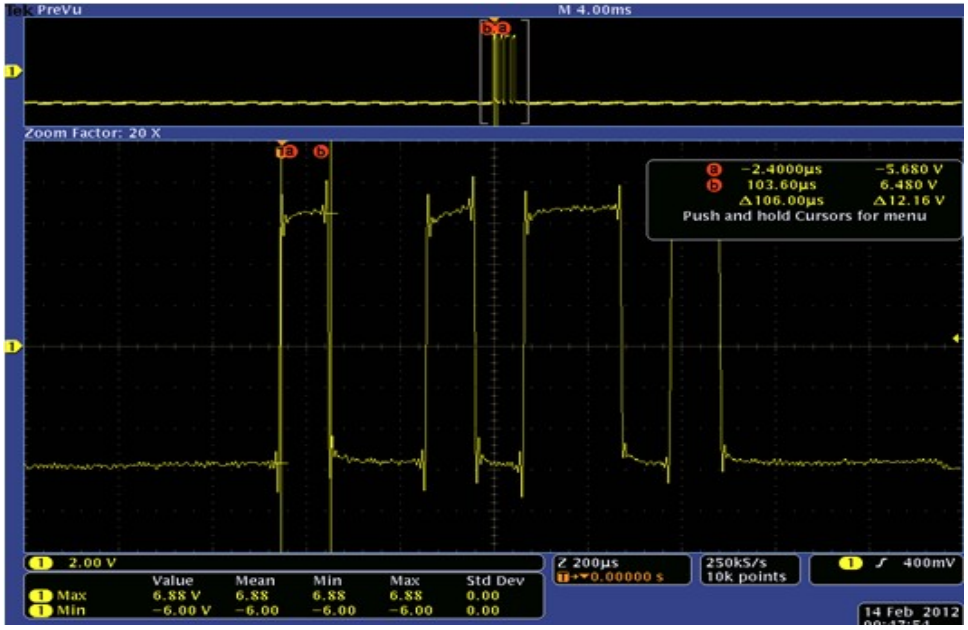


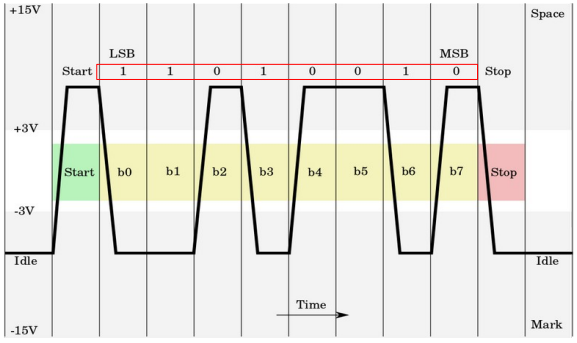
https://en.wikipedia.org/wiki/RS-232

RS232 (serial communication data transmission)

ASCII	Char	Hex	Bin	ASCII	Char	Hex	Bin
65	A	41	0100 0001	97	a	61	0110 0001
66	B	42	0100 0010	98	b	62	0110 0010
67	C	43	0100 0011	99	c	63	0110 0011
68	D	44	0100 0100	100	d	64	0110 0100
69	E	45	0100 0101	101	e	65	0110 0101
70	F	46	0100 0110	102	f	66	0110 0110
71	G	47	0100 0111	103	g	67	0110 0111
72	H	48	0100 1000	104	h	68	0110 1000
73	I	49	0100 1001	105	i	69	0110 1001
74	J	4A	0100 1010	106	j	6A	0110 1010
75	K	4B	0100 1011	107	k	6B	0110 1011
76	L	4C	0100 1100	108	l	6C	0110 1100
77	M	4D	0100 1101	109	m	6D	0110 1101
78	N	4E	0100 1110	110	n	6E	0110 1110
79	O	4F	0100 1111	111	o	6F	0110 1111
80	P	50	0101 0000	112	p	70	0111 0000
81	Q	51	0101 0001	113	q	71	0111 0001
82	R	52	0101 0010	114	r	72	0111 0010
83	S	53	0101 0011	115	s	73	0111 0011
84	T	54	0101 0100	116	t	74	0111 0100
85	U	55	0101 0101	117	u	75	0111 0101
86	V	56	0101 0110	118	v	76	0111 0110
87	W	57	0101 0111	119	w	77	0111 0111
88	X	58	0101 1000	120	x	78	0111 1000
89	Y	59	0101 1001	121	y	79	0111 1001
90	Z	5A	0101 1010	122	z	7A	0111 1010

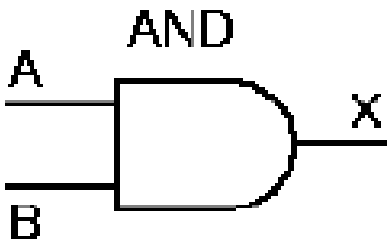
<linuxhint/>



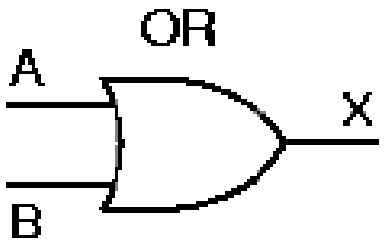


https://en.wikipedia.org/wiki/RS-232

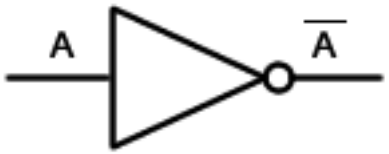
Universal Basis of **all** Digital Hardware: Logic Gates



A	B	X
0	0	0
0	1	0
1	0	0
1	1	1



A	B	X
0	0	0
0	1	1
1	0	1
1	1	1



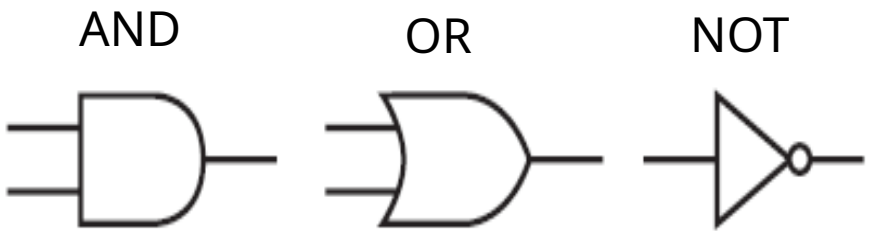
(a)

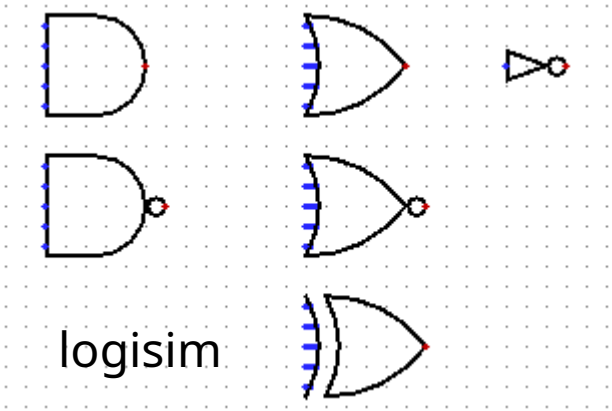
A	$\overline{A}$
0	1
1	0

Different commonly used notations

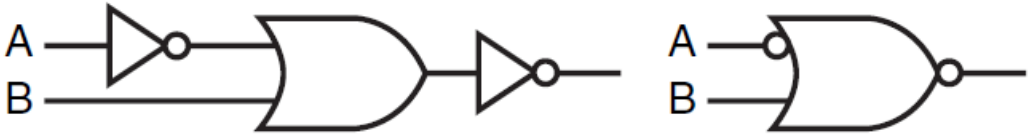
A AND B	$A \cdot B$	$A \wedge B$	&&	&&	&&	and
A OR B	$A + B$	$A \vee B$	||	||	||	or
NOT A	$\overline{A}$	$\neg A$	!	!	!	not
			C/C++	JAVA	JAVASCRIPT	PYTHON
		MATH. LOGIC		PROG. LANG.		

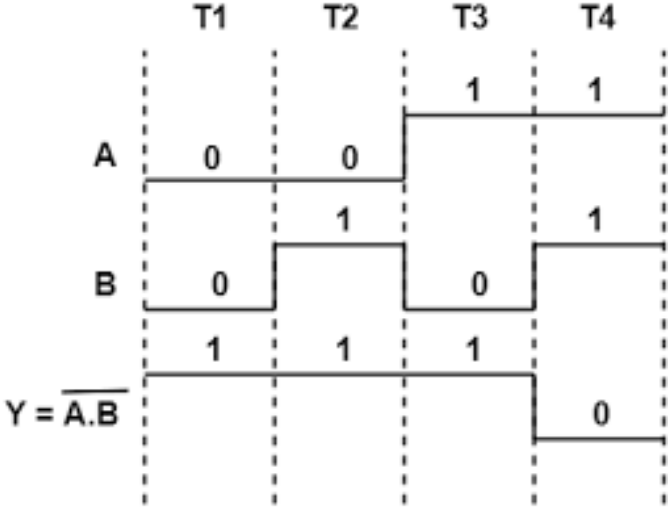
Logic Gates





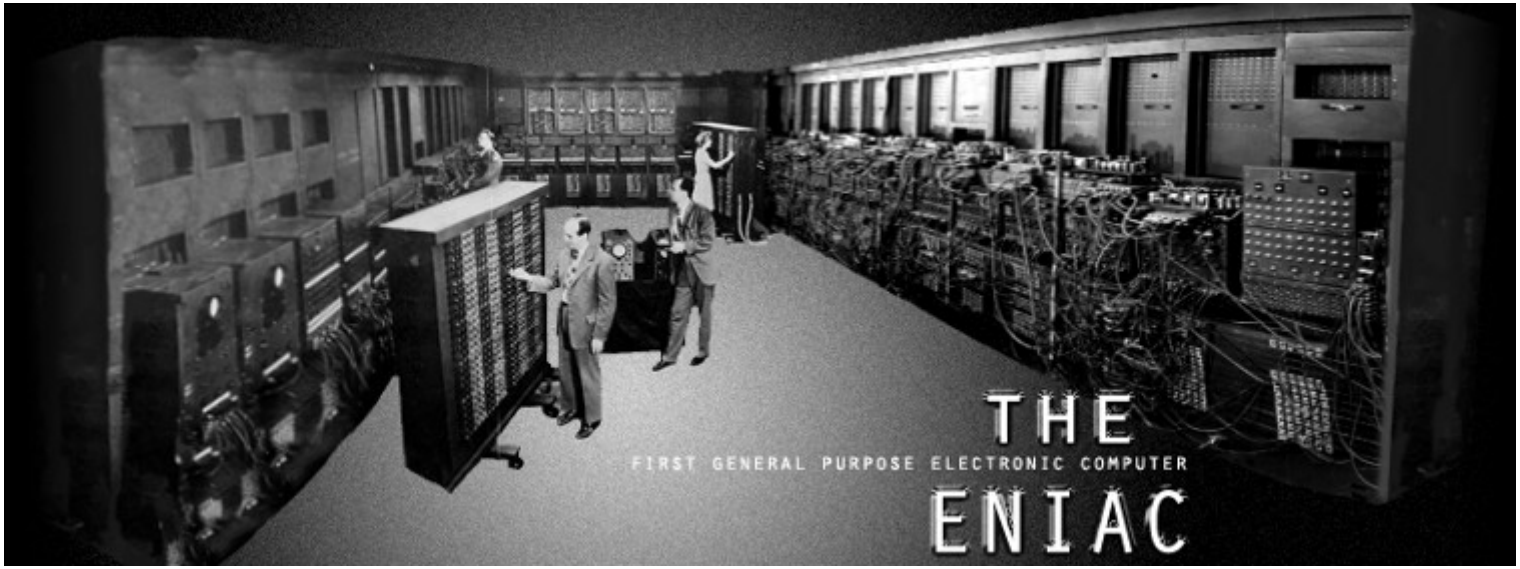
Shorthand notation for combinations including NOT





Note: operate on digital time-varying **signals!**

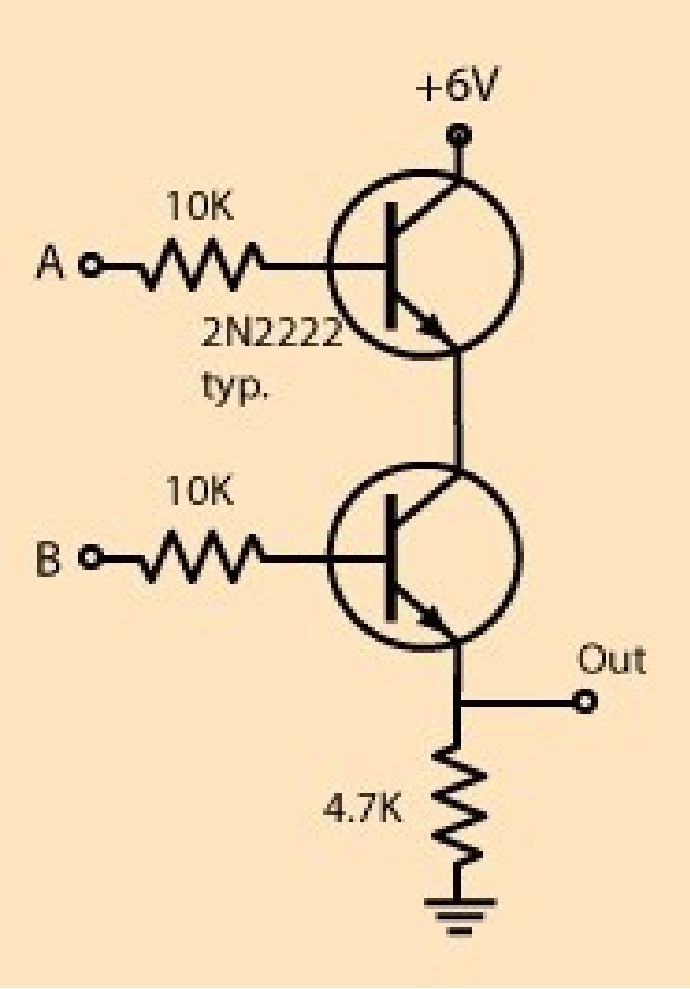
Implementing **Digital** (Logic) Components using **Analog** Electrical Components



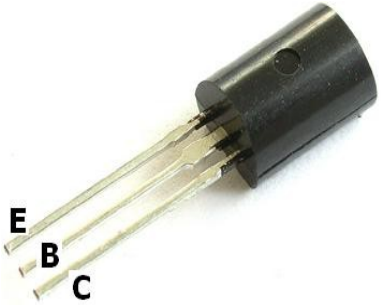
"**E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer"

Implementing **Digital** (Logic) Components using **Analog** Electrical Components

AND



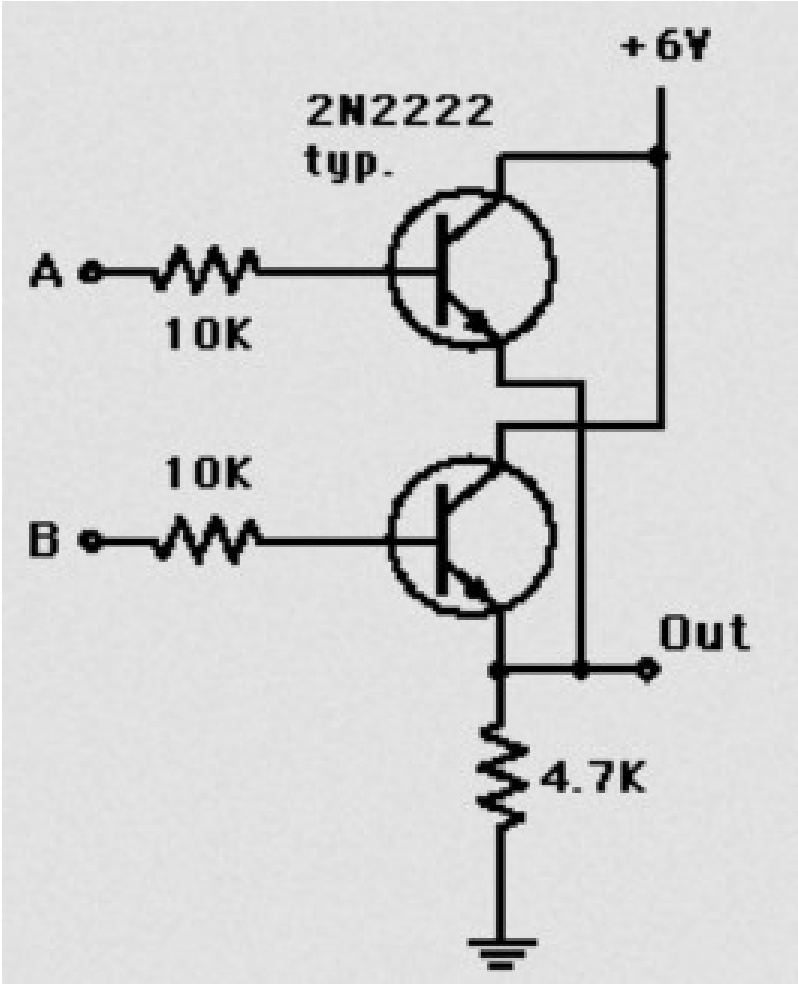
resistor



transistor

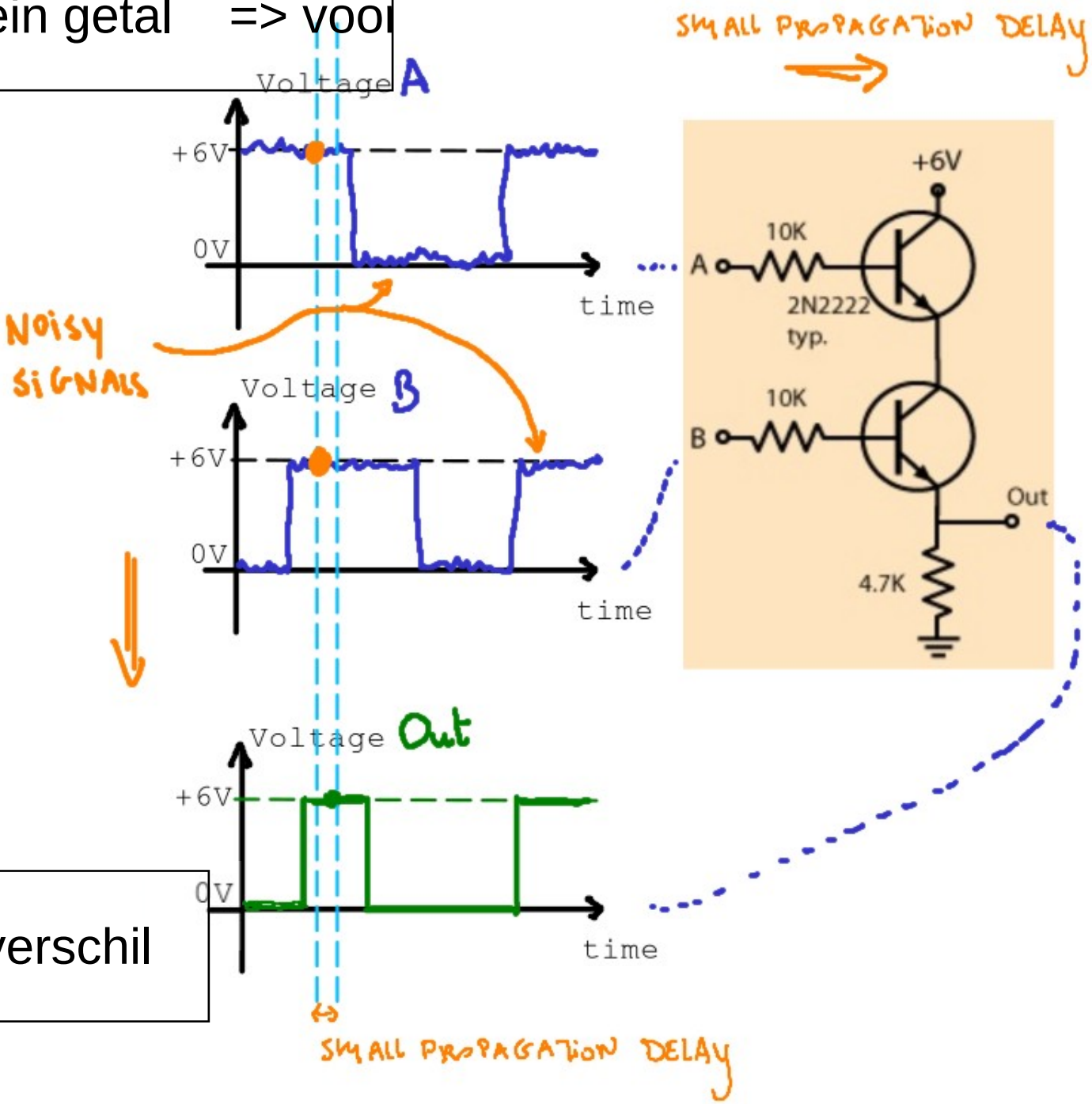
Used to amplify electrical signals.
When used “in saturation” acts as a switch.

OR



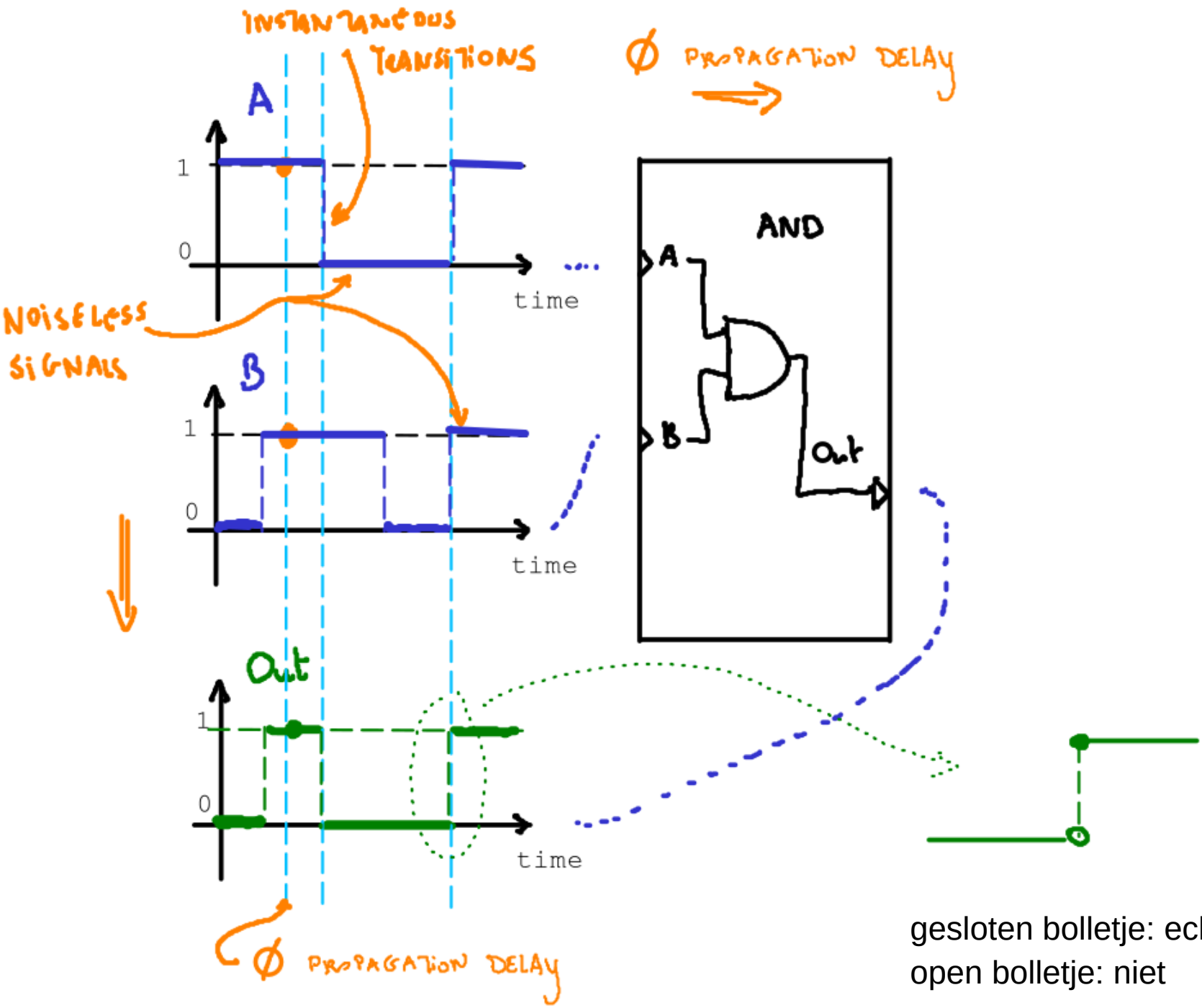
physical (analog) view

= zeer klein getal => voor



= grootte verschil

logical (digital) view (idealized)



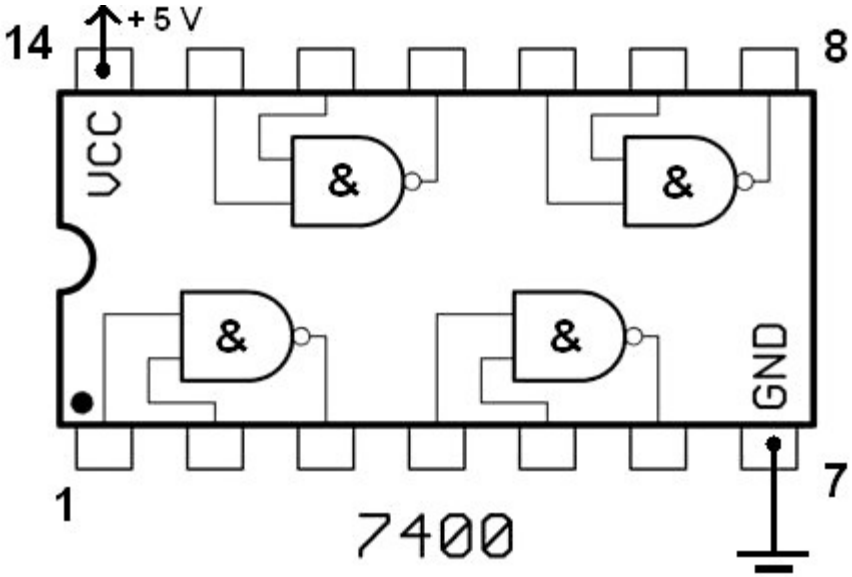
abstraction:
should still be
a function
(unique value)

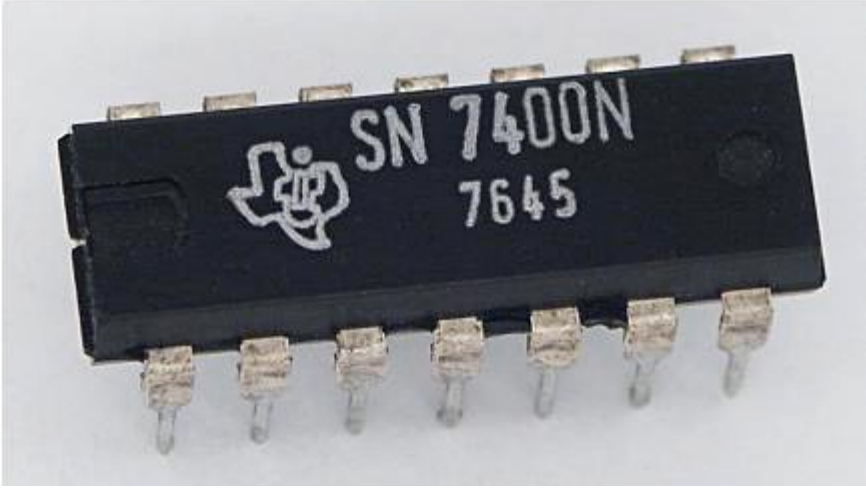
gesloten bolletje: echte waarde
open bolletje: niet

Implementing Logic Components:

SN 7400N with 4 NAND gates (~ 8 transistors)

jaren '70



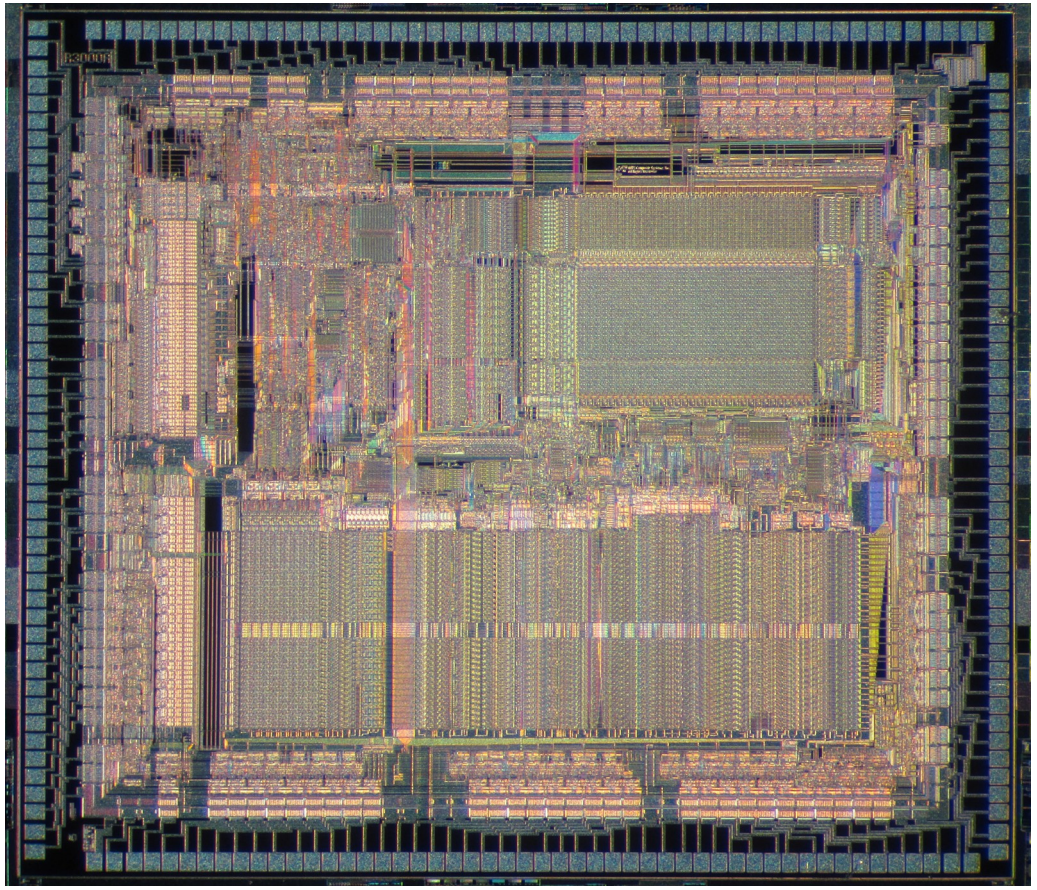


manufactured in the 45th week of 1976

Implementing Logic Components:

jaren '90

32 bit MIPS R3000 processor (115000 transistors)



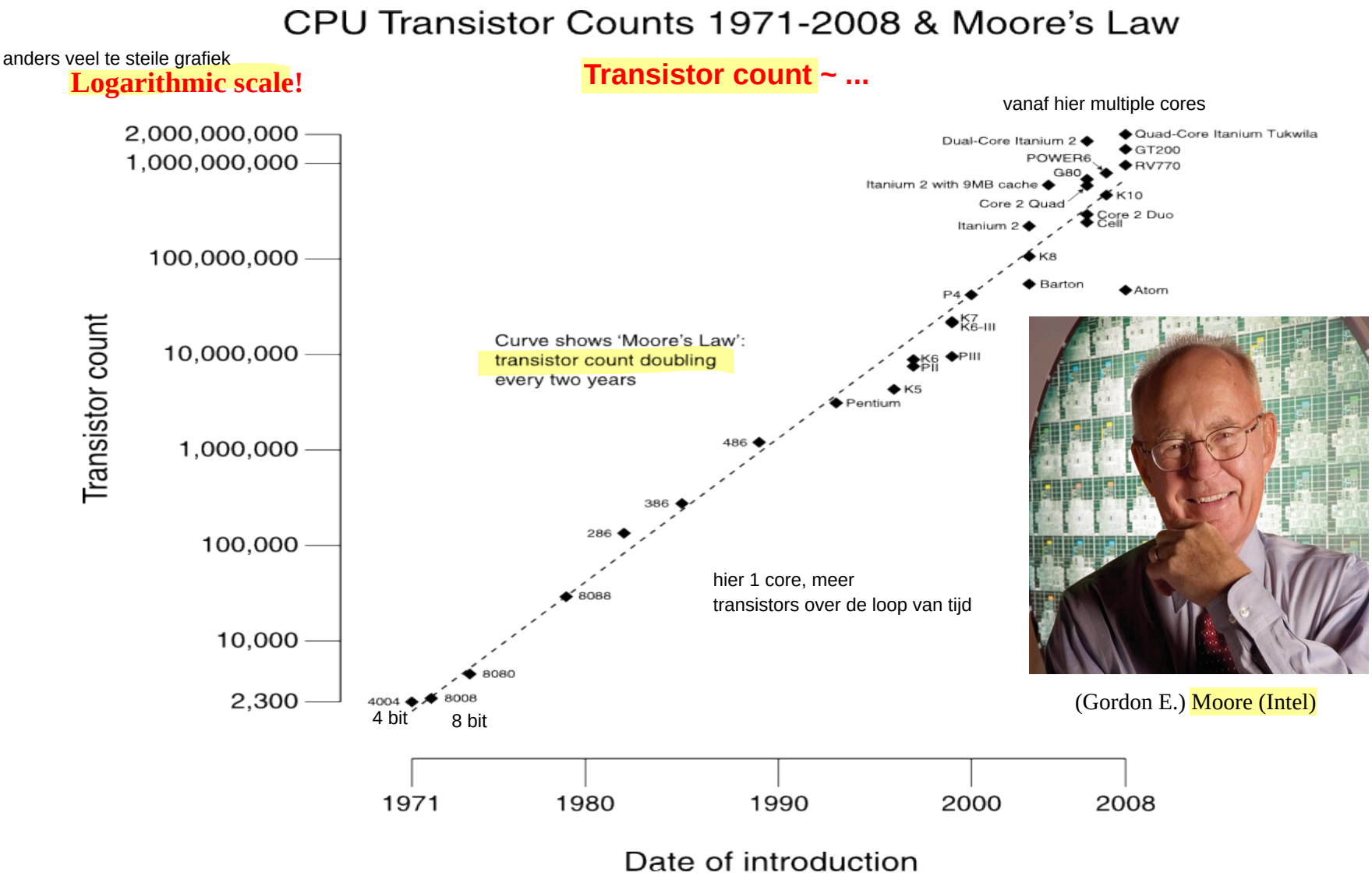
early 1990s

ééns te veel transistors ==> meerdere cores ==> parallel werken

https://www.mips.com/blog/five-most-iconic-devices-to-use-mips-cpus/

MIPS
by Imagination

transistor count ~ computing/memory power
→ **exponential growth** in computing/memory power



hij zei: voor dez prijs & zelfde grootte ==> 2x zoveel transistors

Types of circuits:

- "**combinational**" logic/circuit:
  implements a **function** (input → output)

- "**sequential**" logic/circuit: implements **memory**

A)

f(a1,a2)
= a1 AND a2 === voor alle a1,a2 als element van {TRUE,FALSE}

f(TRUE, TRUE) = TRUE

B)

a1	a2	f(a1,a2)
FALSE	FALSE	FFFT
FALSE	TRUE	
TRUE	FALSE	
TRUE	TRUE	

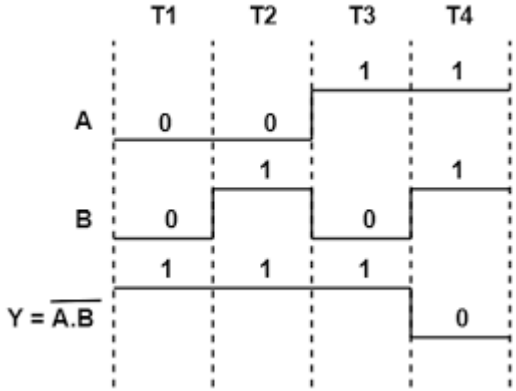
Specifying (modelling) combinational logic:

"truth table"

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Implements a "total" function →
how many table rows for N inputs?

N inputs --> #COMBINATIES = 2^N



Note: these are (digital) "**signals**" !

Specifying (modelling) combinational logic:

Logic (Boolean) formula:

(out1, out2) = f(in1, in2, in3) where

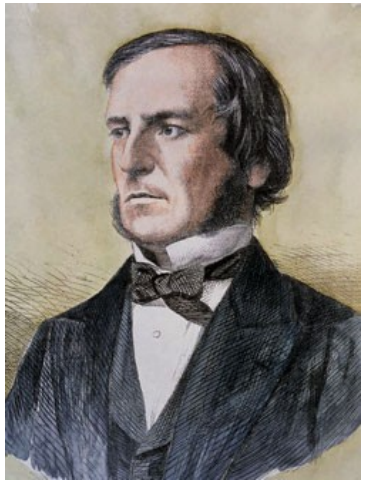
- out1 = (in1 AND in2) OR in3
- out2 = NOT (in2 OR in3)

Derived truth table representation of f:

in1	in2	in3	out1	out2
0	0	0	0	1
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

Boolean algebra: AND (.), OR (+), NOT ($^{-}$)

- Identity law: $A + 0 = A$ and $A \cdot 1 = A$.
- Zero and One laws: $A + 1 = 1$ and $A \cdot 0 = 0$.
- Inverse laws: $A + \overline{A} = 1$ and $A \cdot \overline{A} = 0$
- Commutative laws: $A + B = B + A$ and $A \cdot B = B \cdot A$.
- Associative laws: $A + (B + C) = (A + B) + C$ and $A \cdot (B \cdot C) = (A \cdot B) \cdot C$.
- Distributive laws: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ and $A + (B \cdot C) = (A + B) \cdot (A + C)$.



George Boole
(1815 - 1864)

De Morgan's laws:

- $\overline{A + B} = \overline{A} \,.\, \overline{B}$
- $\overline{A \,.\, B} = \overline{A} + \overline{B}$

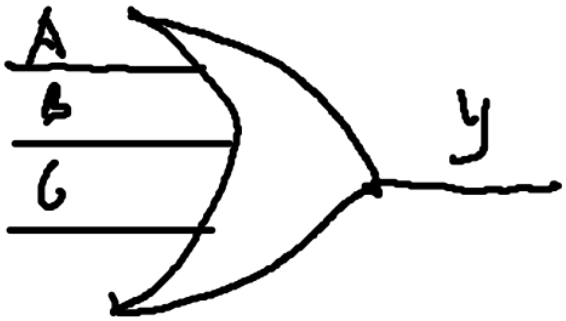
Augustus De Morgan
(1806-1871)

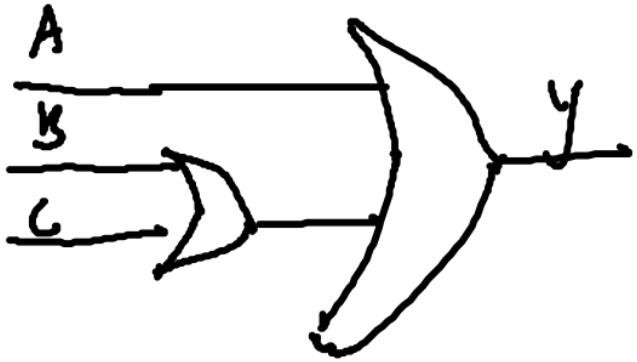
Associativity

aanneemen maar 2 transistors in OR gate
want 1 gate = *2 transistors

VRAAG EX:
#GATES & DELAY

$$y = A + B + C = A + (B + C) = (A + B) + C$$





“truth table”

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Compact representation as Logic Formulae?

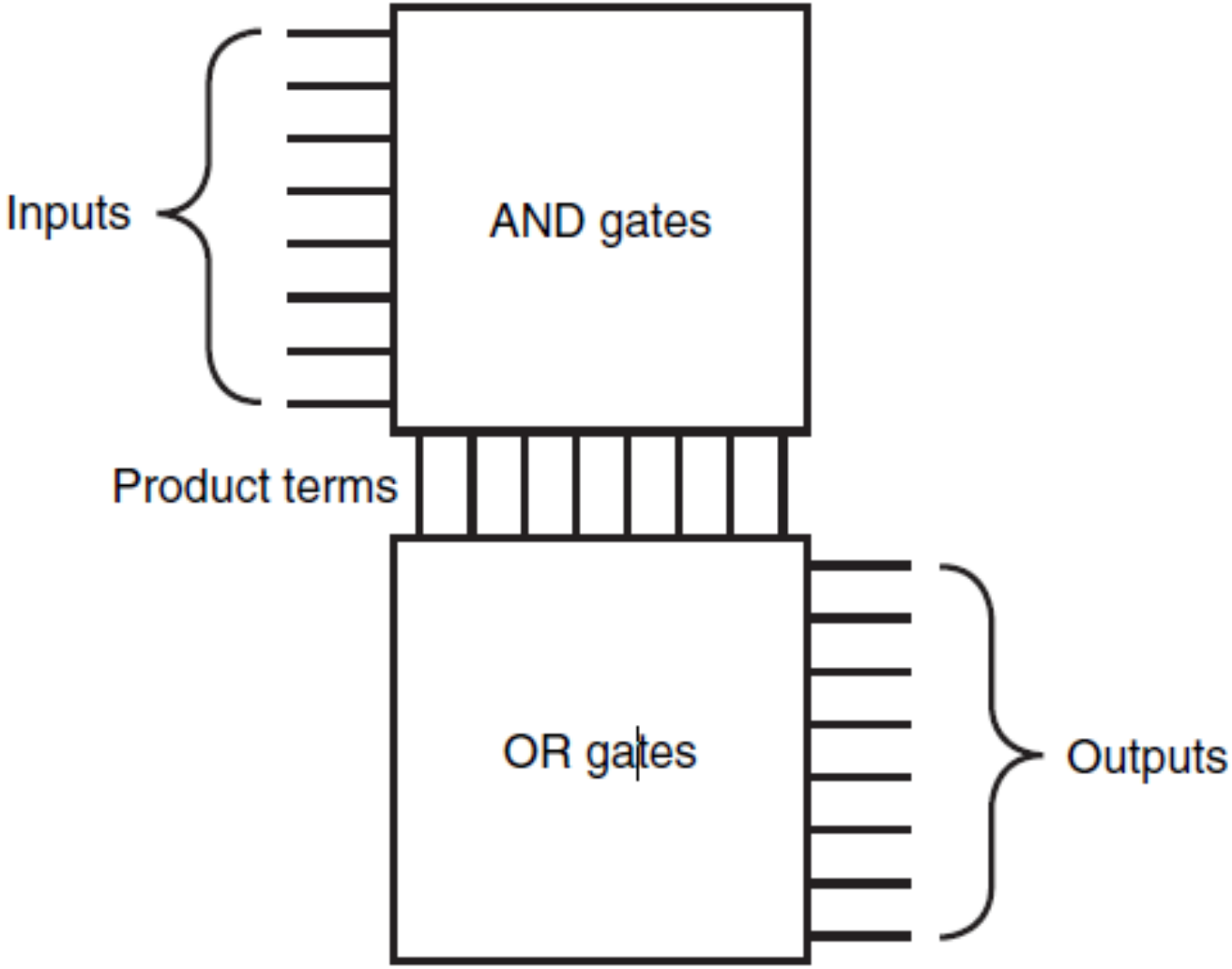
“Sum of Products” (1 outputs) or
“Product of Sums” (0 outputs)
(see lab sessions)

Sum of Products

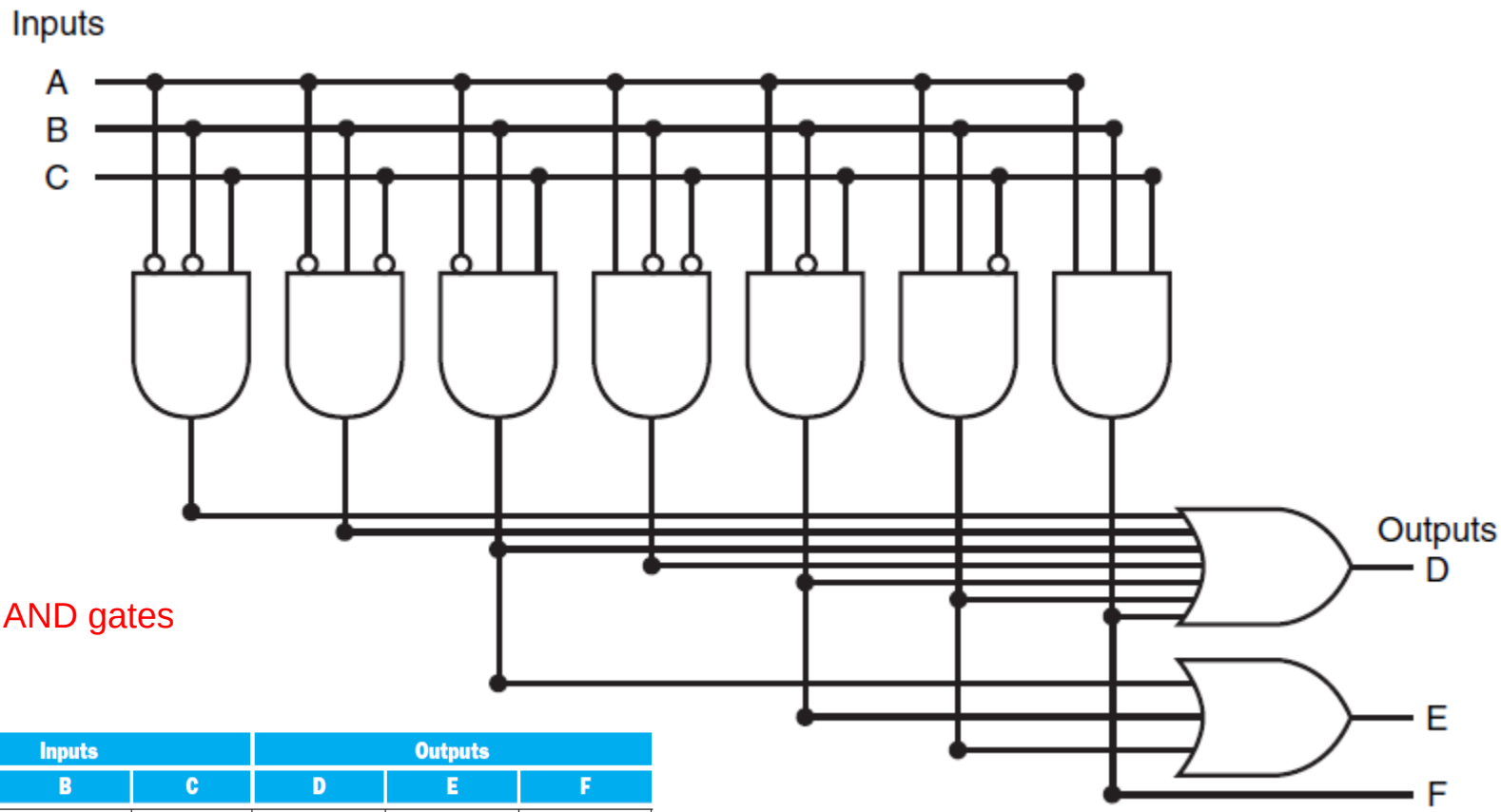
Inputs			Output
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$D = (\overline{A} \cdot \overline{B} \cdot C) + (\overline{A} \cdot B \cdot \overline{C}) + (A \cdot \overline{B} \cdot \overline{C}) + (A \cdot B \cdot C)$$

Programmable Logic Array (PLA)



Programmable Logic Array (PLA)

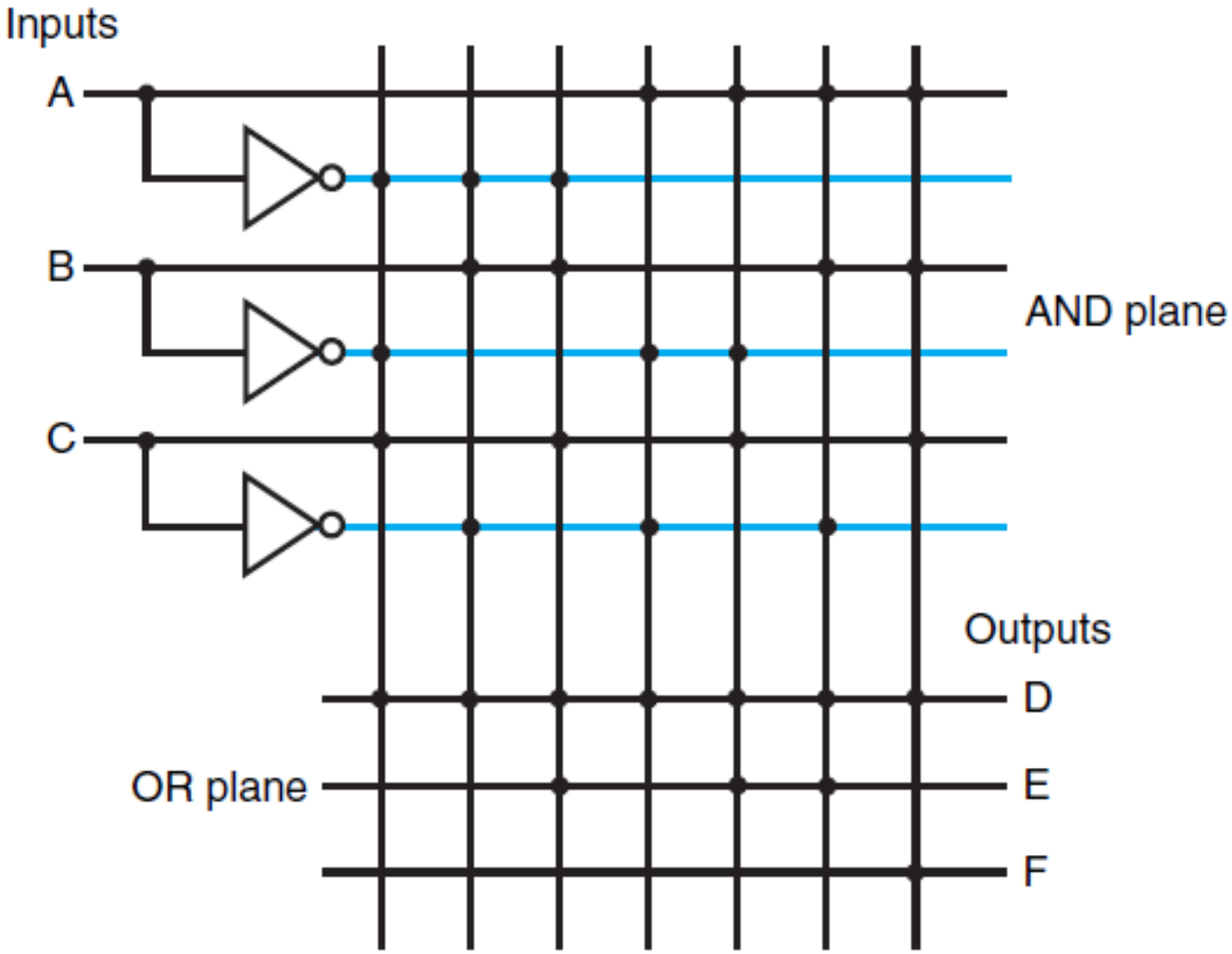


ergste geval: 2^3 AND gates

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

how many logic gates?
how long does it take?

Programmable Logic Array (PLA)



how many logic gates?
how long does it take?

worst case:

how many logic gates?
how long does it take?

i : # INPUTS

o : # OUTPUTS

$$\# \text{ GATES} = 2^i + o$$

(OR, AND)

$$\text{DELAY} = \ell \, \varepsilon$$ (ε IS SINGLE GATE DELAY/LATENCY)

want parallel

Binary representation/encoding of Unsigned Integers

n-bit string "$x_{n-1}x_{n-2}\ldots x_1x_0$" has/encodes **value** x

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

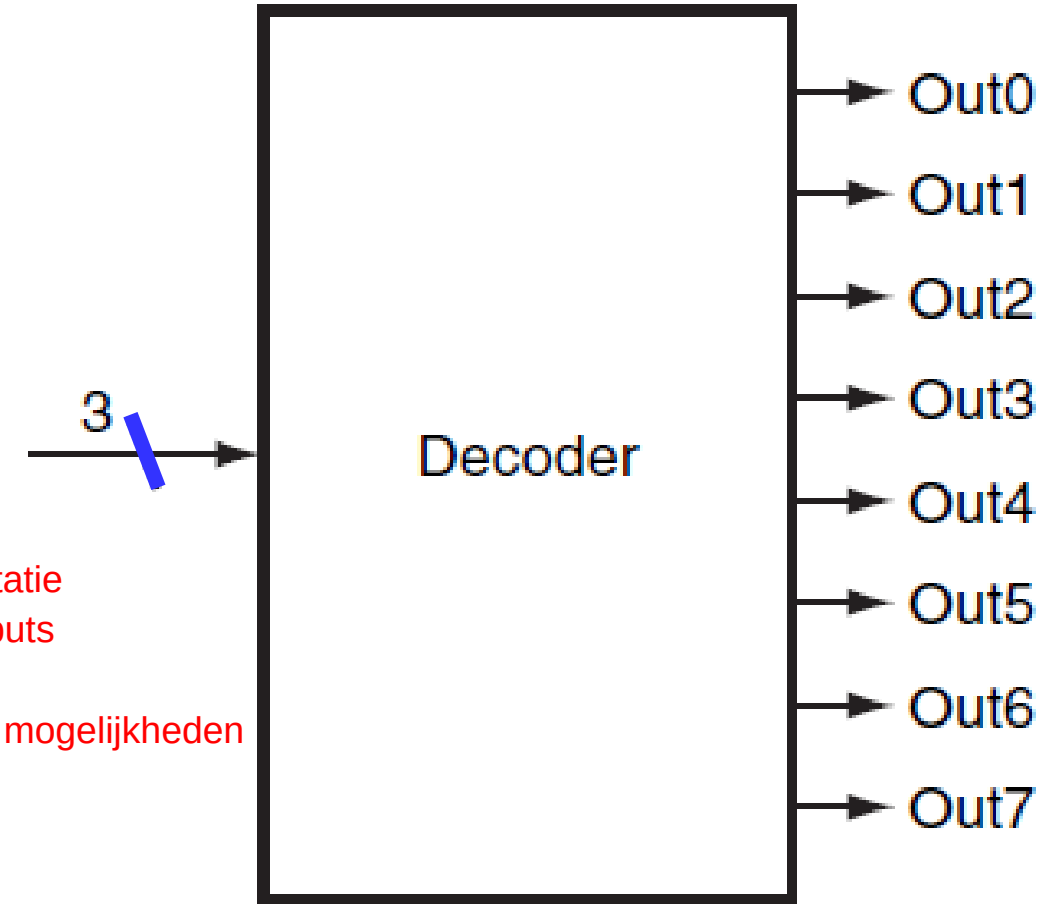
x_0 Least Significant Bit (**LSB**)

x_{n-1} Most Significant Bit (**MSB**)

binair - decimaal (tutorial opzoeken)

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

Decoder (n bits to 2^n outputs)



1 => 100 input

BUS notatie
eig 3 inputs

dus 2^3 mogelijkheden

Note: (3 bit wide) signal "bus"

how many logic gates?
how long does it take?

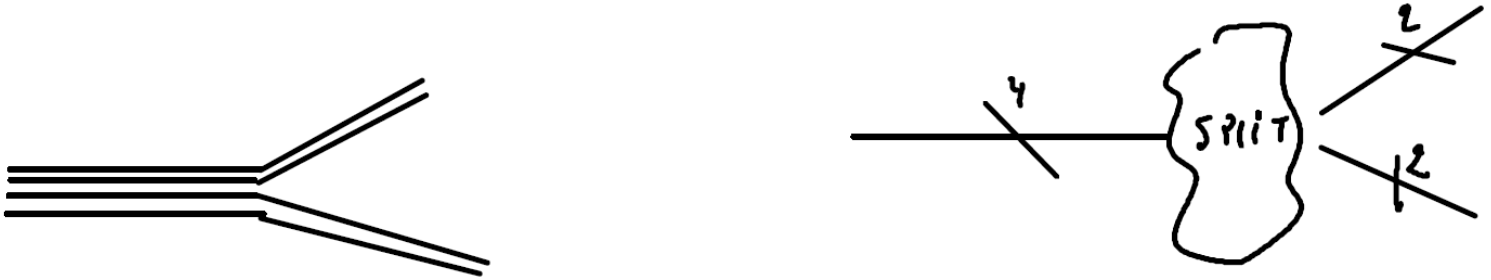
vraag: hoe krijg je 1 bij out 4, welke inputs? (antw: 1 0 0) || hoe lang duurt het om alle 8 de outputs te krijgen (antw: 2 epsi

Decoder (truth table)

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Use “Sum of Products” or “Product of Sums”

split vs. decoder

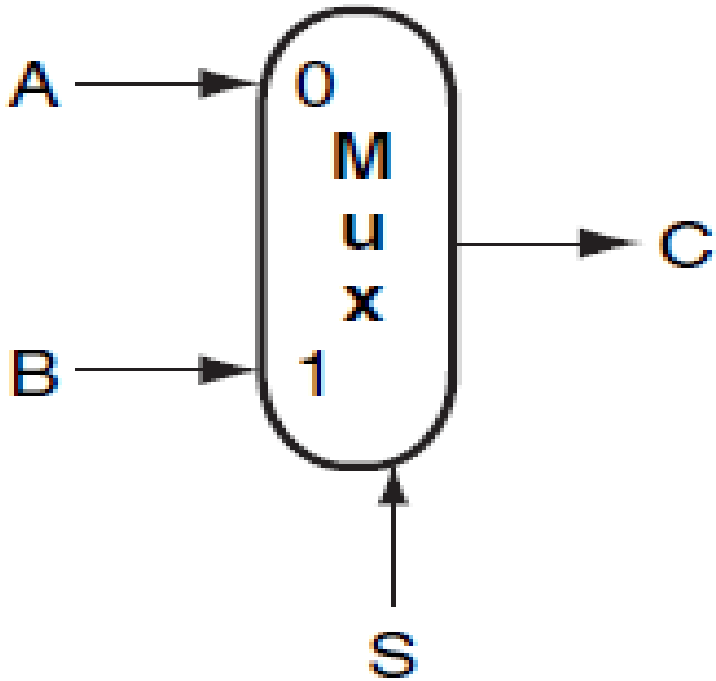


4 bit → $2^4 = 16$ mogelijke bit combinaties

0	0	0	
0	0	1	...
0	1	0	
0	1	1	

Multiplexor (1 bit)

selector: bij 0 --> A MAG DOOR
bij 1 --> B MAG DOOR



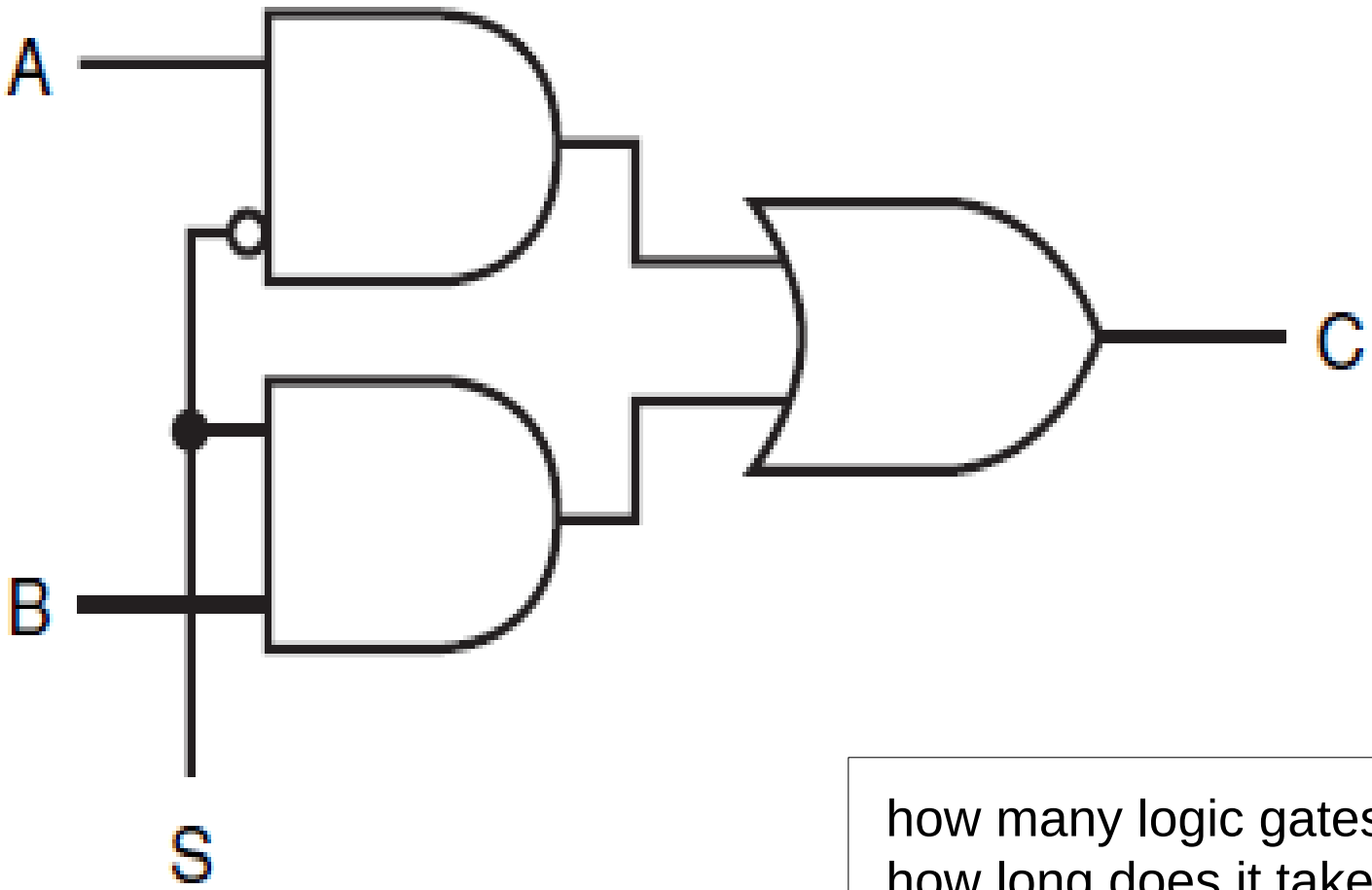
selector
kiest welke door mag

A	B	S	C
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

- **S**election
- From **parallel** to **serial**
- Operates on digital **signals**!

how many logic gates?
how long does it take?

Multiplexor (1 bit) implementation

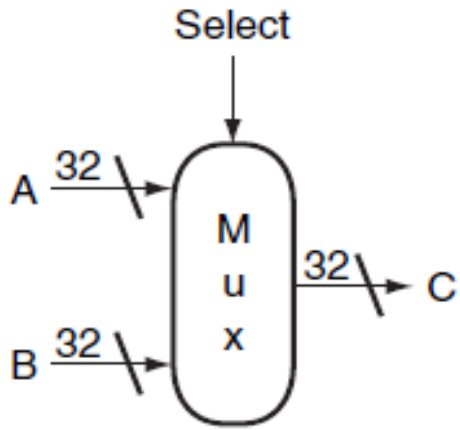


how many logic gates?
how long does it take?

Multiplexor (32 bit), “bus”

#i = 65
o = 32
#GATES = 3
DELAY = 2 epsilon

3*32 gates

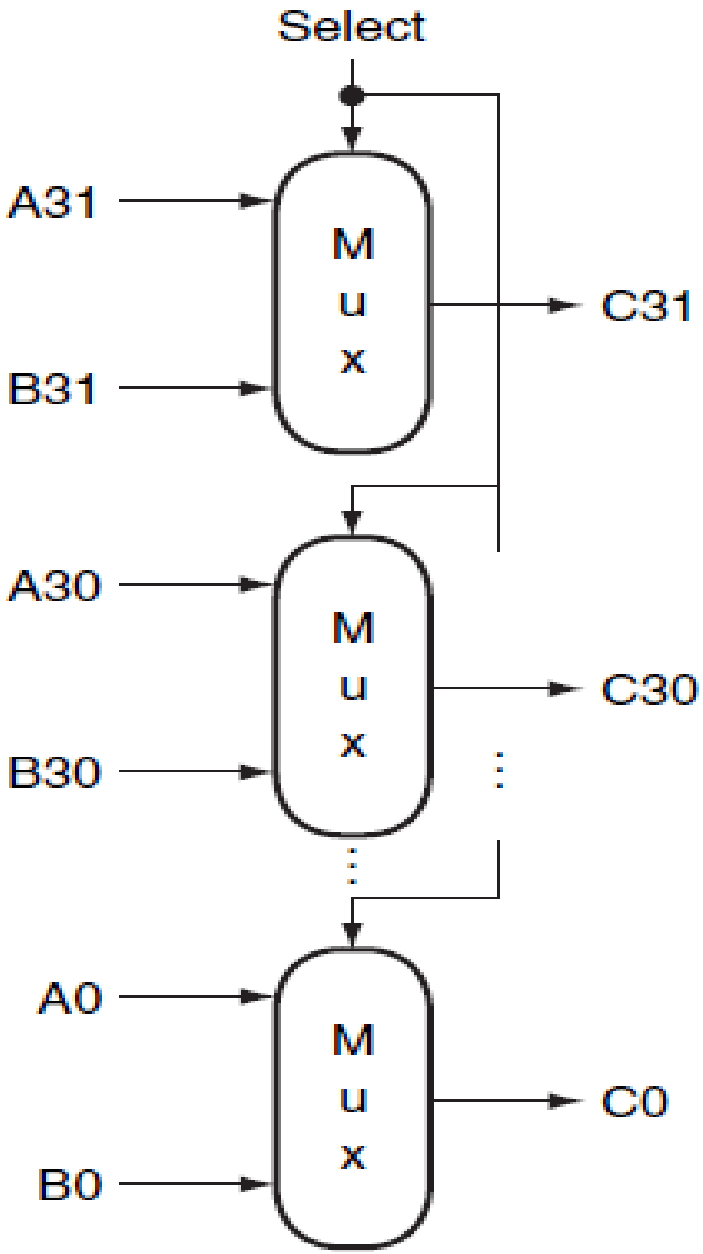


32 1 bit multiplexors, 1 selector

dus 64 + 1 bits

how many logic gates?
how long does it take?

Multiplexor (32 bit) implementation



how many logic gates?
how long does it take?

#waarden met 2 bits

00
01
10
11

#waarden n bits
2^n

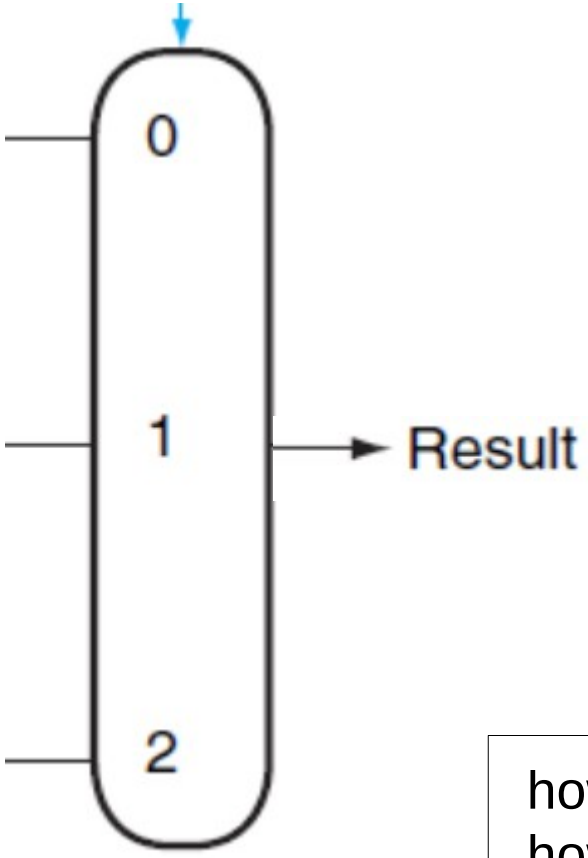
K verschillende waarden (in dit geval 3)
bits nodig = k
==> log2(K)

log2(2^N) = N

ALTIJD NAAR BOVEN AFRONDEN!
want 1 bit voor 1,001 is niet genoeg

==> CEIL(log2(K))
(omgekeerde is FLOOR)

#bit?

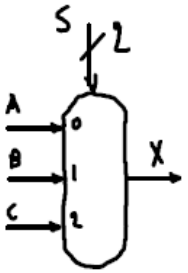


5 input
1 output

(5^3 + 1) GATES

how many logic gates?
how long does it take?

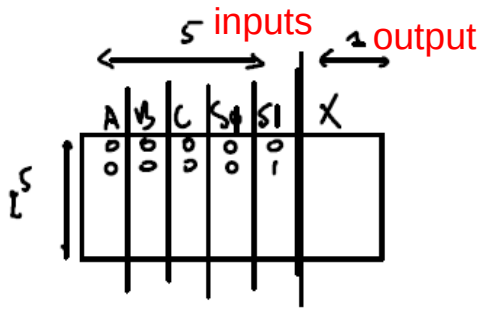
sum of products ... “clever”
(I, II, III, see scribbles)



S_1	S_0	
0	0	A
0	1	B
1	0	C
~~1~~	~~1~~	

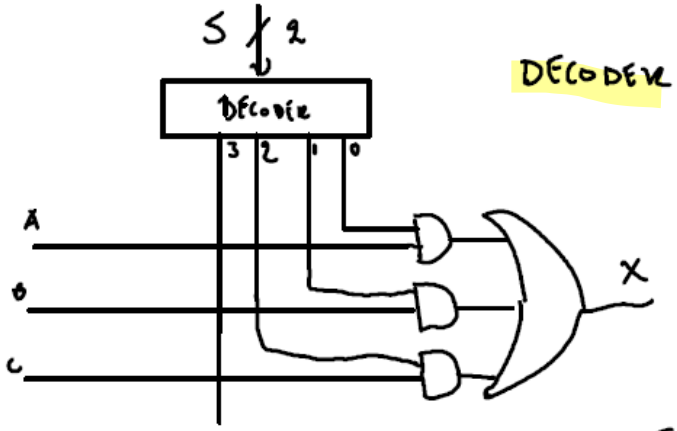
$\Delta t = ?$ $\#G = ?$

I



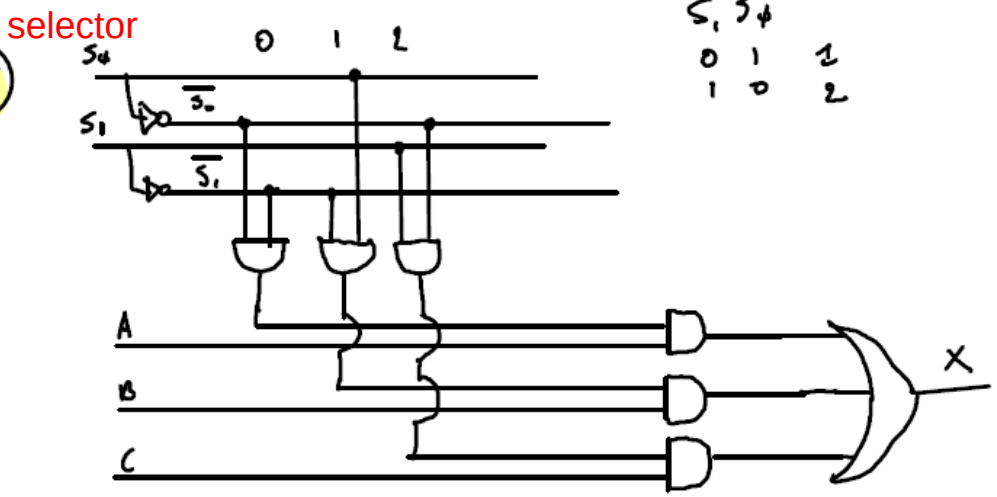
SUM OF PRODUCT

II



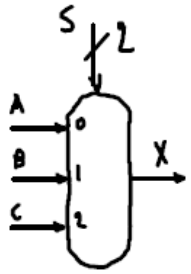
DECODER

III



S_1	S_0	
0	1	1
1	0	2

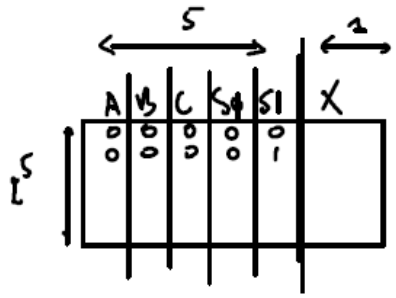
7 gates
==> 3 epsilon



S_1	S_0	
0	0	A
0	1	B
1	0	C
~~1~~	~~1~~	

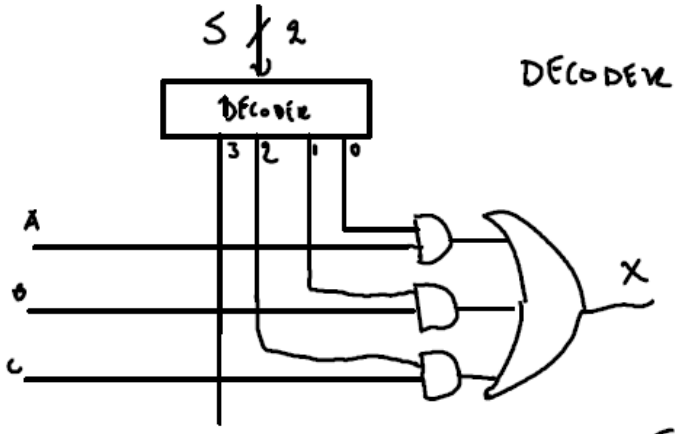
$\Delta t = ?$ $\#G = ?$

I



SUM OF PRODUCT $\Delta t = 2\varepsilon$ $\#G = 2^5 + 1 = 33$

II



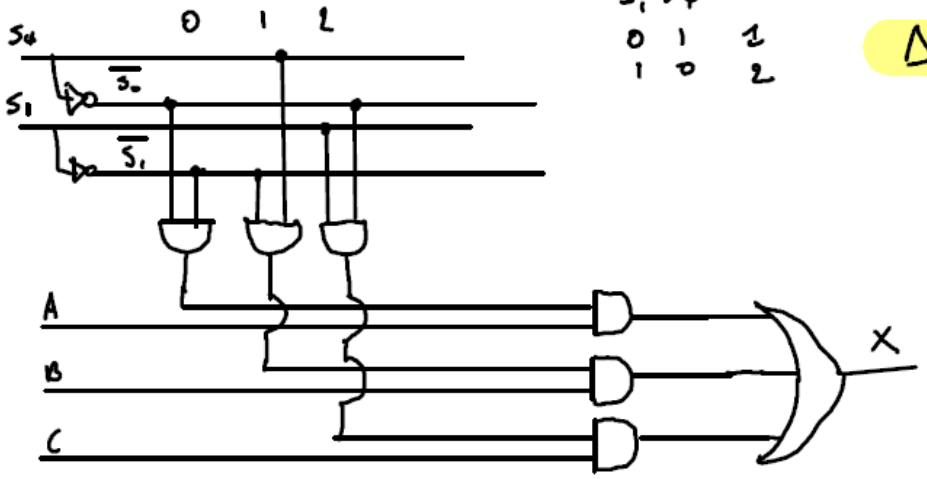
DECODER

NAIVE : SUM OF PRODUCTS

$\Delta t = \Delta t_{DECODER} + 2\varepsilon = 2\varepsilon + 2\varepsilon = 4\varepsilon$

$\#G = \#G_{DECODER} + 4 = (2^2 + 4) + 4 = 12$

III



S_1	S_0	
0	1	1
1	0	2

$\Delta t = 3\varepsilon$ $\#G = 7$

Arithmetic and Logic Unit (ALU)

R-Type Instructions

funct	meaning
0	or: $rd = $rd | $rx
1	xor: $rd = $rd ^ $rx
2	and: $rd = $rd & $rx
3	add: $rd = $rd + $rx
4	srl: $rd = $rx >> 1
5	sra: $rd = $rx / 2
6	not: $rd = ~$rx
7	neg: $rd = -1*$rx

shift right logical

shift right arithmetic

1 0 1 1 ==> _ 1 0 1

--> SHIFT RIGHT:
- logical: meest rechste verdwijnt, meest linke wordt 0
- arithmmetic: meest linkse copied 2de

unsigned
DECODE(1011) = 1+2+0+8 = 11 (basis 2 -> 10)
ENCODE(11) = 1011 (basis 10 -> 2)

1 vooraan zorgt voor negatief getal, als er al een 1 stond, dan willen we het getal ook negatief houden

signed
DECODE(1011) =
ENCODE(1011) =

dus 1111 kan je interpreteren als unsigned (15) en signed (-1) (geheel getal)
PS: 4 bits ==> 2^4 values

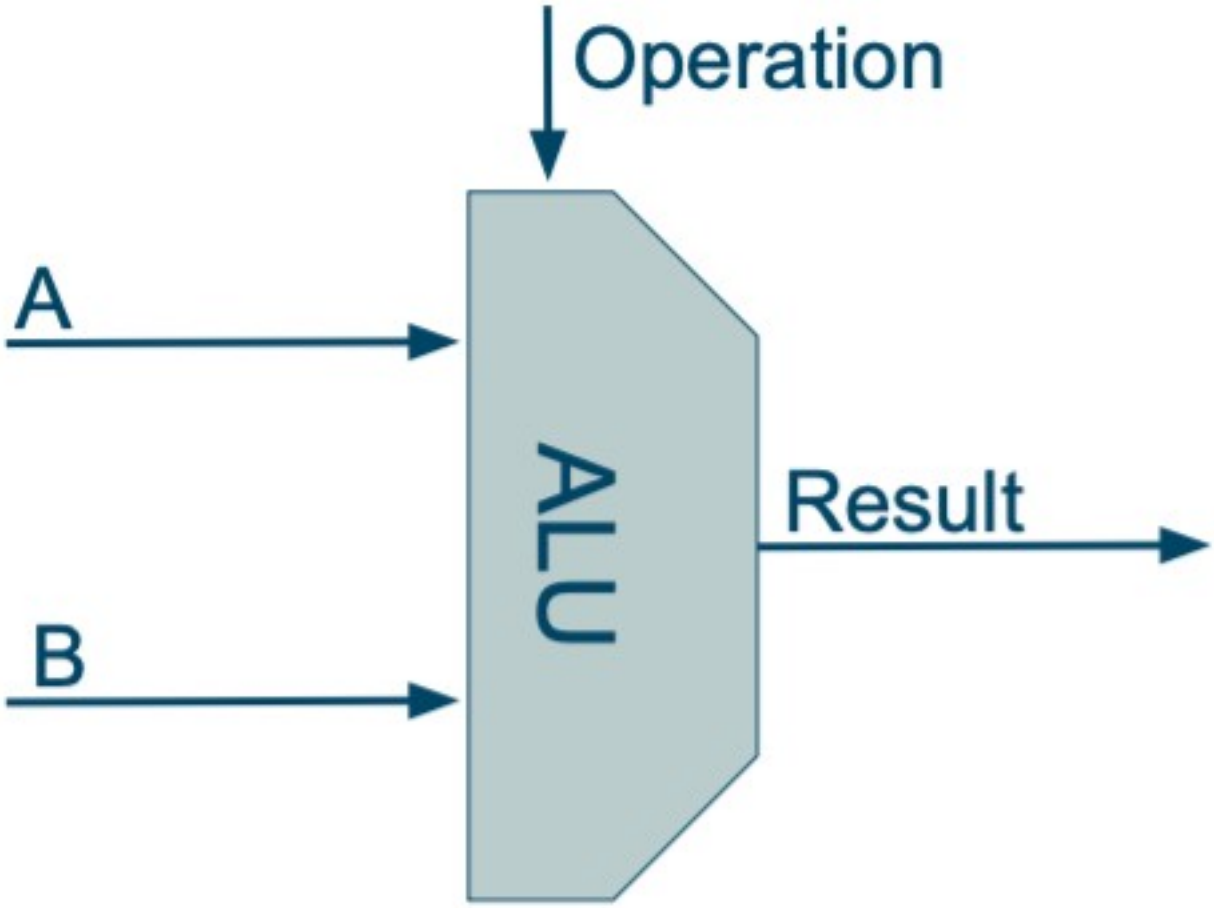
bij shift left arithmetic ==> *2

G = 2 * D + R
D = G DIV 2
K = G MOD 2

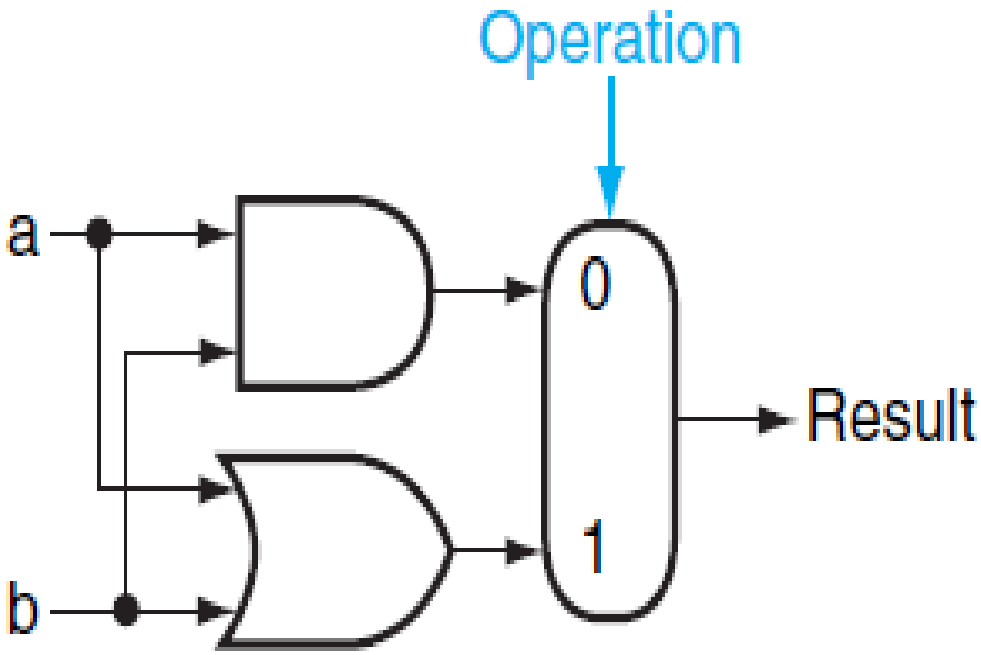
Note: instruction (e.g., not vs. neg) determines data type (e.g., Boolean vs. Int)

5 DIV 2 = 2 (hoeveel keer 2 in 5)
5 MOD 2 = 1 (hoeveel rest bij optimale 5/2)

Arithmetic and Logic Unit (ALU)



1-bit AND, OR



== mux == 3 gates

how many logic gates? 5 gates
how long does it take? 3 epsilon

want 3 keer gate doorgang

Binary representation of information

- in computing and telecommunications
- a **bit** is a basic *unit of information storage*
- “**bi**nary digi**t**”
- the maximum amount of information
- that can be stored in only two distinct states

0 or 1

Bit sequences
(bit) “string” (vs. char string)

0	1 bit	**bit**
0111	4 bits	**nibble**
01100001	8 bits	**byte**
0101010110110111	16 bits	**half-word**
...	32 bits	**word**
...	64 bits	**word**

A **word** is a natural unit of data used by a particular processor design (8, 16, 24, 32, or 64 bits)

Byte = Octet (in French)

Disque Dur Externe WD My Book New 8 To Noir

Disques durs externes - Western Digital ★★★★☆ (19)

PRIX ADHERENT

Voir le produit

Type de disque dur	**Disque Dur Externe**
Technologie de disque dur	**Disque Dur (HDD)**
Capacité de stockage (Go)	**8000 Go**

En stock en ligne
Livraison gratuite

En magasin
Choisir

Origin of the term “byte”

The term byte was coined by Werner Buchholz in June 1956, during the early design phase for the IBM Stretch computer, which had addressing to the bit and variable field length (VFL) instructions with a byte size encoded in the instruction. It is a deliberate respelling of **bite** to avoid accidental mutation to bit.

Binary representation ++

1 Bit	2 Bits	3 Bits	4 Bits	5 Bits
0	00	000	0000	00000
1	01	001	0001	00001
	10	010	0010	00010
	11	011	0011	00011
		100	0100	00100
		101	0101	00101
		110	0110	00110
		111	0111	00111
			1000	01000
			1001	01001
			1010	01010
			1011	01011
			1100	01100
			1101	01101
			1110	01110
			1111	01111
				10000
				10001
				10010
				10011
				10100
				10101
				10110
				10111
				11000
				11001
				11010
				11011
				11100
				11101
				11110
				11111

With 1 bit, can **represent/encode** 2 **distinct entities**

With 2 bits, can represent/encode 4 distinct entities

...

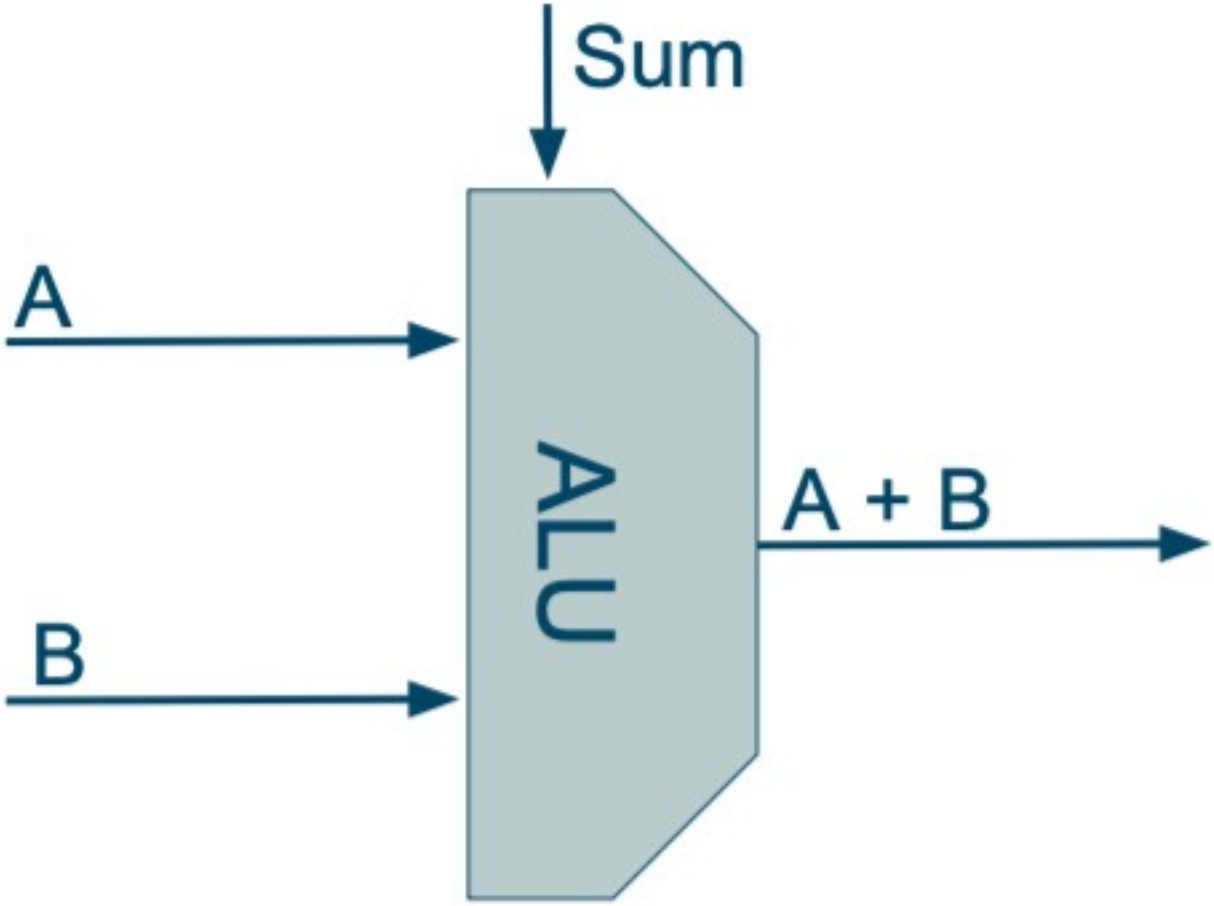
With **N** bits, can represent/encode $\mathbf{2^N}$ distinct entities

Example: `{Red, Green, Blue}`

encoded as `{00, 01, 10}`

code `11` not used

Arithmetic and Logic Unit (ALU)



Addition (decimal)

```
  81730
+ 19223
-------
```

Addition (decimal)

```
"carry"        0
          81730
        + 19223
        ------
              3
```

Addition (decimal)

"carry"

```
     00
  81730
+ 19223
-------
      3
```

Addition (decimal)

"carry"

```
   000
 81730
+ 19223
------
    53
```

Addition (decimal)

"carry" **0000**

```
  81730
+ 19223
-------
    953
```

Addition (decimal)

“carry”

```
  10000
  81730
+ 19223
-------
   0953
```

Addition (decimal)

"carry" **110000**

```
  81730
+ 19223
-------
  00953
```

Addition (decimal)

"carry" **110000**

```
  81730
+ 19223
-------
 100953
```

Binary Addition

```
  00101110
+ 00100111
----------
```

Binary Addition (8 bit)

```
  01011100
  00101110
+ 00100111
----------
  01010101
```

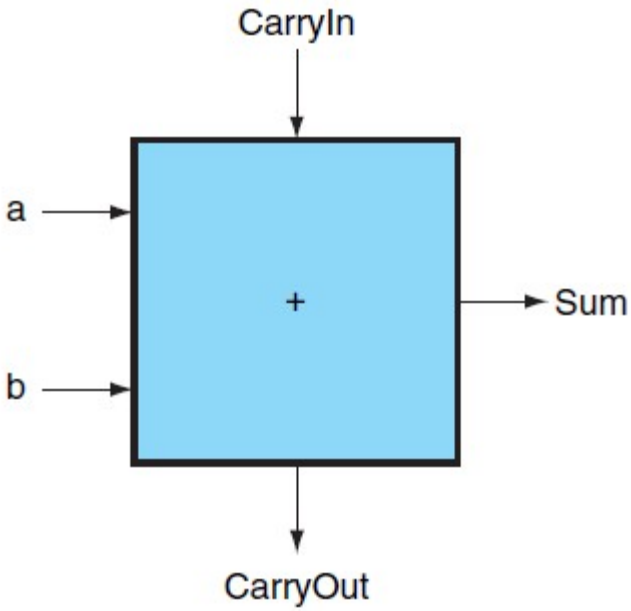
how many inputs?
how many outputs?

how many logic gates?
how long does it take?

Binary Addition (1 bit at a time)

```
  01011100
  00101110
+ 00100111
----------
  01010101
```

(zoek tutorial, nog niet volledig gesnapt)



1-bit adder

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

```
  01011100
  00101110
+ 00100111
----------
  01010101
```

1-bit adder

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

```
  01011100
  00101110
+ 00100111
----------
  01010101
```

1-bit adder

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

```
  01011100
  00101110
+ 00100111
----------
  01010101
```

1-bit adder

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

```
  01011100
  00101110
+ 00100111
----------
  01010101
```

1-bit adder

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Sum

Is 1 when *exactly one input is one* or when *all inputs are one*

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

Sum

Is 1 when *exactly one input is one* or when *all inputs are one*

$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

how many logic gates? 5
how long does it take? 2 epsilon

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

CarryOut

Is 1 when *at least two inputs are one*

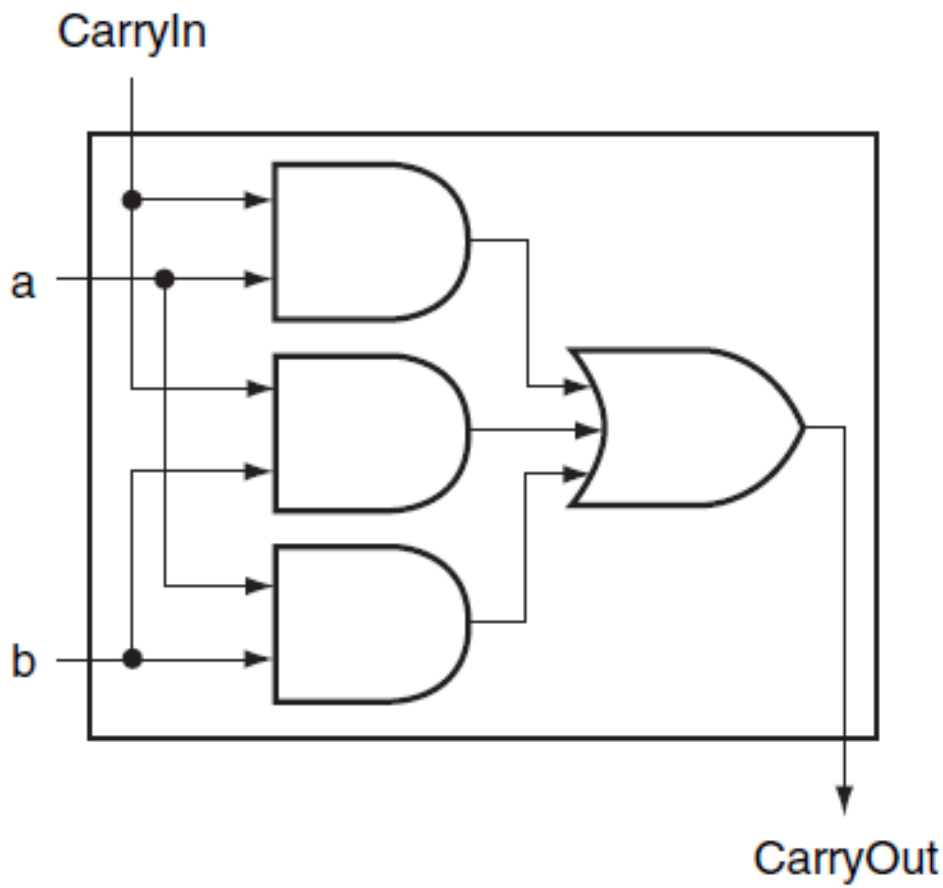
Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{two}$
0	0	1	0	1	$0 + 0 + 1 = 01_{two}$
0	1	0	0	1	$0 + 1 + 0 = 01_{two}$
0	1	1	1	0	$0 + 1 + 1 = 10_{two}$
1	0	0	0	1	$1 + 0 + 0 = 01_{two}$
1	0	1	1	0	$1 + 0 + 1 = 10_{two}$
1	1	0	1	0	$1 + 1 + 0 = 10_{two}$
1	1	1	1	1	$1 + 1 + 1 = 11_{two}$

CarryOut

Is 1 when *at least two inputs are one*

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) \quad + \ \ldots ?$$

1-bit Adder: Carry Out



CarryOut is 1 when

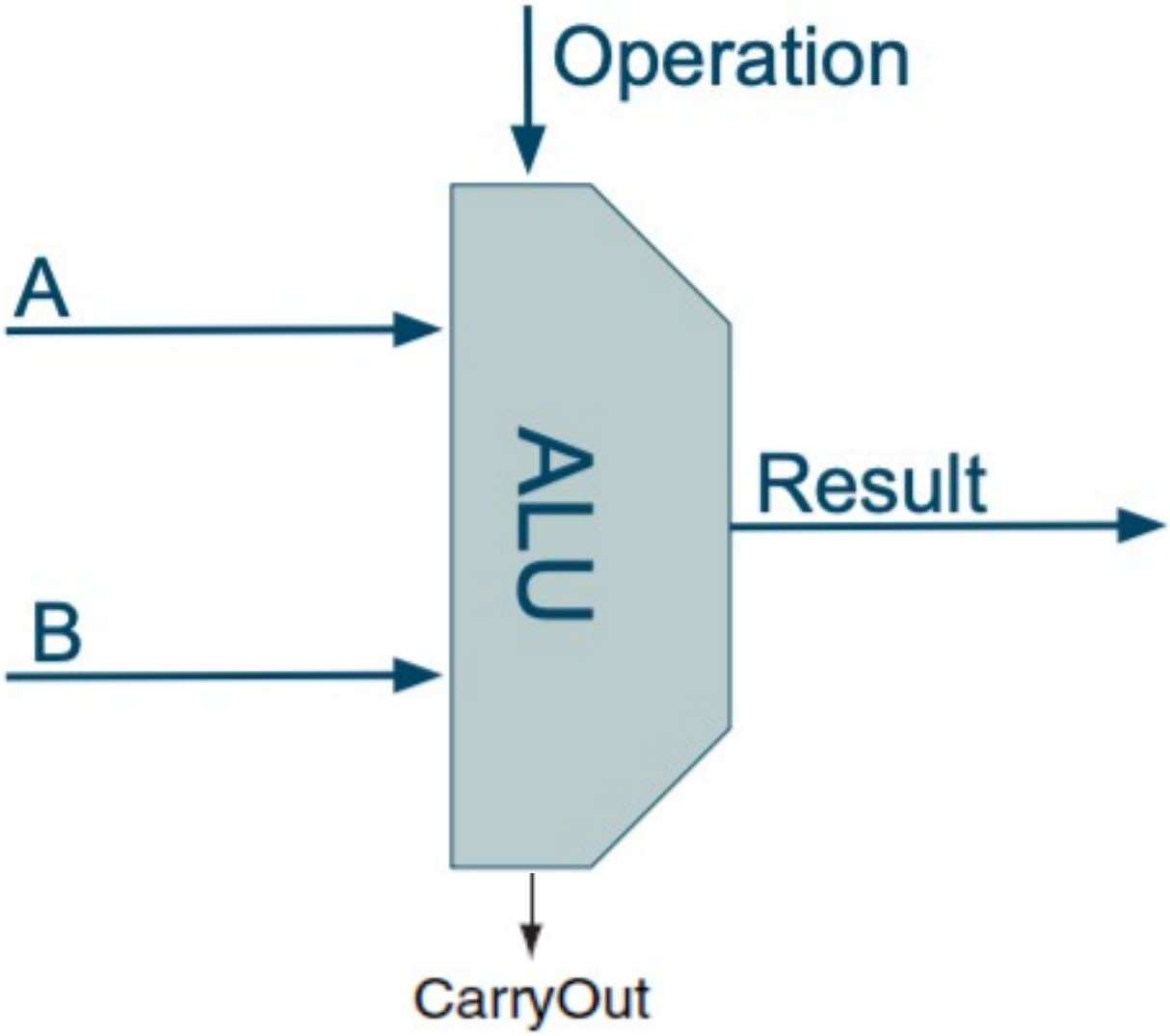
Inputs		
a	b	CarryIn
0	1	1
1	0	1
1	1	0
1	1	1

how many logic gates? 4
how long does it take? 2 epsilon

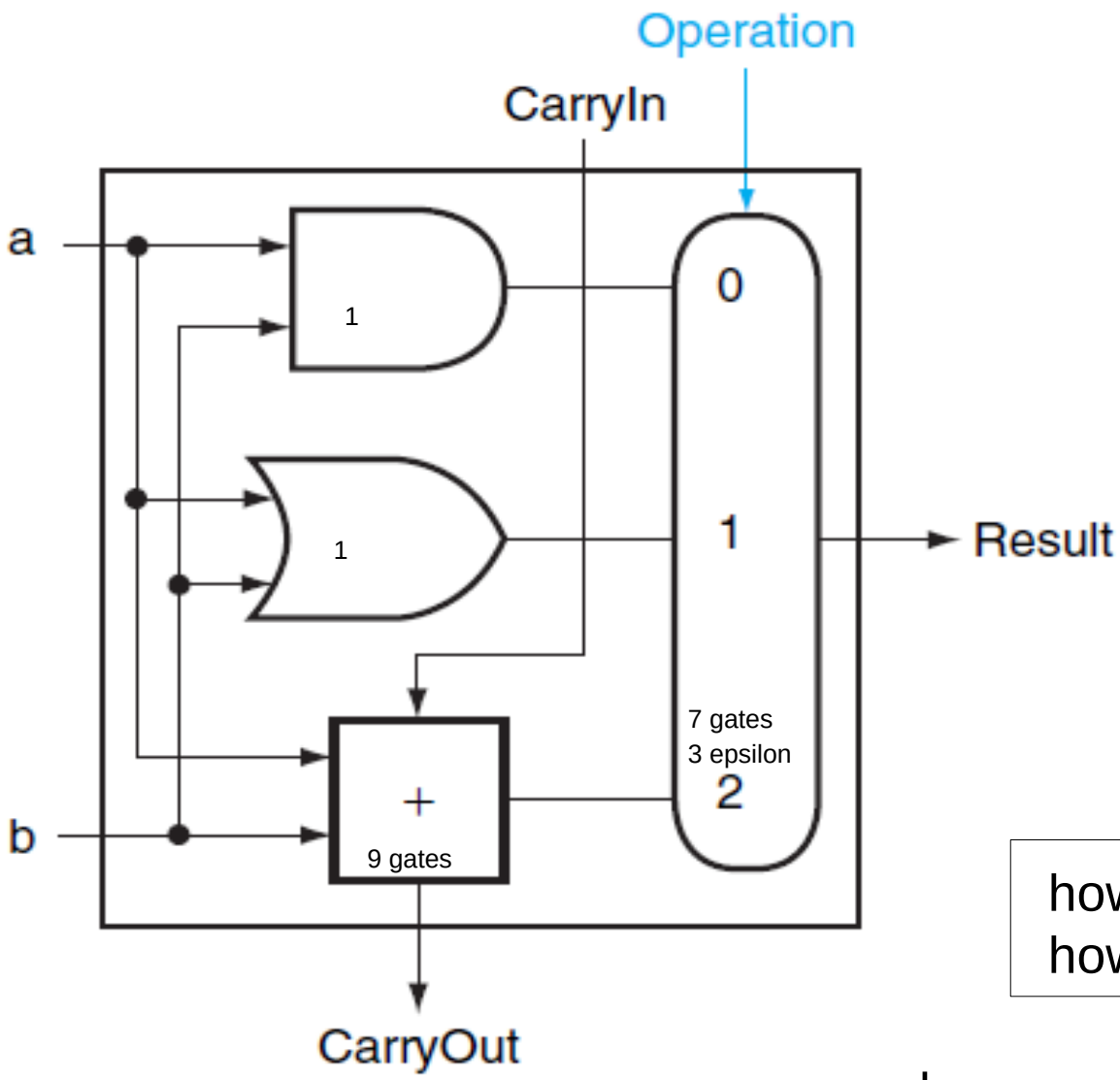
voor heel de doos, 9 gates (sum + carryin); 2 epsilon delay

32 bit adder ==> 1 bit adders aan elkaar hangen dus #GATES = 32 * 9 & DELAY = 32 * 2 epsilon

Arithmetic and Logic Unit (ALU)



1-bit ALU (AND, OR, +)



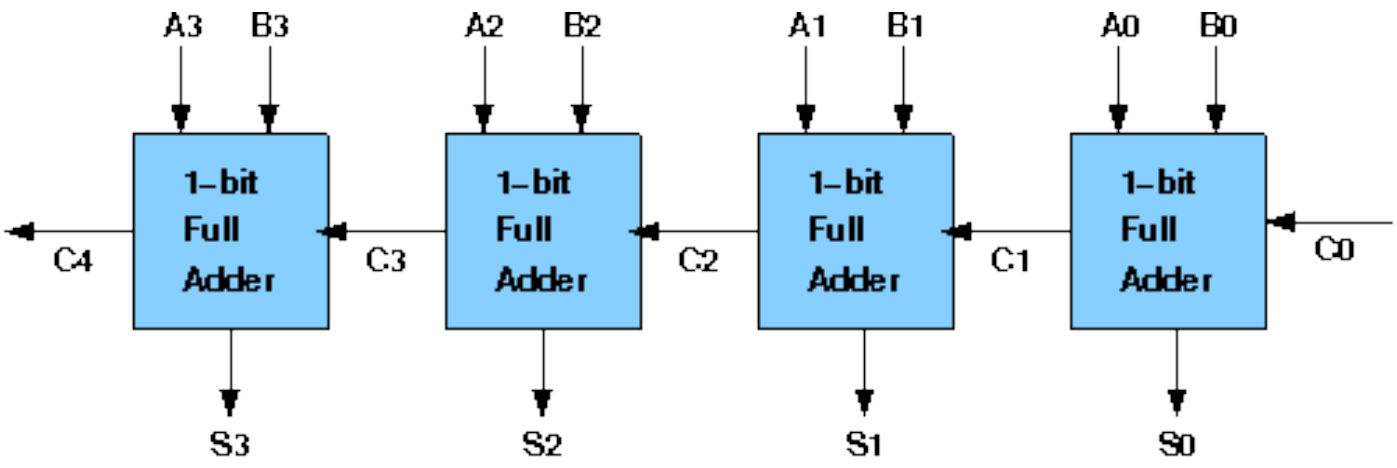
how many logic gates? 18
how long does it take? 4 epsilon

beware: multiplexor has 3 choices!

Ripple Carry Adder

Series of 1-bit full adders

Carry ripples through addition = Slow!

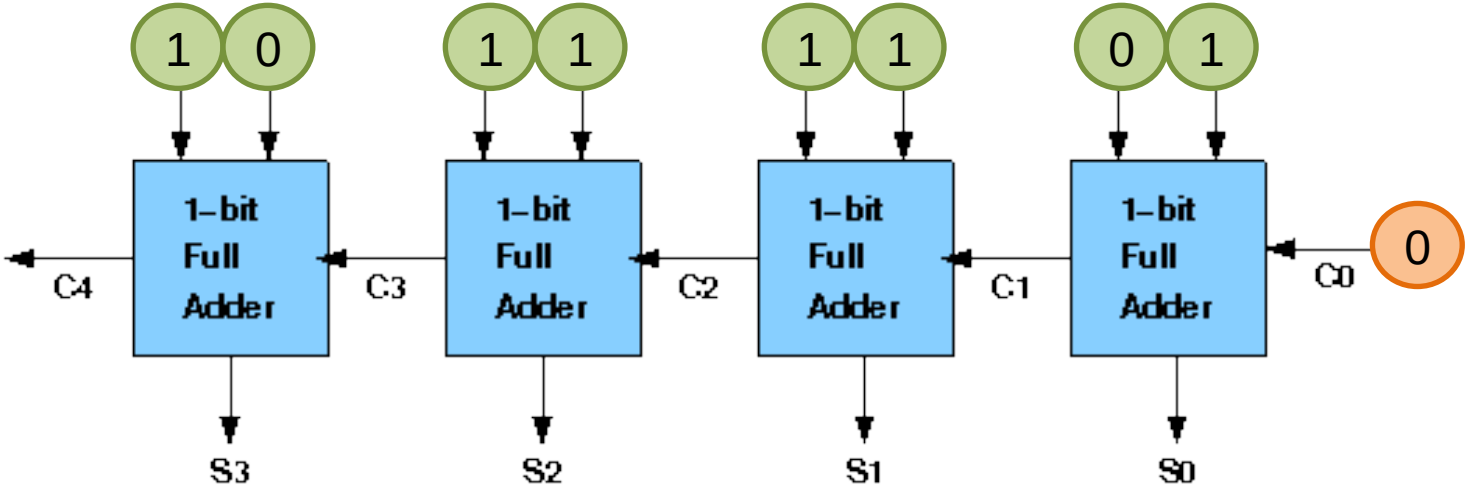


Ripple Carry Adder

Series of 1-bit full adders

Carry ripples through addition = Slow!

```
  0
 1110
+ 0111
```

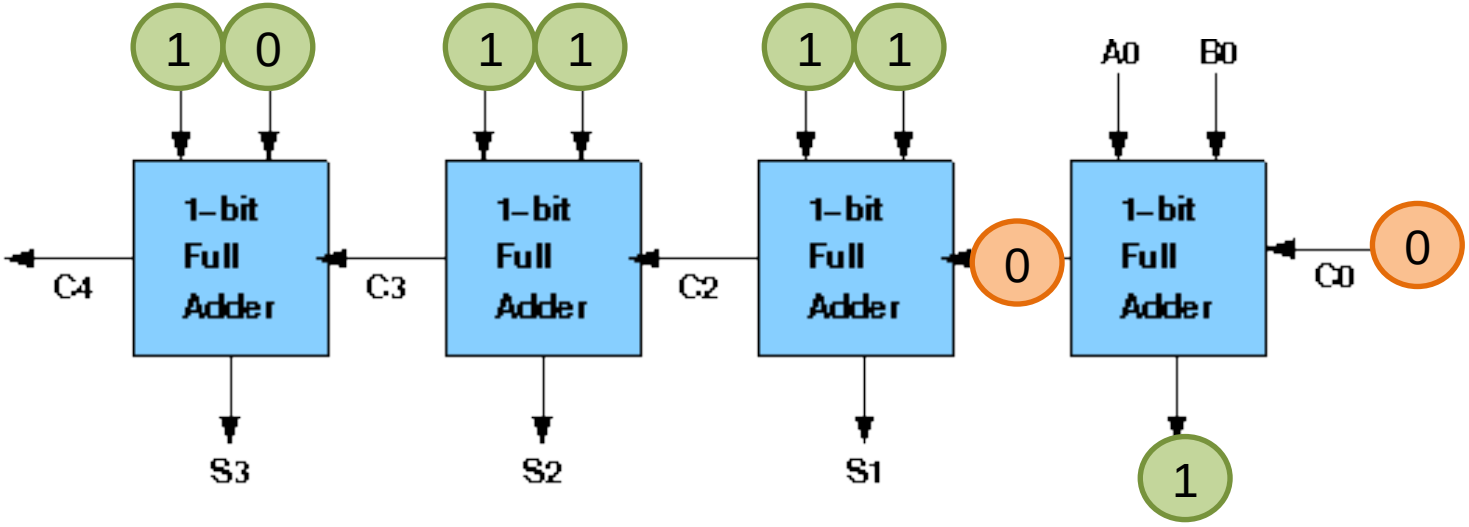


Ripple Carry Adder

Series of 1-bit full adders

Carry ripples through addition = Slow!

```
  00
1110
+ 0111
------
     1
```

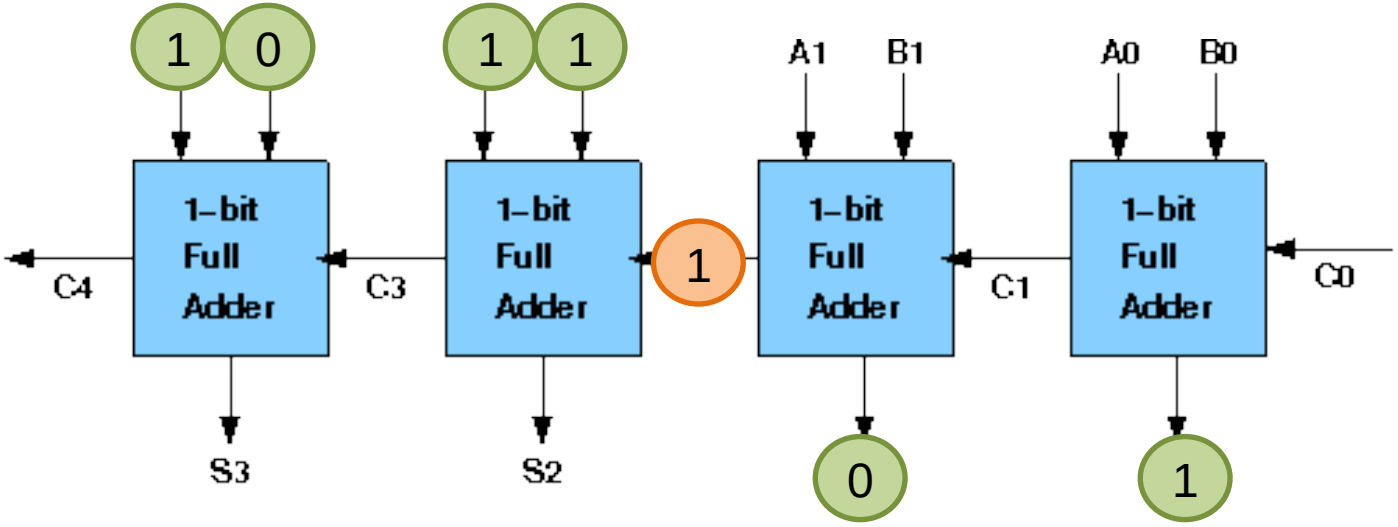


Ripple Carry Adder

Series of 1-bit full adders

Carry ripples through addition = Slow!

```
  100
 1110
+ 0111
 -----
   01
```

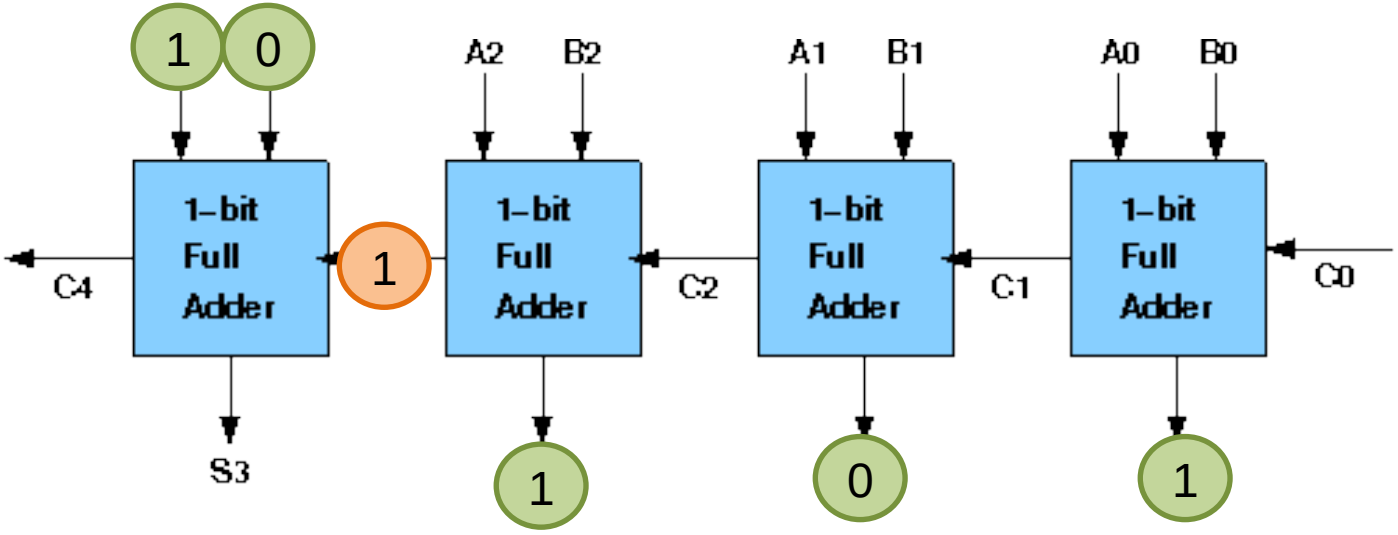


Ripple Carry Adder

Series of 1-bit full adders

Carry ripples through addition = Slow!

```
  1100
  1110
+ 0111
 -----
   101
```

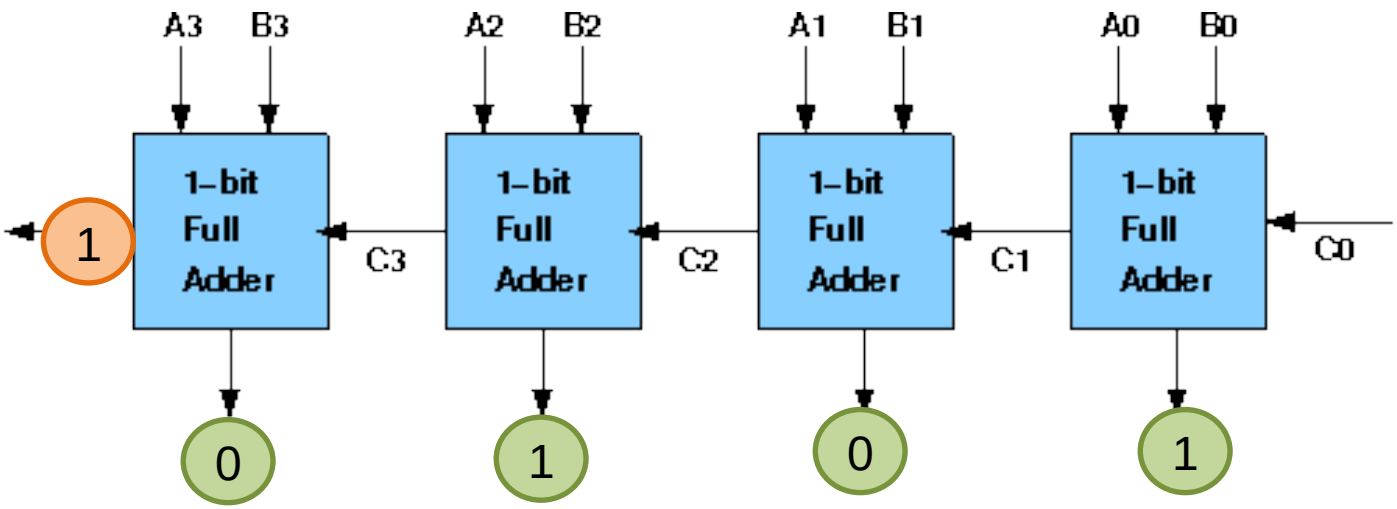


Ripple Carry Adder

Series of 1-bit full adders

Carry ripples through addition = Slow!

```
 11100
  1110
+ 0111
 -----
  0101
```



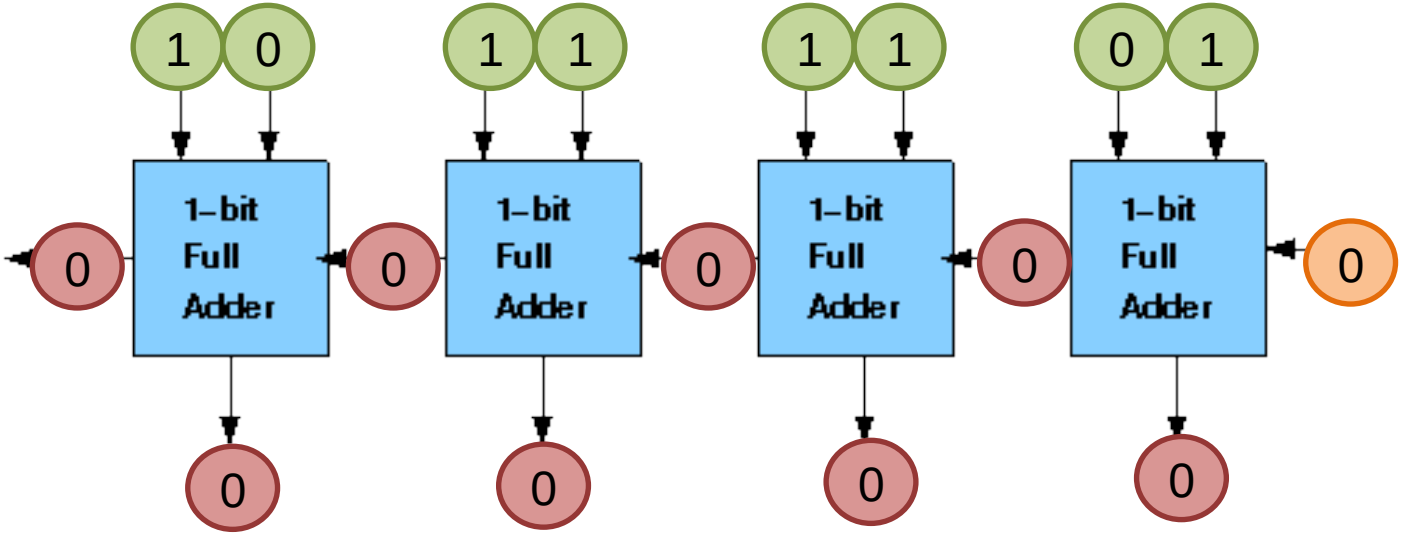
Ripple Carry Adder

Series of 1-bit full adders

Carry ripples through addition = Slow!

Incorrect intermediate result!
(assume result of previous calculation all 0s)

```
 00000
  1110
+ 0111
 ------
  0000
```



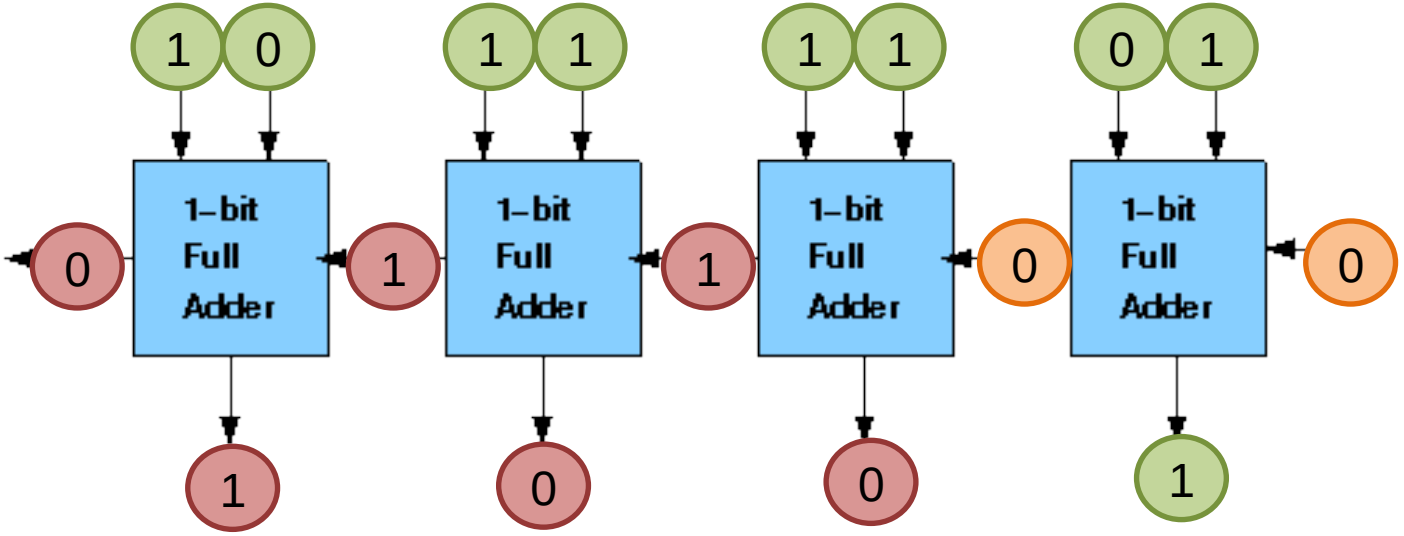
Ripple Carry Adder

```
 01100
  1110
+ 0111
 ------
  0001
```

Series of 1-bit full adders

Carry ripples through addition = Slow!

Incorrect intermediate result!



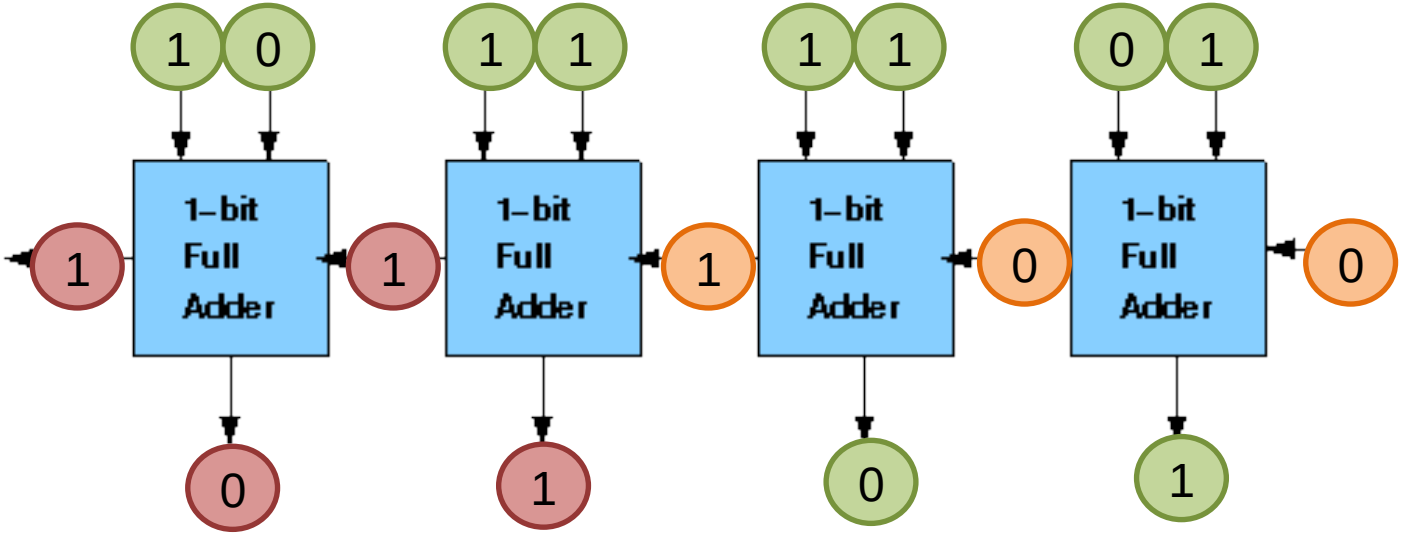
Ripple Carry Adder

```
 11100
  1110
+ 0111
 -----
  0101
```

Series of 1-bit full adders

Carry ripples through addition = Slow!

Incorrect intermediate result!

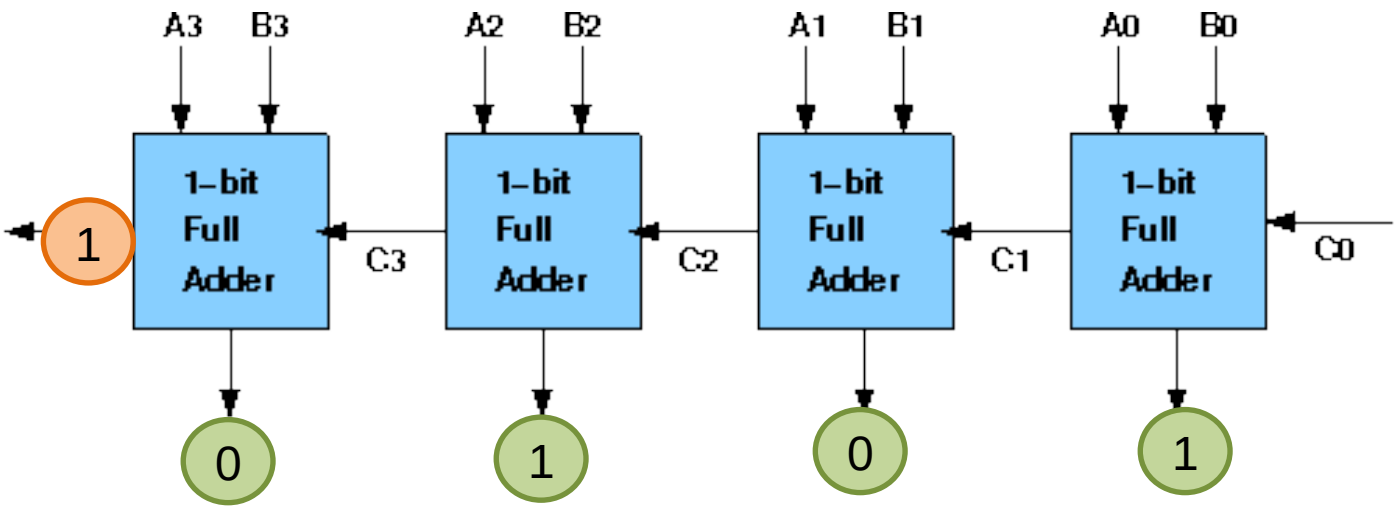


Ripple Carry Adder

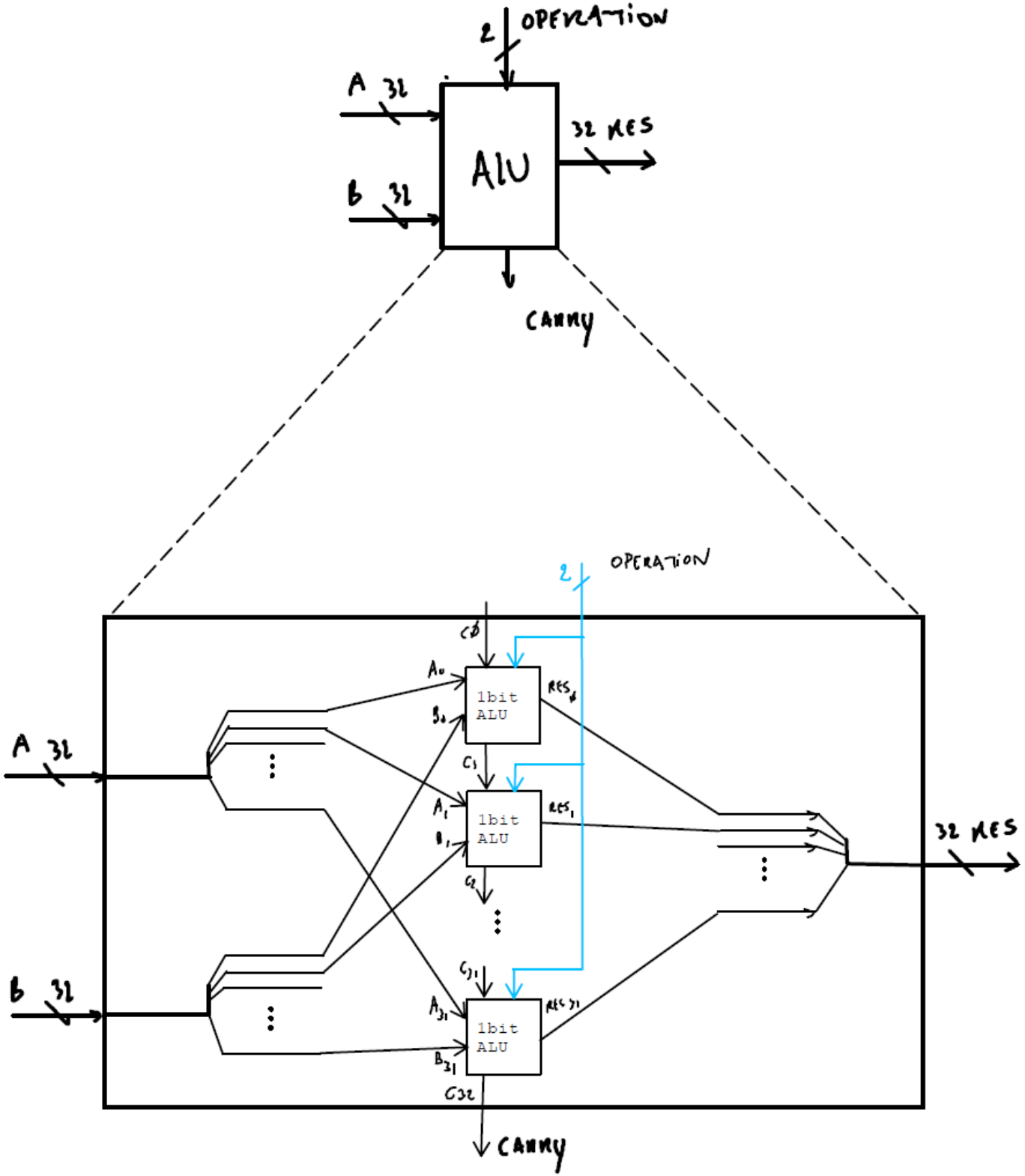
Series of 1-bit full adders

Carry ripples through addition = Slow to get to final correct result!

```
  11100
   1110
+  0111
  -----
   0101
```



how many logic gates?
how long does it take?

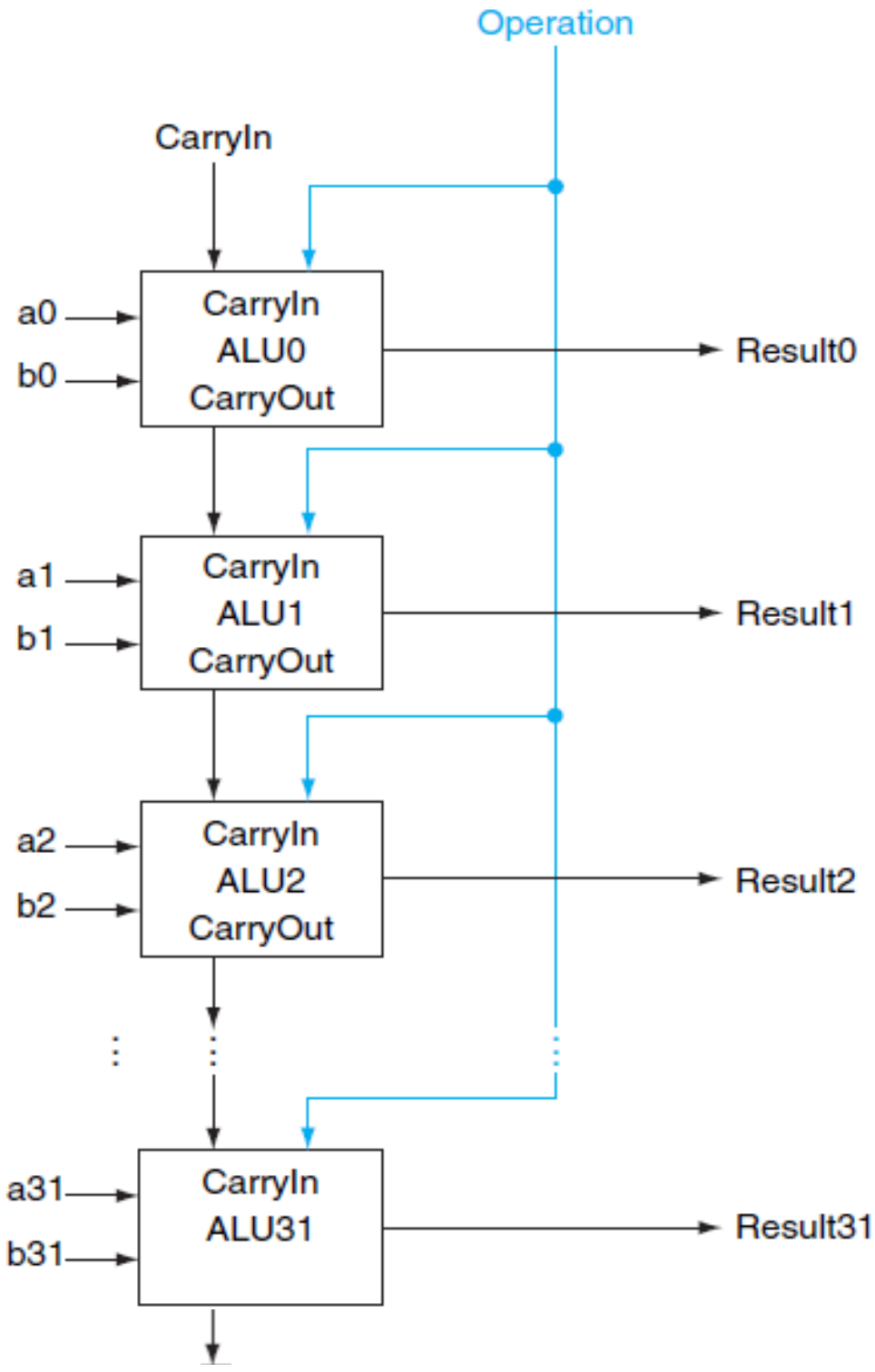

2
OPERATION
A
32
B
32
ALU
32 RES
CARRY
2
OPERATION
C0
A0
B0
1bit ALU
RES0
C1
A1
B1
1bit ALU
RES1
C2
C31
A31
B31
1bit ALU
RES31
C32
A
32
B
32
32 RES
CARRY

32-bit ALU

Adder: Ripple Carry

Inefficient!

how many logic gates?
how long does it take?

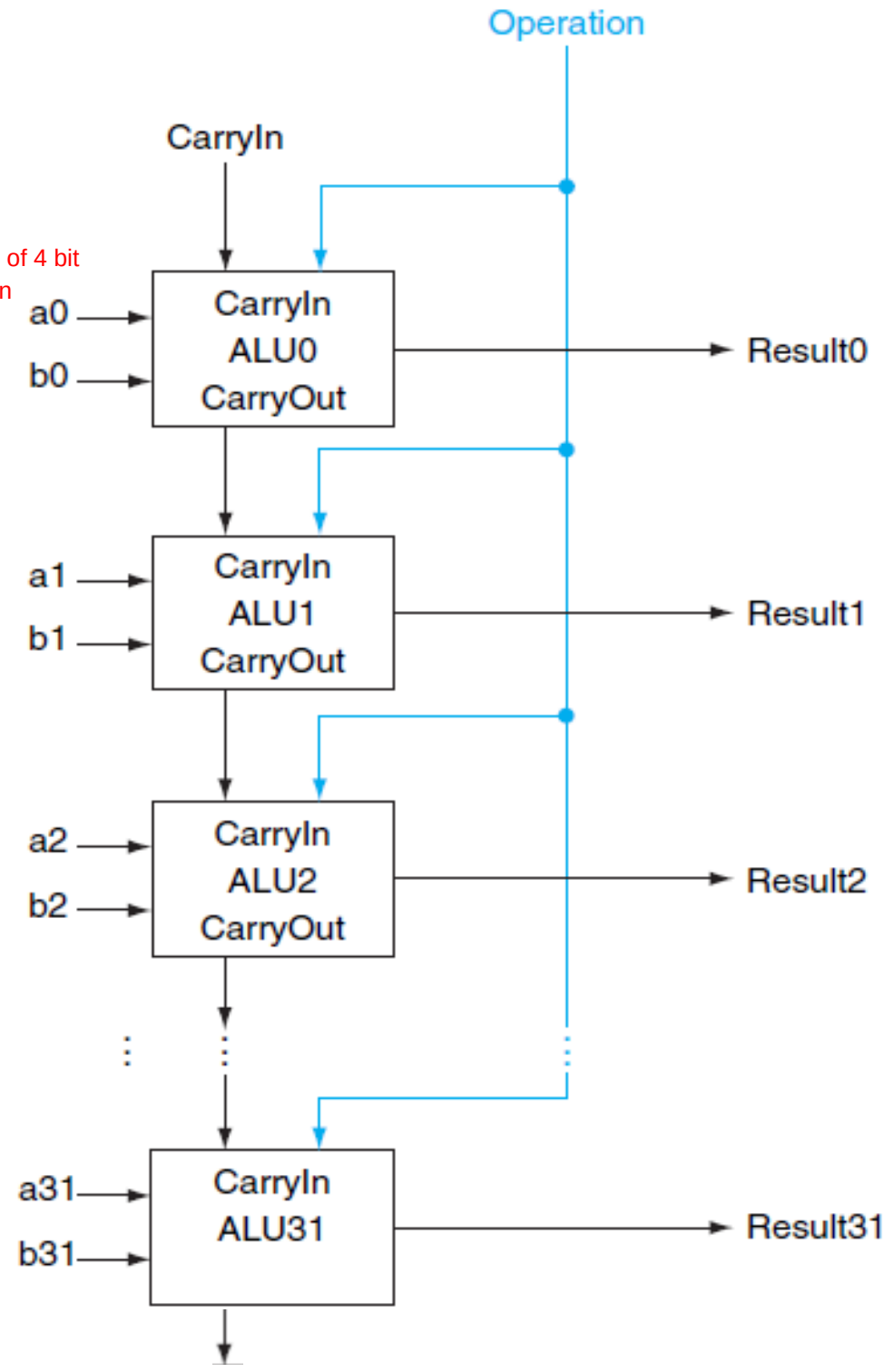


32-bit ALU

Adder: Ripple Carry

Inefficient!

groeperen en 2 bit of 4 bit adder ervan maken



how many logic gates? ~~(2+33)~~ 32+32+1 =65
how long does it take? 2 epsilon

can we make this faster?
→ number of gates?

Solution: Calculate Carry Faster

```
   ???????0  carry in
   00101110  A
 + 00100111  B
 ----------
   01010101
```

Adder: Fast Carry using “infinite hardware”

CarryIn1 = CarryOut0

op voorhand berekenen

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

Adder: Fast Carry using “infinite hardware”

CarryIn1 = (b0.CarryIn0) + (a0.CarryIn0) + (a0.b0)

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

Adder: Fast Carry using “infinite hardware”

CarryIn1 = (b0.CarryIn0) + (a0.CarryIn0) + (a0.b0)

c1 = (a0.c0) + (b0.c0) + (a0.b0)

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

Adder: Fast Carry using “infinite hardware”

CarryIn1 = (b0.CarryIn0) + (a0.CarryIn0) + (a0.b0)

c1 = (a0.c0) + (b0.c0) + (a0.b0)

c2 = (a1.c1) + (b1.c1) + (a1.b1)

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

Adder: Fast Carry using “infinite hardware”

CarryIn1 = (b0.CarryIn0) + (a0.CarryIn0) + (a0.b0)

c1 = (a0.c0) + (b0.c0) + (a0.b0)

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

c2 = (a1.c1) + (b1.c1) + (a1.b1)
= (a1. ((a0.c0) + (b0.c0) + (a0.b0))) + (b1.((a0.c0) + (b0.c0) + (a0.b0))) + (a1.b1)

Adder: Fast Carry using “infinite hardware”

CarryIn1 = (b0.CarryIn0) + (a0.CarryIn0) + (a0.b0)

c1 = (a0.c0) + (b0.c0) + (a0.b0)

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

c2 = (a1.c1) + (b1.c1) + (a1.b1)
= (a1. ((a0.c0) + (b0.c0) + (a0.b0))) + (b1.((a0.c0) + (b0.c0) + (a0.b0))) + (a1.b1)
= (a1 . a0 . c0) + (a1 . b0 . c0) + (a1 . a0 . b0)
+ (b1 . a0 . c0) + (b1 . b0 . c0) + (b1 . a0 . b0) + (a1 . b1)

Adder: Fast Carry using “infinite hardware”

CarryIn1 = (b0.CarryIn0) + (a0.CarryIn0) + (a0.b0)

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

c1 = (a0.c0) + (b0.c0) + (a0.b0)

c2 = (a1.c1) + (b1.c1) + (a1.b1)
= (a1. (a0.c0) + (b0.c0) + (a0.b0)) + (b1.(a0.c0) + (b0.c0) + (a0.b0)) + (a1.b1)
= (a1 . a0 . b0) + (a1 . a0 . c0) + (a1 . b0 . c0)
+ (b1 . a0 . b0) + (b1 . a0 . c0) + (b1 . b0 . c0) + (a1 . b1)

“direct” computation of carry (**sum of products**) ... fast, but complex

c3 = ...
c4 = ...

how many logic gates?
how long does it take?

Size of circuit grows **exponentially** (cfr. naive sum of products)

Adder: Fast Carry using “infinite hardware”

CarryIn1 = (b0.CarryIn0) + (a0.CarryIn0) + (a0.b0)

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

c1 = (a0.c0) + (b0.c0) + (a0.b0)

c2 = (a1.c1) + (b1.c1) + (a1.b1)
= (a1. (a0.c0) + (b0.c0) + (a0.b0)) + (b1.(a0.c0) + (b0.c0) + (a0.b0)) + (a1.b1)
= (a1 . a0 . b0) + (a1 . a0 . c0) + (a1 . b0 . c0)
+ (b1 . a0 . b0) + (b1 . a0 . c0) + (b1 . b0 . c0) + (a1 . b1)

“direct” computation of carry (**sum of products**) ... fast, but complex

c3 = ...
c4 = ...

Size of circuit grows **exponentially**

how many logic gates?
how long does it take?

can we reduce #gates?
→ how long does it take?

Abstraction: Propagate and Generate

$c1 = (a0.c0) + (b0.c0) + (a0.b0) = \underbrace{(a0 + b0)}_{\text{propagate}}.c0 + \underbrace{(a0.b0)}_{\text{generate}}$

Generate: “When do a_i en b_i generate a carry-out?”

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

Abstraction: Propagate and Generate

c1 = (a0.c0) + (b0.c0) + (a0.b0) = (a0 + b0).c0 + (a0.b0)

Generate: “When do a_i en b_i generate a carry-out?”

$$g_i = a_i \,.\, b_i$$

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ s3 s2 s1 s0
```

Abstraction: Propagate and Generate

$c1 = (a0.c0) + (b0.c0) + (a0.b0) = (a0 + b0).c0 + (a0.b0)$

Generate: “When do a_i en b_i generate a carry-out?”

$$g_i = a_i \,.\, b_i$$

Propagate: “When do a_i en b_i propagate a carry?”

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

Abstraction: Propagate and Generate

$c1 = (a0.c0) + (b0.c0) + (a0.b0) = (a0 + b0).c0 + (a0.b0)$

Generate: “When do a_i en b_i generate a carry-out?”

$$g_i = a_i \,.\, b_i$$

Propagate: “When do a_i en b_i propagate a carry?”

$$pi = a_i + b_i$$

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

Abstraction: Propagate and Generate

c1 = (a0.c0) + (b0.c0) + (a0.b0) = (a0 + b0).c0 + (a0.b0)

Generate: “When do a_i en b_i generate a carry-out?”

$$g_i = a_i \,.\, b_i$$

Propagate: “When do a_i en b_i propagate a carry?”

$$pi = a_i + b_i$$

```
  c3 c2 c1 c0
  a3 a2 a1 a0
  b3 b2 b1 b0
+ ----------
  s3 s2 s1 s0
```

generate + propagate * carryin

$$\boxed{\text{carry-out}_i = g_i + p_i \,.\, c_i}$$

note that g_i and p_i do not depend on c_i

$$\boxed{\text{carry-in}_{i+1} = \text{carry-out}_i = g_i + p_i \,.\, c_i}$$

gi en pi hangen puur af van a0 en b0

“generate” $g_i = a_i.b_i$

“propagate” $p_i = a_i + b_i$

$c_{i+1} = g_i + p_i.c_i$

When $g_i = 1$:

$c_{i+1} = g_i + p_i \,.\, c_i = 1 + p_i \,.\, c_i = 1$ independent of c_i : “generate”

When $g_i = 0$ and $p_i = 1$:

$c_{i+1} = 0 + 1 \,.\, c_i = c_i$ independent of a_i and b_i
: “propagate” c_i

Adder: Fast Carry
"Carry-Lookahead" (16-bit adder)

Level of abstraction 1 (4-bit adder)

$$c1 = (a0.b0) + (a0 + b0).c0$$
$$c2 = (a1.b1) + (a1 + b1).c1$$
$$= (a1.b1) + (a1 + b1).((a0.b0) + (a0 + b0).c0)$$

```
    c4 c3 c2 c1 c0
       a3 a2 a1 a0
       b3 b2 b1 b0
+  ------------------
    s4 s3 s2 s1 s0
```

"generate" $g_i = a_i.b_i$
"propagate" $p_i = a_i + b_i$

$$c_{i+1} = g_i + p_i.c_i$$

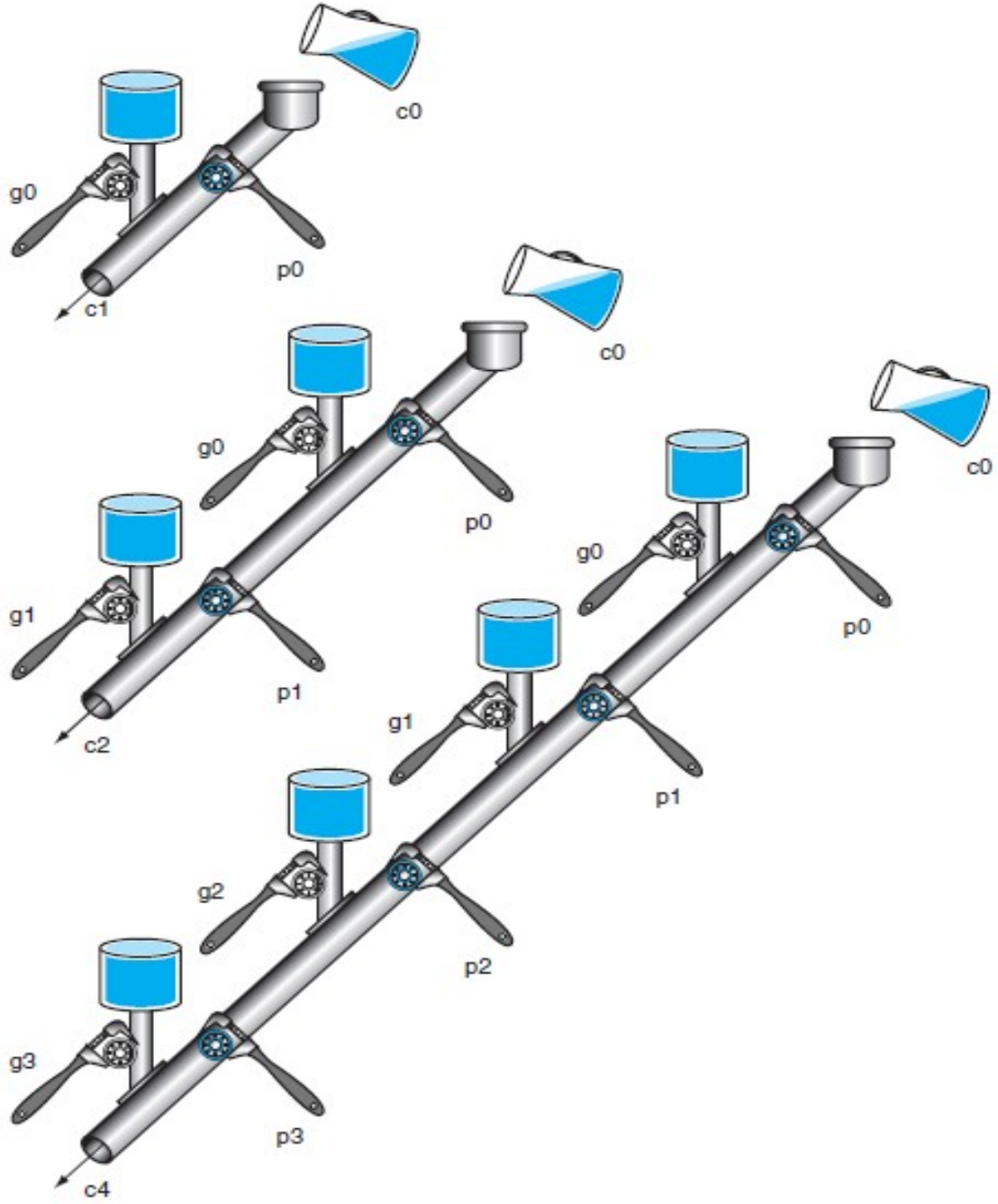
how many logic gates?
how long does it take? 3 epsilon

$$c1 = g0 + p0.c0$$
$$c2 = g1 + (p1.g0) + (p1.p0.c0)$$
$$c3 = g2 + (p2.g1) + (p2.p1.g0) + (p2.p1.p0.c0)$$
$$c4 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0) + (p3.p2.p1.p0.c0)$$

1 grote OR gate per 'c', maar parallel ==> 1 epsilon

OR | AND | OR/AND :==> 3 epsilon

Plumbing Analogy



Calculate Carry

```
  c4 c3 c2 c1 c0
     a3 a2 a1 a0
     b3 b2 b1 b0
+ ------------------
  s4 s3 s2 s1 s0
```

c1 = g0 + p0.c0
c2 = g1 + (p1.g0) + (p1.p0.c0)
c3 = g2 + (p2.g1) + (p2.p1.g0) + (p2.p1.p0.c0)

Calculate Carry

```
  c4 c3 c2 c1 c0
     a3 a2 a1 a0
     b3 b2 b1 b0
+ ------------------
  s4 s3 s2 s1 s0
```

c4 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0) + (p3.p2.p1.p0.**c0**)

note: **only** depends on c0 and a0,b0 a1,b1 a2,b2 a3, b3

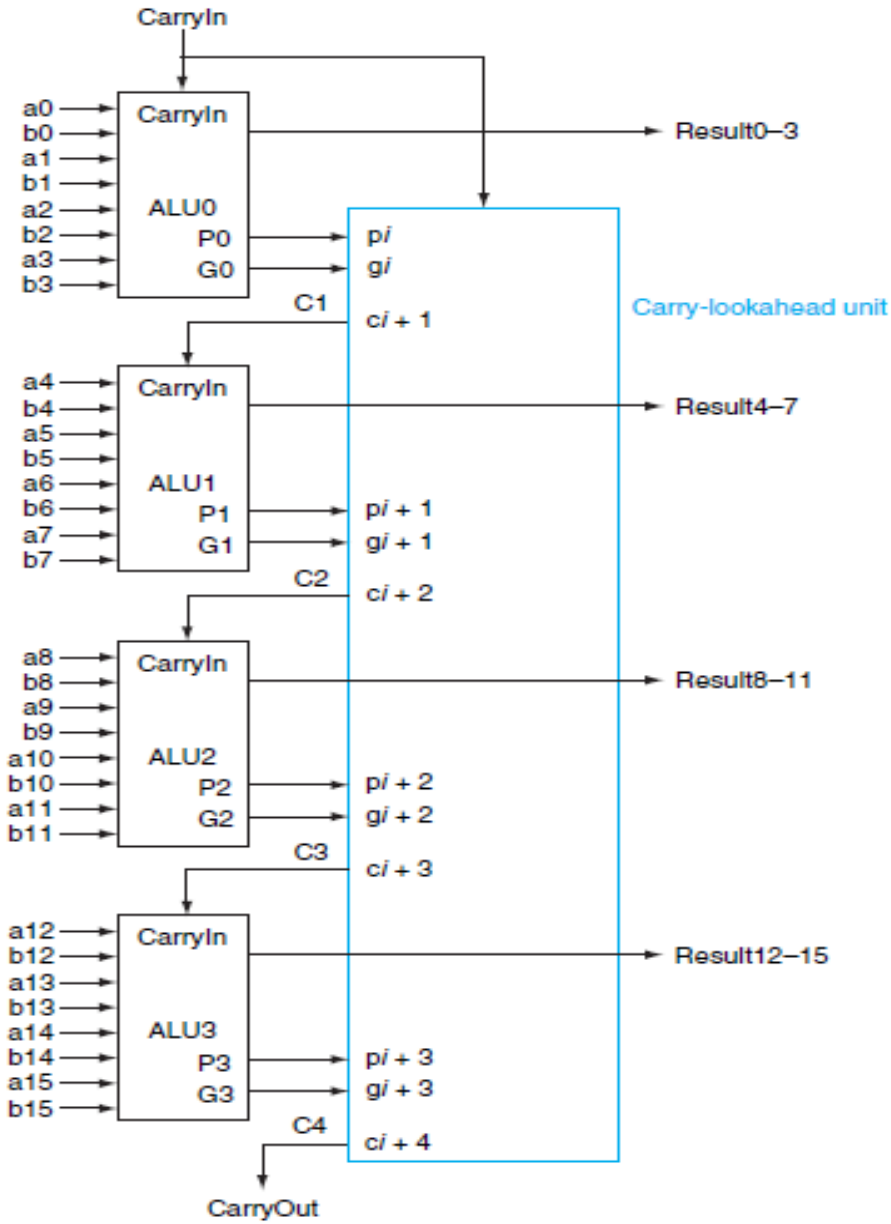
Carry-Lookahead

Level of abstraction 2

hoe alles aan elkaar hangen

4 bit adder
niet ripple 1
bits

16-bit adder using
4 x 4-bit efficient adders



Level of abstraction 2

Super-Propagate and Super-Generate

c4 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0) + (p3.p2.p1.p0.**c0**)

Super-Propagate: “When do $A_{0,4}$ en $B_{0,4}$ propagate a carry?”

<u>Level of abstraction 2</u>
Super-Propagate and Super-Generate

c4 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0) + (p3.p2.p1.p0.**c0**)

Super-Propagate: “When do $A_{0,4}$ en $B_{0,4}$ propagate a carry?”

P0 = p3.p2.p1.p0

<u>Level of abstraction 2</u>
Super-Propagate and Super-Generate

c4 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0) + (p3.p2.p1.p0.**c0**)

Super-Propagate: “When do $A_{0,4}$ en $B_{0,4}$ propagate a carry?”

P0 = p3.p2.p1.p0

Super-Generate: “When do $A_{0,4}$ en $B_{0,4}$ generate a carry-out?”

<u>Level of abstraction 2</u>
Super-Propagate and Super-Generate

c4 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0) + (p3.p2.p1.p0.**c0**)

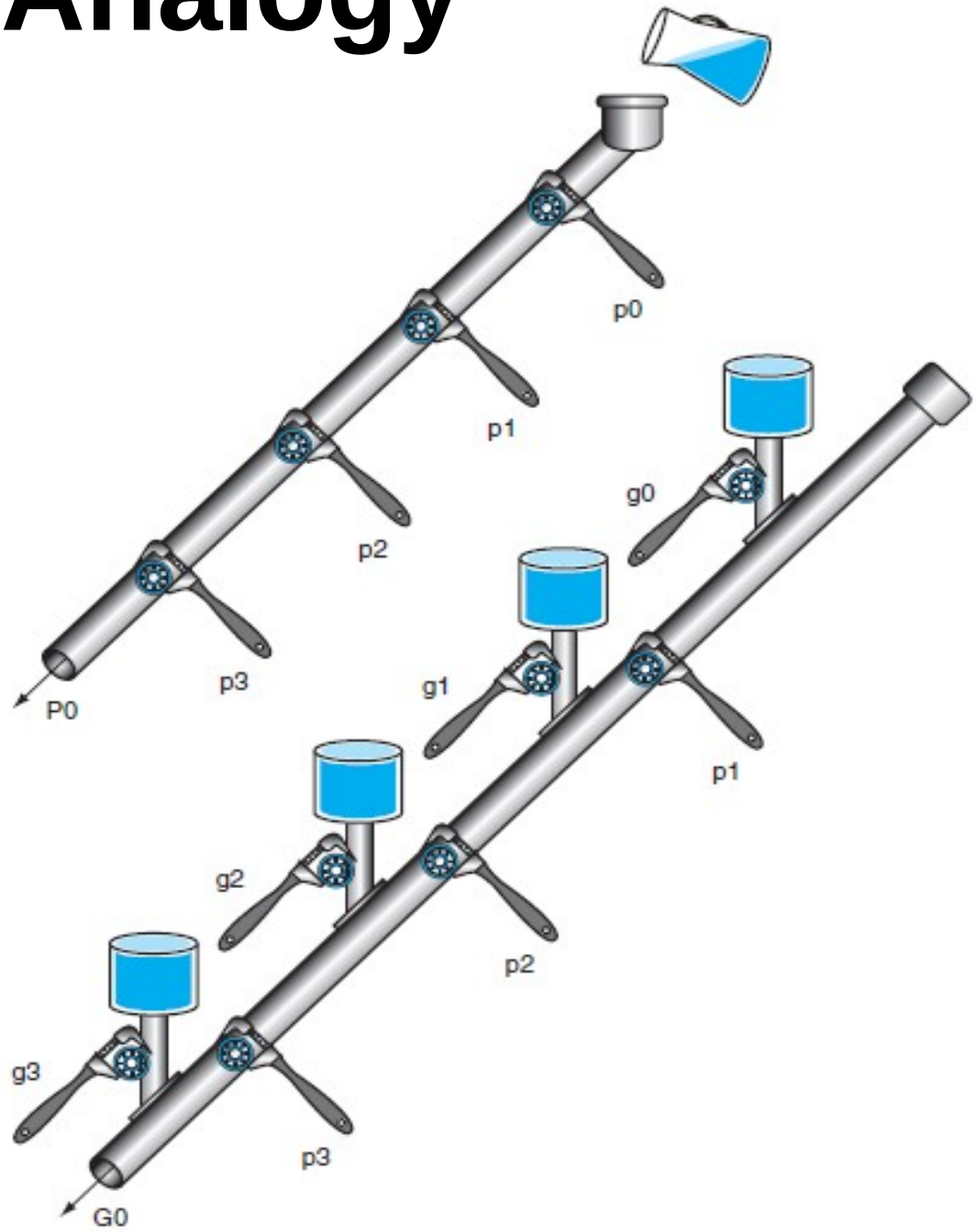
Super-Propagate: “When do $A_{0,4}$ en $B_{0,4}$ propagate a carry?”

P0 = p3.p2.p1.p0

Super-Generate: “When do $A_{0,4}$ en $B_{0,4}$ generate a carry-out?”

G0 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0)

Plumbing Analogy



<u>Level of abstraction 2</u>
Super-Propagate and Super-Generate

c4 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0) + (p3.p2.p1.p0.**c0**)

Super-Propagate: “When do $A_{0,4}$ en $B_{0,4}$ propagate a carry?”

P0 = p3.p2.p1.p0

Super-Generate: “When do $A_{0,4}$ en $B_{0,4}$ generate a carry-out?”

G0 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0)

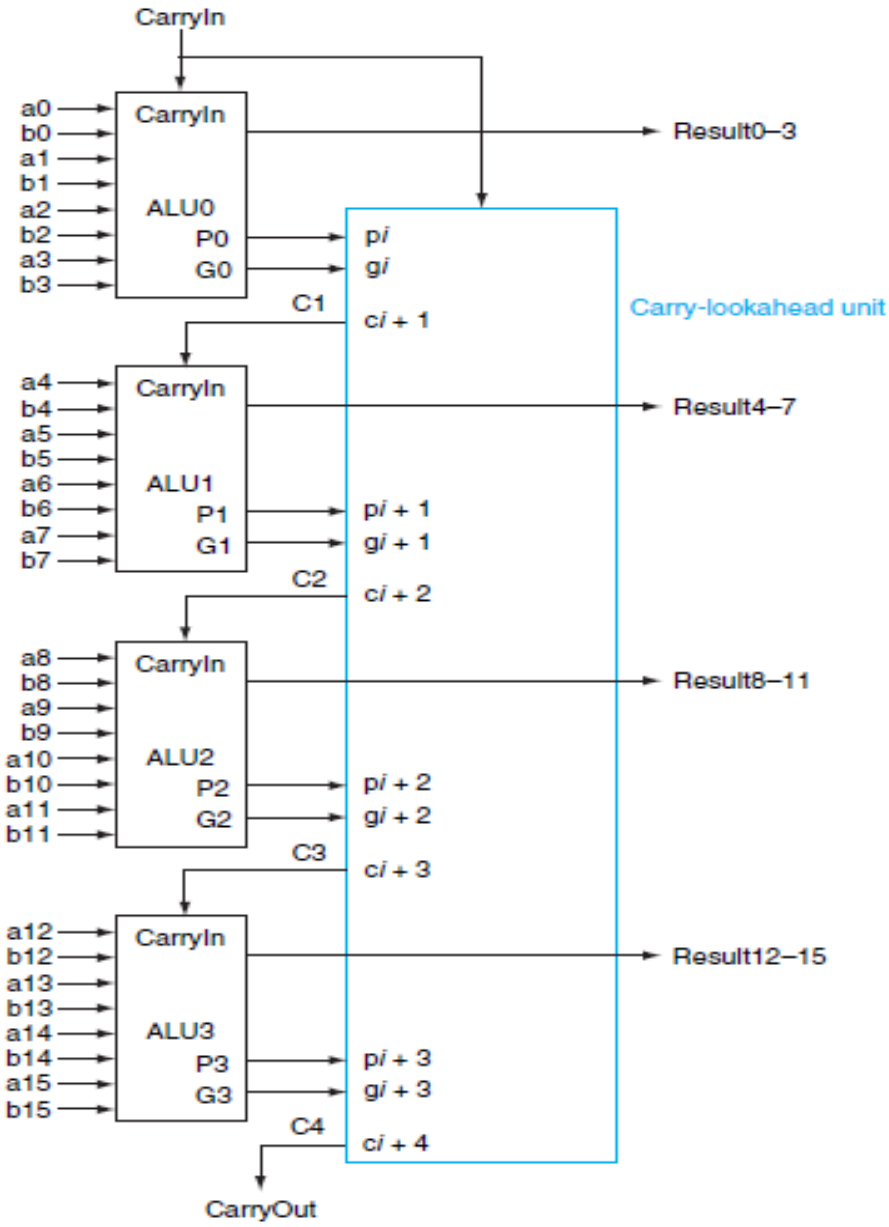
<u>C1</u> = G0 + (P0.c0)

weer hetzelfde als ervoor

Carry-Lookahead

Level of abstraction 2

16-bit adder using
4 x 4-bit efficient adders



Carry-Lookahead

Level of abstraction 2 (16-bit adder)

https://www.youtube.com/watch?v=SQKdnxysXnw&pp=ygUpY2FycnkgbG9vayBhaGVhZCBhZGRlciBhYnN0cmFjdGlvbiBsZXZlbHM%3D

P0 = p3.p2.p1.p0
P1 = p7.p6.p5.p4
P2 = p11.p10.p9.p8
P3 = p15.p14.p13.p12

G0 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0)
G1 = g7 + (p7.g6) + (p7.p6.g5) + (p7.p6.p5.g4)
G2 = g11 + (p11.g10) + (p11.p10.g9) + (p11.p10.p9.g8)
G3 = g15 + (p15.g14) + (p15.p14.g13) + (p15.p14.p13.g12)

<u>C1</u> = G0 + (P0.c0)
C2 = G1 + (P1.G0) + (P1.P0.c0)
C3 = G2 + (P2.G1) + (P2.P1.G0) + (P2.P1.P0.c0)
C4 = G3 + (P3.G2) + (P3.P2.G1) + (P3.P2.P1.G0) + (P3.P2.P1.P0.c0)

Carry-Lookahead

Level of abstraction 2 (16-bit adder)

P0 = p3.p2.p1.p0
P1 = p7.p6.p5.p4
P2 = p11.p10.p9.p8
P3 = p15.p14.p13.p12

G0 = g3 + (p3.g2) + (p3.p2.g1) + (p3.p2.p1.g0)
G1 = g7 + (p7.g6) + (p7.p6.g5) + (p7.p6.p5.g4)
G2 = g11 + (p11.g10) + (p11.p10.g9) + (p11.p10.p9.g8)
G3 = g15 + (p15.g14) + (p15.p14.g13) + (p15.p14.p13.g12)

how many logic gates?
how long does it take?

<u>C1</u> = G0 + (P0.c0)
C2 = G1 + (P1.G0) + (P1.P0.c0)
C3 = G2 + (P2.G1) + (P2.P1.G0) + (P2.P1.P0.c0)
C4 = G3 + (P3.G2) + (P3.P2.G1) + (P3.P2.P1.G0) + (P3.P2.P1.P0.c0)

Binary representation/encoding of Unsigned Integers

n-bit string "$x_{n-1}x_{n-2}\ldots x_1x_0$" has/encodes **value** x

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

x_0 Least Significant Bit (**LSB**)

x_{n-1} Most Significant Bit (**MSB**)

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

Addition of Unsigned Integers in binary representation/encoding

```
      0 Carry        0
   1110 A           14
 + 0111 B         + 7
 ------
        Result
```

Addition of Unsigned Integers in binary representation/encoding

```
  11100 Carry        10
   1110 A            14
 + 0111 B          + 7
 ------            ---
  10101 Result      21
```

2's complement binary representation/encoding of Signed Integers

met n bits kan ik 2^n dingen voorstellen

==> 1 vooraan = -

n-bit string '$x_{n-1}x_{n-2}\ldots x_1x_0$' has **value** x

$$x = \boxed{-x_{n-1}2^{n-1}} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

101: -4+0+1 = -3

VB:
x 011 =3

!x 100
+1 001
---
101 = -3

negatieve versie van getal vinden

	unsigned	signed
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

100	-4
101	-3
110	-2
111	-1
000	0
001	1
010	2
011	3

!! nog eens herbekijken, ik snap het niet !!

Negation for 2's complement

Complement and **add 1**

complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111\ldots111_2 = -1$$

$$\bar{x} + 1 = -x$$

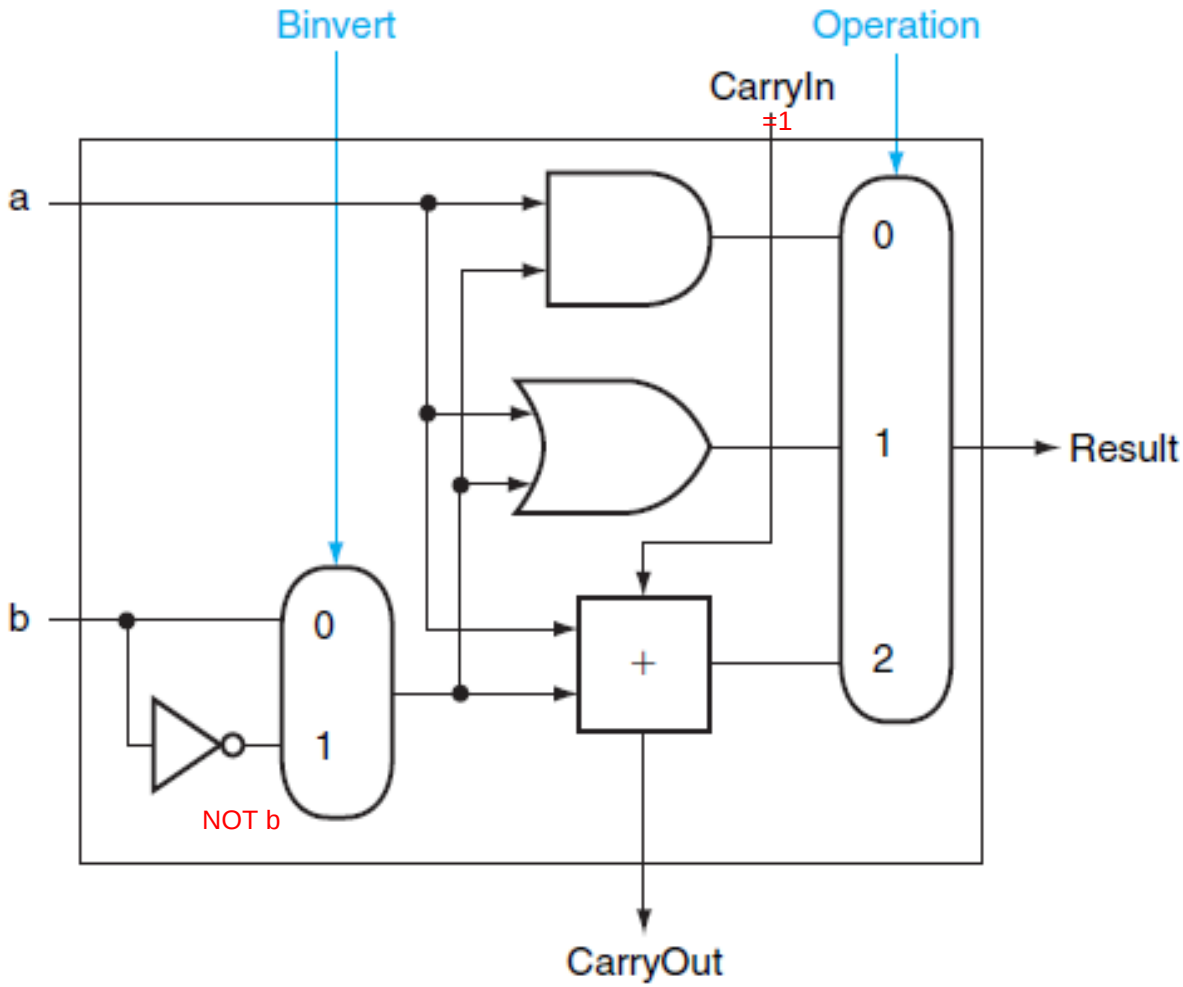
Example: negate +2

- $+2 = 0000\ 0000\ \ldots\ 0010_2$
- $-2 = 1111\ 1111\ \ldots\ 1101_2$

$$\begin{aligned} & \qquad\qquad\qquad\qquad + 1_2 \\ & = 1111\ 1111\ \ldots\ 1110_2 \end{aligned}$$

Note: works for positive and negative numbers

1-bit ALU with subtraction



$$a - b = a + (-b)$$

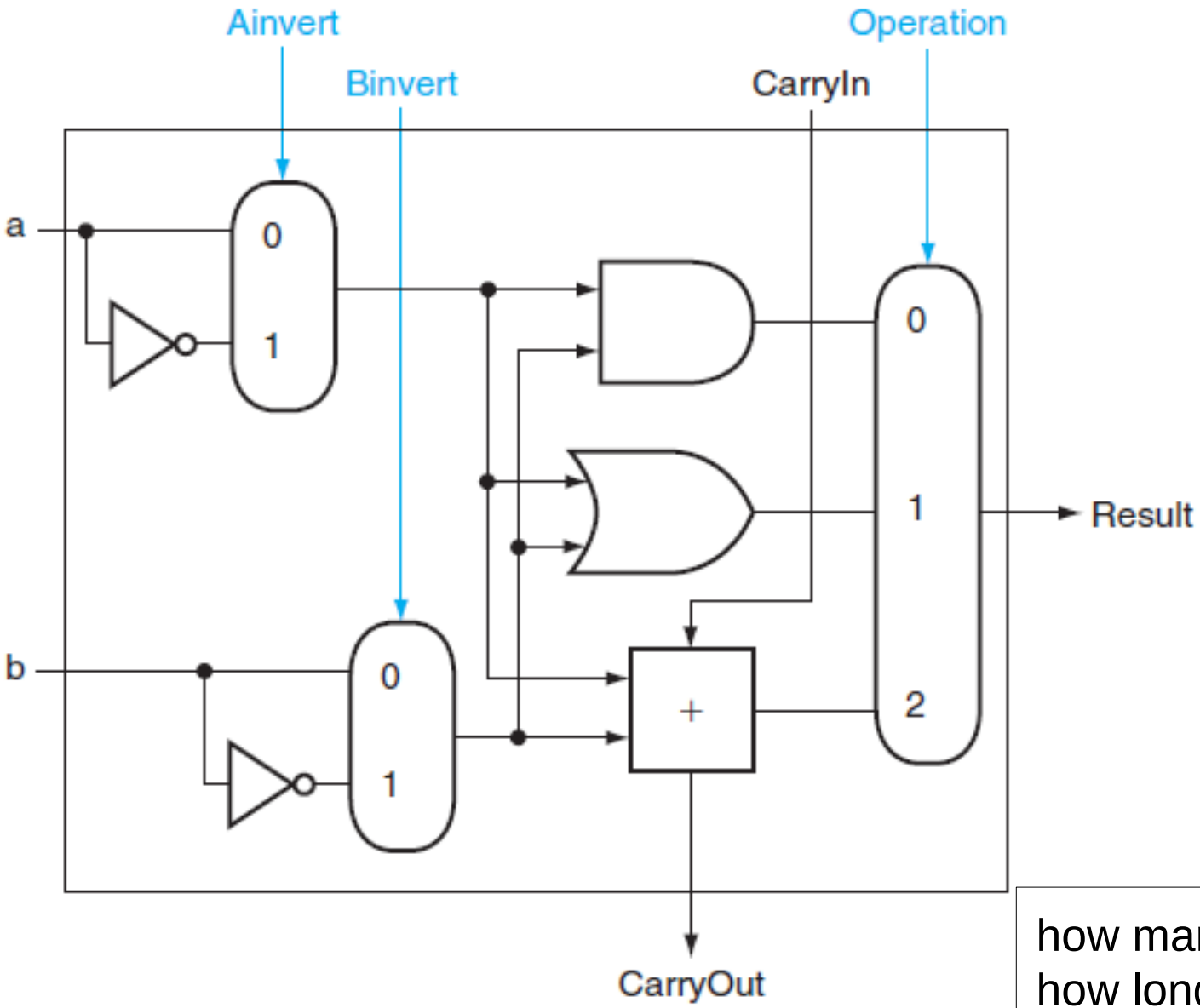
$$= a + (\overline{b} + \overset{\text{carryin}}{1})$$

($\overline{}$ = 1's complement)

Binvert **and** CarryIn: 1

1-bit ALU with NOR

```
a NOR b =  not(a OR b) =
(not a) AND (not b)
```



how many logic gates?
how long does it take?

Overflow Conditions

(for **2's complement signed** binary integers)

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

SIGNED INTEGERS (gehele getallen)

```
oef:
   1
   010     ==2
+  011     ==3
-------------------
    101     == -3
```

OVERFLOW

MSB (eerste) naijken
0
0
1

SIGNED INTEGERS (natuurlijke getallen)

```
oef:
   1
   010     ==2
+  011     ==3
-------------------
    101     ==> 5
```

```
oef:
carryin = 1 & valt weg
   111
   101     ==5
+  011     ==3
-------------------
    000     ==> 0
```

(3 bit unsigned optelling)

MOD(8) optellen

dus 8 = 0, 12 = 4 ... VAN 0 - 7

Unsigned Binary Addition

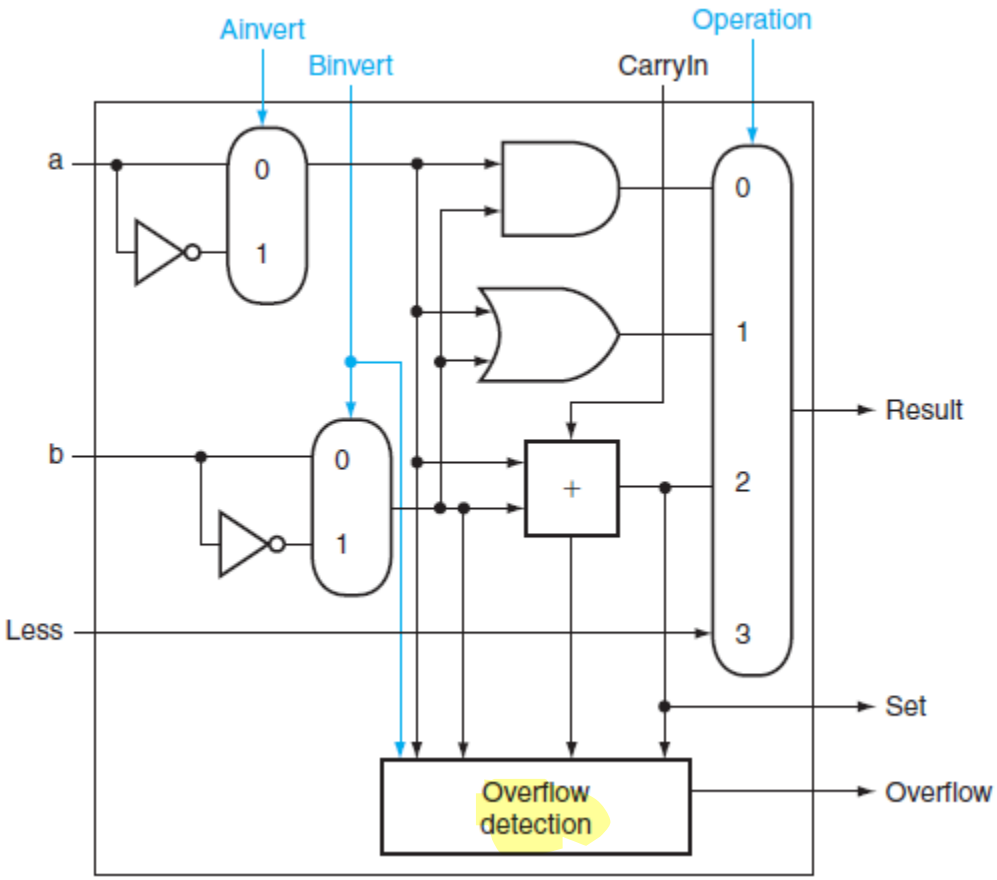
“modulo” calculation, **no overflow!**

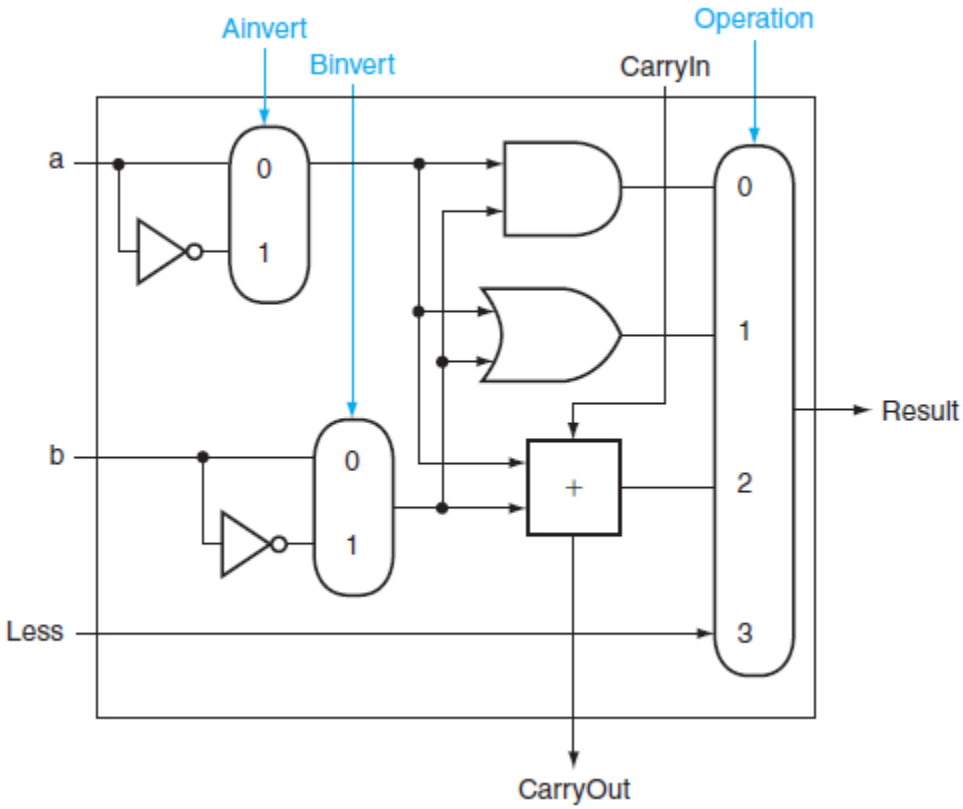
N bits: (2^N) MOD $2^N = 0 = [[2^N]]$
(2^N) DIV $2^N = 1$

$[[2^N \mathbf{+k}]] = (2^N+k)$ MOD $2^N = \mathbf{k}$

used in for example memory address calculation (Program Counter)

1-bit ALU (AND, OR, +, -, slt)





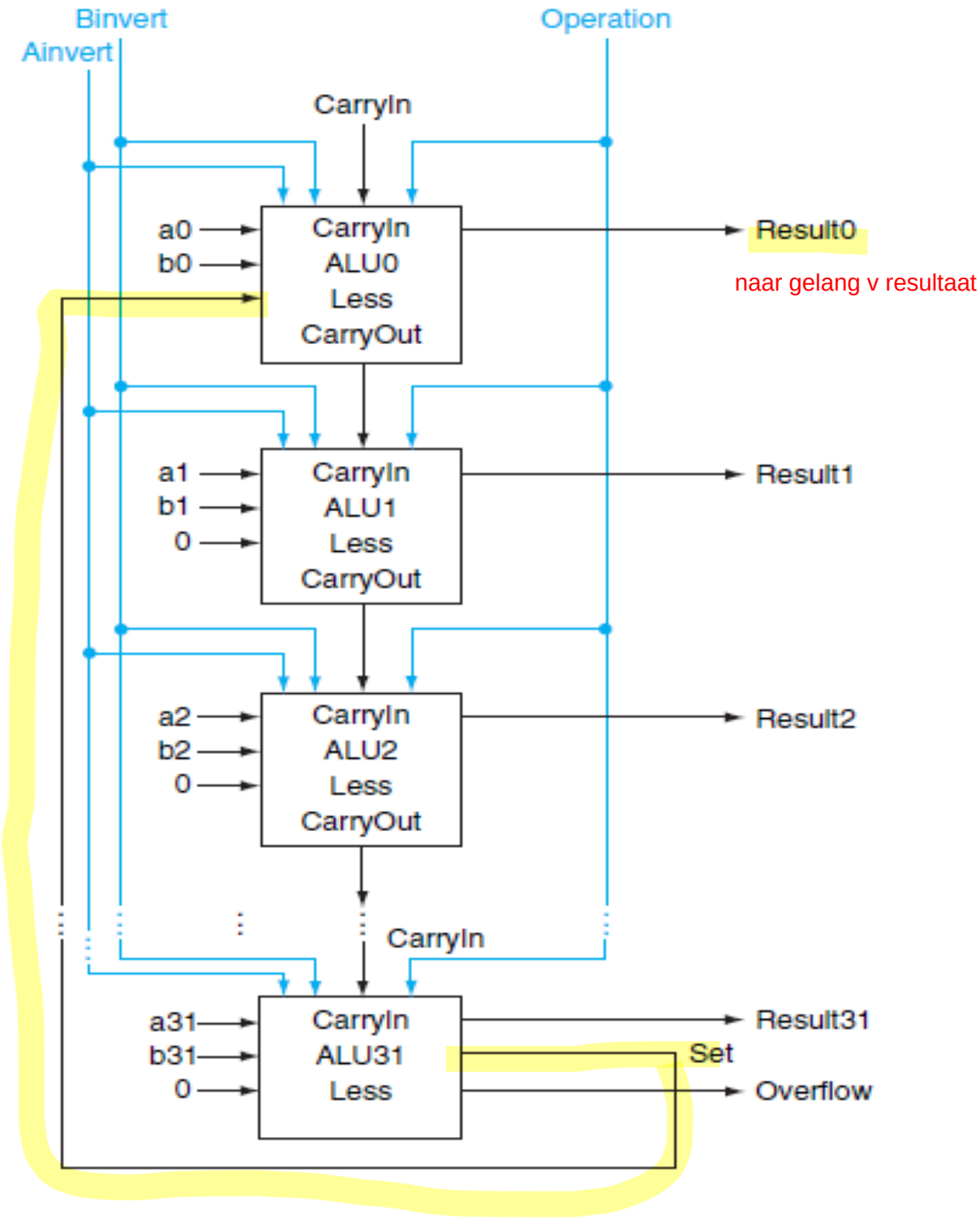
MSb

non-MSb

32-bit ALU: only connect

getal a = 32 bits
getal b = 32 bits
set less than

hoogste bit bekijken
==> zien ofdat we die op 1 of 0 zetten



How to efficiently compare **non-negative** numbers?

(useful for fast comparison of Floating Point number exponents)

```
Is A > B ?

e.g., is 00101000 > 00111010

Solution:

 NOT: check whether A – B is negative (as for slt)
      this works for unsigned numbers, but is slow ...
```

maar ter informatie
niet belangrijk

How to efficiently compare **non-negative** numbers?

Is A > B ?

Solution:

from MSB to LSB (left to right): **compare bit per bit**

```
00101000
00111010
```

Assuming that the two bit strings
(representing non-negative integers) both have N bits

```
for i = N-1 downto 0:
  if bit A_i > B_i then
    A is larger than B
    break
  elif bit A_i < B_i then
    A is smaller than B
    break
  else // bit A_i == B_i
    if i == 0 then
     A is equal to B
//  else
//   continue loop with next i
```

How to efficiently compare **non-negative** numbers?

Is A > B ?

Solution:

but we want a **hardware** solution …

check whether i^{th} bits are equal $(A_i == B_i) = A_i.B_i + \overline{A}_i.\overline{B}_i = e_i$
check whether i^{th} bits < $(A_i < B_i) = \overline{A}_i.B_i$
check whether i^{th} bits > $(A_i > B_i) = A_i.\overline{B}_i$

4 bits example, check A > B:

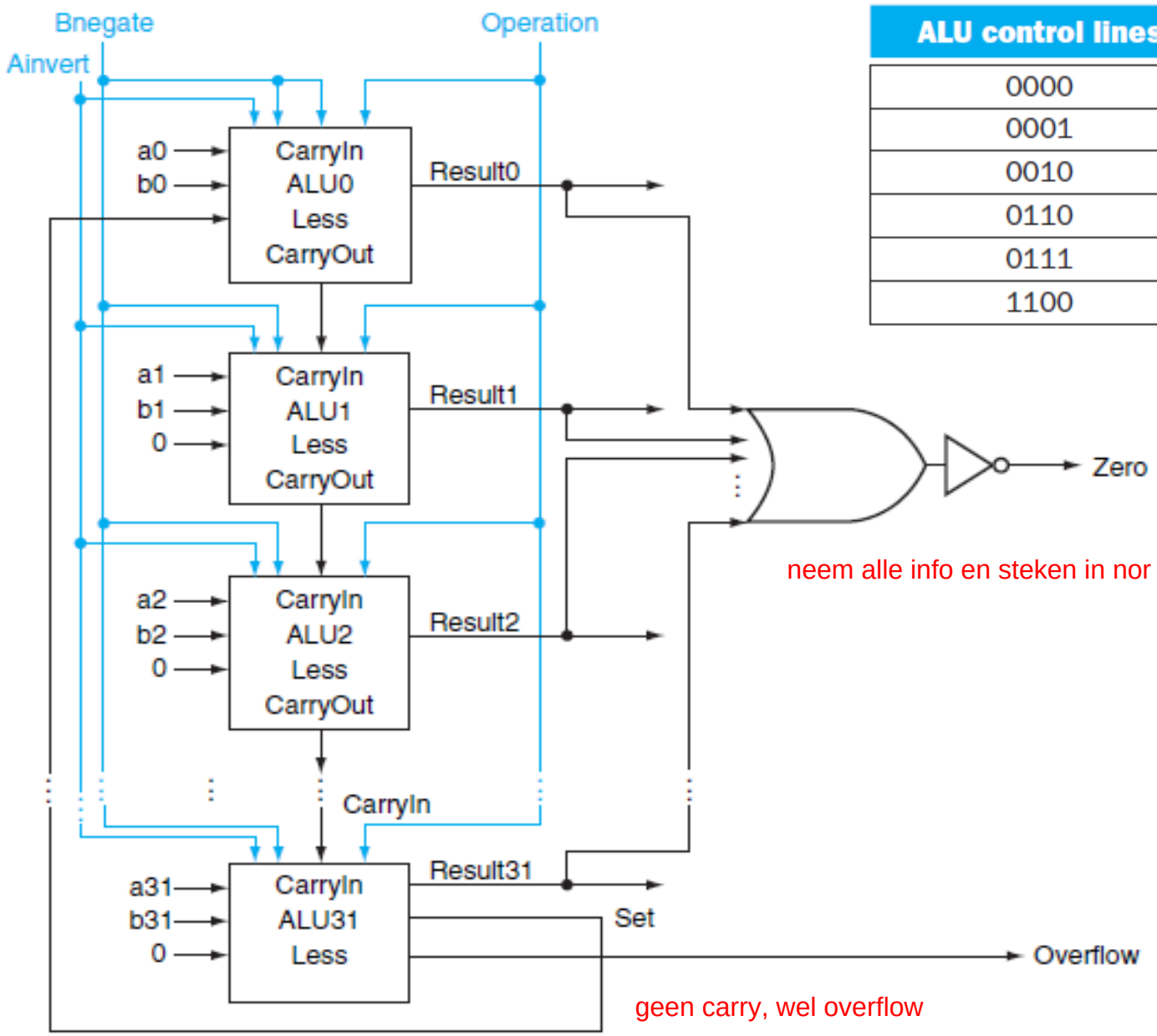
$$A_3.\overline{B}_3 + e_3.A_2.\overline{B}_2 + e_3.e_2.A_1.\overline{B}_1 + e_3.e_2.e_1.A_0.\overline{B}_0$$

Number of gates?
Delay?

maar ter informatie
niet belangrijk

For more information, see https://en.wikipedia.org/wiki/Digital_comparator

32-bit ALU with zero detect



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

neem alle info en steken in nor

geen carry, wel overflow

Higher-level: ALU