



KTH Informations- och
kommunikationsteknik

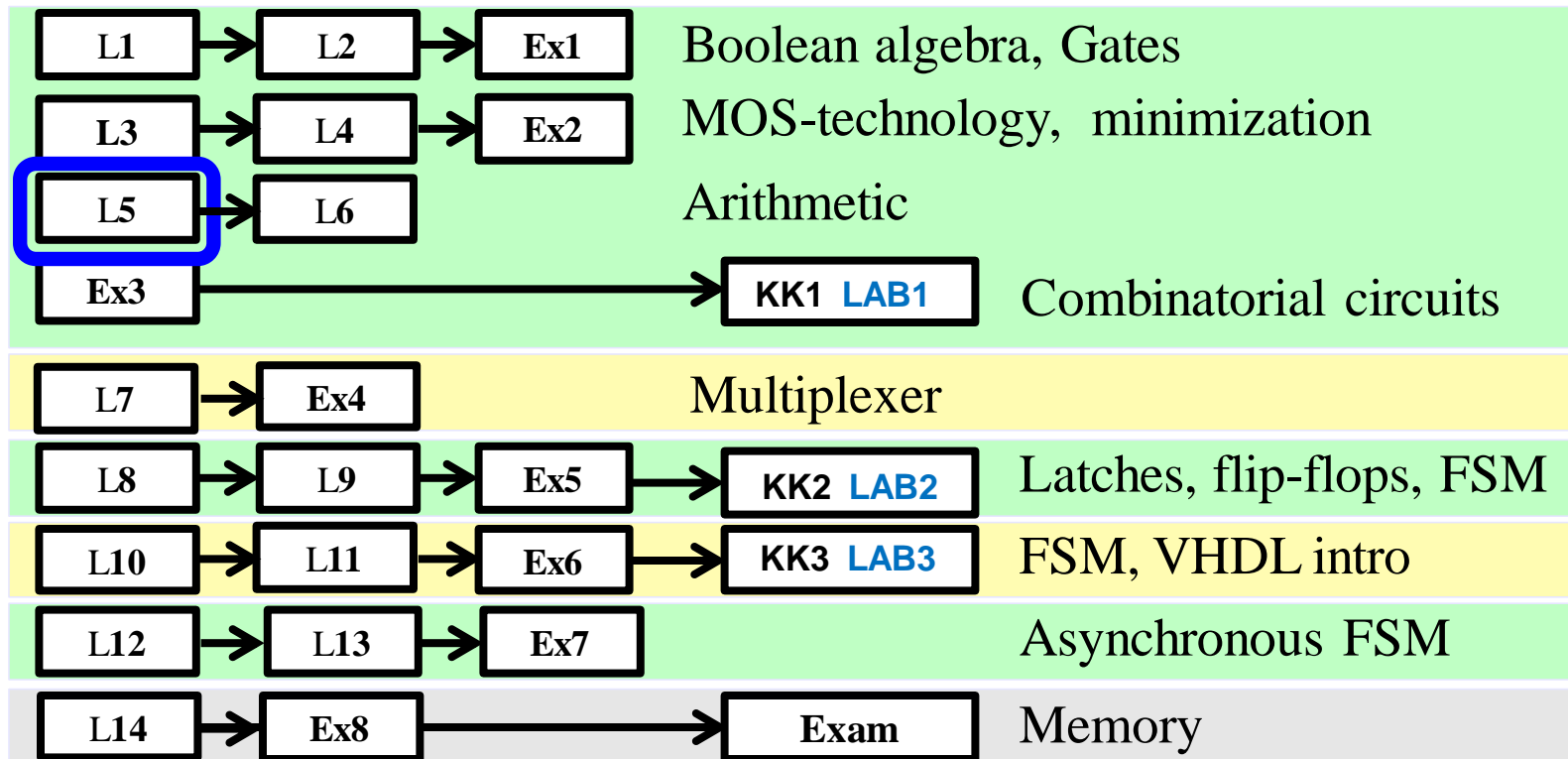
L5: Digital Arithmetic I

Masoumeh (Azin) Ebrahimi

KTH/ICT

mebr@kth.se

IE1204 Digital Design



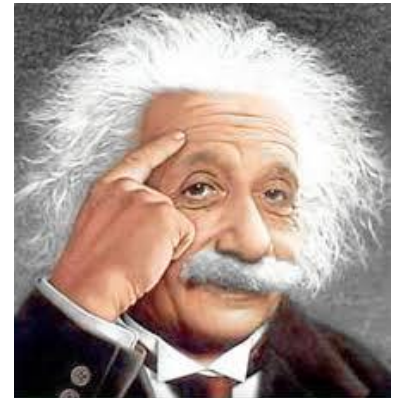
This lecture covers ...

- BV pp. 250-280

How to calculate?!



$$\begin{array}{r} 756 \\ \times 32 \\ \hline 12 \end{array}$$



What about our brain?

**Computers can do arithmetic operations amazingly
Fast and Accurate!**

Number representations

- A number can be represented in binary in many ways
- The most common types of numbers are:
 - Unsigned integers = can only be positive
 - Signed integers = can also be negative
 - sign-and-magnitude, 1's complement, 2's complement
 - Fixed-point numbers
 - Floating-point numbers

Unsigned integers

Unsigned integers:

2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	0	1

$$= 1 * 2^6 + 1 * 2^5 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0 = 109$$

But how do we represent negative numbers???

Sign-and-magnitude

Integer:

S	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
1	1	0	1	1	0	1

$$= - (1 * 2^5 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0) = - 45$$

↙ The magnitude (value) of the number

Bit representing the sign on the number (1 if "-", 0 if "+")

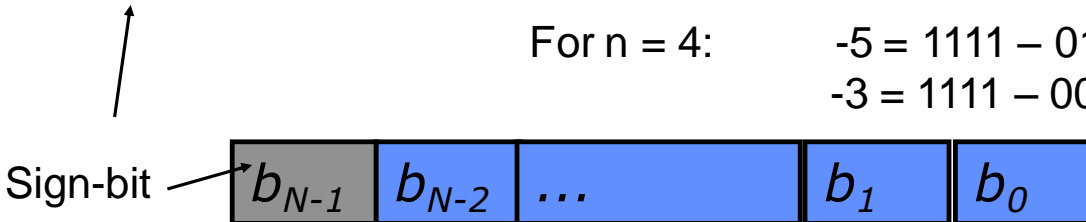
Advantage: simple concept

Disadvantages: two zeros (+/-); complex addition when operands have opposite signs (numbers have to be compared and then the smaller subtracted from the larger)

1	0	0	0	0	0	0
0	0	0	0	0	0	0

1's complement

X	\underline{X}	X_R
0	00...000	0
1	00...001	1
...
$2^{n-1}-1$	01...111	$2^{n-1}-1$
$-2^{n-1}-1$	10...000	2^{n-1}
...
-1	11...110	2^n-2
-0	11...111	2^n-1



In the 1's complement scheme, an n-bit negative number, K, is obtained by subtracting its equivalent positive number, P, from 2^n-1

$$K = (2^n-1) - P$$

Number range: $-(2^{n-1}-1) .. +(2^{n-1}-1)$

To compute 1's complement, all bits are complemented

For n = 4: $-5 = 1111 - 0101 = 1010$
 $-3 = 1111 - 0011 = 1100$

Disadvantages: two zeros; during addition correction is needed when there is a carry-out from the sign bit position (e.g. as in cases $(+5) + (-2)$ and $(-5) + (-2)$)

2's complement

X	<u>x</u>	X _R
0	00...000	0
1	00...001	1
...
2 ⁿ⁻¹ -1	01...111	2 ⁿ⁻¹ -1
-2 ⁿ⁻¹	10...000	2 ⁿ⁻¹
...
-2	11...110	2 ⁿ -2
-1	11...111	2 ⁿ -1

↑
Sign-bit

In the 2's complement scheme, an n-bit negative number, K, is obtained by subtracting its equivalent positive number, P, from 2ⁿ

$$K = 2^n - P$$

Number range: - (2ⁿ⁻¹) .. + (2ⁿ⁻¹-1)

To compute 2's complement, all bits are complemented and the number 1 is added (i.e. add 1 to 1's complement)

For n = 4:

$$-5 = 10000 - 0101 = 1011$$

$$-3 = 10000 - 0011 = 1101$$

Advantage: One zero

Disadvantage: Risk of overflow

The most common representation of signed integers

Number conversion - Positive to Negative numbers

01111 +15

10000 complement

10001 add 1

10001 -15

Number conversion - Negative to Positive numbers

10001 -15

01110 complement

01111 add 1

01111 +15

Group work

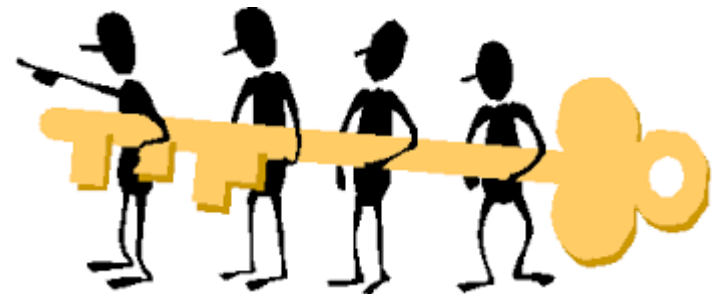
How to represent -10 in 2's complement scheme when $n=5$?

01010 +10

10101 complement

10110 add 1

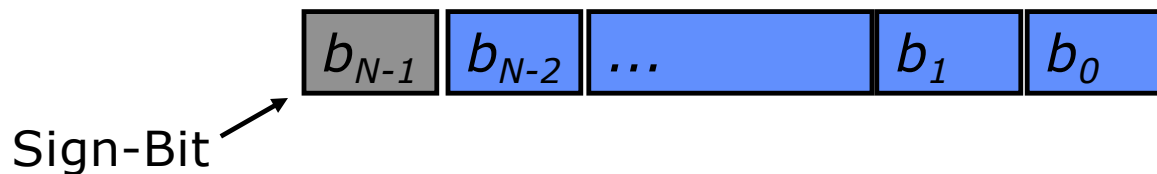
10110 -10



Integer (2's complement)

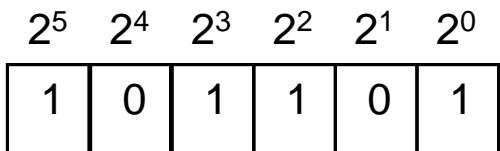
Representation with 2's complement

$$B = b_{N-1} b_{N-2} \dots b_1 b_0 \quad \text{where } b_i \in \{0, 1\}$$



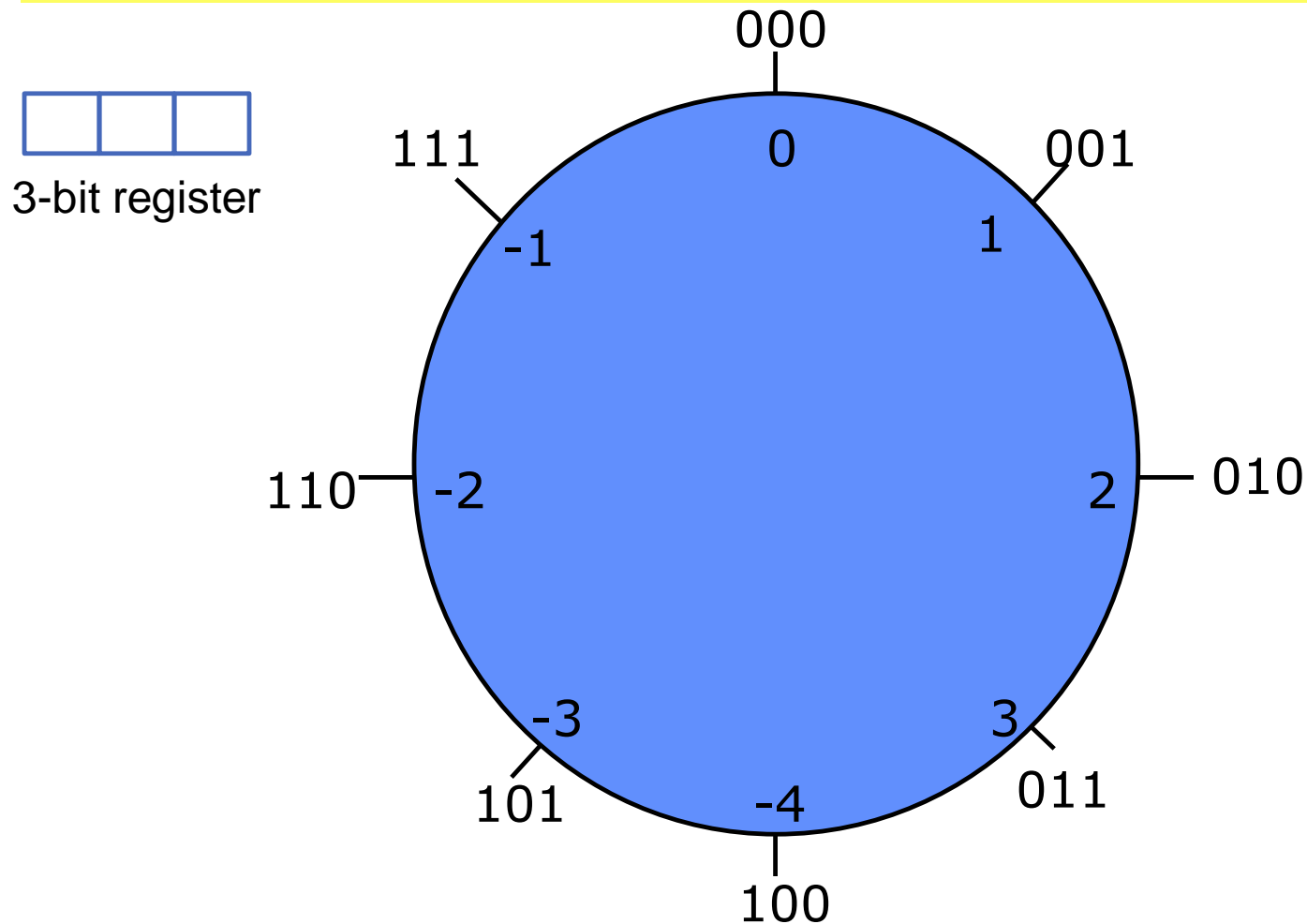
$$\text{Decimal: } D(B) = -b_{N-1} 2^{N-1} + b_{N-2} 2^{N-2} + \dots + b_1 2^1 + b_0 2^0$$

Sign-Bit



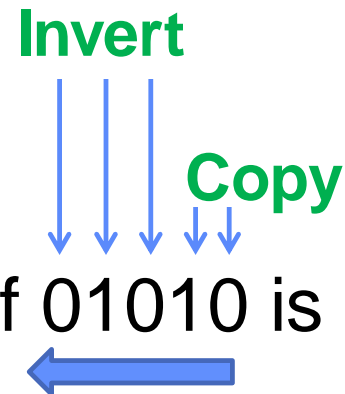
$$= -1 * 2^5 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0 = -19$$

Integers: (2's complement)



Rule for finding 2's complement

- In order to easily find 2's complement of a binary number B you can use the following procedure:
 - Start from the right-hand side
 - Copy all the bits of B that are 0 and the first bit that is 1
 - Complement all other bits



Example: 2's complement of 01010 is 10110

Sign-extension

Integer:

6-bit register

2^5	2^4	2^3	2^2	2^1	2^0
1	0	1	1	0	1

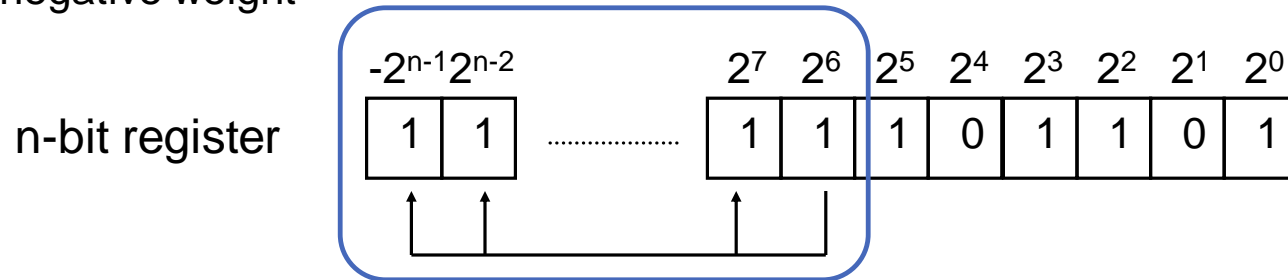
$$= -1 * 2^5 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0 = -19$$

7-bit register

-2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	1	1	0	1

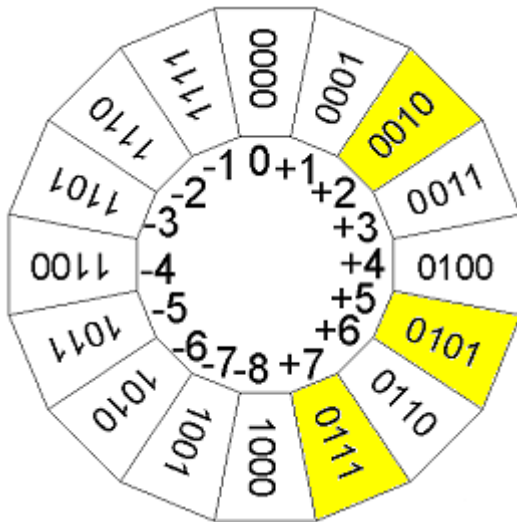
$$= -1 * 2^6 + 1 * 2^5 + 1 * 2^3 + 1 * 2^2 + 1 * 2^0 = -19$$

The sign bit has negative weight



To extend the range of numbers, add additional bit positions to the left of the sign bit, with the same value as the sign bit.

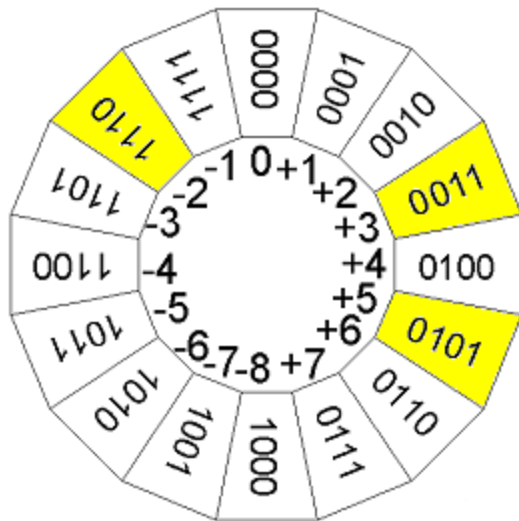
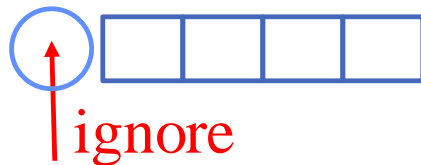
Addition using 2's complement (BV page 264)



$$\begin{array}{r} (+5) \\ + \quad (+2) \\ \hline (+7) \end{array}$$

$$\begin{array}{r} 0101 \\ + \quad 0010 \\ \hline 0111 \end{array}$$

Addition using 2's complement (BV page 264)

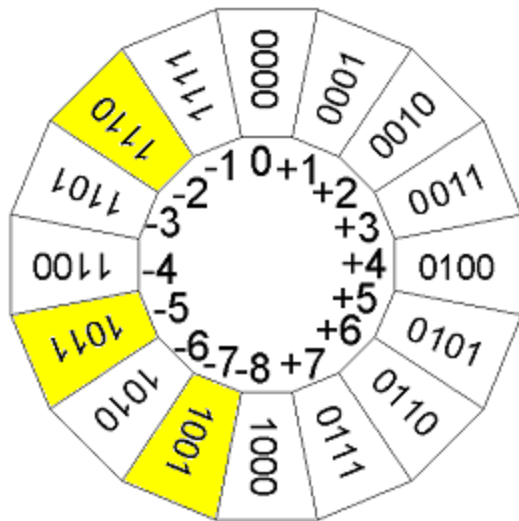
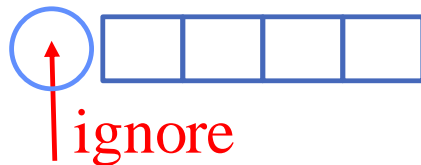


$$\begin{array}{r}
 (+5) \\
 + \quad (-2) \\
 \hline
 (+3)
 \end{array}
 \qquad
 \begin{array}{r}
 \textcolor{red}{1} \text{ } \overline{1} \\
 0101 \\
 + \quad 1110 \\
 \hline
 \textcolor{red}{1} 0011
 \end{array}$$

↑

Carry-bit could be ignored!

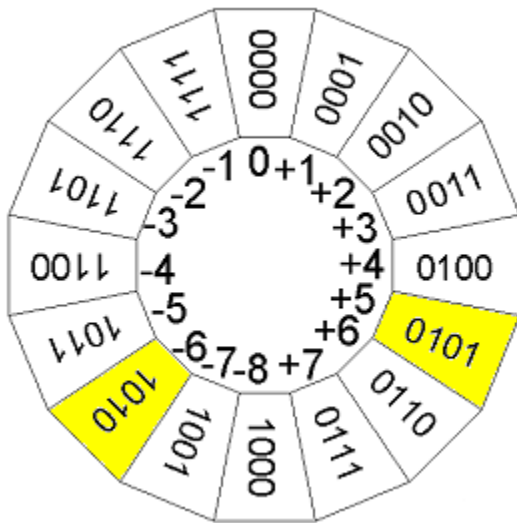
Addition using 2's complement (BV page 264)



$$\begin{array}{r}
 (-5) \\
 + \quad (-2) \\
 \hline
 (-7)
 \end{array}
 \qquad
 \begin{array}{r}
 \underline{1} \quad \underline{11} \\
 \quad \underline{1011} \\
 + \quad 1110 \\
 \hline
 \quad \underline{1} \quad 1001
 \end{array}$$

Carry-bit could be ignored!

Overflow

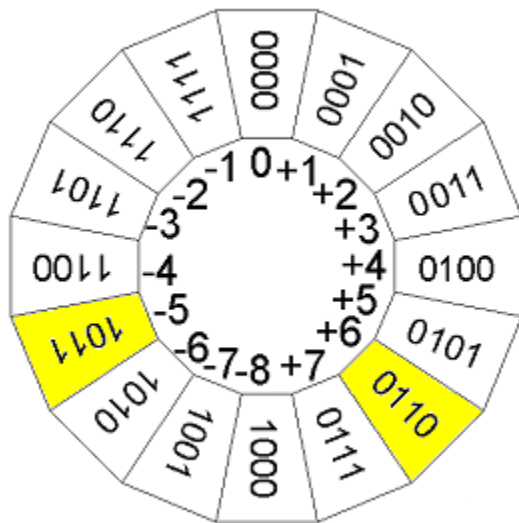
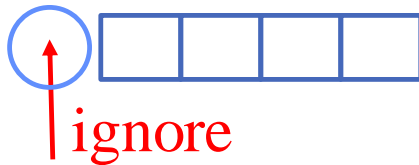


$$\begin{array}{r} (+5) \\ + (+5) \\ \hline (-6) \end{array}$$

$$\begin{array}{r} \overset{1}{\overline{0}}\overset{1}{\overline{1}}\overset{1}{\overline{0}}\overset{1}{\overline{1}} \\ + \overset{1}{\overline{0}}\overset{1}{\overline{1}}\overset{1}{\overline{0}}\overset{1}{\overline{1}} \\ \hline \overset{1}{\overline{1}}\overset{1}{\overline{0}}\overset{1}{\overline{1}}\overset{1}{\overline{0}} \end{array}$$

Overflow - the sign bit does not match with input numbers ...

Overflow (2)



$$\begin{array}{r} (-5) \\ + (-5) \\ \hline (+6) \end{array}$$

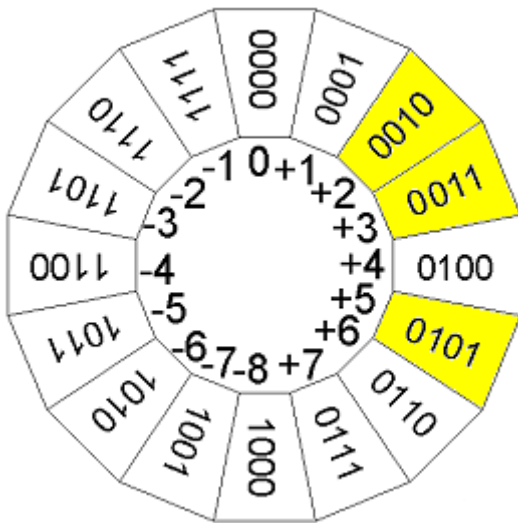
$$\begin{array}{r} \overset{1}{\text{1}} \quad \overset{1}{\text{1}} \quad \overset{1}{\text{1}} \\ \text{1011} \\ + \text{1011} \\ \hline \overset{1}{\text{1}} \quad \boxed{0} \quad \text{110} \end{array}$$

↑

Carry-bit could be ignored!

Overflow - the sign bit does not match with input numbers ...

Subtraction using 2's complement (BV page 265)



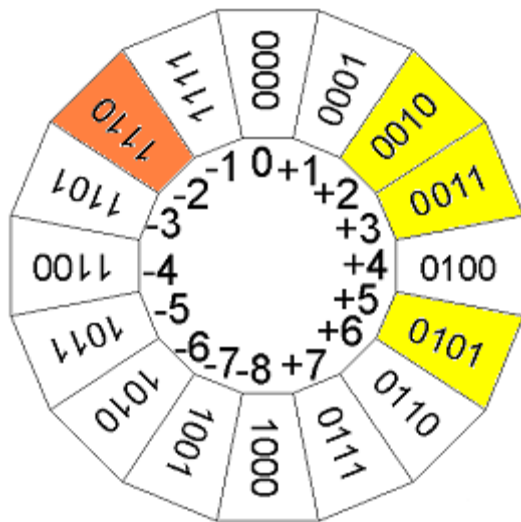
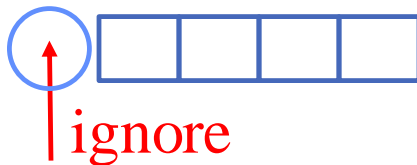
$$\begin{array}{r} (+5) \\ - (+2) \\ \hline (+3) \end{array}$$

"Borrow" one

$$\begin{array}{r} 10 \\ \text{0} \cancel{\text{1}} \text{01} \\ - 0010 \\ \hline \text{????} \end{array}$$

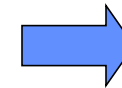
How do you do the subtraction in an easy way?

Subtraction using 2's complement (BV page 265)



$$\begin{array}{r} (+5) \\ - (+2) \\ \hline (+3) \end{array}$$

$$\begin{array}{r} 0101 \\ - 0010 \\ \hline \text{????} \end{array}$$

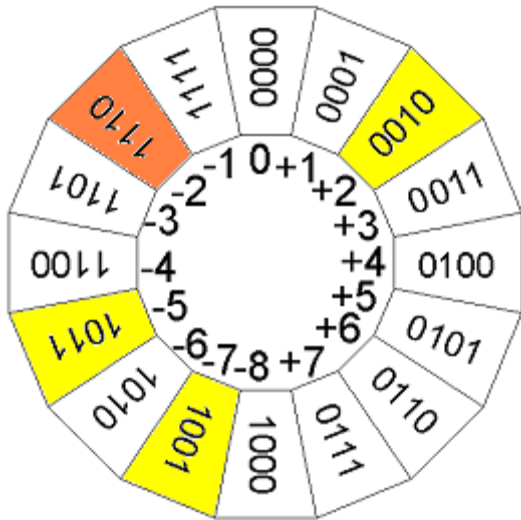
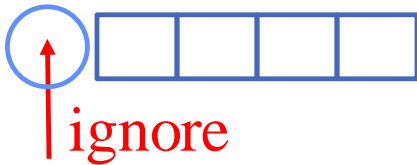


$$\begin{array}{r} \text{1} \text{1} \\ \text{0101} \\ + 1110 \\ \hline \text{1} \text{0011} \end{array}$$

Carry-bit could be ignored!

Making an addition of 2's complement instead!

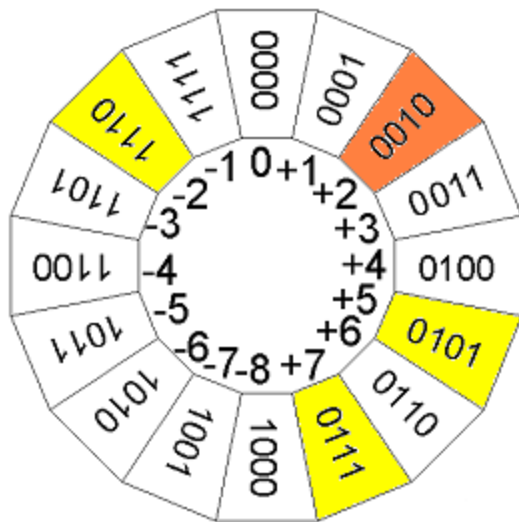
Subtraction using 2's complement (BV page 265)



$$\begin{array}{r}
 (-5) \\
 - \quad (+2) \\
 \hline
 (-7)
 \end{array}
 \quad
 \begin{array}{r}
 1011 \\
 - \quad 0010 \\
 \hline
 \text{????}
 \end{array}
 \rightarrow
 \begin{array}{r}
 \textcolor{red}{1} \overline{11} \overline{1} \\
 \phantom{\overline{11}} \phantom{\overline{1}} 1011 \\
 + \quad 1110 \\
 \hline
 \textcolor{red}{1} 1001
 \end{array}$$

Carry-bit could be ignored!

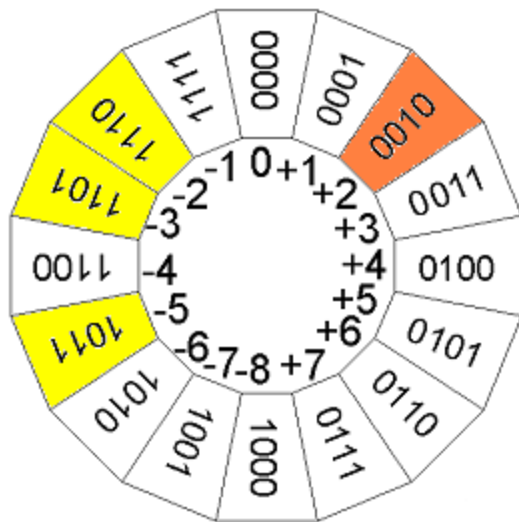
Subtraction using 2's complement (BV page 265)



$$\begin{array}{r}
 (+5) \\
 - (-2) \\
 \hline
 (+7)
 \end{array}
 \quad
 \begin{array}{r}
 0101 \\
 - 1110 \\
 \hline
 \text{????}
 \end{array}
 \quad
 \begin{array}{r}
 0101 \\
 + 0010 \\
 \hline
 0111
 \end{array}$$



Subtraction using 2's complement (BV page 265)



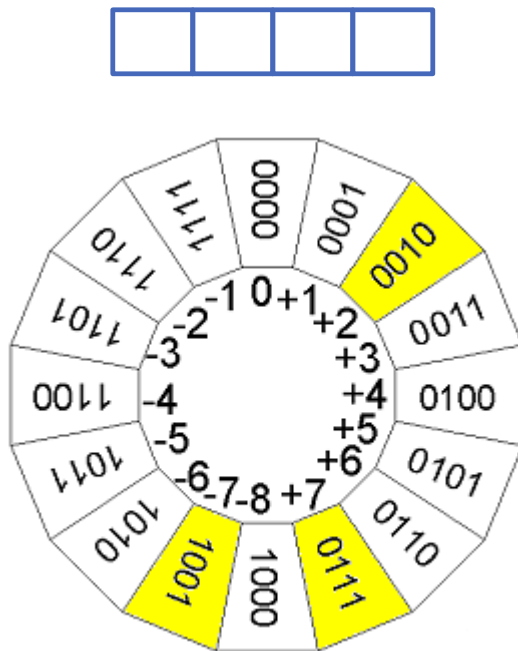
$$\begin{array}{r}
 (-5) \\
 - \quad (-2) \\
 \hline
 (-3)
 \end{array}
 \quad
 \begin{array}{r}
 1011 \\
 - \quad 1110 \\
 \hline
 \text{????}
 \end{array}
 \quad
 \begin{array}{r}
 \overset{1}{\overline{1011}} \\
 + \quad 0010 \\
 \hline
 1101
 \end{array}$$



2's complement representation, a summary

- **Range:** -2^{N-1} up to $2^{N-1} - 1$
- **Negation:** Complement each bit (the Boolean complement), then add 1
- **Expansion of bit-length:** Add additional bit positions to the left of the sign bit, with the same value as the sign bit
- **Overflow rule** If two numbers with the same sign are added, there is an overflow if the result has an opposite sign
- **Subtraction rule:** To subtract B from A, take the 2's complement of B and add it to A.

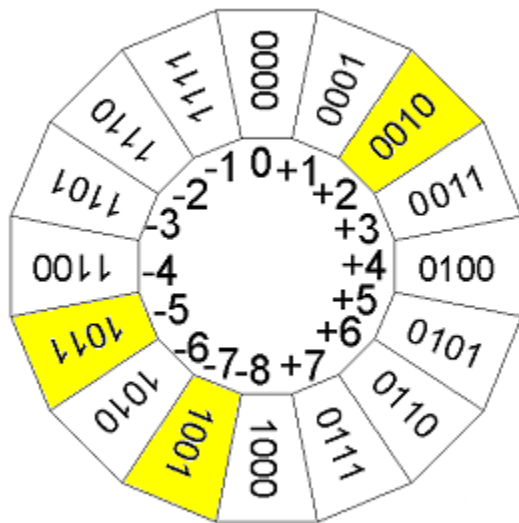
Alternative way to detect overflow (BV page 271)



$$\begin{array}{r}
 (+7) \\
 + \quad (+2) \\
 \hline
 (-7)
 \end{array}
 \qquad
 \begin{array}{r}
 c_4=0 \quad c_3=1 \\
 \begin{array}{c}
 0 \quad 1 \quad 1 \\
 \hline
 0111 \\
 + \quad 0010 \\
 \hline
 1001
 \end{array}
 \end{array}$$

Overflow because c_4 and c_3 are different!

Alternative way to detect overflow (BV page 271)

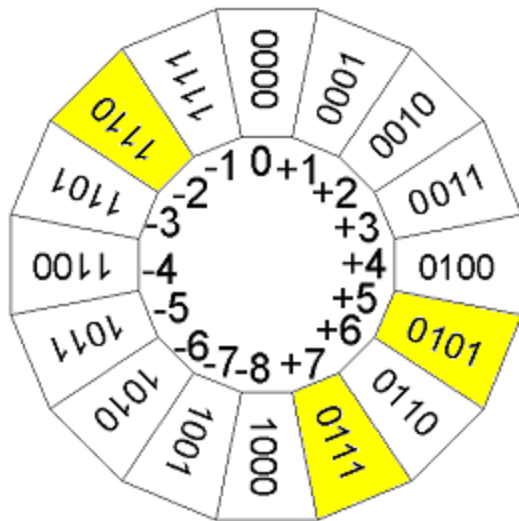
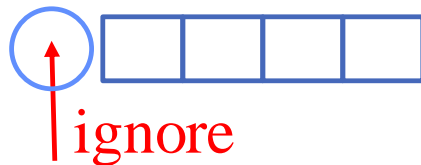


$$\begin{array}{r} (-7) \\ + (+2) \\ \hline (-5) \end{array}$$

$$\begin{array}{r} c_4=0 \quad c_3=0 \\ \quad \underline{0} \quad \underline{0} \\ \quad 1001 \\ + \quad 0010 \\ \hline 1011 \end{array}$$

No Overflow because c_4 and c_3 are the same!

Alternative way to detect overflow (BV page 271)



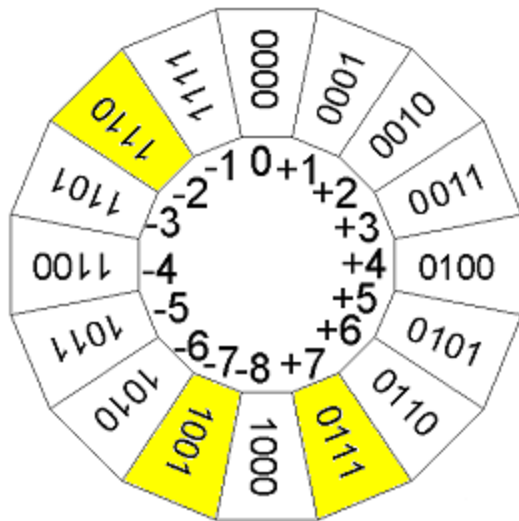
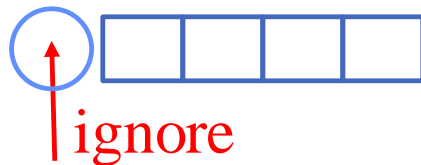
$$\begin{array}{r} (+7) \\ + (-2) \\ \hline (+5) \end{array}$$

$$\begin{array}{r} c_4=1 \quad c_3=1 \\ \underline{1} \quad \underline{1} \quad \underline{1} \\ 0111 \\ + 1110 \\ \hline 1 \quad 0101 \end{array}$$

Carry-bit could be ignored!

No Overflow because c_4 and c_3 are the same!

Alternative way to detect overflow (BV page 271)



$$\begin{array}{r} (-7) \\ + (-2) \\ \hline (+7) \end{array}$$

$$\begin{array}{r} c_4=1 \quad c_3=0 \\ \underline{1} \quad \underline{0} \\ 1001 \\ + 1110 \\ \hline 1 \quad 0111 \end{array}$$

Carry-bit could be ignored!

Overflow because c_4 and c_3 are different!

Logic for detecting overflow

- For 4-bit numbers
 - Overflow if c_3 and c_4 are different
 - Otherwise, there is no overflow

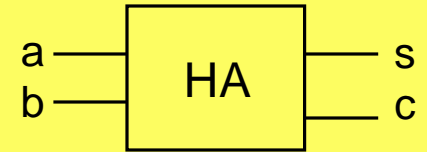
$$\text{Overflow} = c_3 \bar{c}_4 + \bar{c}_3 c_4 = c_3 \oplus c_4$$

- For n -bit numbers

$$\text{Overflow} = c_{n-1} \oplus c_n$$

Hardware arithmetic

Half Adder



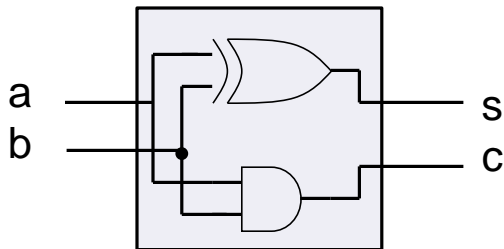
	<u>c</u>		a	b	c	s
	0	a	0	0	0	0
+	0	b	0	1	0	1
<hr/>			1	0	0	1
	c	s	1	1	1	0

a		0	1
b	0	0	0
	1	0	1

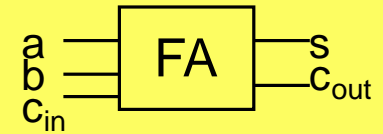
$$c = a b$$

a		0	1
b	0	0	1
	1	1	0

$$s = a \oplus b$$



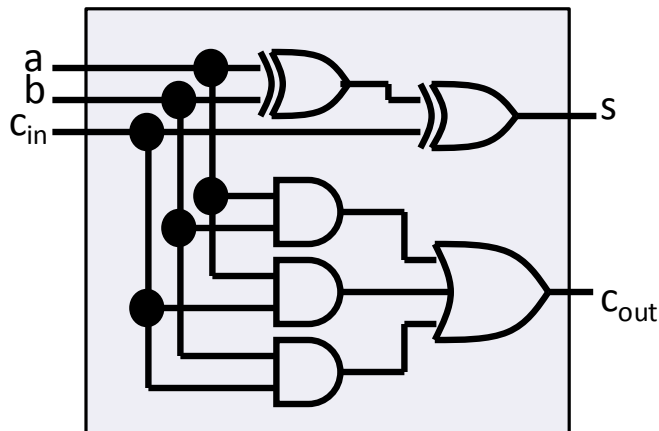
Full Adder



<u>C_{out}</u> <u>C_{in}</u>	a	b	c _{in}	c _o	s
0 a	0	0	0	0	0
0 b	0	0	1	0	1
	0	1	0	0	1
	0	1	1	1	0
C _{out} S	1	0	0	0	1
	1	0	1	1	0
	1	1	0	1	0
	1	1	1	1	1

ab \ c _{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

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

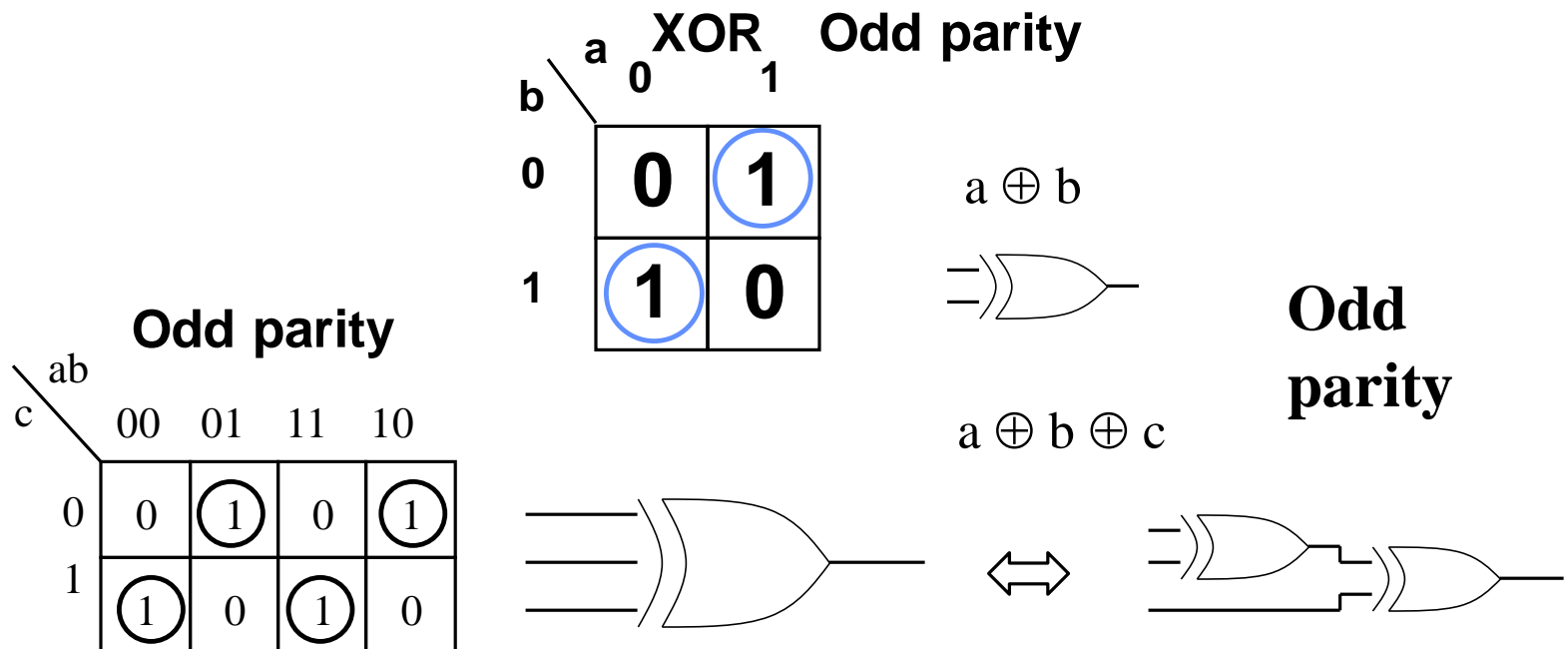


ab \ c _{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

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

Sum function = Odd parity

The Full Adder sum function is the "odd" parity function. This is the XOR function's natural extension to more variables than two. Odd parity is when the number of 1's on the inputs is an odd number.



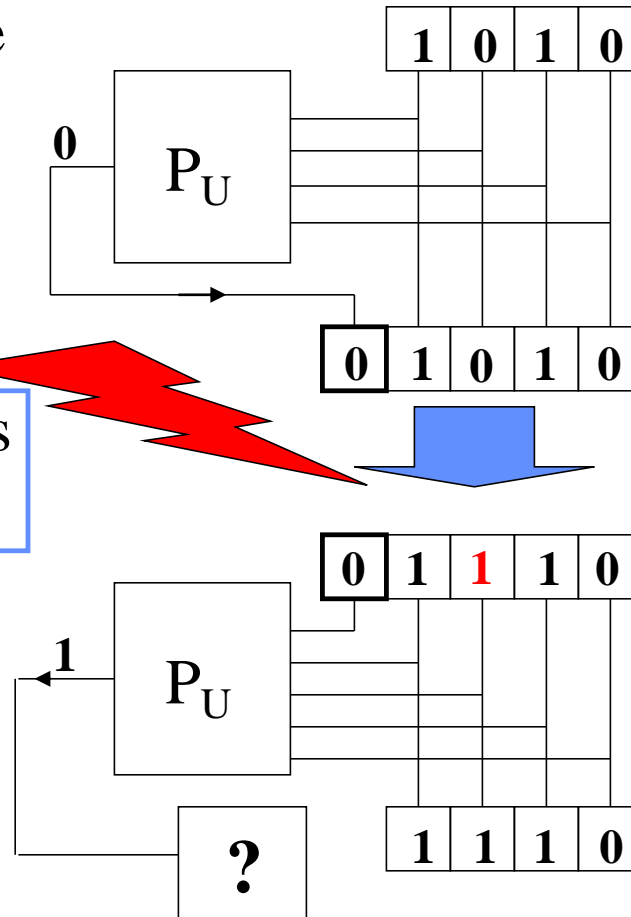
Parity check

With parity function we can check whether data has been disturbed or not.

Disturbance!

Data transmitted always have even parity!

ALARM!? A bit modified!? Data has been disturbed!

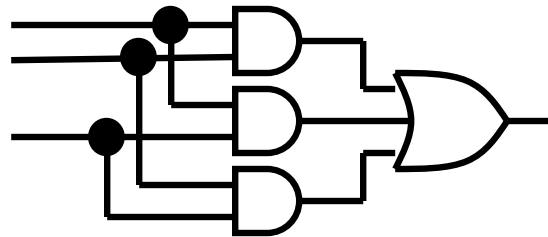


Data - original

Paritybit
is added

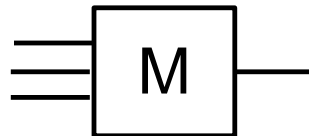
Parity Check
Checks if an
odd number of
"1s" is received
– error!

Carry function = Majority function



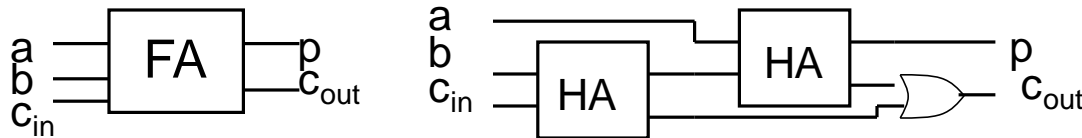
	00	01	11	10
0	0	0	1	0
1	0	1	1	1

- **Majority function.** Output assumes same value 1/0 as a majority of the inputs.



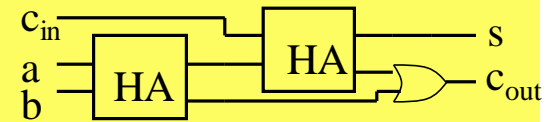
Full adder composed of half-adders

- We may also construct a full adder with the help of two half-adder and an OR gate



- Composition allows us to construct new systems using known building blocks

Full adder



a	b	c_{in}	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

	ab			
c_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

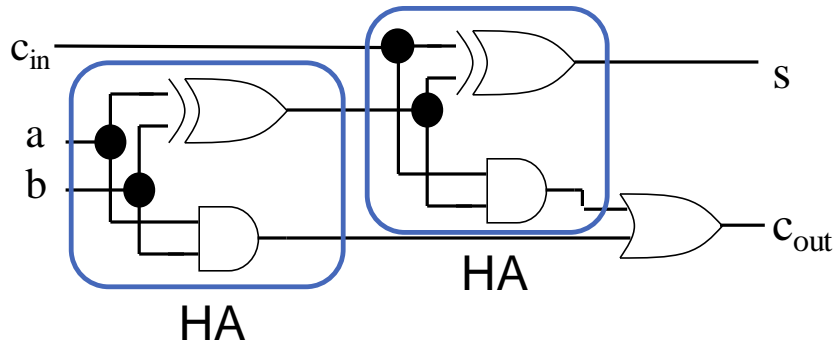
$$c_{in}(a \oplus b)$$

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

	ab			
c_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$c_{out} = ab + c_{in}a + c_{in}b$$

$$= ab + c_{in}(a+b)$$



	ab			
c_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

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

Popular tattoo?



Tattoos are forever! Unfortunately this is not the "best" adder, not if you want fast computers. Exciting continuation of adder circuits follow ...

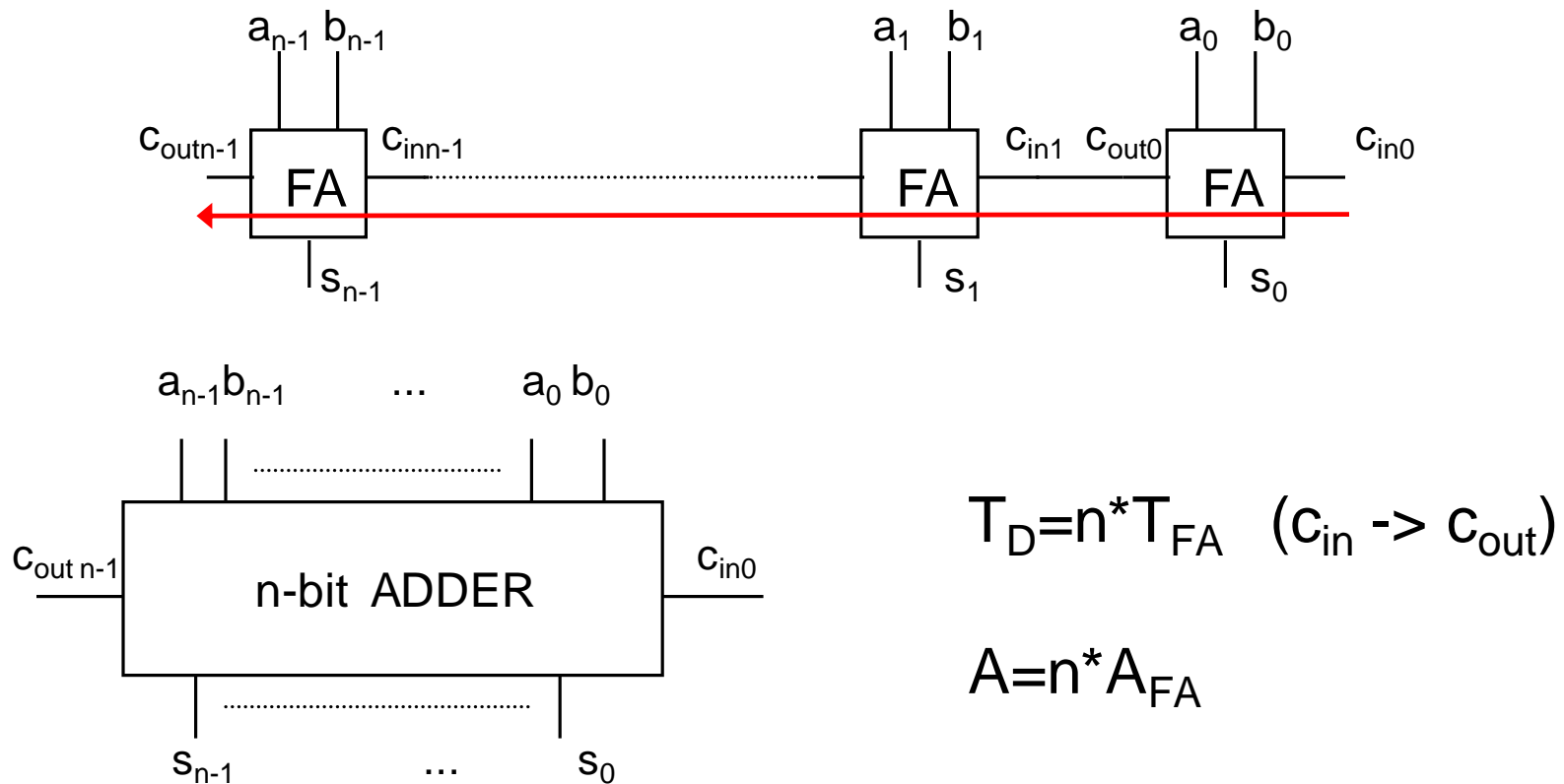
Different Adder Structures

- Ripple-Carry Adder (RCA)
- Carry-Lookahead Adder (CLA)
- Carry-Select Adder (CSA)

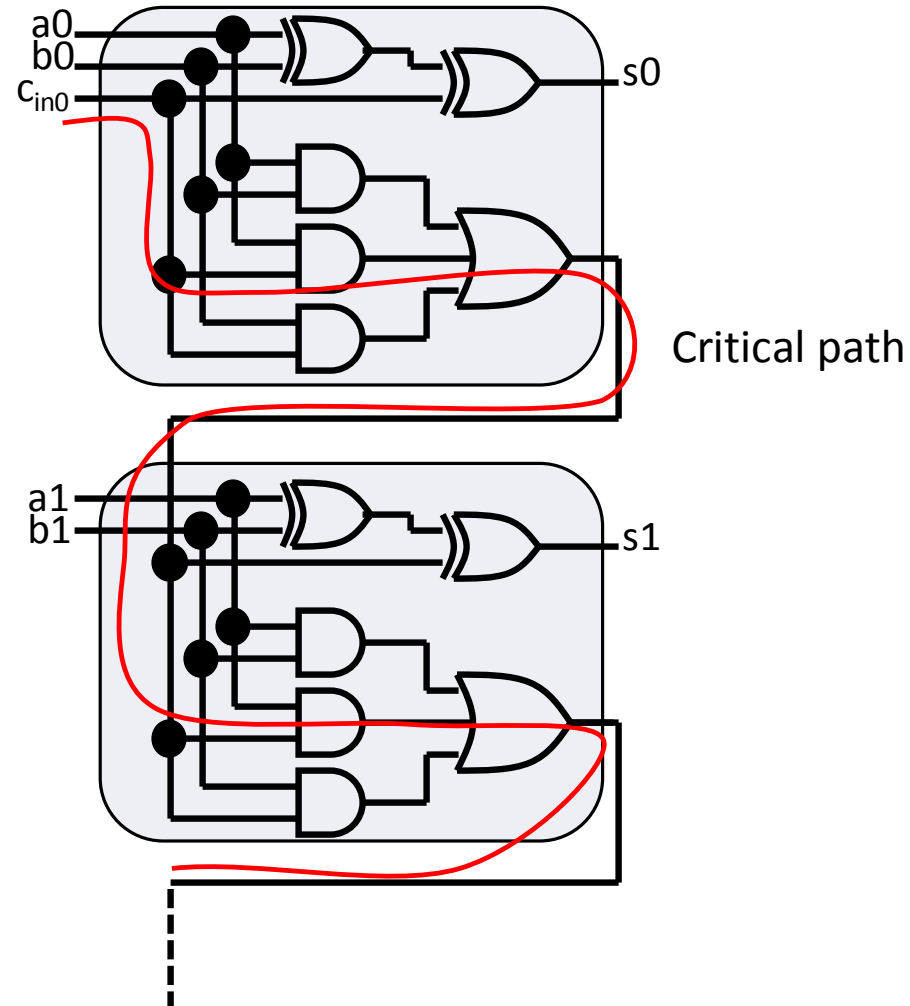
More on composition

- The composition can also be used to construct n -bit adders
- One need n full adders to construct one n -bit adder

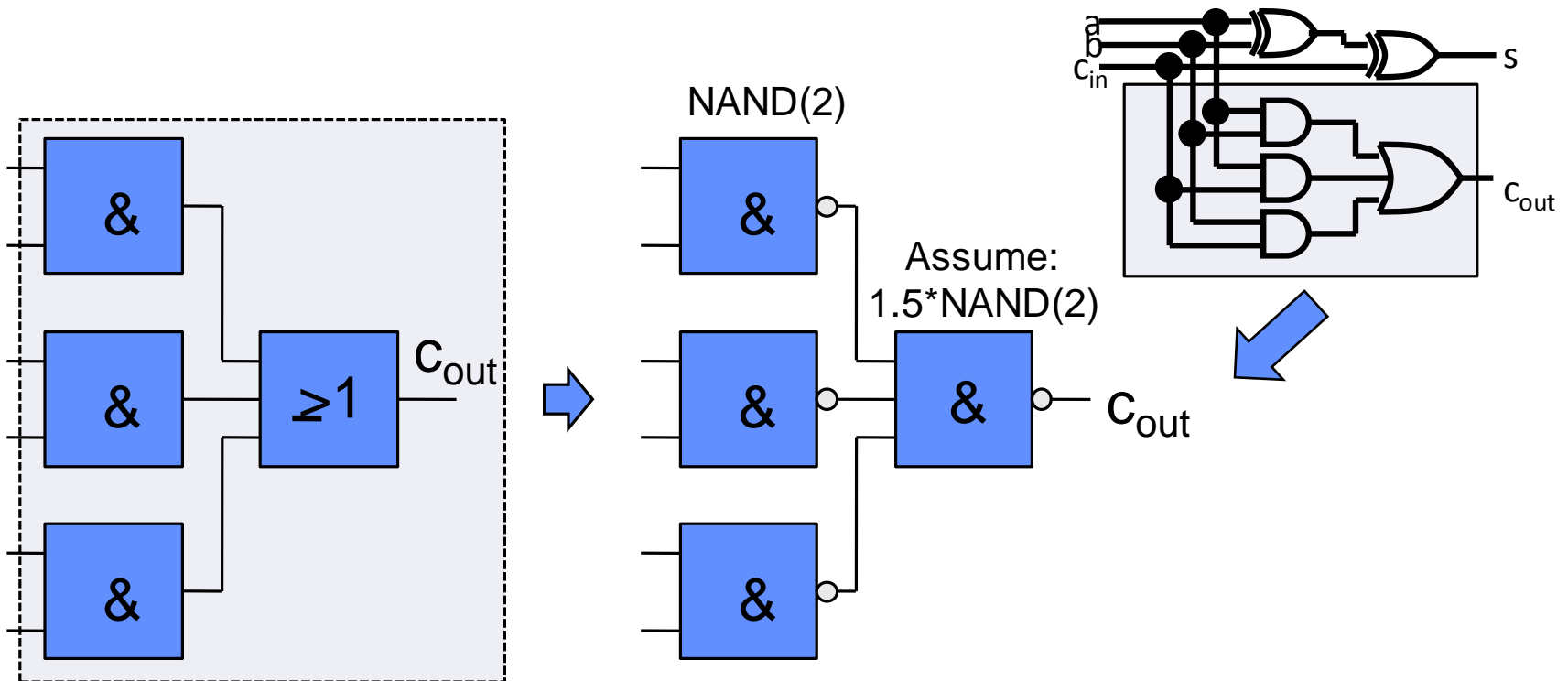
Ripple Carry Adder (RCA)



Ripple Carry Adder (RCA)



AND-OR(3) with NAND gates

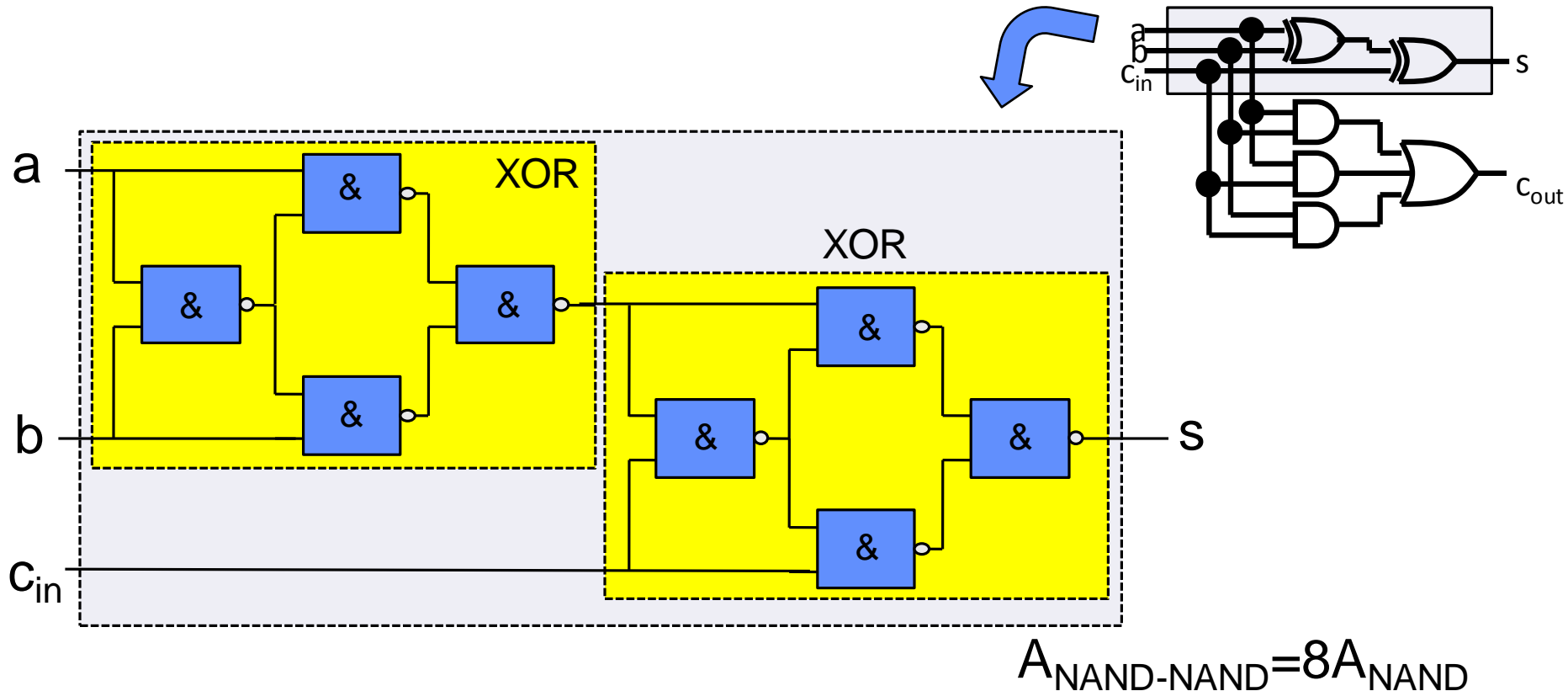


Carry-out function is in the critical path and thus the delay is important!

$$A_{\text{NAND-NAND}} = 4.5 * A_{\text{NAND}}$$

$$T_{\text{NAND-NAND}} = 2.5 * T_{\text{NAND}}$$

XOR(2) with NAND gates



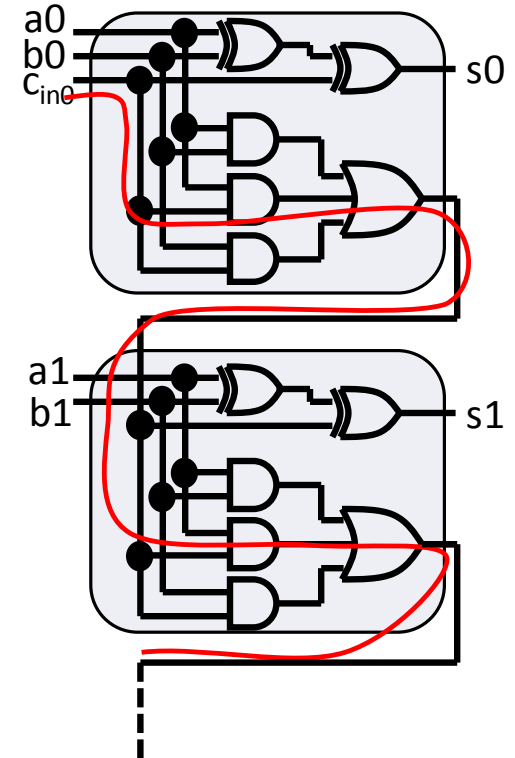
Sum function is not in the critical path
and thus the delay is not important!

$$T_{\text{NAND}} = 6 * T_{\text{NAND}}$$

Area and delay Calculations

Ripple-Carry Adder:

- $T = n * T_{FA} = n * (2.5 * T_{NAND})$
- $A = n * A_{FA} = n * (4.5 * A_{NAND} + 8A_{NAND}) = n * (12.5A_{NAND})$



Can we construct a faster adder?

- The delay of a Ripple Carry Adder (RCA) grows proportionally to the number of bits
- For 32 bits, the delay will be very large

How to reduce the critical path?
Carry-Look-ahead Adder (CLA)

Carry-out function

- Carry-out function for stage i can be expressed as

$$C_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

- We can factor this expression as

$$C_{i+1} = x_i y_i + (x_i + y_i) c_i$$

$$\text{or } C_{i+1} = x_i y_i + (x_i \oplus y_i) c_i$$

- And then re-write it as

$$C_{i+1} = g_i + p_i c_i, \text{ where}$$

$$g_i = x_i y_i \quad (\text{generate function})$$

$$p_i = x_i + y_i \quad (\text{propagate function})$$

$$p_i = x_i \oplus y_i \quad (\text{propagate function})$$

xy \ c _{in}		00	01	11	10
		0	1	1	0
0		0	0	1	0
1		0	1	1	1

$C_{out} = xy + c_{in}x + c_{in}y$

xy \ c _{in}		00	01	11	10
		0	1	1	0
0		0	0	1	0
1		0	1	1	1

Delay of carry-look-ahead adder

- Carry-bit c_0

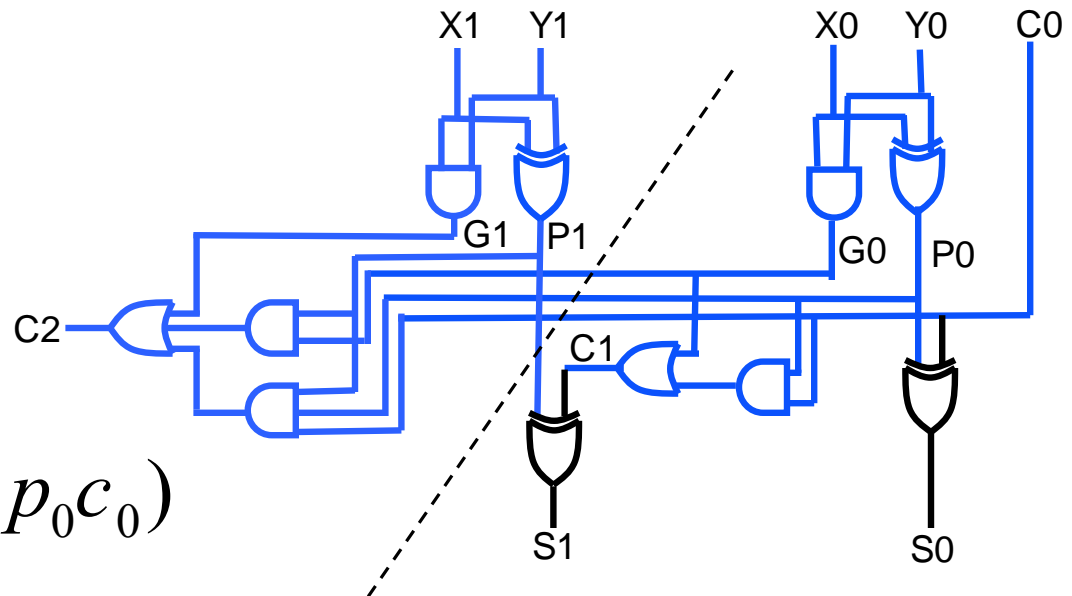
$$c_1 = g_0 + p_0 c_0$$

- Carry-bit c_1

$$c_2 = g_1 + p_1 c_1$$

$$= g_1 + p_1 (g_0 + p_0 c_0)$$

$$= g_1 + p_1 g_0 + p_1 p_0 c_0$$



c_2 will be produced after 3 gate delays
 s_2 will be produced after 4 gate delays

Area of carry-look-ahead adder

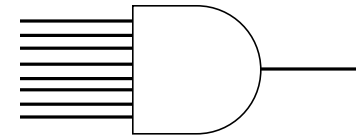
$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$

...

$$\begin{aligned} c_8 = & g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + \\ & + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + \\ & + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + \boxed{p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0} \end{aligned}$$

Ooops!



C_8 will be produced after 3 gate delays (larger gates)

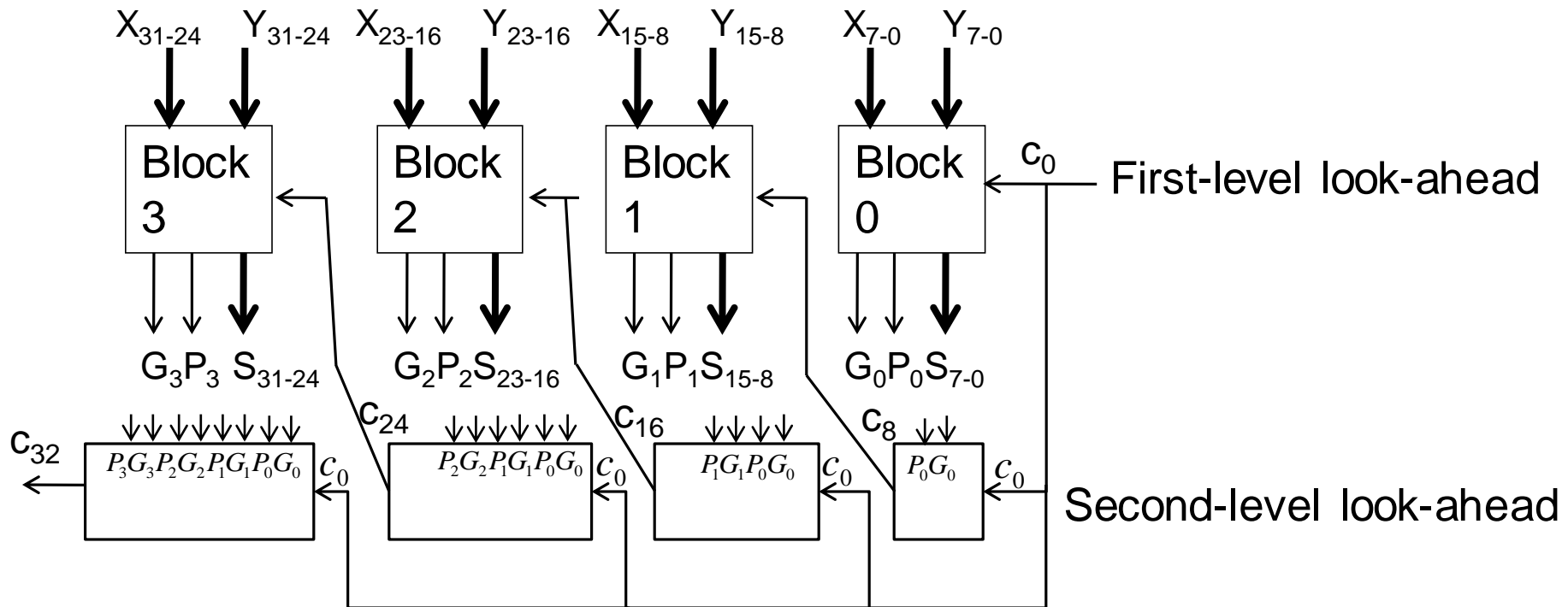
S_8 will be produced after 4 gate delays (larger gates)

$\text{Area}(n = 8) = \text{VERY BIG!}$

Hierarchical carry-look-ahead adder

- To reduce the area, we can use *hierarchical* approach
 - To design a 32-bit adder, we divide it into 4 eight-bit blocks and implement each block as an 8-bit carry look-ahead adder
 - A second-level carry-look-ahead can be used to produce quickly the carry between the blocks

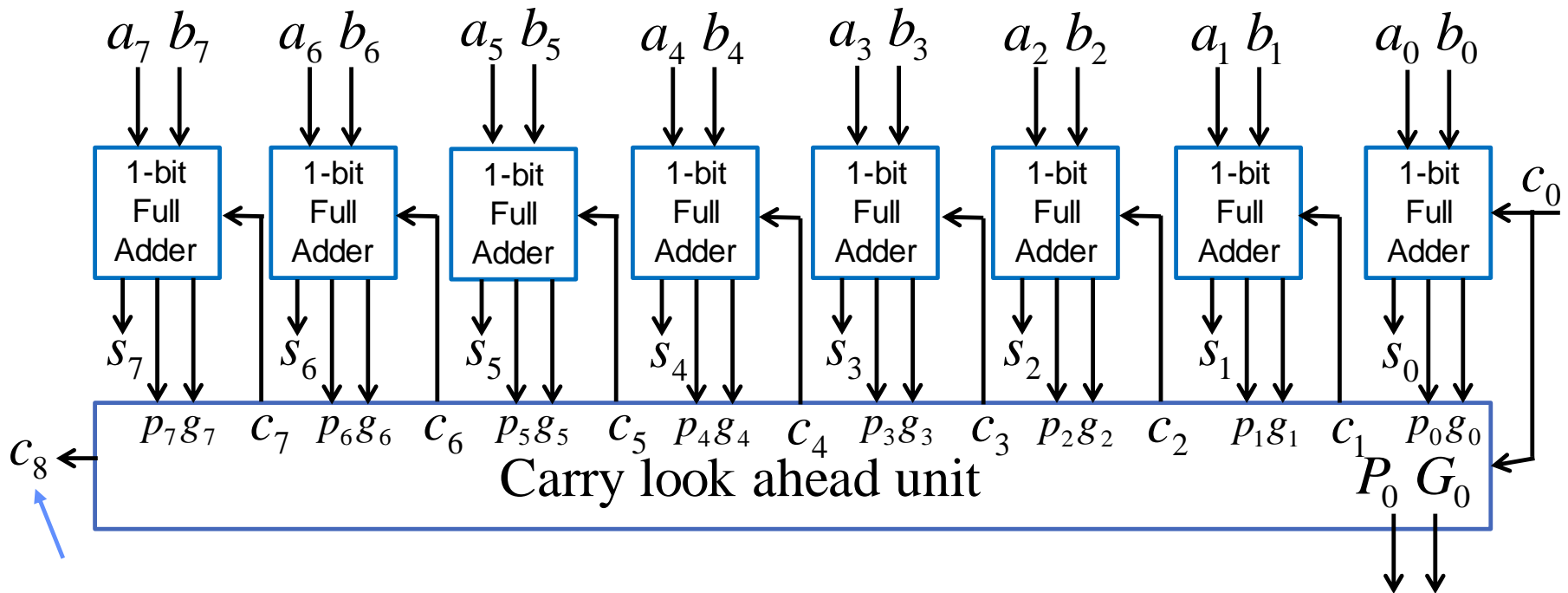
Hierarchical expansion (BV page 277)



$$P_0 = p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 \quad G_0 = g_7 + p_7 g_6 + p_7 p_6 g_5 + \dots + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0$$

Hierarchical expansion (BV page 277)

$$c_8 s_7 s_6 s_5 s_4 s_3 s_2 s_1 s_0 = a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0 + b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$$

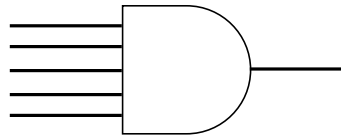


$$c_8 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 + p_7 p_6 p_5 p_4 g_3 + p_7 p_6 p_5 p_4 p_3 g_2 + p_7 p_6 p_5 p_4 p_3 p_2 g_1 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 g_0 + p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0 c_0$$

Hierarchical expansion (2)

- $c_8 = G_0 + P_0 c_0$
- $c_{16} = G_1 + P_1 G_0 + P_1 P_0 c_0$
- $c_{24} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$
- $c_{32} = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 +$
 $P_3 P_2 P_1 P_0 c_0$

etc.



Carry-Select Adder (CSA)

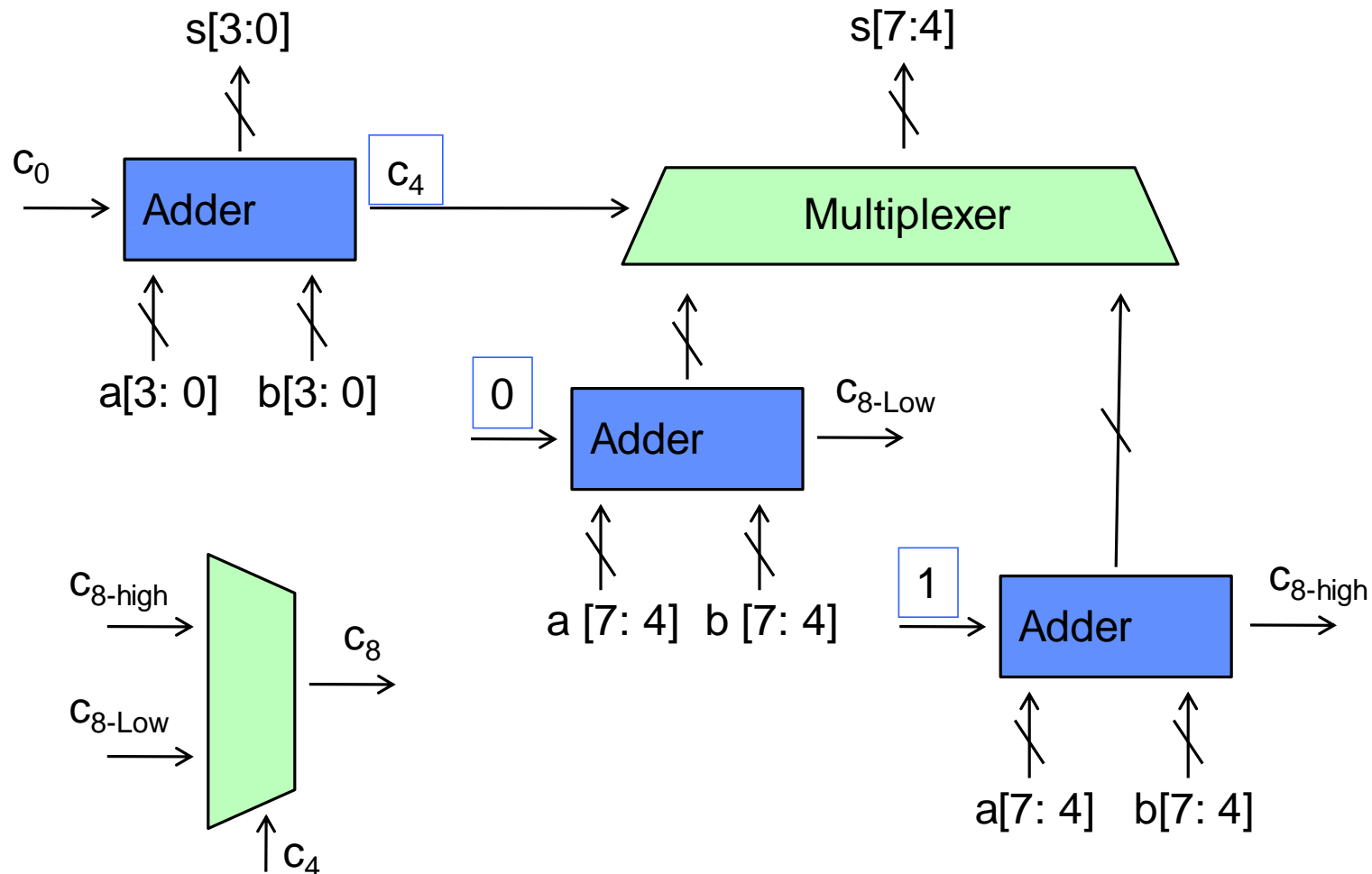
- Ripple Carry Adder (RCA) is simple but slow!
- Carry-Look-ahead Adder (CLA) is fast but complex with large area!

Any other alternative?
Carry-Select Adder (CSA)

Carry-Select-Adder (CSA)

- Idea
 - Divide an adder in two stages with the same number of bits
 - To speed up the process, the result of the second step is estimated in advance for two cases
 - Carry-in = 0
 - Carry-in = 1
 - When the calculation of the carry bit is completed for the first step, one chooses a result of the second step depending on carry-bit value

8-bit Carry-Select Adder



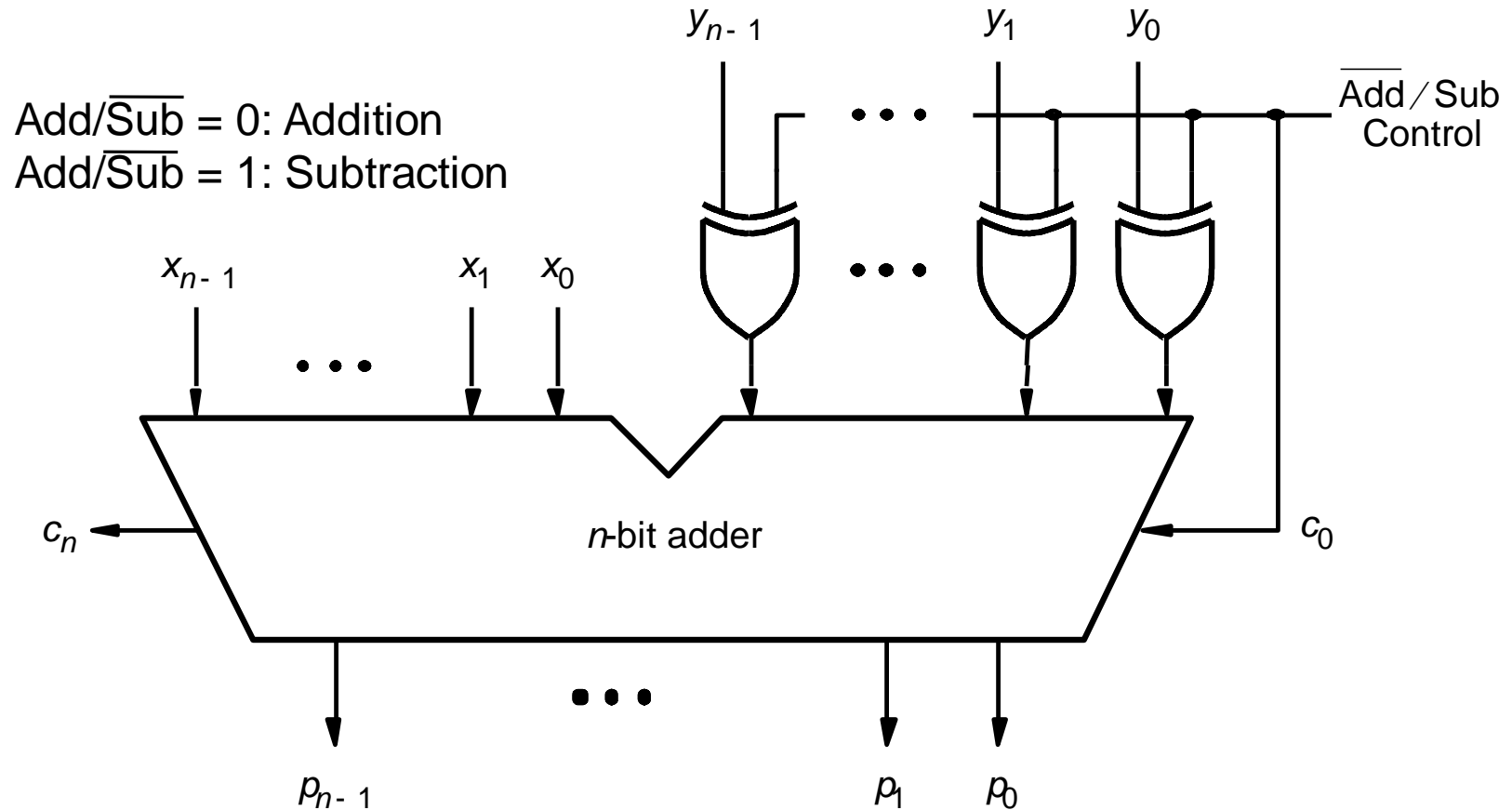
Which adder is the best?

- There is no clear answer!
 - Ripple adder takes the minimum area, but it is slow
 - Carry-look-ahead adder takes a lot of area, but it is fast
 - Carry-select adder is a compromise
- One must make a *trade-off* between area and speed

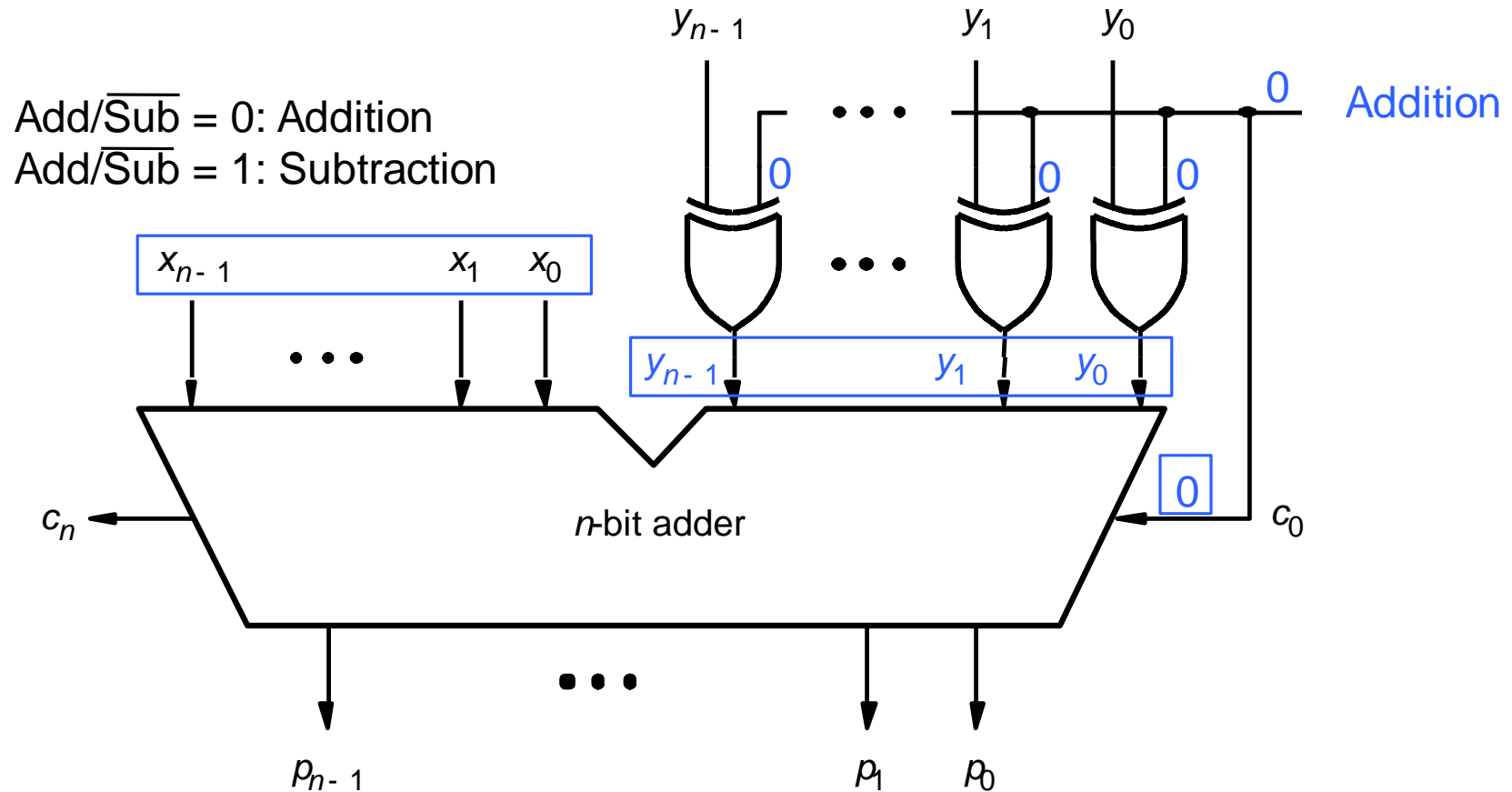
Subtraction

- Subtraction can be done by addition with 2's complement
 - Complement all bits of the second operand
 - Add 1

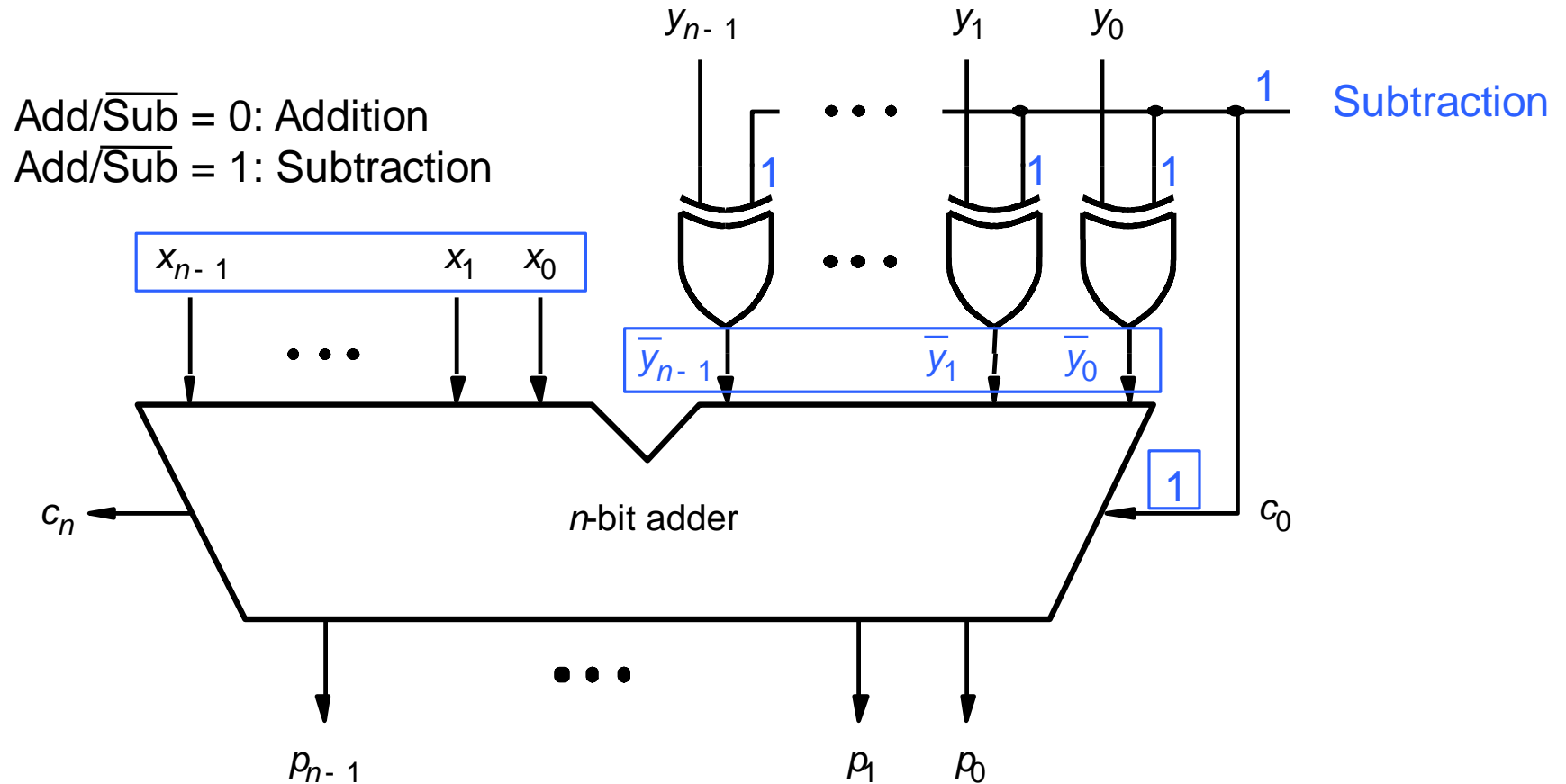
Add / Subtract



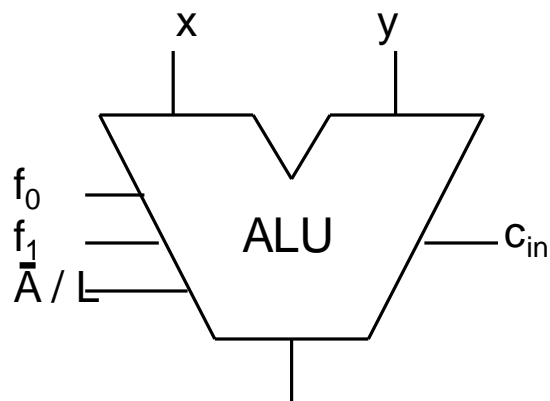
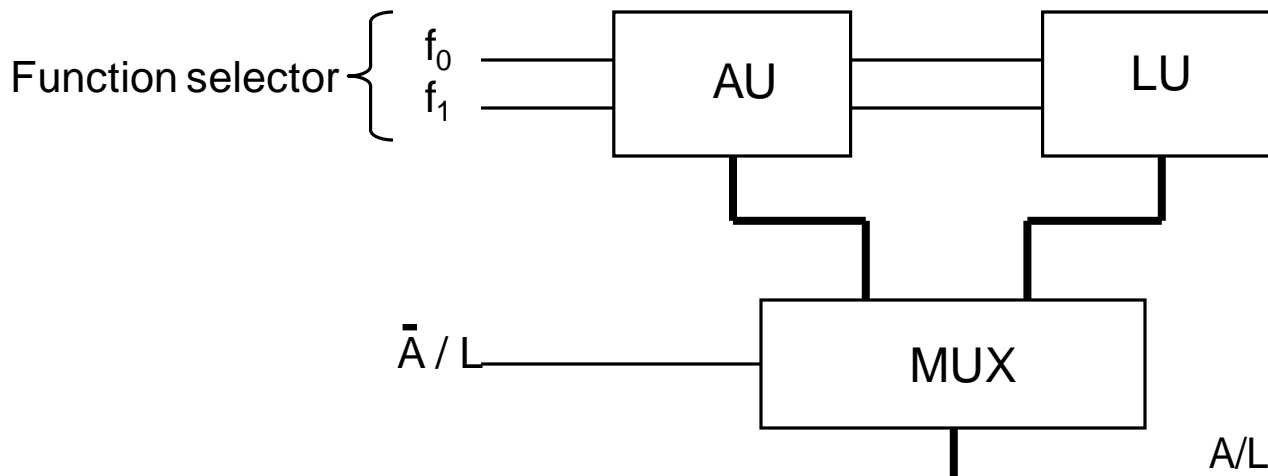
Add



Subtract

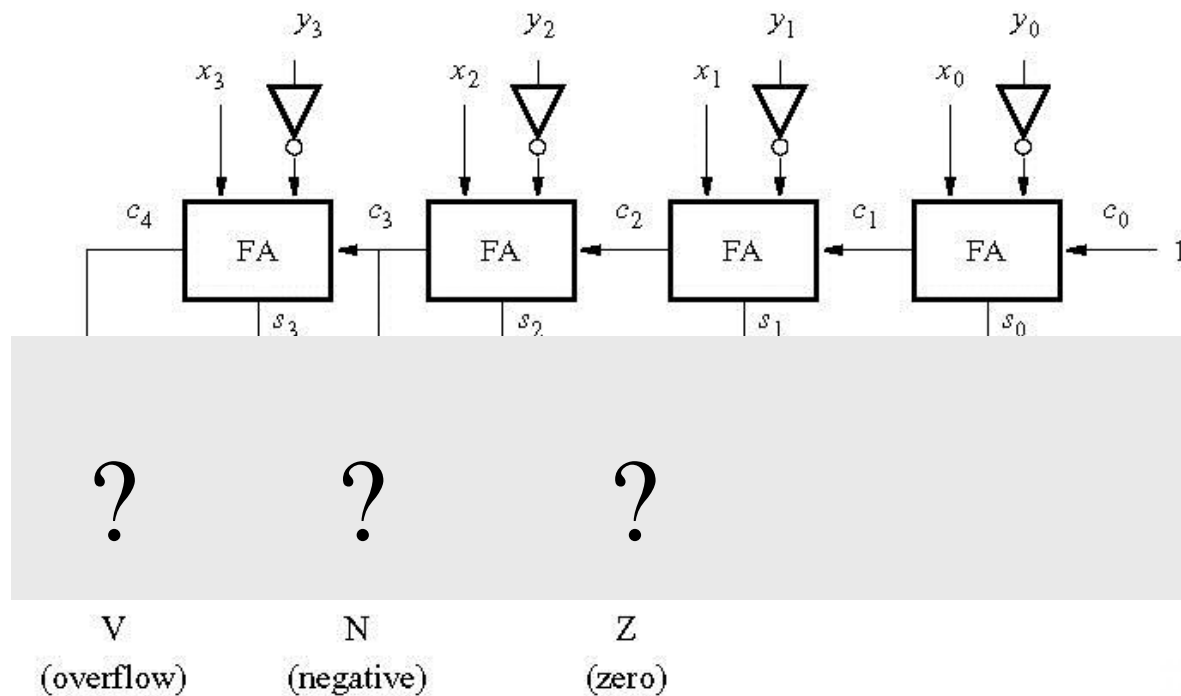


Arithmetic Logic Unit (ALU)



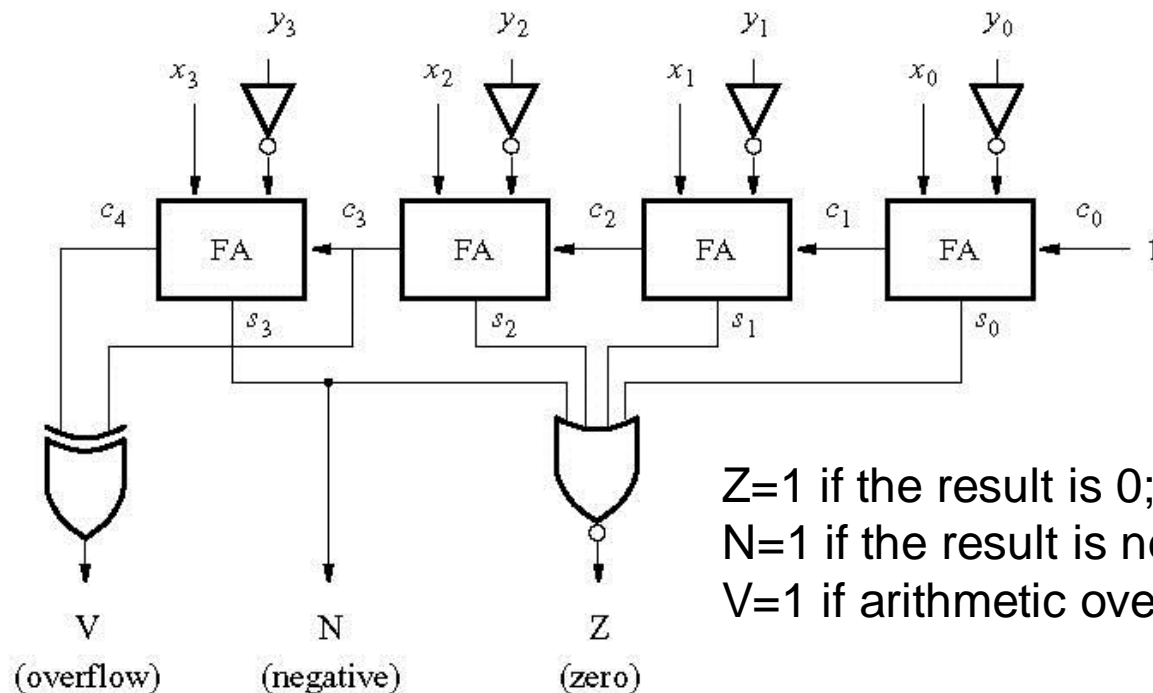
A/L	f1	f0	Funktion
0	0	0	$x+y$
0	0	1	$x+y+C_{in}$
0	1	0	$x-y$
0	1	1	$x-y-\overline{C_{in}}$
1	0	0	$x \text{ or } y$
1	0	1	$x \text{ and } y$
1	1	0	$x \text{ xor } y$
1	1	1	$\text{inv } x$

Comparator (p. 309 BV)



Comparator (p. 309 BV)

- The comparator can be implemented as a subtraction $X - Y$

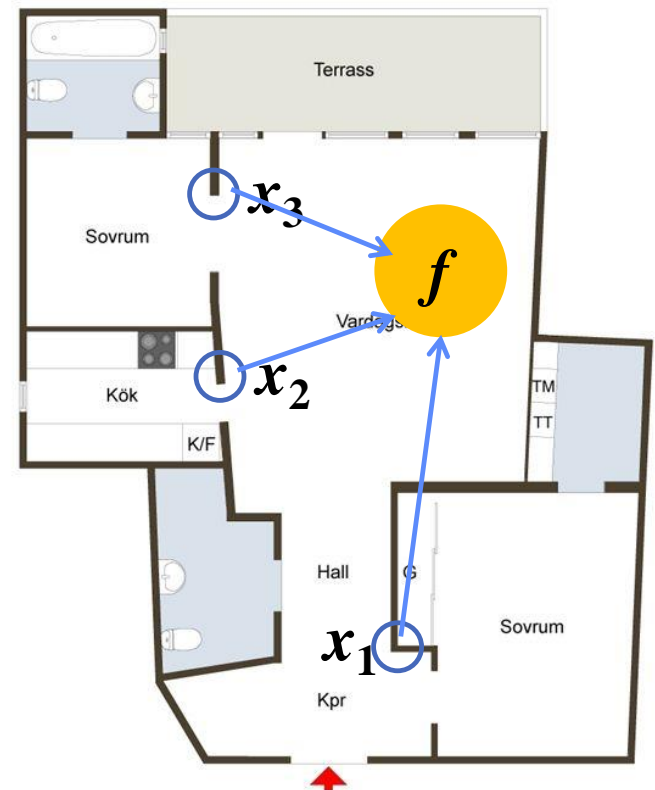
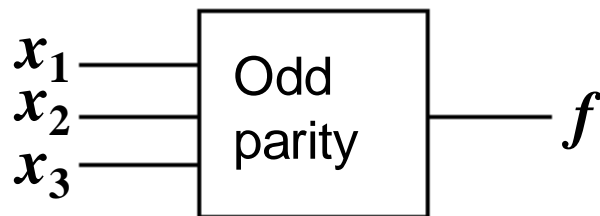


Three-way light control - *revisited*

Brown/Vranesic: 2.8.1

Suppose that we need to be able to turn on / off the light from three different places.

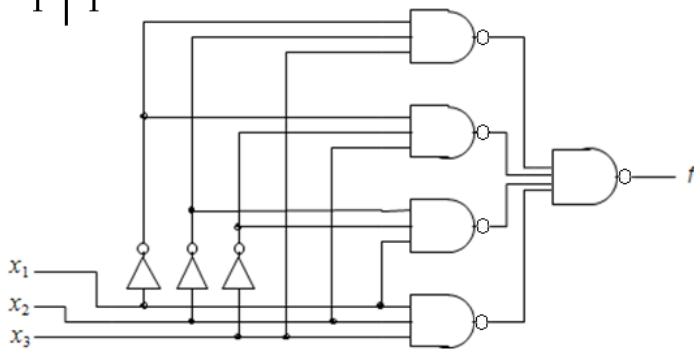
Parityfunction



XOR or NAND?

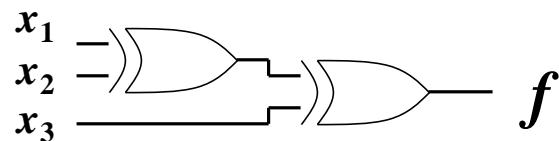
The former solution was based on NAND gates.

x_1	x_2	x_3	f
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



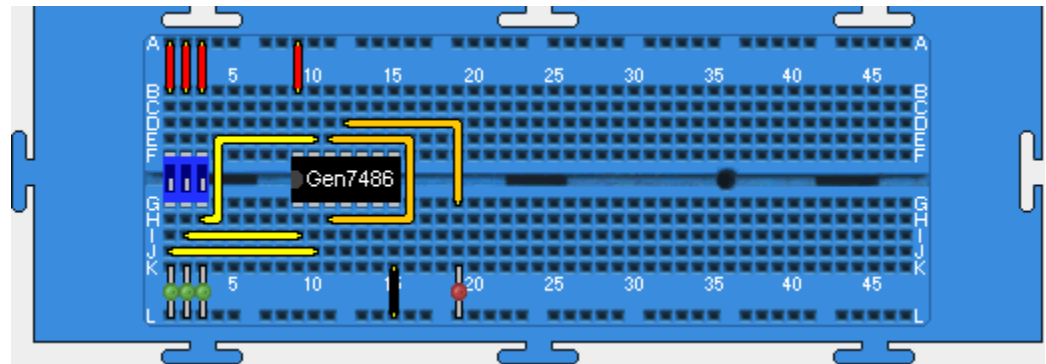
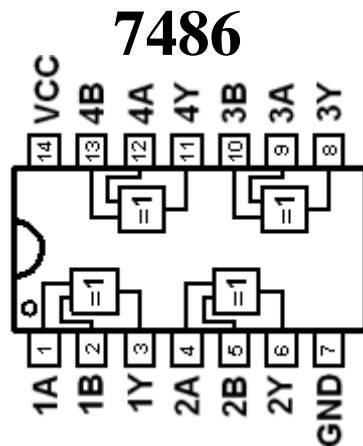
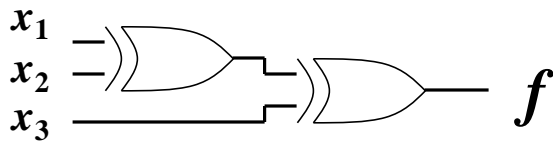
XOR gates will be much more effective than NAND gates!

x_1	$x_2 x_3$				f
	00	01	11	10	
0	0	1	0	1	
1	1	0	1	0	

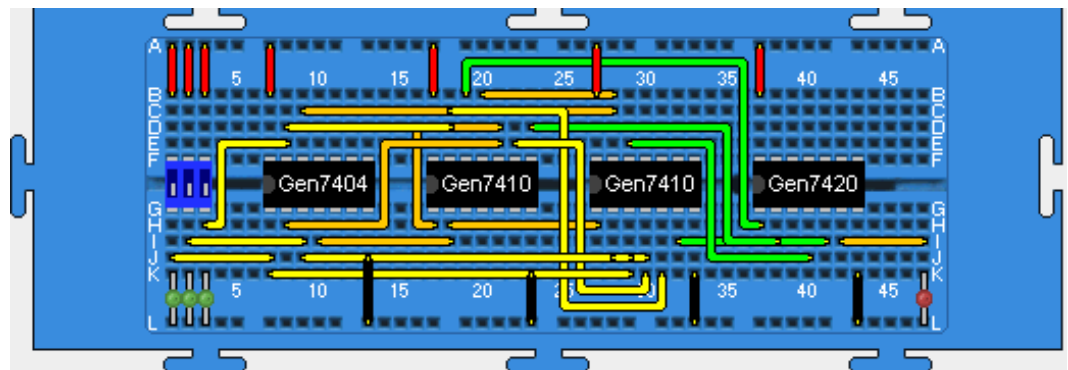


Simpler with XOR gates

With XOR-gates:



(With NAND-gates:)



Summary

- Addition and subtraction of integers
 - 2's complement
 - Subtraction of a number is implemented as addition to its 2's complement
 - Trade-off: Area vs. Speed
 - Different Adder Structures
 - Ripple-Carry Adder (RCA)
 - Carry-Look Ahead Adder (CLA)
 - Carry-Select Adder (CSA)
 - Next lecture: BV pp.291-302
-