Digital Systems I

Instructor: Mohammad Ghamari

Electrical and Computer Engineering Department

These slides are modified version of the slides by Nozar Tabrizi, Introduction to Digital Circuits

Chapter 5

Binary Number Systems
&
Binary Arithmetic

How long does it take for you to do this addition:

950 272 813 746 532 914 +

780 689 290 756 839 387

In this chapter you will be able to design an adder which does this job in, say, 10^{-6} seconds!

Binary Number System

Each bit has its own weight

Example

1110 ← binary number

3210 ← digit position

1110 =

$1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 =$

$1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = (14)_{ten}$

Binary addition (paper-and-pencil)

Example 6: Add: 01 + 01 = 10

1st column: 1 + 1 = 0, carry the one.

2nd column: 1 + 0 + 0 = 1.

Binary addition (paper-and-pencil)

Example 6: Add: 1001 + 011

2^3 2^2 2^1 2^0 ⟵ Weights

3 2 1 0 ⟵ Positions

1 0 **0** 1 +

0 **1** 1

Zero extension:

Add as many 0s to left as you need

Binary addition (paper-and-pencil)

Example 6: Add: 1001 + 0011

2^3	2^2	2^1	2^0		←	Weights
3	2	1	0		←	Positions
0	1	1			←	Carry bits
1	0	**0**	1	+		
0	0	**1**	1			
0	1	1	0	0		

Zero extension:

Add as many 0s to left as you need

Remember

n-bit system: numbers are n bits wide.

In n-bit system, addition result could be (n + 1) bits wide at most. (So, result could be out of range.)

For example, if you add two 4-bit numbers, the result could be 5 bits wide at most.

Binary addition (paper-and-pencil)

Example 7. Add 1001 + 1011

2^3 2^2 2^1 2^0 ⟵ Weights

3 2 1 0 ⟵ Positions

1 0 **0** 1 +

1 0 **1** 1

Binary addition (paper-and-pencil)

Example 7. Add 1001 + 1011

2^3	2^2	2^1	2^0		←	Weights
3	2	1	0		←	Positions
0	1	1			←	Carry bits
1	0	**0**	1	+		
1	0	**1**	1			
1	0	1	0	0		

Result is 5 bits wide (out of range)

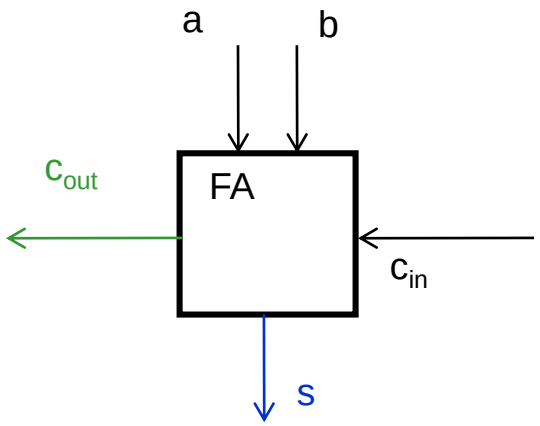
Look what happens in each column

Let's design a Full adder

Truth table

Row	a b c_{in}	c_{out}	s
0	0 0 0	0	0
1	0 0 1	0	1
2	0 1 0	0	1
3	0 1 1	1	0
4	1 0 0	0	1
5	1 0 1	1	0
6	1 1 0	1	0
7	1 1 1	1	1
		MSB	LSB

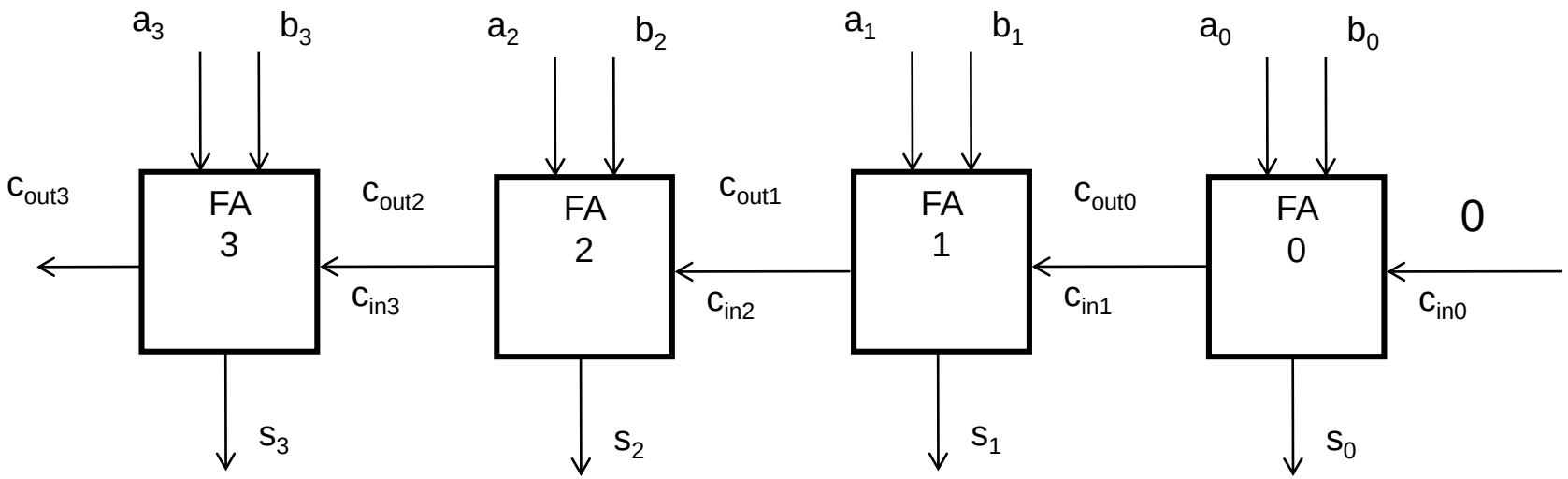
Symbol

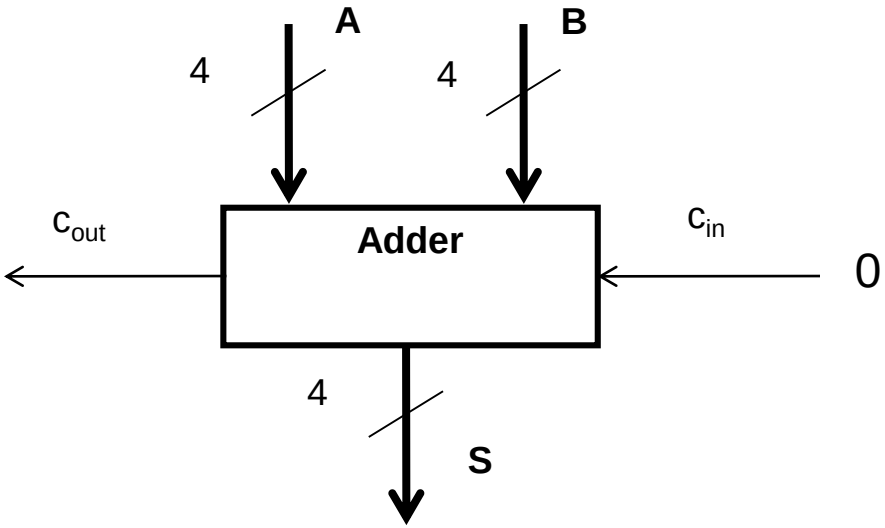


$s\ (a\ b\ c_{in}) = \Sigma\ (1, 2, 4, 7)$

$c_{out}\ (a\ b\ c_{in}) = \Sigma\ (3, 5, 6, 7)$

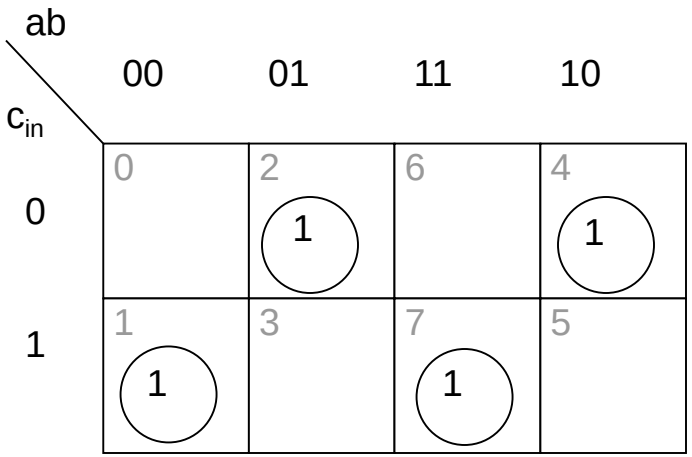
4-bit ripple carry adder (or Paper-&-Pencil algorithm)





Full adder design

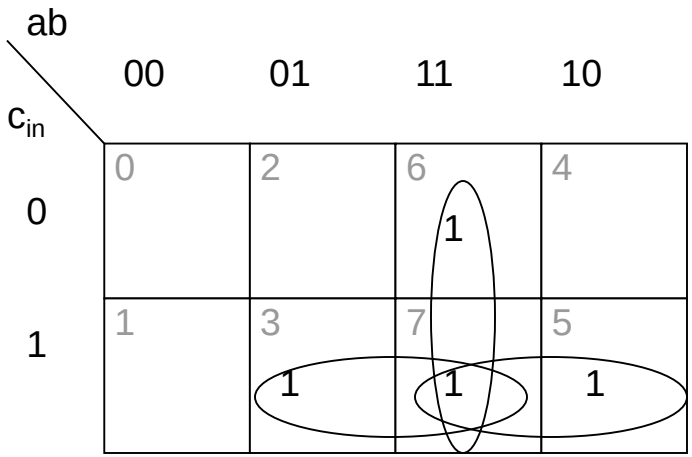
$s(a\ b\ c_{in}) = \Sigma\ (1, 2, 4, 7)$



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

cannot be simplified

$c_{out}(a\ b\ c_{in}) = \Sigma\ (3, 5, 6, 7)$



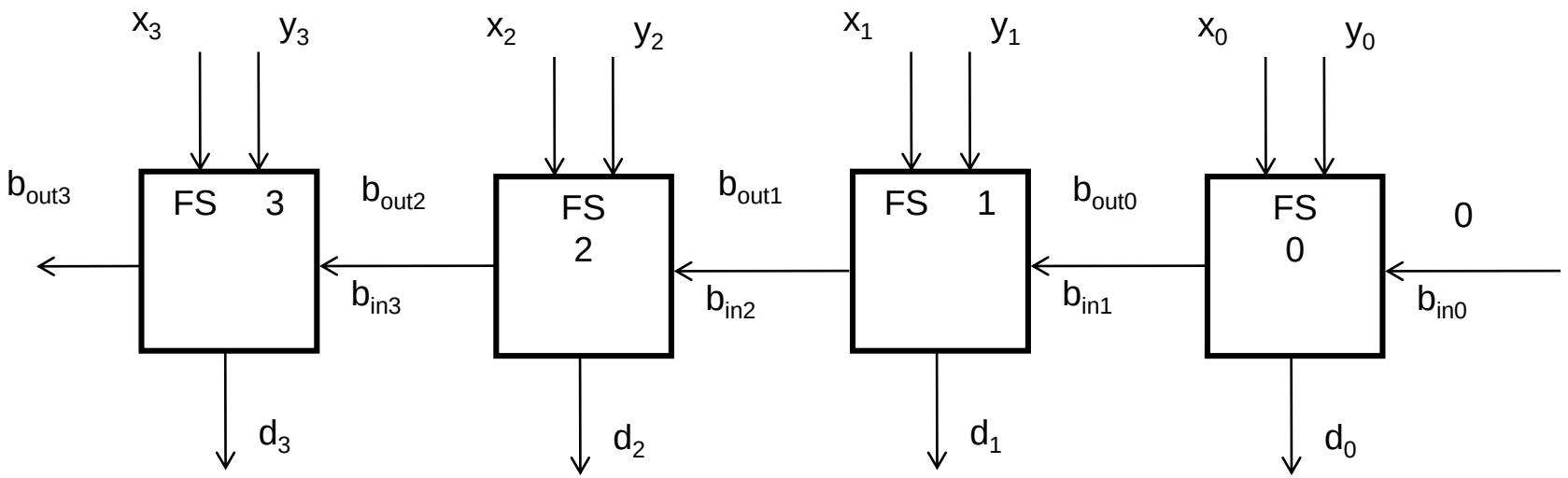
$c_{out}\ (a\ b\ c_{in}) =$

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

(3-bit majority function)

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

Paper-and-pencil algorithm for subtraction



4-bit ripple-borrow subtractor.

(For details see Chapter 5.)

Binary subtraction

Instead of paper-and-pencil algorithm for subtraction, we develop a different technique

"subtraction without subtraction"

or

"addition-based subtraction"

to use the same hardware we use for addition (save hardware)

n-bit addition-based subtraction, X – Y:

1- If operands are not already “n” bits wide, do zero extension to make them “n” bits wide.

2- Perform **X + (bit-wise complement of Y) + 1**

3- “No final carry” in step 2 indicates a “final borrow”, or a greater Y (1st number) than X (2nd number); hence an *impossible subtraction*.

4- But if a final carry is produced in step 2, **ignore** carry; remaining n-bit number is correct subtraction result.

2’s complement of Y = (bit-wise complement of Y) + 1 (5)

Example 12.

Use addition to do subtraction: 1010 (ten) – 101 (5)

Instead of subtraction (1010 – 101), do addition:

X + (bit-wise complement of Y) + 1

But first convert 101 to 0101 (zero extension)

1010 + 1010 + 1 = **1** 0101 (ignore **1**)

Carry => No Borrow

1010 – 0101 = 0101 = 5

Example 13.

Use addition to do subtraction: 1010 (ten) – 1101(13)

Instead of subtraction (1010 – 1101), do addition

X + (bit-wise complement of Y) + 1

Notice both operands are already 4 bits wide

1010 + 0010 + 1 = **0** 1101

No carry => Borrow

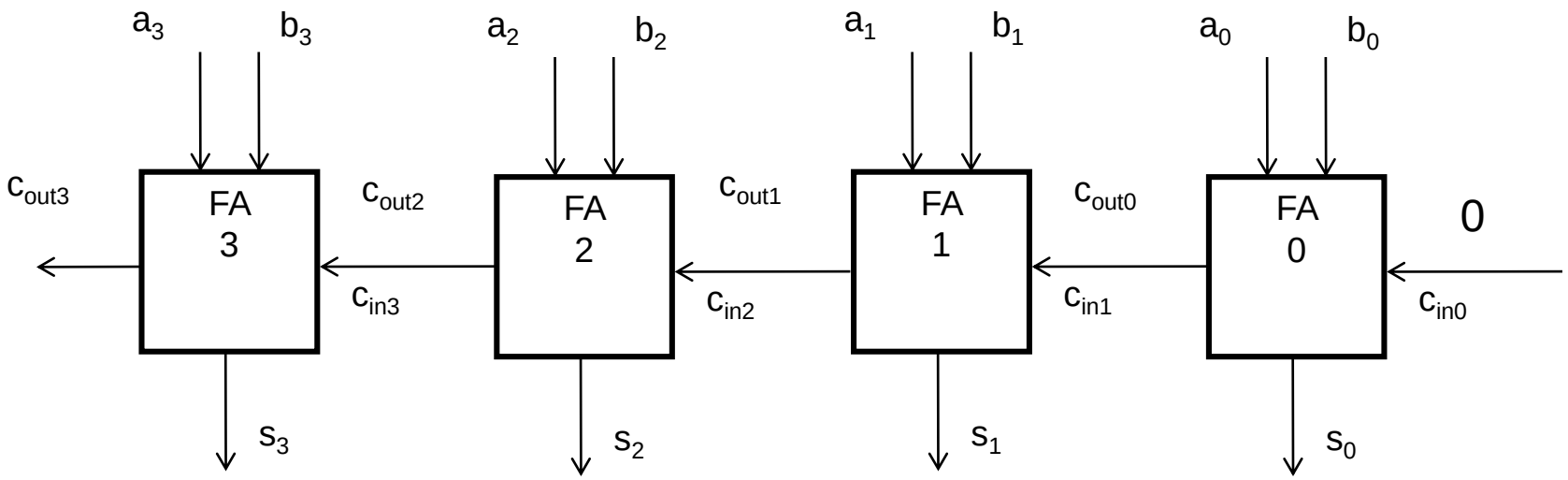
Subtraction is impossible (ten < 13)

Design an Adder-based subtractor ?

X – Y = X + (bit-wise complement of Y) + 1

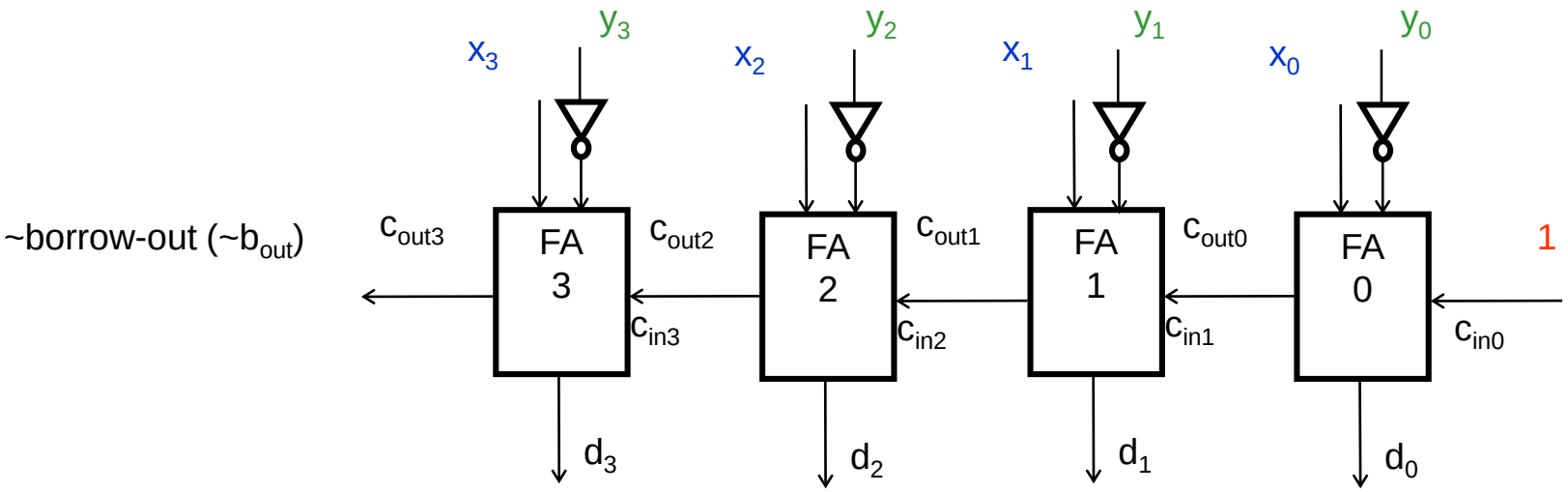
From past slides:

4-bit ripple carry adder (or Paper-&-Pencil algorithm)



Adder-based subtractor design

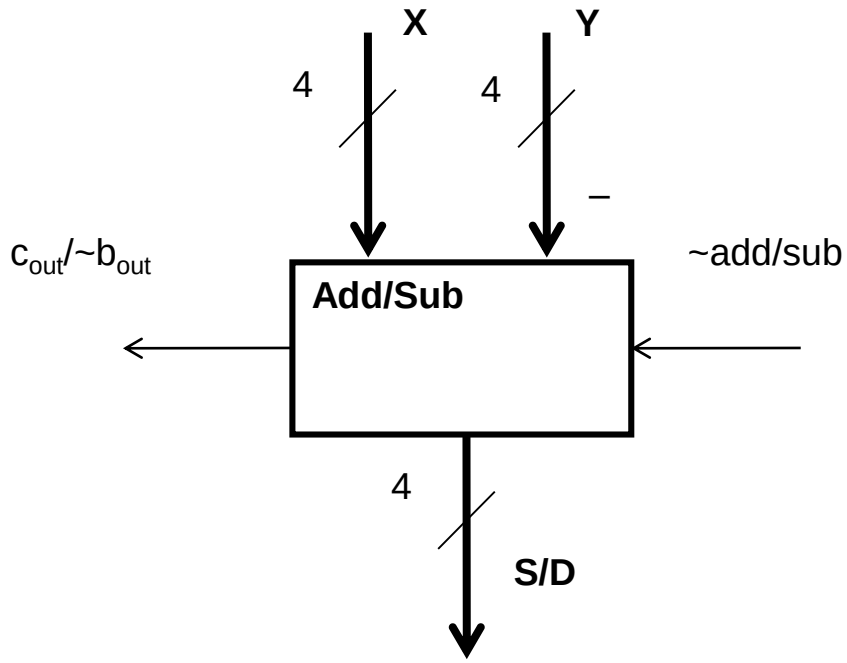
X – Y = X + (bit-wise complement of Y) + 1



~borrow-out is an active-low representing the final borrow-out bit; c_{out3} = 1 means no final borrow

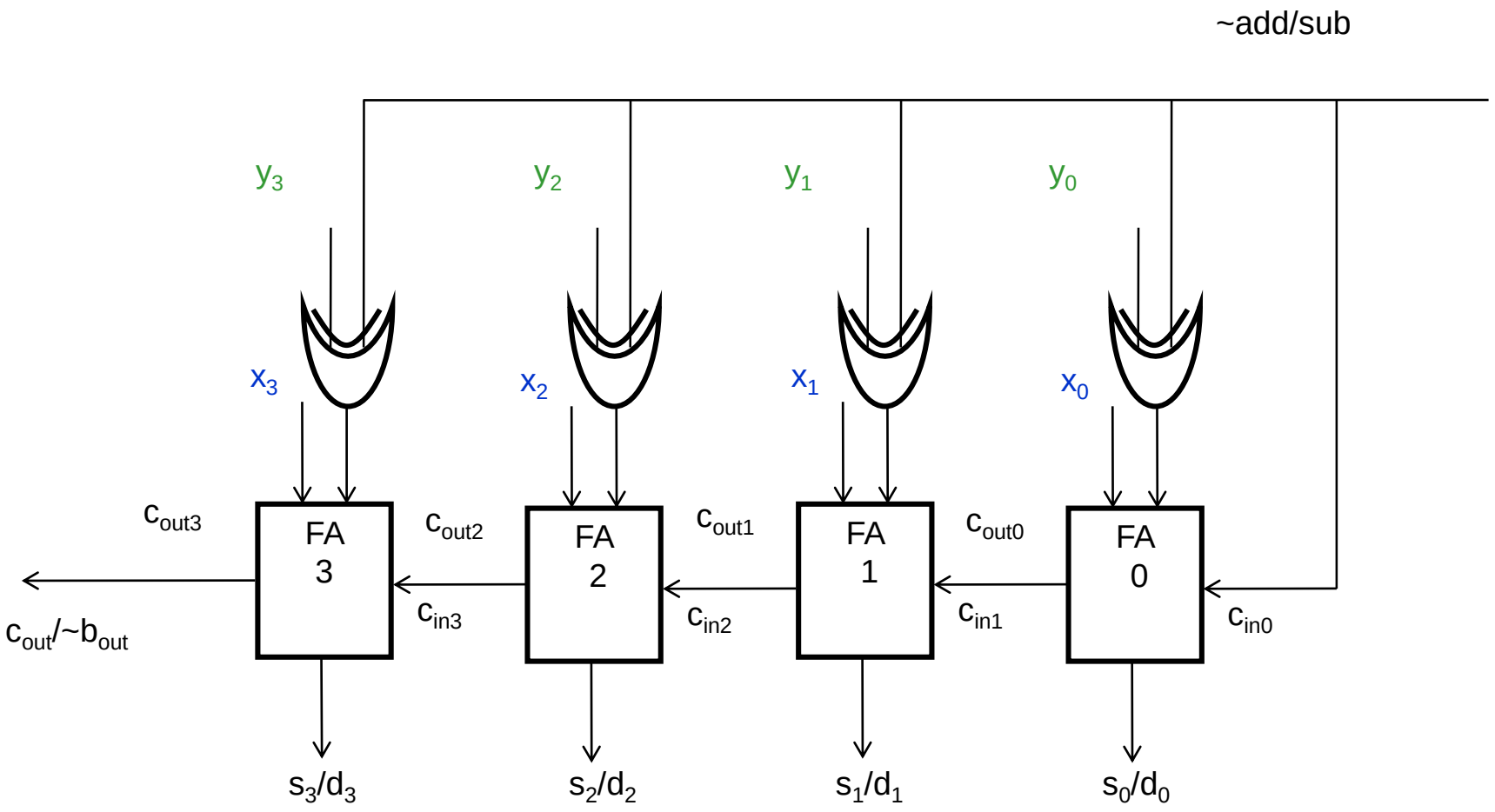
Combine adder & subtractor

Adder/Subtractor Unit: Symbol



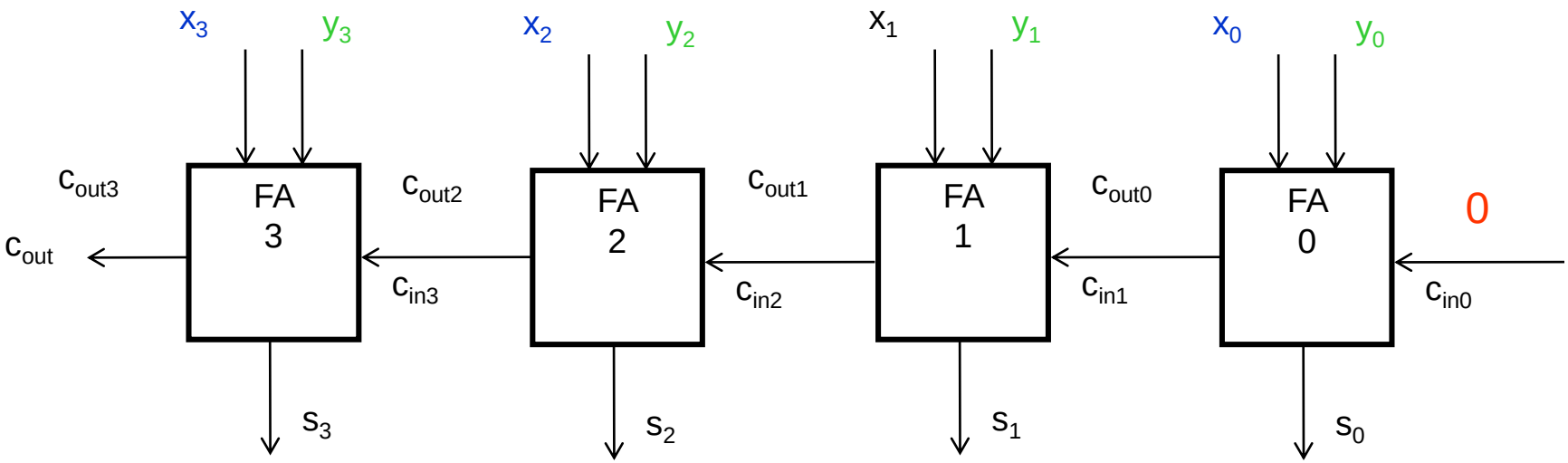
~add/sub is mode/select control signal; ~add/sub = 0 means the add mode is selected

Adder/Subtractor Unit



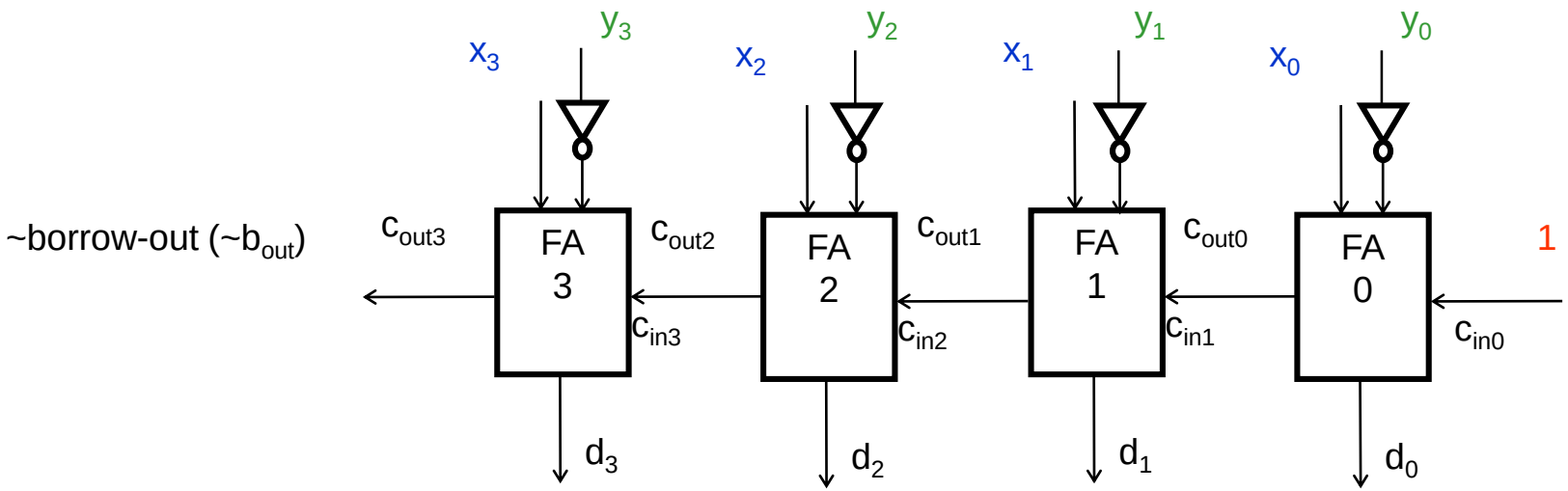
Previous slide become this

if ~add/sub = 0



4-bit adder

Adder/subtractor become this if ~add/sub = 1



4-bit subtractor

n-bit addition-based subtraction, X – Y:

1- If operands are not already “n” bits wide, do zero extension to make them “n” bits wide.

2- Perform **X + (bit-wise complement of Y) + 1**

3- “No final carry” in step 2 indicates a “final borrow”, or a greater Y (1st number) than X (2nd number); hence an *impossible subtraction*.

4- But if a final carry is produced in step 2, **ignore** carry; remaining n-bit number is correct subtraction result.

2’s complement of Y = (bit-wise complement of Y) + 1 (5)

2's complement of Y = (bit-wise complement of Y) + 1 (5)

A shortcut for **2's complement of Y**

Starting with LSB of Y, copy all bits until and including first 1, then complement all remaining bits. **(5')**

Example.

Y = 1010

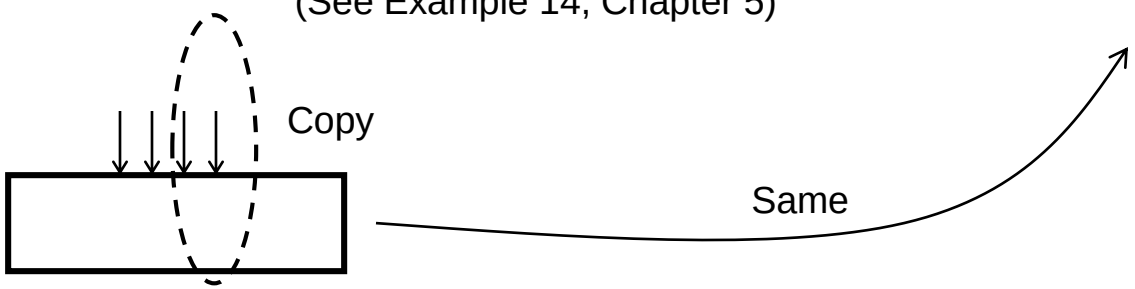
(bit-wise complement of 1010) + 1 = 0101 + 1 = 0110 (5)

Apply (5'): Y = 10**10**

01**10**

Notice: When Y = 0, do not use **5'** to perform subtraction.

(See Example 14, Chapter 5)



Example (use 5')

Use addition to do subtraction: 1101 (=13) – 0110 (=6)

Instead of subtraction (1101 – 0110), do addition:

X + 2's complement of Y

2's complement of Y = 1010 (5')

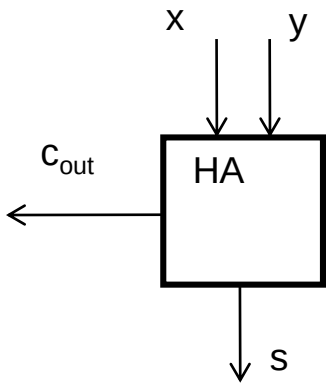
1101 – 0110 = 1101 + 1010 = **1** 0111 (ignore **1**)

Carry => No Borrow

1101 (13) – 0110 (6) = 0111 (7)

Half Adder (HA)

Symbol



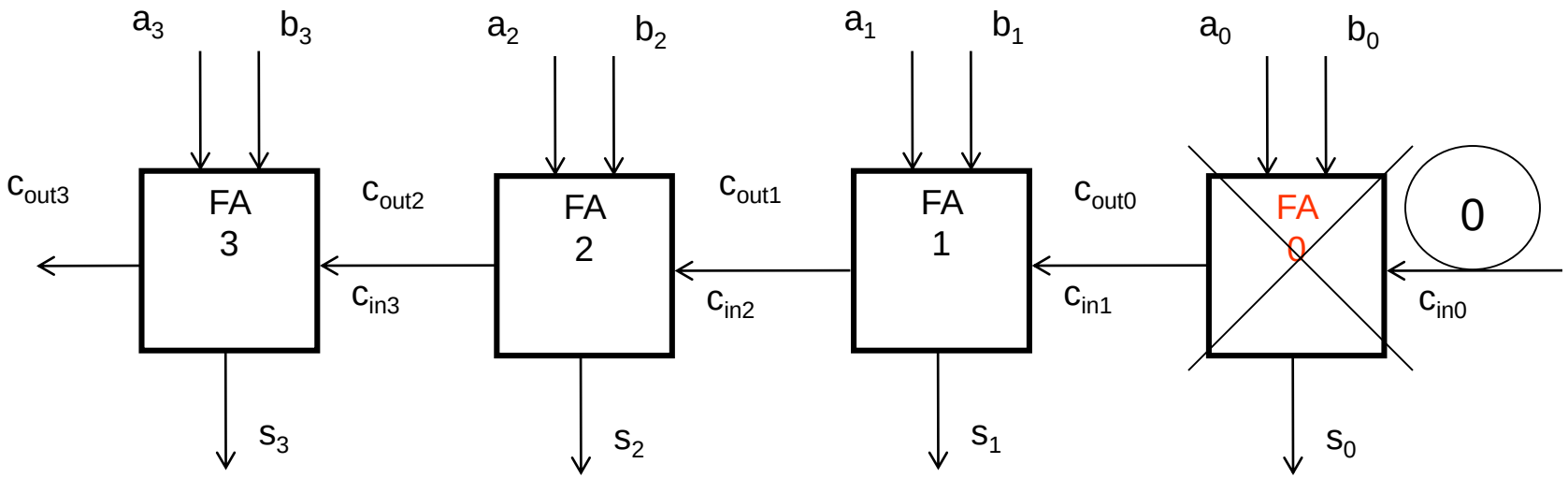
$s(a, b) = \Sigma\ (1, 2)$

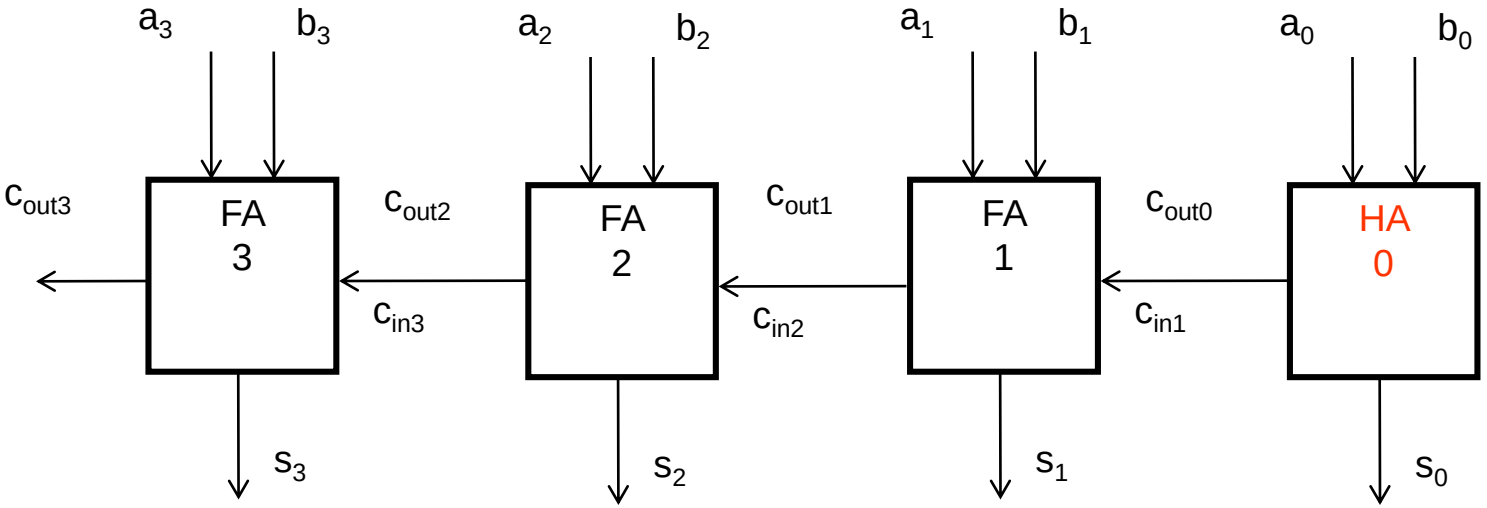
$c_{out}(a, b) = \Sigma\ (3)$

Truth table

Row	x y	c_{out}	s
0	0 0	0	0
1	0 1	0	1
2	1 0	0	1
3	1 1	1	0

Example


a3
b3
a2
b2
a1
b1
a0
b0
cout3
FA
3
cout2
FA
2
cout1
FA
1
cout0
FA
0
0
cin3
cin2
cin1
cin0
s3
s2
s1
s0


a3
b3
a2
b2
a1
b1
a0
b0
cout3
FA
3
cout2
FA
2
cout1
FA
1
cout0
HA
0
cin3
cin2
cin1
s3
s2
s1
s0

Signed Number Systems

2's Complement System

MSB: sign bit (0: positive, 1: negative)

Positive numbers: similar to unsigned numbers (append a 0)

Unsigned 5 = 101

+ 5 = 0101

Negative number = 2's complement of positive number:

-N = (bit-wise complement of N) + 1 **(8)**

Or

Starting from right side of N, copy all bits until and including the first 1, and then complement all remaining bits. **(8')**

Remember: 2's (0…00) = 0…00.

Example 17. (p. 12) n = 4

N = 0110 (+6)

Use (8): -N = 1001 + 1 = 1010 (-6)

Use (8'): -N = 1010 (-6)

If N is complemented twice, it remains unchanged:

-(-N) = 0101 + 1 = 0110 (=+6)

n-bit 2's Complement System

Number of bit patterns: $P = 2^n$

Number of numbers: $B = 2^n$

B = P

Range of numbers: $R = [-(2^{n-1}) : +(2^{n-1} - 1)]$

Smallest number: $Nmin = -(2^{n-1})$

Largest number: $Nmax = 2^{n-1} - 1$

4-bit 2’s Complement System

Example 18. (p. 13) n = 4

P = 16,

B = 16,

P = B

R = [-8 : +7]

Nmin = -8

Nmax = +7

4-bit 2's Complement Numbers

4-bit 2's complement	Decimal equivalent	4-bit 2's complement	Decimal equivalent
0000	0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

Notice

- You cannot negate the most negative number

  Nmin = -8 (4-bit system)

  -Nmin = +8 > Nmax = + 7

  Out of range! (Overflow)
- Also -0…00 = 0…00

2’s Complement Addition

Use unsigned adder to add two numbers in 2’s complement system.

Result is valid in 2’s complement system, no matter what the signs of numbers are, i.e.,

We do not need a special adder for 2’s complement system.

Example 21.

Add: 0011 (= +3) + 0100 (= +4)

sign bit

0 011 + +011 = +3

0 100 +100 = +4

---

0 111 +111 = +7

Example 22.

Add: 0011 (= +3) + 1100 (= -4)

sign bit

0 011 + +011 = +3

1 100 -100 = -4

---

1 111 -001 = -1

4-bit 2's complement	Decimal equivalent	4-bit 2's complement	Decimal equivalent
0000	0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

1111 is a valid result (-1) in 4-bit 2's complement system.

Example 23.

Add: 1101 (= -3) + 1100 (= -4)

4-bit 2's complement	Decimal equivalent	4-bit 2's complement	Decimal equivalent
0000	0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

sign bit

1 101 + -011 = -3

1 100 -100 = -4

---

1 **1** 001 -111 = -7

Carry out, ignore it

Here the addition results in a final carry, but can be removed (it is not part of the addition result.)

Example 25. (p. 15)

Add: 0110 (= +6) + 0101 (= +5)

sign bit

0 110 + +110 = +6

0 101 +101 = +5

---

1 011 -011 = -5

4-bit 2's complement	Decimal equivalent	4-bit 2's complement	Decimal equivalent
0000	0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

Overflow

6+5=+11, we know that the largest number representable in the 4-bit 2's complement system is +7; therefore, **the correct result +11** cannot be represented by 4-bits and this means that overflow has occurred.

Example 26.

Add: 1001 (= -7) + 1100 (= -4)

4-bit 2's complement	Decimal equivalent	4-bit 2's complement	Decimal equivalent
0000	0	1000	-8
0001	+1	1001	-7
0010	+2	1010	-6
0011	+3	1011	-5
0100	+4	1100	-4
0101	+5	1101	-3
0110	+6	1110	-2
0111	+7	1111	-1

sign bit

1 001 + -111 = -7

1 100 -100 = -4

---

1 **0** 101 +101 = +5

Overflow

(-7)+(-4)=-11; but the most negative number in the four-bit 2's complement system is -8; therefore the correct result cannot be represented in four bits, this means that an overflow has happened.

Definition

If the result of an operation does not fit into the designated number of bits, an *overflow* has occurred.

Lemma 1.
In a 2's complement addition an overflow occurs if and only if

1) the two operands have the same sign bit, and

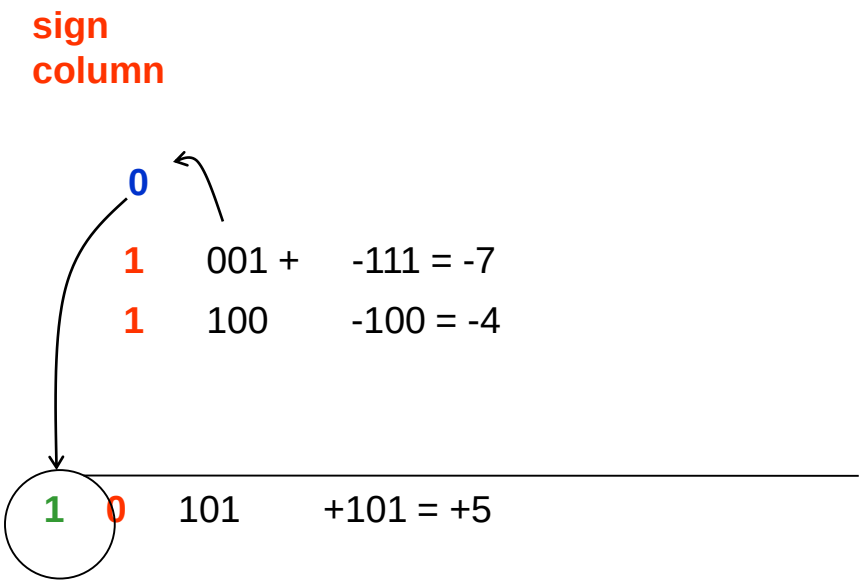
2) this bit is different from the sign bit of result.

Lemma 1 is equivalent to following lemma:

Lemma 2.
In a 2's complement addition an overflow occurs if and only if carry bit arriving at sign column is different from carry bit leaving this column.

Example 27. (p. 17)

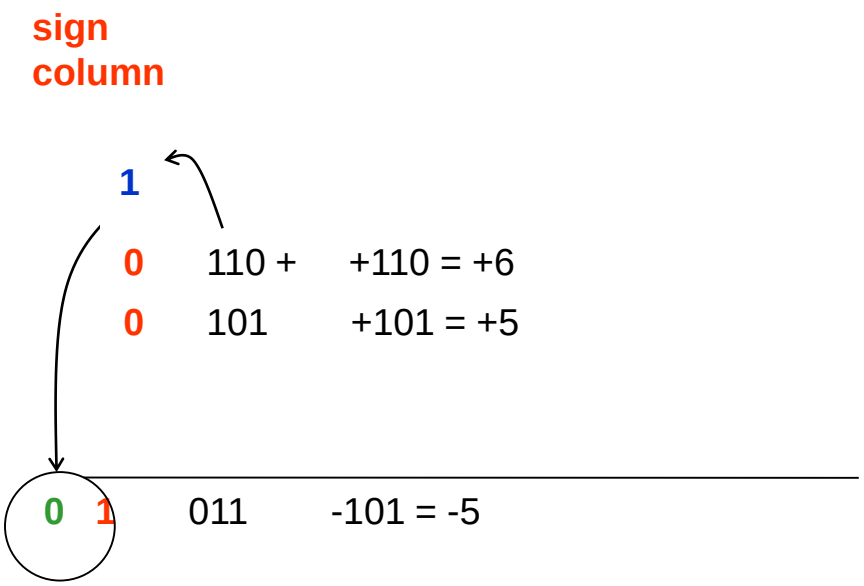
Add: 1001 (= -7) + 1100 (= -4)



For sign column
Carry-in ≠ carry-out => Overflow

Example 28.

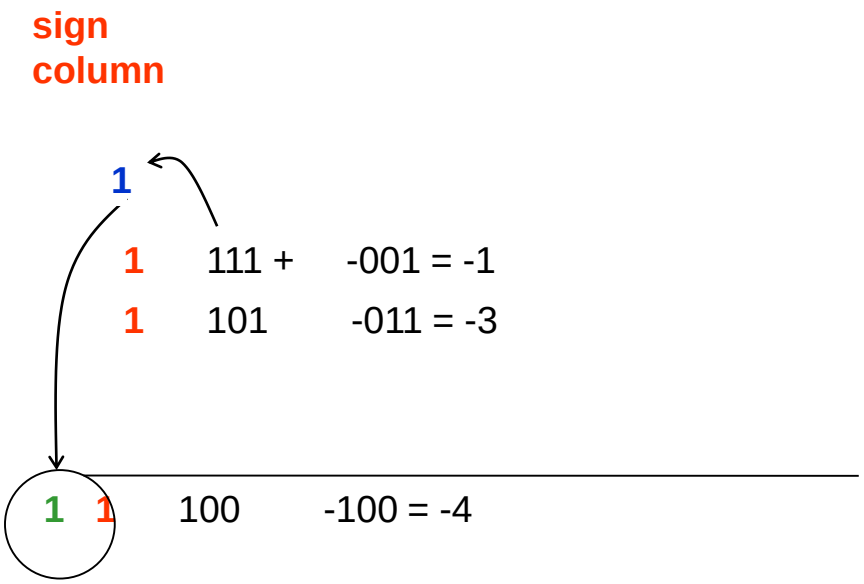
Add: 0110 (= +6) + 0101 (= +5)



For sign column
Carry-in ≠ carry-out => Overflow

Example 29.

Add: 1111 (= -1) + 1101 (= -3)



For sign column
Carry-in = carry-out => No Overflow

Sign Extension

Example 30.

Add in the four-bit 2's complement system:

1001 (= -7) + 011 (= 3)

011 = 0011 =>

1001 + 011 = 1001 + 0011 = 1100 = -4

Sign Extension

Example 31.

Add in the four-bit 2's complement system:

1011 (= -5) + 110 (= -2)

110 = 1110

1011 + 110 = 1011 + 1110 = **1** 1001

We may append as many copies of sign bit as we need to left of number. This is called *sign extension*.

n-bit Signed Addition

Summary

- Do sign extension to make operands n bits wide.
- Add two operands using, say, paper-and-pencil algorithm.
- If overflow, then result is invalid.
- If no overflow, then ignore carry bit. Remaining **n** bits are correct result of addition.

2’s Complement Subtraction

Subtraction may be reworded:

Instead of X – Y

We may write X + (-Y)

Example

1- 4 – 3 = 4 + (-3)

2- 7 – (-5) = 7 + 5

Therefore,

X – Y = X + (bit-wise complement of Y) + 1 (9)

Same as unsigned subtraction

For subtraction, Lemma 1 may be reworded:

Lemma 1'.

In a 2's complement subtraction, an overflow occurs if and only if:

1) the two operands have **different** sign bits, and
2) the sign of the **result** is different from that of the **first operand**.

$$X - Y = X + (\text{bit-wise complement of } Y) + 1 \qquad (9)$$

Remember:

Same-sign operands may **never** produce a subtraction overflow.

Example 33.

Subtract 1100 (= -4) – 101 (= -3)

101 = 1101 (sign extension)

Add 1100 + 0011

sign bit

1 100 +

0 011

---

0 **1** 111

Subtraction:

Same-sign operands

No overflow

Addition:

Different-sign operands

No overflow

Example 34.

Subtract 1001 (= -7) – 0100 (= +4)

Add 1001 + 1100

Subtraction (different signs):

sign of result ≠ sign of X

Overflow

sign bit

1 001 +

1 100

---

1 **0** 101

Addition (same signs):

sign of result ≠ sign of X

Overflow

Lemma 3.

In a 2's complement subtraction,

and also in a 2's complement addition:

An overflow occurs if and only if the carry bit arriving at the sign column is different from the carry bit leaving this column.

$$X - Y = X + (\text{bit-wise complement of } Y) + 1 \qquad \textbf{(9)}$$

n-bit subtraction, X – Y

Summary

1- Do sign extension to make operands n bits wide.

2- Perform X + (bit-wise complement of Y) + 1 (9)

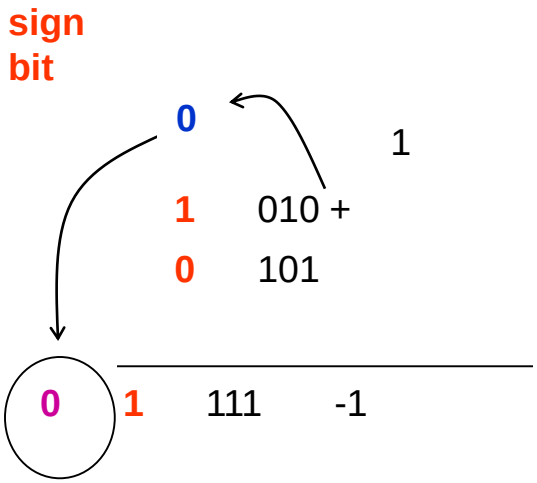
3- Use Lemma 3. If there is an overflow, subtraction result is not representable in n bits.

4- If there is no overflow, ignore carry bit; remaining n bits are correct result of subtraction.

Example.

Subtract 1010 (= -6) – 1011 (= -5)

X – Y = X + (bit-wise complement of Y) + 1 (9)



For sign bit

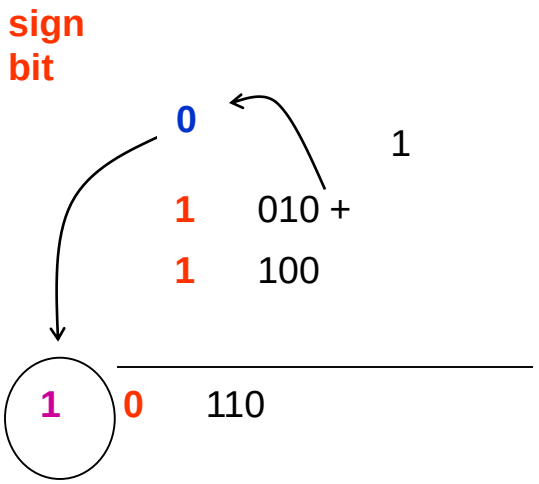
Carry-in = carry-out

No overflow

Example.

Subtract 1010 (= -6) – 0100 (= +4)

X – Y = X + (bit-wise complement of Y) + 1 (9)



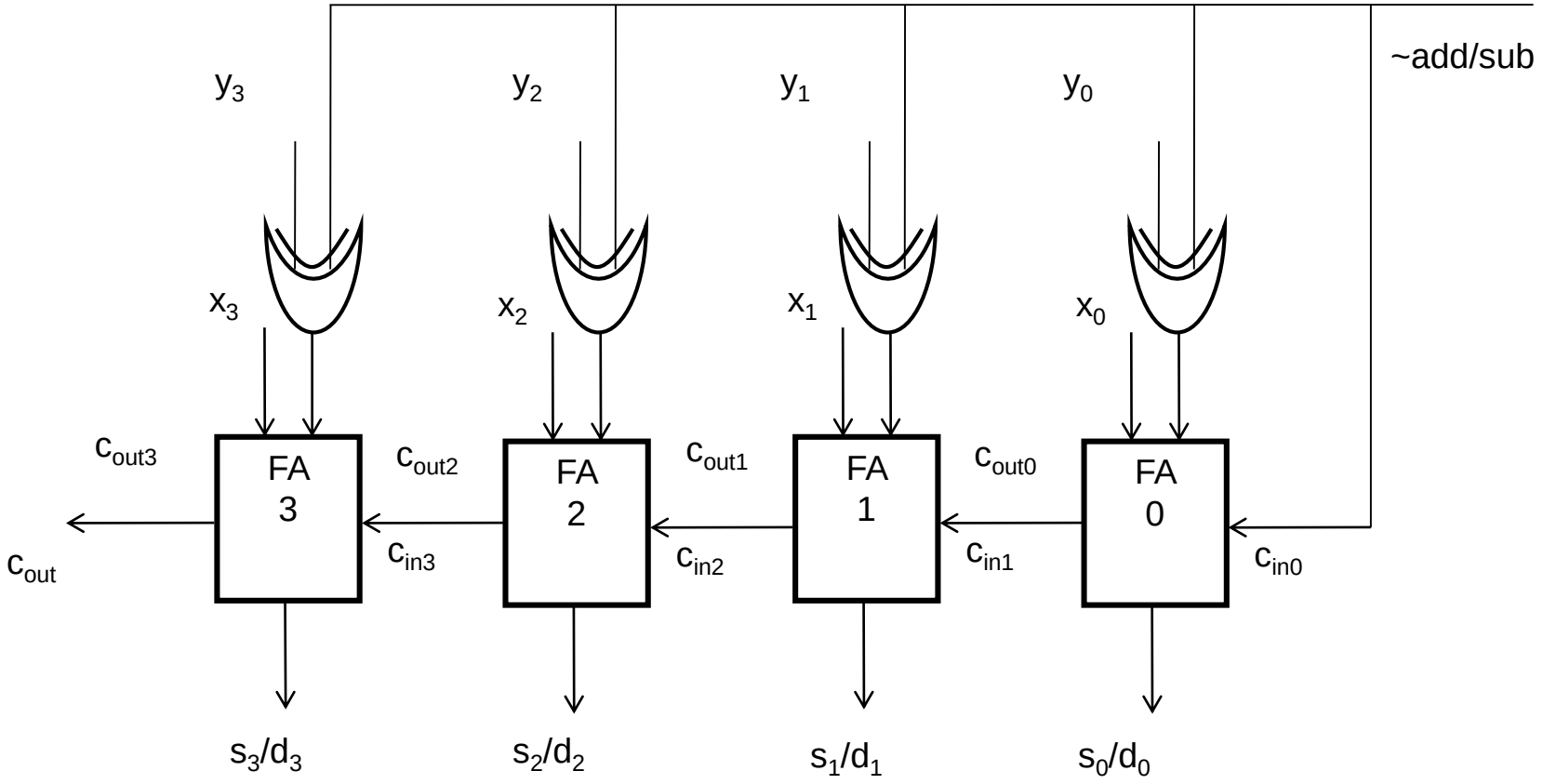
For sign bit

Carry-in ≠ carry-out

Overflow

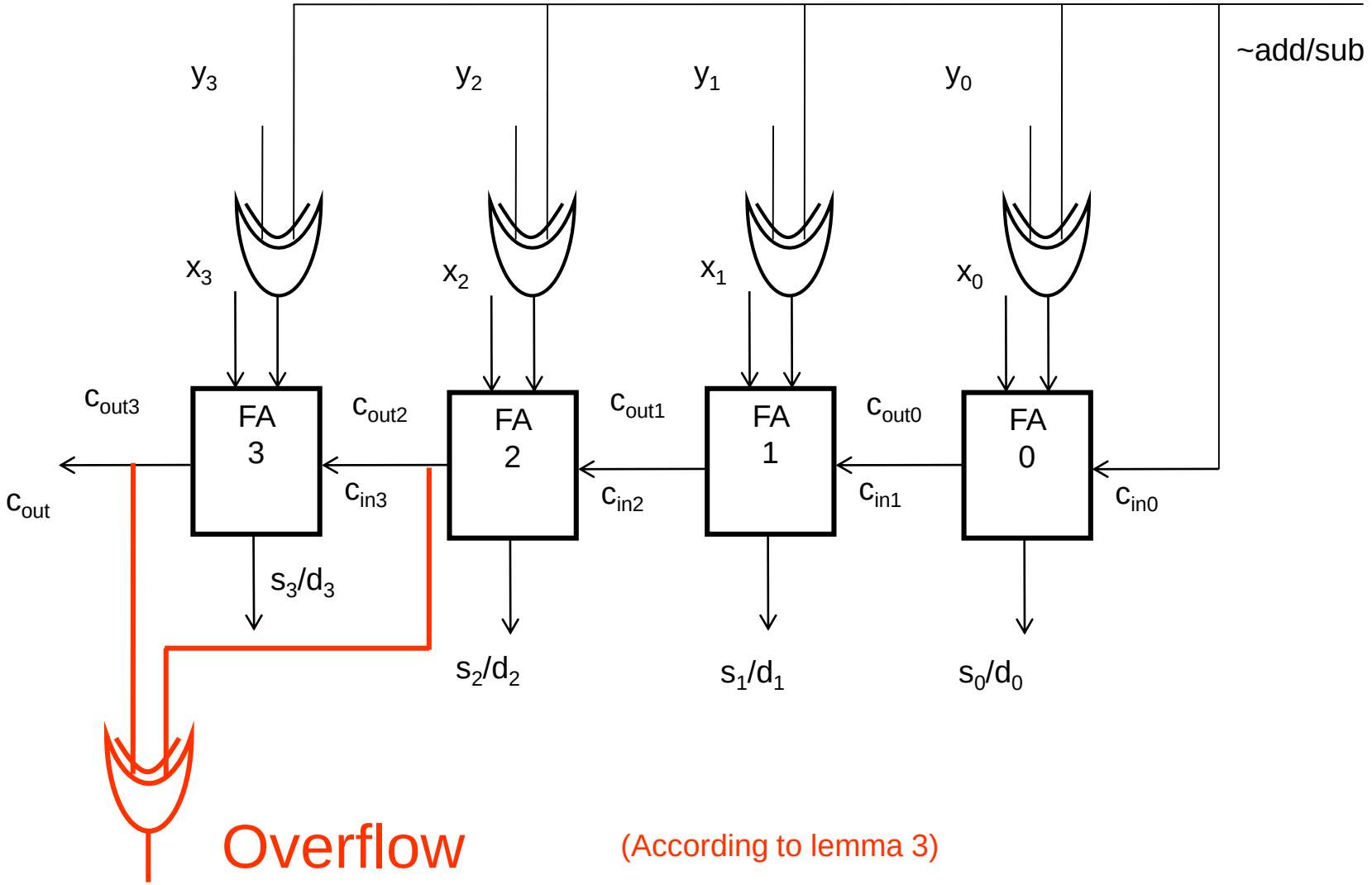
Adder/Subtractor Design for Signed Numbers

2's Complement Adder/Subtractor (same as unsigned)



X – Y = X + (bit-wise complement of Y) + 1 (9)

2's Complement Adder/Subtractor with overflow detector

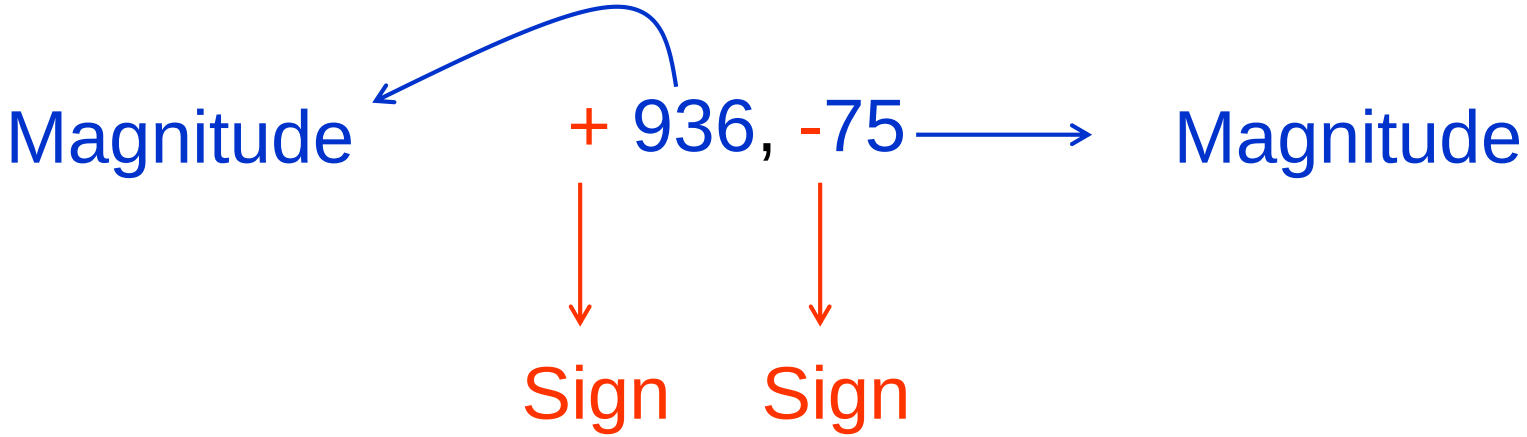


(According to lemma 3)

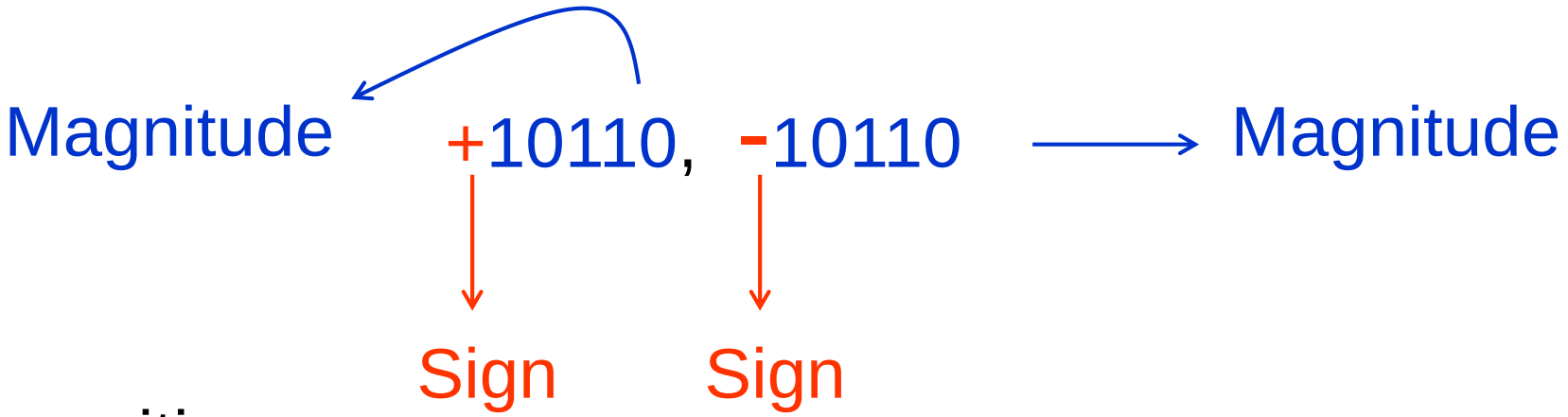
Two more Signed Systems:

- **Sign & Magnitude**
- **1’s Complement**

Sign & Magnitude (S & M)

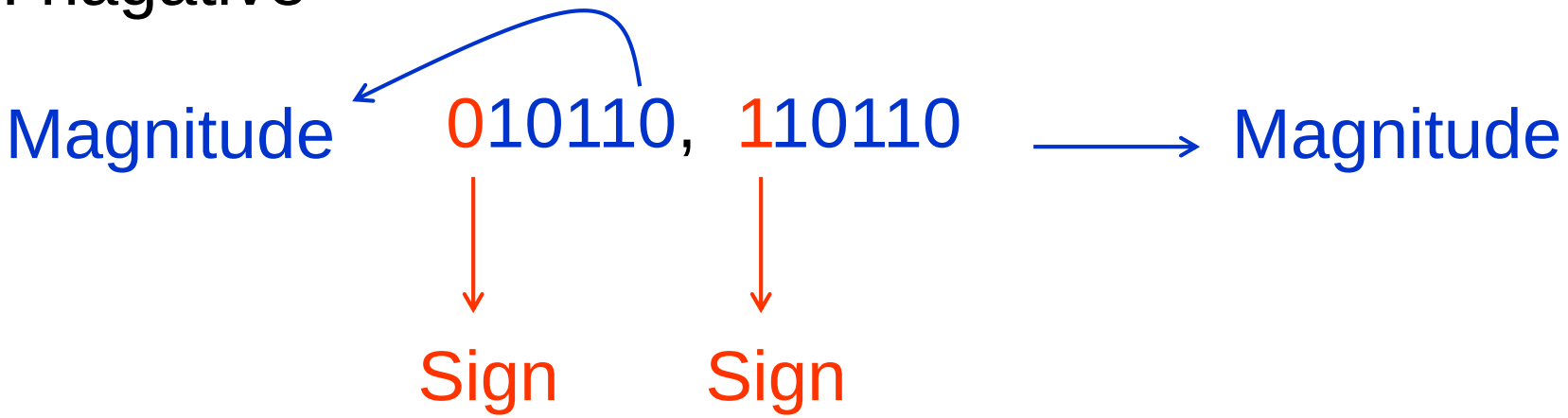


Sign & Magnitude (Binary)



0: positive

1: nagative



Sign & Magnitude (Binary)

Number of bit patterns: $P = 2^n$

Number of numbers: $B = 2^n - 1$

$B \neq P$

Range of numbers: $R = [-(2^{n-1} - 1) : +(2^{n-1} - 1)]$

Smallest number: $Nmin = -(2^{n-1} - 1)$

Largest number: $Nmax = 2^{n-1} - 1$

Sign & Magnitude (Binary)

Example 15. n = 4

P = 16,

B = 15,

P ≠ B

R = [– 7 : +7]

Nmin = – 7

Nmax = +7

Sign & Magnitude (4-bit)

4-bit S&M	Decimal equivalent	4-bit S&M	Decimal equivalent
0000	0	1000	-0
0001	1	1001	-1
0010	2	1010	-2
0011	3	1011	-3
0100	4	1100	-4
0101	5	1101	-5
0110	6	1110	-6
0111	7	1111	-7

1’s Complement

MSB: sign bit

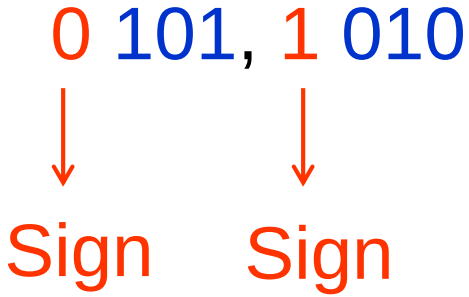
Positive number: same as before

Negative numbers: bitwise complement

Example:

N = 0101(+5)

– N = 1010 (-5)



n-bit 1's Complement

Number of bit patterns: $P = 2^n$

Number of numbers: $B = 2^n - 1$

$P \neq B$

Range of numbers: $R = [-(2^{n-1} - 1) : +(2^{n-1} - 1)]$

Smallest number: $Nmin = -(2^{n-1} - 1)$

Largest number: $Nmax = 2^{n-1} - 1$

1’s Complement

Example 16. n = 4

P = 16,

B = 15,

P ≠ B

R = [– 7 : +7]

Nmin = – 7

Nmax = +7

2's, 1's **Complements** and **S & M (4-bit)**

Bit pattern	S&M	1'S	2's	Bit pattern	S&M	1's	2's
0000	+0	+0	0	1000	-0	-7	-8
0001	+1	+1	+1	1001	-1	-6	-7
0010	+2	+2	+2	1010	-2	-5	-6
0011	+3	+3	+3	1011	-3	-4	-5
0100	+4	+4	+4	1100	-4	-3	-4
0101	+5	+5	+5	1101	-5	-2	-3
0110	+6	+6	+6	1110	-6	-1	-2
0111	+7	+7	+7	1111	-7	-0	-1

Octal and Hexadecimal Numbers

Binary bit patterns are error-prone

10111010010001101010100100010011011101

Use Octal (radix 8) or Hexadecimal (radix 16)

Example

Convert N = 1011110111000101O to octal.

N = 10, 111, 101, 110, 001, 010 = 2 7 5 6 1 2 (Octal)

Possible digits: 0 to 7

For hexadecimal we need 6 more symbols (digits)

A: decimal 10 (= binary 1010),
B: decimal 11 (= binary 1011),
C: decimal 12 (= binary 1100),
D: decimal 13 (= binary 1101),
E: decimal 14 (= binary 1110),
F: decimal 15 (= binary 1111).

Example.

Convert

N = 11011101010100111111011111101

to hexadecimal.

N = 110, 1110, 1010, 1001, 1111, 0111, 1101=

6 E A 9 F 7 D (Hexadecimal)

Example

Convert octal N = 2 7 3 4 5 6 1 to binary:

N = (2 7 3 4 5 6 1) octal =

010 111 011 100 101 110 001

Example.

Convert hexadecimal N = F D 4 B 8 E to binary.

N = (F D 4 B 8 E) hexadecimal =

1111 1101 0100 1011 1000 1110

Reading Assignment

Gray code and ASCII code

Pages 21-23