SMJE 3173 Digital System

Week 5: Arithmetic Circuits Part 1

- Binary Number Representation
  - Sign & Magnitude
  - Ones Complement
  - Twos Complement
- Networks for Binary Addition
  - Half Adder
  - Full Adder
  - Ripple Adder
  - Subtractor

Motivation

- Arithmetic circuits are excellent examples of combinational logic design
- Time vs. Space Trade-offs
  - Doing things fast requires more logic and thus more space
  - Example: carry look ahead logic
- Arithmetic Logic Units
  - Critical component of processor datapath

Unsigned Integers

- Only positive numbers are called ***unsigned*** numbers
- Smallest representable value: bit
- Bit groups represent information –
  - 4 bits in a group : nibble
  - 8 bits in a group : byte
  - 2^{10} in a group : kilo e.g. kilo, kilobytes (kB)
- Number of bits determine max. combinations of information -

  N bits = 2^N values

Number of Bits	**Number of values**	**Machine**
4	16	Intel 4004
8	256	8080, 6800
16	65536	PDP11, 8086, 68000
32	~ 4 x 10^9	IBM 370, 68020, VAX11/780, IEEE single
48	1 x 10^{14}	Unisys
64	1.8 x 10^{19}	Cray, IEEE double

Unsigned Integers

- Value for the bit pattern:

$$V_{unsigned} = \sum_{i=0}^{N-1} b_i \times 2^i$$

$$V = b_0 \text{ x } 2^0 + b_1 \text{x } 2^1 + b_2 \text{ x } 2^2 + \ldots b_{N-1} \text{ x } 2^{N-1}$$

- Example:

$$10110_2 = 1 \text{ x } 2^4 + 0 \text{ x } 2^3 + 1 \text{ x } 2^2 + 1 \text{ x } 2^1 + 0 \text{ x } 2^0 = 22_{10}$$

- How many numbers can 8-bit represent?
- For N bits, range of values:

  0 up to $2^N - 1$

Representation of Negative Numbers

- Representation of positive numbers are the same in most systems
- Major differences are in how negative numbers are represented. There are 3 major schemes:
  - sign and magnitude
  - ones complement
  - twos complement
- Assumptions:
  - 4-bit machine word
  - 16 different values can be represented

    -7, -6, - 5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8
  - roughly half are positive, half are negative

Sign and Magnitude (SAM)

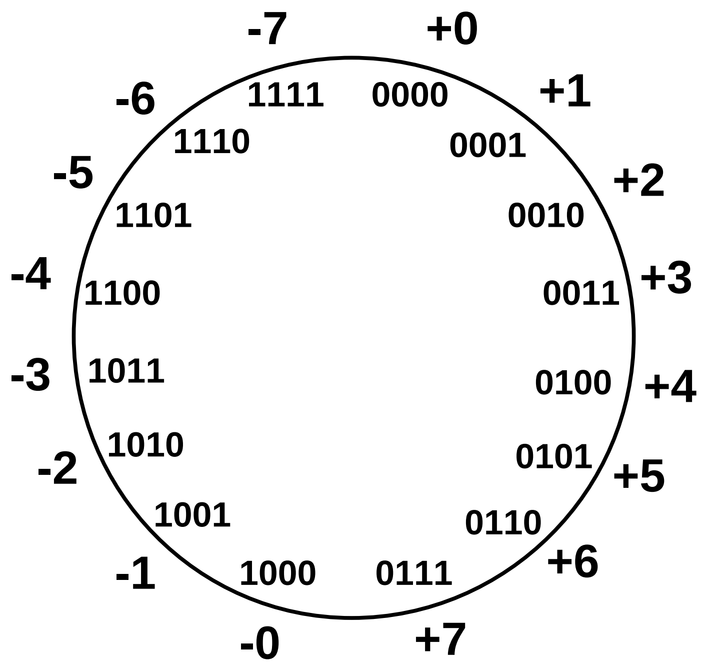
- Easiest to understand
- Method:

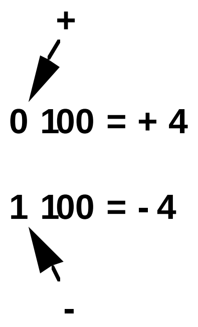
  (i) Leftmost bit (MSB) is sign bit

  - 0 means positive e.g. +18 = 00010010
  - 1 means negative e.g.
- (ii) Represent the magnitude of the number in true binary form

- Example:
  - +18 = 00010010
  - -18 = 10010010

Sign and Magnitude





Summary : (Example for 4-bit number representation; N= 4)

- High order bit is sign: 0 = positive (or zero), 1 = negative
- Three low order bits is the magnitude: 0 (000) thru 7 (111)
- Number range for n bits = +/-(2^{n-1}-1)
- Two representations for 0 (0000 and 1000)

Sign-and-Magnitude Problems

- Easy for humans to understand, but may not be the best for machine operation efficiency
- Cumbersome addition/subtraction
- Must compare magnitudes to determine sign of result
- Need to check both sign and magnitude in arithmetic
- Two representations of zero (+0 and -0)

Ones' Complement Representation

- Ones' complement defined algebraically as

$$\overline{N} = (2^n - 1) - N$$

- For example:

  (i) Find 2's complement of 5 (i.e. -5) in 4-bit representation

  $N = (2^4 - 1) - 5$

  $N = 1111 - 0101 = 1010_{1's} = -5$

- Theoretically, in ones' complement we get the negation of a number by flipping (reversing) all the bits
- The name ones' complement comes from the fact that we could also get the negation of a number by subtracting each bit from 1
- Complement of a complement generates original number

Ones' Complement Representation by algebraic

$$\overline{N} = (2^n - 1) - N$$

- Ones' complement of +7
  - Since n=4, $(2^n - 1) = 1111$

$2^n - 1$	1	1	1	1	
N	0	1	1	1	(+7)
$\overline{N}$	1	0	0	0	(-7)

- Ones' complement of -7

$2^n - 1$	1	1	1	1	
N	1	0	0	0	(-7)
$\overline{N}$	0	1	1	1	(+7)

Shortcut method:

simply perfrom bitwise complement

1001 -> 0110

N	0	1	1	1	(+7)
$N_{2's}$	1	0	0	0	(-7)

Addition and Subtraction : Ones Complement

$$\begin{array}{rr} 4 & 0100 \\ \underline{+\ 3} & \underline{0011} \\ 7 & 0111 \end{array}$$

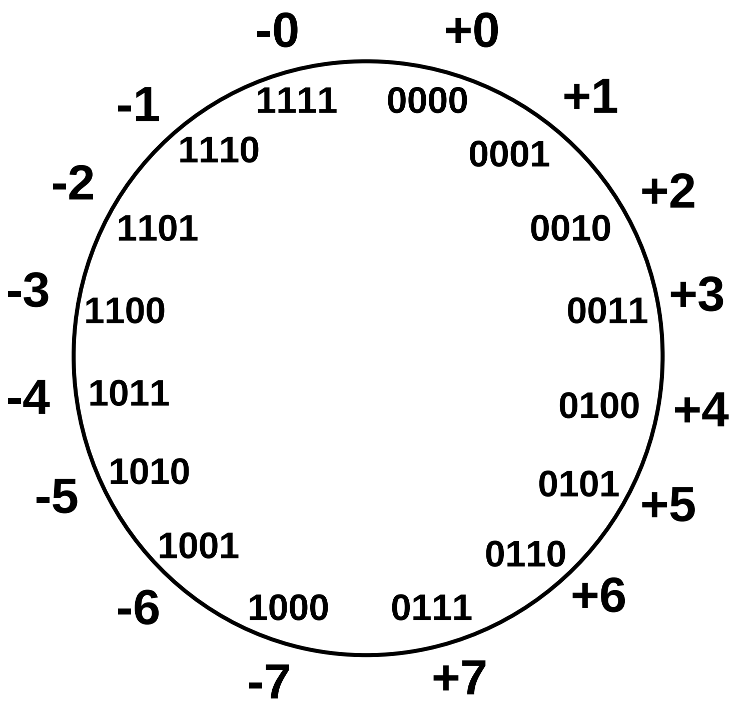
$$\begin{array}{rr} -4 & 1011 \\ +\ \underline{(-3)} & \underline{1100} \\ -7 & 10111 \\ \text{End around carry} & \hookrightarrow \underline{1} \\ & 1000 \end{array}$$

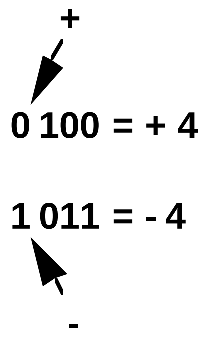
$$\begin{array}{rr} 4 & 0100 \\ \underline{-\ 3} & \underline{1100} \\ 1 & 10000 \\ \text{End around carry} & \hookrightarrow \underline{1} \\ & 0001 \end{array}$$

$$\begin{array}{rr} -4 & 1011 \\ \underline{+\ 3} & \underline{0011} \\ -1 & 1110 \end{array}$$

Ones' Complement Representation

$\overline{N} = (2^n - 1) - N$





Summary

- Some complexities in addition
- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems

Two's Complement

- Two's complement defined as

$$N^* = 2^n - N \text{ for } N \neq 0$$
$$0 \text{ for } N = 0$$

- Example:

Find 2's complement representation of 5 (i.e.-5) in 4 –bits

$5^* = 2^4 - 5$

$= 16 - 5 = 11 = 1011_{2's} = 1101 = -5$

- Two's complement is just a 1 added to 1's complement:

2's = 1's +1

- Complement of a complement generates original number

Two's Complement Representation

$N^* = 2^n - N$

- Two's complement of +7

2^n	1 0000	
N	0111	(+7)
N*	1001	(-7)

- Two's complement of -7

2^n	1 0000	
N	1001	(-7)
N*	0111	(+7)

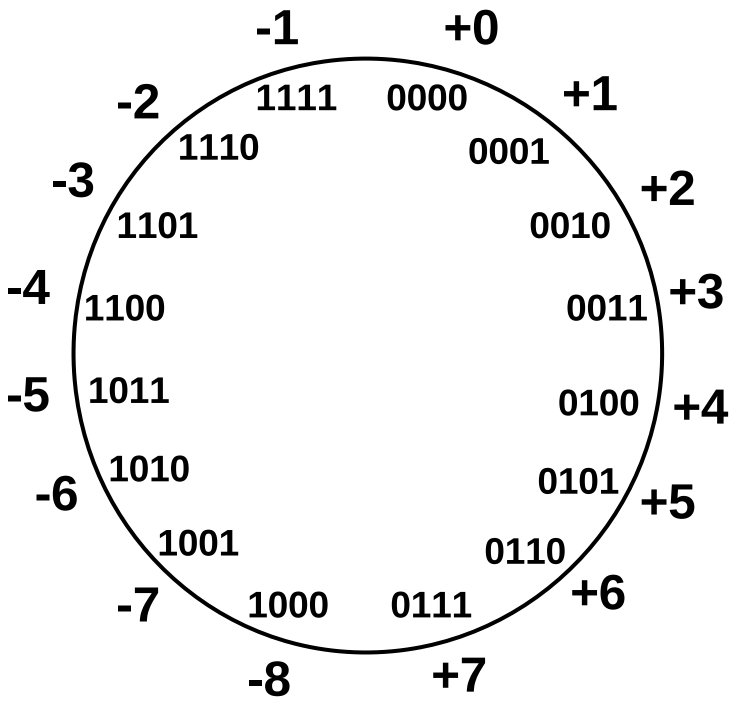
Shortcut method:
Twos complement = bitwise complement + 1

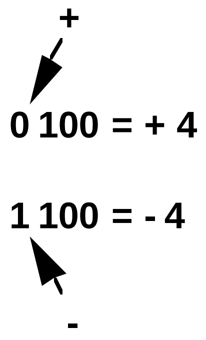
0111 -> 1000 + 1 -> 1001 (representation of -7)

1001 -> 0110 + 1 -> 0111 (representation of 7)

Twos Complement Representation

like 1's comp except shifted one position clockwise





- Only one representation for 0
- One more negative number than positive number

Finding 2's Complement

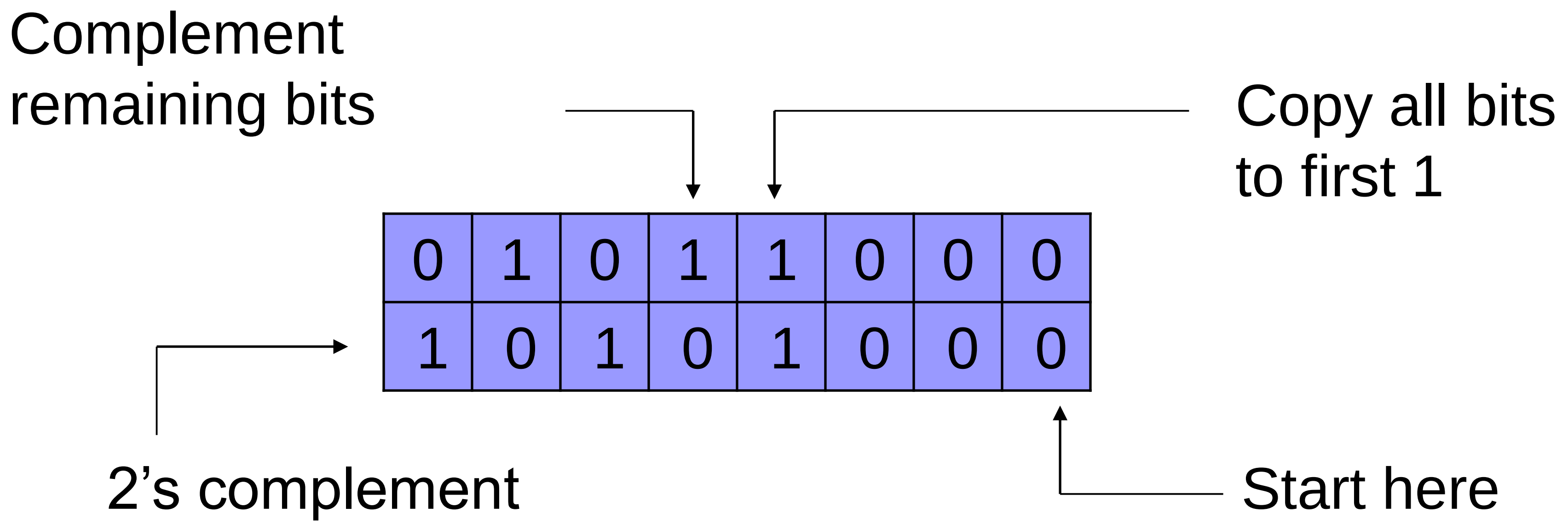


Table 2-6 Decimal and 4-bit numbers.

Decimal	***Two's Complement***	***Ones' Complement***	***Signed Magnitude***
–8	1000	—	—
–7	1001	1000	1111
–6	1010	1001	1110
–5	1011	1010	1101
–4	1100	1011	1100
–3	1101	1100	1011
–2	1110	1101	1010
–1	1111	1110	1001
0	0000	1111 or 0000	1000 or 0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	0101	0101
6	0110	0110	0110
7	0111	0111	0111

Range of Numbers

- 4-bit 2s complement
  - +7 = 0111 = 2^{3}-1
  - -8 = 1000 = -2^{3}
- 8 bit 2s complement
  - +127 = 01111111 = 2^{7} -1
  - -128 = 10000000 = -2^{7}
- 16 bit 2s complement
  - +32767 = 011111111 11111111 = 2^{15} - 1
  - -32768 = 100000000 00000000 = -2^{15}
- N bit 2s complement
  - 011111111..11111111 = 2^{N-1} – 1 (largest positive)
  - 100000000..00000000 = -2^{N-1} (largest negative)

Conversion Between Lengths, e.g. 8 → 16

- Positive number: add leading zeros
  - +18 =   00010010
  - +18 = 00000000 00010010
- Negative numbers: add leading ones
  - -18 =   11101110
  - -18 = 11111111 11101110
  - i.e. pack with msb (sign bit)

called “sign extension”

Addition and Subtraction

- a – b = ?
- Normal binary addition
- Monitor sign bit of result for overflow

- Take negation of b and add to a
  - i.e. a – b = a + (– b)
- So we only need addition and complement circuits

Addition and Subtraction : Twos Complement

$$\begin{array}{rr} 4 & 0100 \\ \underline{+\ 3} & \underline{0011} \\ 7 & 0111 \end{array} \qquad \begin{array}{rr} -4 & 1100 \\ +\ \underline{(-3)} & \underline{1101} \\ -7 & 11001 \end{array}$$

If carry-in to sign = carry-out then ignore carry

$$\begin{array}{rr} 4 & 0100 \\ \underline{-\ 3} & \underline{1101} \\ 1 & 10001 \end{array} \qquad \begin{array}{rr} -4 & 1100 \\ \underline{+\ 3} & \underline{0011} \\ -1 & 1111 \end{array}$$

If carry-in to sign = carry-out then ignore carry

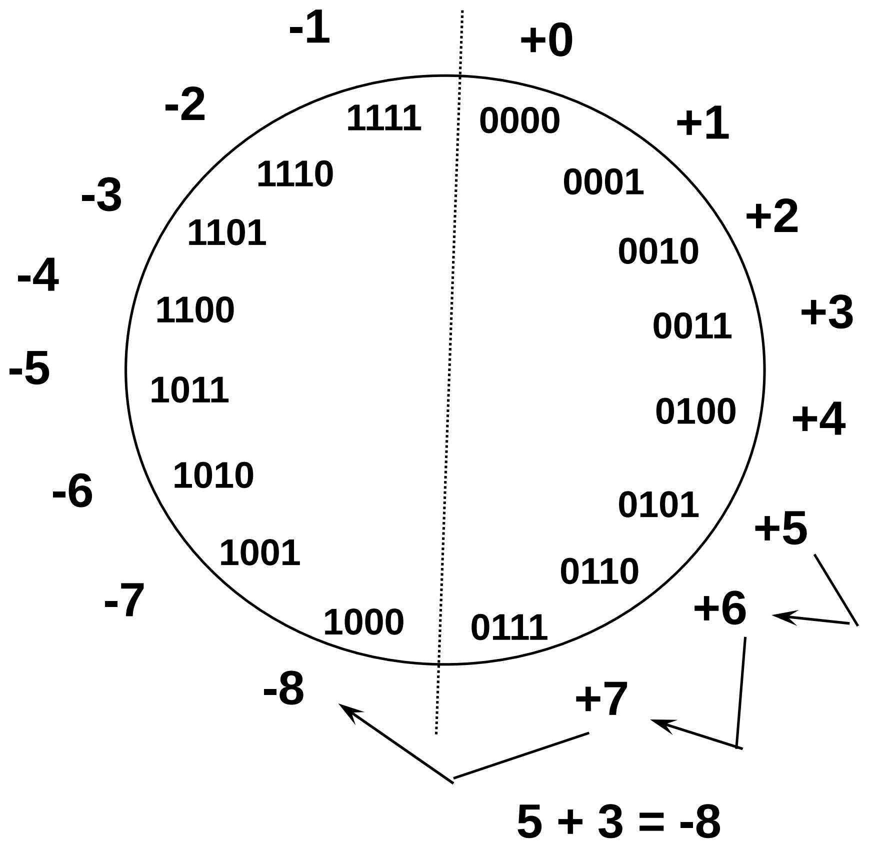
- Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

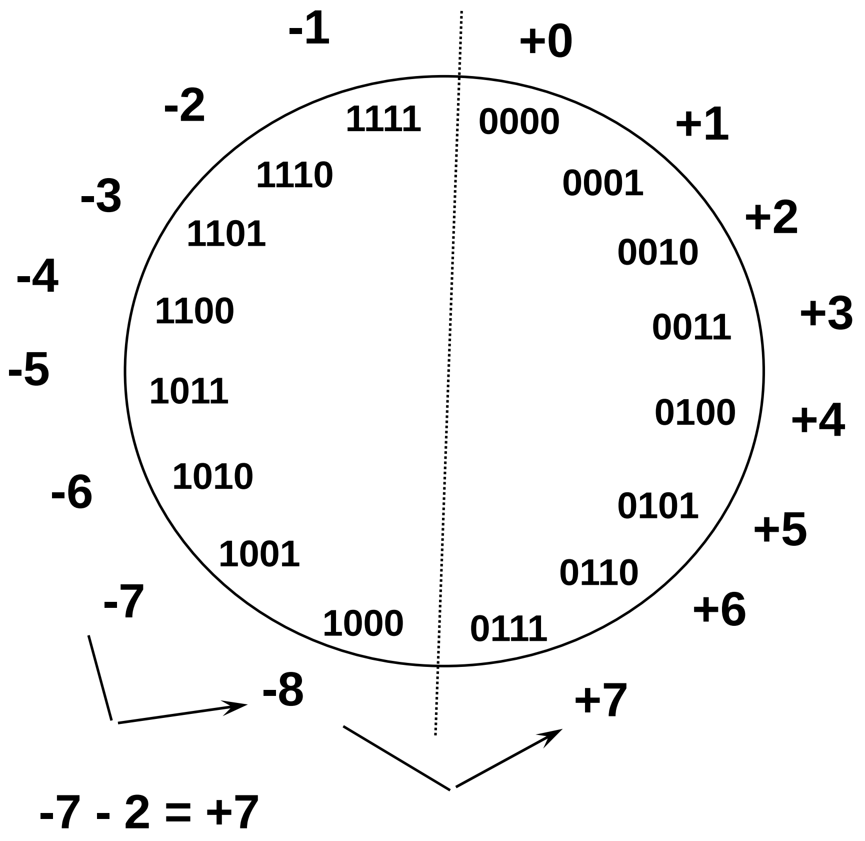
Addition and Subtraction : Twos Complement

- Why can the carry-out be ignored?
- Add 2's complement of N to M
  - This is M - N = M + N*
- If M $\geq$ N, will generate carry
  - M + N* = M + (2^n – N) = M – N + 2^n
  - Discard carry: just like subtracting 2^n
  - Result is positive M - N
- If M < N, no carry

Overflow Conditions

- Add two positive numbers to get a negative number
- or two negative numbers to get a positive number





Twos Complement Overflow

	Expected	$\boxed{0\quad 1}$ 1 1	Actual
	+5	0 1 0 1	
+	+3	0 0 1 1	
	+8	1 0 0 0	-8

Overflow

	Expected	$\boxed{1\quad 0}$ 0 0	Actual
	-7	1 0 0 1	
+	-2	1 1 1 0	
	-9	0 1 1 1	+7

Overflow

	Expected	$\boxed{0\quad 0}$ 0 0	Actual
	+5	0 1 0 1	
+	+2	0 0 1 0	
	+7	0 1 1 1	+7

No Overflow

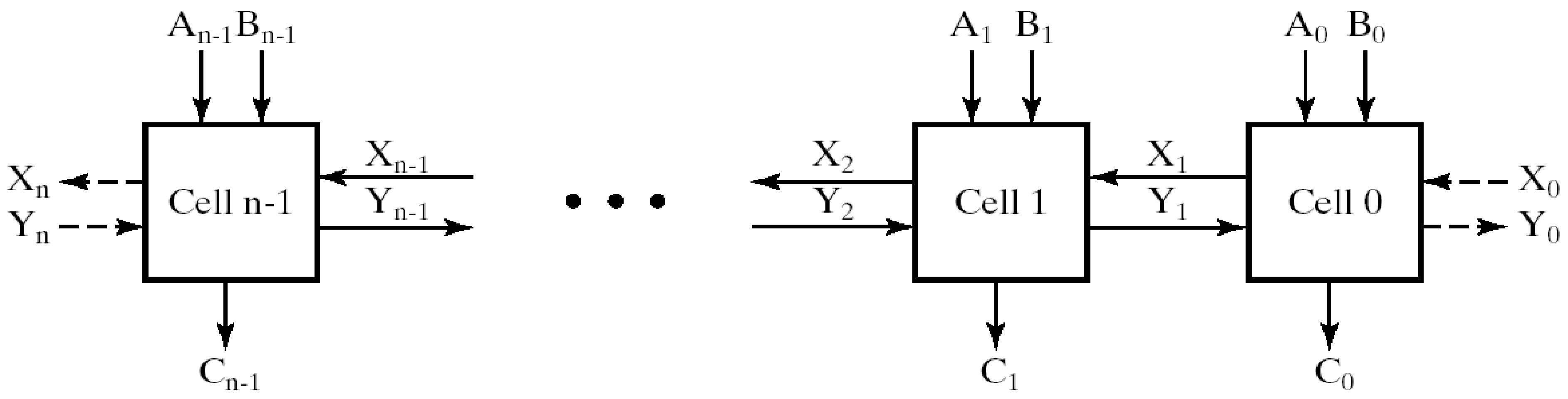
	Expected	$\boxed{1\quad 1}$ 1 1	Actual
	-3	1 1 0 1	
+	-5	1 0 1 1	
	-8	1 0 0 0	-8

No Overflow

Overflow when carry in to sign does not equal carry out

Iterative Circuit

- Like a hierarchy, except functional blocks per bit

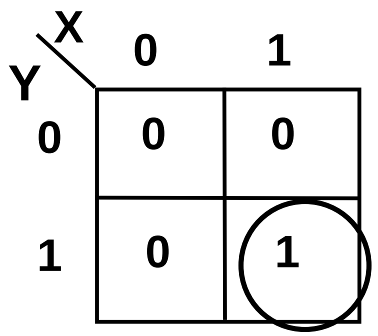


- Adders are a great example of this type of design
- Design 1-bit circuit, then expand
- Look at
  - Half adder – 2-bit adder, no carry in
    - Inputs are bits to be added
    - Outputs: result and possible carry
  - Full adder – includes carry in, really a 3-bit adder

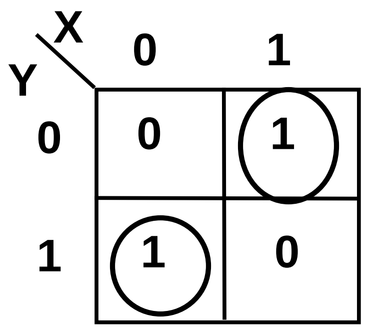
Half Adder

- Simplest adder block is “half adder”
  - Not very useful by itself

X	Y	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

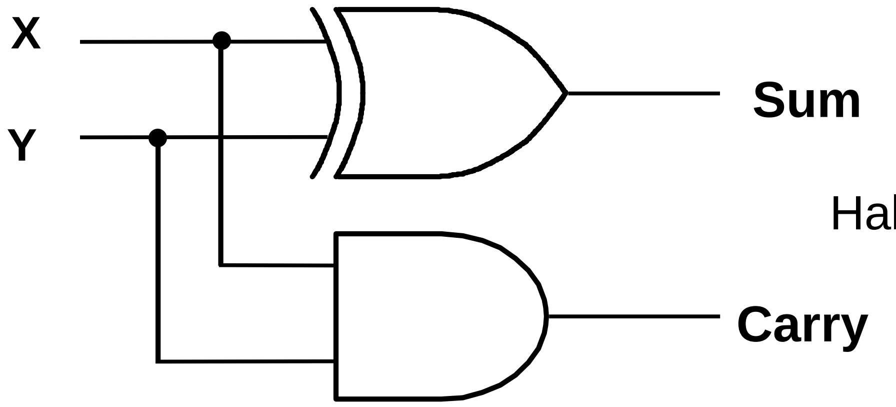


Carry = XY



Sum = X’ Y + X Y’

= X ⊕ Y



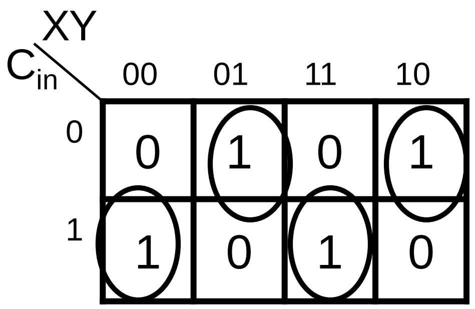
Half-adder Schematic

Full Adders

- Basic building block is full adder
- Many full-adders are combined to add more than 1 bits
- Truth table:

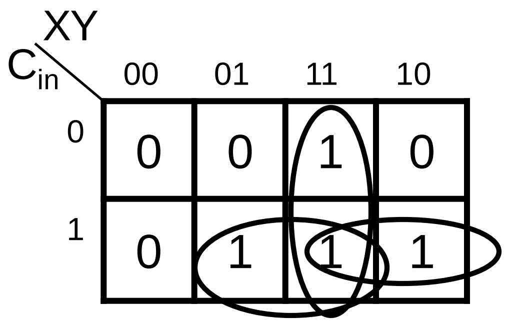
X	Y	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full Adder



C_{in} \ XY	00	01	11	10
0	0	1	0	1
1	1	0	1	0

$$S = X \oplus Y \oplus C_{in}$$

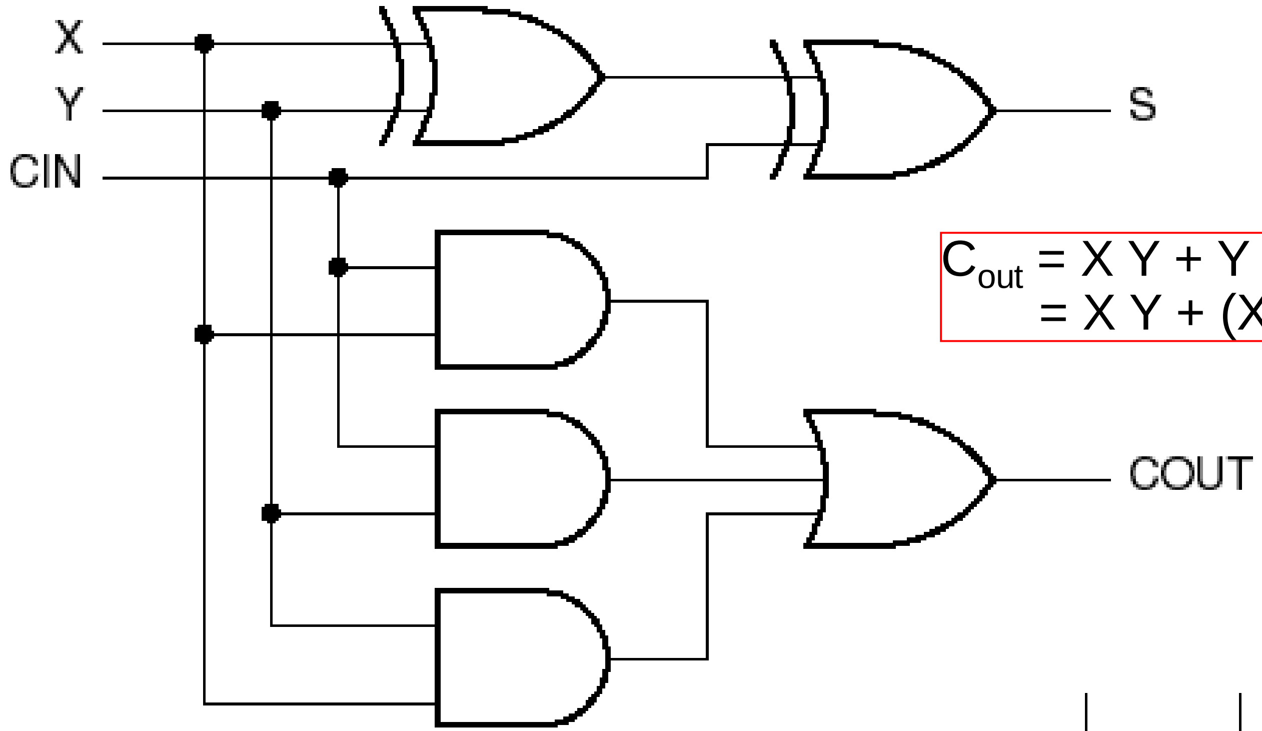


C_{in} \ XY	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$\begin{aligned} C_{out} &= X\,Y + X\,C_{in} + Y\,C_{in} \\ &= X\,Y + (X + Y)C_{in} \end{aligned}$$

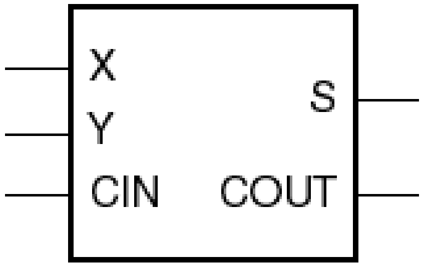
- In a multi-stage adder:
  - Variable i indicates stage number.
  - C_{in} is carry to i-th stage, also known as C_i
  - C_{out} is carry to next stage, also known as C_{i+1}

Full-adder circuit: Straightforward Approach

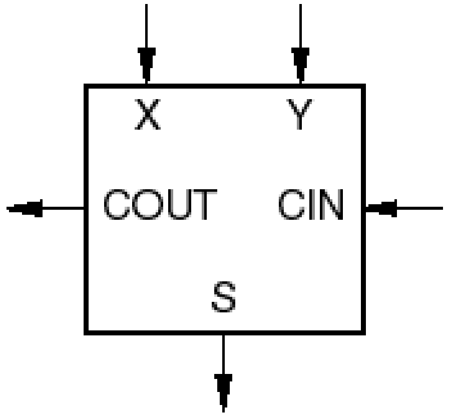


$$C_{out} = X\,Y + Y\,C_{in} + X\,C_{in}$$
$$= X\,Y + (X + Y)C_{in}$$

full adder

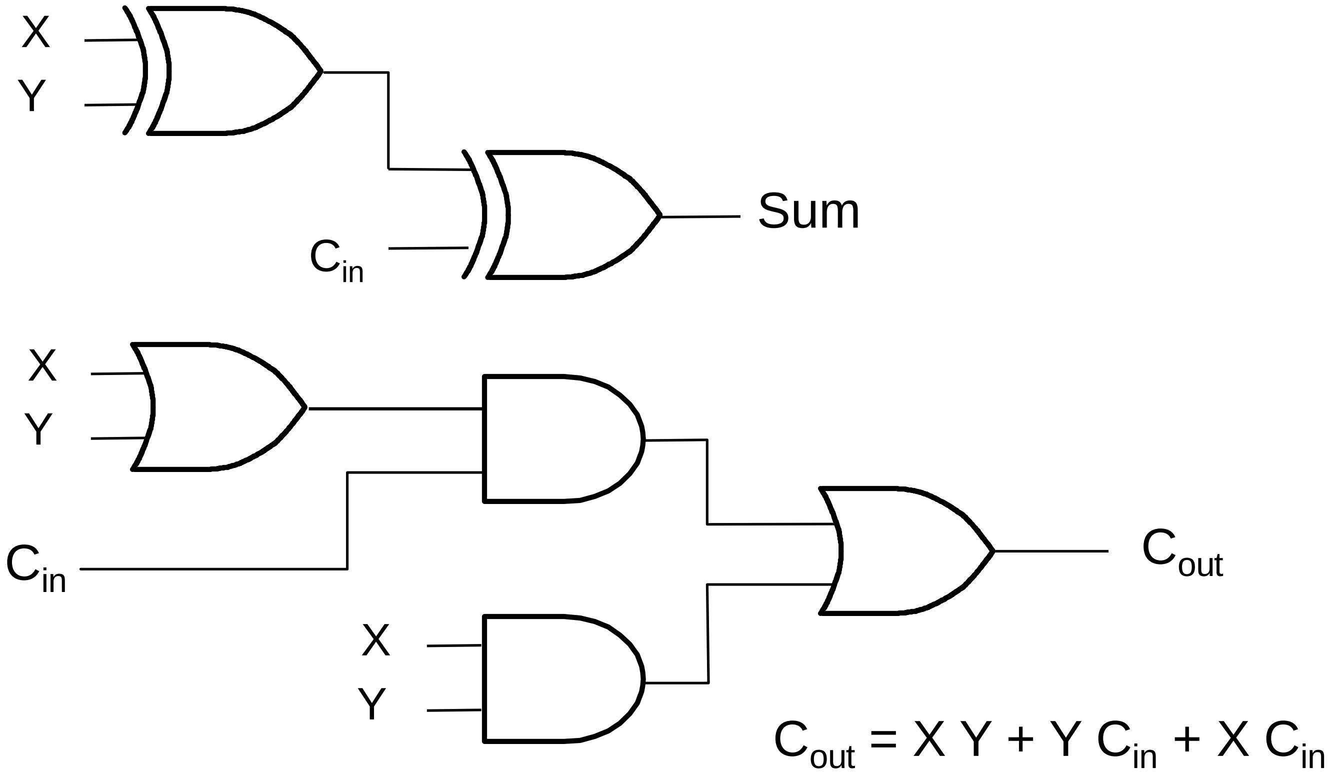


Cost: 6 Gates,
13 inputs



Full Adder: Alternative Implementation

- Cost: 6 Gates, max. 2 inputs per gate, 3 levels of logic
- Advantage: All gates of 2-input type, easier to do VLSI layout



$$C_{out} = X\,Y + Y\,C_{in} + X\,C_{in}$$
$$= X\,Y + (X + Y\,)C_{in}$$

Implementation with Two Half Adders (and an OR)

Cost: 5 Gates, 3 levels of logic

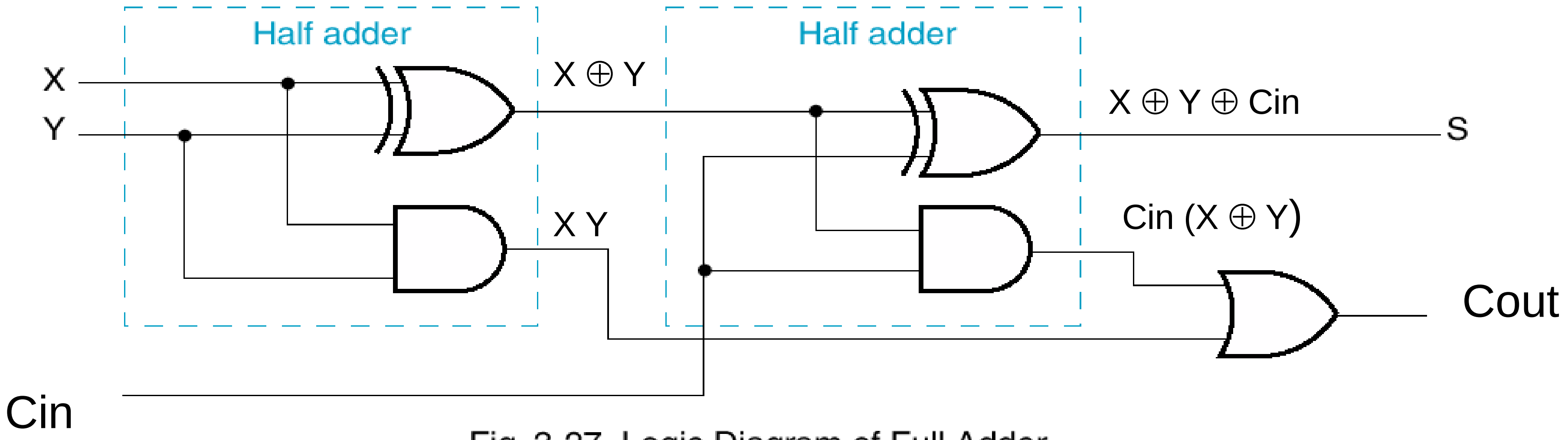
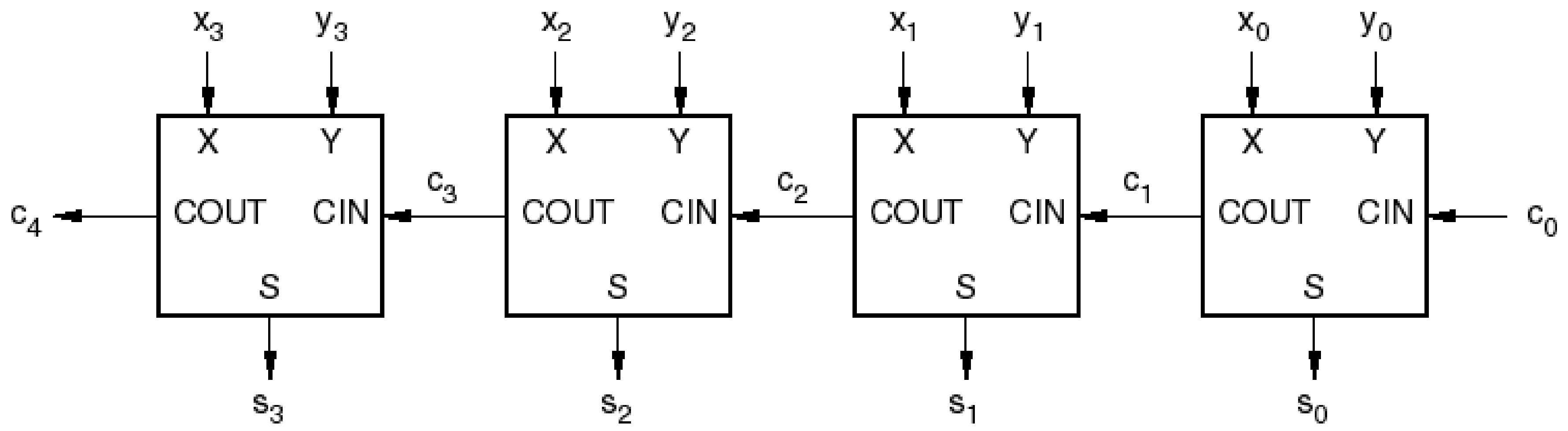


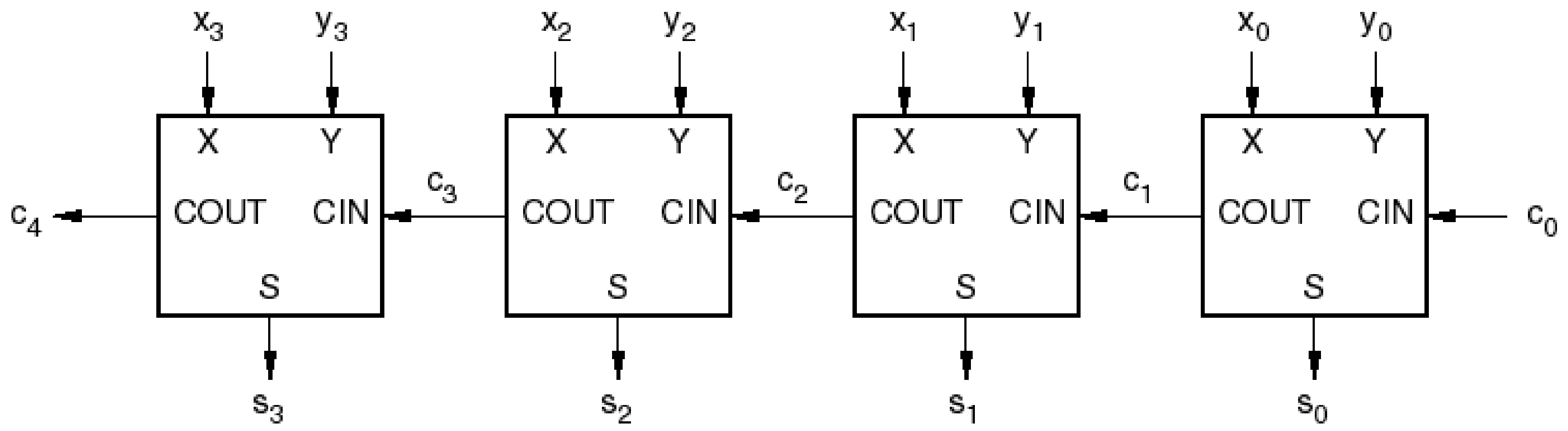
Fig. 3-27 Logic Diagram of Full Adder

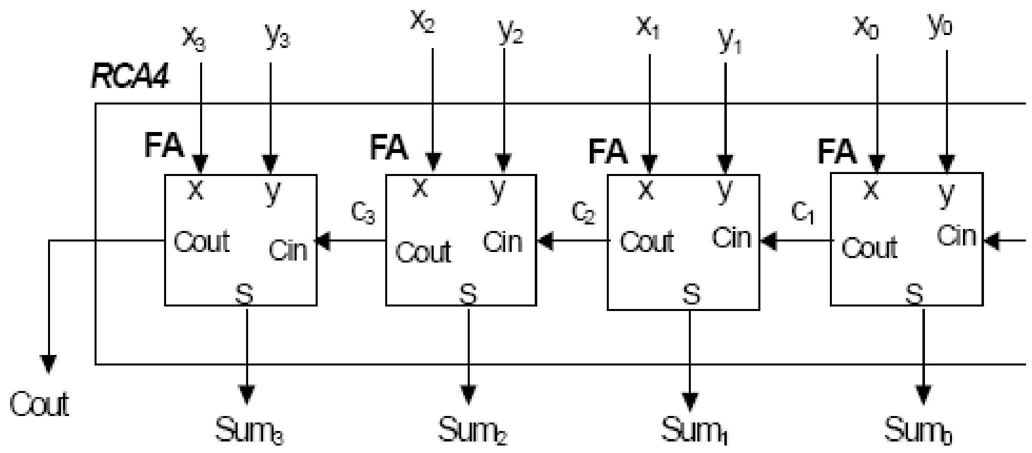
Ripple-Carry Adder



- Straightforward – connect full adders
- Carry-out to carry-in chain
  - C_0 in case this is part of larger chain, maybe just set to zero
- Speed limited by carry chain
- Faster adders eliminate or limit carry chain
  - 2-level AND-OR logic ==> 2^n product terms
  - 3 or 4 levels of logic, carry lookahead

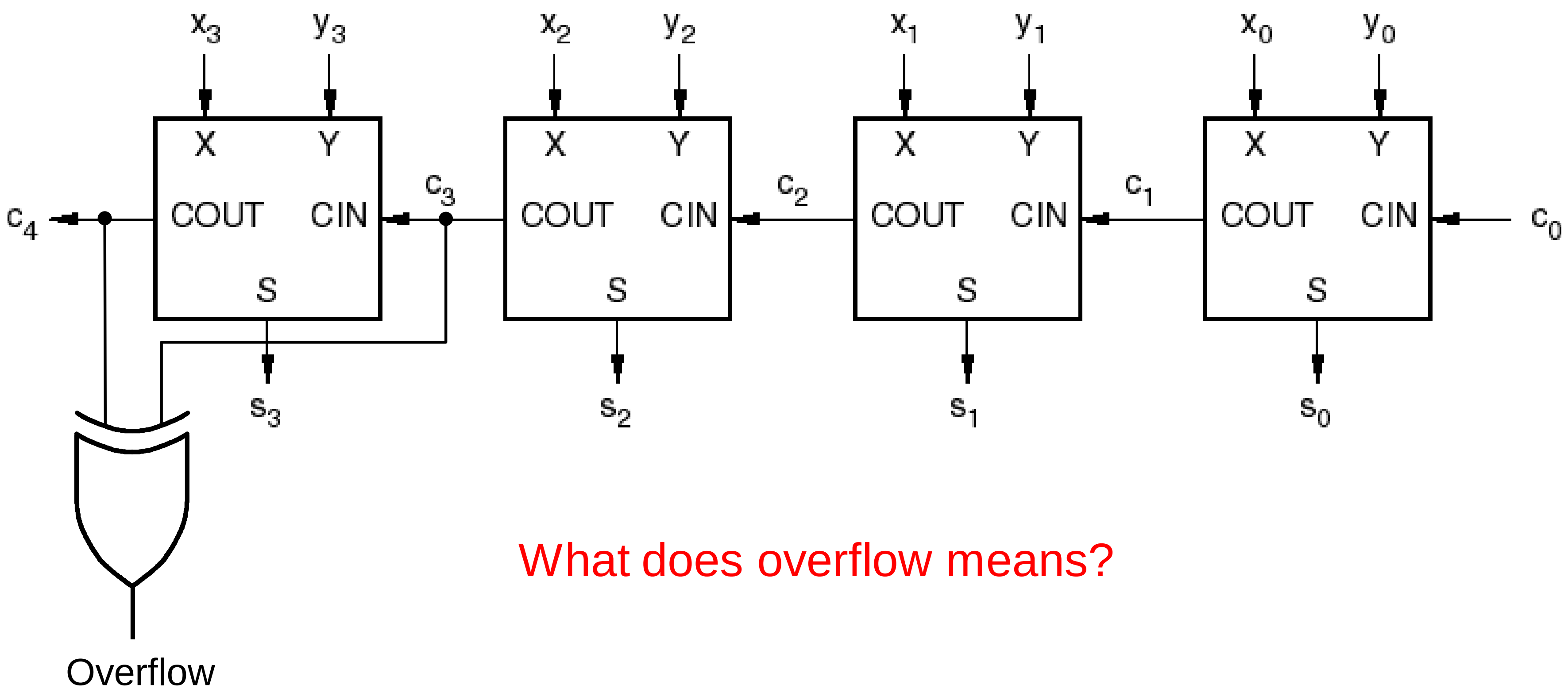
Ripple-Carry Adder





```
module RCA4 (Cin, x, y, Cout, Sum) ;
    input Cin; input [3:0] x, y; output [3:0] Sum; output Cout;
    wire c1, c2, c3;
    FA  u0 (.Cin(Cin), .A(x[0]), .B(y[0]), .Sum(Sum[0]), .Cout(c1));
    FA  u1 ( .Cin(c1), .A(x[1]), .B(y[1]), .Sum(Sum[1]), .Cout(c2)) ;
    FA  u2 ( .Cin(c2), .A(x[2]), .B(y[2]), .Sum(Sum[2]), .Cout(c3))  ;
    FA  u3 ( .Cin(c3), .A(x[3]), .B(y[3]), .Sum(Sum[3]), .Cout(Cout)) ;
endmodule
```

Overflow Detection

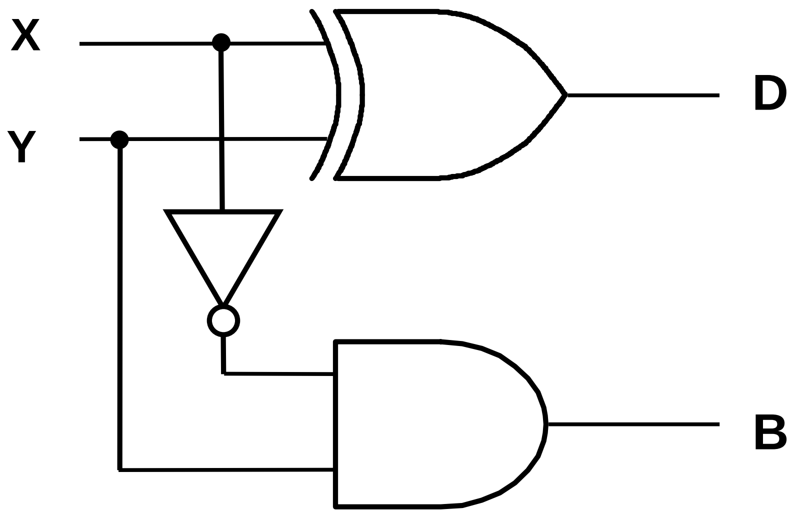


What does overflow means?

- If Overflow = 1, then overflow condition occurs. The output should not be used, i.e. the output is wrong. Condition is that either C_{n-1} or C_n is high, but not both (n = #stages)

Half Subtractor Circuit

X	Y		B	D
0	0	0 - 0	0	0
0	1	0 - 1	1	1
1	0	1 - 0	0	1
1	1	1 - 1	0	0

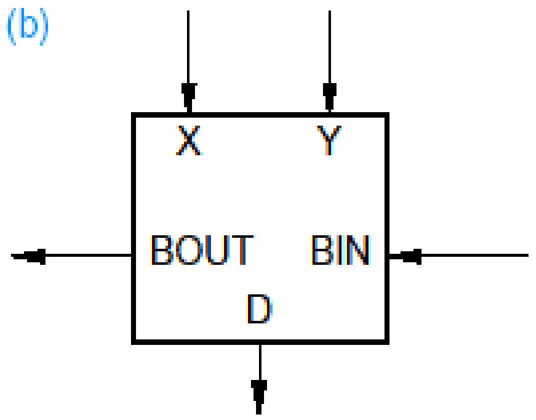


D=>Difference
D = X’Y + XY' = X $\oplus$ Y

B=>Borrow
B = X’Y

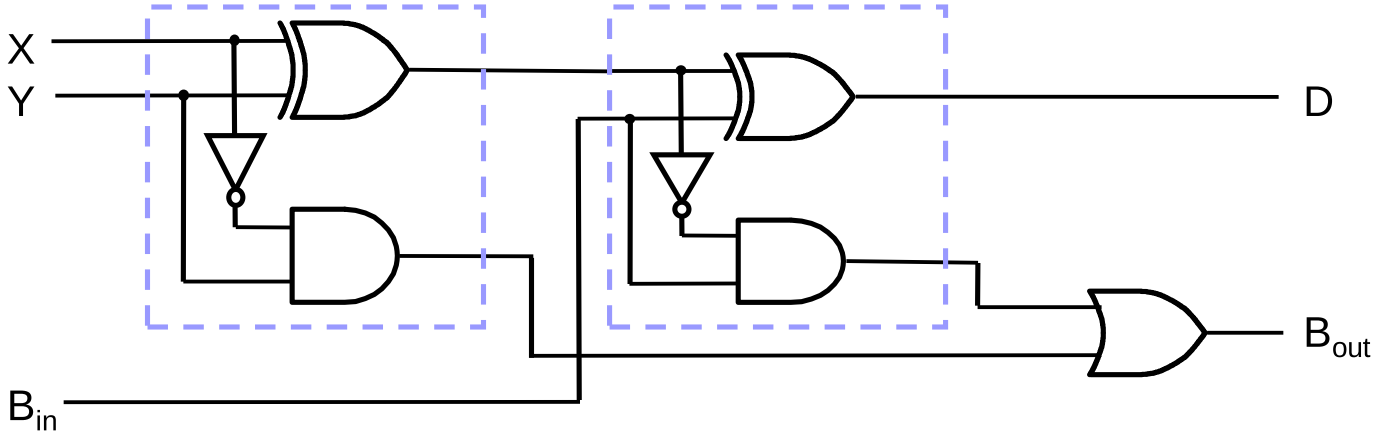
Full Subtractor Circuit

X	Y	B_{in}	B_{out}	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

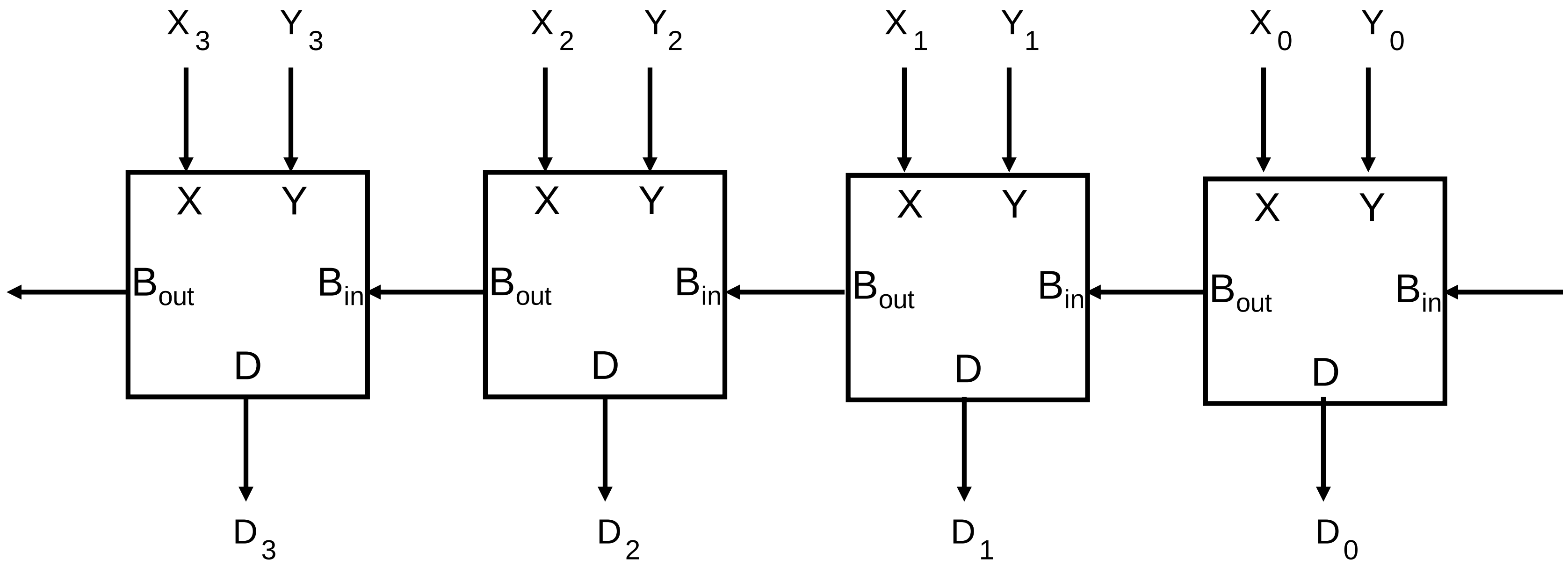


Difference
$D = X \oplus Y \oplus B_{in}$

Borrow out
$B_{out} = X'Y + X'B_{in} + YB_{in}$

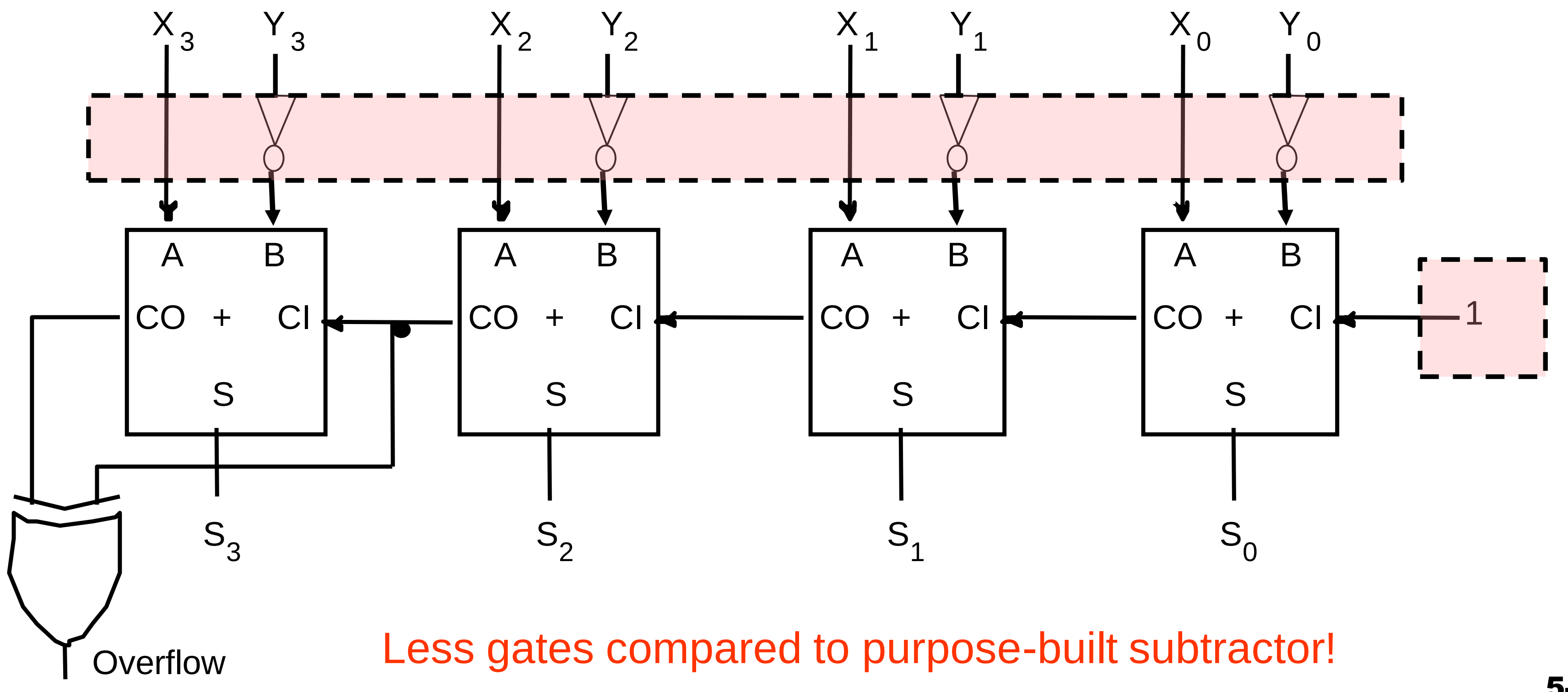


Multi-Stage Full Subtractor



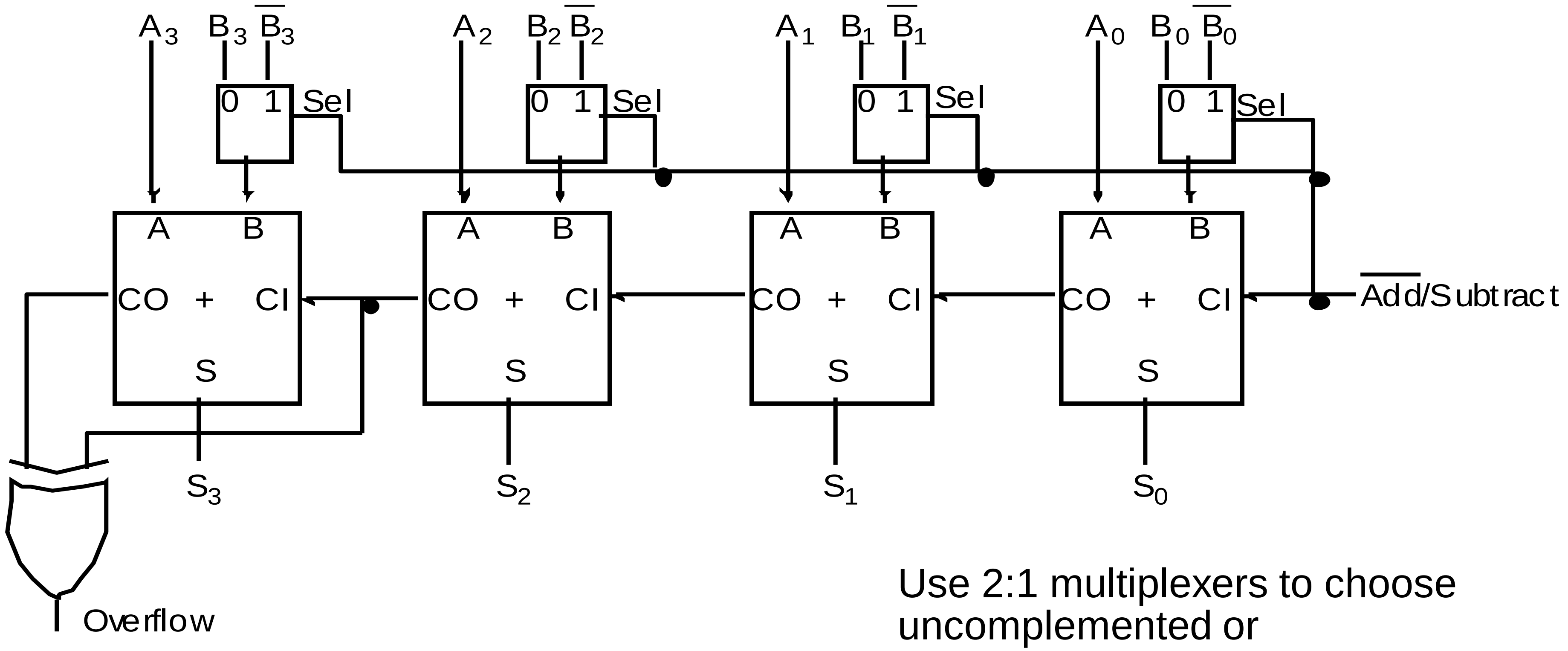
Subtraction Using Adders

- Subtraction is the same as addition of the two's complement.
- The two's complement is the bit-by-bit complement plus 1.
- Therefore, X – Y = X + Y' + 1 .
  - Complement Y inputs to adder, set C_I to 1.



Less gates compared to purpose-built subtractor!

Adder/Subtractor



Use 2:1 multiplexers to choose uncomplemented or complemented inputs

Remember, A - B = A + (-B) = A + $\overline{B}$ + 1
So when Add/Sub = 0, S = A + B
When Add/Sub = 1, S = A + $\overline{B}$ + 1 = A - B

Why use CO2 xor CO3 as overflow detection?

Alternative Design

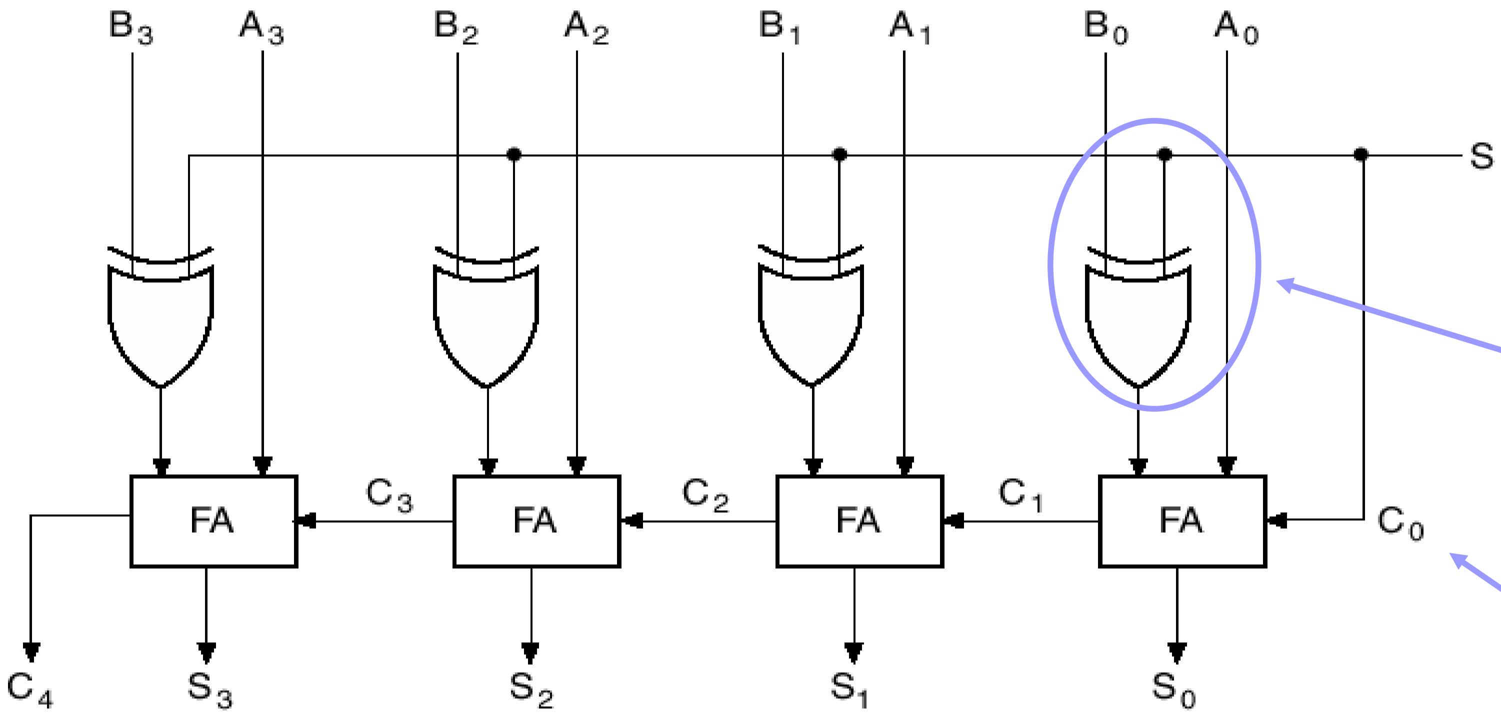


Fig. 3-31 Adder-Subtractor Circuit

S low for add, high for subtract

Inverts each bit of B if S is 1

Adds 1 to make 2's complement

- Output is 2's complement if B > A