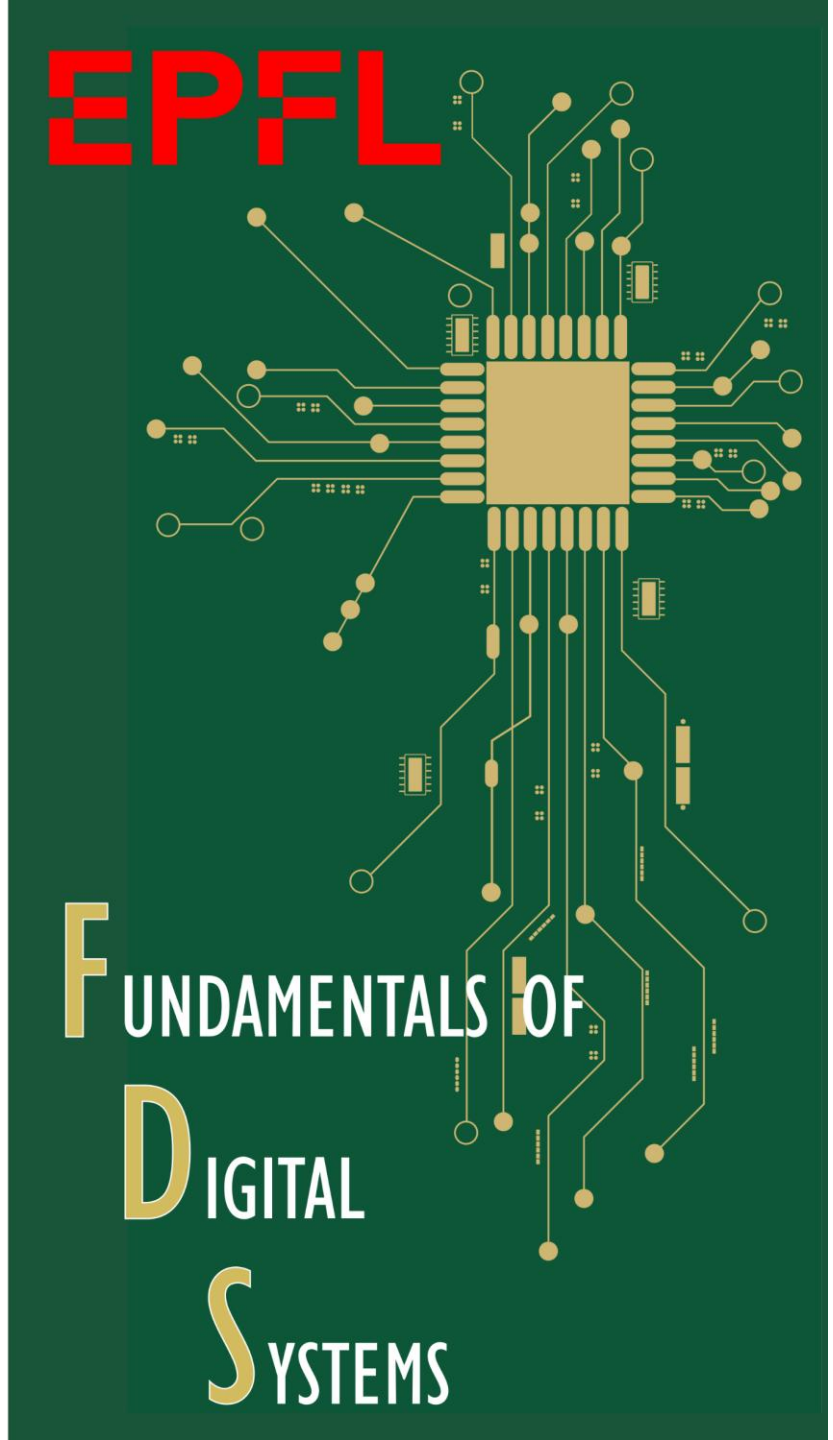


Digital Logic Circuits

Arithmetic Circuits

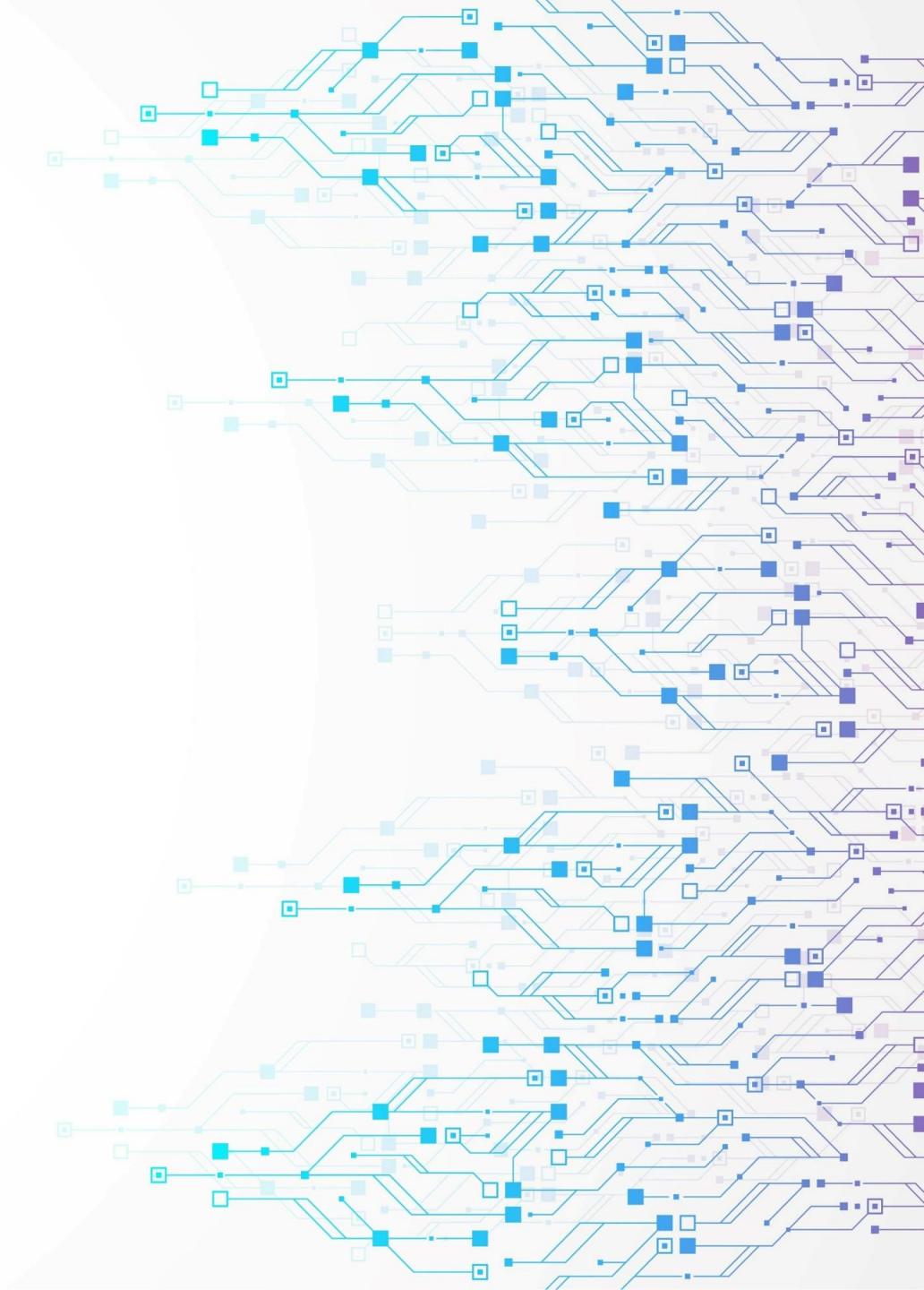
CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025

Previously on FDS

Logic synthesis, NAND and NOR networks, Don’t care conditions, XOR and XNOR gates, seven-segment display, and multiplexers



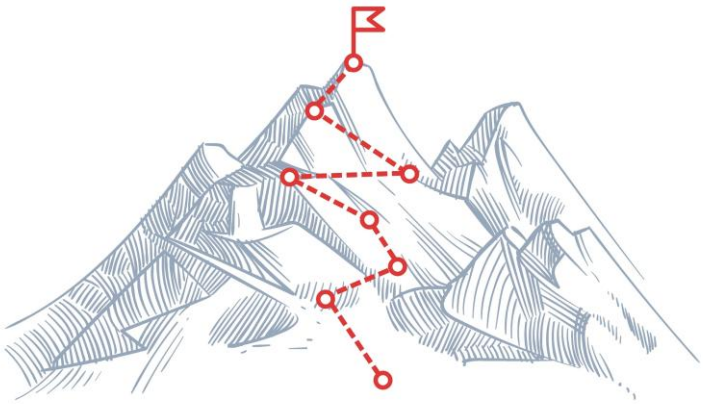
Previously

- Learned **sum-of-products** and **product-of-sums** forms for logic synthesis
  - From truth tables to logic circuits
- Discover techniques to convert an AND/OR/NOT logic network to a NAND-only or NOR-only equivalent
- Discovered the **don’t care conditions** and saw how using them wisely can help build efficient circuits
- Learned about **XOR** and **XNOR** gates
- Built multiplexers (**MUX**)

Let’s Talk About…

…Ways to build arithmetic circuits

Learning Outcomes



- Build basic arithmetic circuits for integer addition and subtraction
  - one-bit addition
  - one-bit subtraction
  - n-bit addition and subtraction in two's complement
- Find the worst-case circuit delay (i.e., the critical path delay)
- Build faster adders; reason about the advantages and disadvantages
- Use multiplexers for shifting efficiently

Quick Outline

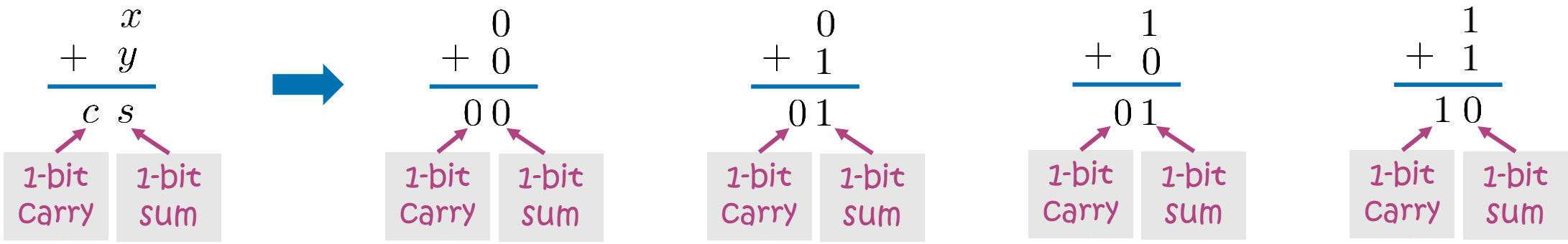
- Adders
  - Half-adder
  - Full-adder
  - Ripple-carry adder
- Subtractors
  - Full-subtractor
  - Ripple-carry subtractor
- Adders-Subtractors
  - Ripple-carry adder-subtractor
- Fast adders
  - Delay as performance metric
    - Full-adder
    - Full adder-subtractor
    - Ripple-carry adder-subtractor
  - Carry-select adder
- Shifters
  - Barrel shifter
    - Right
    - Bidirectional

Adders

- Half-adder
- Full-adder
- Ripple-carry adder

Addition of Two 1-Bit Binary Numbers

- Let us start from the simplest binary addition of one-bit numbers
  - The resulting sum is at most on two bits:
    - the rightmost bit is called **sum** (s)
    - the leftmost bit is called **carry** (c); it is produced as a carry-out when both bits being added are logical one



- Corresponding circuit is called **half-adder (HA)**

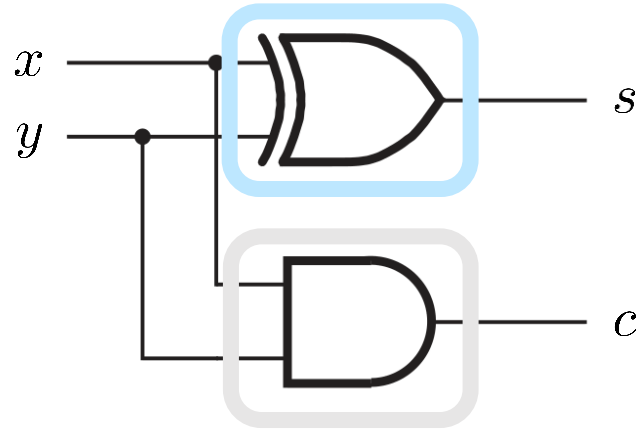
Half-Adder

Addition of Two 1-Bit Binary Numbers without the Input Carry

- Truth table

		sum	carry
x	y	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Digital logic circuit



- Logical expressions

$$s = \bar{x}y + x\bar{y} = x \oplus y$$

$$c = xy$$

- Graphical symbol

Addition of Two N-Bit Binary Numbers

- A binary n-bit adder has two operands $0 \leq x, y \leq 2^n - 1$ and a carry-in $c_{in} \in \{0, 1\}$ as inputs, and produces as outputs
  - sum $0 \leq s \leq 2^n - 1$ and
  - carry-out $c_{out} \in \{0, 1\}$ such that

$$x + y + c_{in} = 2^n c_{out} + s$$

- The solution to the above equation:

$$s = (x + y + c_{in}) \mod 2^n$$

$$c_{out} = \begin{cases} 1 & \text{if } (x + y + c_{in}) \geq 2^n \\ 0 & \text{otherwise} \end{cases}$$

$$= \lfloor (x + y + c_{in}) / 2^n \rfloor$$

Addition of Two N-Bit Binary Numbers

- It is **impractical** to start from the truth tables for n -bit addition
- Iterative approach:
  - Add each pair of bits at the position $i, 0 \leq i < n$
  - The addition at the bit position i needs to include a carry-in at the position i (i.e., carry-out from the position $i-1$); it also generates a carry-in for the position $i+1$

$$\begin{array}{cccccc} & \ldots & c_{i+1} & c_i & \ldots \\ & \ldots & \ldots & x_i & \ldots \\ + & \ldots & \ldots & y_i & \ldots \\ \hline & \ldots & \ldots & s_i & \ldots \end{array}$$

- The 1-bit adder reduces to a primitive module called **full-adder (FA)** with three binary inputs and two binary outputs such that

$$x_i + y_i + c_i = 2c_{i+1} + s_i$$

Full-Adder

Addition of Two 1-Bit Binary Numbers with the Input Carry (i.e., three 1-bit numbers)

- Truth table

x_i	y_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- Logical expressions

$$
\begin{aligned}
s_i &= \overline{x_i}\ \overline{y_i}\ c_i + \overline{x_i}\ y_i\ \overline{c_i} + x_i\ \overline{y_i}\ \overline{c_i} + x_i\ y_i\ c_i \\
&= (x_i\ y_i + \overline{x_i}\ \overline{y_i})c_i + (\overline{x_i}\ y_i + x_i\ \overline{y_i})\overline{c_i} \\
&= \overline{(x_i \oplus y_i)}c_i + (x_i \oplus y_i)\overline{c_i} = x_i \oplus y_i \oplus c_i
\end{aligned}
$$

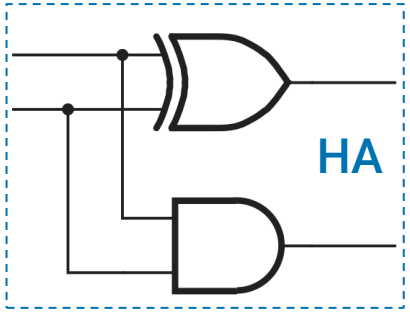
$$
\begin{aligned}
c_{i+1} &= \overline{x_i}\ y_i\ c_i + x_i\ \overline{y_i}\ c_i + x_i\ y_i\ \overline{c_i} + x_i\ y_i\ c_i \\
&= (\overline{x_i}\ y_i + x_i\ \overline{y_i})c_i + x_i\ y_i(\overline{c_i} + c_i) \\
&= (x_i \oplus y_i)c_i + x_i\ y_i
\end{aligned}
$$

- Further optimized:

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

Full-Adder From Half-Adders

Modular Implementation



- *Recall half-adder*

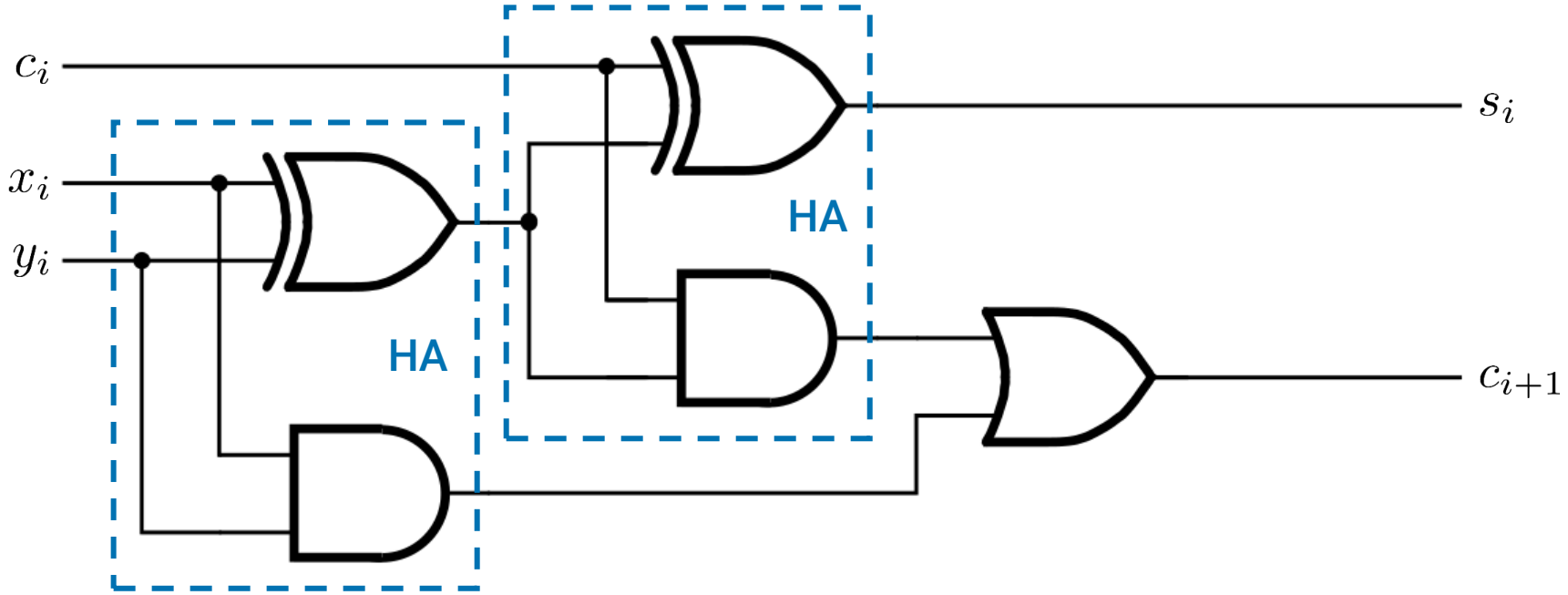
$$s = \bar{x}y + x\bar{y} = x \oplus y$$

$$c = xy$$

- Full-adder

$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = (x_i \oplus y_i)c_i + x_i\ y_i$$



Full-Adder

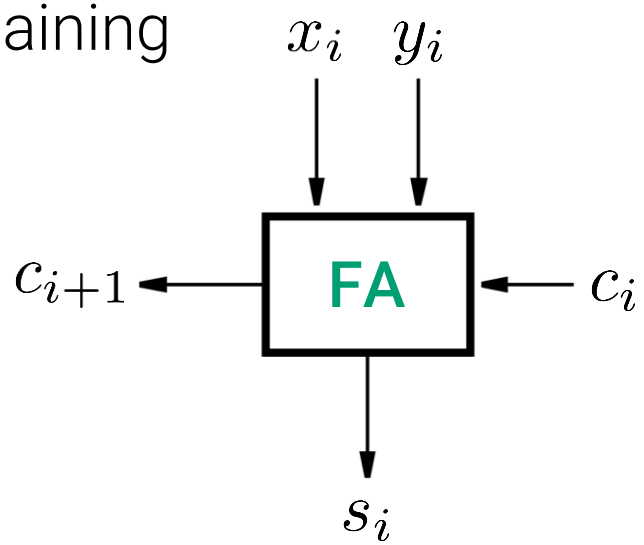
Graphical Symbol

- Final logical expressions

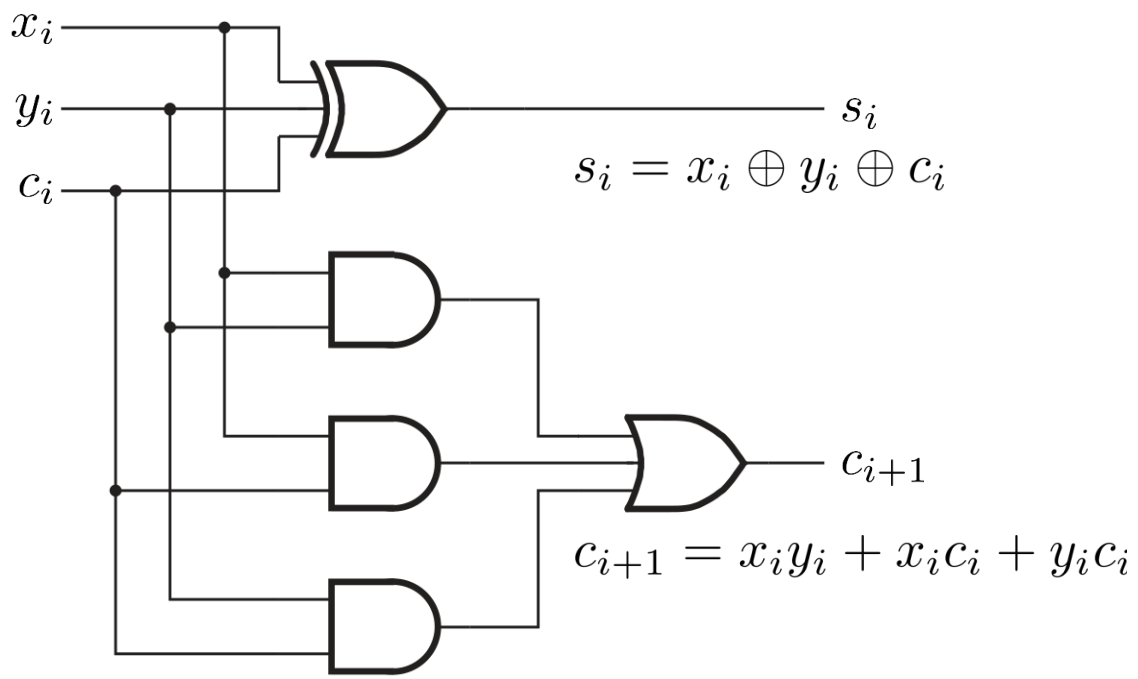
$$s_i = x_i \oplus y_i \oplus c_i$$

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

- Graphical symbol
  - Easy chaining



- Digital logic circuit

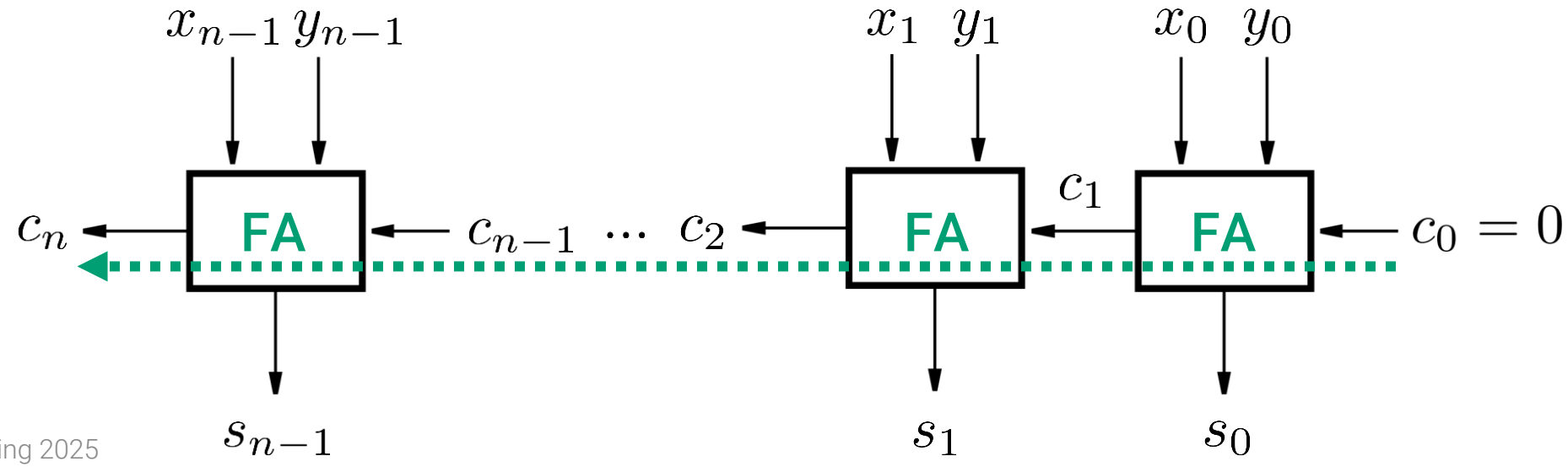


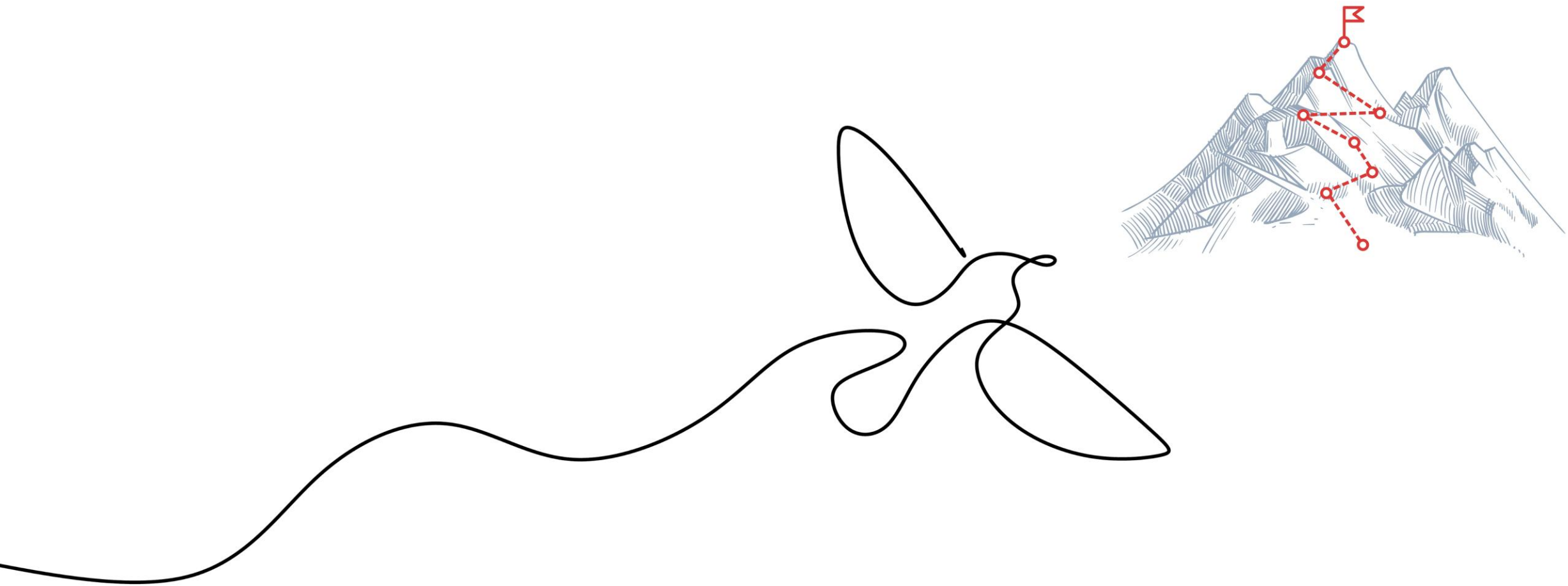
Basic Ripple-Carry Adder (RCA)

Adding Two N-bit Binary Numbers

- Starting from the least-significant digit, we add pairs of digits, progressing to the most-significant digit
- Carry "ripples" through the adder stages

$$\begin{array}{cccccc} & \ldots & c_{i+1} & c_i & \ldots \\ & \ldots & \ldots & x_i & \ldots \\ + & \ldots & \ldots & y_i & \ldots \\ \hline & \ldots & \ldots & s_i & \ldots \end{array}$$



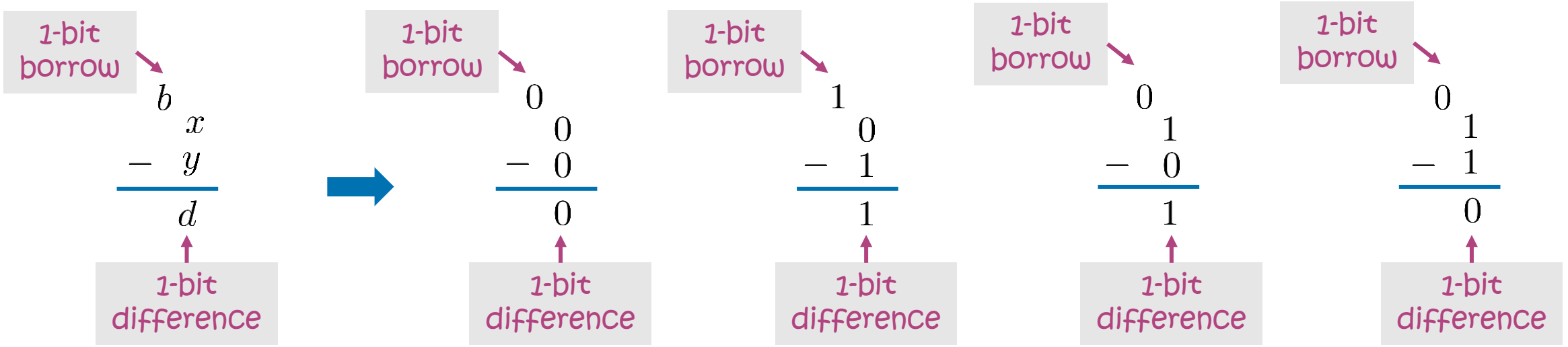


Subtractors

Arithmetic circuits

Subtraction of Two 1-Bit Binary Numbers

- Recall binary subtraction of two 1-bit binary numbers
  - Subtraction generates two bits:
    - **difference** (d), the result of the subtraction,
    - **borrow** (b), produced as a borrow-out when the subtrahend is larger than minuend



Subtraction of Two N-Bit Unsigned Numbers

- It is **impractical** to start from the truth tables for n-bit subtraction
- Iterative approach
  - Subtract each pair of bits at the position $i, 0 \leq i < n$
  - The subtraction at the bit position i needs to include a borrow-in at position i (i.e., borrow-out from the position $i-1$); it also generates a borrow-in for position $i+1$

$$
\begin{array}{cccccc}
 & \ldots & b_{i+1} & b_i & \ldots \\
 & \ldots & \ldots & x_i & \ldots \\
- & \ldots & \ldots & y_i & \ldots \\
\hline
 & \ldots & \ldots & d_i & \ldots
\end{array}
$$

Full Subtractor

Subtraction of Two 1-Bit Binary Numbers Taking into Account the Input Borrow

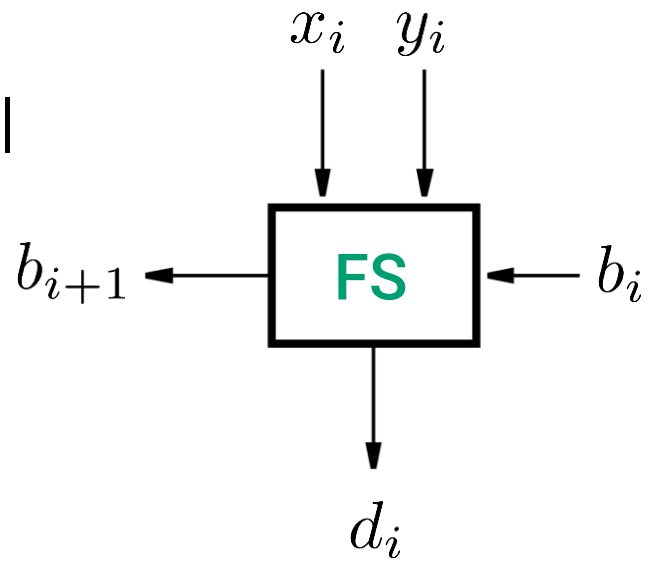
- Truth table

x_i	y_i	b_i	d_i	b_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$
\begin{aligned}
d_i &= \overline{x_i}\ \overline{y_i}\ b_i + \overline{x_i}\ y_i\ \overline{b_i} + x_i\ \overline{y_i}\ \overline{b_i} + x_i\ y_i\ b_i \\
&= (\overline{x_i}\ \overline{y_i} + x_i\ y_i)\ b_i + (\overline{x_i}\ y_i + x_i\ \overline{y_i})\ \overline{b_i} = \overline{(x_i \oplus y_i)}\ b_i + (x_i \oplus y_i)\ \overline{b_i} \\
&= x_i \oplus y_i \oplus b_i
\end{aligned}
$$

$$
\begin{aligned}
b_{i+1} &= \overline{x_i}\ \overline{y_i}\ b_i + \overline{x_i}\ y_i\ \overline{b_i} + \overline{x_i}\ y_i\ b_i + x_i\ y_i\ b_i \\
&\quad 7b.\ \ x + x = x \quad 8b.\ \ x + \bar{x} = 1 \\
&= (\overline{x_i}\ \overline{y_i}\ b_i + \overline{x_i}\ y_i\ b_i) + (\overline{x_i}\ y_i\ \overline{b_i} + \overline{x_i}\ y_i\ b_i) + (\overline{x_i}\ y_i\ b_i + x_i\ y_i\ b_i) \\
&= \overline{x_i}\ b_i + \overline{x_i}\ y_i + y_i\ b_i
\end{aligned}
$$

- Graphical symbol
  - Easy chaining

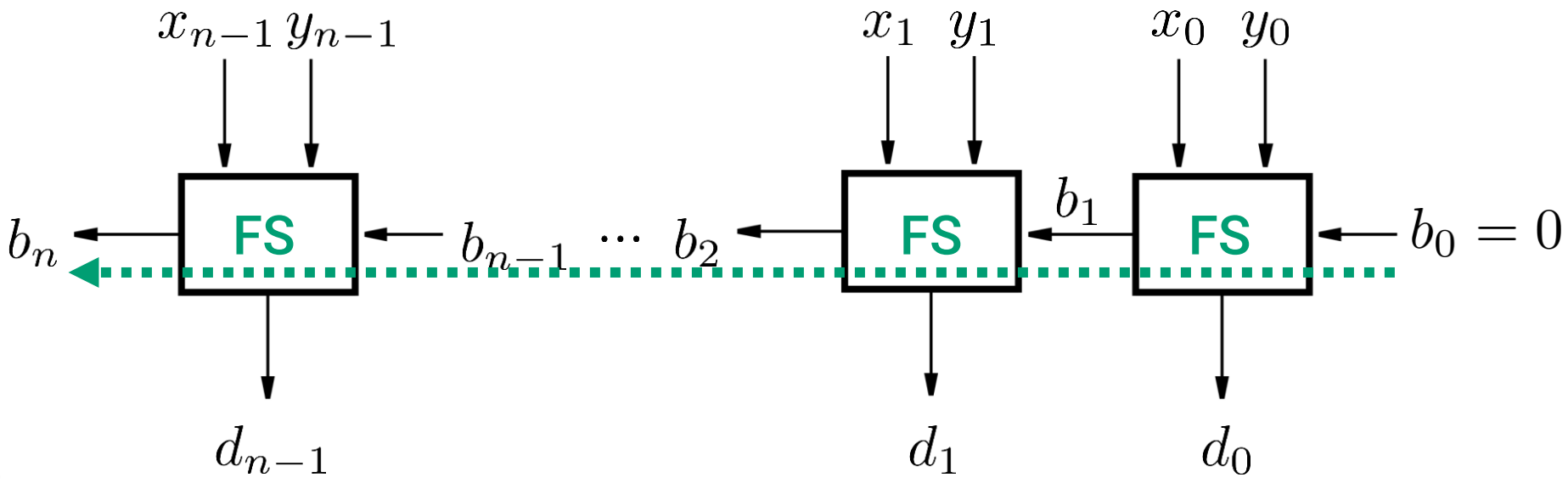


N-Bit Ripple-Carry Subtractor

Subtracting Two N-bit Binary Numbers

- Starting from the least-significant digit, we subtract pairs of digits, progressing to the most-significant digit

$$\begin{array}{cccccc} & \ldots & b_{i+1} & b_i & \ldots \\ & \ldots & \ldots & x_i & \ldots \\ - & \ldots & \ldots & y_i & \ldots \\ \hline & \ldots & \ldots & d_i & \ldots \end{array}$$



Adders-Subtractors

…in Two’s complement

Adders-Subtractors in Two's Complement

- Recall that subtracting two numbers in two's complement format requires using the two's complement of one operand:

$$X - Y = X + \overline{Y} + 1$$

- $\overline{Y}$ is obtained by complementing every bit of Y
- Assume a control signal op determines which **op**eration to perform ($op = 0$: addition, $op = 1$: subtraction)

Adders-Subtractors in Two's Complement

- Assume a control signal op determines which **op**eration to perform ($op = 0$: addition, $op = 1$: subtraction)

$$f(X,Y) = \begin{cases} X + Y, & \text{if } op = 0 \\ X + \overline{Y} + 1, & \text{otherwise} \end{cases}$$

$$f(X,Y,op) = \overline{op}(X+Y) + op(X + \overline{Y} + 1)$$

$$f(X,Y,op) = (\overline{op} + op)X + \overline{op}Y + op\ \overline{Y} + op$$

$$f(X,Y,op) = X + \overline{op}\ Y + op\ \overline{Y} + op$$

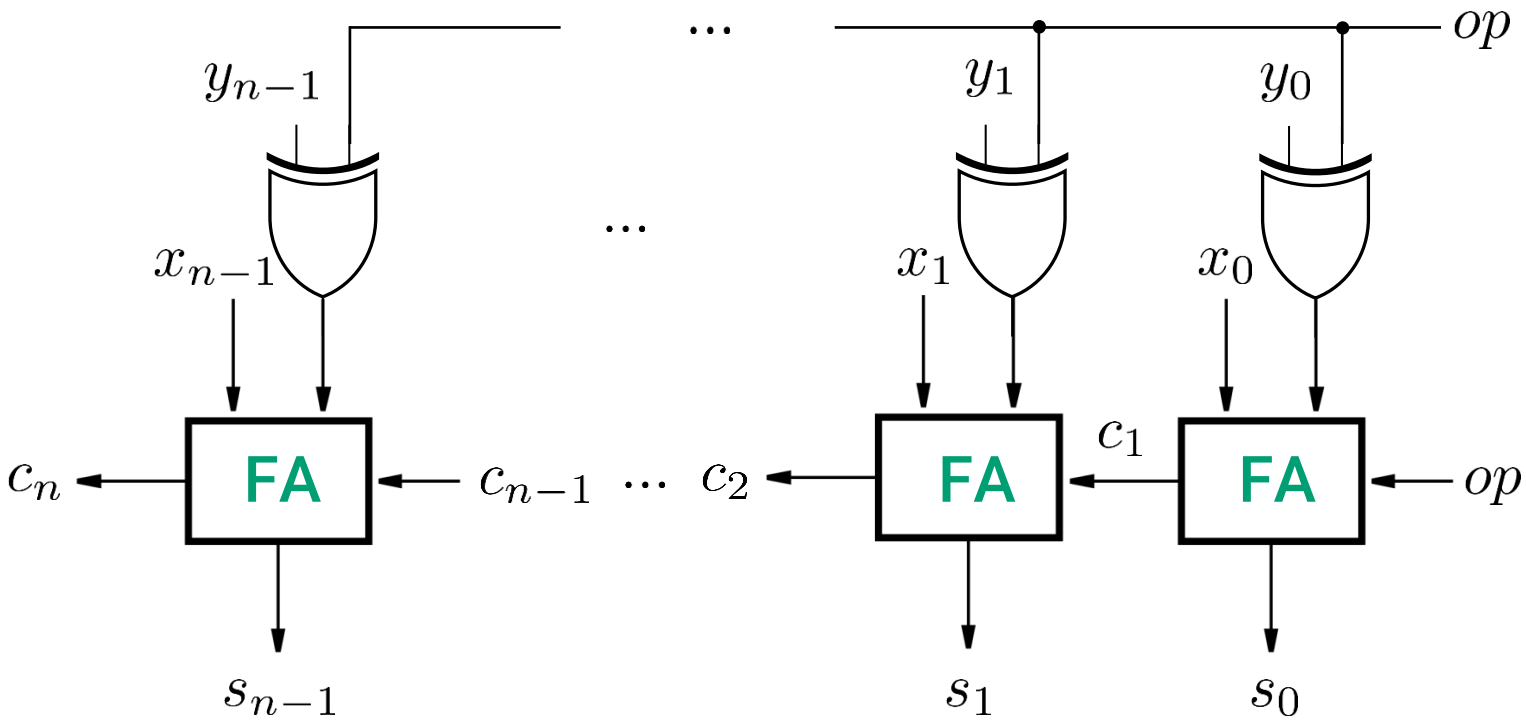
$$f(X,Y,op) = X + op \oplus Y + op$$

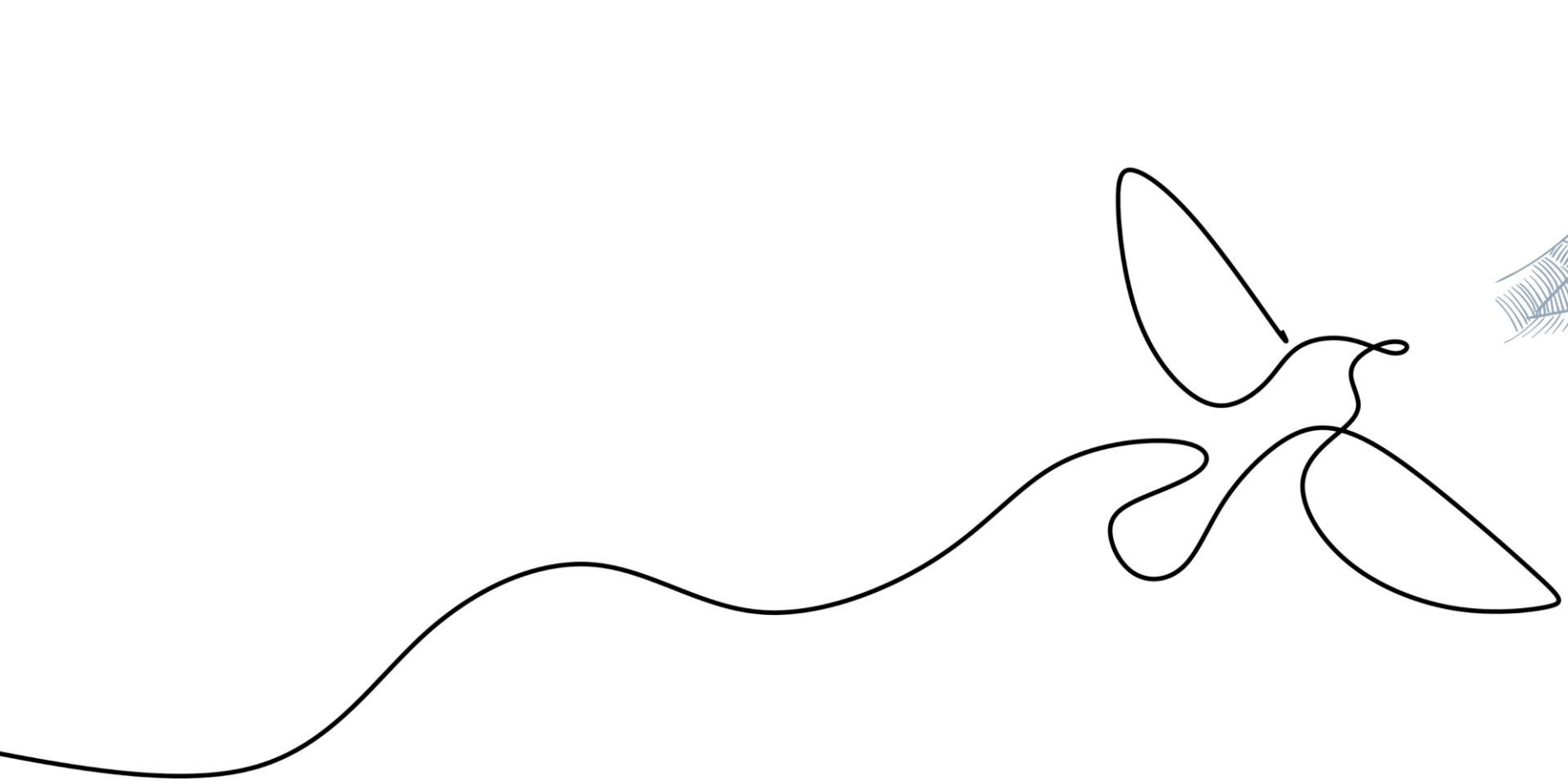
N-Bit Ripple-Carry Adder-Subtractor

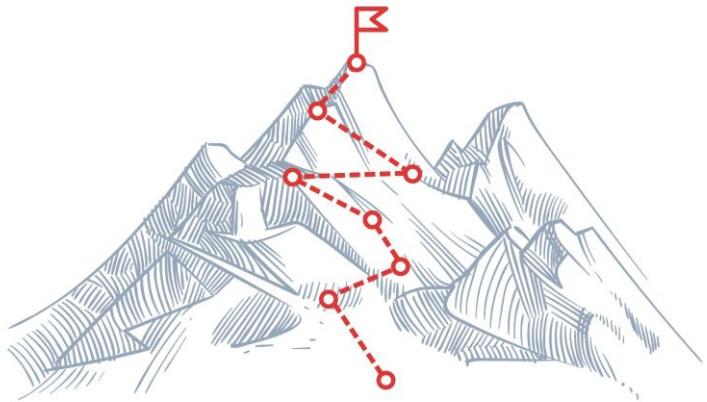
Two's Complement

- One circuit, able to perform two operations

$$f(X, Y, op) = X + op \oplus Y + op$$







Fast Adders

Carry Select vs. Ripple-Carry Adders

Performance Matters

- Addition and subtraction are fundamental operations performed frequently
  - How quickly a result can be produced greatly impacts the system's performance
  - Performance is determined by the worst-case delay
- System's value is measured as a ratio:

$$value = \frac{performance}{price}$$

- A large performance improvement can often be achieved at a modest price/cost increase

Full-Adder

Input-to-Output Delay, Assuming All Inputs are Available at time t = 0

- Delay to generate the sum

$$t(x_i, s_i) = t(y_i, s_i) = t(c_i, s_i) = t(\text{XOR})$$

- Delay to generate carry-out

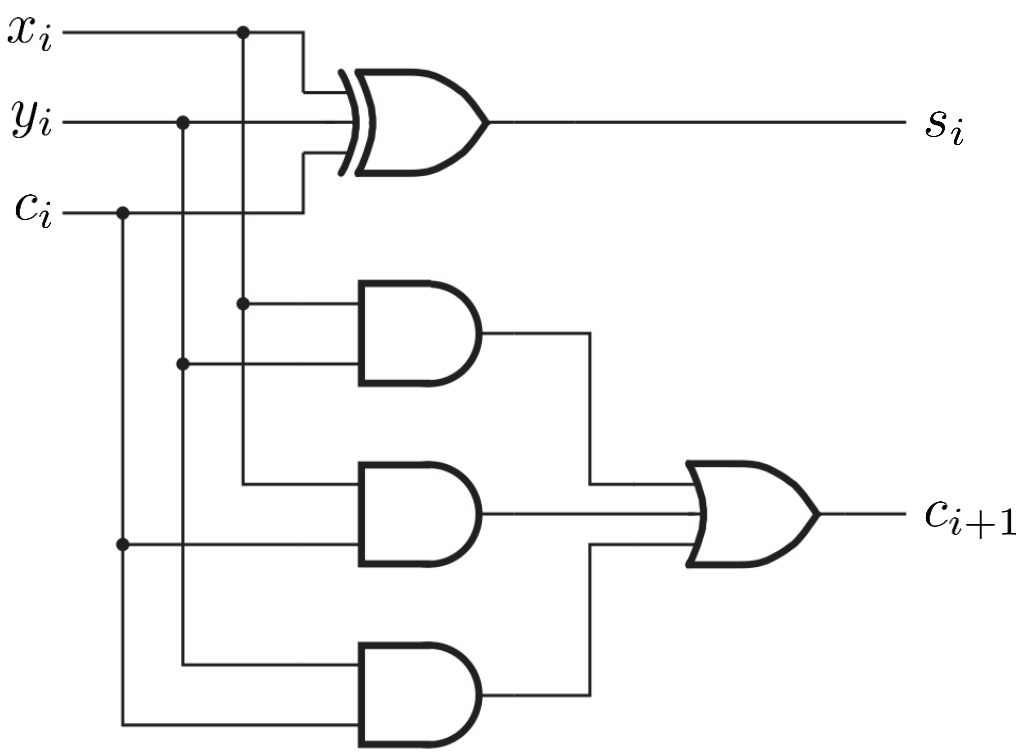
$$t(x_i, c_{i+1}) = t(y_i, c_{i+1}) = t(c_i, c_{i+1}) = t(\text{AND}) + t(\text{OR})$$

- Worst-case delay

$$t_{\max} = \max\left(t(s_i), t(c_{i+1})\right) = \max\left(t(\text{XOR}), t(\text{AND}) + t(\text{OR})\right)$$

  - If all gates had equal delays

$$t_{\max} = t(c_{i+1}) = 2 \text{ Gate Delays}$$



Full Adder-Subtractor

Input-to-Output Delay, Assuming All Inputs are Available at time t = 0

- Delay to generate the sum

$$t(x_i, s_i) = t(c_i, s_i) = t(\text{XOR})$$
$$t(y_i, s_i) = t(op, s_i) = 2t(\text{XOR})$$

- Delay to generate carry-out

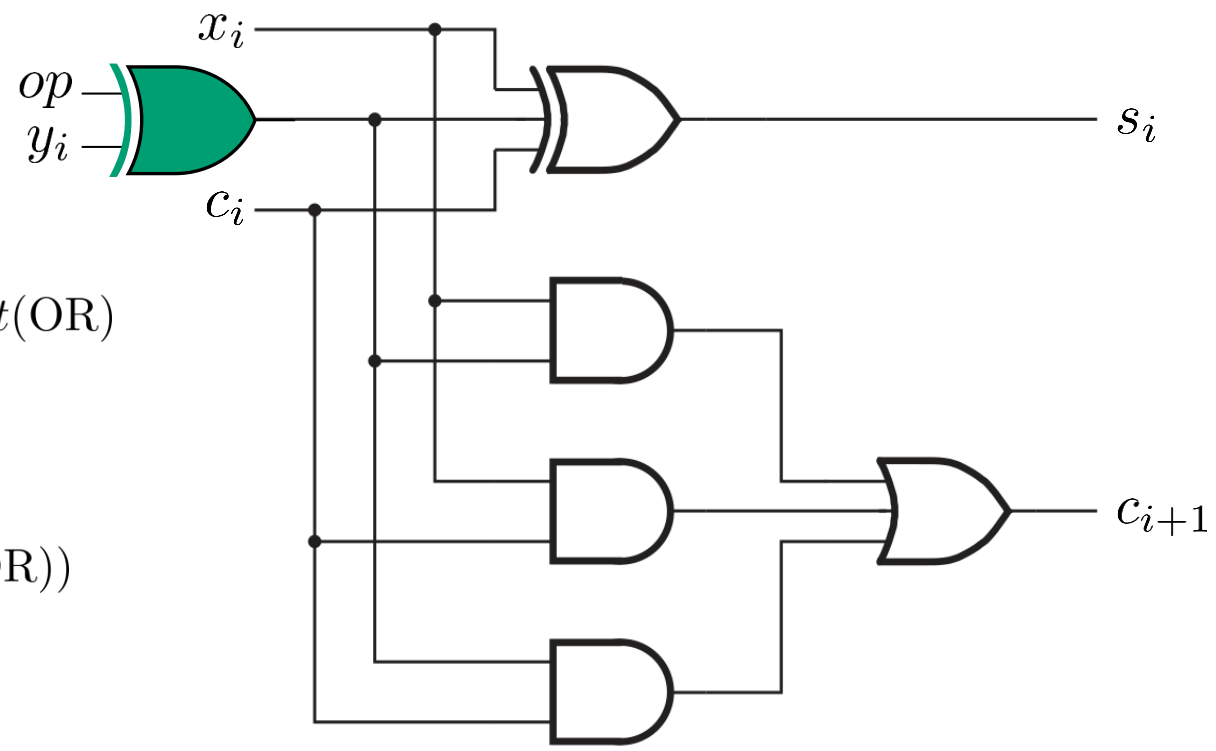
$$t(x_i, c_{i+1}) = t(c_i, c_{i+1}) = t(\text{AND}) + t(\text{OR})$$
$$t(y_i, c_{i+1}) = t(op, c_{i+1}) = t(\text{XOR}) + t(\text{AND}) + t(\text{OR})$$

- Worst-case delay

$$t_{\max} = \max\left(t(s_i), t(c_{i+1})\right)$$
$$= \max\left(2t(\text{XOR}), t(\text{XOR}) + t(\text{AND}) + t(\text{OR})\right)$$

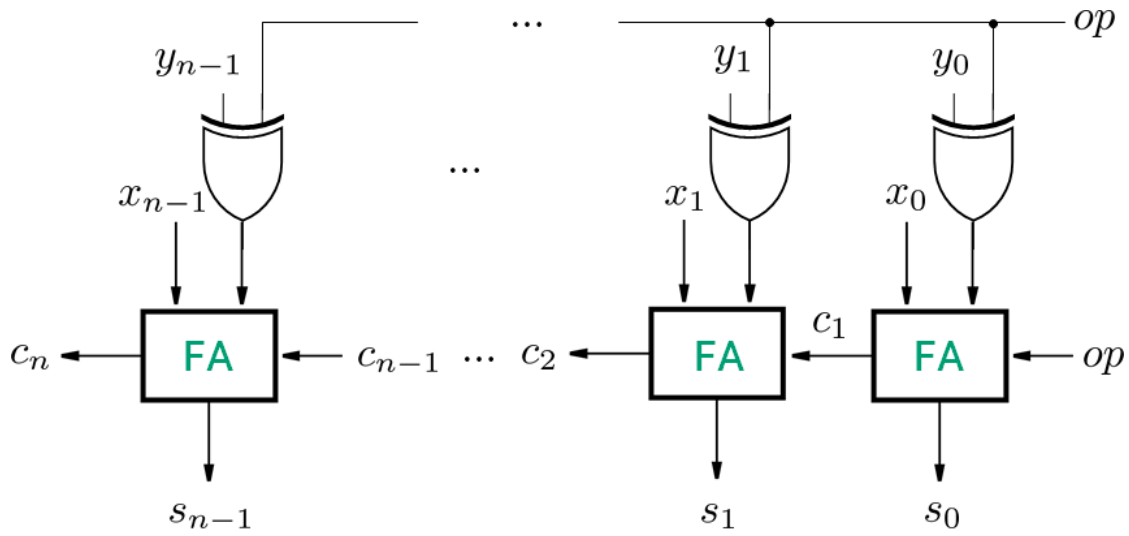
  - If all gates had equal delays:

$$t_{\max} = t(c_{i+1}) = 3 \text{ Gate Delays}$$



Basic Ripple-Carry Adder-Subtractor

- What is the **worst-case delay** to find the sum/difference using a basic ripple carry adder-subtractor?
  - Assume the inputs X, Y, and op are available (no waiting to start)
  - Assume all gates have the same delay
- Note: Worst-case delay is commonly referred to as **critical path delay (CPD)**
- **A:** $(2n+1)$ gate delays

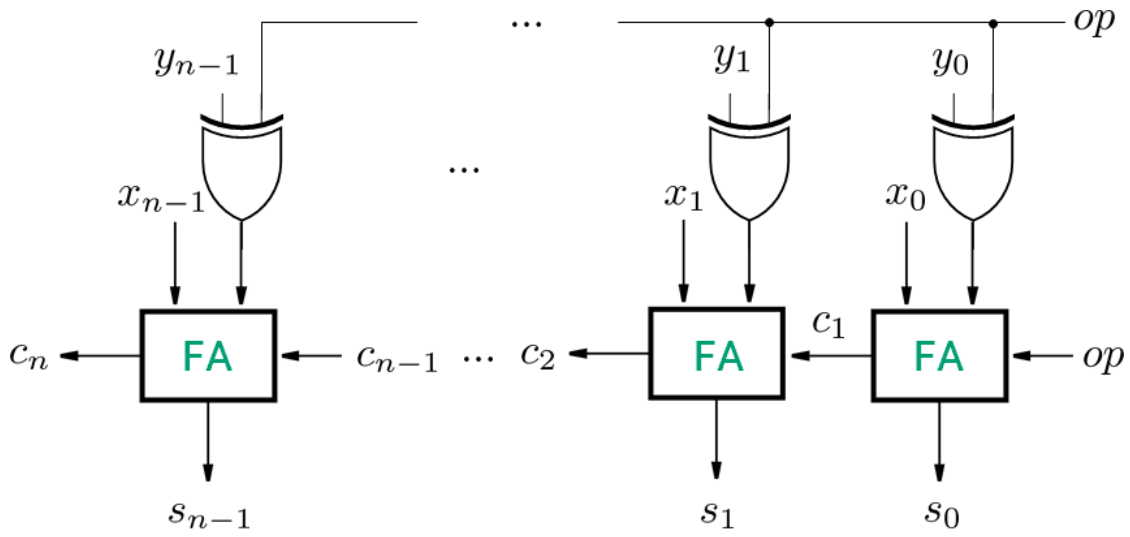


Basic Ripple-Carry Adder/Subtractor

- **Solution:**
  - The worst-case delay is on the path from the op input to the last carry-out output

$$
\begin{aligned}
t_{\max} &= t_{\max}(c_1) \\
&\quad + (n-2)t(c_i, c_{i+1}) \\
&\quad + t_{\max}(t(c_{n-1}, c_n), t(c_{n-1}, s_{n-1}))
\end{aligned}
$$

$$
\begin{aligned}
t_{\max} &= t(\text{XOR}) + t(\text{AND}) + t(\text{OR}) \\
&\quad + (n-2)\,(t(\text{AND}) + t(\text{OR})) \\
&\quad + t(\text{AND}) + t(\text{OR})
\end{aligned}
$$



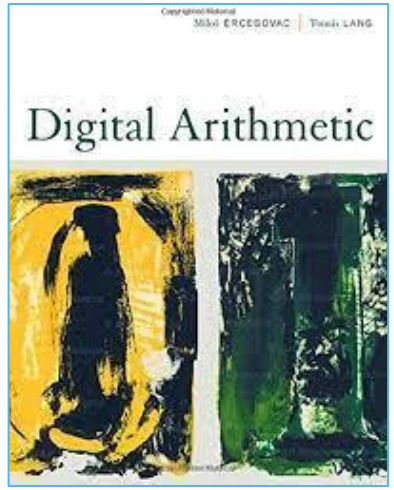
If all gates have equal delays:

$$
\begin{aligned}
t_{\max} &= 3 + (n-2) \cdot 2 + 2 \\
&= (2n+1)\ \text{Gate Delays}
\end{aligned}
$$

RCA Performance Issues

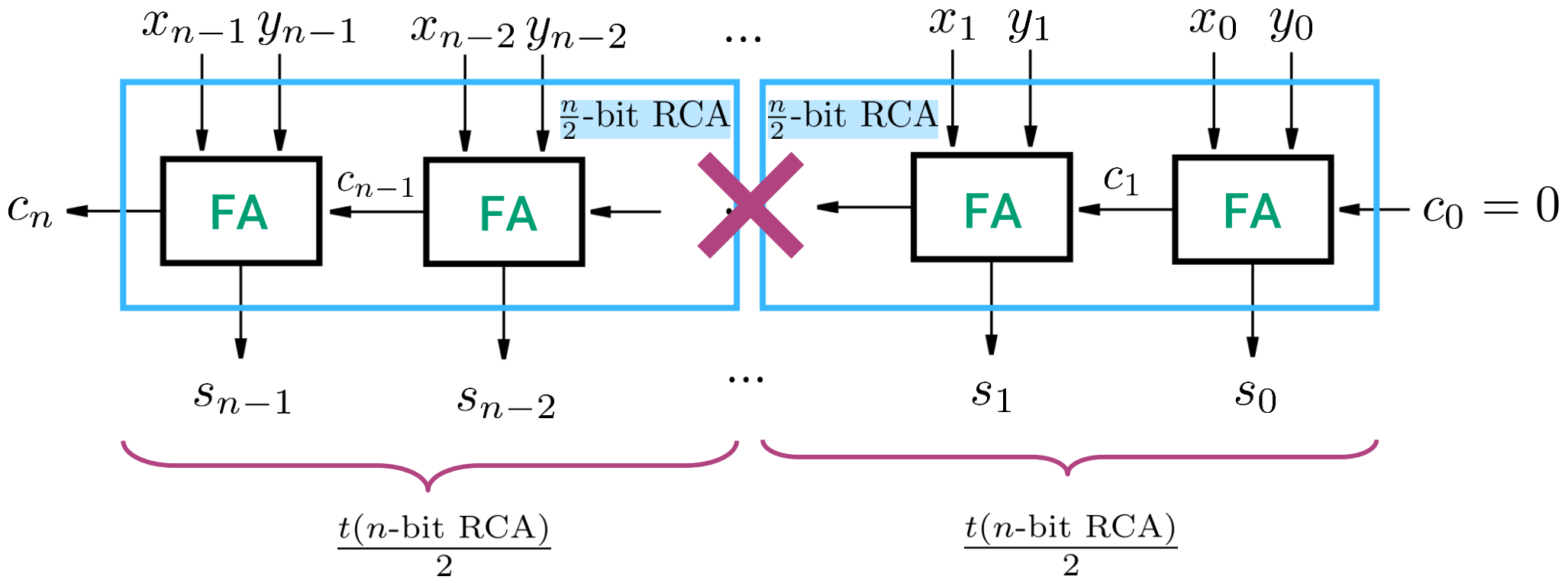
- With the increasing number of bits n, the ripple-carry adder delay increases, and the computation becomes prohibitively slow
- What can be done to make the adder faster?

- A variety of faster implementations exist
  - Switched carry-ripple adder, carry-skip adder, carry-lookahead adder, prefix-adder, conditional-sum adders, …
  - All but the carry-select adder are **out of scope for CS-173**

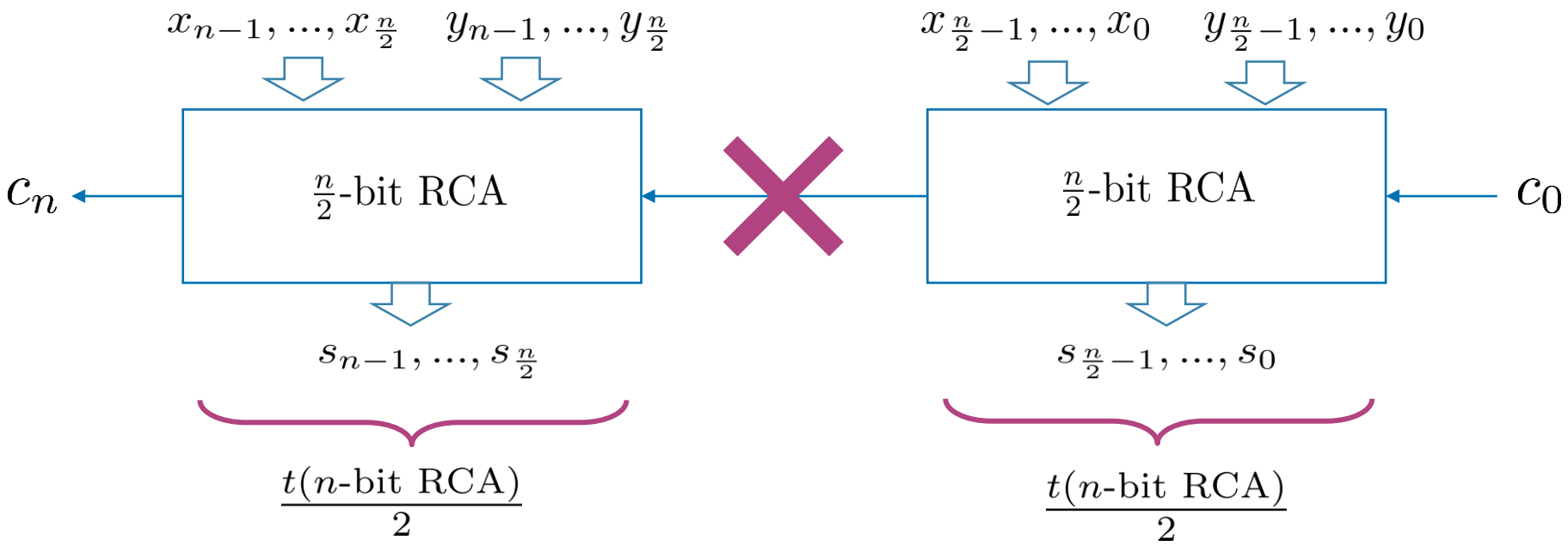


Carry-Select Adder

- **Idea:** Cut the long carry propagation chain in half to save time



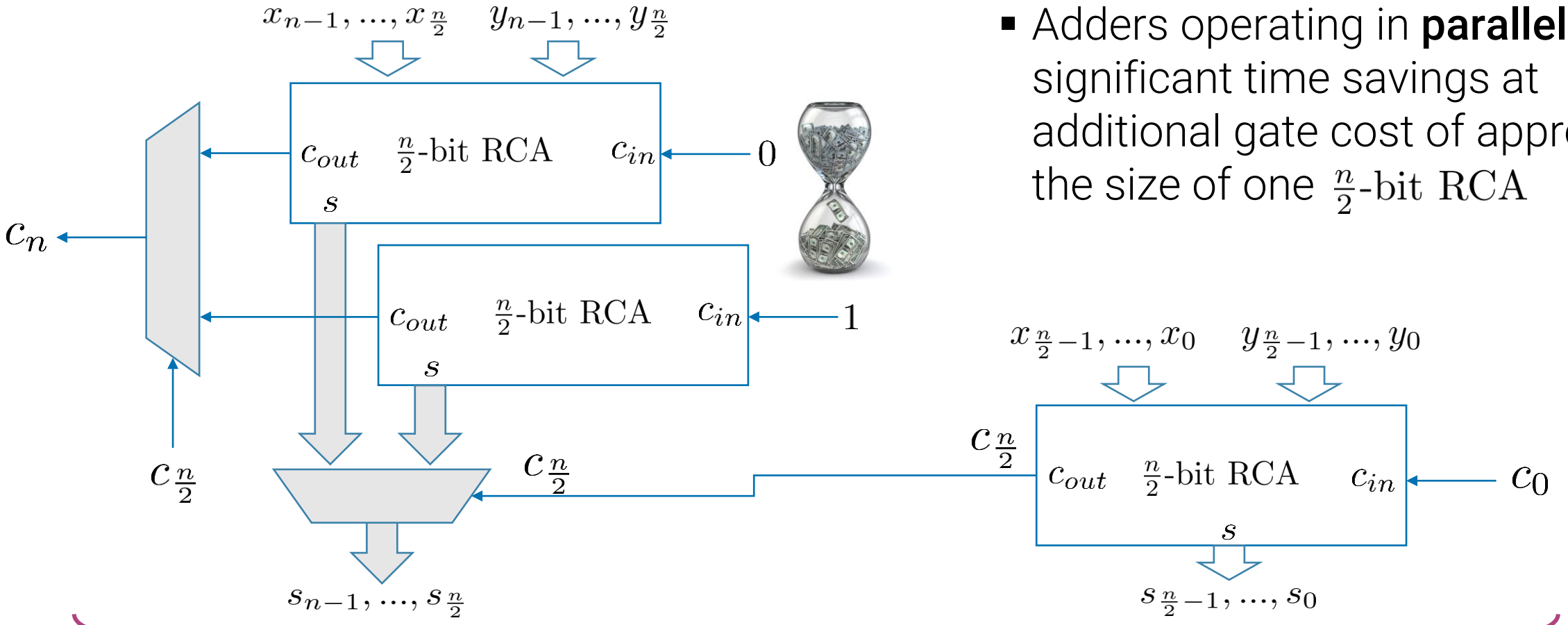
Carry-Select Adder, Contd.



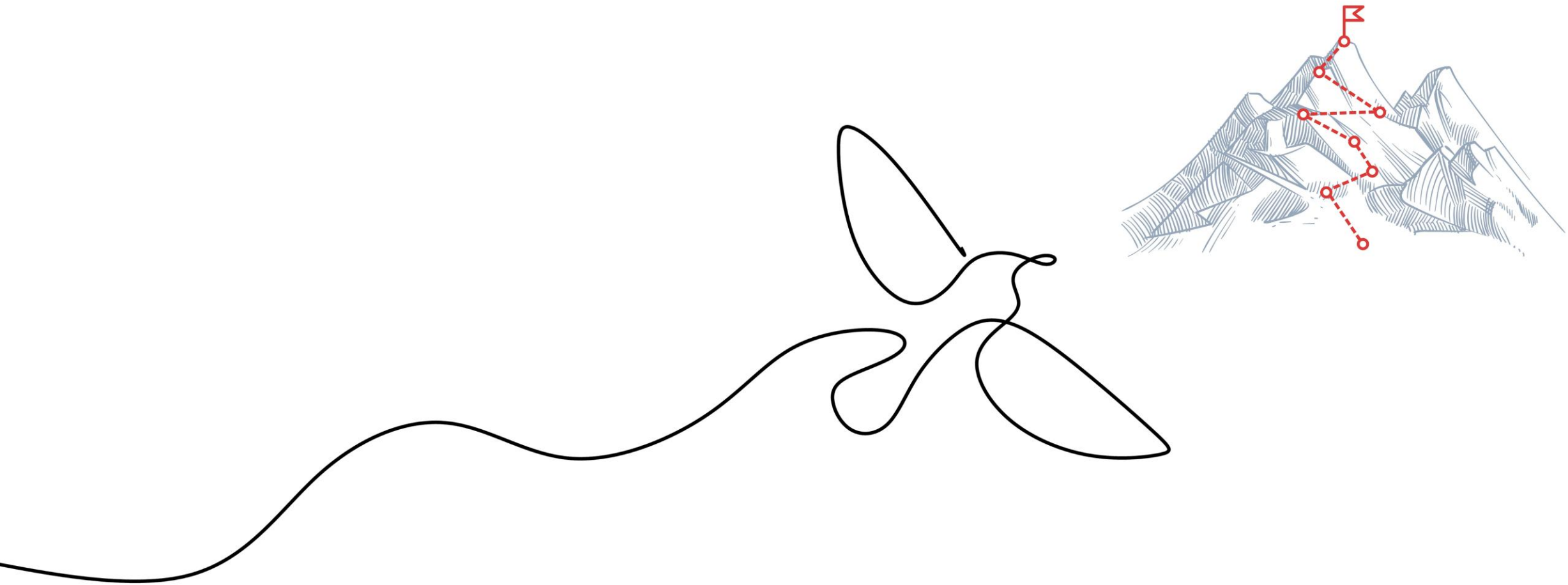
- How can we compute the carry-in for the second half?
  - Compute **twice**: once for carry-in = 0, once for carry-in = 1
  - Compute **in parallel (double the gates) to save time**
  - Once carry-in is known, use it to select the corresponding sum and carry-out

Carry-Select Adder

Block Diagram



- Adders operating in **parallel**: significant time savings at additional gate cost of approx. the size of one $\frac{n}{2}$-bit RCA



Shifting

Barrel Shifters

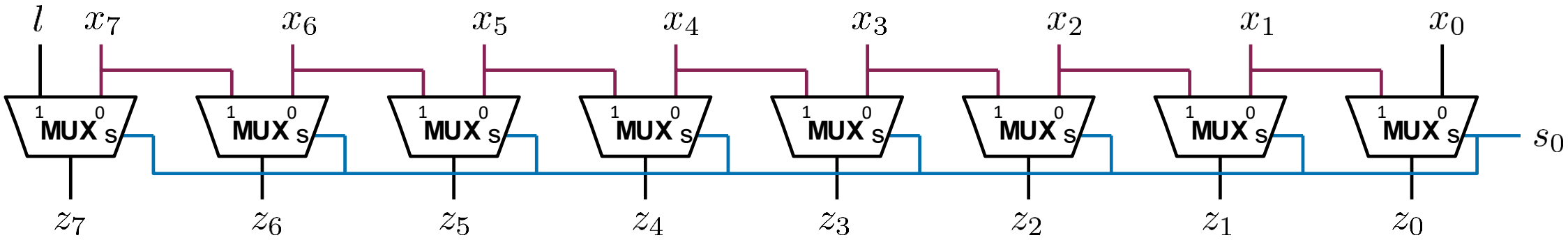
Barrel Shifter

- A **barrel shifter** is a combinational logic circuit with n data inputs, n data outputs, and a set of control inputs that specify how to shift the data between the input and the output
- A barrel shifter inside a processor can typically specify
  - **direction** of shift (left, right)
  - **type** of shift (logical, arithmetic, circular/rotation)
  - **amount** of shift (typically 0 to $n - 1$ bits)
- Implemented as a **sequence of multiplexers** (MUX), each shifting their input by twice as many positions as the previous MUX

Shift Right

By Up to One Position

- **Logical** shift resets the leading bit of the output: $l = 0$
- **Arithmetic** (sign-preserving) shift: $l = x_7 = \text{Most-significant bit}$

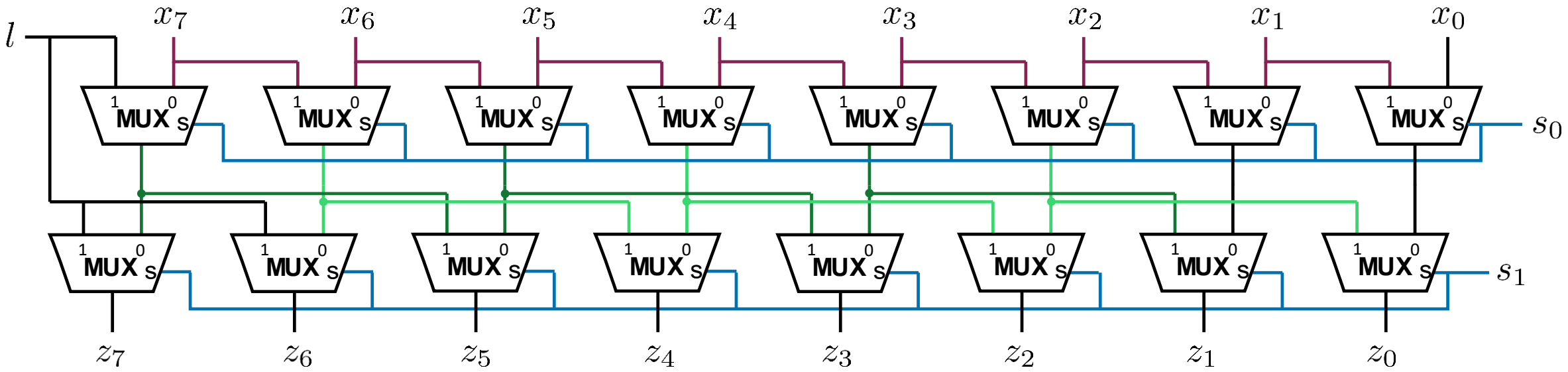


- Truth table

s_0	$z_7 z_6 z_5 z_4 z_3 z_2 z_1 z_0$
0	$x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0$
1	$l x_7 x_6 x_5 x_4 x_3 x_2 x_1$

Shift Right

By Up to Three Positions



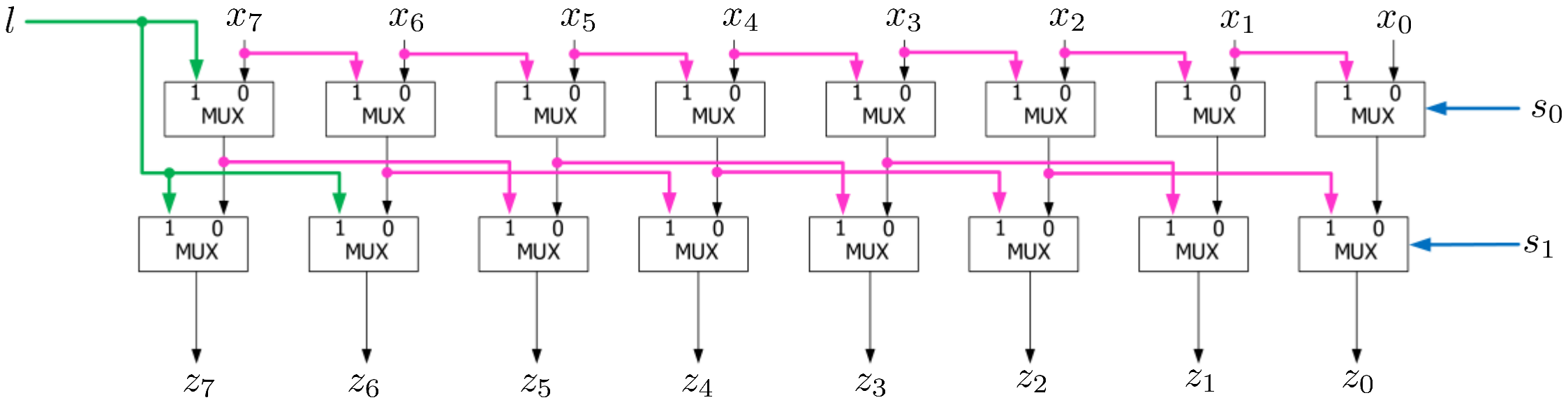
- Truth table
  - Select signals encode the number of positions to shift by (e.g., $s_1s_0 = (10)_2$ shifts by two places)

s_1	s_0	$z_7z_6z_5z_4z_3z_2z_1z_0$
0	0	$x_7x_6x_5x_4x_3x_2x_1x_0$
0	1	$lx_7x_6x_5x_4x_3x_2x_1$
1	0	$llx_7x_6x_5x_4x_3x_2$
1	1	$lllx_7x_6x_5x_4x_3$

Shift Right

By Up to Three Positions, Contd.

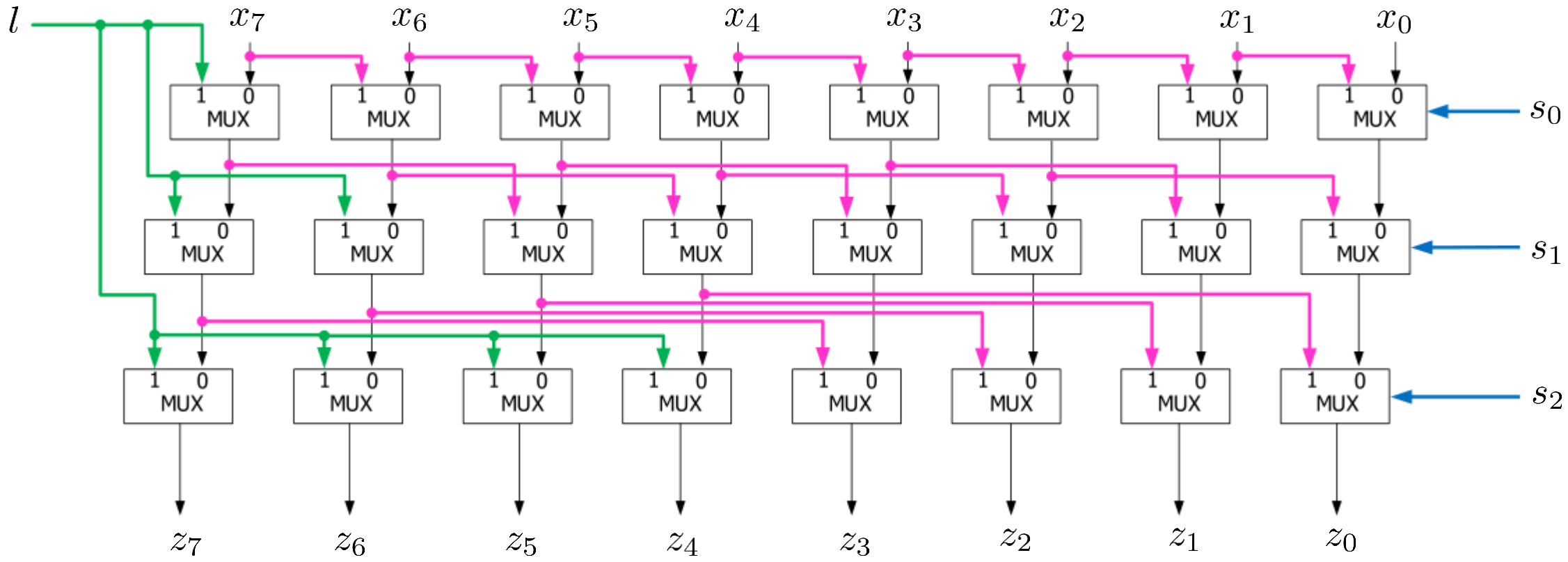
- Nothing new, just a somewhat more compact drawing



Shift Right

By Up to Seven Positions

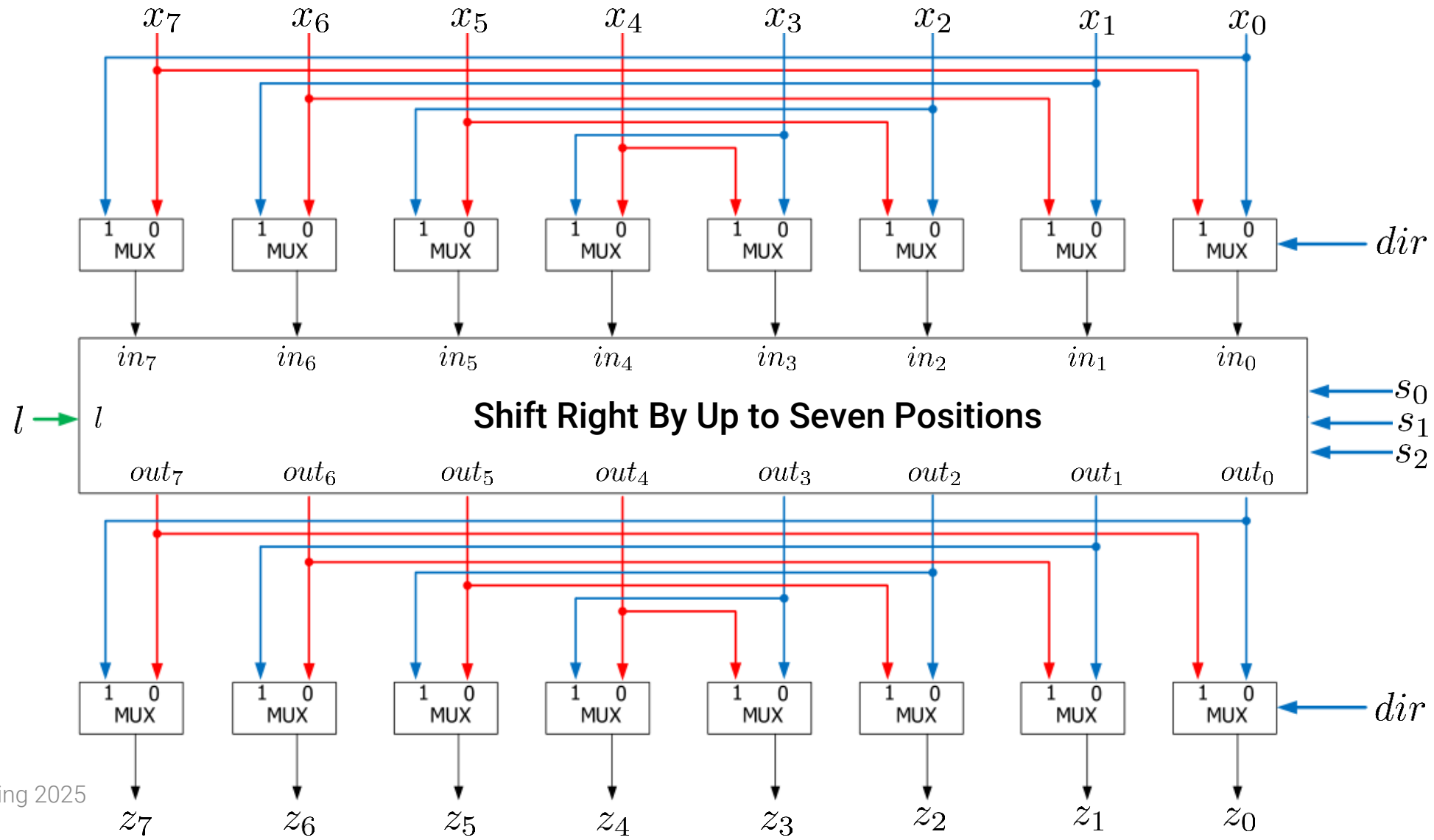
- Three levels of multiplexing



Bidirectional Shifter

By Up to Seven Positions

$dir = 0$: Right
$dir = 1$: Left
$l = 0$: Logical shift right or shift left
$l = x_7$: Arithmetic shift right



Bidirectional Shifter

By Up to Seven Positions, Contd.

$dir = 0$: Right
$dir = 1$: Left
$l = 0$: Logical shift right or shift left
$l = x_7$: Arithmetic shift right

- Truth table (incomplete, only a few example cases are shown):

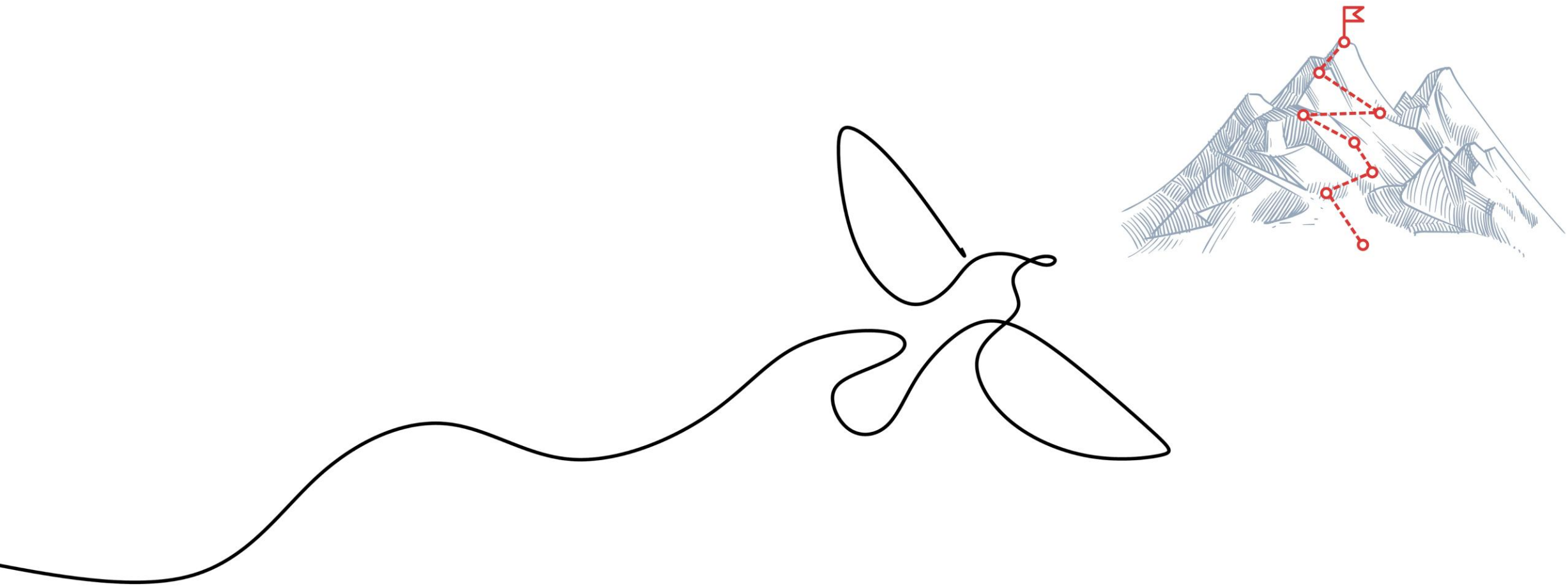
dir	s_2	s_1	s_0	in	out	z
0	0	1	0	$x_7x_6x_5x_4x_3x_2x_1x_0$	$llx_7x_6x_5x_4x_3x_2$	$llx_7x_6x_5x_4x_3x_2$
0	1	0	1	$x_7x_6x_5x_4x_3x_2x_1x_0$	$lllllx_7x_6x_5$	$lllllx_7x_6x_5$
0	1	1	1	$x_7x_6x_5x_4x_3x_2x_1x_0$	$lllllllx_7$	$lllllllx_7$
1	1	0	0	$x_0x_1x_2x_3x_4x_5x_6x_7$	$llllx_0x_1x_2x_3$	$x_3x_2x_1x_0llll$
1	0	1	1	$x_0x_1x_2x_3x_4x_5x_6x_7$	$lllx_0x_1x_2x_3x_4$	$x_4x_3x_2x_1x_0lll$
1	1	1	1	$x_0x_1x_2x_3x_4x_5x_6x_7$	$lllllllx_0$	$x_0lllllll$

Original order

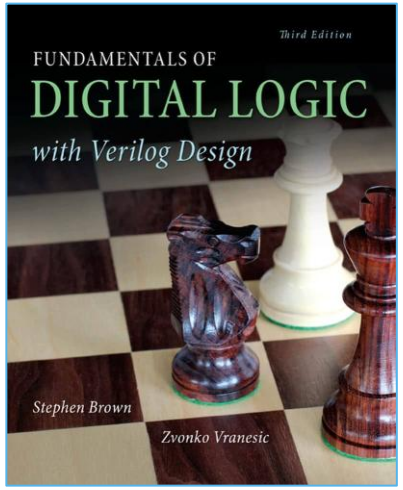
Original order

Swapped order

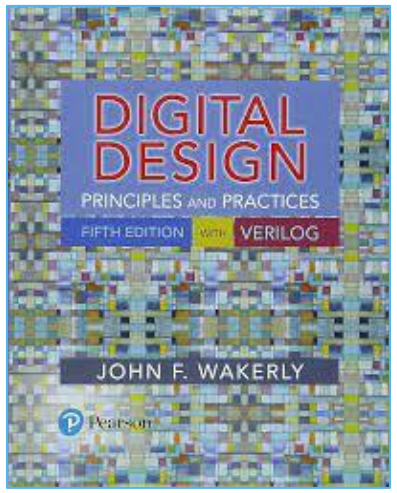
Swapped order



Literature



- Chapter 3: Number representation and arithmetic circuits
  - 3.2.1, 3.2.2, 3.3.3, 3.3.6



- Chapter 8: Combinational arithmetic elements
  - 8.1.1-8.1.3
  - 8.2