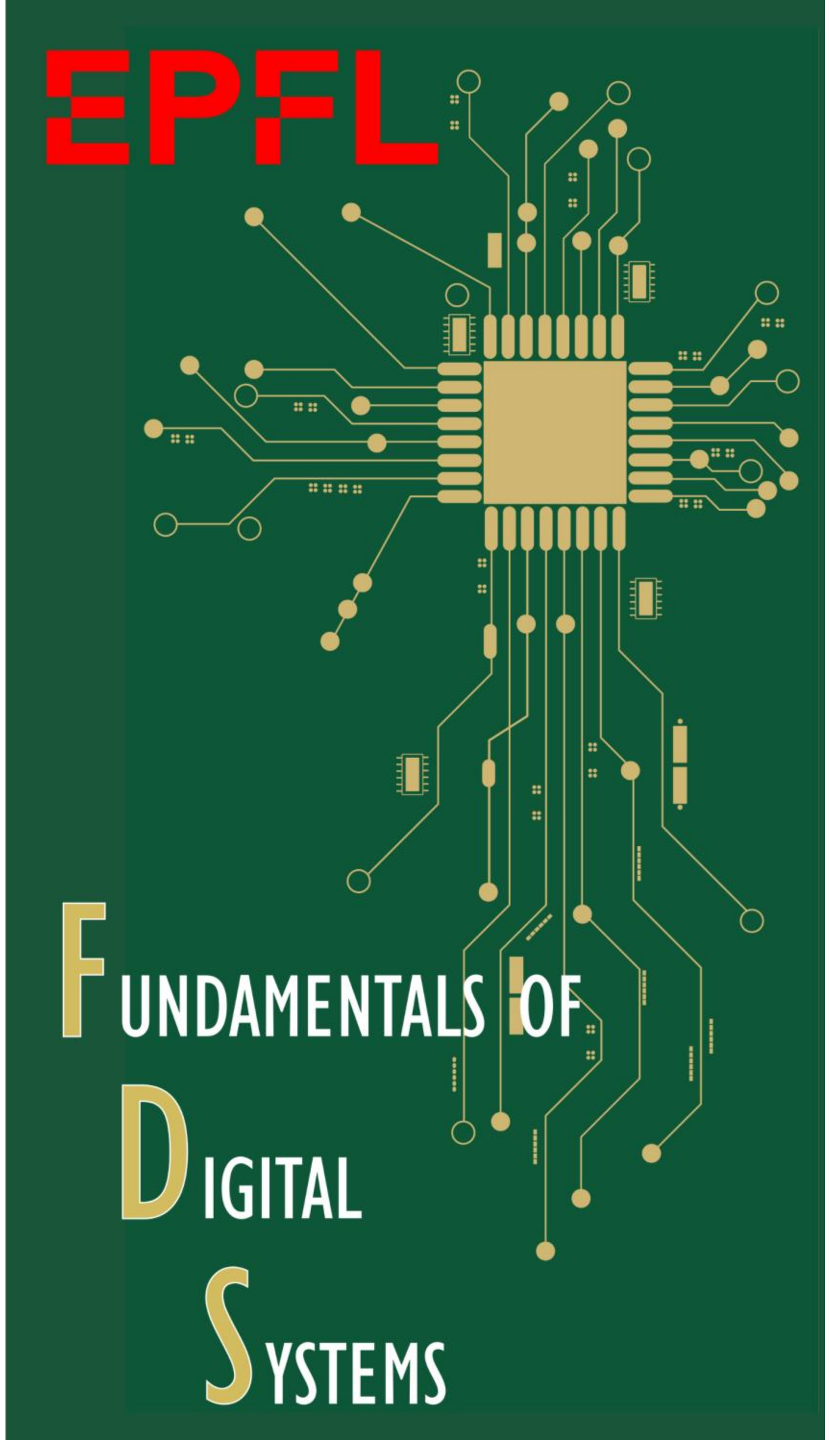


Digital Logic Circuits

Intro to CAD and Verilog

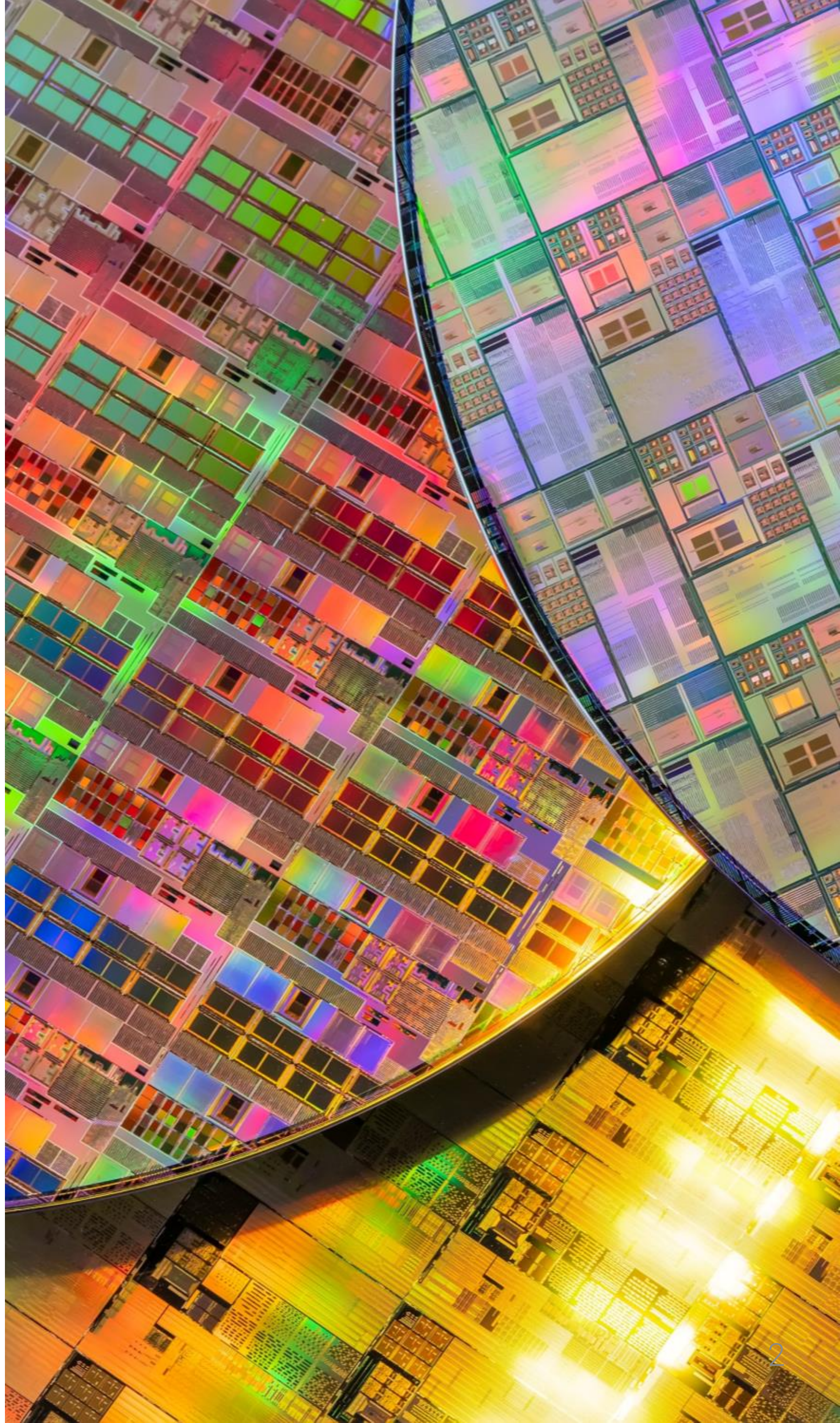
CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

March 2024

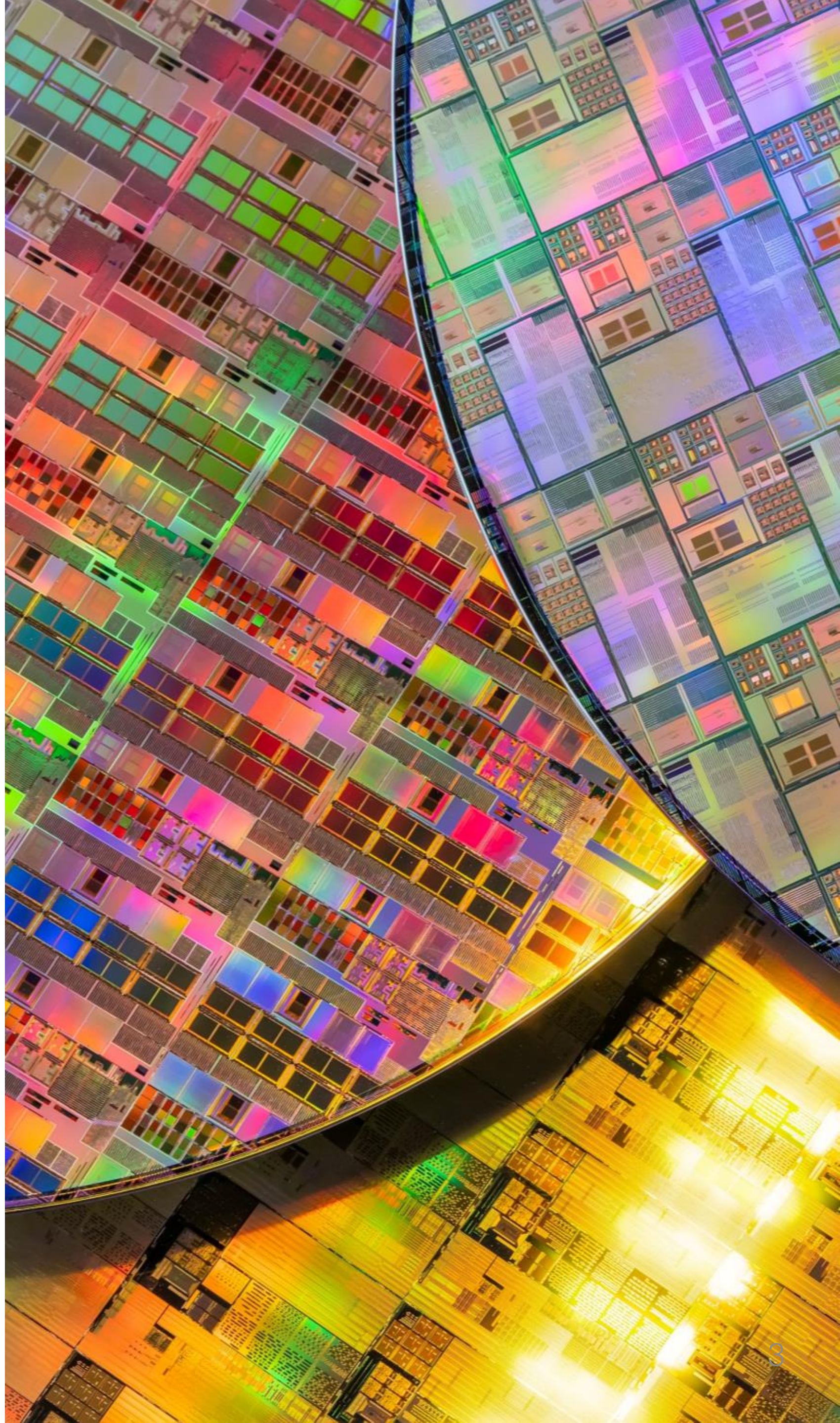
Previously on FDS

Transistors, CMOS logic gates,
real gate behavior, dynamic operation,
dynamic power dissipation



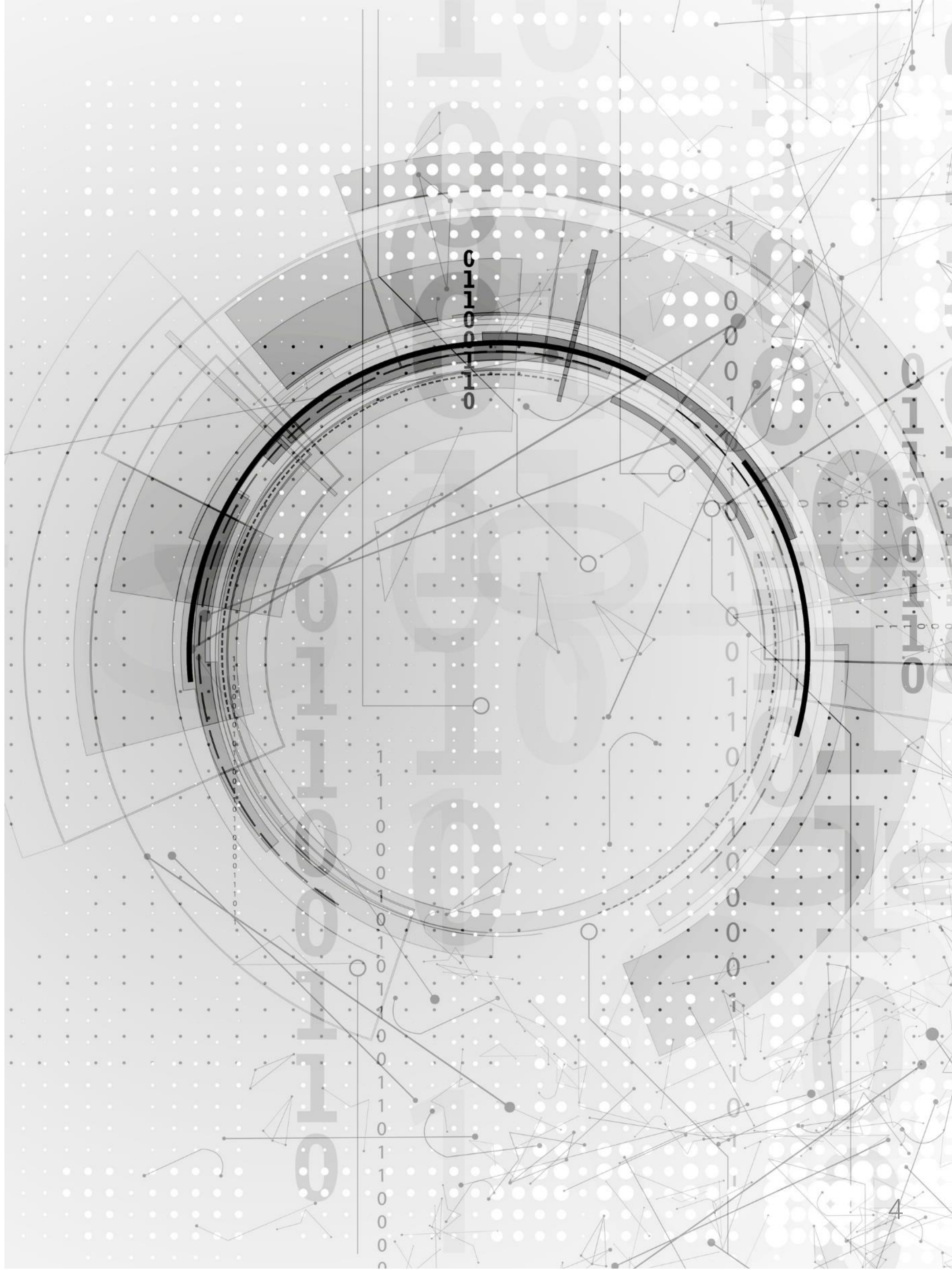
Previously

- Discovered NMOS and PMOS transistors from which real logic gates are built
  - CMOS examples (NOT, NOR, NAND, AND)
- Ideal vs. real gates
  - On-resistance and gate capacitance
  - Rise and fall times
  - Fan-in and fan-out
  - Propagation delays
- Unexpected transitions (hazards)
- Dynamic power consumption

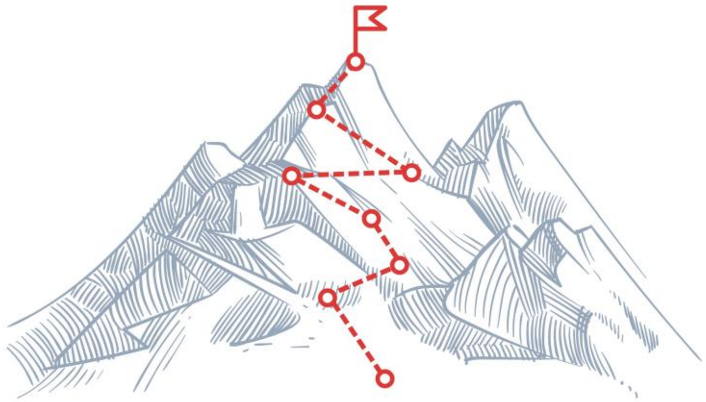


Let’s Talk About…

…Computer-Aided Design of logic circuits and Hardware Design Languages



Learning Outcomes



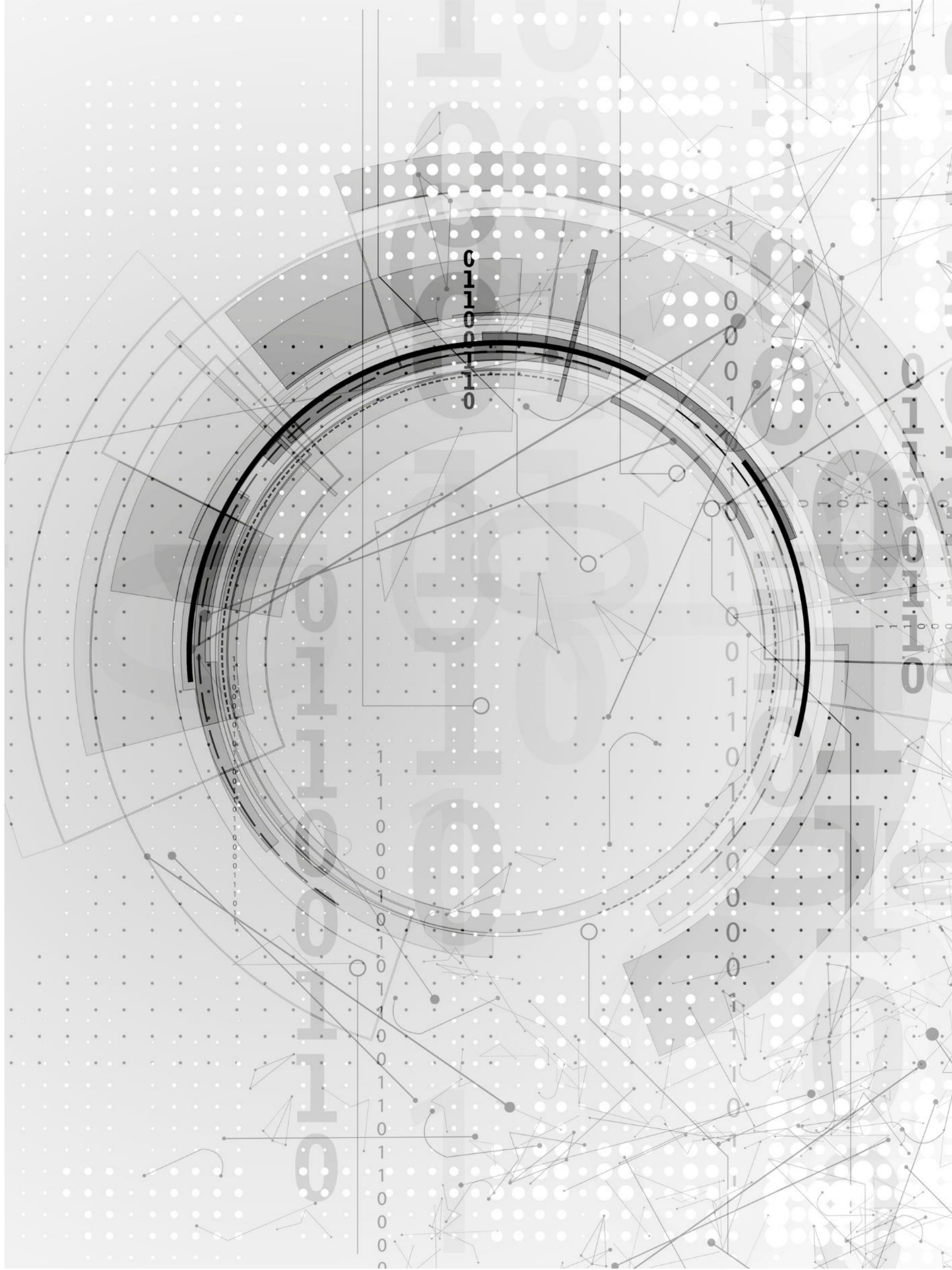
- Learn the basic steps of the computer-aided design process for building complex digital logic circuits
- Get introduced to Verilog hardware description language
- Write a piece of Verilog code that models a logic circuit described as a network of logic gates
  - Structural (i.e., gate-level) modeling in Verilog
- In Verilog, model a complex circuit by instantiating and connecting subcircuits also modeled in Verilog

Quick Outline

- Computer-Aided Design Flow
  - Design entry
    - Schematic capture
    - Hardware description language
  - Logic synthesis
  - Functional simulation
  - Physical design
  - Timing simulation
  - Circuit implementation
- Silicon wafer
- Verilog HDL
  - Brief history
  - Structural modeling
  - Names
  - Modules
  - Ports
  - Subcircuits
  - Examples:
    - Full-adder
    - Four-bit Ripple-Carry Adder

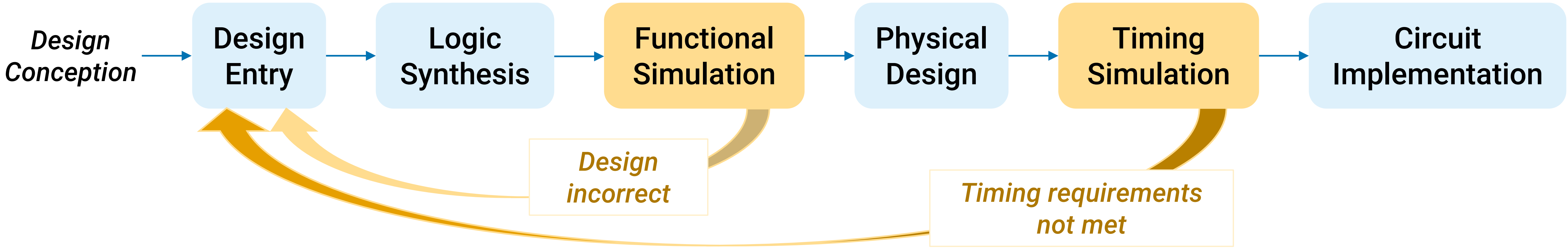
Computer-Aided Design

Introduction

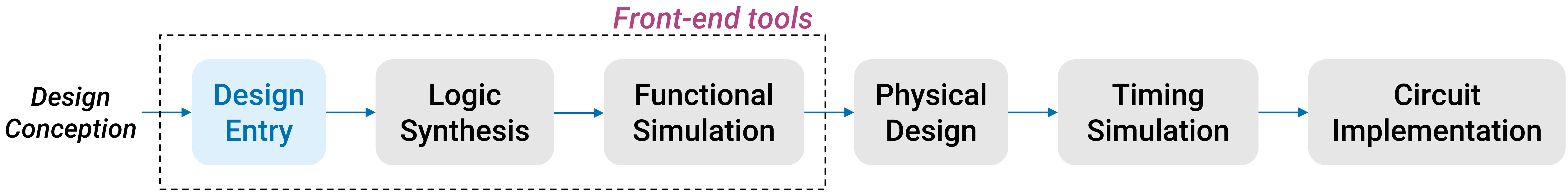


CAD Design Flow
Introduction to CAD Tools

- Logic circuits found in today’s complex computing systems cannot be designed manually
  - Designers of logic circuits heavily rely on the availability of **computer-aided design (CAD)** tools
- CAD design flow:



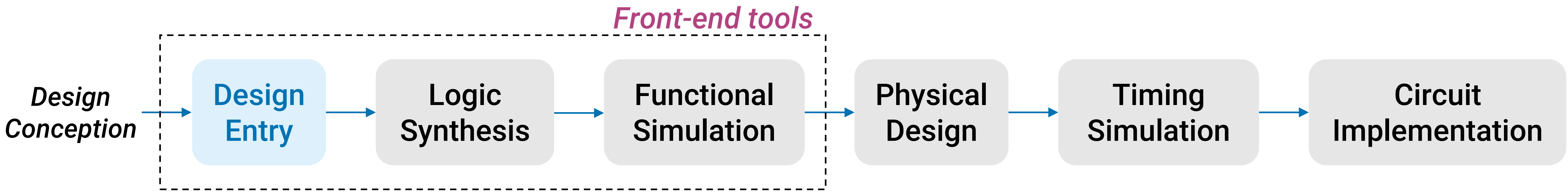
Design Entry
Introduction to CAD Tools



- **Design Entry**
  - The starting point in the process of designing a logic circuit
  - The conception of what the circuit is supposed to do and the formulation of its general organization and structure
  - Performed by the designers without the guidance of CAD tools; requires experience and intuition

Design Entry, Contd.

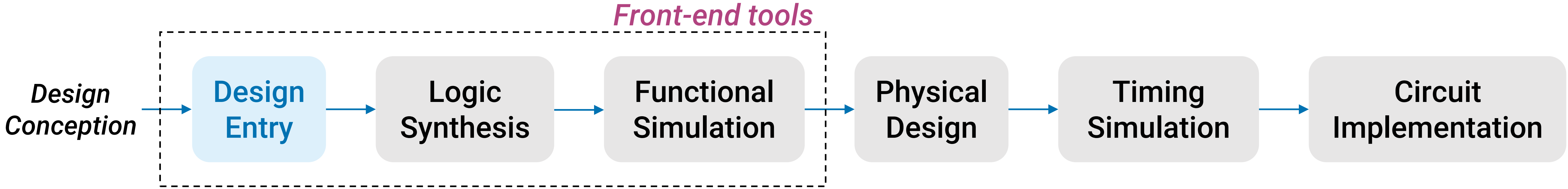
Introduction to CAD Tools



- Approach 1: **Schematic capture**
  - Drawing logic gates and interconnecting them with wires
  - Schematic tools provide libraries of gates and other circuit components
  - **Hierarchical design**
    - Subcircuits previously created can be represented as graphical symbols and included (reused) in the schematic

Design Entry, Contd.

Introduction to CAD Tools

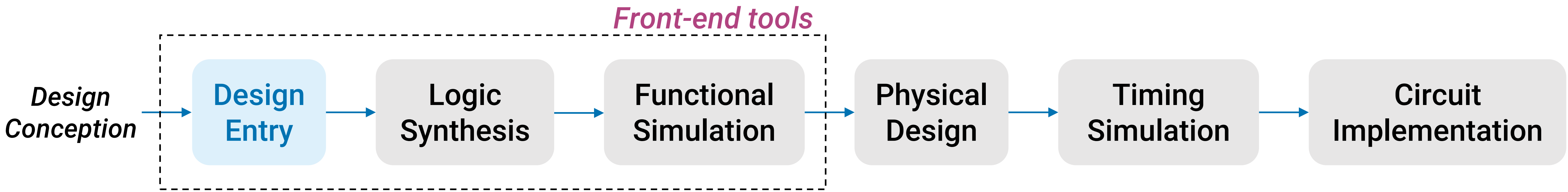


- Approach 2: **Hardware Description Language (HDL)**
  - An HDL is *similar* to a typical computer programming language except that an HDL is used to **describe hardware** rather than a program to be executed on a computer
- Mainstream HDL languages supported by vendors of digital hardware technology and officially endorsed as IEEE standards
  - **Verilog HDL (CS-173)** and VHDL[*]

[*] Very High Speed Integrated Circuit Hardware Description Language

HDL vs. Schematic Capture

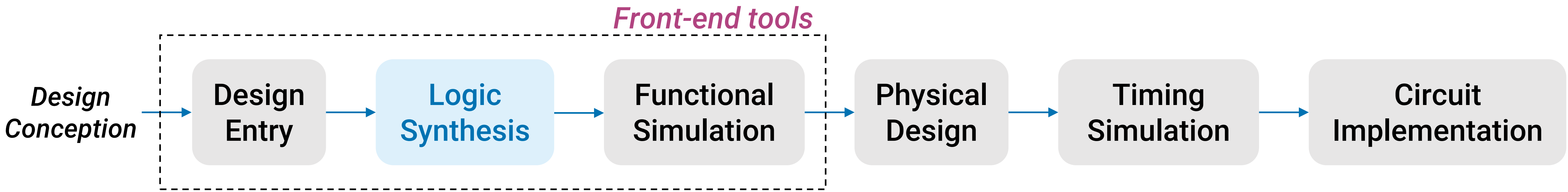
Introduction to CAD Tools



- HDL vs. Schematic Capture
  - HDLs supported by many companies: no need to change the design from one company to another) → Easy **portability**
  - Design entry means writing Verilog source code; the code is plain text, making it easy to include it in the documentation to explain its functionality → Easy **sharing** and **reuse**
  - Similar to schematic capture, HDLs support **hierarchical design**
  - HDL source can be **combined** with schematic capture (e.g., a subcircuit)

Logic Synthesis

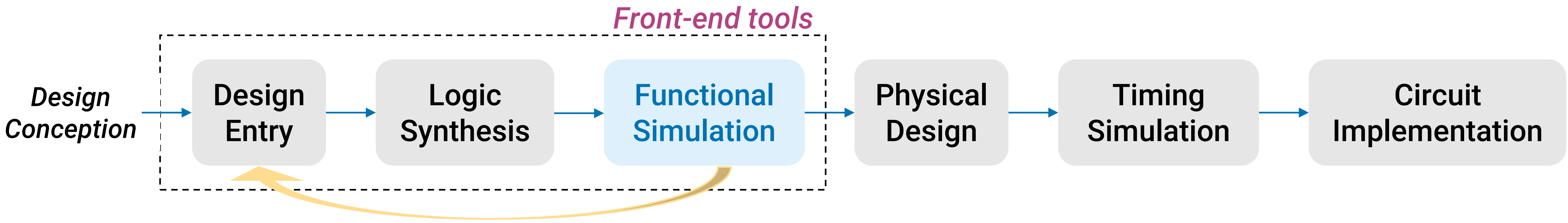
Introduction to CAD Tools



- **Logic Synthesis**
  - Translating HDL code into a network of logic gates
  - The output is the set of logic expressions describing the logic functions the circuit should realize
  - Internally manipulates logic expressions to automatically generate an equivalent but better circuit (e.g., faster, smaller, low power, etc.)

Functional Simulation

Introduction to CAD Tools



- **Functional Simulation**
  - A circuit described in the form of logic functions can be simulated to verify that it will work as expected
  - Functional simulators assume the logic functions will be implemented with **perfect gates (zero-delay model)**
  - For the sequence of **inputs specified by the designers**, the simulator evaluates the circuit outputs and produces the results (e.g., **timing waveforms**) to be analyzed by the designers

Physical Design

Introduction to CAD Tools

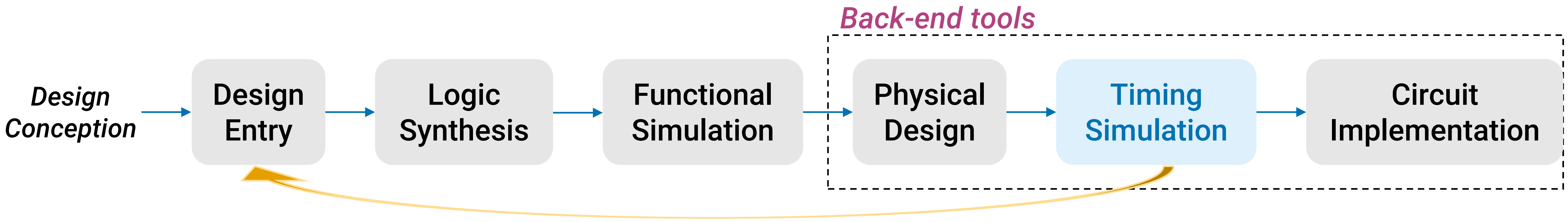
Back-end tools

Design Conception → Design Entry → Logic Synthesis → Functional Simulation → Physical Design → Timing Simulation → Circuit Implementation

- **Physical Design**
  - **Mapping** a circuit described in the form of logic expressions into a realization that uses logic gates or other hardware components available
  - **Placement:** Determine the absolute and relative location of the hardware components on the physical chip
  - **Routing:** Determine the location and shape of the wiring connections that have to be made between the inputs and outputs of the hardware components to connect them appropriately

Timing Simulation

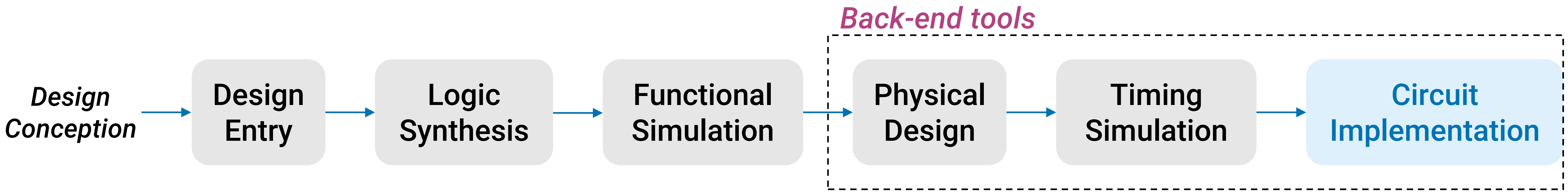
Introduction to CAD Tools



- **Timing Simulation**
  - Real circuits cannot perform their function with zero delay
  - **Logic propagation delay:** Logic elements need time to generate a valid output whenever there are changes in the value of their inputs
  - **Wire propagation delay:** signals propagating along real metal wires that connect various logic elements take some time to reach their destinations
  - Timing simulators evaluate the expected circuit delays; then, the designers check whether the circuit meets the goals (i.e., so-called **timing constraints**)

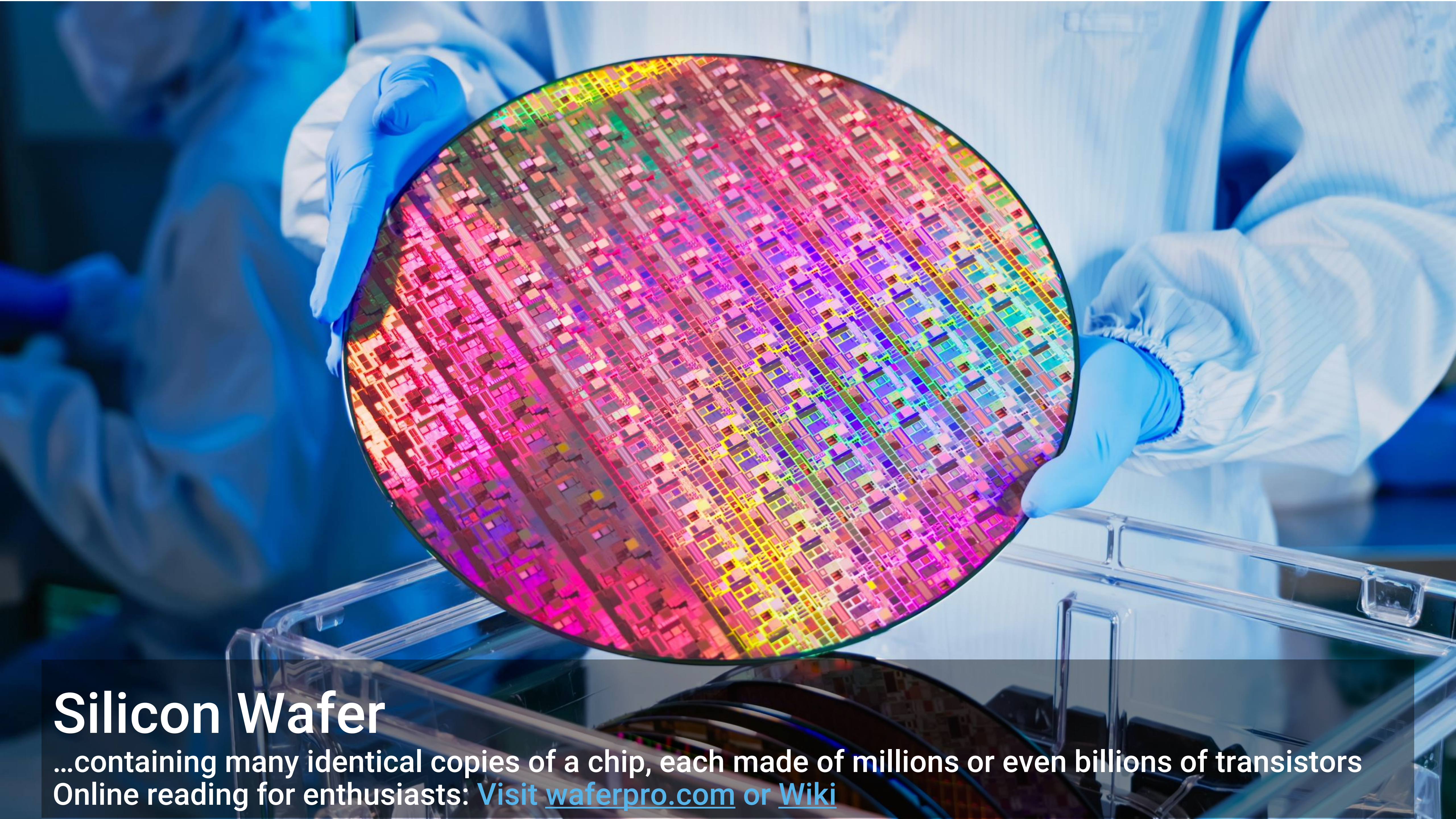
Circuit Implementation

Introduction to CAD Tools



- **Circuit Implementation**
  - Having ascertained that the circuit meets all desired requirements, the circuit is ready to be implemented on an actual chip
  - Options ahead
    - **Chip fabrication ([+] highest performance, [-] extremely expensive)** or
    - **Chip configuration ([+] flexible, [+] affordable, [-] lower performance)**:
      If a programmable hardware device* is used as a baseline, then the desired logic functionality can be implemented by simply reprogramming the device configuration

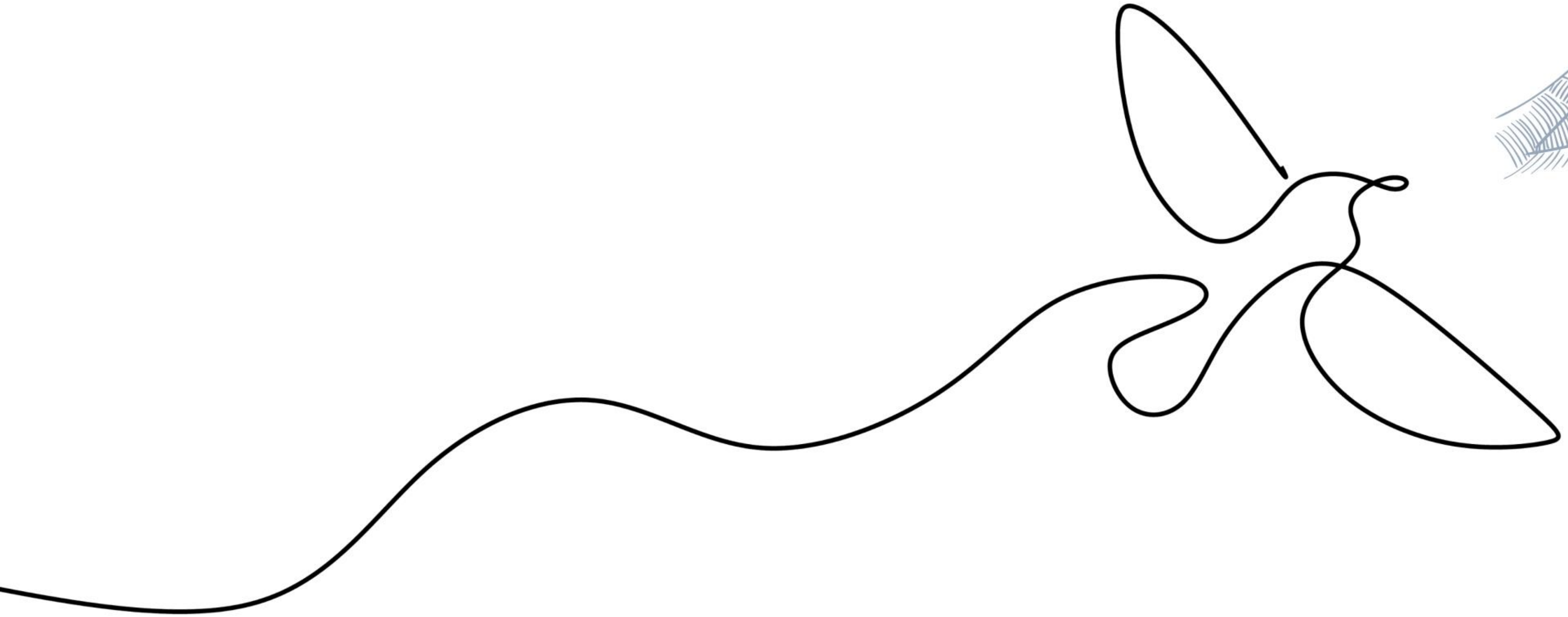
[*] Field-Programmable Gate Array (FPGA): Wiki link

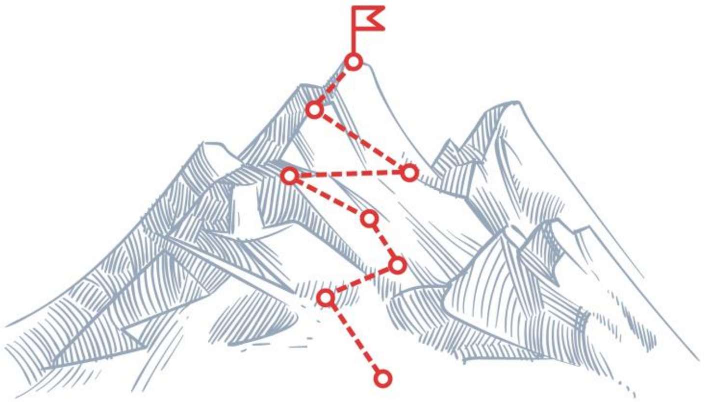


Silicon Wafer

…containing many identical copies of a chip, each made of millions or even billions of transistors

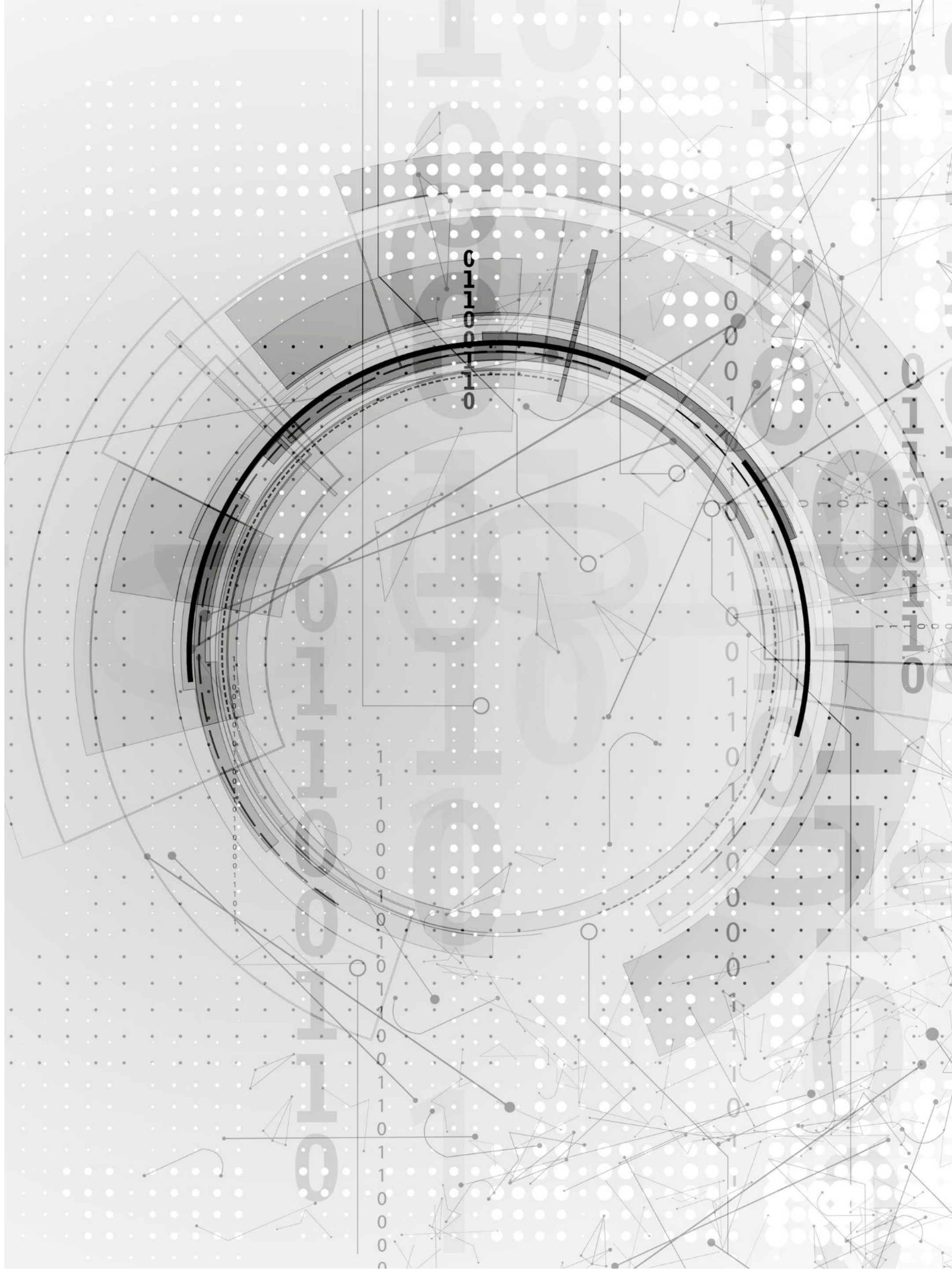
Online reading for enthusiasts: Visit waferpro.com or Wiki





Verilog HDL

- Introduction
- Structural modeling



Brief History of Verilog

- HDLs were introduced in the mid-1980s as languages for describing the behavior of a logic circuit
- Verilog was invented by Phil Moorby and Prabhu Goel ~1984
  - A proprietary language owned by Gateway Design Automation Inc.
  - Extensively modified between 1984 and 1990
  - In 1990, Gateway Design Automation Inc. was acquired by Cadence Design Systems, one of the biggest suppliers of electronic design technologies
  - Cadence recognized the value of Verilog and pushed for making it open
    - IEEE standards followed (Wiki)

Source: realdigital.org

Modeling of Digital Circuits in Verilog

- A logic circuit is specified in the form of a **module**
- **Option 1: Structural modeling**
  - **Gate-level modeling:** Using Verilog constructs to describe the **structure** of the circuit in terms of **circuit elements**, such as logic gates
  - A larger circuit is defined by writing code that **instantiates** and **connects** circuit elements
- **Option 2: Behavioral modeling**
  - Describe a circuit more abstractly, using **logic expressions** and Verilog programming constructs to describe the desired **circuit behavior**, and not its structure in terms of gates

Structural Modeling with Logic Gates

Gate-Level Modeling

- In structural modeling, predefined modules that implement basic **logic gates** are used
- Logic gate instantiation statement:

```
gate_name  [instance_name] (out_port,  in_port{, in_port});
```

- Verilog built-in gates:
  (incomplete list, for the moment)

and	**xor**
nand	**xnor**
or	**buf**
nor	**not**

Note 1: The square brackets indicate an optional field

Note 2: Braces indicate that additional entries are permitted

Structural Modeling with Logic Gates, Contd.

Gate-Level Modeling

- *Recall* logic gate instantiation statement:

Reserved word

Names can be any valid identifiers

List of ports

gate_name [instance_name](out_port, in_port{, in_port});

Instance names can be omitted

Mandatory ;

Note 1: The square brackets indicate an optional field

Note 2: Braces indicate that additional entries are permitted

- Examples
  - **or** Or1 (z, x1, x2, x3), **xor** Xor5 (c, a, b)
  - **not** (f, a), **nand** (g, d, b, w1, w3)

Verilog Syntax

In a Nutshell

- Names
  - Must start with a letter
  - Can contain any letter, number, “_” (underscore), or $
- Verilog is **case sensitive**
  - s $\neq$ S
  - Example5 $\neq$ example5
- Syntax does not enforce a particular style
  - White space characters (i.e., space, TAB) and blank lines are ignored
  - **Readability matters**: Use indentation and blank lines
  - Comments start with “//” (double slash)

Modules in Verilog

Introduction

- A circuit or subcircuit described with Verilog code is a **module**
- Module has a name, inputs, and outputs, referred to as its **ports**

```
module module_name [(port_name{, port_name})];
    [input declarations]
    [output declarations]
    [wire declarations]
    [logic gate instantiations]
    [module instantiations]
    // and many more
endmodule
```

Note 1: The square brackets indicate an optional field

Note 2: Braces indicate that additional entries are permitted

Note 3: In bold, reserved words (i.e., keywords)

Modules in Verilog

Introduction

- A circuit or subcircuit described with Verilog code is a **module**
- Module has a name, inputs, and outputs, referred to as its **ports**

Names can be any valid identifiers

List of ports

Mandatory ;

Reserved keywords

```
module module_name [(port_name{, port_name})];
   [input declarations]
   [output declarations]
   [wire declarations]
   [gate instantiations]
   [module instantiations]
   // and many more
endmodule
```

Wire (net) declarations are often omitted. Verilog assumes signals are nets by default

Anything after // is a comment and it gets ignored

Ports in Verilog

- Ports are the primary ways to communicate with the module
- Port directions
  - **input**: input-only port, receives values from outside
  - **output**: output-only port, sends values to the outside
  - **inout**: bidirectional port, receives and sends values
- Port declaration syntax:

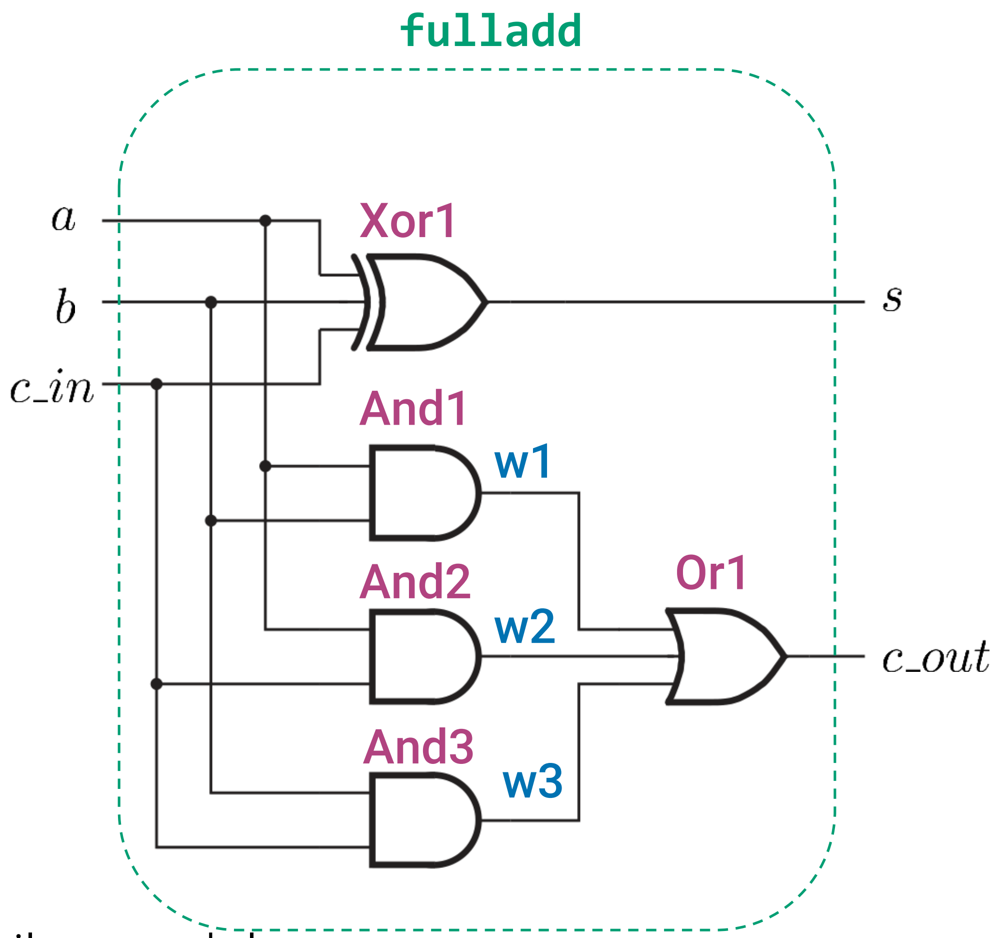
If not specified, implicitly assumes a wire or a bundle of wires (a vector)

port_direction data_type [port_size] port_name;

- Examples:
  - **input** diff; // 1-bit input named **diff**, wire type
  - **inout** [15:0] data; // 16-bit bidirectional vector named **data**, wire type
  - **output** [3:0] f; // 4-bit output named **f**, wire type

Full-Adder in Verilog
Structural (gate-level) model

- Algorithm
  - Name your circuit (i.e., label it)
    - That will be the name of your Verilog module
  - Label all inputs and outputs
    - Those will be the input and output port names of your Verilog module
  - Label all logic gates
    - Those will be the names of your gate instances
    - Same gate type can be instantiated multiple times, provided the instance name is unique
  - Label all internal nets (i.e., wires, signals)
    - Those will be the names of the wires in your Verilog module

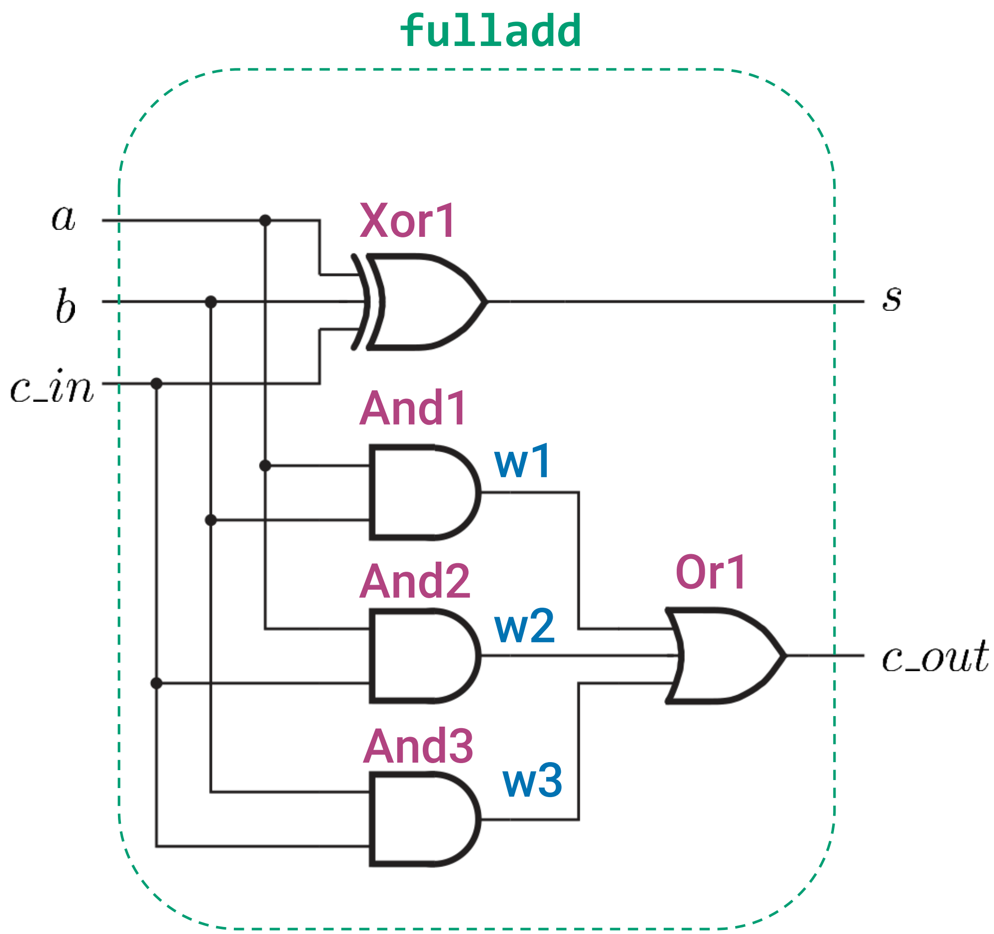


Full-Adder in Verilog

Structural (gate-level) model

EXAMPLES

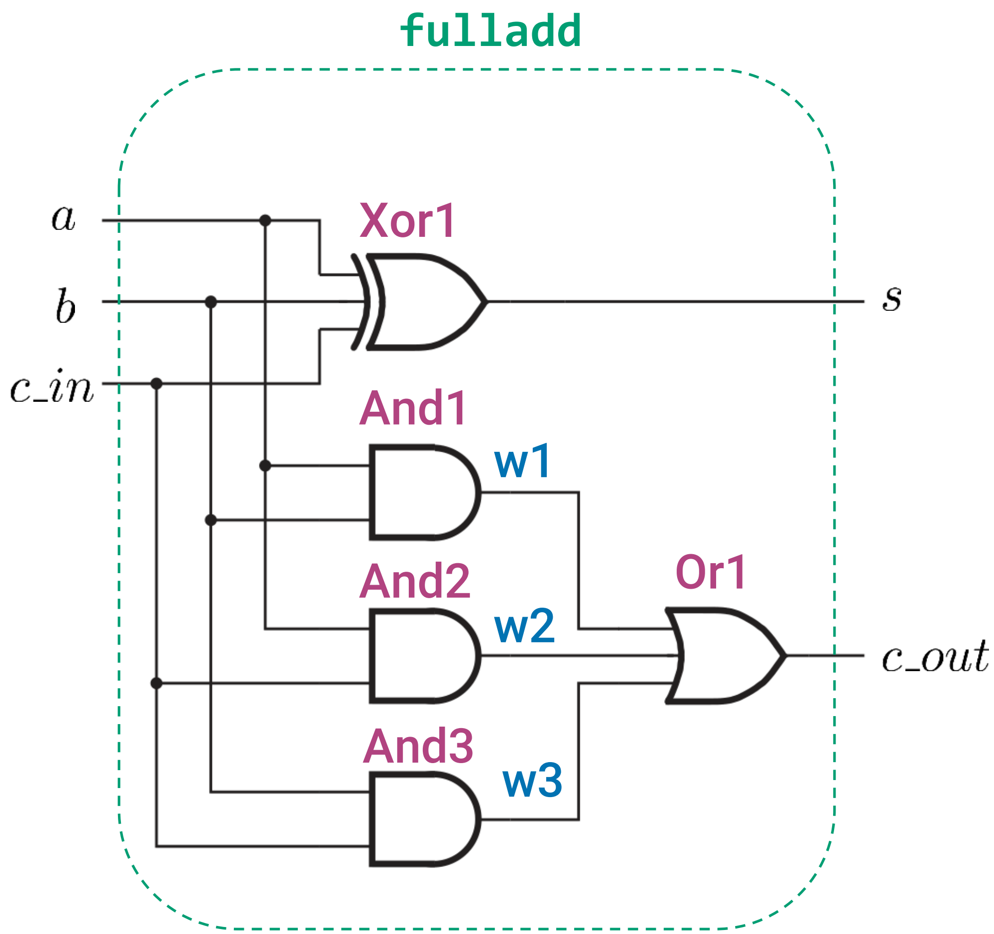
```
// Structural modeling of a full-adder

module fulladd (a, b, c_in, s, c_out);
  // ----- port definitions -----
  input  a, b, c_in;
  output s, c_out;
  // ----- intermediate signals -----
  wire   w1, w2, w3;
  // ----- design implementation -----
  and And1 (w1, a, b);
  and And2 (w2, a, c_in);
  and And3 (w3, b, c_in);
  or  Or1  (c_out, w1, w2, w3);
  xor Xor1 (s, a, b, c_in);
endmodule
```



Full-Adder in Verilog, Simplified

Structural model

```
// Structural modeling of a full-adder
// Simplified version
// - No gate instance names
// - No wire declaration (i.e., implicit)

module fulladd (a, b, c_in, s, c_out);
  input  a, b, c_in;
  output s, c_out;

  and (w1, a, b);
  and (w2, a, c_in);
  and (w3, b, c_in);
  or  (c_out, w1, w2, w3);
  xor (s, a, b, c_in);
endmodule
```



Subcircuits in Verilog

- A Verilog module can be included as a subcircuit in another module
- Modules should be defined in the same source file, in any order
  (or the Verilog compiler must be told where each module is located)
- Module instantiation statement:

```
module_name  instance_name ( .port_name  ([expression] ) { , .port_name ([expression]) } );
```

Subcircuits in Verilog

- A Verilog module can be included as a subcircuit in another module
- Modules should be defined in the same source file, in any order
  (or the Verilog compiler must be told where each module is located)
- Module instantiation statement:

Names can be any valid identifiers

Name of a port in the subcircuit

Specifies a connection to that port

Mandatory ;

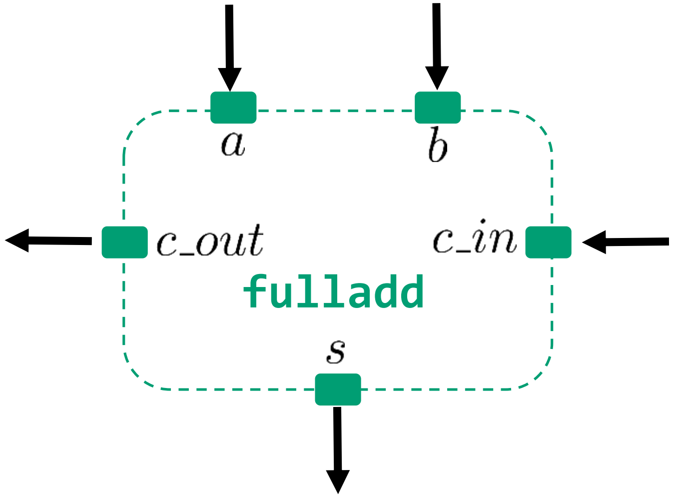
module_name instance_name (.port_name ([expression]) { , .port_name ([expression]) });

The list defines how subcircuit ports connect to the rest of the module; if the ports are listed in the same order as in the subcircuit, then .port_name can be omitted **(not advised, it is error-prone!)**

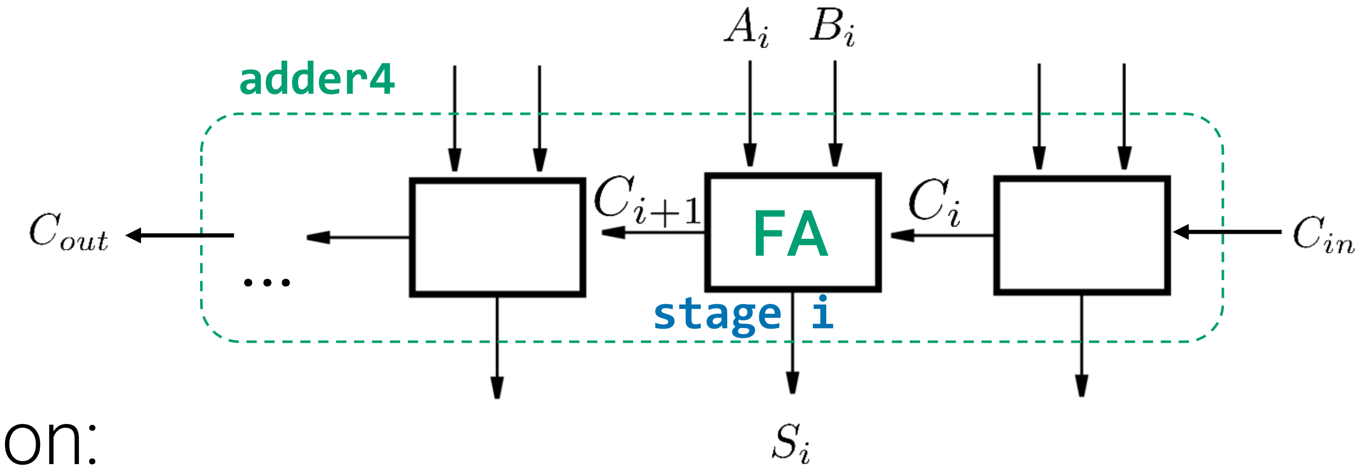
Four-bit Ripple-Carry Adder

Verilog

- *Recall* the names of the ports of the subcircuit and their direction (input, output)



- Plan the structure of the bigger module; give names to the ports and decide how to connect them to the subcircuits

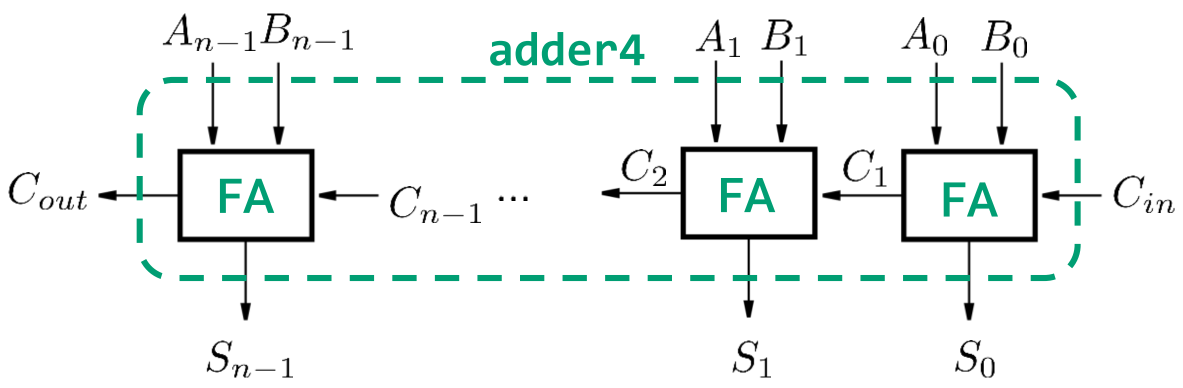


- Example full-adder instantiation:

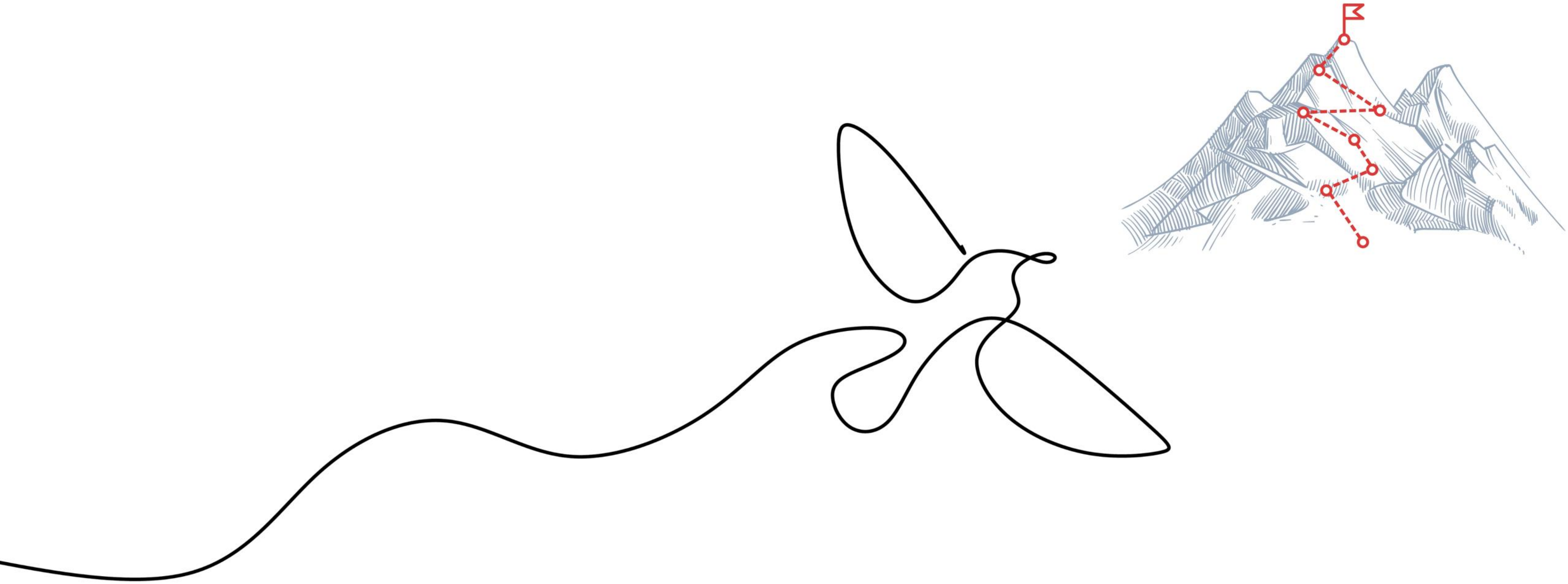
```
fulladd stage2 (.c_in(C[2]), .a(A[2]), .b(B[2]), .s(S[2]), .c_out(C[3]));
```

Port b of fulladd connects to port B[2] of the larger module

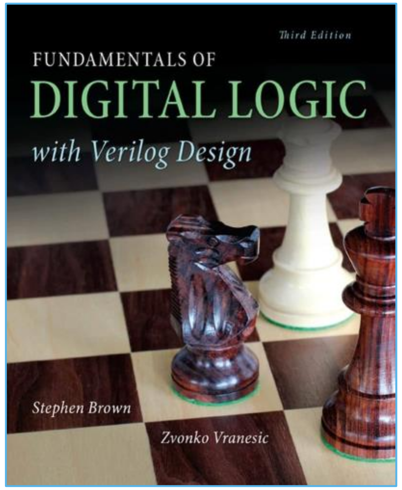
Four-bit Ripple-Carry Adder
Verilog



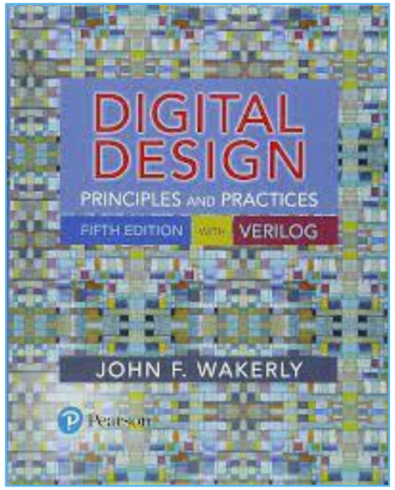

```
module adder4 (Cin, A, B, S, Cout);
  // ----- port definitions -----
  input  Cin;
  input  [3:0] A, B; // 4-bit vectors
  output [3:0] S;    // 4-bit vector
  output Cout;

  // ----- intermediate signals -----
  wire   [3:1] C;    // 3-bit vector

  // ----- design implementation -----
  fulladd stage0 (.c_in(Cin),  .a(A[0]), .b(B[0]), .s(S[0]), .c_out(C[1]));
  fulladd stage1 (.c_in(C[1]), .a(A[1]), .b(B[1]), .s(S[1]), .c_out(C[2]));
  fulladd stage2 (.c_in(C[2]), .a(A[2]), .b(B[2]), .s(S[2]), .c_out(C[3]));
  fulladd stage3 (.c_in(C[3]), .a(A[3]), .b(B[3]), .s(S[3]), .c_out(Cout));
endmodule
```



Literature



- Chapter 2: Introduction to Logic Circuits
  - 2.9, 2.10.1



- Chapter 5: Verilog Hardware Description Language
  - 5.7