CPE 462
VHDL: Simulation and Synthesis

Topic #03 - a) Introduction to VHDL

About VHDL

- VHDL is a hardware description language (HDL)
- Describes the behavior of an electronic circuit or system, from which the physical circuit or system can then be attained (implemented)
- VHDL stands for VHSIC Hardware Description Language
- ... where VHSIC is Very High Speed Integrated Circuits
- First version was released in 87, later upgraded on 93
- Became the first HDL to be standardized by IEEE (IEEE 1076)

What does VHDL do?

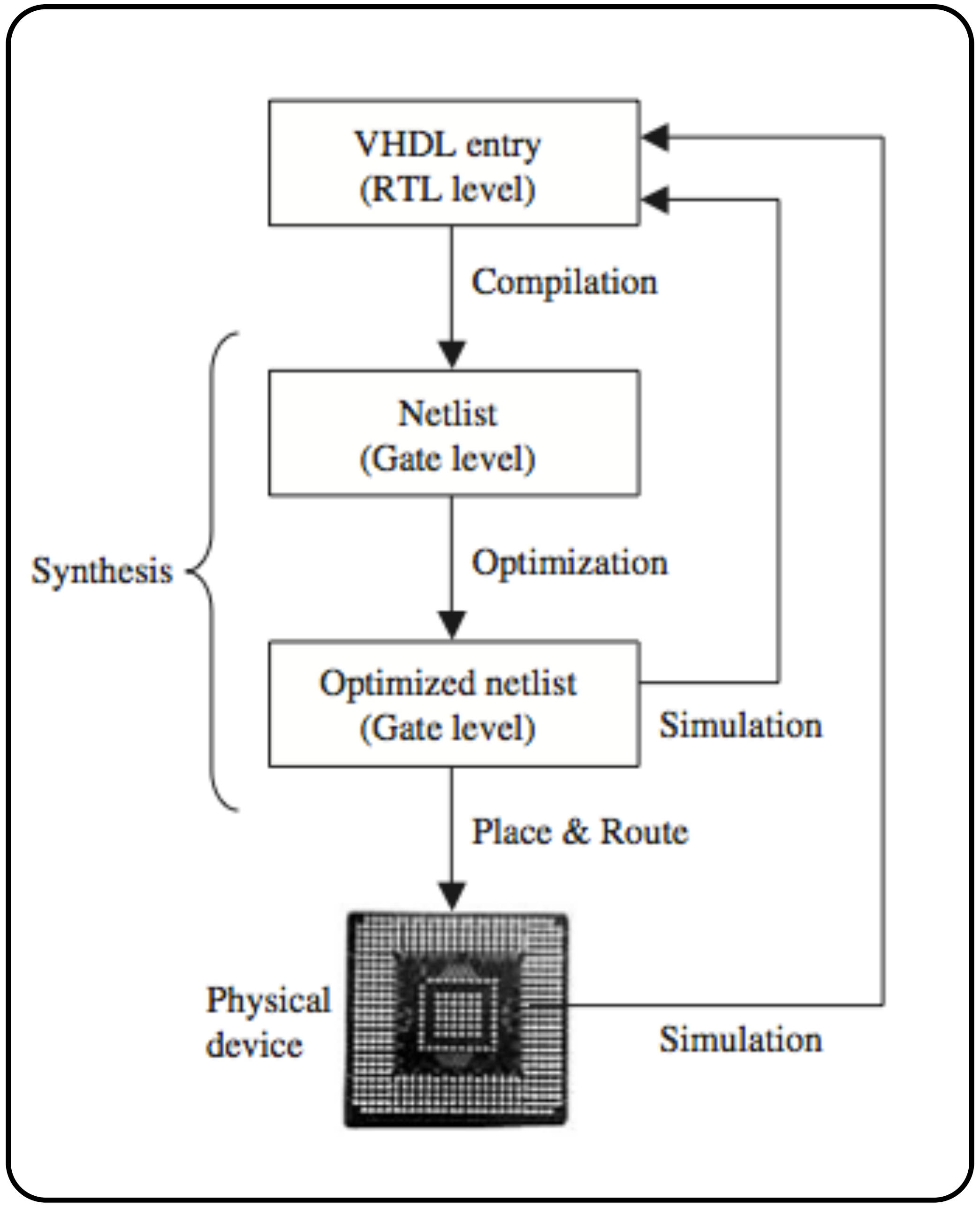
- **Synthesis** is process of converting a higher-level form of a design into a lower-level implementation
  - In software: converting C++ into ASM
  - In hardware: converting some high level language into transistors
- VHDL is intended for circuit synthesis as well as circuit simulation
- However, though VHDL is fully simulatable, not all constructs are synthesizable

Motivation

- VHDL (or its competitor Verilog) is technology/vendor independent language
- Portable and reusable
- Once you have VHDL you can:
  - Either to implement the circuit in a programmable device (e.g. FPGA)
    - Or it can be submitted to a foundry for fabrication of an ASIC chip
- All statements in VHDL are parallel
- All statements in regular computer programs are sequential

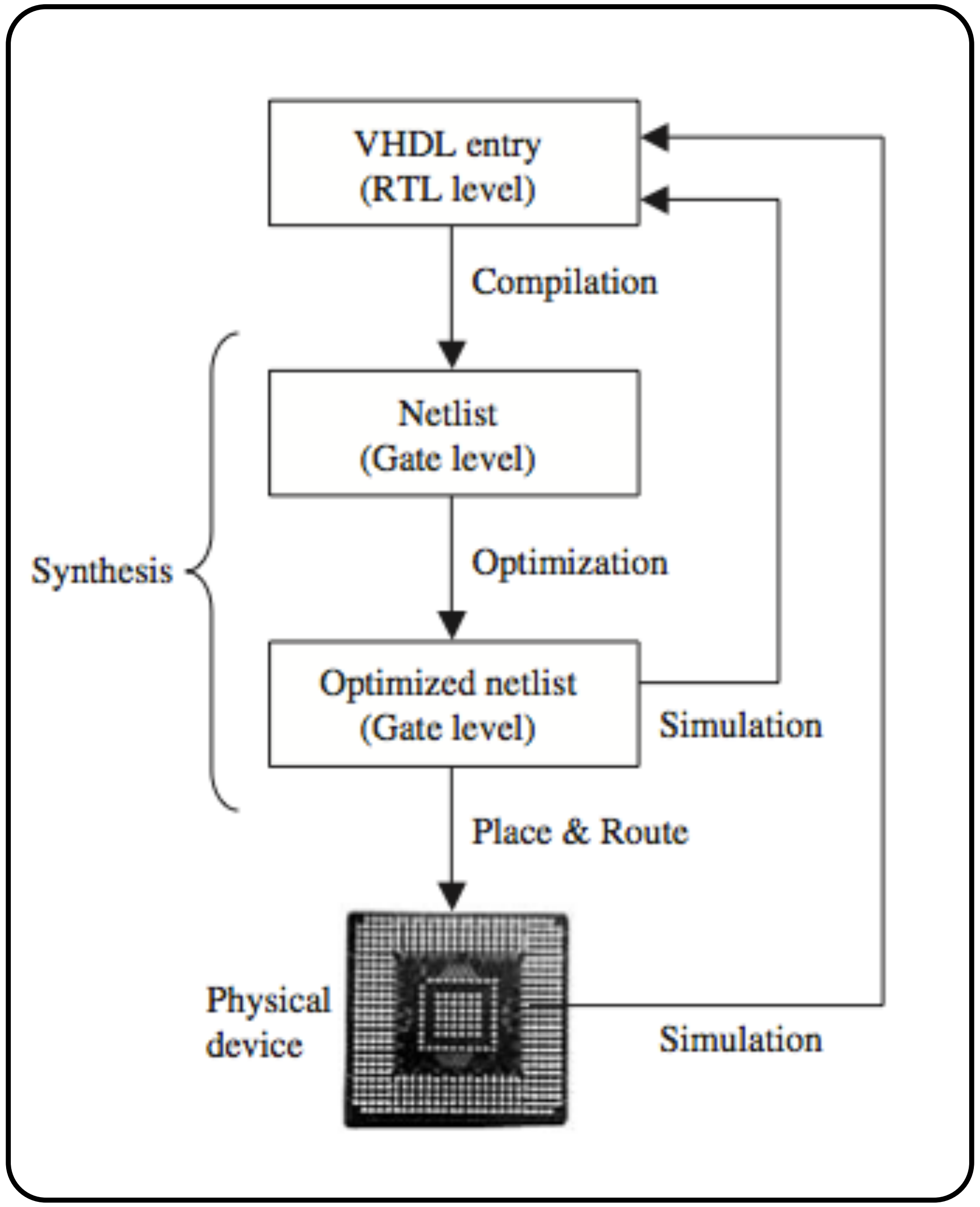
Design flow: VHDL entry

- As mentioned, the main purpose of VHDL is to synthesize a circuit into (or system) in a programmable device or in an ASIC
- We have to follow a sequence of steps
- We start the design by writing the VHDL code
- This is saved in a file with the extension .vhd



Design flow: netlist generation

- Compilation is the conversion of the high-level VHDL language, which describes the circuit at the Register Transfer Level (RTL), into a netlist at the gate level



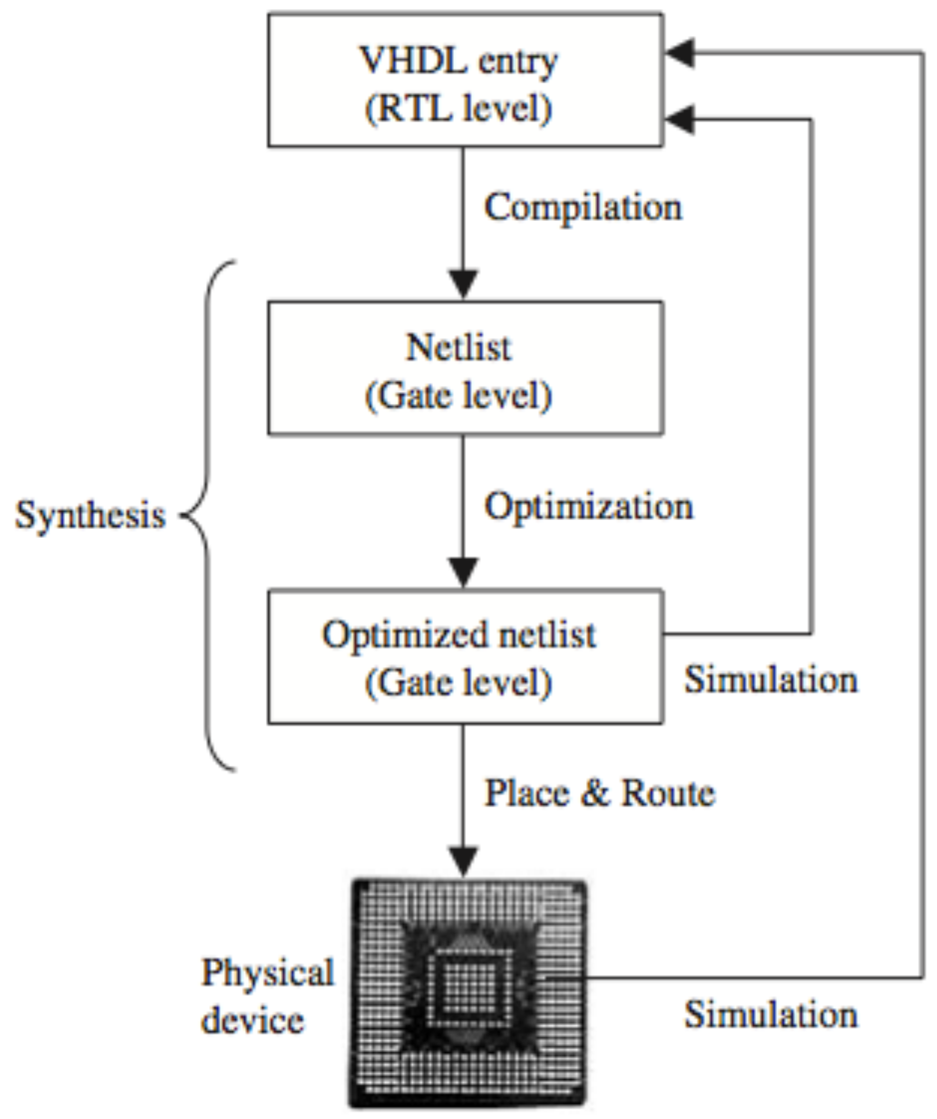
Design flow: optimization

- The second step is optimization, which is performed on the gate-level netlist for speed or for area
- Two different implementations will have different signal propagation times.

$$f = (a \oplus b \oplus cIn)$$
$$cOut = (a \wedge b) \vee (cIn \wedge (a \oplus b))$$

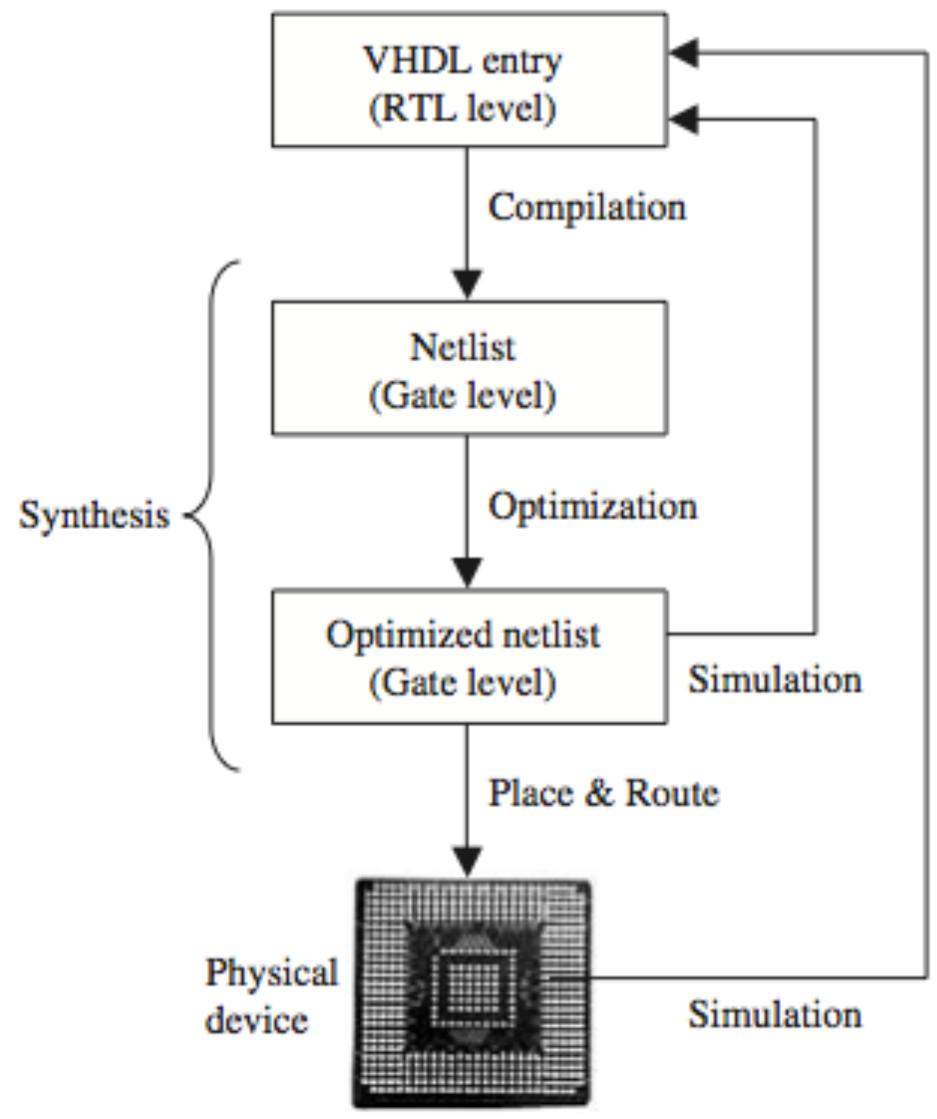
$$f = (a \oplus b) \oplus cIn$$
$$cOut = (a \wedge b) \vee ((a \wedge cIn) \oplus (b \wedge cIn))$$

- At this stage, the design can be simulated



Design flow: place and route

- Finally, a place-and-route software will generate the physical layout for a PLD/FPGA chip or will generate the masks for an ASIC
- Hard problem. May take hours to solve

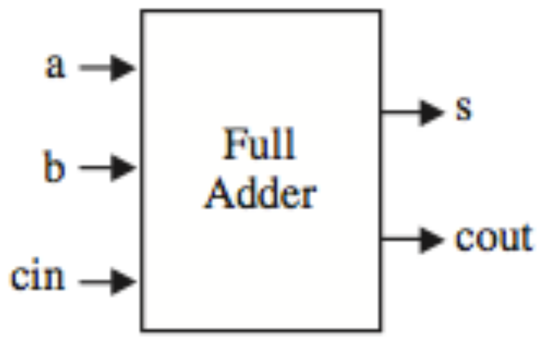


EDA Tools

- **E**lectronic **D**esign **A**utomation
- Major EDA companies are Mentor Graphics and Synopsis
- For small projects all tools for the development process are included in a single package, such as Xilinx ISE package.
- More complex projects require specialized tools
- For example more accurate circuit timings (Prime Time by Synopsys)
- Faster Routing

Translation of VHDL code into a circuit

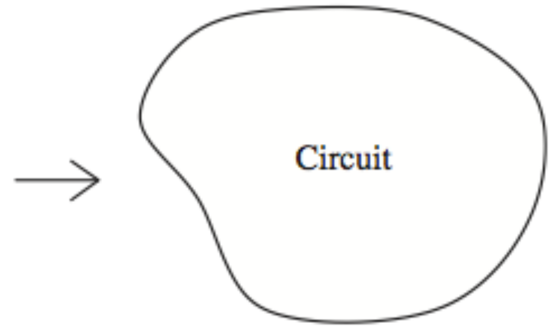
As an example here is a full adder and its equivalent truth table.



a b	cin	s	cout
0 0	0	0	0
0 1	0	1	0
1 0	0	1	0
1 1	0	0	1
0 0	1	1	0
0 1	1	0	1
1 0	1	0	1
1 1	1	1	1

Example of a VHDL code for the full adder unit

```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
        s, cout: OUT BIT);
END full_adder;
--------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```



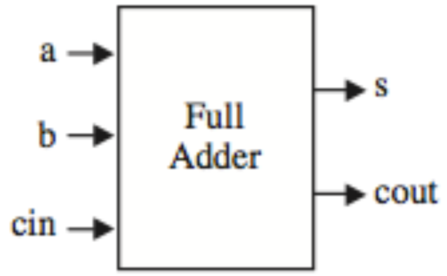
From the VHDL code shown, a physical circuit is inferred!

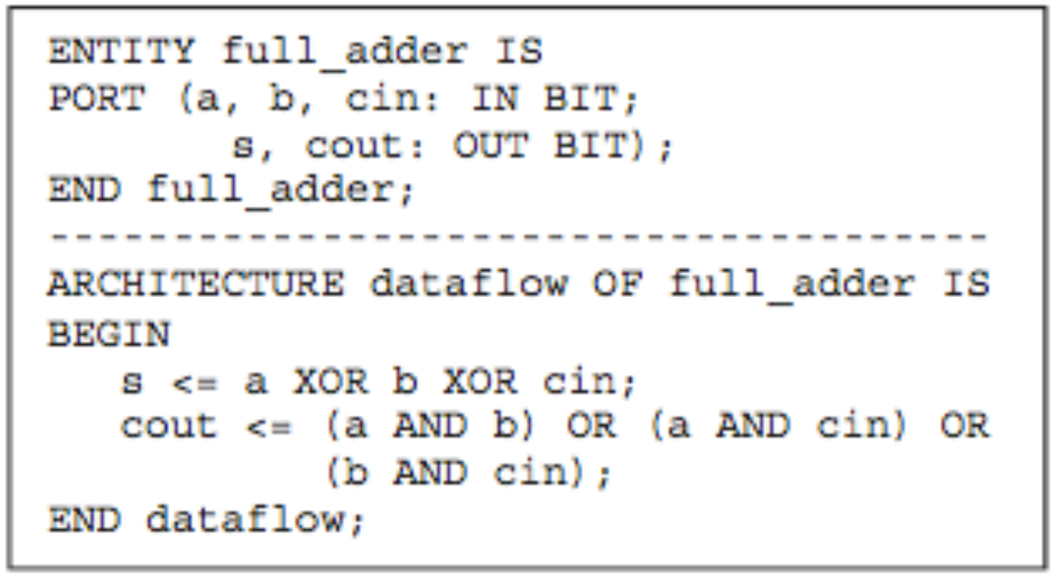
Translation of VHDL code into a circuit

• Consists of an ENTITY, which describes the pins (PORTS) of the circuit

and

• An ARCHITECTURE section, which describes how the circuit should function



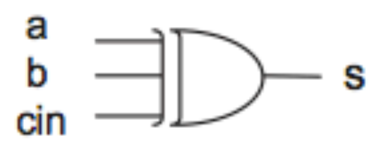


```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
        s, cout: OUT BIT);
END full_adder;
--------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```

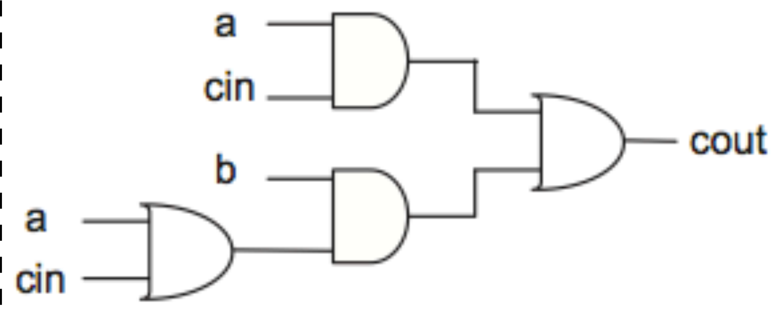
Different synthesis outcomes

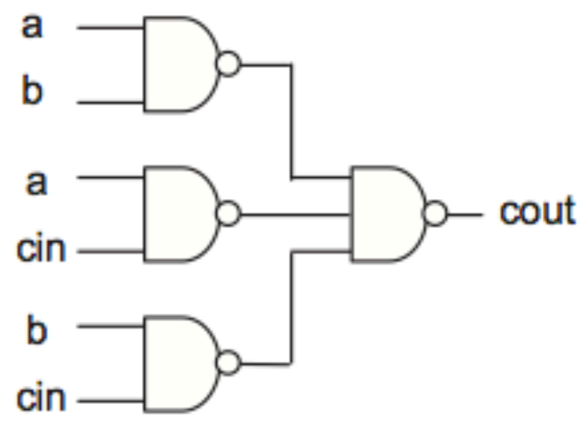
- Several ways of implementing the full adder
- Actual circuit will depend on the compiler/ optimizer being used and on the target technology
- Regardless of the circuit, the output **must** be the same

Reconfigurable device which has adder gates

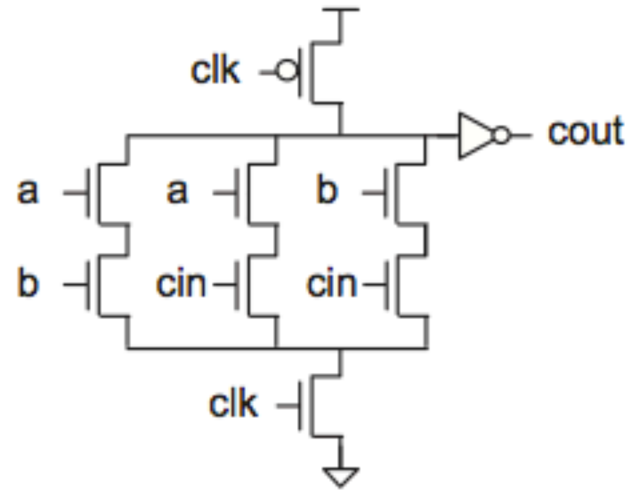


Reconfigurable device with just AND/OR gates





Reconfigurable device with just NAND gates



ASIC with MOS transistors and clocked domino logic

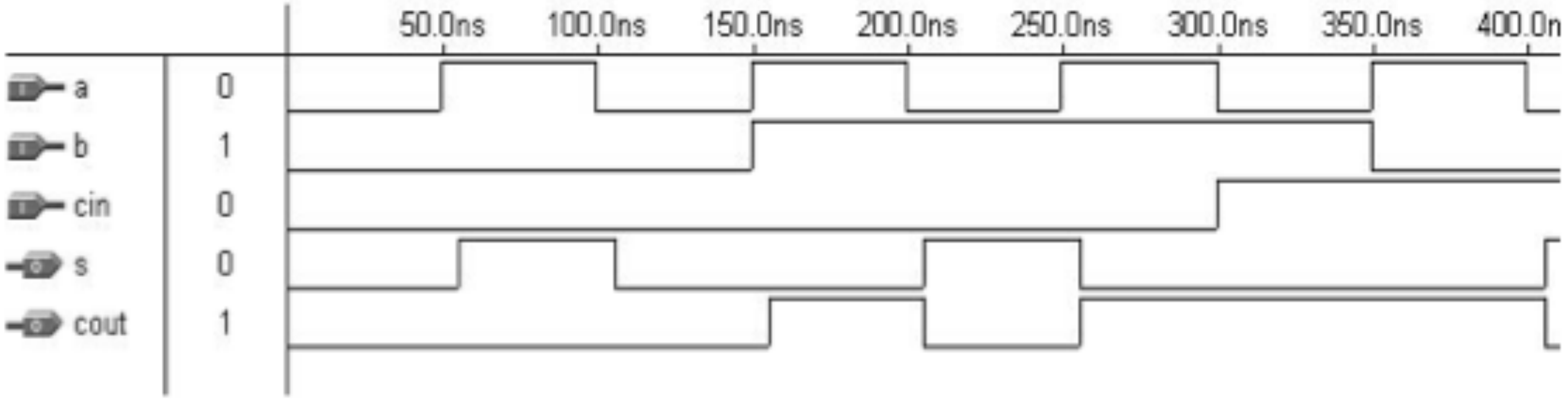
After each step testing is necessary

- You do not want to test your circuit after you manufacture it! Its too expensive
- Functional testing can be done with wave-forms

```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
        s, cout: OUT BIT);
END full_adder;
---------------------------------------
ARCHITECTURE dataflow OF full_adder IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin) OR
           (b AND cin);
END dataflow;
```

Inputs

Outputs



a b	cin	s	cout
0 0	0	0	0
0 1	0	1	0
1 0	0	1	0
1 1	0	0	1
0 0	1	1	0
0 1	1	0	1
1 0	1	0	1
1 1	1	1	1

Code structure

Preliminaries

- VHDL is **not** case sensitive, but its nice to use CAPS to highlight certain code blocks
- Comments are triggered with a double dash (--)
- A semi-colon (;) indicates the end of a statement

Fundamental code sections

There are 3 fundamental sections that comprise a piece of VHDL code:

- **LIBRARY declarations**: lists all libraries used in the design (e.g. ieee, std, work, etc)
- **ENTITY**: Specifies the I/O pins of the circuit
- **ARCHITECTURE**: Contains the VHDL code proper, which describes how the circuit behaves

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY dff IS
   PORT ( d, clk, rst: IN STD_LOGIC;
          q: OUT STD_LOGIC);
END dff;
---------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
   PROCESS (rst, clk)
   BEGIN
      IF (rst='1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk='1') THEN
         q <= d;
      END IF;
   END PROCESS;
END behavior;
---------------------------------------
```

What is a LIBRARY

- A **LIBRARY** is a collection of commonly used pieces of code
- Placing such pieces inside a library allows them to be reused or shared by other designs
- We will create our own libraries in the future
- To declare a LIBRARY (*to make it visible to the design*) two lines of code are needed.

```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY dff IS
   PORT ( d, clk, rst: IN STD_LOGIC;
          q: OUT STD_LOGIC);
END dff;
---------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
   PROCESS (rst, clk)
   BEGIN
      IF (rst='1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk='1') THEN
         q <= d;
      END IF;
   END PROCESS;
END behavior;
---------------------------------------
```

Three fundamental libraries

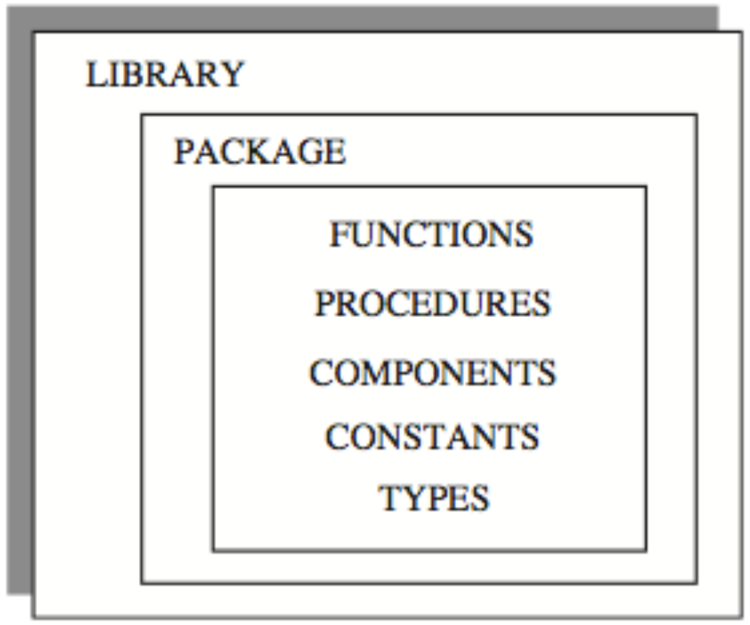
At least three packages, from three different libraries, are usually needed in a design:

- ieee.std_logic_1164 (from the ieee library)
- standard (from the std library)
- work (work library)

However, std and work are made visible by default

The ieee library only must be explicitly written when the STD_LOGIC or STD_ULOGIC data type is employed in the design. Data types will be discussed soon!

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY std;
USE std.standard.all;

LIBRARY work;
USE work.all;
```



ENTITY

An ENTITY is a list with specifications of all input and output pins (PORTS) of the circuit.

```
ENTITY entity_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```

signal_mode can be IN, OUT, INOUT, or BUFFER

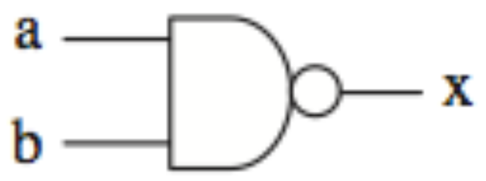
signal_type can be BIT, STD_LOGIC, INTEGER, etc

name can be anything except reserved words

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY dff IS
   PORT ( d, clk, rst: IN STD_LOGIC;
          q: OUT STD_LOGIC);
END dff;
---------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
   PROCESS (rst, clk)
   BEGIN
      IF (rst='1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk='1') THEN
         q <= d;
      END IF;
   END PROCESS;
END behavior;
---------------------------------------
```

ENTITY example

- 3 I/O pins
- 2 Inputs with a BIT data-type
- 1 Output with a BIT data-type
- Name of this ENTITY is *nand_gate*



```
ENTITY nand_gate IS
   PORT (a, b : IN BIT;
         x : OUT BIT);
END nand_gate;
```

ARCHITECTURE

- ARCHITECTURE is how the circuit should behave (function)

```
ARCHITECTURE architecture_name OF entity_name IS
    [declarations]
BEGIN
    (code)
END architecture_name;
```

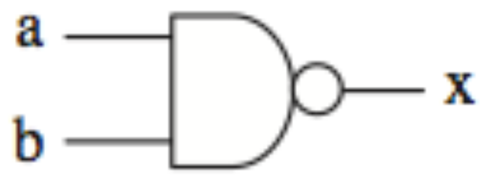
[declarations] (optional), where signals and constants are declared

architecture_name, can be any name except reserved words

entity_name, must be the same name as the entity with the ports declaration

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY dff IS
   PORT ( d, clk, rst: IN STD_LOGIC;
          q: OUT STD_LOGIC);
END dff;
---------------------------------------
ARCHITECTURE behavior OF dff IS
BEGIN
   PROCESS (rst, clk)
   BEGIN
      IF (rst='1') THEN
         q <= '0';
      ELSIF (clk'EVENT AND clk='1') THEN
         q <= d;
      END IF;
   END PROCESS;
END behavior;
---------------------------------------
```

ARCHITECTURE example



- The circuit performs the NAND operation between the two input signals (a, b) and assigns (<=) the result to the output pin (x)
- The architecture name is myarch

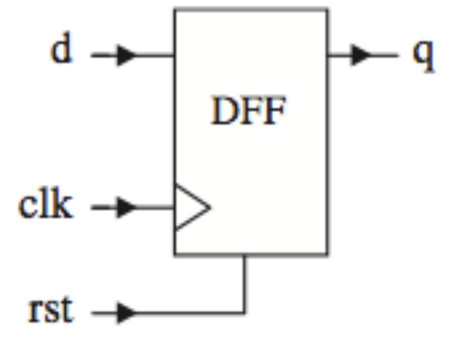
```
ENTITY nand_gate IS
   PORT (a, b : IN BIT;
         x : OUT BIT);
END nand_gate;

ARCHITECTURE myarch OF nand_gate IS
BEGIN
   x <= a NAND b;
END myarch;
```

Additional introductory examples

- We haven't studied everything that will show up on the next examples
- However, they are good examples to learn VHDL structure

D Flip-Flop in VHDL

- DFF is triggered at the rising-edge of the clock (clk)
- When reset (rst) is 1⁄4 of logic 1, the output must be turned low, regardless of clk.
- Otherwise, the output must copy the input when clk changes from “0” to “1”... which is on the rising edge of clk.

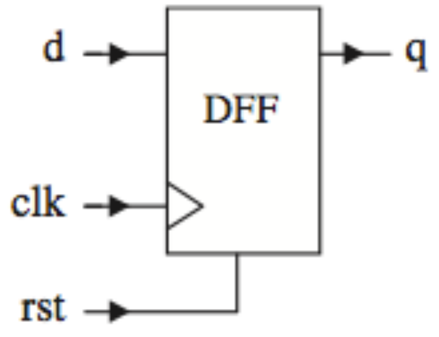


DFF code in VHDL

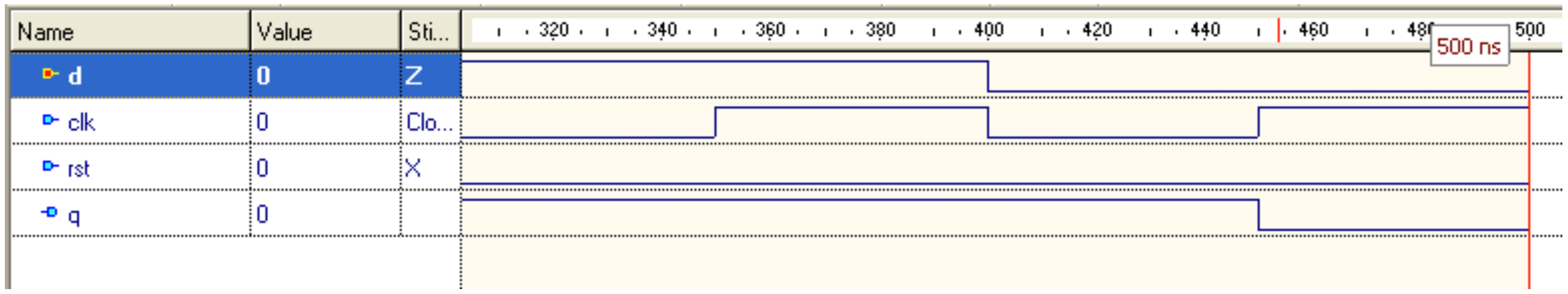
```
library ieee;
use ieee.std_logic_1164.all;

entity DFF is
   port (d, clk, rst: IN std_logic;
   q: OUT std_logic);
end DFF;

architecture myarch of DFF is
begin
   PROCESS (rst, clk)
   begin
       if (rst='1') then
           q<='0';
       elsif (clk'event and clk='1') then
           q<=d;
       end if;
   end process;
end myarch;
```

- Input ports can only be IN. Here all input signals are of type STD_LOGIC.
- Output port can be OUT, INOUT, or BUFFER.
- In VHDL everything happens in parallel. However, a PROCESS forces code to be executed sequentially.
- The PROCESS is entered each time clk or rst changes.
- clk'event means clock has changed... an event has occurred in clk.

DFF Waveform



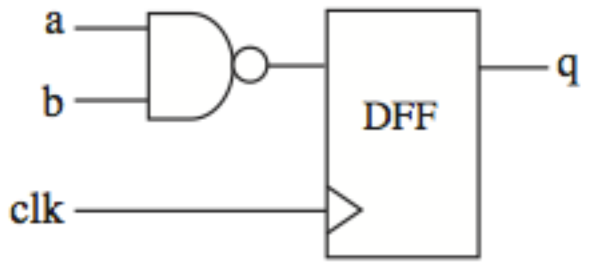
This VHDL implementation works as expected.



Name	Value	Sti...
d	0	Z
clk	0	Clo...
rst	0	X
q	0	

DFF + NAND gate

```
entity example is
    port (a,b,clk : in bit;
    q: out bit);
end example;


architecture example of example is
signal temp : bit;

begin
  temp <= a NAND b;
  process (clk)
    begin
    if (clk'event and clk='1') then q<=temp;
    end if;
  end process;
end example;
```



- The "<=" operator is used to assign a value to a SIGNAL
- In contrast, ":=" would be used for a VARIABLE
- The signal temp, of type BIT, was declared. There is no mode declaration (mode is only used in entities)
- Instead of *q<=temp* we could have written *q<=a NAND b*