Embedded electronics

Rémi Parrot

remi.parrot@ec-nantes.fr

CENTRALE NANTES

Overview

References

Content

Boolean Algebra

VHDL

- Introduction
- Code structure
- Data types
- Operators
- Concurrent code
- Sequential code
- Composition

Mealy and Moore Machines

Conclusion

References

References

- Previous teacher: Joumana Lagha
- Circuit Design with VHDL, by Volnei A. Pedroni

Content

Content

- Lecture + tutorials : 12h
- Lab : 20h
- Exam : 2h

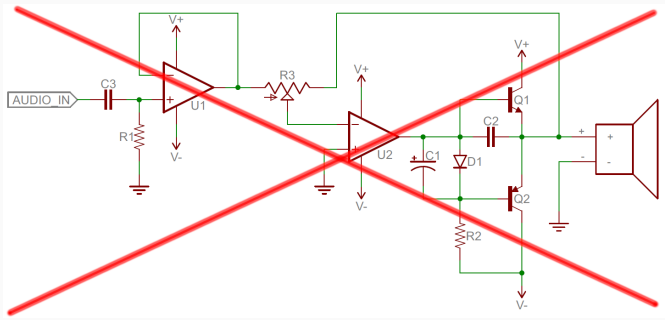
Content

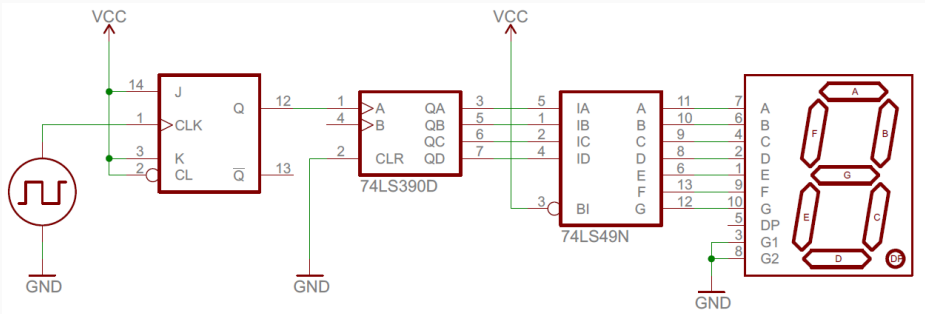
Content

- Designing Circuit

Content

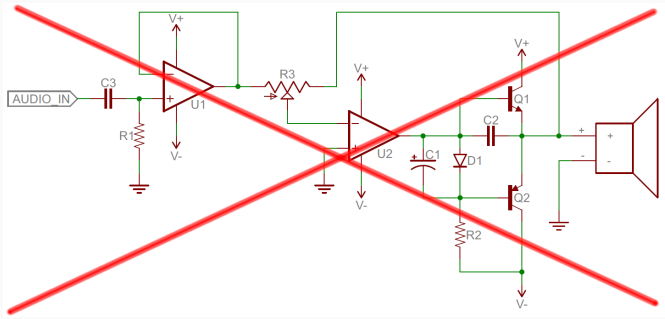
- Designing Circuit
- ~~Analog circuit~~ Logic circuit

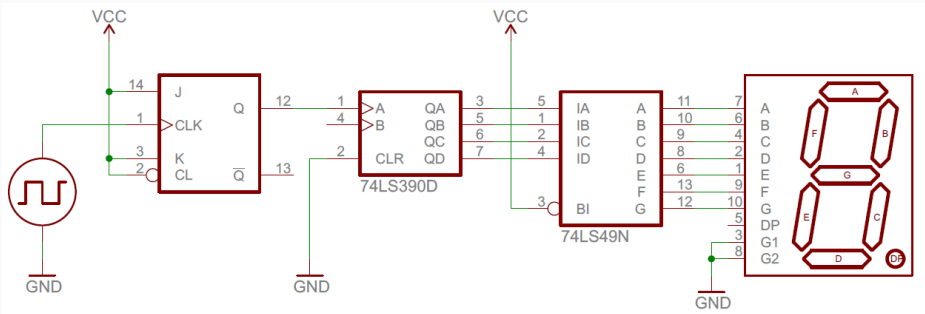




Content

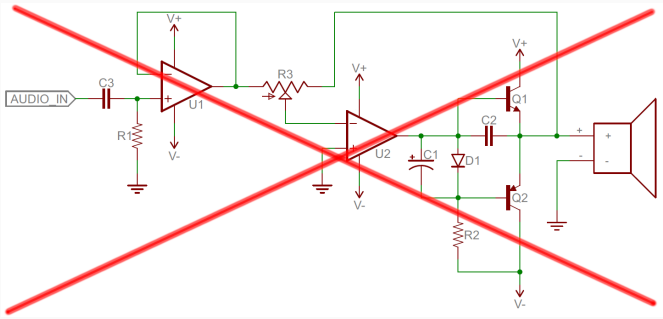
- Designing Circuit
- ~~Analog circuit~~ Logic circuit
- Boolean Algebra

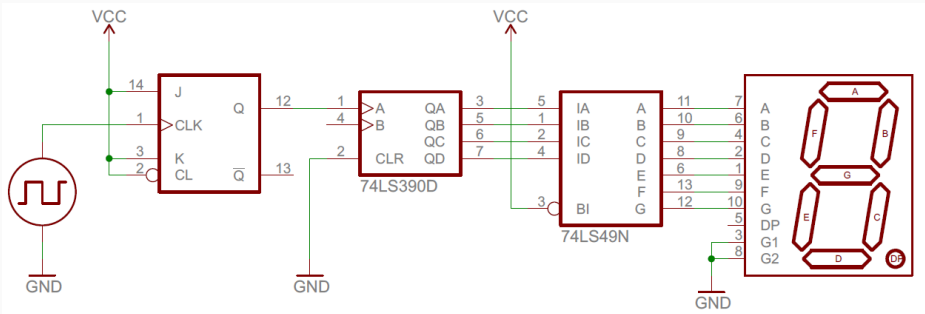




Content

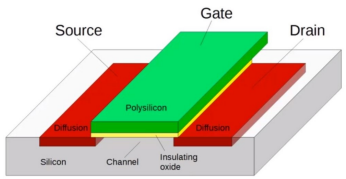
- Designing Circuit
- ~~Analog circuit~~ Logic circuit
- Boolean Algebra
- **Description** Language : VHDL



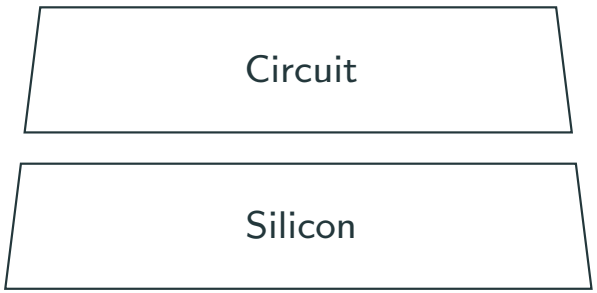


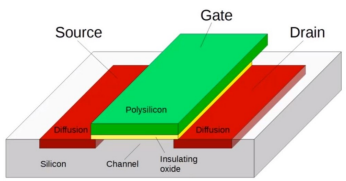
Content - abstraction layers

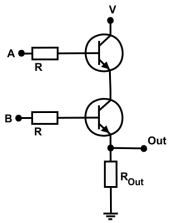
Content - abstraction layers



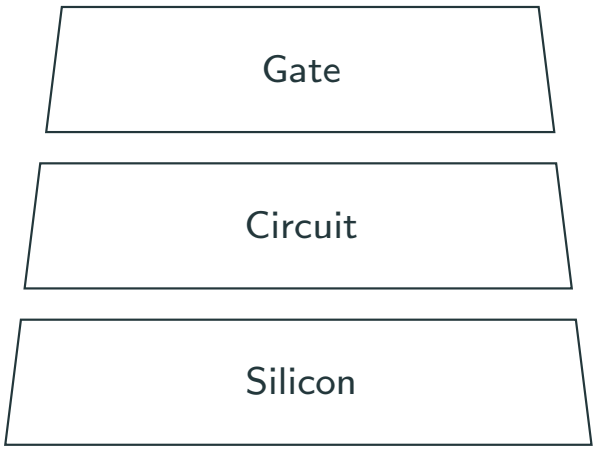
Content - abstraction layers

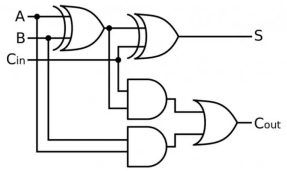


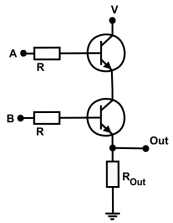


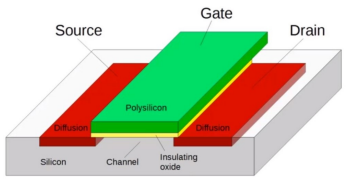


Content - abstraction layers

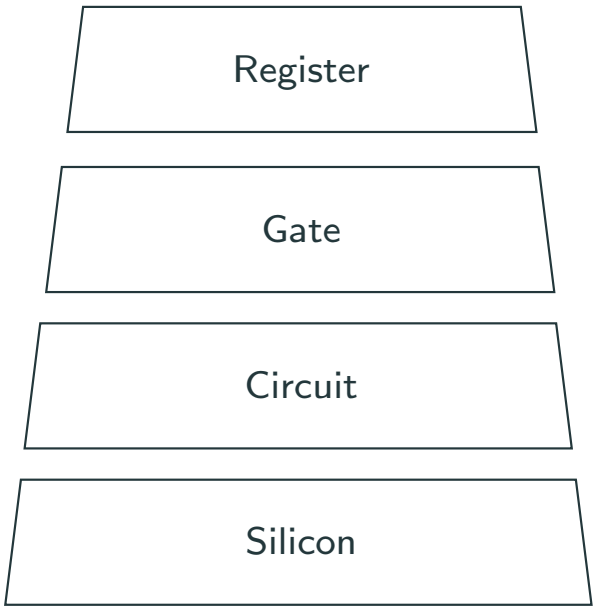


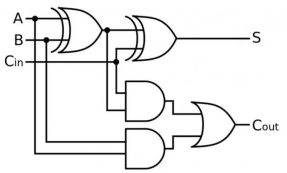


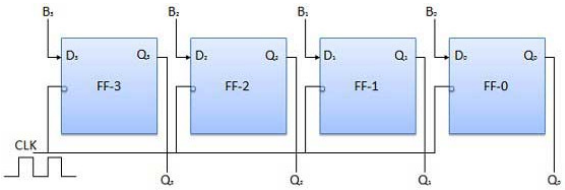


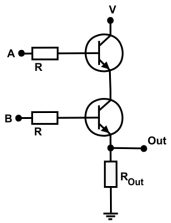


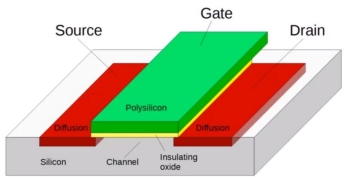
Content - abstraction layers



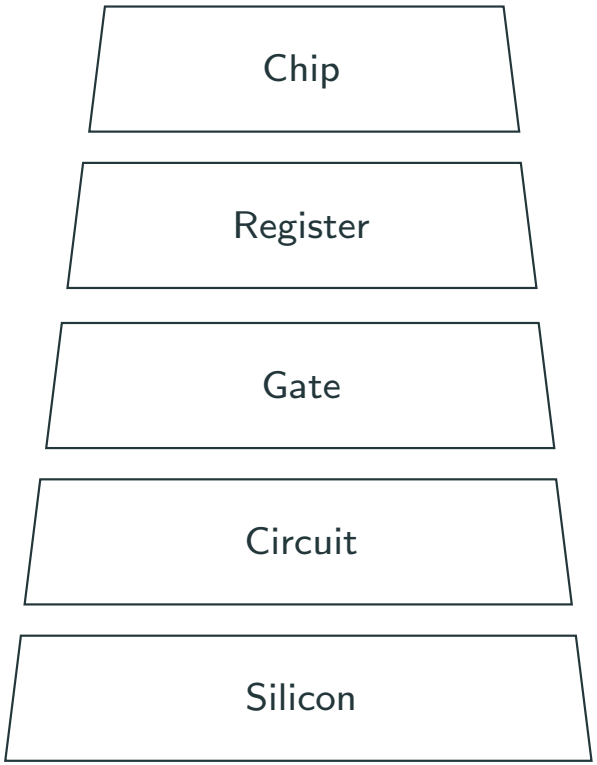


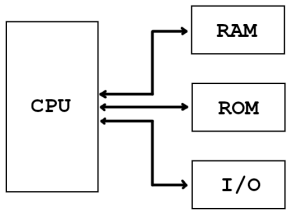


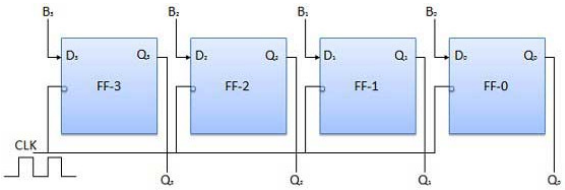


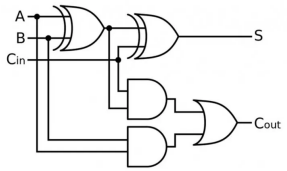


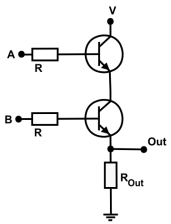
Content - abstraction layers

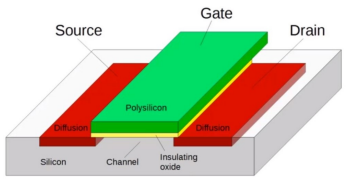




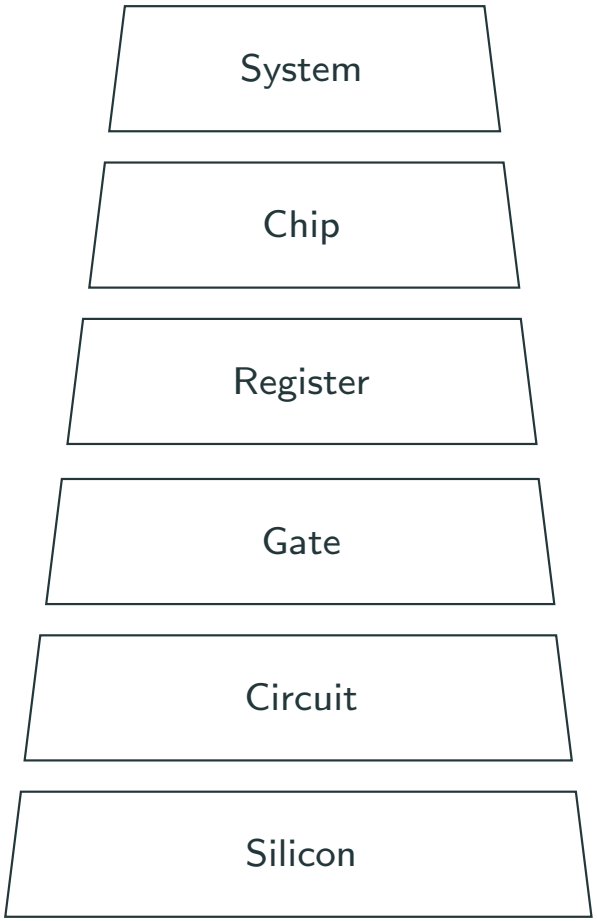


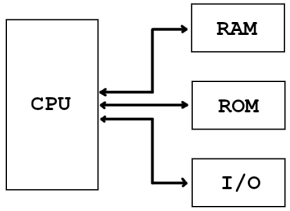


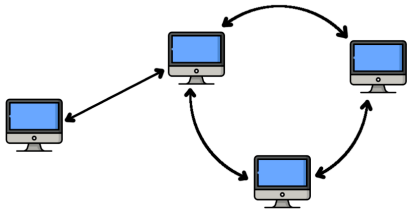


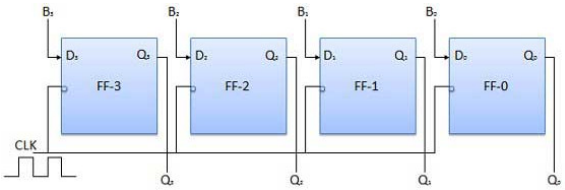


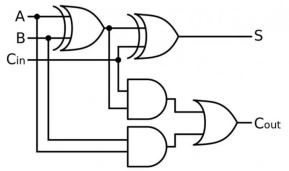
Content - abstraction layers

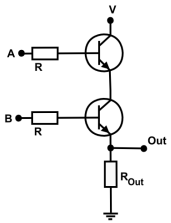


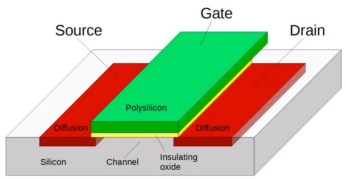




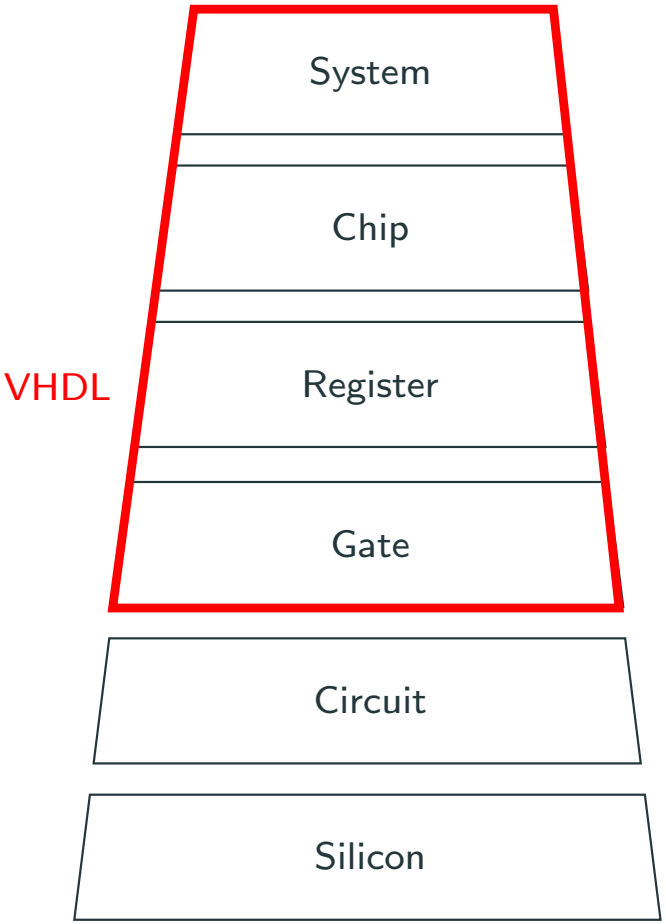


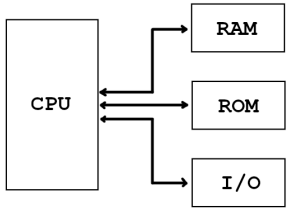


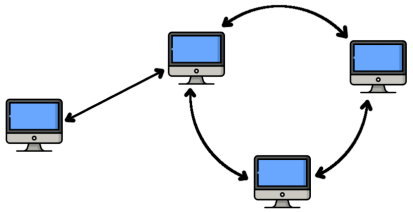


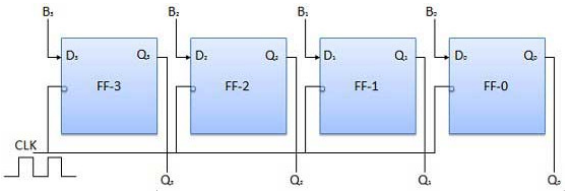


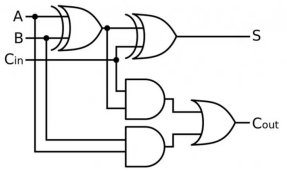
Content - abstraction layers

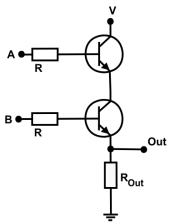


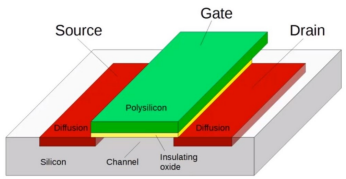












Boolean Algebra

Quick definition

Logic Variable
Variable that can take 2 values ("true" or "false")

- "false" is noted 0
- "true" is noted 1

Quick definition

Logic Variable
Variable that can take 2 values ("true" or "false")

- "false" is noted 0
- "true" is noted 1

Logic Function
Function on logic variables

- *input*: some logic variables;
- *output*: one logic value.

Quick definition

Logic Variable
Variable that can take 2 values ("true" or "false")

- "false" is noted 0
- "true" is noted 1

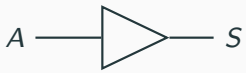
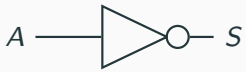
Logic Function
Function on logic variables

- *input*: some logic variables;
- *output*: one logic value.

Example
With only 1 input:

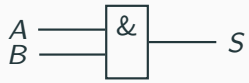
A	constant $f(A) = 0$	constant $f(A) = 1$	identity $f(A) = A$	negation $f(A) = \neg A$
0	0	1	0	1
1	0	1	1	0

Logic Gates

	USA	Europe	Truth table
			A | $S = A$ 0 | 0 1 | 1
NOT			A | $S = \neg A$ 0 | 1 1 | 0

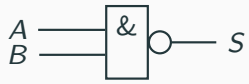
Logic Gates

AND



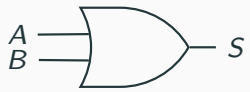
A	B	$S = A.B$
0	0	0
0	1	0
1	0	0
1	1	1

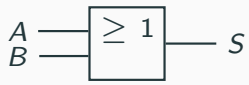
NAND



A	B	$S = \overline{A.B}$
0	0	1
0	1	1
1	0	1
1	1	0

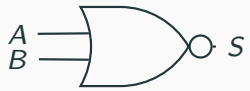
OR

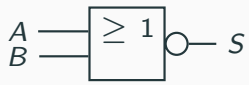




A	B	$S = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

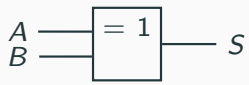
NOR





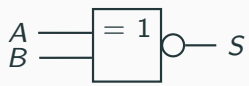
A	B	$S = \overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

XOR



A	B	$S = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

XNOR



A	B	$S = \overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

Some properties

Commutativity

- $A.B = B.A$
- $A + B = B + A$

Associativity

- $A.(B.C) = (A.B).C$
- $A + (B + C) = (A + B) + C$

Neutral element

- $A.1 = A$
- $A + 0 = A$

Absorbing element

- $A.0 = 0$
- $A + 1 = 1$

Involution

- $\overline{\overline{A}} = A$

Inverse element

- $A.\overline{A} = 0$
- $A + \overline{A} = 1$

Idempotence

- $A.A = A$
- $A + A = A$

Distributivity

- $A.(B + C) = A.B + A.C$

Some properties

Commutativity

- $A.B = B.A$
- $A + B = B + A$

Associativity

- $A.(B.C) = (A.B).C$
- $A + (B + C) = (A + B) + C$

Neutral element

- $A.1 = A$
- $A + 0 = A$

Absorbing element

- $A.0 = 0$
- $A + 1 = 1$

Involution

- $\overline{\overline{A}} = A$

Inverse element

- $A.\overline{A} = 0$
- $A + \overline{A} = 1$

Idempotence

- $A.A = A$
- $A + A = A$

Distributivity

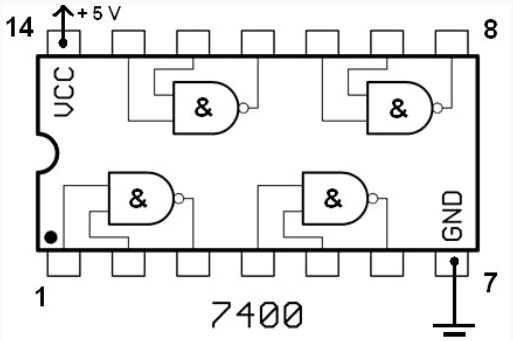
- $A.(B + C) = A.B + A.C$

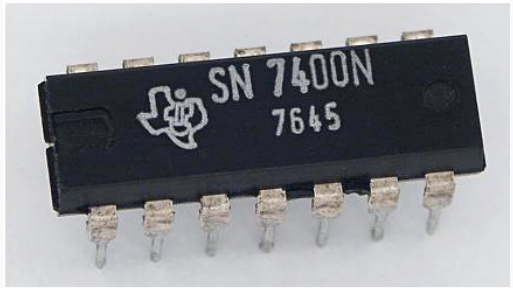
Morgan Laws

- $\overline{A.B} = \overline{A} + \overline{B}$
- $\overline{A + B} = \overline{A}.\overline{B}$

Some properties

Consequence
All logical functions can be built with *NAND gate* (or *NOR gate*).





Exercises

Exercise
Simplify the following expressions:

1. $A + A.B$
2. $A.(A + B)$
3. $(A + B).(A + \overline{B})$

Exercises

Exercise
Simplify the following expressions:

1. $A + A.B = A.1 + A.B = A.(1 + B) = A.1 = A$
2. $A.(A + B)$
3. $(A + B).(A + \overline{B})$

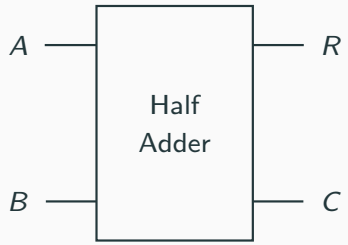
Exercises

Exercise
Simplify the following expressions:

1. $A + A.B = A.1 + A.B = A.(1 + B) = A.1 = A$
2. $A.(A + B) = A.A + A.B = A + A.B = A$
3. $(A + B).(A + \overline{B})$
   $= A.A + A.\overline{B} + B.A + B.\overline{B} = A + A.\overline{B} + A.B + 0 = A.(1 + \overline{B} + B) = A.1 = A$

Exercises

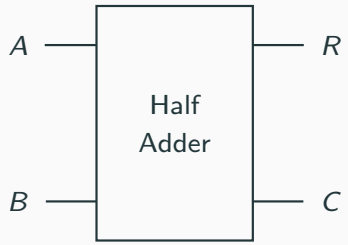
Exercise
Let's build a *one bit adder* with a carry.



1. Write the truth table of the adder

Exercises

Exercise
Let's build a *one bit adder* with a carry.



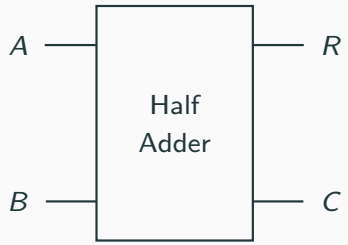
1. Write the truth table of the adder

1.

A	B	R	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Exercises

Exercise
Let's build a *one bit adder* with a carry.



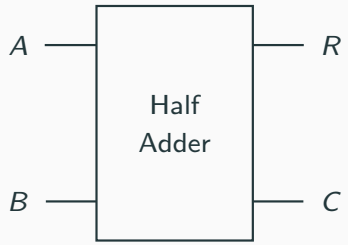
1. Write the truth table of the adder
2. Deduce the logical function $R = f(A, B)$

1.

A	B	R	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Exercises

Exercise
Let's build a *one bit adder* with a carry.



1. Write the truth table of the adder
2. Deduce the logical function $R = f(A, B)$

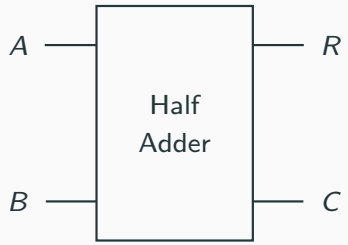
1.

A	B	R	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2. $R = A \oplus B$

Exercises

Exercise
Let's build a *one bit adder* with a carry.



1. Write the truth table of the adder
2. Deduce the logical function $R = f(A, B)$
3. Deduce the logical function $C = g(A, B)$

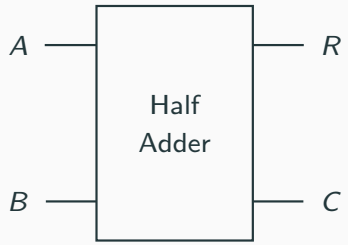
1.

A	B	R	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2. $R = A \oplus B$

Exercises

Exercise
Let's build a *one bit adder* with a carry.



1. Write the truth table of the adder
2. Deduce the logical function $R = f(A, B)$
3. Deduce the logical function $C = g(A, B)$

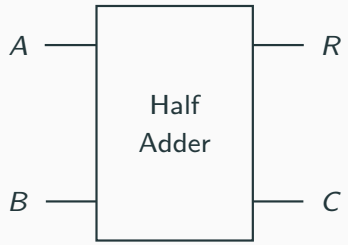
1.

A	B	R	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2. $R = A \oplus B$
3. $C = A.B$

Exercises

Exercise
Let's build a *one bit adder* with a carry.



1. Write the truth table of the adder
2. Deduce the logical function $R = f(A, B)$
3. Deduce the logical function $C = g(A, B)$
4. Draw the full circuit

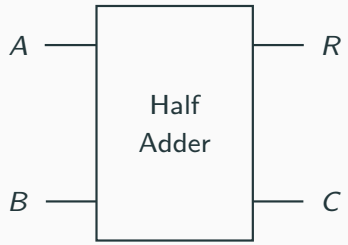
1.

A	B	R	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2. $R = A \oplus B$
3. $C = A.B$

Exercises

Exercise
Let's build a *one bit adder* with a carry.

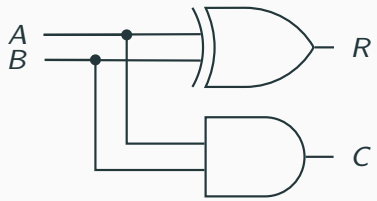


1. Write the truth table of the adder
2. Deduce the logical function $R = f(A, B)$
3. Deduce the logical function $C = g(A, B)$
4. Draw the full circuit

1.

A	B	R	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

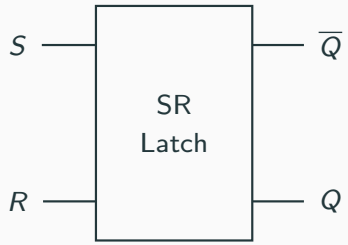
2. $R = A \oplus B$
3. $C = A.B$
4.



Exercises

Flip-flop (or latch)
A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise
Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$

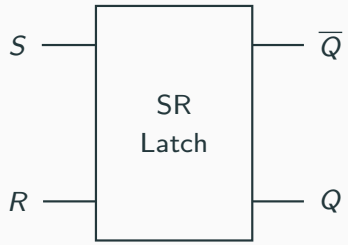
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$

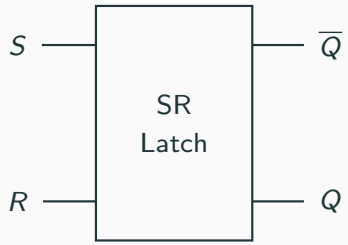
1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

Exercises

Flip-flop (or latch)
A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise
Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$

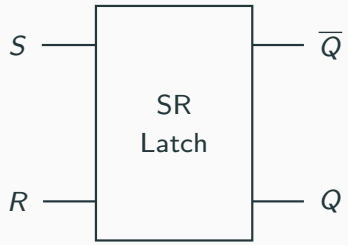
1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

Exercises

Flip-flop (or latch)
A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise
Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$

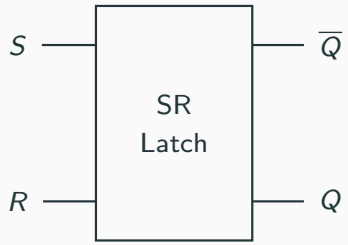
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$

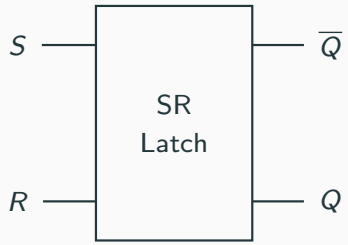
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$
3. $\overline{Q_{next}} = \overline{S + Q}$

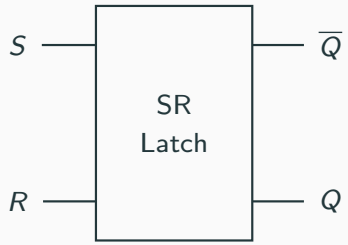
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$
4. Write the truth table $Q_{next} = h(R, \overline{Q})$

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$
3. $\overline{Q_{next}} = \overline{S + Q}$

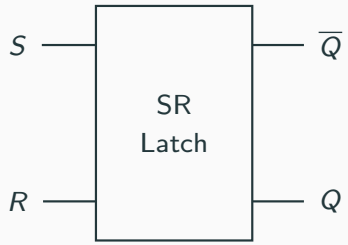
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$
4. Write the truth table $Q_{next} = h(R, \overline{Q})$

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$
3. $\overline{Q_{next}} = \overline{S + Q}$

4.

R	$\overline{Q}$	Q_{next}
0	0	1
0	1	0
1	0	0
1	1	0

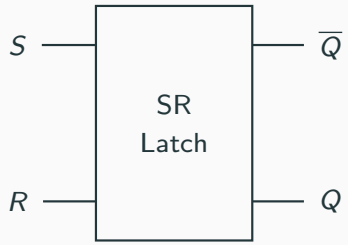
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$
4. Write the truth table $Q_{next} = h(R, \overline{Q})$
5. Deduce the logical function $Q_{next} = h(R, \overline{Q})$

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$
3. $\overline{Q_{next}} = \overline{S + Q}$

4.

R	$\overline{Q}$	Q_{next}
0	0	1
0	1	0
1	0	0
1	1	0

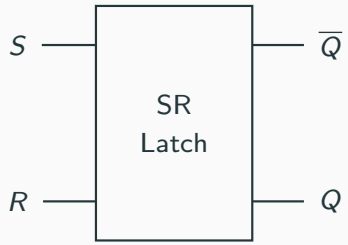
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$
4. Write the truth table $Q_{next} = h(R, \overline{Q})$
5. Deduce the logical function $Q_{next} = h(R, \overline{Q})$

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$
3. $\overline{Q_{next}} = \overline{S + Q}$

4.

R	$\overline{Q}$	Q_{next}
0	0	1
0	1	0
1	0	0
1	1	0

5. $Q_{next} = \overline{R + \overline{Q}}$

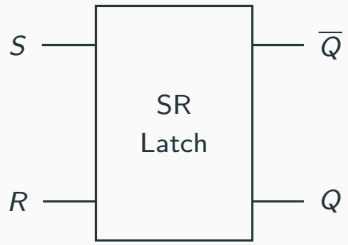
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$
4. Write the truth table $Q_{next} = h(R, \overline{Q})$
5. Deduce the logical function $Q_{next} = h(R, \overline{Q})$
6. Draw the full circuit

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

2. $Q_{next} = S + Q$
3. $\overline{Q_{next}} = \overline{S + Q}$

4.

R	$\overline{Q}$	Q_{next}
0	0	1
0	1	0
1	0	0
1	1	0

5. $Q_{next} = \overline{R + \overline{Q}}$

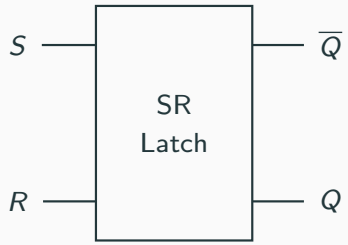
Exercises

Flip-flop (or latch)

A *flip-flop* is a circuit that has *two stables states*. It can be used to *store information*.

Exercise

Let's build a simple *SR latch*.



S	R	Q_{next}	
0	0	Q	Memory
0	1	0	Reset
1	0	1	Set
1	1		(prohibited)

1. Write the truth table $Q_{next} = f(S, Q)$
2. Deduce the logical function $Q_{next} = f(S, Q)$
3. Write the logical function $\overline{Q_{next}} = g(S, Q)$
4. Write the truth table $Q_{next} = h(R, \overline{Q})$
5. Deduce the logical function $Q_{next} = h(R, \overline{Q})$
6. Draw the full circuit

1.

S	Q	Q_{next}
0	0	0
0	1	1
1	0	1
1	1	1

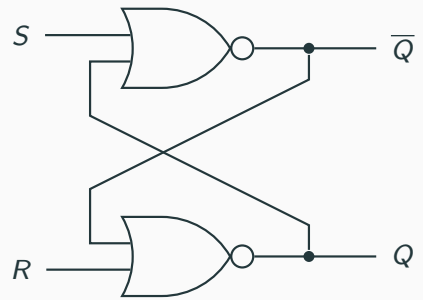
2. $Q_{next} = S + Q$
3. $\overline{Q_{next}} = \overline{S + Q}$

4.

R	$\overline{Q}$	Q_{next}
0	0	1
0	1	0
1	0	0
1	1	0

5. $Q_{next} = \overline{R + \overline{Q}}$

6.



VHDL

VHDL

Introduction

About VHDL

VHDL

- *Hardware Description Language*: describes the *behavior* of the system from which the physical circuit can then be implemented.

About VHDL

VHDL

- *Hardware Description Language*: describes the *behavior* of the system from which the physical circuit can then be implemented.
- VHDL = "VHSIC Hardware Description Language". VHSIC = "Very High Speed Integrated Circuits".

About VHDL

VHDL

- *Hardware Description Language*: describes the *behavior* of the system from which the physical circuit can then be implemented.
- VHDL = "VHSIC Hardware Description Language". VHSIC = "Very High Speed Integrated Circuits".
- Created in the 1980s.

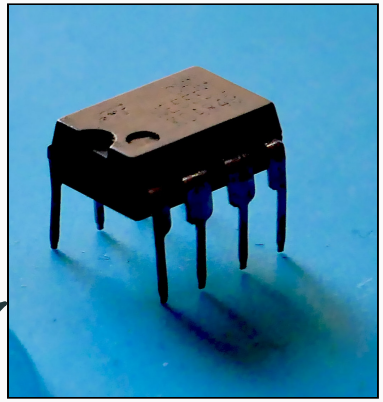
About VHDL

VHDL

- *Hardware Description Language*: describes the *behavior* of the system from which the physical circuit can then be implemented.
- VHDL = "VHSIC Hardware Description Language". VHSIC = "Very High Speed Integrated Circuits".
- Created in the 1980s.
- Two main applications: *programmable logic devices* (CPLD, FPGA) and design of *integrated circuit* (ASIC)

VHDL description

About VHDL



ASIC

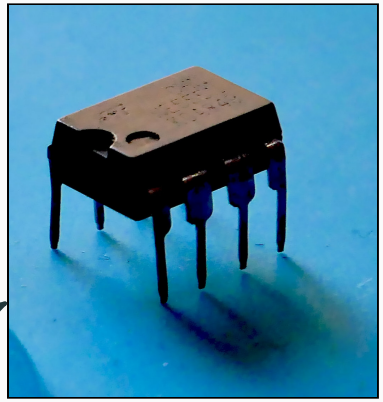
Definitive

VHDL description

About VHDL

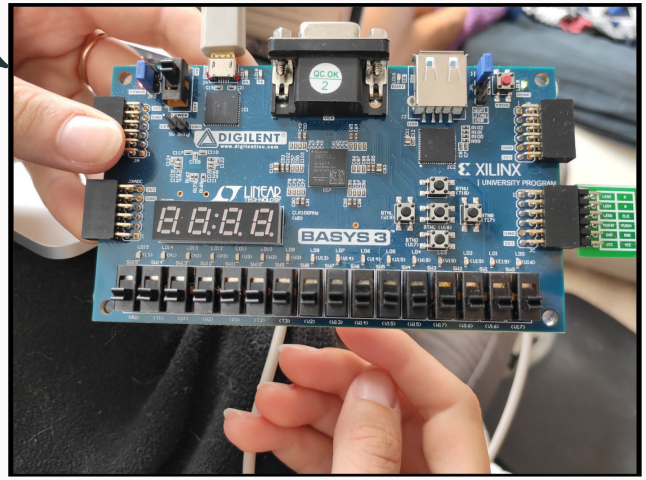
VHDL description

Definitive



ASIC

Temporary



FPGA

EDA Tools

- Electronic Design Automation tools: synthesis, implementation and simulation of VHDL.

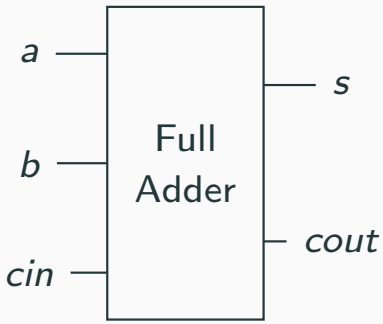
EDA Tools

- Electronic Design Automation tools: synthesis, implementation and simulation of VHDL.
- Some tools are offered as part of vendor's design suite: Altera, Xilinx, ...

EDA Tools

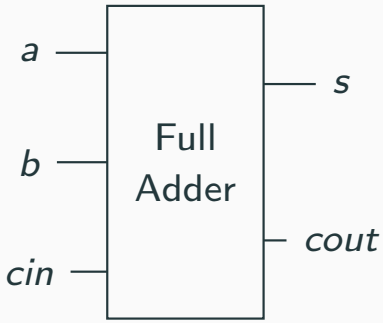
- Electronic Design Automation tools: synthesis, implementation and simulation of VHDL.
- Some tools are offered as part of vendor's design suite: Altera, Xilinx, ...
- During this course we will use Xilinx's Vivado suite, for Xilinx's CPLD/FPGA chips

Translation of VHDL Code into a Circuit



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

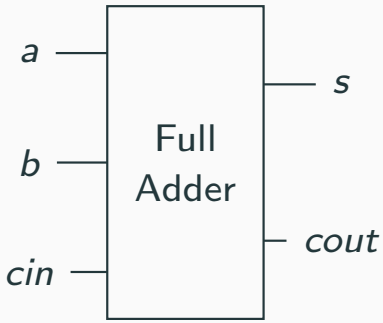
Translation of VHDL Code into a Circuit



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

$$s = a \oplus b \oplus cin$$

Translation of VHDL Code into a Circuit



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

$$s = a \oplus b \oplus cin$$

$$cout = a.b + a.cin + b.cin$$

Translation of VHDL Code into a Circuit

$$s = a \oplus b \oplus cin$$

$$cout = a.b + a.cin + b.cin$$

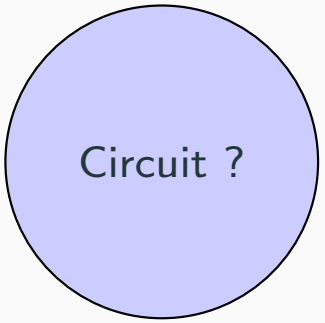
```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
      s, cout: OUT BIT);
END full_adder;
-------------------------------------
ARCHITECTURE dataflow OF full_adder
       IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin)
          OR
         (b AND cin);
END dataflow;
```

Translation of VHDL Code into a Circuit

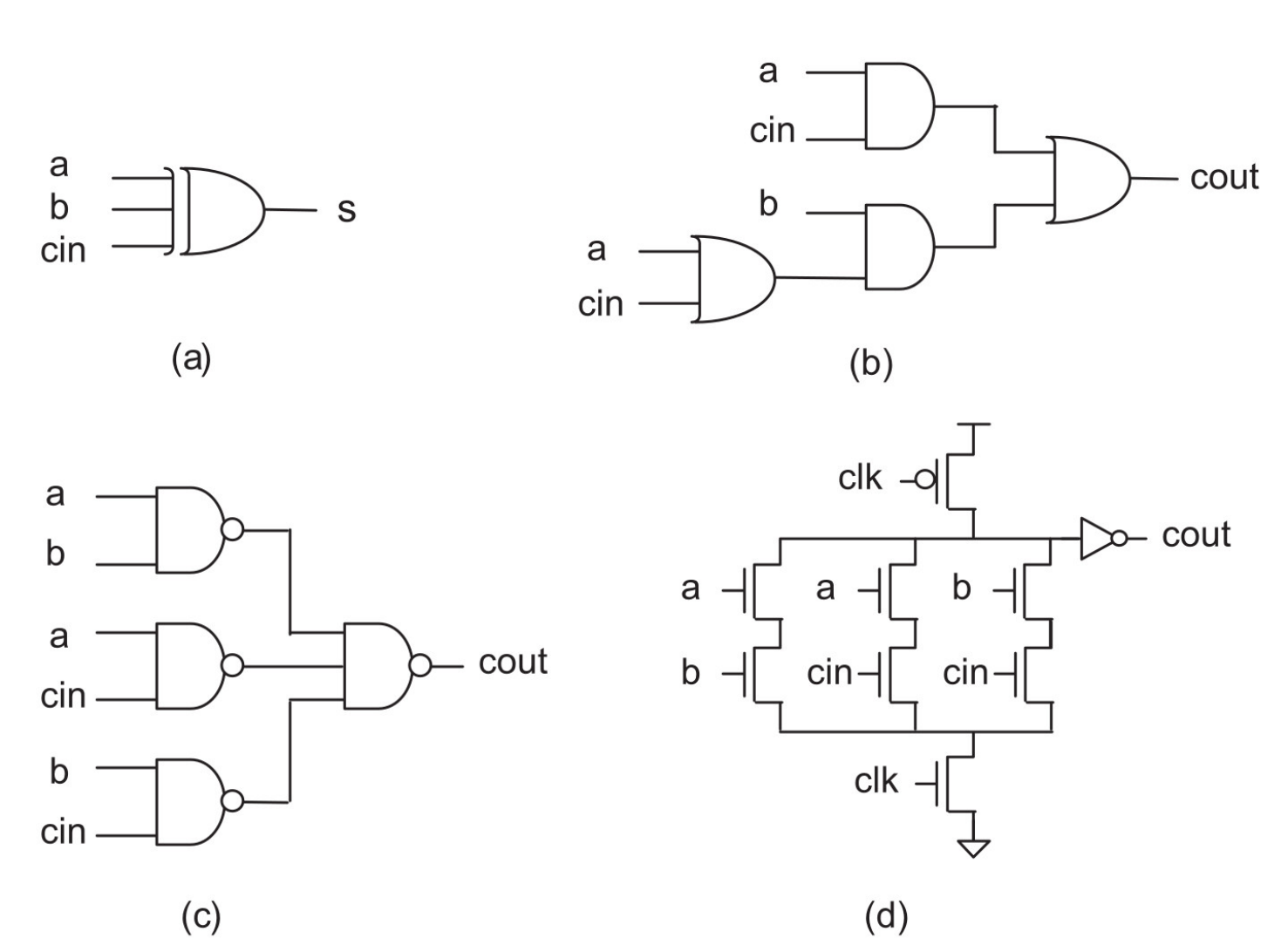
$$s = a \oplus b \oplus cin$$

$$cout = a.b + a.cin + b.cin$$

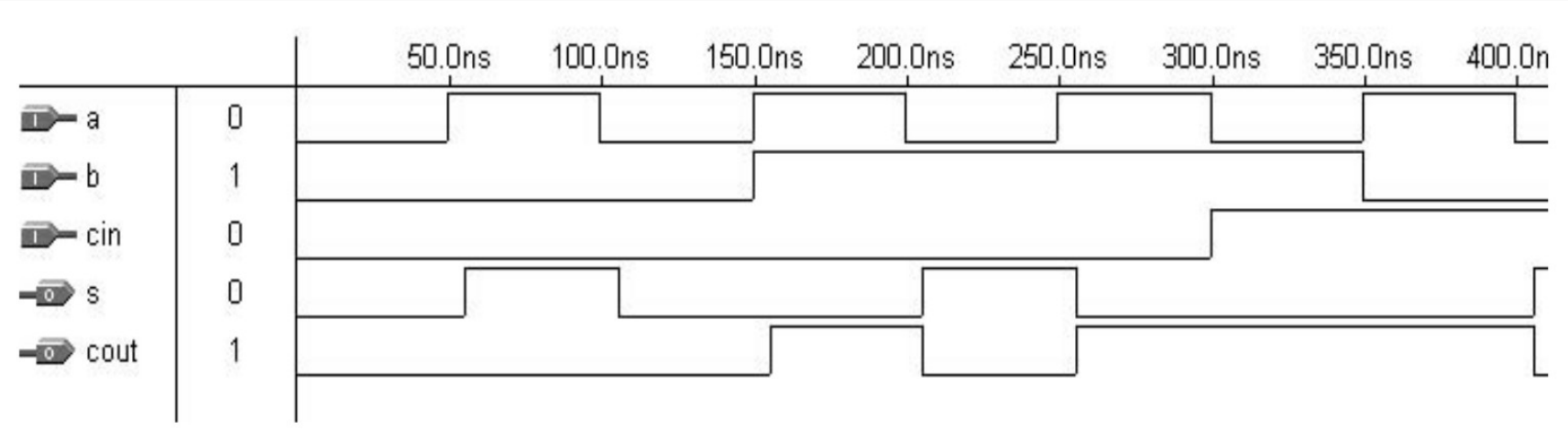
```
ENTITY full_adder IS
PORT (a, b, cin: IN BIT;
      s, cout: OUT BIT);
END full_adder;
-------------------------------------
ARCHITECTURE dataflow OF full_adder
      IS
BEGIN
   s <= a XOR b XOR cin;
   cout <= (a AND b) OR (a AND cin)
          OR
         (b AND cin);
END dataflow;
```



Translation of VHDL Code into a Circuit



Translation of VHDL Code into a Circuit



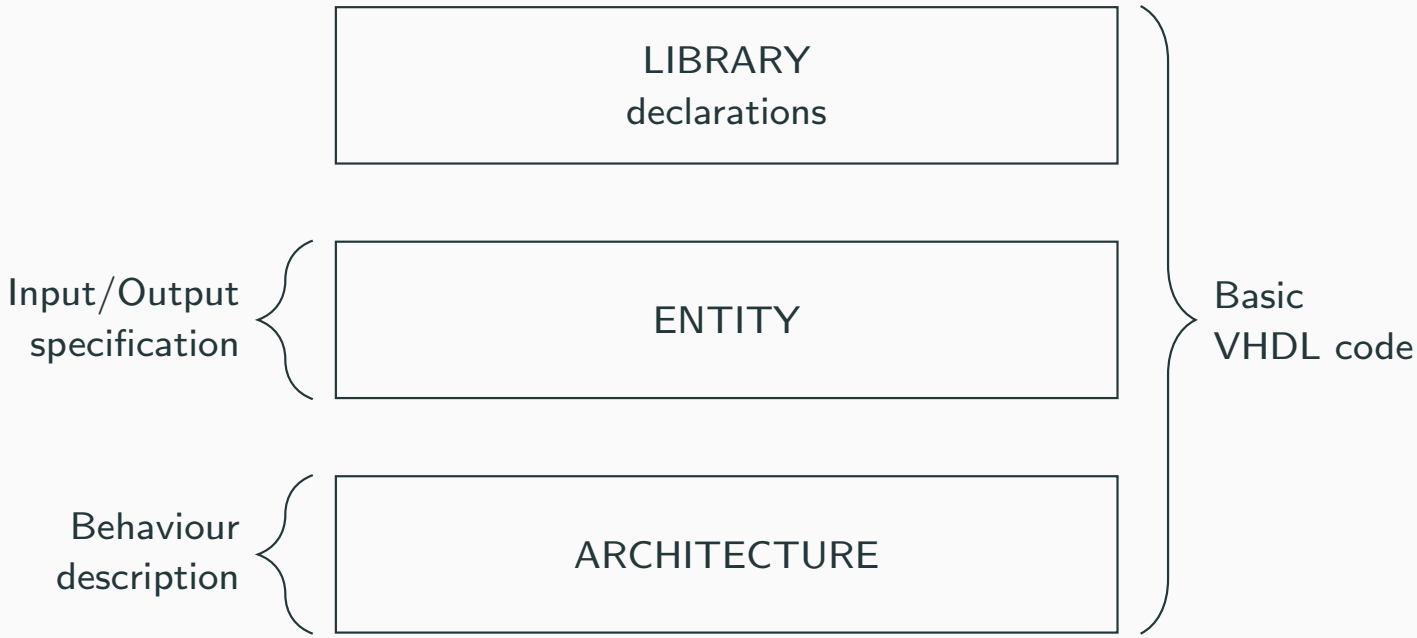
Design examples

- Adders
- Counters
- Comparators
- ALU
- MAC unit
- Decoder/Encoder
- RAM, ROM
- Digital filters
- State machine
- Microprocessor
- Neural network
- ...

VHDL

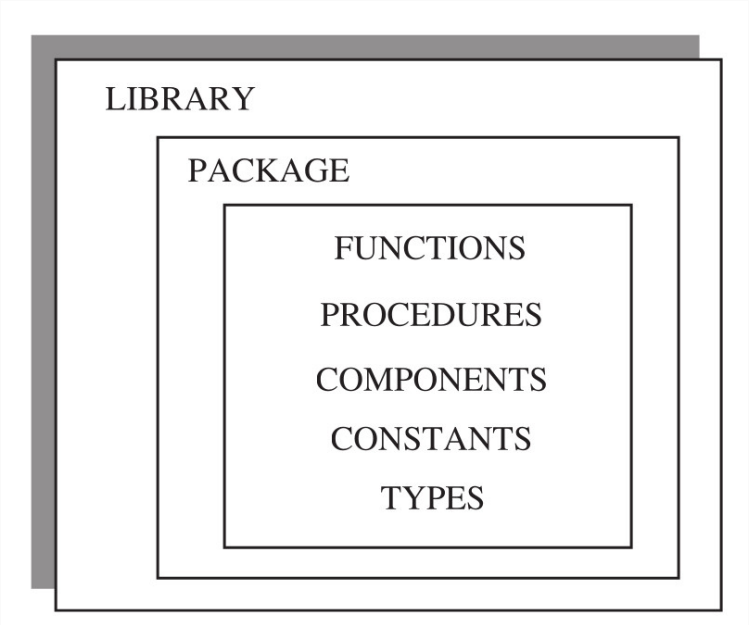
Code structure

Fundamental sections



Library declarations

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



Library declarations

Three packages are usually needed in a design:

```
LIBRARY ieee;                   -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all;    -- the end of a statement or

LIBRARY std;                    -- declaration, while a double
USE std.standard.all;           -- dash (--) indicates a comment.

LIBRARY work;
USE work.all;
```

Library declarations

Three packages are usually needed in a design:

```
LIBRARY ieee;                   -- A semi-colon (;) indicates
USE ieee.std_logic_1164.all;    -- the end of a statement or

LIBRARY std;                    -- declaration, while a double
USE std.standard.all;           -- dash (--) indicates a comment.

LIBRARY work;
USE work.all;
```

Their purposes:

- `ieee.std_logic_1164`: specifies the **STD_LOGIC** and **STD_ULOGIC** datatypes;
- `std`: resource library for the VHDL design environment (loaded by default);
- `work`: current working library (loaded by default).

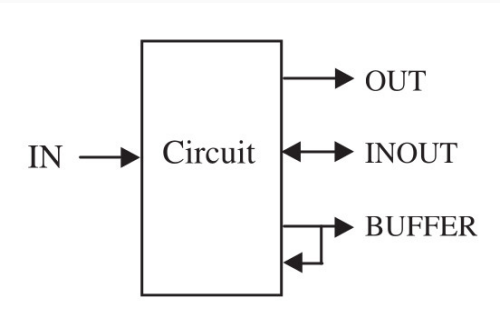
Entity

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

Entity

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

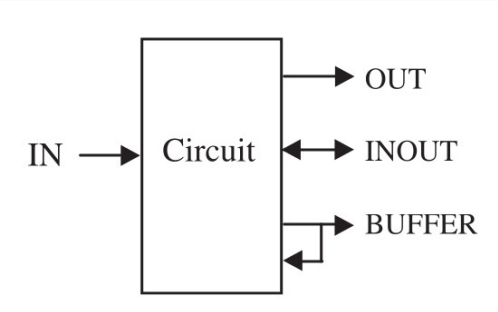
```
ENTITY entity_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```



Entity

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

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




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

Architecture

The **ARCHITECTURE** is a description of how the circuit should behave (function)

Architecture

The **ARCHITECTURE** is a description of how the circuit should behave (function)

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

Architecture

The **ARCHITECTURE** is a description of how the circuit should behave (function)

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

- *declarative* part: where signals and constants (among others) are declared

Architecture

The **ARCHITECTURE** is a description of how the circuit should behave (function)

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

- *declarative* part: where signals and constants (among others) are declared
- *code* part: behaviour is described

Architecture

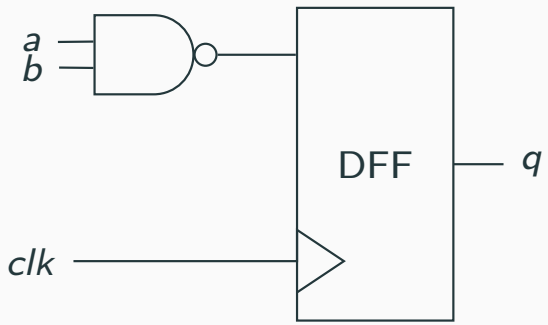
The **ARCHITECTURE** is a description of how the circuit should behave (function)

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

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

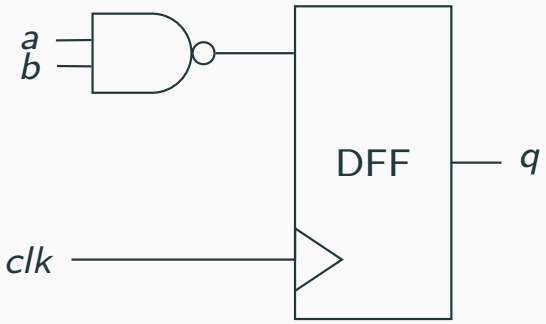
- *declarative* part: where signals and constants (among others) are declared
- *code* part: behaviour is described

Example




```
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;
```

Example

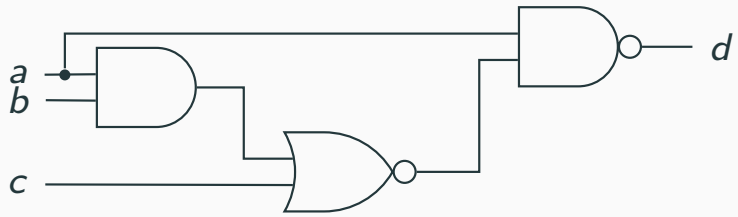



```
ENTITY example IS
   PORT (a, b, clk: IN BIT;
         q: OUT BIT);
END example;
---------------------------------------
ARCHITECTURE example OF example IS
   SIGNAL temp : BIT;
BEGIN
   temp <= a NAND b;
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN q<=temp;
      END IF;
   END PROCESS;
END example;
```

concurrent execution

Exercise

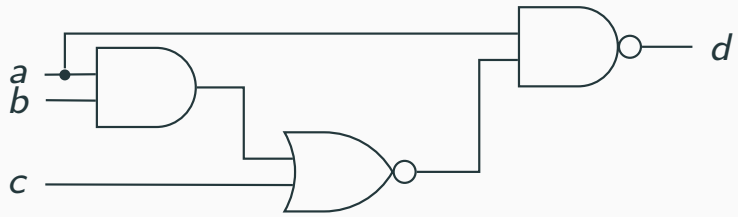
Write the VHDL code of the following circuit:



There are 3 inputs, and 1 output, and 2 internal signals (optionals).
You will only use `BIT` datatype. You will need to use some of the logical operators: `AND`, `OR`, `NAND`, `NOR`, `XOR`.

Exercise

Write the VHDL code of the following circuit:



There are 3 inputs, and 1 output, and 2 internal signals (optionals).
You will only use **BIT** datatype. You will need to use some of the logical operators: **AND**, **OR**, **NAND**, **NOR**, **XOR**.

```
ENTITY example IS
   PORT (a, b, c: IN BIT;
         d: OUT BIT);
END example;
---------------------------------------
ARCHITECTURE my_example OF example IS
   SIGNAL tmp1 : BIT;
   SIGNAL tmp2 : BIT;
BEGIN
   tmp1 <= a AND b;
   tmp2 <= tmp1 NOR c;
   d <= a NAND tmp2;
END my_example;
```

```
ENTITY example2 IS
   PORT (a, b, c: IN BIT;
         d: OUT BIT);
END example2;
---------------------------------------
ARCHITECTURE my_example2 OF example2 IS
BEGIN
   d <= ((a AND b) NOR c ) NAND a;
END my_example2;
```

VHDL

Data types

Pre-defined Data Types

- `std.standard`: Defines **`BIT`**, **`BOOLEAN`**, **`INTEGER`**, and **`REAL`** data types;

Pre-defined Data Types

- `std.standard`: Defines **BIT**, **BOOLEAN**, **INTEGER**, and **REAL** data types;
- `ieee.std_logic_1164`: Defines **STD_LOGIC** and **STD_ULOGIC** data types;

Pre-defined Data Types

- `std.standard`: Defines **`BIT`**, **`BOOLEAN`**, **`INTEGER`**, and **`REAL`** data types;
- `ieee.std_logic_1164`: Defines **`STD_LOGIC`** and **`STD_ULOGIC`** data types;
- `ieee.numeric_std`: Defines **`SIGNED`** and **`UNSIGNED`** data types, plus several data conversion functions, like `to_integer(p)`, `to_unsigned(p,b)`, `to_signed(p,b)`;

Pre-defined Data Types

- `std.standard`: Defines **BIT**, **BOOLEAN**, **INTEGER**, and **REAL** data types;
- `ieee.std_logic_1164`: Defines **STD_LOGIC** and **STD_ULOGIC** data types;
- `ieee.numeric_std`: Defines **SIGNED** and **UNSIGNED** data types, plus several data conversion functions, like `to_integer(p)`, `to_unsigned(p,b)`, `to_signed(p,b)`;
- `ieee.std_logic_signed` and `ieee.std_logic_unsigned`: Contain functions that allow operations with **STD_LOGIC_VECTOR** data to be performed as if the data were of type **SIGNED** or **UNSIGNED**, respectively.

BIT

Definition

`BIT` (and `BIT_VECTOR`): 2-level logic ('0', '1').

BIT

Definition

BIT (and **BIT_VECTOR**): 2-level logic ('0', '1').

Example

```
SIGNAL x: BIT; -- x is declared as a one-digit signal of type BIT.
SIGNAL y: BIT_VECTOR (3 DOWNTO 0); -- y is a 4-bit vector, with the leftmost bit being the MSB.
SIGNAL w: BIT_VECTOR (0 TO 7); -- w is an 8-bit vector, with the rightmost bit being the MSB.
```

```
x <= '1';
-- x is a single-bit signal (as specified above), whose value is '1'.
-- Notice that single quotes (' ') are used for a single bit.
y <= "0111";
-- y is a 4-bit signal (as specified above), whose value is "0111" (MSB='0').
-- Notice that double quotes (" ") are used for vectors.
w <= "01110001";
-- w is an 8-bit signal, whose value is "01110001" (MSB='1').
```

STD_LOGIC

Definition

STD_LOGIC (and **STD_LOGIC_VECTOR**): 8-valued logic system introduced in the IEEE 1164 standard.

'X'	Forcing Unknown	(synthesizable unknown)
'0'	Forcing Low	(synthesizable logic '0')
'1'	Forcing High	(synthesizable logic '1')
'Z'	High impedance	(synthesizable tri-state buffer)
'W'	Weak unknown	(can't tell if it should be 0 or 1)
'L'	Weak low	(that should probably go to 0)
'H'	Weak high	(that should probably go to 1)
'-'	Don't care	

STD_LOGIC

Definition

STD_LOGIC (and **STD_LOGIC_VECTOR**): 8-valued logic system introduced in the IEEE 1164 standard.

'X'	Forcing Unknown	(synthesizable unknown)
'0'	Forcing Low	(synthesizable logic '0')
'1'	Forcing High	(synthesizable logic '1')
'Z'	High impedance	(synthesizable tri-state buffer)
'W'	Weak unknown	(can't tell if it should be 0 or 1)
'L'	Weak low	(that should probably go to 0)
'H'	Weak high	(that should probably go to 1)
'-'	Don't care	

Example

```
SIGNAL x: STD_LOGIC;
-- x is declared as a one-digit (scalar) signal of type STD_LOGIC.
SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0001";
-- y is declared as a 4-bit vector, with the leftmost bit being the MSB.
-- The initial value (optional) of y is "0001".
-- Notice that the ":=" operator is used to establish the initial value.
```

STD_ULOGIC

Definition

`STD_ULOGIC` (and `STD_ULOGIC_VECTOR`): 9-level logic system introduced in the IEEE 1164 standard (‘`U`’, ‘`X`’, ‘`0`’, ‘`1`’, ‘`Z`’, ‘`W`’, ‘`L`’, ‘`H`’, ‘`-`’).

‘`U`’ stands for *Unresolved* $\Longrightarrow$ Driver conflicts

STD_ULOGIC

Definition

STD_ULOGIC (and STD_ULOGIC_VECTOR): 9-level logic system introduced in the IEEE 1164 standard ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-').

'U' stands for *Unresolved* $\Longrightarrow$ Driver conflicts

Resolved logic system

	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

High level Data Types

BOOLEAN
TRUE, FALSE.

High level Data Types

BOOLEAN
TRUE, FALSE.

INTEGER
32-bit integers (from -2,147,483,647 to +2,147,483,647).

High level Data Types

BOOLEAN
TRUE, FALSE.

INTEGER
32-bit integers (from -2,147,483,647 to +2,147,483,647).

NATURAL
Non-negative integers (from 0 to +2,147,483,647).

High level Data Types

BOOLEAN
TRUE, FALSE.

INTEGER
32-bit integers (from -2,147,483,647 to +2,147,483,647).

NATURAL
Non-negative integers (from 0 to +2,147,483,647).

REAL
Real numbers ranging from -1.0E38 to +1.0E38. Not synthesizable.

High level Data Types

BOOLEAN
TRUE, FALSE.

INTEGER
32-bit integers (from -2,147,483,647 to +2,147,483,647).

NATURAL
Non-negative integers (from 0 to +2,147,483,647).

REAL
Real numbers ranging from -1.0E38 to +1.0E38. Not synthesizable.

Physical literals
Used to inform physical quantities, like time, voltage, etc. Useful in simulations. Not synthesizable.

High level Data Types

BOOLEAN
TRUE, FALSE.

INTEGER
32-bit integers (from -2,147,483,647 to +2,147,483,647).

NATURAL
Non-negative integers (from 0 to +2,147,483,647).

REAL
Real numbers ranging from -1.0E38 to +1.0E38. Not synthesizable.

Physical literals
Used to inform physical quantities, like time, voltage, etc. Useful in simulations. Not synthesizable.

Character literals
Single ASCII character or a string of such characters. Not synthesizable.

High level Data Types

BOOLEAN
TRUE, FALSE.

INTEGER
32-bit integers (from -2,147,483,647 to +2,147,483,647).

NATURAL
Non-negative integers (from 0 to +2,147,483,647).

REAL
Real numbers ranging from -1.0E38 to +1.0E38. Not synthesizable.

Physical literals
Used to inform physical quantities, like time, voltage, etc. Useful in simulations. Not synthesizable.

Character literals
Single ASCII character or a string of such characters. Not synthesizable.

SIGNED and UNSIGNED
Data types defined in the `ieee.numeric_std` package. They have the appearance of **STD_LOGIC_VECTOR**, but accept arithmetic operations, which are typical of **INTEGER** data types.

High level Data Types

Example

```
x0 <= '0';            -- bit, std_logic, or std_ulogic value '0'
x1 <= "00011111";     -- bit_vector, std_logic_vector, std_ulogic_vector, signed, or unsigned
x2 <= "0001_1111";    -- underscore allowed to ease visualization
x3 <= "101111";       -- binary representation of decimal 47
x4 <= B"101111";      -- binary representation of decimal 47
x5 <= O"57";          -- octal representation of decimal 47
x6 <= X"2F";          -- hexadecimal representation of decimal 47
n <= 1200;            -- integer
m <= 1_200;           -- integer, underscore allowed
IF ready THEN...      -- Boolean, executed if ready=TRUE
y <= 1.2E-5;          -- real, not synthesizable
q <= d AFTER 10 NS;   -- physical, not synthesizable
```

High level Data Types

Example

```
x0 <= '0';            -- bit, std_logic, or std_ulogic value '0'
x1 <= "00011111";     -- bit_vector, std_logic_vector, std_ulogic_vector, signed, or unsigned
x2 <= "0001_1111";    -- underscore allowed to ease visualization
x3 <= "101111";       -- binary representation of decimal 47
x4 <= B"101111";      -- binary representation of decimal 47
x5 <= O"57";          -- octal representation of decimal 47
x6 <= X"2F";          -- hexadecimal representation of decimal 47
n <= 1200;            -- integer
m <= 1_200;           -- integer, underscore allowed
IF ready THEN...      -- Boolean, executed if ready=TRUE
y <= 1.2E-5;          -- real, not synthesizable
q <= d AFTER 10 NS;   -- physical, not synthesizable
```

Example

Operations between data of differents types:

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
-------------------------------------------------------------------
a <= b(5);  -- legal (same scalar type: BIT)
b(0) <= a;  -- legal (same scalar type: BIT)
c <= d(5);  -- legal (same scalar type: STD_LOGIC)
d(0) <= c;  -- legal (same scalar type: STD_LOGIC)
a <= c;     -- illegal (type mismatch: BIT x STD_LOGIC)
b <= d;     -- illegal (type mismatch: BIT_VECTOR x STD_LOGIC_VECTOR)
e <= b;     -- illegal (type mismatch: INTEGER x BIT_VECTOR)
e <= d;     -- illegal (type mismatch: INTEGER x STD_LOGIC_VECTOR)
```

User-Defined Data Types

Integer

```
TYPE integer IS RANGE -2147483647 TO +2147483647;   -- This is indeed the pre-defined type INTEGER.
TYPE natural IS RANGE 0 TO +2147483647;             -- This is indeed the pre-defined type NATURAL.
TYPE my_integer IS RANGE -32 TO 32;                 -- A user-defined subset of integers.
TYPE student_grade IS RANGE 0 TO 100;               -- A user-defined subset of integers or naturals.
```

User-Defined Data Types

Integer

```
TYPE integer IS RANGE -2147483647 TO +2147483647;  -- This is indeed the pre-defined type INTEGER.
TYPE natural IS RANGE 0 TO +2147483647;            -- This is indeed the pre-defined type NATURAL.
TYPE my_integer IS RANGE -32 TO 32;                -- A user-defined subset of integers.
TYPE student_grade IS RANGE 0 TO 100;              -- A user-defined subset of integers or naturals.
```

Enumerated

```
TYPE bit IS ('0', '1');                            -- This is indeed the pre-defined type BIT
TYPE my_logic IS ('0', '1', 'Z');                  -- A user-defined subset of std_logic.
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;-- This is indeed the pre-defined type BIT_VECTOR.
                                                   -- RANGE <> is used to indicate that the range is
                                                   -- unconstrained.
                                                   -- NATURAL RANGE <>, on the other hand, indicates
                                                   -- that the only restriction is that the range must
                                                   -- fall within the NATURAL range.
TYPE state IS (idle, forward, backward, stop);     -- An enumerated data type,
                                                   -- typical of finite state machines.
TYPE color IS (red, green, blue, white);           -- Another enumerated data type.
```

SIGNED and UNSIGNED

From the package `ieee.numeric_std`

SIGNED and UNSIGNED

From the package `ieee.numeric_std`

Example

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

SIGNED and UNSIGNED

From the package `ieee.numeric_std`

Example

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

Similar to **STD_LOGIC_VECTOR**, and not like **INTEGER**

SIGNED and UNSIGNED

From the package `ieee.numeric_std`

Example

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

Similar to **STD_LOGIC_VECTOR**, and not like **INTEGER**

For *arithmetic operations* purpose

SIGNED and UNSIGNED

From the package `ieee.numeric_std`

Example

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

Similar to **STD_LOGIC_VECTOR**, and not like **INTEGER**

For *arithmetic operations* purpose

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
-- extra package necessary
...
SIGNAL a: SIGNED (7 DOWNTO 0);
SIGNAL b: SIGNED (7 DOWNTO 0);
SIGNAL v: SIGNED (7 DOWNTO 0);
SIGNAL w: SIGNED (7 DOWNTO 0);
...
v <= a + b; --legal (arithmetic operation OK)
w <= a AND b; --illegal (logical operation not OK)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- no extra package required
...
SIGNAL a: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL v: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL w: STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b; --illegal (arithmetic operation not OK)
w <= a AND b; --legal (logical operation OK)
```

SIGNED and UNSIGNED

From the package `ieee.numeric_std`

Example

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

Similar to **STD_LOGIC_VECTOR**, and not like **INTEGER**

For *arithmetic operations* purpose

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
-- extra package necessary
...
SIGNAL a: SIGNED (7 DOWNTO 0);
SIGNAL b: SIGNED (7 DOWNTO 0);
SIGNAL v: SIGNED (7 DOWNTO 0);
SIGNAL w: SIGNED (7 DOWNTO 0);
...
v <= a + b; --legal (arithmetic operation OK)
w <= a AND b; --illegal (logical operation not OK)
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- no extra package required
...
SIGNAL a: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL v: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL w: STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b; --illegal (arithmetic operation not OK)
w <= a AND b; --legal (logical operation OK)
```

Possible to use packages `std_logic_signed` and `std_logic_unsigned` (from the `ieee` library) to do arithmetic operations on **STD_LOGIC_VECTOR**.

Data conversion

Several data conversion functions and type-casting in `ieee.numeric_std`:

function	type of p	b
`to_integer(p)`	**INTEGER**, **UNSIGNED**, **SIGNED**	
`to_unsigned(p,b)`	**INTEGER**	size (bits)
`to_signed(p,b)`	**INTEGER**	size (bits)
`signed(p)(*)`	**STD_LOGIC_VECTOR**	
`unsigned(p)(*)`	**STD_LOGIC_VECTOR**	
`std_logic_vector(p)(*)`	**SIGNED**, **UNSIGNED**	

Data conversion

Several data conversion functions and type-casting in `ieee.numeric_std`:

function	type of p	b
to_integer(p)	INTEGER, UNSIGNED, SIGNED	
to_unsigned(p,b)	INTEGER	size (bits)
to_signed(p,b)	INTEGER	size (bits)
signed(p)(*)	STD_LOGIC_VECTOR	
unsigned(p)(*)	STD_LOGIC_VECTOR	
std_logic_vector(p)(*)	SIGNED, UNSIGNED	

Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
...
SIGNAL a: UNSIGNED (7 DOWNTO 0);
SIGNAL b: UNSIGNED (7 DOWNTO 0);
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
...
y <= STD_LOGIC_VECTOR ((a+b), 8);
-- Legal operation: a+b is converted from UNSIGNED to an 8-bit STD_LOGIC_VECTOR value,
-- then assigned to y.
```

VHDL

Operators

Type of Operators

- Assignment operators
- Logical operators
- Arithmetic operators
- Relational operators
- Shift operators
- Concatenation operators

Assignment operators

<=	Used to assign a value to a **SIGNAL**.
:=	Used to assign a value to a **VARIABLE**, **CONSTANT**, or **GENERIC**. Used also for establishing initial values.
=>	Used to assign values to individual vector elements or with **OTHERS**.

Assignment operators

<=	Used to assign a value to a **SIGNAL**.
:=	Used to assign a value to a **VARIABLE**, **CONSTANT**, or **GENERIC**. Used also for establishing initial values.
=>	Used to assign values to individual vector elements or with **OTHERS**.

Example

```
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0);    -- Leftmost bit is MSB
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7);           -- Rightmost bit is
...
x <= '1';                                     -- '1' is assigned to SIGNAL x using "<="
y := "0000";                                  -- "0000" is assigned to VARIABLE y using ":="
w <= "10000000";                              -- LSB is '1', the others are '0'
w <= (0 =>'1', OTHERS =>'0');                 -- LSB is '1', the others are '0'
```

Logical operators

```
NOT
AND
OR
NAND
NOR
XOR
XNOR
```

The data must be of type **BIT**, **STD_LOGIC**, or **STD_ULOGIC** (or, their respective extensions, **BIT_VECTOR**, **STD_LOGIC_VECTOR**, or **STD_ULOGIC_VECTOR**).

Logical operators

NOT
AND
OR
NAND
NOR
XOR
XNOR

The data must be of type **BIT**, **STD_LOGIC**, or **STD_ULOGIC** (or, their respective extensions, **BIT_VECTOR**, **STD_LOGIC_VECTOR**, or **STD_ULOGIC_VECTOR**).

Example

```
y <= NOT a AND b;
y <= NOT (a AND b);
y <= a NAND b;
```

Arithmetic operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
MOD	Modulus
REM	Remainder
ABS	Absolute value

The data can be of type **INTEGER**, **SIGNED**, **UNSIGNED**, or **REAL**.

Also, if the `std_logic_signed` or the `std_logic_unsigned` package of the `ieee` library is used, then **STD_LOGIC_VECTOR** can also be employed directly in addition and subtraction operations.

Arithmetic operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
MOD	Modulus
REM	Remainder
ABS	Absolute value

The data can be of type **INTEGER**, **SIGNED**, **UNSIGNED**, or **REAL**.

Also, if the `std_logic_signed` or the `std_logic_unsigned` package of the `ieee` library is used, then **STD_LOGIC_VECTOR** can also be employed directly in addition and subtraction operations.

Warning
Must be carefull about the last five, the synthesis (if it is possible) may not be what you expect !

Comparison operators

=	Equal to
/=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

The data can be of any type listed above.

Shift operators

`sll`	Shift left logic	positions on the right are filled with '0's
`srl`	Shift right logic	positions on the left are filled with '0's

The left operand must be of type **BIT_VECTOR**, while the right operand must be an **INTEGER**.

Shift operators

sll	Shift left logic	positions on the right are filled with '0's
srl	Shift right logic	positions on the left are filled with '0's

The left operand must be of type **BIT_VECTOR**, while the right operand must be an **INTEGER**.

Example

```
SIGNAL b : BIT_VECTOR (3 DOWNTO 0) := "1100";
SIGNAL x : BIT_VECTOR (3 DOWNTO 0);
SIGNAL y : BIT_VECTOR (3 DOWNTO 0);
...
x <= b sll 2;  -- x <= "0000"
y <= b srl 1;  -- y <= "0110"
```

Concatenation operator

The concatenation operator is denoted &.

The data can be of type `BIT`, `STD_LOGIC`, or `STD_ULOGIC` (or, their respective extensions, `BIT_VECTOR`, `STD_LOGIC_VECTOR`, or `STD_ULOGIC_VECTOR`).

Concatenation operator

The concatenation operator is denoted &.

The data can be of type **BIT**, **STD_LOGIC**, or **STD_ULOGIC** (or, their respective extensions, **BIT_VECTOR**, **STD_LOGIC_VECTOR**, or **STD_ULOGIC_VECTOR**).

Example

```
SIGNAL a : BIT_VECTOR (3 DOWNTO 0) := "1001";
SIGNAL b : BIT_VECTOR (3 DOWNTO 0) := "1100";
SIGNAL x : BIT_VECTOR (7 DOWNTO 0);
...
x <= a & b; -- x <= "10011100"
```

Signal Attributes

Their syntax is the following: `signal_name'attribute_name`

`s'EVENT`	Returns true when an event occurs on s
`s'STABLE`	Returns true if no event has occurred on s
`s'ACTIVE`	Returns true if s='1'
`s'LAST_EVENT`	Returns the time elapsed since last event
`s'LAST_ACTIVE`	Returns the time elapsed since last s='1'
`s'LAST_VALUE`	Returns the value of s before the last event

Signal Attributes

Their syntax is the following: `signal_name'attribute_name`

`s'EVENT`	Returns true when an event occurs on s
`s'STABLE`	Returns true if no event has occurred on s
`s'ACTIVE`	Returns true if s='1'
`s'LAST_EVENT`	Returns the time elapsed since last event
`s'LAST_ACTIVE`	Returns the time elapsed since last s='1'
`s'LAST_VALUE`	Returns the value of s before the last event

Example

```
IF (clk'EVENT AND clk='1')...     -- EVENT attribute used with IF
IF (NOT clk'STABLE AND clk='1')...  -- STABLE attribute used with IF
WAIT UNTIL (clk'EVENT AND clk='1'); -- EVENT attribute used with wait
```

To go further

- There exists more operators (`sla`, `sra`, `rol`, ...);

To go further

- There exists more operators (`sla`, `sra`, `rol`, ...);
- There exists more attributes (`LOW`, `HIGH`, `LEFT`, ...);

To go further

- There exists more operators (`sla`, `sra`, `rol`, ...);
- There exists more attributes (`LOW`, `HIGH`, `LEFT`, ...);
- You can define your own attribute;

To go further

- There exists more operators (`sla`, `sra`, `rol`, ...);
- There exists more attributes (`LOW`, `HIGH`, `LEFT`, ...);
- You can define your own attribute;
- You can overload the existing operators.

Exercise

Determine the values of the `xi` and `d`.

```
SIGNAL a : BIT := '1';
SIGNAL b : BIT_VECTOR (3 DOWNTO 0) := "1100";
SIGNAL c : BIT_VECTOR (3 DOWNTO 0) := "0010";
SIGNAL d : BIT_VECTOR (7 DOWNTO 0);
...
x1 <= a & c;
x2 <= c & b;
x3 <= b XOR c;
x4 <= a NOR b(3);
x5 <= b sll 2;
x6 <= b srl 1;
x7 <= a AND NOT b(0) AND NOT c(1);
d <= (5=>'0', OTHERS=>'1');
```

Exercise

Determine the values of the `xi` and `d`.

```
SIGNAL a : BIT := '1';
SIGNAL b : BIT_VECTOR (3 DOWNTO 0) := "1100";
SIGNAL c : BIT_VECTOR (3 DOWNTO 0) := "0010";
SIGNAL d : BIT_VECTOR (7 DOWNTO 0);
...
x1 <= a & c;
x2 <= c & b;
x3 <= b XOR c;
x4 <= a NOR b(3);
x5 <= b sll 2;
x6 <= b srl 1;
x7 <= a AND NOT b(0) AND NOT c(1);
d <= (5=>'0', OTHERS=>'1');
```

Solution

```
x1 <= "10010";
x2 <= "00101100";
x3 <= "1110";
x4 <= '0';
x5 <= "0000";
x6 <= "0110";
x7 <= '0';
d <= "11011111";
```

VHDL

Concurrent code

Concurrent code

VHDL code is inherently *concurrent* (parallel).

Concurrent code

VHDL code is inherently *concurrent* (parallel).

The order of the statements does not matter.

```
ENTITY example IS
   PORT (...);
END example;
---------------------------------------
ARCHITECTURE example OF example IS
   ...
BEGIN
   statement 1;
   statement 2;
   statement 3;
END example;
```

≡

```
ENTITY example IS
   PORT (...);
END example;
---------------------------------------
ARCHITECTURE example OF example IS
   ...
BEGIN
   statement 2;
   statement 3;
   statement 1;
END example;
```

Concurrent code

VHDL code is inherently *concurrent* (parallel).

The order of the statements does not matter.

```
ENTITY example IS
   PORT (...);
END example;
-------------------------------------
ARCHITECTURE example OF example IS
   ...
BEGIN
   statement 1;
   statement 2;
   statement 3;
END example;
```

≡

```
ENTITY example IS
   PORT (...);
END example;
-------------------------------------
ARCHITECTURE example OF example IS
   ...
BEGIN
   statement 2;
   statement 3;
   statement 1;
END example;
```

Perfect to build *combinatonial* logic circuits.

Concurrent code

VHDL code is inherently *concurrent* (parallel).

The order of the statements does not matter.

```
ENTITY example IS
   PORT (...);
END example;
-------------------------------------
ARCHITECTURE example OF example IS
   ...
BEGIN
   statement 1;
   statement 2;
   statement 3;
END example;
```

≡

```
ENTITY example IS
   PORT (...);
END example;
-------------------------------------
ARCHITECTURE example OF example IS
   ...
BEGIN
   statement 2;
   statement 3;
   statement 1;
END example;
```

Perfect to build *combinatonial* logic circuits.

Concurrents statements

- Operators;
- The **WHEN** statement (**WHEN/ELSE** or **WITH/SELECT/WHEN**);
- The **GENERATE** statement;
- The **BLOCK** statement.

WHEN statement

It is the *conditional* concurrent statement.

WHEN statement

It is the *conditional* concurrent statement.

It appears in two forms:

WHEN/ELSE (simple WHEN)

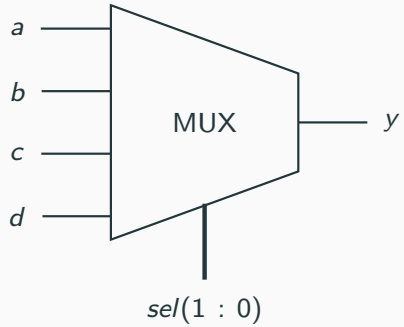
```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WITH/SELECT/WHEN (selected WHEN)

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

WHEN statement

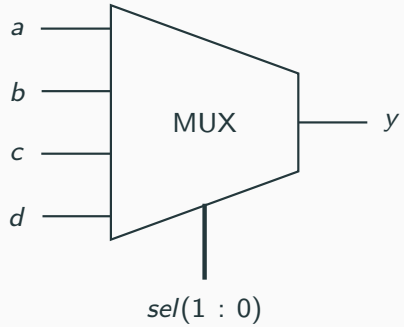
Example
An example of Multiplexer



sel	*y*
00	*a*
01	*b*
10	*c*
11	*d*

WHEN statement

Example
An example of Multiplexer



sel	*y*
00	*a*
01	*b*
10	*c*
11	*d*

```
---------- Solution 1: with WHEN/ELSE ------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
      sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
      y: OUT STD_LOGIC);
END mux;
-------------------------------------------------
ARCHITECTURE mux1 OF mux IS
BEGIN
   y <= a WHEN sel="00" ELSE
        b WHEN sel="01" ELSE
        c WHEN sel="10" ELSE
        d;
END mux1;
-------------------------------------------------
```

```
-------- Solution 2: with WITH/SELECT/WHEN --------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY mux IS
   PORT ( a, b, c, d: IN STD_LOGIC;
      sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
      y: OUT STD_LOGIC);
END mux;
-------------------------------------------------
ARCHITECTURE mux2 OF mux IS
BEGIN
   WITH sel SELECT
      y <= a WHEN "00",-- notice "," instead of ";"
           b WHEN "01",
           c WHEN "10",
           d WHEN OTHERS;-- cannot be "d WHEN "11"
END mux2;
-------------------------------------------------
```

GENERATE statement

It is the *loop* concurrent statement.

FOR/GENERATE

```
label: FOR identifier IN range GENERATE
   (concurrent assignments)
END GENERATE;
```

GENERATE statement

It is the *loop* concurrent statement.

FOR/GENERATE

```
label: FOR identifier IN range GENERATE
   (concurrent assignments)
END GENERATE;
```

Example

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (15 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE
   z(i) <= x(i) AND y(i+8);
END GENERATE;

G2: FOR i IN 0 TO 7 GENERATE
   z(i+8) <= x(i) OR y(i);
END GENERATE;
```

GENERATE statement

It is the *loop* concurrent statement.

FOR/GENERATE

```
label: FOR identifier IN range GENERATE
   (concurrent assignments)
END GENERATE;
```

Example

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (15 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE
   z(i) <= x(i) AND y(i+8);
END GENERATE;

G2: FOR i IN 0 TO 7 GENERATE
   z(i+8) <= x(i) OR y(i);
END GENERATE;
```

Warning

- Limits of the range must be static;
- No multiply-driven signals allowed.

Exercise

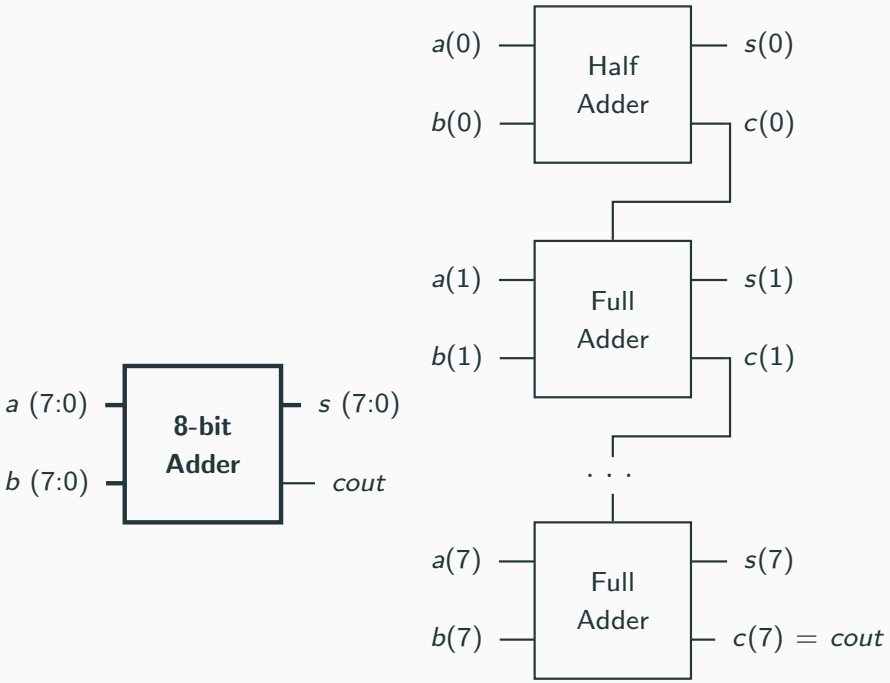
Build an 8-bit Adder using only logical operations.

Exercise

Build an 8-bit Adder using only logical operations.

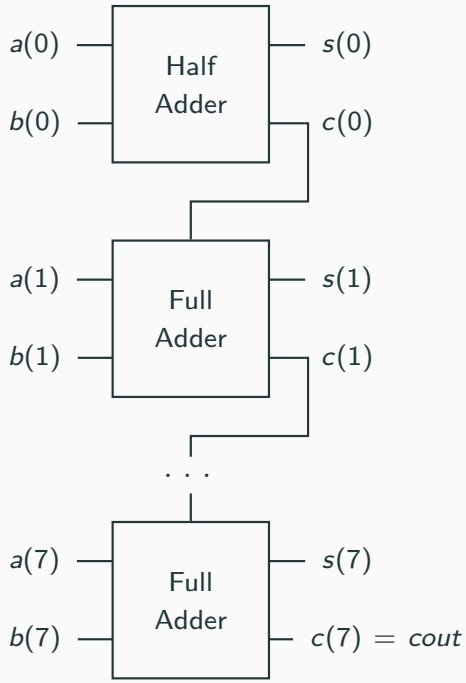
Exercise

Build an 8-bit Adder using only logical operations.



Exercise

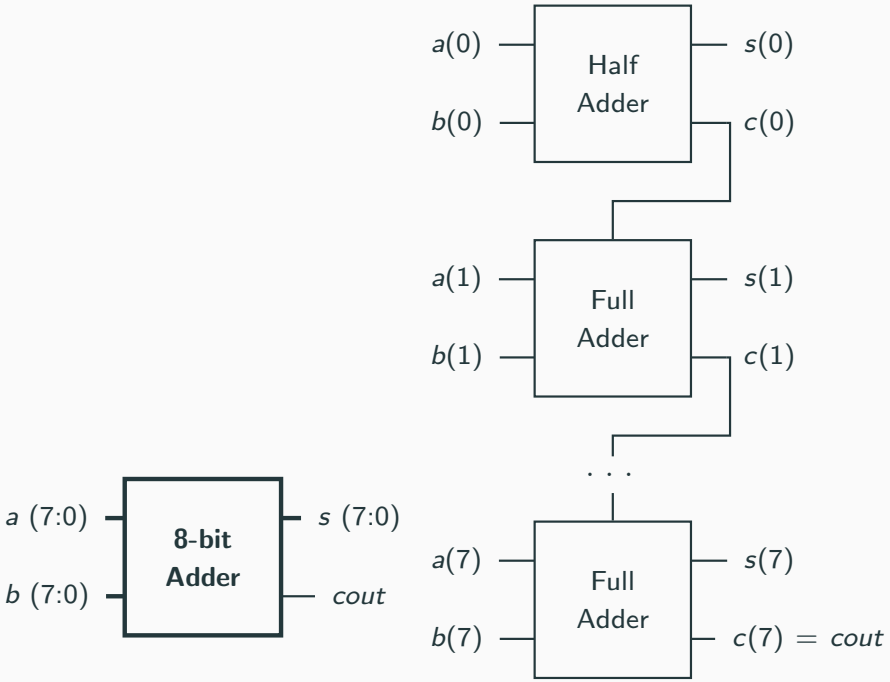
Build an 8-bit Adder using only logical operations.



$$s(0) = a(0) \oplus b(0)$$
$$c(0) = a(0).b(0)$$

Exercise

Build an 8-bit Adder using only logical operations.



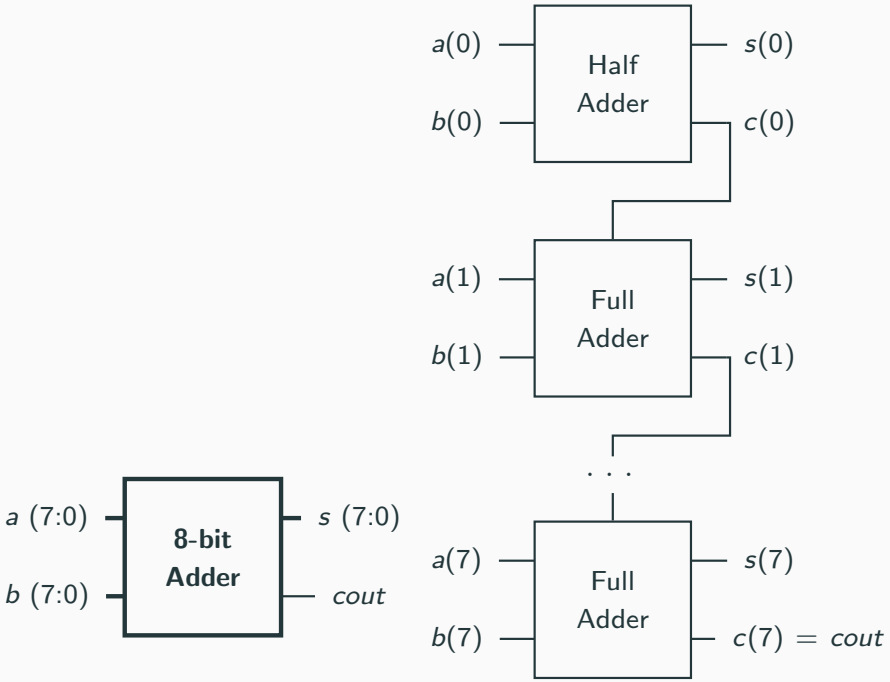
$$s(0) = a(0) \oplus b(0)$$
$$c(0) = a(0).b(0)$$

$$s(1) = a(1) \oplus b(1) \oplus c(0)$$
$$c(1) = a(1).b(1) + a(1).c(0) + b(1).c(0)$$

...

Exercise

Build an 8-bit Adder using only logical operations.



$$s(0) = a(0) \oplus b(0)$$
$$c(0) = a(0).b(0)$$

$$s(1) = a(1) \oplus b(1) \oplus c(0)$$
$$c(1) = a(1).b(1) + a(1).c(0) + b(1).c(0)$$

...

```
ENTITY adder_8_bits IS
PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      s: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
      cout: OUT STD_LOGIC);
END adder_8_bits;
-------------------------------------
ARCHITECTURE my_adder OF adder_8_bits IS
   SIGNAL c : STD_LOGIC_VECTOR (7 DOWNTO 0);
   SIGNAL i : INTEGER RANGE 0 TO 7;
BEGIN
   -- Half Adder on bit 0
   s(0) <= a(0) XOR b(0);
   c(0) <= a(0) AND b(0);
   -- Generate all the Full Adders
   G1: FOR i IN 1 TO 7 GENERATE
      -- Full Adder on bit i
      s(i) <= a(i) XOR b(i) XOR c(i-1);
      c(i) <= (a(i) AND b(i)) OR (a(i) AND c(i-1))
            OR (b(i) AND c(i-1));
   END GENERATE;
   -- Last carry is cout
   cout <= c(7);
END my_adder;
```

VHDL

Sequential code

Sequential code

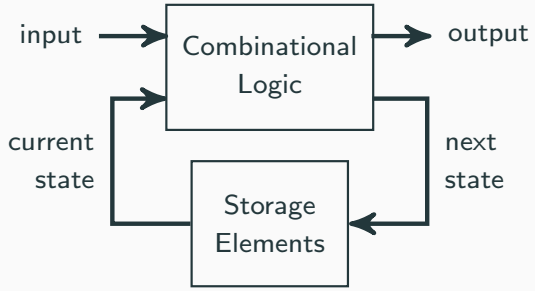
When the output depends on *previous* inputs.

Sequential code

When the output depends on *previous* inputs.

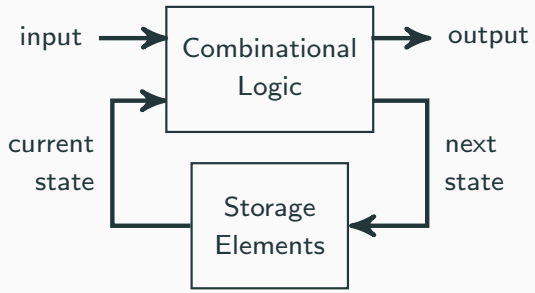
Sequential code

When the output depends on *previous* inputs.



Sequential code

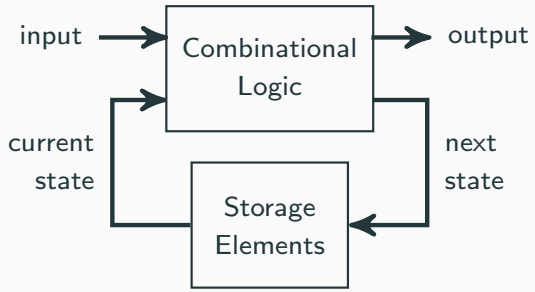
When the output depends on *previous* inputs.



In *sequential* code, the statements are executed *in order*.

Sequential code

When the output depends on *previous* inputs.



In *sequential* code, the statements are executed *in order*.

In VHDL, the only code executed *sequentially* is in the sections **PROCESS**, **FUNCTION**, or **PROCEDURE**.

PROCESS

The section **PROCESS** has the following syntax:

```
[label:] PROCESS (sensitivity list)
   [VARIABLE name type [range] [:= initial_value;]]
BEGIN
   (sequential code)
END PROCESS [label];
```

PROCESS

The section **PROCESS** has the following syntax:

```
[label:] PROCESS (sensitivity list)
   [VARIABLE name type [range] [:= initial_value;]]
BEGIN
   (sequential code)
END PROCESS [label];
```

VARIABLE

VARIABLE are the equivalent of **SIGNAL** for a **PROCESS**. But there are some difference:

- They can only be used inside a **PROCESS**, **FUNCTION**, or **PROCEDURE**;
- They are defined locally;
- The assignement operator is `:=` .

PROCESS

The section **PROCESS** has the following syntax:

```
[label:] PROCESS (sensitivity list)
   [VARIABLE name type [range] [:= initial_value;]]
BEGIN
   (sequential code)
END PROCESS [label];
```

VARIABLE

VARIABLE are the equivalent of **SIGNAL** for a **PROCESS**. But there are some difference:

- They can only be used inside a **PROCESS**, **FUNCTION**, or **PROCEDURE**;
- They are defined locally;
- The assignement operator is `:=` .

Sequential statements

- The **IF** statement;
- The **WAIT** statement;
- The **CASE** statement;
- The **LOOP** statement.

IF statement

It is a *conditional* sequential statement.

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

IF statement

It is a *conditional* sequential statement.

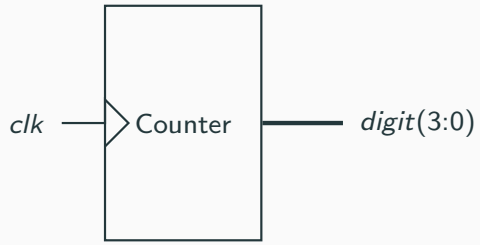
```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

Example

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

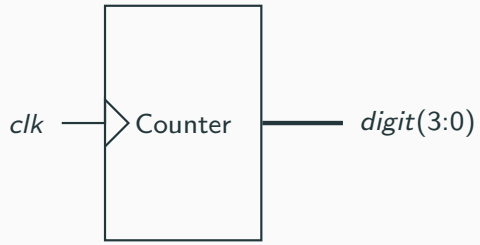
IF statement

Example
A 1-digit counter (from 0 to 9)



IF statement

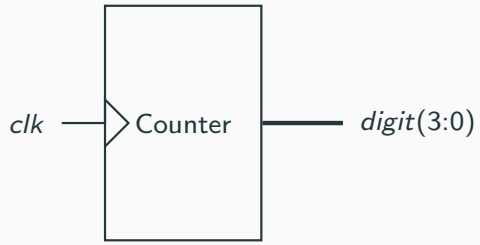
Example
A 1-digit counter (from 0 to 9)




```
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY counter IS
   PORT (clk : IN STD_LOGIC;
      digit : OUT INTEGER RANGE 0 TO 9);
END counter;
-----------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   count: PROCESS(clk)
      VARIABLE temp : INTEGER RANGE 0 TO 10;
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         temp := temp + 1;
         IF (temp=10) THEN temp := 0;
         END IF;
      END IF;
      digit <= temp;
   END PROCESS count;
END counter;
-----------------------------------------------
```

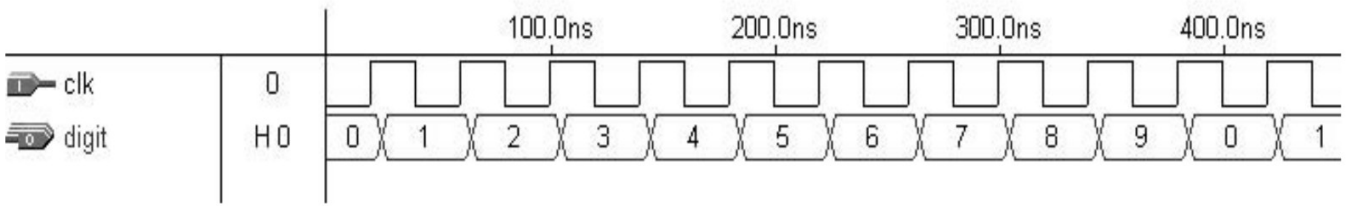
IF statement

Example
A 1-digit counter (from 0 to 9)




```
-----------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY counter IS
   PORT (clk : IN STD_LOGIC;
      digit : OUT INTEGER RANGE 0 TO 9);
END counter;
-----------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   count: PROCESS(clk)
      VARIABLE temp : INTEGER RANGE 0 TO 10;
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         temp := temp + 1;
         IF (temp=10) THEN temp := 0;
         END IF;
      END IF;
      digit <= temp;
   END PROCESS count;
END counter;
-----------------------------------------------
```

Result



CASE statement

It is a *conditional* sequential statement, very similar to the **WHEN** statement (concurrent equivalent).

```
CASE identifier IS
WHEN value => assignments;
WHEN value => assignments;
...
END CASE;
```

CASE statement

It is a *conditional* sequential statement, very similar to the **WHEN** statement (concurrent equivalent).

```
CASE identifier IS
WHEN value => assignments;
WHEN value => assignments;
...
END CASE;
```

Example

```
CASE control IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=c;
   WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

CASE statement

It is a *conditional* sequential statement, very similar to the **WHEN** statement (concurrent equivalent).

```
CASE identifier IS
WHEN value => assignments;
WHEN value => assignments;
...
END CASE;
```

Example

```
CASE control IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=c;
   WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

- Another important keyword is **NULL**: used when no action is to take place;

CASE statement

It is a *conditional* sequential statement, very similar to the **WHEN** statement (concurrent equivalent).

```
CASE identifier IS
WHEN value => assignments;
WHEN value => assignments;
...
END CASE;
```

Example

```
CASE control IS
   WHEN "00" => x<=a; y<=b;
   WHEN "01" => x<=b; y<=c;
   WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

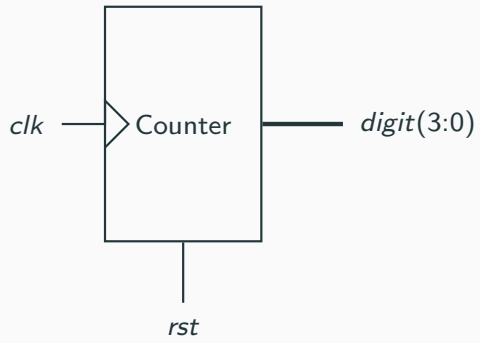
- Another important keyword is **NULL**: used when no action is to take place;
- **CASE** allows multiple assignments for each test condition, while **WHEN** allows only one.

CASE statement

Example

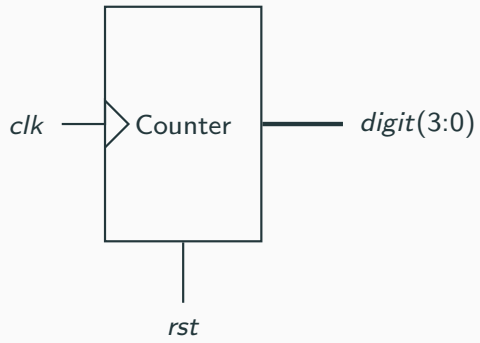
A 1-digit counter (from 0 to 9)

with a reset



CASE statement

Example
A 1-digit counter (from 0 to 9) with a reset




```
-------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY counter IS
   PORT (clk, rst : IN STD_LOGIC;
      digit : OUT INTEGER RANGE 0 TO 9);
END counter;
-------------------------------------------
ARCHITECTURE counter OF counter IS
BEGIN
   count: PROCESS(clk, rst)
      VARIABLE temp : INTEGER RANGE 0 TO 10;
   BEGIN
      CASE rst IS
         WHEN '1' => temp :=0;
         WHEN '0' =>
            IF (clk'EVENT AND clk='1') THEN
               temp := temp + 1;
               IF (temp=10) THEN temp := 0;
               END IF;
            END IF;
         WHEN OTHERS => NULL;
      digit <= temp;
   END PROCESS count;
END counter;
-------------------------------------------
```

WAIT statement

It is a sequential statement to elapse time.

It appears in three forms:

```
WAIT UNTIL signal_condition;
```

```
WAIT ON signal1 [, signal2, ... ];
```

```
WAIT FOR time;
```

WAIT statement

It is a sequential statement to elapse time.

It appears in three forms:

```
WAIT UNTIL signal_condition;
```

```
WAIT ON signal1 [, signal2, ... ];
```

```
WAIT FOR time;
```

Example

```
PROCESS -- no sensitivity list
BEGIN
   WAIT UNTIL (clk'EVENT AND clk='1');
   IF (rst='1') THEN
      x <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      x <= a;
   END IF;
END PROCESS;
```

```
PROCESS -- no sensitivity list
BEGIN
   WAIT ON clk, rst;
   IF (rst='1') THEN
      output <= "00000000";
   ELSIF (clk'EVENT AND clk='1') THEN
      output <= input;
   END IF;
END PROCESS;
```

For simulation only:

```
WAIT FOR 5NS;
```

LOOP statement

It is the *loop* sequential statement.

It appears in two forms:

```
[label:] FOR identifier IN range LOOP
   (sequential statements)
END LOOP [label];
```

```
[label:] WHILE condition LOOP
   (sequential statements)
END LOOP [label];
```

LOOP statement

It is the *loop* sequential statement.

It appears in two forms:

```
[label:] FOR identifier IN range LOOP
   (sequential statements)
END LOOP [label];
```

```
[label:] WHILE condition LOOP
   (sequential statements)
END LOOP [label];
```

Example

FOR/LOOP

```
FOR i IN 0 TO 5 LOOP
   x(i) <= enable AND w(i+2);
   y(0, i) <= w(i);
END LOOP;
```

WHILE/LOOP

```
WHILE (i < 10) LOOP
   WAIT UNTIL clk'EVENT AND clk='1';
   i:=i+1;
   (other statements)
END LOOP;
```

LOOP statement

It is the *loop* sequential statement.

It appears in two forms:

```
[label:] FOR identifier IN range LOOP
   (sequential statements)
END LOOP [label];
```

```
[label:] WHILE condition LOOP
   (sequential statements)
END LOOP [label];
```

Example

FOR/LOOP

```
FOR i IN 0 TO 5 LOOP
   x(i) <= enable AND w(i+2);
   y(0, i) <= w(i);
END LOOP;
```

WHILE/LOOP

```
WHILE (i < 10) LOOP
   WAIT UNTIL clk'EVENT AND clk='1';
   i:=i+1;
   (other statements)
END LOOP;
```

- Limits of the range of **FOR/LOOP** must be static;

LOOP statement

It is the *loop* sequential statement.

It appears in two forms:

```
[label:] FOR identifier IN range LOOP
   (sequential statements)
END LOOP [label];
```

```
[label:] WHILE condition LOOP
   (sequential statements)
END LOOP [label];
```

Example

FOR/LOOP

```
FOR i IN 0 TO 5 LOOP
   x(i) <= enable AND w(i+2);
   y(0, i) <= w(i);
END LOOP;
```

WHILE/LOOP

```
WHILE (i < 10) LOOP
   WAIT UNTIL clk'EVENT AND clk='1';
   i:=i+1;
   (other statements)
END LOOP;
```

- Limits of the range of **FOR/LOOP** must be static;
- There exists a statement to exit the loop (**EXIT**), and a statement to skip loop steps (**NEXT**).

VHDL

Composition

Composition

Composition

Idea
Compose basic “brick” of code in order to build bigger system.

Composition

Idea
Compose basic "brick" of code in order to build bigger system.

Benefits

- Reusability of code;
- More understandable code;
- Modularity;

Composition

Idea
Compose basic "brick" of code in order to build bigger system.

Benefits

- Reusability of code;
- More understandable code;
- Modularity;

How to do it in VHDL

- `COMPONENT`;
- `FUNCTION`;
- `PROCEDURE`.

COMPONENT

COMPONENT

Definition

COMPONENT = *conventional* code (**LIBRARY** declarations + **ENTITY** + **ARCHITECTURE**).

Declare a **COMPONENT** make it usable within another circuit, thus allowing the construction of *hierarchical* designs.

COMPONENT

Definition

COMPONENT = *conventional* code (**LIBRARY** declarations + **ENTITY** + **ARCHITECTURE**).

Declare a **COMPONENT** make it usable within another circuit, thus allowing the construction of *hierarchical* designs.

Declaration

```
COMPONENT component_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END COMPONENT;
```

COMPONENT

Definition

COMPONENT = *conventional* code (**LIBRARY** declarations + **ENTITY** + **ARCHITECTURE**).

Declare a **COMPONENT** make it usable within another circuit, thus allowing the construction of *hierarchical* designs.

Declaration

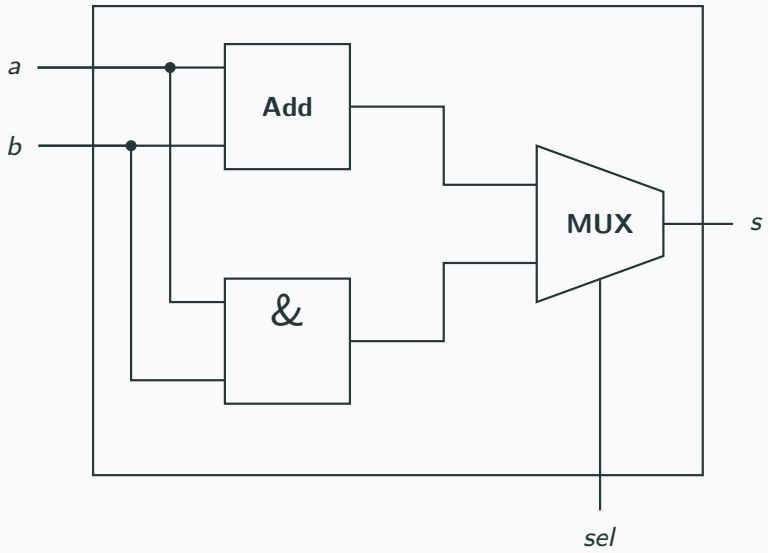
```
COMPONENT component_name IS
   PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END COMPONENT;
```

Instantiation

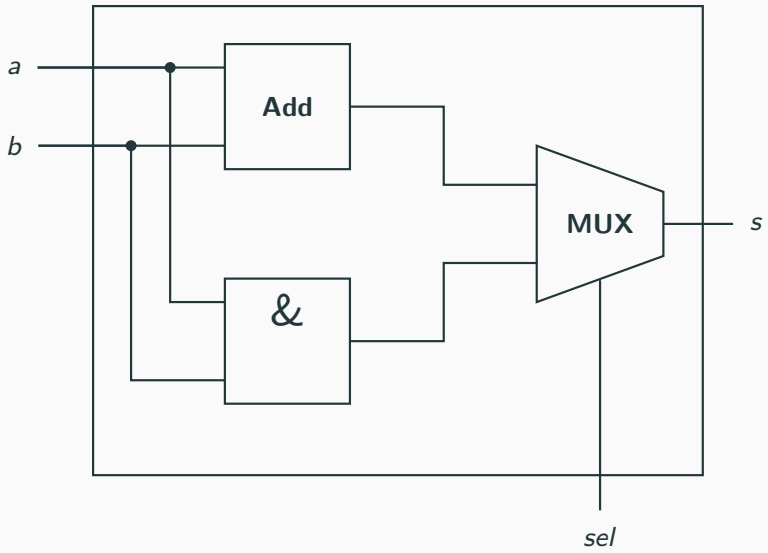
```
label: component_name PORT MAP (port_list);
```

Example

Example



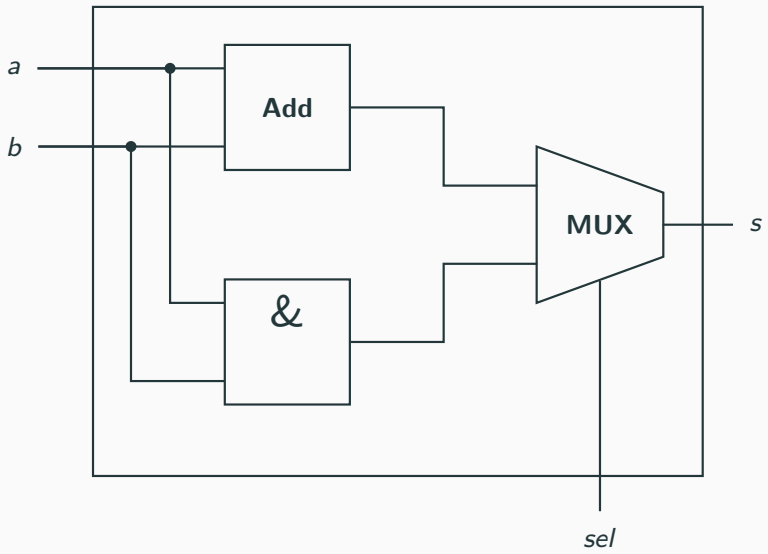
Example




```
---------- File adder.vhd: ------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY adder IS
PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      s: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END adder
-----------------------------------------------
ARCHITECTURE my_adder OF adder IS
BEGIN
   ...
END my_adder
-----------------------------------------------
```

```
---------- File and_gate.vhd: --------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY and_gate IS
PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      s: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END and_gate
-----------------------------------------------
ARCHITECTURE my_and_gate OF and_gate IS
BEGIN
   ...
END my_and_gate
-----------------------------------------------
```

Example



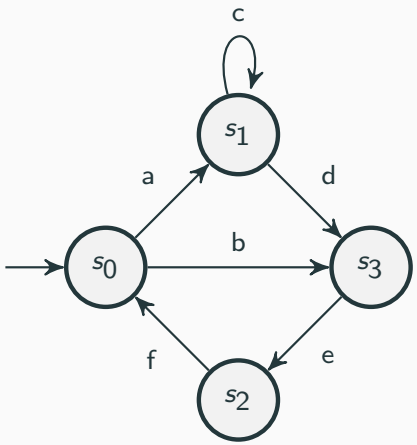

```
------------ File alu.vhd: -----------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY alu IS
PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      sel: IN STD_LOGIC;
      s: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END alu;
-------------------------------------------
ARCHITECTURE my_alu OF alu IS
   COMPONENT adder IS
      PORT (a,b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
         s: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
   COMPONENT and_gate IS
      PORT (a,b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
         s: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
   END COMPONENT;
   SIGNAL tmp1, tmp2 : STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
   U1: adder PORT MAP (a, b, tmp1);
   U2: and_gate PORT MAP (a, b, tmp2);
   s <= tmp1 WHEN sel='0' ELSE
        tmp2;
END my_alu;
-------------------------------------------
```

Mealy and Moore Machines

State Machines

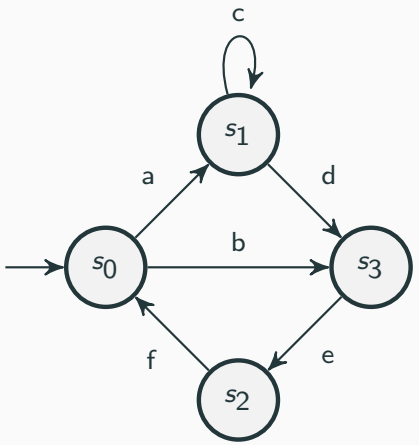
State Machines

Definition
A state machine is a mathematical model used for designing sequential logic circuits.



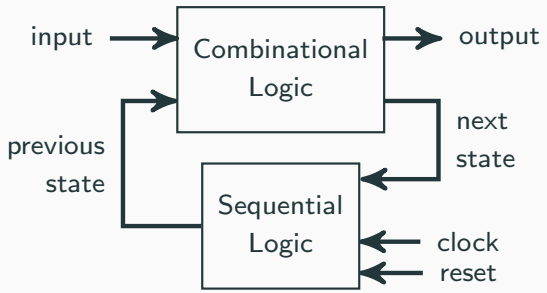
State Machines

Definition
A state machine is a mathematical model used for designing sequential logic circuits.



Implementation

- *combinational*:
  - *next state* = f(input, previous state)
  - *output* = g(input, previous state)
- a *sequential* part to synchronize the *state* changing.

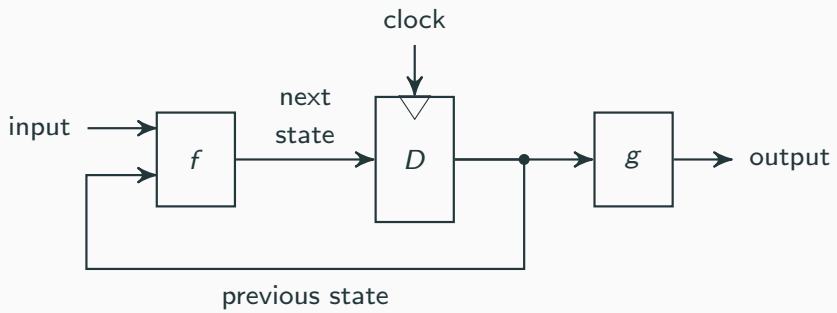


Mealy and Moore Machines

Mealy and Moore Machines

Moore Machine

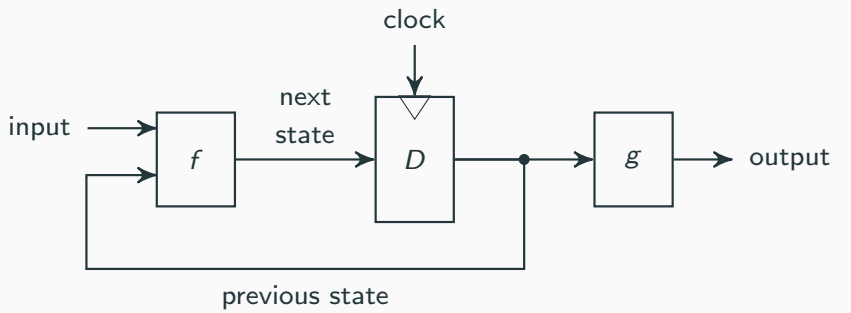
- *combinational*:
  - *next state* = f(input, previous state)
  - *output* = g(previous state)
- *sequential*: *D flip-flop*.



Mealy and Moore Machines

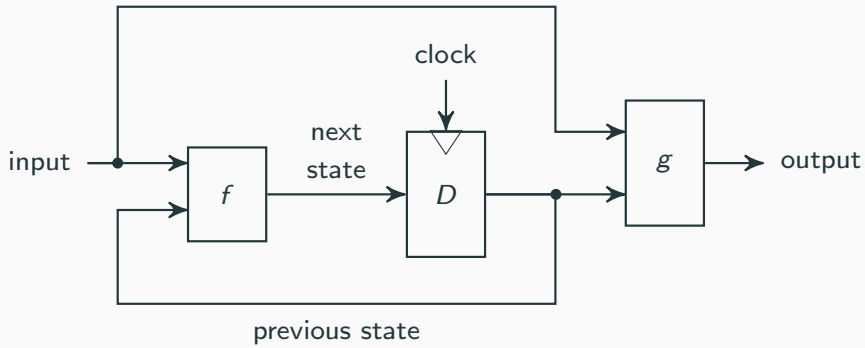
Moore Machine

- *combinational*:
  - *next state* = f(input, previous state)
  - *output* = g(previous state)
- *sequential*: *D flip-flop*.



Mealy Machine

- *combinational*:
  - *next state* = f(input, previous state)
  - *output* = g(input, previous state)
- *sequential*: *D flip-flop*.



Mealy and Moore Machines

Moore Machine

```
...
ENTITY moore_machine IS
PORT (input: IN <data_type_in>;
      clk: IN STD_LOGIC;
      output: OUT <data_type_out>);
END moore_machine;
-----------------------------------------------
ARCHITECTURE my_moore OF moore_machine IS
   COMPONENT f IS
      PORT (input: IN <data_type_in>;
         previous_state: IN <data_type_state>;
         next_state: OUT <data_type_state>);
   END COMPONENT;
   COMPONENT g IS
      PORT (current_state: IN <data_type_state>;
         output: OUT <data_type_out>);
   END COMPONENT;
   SIGNAL p_state, n_state: <data_type_state>;
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1')
         p_state <= n_state;
      END IF;
   END PROCESS;
   U1: f PORT MAP (input, p_state, n_state);
   U2: g PORT MAP (n_state, output);
END my_moore;
```

Mealy and Moore Machines

Moore Machine

```
...
ENTITY moore_machine IS
PORT (input: IN <data_type_in>;
      clk: IN STD_LOGIC;
      output: OUT <data_type_out>);
END moore_machine;
-----------------------------------------------
ARCHITECTURE my_moore OF moore_machine IS
   COMPONENT f IS
      PORT (input: IN <data_type_in>;
         previous_state: IN <data_type_state>;
         next_state: OUT <data_type_state>);
   END COMPONENT;
   COMPONENT g IS
      PORT (current_state: IN <data_type_state>;
         output: OUT <data_type_out>);
   END COMPONENT;
   SIGNAL p_state, n_state: <data_type_state>;
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1')
         p_state <= n_state;
      END IF;
   END PROCESS;
   U1: f PORT MAP (input, p_state, n_state);
   U2: g PORT MAP (n_state, output);
END my_moore;
```

Mealy Machine

```
...
ENTITY mealy_machine IS
PORT (input: IN <data_type_in>;
      clk: IN STD_LOGIC;
      output: OUT <data_type_out>);
END mealy_machine;
-----------------------------------------------
ARCHITECTURE my_mealy OF mealy_machine IS
   COMPONENT f IS
      PORT (input: IN <data_type_in>;
         previous_state: IN <data_type_state>;
         next_state: OUT <data_type_state>);
   END COMPONENT;
   COMPONENT g IS
      PORT (input: IN <data_type_in>;
         current_state: IN <data_type_state>;
         output: OUT <data_type_out>);
   END COMPONENT;
   SIGNAL p_state, n_state: <data_type_state>;
BEGIN
   PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1')
         p_state <= n_state;
      END IF;
   END PROCESS;
   U1: f PORT MAP (input, p_state, n_state);
   U2: g PORT MAP (input, n_state, output);
END my_mealy;
```

Conclusion

Conclusion - Recap

Boolean Algebra

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

- Structure of a VHDL file;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

- Structure of a VHDL file;
- Representation of data, and Data Types in VHDL;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

- Structure of a VHDL file;
- Representation of data, and Data Types in VHDL;
- Basic operators;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

- Structure of a VHDL file;
- Representation of data, and Data Types in VHDL;
- Basic operators;
- Concurrent code: the `WHEN` and `GENERATE` statements;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

- Structure of a VHDL file;
- Representation of data, and Data Types in VHDL;
- Basic operators;
- Concurrent code: the `WHEN` and `GENERATE` statements;
- Sequential code: write a `PROCESS` using `VARIABLE` and the `IF`, `WAIT`, `CASE` and `LOOP` statements;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

- Structure of a VHDL file;
- Representation of data, and Data Types in VHDL;
- Basic operators;
- Concurrent code: the `WHEN` and `GENERATE` statements;
- Sequential code: write a `PROCESS` using `VARIABLE` and the `IF`, `WAIT`, `CASE` and `LOOP` statements;
- Composition: use `COMPONENT`;

Conclusion - Recap

Boolean Algebra

- Classical logic gates, and their truth tables;
- Simplify boolean expression using properties of the algebra, and Morgan's Laws;
- Design simple logical circuit given a specification.

VHDL

- Structure of a VHDL file;
- Representation of data, and Data Types in VHDL;
- Basic operators;
- Concurrent code: the `WHEN` and `GENERATE` statements;
- Sequential code: write a `PROCESS` using `VARIABLE` and the `IF`, `WAIT`, `CASE` and `LOOP` statements;
- Composition: use `COMPONENT`;
- Simulation: write a Simulation File in VHDL.

Conclusion - Recap

State machine

Conclusion - Recap

State machine

- Definition of State machine;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;
- Multiplexer;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;
- Multiplexer;
- Barrel shifter;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;
- Multiplexer;
- Barrel shifter;
- Counter;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;
- Multiplexer;
- Barrel shifter;
- Counter;
- Simple ALU;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;
- Multiplexer;
- Barrel shifter;
- Counter;
- Simple ALU;
- Chaser;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;
- Multiplexer;
- Barrel shifter;
- Counter;
- Simple ALU;
- Chaser;
- Traffic light;

Conclusion - Recap

State machine

- Definition of State machine;
- Mealy and Moore machine;
- Implementation of Mealy and Moore machine in VHDL;

Examples

- Half Adder and Full Adder;
- n bits Adder;
- SR flip-flop;
- D flip-flop;
- Multiplexer;
- Barrel shifter;
- Counter;
- Simple ALU;
- Chaser;
- Traffic light;
- PWM;