Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
Fondazione ICSC

Advanced course to FPGA programming

Piero Vicini, Ottorino Frezza, Francesca Lo Cicero, Francesco Simula,
INFN Rome1

INFN Istituto Nazionale di Fisica Nucleare

Course Overview

Introduction to FPGA

- Architecture
- Advantages and limitations
- Design Flow
- Simulation tool

Tools & Programming languages

- VHDL syntax fundamentals
- Vivado tool overview
  - Design Flow
    - Project creation
    - Ip core integration
    - Simulation
    - Synthesis
    - Implementation
    - Tcl scripting
  - Timing analysis
  - Custom IP Design & Integration
  - FPGA Test & Debug

FPGA interconnection

- Quick intro
- AURORA Xilinx
  - Test and Debug
  - Fine tuning
  - Timing and Resources analysis
  - Optimization

HDLs why?

- Computer programming languages, like C++ and Java
  - Operations are performed in **sequential order**, one operation at a time.
  - Since an operation frequently depends on the result of an earlier operation, **the order of execution cannot be altered** at will.

- A typical digital system is normally built by **smaller parts**, with **customized wiring** that connects the input and output ports of these parts.
  - When a signal changes, the parts connected to the signal are activated and a set of new operations is initiated accordingly.
  - These operations are **performed concurrently**, and each operation will take a specific amount of time, which represents the propagation delay of a particular part, to complete.
  - After completion, each part updates the value of the corresponding output port.
  - If the value is changed, the output signal will in turn activate all the connected parts and initiate another round of operations

The sequential model used in traditional programming languages cannot capture the characteristics of digital hardware, and there is a **need for special languages** (i.e., **HDL**s) that are designed to model digital hardware.

HDLs how?

The fundamental characteristics of a digital circuit are defined by the concepts of module, connectivity, concurrency and timing.

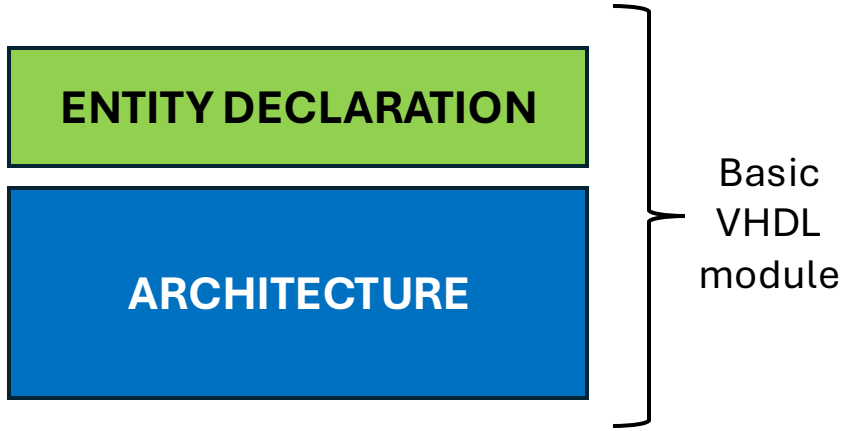
- **Module** is the basic building block; it is self-contained, independent, and has no implicit information about other entities.
- **Connectivity** models the connecting wires among the parts. It is the way that entities interact with one another.
- Since the connections of a system are seldom formed as a single thread, many entities may be active at the same time and **many operations are performed in parallel**. **Concurrency** describes this type of behavior.
- **Timing** is related to concurrency: it specifies the initiation and completion of each operation and implicitly provides a schedule and order of multiple operations.

The goal of an HDL is to describe and model digital systems faithfully and accurately. To achieve this, the cornerstone of the language should be based on the model of hardware operation, and its semantics should be able to capture the fundamental characteristics of the circuits.

VHDL module

A typical VHDL module has two main portions:

- Entity declaration: defines the module's input and output ports and configuration parameters (generic)
- Architecture block: defines the functionality of the device.



- By VHDL default, the modules will be stored in a library named **work**.

VHDL module

ENTITY DECLARATION

Name: Every word except some reserved words, not case sensitive

```
entity entity_name is
[generic (
generic_name0 : generic_type := default_value;
.....
generic_name1 : generic_type := default_value
);]
port (
port_name0 : mode port_type;
port_name1 : mode port_type
);
end entity_name;
```

[Optional] Parametrize a design, conferring the code more flexibility and reusability

Note that there is no semicolon (;) in the last port/generic declaration

Es: integer

The **mode** term indicates the direction of the signal, which can be in, out or inout.
The **in** and **out** keywords indicate that the signal flows “into” and “out of’ the circuit, respectively.
The **inout** keyword indicates that the signal flows in both directions and that the corresponding port is a bidirectional port.

VHDL module

ARCHITECTURE

```
architecture arch_name of entity_name is
[declaration]
begin
  Functional code
end arch_name;
```

[Optional]

- **Component declarations** required to describe a hierarchical design
- **Signal declarations** used for local connections between components.
- **Constant declarations**

Defines module functionality

All statements within architecture are executed concurrently:

- Concurrent statements are executed at the same time;
- The order of execution is solely specified by events occurring on signals that the statements are sensitive to; it is independent of the order in which the statements appear.

An **entity** can have multiple **architectures**.

Configuration

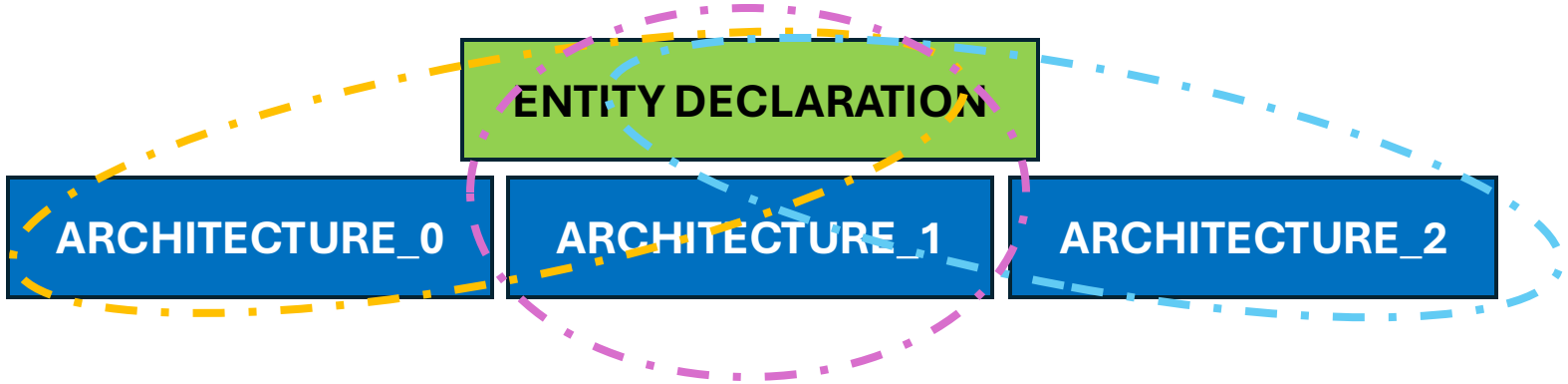
Configuration refers to the mechanism to bind an entity with its architecture.

- Allow selection of a specific architecture for an entity
- Cannot be written inside an entity or architecture

```
configuration <config_name> of <entity_name> is
    for <architecture_name>
 for INSTANCE_NAME : COMPONENT_NAME
   use entity LIBRARY_NAME.ENTITY_NAME(ARCHITECTURE_NAME);
 end for;
    end for;
end configuration <config_name>;
```

Where configuration are placed

- In separate .vhd files (recommended)
  - Modular, easy to manage
- In the same file as the design unit
  - Keep code together but less modular



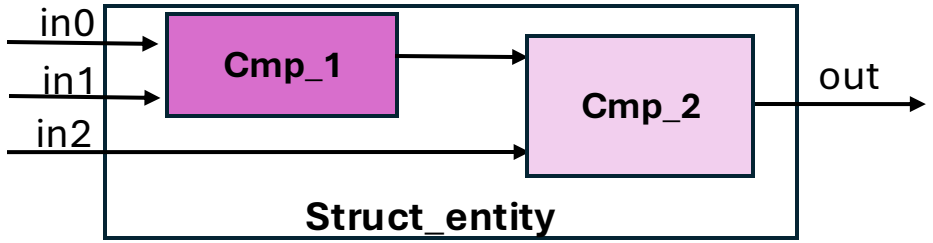
Configuration

```
entity Example is
    port (
        a, b : in  std_logic;
        y    : out std_logic
    );
end entity Example;

architecture and_arch of Example is
begin
    y <= a and b;
end architecture and_arch;

architecture or_arch of Example is
begin
    y <= a or b;
end architecture or_arch;
```

```
-- Configurations
configuration CFG_AND of Example is
    for and_arch
    end for;
end configuration CFG_AND;

configuration CFG_OR of Example is
    for or_arch
    end for;
end configuration CFG_OR;
```

Structural description

Complex project can be split in two or more simple design (**components**) to easy handle the complexity.



- **Structural style** describes the interconnection of components within an architecture.

To instance a component inside a design, you shall:

- Declare the components in the declarative part of the architecture

```
component Cmp_1 is
port (
  port_name0 : mode port_type;
  port_name1 : mode port_type
);
end component ;
```

```
architecture arch_name of struct_entity_name
is
[declaration]
begin
  Functional code
end arch_name;
```

- Instance the components in the architecture statement section (mapping ports)

```
component_inst_name : Cmp_1
Port map(
  port_name0 =>signal/port,
  port_name1 =>signal/port);
```

← Nominal port map

Constants

Constant objects are named value that cannot change during simulation or synthesis.

- Constants allow the designer to create a better-documented model that is easy to maintain and update.

- A constant in an architecture can be used by any statement inside the architecture

Constant declaration:

```
constant constant_name: constant_type:= value;
```

```
architecture arch_name of struct_entity_name is
  constant constant_name: constant_type:= value;
begin
   Functional code
end arch_name;
```

Signals

Signals are used to represent connections and data storage in VHDL

Signal declaration:

```
signal signal_name: signal_type [range] [:= default value];
```

Optional; it used in simulation and may be used in synthesis, depending on the platform (not safe).

```
architecture arch_name of simplified_entity_name is
  Signal signal_name: signal_type;
begin
  Functional code
end arch_name;
```

- Signals can be seen as real, physical signals
- To define more than one signal of the same type, separates the signal names with a comma.
- Signals declared in architecture declaration section can be referenced in architecture.

Signal assignment

Signal assignment is used to assign values to signal

- Identified by the symbol **<=**.
- Sensitive to changes on any signals that are to the right of the <= symbol.

Signal assignment:

```
signal_name <= value;
```

Constant or another signal

- The designer has the possibility to perform a signal assignment after certain amount of time, implementing the **delay in the assignment** (**only in simulation**).

```
signal_name <= value after time_value;
```

Inertial delay

```
signal_name <= transport value after time_value;
```

Transport delay

- In Architecture's functional code, signal assignment are **concurrent statements.**
  - Concurrent statements are executed at the same time;
  - The order of execution is solely specified by events occurring on signals that the statements are sensitive to; it is independent of the order in which the statements appear.

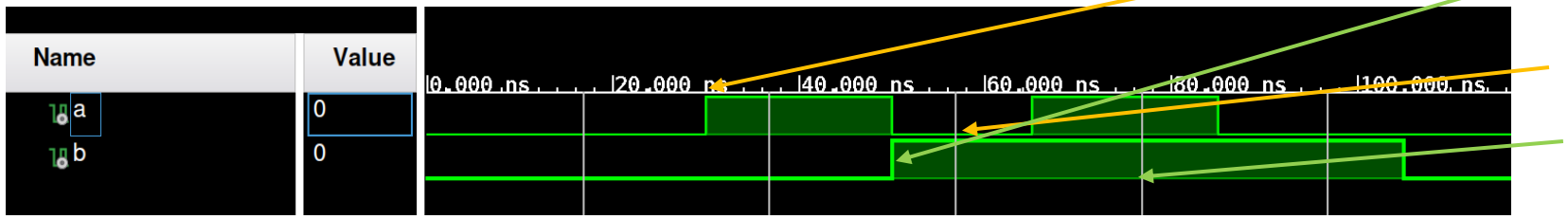
Signal delay model

- **Inertial delay** models physical devices with inertia
  - Ignores glitches (short pulses)
  - Only stable changes (longer than the delay) are propagated

```
signal_name <= value after time_value;
```

```
b <= a after 20 ns;
```

- **b** takes the value of **a** after 20 ns second of inertial delay.
- A pulse shorter than 20 ns on a will be **ignored**.



a changes from 0 to 1,
and **b** change its value after 20 ns;

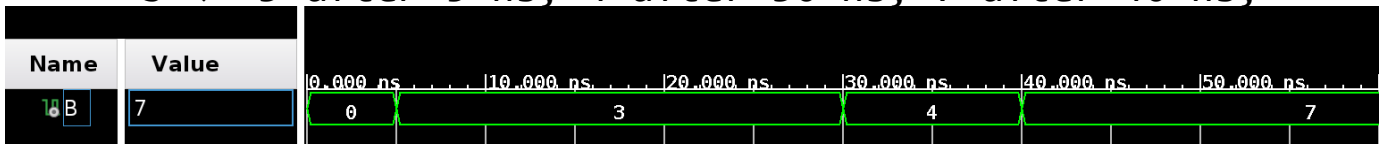
a changes value going to 0 and then to 1 in 15 ns (glitch).
This delay is less than inertial delay of 20 ns, so **b** remains unchanged.

- Multiple inertial delayed assignments define a sequence of value changes for a signal.

```
signal_name <= value0 after time_value0, value1 after timevalue1, ...;
```

  - Time values are relative to the current simulation time
  - Useful for generating stimuli in testbenches.

```
b <= 3 after 5 ns, 4 after 30 ns, 7 after 40 ns;
```

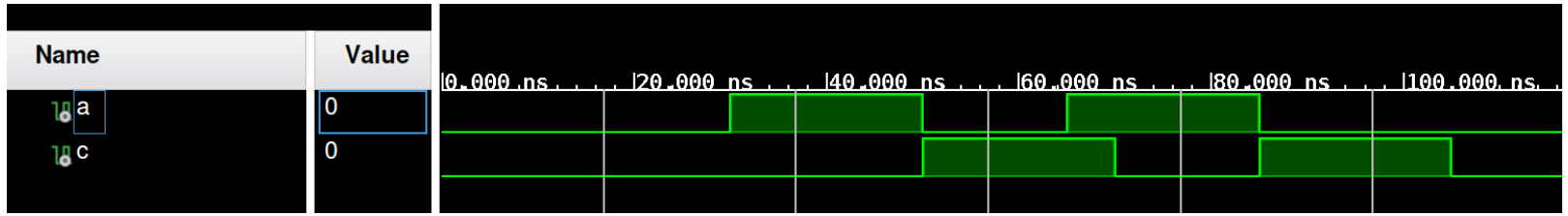


Signal delay model

- **Transport delay** idealizes propagation delay of a signal in hardware.

```
signal_name <= transport value after time_value;
```

```
c <= transport a after 20 ns;
```



No matter how fast a changes his value, **c** will follow the behavior of **a** after the amount of time specified in the delay statement.

Conditional and selected signal assignment

Conditional signal assignment statement assigns a value to the target signal based on conditions.

- The statement WHEN conditions are executed one at a time in sequential order until the conditions are met.
- The first statement that matches the conditions required assigns the value to the target signal.

```
Target_signal <=
   Value_0 when condition_0 else
   Value_1 when condition_1 else
   ...
   Value_n when condition_n else
   Default_value;
```

Selected signal assignment selects among a number of options to assign the correct value to the target signal.

- All of the possible values of the expression must have a matching choice in the selected signal assignment (or an **OTHERS** clause must exist).

```
with selector select
target_signal <=
  value_0 when selector_value0,
  ...
  value_n when selector_valuen,
  default_value when others;
```

Data type

Type specification specifies the characteristics of the object (port/signal/constant)

- **Pre-defined type**
- **User-defined type**

```
entity simplified_entity_name is
port (
port_name : mode port_type;
port_name : mode port_type
);
end simplified_entity_name;
```

```
architecture arch_name of etity_name is
  Signal signal_name:   signal_type;
  constant constant_name: constant_type := value;
begin
  Functional code
end arch_name;
```

- In VHDL, selecting the appropriate data type is fundamental for accurately modeling hardware behavior and ensuring efficient synthesis and simulation.
- Operation between different data types are not allowed!

VHDL predefined package

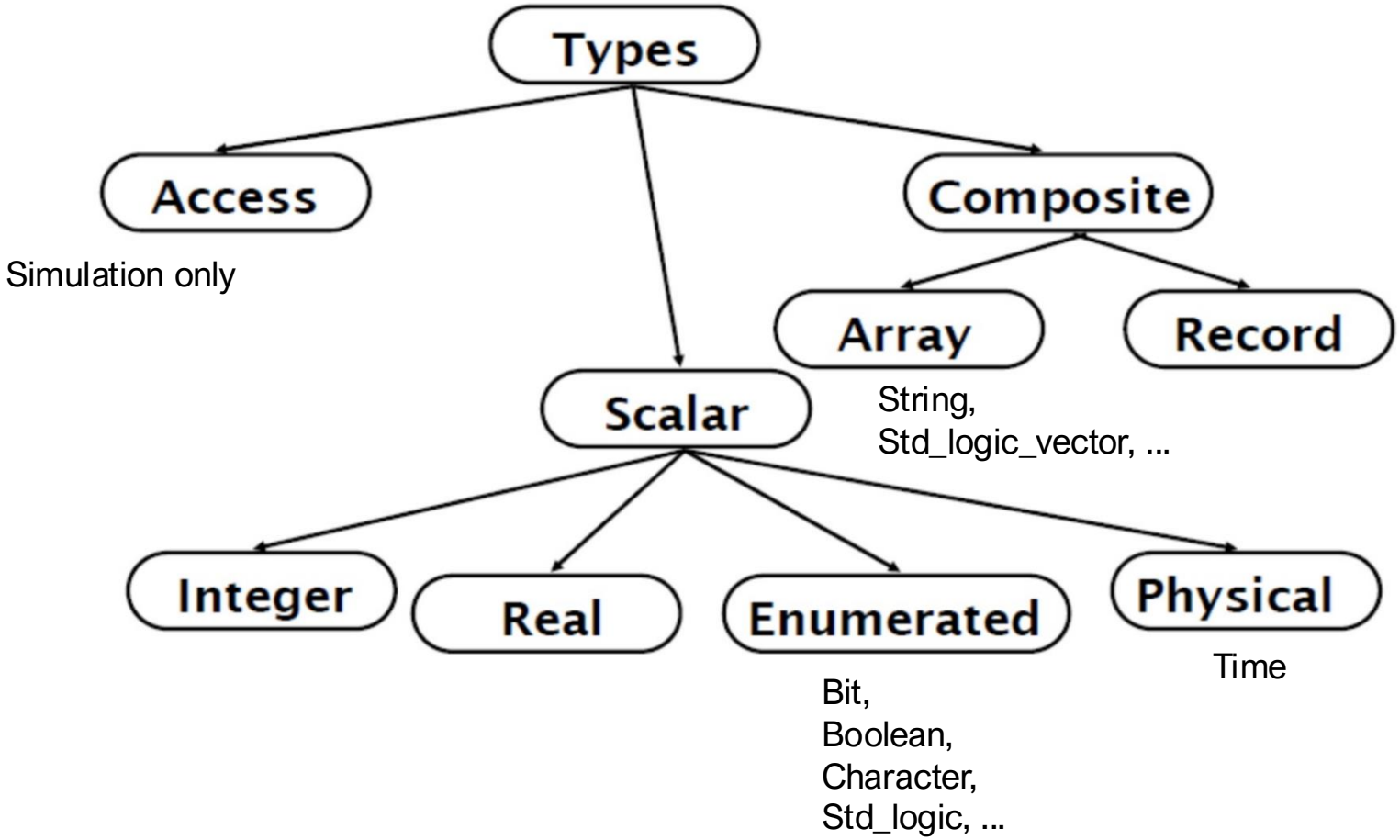
The **standard** and **IEEE** libraries were created to provide a set of predefined, commonly used data types, making it easier to write, read, and maintain VHDL designs

- packages group related types, constants, and functions together
  - **Library std**
    - **Standard** (visible by default) : type definition (INTEGER, BIT, BOOLEAN, CHARACTER) and logic, arithmetic, comparison, shift and concatenation operators
    - **Textio**: text and files
  - **Library IEEE**
    - **std_logic_1164** defines a standard for describing digital logic values in VHDL (IEEE STD 1164). It contains definitions for std_logic (single bit) and for std_logic_vector (array).
    - **numeric_std**: defines signed and unsigned types, and operators

To use a predefined package, you must include the library containing it and use statements before the entity declaration (std and work libraries are visible by default).

```
library library_name ;
use library_name.package_name.all ;
```

VHDL Predefined Data type



VHDL Real and Physical data type

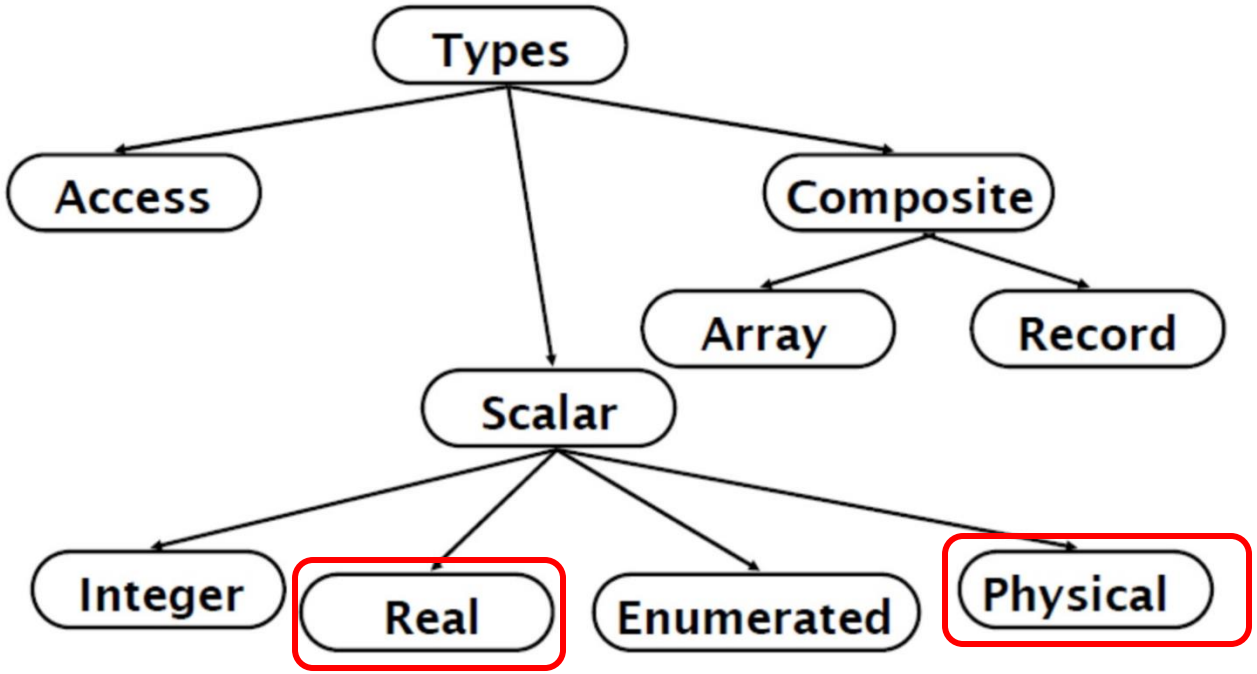
Pre-defined type in Standard Library (visible by default)

- Type **Real**
  - range defined by standard: -1.0E38 to 1.0E38 (Floating Point numbers)

```
ARCHITECTURE test OF test IS
SIGNAL a : REAL;
BEGIN
  a <= 1.0;        --OK
  a <= 1;          --Error
  a <= -1.0E10;    --OK
  a <= 1.5E-20;    --OK
END test;
```

- Type **time** (physical)

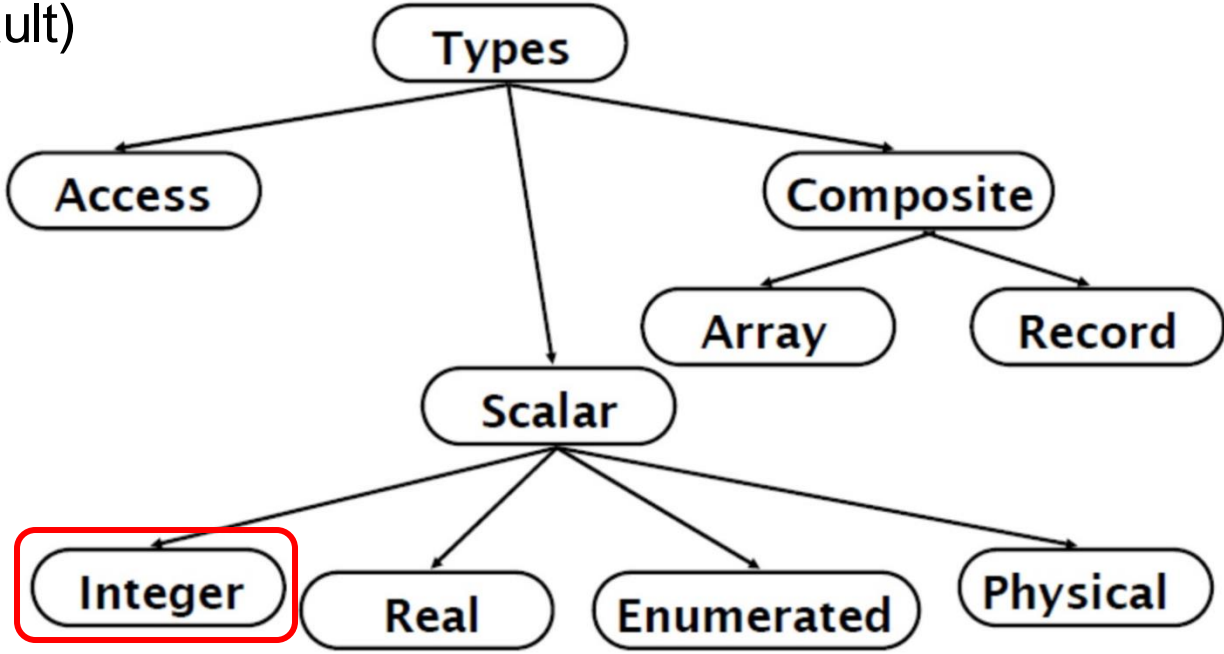
```
  UNITS
  fs;                --femtosecond
  ps   =  1000 fs;
  ns   =  1000 ps;
  us   =  1000 ns;
  ms   =  1000 us;
  sec  =  1000 ms;
  min  =  60 sec;
  hr   =  60 min
  END UNITS;
```



VHDL Integer data type

Pre-defined type in Standard Library (visible by default)

- Type **Integer**
  - Range defined by standard: -2147483647 to 2147483647 (**32-bit** representation)
  - Always define the range (otherwise the compiler will employ 32 bits to represent them)




```
Signal signal_integer : integer range 0 to 255;
```

A **subtype** of a given type restricts the type range.
Integer has two predefined subtypes:

- subtype **natural** is integer range 0 to integer'high;
- subtype **positive** is integer range 1 to integer'high;

→ 2147483647

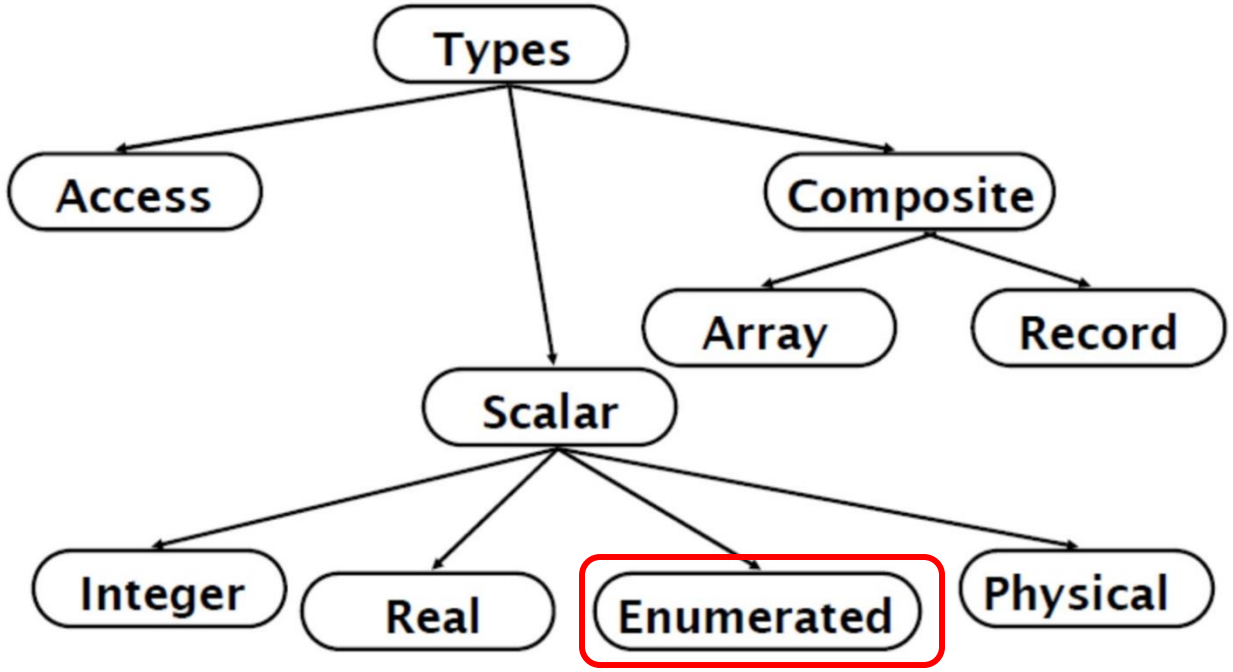
VHDL Enumerated data type

Enumerated data type consists of a set of named values called elements.

Pre-defined type in Standard Library (visible by default)

- Type **BIT** is (‘0’, ‘1’)
- Type **BOOLEAN** is (false, true)
- Type **CHAR** is (NUL, SOH, ..., DEL...)
  - Symbols are from the ISO 8859-1 character set
  - The first 128 symbols comprising regular ASCII CODE

00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	NL	VT	NP	CR	SO	SI
10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

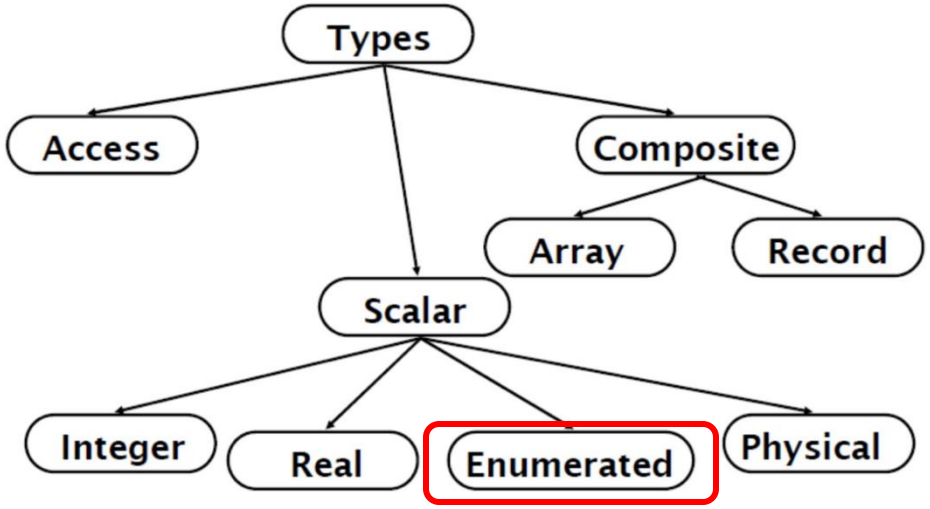


Std_ulogic and Std_logic

Pre-defined type in ieee.std_logic_1164

- Type **Std_ulogic**
  - 1 logic one
  - 0 logic zero
  - Z high-impedance
  - U uninitialized (sim. only)
  - X unknown (driven)
  - - don't care
  - H weak high
  - L weak low
  - W weak signal

- **Pre-defined type**



Problem: If two sources try to drive the same std_ulogic signal, it causes a compilation error.

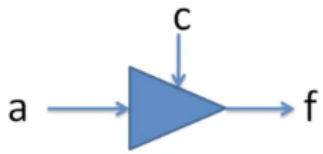
- Type **std_logic** is resolved std_ulogic

```
----------------------------------------------------------------
CONSTANT resolution_table : stdlogic_table := (
    ---------------------------------------------------------
    -- U    X    0    1    Z    W    L    H    -       |   |
    ---------------------------------------------------------
    ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
    ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
    ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
    ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
    ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
    ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
    ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |);
    ---------------------------------------------------------
```

Std_logic High impedence

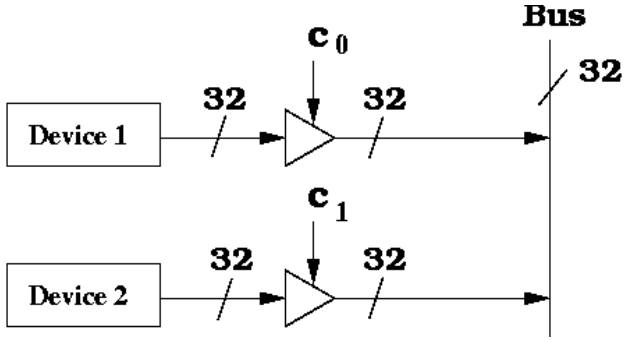
High impedance (Hi-Z) represents a disconnected or floating state on a signal or pin.

- A **tri-state buffer** is a digital circuit with three possible output states:
  - Logic High ('1') → the buffer drives the output high.
  - Logic Low ('0') → the buffer drives the output low.
  - High Impedance ('Z') → the buffer "disconnects" from the bus, allowing other devices to drive it.



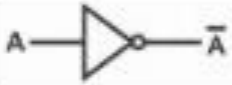
c	a	f
0	0	Z
0	1	Z
1	0	0
1	1	1

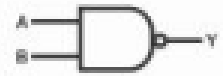
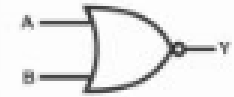
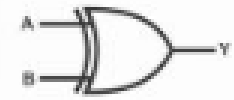
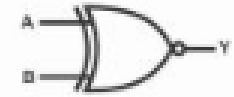
  - Prevents bus conflicts.
  - Allows multiple devices to share the same line safely.
  - Essential in shared digital systems, like microprocessors, memory interfaces, and buses.



- When c0 = '1', Device 1 drives bus line.
- When c0 = '0', the buffer goes into high impedance ('Z'), letting Device 2 use the bus.

Std_ulogic and Std_logic
Basic logic gates

LOGIC FUNCTION	LOGIC SYMBOL	BOOLEAN EXPRESSION	TRUTH TABLE		
			INPUTS		OUTPUTS
			B	A	Y
AND		A+B=Y	0	0	0
			0	1	0
			1	0	0
			1	1	1
OR		A+B=Y	0	0	0
			0	1	1
			1	0	1
			1	1	1
inverter		$A=\overline{A}$		0	1
				1	0

NAND		$\overline{A+B}=Y$		0	1
			0	1	1
			0	0	1
			1	1	0
NOR		$\overline{A+B}=Y$	1	0	1
			0	1	0
			0	0	0
			1	1	0
XOR		$A \oplus B=Y$	1	0	0
			0	1	0
			0	0	1
			1	1	1
XNOR		$\overline{A \oplus B}=Y$	1	0	0
			0	0	1
			0	1	0
			1	0	0
			1	1	1

Hands-on

MUX

ghdl_examples/Mux

GHDL and GTKWave at work

Few basic examples of use

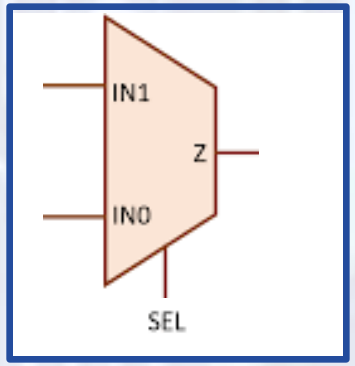
- Basic VHDL
- Scripting
- Execution and waveform analysis

Hands-on

- Launch JupyterLab: `https://xilinx01-2.mib.infn.it`
- Launch Desktop
- Open terminal window and download material
  - `Git clone https://gitlab.com/piero.vicini/ghdl_examples`
- 3 directories: Register, Counter, Coffee-maker.
- Directories contain:
  - nomefile.vhd (src file)
  - nomefile_TB.vhd (testbench file)
  - Makefile (script for compilation and run)

2-to-1 multiplexer (MUX)

Exercise:
- go to directory MUX, have a look to the code, execute simulation and visualize the output
- or Do it yourself...



Specifications:
Module selects one of two inputs and forwards it to the output based on a 1-bit selection input.

Input:
- Two bit (IN0, IN1)
- one bit selector (SEL)

Output: single bit (Z)

Function: The control inputs (SEL) select which one of the data inputs is connected to the output.
- If (SEL) = 0 → Z = IN0
- If (SEL) = 1 → Z = IN1

VHDL Array data type

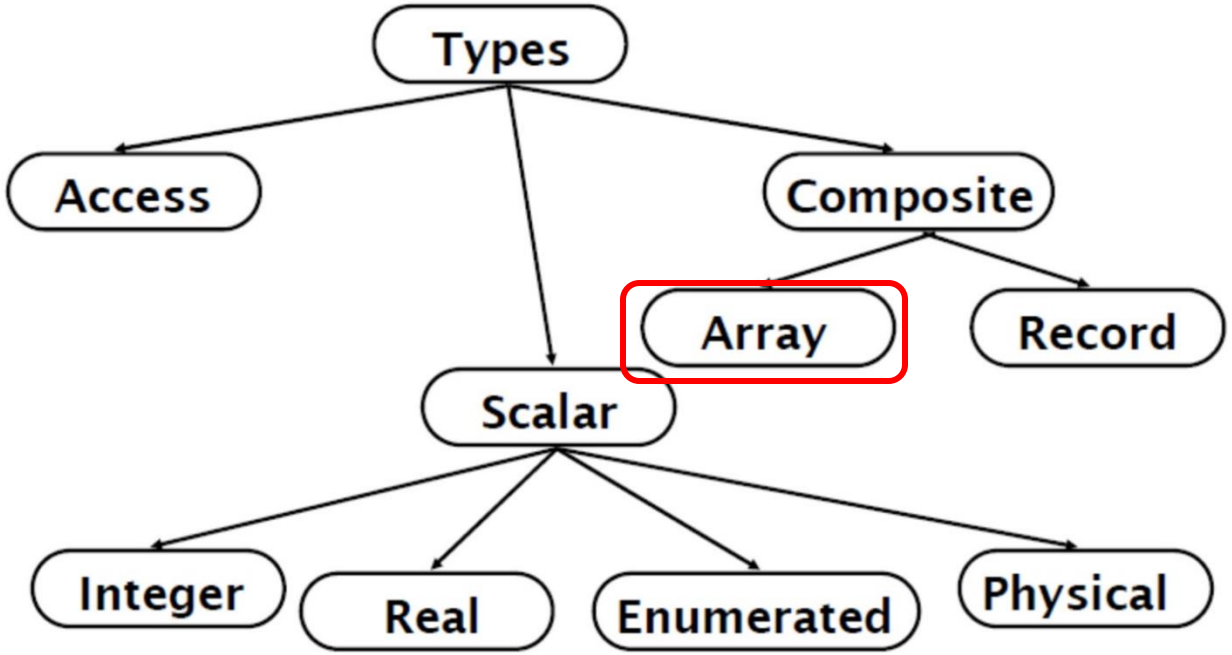
Array types is used to collect one or more elements of a similar type in a single construct.

Pre-defined array

- Type **bit_vector** is array (integer range **<>**) of bit
- Type **Boolean_vector** is array (integer range <>) of boolean
- Type **string** is array (positive range <>) of char
- Type **std_logic_vector** is array (natural range <>) of std_logic
- Type **signed/unsigned** is array (integer range <>) of std_logic

Unconstrained array (**<>**): Actual range is established later, when type is used in the code

Pre-defined type



VHDL Array data type: std_logic_vector

- Type **std_logic_vector** is array (natural range <>) of std_logic
- Pre-defined type in library ieee.std_logic_1164
- To access the value of an element from this vector, you can use the index

- Natural range can be ascending or descending
  - The choice between ascending and descending order is often a question of the designer's preferences. The most important thing is to choose one style and then follow it consistently; mixing the two different styles in one project can easily lead to trouble.

```
library ieee;
use ieee.std_logic_1164.all;
…
constant bin_data : std_logic_vector(7 downto 0) := "00000101"; -- Descending range
signal bin_data_1 : std_logic_vector(7 downto 0);
signal bin_data_2 : std_logic_vector(0 to 7);                   -- Ascending range
signal a,b,c : std_logic;
…
bin_data_2 <= "00000101";     -- Note double quote
bin_data_1 <= (others=>'0'); -- This syntax assigns '0' to all bits of the vector bin_data_1.
a <= bin_data (0);            -- a = '1'
b <= bin_data_2 (0);          -- b = '0'
```

VHDL Array data type: std_logic_vector

- Type **std_logic_vector** is array (natural range <>) of std_logic
- Pre-defined type in library ieee.std_logic_1164
- To access the value of an element from this vector, you can use the index

- Natural range can be ascending or descending
  - The choice between ascending and descending order is often a question of the designer's preferences. The most important thing is to choose one style and then follow it consistently; mixing the two different styles in one project can easily lead to trouble.

```
library ieee;
use ieee.std_logic_1164.all;
…
constant bin_data : std_logic_vector(7 downto 0) := "00000101"; -- Descending range
signal bin_data_1 : std_logic_vector(7 downto 0);
signal bin_data_2 : std_logic_vector(0 to 7);                   -- Ascending range
signal a,b,c : std_logic;
…
bin_data_2 <= "00000101";     -- Note double quote
bin_data_1 <= (others=>'0'); -- This syntax assigns '0' to all bits of the vector bin_data_1.
a <= bin_data (0);            -- a = '1'
b <= bin_data_2 (0);          -- b = '0'
```

std_logic_vector is just a vector of bits with no numeric meaning!

VHDL std_logic_vector as integer

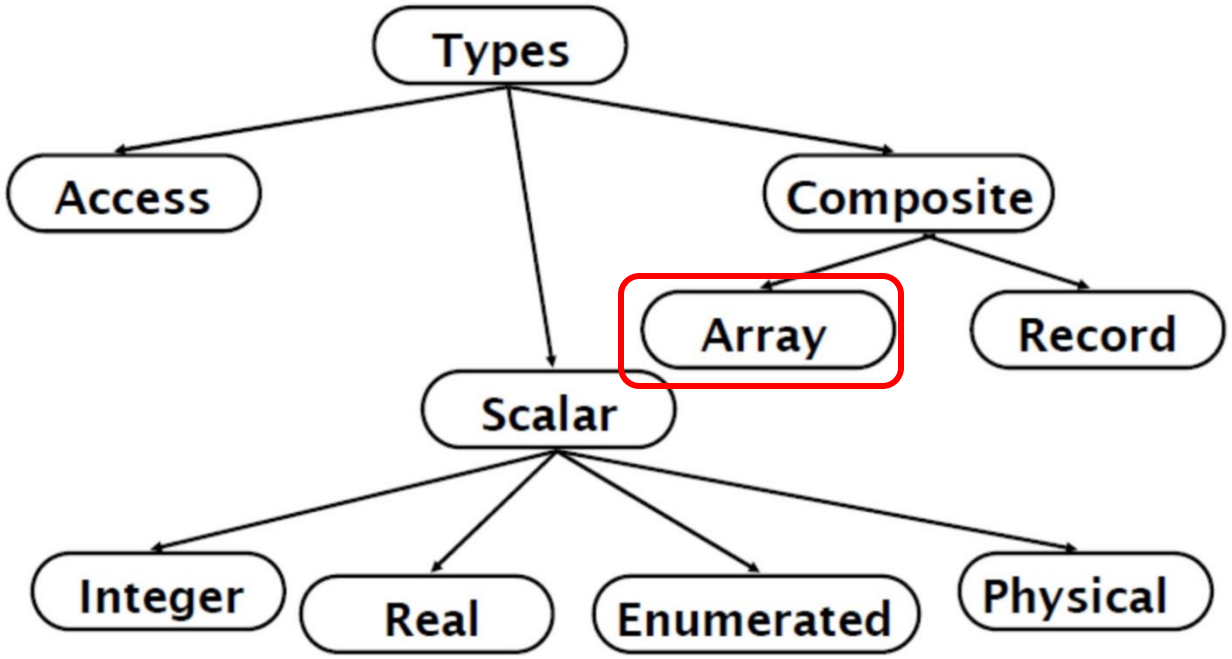
Std_logic_unsigned, signed and arith library were created by Synopsys in the very early. They were distributed free of charge and compiled into the IEEE library (even though they're not an IEEE standard).

Library **ieee.std_logic_unsigned**

- provides operations for treating **std_logic_vector** signals as unsigned integer
- allows to perform arithmetic operations like addition, subtraction, multiplication, and comparisons .

Library **ieee.std_logic_signed**

- provides operations for treating **std_logic_vector** signals as signed integer
- allows to perform arithmetic operations like addition, subtraction, multiplication, and comparisons .

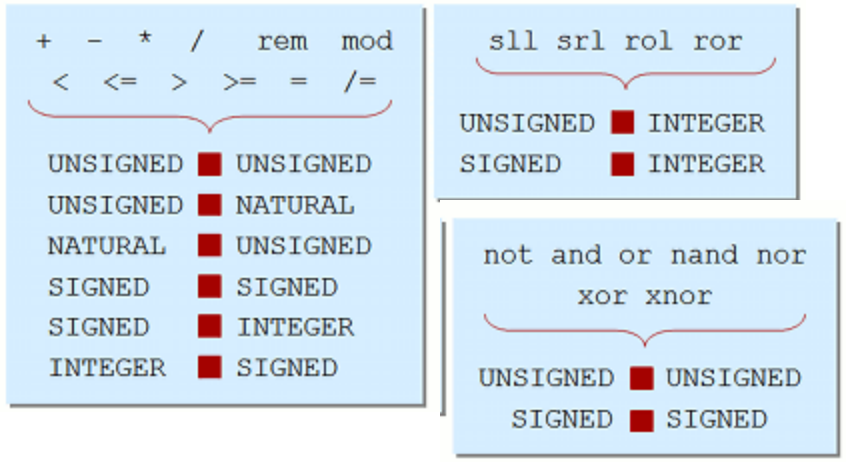


VHDL Array data type: signed/unsigned

- Types signed/unsigned are array (natural range <>) of std_logic **with a numeric interpretation**
- Pre-defined type in library **ieee.numeric_std**
- For signed signals, internally the FPGA will use Two's Complement representation

```
library IEEE;
use IEEE.NUMERIC_STD.ALL;
…
signal signal_unsigned: unsigned(3 downto 0);
…
signal_unsigned <= "0001"; -- Note double quote
```

Available operators



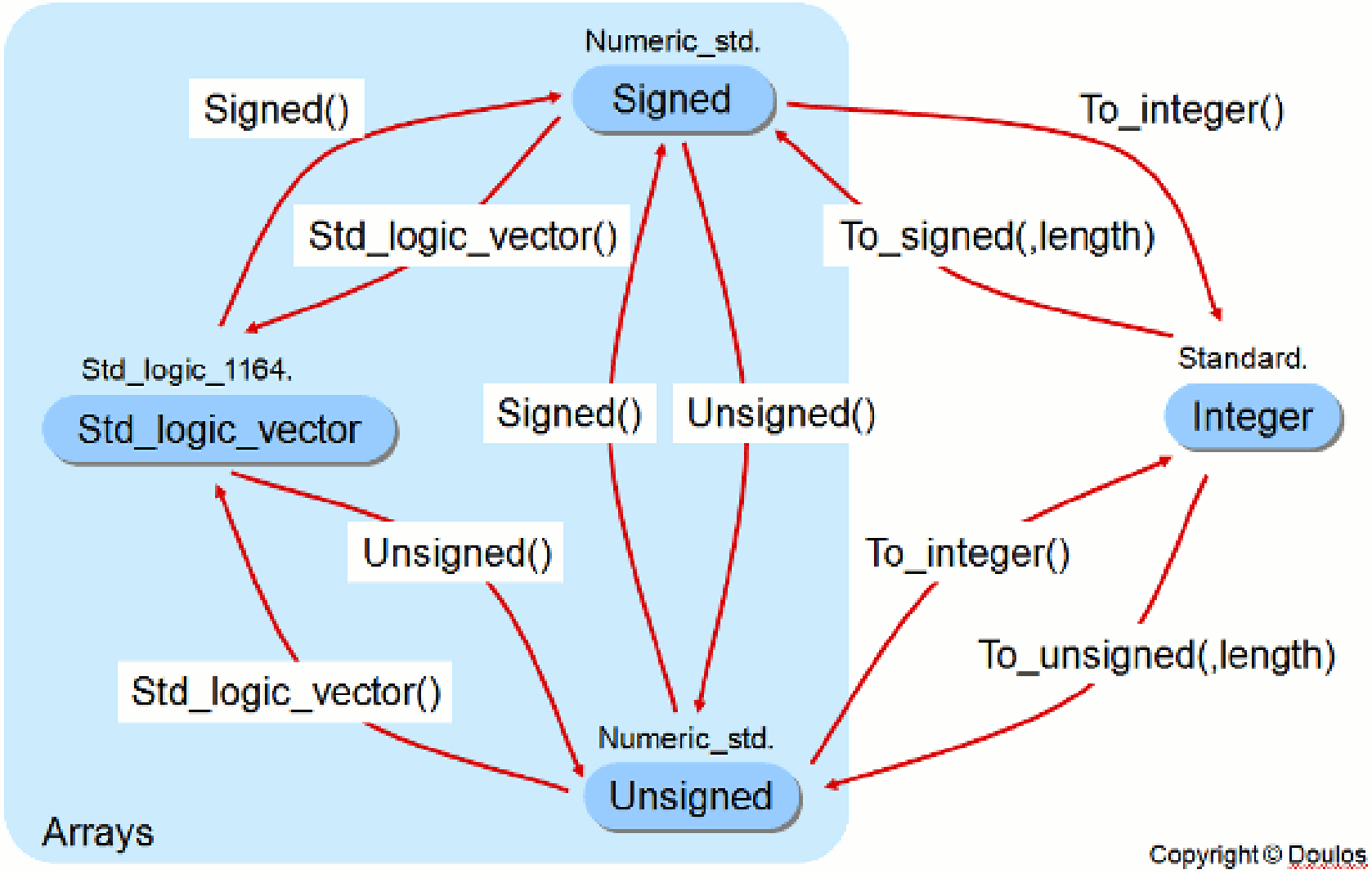
Type Compatibility and Result Rules for signed and unsigned in VHDL

Operation	Left Operand	Right Operand	Result
Addition (+) and Subtraction (-)	unsigned	unsigned	unsigned
	unsigned	natural	unsigned
	natural	unsigned	unsigned
	signed	signed	signed
	signed	integer	signed
	integer	signed	signed
	unsigned	std_logic	unsigned
	std_logic	unsigned	unsigned
	signed	std_logic	signed
	std_logic	signed	signed

VHDL Numeric Types: Comparison of integer, unsigned, and signed

Feature	integer	unsigned	signed
Type	Scalar (not a vector)	Vector (std_logic_vector)	Vector (std_logic_vector)
Defined in	VHDL standard	ieee.numeric_std	ieee.numeric_std
Bit-level?	❌ No	✅ Yes	✅ Yes
Supports sign?	✅ Yes	❌ No (positive only)	✅ Yes (positive and negative)
Representation	Decimal numbers	Binary (no sign)	Binary (two's complement)
Range	-2,147,483,647 to +2,147,483,647 (typical)	0 to 2^n - 1	-2^{n-1} to 2^{n-1} - 1
Arithmetic	Native arithmetic	Supported via numeric_std	Supported via numeric_std

VHDL Integer type conversion



Hands-on

Adder ghdl_examples/Adder

VHDL sequential behaviour: process

Process statement contains only sequential statements.

- A process contains:
  - sensitivity list
  - declarative part
  - sequential statement section

```
[process_label] : process(sensitivity_list)
[declarative part]
begin
-- sequential statement
end process [process_label];
```

The process label is optional, you can avoid using the label. Labeling all process you use, the code will be clear, and it will be simple to arrange the simulation environment.

In the process sensitivity list are declared all the signal which the process is sensitive to : the process is evaluated any time a transaction is scheduled on the signals in the sensitivity list.

Optinal declarative part is used to declare **local** variables, types and constant.

- The process statement is a concurrent statement.

VHDL variable

A **variable** in VHDL is a local storage element that exists only inside a process, function, or procedure. It is used to store local values.

```
variable var_name : type [:= initial_value];
```

- Changes take effect **immediately**, unlike signals which update at the end of the process or delta cycle.
- Variables cannot be assigned to output ports; use signals for that

Update timing

Feature	**Variable**	**Signal**
Assignment operator	:=	<=
Update timing	Immediate	Scheduled (end of process)
Scope	Local to process/function	Can be global to architecture
Synthesis	Synthesizable if inside a process	Always synthesizable

VHDL if statement

IF statements are used in VHDL to test for various conditions.

```
If <condition> then
   Sequence of statement
[elsif condition_1 then
   Sequence of statement
]
[else
   Sequence of statement
]
end if;
```

- The <condition> can be a Boolean true or false, or it can be an expression which evaluates to true or false.

Logical operators:

not *a*	true if *a* is false
a **and** *b*	true if *a* and *b* are true
a **or** *b*	true if *a* or *b* are true
a **nand** *b*	true if *a* or *b* is false
a **nor** *b*	true if *a* and *b* are false
a **xor** *b*	true if exactly one of *a* or *b* are tru
a **xnor** *b*	true if *a* and *b* are equal

Relational operators:

=	equal
/=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

VHDL if statement

IF statements are used in VHDL to test for various conditions.

```
If <condition> then
   Sequence of statement
[elsif condition_1 then
   Sequence of statement
]
[else
   Sequence of statement
]
end if;
```

- **elsif** is optional and may be used multiple times.
- **else** is optional but pay attention: if the code doesn't explicitly specify the value of the output for one case, the previous value of the output will be held. This is equivalent to the memory effect that a D latch exhibits!

Latch are not suitable for synthesis

- If statements are synthesized by generating a multiplexer for each signal assigned within the if statement. The select input on each mux is driven by logic determined by the if condition, and the data inputs are determined by the expressions on the right-hand sides of the assignments.

VHDL if statement example

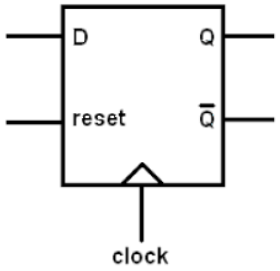
```vhdl
library ieee ;
use ieee.std_logic_1164.all;

entity mux_2 is
port(
  a : in std_logic_vector(2 downto 0);
  b : in std_logic_vector(2 downto 0);
  s : in std_logic;
  m : out std_logic_vector(2 downto 0));
end mux_2;

architecture rtl of mux_2 is
begin
 p_mux : process(a,b,s)
 begin
   if(s='0') then
      m <= a ;
   else
      m <= b ;
   end if;
 end process p_mux;
end rtl;
```

=

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

entity mux_2 is
port(
  a : in std_logic_vector(2 downto 0);
  b : in std_logic_vector(2 downto 0);
  s : in std_logic;
  m : out std_logic_vector(2 downto 0));
end mux_2;

architecture rtl of mux_2 is
begin
  m <= a when (s='0') else b;
end rtl;
```

VHDL synchronous process example

```
entity reg8 is
Port (
 clk : in STD_LOGIC;
 reset : in STD_LOGIC;
 d : in STD_LOGIC_VECTOR(7 downto 0);
 q : out STD_LOGIC_VECTOR(7 downto 0));
end reg8;

architecture Behavioral of reg8 is
signal reg : STD_LOGIC_VECTOR(7 downto 0);

begin
 process(clk, reset)              -- asynchronous reset in sensitivity list
 begin
    if reset = '1' then           -- Reset active high
      reg <= (others => '0');     -- async reset clears register immediately
    elsif rising_edge(clk) then   -- rising_edge(clk) means clk'event and clk='1'
      reg <= d;                   -- on clock rising edge, capture input d
    end if;
 end process;
q <= reg;
end Behavioral;
```



FLIP FLOP D

VHDL case-when statement

The **Case-When** statement will cause the program to take one out of multiple different paths, depending on the value of a signal, variable, or expression.

- It's a more elegant alternative to an If-Then-Elsif-Else statement with multiple Elsif's.

```
case <expression> is
when <choice> => code for this branch
when <choice> => code for this branch
...
end case;
```

- The <expression> is usually a variable or a signal.
- The Case statement may contain multiple when choices, but only one choice will be selected.

  The <choice> may be
  - a unique value (like "11")
  - a range (like 5 to 10)
  - several values (like 1|3|5)
  - **others** (selected whenever no other choice was matched)

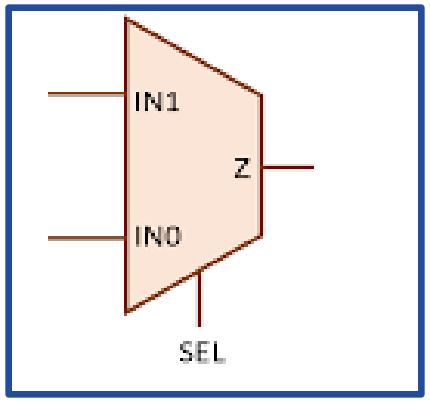
VHDL case-when statement example

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

entity mux4_case is
port(
 a : in std_logic_vector(3 downto 0);
 b : in std_logic_vector(3 downto 0);
 c : in std_logic_vector(3 downto 0);
 d : in std_logic_vector(3 downto 0);
 s : in std_logic_vector(1 downto 0);
 m : out std_logic_vector(3 downto 0));
end mux4_case;

architecture rtl of mux4_case is
begin
 p_mux : process(a,b,c,d,s)
 begin
   case s is
     when "00" => m <= a;
     when "01" => m <= b;
     when "10" => m <= c;
     when others => m <= d;
   end case;
 end process p_mux;
end rtl;
```

=

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

entity mux4_case is
port(
 a : in std_logic_vector(3 downto 0);
 b : in std_logic_vector(3 downto 0);
 c : in std_logic_vector(3 downto 0);
 d : in std_logic_vector(3 downto 0);
 s : in std_logic_vector(1 downto 0);
 m : out std_logic_vector(3 downto 0));
end mux4_case;

architecture rtl of mux4_case is
begin
 p_mux : process(a,b,c,d,s)
 begin
   If    s="00" then  m <= a;
   elsif s="01" then  m <= b;
   elsif s="10" then  m <= c;
   else m <= d;
   end if;
 end process p_mux;
end rtl;
```

VHDL case-when statement example



Truth table

SEL	IN0	IN1	OUT_Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

```
architecture truth_table of MUX is
   signal temp_sel : std_logic_vector(2 downto 0) ;

begin
 temp_sel <= SEL & IN0 & IN1 ;    -- concatenation
   process(SEL,IN0,IN1)
   begin
      case(temp_sel ) is
         when "000"  => OUT_Z <= '0' ;
         when "001"  => OUT_Z <= '0' ;
         when "010"  => OUT_Z <= '1' ;
         when "011"  => OUT_Z <= '1' ;
         when "100"  => OUT_Z <= '0' ;
         when "101"  => OUT_Z <= '1' ;
         when "110"  => OUT_Z <= '0' ;
         when "111"  => OUT_Z <= '1' ;
         when others => OUT_Z <= 'X' ; -- catch-all
      end case ;
   end process ;

end architecture truth_table;
```

Hands-on

Register

ghdl_examples/Register

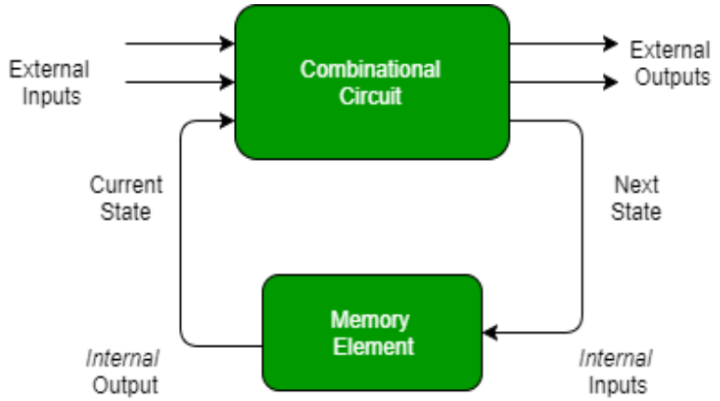
Counter

ghdl_examples/Counter

Sequential circuit

Sequential circuits: Output depends on inputs + history of past inputs (memory).

- Example: counters, registers, sequence detectors.



Combinational circuits: Output depends only on current inputs

A sequential circuit can be

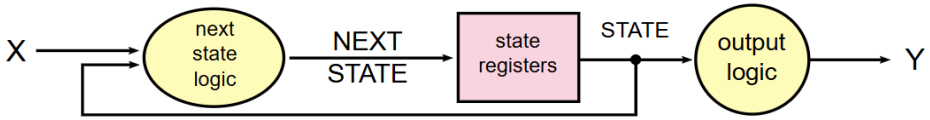
- **Asynchronous**: the changes in all the state variables are not synchronized and can occur at any time. The transitions between states are driven by the inputs themselves, which can lead to more complex behavior.
- **Synchronous**: changes on all the state variables are synchronized with a clock signal.

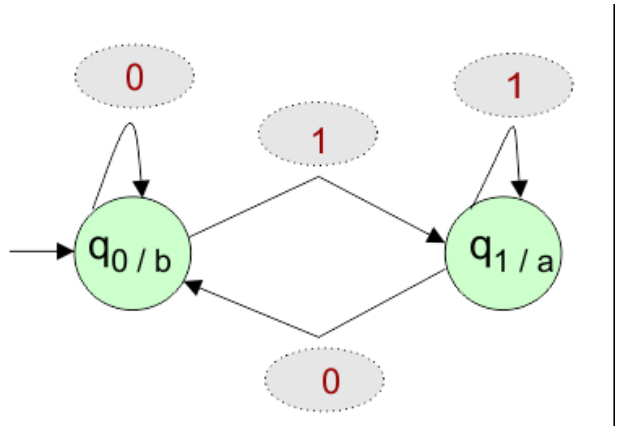
A **Finite State Machine (FSM)** is the mathematical abstraction of a sequential circuit:.
It consists of a finite number of states, transitions, inputs, and outputs.

- **States**: Represent the current condition of the system.
- **Transitions**: Rules for moving from one state to another, based on inputs.
- **Inputs**: Signals that trigger state changes.
- **Outputs**: Signals produced

Moore FSM

Moore Machine: Output depends only on the current state.

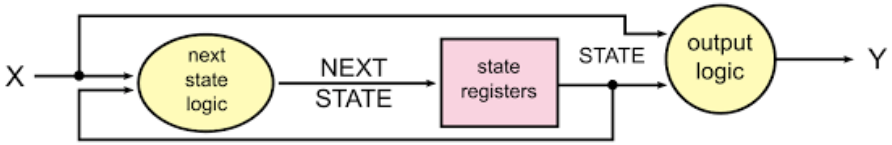


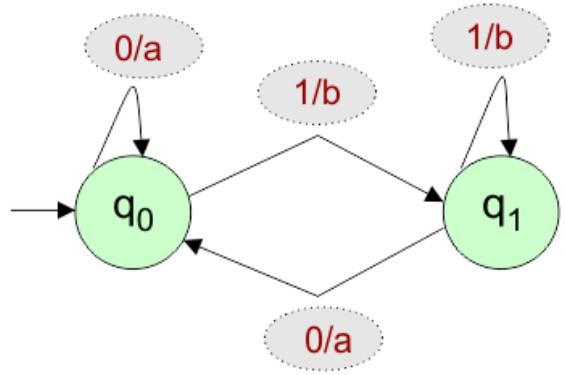



```
--Three Processes
architecture RTL of MOORE is
 signal current_state, next_state: <state_type>;
begin
  REG:process(clk, rst)
    begin
        if rst = '1' then
            current_state <= S0;
        elsif rising_edge(clk) then
            current_state <= next_state;
        end if;
    end process;
CMB_STATE: process(current_state, inputs) -- Combinational Process with Next State Logic
  Begin
     ...
  end process;
CMB_OUTPUT: process (current_state)        -- Combinational Process with Output Logic
  begin
     ...
  end process;
end RTL ;
```

Mealy FSM

Mealy Machine: Output depends on current state and input.






```
--Three Processes
architecture RTL of MEALY is
 signal current_state, next_state: <state_type>;
begin
  REG:process(clk, rst)
    begin
        if rst = '1' then
            current_state <= S0;
        elsif rising_edge(clk) then
            current_state <= next_state;
        end if;
    end process;
CMB_STATE: process(current_state, inputs) -- Combinational Process with Next State Logic
  Begin
     ...
  end process;
CMB_OUTPUT: process (current_state, inputs) -- Combinational Process with Output Logic
  begin
      ...
  end process;
end RTL ;
```

FSMs

Feature	Moore FSM	Mealy FSM
Output depends on	State only	State + Input
Number of states	More	Fewer
Output timing	On state change	Immediate (on input)
Complexity	Simpler	Slightly more complex
Reaction speed	Slower	Faster

Hands-on

Coffee Maker

ghdl_examples/coffee_maker