# Design units

Lecture 2

# Design units

- A digital system can be modelled in VHDL as an *entity*:
  - entity = the fundamental building block of any design
  - Or, any design consists of one entity or a set of entities
  - An entity can be a component of another project, or it can be the top level module of the design
- An entity is described as a set of VHDL design units.
- The design units can be compiled separately

# Design units

- In VHDL there are the following design units:
  1. Entity declaration (or simply entity)
  2. Architecture body (or architecture)
  3. Package declaration
  4. Package body
  5. Configuration declaration (configuration)
- Primary design units: entity declaration, configuration declaration, and  package declaration
- Secondary design units: architecture body and package body

# Entity declaration

It specifies *the name* of the entity *and* its *interface* (mainly the ports of the entity)

Syntax:

```
ENTITY_delcaration ::=
ENTITY name IS
        entity_header
        [entity_declarative_part]
[BEGIN
        entity_statement_part]
END [ENTITY][name];
Entity_header ::=
[GENERIC_clause][PORT_clause]
```

BNF description:

::= -is defined

[] – optional sequence

{} – optional repeatable sequence

| - logic alternatives

VHDL **is NOT** case sensitive

Convention: we will write the keywords in capital letters

# Entity declaration (cont’d)

- The entity declaration contains the keyword ENTITY followed by the name of the entity
- The header contains
  - the GENERIC clause:
    - Declares the generic parameters (or generics): belong to the class of constants and are visible in all architectures of that entity
    - The value of a generic parameter can be modified when the entity containing the generic is a component of a bigger entity.
    - Generics are used most often for specifying delays, but can have other utilizations
  - The PORT clause:
    - Declares the ports of the entity, i.e. the signals by which the entity communicates with the external world
    - The ports belong to the signal class
    - Each port has a mode (direction):
      - IN (for input ports), OUT for output ports and INOUT or BUFFER for bidirectional ports
- entity_declarative_part is optional
  - May contain declarations of constants, signals, types, subprograms

# Entity declaration (cont’d)

- entity_statement_part – also optional
  - Only passive processes or equivalent statements are allowed in an entity declaration
  - Passive = should not contain signal assignment statements, i.e., the statements from entities cannot assign (give) values to signals
  - Used mostly for constrains verifications (e.g. setup and hold time, forbidden input combinations, etc)
  - The behaviour of an entity cannot be described by its internal statements => we will need architectures for describing what an entity does !
- The entity declarations ends with the keyword END followed optionally by the keyword ENTITY or/and by the name of the entity;
  - The name of the entity, if it appears, should be the same like in the declaration
- The symbol ; at the end is mandatory
- See examples.

# Architecture body

Implementation of an entity is described in *architecture body*

An entity can have any number of architectures

```
 Architecure_body::=
          ARCHITECTURE architecture_name OF entity_name IS
                   architecture_declarative_part
          BEGIN
                   architecture_statement_part
          END [ARCHITECTURE] [architecture_name];
```

# Architectures

- Keyword ARCHITECTURE is followed by the architecture identifier (name) and the name of the associated  entity
- architecture declarative part
  - MAY be declared here : types, constants, signals, components, subprograms
  - MAY NOT be declared here: variables
- architecture_statement_part
  - The statements inside an architecture are concurrent (i.e., they execute in parallel) => their order is not important
- After END:
  - optional the keyword ARCHITECTURE and/or architecture_name (the same like at the beginning)
  - the symbol ; (mandatory)

# Statements in architectures

1. PROCESS :
   - A composed concurrent statement, inside which the statements are executed *sequentially*
   - Is the the basic statement for behavioural modelling (sequential)
2. BLOCK : composed statement, used mainly for dataflow modelling (concurrent)
3. Concurrent procedure calls
4. Concurrent ASSERT statements
5. Concurrent signal assignment statements
6. Component instantiation statements: for *structural modelling*
7. GENERATE statement (it is a macroinstruction)

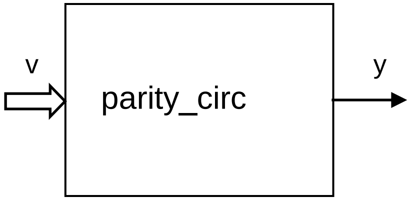
# Modelling styles

- In VHDL there exist 3 modelling styles:
  1. Behavioural modelling (sequential)
     - inside processes and subprograms
  2. Dataflow modelling (concurrent):
     - Statements 3-5 from the previous list
  3. Structural modelling: component instantiation statement
- Statements 2 and 7 are special cases, which do not belong to only one category
- In VHDL an architecture may contain any combination of the 3 modelling styles, however
  - We will discuss them separately for clarity
- We will show an example of a parity circuit modelled in the three styles

# Example [EKP98]

- We will describe a parity generator circuit. It has four inputs of type BIT and one output that is one when the input contains an even number of ‘1’.
- This word description serves as a specification of the circuit, which can be used for behavioural modelling.
- We don’t need the circuit diagram for the behavioural modelling of the circuit !
- Hence, we model the circuit at a high(er) level of abstraction.

# Behavioural modelling of the parity circuit



```
ENTITY parity_circ IS
       PORT(v: IN BIT_VECTOR(3 DOWNTO 0);
              y: OUT BIT);
END ENTITY;
----------------
ARCHITECTURE behavioural OF parity_circ IS
BEGIN
PROCESS(v)
-- signal v is the SENSITIVITY LIST of the process
-- PROCESS declarations are here:
-- We can declare constants, variables, types,
--subprograms,..., but
-- we MAY NOT declare signals !
       VARIABLE nr_of_1 : INTEGER:=0;
              -- counts the 1s in the vector v
```

# Behavioural modelling …

```
BEGIN
       nr_of_1:=0;
-- should be initialized every time when signal v changes,
-- which is, at each execution of the process
       FOR i IN 0 TO 3 LOOP
             IF v(i)='1' THEN
                    nr_of_1 := nr_of_1 +1;
             -- there are not operators like ++, +=, etc :)
             END IF;
       END LOOP;
       IF nr_of_1 MOD 2=0 THEN
             y<= '1' AFTER 1 ns;
       ELSE
             y<='0' AFTER 1 ns;
       END IF;
END PROCESS;
END;
```

# Behavioural modelling: comments

- Entity:
  - Is named *parity_circ*
  - It has an input *v* which is a 4-bit vector
  - It has an output *y* of type BIT
    - BIT is a predefined type, having the values ‘0’ si ‘1’
    - BIT_VECTOR is also a predefined type, as an ARRAY of BIT
- Architecture
  - Has no declarations in the declaration part
  - Contains a single statement: (PROCESS)

# Comments (cont’d)

- PROCESS
  - It has a **sensitivity list**, containing the signal *v*, meaning that the process is activated (resumes execution) every time when the signal *v* changes (we say that there is an **event** on *v*)
  - The sensitivity list may contain several signals; the process resumes each time when there is an event on one of the signals from the sensitivity list
  - The declaration part of the process contains a variable declaration: the variable nr_of_1 of type integer is declared here. It counts the number of 1s in the input vector.
  - In processes we **may declare variables**, constants, subprograms, types, subtypes, but
  - We **may not declare signals or components**
  - The statements inside a process (between BEGIN and END PROCESS) are executed sequentially (in the specified order)
  - Here the counter (nr_of_1) is incremented every time when a 1 is found in the input vector *v*; if the number of 1s in *v* is even, then the output *y* will be ‘1’ after one nanosecond (1 ns) delay, otherwise *y* will be ‘0’
  - The process is an infinite loop: after END PROCESS its execution will continue from BEGIN (if there are events on *v* !)

# Structural modelling

We represent the circuit diagram and we infer that we need XOR type gates and one INVERTER gate

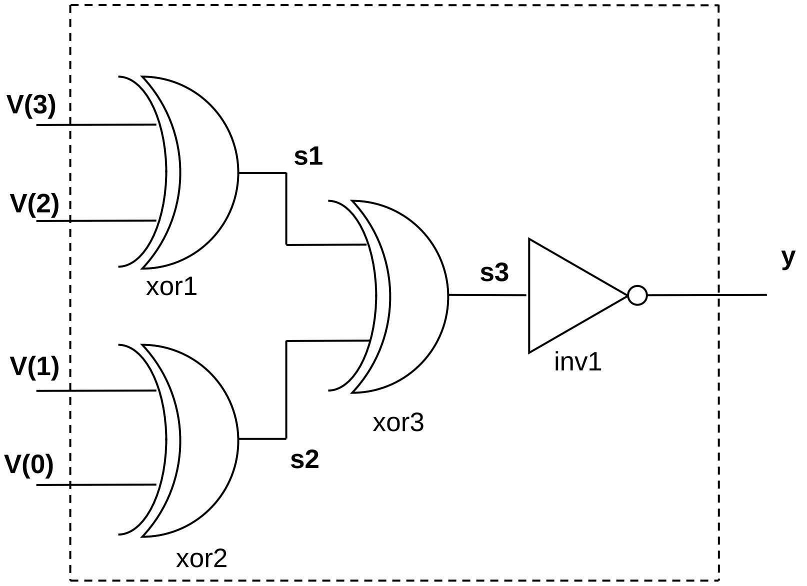
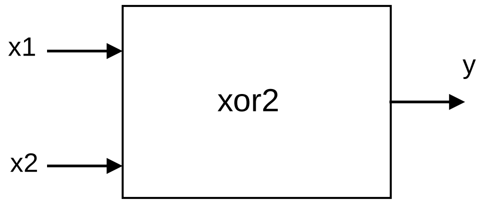


Fig 1.Circuit diagram for the parity generator circuit, implemented with XOR gates (after fig 2.3 from [EKP98]).

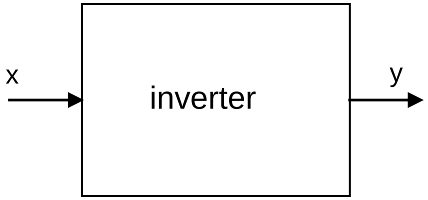
# Gates description: xor2

```
ENTITY xor2 IS
       GENERIC(del: TIME:=3ns);
       PORT(x1,x2: IN BIT;
              y: OUT BIT);
END xor2;

ARCHITECTURE behave OF xor2 IS
-- signal declarations, etc,
-- variabiles MAY NOT be declared in architecture !

BEGIN
       y <= x1 XOR x2 AFTER del;
END behave;
```



# Gates description : inverter

```
ENTITY inverter IS
       GENERIC(del: TIME:=4ns);
       PORT(x: IN BIT;
              y: OUT BIT);
END inverter;

ARCHITECTURE behave OF inverter IS
-- declarations
BEGIN
       y <= NOT x AFTER del;
END behave;
```



# Structural modelling of the parity circuit

```
ARCHITECTURE struct OF parity_circ IS
        COMPONENT xor_gate IS
                GENERIC(del: TIME:=3ns);
                PORT(x1,x2: IN BIT; y: OUT BIT);
        END COMPONENT;
        COMPONENT inv_gate IS
                GENERIC(del: TIME:=4ns);
                PORT(x: IN BIT; y: OUT BIT);
        END COMPONENT;
        SIGNAL s1, s2, s3: BIT;
BEGIN
xor1: xor_gate PORT MAP(y => s1, x2=> v(2), x1=> v(3));
        --  named association
xor2: xor_gate PORT MAP(v(1), v(0), s2);--positional association
xor3: xor_gate GENERIC MAP(del => 4ns) PORT MAP(s1, s2, s3);
inv1: inv_gate PORT MAP(x=>s3, y=>y);
END ARCHITECTURE struct;
```

# Structural description

- Contains the components of the design and their interconnections
- The behaviour of the system is not explicitly specified, but it results from the structure (similar to a circuit design)
- In the example, the architecture contains two component declarations and signal declarations in the declarative part
- The components are not active elements, but rather templates
- The declared signals are internal for the architecture
- In this example, the architecture body contains only *component instantiation statements*

# Structural description

- The syntax for the component instantiation statement: a mandatory label (the labels should be different inside an architecture), followed by the name of the component (same like in component declaration) and a PORT MAP clause
- PORT MAP makes an association between each *formal port* (the ports from the component declaration) and the corresponding *actual port* (internal signal or port of the modelled entity that is connected to the formal port).
- In VHDL there are rules concerning the types and mode (direction) of the formal and actual ports:
  - The type of the formal and actual port must match
  - A formal OUT port may not be connected to an IN actual port; neither can a formal IN port be connected to an actual OUT port
- The signals internal to an architecture have no mode; it means that they can be connected to both IN and OUT formal ports

# Structural description

- The PORT MAP can be:
  - Positional: the name of the formal port is not specified; the name of the actual port is specified on the position corresponding to the formal port (from the component declaration)
    - NOT recommended, it can generate errors that are very difficult to detect !
  - By name: it is specified both the name of the formal port and of the actual port
    - The order of ports is not important in this case
    - The syntax is always **formal_port => actual_port**
    - The mapping symbol => is from the formal port to the actual port, no matter if the ports are IN or OUT
- GENERIC MAP
  - can be used to modify the values of some generic parameters for a certain component instantiation

# Structural description

- In order to simulate a structural description, we need also the entity and architecture for each of the components used (xor_gate and inv_gate in our example).
- See *gates description* for this example
- If the associated entity has the same name (and the same ports and generics) like the component declared in an architecture, then the simulator can realize an association between them *(default binding)*
- If, like in our example, entities have other names than the components (which can happen if two different teams work to a design or if we use components from different libraries), then we need a configuration in order to realize the association (the binding)

# A configuration of the parity  circuit

```
CONFIGURATION cfg_parity_circ OF parity_circ IS
        FOR struct
               FOR ALL: xor_gate USE ENTITY WORK.xor2(behave);
               END FOR;
               FOR inv1: inv_gate USE ENTITY WORK.inverter(behave);
               END FOR;
        END FOR;
END CONFIGURATION;
```

# Configurations

- In example we have a *configuration declaration*
- There is also the *configuration specification,* which is declared inside the declaration part of an architecture
- The configuration declaration specifies the name  of the configuration and the associated entity
- Then it is specified the architecture for which is the configuration (an entity can have different architectures)
- Each component instantiation is associated with an entity(architecture) pair. In this example, from the current working library, with the logic name WORK.
- If the names of ports and/or generics differ between component and entity, the association between those names is made in the configuration (not in the example)
- VHDL is extremely reach concerning the configurations !

# Two dataflow architectures of the parity circuit

```
ARCHITECTURE dataflow1 OF parity_circ IS
       SIGNAL s1, s2, s3: BIT;
BEGIN
       y<=NOT s3 AFTER 11ns;
       s3<=s2 XOR s1;
       s2<=v(0) XOR v(1);
       s1<=v(3) XOR v(2);
END;

ARCHITECTURE dataflow2 OF parity_circ IS

BEGIN
       y<= NOT(v(0) XOR v(1) XOR v(2) XOR v(3)) AFTER 11ns;
END ARCHITECTURE;
```

# Dataflow modelling

- Both architectures from our example contain only concurrent signal assignment statements
- First architecture is very close to the structural description
- The statements in the architecture are concurrent, the order in which they appear is not important
- Second architecture contains only one statement
- Dataflow modelling is based on conditional or selected signal assignment statements.
- Each of them is equivalent with a process containing an IF, or respectively a CASE statement.

# PACKAGE

*Definition:* “a *package* is a collection of declarations such as subprograms, types, subtypes, constants, components, and possibly others, which are grouped in a way that allows different design units to share them” [EKP98].

PACKAGE declaration: is the interface to the package, containing those declarations that are made visible from outside. Its syntax is given bellow [EKP98]

Hidden details, that are not visible from outside, are grouped in the PACKAGE BODY.

A package declaration can have *at most* one package body.

Package body *must* exist if in package declaration appear subprogram declarations or *deferred constants*.

```
Package_declaration::=

PACKAGE name IS

        package_declarative_part

END [PACKAGE] [name];
```

```
Package_body::=

PACKAGE BODY name IS

        package_body_declarative_part

END [PACKAGE BODY] [name];
```

# PACKAGE

- If there is a PACKAGE BODY, then its name is the same with the name from package_declaration
- In order to make visible in a design unit the content of a package or only an  identifier, we must use the clause USE before that design unit. Two forms are possible:
  - USE package_name.ALL;
  - USE package_name.identifier;
- In the first case the entire content of the package becomes visible in the design unit, while in the second case, only the specified identifier (e.g. a type, a subprogram, a component, a constant, etc) is visible.
- **Deferred constants**: sometimes we want to declare a constant in a PACKAGE declaration without specifying its value => it must exist also a package body.
- The value of the constant will be given only in the package body
- This is useful if we want to change the value of the constant later on.
- If the constant is declared in the package declaration, all design units that use the package must be recompiled when we recompile the package declaration
- With deferred constants, we have to recompile only the package body.

# PACKAGE

- What to put in a package body:
  - Deferred constants
  - The definition of the subprograms declared in package declaration: a subprogram is only declared in package declaration, its body being given in the package body. Hence, the subprogram’s body is NOT visible from outside
  - Other items: for example subprograms that we use for implementing other subprograms, but we don’t want to make them visible from outside. Example: in package declaration we declare a function that adds two bit vectors resulting another bit vector. In order to implement the function (in the package body !) we need a function that converts bit-vectors to integers and another function, that converts integers to bit vectors. We can put those converting functions in the package body
- In VHDL there are two predefined packages:
  - STANDARD : contains standard types like BIT, BIT_VECTOR, integer, Boolean, real, character, string, time
  - TEXTIO: for working with text files
- Other packages of interest:
  - STD_LOGIC_1164 from the library IEEE : it extends the BIT type and the operations associated with it to a 9-value logic type.