



FPGA and VHDL

[VHDL]

- Hardware description language
- Used for describing digital design in general

[Entity]

- All designs are expressed in terms of entities. An entity is the most basic building block in a design. The uppermost level of the design is the top-level entity. If the design is hierarchical, then the top-level description will have lower-level descriptions contained in it. These lower-level descriptions will be lower-level entities contained in the top-level entity description.

[Architecture]

- All entities that can be simulated have an architecture description. The architecture describes the behavior of the entity. A single entity can have multiple architectures. One architecture might be behavioral while another might be a structural description of the design.

[Configuration]

- A configuration statement is used to bind a component instance to an entity-architecture pair. A configuration can be considered like a parts list for a design. It describes which behavior to use for each entity, much like a parts list describes which part to use for each part in the design.

[Package]

- A package is a collection of commonly used data types and subprograms used in a design. Think of a package as a toolbox that contains tools used to build designs.

[Driver]

- This is a source on a signal. If a signal is driven by two sources, then when both sources are active, the signal will have two drivers.

[Bus]

- The term “bus” usually brings to mind a group of signals or a particular method of communication used in the design of hardware. In VHDL, a bus is a special kind of signal that may have its drivers turned off.

[Attribute]

- An attribute is data that are attached to VHDL objects or predefined data about VHDL objects. Examples are the current drive capability of a buffer or the maximum operating temperature of the device.

[Generic]

- A generic is VHDL's term for a parameter that passes information to an entity. For instance, if an entity is a gate level model with a rise and a fall delay, values for the rise and fall delays could be passed into the entity with generics.

[Process]

- A process is the basic unit of execution in VHDL. All operations that are performed in a simulation of a VHDL description are broken into single or multiple processes.

[Describing Hardware in VHDL]

- VHDL Descriptions consist of primary design units and secondary design units.
- The primary design units are the Entity and the Package.
- The secondary design units are the Architecture and the Package Body.
- Secondary design units are always related to a primary design unit.
- Libraries are collections of primary and secondary design units.
- A typical design usually contains one or more libraries of design units.

[Entity]

- A VHDL entity specifies the name of the entity, the ports of the entity, and entity-related information.
- All designs are created using one or more entities.
- Example:

```
ENTITY mux IS
    PORT ( a, b, c, d : IN BIT;
          s0, s1 : IN BIT;
          x, : OUT BIT);
END mux;
```

[Architectures]

- The architecture describes the underlying functionality of the entity and contains the statements that model the behavior of the entity.
- An architecture is always related to an entity and describes the behavior of that entity

[Architecture Example]

```
ARCHITECTURE dataflow OF mux IS
  SIGNAL select : INTEGER;
BEGIN
  select <= 0 WHEN s0 = '0' AND s1 = '0' ELSE
    1 WHEN s0 = '1' AND s1 = '0' ELSE
    2 WHEN s0 = '0' AND s1 = '1' ELSE
    3;
  x <= a AFTER 0.5 NS WHEN select = 0 ELSE
    b AFTER 0.5 NS WHEN select = 1 ELSE
    c AFTER 0.5 NS WHEN select = 2 ELSE
    d AFTER 0.5 NS;
END dataflow;
```

Concurrent Signal Assignment

- In a typical programming language such as C or C++, each assignment statement executes one after the other and in a specified order.
 - The order of execution is determined by the order of the statements in the source file.
- Inside a VHDL architecture, there is no specified ordering of the assignment statements.
 - The order of execution is solely specified by events occurring on signals that the assignment statements are sensitive to.

```
select <= 0 WHEN s0 = '0' AND s1 = '0' ELSE
        1 WHEN s0 = '1' AND s1 = '0' ELSE
        2 WHEN s0 = '0' AND s1 = '1' ELSE
        3;
```

[Event Scheduling]

- The assignment to signal **x** does not happen instantly.
- Each of the values assigned to signal **x** contain an **AFTER** clause.
- The mechanism for delaying the new value is called scheduling an event.
- By assigning port **x** a new value, an event was scheduled 0.5 nanoseconds in the future that contains the new value for signal **x**.
 - When the event matures (0.5 nanoseconds in the future), signal **x** receives the new value.

```
x <= a AFTER 0.5 NS WHEN select = 0
```

[Statement Concurrency]

- The first assignment is the only statement to execute when events occur on ports `s0` and `s1`
- The second signal assignment statement does not execute unless an event on signal **select** occurs or an event occurs on ports **a**, **b**, **c**, **d**.
- The two signal assignment statements in architecture **behave** form a behavioral model, or architecture, for the **mux** entity.

[Structural Designs]

- Another way to write the **mux** design is to instantiate subcomponents that perform smaller operations of the complete model.
- With a model as simple as the 4-input multiplexer that we have been using, a simple gate level description can be generated to show how components are described and instantiated.

[Example Structural Design]

```
ARCHITECTURE netlist OF mux IS
  COMPONENT andgate
    PORT(a, b, c : IN bit; c : OUT BIT);
  END COMPONENT;
  COMPONENT inverter
    PORT(in1 : IN BIT; x : OUT BIT);
  END COMPONENT;
  COMPONENT orgate
    PORT(a, b, c, d : IN bit; x : OUT BIT);
  END COMPONENT;
  SIGNAL s0_inv, s1_inv, x1, x2, x3, x4 : BIT;
BEGIN
  U1 : inverter(s0, s0_inv);
  U2 : inverter(s1, s1_inv);
  U3 : andgate(a, s0_inv, s1_inv, x1);
  U4 : andgate(b, s0, s1_inv, x2);
  U5 : andgate(c, s0_inv, s1, x3);
  U6 : andgate(d, s0, s1, x4);
  U7 : orgate(x2 => b, x1 => a, x4 => d, x3 => c, x => x);
END netlist;
```

[Sequential Behavior]

- The third way to describe the functionality of the `mux` is to use a process statement to describe the functionality in an algorithmic representation.
- The architecture contains only one statement, called a process statement.
- It starts at the line beginning with the keyword **PROCESS** and ends with the line that contains **END PROCESS**.
- All the statements between these two lines are considered part of the process statement.

```

ARCHITECTURE sequential OF mux IS
    (a, b, c, d, s0, s1 )
    VARIABLE sel : INTEGER;
    BEGIN
    IF s0 = '0' and s1 = '0' THEN
        sel := 0;
    ELSIF s0 = '1' and s1 = '0' THEN
        sel := 1;
    ELSIF s0 = '0' and s1 = '1' THEN
        sel := 2;
    ELSE
        sel := 3;
    END IF;
    CASE sel IS
        WHEN 0 =>
            x <= a;
        WHEN 1 =>
            x <= b;
        WHEN 2 =>
            x <= c;
        WHEN OTHERS =>
            x <= d;
    END CASE;
    END PROCESS;
END sequential

```

[Process Statements]

- The process statement consists of a number of parts.
 - The first part is called the sensitivity list;
 - The second part is called the process declarative part;
 - The third is the statement part.

[Sensitivity List]

- The list of signals in parentheses after the keyword **PROCESS** is called the sensitivity list.
- This list enumerates exactly which signals cause the process statement to be executed.
 - Here, the list consists of **a**, **b**, **c**, **d**, **s0**, and **s1**.
- Only events on these signals cause the process statement to be executed.

```
ARCHITECTURE sequential OF mux IS  
    (a, b, c, d, s0, s1 )
```

[Process Declarative Region]

- The process declarative part consists of the area between the end of the sensitivity list and the keyword **BEGIN**.
- Here, the declarative part contains a variable declaration that declares local variable **sel**. This variable is used locally to contain the value computed based on ports **s0** and **s1**.

[Process Statement Part]

- The statement part of the process starts at the keyword **BEGIN** and ends at the **END PROCESS** line.
- All the statements enclosed by the process are sequential statements.
- This means that any statements enclosed by the process are executed one after the other in a sequential order just like a typical programming language.
 - Remember that the order of the statements in the architecture did not make any difference; however, this is not true inside the process.
 - The order of execution is the order of the statements in the process statement.

[Process Execution]

- Assume that **s0** changes to 0.
- Because **s0** is in the sensitivity list for the process statement, the process is invoked.
- Each statement in the process is then executed sequentially.
- In this example the **IF** statement is executed first followed by the **CASE** statement.
- Each check that the **IF** statement performs is done sequentially starting with the first in the model.

[Sequential Statements]

- This statement will execute sequentially. Once it is executed, the next check of the **IF** statement is not performed.
 - Whenever a check succeeds, no other checks are done.
- The **CASE** statement will evaluate the value of **sel** computed earlier by the **IF** statement and then execute the appropriate statement that matches the value of **sel**.
 - For example, if the value of **sel** is 1 therefore the following statement will be executed:
`x <= b;`
- The value of port **b** will be assigned to port **x** and process execution will terminate because there are no more statements in the architecture.

Architecture Selection

- Which architecture should be used to model the **mux** device?
- If the model is going to be used to drive a layout tool, then the structural architecture netlist is probably most appropriate.
- If not, behavioral and sequential are probably more efficient in memory space required and speed of execution.
- Would the modeler rather write concurrent or sequential VHDL code?
- Typically, modelers are more familiar with sequential coding styles, but concurrent statements are very powerful tools for writing small efficient models.

[Configuration Statements]

- The configuration statement maps component instantiations to entities.
- With this powerful statement, the modeler can pick and choose which architectures are used to model an entity at every level in the design.
- The function of the configuration statement is to spell out exactly which architecture to use for every component instance in the model.
- This occurs in a hierarchical fashion. The highest-level entity in the design needs to have the architecture to use specified, as well as any components instantiated in the design.

[Example Configuration]

```
CONFIGURATION muxcon1 OF mux IS
  FOR netlist
    FOR U1,U2 :
      inverter USE ENTITY WORK.myinv(version1);
    END FOR;
    FOR U3,U4,U5,U6 : andgate USE ENTITY
      WORK.myand(version1);
    END FOR;
    FOR U7 : orgate USE ENTITY WORK.myor(version1);
    END FOR;
  END FOR;
END muxcon1;
```

[Power of Configurations]

- By compiling the entities, architectures, and the configuration specified earlier, you can create a simulatable model.
 - what if you did not want to simulate at the gate level?
 - What if you really wanted to use behavioral model instead?
- The power of the configuration is that you do not need to recompile your complete design
 - you only need to recompile the new configuration.