§ 2 Basic Features of VHDL

§ 2.1 VHDL Background Information

1. What is a HDL?

   A high level programming language that offers *special constructs* with which you can model microelectronic circuits.

   The special constructs allow you to:

   a. Describe the operation of a circuit at various levels of
      i. The behavior abstraction of a circuit
      ii. The function abstraction of a circuit
      iii. The structure abstraction of a circuit

   b. Timing of a circuit

   c. Concurrency of circuit operation

2. Why HDL?

   a. HDL facilitates a top-down design methodology using synthesis
      i. Design at high implementation-independent level
      ii. Delay decision on implementation details
      iii. Easily explore design alternatives
      iv. Solve architecture problems before implementation
      v. Automatic mapping of a high-level description to a technology specific implementation

   b. Provide greater flexibility
      i. Design reuse
      ii. Move design between multiple vendors' tools

   c. Permits you to take advantage of mature software design practice
      i. Quickly capture design intent
      ii. Quickly manage design data

      Or in other words, documenting a design and modeling it.

   d. Prototyping of complicated system is extremely expensive

      ➔ Replace the prototyping process with validation through simulation

      ➔ HDLs can be used for both logic synthesis and test generation

3. HDLs

   Early HDLs (CDL, ISP, AHPL), mainly in 70's, primarily target at design architecture verification

   ➔ Can't model designs with high accuracy

   ➔ Can't provide precise timing

   ➔ Language construct imply a certain hardware structure

Newer HDLs (VHDL, Verilog) have universal timing model and imply no particular hardware structure.

4. VHDL history
    a. Begin in 1983 US DOD sponsored the development of VHSIC (very high speed integrated circuit) HDL (VHDL) program (Intermetrics, IBM, TI)
        ➔ Original intent: a means of communicating designs among contractors in the VHSIC program
    b. First major stage of language development in August 1985 on the release of version 7.2
    c. IEEE sponsored further development
        ➔ Goal: The development of an improved standard version of the language
        ➔ In May, 1987, LRM (Language Reference Manual) released for industrial review
        ➔ In December 1987, the version of VHDL became IEEE 1076-1987 standard and official
        ➔ From 1988 to 1992, minor changes incorporated, and balloted in 1993
        ➔ In 1994, revised standard (VHDL 1076-1993)

§ 2.2 VHDL Basic

1. How to detect errors?
    ➔ Simulate and test on different levels
2. The two HDL languages can do:
    ➔ Behavioral level design (very high level)
    ➔ Dataflow level design (RTL level)
    ➔ Structural level design (Gate level)
3. Two programs need to write in order to simulate and test
    ➔ What you want?
        The VHDL code for components
    ➔ How you test / verify?
        VHDL code for testing, i.e., testbench
4. If okay, completes logic design
    Technology library in this course for synthesis: 0.18 CMOS technology

§ 2.3 Major Language Constructs

1. Design example: A circuit that counts the number of 1's in an input vector of three bits.

2. Design Entities

   In VHDL, a given *logic circuit* is a design entity

   Design entity consists of two different types of descriptions:

Interface description	+	Architecture bodies
(Only one)		(One or more)

3. Interface description

   Declares the entity and describes its inputs and outputs.

   Also, this is a place where documentation information about the nature of the entity can be recorded.

   -- starts comments, for any line

4. Architecture bodies

   Specifies either the behavior of the entity or a structural decomposition of the entity using more primitive components

   Step 1: At the beginning of the design process

   Algorithm in mind that would like to implement

   ➔ Check its accuracy of the algorithm

➔ Implementation detail not specified

➔ Behavioral body

i. Describe the operation of the algorithm perfectly.

ii. Correspondence to real hardware is weak.

5. Advance to logic design stage

6. Structured design

   Level 1 partitioning:

      Majority function (MAJ)

      Odd-parity function (OPAR)

   Level 2 partitioning:

      Basic gates, e.g., AND2, OR2, AND3, OR4, INV

   Proceed with the design and implementation of **MAJ**

   1. Construct building blocks

   2. construct **MAJ**

3. construct **OPAR** (omitted)

4. construct **Top Level Entity**

§ 2.4 Model testing and testbench

1. VHDL model must be tested.
   - ➔ Done by forming a top-level entity
   - ➔ Naming convention:
2. The entity declaration for testbench contains NO PORT statement because the test signals are generated internally to the testbench.
3. Within the testbench architecture, the test input(s) and test output(s) for the entity are declared as signals.
4. Next, components declaration and binding to that in the design library.
5. After BEGIN, component is instantiated and mapped to PORT signals.
6. Finally, one or several processes contain a sequence of test vectors to drive the ENTITY.
   - ➔ Each process runs just once at the beginning of simulation and then suspended (wait).

§ 2.5 Other VHDL program modules

1. Block statements

(i) A block is a bounded region of text that contains a declaration section and an executable section.

➔ Architecture body itself is a block

➔ Within an architecture body or block body, internal blocks can exist.

```
A: BLOCK
        … …   -- Block A declaration section
        … …
BEGIN
        … …   -- Block A executable section
        … …
END BLOCK A;

B: BLOCK
        … …   -- Block B declaration section
        … …
BEGIN
        … …   -- Block B executable section
        … …
END BLOCK B;
```

(ii) Why block?

a) It supports a natural form of design decomposition;

b) A “guarded” condition can be associated with a block when a guard condition is true. It enables certain types of statements inside the block

```
e.g.,   D0: BLOCK (clk = '1' AND NOT clk'STABLE)
                q <= GUARDED d;
        END BLOCK D0;
```

c) A guarded statement will be executed when
   (1) The guard is TRUE and a signal on the RHS of the guarded statement changes
   (2) The guard changes from FALSE to TRUE

d) Very useful for modeling register primitives.

2. Processes
   (i) Begin with keyword **PROCESS (A)**.
   (ii) **A** is called sensitivity list of the process.
   (iii) When a signal in the sensitivity list changes, the process is activated and statements within the process block are executed.
   (iv) Questions? How about **PROCESS ( )**?

```
Label: PROCESS (Sensitivity List)
        -- Constant declaration
        -- Variable declaration
        -- Subprogram declaration
        -- Signals are NOT permitted!
BEGIN
        --
        -- sequential statements
        --
END PROCESS Label;
```

   (v) Every process statement is executed once at the beginning of the simulation. Thereafter, only when a signal in the sensitivity list changes value (when there is an event on one or more signals), will the process be executed again.
   (vi) Variables within processes are static
      ➔ Initialized only once at the beginning of the simulation
      ➔ Retain their values between process activations.

3. Data Types (Details self-study)
   (i) Scalar (Consist of a single element)
      a) Enumeration (discrete): A type whose values are defined by simply listing them in an ordered list.

         Pre-defined enumeration data type in language:
```
e.g.:   TYPE BIT IS ('0', '1');
        TYPE BOOLEAN IS (FALSE, TRUE);
        TYPE TRISTATE IS ('Z', '0', '1');
```
         User-defined enumeration data type:
```
e.g.:   TYPE state IS (s0, s1, s2, s3);
```
         Type STD_ULOGIC: nine value type defined by IEEE standard 1164, which is important for synthesis

         STD_LOGIC is a subtype of STD_ULOGIC, with an associated resolution function, is the type actually used for signal and variable declaration.

```
TYPE STD_ULOGIC IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

         If the same identifier or character literal is declared for more than one enumeration type, it is said to be **overloaded**.

      b) Integer: discrete, numeric
      c) Physical: numeric
      d) Floating point (or real): numeric
   (ii) Composite
      a) Array – All elements have the same type, homogeneous
```
TYPE Reg_32 IS Array (31 DOWNTO 0) OF BIT;
SUBTYPE my_byte IS BIT_VECTOR (7 DOWNTO 0);
```
         Arrays can have tick attributes to describe:
```
e.g.,   SIGNAL dbus: BIT_VECTOR (15 DOWNTO 0);

        dbus'RIGHT = 0;
        dbus'LEFT = 15;
        dbus'LENGTH = 16
```
      b) Record – Elements may have different types.
   (iii) Access

      Type that provides access to other types
   (iv) File

      Provide access to data files.

4. Language Statement

There are two classes of language statements: sequential statements and concurrent statements.

Sequential Statements include processes and subprograms. It is like high-level software languages such as C/C++. It is mainly used to describe algorithms.

Current statements are included in the architecture bodies. It is mainly used to model signals. All concurrent statements are executed once at the beginning of the simulation. The order in which the concurrent statement appear in the architecture body is not important. It is executed only when the RHS signals generate events.

(i) Assignment statement

a) Variable assignment (**:=**) Variable resumes its new value instantaneously.
b) Signal assignment (**<=**) will schedule a new value for a signal to assume at some future time. The current value of signal is never changed by a signal assignment statement.
   If no specific time value is specified, the default is an infinite small value (**δ, delta delay**).
   For example:
c) The base type of the value assigned to a signal must be the same as the base type declared for the signal.
d) Transaction representation:
   They are represented as a value-time pair in parenthesis, i.e., (value, time).
   For example:

(ii) Signal Drivers

If a process contains one or more signal assignment statement for a signal, the VHDL simulator creates a single value holder called a signal driver (for a signal).

Signal driver maintains an ordered list of scheduled value assignments for the signal.

Each scheduled signal assignment is called a **Transaction**. New value assigned to a signal by a transaction may/may not be different from the current value.

If signal undergoes a change in value due to a transaction, an **Event** occurs.

If more than one process contains signal assignment statement for the same signal, simulator creates a separate driver for the signal for each such process.

➔ At any given time, there may be more than one driver (each associated with a different process) scheduling values for the signal.

➔ The problem is solved by specifying a resolution function.

e.g., SIGNAL a: wired_or BIT;

5. Characterizing Hardware Language
   (i) Timing and concurrency are main characteristics of HDLs.
   (ii) Why need timing?
      ➔ Because value transfer is done through wired or busses.
      ➔ Signals in VHDL represent real wire, there is delay associated with value transfer through wires
      ➔ Hence, we need timing.
   (iii) How delay comes?
      ➔ Wire has capacitance → sometimes proportional to the length
      ➔ Wire capacitance + pull-up / pull-down resistance → propagation delays through wires
      ➔ Delay depends on technology, material, and size.
   (iv) Software → sequential manner
   Hardware → interconnection of components → concurrently active → VHDL simulator makes user think the execution is done concurrently.

6. Delay Modeling
   (i) The way delays are handled in signal assignments.
   (ii) Signal assignments can have **inertial** or **transport** delays.
   (iii) The **inertial** delay can have an additional reject specification.
   (iv) Default delay mechanism for signal assignment is **inertial**.
      ➔ **Transport** delay must be explicitly specified.
   (v) **Inertial** delay can be used to model capacitive networks.
      For example:

Delays through the capacitive networks and through gates with threshold values can be more accurately modeled with an inertial delay and a pulse rejection value that is less than the value of the inertial delay.

➔ **Target1 <= REJECT 3 NS INERTIAL waveform AFTER 5 NS;**

Pulse of 3 NS will be rejected!

➔ **Target2 <= waveform AFTER 5 NS;**

If a pulse whose width is less than 5 NS occurs on the waveform, it will be rejected and does not appear on target2

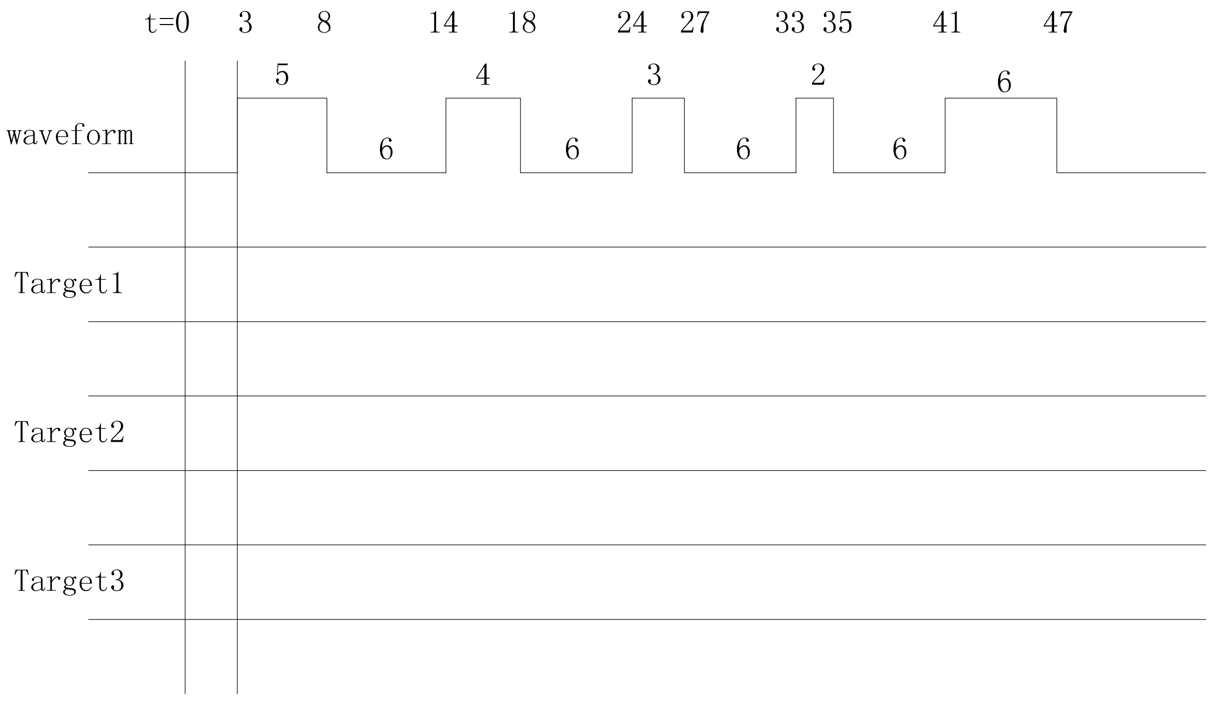
Pulse of exactly 5 NS will not be rejected.

(vi) Transport delay

It is used to model delay through transmission lines and networks with virtually infinite frequency response.

➔ **Target3 <= TRANSPORT waveform AFTER 5 NS;**

```
waveform  <=    '1' AFTER 3 NS, '0' AFTER 8 NS, '1' AFTER 14 NS,
                '0' AFTER 18 NS, '1' AFTER 24 NS, '0' AFTER 27 NS,
                '1' AFTER 33 NS, '0' AFTER 35 NS, '1' AFTER 41 NS,
                '0' AFTER 47 NS;
```



(vii) Example:

```
ARCHITECTURE test OF example IS
        SINGAL a, b, c: BIT := ‘0’;
BEGIN
        a <= ‘1’ AFTER 15 NS;
        b <= NOT a AFTER 5 NS;
        c <= a AFTER 10 NS;
END test;
```

7. Delta Delay (δ)

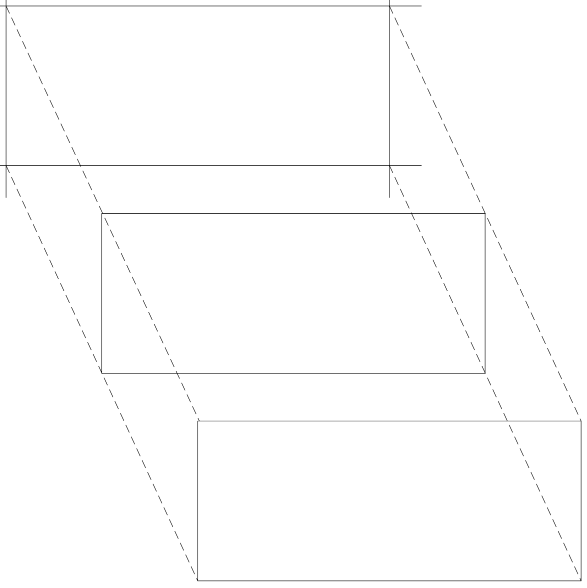
Used internally in HDL simulators to model hardware concurrency.

```
ARCHITCTURE test OF concurrent IS
        SIGNAL a, b, c : BIT := ‘0’;
BEGIN
                a <= ‘1’;
                b <= NOT a;
                c <= NOT b;
        END test;
```

8. Sequential placement of transactions
   - (i) Signal assignment can be placed in the sequential body
     - ➔ The order in which signal assignments appear is important
     - ➔ It is legal to make multiple assignments to unresolved signal
     - ➔ New transaction may be placed while old transaction not expired
     - ➔ Whether to **overwrite** or **append** depends
       - ① Timing of new transaction
       - ② Type of the assignment
       - ③ Value of signal

   - (ii) Rules

	Transport Delay	Inertial Delay
New Transaction is BEFORE already Existing		
New Transaction is AFTER already Existing		



(iii) Examples:

Example 1:

```
ARCHITECTURE sequential OF example IS
        SIGNAL x : RIT := ‘Z’;
BEGIN
        PROCESS ( )
        BEGIN
                x <= ‘1’ AFTER 5 NS;
                x <= TRANSPORT ‘0’ AFTER 3 NS;
                WAIT;
        END PROCESS;
END sequential;
```

Example 2:

```
x <= ‘1’ AFTER 5 NS;
x <= TRANSPORT ‘0’ AFTER 8 NS;
WAIT;
```

Example 3:

```
x <= ‘1’ AFTER 5 NS;
x <= ‘0’ AFTER 3 NS;
WAIT;
```

Example 4:

```
x <= ‘0’ AFTER 5 NS;
x <= ‘0’ AFTER 8 NS;
WAIT;
```

Example 5:

```
x <= ‘1’ AFTER 5 NS;
x <= REJECT 2 NS INERTIAL ‘0’ AFTER 8 NS;
WAIT;
```

Example 6:

```
x <= ‘1’ AFTER 5 NS;
x <= REJECT 4 NS INERTIAL ‘0’ AFTER 8 NS;
WAIT;
```

Example 7:

```
ARCHITECTURE current OF example IS
        SIGNAL a, b : BIT ;
BEGIN
       a <= ‘0’, ‘1’ AFTER 5 NS, ‘0’ AFTER 10 NS, ‘1’ AFTER 15 NS;
        b <= ‘0’, a AFTER 3 NS;
END current;
```

Example 8:

```
ARCHITECTURE current OF example IS
        SIGNAL a, b, c : RIT := ‘0’;
BEGIN
        a <= NOT a AFTER 10 NS WHEN NOW <= 25 NS ELSE a;
        b <= ‘1’, a AFTER 20 NS, ‘0’ AFTER 35 NS;
        c <= ‘Z’, a AFTER 5 NS, NOT b AFTER 10 NS;
END current;
```

9. Sequential Control Statements

➔ Used in processes and subprograms to define algorithms

➔ Execute in order of appearance

(i) WAIT Statement

➔ Not supported in synthesis, useful in testbench

➔ E.g.:

```
WAIT ON x, y UNTIL z = 0 FOR 100 NS;
```

Process resumes after 100 NS, or when an event occurs on x or y with z = 0, whichever comes first.

```
WAIT;
```

Wait forever, permanent suspend process

(ii) IF Statement

➔ Format:

```
IF condition1 THEN
        -- Sequence of statements under condition1
ELSIF condition2 THEN
        -- Sequence of statements under condition2
-- Any number of ELSIF clauses
ELSE
        -- Sequence of statements under all other conditions
END IF;
```

➔ Note: (1) Always has else statement;

(2) Notice condition has priority

(3) Can be nested.

➔ E.g., Positive edge triggered D-FF with asynchronous set/reset.

```
ENTITY d_ff IS
        PORT (d, set, rst, clk : IN BIT; q, qb : OUT BIT);
END d_ff;

ARCHITECTURE behavioral OF d_ff IS
```

```
        END behavioral;
```

(iii) CASE Statement

➔ Format:

```
        CASE expression IS
                WHEN choice 1 =>
                        -- Sequence of statements under choice 1
                WHEN choice 2 =>
                        -- Sequence of statements under choice 2
                … …
                WHEN OTHERS =>
                        -- Last sequence of statements
        END CASE;
```

➔ Note:

1. All possible choices for values must be included exactly once
2. The sets of choices must be mutually exclusive
3. Give equal weight to each choice
4. Useful in state machine
5. Can be nested with **IF statement**

➔ E.g.: Microwave oven controller

10. Subprograms

➔ Functions and procedures

➔ Function computes and return a value to invoking expression, does not modify any of its arguments.

➔ Procedures are both sequential and concurrent statements.

Don't return a value to the invoking program.

May / May not modify their arguments.

(i) Functions

➔ Things to specify

Name, input parameters, type of return value, and algorithm to compute the returned value. Any declaration if needed.

➔ Format:

```
FUNCTION name (Formal Parameters) RETURN type IS
        -- Parameters always as IN type
        -- constant, variable
        -- No Signal Allowed
BEGIN
        -- sequential statements
        RETURN (value);
END name;
```

➔ E.g., the majority function in the 1's counter

(ii) Procedures

➔ Can modify one or more input parameters

➔ Things to specify

Name, input and **output** parameters, algorithm, any algorithm if needed.

➔ Format:

```
PROCEDURE name (Formal Parameters)
        -- Parameters can be IN, OUT, INOUT type
        -- constant, variable
        -- No Signal Allowed
BEGIN
        -- sequential statements
END name;
```

➔ E.g., Procedure to add

➔ When invoking, actual parameters are associated with formal parameters using wither positional association list or a named association list. Base type must be matched.

(i) Overloading

➔ The capability to change the meaning of literals and the names of operators, functions and procedures by re-declarations.

➔ E.g., F <= A AND B; A, B : QIT (‘X’, ‘0’, ‘1’, ‘Z’)

AND operator is defined for BIT and Boolean, does not support QIT

➔ One way to do is write function for the operation and use A, B as arguments to the function: F <= FUNC_AND (A, B);

→ Not convenient, difficult to read, especially under complicated expressions

➔ Better way to do: overload AND (OR) operators

```
FUNCTION  “AND” (a, b : QIT)  RETURN QIT IS
        TYPE qit_table IS ARRAY(QIT, QIT) OF QIT;
        CONSTANT qit_and_table : qit_table :=
                --      0      1      X      Z
                   (( ‘0’    ‘0’    ‘0’    ‘0’),   -- 0
                    ( ‘0’    ‘1’    ‘X’    ‘X’),   -- 1
                    ( ‘0’    ‘X’    ‘X’    ‘X’),   -- X
                    ( ‘0’    ‘X’    ‘X’    ‘X’));  -- Z
BEGIN
        RETURN qit_and_table (a, b);
END “AND”;

F <= A AND B;
```

➔ Is the built-in AND operator still available for the use of BIT and Boolean?

→ Yes, VHDL analyzer can infer which operator is required in an expression by the parameter profiles and result profiles for the operations having the same name.

Parameter profiles: the number, order, type of operand

Result profiles: the type of returned result

➔ Subprogram name can also be overloaded.

12. Package

It’s tedious to repeat declarations each time.

→ VHDL provides package to hold frequently used declarations

→ Must be declared before it can be used.

→ IEEE has developed standard 1164 as a standard nine-value logic system, particularly for logic synthesis

STD_LOGIC_1164	Basic value system and associated functions
NUMERIC_STD	Overloaded arithmetic and other operators for synthesis

→ Vendors package (Synopsys)

STD_LOGIC_ARITH
STD_LOGIC_UNSIGNED
STD_LOGIC_SIGNED

→ Declared own package

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
PACKAGE my_package IS
        TYPE qit IS (‘X’, ‘0’, ‘1’, ‘Z’);
        SUBTYPE rit IS qit RANGE ‘0’ TO ‘Z’;
        TYPE qit_table IS ARRAY (qit, qit) OF qit;
        PROCEDURE int2qit (int: IN INTEGER; qin: OUT qit_vector);
        FUNCTION “AND” (a, b: qit) RETURN qit;
END my_package;

PACKAGE BODY my_package IS
        FUNCTION “AND” (a, b: qit) RETURN qit IS
                -- Statements for “AND” function
        END “AND”;

        -- Other procedures;
END my_package;
```

→ When use your package

1. Compile the package file
2. In your component, specify “use *lib_name.pkg_name.element_name*;
   For example: USE work.my_package.all;

13. Configuration

→ VHDL structural architectures are developed by instantiating components that are declared in the declaration region of the architecture

→ Before a structural model can be simulated, each instance component must be bound to a library model

→ One way to do: place configuration specification statements directly in the structural model

E.g.: FOR ALL: inv USE entity work.inv (dataflow);

→ Another approach

Through the use of configuration declaration:

- Component instances in the structural architecture are left unbound, and the component specification statements are collected in a separate analyzable unit called a configuration declaration.
- Can be placed at the end of VHDL model or in a separate .vhd file
- Must be analyzed after the architecture they are configuring
- When the model is being simulated, it is the configuration declaration that is bound to the component being tested in the testbench
- Very useful when multiple simulations of the same structural architecture with the components bound to different library components for each simulation
    - Structural architecture only need to analyzer once
    - Analyze different configuration results in different structure
    - Reduce errors and analysis time and promotes reuse of the models.

14. File I/O

Straightforward, refer to the sample code provided.

15. Signal Attributes

→ Used for signal objects for finding events, transactions, or timing of events or transactions on signals

→ Very useful when model H/W properties

→ Attributes deal with events on a signal:

’STABLE ➔ return Boolean Signal

’EVENT, ’LAST_EVENT, ’LAST_VALUE ➔ return a value

→ Attributes deal with transactions on a signal:

’QUIET ➔ return Boolean Signal

’ACTIVE ➔ return Boolean value

’LAST_ACTIVE ➔ return a Time value

’TRANSACTION ➔ return a Bit signal

→ Common applications of signal attributes include:

Edge detection, pulse width verification, glitch detection, etc.

→ Although s’EVENT and NOT s’STABLE are equivalent in most cases, ’STABLE generates a signal, it is recommended in concurrent statements

→ Model Setup Time and Hold Time

Setup Time (Before the clock edge): Minimum required time between changes on the data input and the triggering edge of the clock

Hold Time (After the clock edge): Minimum time that data input of a FF should stay stable after the effective edge of the clock.

Setup: (clock = '1' AND NOT clock' STABLE) AND (data' STABLE (setup_time))

Clock rise from 0 to 1 (clock' EVENT)	Data input has been stable at least for The amount of setup time

Hold: (data' EVENT) AND (clock = '1') AND (clock' STABLE (hold_time))

There is a change on data input (NOT data' STABLE)	Logic clock value is '1'	Clock has got a new value more recent than the account of hold value

16. Generic parameter

→ Parameterize component models to better utilize gates or component models in different design environment

→ The specific behavior of those models depends on the parameters that are determined by the time they are used

→ For example: the S-R latch in the lab

It can be constructed using NAND2 gates, to make it work, the gates should have different delay parameters. Where to specify the delay parameters?

➔ Choice 1: Different code for each gate with different delay 🡄 Inconvenient

➔ Choice 2: Use generic, delays are given when they are used

→ GENERIC is used as a means of communicating non-hardware and non-signal information between designs, such as timing and delay

→ Similar to the PORT, we use GENERIC MAP

→ For example: the NAND2

```
ENTITY nand2 IS
        GENERIC (tplh: TIME := 6 ns; tphl: TIME := 4 ns);
        PORT (a, b: IN BIT; o: OUT BIT);
END nand2;

ARCHITECTURE ave_delay OF nand2 IS
BEGIN
        O <= a AND b AFTER (tplh+tphl) / 2;
END ave_delay;
```

→ Default values will be used for tplh and tphl if they are not specified by another method

→ Default values will be overwritten if new generic values are specified when the new component is used

→ GENERIC is used to pass values (generic parameters) through components.

→ The S-R latch example:

```
        ARCHITECTURE default OF sr_latch IS
                COMPONENT n2
                        PORT (a, b: IN BIT; o: OUT BIT);
                END COMPONENT;
                FOR ALL: n2 USE ENTITY work.nand2 (ave_delay);
                SIGNAL im1, im2, im3, im4: BIT;
        BEGIN
                u0:     n2 PORT MAP (s, c, im1);
                u1:     n2 PORT MAP (c, r, im2);
                u2:     n2 PORT MAP (im4, im1, im3);
                u3:     n2 PORT MAP (im3, im2, im4);
                s0:     q <= im3;
                s1:     qbar <= im4;
        END default;
```

--Using generic parameters

```
ARCHITECTURE fixed OF sr_latch IS
        COMPONENT n2
                GENERIC (tplh, tphl: TIME);
                PORT (a, b: IN BIT; o: OUT BIT);
        END COMPONENT;
        FOR ALL: n2 USE ENTITY work.nand2 (ave_delay);
        SIGNAL im1, im2, im3, im4: BIT;
BEGIN
        u0:     n2      GENERIC MAP (2 ns, 4 ns)
                        PORT MAP (s, c, im1);
        u1:     n2      GENERIC MAP (2 ns, 4 ns)
                        PORT MAP (c, r, im2);
        u2:     n2      GENERIC MAP (3 ns, 5 ns)
                        PORT MAP (im4, im1, im3);
        u3:     n2      GENERIC MAP (3 ns, 5 ns)
                        PORT MAP (im3, im2, im4);
        s0:     q <= im3;
        s1:     qbar <= im4;
END fixed;
```

17. Generate statement

→ Purpose:

1. Reduce the number of lines of code (by removing repetition)
2. Make the code flexible

→ It is a concurrent statement

→ Several generate statement can be nested

→ One form using generate statement

Use in a FOR loop

➔ H/W just replicated

➔ Application: 32-bit RCA

→ Another form of using: use IF followed by a condition for the generation

18. How to write the testbenches

→ Test vector selection

1. Exhaustive ➔ combination explosion ➔ impractical
2. Arbitrary ➔ can’t rely on the outcome
3. Make intelligent choice ➔ Great if you can do it
4. Test at different levels
5. Random ➔ Different from arbitrary, give equal chances

Basically, do (4), try to do (3), combined with (5).

Machines can do (4) and (5)

ATPG: Automatic Test Pattern Generators

19. Output Stage

1. Totempole (or Push-pull output)

   Like TTL, NAND Gate
2. High impedance (Tri-state)
3. Open collector outputs