بسم الله الرحمن الرحيم
الحمد لله الذي له ما في السماوات و ما في الأرض ، و له الحمد في
الآخرة وهو الحكيم الخبير

This is a compiled set of my lecture notes that I used in teaching COE 405 "*Design and Modeling of Digital Systems*". These notes have evolved since I developed and offered the course for the first time in the **981** semester. The notes have been also enhanced in the semesters; **011, 041, 042** and **052**.

These lectures were also largely used by my colleague **Dr. Aiman El-Maleh** when he taught the same course (COE 405) in **021** and **031**.

With the exception of the "**Introduction**" and the "**VHDL Synthesis**" Lectures, all these lectures were developed by me using the following references:

1. Zainalabedin Navabi, "VHDL: Analysis and Modeling of Digital Systems", McGraw-Hill, Inc., 2nd edition, 1997. (*Current textbook of the course*)
2. Douglas Perry, "VHDL" McGraw-HILL 1998.
3. K. Skahill, "VHDL for Programmable Logic," Addison Wesley, 1996.
4. R. Lipsett, C. Schaefer, and C. Ussery, "VHDL: Hardware Description and Design," Kluwer Academic Publishers, 1990.
5. James R. Armstrong, "Chip-Level Modeling with VHDL," Prentice Hall 1989.

The cover pages of these two lectures ("**Introduction**" and the "**VHDL Synthesis**") acknowledge their original authors.

Bookmarks of various topics and subtopics have been provided for Easy navigation.

To encourage thorough reading, *I promise to pay* **5 SR** *for the first student who identifies any non-trivial coding error* in these notes.

Wishing the best to all,

Alaa Amin

Dr. Alaaeldin Amin

COE 405

*Design and Modeling of Digital Systems **

Dr. Alaaeldin A. Amin

Computer Engineering Department

E-mail: amin@ccse.kfupm.edu.sa

Home Page : http://www.ccse.kfupm.edu.sa/~amin

*** This Lecture is Mostly taken from from : Dr. A. Selçuk Öğrenci**

ogrenci@boun.edu.tr

Course Objective

- **Learning VHDL**
  - Write Functionally Correct and well-documented VHDL Code of Combinational or Sequential Digital Systems Intended for Modeling or Synthesis Purposes
  - Define and Use the 3 Major Modeling Styles (Structural, DataFlow, Behavioral)
  - Define a Suitable Test Bench for Model Verification

Course Objective

- More on Digital System Design (Data-Path & Control)
- More (Applications) on Computer Arithmetic

TOPICS

Structured Design Methodologies	Digital System Design, Abstraction hierarchy, Types of Behavioral Descriptions The Digital Design Space & Design Decomposition.	Handout, Ch 1

TOPICS

VHDL Quick Overview	Design Partitioning & Top-Down Design. Design Entities, Signals vs. Variables, Architectural Bodies, Different design views, behavioral model, dataflow model, structural models.	Handout , Ch 3
Digital System Design	Data Path and Control Path. Sample designs. VHDL Models	Handout
Design & Modeling Tools	Tutorials on available Simulators	

TOPICS

VHDL Language Basics	Lexical Elements, Data Types (Scalars & Composites), Type Conversion, Attributes, Classes of objects. Operators & Precedence, Overloading.	Handout, Ch. 7
Signals, Delays & Concurrency.	Variables vs. Signals, sequential vs concurrent constructs, Signal Propagation Delay & Delay types, Transactions, Events and Transaction Scheduling, Signal Attributes.	Ch 4, handout
Structural Models	Structural Models, Configuration Statement, Modeling Iterative/Regular Structures, and Test Benches.	Ch 5

TOPICS

Design Organization & Parameterization	Packages & Libraries. Design Parameterization, Design Configuration & General purpose test bench.	Ch 6
Dataflow Models	Concurrent Signal Assignment, Block statements, Guards, Resolution functions, Resolved Signals and Signal Kinds, Data Flow Moore & Mealy Models, Data & Control Path Data Flow Models.	Ch 8
Behavioral Models	Process & Wait Statements, Assert Statement, General Algorithmic Model, Moore and Mealy Machine Algorithmic Models, Data & Control Path Design.	Ch. 9

TOPICS

Introduction to VHDL Synthesis	Combinational, sequential logic synthesis, state machine synthesis.	Handout Notes
CPU Design Example		Ch10, 11.

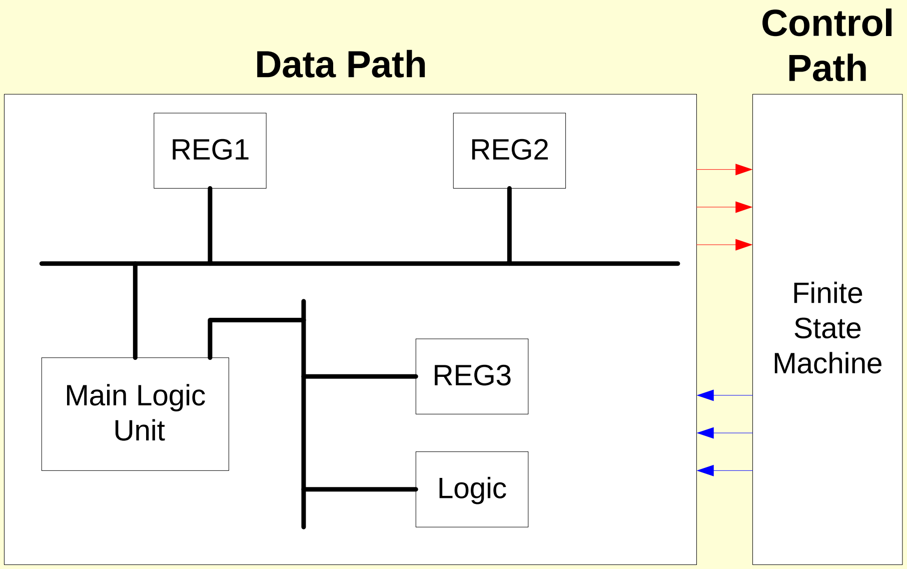
Grading Policy

Option 1		Option 2	
•Homework Assignments	15%	•Homework Assignments	20%
Midterm Exam	20%	Midterm Exam	20%
Project	25%	Project	25%
Quizzes	15%	Project Presentation	10%
Final	25%	Quizzes	25%

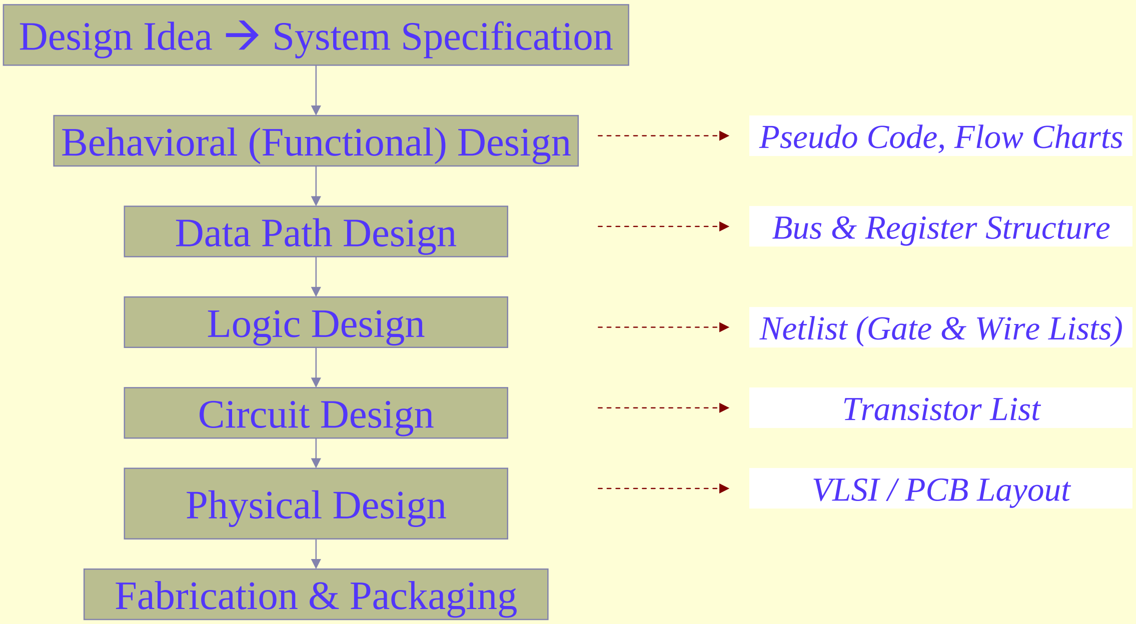
Digital System Design

- **Realization of a specification Subject to the Optimization of**
  - Area (Chip, PCB)
  - Speed
  - Power dissipation
  - Design time
  - Testability

Digital System Design



Digital System Design Cycle


Design Idea → System Specification

Behavioral (Functional) Design → *Pseudo Code, Flow Charts*

Data Path Design → *Bus & Register Structure*

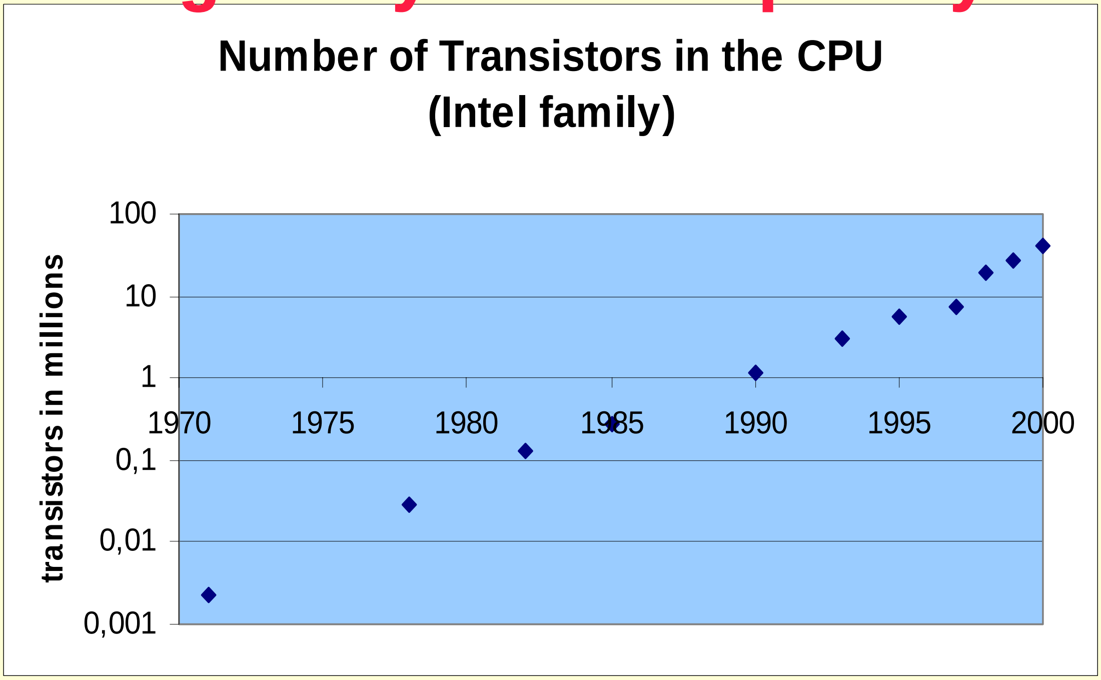
Logic Design → *Netlist (Gate & Wire Lists)*

Circuit Design → *Transistor List*

Physical Design → *VLSI / PCB Layout*

Fabrication & Packaging

Digital System complexity

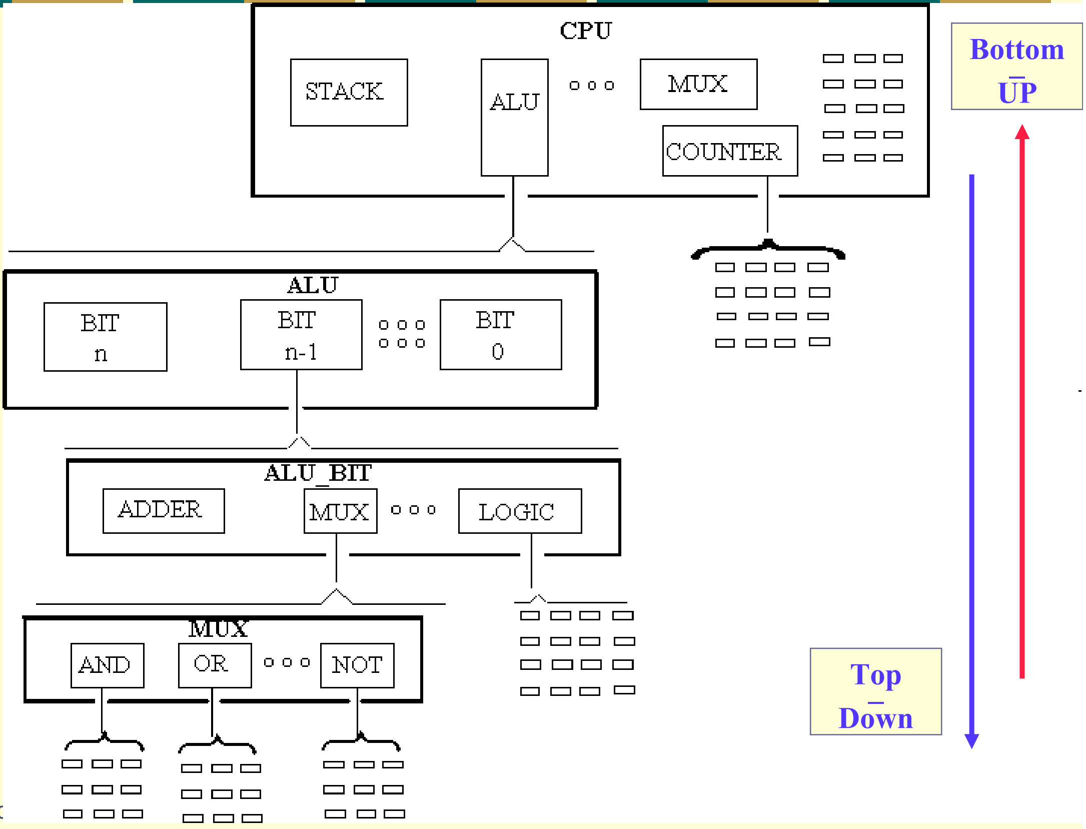


How to deal with the complexity?

Moore's Law: Number of transistors that can be packed on a chip doubles every 18 months while the price stays the same.

- **Hierarchy: structure of a design at different levels of description**
- **Abstraction: hiding the lower level details.**

Hierarchy



CPU
STACK
ALU
MUX
COUNTER
ALU
BIT n
BIT n-1
BIT 0
ALU_BIT
ADDER
MUX
LOGIC
MUX
AND
OR
NOT

Bottom UP

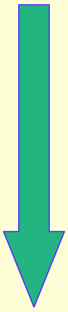
Top Down

Abstractions

- An ***Abstraction*** is a Simplified Model of Some Entity Which *Hides Certain Amount of the Internal Details of this Entity*
- Examples are: NAND gate, Transistor, Abstract Data Type, etc.
- Lower Level Abstractions Give More Details of the Modeled Entity.

Hardware Levels of Abstraction

- **Several Levels of Abstractions (*Details*) are Commonly Used:**
  - System Level
  - Chip Level
  - Register Level
  - Gate Level
  - Circuit (Transistor) Level
  - Layout (Geometric) Level



More Details

(Less Abstract)

Design Domains & Levels of Abstraction

- **Designs Can Be Expressed / Viewed in one of 3 Possible Domains**
  - Behavioral Domain (*Behavioral View*)
  - Structural/Component Domain (*Structural View*)
  - Physical Domain (*Physical View*)
- **A Design Modeled in a Given Domain Can be Represented at Several Levels of Abstraction (*Details*)**

Design Domains & Levels of Abstraction

Design Domain

Abstraction Level	***Behavioral***	***Structural***	***Physical***
System	English Specs	Computer, Disk Units, Radar, etc.	Boards, MCMs, Cabinets
Chip	Algorithms, Flow Charts	Processors, RAMs, ROMs	Chips, Floor Plans, PCBs
Register	Data Flow, Reg. Transfer	Registers, ALUs, Counters, MUX, etc.	Std. Cells, Floor Plans
Gate	Boolean Equations	AND, OR, XOR, FFs, etc	Cells, Gates, FFs, PCBs
Circuit (Tr)	Diff, and element Equations	Transistors, R, L, C, etc …	Mask Geometry (Layout)

Black Box View

Grey Box View

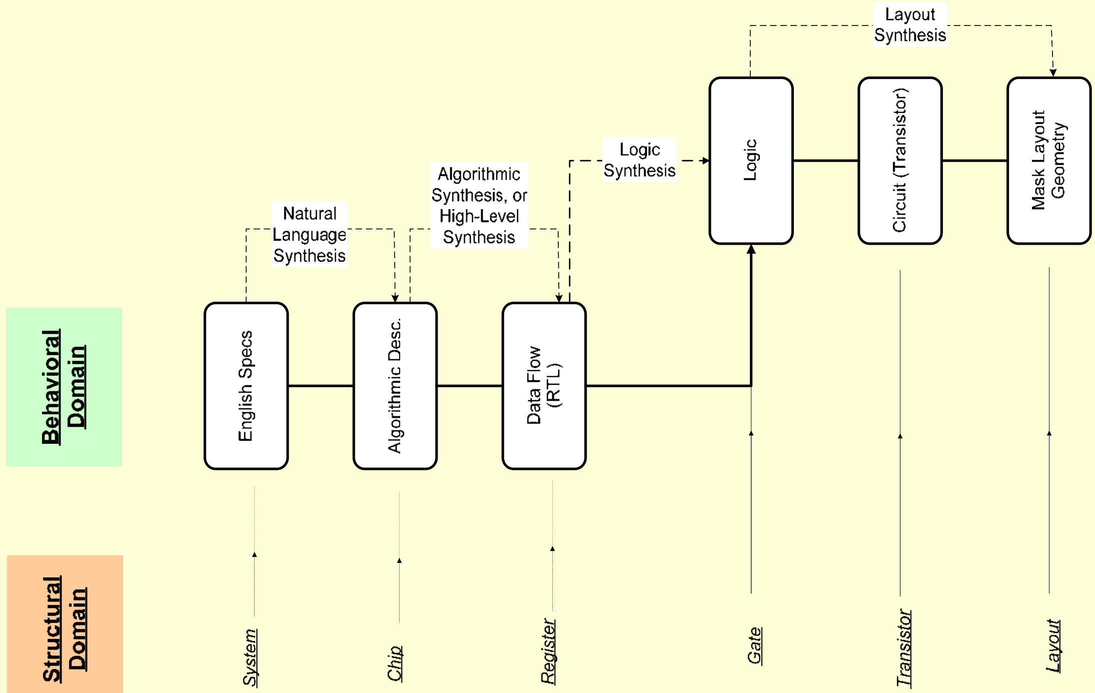
White Box View

Design methods

- **Full custom**
  - Maximal freedom
  - High performance blocks
  - Slow
- **Semi-custom**
  - Gate Arrays
    - Mask Programmable (MPGAs)
    - Field Programmable (FPGAs))
  - Standard Cells
  - Silicon Compilers & Parametrizable Modules (adder, multiplier, memories)

Design vs. Synthesis

- **Synthesis:**
  - The Process of Transforming H/W from One Level of Abstraction to a ***Lower*** One
- **Design:**
  - A Sequence of Synthesis Steps Down to a Level of Abstraction Which is Manufacturable



Layout Synthesis

Logic Synthesis

Algorithmic Synthesis, or High-Level Synthesis

Natural Language Synthesis

Logic

Circuit (Transistor)

Mask Layout Geometry

Behavioral Domain

English Specs

Algorithmic Desc.

Data Flow (RTL)

Structural Domain

System

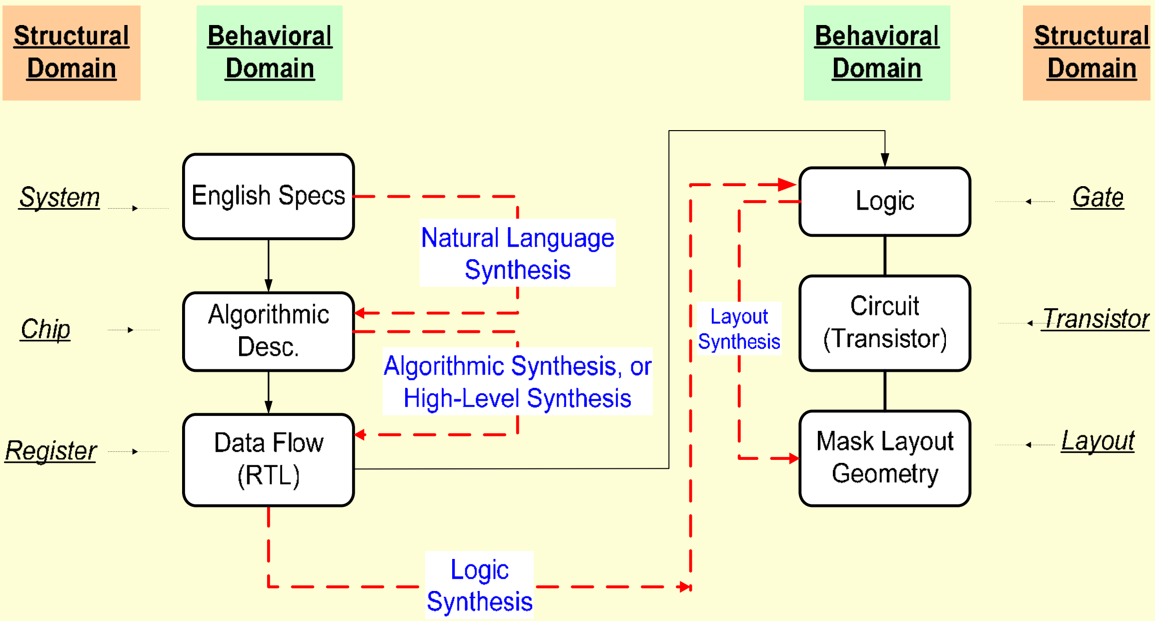
Chip

Register

Gate

Transistor

Layout



Structural Domain

Behavioral Domain

Behavioral Domain

Structural Domain

System

Chip

Register

English Specs

Algorithmic Desc.

Data Flow (RTL)

Natural Language Synthesis

Algorithmic Synthesis, or High-Level Synthesis

Logic Synthesis

Layout Synthesis

Logic

Circuit (Transistor)

Mask Layout Geometry

Gate

Transistor

Layout

Design Automation & CAD Tools

- **Design Entry (Description) Tools**
  - Schematic Capture
  - Hardware Description Language (HDL)
- **Simulation (Design Verification) Tools**
  - Simulators (Logic level, Transistor Level, High Language Level “HLL”)
- **Synthesis Tools**
- **Test Vector Generation Tools**

HARDWARE DESCRIPTION LANGUAGES

- **HDL are used to describe the hardware for the purpose of modeling, simulation, testing, design, and documentation.**
  - *Modeling*: behavior, flow of data, structure
  - *Simulation:* verification and test
  - *Design*: synthesis

Purpose of VHDL

- **Problem**
  - Need a method to quickly design, implement, test, and document increasingly complex digital systems
  - Schematics and Boolean equations inadequate for million-gate IC
- **Solution**
  - A hardware description language (HDL) to express the design
  - Associated computer-aided design (CAD) or electronic design automation (EDA) tools for synthesis and simulation
  - Programmable logic devices for rapid implementation of hardware
  - Custom VLSI application specific integrated circuit (ASIC) devices for low-cost mass production

History of VHDL

- **Two widely-used HDLs today**
  - VHDL
  - Verilog HDL (from Cadence, now IEEE standard)

- **VHDL - VHSIC Hardware Description Language**

Very High Speed Integrated Circuit

- **VHDL history**
  - Created by DOD to document military designs for portability
  - IEEE standard 1076 (VHDL) in 1987
  - Revised IEEE standard 1076 (VHDL) in 1993
  - IEEE standard 1164 (object types standard) in 1993
  - IEEE standard 1076.3 (synthesis standard) in 1996

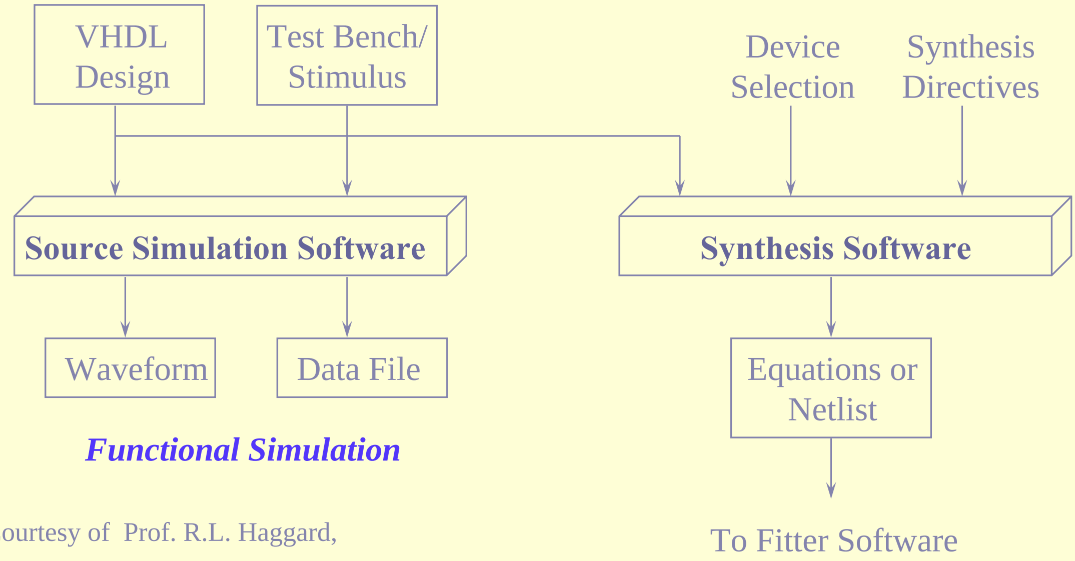
VHDL: Why to use?

- **Reasons to use VHDL**
  - Power and flexibility
  - Device-independent design
  - Portability among tools and devices
  - Device and tool benchmarking capability
  - VLSI ASIC migration
  - Quick time-to-market and low cost (with programmable logic)
- **Problems with VHDL**
  - Loss of control with gate-level implementation (so what?)
  - Inefficient logic implementations via synthesis (engineer-dependent)
  - Variations in synthesis quality among tools (always improving)

Design Flow in VHDL

- **Define the design requirements**
- **Describe the design in VHDL**
  - Top-down, hierarchical design approach
  - Code optimized for synthesis or simulation
- **Simulate the VHDL source code**
  - Early problem detection before synthesis
- **Synthesize, optimize, and fit (place and route) the design for a device**
  - Synthesize to equations and/or netlist
  - Optimize equations and logic blocks subject to constraints
  - Fit into the components blocks of a given device
- **Simulate the post-layout design model**
  - Check final functionality and worst-case timing
- **Program the device (if PLD) or send data to ASIC vendor**

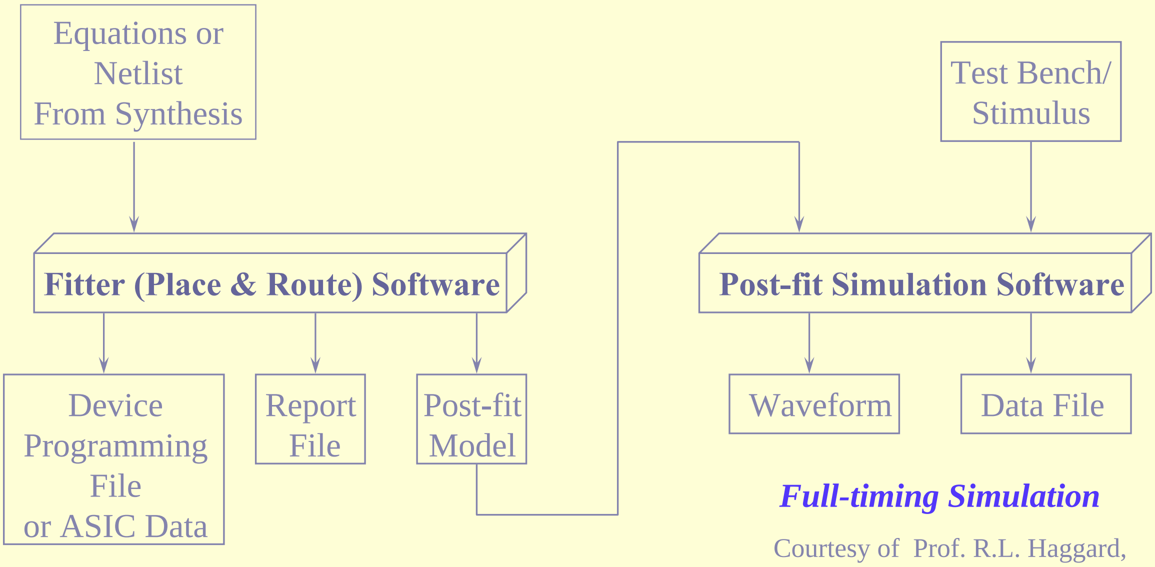
Design Tool Flow (1)



Functional Simulation

Courtesy of Prof. R.L. Haggard,
Tennessee Technological University

Design Tool Flow (2)


Equations or Netlist From Synthesis
Test Bench/ Stimulus
Fitter (Place & Route) Software
Post-fit Simulation Software
Device Programming File or ASIC Data
Report File
Post-fit Model
Waveform
Data File

Full-timing Simulation

Courtesy of Prof. R.L. Haggard,
Tennessee Technological University

STYLES in VHDL

- **Levels of Abstraction (Architectural Styles):**
- **Behavioral**
  - High level, algorithmic, sequential execution
  - Hard to synthesize well
  - Easy to write and understand (like high-level language code)
- **Dataflow**
  - Medium level, register-to-register transfers, concurrent execution
  - Easy to synthesize well
  - Harder to write and understand (like assembly code)
- **Structural**
  - Low level, netlist, component instantiations and wiring
  - Trivial to synthesize
  - Hardest to write and understand (very detailed and low level)

SUMMARY

- **The VLSI digital design problem is described.**
- **VLSI design automation and CAD tools are mentioned.**
- **Purpose and background of VHDL have been pointed out.**
- **VHDL and programmable logic are the best current solution for rapid design, implementation, testing, and documenting of complex digital systems.**
- **A standard 6-step design synthesis process is used with VHDL.**
- **The general flow of information through standard VHDL synthesis CAD tools was described.**

VHDL Lexical Elements

OUTLINE

- **Design File**
  - **Lexical Elements &&**
  - **Separators**
- **Reserved Words**
- **User-Defined Identifiers**
- **Literals**
  - **1. Character Literals**
  - **2. String Literals**
  - **3. Bit-String Literals**
  - **4. Abstract (Numeric) Literals**
    - **Based Literals**

Dr. Alaaeldin Amin

VHDL Lexical Elements

Design File = Sequence of
•Lexical Elements &&
•Separators

(a) Separators: *Any # of Separators Allowed Between Lexical Elements*

1. Space character
2. Tab
3. Line Feed / Carriage Return (EOL)

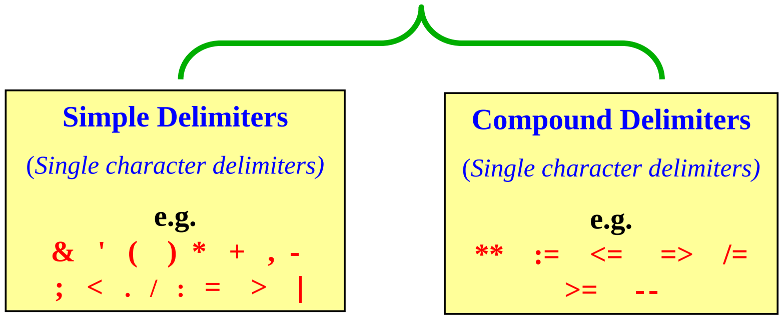
(b) Lexical Elements: **Three types:**

1. Delimiters " *Meaningful Separator Characters*"
2. Identifiers
3. Literals → Types of literals:

(i) Character Literal
(ii) String Literal
(iii) Bit String Literal
(iv) Abstract (Numeric) Literal

VHDL Lexical Elements

(i) Delimiters are *Separators Which Have Meaning*



Simple Delimiters

(Single character delimiters)

e.g.

& ' ( ) * + , -
; < . / : = > |

Compound Delimiters

(Single character delimiters)

e.g.

** := <= => /=
>= --

Note: The operator <= has two meanings:

1. Less Than or Equal, and
2. Signal Assignment Operator

(ii) Identifiers: Two Types:

1. Key/Reserved Words *(No Declaration Required)*
2. User-Defined

VHDL Lexical Elements

Reserved Words

abs
access
after
alias
all
and
architecture
array
assert
attribute
begin
block
body
buffer
bus
case
component
configuration
constant

disconnect
downto
else
elsif
end
entity
exit
file
for
function
generate
generic
guarded
if
in
inout
is

label
library
linkage
loop
map
mod
nand
new
next
nor
not
null
of
on
open
or
others
out

package
Poll
procedure
process
range
record
register
rem
report
return
select
severity
signal
subtype
then
to
transport
type

units
until
use
variable
wait
when
while
with
xor

VHDL Lexical Elements

User-Defined Identifiers

Definition:

Identifier ::= Letter **{** **[***underscore***]** *Letter_or_Digit***}**

Identifiers in VHDL Must Satisfy the Following:

- ***Start*** with a Letter
- Followed by any # of Alpha-Numeric Characters
- ***No*** 2-Consecutive Underscores are Allowed
- Underscore ***Cannot*** be the Last Character in an Identifier
- Case insensitive
- No VHDL reserved/key word.

Examples:

- mySignal_23 -- Valid identifier
- rdy, RDY, Rdy -- Valid identical identifiers
- vector_&_vector -- **Invalid →** special character
- last of Zout -- **Invalid →** white spaces
- idle__state --**Invalid→**consecutive underscores
- 24th_signal -- **Invalid →** Doesn't ***Start***
  -- with a Letter
- open, register -- **Invalid →** VHDL keywords

VHDL Lexical Elements

Extended Identifiers (VHDL-93 Only)

Definition:

extended_identifier ::= \ graphic_character **{***graphic_character***}** **

Extended Identifiers in VHDL are characterized by the following:

- Defined in VHDL-93 only
- Enclosed in back slashes
- Case sensitive
- Graphical characters allowed
- May contain spaces and consecutive underscores
- VHDL keywords allowed

Examples:

- \mySignal_23\ -- extended identifier
- \rdy\, \RDY\, \Rdy\ -- 3 different identifiers
- \vector_&_vector\ -- legal
- \last of Zout\ -- legal
- \idle__state\ -- legal
- \24th_signal\ -- legal
- \open\, \register\ -- legal

Comments

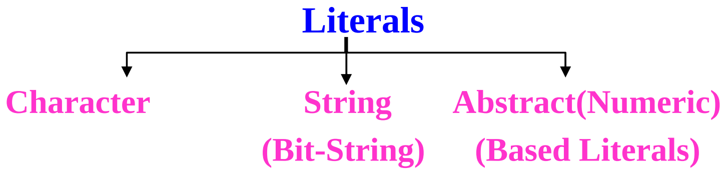
- Start with **--** *"2 Consecutive Dashes"*
- Comment Must be the LAST Lexical Element on the Line
- IF Line starts with **--**, It is a *Full-Line Comment*.

Examples:

- - *This is a Full-Line Comment*

C := A*B; **- -** *This is an In-Line Comment*

VHDL Lexical Elements



Literals

Character

String
(Bit-String)

Abstract(Numeric)
(Based Literals)

(i) Character Literal

- ***Single*** Character Enclosed in ***Single Quotes***
- Used to Define *Constant* Values of *Objects of Type* *Character*
- Literal values are *Case Sensitive*; ‘**z**’ NOT SAME as ‘**Z**’

Examples of Character Literals:

‘A’ ‘B’ ‘e’ ‘ ’ ‘1’ ‘9’ ‘*’ ……….etc.

(ii) String Literal

- ***Sequence*** of Characters Enclosed in ***Double Quotes***
- IF a Quotation Char is part of the character sequence, 2 Consecutive Quotation Marks Are Used

  – *No **2** Strings are Allowed on the same Line*

Examples:

```
"A String"          -- 8-Char  String
""                  -- Empty  String
" " " "         -- 4-Double Quotes → String of Length 1
"A+B=C;#3=$"        -- String  with Special Chars
```

VHDL Lexical Elements (Literals)

(ii) String Literal (Contd)

- ***Strings Must Be Typed on One Line***
- Longer Strings Are *Concatenated* from Shorter Ones Using the **&** operator.

Examples:

```
"This is a Very Long String Literal"     &
"Formed By Concatenation"
```

(iii) Bit String Literals

- *Is a String Literal representing a “Bit Pattern”*
- *Bit String Literal are characterized by :*
  - Preceded By A *Base Identifier* ∈ **{B, O, X}** {**B** for Binary, **O** for Octal and **X** for Hex}
  - Allowed Chars are ***Digits*** of the Base Number System or *Underscores*.
  - The Length of the String Does Not Include the Number of Underscores

Examples:

- **B**"11011001" -- Length 8 (Binary)
- **B**"1101_1001" -- Length 8 (Binary)
- **X**"D9" -- Length 8 (HEX → Equiv to Above String)
- **O**"331" -- Length 9 (Octal)

VHDL Lexical Elements

(iii) Bit String Literals (Contd)

- *Used to Specify Initial Contents of Registers*
- *Value of Bit-String is Equivalent to a String of Bits, However, Interpreting This Value is a User Choice*

Examples:

- X"A" -- Represents the String 1010
  -- Interpreted as a Decimal Value of ten
  -- if it represents an Unsigned Number.
  -- Interpreted as -6 if it represents a
  -- signed 2`s Complement Number

(iv) Abstract (Numeric) Literals

1. *Default is Decimal*
2. *Other Bases Are Possible (Bases Between 2 and 16)*
3. *Underscore Char May Be Used to Enhance Readability*
4. *Scientific Notations Must Have Integer Exponent*
5. *Integer Literals **Should Not Have** Base Point*
6. *Integer Literals **Should Not Have** -ive Exponents*
7. *Real Literals, **Should Have** a Base Point which **Must Be Followed** By AT LEAST ONE DIGIT*
8. *No Spaces Are Allowed*

VHDL Lexical Elements

(iv) Abstract (Numeric) Literals (Contd)

Examples:

0 1 123_987_456 73E13 -- Integers

0.0 2.5 2.7_456 73.0E-2 12.5E3 -- Reals

Special Case (BASED LITERALS)

- ***General Base** Abstract Literals (Including Decimal)*

Based_Literal::=Base#Based_Integer[.Based_Integer]#[Exponent]

Based_Integer::=Extd_Digit { [Underscode] Extd_Digit }

Extd_Digit::=digit | Letters_A-F

- *Both **Base** and **Exponent** are Expressed in **Decimal***
- ***Base** Must be Between 2 & 16*
- *Digits Are Extended to Use the HEX Characters A-F*

Examples:

The Following Represent Integer Value of 196

2#1100_0100**#** , **16#**C4**#**

4#301**#E1** , **10#**196**#**

The Following Represent Real Value of 4095.0

2#1.1111_1111_111**#E11** , **16#**F.FF**#E2**

10#4095.0**#**

VHDL Lexical Elements

- Formal grammar of the IEEE Standard 1076-1993 VHDL language in BNF format
  - Appendix E
  - http://www.iis.ee.ethz.ch/~zimmi/download/vhdl93_syntax.html

VHDL: A Quick Overview

OUTLINE

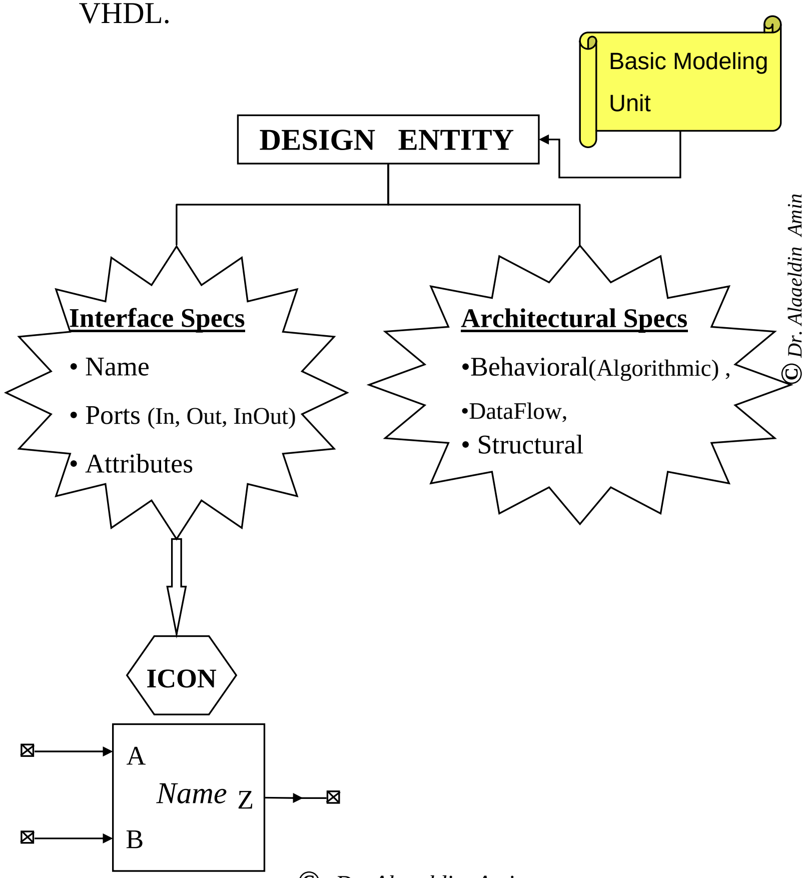
VHDL Basic Modeling Unit (Design Entity)

- **Example (Ones Count Circuit)**
  - **Interface Specs**
  - **1. Behavioral View**
  - **2. Data Flow View (2-Level Implementation)**
  - **3. Data Flow View (using functions)**
  - **4. Behavioral View (using Truth Table)**
  - **5. Structural View**

Dr. Alaaeldin Amin

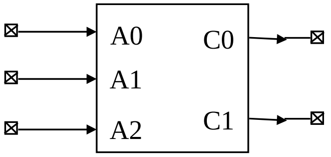
Hardware MOdeling using vhdl

- **VHDL** is **<u>NOT</u>** ***<u>CaSe-SeNsItIvE</u>*** **, Thus:**
  Begin = begin = beGiN
- Semicolon " **;** " Terminates Declarations or Statements.
- Line Feeds and Carriage Returns are not Significant in VHDL.

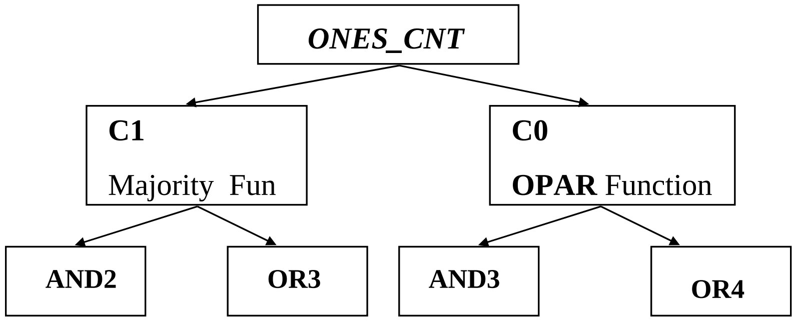


Example

“Ones Count CIRCUIT”



- Value of **C1 C0** = No. of Ones in the Inputs **A2, A1**, and **A0**
- **C1** is the Majority Function (=1 *IFF* Two or More Inputs =1)
- **C0** is a 3-Bit Odd-Parity Function (OPAR3))
- C1 = A1 A0 + A2 A0 + A2 A1
- C0 = A2 A1’ A0’ + A2’ A1’ A0 + A2 A1 A0 + A2’ A1 A0’



ONES_CNT

C1
Majority Fun

C0
OPAR Function

AND2

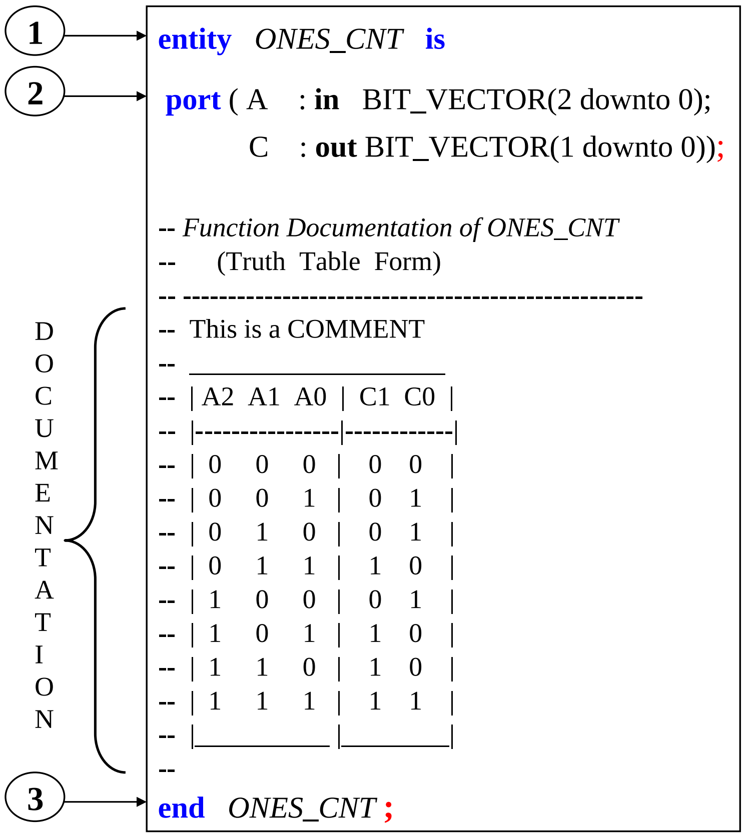
OR3

AND3

OR4

Example “Ones Count CIRCUIT

INTERFACE SPECs



```
entity   ONES_CNT   is

 port ( A    : in   BIT_VECTOR(2 downto 0);
        C    : out BIT_VECTOR(1 downto 0));


-- Function Documentation of ONES_CNT
--      (Truth  Table  Form)
-- ---------------------------------------------------
--  This is a COMMENT
--  ____________________
-- | A2  A1  A0 |  C1  C0 |
-- |-----------------|-------------|
-- |  0    0    0  |   0    0   |
-- |  0    0    1  |   0    1   |
-- |  0    1    0  |   0    1   |
-- |  0    1    1  |   1    0   |
-- |  1    0    0  |   0    1   |
-- |  1    0    1  |   1    0   |
-- |  1    1    0  |   1    0   |
-- |  1    1    1  |   1    1   |
-- |___________|__________|
--
end   ONES_CNT ;
```

example “Ones Count CIRCUIT

Architectural Body

(((((Behavioral view-1)))))

```
architecture  Algorithmic  of  ONES_CNT  is
begin

Process(A) -- Sensitivity List Contains only Vector A
    Variable  num: INTEGER range 0 to 3;

    begin
          num :=0;
          For i   in   0 to 2
          Loop
                 IF A(i) = '1' then
                              num := num+1;
                 end if;
          end Loop;
--
--        Transfer "num" Variable Value to a SIGNAL
--
          CASE num is
                    WHEN 0 => C <= "00";
                    WHEN 1 => C <= "01";
                    WHEN 2 => C <= "10";
                    WHEN 3 => C <= "11";
          end CASE;
--
    end process;

end Algorithmic;
```

example “Ones Count CIRCUIT

Architectural Body

2- Data Flow view (2-Level Implementation)

- C1 = A1 A0 + A2 A0 + A2 A1
- C0 = A2 A1’ A0’ + A2’ A1’ A0 + A2 A1 A0 + A2’ A1 A0’

```
architecture  Two_Level of  ONES_CNT  is

begin

     C(1) <=(A(1) and A(0)) or (A(2) and A(0))
              or (A(2) and A(1));
--
     C(0) <= (A(2) and not A(1) and not A(0))
              or (not A(2) and not A(1) and A(0))
              or (A(2) and A(1) and A(0))
              or (not A(2) and A(1) and not A(0));

end Two_Level;
```

Example “Ones Count CIRCUIT

Architectural Body
3- Data Flow view
(Using Functions)

```
architecture  Macro  of  ONES_CNT  is

begin

     C(1) <= MAJ3(A);
--
     C(0) <= OPAR3(A);

end Macro ;
```

- ***Functions OPAR3 and MAJ3 Must Have Been Declared and Defined Previously***

example “Ones Count CIRCUIT

Architectural Body
(((((Behavioral view -4)))))

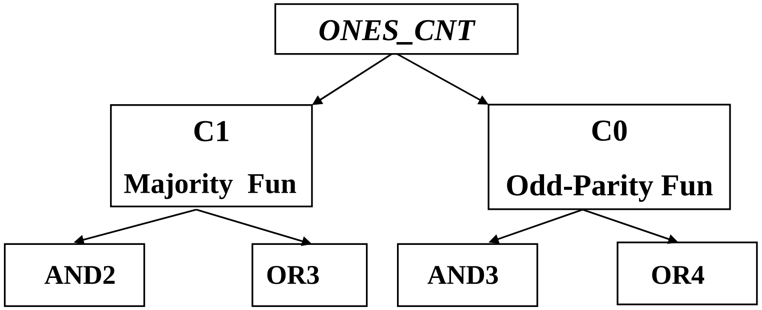
```
architecture  Truth_Table  of  ONES_CNT  is
begin
--
Process(A) -- Sensitivity List Contains only Vector A
    Variable num: BIT_VECTOR(2 downto 0);
    begin
          num :=A;
          CASE num  is
               WHEN "000" => C <= "00";
               WHEN "001" => C <= "01";
               WHEN "010" => C <= "01";
               WHEN "011" => C <= "10";
               WHEN "100" => C <= "01";
               WHEN "101" => C <= "10";
               WHEN "110" => C <= "10";
               WHEN "111" => C <= "11";
          end CASE;
--
    end process;

end  Truth_Table;
```

VHDL STRUCTURAL DESCRIPTION

“Ones Count CIRCUIT example ”

- C1 = A1 A0 + A2 A0 + A2 A1 =**MAJ3**(A)
- C0 = A2 A1’ A0’ + A2’ A1’ A0 + A2 A1 A0 + A2’ A1 A0’ = **OPAR3**(A)

Structural Design Hierarchy


ONES_CNT

C1
Majority Fun

C0
Odd-Parity Fun

AND2

OR3

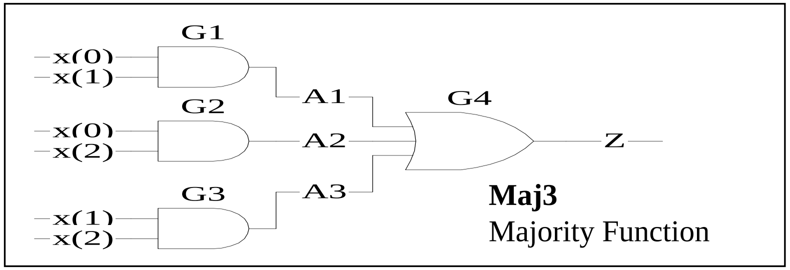
AND3

OR4

```
entity   MAJ3   is
    PORT( X: in BIT_Vector(2 downto 0);
             Z: out BIT);
end  MAJ3 ;
```

```
entity  OPAR3  is
    PORT( X: in BIT_Vector(2 downto 0);
             Z: out BIT) ;
end OPAR3 ;
```

VHDL STRUCTURAL DESCRIPTION



Maj3
Majority Function

```
architecture   Structural   of   MAJ3   is
COMPONENT AND2
     PORT( I1, I2: in BIT;
                  O: out BIT);
END COMPONENT;
COMPONENT OR3
     PORT( I1, I2, I3: in BIT;
                  O: out BIT);
END COMPONENT;
```

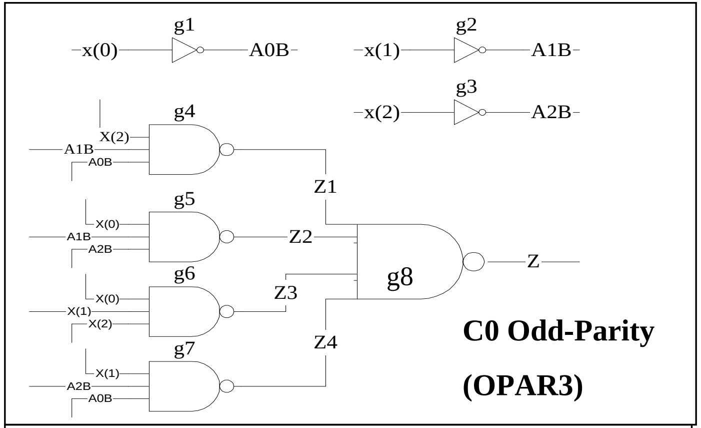
Declare Components To Be Instantiated

```
--
SIGNAL  A1, A2, A3:   BIT;
```

Declare Maj3 Local Signals

```
begin
-- Instantiate Gates
     g1: AND2 PORT MAP (X(0), X(1), A1);
     g2: AND2 PORT MAP (X(0), X(2), A2);
     g3: AND2 PORT MAP (X(1), X(2), A3);
     g4: OR3  PORT MAP (A1, A2, A3, Z);
end Structural;
```

Wiring of Maj3 Compts.



C0 Odd-Parity

(OPAR3)

```
architecture  Structural  of  OPAR3  is
Component INV
    PORT( Ipt: in BIT; Opt: out BIT);
end Component ;
Component NAND3
    PORT( I1, I2, I3: in BIT;  O: out BIT);
end Component ;
Component NAND4
    PORT( I1, I2, I3, I4: in BIT; O: out BIT);
end Component ;
--
SIGNAL A1B, A2B, A0B, Z1, Z2, Z3, Z4: BIT;
begin
            g1: INV PORT MAP (X(0), A0B);
            g2: INV PORT MAP (X(1), A1B);
            g3: INV PORT MAP (X(2), A2B);
            g4: NAND3 PORT MAP (X(2), A1B, A0B, Z1);
            g5: NAND3 PORT MAP (X(0), A1B, A2B, Z2);
            g6: NAND3 PORT MAP (X(0), X(1), X(2), Z3);
            g7: NAND3 PORT MAP (X(1), A2B, A0B, Z4);
            g8: NAND4 PORT MAP (Z1, Z2, Z3, Z4, Z);
end Structural;
```

Top Structural level of ones_cnt

```
architecture  Structural  of  ONES_CNT  is
COMPONENT MAJ3
    PORT( X: in BIT_Vector(0 to 2);
              Z: out BIT);
END COMPONENT;
COMPONENT OPAR3
    PORT( X: in BIT_Vector(0 to 2);
              Z: out BIT);
END COMPONENT;
--
begin
-- Instantiate Components
--
    c1: MAJ3 PORT MAP (A, C(1));
    c2: OPAR3  PORT MAP (A, C(0));
end Structural;
```

Behavioral definition of lower level components

```
entity   INV   is
     PORT( Ipt: in BIT;  Opt: out BIT);
end INV;
--
architecture   behavior  of  INV   is
begin
     Opt <= not Ipt;
end behavior;
```

```
entity  NAND2 is
     PORT( I1, I2: in BIT; O: out BIT);
end NAND2;
--
architecture   behavior   of   NAND2   is
begin
     O <= not (I1 and I2);
end behavior;
```

Similarly Other Lower Level Gates Are Defined

VHDL Lexical Elements

OUTLINE

- **Design File**
  - **Lexical Elements &&**
  - **Separators**
- **Reserved Words**
- **User-Defined Identifiers**
- **Literals**
  - **1. Character Literals**
  - **2. String Literals**
  - **3. Bit-String Literals**
  - **4. Abstract (Numeric) Literals**
    - **Based Literals**

VHDL Lexical Elements

Design File = Sequence of
•Lexical Elements &&
•Separators

(a) Separators: *Any # of Separators Allowed Between Lexical Elements*

1. Space character
2. Tab
3. Line Feed / Carriage Return (EOL)

(b) Lexical Elements: **Three types:**

1. Delimiters " *Meaningful Separator Characters*"
2. Identifiers
3. Literals → Types of literals:

(i) Character Literal
(ii) String Literal
(iii) Bit String Literal
(iv) Abstract (Numeric) Literal

VHDL Lexical Elements

(i) Delimiters are *Separators Which Have Meaning*

Simple Delimiters

(Single character delimiters)

e.g.

& ' ( ) * + , -
; < . / : = > |

Compound Delimiters

(Single character delimiters)

e.g.

** := <= => /=
>= --

Note: The operator <= has two meanings:

1. Less Than or Equal, and
2. Signal Assignment Operator

(ii) Identifiers: Two Types:

1. Key/Reserved Words *(No Declaration Required)*
2. User-Defined

VHDL Lexical Elements

Reserved Words

abs
access
after
alias
all
and
architecture
array
assert
attribute
begin
block
body
buffer
bus
case
component
configuration
constant

disconnect
downto
else
elsif
end
entity
exit
file
for
function
generate
generic
guarded
if
in
inout
is

label
library
linkage
loop
map
mod
nand
new
next
nor
not
null
of
on
open
or
others
out

package
Poll
procedure
process
range
record
register
rem
report
return
select
severity
signal
subtype
then
to
transport
type

units
until
use
variable
wait
when
while
with
xor

VHDL Lexical Elements

User-Defined Identifiers

Definition:

Identifier ::= Letter **{** **[***underscore***]** *Letter_or_Digit***}**

Identifiers in VHDL Must Satisfy the Following:

- ***Start*** with a Letter
- Followed by any # of Alpha-Numeric Characters
- ***No*** 2-Consecutive Underscores are Allowed
- Underscore ***Cannot*** be the Last Character in an Identifier
- Case insensitive
- No VHDL reserved/key word.

Examples:

• mySignal_23	-- Valid identifier
• rdy, RDY, Rdy	-- Valid identical identifiers
• vector_&_vector	-- **Invalid →** special character
• last of Zout	-- **Invalid →** white spaces
• idle__state	--**Invalid→**consecutive underscores
• 24th_signal	-- **Invalid →** Doesn't ***Start*** -- with a Letter
• open, register	-- **Invalid →** VHDL keywords

VHDL Lexical Elements

Extended Identifiers (VHDL-93 Only)

Definition:

extended_identifier ::= \ *graphic_character* **{***graphic_character***}** \

Extended Identifiers in VHDL are characterized by the following:

- Defined in VHDL-93 only
- Enclosed in back slashes
- Case sensitive
- Graphical characters allowed
- May contain spaces and consecutive underscores
- VHDL keywords allowed

Examples:

• \mySignal_23\	-- extended identifier
• \rdy\, \RDY\, \Rdy\	-- 3 different identifiers
• \vector_&_vector\	-- legal
• \last of Zout\	-- legal
• \idle__state\	-- legal
• \24th_signal\	-- legal
• \open\, \register\	-- legal

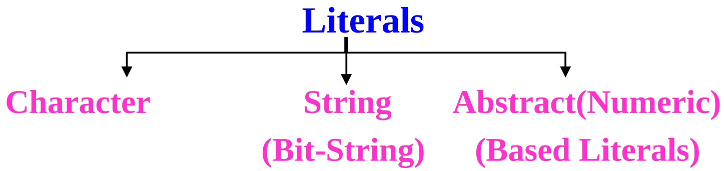
Comments

- Start with **--** *"2 Consecutive Dashes"*
- Comment Must be the LAST Lexical Element on the Line
- IF Line starts with **--**, It is a *Full-Line Comment*.

Examples:

```
- - This is a Full-Line Comment
C := A*B; - - This is an In-Line Comment
```

VHDL Lexical Elements



(i) Character Literal

- ***Single*** Character Enclosed in ***Single Quotes***
- Used to Define *Constant* Values of *Objects of Type* *Character*
- Literal values are *Case Sensitive*; ‘**z**’ NOT SAME as ‘**Z**’

Examples of Character Literals:

‘A’ ‘B’ ‘e’ ‘ ’ ‘1’ ‘9’ ‘*’ ……….etc.

(ii) String Literal

- ***Sequence*** of Characters Enclosed in ***Double Quotes***
- IF a Quotation Char is part of the character sequence, 2 Consecutive Quotation Marks Are Used

– *No **2** Strings are Allowed on the same Line*

Examples:

```
"A String"        -- 8-Char  String
""                -- Empty  String
" " " "      -- 4-Double Quotes → String of Length 1
"A+B=C;#3=$"      -- String  with Special Chars
```

VHDL Lexical Elements (Literals)

(ii) String Literal (Contd)

- ***Strings Must Be Typed on One Line***
- Longer Strings Are *Concatenated* from Shorter Ones Using the **&** operator.

Examples:

```
"This is a Very Long String Literal"     &
"Formed By Concatenation"
```

(iii) Bit String Literals

- *Is a String Literal representing a “Bit Pattern”*
- *Bit String Literal are characterized by :*
  - Preceded By A *Base Identifier* ∈ **{B, O, X}** {**B** for Binary, **O** for Octal and **X** for Hex}
  - Allowed Chars are ***Digits*** of the Base Number System or *Underscores*.
  - The Length of the String Does Not Include the Number of Underscores

Examples:

- **B**"11011001" -- Length 8 (Binary)
- **B**"1101_1001" -- Length 8 (Binary)
- **X**"D9" -- Length 8 (HEX → Equiv to Above String)
- **O**"331" -- Length 9 (Octal)

VHDL Lexical Elements

(iii) Bit String Literals (Contd)

- *Used to Specify Initial Contents of Registers*
- *Value of Bit-String is Equivalent to a String of Bits, However, Interpreting This Value is a User Choice*

Examples:

- X"A" -- Represents the String 1010
  -- Interpreted as a Decimal Value of ten
  -- if it represents an Unsigned Number.
  -- Interpreted as -6 if it represents a
  -- signed 2`s Complement Number

(iv) Abstract (Numeric) Literals

1. *Default is Decimal*
2. *Other Bases Are Possible (Bases Between 2 and 16)*
3. *Underscore Char May Be Used to Enhance Readability*
4. *Scientific Notations Must Have Integer Exponent*
5. *Integer Literals **Should Not Have** Base Point*
6. *Integer Literals **Should Not Have** -ive Exponents*
7. *Real Literals, **Should Have** a Base Point which **Must Be Followed** By AT LEAST ONE DIGIT*
8. *No Spaces Are Allowed*

VHDL Lexical Elements

(iv) Abstract (Numeric) Literals (Contd)

Examples:

0 1 123_987_456 73E13 -- Integers

0.0 2.5 2.7_456 73.0E-2 12.5E3 -- Reals

Special Case (BASED LITERALS)

- ***General Base*** *Abstract Literals (Including Decimal)*

Based_Literal::=Base#Based_Integer[.Based_Integer]#[Exponent]

Based_Integer::=Extd_Digit { [Underscode] Extd_Digit }

Extd_Digit::=digit | Letters_A-F

- *Both **Base** and **Exponent** are Expressed in **Decimal***
- ***Base** Must be Between 2 & 16*
- *Digits Are Extended to Use the HEX Characters A-F*

Examples:

The Following Represent Integer Value of 196

2#1100_0100**#** , **16#**C4**#**

4#301**#E1** , **10#**196**#**

The Following Represent Real Value of 4095.0

2#1.1111_1111_111**#E11** , **16#**F.FF**#E2**

10#4095.0**#**

VHDL Lexical Elements

- Formal grammar of the IEEE Standard 1076-1993 VHDL language in BNF format
  - Appendix E
  - http://www.iis.ee.ethz.ch/~zimmi/download/vhdl93_syntax.html

VHDL Data Types & Objects

OUTLINE

- **Objects**
  - **Constants,**
  - **Variables**
  - **Signals**
- **Data Types**
  - Scalars
    - Numeric (Integer, Real)
    - Enumerations
    - Physical
  - **Composite**
    - **Arrays, and**
    - **Records**
- **VHDL Operations**

© Dr. Alaaeldin Amin

Dr. Alaaeldin Amin

Data Types & objects

objects

- VHDL ***OBJECT*** : Something that can ***Hold a Value*** of a Given ***Data Type***.
- VHDL has 3 Object Categories:
  - CONSTANTS
  - VARIABLES
  - SIGNALS

Examples

Constant Rom_Size **: Integer** *:= 2**16*;

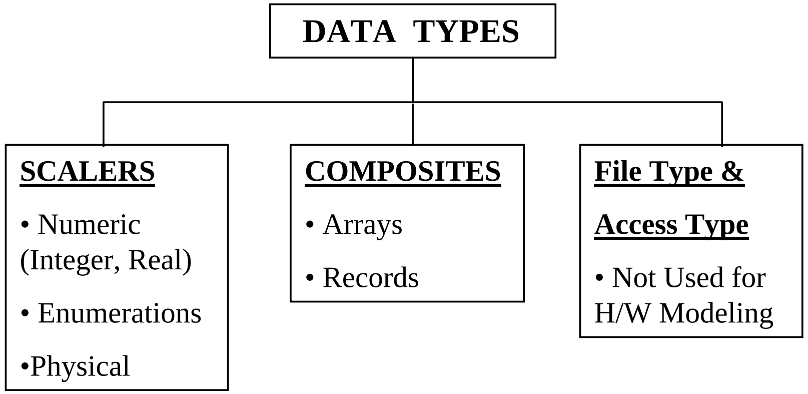
Variable Busy, Active **: Boolean** *:= False*;

Signal Reset**: Bit** *:= '0'*;

- Every Object & Expression Must Unambiguously Belong to One *Named* ***Data Type***
- A ***Data Type*** Defines a *Set of Values* & a *Set of Operations*.
- VHDL is a **Strongly-Typed** Language. *Types Cannot be Mixed* in Expressions or in Assigning Values to Objects in General

© Dr. Alaaeldin Amin

DATA TYPES



DATA TYPES

SCALERS

- Numeric (Integer, Real)
- Enumerations
- Physical

COMPOSITES

- Arrays
- Records

File Type & Access Type

- Not Used for H/W Modeling

SCALER DATA TYPES

SYNTAX

TYPE *Identifier* **IS** *Type-Definition*

(I) Numeric Data Type

- *Type-Definition* is a *Range_Constraint* as follows:

Type-Definition := **Range** *Initial-Value* < **To** | **DownTo**> *Final-Value*

Ascending Range *Descending Range*

Examples

```
TYPE address IS RANGE 0 To     127;
TYPE index   IS RANGE 7 DownTo 0;
TYPE voltage IS RANGE -0.5 To 5.5;
```

Number Formats:

- Integers Have no Base Point.
- Integers may be Signed or Unsigned (e.g. **-5** **356**)
- Integers may not have -ive Exponents (Scientific Notation),
- A Real Number must have a Base Point, and may have -ive Exponents (Scientific Notation).
- Real Numbers May be Signed or Unsigned (e.g. **-3.75** **1.0E-9** **1.5E-12**)

Based Numbers:

- Numbers Default to Base 10 (Decimal)
- VHDL Allows Expressing Numbers Using Other Bases

Syntax

B#*nnnn*# -- Number *nnnn* is in Base **B**

Examples

```
16#DF2#          -- Base 16 Integer  (HEX)
8#7134#          -- Base 8 Integer  (OCTAL)
2#10011#         -- Base 2 Integer  (Binary)
16#65_3EB.37#    -- Base 16 REAL  (HEX)
```

Predefined numeric data types

1- **INTEGER** -- *Range is Machine limited but At Least -(2^{31} - 1) To (2^{31} - 1)*

2- **Positive** -- *INTEGERS > 0*

3- **Natural** -- *INTEGERS ≥ 0*

4- **REAL** -- *Range is Machine limited*

(II) Enumeration Data Type

- Parenthesized Ordered List of Literals. Each May be an *Identifier* or a *Character Literal*. The List Elements are Separated By Commas
- A Position # is Associated with Each Element in The List
- *Position #'s* Begin with ***0*** for the ***Leftmost*** Element
- Variables & Signals of type ENUMERATION will have the *Leftmost Element* as their *Default* (Initial) Value unless, otherwise Explicitly Assigned.

Examples

TYPE Color **IS** **(** Red, Orange, Yellow, Green, Blue, Indigo, Violet)**;**

TYPE Tri_Level **IS** **(** '0', '1', 'Z')**;**

TYPE Bus_Kind **IS** **(** Data, Address, Control)**;**

TYPE state **IS** **(**Init, Xmit, Receiv, Wait, Terminal)**;**

Predefined Enumerated Data Types

1- **TYPE BIT IS** ('0' , '1') **;**

2- **TYPE BOOLEAN IS** (False, True) **;**

3- **TYPE CHARACTER IS** (*128 ASCII Chars......*) **;**

4- TYPE Severity_Level IS **(**Note, Warning, Error, Failure) **;**

```
5- TYPE Std_U_Logic IS (
        'U' , -- Uninitialized
        'X' , -- Forcing Unknown
        '0' , -- Forcing 0
        '1' , -- Forcing 1
        'Z' , -- High Impedence
        'W' , -- Weak Unknown
        'L' , -- Weak 0
        'H' , -- Weak 1
        '-' , -- Don't Care
        ) ;
```

6- **SUBTYPE Std_Logic IS resolved** Std_U_Logic **;**

(III) Physical Data Type

- Specifies a ***Range Constraint*** , One ***Base Unit***, and 0 or More ***Secondary Units***.
- Base Unit is Indivisible, i.e. No Fractional Quantities of the Base Units Are Allowed.
- Secondary Units Must be Integer Multiple of the ***Indivisible*** Base Unit.

Examples

```
TYPE  Resistance IS  Range 1  To  10E9
    Units
      Ohm;        -- Base Unit
      Kohm = 1000  Ohm;     -- Secondary  Unit
      Mohm = 1000  Kohm;    -- Secondary  Unit
    end  Units ;
```

Predefined Physical data types

- *Time* is the ONLY Predefined Physical Data Type

```
TYPE  Time  IS Range 0 To  1E20
        Units
          fs;          -- Base Unit (Femto*  Second)
          ps = 1000  fs;   -- Pico_Second
          ns = 1000  ps;   -- Nano_Second
          us = 1000  ns;   -- Micro_Second
          ms = 1000  us;  -- Milli_Second
          sec = 1000  ms; -- Second
          min = 60  sec;    -- Minuite
          hr   = 60  min;    -- Hour
        end  Units ;
```

* *Femto = 1E-15*

Composite Data Types

(I) Arrays

Elements of an Array Have the ***Same*** Data Type

- Arrays May be ***Single / Multi - Dimensional***
- Array Bounds may be either ***Constrained*** or ***Unconstrained***.

(a) Constrained Arrays

- Array *Bounds Are Specified*

Syntax:

TYPE *id* Is Array (*Range_Constraint*) of ***Type***;

Examples

TYPE ***word*** Is Array (0 **To** 7) of Bit;

TYPE pattern Is Array (31 **DownTo** 0) of Bit;

2-D Arrays

TYPE col Is Range 0 **To** 255;

TYPE row Is Range 0 **To** 1023;

TYPE Mem_Array Is Array **(row, col)** of Bit;

TYPE Memory Is Array **(row)** of ***word***;

© Dr. Alaaeldin Amin

(b) Unconstrained Arrays

- Array *Bounds Not Specified* Through Using the notation **RANGE<>**
- ***Type*** of each Dimension is *Specified*, but the *exact* ***Range*** and ***Direction*** are *not Specified*.
- Useful in Interface_Lists → Allows Dynamic Sizing of Entities , e.g. Registers.
- Bounds of Unconstrained Arrays in Such Entities Assume the Actual Array Sizes When Wired to the Actual Signals.

Example

TYPE Screen **Is Array (Integer Range <> , Integer Range<>) of** BIT;

© Dr. Alaaeldin Amin

Predefined array types

Two ***UNCONSTRAINED*** Array Types Are Predefined

1) BIT_VECTOR

TYPE Bit_Vector **Is Array (** Natural **Range<>) of** Bit;

2) String

TYPE String **Is Array (** Positive **Range<>) of** Character;

Example

SUBTYPE Pixel **Is** Bit_Vector (7 **DownTo** 0);

Referencing arrays & array elements

- VHDL allows referencing an Array in its *Entirety* or By a *SLICE*, or *Element*.

EXAMPLE

```
TYPE   clock_state  IS  (Low, Rising, High, Falling);
TYPE   Conversion_Array IS Array  (Clock_state) of  Bit;
Signal   C_A : Conversion_Array :=  ('0' , '1', '0', '1') ;

C_A <= ('1', '1', '0', '0');            -- Positional Association List

C_A <= (High => ‘0', Rising => '1', Low => ‘1',  Falling =>
          '0');                         -- Named Association List
C_A <= (Low => '0', Falling => '0', OTHERS=> '1');
                                        --   Alternative 3
C_A(Low)  <= '0';
```

```
TYPE   Register4 IS Array  (3 Downto 0) of  Bit;
TYPE   Reg4      IS Array  (0 To 3)       of  Bit;
Signal   A: Register4 := ('0' , '1', '0', '1') ;--A(0)='1', A(3)='0'
Signal   B: Reg4      := ('0' , '1', '0', '1') ;--B(0)='0', B(3)='1'
```

Referencing arrays & array elements

2-D Arrays

```
   TYPE Reg32 Is Array (31 DownTo 0)  of   Bit;
   TYPE ROM  Is Array (0 To 3)  of  Reg32;
   TYPE ROM2 Is Array (0 To 4 ,  0 To 2)  of  Bit;

Signal   A: ROM := (X"2F3C_5456" ,  X"FF32_E7B8"  ,
                    X"109A_BA15" ,  X"FFFF_FFFF" );


Signal   B:  ROM2   :=        ( ('1', '0', '0'),
                                 ('0' , '1', '0'),
                                 ('0' , '1', '1',
                                 ('1' , '0', '1'),
                                 ('1' , '1', '1') ) ;
 B(1 , 2) <= '0' ; -- Referencing a 2-D Array Element
```

Examples on Referencing arrays

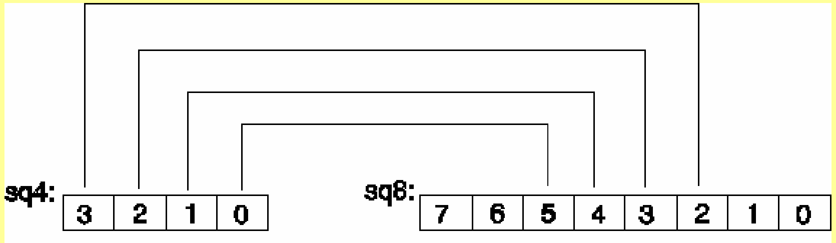
```
TYPE   qit  IS  (‘0’ , ‘1’ , ‘Z’ , ‘X’);
TYPE qit_nibble IS ARRAY ( 3 DOWNTO 0 ) OF qit;
TYPE qit_byte IS ARRAY ( 7 DOWNTO 0 ) OF qit;
TYPE qit_word IS ARRAY ( 15 DOWNTO 0 ) OF qit;
TYPE qit_4by8 IS ARRAY ( 3 DOWNTO 0 , 0 TO 7 )
         OF qit;       -- 2-D array
TYPE qit_nibble_by_8 IS ARRAY ( 0 TO 7 ) OF
         qit_nibble;
SIGNAL sq1 : qit ;
SIGNAL sq4 : qit_Nibble ;
SIGNAL sq8 : qit_byte := "ZZZZZZZZ“  ;
SIGNAL sq16 : qit_word ;
SIGNAL sq_nibble_8 : qit_nibble_by_8 ;
     sq8 <= "Z101000Z" ;
     sq8 <= sq16 (11 DOWNTO 4); -- middle 8 bit slice of
                                 -- sq16 to sq8
```

Direction of indexing must be as declared

```
sq16 (15 DOWNTO 12) <= sq8(5 DOWNTO 2) ;
     -- sq8 Middle Nibble into left 4 bit slice of sq16
sq4 <= sq_nibble_8(2) ;-- third nibble of sq_nibble_8 into
                             -- sq4
sq1 <= sq_nibble_8 (2)(1) ;
```

More Examples on Referencing arrays

```
TYPE   qit  IS  (‘0’ , ‘1’ , ‘Z’ , ‘X’);
TYPE qit_nibble IS ARRAY ( 3 DOWNTO 0 ) OF qit;
TYPE qit_byte IS ARRAY ( 7 DOWNTO 0 ) OF qit;
     --
     SIGNAL sq4 : qit_Nibble ;
     SIGNAL sq8 : qit_byte ;
          --
          sq8 <= sq8 (0) & sq8 (7 DOWNTO 1) ; -- right
             rotate sq8


          sq4 <= sq8 (2) & sq8 (3) & sq8 (4) & sq8 (5) ; --
             reversing
```



- **Concatenation operator “&”can be used for shift and rotate**

Examples on Referencing arrays

```
TYPE qit_4by8 IS ARRAY ( 3 DOWNTO 0 , 0 TO 7 )
   OF qit;         -- 2-D array
--
SIGNAL sq_4_8 : qit_4by8 :=
    (                          Outer Parenthesis

      ( '0', '0', '1', '1', 'Z', 'Z', 'X', 'X' ), -- sq_4_8 (3, 0 TO 7)

      ( 'X', 'X', '0', '0', '1', '1', 'Z', 'Z' ), -- sq_4_8 (2, 0 TO 7)

      ( 'Z', 'Z', 'X', 'X', '0', '0', '1', '1' ), -- sq_4_8 (1, 0 TO 7)

      ( '1', '1', 'Z', 'Z', 'X', 'X', '0', '0' ) -- sq_4_8 (0, 0 TO 7)

    );
```

- **Example shows initialization of *sq_4_8***
- **Use nested parenthesis for multidimensional arrays**
- **Deepest set of parenthesis corresponds to right most index.**

Composite data types

(II) Records

- Elements of a Record Are Heterogeneous (not Necessarily of the ***Same*** Data Type)

Examples

```
TYPE opcode Is ( STA, LDA, ADD, JMP);
TYPE mode Is Range 0 To 3;
SubType  Address Is Bit_Vector(7 DownTo 0);
TYPE Instruction is
        Record
                OPC: opcode ;
                M   : mode ;            -- Addressing mode
                OP1, OP2 : Address ;
        End record ;
```

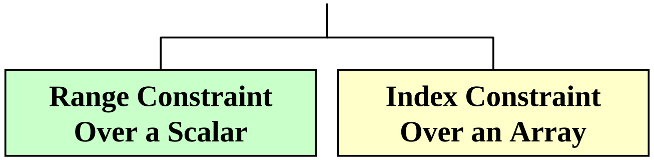
Referencing Record Elements:

```
TYPE Instr_Q  Is Array (0 To 15)   of Instruction ;
SIGNAL  IQ : Instr_Q ;
IQ(0) <= (LDA, 2,  x"F3", x"13"); --Positional Association
Alternatively           IQ(0).OPC <= LDA;
                        IQ(0).M <= 2;
                        IQ(0).OP1 <=  X”F3”;
                        IQ(0).OP2 <=  X"13" ;
Alternatively
IQ(0) <= (M => 2,  OP2 => X"13", OPC => LDA,
            OP1 => X"F3"); --Named  Association
```

subtypes

- A ***SUBTYPE*** Defines a **SUBSET** of Values Defined by a **TYPE** Declaration.
- Subtypes of Subtypes are also possible

SUBTYPE = *Constrained* "TYPE" or **"SubType"**


Range Constraint Over a Scalar
Index Constraint Over an Array

Example (i) **Range Constrained Subtypes**

SubType Lower_Case **Is** Character **Range** 'a' **To** 'z';
SubType Positive **Is** *Integer* **Range** 1 **To** Integer'High**;**
SubType Natural **Is** *Integer* **Range** 0 **To** Integer'High**;**

Predefined Types

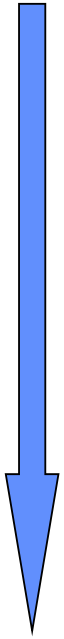
Example (ii) **Index Constrained Subtypes**

SubType Byte **Is** Bit_Vector (7 **DownTo** 0**);**

© Dr. Alaaeldin Amin

VHDL Operations

```
                     PREDEFINED OPERATORS

      LOGICAL OPERATORS: NOT AND OR NAND NOR XOR
           OPERAND TYPE: BIT BOOLEAN
            RESULT TYPE: BIT BOOLEAN

   RELATIONAL OPERATORS: = /  = < <= > >=
           OPERAND TYPE: any type
            RESULT TYPE: BOOLEAN

   ARITHMETIC OPERATORS: + - * / **
                       : MOD REM ABS
           OPERAND TYPE: INTEGER REAL physical
            RESULT TYPE: INTEGER REAL physical

 CONCATENATION OPERATOR: &
           OPERAND TYPE: array of any type
            RESULT TYPE: array of any type
```



Higher Precedence Operators

"mod" and the "rem" Operations

Integer division **"/"** and the ***reminder*** operator **(rem)** are related by the following formula:

A = (A/B)*****B **+** (A **rem** B),

where

1. **Sign of (A rem B)** = **sign of A**, and
2. **|A rem B | < |B|**

The mod operation must satisfy:

A = B*N + **(A mod B),**

where

1. **N** *is Integer*.
2. **Sign of (A mod B)** = **sign of B**, and
3. **|A mod B | < |B|**

Example

A	B	A / B	A rem B	A mod B
11	4	2	3	3
-11	4	-2	-3	1
11	-4	-2	3	-1
-11	-4	2	-3	-3

Type Compatibility &

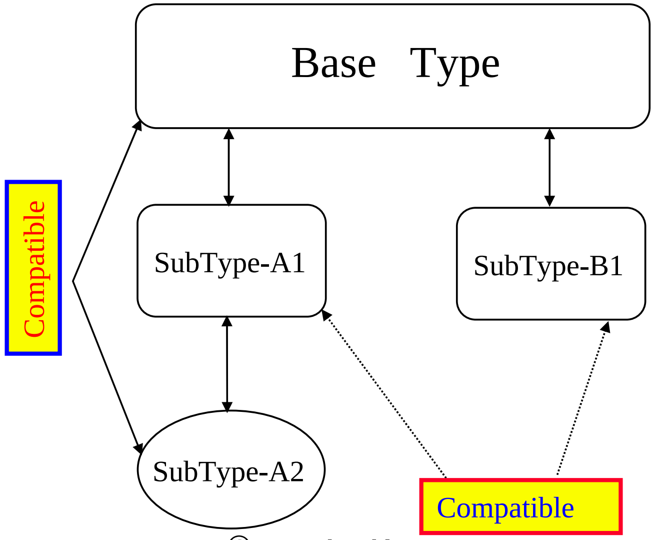
Type Conversion

OUTLINE

- **Introduction**
- **Closely Related Types**
- **Mixed Type Arithmetic**
- **Type Conversion Using Functions**
- **Type Attributes**
- **Array Attributes**

Type Compatibility & conversion

- VHDL is a ***Strongly-Typed*** Language.
- Compiler Flags an Error Whenever Different Types are Mixed.
- Subtypes are Type compatible with their *higher level* Subtypes and their Root Type.
- Two Subtypes of the Same type Are also Type-Compatible
- Type of an expression assigned to an object must be the same as the type of the object.
- Operands of predefined operators must be of the same type



Closely Related Types

- A Type is Closely Related to itself.
- Any Two Numeric Types Are Closely Related
- Type Casting May be used for type conversion
  - I := Integer(X); -- Rounds X
  - X :=Real(I);
  - I := Integer(3.5); -- Ambiguous (Implementation dependent)
- Array Types Are Closely Related Iff :
  - Same Dimensionality
  - Index Types Are Closely Related for Each Dimension
  - Elements Types Are The Same

Example

```
SubType  Minuites  Is  Integer Range  0  To  59;
SubType  Seconds   Is  Integer Range  0  To  59;
SubType  X_int  Is   Minuites  Range  1  To  30;

Variable  x  : X_int ;
Variable  M : Minuites ;
Variable  S  : Seconds ;
Variable  I   : Integer ;
Variable  r   : Real ;

I := 60*M + S; -- Legal I, M & S are compatible types
I := 60*Integer(r) + S ;  -- Valid
r := Real (M);          -- Legal
r := 3*Real(x);    -- Illegal – 3 is Integer
M := Minuites(r/60);  --Illegal Incompatible Types (60 Integer)
M := Minuites(r/Real(60));  --Legal
```

© Dr. Alaaeldin Amin

Mixed Type Arithmetic

- ***Explicit*** *Type Conversion* is Done Between Closely-Related Types, e.g. *REALs & INTEGERs*

Example:

```
Variable x, y : Real;
Variable n,m : Integer;
    n := INTEGER (x) * m;  -- x is first converted to Integer
    y := REAL (n) * x;        -- n is first converted to Real
```

Example:

```
TYPE qit_byte IS ARRAY ( 7 DOWNTO 0 ) OF qit;
TYPE qit_octal IS ARRAY ( 7 DOWNTO 0 ) OF qit;
Signal qb: qit_byte;
Signal qo: qit_octal;

qb <= qit_byte(qo); - - Explicit Type Conversion (Type Casting)
                    - -  of closely-related types
```

- Custom Type Conversions can be defined Using either:
  - **Constant** Conversion Arrays, or
  - **Subprograms** (Functions or Procedures)
- Type Conversion Arrays or Subprograms may be placed within packages, e.g. functions already in predefined standard packages may also be used, e.g. the package

  std_logic_1164 defined within the "**ieee**" Library

© Dr. Alaaeldin Amin

Type Conversion Using Functions

Example:

```
Type  MVL4 ('X', '0', '1', 'Z');
Function MVL4_To_Bit(B: in MVL4) Return Bit IS
Begin
    Case B is
          when 'X' => return '0';
          when '0' => return '0';
          when '1' => return '1';
          when 'Z' => return '0';
    End Case;
End MVL4_To_Bit ;
-----------------------------------------------------------------------------------------------
Function Bit_To_ MVL4(B: in Bit) Return MVL4 IS
Begin
    Case B is
          when '0' => return '0';
          when '1' => return '1';
    End Case;
End MVL4_To_Bit ;
-----------------------------------------------------------------------------------------------
Signal B4: MVL4;
Signal  B: Bit;
    B <= MVL4_To_Bit(B4);
    B4 <= Bit_To_ MVL4(B);
```

Array Attributes

- A Predefined Attribute is a *named* Feature of an Array or Data Type

•Value of **ARRAY Attribute** is Referenced as:

Array_Name'Attribute_ID

Pronounced **Tick**

•Find (1) Range, (2)Length, (3)Boundary of an Array Object or Array Type

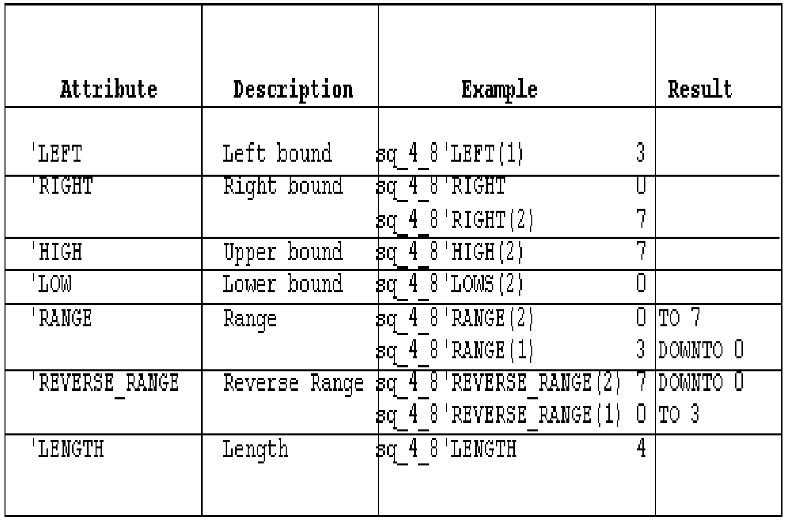
A'LEFT(N)	Left Bound Of The Nth Index Of the Array Object Or Subtype. Optional Parameter. Default Is 1.
A'RIGHT(N)	Right Bound Of The Nth Index Of Array Object Or Subtype. Optional Parameter. Default Is 1.
A'HIGH(N)	Upper Bound Of Nth Index Of Array Object or Subtype. Optional Parameter. Default is 1.
A'LOW(N)	Lower Bound Of Nth Index Of Array Object Or Subtype. Optional Parameter. Default Is 1.
A'RANGE(N)	Range Of Nth Index Of Array Object Or Constrained Array Subtype Ascending: *Left Bound* **To** *Right Bound* *Descend: Left Bound* **Downto** *Right Bound*
A'REVERSE _ RANGE(N)	Identical To A'RANGE(N) *Except* Range Is Reverse; *i.e.*. Ascending, *Right Bound* **Downto** *Left Bound* Descending, *Right Bound* To *Left Bound*.
A'LENGTH(N)	Number Of Values In The Nth Index Of Array Object Or Constrained Array Subtype. Optional Parameter. Default Is 1.

Array Attributes

- Find (1) Range, (2)Length, (3)Boundary of an Array Object or Array Type
- Follow attribute by ( ) to specify index

TYPE qit_4by8 IS ARRAY (3 DOWNTO 0, 0 TO 7) OF qit;

SIGNAL sq_4_8 : qit_4by8;



Attribute	Description	Example	Result
'LEFT	Left bound	sq_4_8'LEFT(1) 3	
'RIGHT	Right bound	sq_4_8'RIGHT 0 sq_4_8'RIGHT(2) 7	
'HIGH	Upper bound	sq_4_8'HIGH(2) 7	
'LOW	Lower bound	sq_4_8'LOWS(2) 0	
'RANGE	Range	sq_4_8'RANGE(2) 0 sq_4_8'RANGE(1) 3	TO 7 DOWNTO 0
'REVERSE_RANGE	Reverse Range	sq_4_8'REVERSE_RANGE(2) 7 sq_4_8'REVERSE_RANGE(1) 0	DOWNTO 0 TO 3
'LENGTH	Length	sq_4_8'LENGTH 4	

Type Attributes

- A Predefined Attribute is a *named* Feature of a Type
- Value of Attribute is Referenced as:
- Type_Name'Attribute_ID

Pronounced **Tick**

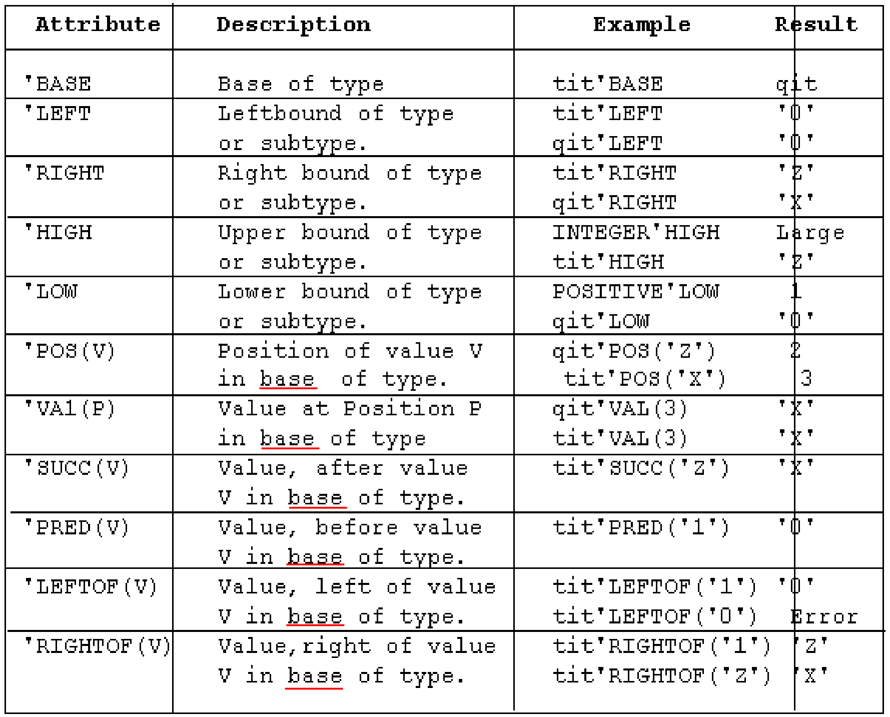
Attributes of Types and SubTypes:

T'LEFT :	Left Bound Of Scalar Type T.
T'RIGHT:	Right Bound Of Scalar Type T.
T'HIGH:	Upper Bound Of Scalar Type T.
T'LOW:	Lower Bound Of Scalar Type T.
T'POS(X):	Position Within The Enumeration Or Physical Type Of The Value Of The Parameter X.
T'VAL(X):	Value of Enumeration (Or Physical) Type Element at Position X
T'SUCC(X):	Value of Enumeration (Or Physical) Type Element at Position (X+1)
T'PRED(X):	Value of Enumeration (Or Physical) Type Element at Position (X-1)
T'LEFTOF(X):	Value of Enumeration (Or Physical) Type Element to the Left of X
T'RIGHTOF(X):	Value of Enumeration (Or Physical) Type Element to the Right of X
T'BASE:	Base Type Of Type T.

Examples

TYPE qit IS ('0', '1', 'Z', 'X');

SUBTYPE tit IS qit RANGE '0' TO 'Z';



Attribute	Description	Example	Result
'BASE	Base of type	tit'BASE	qit
'LEFT	Leftbound of type or subtype.	tit'LEFT qit'LEFT	'0' '0'
'RIGHT	Right bound of type or subtype.	tit'RIGHT qit'RIGHT	'Z' 'X'
'HIGH	Upper bound of type or subtype.	INTEGER'HIGH tit'HIGH	Large 'Z'
'LOW	Lower bound of type or subtype.	POSITIVE'LOW qit'LOW	1 '0'
'POS(V)	Position of value V in base of type.	qit'POS('Z') tit'POS('X')	2 3
'VA1(P)	Value at Position P in base of type	qit'VAL(3) tit'VAL(3)	'X' 'X'
'SUCC(V)	Value, after value V in base of type.	tit'SUCC('Z')	'X'
'PRED(V)	Value, before value V in base of type.	tit'PRED('1')	'0'
'LEFTOF(V)	Value, left of value V in base of type.	tit'LEFTOF('1') tit'LEFTOF('0')	'0' Error
'RIGHTOF(V)	Value,right of value V in base of type.	tit'RIGHTOF('1') tit'RIGHTOF('Z')	'Z' 'X'

FOR the enumeration types: **'PRED = 'LEFTOF**

'SUCC = 'RIGHOF

'LEFT = 'LOW

'RIGHT = 'HIGH

- Position wise, enumeration elements are in ascending order

- For types with ascending range **low to high** is **left to right**

COE 405
VHDL Objects and Signals

Dr. Alaaeldin A. Amin
Computer Engineering Department

E-mail: amin@ccse.kfupm.edu.sa

Home Page : http://www.ccse.kfupm.edu.sa/~amin

Outline

- VHDL Objects
- Variables vs. Signals
- Signal Assignment
- Signal Transaction & Event
- Delta Delay
- Transport and Inertial Delay
- Sequential Placement of Transactions
- Signal Attributes

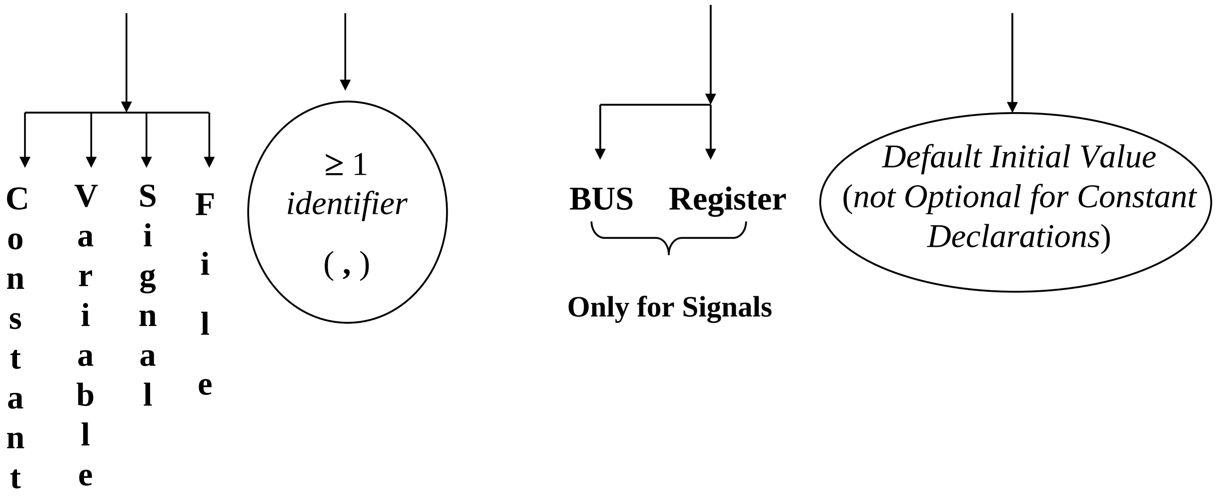
VHDL Objects …

- VHDL *OBJECT* : Something that can hold a value of a given ***Data Type***.
- VHDL has 3 classes of objects
  - CONSTANTS
  - VARIABLES
  - SIGNALS
- Every object & expression must unambiguously belong to one *named* ***Data Type***
- Every object must be ***Declared.***

… VHDL Object …

Syntax

***Obj_Class* <*id_list*> : Type/SubType [*signal_kind*] [:= *expression*];**



… VHDL Object …

- Value of Constants *must* be specified when declared
- *Initial* values of Variables or Signals *may* be specified when declared
- If not explicitly specified*, Initial* values of Variables or Signals *default* to the value of the Left Element in the type range specified in the declaration.
- Examples:
  - **Constant** Rom_Size **:** Integer := 2**16**;**
  - **Constant** Address_Field **:** Integer := 7**;**
  - **Constant** Ovfl_Msg **:** String (**1 To** 20) := "Accumulator OverFlow"**;**
  - **Variable** Busy, Active **:** Boolean := False**;**
  - **Variable** Address **:** Bit_Vector (0 **To** Address_Field) := "00000000"**;**
  - **Signal** Reset**:** Bit := '0'**;**

Variables vs. Signals

VARIABLES	SIGNALS
* Variables are only *Local* and May Only Appear within the Body of a Process or a SubProgram * Variable Declarations Are Not Allowed in Declarative Parts of Architecture Bodies or Blocks.	* Signals May be Local or Global. * Signals May not be Declared within Process or Subprogram Bodies. * All Port Declarations Are for Signals.
A Variable Has No HardWare Correspondence	A Signal Represents a Wire or a Group of Wires (BUS)
Variables Have No ***Time*** Dimension Associated With Them. (*Variable Assignment occurs instantaneously*)	Signals Have ***Time*** Dimension (*A Signal Assignment is Never Instantaneous* (Minimum Delay = δ Delay)
Variable Assignment Statement is always ***SEQUENTIAL***	Signal Assignment Statement is Mostly ***CONCURRENT*** (*Within Architectural* Body). It Can Be **SEQUENTIAL** (*Within Process Body*)
Variable Assignment Operator is **:=**	Signal Assignment Operator is <=

Variables vs. Signals

❖**Variables Within Process Bodies are STATIC, i.e. a Variable Keeps its Value from One Process Call to Another.**

❖**Variables Within Subprogram Bodies Are Dynamic, i.e. Variable Values are Not held from one Call to Another.**

Simulation Algorithm

- **Initialization phase**
  - each signal is assigned its initial (or default) value
  - simulation time set to 0
  - for each concurrent construct / process
    - Activate / execute
      - execution usually involves scheduling transactions on signals for later times
- **Repeat The folllowing (Simulation cycle** (1 δ time) **)**
  - Advance simulation time to time of next transaction.
  - for each transaction at this time
    - update signal values
    - Generate events for signals whose new values are different from old values
    - Schedule Activate processes & concurrent constructs sensitive to any of these signal events.
- **Simulation finishes when there are no further scheduled transactions**

Signal Assignments …

- Syntax:

  Target Signal <= [Transport] *Waveform* ;

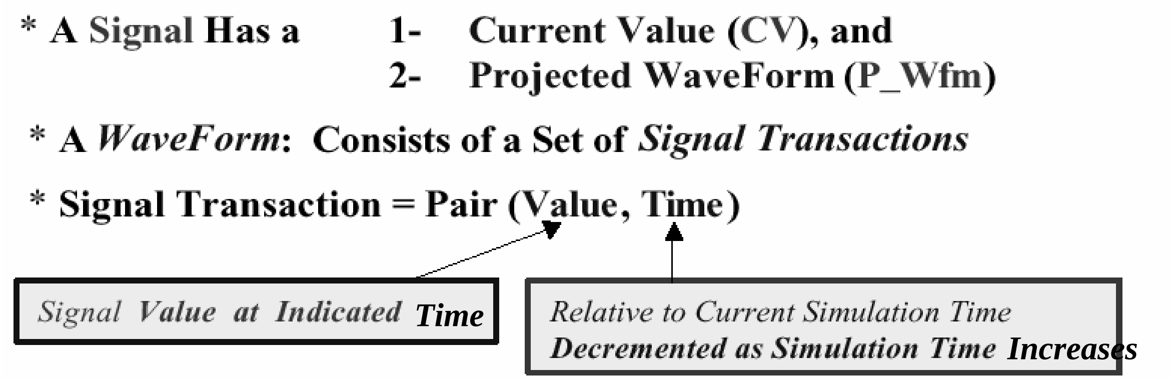
  Waveform := *Waveform_element* {, *Waveform_element* }

  Waveform_element := Value_Expression [*After Time_Expression*]

- **Examples**:
  - X <= ‘0’ ; -- Assignment executed After δ delay
  - S <= ‘1’ After 10 ns;
  - Q <= Transport ‘1’ After 10 ns;
  - S <= ‘1’ After 5 ns, ‘0’ After 10 ns, ‘1’ After 15 ns;
- Signal assignment statement
  - mostly **concurrent** (within architecture bodies)
  - can be **sequential** (within process body ONLY)

… Signal Assignments

- Concurrent signal assignments are order independent
- Sequential signal assignments are order dependent
- Concurrent signal assignments are executed
  - Once *at the beginning* of simulation (Initialization phase)
  - Any time a signal on the right hand side changes



* **A Signal Has a 1- Current Value (CV), and**
2- Projected WaveForm (P_Wfm)

* **A *WaveForm*: Consists of a Set of *Signal Transactions***

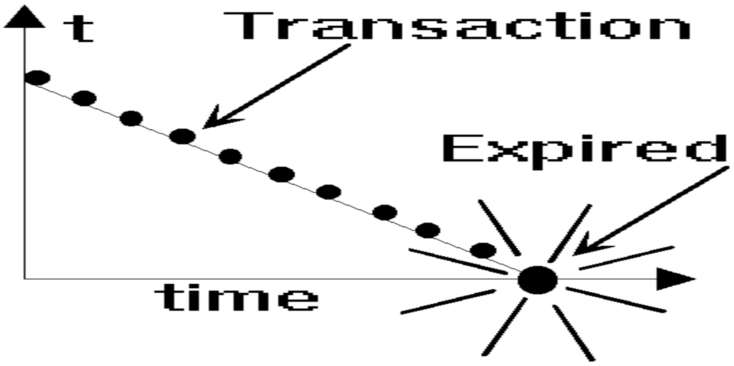
* **Signal Transaction = Pair (Value, Time)**

Signal Value at Indicated **Time**

Relative to Current Simulation Time
Decremented as Simulation Time Increases

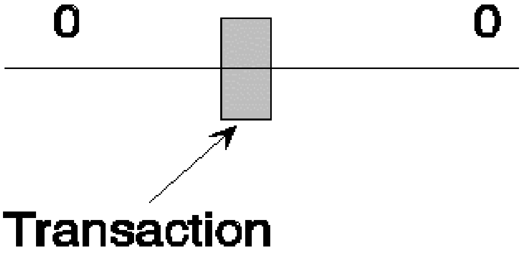
Signal Transaction

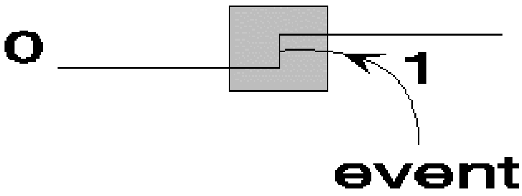
- When the time element of a signal transaction expires (t=0)
  - Its associated value is made the current value (CV) of a signal
  - The transaction is deleted from the list of transactions forming the Projected Waveform (P_Wfm) of the signal



Signal Transactions & Events …

- When a new value is assigned to a signal, it is said that
  - a Transaction has been *Scheduled* for this signal
  - or a Transaction has been placed on this *Signal Driver*
- A Transaction which does not cause a signal transition (Event) is still a Transaction
- A Transaction *May/May not* cause a signal transition (Event) on the target signal





… Signal Transaction & Events …

- A <= ‘1’ After 10 ns, ‘0’ After 20 ns, ‘1’ After 30 ns;
- Executing the above assignment defines the following values & waveform for signal A

	t=0	t=5 ns	t=10 ns	t=20 ns	t=30 ns
A (CV)	‘0’	‘0’	‘1’	‘0’	‘1’
A (P_Wfm)	(‘1’, 10ns) (‘0’, 20ns) (‘1’, 30ns)	(‘1’, 5ns) (‘0’, 15ns) (‘1’, 25ns)	(‘0’, 10ns) (‘1’, 20ns)	(‘1’, 10ns)	

Scope of Signal Declarations

- Signals declared within a Package are Global usable by all Entities using this package
- Signals declared within an Entity are usable by all architectural bodies of this entity
- Signals declared within an Architectural body are only usable within this Architectural body

Delta Delay …

- If no Time Delay is *explicitly specified*, Signal assignment is executed after an infinitesimally small δ-delay
  - Delta is the duration of a simulation cycle , and is *<u>not physical / real time</u>*
  - An infinite number of deltas still add up to zero seconds
  - Delta is used for scheduling

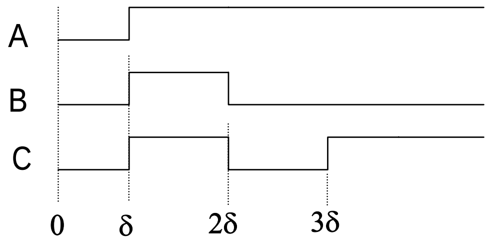
Delta Delay - Example…

```
ARCHITECTURE concurrent
OF timing_demo IS
SIGNAL a, b, c : BIT := '0';
BEGIN
        a <= '1';
        b <= NOT a;
        c <= NOT b;
END concurrent;
```

Notes:

- The 3 Assignment Statements Are *Concurrent* (Order-Independent)
- They Are All *Executed* at the *Beginning of Simulation* Assuming the *Default Value of ‘0*’.

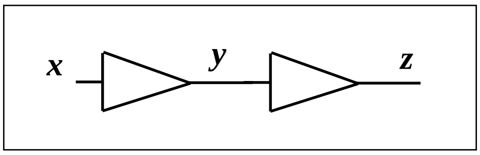
Time	A		B		C	
	CV	P-Wave	CV	P-Wave	CV	P-Wave
0(-)	‘ 0 ’		‘0’		‘0’	
0	‘ 0 ’	(‘1 ’, δ)	‘0’	(‘1 ’, δ)	‘0’	(‘1 ’, δ)
δ	‘1’		‘1’	(‘0 ’ , δ)	‘1’	(‘0 ’ , δ)
2δ	‘1’		‘ 0 ’		‘ 0 ’	(‘1 ’, δ)
3δ	‘1’		‘ 0 ’		‘1’	



Delta Delay in Sequential Signal Assignments …

- Effect of δ-delay should be carefully considered when signal assignments are embedded within a process

```
Entity BUFF2 IS
Port (X: IN BIT;
               Z: OUT BIT);
END BUFF2;
```

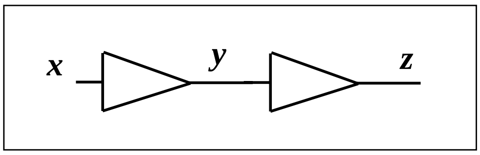


```
Architecture Wrong of BUFF2 IS
Signal y: BIT;
Begin
  Process(x)
  Begin
      y <= x;
      z <= y;
   End Process;
End Wrong;
```

- **Process activated on x-events only**
  - **y ← x(δ)**
  - **z ← y(0)**
- **z gets OLD value of y and not new value of x**

… Delta Delay in Sequential Signal Assignments

```
Architecture OK of BUFF2 IS
Signal y: BIT;
Begin
  Process(x , y)
  Begin
      y <= x;
      z <= y;
   End Process;
End OK;
```



- **Process activated on both x and y events**
- **x changes and process activated**
  - **y ← x; -- y gets x value after δ**
  - **z ← y; -- z gets y(0) value after δ**
- **Process terminated**
- **After δ, y changes and process activated**
- **z gets new y (=x) after δ**
- **Process terminated**

Oscillation in Zero Real Time

```
Architecture forever of oscillating IS
Signal x: BIT :=‘0’;
Signal y: BIT :=‘1’;
Begin
     x <= y;
     y <= NOT x;
End forever;
```

Delta	x	y
+0	0	1
+1	1	1
+2	1	0
+3	0	0
+4	0	1
+5	1	1
+6	1	0
+7	0	0
+8	0	1

COE 405
VHDL Design Organization

Dr. Alaaeldin A. Amin
Computer Engineering Department

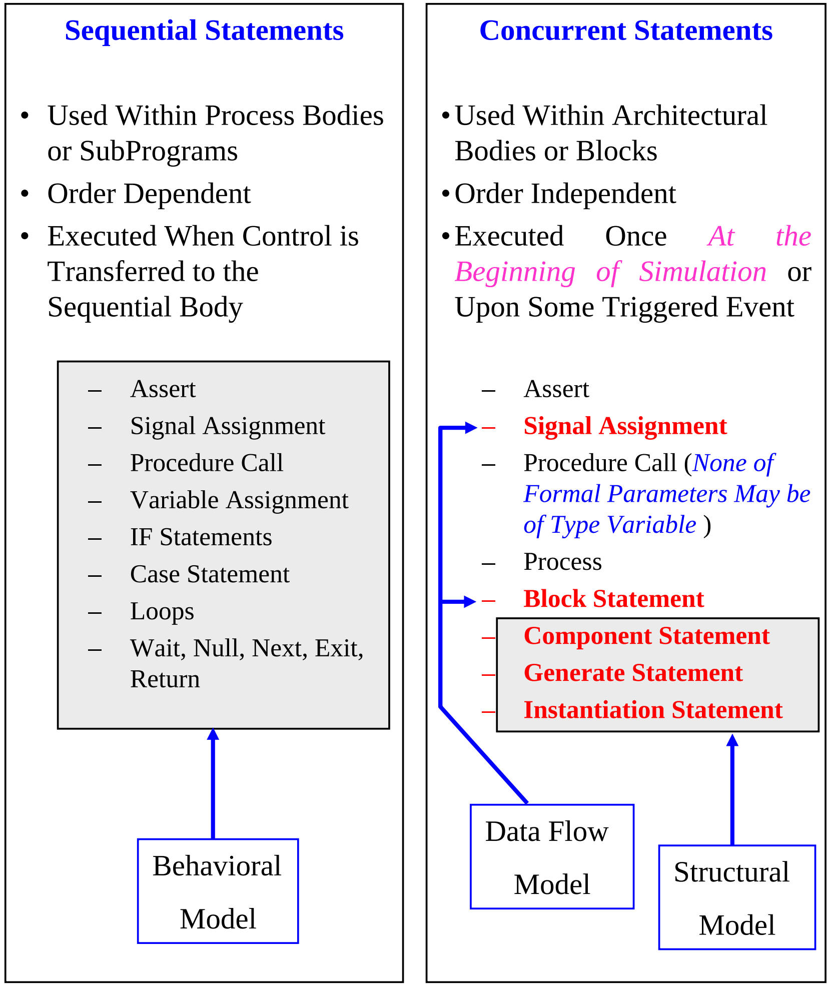
E-mail: amin@ccse.kfupm.edu.sa

http://www.ccse.kfupm.edu.sa/~amin

OUTLINE

- Concurrent vs. Sequential Constructs / Statements
- Concurrent Signal Assignments
- Sequential Statements
- Sequential Bodies
- Overloading
- Packages
- Libraries
- Process Statement
- Modeling FSMs

Concurrent Versus Sequential statements



Sequential Statements

- Used Within Process Bodies or SubPrograms
- Order Dependent
- Executed When Control is Transferred to the Sequential Body

- – Assert
- – Signal Assignment
- – Procedure Call
- – Variable Assignment
- – IF Statements
- – Case Statement
- – Loops
- – Wait, Null, Next, Exit, Return

Behavioral Model

Concurrent Statements

- Used Within Architectural Bodies or Blocks
- Order Independent
- Executed Once *At the Beginning of Simulation* or Upon Some Triggered Event

- – Assert
- – **Signal Assignment**
- – Procedure Call (*None of Formal Parameters May be of Type Variable*)
- – Process
- – **Block Statement**
- – **Component Statement**
- – **Generate Statement**
- – **Instantiation Statement**

Data Flow Model

Structural Model

Concurrent Signal Assignment

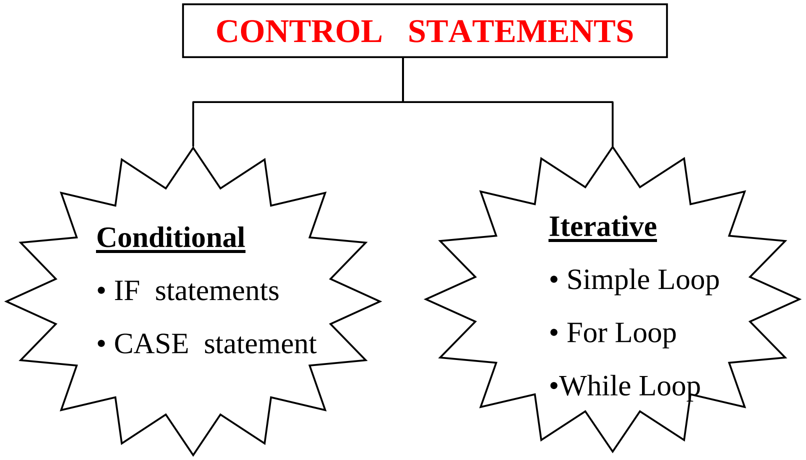
Syntax 1

```
Label :  target   <= [Guarded] [Transport]
                  Wave1 when Cond1 Else
                  Wave2 when Cond2 Else
                  ................................
                  Waven-1 when Condn-1  Else
                  Waven ;
```

Syntax 2

```
With Expression Select
    target <= [Guarded] [Transport]
              Wave1 when Choice1 ,
              Wave2 when Choice2 ,
              ................................
              Waven-1 when Choicen-1 ,
              Waven when OTHERS;
```

Sequential Statements



(I) Conditional control

a) **IF Statements**

Syntax: 3-Possible Forms

(i)

```
IF condition Then
    statements;
End IF;
```

(ii)

```
IF condition Then
    statements;
Else
    statements;
End IF;
```

(iii)

```
IF   condition Then
    statements;
Elsif condition Then
    statements;
Elsif condition Then
    statements;
          :
          :
Elsif condition Then
    statements;
End IF;
```

b) CASE Statement

Syntax:

(i)

```
CASE Expression is
  when value => statements;
  when value1 | value2| ...|valuen => statements;
  when discrete range of values => statements;
  when others => statements;
End CASE;
```

Notes:

- Values/Choices Should not Overlap (*Any value of the Expression should Evaluate to only one Arm of the Case statement*).
- All Possible Choices for the *Expression* Should Be Accounted For *Exactly Once*.
- If "**others**" is used, It must be the last "arm" of the CASE statement.
- There can be Any Number of Arms in Any Order (*Except for the* ***others*** *arm which should be Last*)

Example:

```
 CASE x is
  when 1 => out :=0;
  when 2 | 3 => out :=1;
  when 4 to 7 => out :=2;
  when others => out :=3;
 End CASE;
```

(2) LOOP control

a) Simple Loops

Syntax:

Optional

```
Loop_Label: LOOP
              statements;
        End LOOP Loop_Label;
```

Notes:

- The *Loop_Label* is Optional
- The **exit** statement may be used to exit the Loop. It has two possible Forms:

  1- **exit** *Loop_Label;* -- *This may be used in an if statement*

  2- **exit** *Loop_Label* **when** *condition;*

Example:

```
Process
    variable A:Integer :=0;
    variable B:Integer :=1;
Begin
    Loop1: LOOP
                A := A + 1;
                B := 20;
                Loop2: LOOP
                        IF B < (A * A) Then
                            exit Loop2;
                        End IF;
                        B := B - A;
                    End LOOP Loop2;
                exit Loop1 when  A > 10;
        End  LOOP Loop1;
End Process;
```

b) **For Loop**

Syntax:

Optional

Need Not Be Declared

```
Loop_Label: FOR Loop_Variable  in  range  LOOP
                statements;
        End LOOP Loop_Label;
```

Example:

```
Process
    variable B:Integer :=1;
Begin
    Loop1: FOR  A  in  1  TO  10  LOOP
                B := 20;
                Loop2: LOOP
                        IF B < (A * A) Then
                            exit Loop2;
                        End IF;
                        B := B - A;
                    End LOOP Loop2;
        End  LOOP Loop1;
End Process;
```

c) **WHILE Loop**

Optional

Syntax:

```
Loop_Label: WHILE  Condition  LOOP
                statements;
            End LOOP Loop_Label;
```

Example:

```
Process
    variable B:Integer :=1;
Begin
  Loop1: FOR A in 1 TO 10  LOOP
                B := 20;
            Loop2: WHILE B < (A * A) LOOP
                   B := B - A;
                End LOOP Loop2;
        End LOOP Loop1;
End Process;
```

c) Next Statement

Syntax:

```
Next [Loop_Label][When Condition];
```

- Skip Current Loop Iteration When *Condition* is True
- If *Loop_Label* is Absent, innermost Loop iteration is Skipped When *Condition* is True
- IF *Condition* is Absent, Appropriate Loop Iteration is Skipped .

c) Null Statement

Syntax: **Null;**

- Does Nothing
- Useful in **CASE** Statements If No Action Is Required.

Subprograms

FUNCTIONS

Syntax :

Only ***Input*** Constants (*Default*) or Signals (No Input Variables)

```
FUNCTION function_Name(Input Parameter_List) RETURN
type IS
          {Function Declarative Part}
Begin
          Function Algorithm;
    RETURN Expression;
End function_Name;
```

Examples :

Default is Constant

```
FUNCTION maj3(Signal x, y, z :Bit ) RETURN Bit IS
          variable M : Bit;
Begin
    M := (x and y) or (x and z) or (z and y);
    RETURN M;
End maj3;
```

```
FUNCTION maj3(Signal x, y, z :Bit ) RETURN Bit IS
Begin
   RETURN (x and y) or (x and z) or (z and y);
End maj3;
```

Function Usage Notes

- The Only Allowed Mode For Function Parameters is "***IN***". *No* ***Out*** *or* ***INOUT*** *Parameters Are Allowed*.
- The Only Allowed Object Class for Parameters are **Constants** and **Signals**. If Not Specified, "Constant Is Assumed", (No Variables Parameters are Allowed)
- Since Only parameters of Mode "IN" Are Allowed, Functions Have No Side Effects.
- Parameters of mode "IN" Can only be Read but not Written into
- At least One Return Statement must be included
- Functions Can Be *Recursively* Defined

Subprograms

PROCEDURES

Both ***Input& Output*** Parameters Allowed

Syntax :

```
PEOCEDURE  Procedure_Name (Interface_List)   IS
           {Procedure Declarative Part}
Begin
           Procedure  Algorithm;
End Procedure_Name;
```

Example :

```
TYPE  Bit4  IS ('X', '0', '1', 'Z');
TYPE  Bit4_Vector  IS  array(Integer  range<>) of Bit4;
PROCEDURE Ones_N_Zeros_CNT (X : in Bit4_Vector;
N_Ones, N_Zeros : Out Integer) IS
           variable  N0, N1 : Integer :=0;
Begin
    FOR  i  in X'Range   LOOP
           IF  X(i) = '1'  THEN
                      N1 := N1 + 1;
           ElsIF  X(i) = '0'  THEN
                      N0 := N0 + 1;
           END IF;
    End Loop;
    N_Zeros := N0;
    N_Ones  := N1;
End Ones_N_Zeros_CNT ;
```

Procedures Usage Notes

- Allowed Modes For Procedure Parameters are "*In*", "*Out*", and "*InOut*".
- "IN" Parameters can only be Read, while "OUT" Parameters can only be Written Into
- Allowed Object Classes for Procedure Parameters are **Constants, Variables** and **Signals**. If Mode=In, the Default is **Constant**. If Mode=Out or InOut, the Default is **Variable**. Thus, *Signal Type Parameters Have to be Explicitly Declared*.
- A Signal Formal Parameter can be of Mode **in**, **out** or **inout**.
- Procedure Calls May Be Either Sequential or Concurrent. IF Concurrent, Only *Parameters of Type Constant or Signal May be Used* (*Variables are not Defined Within Concurrent Bodies*)
- Procedures May be Declared within Other Procedures
- Procedure Variables are Dynamic (Don't Maintain Their Values Between Calls)

Parameter Default Values

- Default Values May Be Specified for Parameters of **Mode *In* only**.
- The Parameter Must be either Constant or Variable (Not a Signal)

Example

```
Procedure increment(a: inout word32;
                         by: in word32:=X”0000_0001”) is
   Variable Sum: word32 ;
   Variable Carry: Bit:= '0' ;
Begin
  For i in a'reverse_Range Loop
   Sum(i) := a(i) xor by(i) xor Carry;
   Carry  := (a(i) and by(i))  or (Carry and (a(i) xor
   by(i))) ;
  End Loop;
  a := Sum;
End Procedure increment;
```

CALL Examples

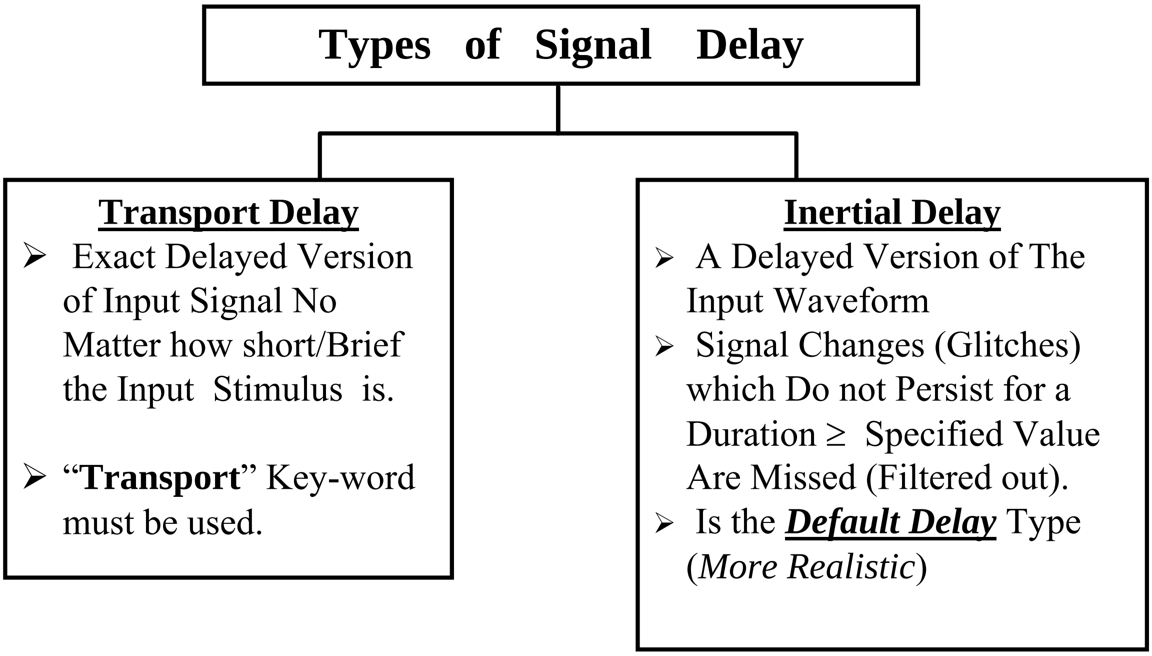
```
increment(count , X”0000_0004”); -- Increment by 4

increment(count); -- Increment by Default Value (1)

increment(count , by => open); -- Increment by Default
```

Good Practice

- Write Generic Subprograms that work for:
  - Any Array Size (***Size-Flexible***)
  - Any Index Range (***Range-Flexible***)

  - **Use Unconstrained Array Parameters**

Signal Assignment & Delay Types



Types of Signal Delay

<u>Transport Delay</u>

- Exact Delayed Version of Input Signal No Matter how short/Brief the Input Stimulus is.
- "**Transport**" Key-word must be used.

<u>Inertial Delay</u>

- A Delayed Version of The Input Waveform
- Signal Changes (Glitches) which Do not Persist for a Duration ≥ Specified Value Are Missed (Filtered out).
- Is the ***<u>Default Delay</u>*** Type (*More Realistic*)

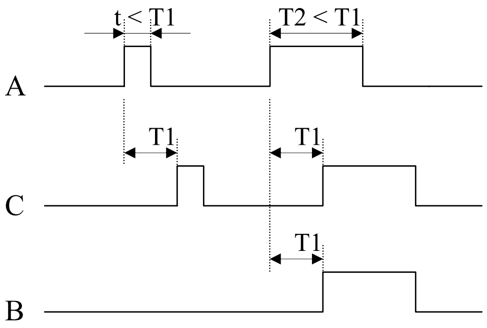
Signal Assignment & Delay Types

<u>Examples</u>

```
B <= A After T1;                -- Default Inertial Delay
C <= Transport A After T1;      -- Transport Delay
```

<u>Problem</u>

What if the period of the glitch to be filtered is smaller than the signal delay ?



- VHDL-93 provides additional Reject specification

<u>Example</u>: S<= REJECT 3 ns INERTIAL waveform after 5 ns; -- VHDL'93 only

…Transport & Inertial Delay…

```
Entity example Is
End example;
Architecture ex1 of example is
SIGNAL a, b, c, wave : BIT;
BEGIN
    a <= wave after 5 ns;
    b <= REJECT 2 ns INERTIAL   wave after 5 ns;
    c <= transport wave after 5 ns;
    wave  <= '1' after 5 ns, '0' after 8 ns, '1' after 15 ns,
              '0' after 17 ns, '1' after 25 ns;
END ex1;
```

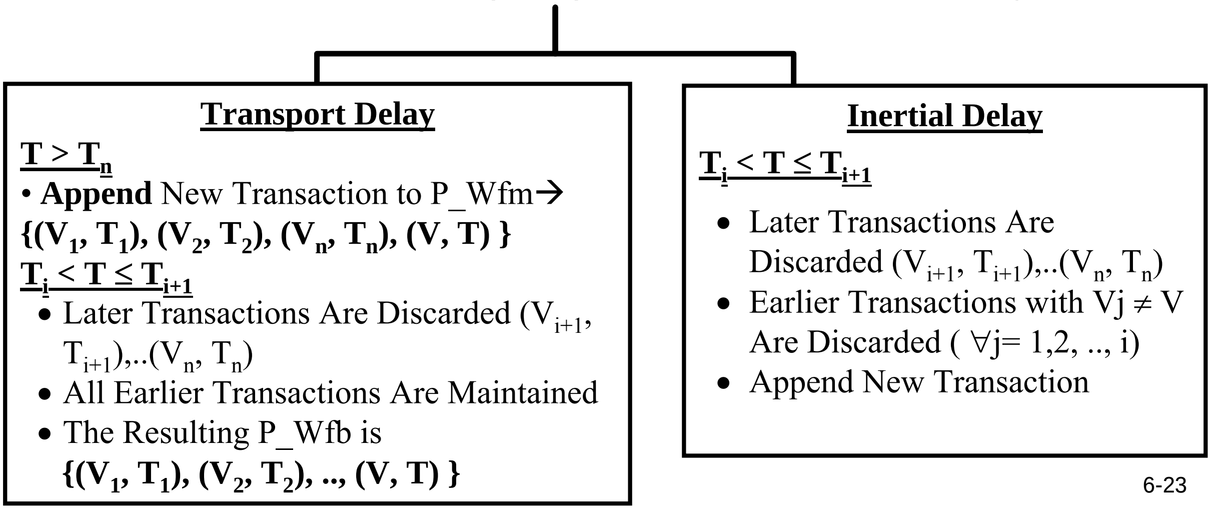
Sequential Placement of Transactions

- The Scheduling Mechanism Reflects The Difference in Nature Between *Inertial* & *Transport* Delays.
- Let the Projected Waveform of a Signal Consist of the Transactions:

(V1, T1), (V2, T2), …(Vi, Ti),…, (Vn, Tn)

such that Tn > Tn-1 > .. > Ti > .. T2 > T1

- IF a new Transaction be (V , T) is to Scheduled on this Signal, Then:



Transport Delay

$T > T_n$

• **Append** New Transaction to P_Wfm→

$\{(V_1, T_1), (V_2, T_2), (V_n, T_n), (V, T)\}$

$T_i < T \leq T_{i+1}$

- Later Transactions Are Discarded (V_{i+1}, T_{i+1}),..(V_n, T_n)
- All Earlier Transactions Are Maintained
- The Resulting P_Wfb is **$\{(V_1, T_1), (V_2, T_2), .., (V, T)\}$**

Inertial Delay

$T_i < T \leq T_{i+1}$

- Later Transactions Are Discarded (V_{i+1}, T_{i+1}),..(V_n, T_n)
- Earlier Transactions with Vj ≠ V Are Discarded (∀j= 1,2, .., i)
- Append New Transaction

Sequential Placement of Transactions

<u>Lemma</u>

- The Semantics of Inertial-Delay Signal Assignment is Such That if a Number of waveform elements are assigned to a signal as in:

  *S <= 1 After 1 NS***,** *3 After 3 NS***,** *5 After 5 NS***;**

- ***<u>Only the First Element is Considered to Have Inertial Delay, i.e. Elements After the First One Are Considered to Have Transport Delay</u>***
- <u>Note</u>: Wfm Elements Must Have Ascending Time Delays

Sequential Placement of Transactions

A <= 1 After 1 NS**,** 3 After 3 NS**,** 5 After 5 NS;

Is NOT Equivalent To:	**Is Equivalent To:**
A <= 1 After 1 NS;	A <= 1 After 1 NS;
A <= 3 After 3 NS;	A <= Transport 3 After 3 NS;
A <= 5 After 5 NS;	A <= Transport 5 After 5 NS;

Sequential Placement of Transactions

Process
Begin
A <= Transport 1 After 5 NS;
A <=Transport 2 After 10 NS;
Wait;
End Process;
	Process
	Begin
	B <= Transport 2 After 10 NS;
	B <= Transport 1 After 5 NS;
	Wait;
	End Process;

Line	**P_WFM(A)**	**Time**	**P_WFM(B)**
1	(1, 5NS)	t0	(2, 10 NS)
2	(1, 5NS) (2, 10 NS)	t1	(1, 5 NS) {*Previous Transaction Discarded*}

Line 1= After Executing the First Statement

Line 2= After Executing the Second Statement

Sequential Placement of Transactions

Process
Begin
A <= 1 After 5 NS;
A <= 2 After 10 NS;
Wait;
End Process;
	Process
	Begin
	B <= 2 After 10 NS;
	B <= 1 After 5 NS;
	Wait;
	End Process;

Line	**P_WFM(A)**	**Time**	**P_WFM(B)**
1	(1, 5NS)	t0	(2, 10 NS)
2	(2, 10 NS) {Discard Previous Transaction Since 1 ≠ 2}	t1	(1, 5 NS) {*Later Transaction Discarded*}

Line 1 = After Executing the First Statement

Line 2= After Executing the Second Statement

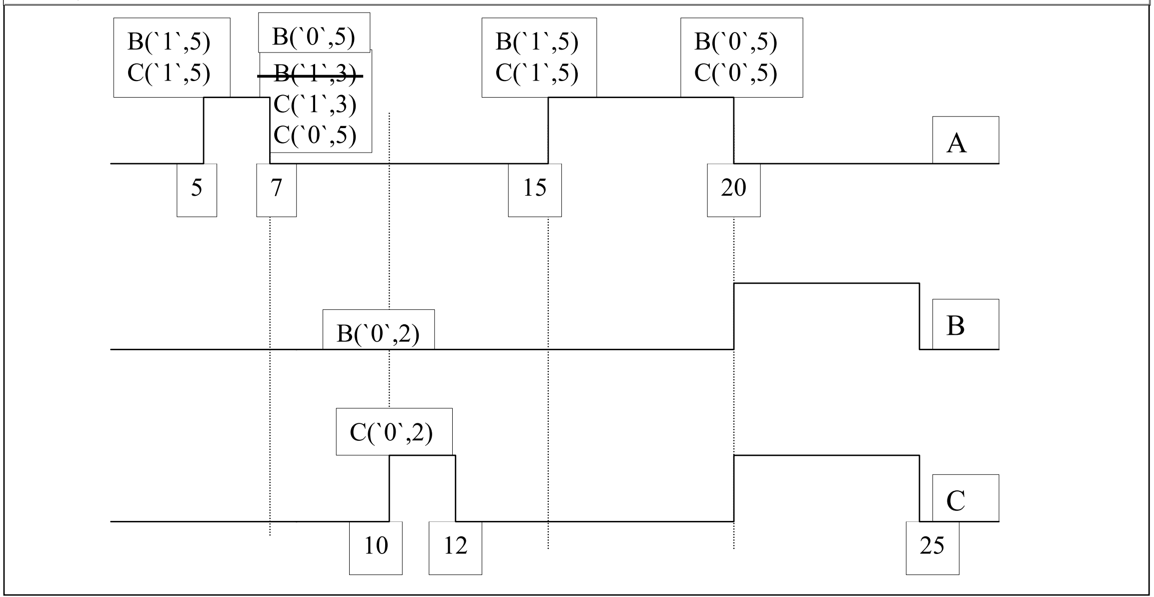
Sequential Placement of Transactions

```
Process
Begin
    S1:     A <= 1 After 1 NS, 3 After 3 NS, 5 After 5 NS;
    S2:     A <= 3 After 4 NS, 4 After 5 NS;
    Wait;
End Process;
```

Statement	P_WFM(A)
S1	(1, 1NS) (3, 3NS) (5, 5NS)
S2	Upon Adding (3, 4 NS)) **(Inertial)** • Discard (5, 5 NS) Since it is later than (3, 4 NS) • Discard (1, 1 NS) Since 1 ≠ 3 → (3, 3 NS) (3, 4 NS) Upon Adding (4, 5 NS) **(Transport)** Maintain All Earlier Transactions (3NS & 4NS) → (3, 3 NS) (3, 4 NS) (4, 5 NS)

Sequential Placement of Transactions

```
Architecture Ex OF Delay IS
Begin
   A <= `1` After 5 NS, `0` After 7 NS, `1` After 15 NS, `0` After 20 NS;
   B <= A After 5 NS;
   C <= Transport A After 5 NS;
End;
```



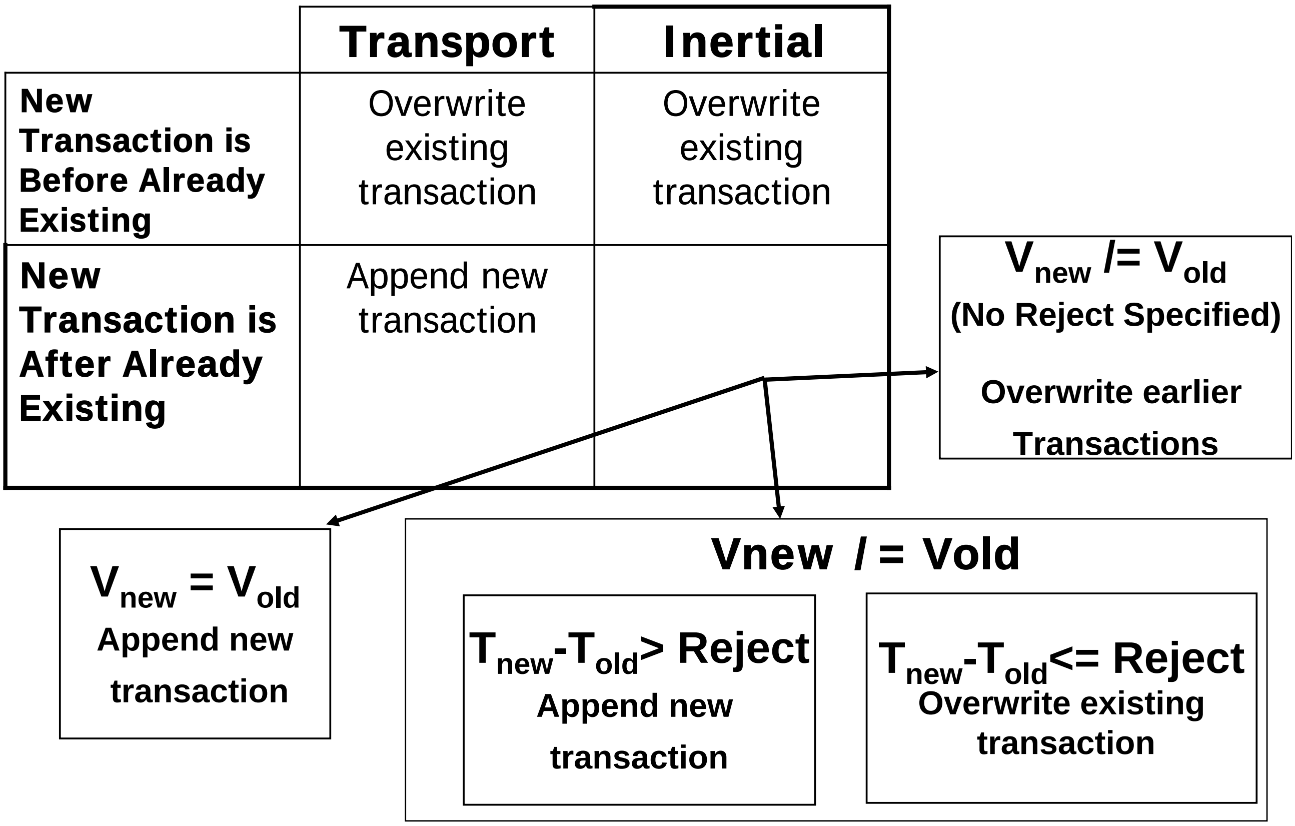
Sequential Placement of Transactions

	t=0	t=5 ns	t=7 ns	t=15 ns	t=20 ns
A (CV)	\` 0 \`	\` 1 \`	\` 0 \`	\` 1 \`	\` 0 \`
A(P_Wfm)	(\`1\`, 5 ns) (\`0\`, 7 ns) (\`1\`,15 ns) (\`0\`,20 ns)	(\`0\`, 2 ns) (\`1\`,10 ns) (\`0\`,15 ns)	(\`1\`,8 ns) (\`0\`,13ns)	(‘0\`,5 ns)	--

	t=0	t=5 ns	t=7 ns	t=12 ns	t=15 ns	t=20 ns
B(CV)	\` 0 \`	\` 0 \`	\` 0 \`	\` 0 \`	\` 0 \`	\` 1 \`
B(P_Wfm)	--	(\`1\`, 5ns)	(\`0 \`, 5 ns) ~~(\`1 \`,3 ns)~~	--	(\`1\`,5 ns)	(\`0\`,5ns)

	t=0	t=5 ns	t=7 ns	t=10 ns	t=12	t=15 ns	t=20 ns
C(CV)	\` 0 \`	\` 0 \`	\` 0 \`	\` 1 \`	\`0\`	\` 0 \`	\` 1 \`
C(P_Wfm)	--	(\`1\`,5ns)	(\`0\`,5ns) (\`1\`,3ns)	(\`0\`,2 ns)	--	(\`1\`,5 ns)	(\`0\`,5 ns)

Sequential Placement VHDL-93



	Transport	**Inertial**
New Transaction is Before Already Existing	Overwrite existing transaction	Overwrite existing transaction
New Transaction is After Already Existing	Append new transaction	

V_{new} /= V_{old}
(No Reject Specified)

Overwrite earlier Transactions

V_{new} = V_{old}
Append new transaction

Vnew / = Vold

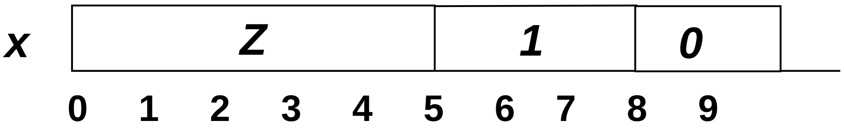
T_{new}-T_{old}> Reject
Append new transaction

T_{new}-T_{old}<= Reject
Overwrite existing transaction

Example: Inertial, New Transaction After Already Existing (Diff. Value with Reject)

```
ARCHITECTURE sequential OF appending IS
Type tit is (‘0’, ‘1’, ‘Z’);
SIGNAL x : tit := 'Z';
BEGIN
  PROCESS
  BEGIN
          x <= ‘1' AFTER 5 NS;
          x <= Reject 2 ns Inertial ‘0' AFTER 8 NS;
          WAIT;
  END PROCESS;
END sequential;
```

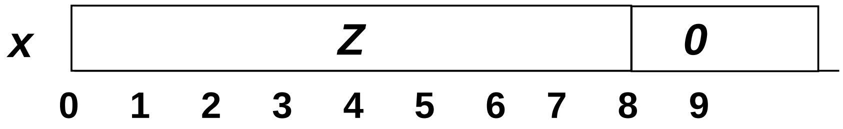
- *Time difference between new and existing transaction is greater than reject value*
- *Appends transaction*



Example: Inertial, New Transaction After Already Existing (Diff. Value with Reject)

```
ARCHITECTURE sequential OF  appending IS
Type tit is (‘0’, ‘1’, ‘Z’);
SIGNAL x : tit := 'Z';
BEGIN
  PROCESS
  BEGIN
          x <= ‘1' AFTER 5 NS;
          x <= Reject 4 ns Inertial ‘0' AFTER 8 NS;
          WAIT;
  END PROCESS;
END sequential;
```

- *Time difference between new and existing transaction is less than reject value*
- *Discards old value*



Signal Attributes…

Attribute	Example	Meaning	T/E	Kind	Type
`Event	S`Event	If S changes in the Current simulation cycle, S`EVENT **will** be **TRUE** for this cycle (δ time).	EV	Value	Boolean
`Active	S`Active	If a transaction is scheduled on S in the current simulation cycle, S`ACTIVE will be TRUE for this simulation cycle, i.e. for a δ-time,	TR	Value	Boolean
`Last_event	S`Last_event	Time elapsed since the last value change on S. If S`Event is TRUE, then S`Last_event = 0.	EV	Value	Time
`Last_active	S`Last_active	Time elapsed since the last transaction occurred on S. If S`Active is TRUE, then S`Last_active =0.	TR	Value	Time
`Last_value	S`Last_value	The value of S before its most recent event.	EV	Value	Same as S
`Delayed[(Time)]	S`Delayed(5 Ns)	A copy of S, delayed by 5 NS or by δ-time if no parameter or 0 parameter is specified →(Equivalent to *TRANSPORT* delay of S).	--	**Signal**	Same as S
`Stable[(Time)]	S`Stable(15 Ns)	A signal that is TRUE if S has not changed in the last 15 NS. If used with no parameter or 0, the resulting signal is TRUE if S has not changed in the current simulation time.	EV	**Signal**	Boolean
`Quiet[(Time)]	S`Quiet(5 Ns)	A signal that is TRUE if no transaction has been scheduled on S in the last 5 NS. If no parameter or 0, for current simulation cycle is assumed.	TR	**Signal**	Boolean
`Transaction	S`Transaction	A signal that toggles each time a transaction is scheduled on S.	TR	**Signal**	Bit

COE 405
VHDL Design Organization

Dr. Alaaeldin A. Amin
Computer Engineering Department

E-mail: amin@ccse.kfupm.edu.sa

http://www.ccse.kfupm.edu.sa/~amin

OUTLINE

- Concurrent vs. Sequential Constructs / Statements
- Concurrent Signal Assignments
- Sequential Statements
- Sequential Bodies
- Overloading
- Packages
- Libraries
- Process Statement
- Modeling FSMs

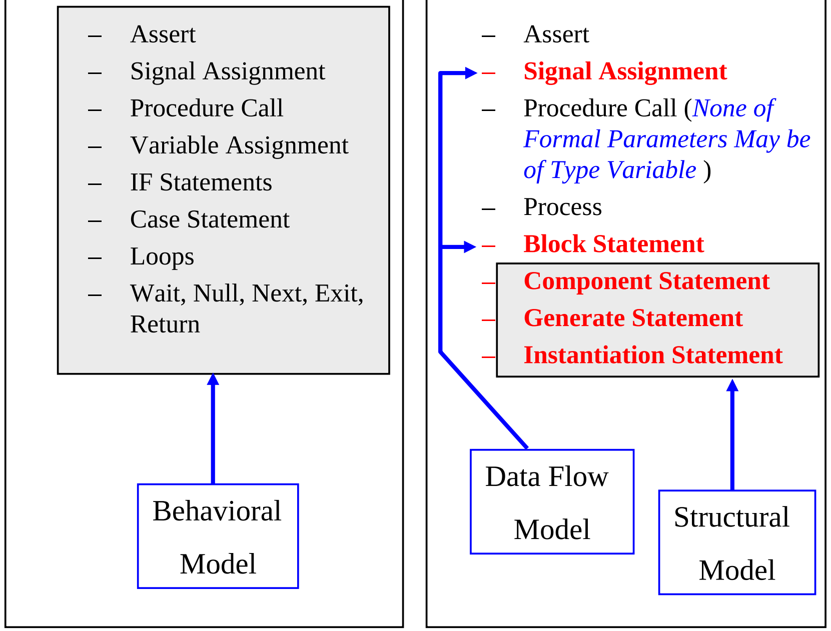
Concurrent Versus Sequential statements

Sequential Statements

- Used Within Process Bodies or SubPrograms
- Order Dependent
- Executed When Control is Transferred to the Sequential Body

Concurrent Statements

- Used Within Architectural Bodies or Blocks
- Order Independent
- Executed Once *At the Beginning of Simulation* or Upon Some Triggered Event



– Assert
– Signal Assignment
– Procedure Call
– Variable Assignment
– IF Statements
– Case Statement
– Loops
– Wait, Null, Next, Exit, Return

Behavioral Model

– Assert
– **Signal Assignment**
– Procedure Call (*None of Formal Parameters May be of Type Variable*)
– Process
– **Block Statement**
– **Component Statement**
– **Generate Statement**
– **Instantiation Statement**

Data Flow Model

Structural Model

Concurrent Signal Assignment

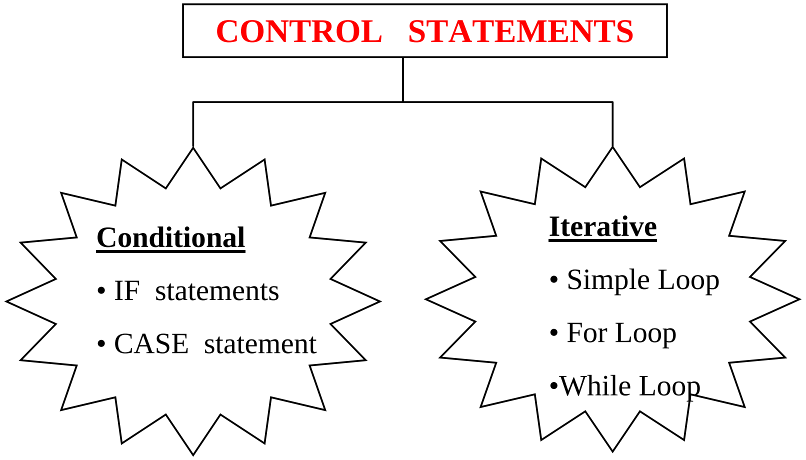
Syntax 1

Label : **target** <= [Guarded] [Transport]
Wave1 **when** *Cond1* **Else**
Wave2 **when** *Cond2* **Else**
…………………………….
Waven-1 **when** *Condn-1* **Else**
Waven **;**

Syntax 2

With *Expression* **Select**
target <= [Guarded] [Transport]
Wave1 **when** *Choice1* **,**
Wave2 **when** *Choice2* **,**
…………………………….
Waven-1 **when** *Choicen-1* **,**
Waven **when OTHERS;**

Sequential Statements



CONTROL STATEMENTS

Conditional

- IF statements
- CASE statement

Iterative

- Simple Loop
- For Loop
- While Loop

(I) Conditional control

a) IF Statements

Syntax: 3-Possible Forms

(i)

```
IF condition Then
     statements;
End IF;
```

(ii)

```
IF condition Then
     statements;
Else
     statements;
End IF;
```

(iii)

```
IF   condition Then
     statements;
Elsif condition Then
     statements;
Elsif condition Then
     statements;
          :
          :
Elsif condition Then
     statements;
End IF;
```

b) CASE Statement

Syntax:

(i)

```
CASE Expression is
  when value => statements;
  when value1 | value2| ...|valuen => statements;
  when discrete range of values => statements;
  when others => statements;
End CASE;
```

Notes:

- Values/Choices Should not Overlap (*Any value of the Expression should Evaluate to only one Arm of the Case statement*).
- All Possible Choices for the *Expression* Should Be Accounted For *Exactly Once*.
- If "**others**" is used, It must be the last "arm" of the CASE statement.
- There can be Any Number of Arms in Any Order (*Except for the* ***others*** *arm which should be Last*)

Example:

```
CASE x is
  when 1 => out :=0;
  when 2 | 3 => out :=1;
  when 4 to 7 => out :=2;
  when others => out :=3;
End CASE;
```

(2) LOOP control

a) Simple Loops

Syntax:

Optional

```
Loop_Label: LOOP
            statements;
        End LOOP Loop_Label;
```

Notes:

- The *Loop_Label* is Optional
- The **exit** statement may be used to exit the Loop. It has two possible Forms:

  1- **exit** *Loop_Label;* -- *This may be used in an if statement*

  2- **exit** *Loop_Label* **when** *condition;*

Example:

```
Process
    variable A:Integer :=0;
    variable B:Integer :=1;
Begin
   Loop1: LOOP
                 A := A + 1;
                 B := 20;
                 Loop2: LOOP
                           IF B < (A * A) Then
                               exit Loop2;
                           End IF;
                           B := B - A;
                        End LOOP Loop2;
                 exit Loop1 when  A > 10;
          End  LOOP Loop1;
End Process;
```

b) **For Loop**

Syntax:

Optional

Need Not Be Declared

```
Loop_Label: FOR Loop_Variable  in  range  LOOP
                  statements;
             End LOOP Loop_Label;
```

Example:

```
Process
    variable B:Integer :=1;
Begin
  Loop1: FOR  A  in  1  TO  10  LOOP
              B := 20;
              Loop2: LOOP
                        IF B < (A * A) Then
                            exit Loop2;
                        End IF;
                        B := B - A;
                     End LOOP Loop2;
       End  LOOP Loop1;
End Process;
```

c) **WHILE Loop**

Syntax:

Optional

```
Loop_Label: WHILE  Condition  LOOP
                  statements;
             End LOOP Loop_Label;
```

Example:

```
Process
    variable B:Integer :=1;
Begin
  Loop1: FOR A in 1 TO 10 LOOP
            B := 20;
            Loop2: WHILE B < (A * A) LOOP
                      B := B - A;
                   End LOOP Loop2;
         End LOOP Loop1;
End Process;
```

c) **Next Statement**

Syntax:

```
Next [Loop_Label][When Condition];
```

- Skip Current Loop Iteration When *Condition* is True
- If *Loop_Label* is Absent, innermost Loop iteration is Skipped When *Condition* is True
- IF *Condition* is Absent, Appropriate Loop Iteration is Skipped .

c) **Null Statement**

Syntax: **Null;**

- Does Nothing
- Useful in **CASE** Statements If No Action Is Required.

Subprograms

FUNCTIONS

Syntax :

Only ***Input*** Constants (*Default*) or Signals (No Input Variables)

```
FUNCTION function_Name(Input Parameter_List) RETURN
type IS
        {Function Declarative Part}
Begin
        Function Algorithm;
    RETURN Expression;
End function_Name;
```

Examples :

Default is Constant

```
FUNCTION maj3(Signal x, y, z :Bit ) RETURN Bit IS
        variable M : Bit;
Begin
    M := (x and y) or (x and z) or (z and y);
    RETURN M;
End maj3;
```

```
FUNCTION maj3(Signal x, y, z :Bit ) RETURN Bit IS
Begin
   RETURN (x and y) or (x and z) or (z and y);
End maj3;
```

Function Usage Notes

- The Only Allowed Mode For Function Parameters is "*IN*". *No **Out** or **INOUT** Parameters Are Allowed*.
- The Only Allowed Object Class for Parameters are **Constants** and **Signals**. If Not Specified, "Constant Is Assumed", (No Variables Parameters are Allowed)
- Since Only parameters of Mode "IN" Are Allowed, Functions Have No Side Effects.
- Parameters of mode "IN" Can only be Read but not Written into
- At least One Return Statement must be included
- Functions Can Be *Recursively* Defined

Subprograms

PROCEDURES

Syntax :

Both *Input& Output* Parameters Allowed

```
PEOCEDURE  Procedure_Name (Interface_List)   IS
           {Procedure Declarative Part}
Begin
           Procedure  Algorithm;
End Procedure_Name;
```

Example :

```
TYPE  Bit4  IS ('X', '0', '1', 'Z');
TYPE  Bit4_Vector  IS  array(Integer  range<>) of Bit4;
PROCEDURE Ones_N_Zeros_CNT (X : in Bit4_Vector;
N_Ones, N_Zeros : Out Integer) IS
           variable  N0, N1 : Integer :=0;
Begin
    FOR  i  in X'Range   LOOP
           IF  X(i) = '1'  THEN
                      N1 := N1 + 1;
           ElsIF  X(i) = '0'  THEN
                      N0 := N0 + 1;
           END IF;
    End Loop;
    N_Zeros := N0;
    N_Ones  := N1;
End Ones_N_Zeros_CNT ;
```

Procedures Usage Notes

- Allowed Modes For Procedure Parameters are "*In*", "*Out*", and "*InOut*".
- "IN" Parameters can only be Read, while "OUT" Parameters can only be Written Into
- Allowed Object Classes for Procedure Parameters are **Constants, Variables** and **Signals**. If Mode=In, the Default is **Constant**. If Mode=Out or InOut, the Default is **Variable**. Thus, *Signal Type Parameters Have to be Explicitly Declared*.
- A Signal Formal Parameter can be of Mode **in**, **out** or **inout**.
- Procedure Calls May Be Either Sequential or Concurrent. IF Concurrent, Only *Parameters of Type Constant or Signal May be Used* (*Variables are not Defined Within Concurrent Bodies*)
- Procedures May be Declared within Other Procedures
- Procedure Variables are Dynamic (Don't Maintain Their Values Between Calls)

Parameter Default Values

- Default Values May Be Specified for Parameters of **Mode *In* only**.
- The Parameter Must be either Constant or Variable (Not a Signal)

Example

```
Procedure increment(a: inout word32;
                        by: in word32:=X"0000_0001") is
   Variable Sum: word32 ;
   Variable Carry: Bit:= '0' ;
Begin
  For i in a'reverse_Range Loop
   Sum(i) := a(i) xor by(i) xor Carry;
   Carry  := (a(i) and by(i))  or (Carry and (a(i) xor
   by(i))) ;
  End Loop;
  a := Sum;
End Procedure increment;
```

CALL Examples

```
increment(count , X"0000_0004"); -- Increment by 4

increment(count); -- Increment by Default Value (1)

increment(count , by => open); -- Increment by Default
```

Good Practice

- Write Generic Subprograms that work for:
  - Any Array Size (***Size-Flexible***)
  - Any Index Range (***Range-Flexible***)

↓

- **Use Unconstrained Array Parameters**

OverLoading

Definition:

A Character Literal
An Identifier
A Procedure Name
A Function Name
An Operator Symbol

Can Be Defined To

Have *More Than One Meaning*.

Example :

- A Char Literal Can Be Defined as an Element in More than one Enumeration Data Type.

```
Type Tri_State IS  ('0', '1', 'Z');
Type MVL4 IS ('X, '0', '1', 'Z');
```

Thus '**0**' is *Overloaded* Being *Member of* {Bit, Tri_State, MVL4}

- VHDL Differentiates Between Overloaded Char Literals and Identifiers Based on Context.
- VHDL Differentiates Between Overloaded Subprogram Names Based on The Type and Number of Passed Parameters, and the Type of Returned Data Type in Case of Functions.

OverLoading

Example :

```
SubType Word32 IS Bit_Vector(31 DownTo 0);
Function Check_Bounds(Value: Integer) Return Boolean IS
Function Check_Bounds(Value: Word32) Return Boolean IS

Valid_int := Check_Bounds(4095);
Valid_Bin := Check_Bounds(X"000F_FFFF");
```

Type of Passed Parameters Determine Which Function is Used

- Meanings of *Predefined Operators* Can Be Further *Extended* To Cover Other Data Types not Covered By the Original Operator.
- A *Function* Whose *Name* is *a String* Representing the Operator is Defined for the New Operand Types.

Example Extend the "+" Operator to Add Two 32-Bit Binary Numbers.

```
Function "+" (a, b: Word32) Return Word32 IS
Begin
     Return(int2bin(Bin2Int(a) + Bin2Int(b)));
End "+";
```

Usage:

(a) **X"1F0E_1015" + X"3AF0_0C12"**
(b) **"+"(X"1F0E_1015" , X"3AF0_0C12");**

OverLoading

Example : OverLoad The AND Operator To Operate on Type MVL4 Operands

```
Function "AND" (L, R: MVL4) Return MVL4 IS
Type T_Table IS Array(MVL4, MVL4) OF MVL4;
Constant AND_Table : T_Table :=
-- -----------------------------------------------------------------
--              ('X', '0', '1', 'Z')
-- -----------------------------------------------------------------
            (('X', '0', 'X', 'X') ,   -- 'X'
             ('0',  '0', '0',  '0') ,   -- '0'
             ('X', '0', '1',  'X') ,   -- '1'
             ('X', '0', 'X', 'X'));   -- 'Z'
Begin
            Return (AND_Table(L, R));
End "AND" ;
```

Note : The overloaded AND Operator in the Above example is still *CASE-INSENSITIVE*, even though the operator name is placed between double quotes.

Example :

```
Variable v1, v2, v3: MVL4;
…………………………………….
v3 := “and”(v1, v2);
v3  := “AND”(v1, v2);
v3 := v1 and v2;
v3 := v1 AND v2
```

Packages

- **Packages** Group Frequently Used ***Declarations*** of Data Types, Subprograms, Constants, Signals, and Components.
- A Package Has a Name Consists of a ***Declaration****-Part* and a ***Body****-Part*. The Package Declaration Takes the Following General Form:

```
Package Package_Name IS
        Declarations;
End Package_Name;
```

- A Package **Body** Should Have the Same *Package_Name* as the Package Declaration Part. The Package Body Contains the *Subprogram Bodies* Whose Corresponding Declarations Appeared in the Package Declaration. The Package Body Takes the Following General Form:

```
Package Body Package_Name IS
        Subprogram_Bodies;
End Package_Name;
```

- Declarations Made within an Entity Are Visible only Within the Architectural Bodies of this Entity.
- Declarations Appearing Within an Architectural Body Are Visible Only Within this Body and are not visible to Other Architectural Bodies Even if they Describe the Same Design Entity.
- Declarations Within a Package Construct, However, Can Be made Visible To Any Number of Design Entities By *Preceding these Design Entities* by a **USE** Clause for this Package

Example: *Declaration Part of Package Sample*

```
Package  Sample IS
     File RSPNS: text open write_mode is "./Digits.out";
     Constant Dash30: String( 1 to  30):=(Others => '-');
     Constant Dash60: String( 1 to  60):=(Others => '-');
     Constant eqs: String(1 to 3) := " = ";
     Shared Variable ll: Line;
     Type  Tri_Level  IS  ('0', '1', 'Z');
     SubType  Bit32  IS  Bit_Vector (31 downto 0);
     Function Invert (X: Tri_Level) Return  Tri_Level;
     Procedure Bin2Int(Bin: in Bit_Vector; Int: out Integer);
End  Sample;
```

- The following USE Statement makes all these declarations Visible

```
USE  Work.Sample.ALL ;
```

```
Package BODY   Sample IS
--
Function Invert (X: Tri_Level) Return  Tri_Level IS
   Variable  y: Tri_Level;
Begin
    Case  X  IS
          when '0' => y:= '1';
          when '1' => y:= '0';
          when 'Z' => y:= 'Z';
   End Case;
   Return (y);
End Invert;
--
Procedure Bin2Int (Bin: in Bit_Vector; Int: out Integer) IS
   Variable  result: Integer:=0;
   Variable  Tmp: Integer:=1;
Begin
   For  i  in  Bin'Low  To  Bin' high Loop
        IF Bin(i) = '1' Then result := result +Tmp; End IF
        Tmp := 2 * Tmp;
   End  Loop;
   Int := result ;
End Bin2Int;

End  Sample;
```

Predefined Packages - Examples

Standard Package

- Defines primitive types, subtypes, and functions.
- e.g. **Type Boolean IS (false, true);**
- e.g. **Type Bit is ('0', '1');**

TEXTIO Package

- Defines types, procedures, and functions for standard text I/O from ASCII files.

Design Libraries…

- VHDL supports the use of design libraries for categorizing components or utilities.
- Applications of libraries include
  - Sharing of components between designers
  - Grouping components of standard logic families
  - Categorizing special-purpose utilities such as subprograms or types

Predefined libraries

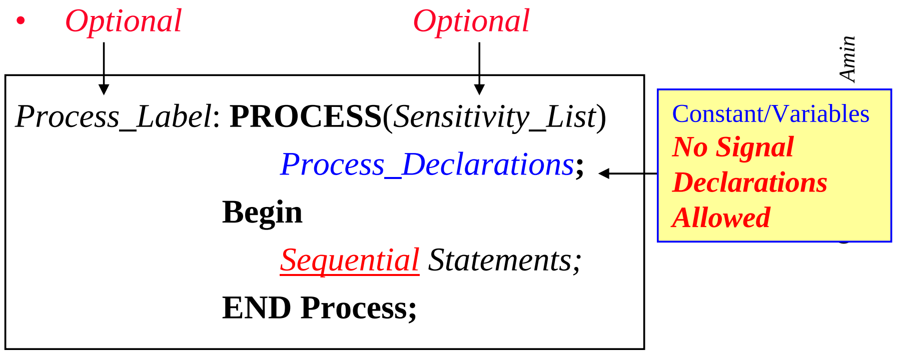
- **STD Library**
  - Contains the **STANDARD** and **TEXTIO** packages
  - Contains all the standard types & utilities
  - Visible to all designs
- **IEEE library**
  - Contains VHDL-related standards
  - Contains the std_logic_1164 (IEEE 1164.1) package
    * Defines a nine values logic system
- **WORK library**
  - Root library for the user

Visibility of Design Libraries & Packages

- To make a library visible to a design *entity*
  - **LIBRARY** *library_name*;
- To make a Package visible to a design entity
  - **Use** *library_name*.*Package_Name*.**ALL**;
- The following statement is assumed by all designs
  - **LIBRARY** WORK;
- To use the std_logic_1164 package
  - **LIBRARY** IEEE
  - **USE** IEEE.std_logic_1164.ALL

Process Statement

- Main Construct for Behavioral Modeling.
- Other Concurrent Statements Can Be Modeled By an Equivalent Process.
- Process Statement is a **Concurrent** Construct which Performs a Set of Consecutive (Sequential) Actions once it is Activated. Thus, Only Sequential Statements Are Allowed within the Process Body.



- *Optional* *Optional*

Process_Label: **PROCESS**(*Sensitivity_List*)
Process_Declarations;
Begin
Sequential Statements;
END Process;

Constant/Variables
No Signal Declarations Allowed

- Whenever a SIGNAL in the *Sensitivity_List* of the Process Changes, The Process is **Activated**.
- After Executing the Last Statement, the Process is **SUSPENDED** Until one (or more) Signal in the Process *Sensitivity_List* Changes Value where it will be **REACTIVATED**.

- A Process Statement ***Without*** a *Sensitivity_List* is ALWAYS ACTIVE, i.e. After the Last Statement is Executed, Execution returns to the First Statement and Continues (*Infinite Looping*).
- It is ILLEGAL to Use WAIT-Statement Inside a Process Which Has a *Sensitivity_List* .
- In case no *Sensitivity_List* exists, a Process may be activated or suspended Using the ***WAIT***-Statement :

Syntax :

- **WAIT;** -- *Process Suspended Indefinitely*
- **WAIT ON** *Signal_List*; -- *Waits for Events on one of the* -- *Signals in the List Equiv. To Process With Sensitivity_List.*
- **WAIT UNTIL** *Condition*; -- *Event Makes Condition True*
- **WAIT FOR** *Time_Out_Expression*;

Notes :

- When a **WAIT**-Statement is Executed, The process **Suspends** and Conditions for its **Reactivation** Are Set.
- Process Reactivation conditions may be Mixed as follows

WAIT ON *Signal_List* **UNTIL** *Condition* **FOR** *Time_Expression* **;**

- Process Reactivated IF:
  - Event Occurred on the *Signal_List* while the *Condition* is True, *OR*
  - *Wait* Period Exceeds "*Time_Expression* "

- ***UNLESS SUSPENDED***, **Process Execution**
  - **(1) Takes Zero Real Time (***Process Executed in one Simulation Cycle* → **Delta Time).**
  - **(2) Repeats Forever (***Infinite Loop***)**

Example:

```
Process
Begin
    A<= '1';
    B <= '0';
End Process;
```

- **Sequential Processing:**
  - **First A is Scheduled to Have a Value '1'**
  - **Second B is Scheduled to Have a Value '0'**
  - **A & B Get their New Values At the SAME TIME (1 Delta Time Later)**

Example:

```
Process
Begin
    A<= '1';
    IF (A= '1') Then Action1;
                 Else Action2;
End IF;
End Process;
```

- **Assuming a '0' Initial Value of A,**
  - **First A is Scheduled to Have a Value '1' One Delta Time Later**
  - **Thus, Upon Execution of IF_Statement, A Has a Value of '0' and *Action 2* will be Taken.**
  - **If A *was Declared as a Process* Variable, *Action1* Would Have Been Taken**

Examples : An Edge-Triggered D-FF

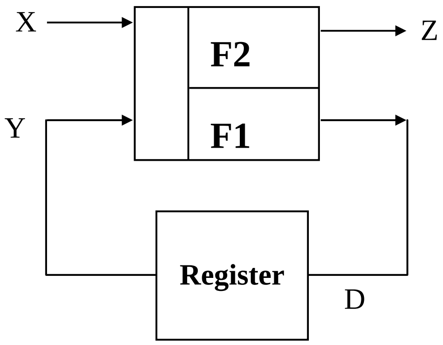
```
D_FF: PROCESS(CLK)
        Begin
            IF (CLK'Event and CLK = '1') Then
                Q <= D After TDelay;
            END IF;
        END Process;
```

```
D_FF: PROCESS           -- No Sensitivity_List
        Begin
            WAIT UNTIL  CLK = '1';
                Q <= D After TDelay;
        END Process;
```

```
D_FF: PROCESS(Clk, Clr)  -- FF With Asynchronous Clear
        Begin
            IF   Clr= '1'   Then
                Q <= '0' After TD0;
            ELSIF (CLK'Event and CLK = '1') Then
                Q <= D After TD1;
            END IF;
        END Process;
```

```
wait on  X,Y until (Z = 0) for 70 NS; -- Process Resumes
After 70 NS  OR (in Case X or Y Changes Value and Z=0 is
True) Whichever Occurs First
```

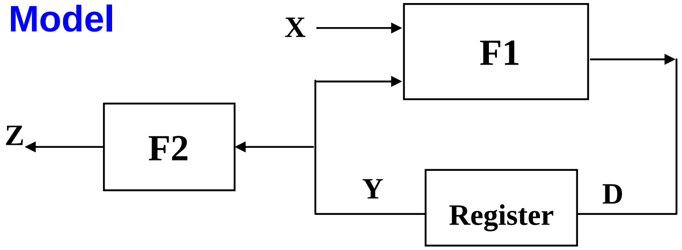
Generalized VHDL Mealy Model



```
Architecture  Mealy of  fsm  is
    Signal D, Y: Std_Logic_Vector( ...); -- Local Signals
Begin
   FFs: Process( Clk)
   Begin
      IF (Clk'EVENT and Clk = '1')  Then  Y <= D;
      End IF;
   End Process;

Transitions: Process(X, Y)
   Begin
      D <= F1(X, Y);
   End Process;

Output: Process(X, Y)
   Begin
      Z <= F2(X, Y);
   End Process;

End Mealy;
```

Can Be One Process

Generalized VHDL MOORE Model



```
Architecture  Moore of  fsm  is
      Signal D, Y: Std_Logic_Vector( ...); -- Local Signals
Begin
    FFs: Process( Clk)
    Begin
         IF (Clk'EVENT and Clk = '1')  Then  Y <= D;
         End IF;
    End Process;

Transitions: Process(X, Y)
    Begin
         D <= F1(X, Y);
    End Process;

Output: Process(Y)
    Begin
         Z <= F2(Y);
    End Process;

End Moore;
```

FSM example

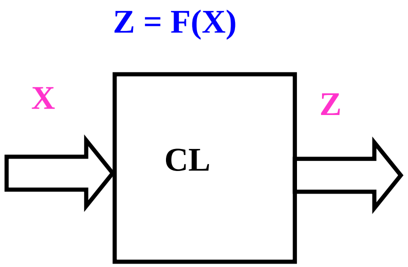
```
entity  fsm  is

 port ( Clk, Reset    : in   Std_Logic;
            X              : in  Std_Logic_Vector(0 to 1);
            Z              : out Std_Logic_Vector(1 downto 0));
end  fsm;
```

```
Architecture  behavior of fsm  is
   Type States is  (st0, st1, st2, st3);
   Signal Present_State ,  Next_State : States ;
Begin
   register: Process(Reset, Clk)
   Begin
      IF Reset = '1'  Then
            Present_State <= st0; --  Machine
                                   -- Reset to st0
      elsIF (Clk'EVENT and Clk = '1')  Then
            Present_State <= Next_state;
      End IF;
   End Process;
```

1st. Process

```
Transitions: Process(Present_State, X)
   Begin
      CASE Present_State is
            when  st0  =>
                        Z <= "00";
                        IF X = "11" Then Next_State <= st0;
                        else Next_State <= st1;
                        End IF;
             when st1  =>
                        Z <= "01";
                        IF X = "11" Then Next_State <= st0;
                        else Next_State <= st2;
                        End IF;
             when st2  =>
                        Z <= "10";
                        IF X = "11" Then Next_State <= st2;
                        else Next_State <= st3;
                        End IF;
             when st3  =>
                        Z <= "11";
                        IF X = "11" Then Next_State <= st3;
                        else Next_State <= st0;
                        End IF;
      End CASE;
 End Process;
End behavior;
```

Behvioral Modeling of Combinational Logic



Modeling Strategy

1. Put All Input Signals (**X**) in the Process Sensitivity List.

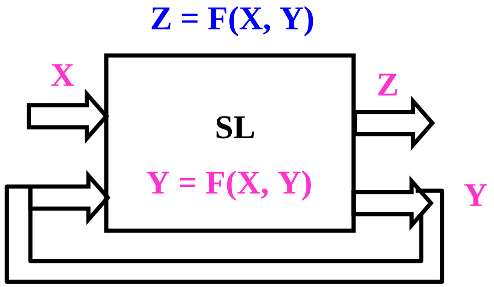
2. Define Local Process **Variables** **(Zvar)** Corresponding To the Output **Signals** **Z**

3. In the Process Body, Compute The Local Output **Variables** **(Zvar)** as Function of the Inputs **X**.

4. Circuit delay is modeled by assigning **ZVAR** to the Signals **Z** After Delay DEL.

```
Process(X)
            -- Declare Process Variables
            Variable ZVAR: Bit;
Begin
            -- Compute ZVAR=F(X)
            Z <= ZVAR  After Del;
End Process ;
```

Behvioral Modeling of Sequential Logic



Modeling Strategy

1. Put **Both** the Input Signals (**X**) **AND** the **Feedback** Signals **(Y)** in the Process Sensitivity List.

2. Define **Local** Process **Variables** **(Yvar and Zvar)** Corresponding To the **Feedback** **Signals** **(Y)** and Output **Signals** **Z** Respectively.

3. In the Process Body, Compute The Local **Variables** (**Yvar and Zvar)** as Function of **X and Yvar**.

4. Circuit delay is modeled by assigning **Yvar and Zvar** to the Signals **Y** and **Z** After the Corresponding Delays.

```
Process(X, Y)
            -- Declare Process Variables
            Variable Yvar, Zvar: Bit;
Begin
            -- Compute YVAR and Zvar=F(X, Y)
            Z <= Zvar  After Delz;
            Y <= Yvar  After Dely;
End Process ;
```

COE 405
VHDL Structural Modeling

Dr. Alaaeldin A. Amin
Computer Engineering Department

E-mail: amin@ccse.kfupm.edu.sa

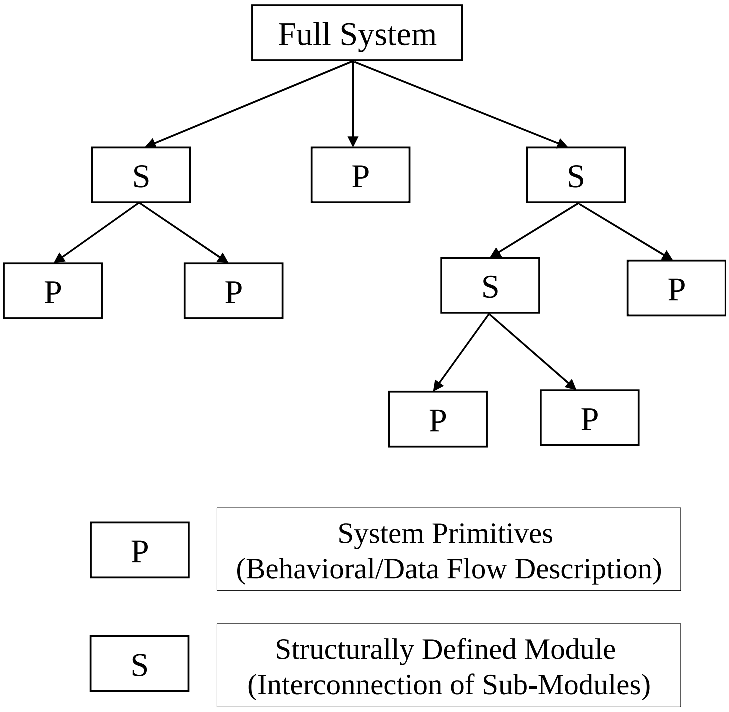
Home Page : http://www.ccse.kfupm.edu.sa/~amin

Outline

- Structural Models
- Example 4-Bit Comparator
- Elements of a Structural Model
  - Component Declarations
  - Instantiation Statements
  - Configuration
  - Binding of Component Instances
- Iterative Networks
  - 4-bit comparator
  - For…Generate Statement
  - IF…Generate Statement
- Binding Alternative
  - SR Latch Example
- Top Down Wiring
  - Sequential Comparator
  - Byte Latch
  - Byte Comparator

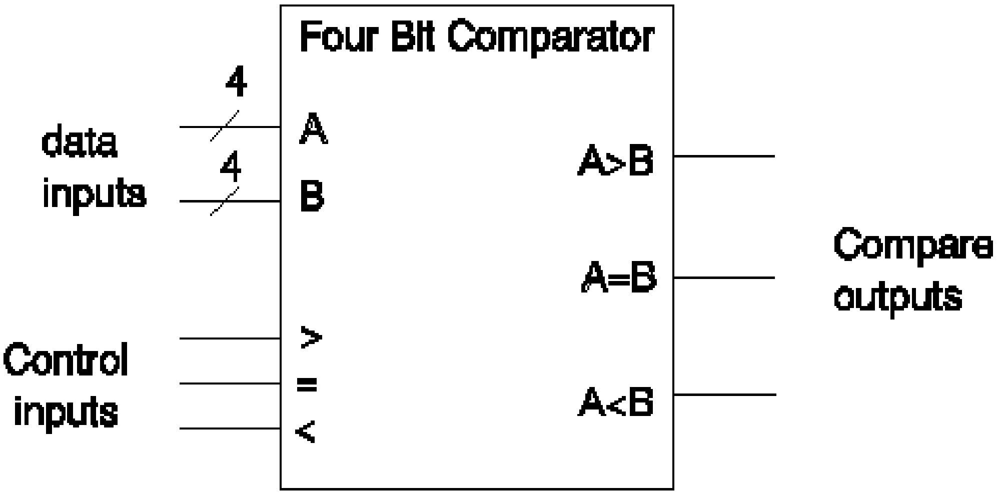
Structural Models

- Structural Model Starts Down-Up
- Structural Model → Set of Hierarchically Interconnected Modules.
- Leaf Cells → behavioral Description
- Wire gates into components
- *Iterative* VHDL constructs Wire components into larger designs
- Design is Top→ Down
- Implementation Bottom→ Up


Full System
S
P
S
P
P
S
P
P
P
P
System Primitives
(Behavioral/Data Flow Description)
S
Structurally Defined Module
(Interconnection of Sub-Modules)

Example 4-Bit Comparator

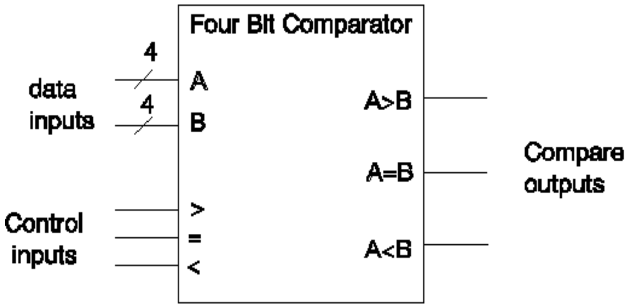
TOP LEVEL



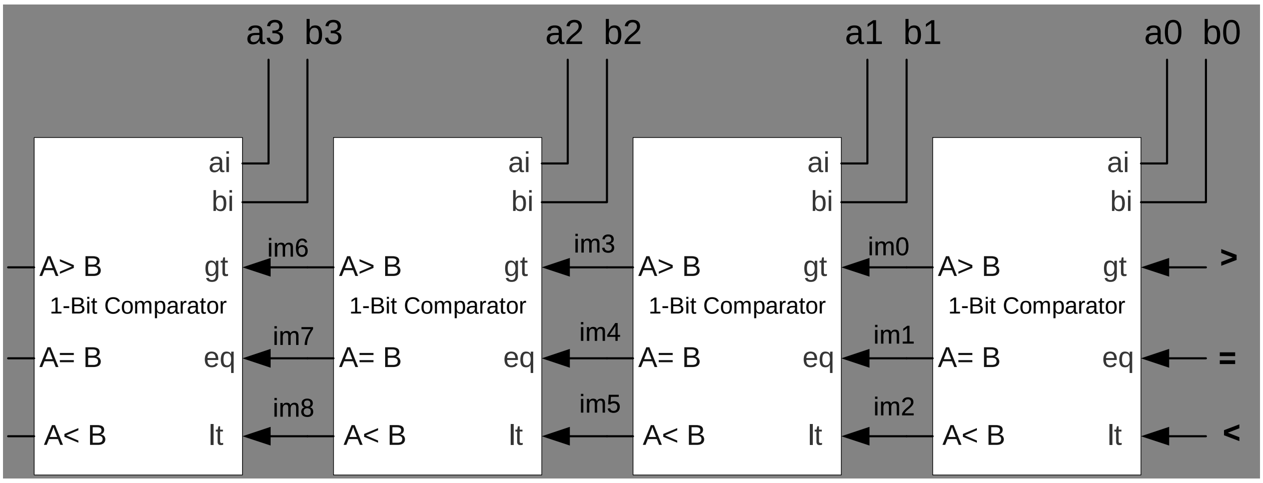
- Cascadable 4-bit comparator
- Top Level Input Control Signal Values
  - **>** = 0
  - **<** = 0
  - **=** = 1

4-Bit Comparator

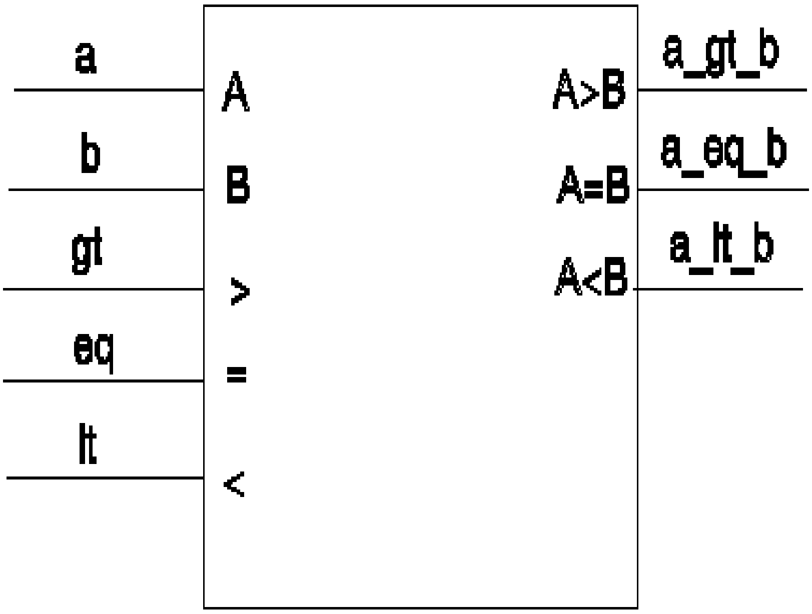
- Cascading single bit comparators into a 4- bit comparator
- Intermediate Signals Needed for Wiring lower level Components into higher level ones



1-LEVEL Down

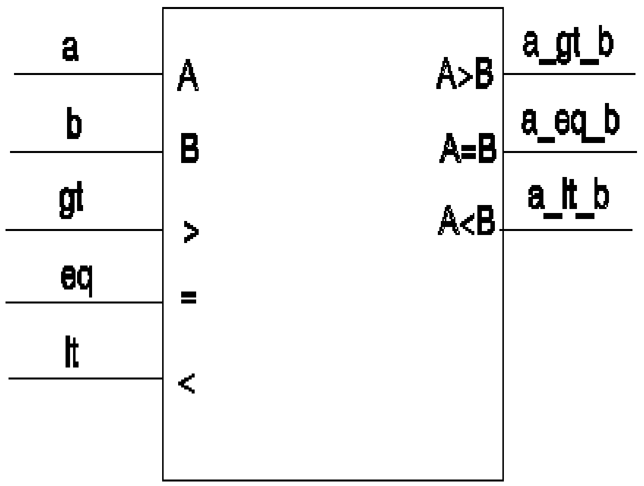


Cascadable Single-Bit Comparator



- When a > b the a_gt_b becomes 1
- When a < b the a_lt_b becomes 1
- If a = b outputs become the same as corresponding inputs

Cascadable Single-Bit Comparator



> \ a,b	00	01	11	10
0				1
1	1		1	1

a>b

= \ a,b	00	01	11	10
0				
1	1		1	

a=b

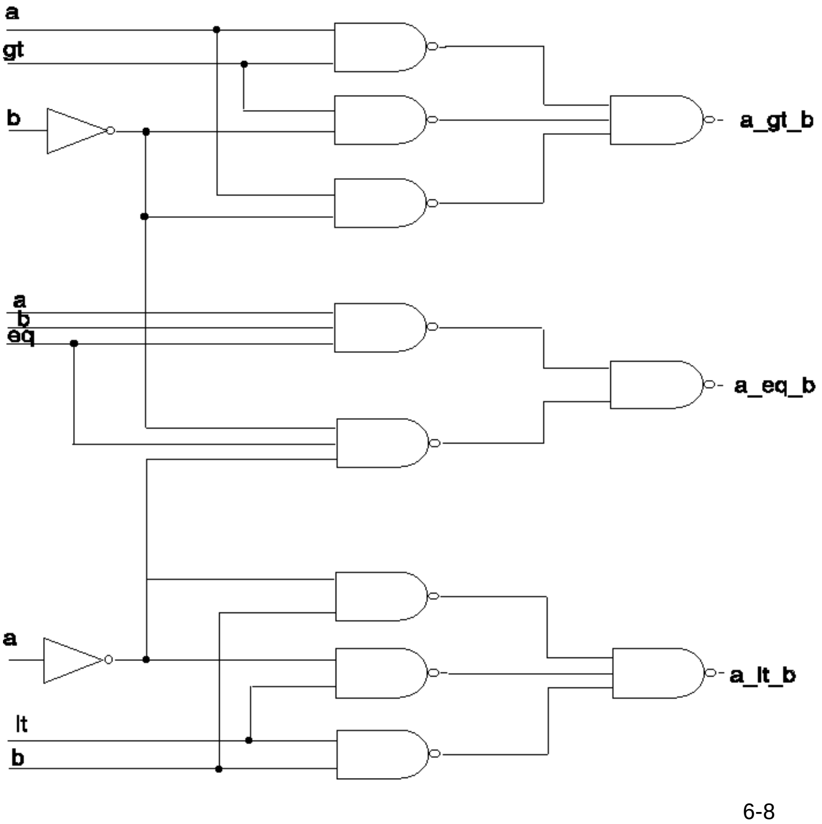
< \ a,b	00	01	11	10
0		1		
1	1	1	1	

a<b

Structural Single-Bit Comparator

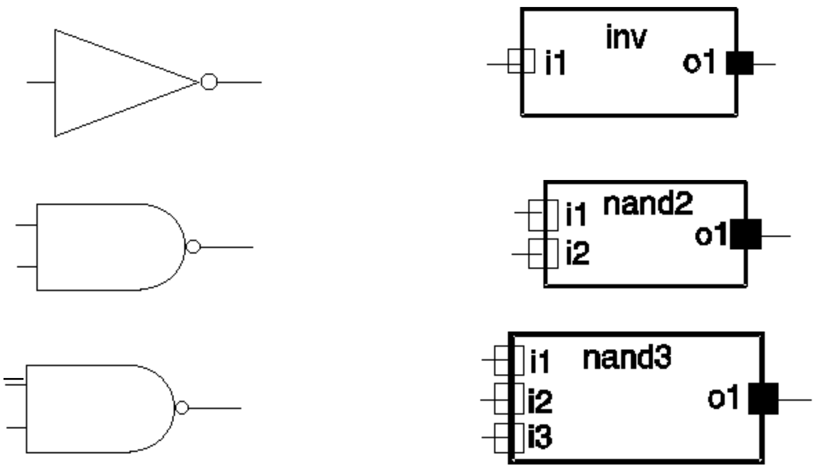
1-LEVEL Down

- Design uses basic gates
- The **a_lt_b** and **a_gt_b** outputs use the same logic

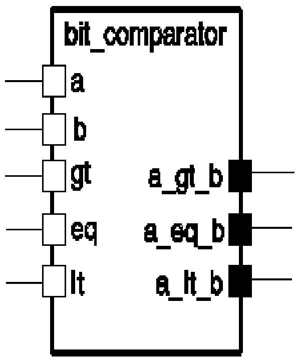


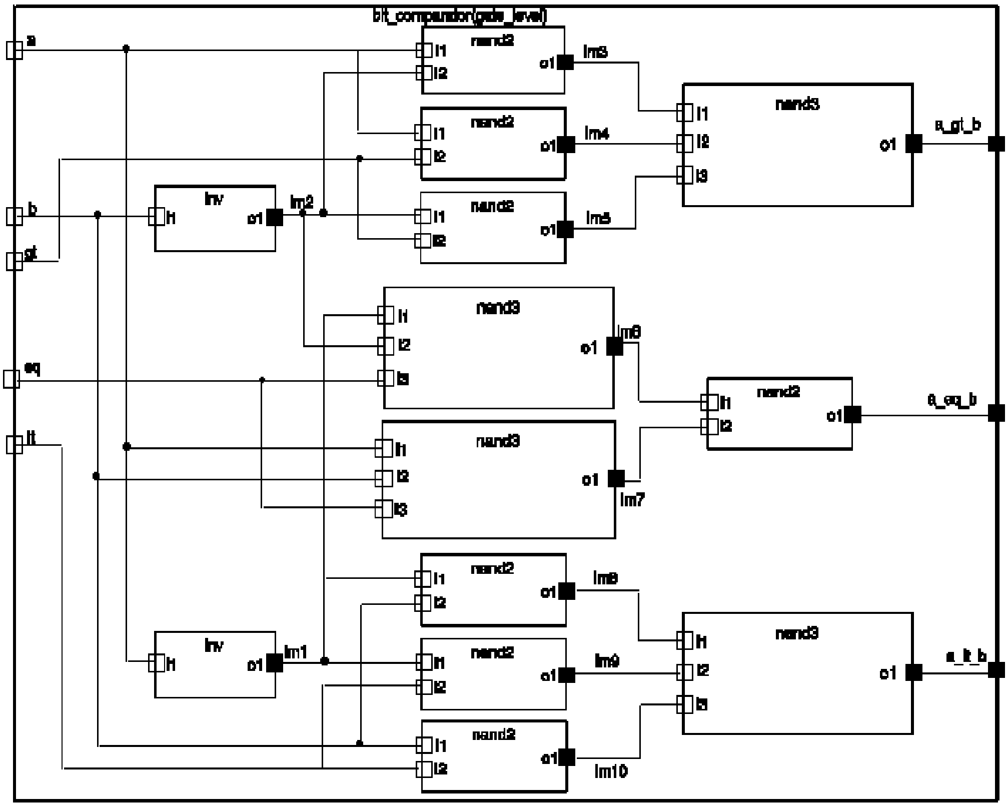
Basic Components & Graphical Notation

- Three basic components are used
- Will use as the basic components of several designs
- A graphical notation helps clarify wiring



Structural Single-Bit Comparator





Structural Model of Single-Bit Comparator

```
ENTITY bit_comparator IS
    PORT (a, b, gt,  eq,  lt : IN BIT;  a_gt_b,  a_eq_b, a_lt_b : OUT BIT);
END bit_comparator;
-----------------------------
ARCHITECTURE gate_level OF bit_comparator IS
--
COMPONENT n1 PORT (i1: IN BIT; o1: OUT BIT); END COMPONENT ;
COMPONENT n2 PORT (i1,i2: IN BIT; o1:OUT BIT); END COMPONENT;
COMPONENT n3 PORT (i1, i2, i3: IN BIT; o1: OUT BIT); END OMPONENT;

--  Component Configuration

FOR ALL : n1 USE  ENTITY WORK.inv (single_delay);
FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
FOR ALL : n3 USE ENTITY WORK.nand3 (single_delay);

--Intermediate signals
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
```

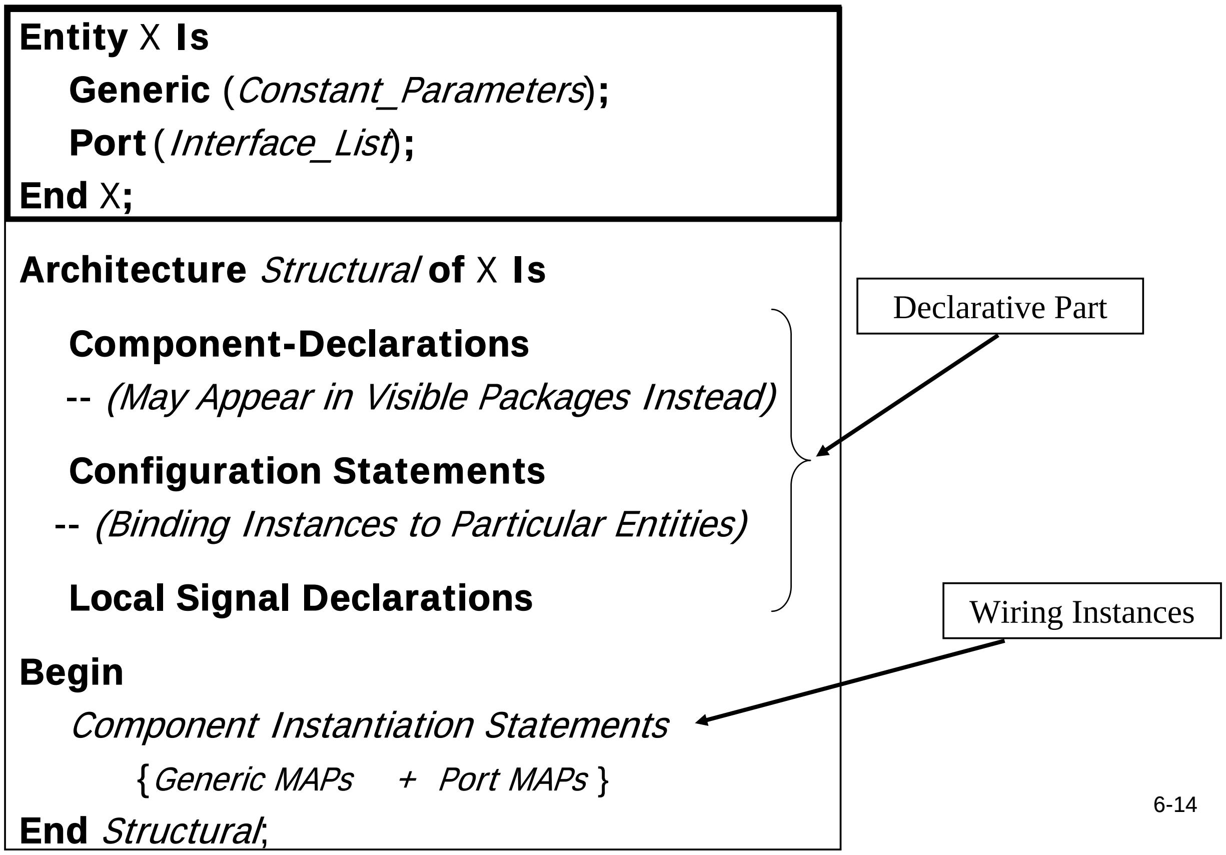
Netlist Description of Single-Bit Comparator

```
BEGIN
    g0 : n1 PORT MAP (a, im1); -- Generates a_Bar
    g1 : n1 PORT MAP (b, im2); -- Generates b_Bar
-- a_gt_b output
    g2 : n2 PORT MAP (a, im2, im3);
    g3 : n2 PORT MAP (a, gt, im4);
    g4 : n2 PORT MAP (im2, gt, im5);
    g5 : n3 PORT MAP (im3, im4, im5, a_gt_b);
--  a_eq_b output
    g6 : n3 PORT MAP (im1, im2, eq, im6);
    g7 : n3 PORT MAP (a, b, eq, im7);
    g8 : n2 PORT MAP (im6, im7, a_eq_b);
-- a_lt_b output
    g9 : n2 PORT MAP (im1, b, im8);
    g10 : n2 PORT MAP (im1, lt, im9);
    g11 : n2 PORT MAP (b, lt, im10);
    g12 : n3 PORT MAP (im8, im9, im10, a_lt_b);
END gate_level;
```

Model without Explicit Configuration Statement

```
ARCHITECTURE netlist OF bit_comparator IS
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
BEGIN
-- a_gt_b output
    g0 : ENTITY Work.inv(single_delay)  PORT MAP (a, im1);
    g1 : ENTITY Work.inv(single_delay) PORT MAP (b, im2);
    g2 : ENTITY Work.nand2(single_delay) PORT MAP (a, im2, im3);
    g3 : ENTITY Work.nand2(single_delay) PORT MAP (a, gt, im4);
    g4 : ENTITY Work.nand2(single_delay) PORT MAP (im2, gt, im5);
    g5 : ENTITY Work.nand3(single_delay) PORT MAP (im3, im4, im5,
    a_gt_b);
--  a_eq_b output
    g6 : ENTITY Work.nand3(single_delay) PORT MAP (im1, im2, eq, im6);
    g7 : ENTITY Work.nand3(single_delay) PORT MAP (a, b, eq, im7);
    g8 : ENTITY Work.nand2(single_delay) PORT MAP (im6, im7, a_eq_b);
-- a_lt_b output
    g9 : ENTITY Work.nand2(single_delay) PORT MAP (im1, b, im8);
    g10 : ENTITY Work.nand2(single_delay) PORT MAP (im1, lt, im9);
    g11 : ENTITY Work.nand2(single_delay) PORT MAP (b, lt, im10);
    g12 : ENTITY Work.nand3(single_delay) PORT MAP (im8, im9, im10,
    a_lt_b);
END netlist;
```

General Model for Structural Architectural Bodies



Entity X **Is**

Generic (*Constant_Parameters*)**;**

Port (*Interface_List*)**;**

End X**;**

Architecture *Structural* **of** X **Is**

Component-Declarations

-- *(May Appear in Visible Packages Instead)*

Configuration Statements

-- *(Binding Instances to Particular Entities)*

Local Signal Declarations

Begin

Component Instantiation Statements

{*Generic MAPs* + *Port MAPs* }

End *Structural*;

Declarative Part

Wiring Instances

Component Declarations

General Syntax

```
Component Comp_Name [is]
          Generic (Constant_Parameters );
          Port ( Interface_List);
End Component;
```

Difference Bet Component & Entity Declarations

1- Component Declaration May Appear In a Package, while Entity Declarations May Not.

2. Entity Declaration Declares Something that Really Exists, while Component Declarations only Declare Templates That Have No Physical Existence.

Component Declarations

Ports & Interface Lists

Each Port Declaration Includes

1- Port Object ***Name*** (*Only Signal Objects Can Be Ports*).

2. Port (Signal) Direction of Flow (***Mode***).

3. The ***Type*** of the Port Signal

Four Possible Signal Modes

1. **IN** Mode -- This is The Default Mode
2. **OUT** Mode
3. **INOUT** Mode
4. **Buffer** Mode

- *Both **inout** & **Buffer** Modes can be read & written into*
- ***Buffer** may never have more than 1 signal driver even if it is resolved type.*
- ***Buffer** ports can only be connected to another buffer port or a signal with a single driver*

Component Declarations (Port Modes)

An in port

- **can be read but not updated within the module, carrying information into the module. (An in port cannot appear on the left hand side of a signal assignment.)**

An out port

- **can be updated but not read within the module, i.e. it carries information out of the module. (*An out port cannot appear on the right hand side of a signal assigment.*)**

A buffer port

- **likewise carries information out of a module, but can be both updated & read (*used as input*) within the module. Can only be connected to a signal of buffer mode.**

An inout port

- **is bidirectional and can be both read and updated, with multiple update sources possible.**

Component Declarations

<u>Generics& Interface Lists</u>

- **May Appear in Design *Entities* & *Component* Declarations.**
- **Generics Provide Means for Parameterization of:**
  - **Timing**
  - **Range of SubTypes**
  - **The Number of Instantiated Components, e.g. n-Bit Adder**
  - **Array Sizes**
  - **Documenting Physical Quantities, e.g. Temperature**

Component Declarations

Example

```
Component Decoder Is
  Generic (N: Positive);
  Port (En: bit;  Sel:  bit_vector(N-
    1 DownTo 0);
    Dout:  Out  bit_vector(N-1
    DownTo 0));
End Component ;
```

Default Mode = **IN**

Example

```
Component ALU Is
  Generic (Size: Positive:8) ;
  Port(Sel: bit_vector(3 downto 0);
  RA, RB: bit_vector(Size-1
    DownTo 0) ;
  C, M :   Bit;  Cout: Out Bit ;
  Result: Out Bit_Vector(Size-1
    DownTo 0));
End Component ;
```

Component Instantiations

Required

1. Named Association, **or**
2. Positional Association

Syntax

Label **:** *Compt_Name*

Generic Map *Association_List*

Port Map *Association_List* **;**

- Objects Associated with Instantiated Ports Must Be of Type **SIGNAL** (*an Expression, Variable or Constant Types Are Not Allowed*)
- If a Generic Map is Not Specified, Generic *Default Values* are Used).

Component Declarations

Generics& Interface Lists

- Component **Generic**-Specified Values Overwrite Entity Default Values
- Association with **OPEN** Causes Default Values to be Used → Generic Map can specify only some of the parameters
- Using **OPEN** causes use of Entity default values
- Exclusion of **Generic Map** Leaves All Parameters **OPEN**

Example

Generic(tplh1, tplh2, tplh3, tphl1, tphl2, tphl3 : Time := 3 ns);

Generic Map (**Open**, **Open**, 8 NS, **Open**, **Open**, 10 NS)

Alternatively

Generic Map (tplh3 => 8 NS, tphl3 => 10 NS)

Component Declaration & Instantiation Example

Example

```
Package xxx  IS
    Type .....  -- Type Declarations
    Constant .....  -- Constant Declarations
    Component N2      -- Component  Declarations
          Port (I1, I2 : Bit ; O1 :out Bit);
    End Component;
End Package;
```

Component Instantiation Statement

Example

```
Use Work.xxx.ALL ;       -- Makes Package xxx visible
Entity y IS  .....  End y;
Architecture Struc of  y IS
    Signal S1, S2, S3, S4 : Bit; -- Declares local interconnect wires
    For G1 : N2  Use  Work.AND2(DF) ; -- Configuration Binding # 1
    For G2 : N2  Use  Work.OR2(Behavior) ; -- Configuration Binding # 2
Begin
    G1: N2  Port Map ( I1 => S1 , I2 => S2, O1 =>S3 ) ; -- Instance # 1
    G2: N2  Port Map (S3 , S1, S4) ; -- Instance # 2
End Struc;
```

Positional Association

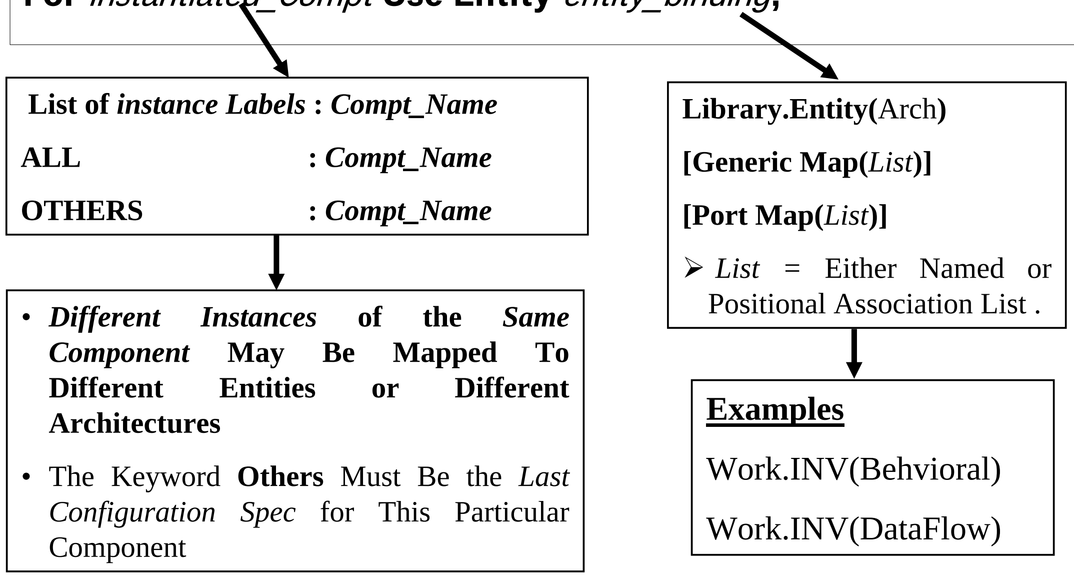
Named Association

… Configuration Specification

- Specifies for ***Each*** *Component Instance* the Actual *Entity* it Represents
  - Which Entity Declaration
  - In Which Library
  - Which Architecture to Use for this Entity

Syntax

For *instantiated_Compt* **Use Entity** *entity_binding***;**



List of *instance Labels* : *Compt_Name*

ALL : *Compt_Name*

OTHERS : *Compt_Name*

- ***Different Instances* of the *Same Component* May Be Mapped To Different Entities or Different Architectures**
- The Keyword **Others** Must Be the *Last Configuration Spec* for This Particular Component

Library.Entity(Arch**)**

[Generic Map(*List***)]**

[Port Map(*List***)]**

- *List* = Either Named or Positional Association List .

Examples

Work.INV(Behvioral)

Work.INV(DataFlow)

Configuration Specification …

Example

```
For g1, g3, g9 : Inverter
                 Use Entity Work.Inv(DataFlow)  ;
For Others      : Inverter - - All Other Inverter Instances
                 Use Entity Work.Inv(Delayed)  ; - - Use Diff Arch
```

Example

```
For ALL : Inverter
        Use Entity Work.Inv(DataFlow)  ;
```

Example

```
Entity Inverter IS
   Port (I: in Bit; O: out Bit);
End Entity;
   …………………………………….
Component Inv is
   Port (IN1 : in Bit; Out1: out Bit);
End Component
   …………………………………….
For ALL : Inv      Use Entity Work.Inverter(DF)
                   Port Map (I => IN1 , O => Out1)  ;
```

Binding of Component Instances

A Component Instance May be Bound to A Particular Entity Architecture in Several Ways:

1- Default Binding -- (*Hard Binding*

2. Configuration Specification -- (*Firm Binding*)

3. Configuration Declaration. -- (*Soft Binding*)

Default Binding

- **Component Instance** is Bound to an **Entity** which :

  1. Has the Same NAME as the Component
  2. Has the Same Number of Ports as the Component
  3. ***Ports*** Must Have The ***Same*** *Name* ***,*** *Type* and be of *Compatible modes*

- If the entity has more than one Architecture , the *Most Recently Analyzed Architecture* Will Be Used

Port Compatibility

Formal → Actual

(entity) → (Instance)

Actual`s Mode (Instance Ports)

Formal`s Mode Entity Formal Ports

	IN	OUT	INOUT	BUFFER
IN	X		X	
OUT		X	X	
INOUT			X	
BUFFER				X

… Port Map Association

Association is done in two steps:

1. **Instance-To-Component**
   (*Actuals* → *Component Local Ports*)
2. **Componenet-To-Entity** (**Configuration Port Map**)
   (*Component Local Ports* → *Entity Formal Ports*)

- **PORT MAP** in **Instantiation Statements** defines actual signal names corresponding to Component Port Names

actual

- Specifying **PORT MAP** in **Configuration** Statements is Optional

actual

- To Override Entity Local Port Names by Component Port Names, USE **PORT MAP** in **Configuration** Statement

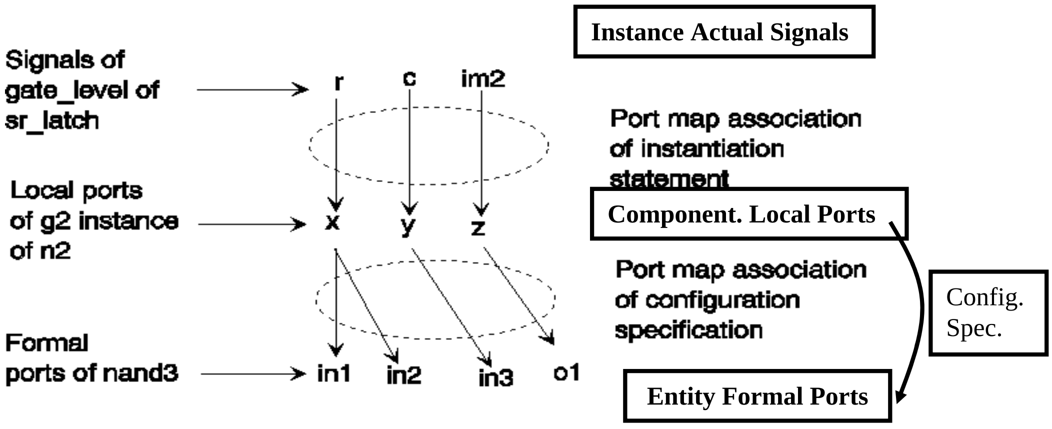
actual

- IF No **Port Map** is Specified in Configuration Statement, Local Port Names of COMPONENT declaration are the Default and they must be the Same as the Formals of the Design Entity.

actual

- A Locally Declared **Signal** Can Be Associated with A Formal Port of **Any Mode** As Long As It Has The **Same Type**

Port Map Association …



Instance Actual Signals
Signals of gate_level of sr_latch
r
c
im2
Port map association of instantiation statement
Local ports of g2 instance of n2
x
y
z
Component. Local Ports
Port map association of configuration specification
Config. Spec.
Formal ports of nand3
in1
in2
in3
o1
Entity Formal Ports

Iterative Logic… The Generate Statement

- ❑ Allows Generation of Regular Structure through Repeated Instantiation of Components in Some Regular Pattern.
- ❑ Useful for Modeling *Size-Flexible* (Definable) Regular Structures when used with *Generic Parameters*

Syntax

Required

Label : **FOR/IF** *Generation Scheme* **GENERATE**
Concurrent Statements; -- Typically Component
-- Instantiation Statements
End Generate *Label* ;

Optional

For Generatation Scheme → (*Repetitive Generation*)

For *identifier* **IN** *Discrete Range*

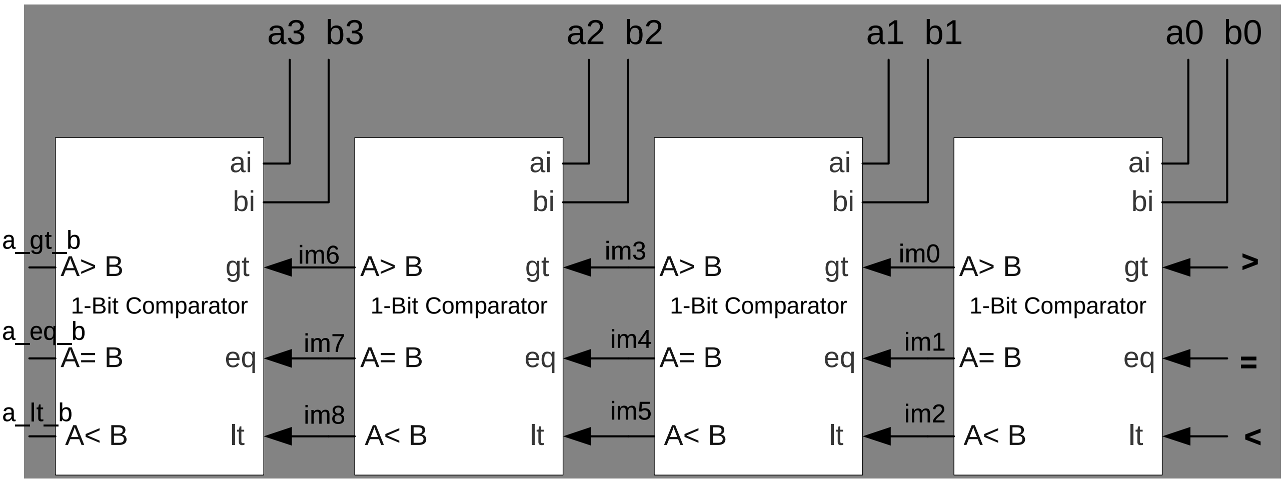
- **Must Iterate Over ALL Values of the** *Discrete Range*
- Unlike Sequential Loop Statement, **No EXIT** or **NEXT Statements Are Allowed.**

IF Generatation Scheme → (*Conditional Generation*)

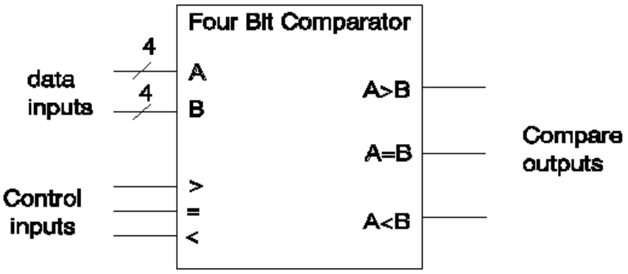
IF *Boolean condition*

- Unlike Sequential If Statement, IF-Generate Cannot Have **ELSE** or **ELSIF**
- Used Where Some Irregularities Exist at the terminal Points of the Regular Structure

4-Bit Comparator



- Cascading single bit into a four bit comparator
- Port Signals are visible in the ARCHITECTURE



4-Bit Comparator: “For ……. Generate” Statement

```
ENTITY nibble_comparator IS
   PORT (a, b : IN BIT_VECTOR (3 DOWNTO 0); -- a and b data inputs
           gt, eq,  lt : IN BIT;  -- previous greater, equal & less than
           a_gt_b, a_eq_b, a_lt_b : OUT BIT);  -- a > b,  a = b, a < b
END nibble_comparator;
   --
ARCHITECTURE iterative OF nibble_comparator IS

   COMPONENT comp1 [IS] -- “IS” May Be Used in VHDL-93
     PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
   END COMPONENT;
   FOR ALL : comp1 USE ENTITY WORK.bit_comparator (gate_level);
   SIGNAL im : BIT_VECTOR ( 0 TO 8);
BEGIN
  -- First Instance of Single bit comparator
   c0: comp1 PORT MAP (a(0), b(0), gt, eq, lt, im(0), im(1), im(2));
```

4-Bit Comparator: “For ……. Generate” Statement

```
   c1to2: FOR i IN 1 TO 2
             GENERATE
                c: comp1 PORT MAP ( a(i), b(i), im(i*3-3),
                                   im(i*3-2), im(i*3-1), im(i*3+0),
                                        im(i*3+1), im(i*3+2) );
             END GENERATE;
 -- Last Instance of Single bit comparator
   c3: comp1 PORT MAP (a(3), b(3), im(6), im(7),
         im(8), a_gt_b, a_eq_b, a_lt_b);
END iterative;
```

- BIT_VECTOR for Ports a & b
- Separate first and last bit-slices
- Arrays for intermediate signals facilitate iterative wiring
- Can easily expand to an n-bit comparator

- **Generate statement is a concurrent statement**
- **Generate statement brackets concurrent statements**

4-Bit Comparator: “IF . Generate” Statement

```
ARCHITECTURE iterative OF nibble_comparator IS
   --
   COMPONENT comp1  [IS] – “IS” May Be Used in VHDL-93
        PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
   END COMPONENT;
   --
   FOR ALL : comp1 USE ENTITY WORK.bit_comparator (gate_level);
   CONSTANT n : INTEGER := 4;
   SIGNAL im :  BIT_VECTOR ( 0  TO  (n-1)*3-1);
   --
BEGIN
   c_all : FOR i IN 0 TO n-1 GENERATE

              L : IF i = 0 GENERATE
                        least: comp1 PORT MAP (a(i), b(i), gt, eq, lt,
                                                im(0), im(1), im(2) );
                 END GENERATE;
```

4-Bit Comparator: “IF . Generate” Statement

```
         m: IF i = n-1 GENERATE
           most: comp1 PORT MAP (a(i), b(i), im(i*3- 3),
                                 im(i*3-2), im(i*3-1), a_gt_b,
                                 a_eq_b, a_lt_b);
             END GENERATE;
--
         r : IF i > 0 AND i < n-1 GENERATE
                 rest: comp1 PORT MAP (a(i), b(i), im(i*3-3),
                            im(i*3-2), im(i*3-1), im(i*3+0),
                            im(i*3+1), im(i*3+2) );
            END GENERATE;
--
END GENERATE; -- Outer Generate
END iterative;
```

Alternative Architecture (Single Generate)

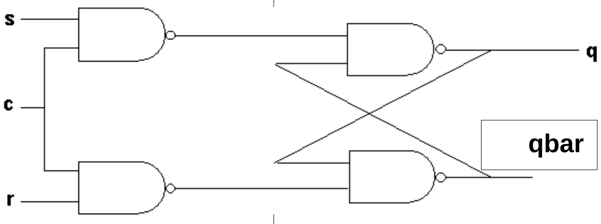
```
ARCHITECTURE Alt_iterative OF nibble_comparator IS
constant n: Positive :=4;
COMPONENT comp1 IS
   PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END COMPONENT;
FOR ALL : comp1 USE ENTITY WORK.bit_comparator (gate_level);
SIGNAL im : BIT_VECTOR ( 0 TO 3*n+2);
BEGIN
im(0 To 2) <= gt&eq&lt;
cALL: FOR i IN 0 TO n-1 GENERATE
           C : comp1 PORT MAP (a(i), b(i), im(i*3), im(i*3+1), im(i*3+2),
                               im(i*3+3), im(i*3+4), im(i*3+5) );
     END GENERATE;
a_gt_b <= im(3*n);
a_eq_b  <= im(3*n+1);
a_lt_b  <= im(3*n+2);
END Alt_iterative ;
```

6-35

Different Binding Schemes

USE 4 2-Input NAND gates to design an SR latch

```
Entity sr_latch IS
   Port (S, R, C : IN Bit; q : Out Bit);
END sr_latch;

ARCHITECTURE Wrong OF sr_latch IS
COMPONENT n2 IS
   PORT (i1, i2: IN BIT; o1: OUT BIT);
END COMPONENT;

FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
SIGNAL im1, im2, im4 : BIT;
BEGIN
g1 : n2 PORT MAP (s, c, im1);
g2 : n2 PORT MAP (r, c, im2);
g3 : n2 PORT MAP (im1, im4, q);
g4 : n2 PORT MAP (q, im2, im4);
END Wrong;
```



-- Error … q declared as Output

SR Latch Modeling …

***Fix #1** (Same Architecture -- Change mode of q to Inout)*

```
Entity sr_latch IS
Port (S, R, C : IN Bit; q : InOut Bit);
END sr_latch;
```

***Fix #2** (Same Architecture -- Change mode of q to Buffer)*

```
Entity sr_latch IS
Port (S, R, C : IN Bit; q : Buffer Bit);
END sr_latch;
```

… SR Latch Modeling …

Fix #3 ***(Use Local Signal - Mode of q is maintained as Out)***

```
ARCHITECTURE Problem OF sr_latch IS
COMPONENT n2 IS
            PORT (i1, i2: IN BIT; o1: OUT BIT);
END COMPONENT;
FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
SIGNAL im1, im2, im3, im4 : BIT;
BEGIN
g1 : n2 PORT MAP (s, c, im1);
g2 : n2 PORT MAP (r, c, im2);
g3 : n2 PORT MAP (im1, im4, im3);
g4 : n2 PORT MAP (im3, im2, im4);
q <= im3;
END Problem;
```

- **Correct Syntax → Problem Oscillating**
- **If all delays are equal then (0,0) on (q, qbar) will oscillate**
- **Remedy by using gates of different delay values**

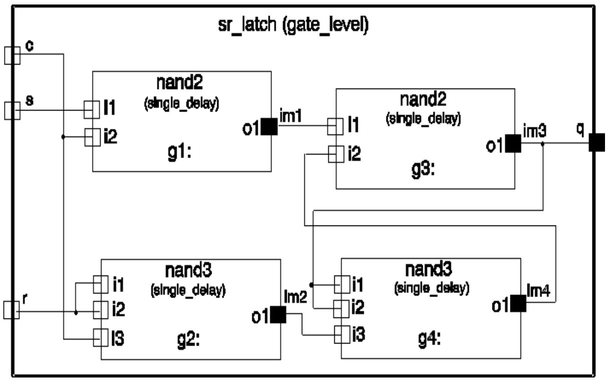
… SR Latch Modeling

```
ARCHITECTURE fast_single_delay OF nand2 IS
BEGIN
    o1 <= i1 NAND i2 AFTER 1 NS;
END fast_single_delay;
```

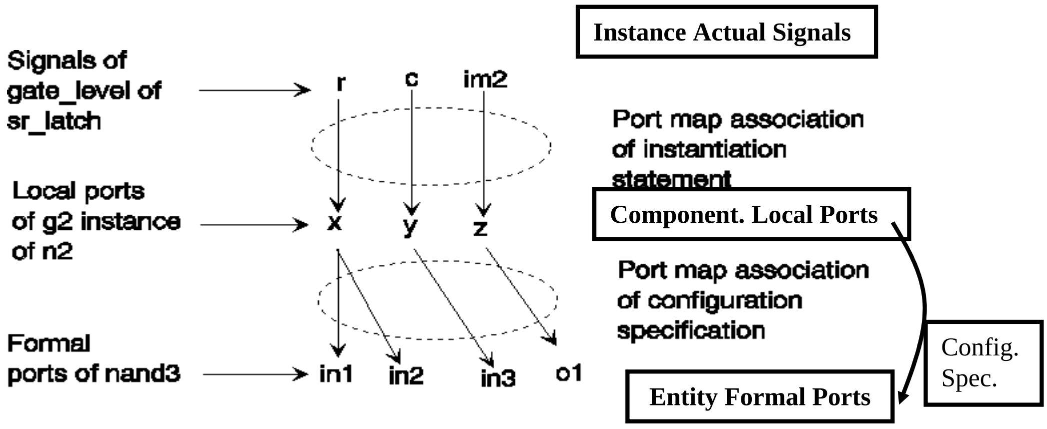
```
ARCHITECTURE Problem OF sr_latch IS
COMPONENT n2 IS
            PORT (i1, i2: IN BIT; o1: OUT BIT);
END COMPONENT;
FOR g1, g3 : n2 USE ENTITY WORK.nand2 (fast_single_delay);
FOR g2, g4 : n2 USE ENTITY WORK.nand2 (single_delay);
SIGNAL im1, im2, im3, im4 : BIT;
BEGIN
g1 : n2 PORT MAP (s, c, im1);
g2 : n2 PORT MAP (r, c, im2);
g3 : n2 PORT MAP (im1, im4, im3);
g4 : n2 PORT MAP (im3, im2, im4);
q <= im3;
END Problem;
```

Binding 3-Input NAND Entity (*Different Delay*) to 2-Input "NAND" Component

```
ARCHITECTURE gate_level OF sr_latch IS
COMPONENT n2
    PORT (x, y: in BIT; z: out BIT);
END COMPONENT;
FOR g1, g3 : n2 USE ENTITY
WORK.nand2 (single_delay) PORT MAP
(x, y, z);
FOR g2, g4 : n2 USE ENTITY
WORK.nand3 (single_delay) PORT MAP
(x, x, y, z);
SIGNAL im1, im2, im3, im4 : BIT;
BEGIN
g1 : n2 PORT MAP (s, c, im1);
g2 : n2 PORT MAP (r, c, im2);
g3 : n2 PORT MAP (im1, im4, im3);
g4 : n2 PORT MAP (im3, im2, im4);
q <= im3;
END gate_level;
```



Port Map Association …


Signals of gate_level of sr_latch
r
c
im2
Port map association of instantiation statement
Local ports of g2 instance of n2
x
y
z
Port map association of configuration specification
Formal ports of nand3
in1
in2
in3
o1

Instance Actual Signals

Component. Local Ports

Config. Spec.

Entity Formal Ports

… Port Map Association

Association is done in two steps:

1. **Instance-To-Component**
   (*Actuals* → *Component Local Ports*)
2. **Componenet-To-Entity** (Configuration **Port Map**)
   (*Component Local Ports* → *Entity Formal Ports*)

- **PORT MAP** in **Instantiation Statements** defines actual signal names corresponding to Component Port Names

actual

- Specifying **PORT MAP** in **Configuration** Statements is Optional

actual

- To Override Entity Local Port Names by Component Port Names, USE **PORT MAP** in **Configuration** Statement

actual

- IF No **Port Map** is Specified in Configuration Statement, Local Port Names of COMPONENT declaration are the Default and they must be the Same as the Formals of the Design Entity.

actual

- A Declared **Signal** Can Be Associated with A Formal Port of **Any Mode** As Long As It Has The **Same Type**

Default Biniding

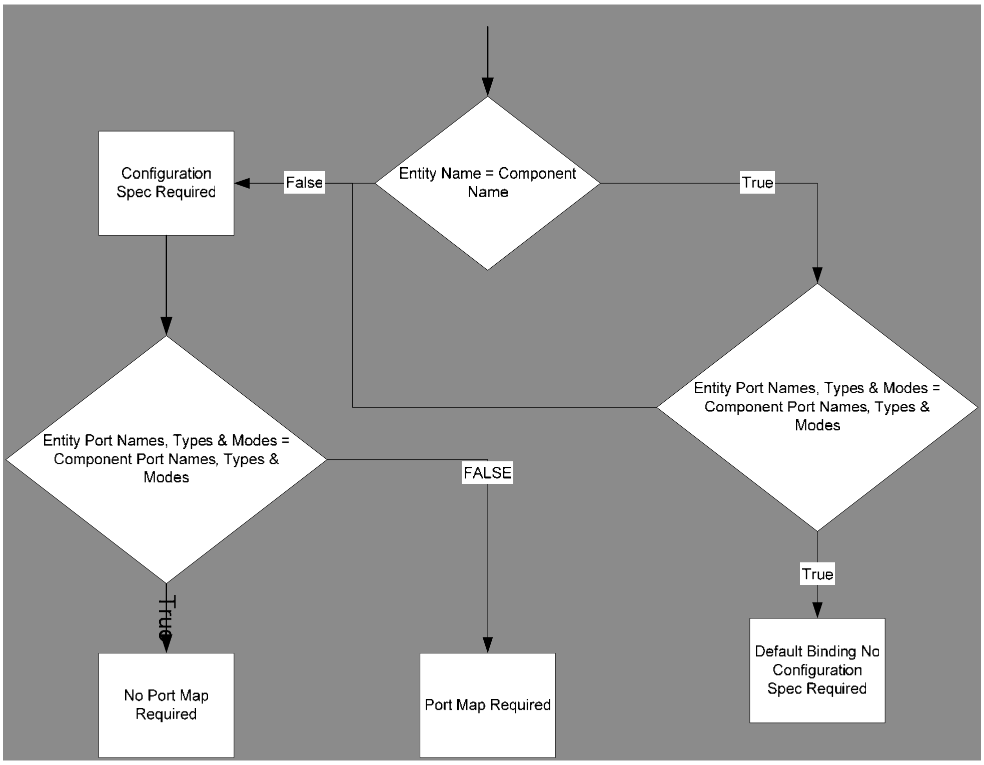
- *Default Binding*: Component instance is Bound to an Entity which :
  - Has the same NAME as the Component
  - Has the same number of Ports as the Component
  - *Ports* must have the *Same Name*, *Type* and be of *Compatible modes*
  - The most Recently analyzed Architecture will Be used
  - A declared **Signal** can be associated with A Formal Port of **Any Mode** as long as it has the **Same Type**
- *Port Compatibility:* Formal => Actual

Actual`s Mode (Instance Ports)

Formal`s Mode
Entity Formal Ports

	IN	OUT	INOUT	BUFFER
IN	X		X	
OUT		X	X	
INOUT			X	
BUFFER				X

Use of Configuration Specifications



Entity Name = Component Name

False

Configuration Spec Required

True

Entity Port Names, Types & Modes = Component Port Names, Types & Modes

FALSE

True

No Port Map Required

Port Map Required

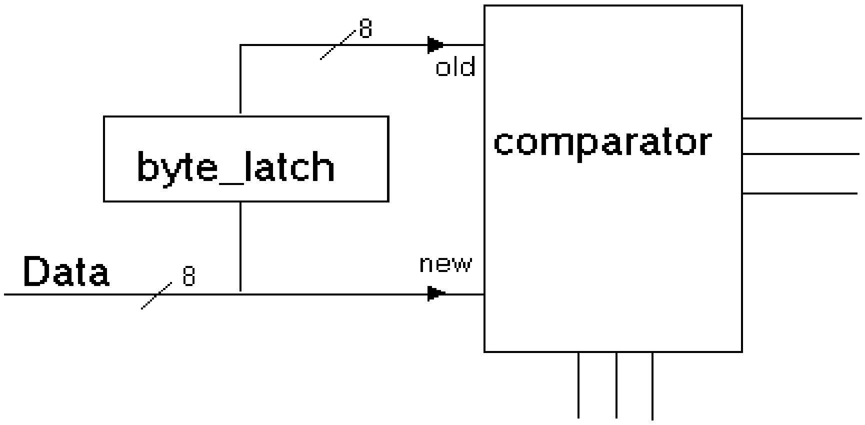
Entity Port Names, Types & Modes = Component Port Names, Types & Modes

True

Default Binding No Configuration Spec Required

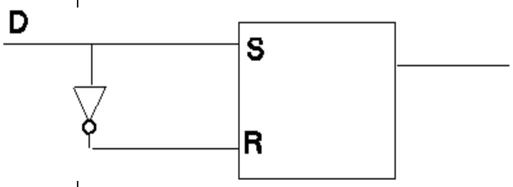
Sequential Comparator …

- Compare consecutive sets of data on an 8-bit input bus.
- Data on the input bus are synchronized with a clock signal



D-Latch

```
ENTITY d_latch IS
  PORT (d, c : IN BIT; q: OUT BIT);
END d_latch;
--
ARCHITECTURE sr_based OF d_latch IS
COMPONENT sr_latch
  PORT (s, r, C : IN BIT; q : OUT
  BIT);
END COMPONENT;
COMPONENT inv
  PORT (i1 : IN BIT; o1 : OUT BIT);
END COMPONENT;
SIGNAL dbar: BIT;
BEGIN
  c1 : sr_latch PORT MAP(d, dbar, c,
  q);
  c2 : inv PORT MAP (d, dbar);
END sr_based;
```

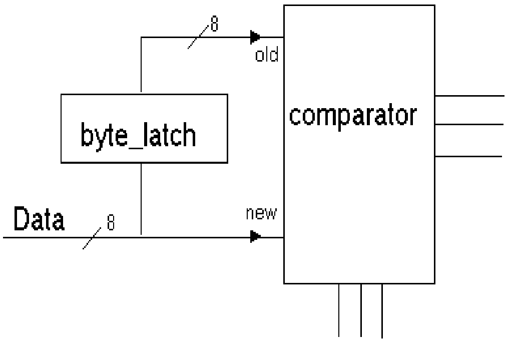


Byte Latch

```
ENTITY byte_latch IS
PORT (di : in BIT_VECTOR (7 DOWNTO 0);   clk : in BIT;
      qo : OUT BIT_VECTOR( 7 DOWNTO 0));
END byte_latch;
--
ARCHITECTURE iterative OF byte_latch IS
COMPONENT d_latch
      PORT (d, c : IN BIT; q : OUT BIT);
END COMPONENT;
BEGIN
g :   FOR i IN di'RANGE GENERATE
         L7DT0 : d_latch PORT MAP (di(i), clk, qo(i));
      END GENERATE;
END iterative;
```

… Sequential Comparator …

```
ENTITY old_new_comparator IS
  PORT (i : in BIT_VECTOR (7 DOWNTO 0);
          clk : in BIT; compare : out  BIT);
END old_new_comparator;
--
ARCHITECTURE wiring Of
    old_new_comparator IS
COMPONENT byte_latch
  PORT (di : in bit_vector (7 downto 0);
      clk : in bit ;  qo : out bit_vector (7 downto 0));
END COMPONENT;
COMPONENT byte_comparator
     PORT (a, b : in bit_vector (7 downto 0); gt,
    eq, lt : in bit; a_gt_b, a_eq_b, a_lt_b : out bit);
END COMPONENT;
SIGNAL con1 : BIT_VECTOR (7 DOWNTO 0);
SIGNAL vdd : BIT := '1';
SIGNAL gnd : BIT := '0';
```



… Sequential Comparator

```
BEGIN
ℓ: byte_latch PORT MAP (i, clk, con1);
c : byte_comparator
    PORT MAP (con1, i, gnd, vdd, gnd, OPEN, compare, OPEN);
END wiring;
```

- *Unconnected Outputs* → *OPEN*
- *Unconnected Inputs (Only with Default Specified)* → *OPEN*

COE 405
Test Benches &&
File I/O

Dr. Alaaeldin A. Amin
Computer Engineering Department

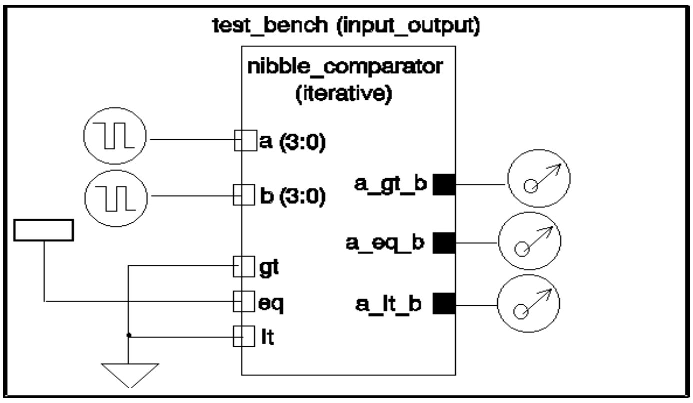
E-mail: amin@ccse.kfupm.edu.sa

Home Page : http://www.ccse.kfupm.edu.sa/~amin

Outline

- Test Benches
- VHDL Files
- File Types & External I/O
  - Opening & Closing Files
  - Text Input and Output
  - Byte Comparator

Test Benches



- A ***Testbench*** is an *Entity without Ports* that has a *Structural Architecture*
- The Testbench Architecture, in general, has 3 major components:
  - Instance of the Entity Under Test (EUT)
  - Test Pattern Generator (Generates Test Inputs for the Input Ports of the EUT)
  - Response Evaluator (Compares the EUT Output Signals to the Expected Correct Output)
- Input & Output ports of the EUT must be declared as local signals

Testbench Example …

```
Entity nibble_comparator_test_bench IS
End nibble_comparator_test_bench ;
--
ARCHITECTURE input_output OF nibble_comparator_test_bench IS
--
COMPONENT comp4 IS
   PORT (a, b : IN bit_vector (3 DOWNTO 0); gt, eq, lt : IN BIT;
            a_gt_b, a_eq_b, a_lt_b : OUT BIT);
END COMPONENT;
--
FOR a1 : comp4 USE ENTITY WORK.nibble_comparator(iterative);
--
SIGNAL a, b : BIT_VECTOR (3 DOWNTO 0);
SIGNAL eql, lss, gtr, gnd : BIT;
SIGNAL vdd : BIT := '1';
--
BEGIN
a1: comp4 PORT MAP (a, b, gnd, vdd, gnd, gtr, eql, lss);
```

eq

…Testbench Example

```
a2: a <= "0000",              -- a = b (steady state)
"1111" AFTER 0500 NS, -- a > b (worst case)
"1110" AFTER 1500 NS, -- a < b (worst case)
"1110" AFTER 2500 NS, -- a > b (need bit 1 info)
"1010" AFTER 3500 NS, -- a < b (need bit 2 info)
"0000" AFTER 4000 NS, -- a < b (steady state, prepare FOR next)
"1111" AFTER 4500 NS, -- a = b (worst case)
"0000" AFTER 5000 NS, -- a < b (need bit 3 only, best case)
"0000" AFTER 5500 NS, -- a = b (worst case)
"1111" AFTER 6000 NS; -- a > b (need bit 3 only, best case)
--
a3 : b <= "0000",             -- a = b (steady state)
"1110" AFTER 0500 NS, -- a > b (worst case)
"1111" AFTER 1500 NS, -- a < b (worst case)
"1100" AFTER 2500 NS, -- a > b (need bit 1 info)
"1100" AFTER 3500 NS, -- a < b (need bit 2 info)
"1101" AFTER 4000 NS, -- a < b (steady state, prepare FOR next)
"1111" AFTER 4500 NS, -- a = b (worst case)
"1110" AFTER 5000 NS, -- a < b (need bit 3 only, best case)
"0000" AFTER 5500 NS, -- a = b (worst case)
"1110" AFTER 6000 NS; -- a > b (need bit 3 only, best case)
END input_output;
```

VHDL Files

- Files provide a way for a VHDL design to communicate with the host environment.
- File declarations make a file available for use to a design.
- Files can be opened for reading and writing:
  - In VHDL87, files are opened and closed when their associated objects come into and out of scope.
  - In VHDL93 explicit **file_open()** and **file_close()** procedures were added.
  - The contents of a file may only be accessed sequentially.
- The package **standard** defines basic file I/O routines for VHDL types.
- The package **textio** defines more powerful routines handling I/O of text files.

VHDL Files

VHDL '87:

FILE identifier **:** file_type **is [mode] "**file_name**";**

VHDL '93:

FILE identifier **:** file_type **[[open mode] is "**file_name**";**

- A file type must be defined for each VHDL type that is to be input from or output to a file:

TYPE bit_file is **FILE** of bit;

File Type & External File I/O …

- Specifying files is a two step process of
  - File **type** declaration
  - File declaration

Type of data contained in this file type

File Type Declaration Example

- **Type** logic_data **is FILE of Character**;

File Declaration Examples

- **FILE** file1 : logic_data; -- *file must be explicitly opened to be associated with a physical file*
- **FILE** file2 : logic_data **IS** "Input.dat"; -- *file IMPLICITLY opened in the default "Read_Mode"*
- **FILE** file3 : logic_data **OPEN READ_MODE IS** "Input.dat";
- **FILE** file4 : logic_data **OPEN Write_Mode IS** "output.dat";
  - File can be opened in READ_MODE, WRITE_MODE, or APPEND_MODE
  - file1, file2, file3 and file4 are *LOGICAL file names*

… Opening & Closing Files …

- **FILE_OPEN** MUST be used with *declared files* which are not IMPLICITLY opened

- **FILE** file1 : logic_data; -- Not Implicitly Opened. → *file must be explicitly opened to be associated with a* ***physical*** *file*
- **FILE** file2 : logic_data **IS** “Input.dat”; -- *file IMPLICITLY opened in the default “Read_Mode”*
- **FILE** file3 : logic_data **OPEN READ_MODE IS** “Input.dat”;
- **FILE** file4 : logic_data **OPEN Write_Mode IS** “output.dat”;

- **FILE_OPEN**(file1, “input.dat”, READ_MODE);
- **FILE_OPEN**(file1, “output.dat”, WRITE_MODE);
- **FILE_CLOSE**(file1);

TEXT Input & Output

- The **TEXTIO** package of the **std** library contains basic functions and procedures:

use std.textio.all;

- The basic **read** and **write** operations of the FILE type are not very useful because they work with binary files.
- The following data types are supported by the **TEXTIO** routines:
  - Bit, Bit_vector
  - Boolean
  - Character, String
  - Integer, Real
  - Time

TEXT Input & Output

- The **TEXTIO** package provides additional **TYPES** and read/write **subprograms** for manipulating text more easily and efficiently:
  - The **LINE** type is a text buffer used to interface VHDL I/O and the file. Only the **LINE** type may read from or write to a file.
  - A new **File** type of **TEXT** is also defined. A file of type **TEXT** may only contain ASCII characters.

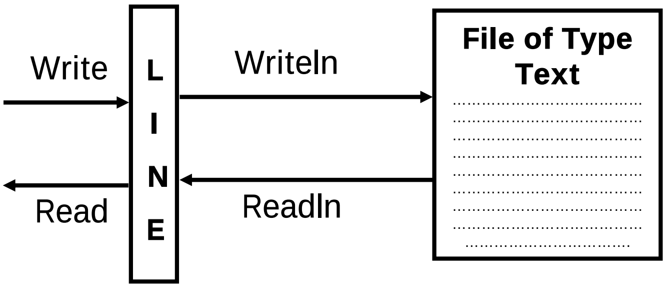
TEXT Input & Output

- Procedures defined by **TEXTIO** package include:

- **Readline(f , k)** **--** reads a **LINE** of file **f** and places it in
  -- buffer **k** (of type **LINE**)
- **Read(k , v1, v2,...) --** reads values **vi** from the **LINE k**
- **write(k, v1, v2,...) --** writes values **vi** to the **LINE k**
- **writeline(f,k) --** writes the **LINE k** to file **f**
- **endfile(f) --** returns true at the end of file

A line is first read from the file via ***readline*** command and is dis-assembled in variables with successive ***read*** statements.

A line is assembled first via ***write*** commands and is finally written to the file with a ***writeline*** statement.



TEXT Input & Output (EXAMPLE)

```
use std.textio.all;
type state is (reset, good);
procedure display_state(current_state: in state) is
variable k : line;
file flush : text OPEN Write_Mode is "debug.txt";
variable state_string : string(1 to 7);
begin
   case current_state is
       when reset => state_string := "reset  ";
       when good  => state_string := "good   ";
   end case;
write (k, state_string, left, 7);
writeline (flush, k);
end display_state;
```

TEXT Input & Output (EXAMPLE)

```
stimuli : process
variable l_in : LINE; variable char : character;
variable data_0,data_1: Bit_vector(7 downto 0);
file stim_in : TEXT is "stim_in.txt";
begin
  w_value_0 <= (others => '0');
  w_value_1 <= (others => '0');
  wait for period;
  while not endfile(stim_in) loop
    readline (stim_in,l_in); read (l_in,data_0);
    w_value_0 <= data_0;
    read (l_in, char); read (l_in, data_1);
    w_value_2 <= data_1;
    wait for period;
  end loop;
  wait;
end process stimuli;
```

TEXT Input & Output (EXAMPLE)

```
Fultst: Process
------------------------
VAriable ll: Line;
Variable N1: Integer range 0 to 255;
Variable P1, rslt: Integer;
begin
Lp0: While not EndFile (Stim_in) Loop
              readline (stim_in , ll);
               read(ll, N1);
              N <=  Int2Bin (N1 , 8);  -- N is a Bit_Vector 7 downto 0)
              wait for Period;
     ….. “More Processing”
 End Loop Lp0;
File_Close(stim);
end Process Fultst;
```

TEXTIO Package (READ)

```
PROCEDURE READLINE(file f: TEXT; L: out LINE);
PROCEDURE READ(L:inout LINE; VALUE: out bit);
PROCEDURE READ(L:inout LINE; VALUE: out bit_vector);
PROCEDURE READ(L:inout LINE; VALUE: out BOOLEAN);
PROCEDURE READ(L:inout LINE; VALUE: out character);
PROCEDURE READ(L:inout LINE; VALUE: out integer);
PROCEDURE READ(L:inout LINE; VALUE: out real);
PROCEDURE READ (L:inout LINE; VALUE: out string);
PROCEDURE READ (L:inout LINE; VALUE: out time);
```

TEXTIO Package (Write)

```
PROCEDURE WRITELINE(file f : TEXT; L : inout LINE);
PROCEDURE WRITE(L : inout LINE; VALUE : in bit;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0);
PROCEDURE WRITE(L : inout LINE; VALUE : in bit_vector;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0);
PROCEDURE WRITE(L : inout LINE; VALUE : in BOOLEAN;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0);
PROCEDURE WRITE(L : inout LINE; VALUE : in character;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0);
PROCEDURE WRITE(L : inout LINE; VALUE : in integer;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0);
```

TEXTIO Package (Write)

```
PROCEDURE WRITE(L : inout LINE; VALUE : in real;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0;
          DIGITS: in NATURAL := 0);
   PROCEDURE WRITE(L : inout LINE; VALUE : in string;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0);
   PROCEDURE WRITE(L : inout LINE; VALUE : in time;
          JUSTIFIED: in SIDE := right;
          FIELD: in WIDTH := 0;
          UNIT: in TIME := ns);
```

Test Bench Package (Write)

```
Package  T_Bench IS
    File Stims_in: text open read_mode is "./Data_in.txt";
    File RSPNS: text open write_mode is "./Data_out.txt";
    Shared Variable ll: Line;
---------------------------------------
    Constant Dash10: String( 1 to  10):=(Others => '-');
    Constant Dash30: String( 1 to  30):=(Others => '-');
    Constant Dash60: String( 1 to  60):=(Others => '-');
    Constant eqs: String(1 to 3) := " = ";
End T_Bench;
```

SAMPLE Test Bench (1)

Repeated ADD Multiplier

```
Use std.TEXTIO.ALL;
Use work.Bin_Arith.ALL;
Use work.T_Bench.ALL;
entity tst_MR is
end tst1_MR;
----------------
Architecture struc of tst_MR is
--
Component Add_Multr is
     Generic (n : positive:=4);
     Port (    A, X : in Unsigned(n-1 downto 0);
          Strt, Reset, clk: in Bit;
          P : out Unsigned(2*n-1 downto 0);
          Done: out Boolean := False);
end Component;
--
Signal A, X : Unsigned(3 downto 0);
Signal Strt, Reset: Bit;
Signal Clk   : Bit := '0';
Signal P :  Unsigned(7 downto 0);
Signal Done, Stop_Clk:  Boolean := False;
--
begin
Clk <= not clk after 10 ns when not(stop_Clk) else '0';
--
MR1: Add_Multr
     Generic Map (n => 4)
     Port Map( A, X, Strt, Reset, clk, P, Done);
tst:Process
     variable nn : Positive:=4;
     variable Ntot: Positive :=2**nn;
     constant crrct: string(1 to 16):= " Correct Result ";
     constant wrng: string(1 to 21) := " *** WRONG Result ***";
     begin
        Reset <= '1';
        wait For 5 ns;
        Reset <= '0';
        Strt <= '0';
        wait For 5 ns;
outLp:       For i in Ntot-1 downto 0 Loop
          A <= Int2Unsigned(i,nn);
inrLp:         For j in i downto 0 Loop
               X <= Int2Unsigned(j,nn);
```

```
               wait For 0 ns; -- Causes Delta time to pass For A &
                              --X to assume their new values
               write(ll,"A"&eqs); -- writes "A = "
               write(ll,Bit_Vector(A)); -- writes Bit_Vector value
                                        -- of A
               write(ll,"("&eqs); -- writes "( = "
               write(ll, i); -- writes the integer value of A
               write(ll,R_Pran&Spc&"  X  ");
-----
               write(ll,"B"&eqs); -- writes "B = "
               write(ll,Bit_Vector(X)); -- writes Bit_Vector value
                                        -- of B
               write(ll,"("&eqs); -- writes "( = "
               write(ll, j); -- writes the integer value of B
               write(ll,")" &eqs); -- writes ")"
               strt <='1';
               wait until not(Done) and clk='1' and clk'event;
               strt <='0';
               wait until Done;
               write(ll,Bit_Vector(P)); -- writes Bit_Vector value
                                        -- of P
               write(ll,"("&eqs); -- writes "( = "
               write(ll, IntVal(P)); -- writes the integer value
                                     -- of P
               write(ll,R_Pran&Spc&Arrow); -- writes ")"
               IF IntVal(P)= i*j then
                 write(ll, crrct);
               else
                 write(ll, wrng);
               end if;
               writeline(Rspns, ll);
               write(ll, dash40);
               writeline(Rspns, ll);
           End Loop;
           write(ll, dash85);
           writeline(Rspns, ll);
           write(ll, dash85);
           writeline(Rspns, ll);
        End Loop;
        stop_Clk <= True;
        File_Close(Rspns);
        wait;
     End Process;


end struc;
```

SAMPLE Test Bench (2)

Repeated ADD Multiplier

```
----------------
Architecture  struc2 of tst_MR is
--
Component Add_Multr is
      Generic (n : positive:=4);
      Port (      A, X : in Unsigned(n-1 downto 0);
            Strt, Reset, clk: in Bit;
            P : out Unsigned(2*n-1 downto 0);
            Done: out Boolean := False);
end Component;
--
Signal A, X : Unsigned(3 downto 0);
Signal Strt, Reset: Bit;
Signal Clk   : Bit := '0';
Signal P :  Unsigned(7 downto 0);
Signal Done, Stop_Clk:  Boolean := False;
--
begin
Clk <= not clk after 10 ns when not(stop_Clk) else '0';
--
MR1: Add_Multr
      Generic Map (n => 4)
      Port Map(   A, X, Strt, Reset, clk, P, Done);
tst:Process
      variable Lin : Line;
      variable nn : Positive:=4;
      variable i, j: Positive ;
      constant crrct: string(1 to 16):= " Correct Result ";
      constant wrng: string(1 to 21) := " *** WRONG Result ***";
      begin
      Readline(Stims, Lin);
      Read(Lin, nn);
         Reset <= '1';
         wait For 5 ns;
         Reset <= '0';
         Strt <= '0';
         wait For 5 ns;
outLp:         While not(EndFile(Stims)) loop
            Readline(Stims, Lin);
            Read(Lin, i);
            Read(Lin, j);
            A <= Int2Unsigned(i,nn);
            X <= Int2Unsigned(j,nn);
            wait for 0 ns; -- Causes Delta time to pass for A & X to assume
                           -- their new values
            write(ll,"A"&eqs); -- writes "A = "
            write(ll,Bit_Vector(A)); -- writes Bit_Vector value of A
            write(ll,"("&eqs); -- writes "( = "
            write(ll, i); -- writes the integer value of A
            write(ll,R_Pran&Spc&"  X  ");
-----
            write(ll,"B"&eqs); -- writes "B = "
            write(ll,Bit_Vector(X)); -- writes Bit_Vector value of B
```

```
            write(ll,"("&eqs); -- writes "( = "
            write(ll, j); -- writes the integer value of B
            write(ll,")" &eqs); -- writes ")"
            strt <='1';
            wait until not(Done) and clk='1' and clk'event;
            strt <='0';
            wait until Done;
            write(ll,Bit_Vector(P)); -- writes Bit_Vector value of P
            write(ll,"("&eqs); -- writes "( = "
            write(ll, IntVal(P)); -- writes the integer value of P
            write(ll,R_Pran&Spc&Arrow); -- writes ")"
            IF IntVal(P)= i*j then
              write(ll, crrct);
            else
              write(ll, wrng);
            end if;
            writeline(Rspns, ll);
            write(ll, dash85);
            writeline(Rspns, ll);
            write(ll, dash85);
            writeline(Rspns, ll);
         End Loop;
         stop_Clk <= True;
         File_Close(Stims);
         File_Close(Rspns);
         wait;
      End Process;


end struc2;
```

Chapter 6

Design_Organization & Parameterization

OUTLINE

- **Subprograms *for Test Benches***
- **Design Parametrization**
- **Design libraries**
  - **IEEE Library**
    - **Std_Logic_1164 Package (9-Valued Logic)**
    - **Numeric_Std" Package**

TEST BENCH SubPrograms

```
ARCHITECTURE procedural OF nibble_comparator_test_bench is
TYPE integers IS ARRAY (0 TO 12) OF INTEGER;

PROCEDURE apply_data (SIGNAL target : OUT BIT_VECTOR
                        (3 DOWNTO 0); CONSTANT values : IN integers;
                        CONSTANT period : IN TIME) IS
VARIABLE buf : BIT_VECTOR (3 DOWNTO 0);
BEGIN
FOR i IN values'LOW TO values'High LOOP
    int2bin (values(i), buf); -- buf := int2bin(values(i));
    target <= TRANSPORT buf AFTER i * period;
END LOOP;
END apply_data;

Component comp4 PORT (a, b : IN bit_vector (3 DOWNTO 0);
              gt, eq, lt : IN BIT;   a_gt_b, a_eq_b, a_lt_b : OUT  BIT);
END Component;

FOR a1 : comp4 USE ENTITY work.nibble_comparator(structural);

SIGNAL a, b : BIT_VECTOR (3 DOWNTO 0);
SIGNAL eql, lss, gtr, gnd : BIT:= '0';          SIGNAL vdd : BIT := '1';

BEGIN
a1: comp4 PORT MAP (a, b, gnd, vdd, gnd, gtr, eql, lss);
apply_data (a, 0&15&15&14&14&14&14&10&00&15&00&00&15,
              500 NS);
apply_data (b, 0&14&14&15&15&12&12&12&15&15&15&00&00,
              500 NS);
END procedural;
```

Type Mismatch??

- Two *concurrent* procedure calls of *apply_data.*
- *P*rocedure uses *integers* and places *binary equivalent* on target signal
- Invoked at initialization or when inputs change (*Here Called Once*)

Design_Parametrization

```
ENTITY   inv_t   IS
     GENERIC (tplh : TIME := 3 NS; tphl : TIME := 5 NS);
     PORT (i1 : in BIT; o1 : out BIT);
END inv_t;
--
ARCHITECTURE average_delay OF inv_t IS
BEGIN
     o1 <= NOT i1 AFTER (tplh + tphl) / 2;
END average_delay;
______________________________________________________

ARCHITECTURE Asym_delay OF inv_t IS
BEGIN
     Process(i1)
     Variable V: Bit;
     Begin
         V := NOT i1;
         Case V is
              When '0'  => o1 <= '0' after tphl;
              When '1'  => o1 <= '1' after tplh;
         End Case;
     END;
END Asym_delay;
```

```
ENTITY nand2_t IS
      GENERIC (tplh : TIME := 4 NS; tphl : TIME := 6 NS);
      PORT (i1, i2 : IN BIT; o1 : OUT BIT);
      END nand2_t;
--
ARCHITECTURE average_delay OF nand2_t IS
BEGIN
      o1 <= i1 NAND i2 AFTER (tplh + tphl) / 2;
END average_delay;
```

- **GENERIC**s allow passing various design parameters
- New versions of gate descriptions (Architecture bodies) contain timing information.
- **GENERIC**s can include default values

```
ENTITY nand3_t IS
      GENERIC (tplh : TIME := 5 NS; tphl : TIME := 7 NS);
      PORT (i1, i2, i3 : IN BIT; o1 : OUT BIT);
END nand3_t;
--
ARCHITECTURE average_delay  OF  nand3_t  IS
BEGIN
     o1 <= NOT ( i1 and i2 and i3 ) AFTER (tplh + tphl)/2;
END average_delay;
```

Values Passed to Generic Parameters

Several alternatives for passing values to **GENERIC**s

1. Using entity defaults
2. Passing_Values Through Component Defaults
3. Assigning fixed values
4. Passing values from higher level components

1. Using Entity Default_Values

*No **Generics** Specified in Component Declarations*

```
ARCHITECTURE default_delay OF bit_comparator IS
Component n1 is PORT (i1: IN BIT; o1: OUT BIT);
END Component;
Component n2 is PORT (i1, i2: IN BIT; o1: OUT BIT);
END Component;
Component n3 is PORT (i1, i2, i3: IN BIT; o1: OUT BIT);
END Component;
FOR ALL : n1 USE ENTITY WORK.inv_t (average_delay);
FOR ALL : n2 USE ENTITY WORK.nand2_t (average_delay);
FOR ALL : n3 USE ENTITY WORK.nand3_t (average_delay);
-- Intermediate signals
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
```

```
BEGIN
-- a_gt_b output
g0 : n1 PORT MAP (a, im1);
g1 : n1 PORT MAP (b, im2);
g2 : n2 PORT MAP (a, im2, im3);
g3 : n2 PORT MAP (a, gt, im4);
g4 : n2 PORT MAP (im2, gt, im5);
g5 : n3 PORT MAP (im3, im4, im5, a_gt_b);
-- a_eq_b output
g6 : n3 PORT MAP (im1, im2, eq, im6);
g7 : n3 PORT MAP (a, b, eq, im7);
g8 : n2 PORT MAP (im6, im7, a_eq_b);
-- a_lt_b output
g9 : n2 PORT MAP (im1, b, im8);
g10 : n2 PORT MAP (im1, lt, im9);
g11 : n2 PORT MAP (b, lt, im10);
g12 : n3 PORT MAP (im8, im9, im10, a_lt_b);
END default_delay;
```

- Component declarations do not contain **GENERIC**s
- Component instantiation are as before (Only Port maps)
- Entity default values are used.

2. Passing_Values Through Component Defaults

```
ARCHITECTURE iterative OF nibble_comparator IS
Component comp1 is
Generic (tplh1: Time := 2 ns; tplh2: Time:= 3 ns; tplh3: Time:=
  4 ns; tphl1: Time:= 4 ns; tphl2: Time:= 5 ns; tphl3: Time:= 6 ns);
Port (a, b, gt, eq, lt : in Bit; a_gt_b, a_eq_b, a_lt_b : Out Bit);
END Component;
FOR ALL : comp1 USE ENTITY WORK.bit_comparator_t
(passed_delay);
SIGNAL im : BIT_VECTOR ( 0 TO 8);
BEGIN
c0: comp1 Port Map (a(0), b(0), gt, eq, lt, im(0), im(1), im(2));
            -- No Generic Map…
c1to2: FOR i IN 1 TO 2 GENERATE
c: comp1 PORT MAP (a(i), b(i), im(i*3-3), im(i*3-2), im(i*3-
1), im(i*3+0), im(i*3+1), im(i*3+2) ); -- No Generic Map…
END GENERATE;
c3: comp1 PORT MAP (a(3), b(3), im(6), im(7), im(8),
a_gt_b, a_eq_b, a_lt_b); -- No Generic Map…
END iterative;
```

- Exclusion of **GENERIC Map** from instances leaves all parameters **OPEN** → *Declared Component Default Generic Parameter Values are used.*

3. Assigning Fixed Values to Generic Parameters

```
ARCHITECTURE fixed_delay OF bit_comparator IS

Component n1 is
Generic (tplh, tphl : Time); Port (i1: in Bit; o1: out Bit);
END Component;

Component n2 is
Generic (tplh, tphl : Time); Port (i1, i2: in Bit; o1: out Bit);
END Component;

Component n3 is
Generic (tplh, tphl : Time); Port (i1, i2, i3: in Bit; o1: out Bit);
END Component;

FOR ALL : n1 USE ENTITY WORK.inv_t (average_delay);
FOR ALL : n2 USE ENTITY WORK.nand2_t (average_delay);
FOR ALL : n3 USE ENTITY WORK.nand3_t (average_delay);

-- Intermediate signals
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;

BEGIN
-- a_gt_b output
g0 : n1 Generic Map (2 NS, 4 NS) Port Map (a, im1);
g1 : n1 Generic Map (2 NS, 4 NS) Port Map (b, im2);
g2 : n2 Generic Map (3 NS, 5 NS) Port Map (a, im2, im3);
g3 : n2 Generic Map (3 NS, 5 NS) Port Map P (a, gt, im4);
g4 : n2 Generic Map (3 NS, 5 NS) Port Map (im2, gt, im5);
g5 : n3 Generic Map (4 NS, 6 NS) Port Map (im3, im4, im5,
a_gt_b);
```

```
-- a_eq_b output
g6 : n3 Generic Map (4 NS, 6 NS) Port Map (im1, im2, eq, im6);

-- a_lt_b output
g9 : n2 Generic Map (3 NS, 5 NS) Port Map (im1, b, im8);

END fixed_delay;
```

- Component declarations contain **GENERIC**s
- Component instantiation contain **GENERIC MAP Values**
- **GENERIC MAP** specified **Values** overwrite default values

4. Passing_Values From Higher Level Specs

Higher Level Entity

Generic Parameters To be Passed to Lower Instances

```
ENTITY bit_comparator_t   IS
    GENERIC (tplh1, tplh2, tplh3, tphl1, tphl2, tphl3 : TIME);
    PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b,
          a_lt_b : OUT BIT);
END bit_comparator_t;
```

- To pass values, higher level units must contain **GENERIC**s
- A timed *bit_comparator* is developed

```
ARCHITECTURE passed_delay OF bit_comparator_t IS
```

```
Component n1is
Generic (tplh, tphl : Time); Port (i1: in Bit; o1: out Bit);
END Component;
```

```
Component n2 is
Generic (tplh, tphl : Time); Port (i1, i2: in Bit; o1: out Bit);
END Component;
```

```
Component n3 is
Generic (tplh, tphl : Time); Port (i1, i2, i3: in Bit; o1: out Bit);
END Component;
```

```
FOR ALL : n1 USE ENTITY WORK.inv_t (average_delay);
FOR ALL : n2 USE ENTITY WORK.nand2_t (average_delay);
FOR ALL : n3 USE ENTITY WORK.nand3_t (average_delay);
```

```
-- Intermediate signals
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : Bit;
```

Passed Parameters

```
BEGIN
-- a_gt_b output
```

```
g0 : n1 Generic Map (tplh1, tphl1) Port Map (a, im1);
g1 : n1 Generic Map (tplh1, tphl1) Port Map (b, im2);
g2 : n2 Generic Map (tplh2, tphl2) Port Map (a, im2, im3);
g3 : n2 Generic Map (tplh2, tphl2) Port Map (a, gt, im4);
g4 : n2 Generic Map (tplh2, tphl2) Port Map (im2, gt, im5);
g5 : n3 Generic Map (tplh3, tphl3) Port Map (im3, im4, im5,
                                                        a_gt_b);
```

```
-- a_eq_b output
g6 : n3 Generic Map (tplh3, tphl3) Port Map (im1, im2, eq, im6);
```

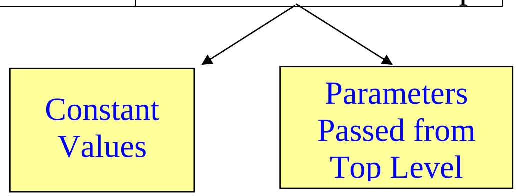
```
-- a_lt_b output
g9 : n2 Generic Map (tplh2, tphl2) Port Map (im1, b, im8);
```

```
END passed_delay;
```

- **Component** *declarations* include **GENERIC**s
- **Component** *instantiations* include passed values
- **GENERIC** maps are required

5. Summary

Entity Generic	Component Generic	Instance Generic Map	
√	X	X	Default Entity Generic Values
√	√	X	Default Component Generic Values
√	√	√	As Specified by Instance Generic Map



6. Combining Parameter with Default Values (Association with OPEN) & Passing Values to Other Parameter

Alternative 1: (Generics Map Positional Association → OPEN Keyword)

```
ARCHITECTURE iterative OF nibble_comparator IS
BEGIN
c0: comp1
    GENERIC MAP (Open, Open, 8 NS, Open, Open, 10 NS)
    PORT MAP (a(0), b(0), gt, eq, lt, im(0), im(1), im(2));
………………….
END iterative;
```

Alternative 2: (Generic Map Named Association)

```
ARCHITECTURE iterative OF nibble_comparator IS
………………….
BEGIN
c0: comp1
GENERIC MAP (tplh3 => 8 NS, tphl3 => 10 NS)
PORT MAP (a(0), b(0), gt, eq, lt, im(0), im(1), im(2));

……………………
END iterative;
```

- A **GENERIC Map** May specify only some of the parameters
- Using **OPEN** causes use of default Component Values
- Alternatively, association by name can be used

Design Libraries

- A *Design Library* is A Set of *Pre-Compiled* *Pre-Analyzed* *Design Units*.
- **Design Units**:
  - Entities,
  - Architecture Bodies,
  - Package Declarations,
  - Package Bodies, and
  - Configurations

Two Types of Libraries:

1. Working Library (**WORK**) {*A Predefined library into which a Design Unit is Placed after Compilation.*},
2. Resource Libraries {Contain design units that can be referenced within the design unit being compiled}.

- Only one library can be the **Working** Library
- Any number of Resource Libraries May be Used by a Design Entity
- There is a Number of Predefined Resource Libraries
- The **Library** Clause is Used To Make A Given Library Visible
- The **Use**-Clause Causes Package Declarations Within a Library to be Visible

- Library Management Tasks, e.g. Creation or Deletion, are not Part of the VHDL Language Standard → Tool Dependent

Predifined Libraries:

- The **STD** Library Contains 2 *Packages* **Standard** and **Textio** (*See Appendix F in the Textbook*).
- The *Standard* Package Contains All the Predefined Data Types, e.g. BIT, BIT_Vector, Character, Integer, Boolean, etc.
- The *Textio* Package Contains some Utilities for reading and writing of Data into Files
- By Default, Every Design Unit is Assumed to Contain the Following Declarations:

```
LIBRARY     STD , work ;
USE         STD.Standard.All ;
```

- Another Standard Library is the **IEEE** Library which Contains the Standard *Package* **Std_Logic_1164** which is becoming a De Facto Standard for all Synthesis Tools (*See Appendix G in the Textbook*).
- The **Std_Logic_1164** Package Defines a new Logic System consisting of 9-Valued Data Type and Related Utility Functions and Procedures For This 9-Vlued System.
- Subtypes of this 9-Valued Are Also Defined Together with Overloaded Functions and Operations on these Subtypes

```
LIBRARY   IEEE ;
USE       IEEE.STD_Logic_1164.All ;
```

STD_LOGIC_1164 9-Valued Logic System

TYPE **std_ulogic** **IS** **(**

'U', -- Uninitialized {*Important For Sequential Systems*}

'X', -- Forcing Unknown {*Contention*}

'0', -- Forcing 0

'1', -- Forcing 1

'Z', -- High Impedance

'W', -- Weak Unknown

'L', -- Weak 0 {*,e.g. Logic 0 Held Dynamically on a Capacitor*}

'H', -- Weak 1 {*,e.g. Logic1 Held Dynamically on a Capacitor*}

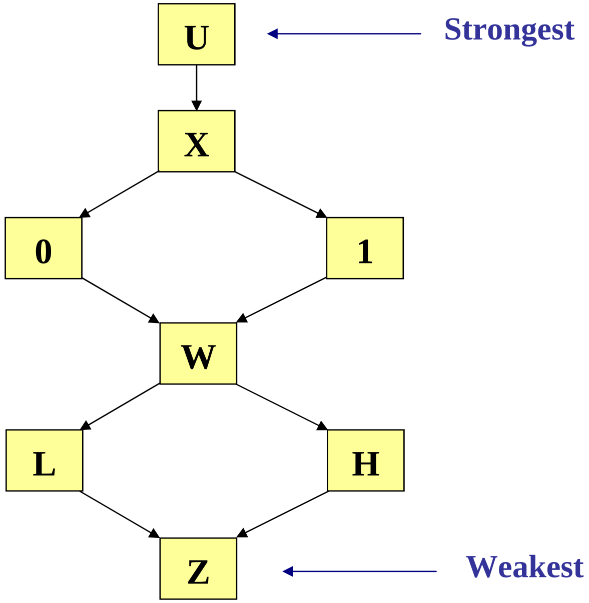
'-' -- Don't care{*Used To Optimize Synthesis*}

);

TYPE ***std_ulogic_vector*** **IS Array** (Natural Range <>) **OF** ***std_ulogic*** **;**

Resolving Multiple Values on the Same Signal Driver:

1. Strong Dominates Weak
2. Weak Dominates Z
3. Conflicts of Equal Strength is Unknown at That Strength



TYPE *stdlogic_table* **IS ARRAY**(*std_ulogic*, *std_ulogic*) **OF** *std_ulogic* **;**

```
-- resolution function
-------------------------------
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_ulogic* is the base type defined in the 1164 standard
  - This is an unresolved type
  - *std_ulogic* cannot have multiple drivers
- *Std_logic* is the *resolved* **subtype** of *std_ulogic*
- A *resolved* type is always a **subtype** of another *unresolved* type

SUBTYPE *std_logic* **IS resolved std_ulogic ;**
TYPE *std_logic_vector* **Is Array (Natural Range <>) OF** *std_logic* **;**

```
FUNCTION resolved ( s : std_ulogic_vector ) RETURN
                                        std_ulogic  IS
 VARIABLE result : std_ulogic := 'Z';  -- weakest state default
   BEGIN
     IF   (s'LENGTH = 1) THEN   RETURN s(s'LOW); ELSE
        FOR i IN s'RANGE LOOP
           result := resolution_table (result, s(i));
        END LOOP;
     END IF;
     RETURN result;
   END resolved;
```

❖ You should use ***std_ulogic*** and ***std_ulogic_vector*** for signals that only require one driver
  – accidental connections between signals that should only have one driver can be detected by the compiler

Illustrative Examples

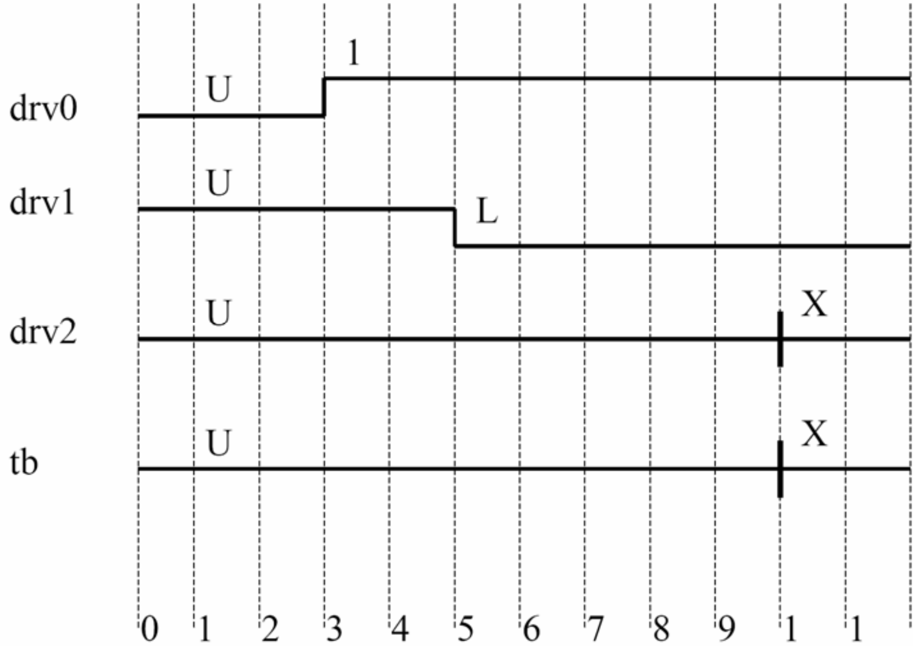
```
Signal ta,tb : bit;
begin
   ta <= transport '1' after 2 ns;
   tb <= transport '1' after 3 ns;
p1:process
begin
   tb <= transport '0' after 5 ns;
   wait;
end process p1;
```

.
.
.

```
--ERROR: Nonresolved Signal tb
'tb' has multiple drivers
```

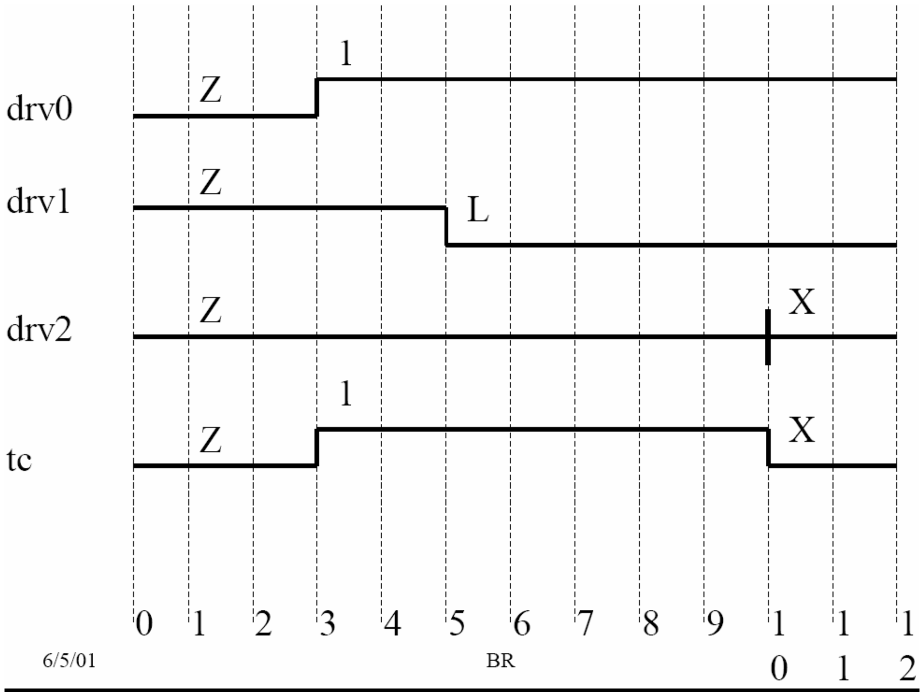
```
Signal tb : std_logic;
begin
-- 'tb' has multiple drivers
   tb <= transport '1' after 3 ns;   -- DRV0
p1:process
begin
   tb <= transport ‘L' after 5 ns;   -- DRV1
   wait;
end process p1;
   tb <= transport ‘X’ after 10 ns; -- DRV 2
end;
```

What does the waveform of signal tb look like?



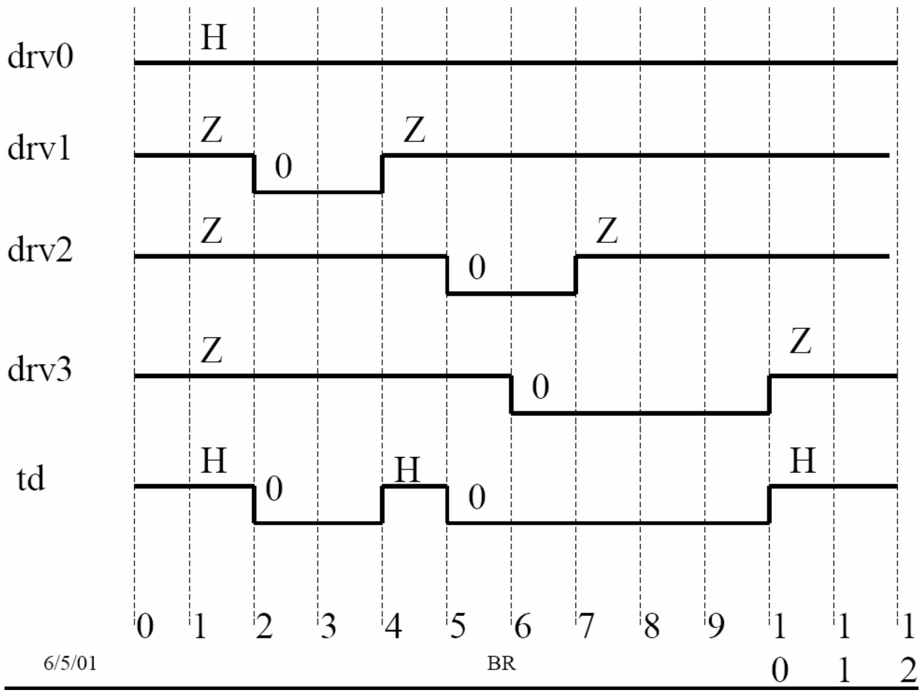
```
Signal tc : std_logic := ‘Z’;
begin
-- 'tc' has multiple drivers
   tc <= transport '1' after 3 ns; -- DRV0
p1:process
begin
   tc <= transport ‘L' after 5 ns; -- DRV1
   wait;
end process p1;
   tc <= transport ‘X’ after 10 ns; -- DRV 2
end;
```

What does the waveform of signal tc look like?



```
Signal td : std_logic := 'Z';
begin
-- emulates a pullup resistor using 'H' drive
td <= 'H';     -- DRV0
               -- DRV1
td <= transport '0' after 2 ns, 'Z' after 4 ns;
               -- DRV2
td <= transport '0' after 5 ns, 'Z' after 7 ns;
               -- DRV3
td <= transport '0' after 6 ns,'Z' after 10 ns;
end;
```

What does the waveform of signal td look like?



```
Function  resolved ( s : std_ulogic_vector ) RETURN  std_ulogic  is
-- 'S' is a list of all driver values for the signal to be resolved.
 VARIABLE result : std_ulogic := 'Z';  -- weakest state default
   BEGIN
      IF    (s'LENGTH = 1) THEN    RETURN s(s'LOW); ELSE
          FOR i IN s'RANGE LOOP
             result := resolution_table (result, s(i));
          END LOOP;
       END IF;
       RETURN result;
   END resolved;
```

SUBTYPE *std_logic* IS resolved std_ulogic ;

Type *std_logic_vector* Is Array (Natural Range <>) of *std_logic* ;

- ❖***std_ulogic*** is an *Unresolved* 9-Valued System
- ❖The *resolved* function resolves ***std_ulogic*** Values
- ❖The ***std_logic*** and ***std_logic_vector*** are *de-facto* industry standard

What Else in the 1164 Package?

- ❖ Boolean functions (AND, OR, etc)
- ❖ Subtypes with restricted members (X01, X01Z, UX01, UX01Z)
  - – conversion functions to/from vector types to these types
  - – useful in models where you do not want to deal with the full range of types
- ❖ Vectorized conversion functions
- ❖ Misc functions
  - ▪ rising_edge, falling_edge, is_X
- ❖ Overloading logical operators
- ❖ Vectorized operators are overloaded
- ❖ Vectorized operators are for resolved and unresolved types
- ❖ Conversion functions for common logic value systems
- ❖ Conversion to systems without strength (Strength Strippers)

Subsets of the *std_logic* Data Type

SUBTYPE X01 IS *resolved* ***std_ulogic*** **RANGE** 'X' TO '1';

-- ('X','0','1')

SUBTYPE X01Z IS *resolved* ***std_ulogic*** **RANGE** 'X' TO 'Z';

--('X','0','1','Z')

SUBTYPE UX01 IS *resolved* ***std_ulogic*** **RANGE** 'U' TO '1';

-- ('U','X','0','1')

SUBTYPE UX01Z IS *resolved* ***std_ulogic*** **RANGE** 'U' TO 'Z';

-- (‘U’,'X','0','1', ‘Z’)

- Subtypes provide mapping to common value systems ***without Strength*** X01, X01Z, UX01, UX01Z
- The ***std_logic*** is a *Superset* of most common value systems

---

-- Overloaded Logical Operators

```
-------------------------------------------------------------------

FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;
FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

-------------------------------------------------------------------
```

-- Vectorized Overloaded Logical Operators

```
-------------------------------------------------------------------
FUNCTION "and"  ( l, r : std_logic_vector  ) RETURN
std_logic_vector;
FUNCTION "and"  ( l, r : std_ulogic_vector ) RETURN
std_ulogic_vector;

FUNCTION "nand" ( l, r : std_logic_vector  ) RETURN
std_logic_vector;
FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN
std_ulogic_vector;
```

```
FUNCTION "or"   ( l, r : std_logic_vector  ) RETURN
std_logic_vector;
FUNCTION "or"   ( l, r : std_ulogic_vector ) RETURN
std_ulogic_vector;

FUNCTION "nor"  ( l, r : std_logic_vector  ) RETURN
std_logic_vector;
FUNCTION "nor"  ( l, r : std_ulogic_vector ) RETURN
std_ulogic_vector;

FUNCTION "xor"  ( l, r : std_logic_vector  ) RETURN
std_logic_vector;
FUNCTION "xor"  ( l, r : std_ulogic_vector ) RETURN
std_ulogic_vector;

Function "xnor" ( l, r : std_logic_vector  ) return
std_logic_vector;
Function "xnor" ( l, r : std_ulogic_vector ) return
std_ulogic_vector;

FUNCTION "not"  ( l : std_logic_vector  ) RETURN
std_logic_vector;
FUNCTION "not"  ( l : std_ulogic_vector ) RETURN
std_ulogic_vector;
```

```
-------------------------------------------------------------------
            -- Conversion Functions
-------------------------------------------------------------------
FUNCTION To_bit      ( s : std_ulogic;       xmap : BIT := '0')
RETURN BIT;
FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT
:= '0') RETURN BIT_VECTOR;

FUNCTION To_bitvector ( s : std_ulogic_vector; xmap :
BIT := '0') RETURN BIT_VECTOR;

FUNCTION To_StdULogic   ( b : BIT   ) RETURN std_ulogic;

FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        )
RETURN std_logic_vector;

FUNCTION To_StdLogicVector  ( s : std_ulogic_vector )
RETURN std_logic_vector;

FUNCTION To_StdULogicVector ( b : BIT_VECTOR        )
RETURN std_ulogic_vector;

FUNCTION To_StdULogicVector ( s : std_logic_vector  )
RETURN std_ulogic_vector;

-------------------------------------------------------------------
        -- strength strippers and type convertors
-------------------------------------------------------------------

FUNCTION To_X01  ( s : std_logic_vector  ) RETURN
std_logic_vector;
```

```
FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN
std_ulogic_vector;

FUNCTION To_X01  ( s : std_ulogic        ) RETURN  X01;

FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN
std_logic_vector;
FUNCTION To_X01  ( b : BIT_VECTOR        ) RETURN
std_ulogic_vector;
FUNCTION To_X01  ( b : BIT               ) RETURN  X01;

FUNCTION To_X01Z ( s : std_logic_vector  ) RETURN
std_logic_vector;

FUNCTION To_X01Z ( s : std_ulogic_vector ) RETURN
std_ulogic_vector;

FUNCTION To_X01Z ( s : std_ulogic        ) RETURN  X01Z;

FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN
std_logic_vector;

FUNCTION To_X01Z ( b : BIT_VECTOR        ) RETURN
std_ulogic_vector;

FUNCTION To_X01Z ( b : BIT               ) RETURN  X01Z;

FUNCTION To_UX01  ( s : std_logic_vector  ) RETURN
std_logic_vector;

FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN
std_ulogic_vector;

FUNCTION To_UX01  ( s : std_ulogic        ) RETURN
UX01;
```

```
FUNCTION To_UX01  ( b : BIT_VECTOR       ) RETURN std_logic_vector;
FUNCTION To_UX01  ( b : BIT_VECTOR       ) RETURN std_ulogic_vector;
FUNCTION To_UX01  ( b : BIT              ) RETURN  UX01;

    -------------------------------------------------------------------
                          -- Edge Detection
    -------------------------------------------------------------------
FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;

    -------------------------------------------------------------------
                    -- object contains an unknown
    -------------------------------------------------------------------
FUNCTION Is_X ( s : std_ulogic_vector ) RETURN BOOLEAN;
FUNCTION Is_X ( s : std_logic_vector  ) RETURN BOOLEAN;
FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN;
```

IEEE Library: "Numeric_Std" Package

- The *numeric_std* package is in the IEEE Library
  - defines the *unsigned* and *signed* types based on the *std_logic* type
  - Defines numeric operations such as +, -, *, /, abs, etc. for these types
- Use the numeric_std package when need to perform arithmetic operations (or synthesize arithmetic operators) on std_logic types

library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

Signed vs. Unsigned

- *Unsigned* is an unsigned binary integer with the MSB as the left-most bit.
- *Signed* is defined as a 2’s complement value with the most significant bit as the left-most bit.
- Need signed & unsigned types because arithmetic results of operations can be different depending on the types.

type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;

type SIGNED **is array** (NATURAL range <>) of STD_LOGIC;

Main Operations

- Abs, unary –
- +, -, *, / (division), rem, mod
- >, <, <=, >=, =, /=
- Shift_left, shift_right, rotate_left, rotate_right →

  Operate on Unsigned.
- XSll, xsrl, xsra, xrol, xror → *Shift & Rotate Ops*. For

  "Std_logic_vector"
- Resize Unsigned and signed to specified vector size
- To_integer, to_unsigned, to_signed
- Not, and, or, nand, nor, xor, xnor
- Std_match
- To_01

Metalogical & Z Values

- A *metalogical* value is defined as ‘X’, ‘W’, ‘U’, or ‘-’
- A high impedance value is ‘Z’
- If any bit in an operand to a **numeric_std** *function* contains a metalogical or high impedance value (‘Z’), the result is returned with all bits set to ‘X’
- One exception, the ‘std_match’ function
- A value is *well-defined* if it contains no metalogical or high impedance values.

Conversions

- Std_ulogic_vector, std_logic_vector, unsigned, signed are all closely related types (subtypes of std_ulogic).
- Use explicit *type casts* when assigning one type to another

<u>Example</u>

- **signal a_us, b_us: unsigned(7 downto 0);**
- **signal a_s,   b_s: signed(7 downto 0);**
- **signal a, b: std_logic_vector(7 downto 0);**
- **a <= std_logic_vector(a_s);**
- **a_s <= signed(a_us);**
- **a_us <= unsigned(a);**
- **b_s <= signed(b);**

Integer Conversion

function **To_Integer** (ARG: **UNSIGNED**) return **NATURAL**;
function **To_Integer** (ARG: **SIGNED**) return **INTEGER**;
function **To_Unsigned** (arg, size: **Natural**) return **Unsigned**;
function **To_Signed**(Arg: Integer; Size: **Natural**) Return **Signed**;

- Basically the same functions as in the std_logic_1164 package.

Different Forms of +

- function "+" (L, R: UNSIGNED) return UNSIGNED;
- function "+" (L, R: SIGNED) return SIGNED;
- function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
- function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
- function "+" (L: INTEGER; R: SIGNED) return SIGNED;
- function "+" (L: SIGNED; R: INTEGER) return SIGNED;

- Note different combinations of allowable operands.
- For synthesis, there is a problem – *do not have access to carry-in, or carry-out which would be very useful. Would have to use operands with 2-extra bits to get access to both carry-in and carry-out*.

Mixed Signed / Unsigned Operands

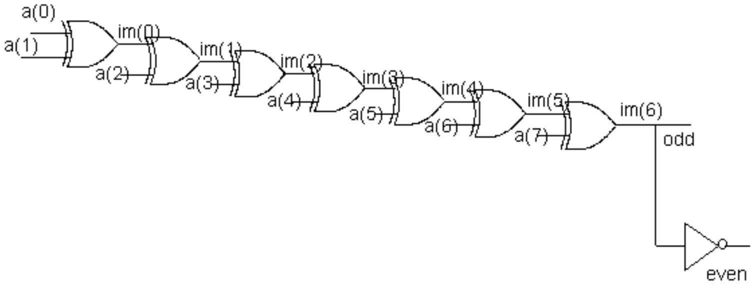
- The defined forms of '+' do not have mixed unsigned/signed operands
- Must do explicit conversions to perform mixed unsigned/signed operands
- This allows the user to decide how the sign bit is handled.

Resize

BR 1/02 10

```
function RESIZE (ARG: SIGNED; NEW_SIZE: natural)
return SIGNED;
-- Result subtype: SIGNED(NEW_SIZE-1 downto 0)

function RESIZE (ARG: UNSIGNED; New_Size: natural)
return UNSIGNED;
-- Result subtype: UNSIGNED(NEW_SIZE-1 downto 0)
```

- Changing size of an input vector larger than old, then sign extend the operand for signed numbers, else fill with zeros.
- If size decreases, for signed case keep sign bit but drop leftmost part. For unsigned case, just drop leftmost part.



```
ENTITY parity IS
     Port (A : In Bit_Vector (7 Downto 0); Odd, Even : Out Bit);
END parity;
--

ARCHITECTURE iterative OF parity IS

Component X2 Port (I1, I2: In Bit; O1: Out Bit); End Component;
Component N1 Port (I1: In Bit; O1: Out Bit); End Component;

SIGNAL im : BIT_VECTOR ( 0 TO 6 );

BEGIN
     first: x2 PORT MAP (a(0), a(1), im(0));
     middle: FOR i IN 1 TO 6 GENERATE
                    m: x2 PORT MAP (im(i-1), a(i+1), im(i));
               END GENERATE;
     last: odd <= im(6);
     inv: n1 PORT MAP (im(6), even);
END iterative;
```

```
CONFIGURATION parity_binding OF parity IS
FOR iterative
     FOR first : x2
          USE ENTITY WORK.xor2_t (average_delay)
          GENERIC MAP (5 NS, 5 NS);
     END FOR;
     FOR middle(1 TO 5)
          FOR m : x2
               USE ENTITY WORK.xor2_t (average_delay)
               GENERIC MAP (5 NS, 5 NS);
          END FOR;
     END FOR;
     FOR middle ( 6)
          FOR m : x2
               USE ENTITY WORK.xor2_t (average_delay)
               GENERIC MAP (6 NS, 7 NS);
          END FOR;
     END FOR;
     FOR inv : n1
          USE ENTITY WORK.inv_t (average_delay) GENERIC
          MAP (5 NS, 5 NS);
     END FOR;
END FOR;
END parity_binding;
```

- Due to fanout, last gate has a higher delay
- Element 6 of **Generate** statement specifies 6 NS, 7 NS delays
- Other generated elements use 5 NS, 5 NS
- **Generate** Index Value May be Used to Bind different instances of the **Generate** Statement to Different Entities/Architectures
- Can use **OTHERS** for indexing all other instantiations

Signal Resolution

and

Data Flow Models

Dr. Alaaeldin Amin

OutLine

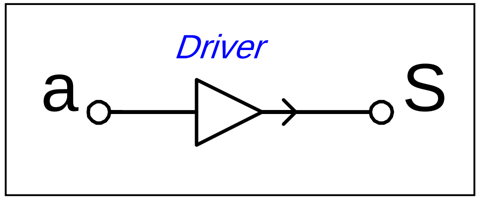
- Signal Resolution Function
- Declaring Resolved Signals
- Data Flow Models
  - * Signal Assignment Statement
  - * Block Construct
- Guard Condition & Guarded Signals
- Use of Nested Blocks & Example
- Resolution of Guarded/Non-Guarded Signals
  - * BUS & Register Signal Kinds
- Data Flow FSM Model & Examples
- Disconnection of BUS Signals

Signal Resolution Function

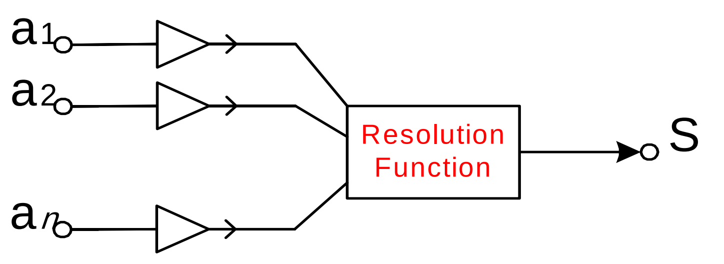
- Each Signal Assignment Statement Defines a Signal **Driver** (*Source*)

Example:

S <= a After T ;



- Multiple *Concurrent* Assignment Statements To The *Same Signal* Defines Multiple Drivers (*Signal Sources*).
- Such Multi-Driver Signals Are Commonly Encountered in Buses with Multiple Drivers
- Electrically, Tri-State or Open-Collector Drivers Are Used to ***Resolve*** ***Conflicts*** of the Different Drivers
- VHDL Model Requires the Definition of a ***Resolution Function*** ***To Resolve Values Being Assigned to the Common Signal By All Its Drivers***



Signal Resolution Function

- A User-Defined *Resolution Function* (RF) Must be Defined to Resolve the Final Value of Such Signals. Such Signals are Called *Resolved* Signals.
- AN Error is Reported whenever an RF is not Defined for Multi-Driver Signals.
- The RF is Invoked Each time any of the signal Drivers Receives a New Value.

RF Example

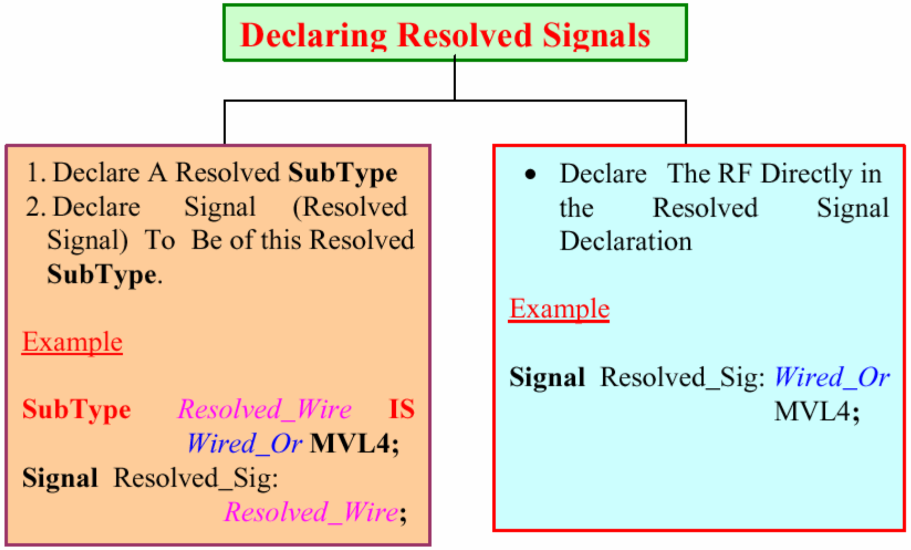
Input : 1-D *Unconstrained* Array of Values of the Type to be Resolved

Output (Returned Value): A Value of the *Resolved* Type.

```
Type MVL4 IS (` X `, `0`, `1`, `Z`);
Type MVL4_Vector IS Array (Natural Range <>) OF MVL4;

Function Wired_OR(Vin: MVL4_Vector) Return MVL4 IS
          Variable R: MVL4 :=`0`;  -- Initial Default Result
Begin
    For  I  IN Vin`Range
        Loop
           IF  Vin(i) = `1` Then  R:= `1`; Exit;
           Elsif Vin(i) = `X` Then  R:= `X`;
           Else Null;
           EndIF;
        End Loop;
Return R;
End Wired_Or
```

Notes

1. The RF Should Handle **Any #** of Signal Drivers. Thus The Input To RF Must Be an *Unconstrained Array*.
2. RF Should Not Depend on The *Order of Values in the Input Vector*. Thus, an Input (‘0’, ‘1’) Should Return Identical Value Like The Vector (‘1’, ‘0’).
3. Multiple Assignments To the Same Target Signal in A Process Body *Does Not Require an RF* Since a Process Body Is Sequential Not Concurrent.



Declaring Resolved Signals

1. Declare A Resolved **SubType**
2. Declare Signal (Resolved Signal) To Be of this Resolved **SubType**.

Example

SubType *Resolved_Wire* **IS** *Wired_Or* **MVL4;**
Signal Resolved_Sig: *Resolved_Wire*;

- Declare The RF Directly in the Resolved Signal Declaration

Example

Signal Resolved_Sig: *Wired_Or* MVL4**;**

DATA FLOW MODEL

- Represents Register Transfer operations
- There is Direct Mapping between Data Flow Statements && Register Structural Model
  - Implied Module Connectivity
  - Implied Muxes & Buses

Main Data Flow VHDL Constructs:

1. **Concurrent Signal Assignment Statements**
2. **Block Statement**

Signal Assignment

Unconditional → *Both Sequential & Concurrent*

Conditional *Only Concurrent* → *Conditions Must Be Boolean, May Overlap and Need Not Be Exhaustive*

Selected *Only Concurrent* → *Cases Must Not Overlap and Must Be Exhaustive*)

Conditional Signal Assignment

```
[ Label: ]  target <= [Guarded] [Transport ]
                 Wave1  when  Cond1  Else
                 Wave2  when  Cond2  Else
                 ……………………………..
                 Waven-1  when  Condn-1  Else
                 Waven  ; -- Mandatory Wave
```

Optional

Selected Signal Assignment

```
[ Label:]
With Expression  Select
          target <= [Guarded] [Transport]
                 Wave1  when  Choice1 ,
                 Wave2  when  Choice2 ,
                 ………………………….
                 Waven-1  when  Choicen-1 ,
                 Waven when  OTHERS ;
```

VHDL-93 **Any** *Wavei* **Can Be Replaced By the Keyword** **UNAFFECTED** (Which Doesn't Schedule Any Transactions on the Target Signal.)

Examples

Ex A 2x4 Decoder

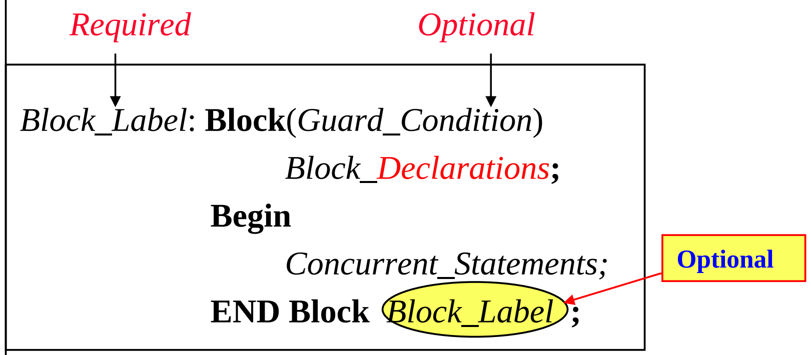
```
Signal D  :  Bit_Vector(1 To 4) := "0000";
Signal S0, S1  :  Bit;
…………………………………………
Decoder:   D <= "0001" after T  When S1='0' and S0='0' else
                "0010" after T  When S1='0' else
                "0100" after T  When S0='0' else
                 "1000" ;
```

Ex 4-Phase Clock Generator

```
Signal Phi4  :  Bit_Vector(1 To 4) := "0000";
…………………………………………
ClkGen:  With  Phi4  Select
           Phi4   <=  "1000" after T  When "0000" ,
                      "0100" after T  When "1000" ,
                      "0010" after T  When "0100" ,
                      "0001" after T  When "0010" ,
                      "1000" after T  When "0001" ,
                      "0000" When   Others  ; -- Exhaustive
```

Block Statement

- Block Statement is a **Concurrent** VHDL Construct Which is Used Within an Architectural Body to Group (Bind) a Set of **Concurrent** Statements.



Required

Optional

```
Block_Label: Block(Guard_Condition)
                 Block_Declarations;
         Begin
                 Concurrent_Statements;
         END Block  Block_Label ;
```

Optional

- A *Guard Condition* May be Associated with a Block Statement to Allow Enabling/Disabling of Certain Signal Assignment Statements.
- The Guard Condition Defines an *Implicit Signal* Called ***GUARD*.**
- In the Simplest Case, Binding (*Packing !*) Statements Within A **Block** Has No Effect On the Model.
- Blocks Can Be Nested.

Example :

```
Architecture DF of D_Latch is
 Begin
B : Block (Clk = '1')
     Signal  I_State :Bit;
      Begin
          I_State <=  Guarded  D ;
           Q     <= I_State after 5 ns;
           QB  <= not I_State after 5 ns;
      END  Block B ;
 END DF ;
```

Guard Condition

Block Local Signal

Notes

1. **If *Guard Condition* (Clk='1') is TRUE, *Guarded Statements* within block are Enabled (*Made Active*)**
2. **Drivers of Guarded Signals are Enabled when Guard condition is TRUE → New Signal Transaction is scheduled**
3. **Drivers of Guarded Signals are Turned OFF when Guard condition is False → No Signal Transaction is scheduled even if signals on RHS change value.**
4. **UnGuarded Signal Targets (e.g., Q, QB) are independent of the Guard Condition**

Examples

+ive Edge-Triggered D-FF

```
Entity DFF is
   Generic(TDel: Time:= 5 NS);
   Port(D, Clk: Bit; Q, QB: out Bit);
End DFF;
```

- We will show several dataflow architectures of D-FF with and without Block statement
- Will show why some of these architectures do not work

Works Fine

Arch 1

```
Architecture DF1_NO_Block of DFF is
Signal I_State: Bit:='1';
begin
   I_State <= D when (Clk='1' and
   Clk'Event) else I_state;
   Q <= I_state after TDel ;
   QB <= not I_state after TDel ;
End  DF1_NO_Block ;
```

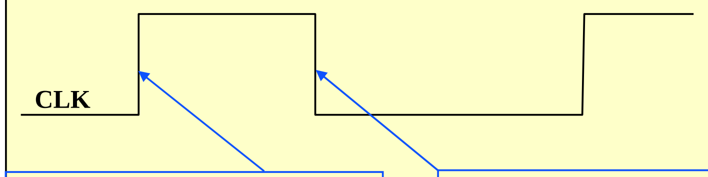
Examples

Works Fine

Arch 1

```
Architecture DF1_NO_Block of DFF is
Signal I_State: Bit:='1';
begin
   I_State <= D when (Clk='1' and
   Clk'Event) else I_state;
   Q <= I_state after TDel;
   QB <= not I_state after TDel;
End  DF1_NO_Block ;
```

Clk='1' and Clk'Event



Signal Evaluated here →
(Clk='1' and Clk'Event) = TRUE

Signal Evaluated here →
(Clk='1' and Clk'Event) = FALSE

Signal Evaluated 2-Times Per Clock Cycle

- Clk ↑True → Correct Value is scheduled on I_State
- Clk ↓False → Current Value to I_State

is scheduled on I_State (i.e., No Change)

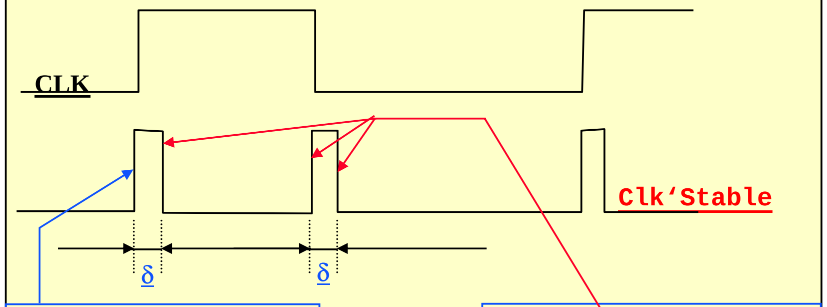
Examples

Doesn't Work

Arch 2

```
Architecture DF2_NO_Block of DFF is
Signal I_State: Bit:='1';
begin
   I_State <= D after TDel when (Clk='1'
   and (not(Clk'Stable))) else I_state;
   Q <= I_state;
   QB <= not I_state;
End  DF2_NO_Block ;
```

Clk='1' and not(Clk'Stable)



Signal Evaluated here →
(Clk='1' and not Clk'Stable)= TRUE

Signal Evaluated here →
(Clk='1' and not Clk'Stable)= FALSE

Signal Evaluated 4-Times Per Clock Cycle

- **Clk** ↑ True → correct value scheduled on I_State after TDel
- **Clk** ↑+ δ False → Wrong Value scheduled on I_State → Overwrites the Previously Scheduled Correct Value
- **Clk** ↓ False → Schedules the now incorrectly assigned value
- **Clk** ↓ + δ False → Recent assigned incorrect value

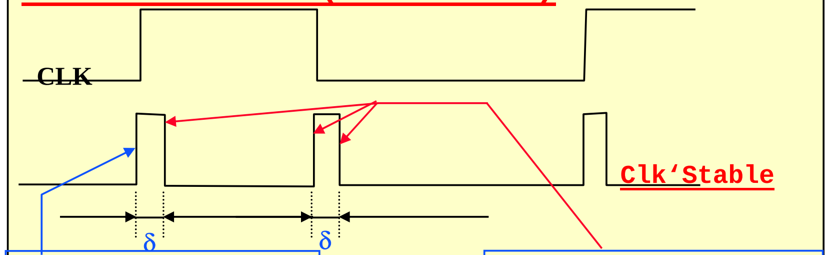
Examples

Works Fine

Arch 3

```
Architecture DF3_NO_Block of DFF is
Signal I_State: Bit:='1';
begin
   I_State <= D  when (Clk='1' and
   (not(Clk'Stable)) else I_state;
   Q <= I_state after TDel;
   QB <= not I_state after TDel;
End  DF3_NO_Block ;
```

Clk='1' and not(Clk'Stable)



Signal Evaluated here →

(Clk='1' and not Clk'Stable)= TRUE

Signal Evaluated here →

(Clk='1' and not Clk'Stable)= FALSE

Signal Evaluated 4-Times Per Clock Cycle

- **Clk** ↑ True → correct value scheduled on I_State after δ
- **Clk** ↑+ δ False → Just assigned Correct Value scheduled on I_State
- **Clk** ↓ False → Schedules the now correctly assigned value
- **Clk** ↓ + δ False → Recent assigned correct value rescheduled

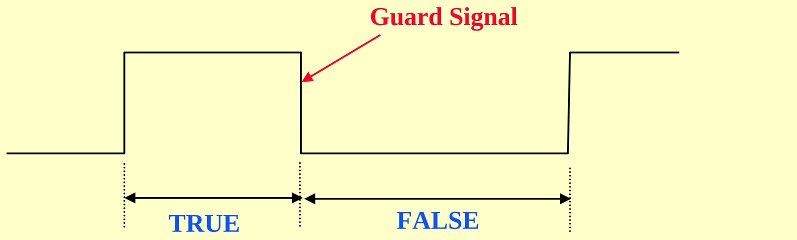
Examples

Doesn't Work

Arch4

```
Architecture DF1_Block of DFF is
Signal I_State: Bit:='1';
begin
 D_Blk: Block(Clk='1' and Clk'Event)
    Begin
   Q  <= Guarded D after Tdel;
   QB <= Guarded not D after Tdel;
    End Block;
End  DF1_Block ;
```

GUARD <= Clk='1' and Clk'Event



Signal Evaluated Continuously while Clk = '1' !!!

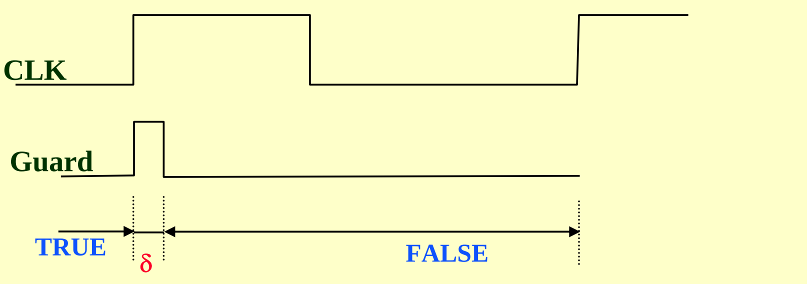
Examples

Works Fine

Arch5

```
Architecture DF2_Block of DFF is
Signal I_State: Bit:='1';
begin
 D_Blk: Block(Clk='1' and not
   Clk'Stable)
    Begin
   Q  <= Guarded D after Tdel;
   QB <= Guarded not D after Tdel;
    End Block;
End  DF2_Block ;
```

GUARD <= Clk='1' and not Clk‘Stable



Signal Evaluated Once Per Clock Cycle

(At Rising Edge of the Clock)

example "Nested Blocks"

```
Architecture  Block_Structure of   Demo  is
begin
A: Block -- 1
    Outer Block Declarative Section;
 Begin
     Concurrent Statements of Outer Block;
   B:Block -- 1.1
          Inner Block "A" Declarative Section;
      begin
          Concurrent Statements of Inner Block "A";
          ................................
      end Block B;
   C:Block -- 1.2
          Inner Block "B" Declarative Section;
      begin
          Concurrent Statements of Inner Block "B";
          ................................
      end Block C;
end Block A;
D: Block -- 2
          ................................
end Block D;
end   Block_Structure;
```

Use of Nested Blocks For Composite Enabling Conditions

```
ARCHITECTURE guarding OF DFF IS
  BEGIN
  edge: BLOCK ( CLK = '1'  and  not CLK 'STABLE )
  BEGIN
        gate: BLOCK ( En = '1' AND  GUARD )
        BEGIN
        q  <= GUARDED d AFTER delay1;
        qb <= GUARDED  NOT  d  AFTER delay2;
        END BLOCK gate;
  END BLOCK edge;
  END guarding;
```

- Blocks Can be Nested
- **Implicit *GUARD* signals** in each block
- **Combining guard expressions** must be done ***explicitly***
- Inner Guard Signal <= (En= '1') **AND** (CLK = '1' ***and not*** CLK 'STABLE)

EXAMPLE

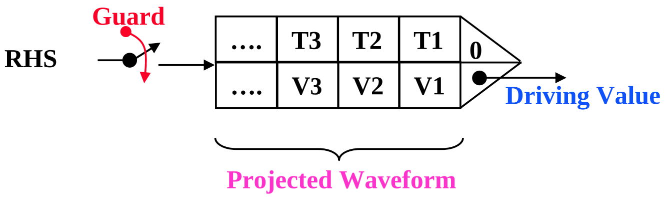
- **Model** A System with 2 **8-Bit Registers** R1 and R2, a 2-Bit Command signal “**COM**” and an external 8-Bit Input “**INP**”
- When Com= “00”→ R1 is Loaded with External Input
- When Com= “01”→ R2 is Loaded with External Input
- When Com= “10”→ R1 is Loaded with **R1+R2**
- When Com= “11”→ R1 is Loaded with **R1-R2**

```
Use Work.Utils_Pkg.ALL
Entity DF_Ex is
      Port (Clk: Bit; Com: Bit_Vector (1 DownTo 0);
           Input: Bit_vector(7 DownTo 0));
End DF_Ex;
--
Architecture DF of DF_Ex is
Signal  Mux_R1, R1, R2, R2C, R2TC, Mux_Add,
    Sum: Bit_Vector(7 DownTo 0);
Signal  D00, D01, D10, D11, LD_R1: Bit;
Begin
    D00 <= not Com(0) and not Com(1); -- Decoder
    D01 <= not Com(0) and Com(1); -- Decoder
    D10 <= Com(0) and not Com(1); -- Decoder
    D11 <= Com(0) and Com(1); -- Decoder
- -
```

EXAMPLE

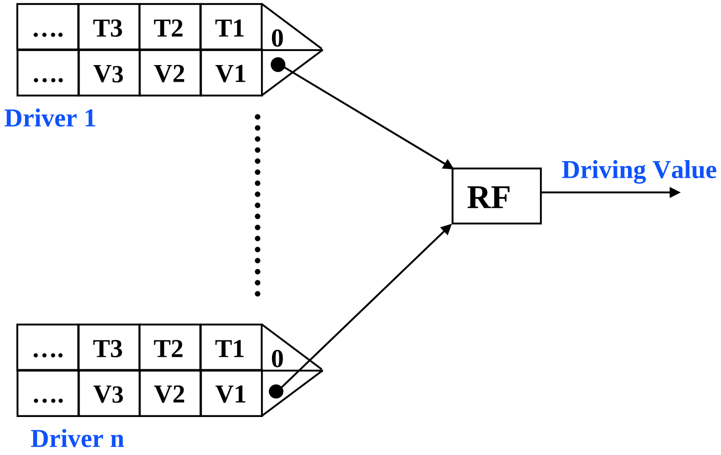
```
R2C  <= not R2;
R2TC <= INC(R2C); -- Increment Function Defined
                  --  in the Package
Mux_Add <=R2TC when D11 = ‘1’ Else
          R2 ;
Sum <= ADD(R1,  Mux_Add); -- ADD Function
                          -- Defined in Package
Mux_R1 <= INP when D00 = ‘1’ Else
          Sum;
R1E  <= D00 OR D10  OR D11;
Rising Edge: BLOCK(Clk=‘1’ and not Clk’Stable)
  R1_Reg: BLOCK(R1E=‘1’ AND GUARD)
        R1 <= Guarded Mux_R1 ;
  End Block R1_Reg ;

  R2_Reg: BLOCK(D01=‘1’ AND GUARD)
        R2 <= Guarded INP ;
  End Block R2_Reg ;
End Block Rising Edge;
```

Non-Resolved (1-Driver) & Guarded Signals



- (GUARD = False) → LHS Signal is ***Disconnected*** from its Driver Signals on the RHS
- *No New Transactions* May Be Placed on the LHS Signal Driver
- *Pending Transaction*s on the PWFM of the Signal Continue to Affect the Signal Value as they Expire.
- (GUARD = True) → LHS Signal is ***Connected*** to its Driver on the RHS
- *New Transactions* are Placed on the LHS Signal Driver as dictated by the RHS.

Resolution of Non-Guarded Signals (More than 1 Driver)



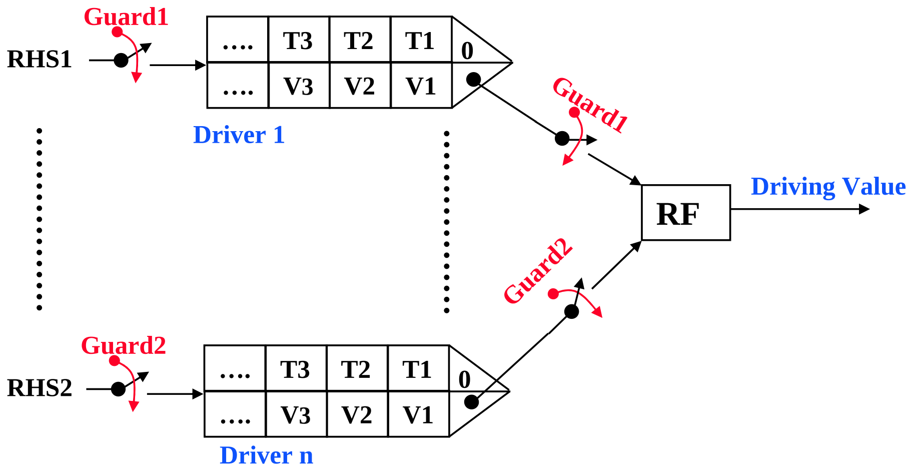
- Signal Drivers are *ALWAYS Enabled* (Connected)

•Resolved Non-Guarded LHS Signal Values Are Determined by ***The Resolution Function*** from CVs of **ALL** Driver Signals on the RHS

•*Expired Transactions* on **Any** of the Signal Driver, Activate the RF to Determine the new value of the output signal

•*Pending Transaction*s on the PWFM of the Signal Continue to Affect the Signal Value (Through RF) as they Expire.

Resolution of Guarded Signals



•**Only** **Enabled** Drivers (with GUARD = True) Participate in Determining Value of Target Signal (i.e. not *ALL* Drivers)

•If a Driver is **Disabled** (GUARD = False) It is Considered **Turned-Off (DISCONNECTED)**

• **Possible to have ALL Drivers Disconnected**

•A ***Resolved Guarded Signal*** Must be Declared of either ***REGISTER*** *kind* or ***BUS*** Kind.

•***Register Signal*** drivers *DO NOT Invoke* the *RF in Case All Drivers Are Turned Off* → Signal **Retains its Previous Value**.

•Signals of ***BUS*** Kind *Invoke the RF is in case All Signal Drivers Are Turned Off* → *RF is Invoked with a NULL input* → *Default Value is Returned*.

Types of resolved Signals

1. Non-Guarded Resolved Signals

- Resolved-value of the signal is determined by the RF based on all values of its drivers, i.e. *ALL DRIVERS ARE ACTIVE*.

2. Guarded Resolved Signals (Guarded Signal Assignments within Guarded Blocks)

- Resolved-value of the signal is determined by the RF based on all values of its ***CONNECTED*** drivers, i.e. *ONLY DRIVERS with TRUE Guard Conditions ARE Considered ACTIVE*.
- IF ALL Drivers are ***DISCONNECTED*** from the SIGNAL (*All Guard Conditions Are False*). VHDL allows ***3 possible scenarios*** in this case:
  - i. Signals of the "***BUS***" *kind*: Signal value is determined by the RF assuming a *NULL* input, i.e. the signal will assume *default* value of the RF.
  - ii. Signals of the "***REGISTER***" *kind*: No CALL is made to the RF upon the Disconnection, i.e. the signal maintains its last value before disconnection.
  - iii. Signal with NO specific ***kind***: In this case, a ***FALSE GUARD VALUEs DO NOT DISCONNECT*** the signal Driver and the signal value is determined by the RF assuming *ALL DRIVERS TO BE ACTIVE*. ***This is not recommended for use***.

Syntax

Signal *<sig_name> : <resolved sig_subtype> [kind] [:=Initial_Value]* **;**

Signal_kind **::= BUS | Register**

Examples:

Signal x : Wired_MVL4 *BUS* ;

Signal y : Wired_MVL4 *Register* ;

Note:

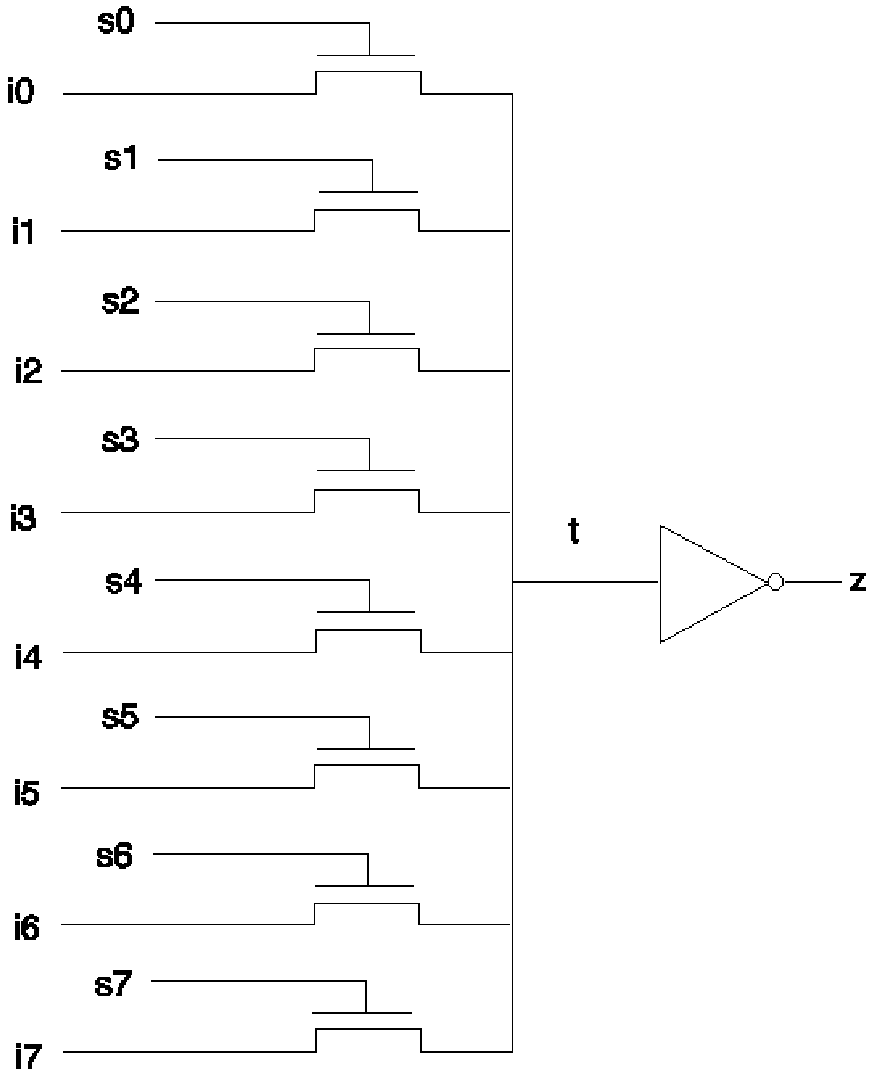
1. Only Signals of Kind BUS May be Specified as Port Signals

2. Signals of Register Kind May NOT be Specified as Port Signals)

Example

```
Entity ex is
        Port( s1, s2 : in MVL4;
              Z: out  wired_MVL4  BUS) ;
End ex;
```

Example MOS (PTL) Multiplexer

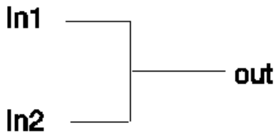


Example MOS (PTL) Multiplexer

```
TYPE  qit  IS  (‘0’ , ‘1’ , ‘Z’ , ‘X’);
Type qit_2d is Array (qit, qit) of qit;
Type qit_Vector is Array (Natural Range <>) of qit;
FUNCTION wire (a, b : qit) RETURN qit IS
CONSTANT qit_and_table : qit_2d := (
                                   ('0','X','0','X'),
                                   ('X','1','1','X'),
                                   ('0','1','Z','X'),
                                   ('X','X','X','X'));
BEGIN
          RETURN qit_and_table (a, b);
END wire;
```

	In1:	0	1	Z	X
In2:	0	0	X	0	X
	1	X	1	1	X
	Z	0	1	Z	X
	X	X	X	X	X

Out



Example MOS (PTL) Multiplexer

```
FUNCTION wiring ( drivers : qit_vector) Return qit IS
Variable accumulate : qit := 'Z'; -- Default
BEGIN
FOR i IN drivers'RANGE LOOP
          accumulate := wire (accumulate, drivers(i));
END LOOP;
          RETURN accumulate;
END wiring;
```

```
SUBTYPE wired_qit IS wiring qit;
TYPE wired_qit_vector IS Array (Natural Range <>)
OF wired_qit;
```

Example MOS (PTL) Multiplexer
Model 1 (BUS Signal Kind)

```
USE WORK.basic_utilities.ALL;
   -- FROM PACKAGE USE: wired_qit
Architecture multiple_guarded_assignments OF
     mux_8_to_1 IS
SIGNAL t : Wired_qit  BUS;
BEGIN
b7: Block (s7 = '1' ) Begin t <= Guarded  i7; End Block;
b6: Block (s6 = '1') Begin t <= Guarded i6; End Block ;
b5: Block (s5 = '1') Begin t <= Guarded i5; End Block ;
b4: Block (s4 = '1') Begin t <= Guarded i4; End Block ;
b3: Block (s3 = '1') Begin t <= Guarded i3; End Block ;
b2: Block (s2 = '1') Begin t <= Guarded i2; End Block ;
b1: Block (s1 = '1') Begin t <= Guarded i1; End Block ;
b0: Block (s0 = '1') Begin t <= Guarded i0; End Block ;
--
 z <= not t after 1 NS;
 END multiple_guarded_assignments;
```

- **Disconnection** is realized by **block** statements
- *If all drivers are disconnected*

  Hardware returns to 'Z’ → Modeling This Requires Using **BUS** Signal Kind.

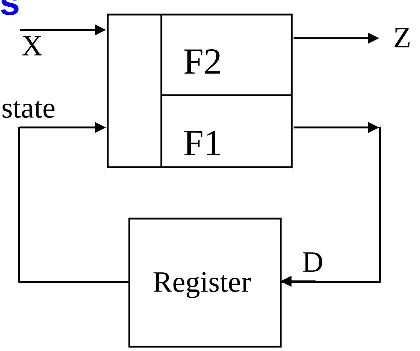
Example MOS (PTL) Multiplexer
Model 2 (Register Signal Kind)

```
USE WORK.basic_utilities.ALL;
    -- FROM PACKAGE USE: wired_qit
Architecture multiple_guarded_assignments OF mux_8_to_1 IS
SIGNAL t : Wired_qit  REGISTER ;
BEGIN
b7: Block (s7 = '1' ) Begin t <= Guarded  i7; End Block;
b6: Block (s6 = '1' ) Begin t <= Guarded i6; End Block ;
b5: Block (s5 = '1' ) Begin t <= Guarded i5; End Block ;
b4: Block (s4 = '1' ) Begin t <= Guarded i4; End Block ;
b3: Block (s3 = '1' ) Begin t <= Guarded i3; End Block ;
b2: Block (s2 = '1' ) Begin t <= Guarded i2; End Block ;
b1: Block (s1 = '1' ) Begin t <= Guarded i1; End Block ;
b0: Block (s0 = '1' ) Begin t <= Guarded i0; End Block ;
--
 z <= not t after 1 NS;
 END multiple_guarded_assignments;
```

- **Disconnection** is realized by **block** statements
- *If all drivers are disconnected* Real hardware Maintains State for few milliseconds (As Charge on the Capacitance of Node “t”.
- Use **Register** to implement this behavior

Mealy Machine Example Using Block Statements

```
entity Mealy_Mc is
    Port(Clk, X: in  Bit;
            Z : out Bit);
end Mealy_Mc;
```



```
Architecture Mealy_Block of  Mealy_Mc is
   Type State is (St0, St1, St2);
   Type St_Vector is array (Natural range <>) of State ;
   Function  State_RF (Signal X: St_Vector ) Return
         State is
    Begin
       Return X(X'Left);
    End State_RF  ;
   Signal Pstate: State_RF State REGISTER := St0;
Begin
  B1: Block(not Clk’STABLE  and  Clk = ‘1’)
      begin
        S0:Block((Pstate = St0) and Guard)
             begin
                Pstate <= Guarded St1 when X='0'
                                             else St2;
             end block S0;
```

Guard Signal

```
        S1:Block((Pstate = St1) and Guard)
              begin
                  Pstate <= Guarded St2 when X='0'
                                               else St0;
              end block S1;

        S2:Block((Pstate = St2) and Guard)
              begin
                  Pstate <= Guarded St1 when X='1'
                                               else St2;
              end block S2;
End Block B1;

    Z <= '1' when Pstate =St1 and X='0' else
         '0' when Pstate =St1 and X='1' else
         '0' when Pstate =St2 and X='0' else
         '1' when Pstate =St2 and X='1' else
         '0' ;
 End Mealy_Block;
```

Notes:

- Since there are 3 concurrent Signal assignments to the Signal ***Pstate*** , it is declared as a Resolved Signal with the RF being "*State_RF*".

- Signal ***Pstate*** is also declared to be of ***REGISTER*** *kind*. This Means that the Signal is ***Guarded*** and ***Resolved*** and that the *RF is not Invoked in Case All its Drivers Are Turned Off* (e.g. when CLK = ‘0’) in which case the Signal **Retains its Previous Value**.

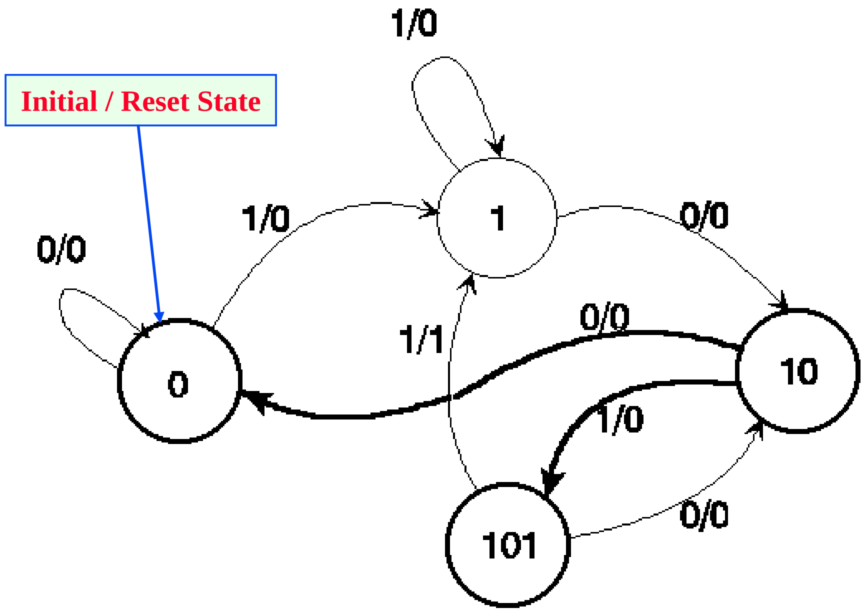
- The Outer Block Statement “*B1*” Defines an *IMPLICIT* **Guard Signal** Which is TRUE only On the Rising Edge of the Clock.

- The Implicit **Guard** Signal ANDed with the Present State Define the Guard Condition for the Nested Block Statements.

- ONE Inner Block Statement is Assigned to Each Possible Present State

- The State Machine Model Used Allows ***only One Driver*** *of the Resolved Signal Pstate to be* ***Active*** at any Given Time. Thus the **‘Left** Attribute is Used in the RF to Derive the Signal Value Forced By this Driver.

Sequence Detector Example Overlapped Detection of the Sequence "1011"



- A simple 1011 Mealy Sequence Detector
- Single Input x and A single Output z
- For x= 011011011011110111

  z= 000001001001000010

```
Entity detector IS
        PORT (x, clk : IN Bit; z : out Bit);
END detector;
```

```
Architecture Singular_state_machine OF Detector IS
TYPE State IS (Reset, Got1, Got10, Got101);
Type State_vector Is Array (Natural Range <>) Of State;
Function One_of (Sources : State_vector) Return State Is
BEGIN
        RETURN Sources(Sources'Left);
End One_of;
Signal Pstate : One_of  State  Register := Reset;
Begin
Clocking : BLOCK (Clk = '1' AND NOT Clk'Stable)
Begin
S1: BLOCK ( Pstate = Reset AND GUARD )
BEGIN
Pstate <= GUARDED Got1 When X = '1' Else Reset;
End Block S1;
S2: Block ( Pstate = Got1 And Guard )
Begin
Pstate <= GUARDED Got10 When X = '0' Else Got1;
End Block S2;
S3: Block ( Pstate = Got10 And Guard )
Begin
Pstate <= Guarded Got101 When X = '1' Else Reset;
End Block S3;
```

```
S4: Block ( Pstate = Got101 And Guard)
Begin
Pstate <= Guarded Got1 When X = '1' Else Got10;
End Block S4;
End Block Clocking;
--
Z <= '1' When ( Pstate = Got101 And X = '1') Else '0';
--
End Singular_state_machine;
```

- **Pstate receives four concurrent assignments**
- **Pstate must be resolved; use *one_of* as an RF**

Multiplier Design

Design a Multiplier Circuit Which Multiplies 2 Unsigned n-Bit Numbers **A** (*multiplicand*) & **B** (*multiplier*).

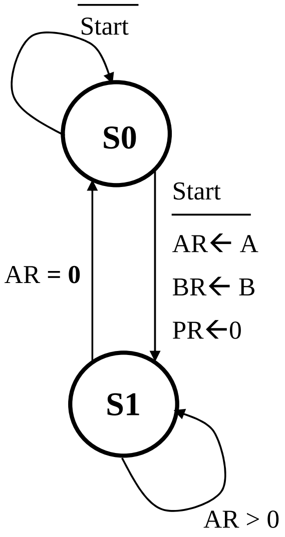
The Product (**P**) is Evaluated by Repeated Additions of the Multiplicand (**B)** to itself a Number of Times Equals the Multiplier (**A)** Value.

Example

1. A=3, B=4 → **P** = 4 + 4 +4
2. A=0, B=4 → **P** = 0
3. A=3, B=0 → **P** = 0 + 0 + 0

Required Data Path Modules:

1. A-Register (n-Bits) → AR
2. B-Register (n-Bits) → BR
3. P-Register (*2n*-Bits) → PR
4. Adder

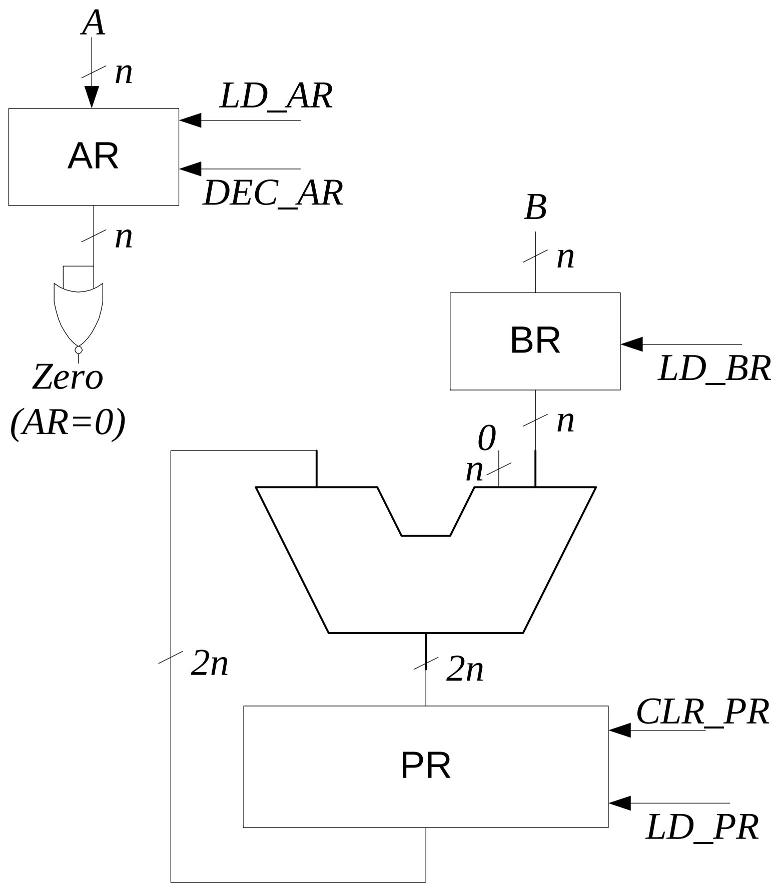


Controller Model

```
Architecture DF of CPath_Mult is
Type States is (Initial, Iterative);
Type State_Vector is Array (Natural Range <>) of States;
Function RF(V:State_Vector) Return States is
Begin
        Return V(V'Left);
end RF;
--_______________________________________________
Signal Pstate:  RF States Register := Initial;
Begin
edge: Block(Clk='1'  and  not Clk'Stable)
  Begin
  S0: Block(Pstate= Initial and Guard)
     Begin
     Pstate <= Guarded Iterative when Start='1' Else  Initial;
     end Block S0;
```

```
  S1: Block(Pstate= Iterative and Guard)
     Begin
       Pstate <= Guarded Iterative when Zero /='1' Else Initial;
     end Block S1;
--        --------------
    LD_AR  <= '1' when Pstate= Initial and Start='1' else '0';
    LD_BR  <= '1' when Pstate= Initial and Start='1' else '0';
    Clr_PR <= '1' when Pstate= Initial and Start='1' else '0';
    LD_PR  <= '1' when Pstate=Iterative and Zero /= '1' else '0';
    DEC_AR <= '1' when Pstate=Iterative and Zero /= '1' else '0';
End Block edge;
End  DF;
```

Data Path Design



Data Path Model

```
Entity DPath_Mult is
Generic(N: Positive:= 8);
Port(LD_AR, LD_BR, CLR_PR, LD_PR,Dec_AR, Clk: Bit ;
A, B: in Bit_Vector(N-1 DownTo 0);    Zero: out Bit :='0';
P: out Bit _Vector(2*N-1 DownTo 0));
End DPath_Mult ;
-- ---------------------
Architecture DF of DPath_Mult is
Signal AR, BR       : Bit _Vector(N-1 DownTo 0);
Signal PR           : Bit _Vector(2*N-1 DownTo 0);
Signal ARE,BRE,PRE : Boolean:=False ;
--  --------------------
Begin
ARE <= LD_AR='1' or DEC_AR='1' ;
BRE <= LD_BR='1' ;     -- Inner Block (Register) Enable Signals
PRE <= LD_PR='1' or CLR_PR='1' ;
--
```

Data Path Model

```
edge: Block(Clk='1' and not Clk'Stable)
Begin
AReg: Block(ARE and Guard)
  Begin
  AR <= Guarded A when LD_AR='1' Else
          Int2Bin(Int_Val(AR)-1 , AR'Length)
                   when  Int_Val(AR)> 0 else
          Unaffected;
  Zero <= '1' when (Int_Val(AR)=0) else '0';
 end Block AReg;
BReg: Block(BRE and Guard)
          Begin
                 BR <= Guarded B;
          end Block BReg;
PReg: Block(PRE and Guard)
 Begin
   PR <= Guarded Int2Bin((Int_Val(PR)+ Int_Val(BR)),
          PR'Length) when LD_PR='1' Else Int2Bin(0, 2*N);
 end Block PReg;
End Block edge;
          P <= PR ;
End  DF;
```

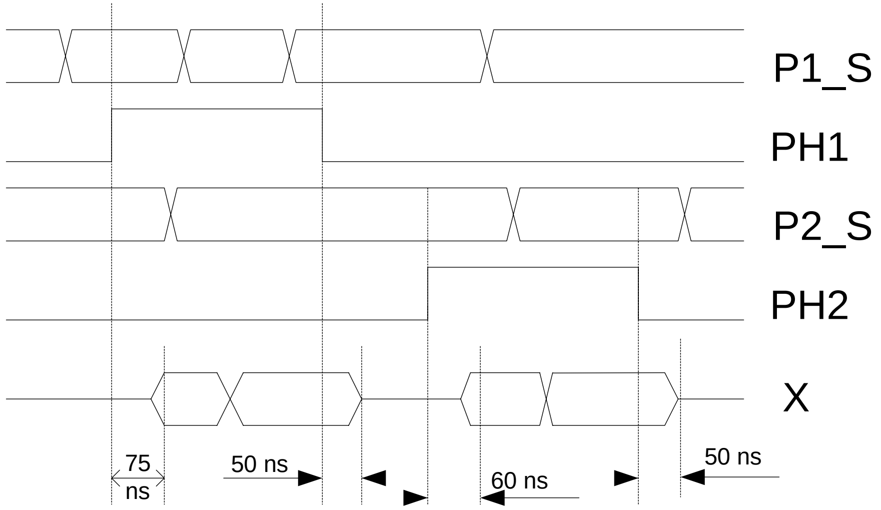
Disconnection of BUS Signals

- *Guarded Resolved Signal assignment can specify disconnection delay. The DISCONNECTION statement is placed in the declarative part of the Architecture and applies to all assignments to this signal. Architecture DF of Ex is*

EXAMPLE

```
Architecture  DF of  Example is
Signal            X  : WX_Vector(7 downTo 0) BUS ;
DISCONNECT  X  : WX_Vector  after  50 ns ;

Begin
 B1: Block(Ph1='1')
          Signal  P1_S :  WX_Vector(7 downTo 0)  ;
   Begin
          P1_S <= ….
          X <= Guarded  P1_S after 75 ns;
   End Block B1   ;

 B2: Block(Ph2='1')
          Signal  P2_S :  WX_Vector(7 downTo 0)  ;
   Begin
          P2_S <= ….
          X <= Guarded  P2_S after 60 ns;
   End Block B2  ;
END DF ;
```

Disconnection of BUS Signals



Example
"Register Signals"
+ive Edge-Triggered Shift Register with Parallel Load

Register INPUTS In Order of Priority

1. **Ena** : If Ena=0, The register Cannot not Change its state.
2. **LD** : IF LD = 1, Data on the parallel inputs (Din) are Loaded into the Register independent of the Clock Signal (Asynchronous Load)
3. **Dir** : Determines the Direction of the Shift or Rotate Operation. Dir=0 indicates a Left shift/Rotate while Dir = 1, indicates a Right Shift /Rotate.
4. Shift Mode Signals **M1 & M2**

   M1M2 : 00 A 0 is Shifted-In

   M1M2 : 01 A 1 is Shifted-In

   M1M2 : 10 The **Sin** input is Shifted-In

   M1M2 : 11 Rotate Operation.

Example

```
Type  MVL4 is (‘X’, ‘0’, ‘1’, ‘Z’);

Type MVL4_Vec is Array(Natural range <>) of MVL4 ;

Type MVL4_Tab is array(MVL4 , MVL4) of MVL4;
```

```
Constant Tab_X  :   MVL4_Tab  :=
--                                   'x', '0', '1', 'Z'
                                   ------------------------
                                   (('x', 'x', 'x', 'x'),  -- 'x'
                                    ('x', '0', 'x', '0'),  -- '0'
                                    ('x', 'x', '1', '1'),  -- '1'
                                    ('x', '0', '1', 'z')); -- 'z'
```

```
Function WiredX (INP : MVL4_Vec) Return MVL4 is
Variable Result: MVL4:='z';-- Initialize
Begin
   For i in INP'Range Loop
     Result:= TAB_X(Result , INP(i));
   End Loop;
   Return Result;
end  WiredX ;
```

```
SubType WX is  WiredX   MVL4 ;

Type WX_Vector is Array(Natural range <>) of WX ;
```

RF Function

```
Entity ShiftReg is
    Port ( Ena, Ld, Clk, Dir, M1, M2 : in Bit;
           Sin : in MVL4 ;
           Din : in WX_Vector(7 downto 0);
           Q   : Out WX_Vector(7 downto 0));
END ShiftReg ;
```

```
Architecture Wrong_DF of ShiftReg is
    Signal I_State : WX_Vector(7 downto 0);  -- Guarded Resolved
--                                 Should have its “Kind” Specified
Begin
Load: Block(Ena=‘1’ and Ld=‘1’)
          begin
             I_State <= Guarded Din;
          end block Load;
Shift: Block(Ena=‘1’ and Ld=‘0’ and Clk=‘1’ and not
                          Clk'Stable))
          Signal tmp: Bit_Vector() to 2);
          begin
             tmp <= Dir & M1 & M2 ;
             With tmp Select
              I_State <= Guarded
                  I_State(6 downto 0) & ‘0’ When “000” ,
                  I_State(6 downto 0) & ‘1’ When “001” ,
                  I_State(6 downto 0) & Sin When “010” ,
                  I_State(6 downto 0) & I_State(7) When “011” ,
                  ‘0’ & I_State(7 downto 1)  When “100” ,
                  ‘1’ & I_State(7 downto 1)  When “101” ,
                  Sin & I_State(7 downto 1)  When “110” ,
                  I_State(0)& I_State(7 downto 1)  When “111”;
          end block Shift;
          Q <= I_State After 5 ns ;
End Wrong_DF ;
```

```
Architecture Correct_DF of ShiftReg is
    Signal I_State : WX_Vector(7 downto 0) Register; -- Guarded
--                                          Resolved with “Kind”= Register
Begi
Load: Block(Ena=‘1’ and Ld=‘1’)
              begin
                  I_State <= Guarded Din;
              end block Load;
Shift: Block(Ena=‘1’ and Ld=‘0’ and Clk=‘1’ and not
                                            Clk'Stable))
     Signal tmp: Bit_Vector(0 to 2);
     begin
         tmp <= Dir & M1 & M2 ;
         With tmp Select
                  I_State <= Guarded
                        I_State(6 downto 0) & ‘0’ When “000” ,
                        I_State(6 downto 0) & ‘1’ When “001” ,
                        I_State(6 downto 0) & Sin When “010” ,
                        I_State(6 downto 0) & I_State(7) When “011” ,
                        ‘0’ & I_State(7 downto 1)  When “100” ,
                        ‘1’ & I_State(7 downto 1)  When “101” ,
                        Sin & I_State(7 downto 1)  When “110” ,
                        I_State(0)& I_State(7 downto 1)  When “111”;
              end block Shift;
              Q <= I_State After 5 ns ;
End Correct _DF ;
```

COE 405
VHDL Coding for Synthesis

Dr. Alaaeldin A. Amin
Computer Engineering Department

E-mail: amin@ccse.kfupm.edu.sa

Home Page : http://www.ccse.kfupm.edu.sa/~amin

Outline…

- Synthesis overview
- Synthesis of primary VHDL constructs
- Combinational circuit synthesis
  - Multiplexor, Decoder, Priority encoder, Adder, Tri-state buffer, Bi-directional buffer
- Sequential circuit synthesis
  - Latch
  - Flip-flop with asynchronous reset
  - Flip-flop with synchronous reset
  - Loadable register
  - Shift register
  - Register with tri-state output
  - Finite state machine
- Efficient coding styles for synthesis

General Overview of Synthesis…

- Synthesis is the process of translating from an abstract description of a hardware device into an optimized, technology specific implementation.
- Synthesis may occur at many different levels of abstraction
  - Behavioral synthesis
  - Register Transfer Level (RTL) synthesis
  - Boolean equations descriptions, netlists, block diagrams, truth tables, state tables, etc.
- RTL synthesis implements the register usage, the data flow, the control flow, and the machine states as defined by the syntax & semantics of the HDL.

…General Overview of Synthesis

Factors Affecting the synthesis algorithm

- HDL coding style
  - Should be technology independent.
  - Determines the initial starting point for the synthesis algorithms & plays a key role in generating the final synthesized hardware.
- Design constraints
  - Timing goals
  - Area goals
  - Power management goals
  - Design-For-Test rules
- Target technology
  - Target library design rules

VHDL Synthesis Subset

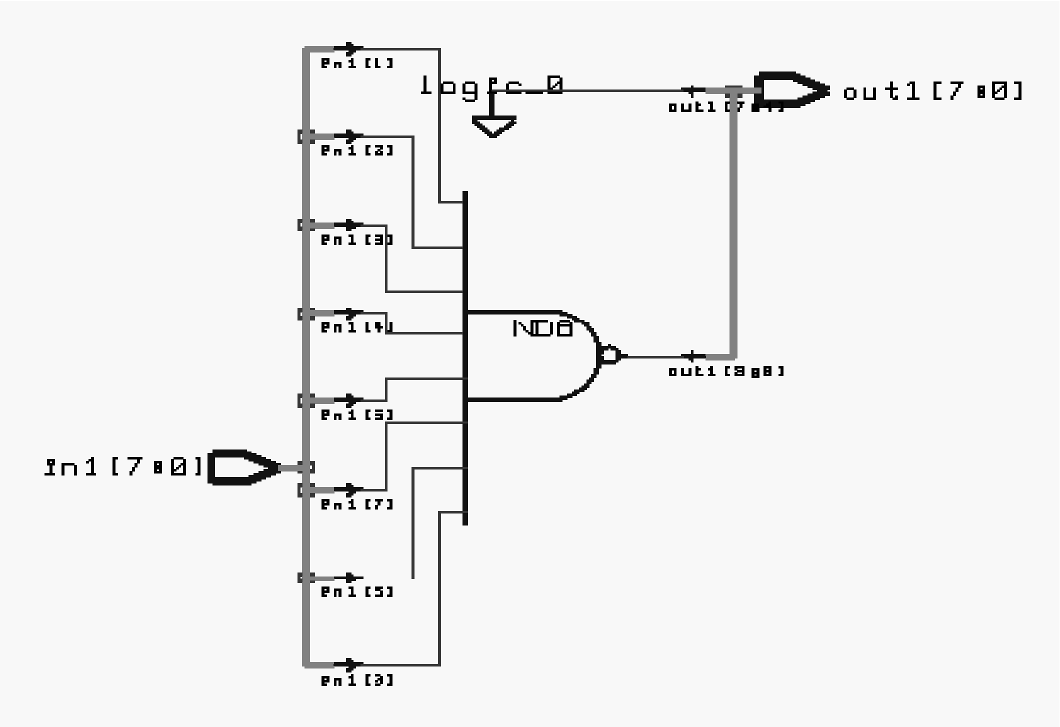
- VHDL is a complex language but only a subset of it is synthesizable.
- Primary VDHL constructs used for synthesis:
  - Constant definition
  - Port map statement
  - Signal assignment: A <= B
  - Comparisons: = (equal), /= (not equal), > (greater than), < (less than), >= (greater than or equal), <= (less than or equal)
  - Logical operators: **AND**, **OR**, **NAND**, **NOR**, **XOR**, **XNOR**, **NOT**
  - 'if' statement
    - **if** (presentstate = CHECK_CAR) **then** ....
      end if | **elsif** ....
  - 'for' statement (used for looping in creating arrays of elements)
  - Other constructs are '**with**', '**when**', '**when else**', '**case**' , '**wait** '. Also "**:=**" for variable assignment.

Synthesis of Combinational Logic

Constant Definition...

```
library ieee;
use ieee.std_logic_1164.all;
entity constant_ex is
    port (in1 : in std_logic_vector (7 downto 0);
            out1 : out std_logic_vector (7 downto 0));
end constant_ex;
 architecture constant_ex_a of constant_ex is
    constant A : std_logic_vector (7 downto 0) := "00000000";
     constant B : std_logic_vector (7 downto 0) := "11111111";
    constant C : std_logic_vector (7 downto 0) := "00001111";
 begin
    out1 <= A when in1 = B else C;
 end constant_ex_a;
```

…Constant Definition



Port Map Statement…

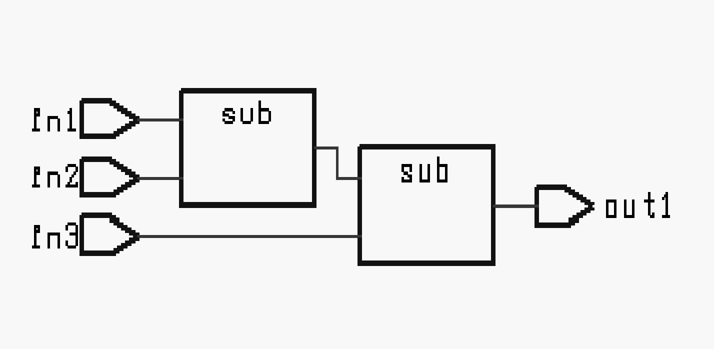
```
library ieee;
 use ieee.std_logic_1164.all;
entity sub is
   port (a, b : in std_logic; c : out std_logic);
 end sub;
architecture sub_a of sub is
begin
   c <= a and b;
end sub_a;
```

…Port Map Statement…

```
library ieee;
use ieee.std_logic_1164.all;
entity portmap_ex is
   port (in1, in2, in3 : in std_logic; out1 : out std_logic);
end portmap_ex;
architecture portmap_ex_a of portmap_ex is
   component sub
         port (a, b : in std_logic; c : out std_logic);
   end component;
   signal temp : std_logic;
```

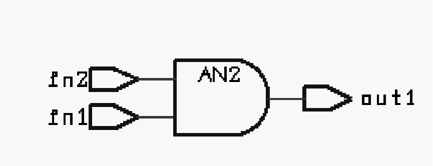
…Port Map Statement…

```
begin
   u0 : sub port map (in1, in2, temp);
   u1 : sub port map (temp, in3, out1);
end portmap_ex_a;
use work.all;
configuration portmap_ex_c of  portmap_ex is
    for portmap_ex_a
         for u0,u1 : sub use entity sub (sub_a);
         end for;
    end for;
end portmap_ex_c;
```



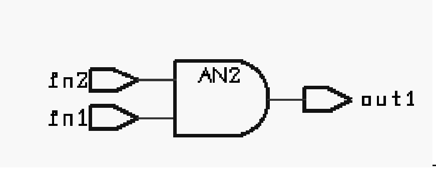
When Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity when_ex is
   port (in1, in2 : in std_logic; out1 : out std_logic);
end when_ex;
architecture when_ex_a of when_ex is
begin
   out1 <= '1' when in1 = '1' and in2 = '1' else '0';
end when_ex_a;
```



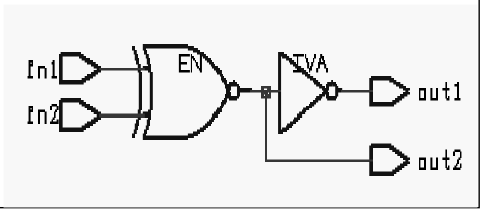
With Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity with_ex is
   port (in1, in2 : in std_logic; out1 : out std_logic);
end with_ex;
architecture with_ex_a of with_ex is
begin
   with in1 select out1 <= in2 when '1',
                                        '0' when others;
end with_ex_a;
```



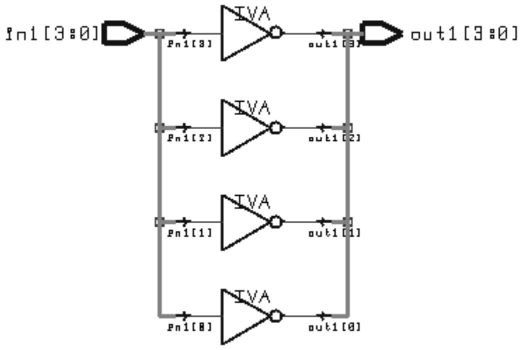
Case Statement…

```
library ieee;
use ieee.std_logic_1164.all;
entity case_ex is
     port (in1, in2 : in std_logic; out1,out2 : out std_logic);
end case_ex;
architecture case_ex_a of case_ex is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
       case b is
            when "00"|"11" => out1 <= '0'; out2 <= '1';
            when others => out1 <= '1'; out2 <= '0';
       end case;
    end process;
     b <= in1 & in2;
end case_ex_a;
```



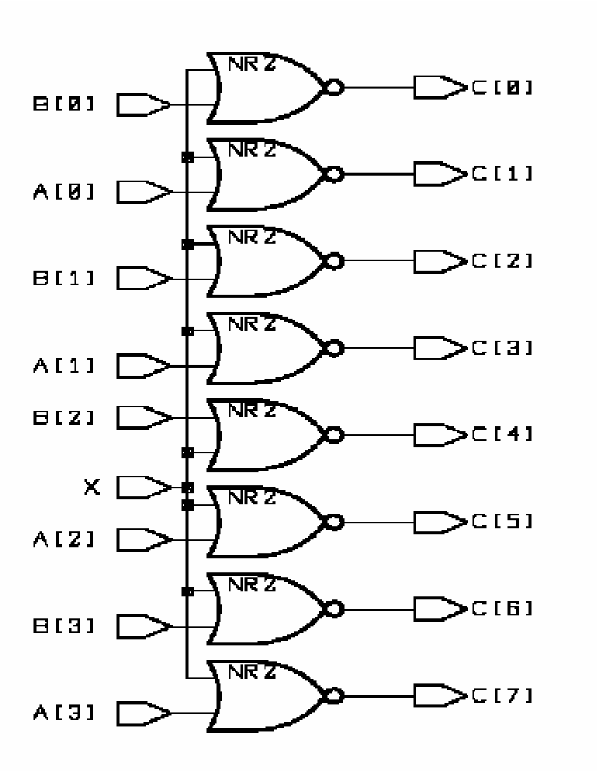
For Statement…

```
library ieee;
use ieee.std_logic_1164.all;
entity for_ex is
    port (in1 : in std_logic_vector (3 downto 0); out1 : out
    std_logic_vector (3 downto 0));
end for_ex;
architecture for_ex_a of for_ex is
begin
    process (in1)
    begin
       for0 : for i in 0 to 3 loop
                    out1 (i) <= not in1(i);
                end loop;
     end process;
end for_ex_a;
```



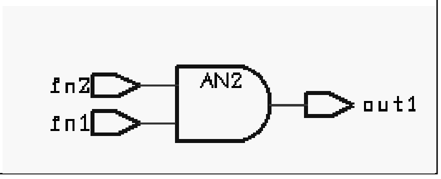
Generate Statement

```
Signal A,B: Bit_Vector (3 Downto 0);
Signal C: Bit_Vector(7 Downto 0);
Signal X:BIT;

. . .

Gen_label:
    For I In 3 Downto 0 Generate
            C(2*I+1) <= A(I) Nor X;
            C(2*I) <= B(I) Nor X;
    End Generate Gen_label ;
```



If Statement

```
library ieee;
use ieee.std_logic_1164.all;
entity if_ex is
    port (in1, in2 : in std_logic; out1 : out std_logic);
end if_ex;
architecture if_ex_a of if_ex is
    begin
          process (in1, in2)
          begin
                    if in1 = '1' and in2 = '1' then out1 <= '1';
                    else out1 <= '0';
                    end if;
           end process;
end if_ex_a;
```

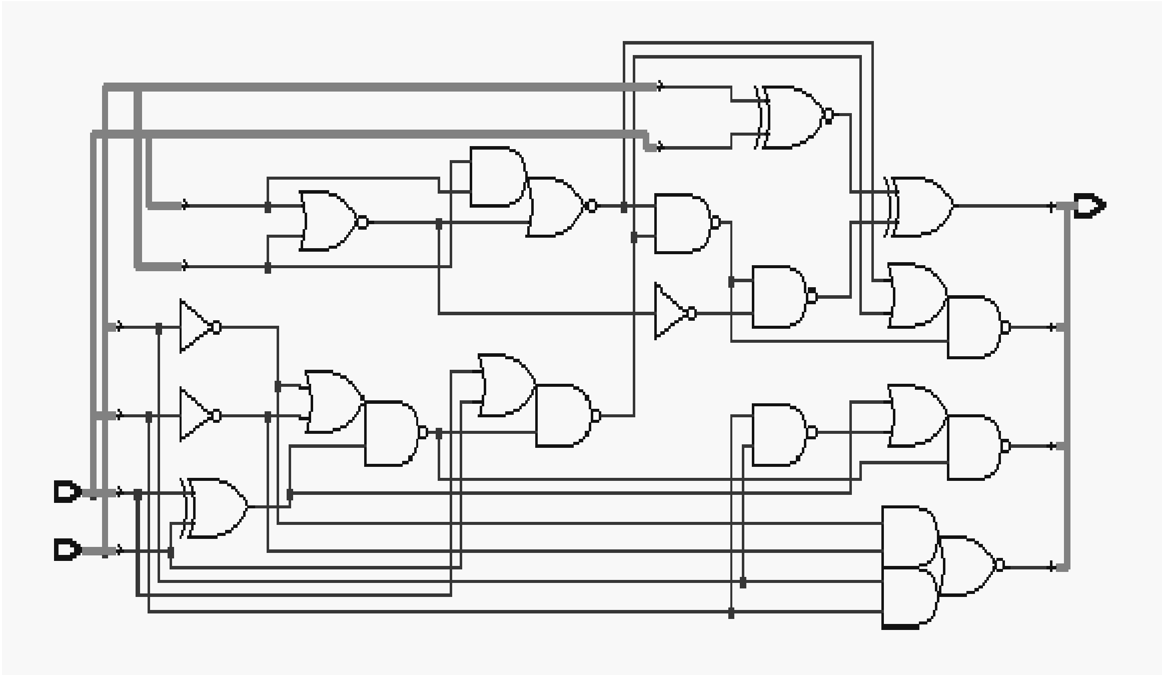


Variable Definition…

```
library ieee;
use ieee.std_logic_1164.all;
entity variable_ex is
   port ( a : in std_logic_vector (3
     downto 0); b : in std_logic_vector (3
     downto 0); c : out std_logic_vector (3
     downto 0));
end variable_ex;
architecture variable_ex_a of variable_ex
     is
begin
     process (a,b)
           variable carry : std_logic_vector
                                  (4 downto 0);
           variable sum  : std_logic_vector
                                  (3 downto 0);
```

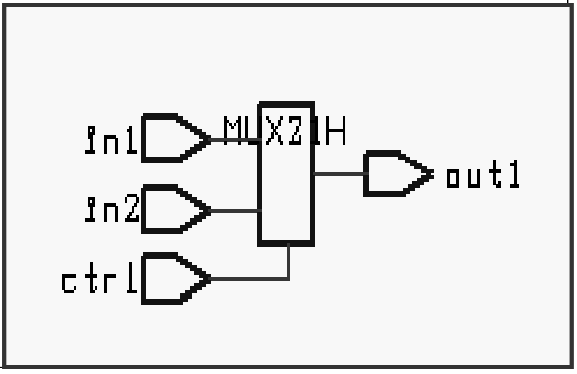
```
     begin
          carry (0) := '0';
          for i in 0 to 3 loop
               sum (i) := a(i) xor b(i)
                                   xor carry(i);
               carry (i+1) := (a(i) and
                        b(i)) or (b(i) and
                                   carry (i)) or
                        (carry (i) and a(i));
          end loop;
     c <= sum;
     end process;
end variable_ex_a;
```

…Variable Definition



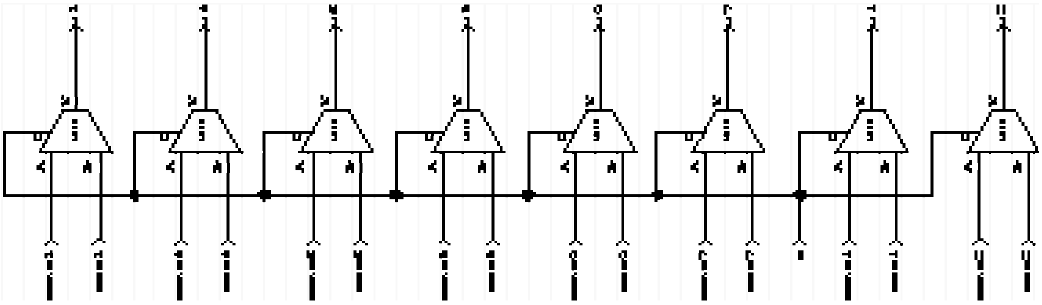
Multiplexor Synthesis…

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is
    port (in1, in2, ctrl : in std_logic; out1 : out std_logic);
end mux;
architecture mux_a of mux is
begin
    process (in1, in2, ctrl)
    begin
            if ctrl = '0' then out1 <= in1;
            else out1 <= in2;
            end if;
    end process;
end mux_a;
```



…Multiplexor Synthesis

```
entity mux2to1_8 is
    port ( signal s : in std_logic; signal zero,one: in std_logic_vector(7
    downto 0); signal y: out std_logic_vector(7 downto 0) );
end mux2to1_8;
architecture behavior of mux2to1 is
begin
            y <= one when (s = '1') else zero;
end behavior;
```



2x1 Multiplexor using Booleans

```
architecture boolean_mux of mux2to1_8 is
    signal temp: std_logic_vector(7 downto 0);
begin
     temp <= (others => s);
    y <= (temp and one) or (not temp and zero);
end boolean_mux;
```

- **The *s* signal cannot be used in a Boolean operation with the *one* or *zero* signals because of type mismatch (*s* is a std_logic type, *one/zero* are std_logic_vector types)**
- **An internal signal of type std_logic_vector called *temp* is declared. The *temp* signal will be used in the Boolean operation against the *zero/one* signals.**
- **Every bit of *temp* is set equal to the *s* signal value.**

2x1 Multiplexor using a Process

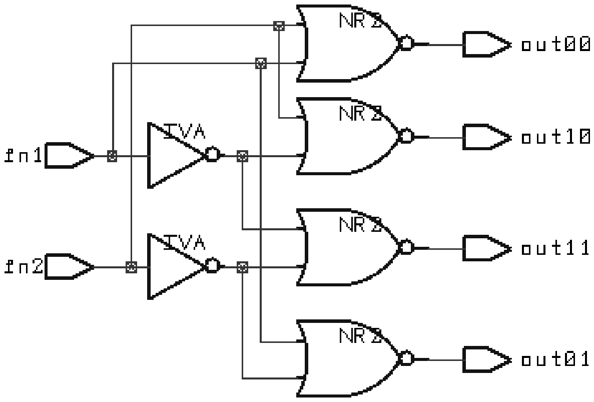
```
architecture process_mux of mux2to1_8 is
begin
    comb: process (s, zero, one)
    begin
            y <= zero;
            if (s = '1') then
                y <= one;
            end if;
    end process comb;
end process_mux ;
```

Decoder Synthesis…

```
library ieee;
use ieee.std_logic_1164.all;
entity decoder is
     port (in1, in2 : in std_logic; out00, out01, out10, out11 : out std_logic);
end decoder;
architecture decoder_a of decoder is
begin
    process (in1, in2)
    begin
           if in1 = '0' and in2 = '0' then out00 <= '1';
           else out00 <= '0';
           end if;
           if in1 = '0' and in2 = '1' then out01 <= '1';
           else out01 <= '0';
           end if;
```

…Decoder Synthesis

```
           if in1 = '1' and in2 = '0' then out10 <= '1';
           else out10 <= '0';
           end if;
           if in1 = '1' and in2 = '1' then out11 <= '1';
           else out11 <= '0';
           end if;
    end process;
end decoder_a;
```



3-to-8 Decoder Example…

```
entity dec3to8 is
    port (sel: in std_logic_vector(2 downto 0); en: in std_logic;
           y: out std_logic_vector(7 downto 0))
end dec3to8;
architecture behavior of dec3to8 is
begin
   process (sel, en)
   begin
        variable yv: Std_logic_vector(7 downto 0);
        yv := “1111111”;
        if (en = ‘1’)  then
            case sel is
                when “000” => yv(0) := ‘0’;     when “001” => yv(1) := ‘0’;
                when “010” => yv(2) := ‘0’;     when “011” => yv(3) := ‘0’;
                when “100” => yv(4) := ‘0’;     when “101” => yv(5) := ‘0’;
                when “110” => yv(6) := ‘0’;     when “111” => yv(7) := ‘0’;
            end case;
        end if;
       y <= yv;
    end process;
end behavior;
```

Architecture of Generic Decoder

```
architecture behavior of generic_decoder  is
begin
    process (sel, en)
    begin
          y <= (others => '1') ;
           for  i    in  y'range  loop
                     if ( en = '1' and bvtoi(To_Bitvector(sel)) = i ) then
                               y(i) <= '0' ;
                     end if ;
          end loop;
    end process;
end behavior;
```

bvtoi is a function to convert from bit_vector to integer

A Common Error in Process Statements…

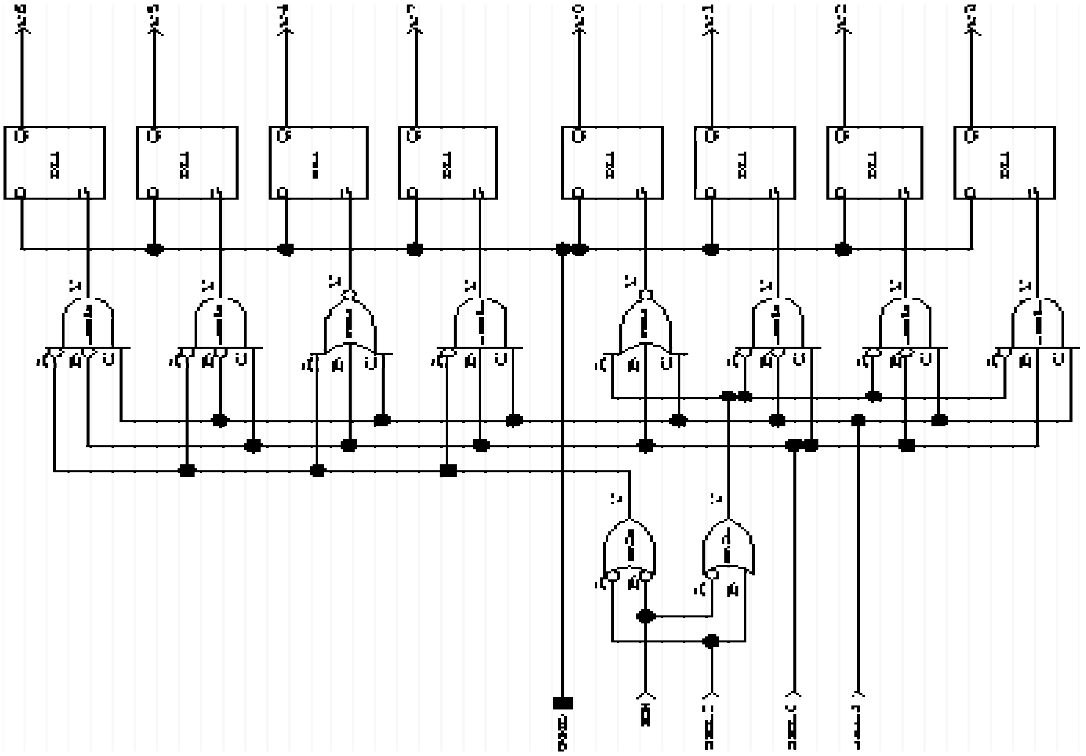
- When using processes, a common error is to forget to assign an output a **default value**.
  - ALL outputs should have DEFAULT values
- If there is a logical path in the model such that an output is not assigned any value
  - the synthesizer will assume that the output must retain its current value
  - a latch will be generated.
- Example: In *dec3to8.vhd* do not assign *'y'* the default value of B"11111111"
  - If *en* is 0, then 'y' will not be assigned a value
  - In the new synthesized logic, all '*y*' outputs are latched

…A Common Error in Process Statements…

```
entity dec3to8 is
    port (signal sel: in std_logic_vector(3 downto 0); signal en: in std_logic;           signal
    y: out std_logic_vector(7 downto 0))
end dec3to8;
architecture behavior of dec3to8 is
begin
    process (sel, en)
    --      y <= “1111111”;
            if (en = ‘1’)  then
                        case sel is
                          when “000” => y(0) <= ‘0’;     when “001” => y(1) <= ‘0’;
                          when “010” => y(2) <= ‘0’;     when “011” => y(3) <= ‘0’;
                          when “100” => y(4) <= ‘0’;     when “101” => y(5) <= ‘0’;
                          when “110” => y(6) <= ‘0’;     when “111” => y(7) <= ‘0’;
                        end case;
            end if;
    end process;
end behavior;
```

No default value assigned to y!!

…A Common Error in Process Statements

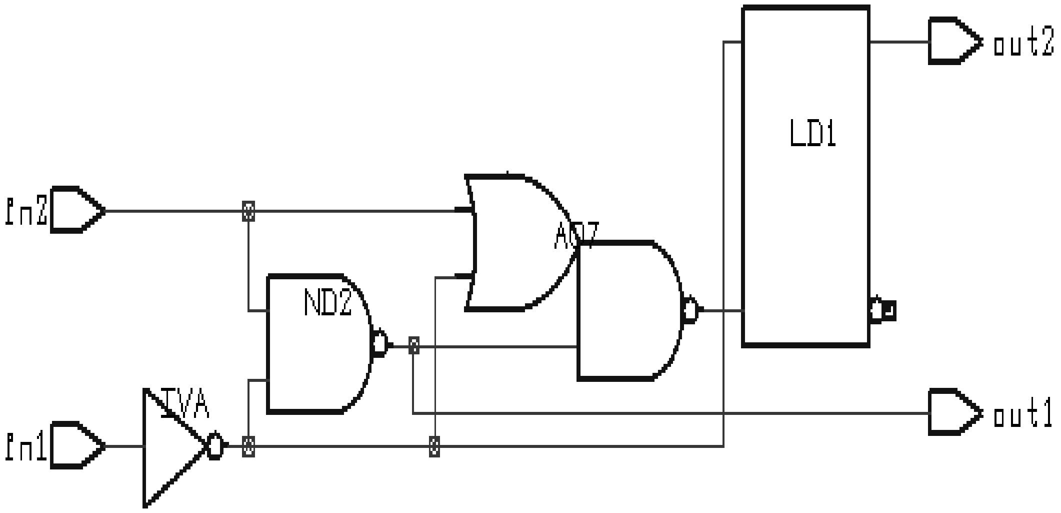


Another Incorrect Latch Insertion Example…

```
entity case_example is
    port (in1, in2 : in std_logic; out1, out2 : out std_logic);
end case_example;
architecture case_latch of case_example is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
          case b is
                    when "01" => out1 <= '0'; out2 <= '1';
                    when "10" => out1 <= '1'; out2 <= '0';
                    when others => out1 <= '1';
          end case;
    end process;
    b <= in1 & in2;
end case_latch;
```

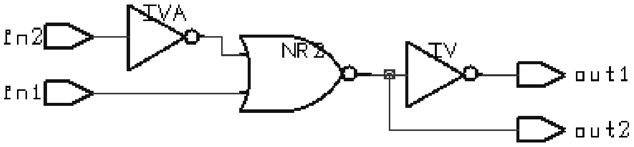
out2 has not been assigned a value for others condition!!

…Another Incorrect Latch Insertion Example



Avoiding Incorrect Latch Insertion

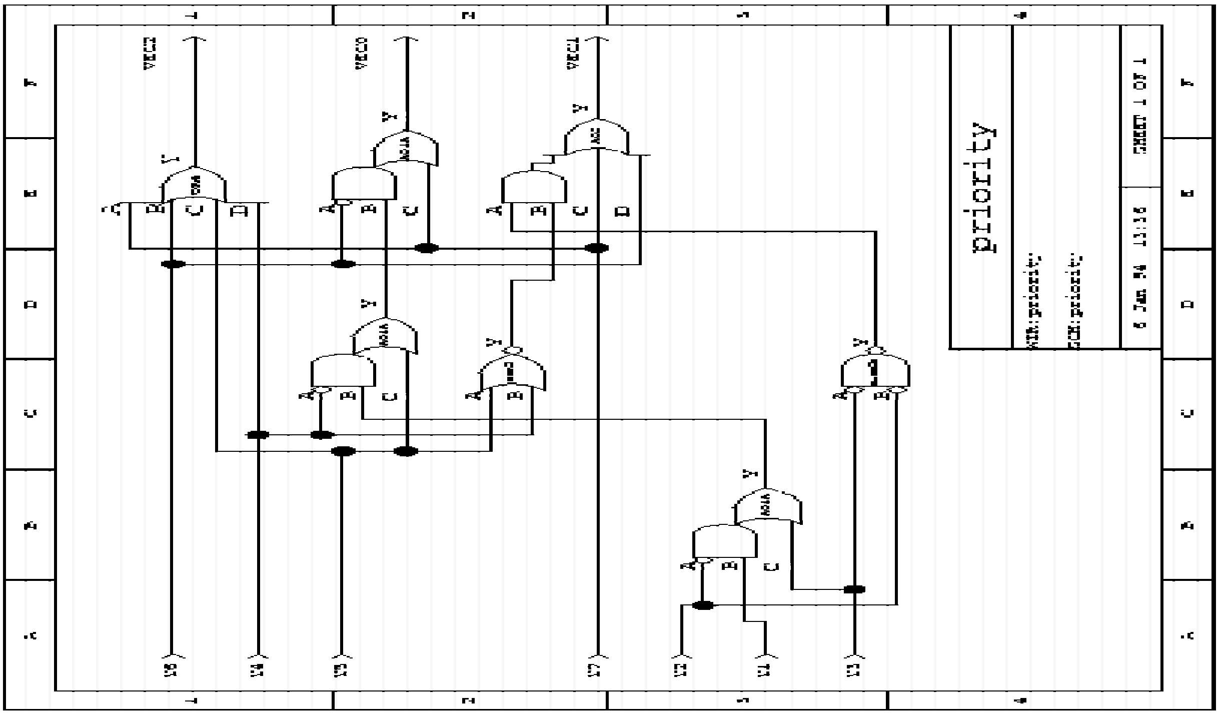
```
architecture case_nolatch of case_example is
    signal b : std_logic_vector (1 downto 0);
begin
    process (b)
    begin
           case b is
                      when "01" => out1 <= '0'; out2 <= '1';
                      when "10" => out1 <= '1'; out2 <= '0';
                      when others => out1 <= '1'; out2 <= '0';
           end case;
    end process;
    b <= in1 & in2;
end case_nolatch;
```



Eight-Level Priority Encoder…

```
Entity priority is
    Port (Signal y1, y2, y3, y4, y5, y6, y7: in std_logic;
                Signal vec: out std_logic_vector(2 downto 0));
End priority;
Architecture behavior of priority is
Begin
    Process(y1, y2, y3, y4, y5, y6, y7)
    begin
           if (y7 = ‘1’)  then vec <= “111”;      elsif (y6 = ‘1’)  then vec <= “110”;
           elsif (y5 = ‘1’)  then vec <= “101”;  elsif (y4 = ‘1’)  then vec <= “100”;
           elsif (y3 = ‘1’)  then vec <= “011”;  elsif (y2 = ‘1’)  then vec <= “010”;
           elsif (y1= ‘1’)  then vec <= “001”;   else vec <= “000”;
           end if;
    end process;
End behavior;
```

…Eight-Level Priority Encoder…



Eight-Level Priority Encoder…

```
Architecture behavior2 of priority is
Begin
    Process(y1, y2, y3, y4, y5, y6, y7)
    begin
          vec <= “000”;
          if (y1 = ‘1’)  then vec <= “001”;  end if;
          if (y2 = ‘1’)  then vec <= “010”;  end if;
          if (y3 = ‘1’)  then vec <= “011”;  end if;
          if (y4 = ‘1’)  then vec <= “100”;  end if;
          if (y5 = ‘1’)  then vec <= “101”;  end  if;
          if (y6 = ‘1’)  then vec <= “110”;  end if;
          if (y7= ‘1’)  then vec <= “111”;   end if;
    end process;
End behavior2;
```

Equivalent 8-level priority encoder.

Ripple Carry Adder…

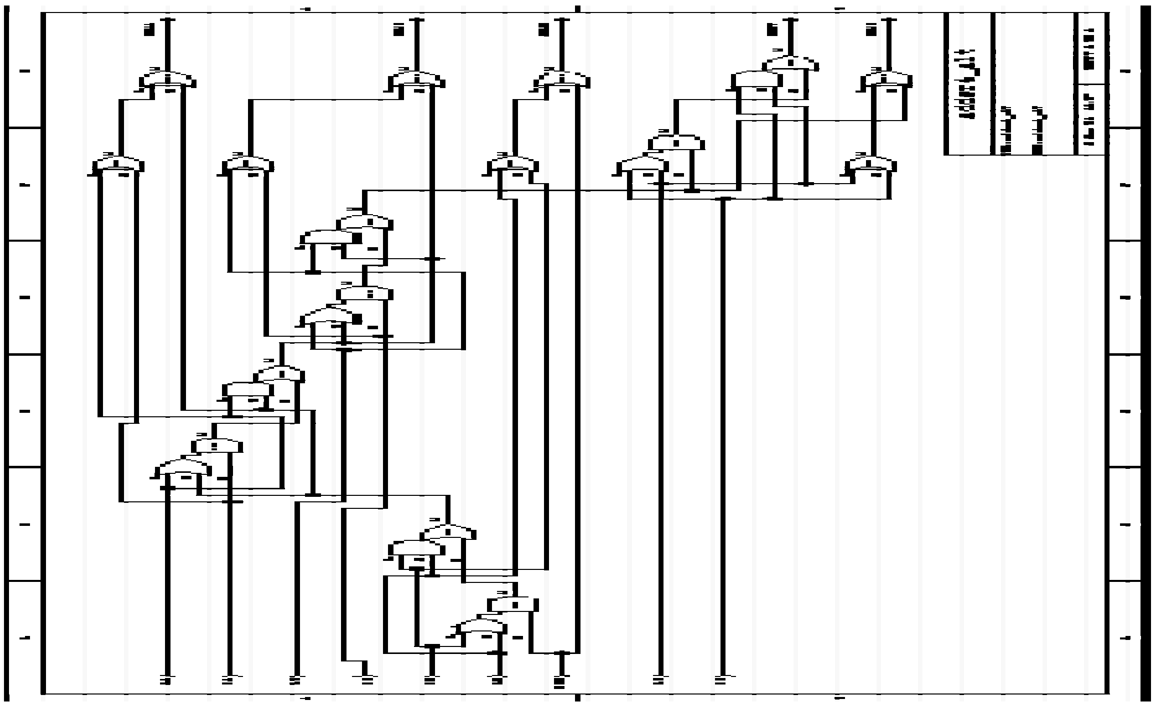
```
library ieee;
use ieee.std_logic_1164.all;
entity adder4 is
    port (Signal  a, b:  in std_logic_vector (3 downto 0);
            Signal cin : in std_logic_vector;
            Signal  sum:  out std_logic_vector (3 downto 0);
            Signal cout : out  std_logic_vector);
end adder4;
architecture behavior  of adder4 is
Signal  c:  std_logic_vector (4 downto 0);
begin
```

C is a temporary signal to hold the carries.

…Ripple Carry Adder…

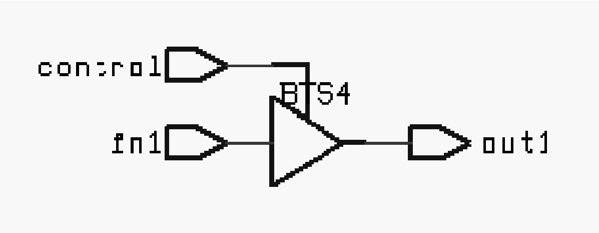
```
    process (a, b, cin, c)
    begin
          c(0) <= cin;
          for I in 0 to 3 loop
                    sum(I) <= a(I) xor b(I) xor c(I);
                    c(I+1) <= (a(I) and b(I)) or (c(I) and (a(I) or b(I)));
          end loop;
    end process;
    cout <= c(4);
End behavior;
```

…Ripple Carry Adder



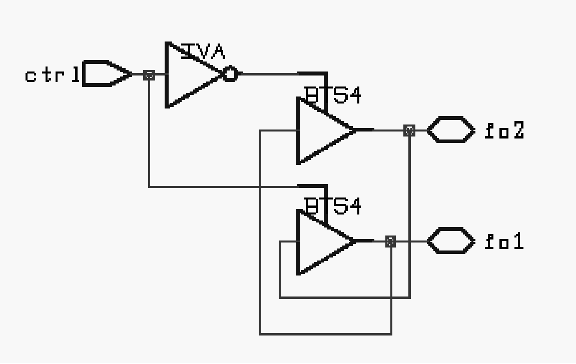
Tri-State Buffer Synthesis

```
library ieee;
use ieee.std_logic_1164.all;
entity tri_ex is
    port (in1, control : in std_logic; out1 : out std_logic);
end tri_ex;
architecture tri_ex_a of tri_ex is
begin
     out1 <= in1 when control = '1' else 'Z';
end tri_ex_a;
```



Bi-directional Buffer Synthesis

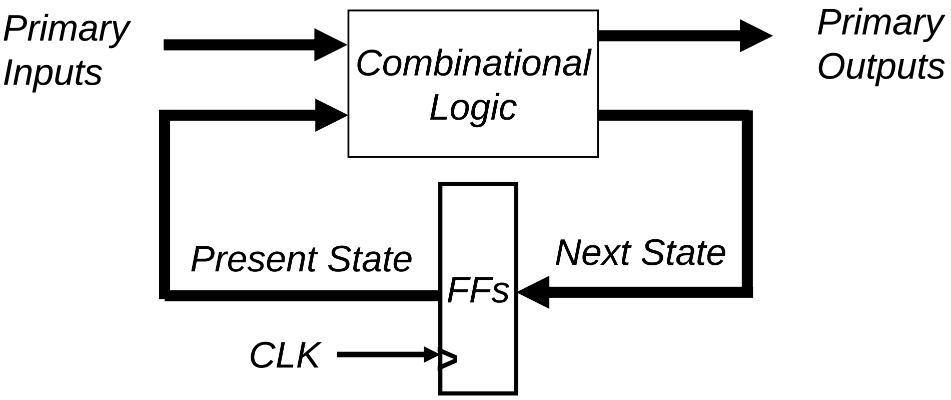
```
library ieee;
use ieee.std_logic_1164.all;
entity inout_ex is
    port (io1, io2 : inout std_logic; ctrl : in std_logic);
end inout_ex;
architecture inout_ex_a of inout_ex is
begin
    io1 <= io2 when ctrl = '1' else 'Z';
    io2 <= io1 when ctrl = '0' else 'Z';
end inout_ex_a;
```



Eefficient Coding Style

Sequential Circuits

- Sequential circuits consist of both combinational logic and storage elements.
- Sequential circuits can be
  - *Moore*-type: outputs are a combinatorial function of Present State signals.
  - *Mealy*-type: outputs are a combinatorial function of both Present State signals and primary inputs.

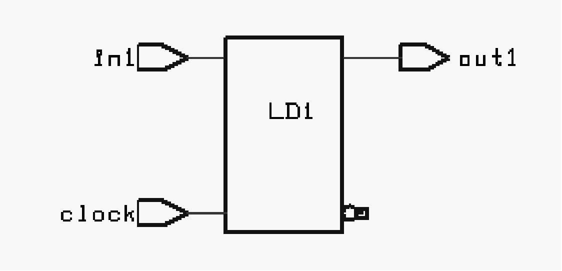


Template Model for a Sequential Circuit

```
entity model_name is
    port ( list of inputs and outputs );
end model_name;
architecture behavior of model_name is
    internal signal declarations
begin
    -- the state process defines the storage elements
    state: process ( sensitivity list -- clock, reset, next_state inputs)
    begin
           vhdl statements for state elements
    end process state;
    -- the comb process defines the combinational logic
    comb: process ( sensitivity list -- usually includes all inputs)
    begin
           vhdl statements which specify combinational logic
    end process comb;
end behavior;
```

Latch Synthesis…

```
library ieee;
use ieee.std_logic_1164.all;
entity latch_ex is
    port (clock, in1 : in std_logic; out1 : out std_logic);
end latch_ex;
architecture latch_ex_a of latch_ex is
begin
    process (clock, in1)
    begin
            if (clock = '1') then
                    out1 <= in1;
           end if;
    end process;
end latch_ex_a;
```

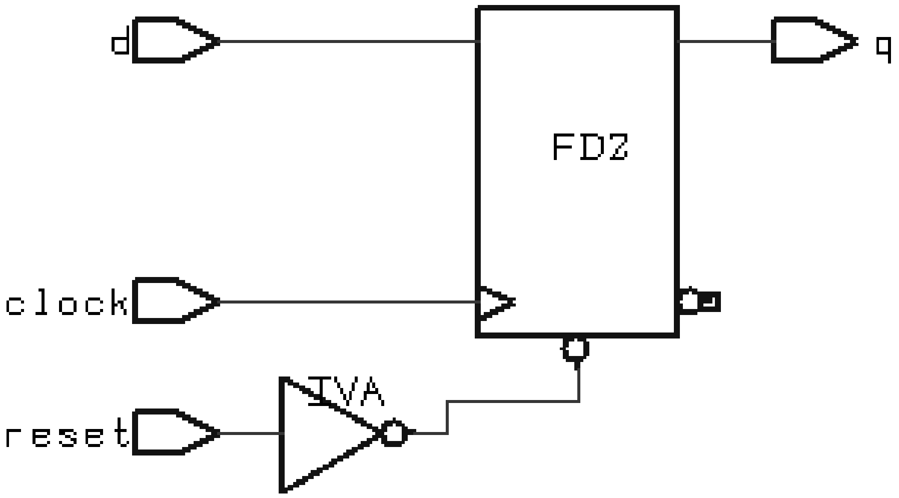


Flip-Flop Synthesis with Asynchronous Reset…

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_asyn is
    port( reset, clock, d: in std_logic; q: out std_logic);
end dff_asyn;
architecture dff_asyn_a of dff_asyn is
begin
    process (reset , clock)
    begin
            if (reset = '1') then
                    q <= '0';
            elsif rising_edge(clock) then
                    q <= d;
            end if;
    end process;
end dff_asyn_a;
```

•*Note that the reset input has precedence over the clock in order to define the asynchronous operation.*

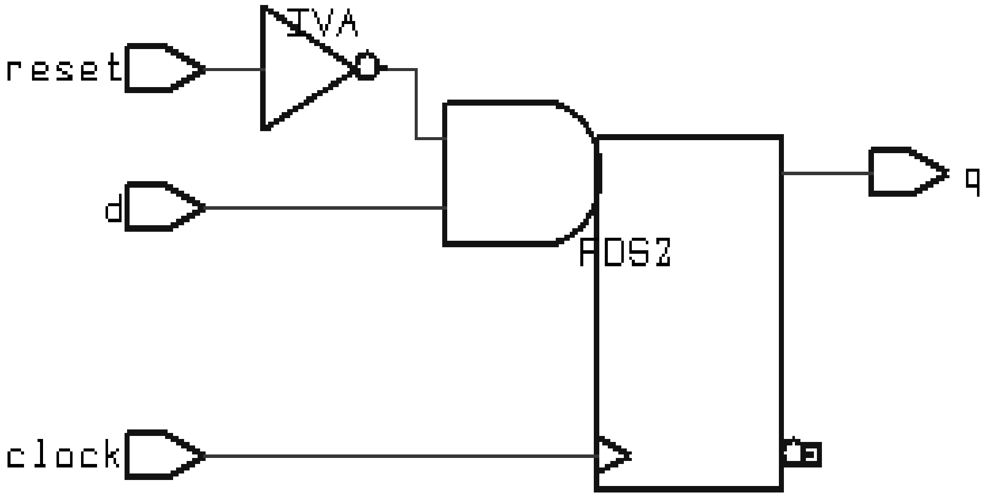
…Flip-Flop Synthesis with Asynchronous Reset



Flip-Flop Synthesis with Synchronous Reset…

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_syn is
   port( reset, clock, d: in std_logic; q: out std_logic);
end dff_syn;
architecture dff_syn_a of dff_syn is
begin
   process (clock) – Only The clock needs to be in the sensitivity
   list
   begin
         if rising_edge(clock) then
                if (reset = '1') then q <= '0';
                else q <= d;
```

…Flip-Flop Synthesis with Synchronous Reset



8-bit Loadable Register with Asynchronous Clear…

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8bit is
    port( reset, clock, load: in std_logic;
          din: in std_logic_vector(7 downto 0);
          dout: out std_logic_vector(7 downto 0));
end reg8bit;
architecture behavior of reg8bit is
    signal n_state, p_state: std_logic_vector(7 downto 0);
begin
    dout <= p_state;
    comb: process (p_state, load, din)
    begin
          n_state <= p_state;
          if (load = '1') then n_state <= din end if;
    end process comb;
```

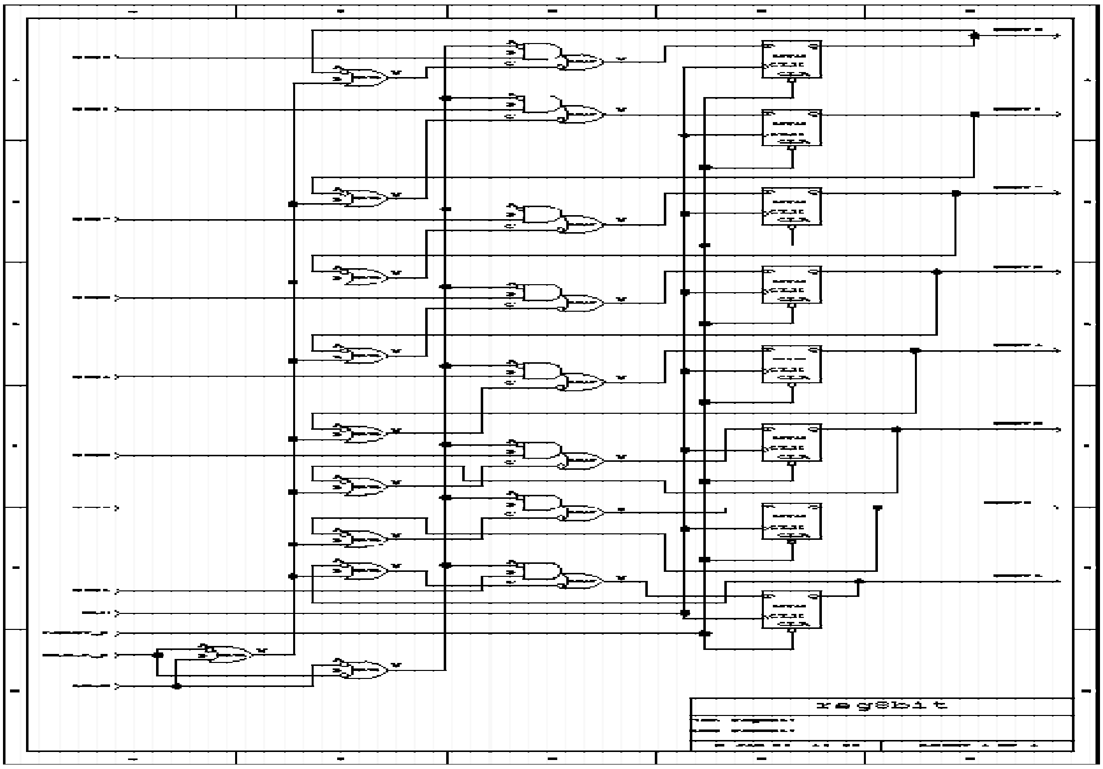
…8-bit Loadable Register with Asynchronous Clear…

```
   state:  process (clk, reset)
   begin
        if (reset = ‘0') then   p_state   <= (others => '0‘);
        elsif rising_edge(clock) then
                  p_state  <= n_state;
        end if;
   end process state;
End behavior;
```

- *The state process defines a storage element which is 8-bits wide, rising edge triggered, and had a low true asynchronous reset.*

- *Note that the reset input has precedence over the clock in order to define the asynchronous operation.*

…8-bit Loadable Register with Asynchronous Clear



4-bit Shift Register…

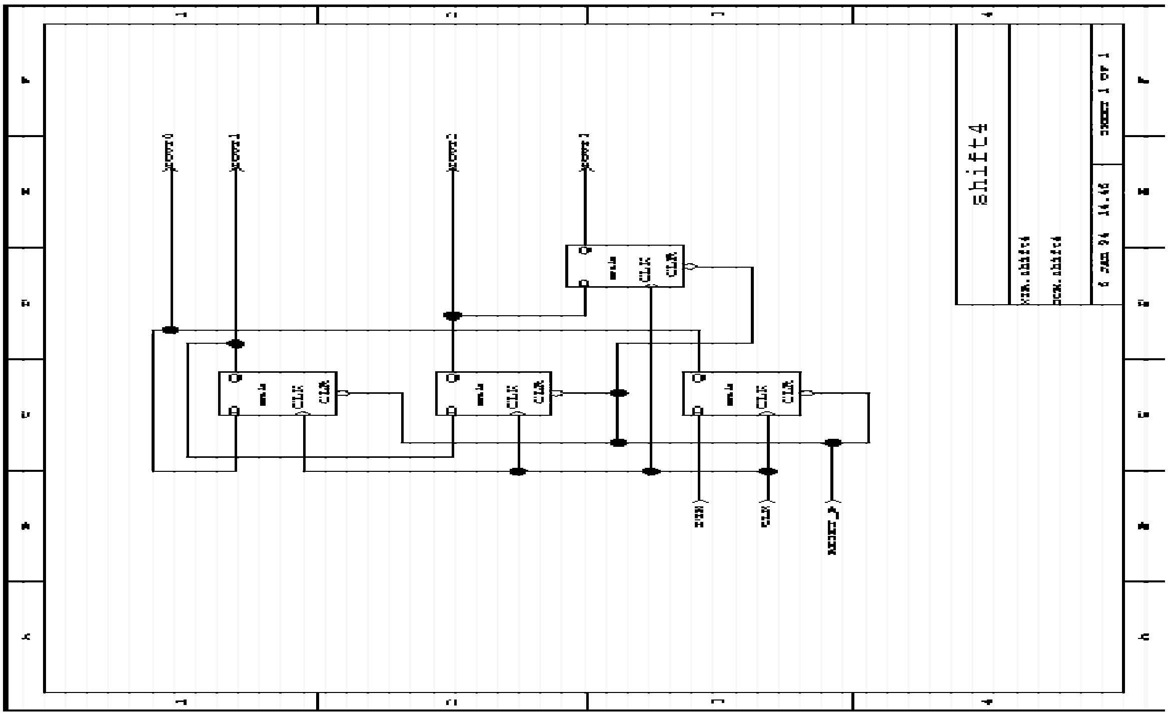
```
library ieee;
use ieee.std_logic_1164.all;
entity shift4 is
    port( reset, clock: in std_logic;   din: in std_logic;
      dout: out std_logic_vector(3 downto 0));
end shift4;
architecture behavior of shift4 is
    signal n_state, p_state: std_logic_vector(3 downto 0);
begin
    dout <= p_state;
    state: process (clock, reset)
    begin
            if (reset = ‘0') then   p_state  <= (others => '0‘);
            elsif rising_edge(clock) then
                        p_stateq <= n_state;
            end if;
    end process state;
```

…4-bit Shift Register…

```
            comb: process (p_state, din)
            begin
                        n_state(0) <= din;
                        for I in 3 downto 0 loop
                                    n_state(I)  <= p_state(I-1);
                        end loop;
            end process comb;
End behavior;
```

- *Serial input din is assigned to the D-input of the first D-FF.*
- *For loop is used to connect the output of previous flip-flop to the input of current flip-flop.*

…4-bit Shift Register



Register with Tri-State Output…

```
library ieee;
use ieee.std_logic_1164.all;
entity tsreg8bit is
    port( reset, clock, load, en: in std_logic;
    signal din: in std_logic_vector(7 downto 0);
    signal dout: out std_logic_vector(7 downto 0));
end tsreg8bit;
architecture behavior of tsreg8bit is
    signal n_state, p_state: std_logic_vector(7 downto 0);
begin
    dout <= p_state when (en=‘1’) else “ZZZZZZZZ”;
    comb: process (p_state, load, din)
    begin
          n_state <= p_state;
          if (load = '1') then n_state <= din end if;
    end process comb;
```

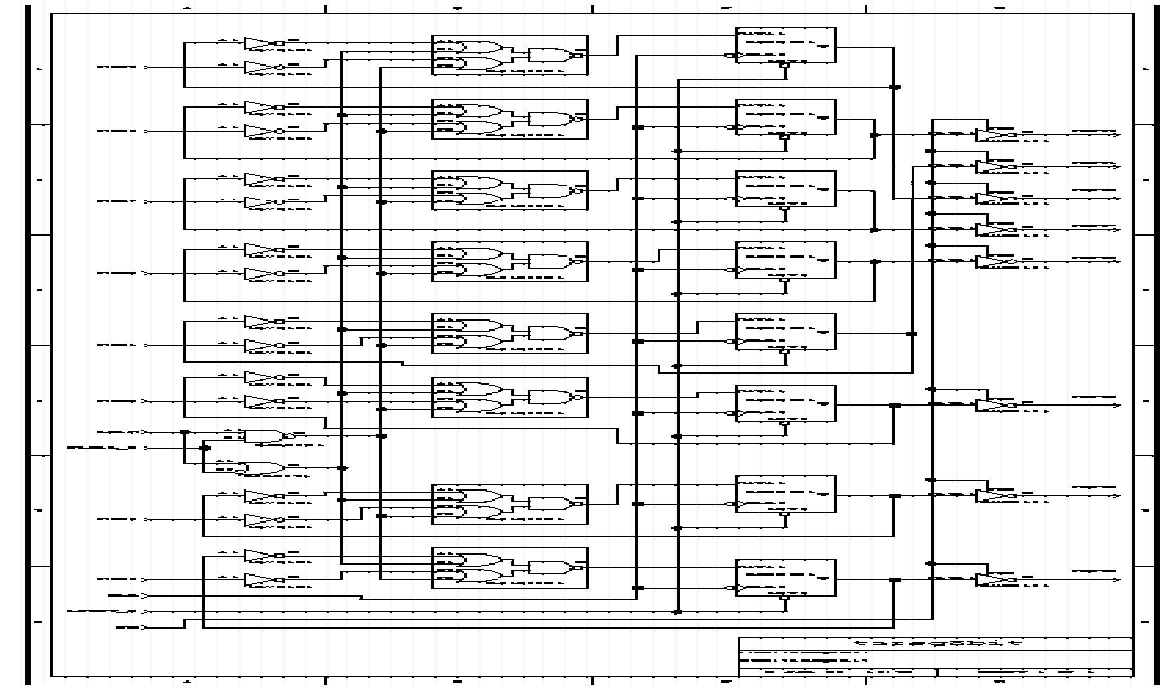
- *Z assignment used to specify tri-state capability.*

…Register with Tri-State Output…

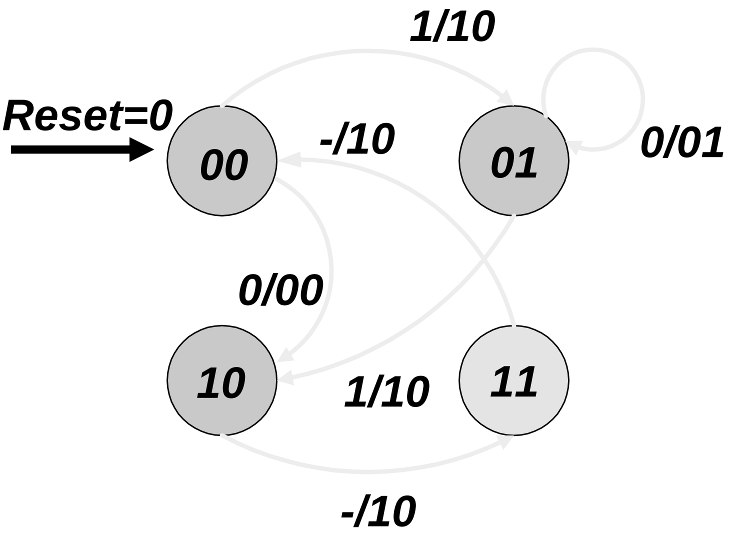
```
   state:  process (clock, reset)
        begin
                 if (reset = ‘0') then   p_state   <= (others
   => '0‘);
                 elsif rising_edge(clock) then
                         p_state  <= n_state;
                 end if;
        end process state;
End behavior;
```

…Register with Tri-State Output

Mapped to ITD stdcell library because Actel ACT1 does not have tristate capability.



Finite State Machine Synthesis…



- *Mealy model*
- *Single input, two outputs*
- *Synchronous reset*

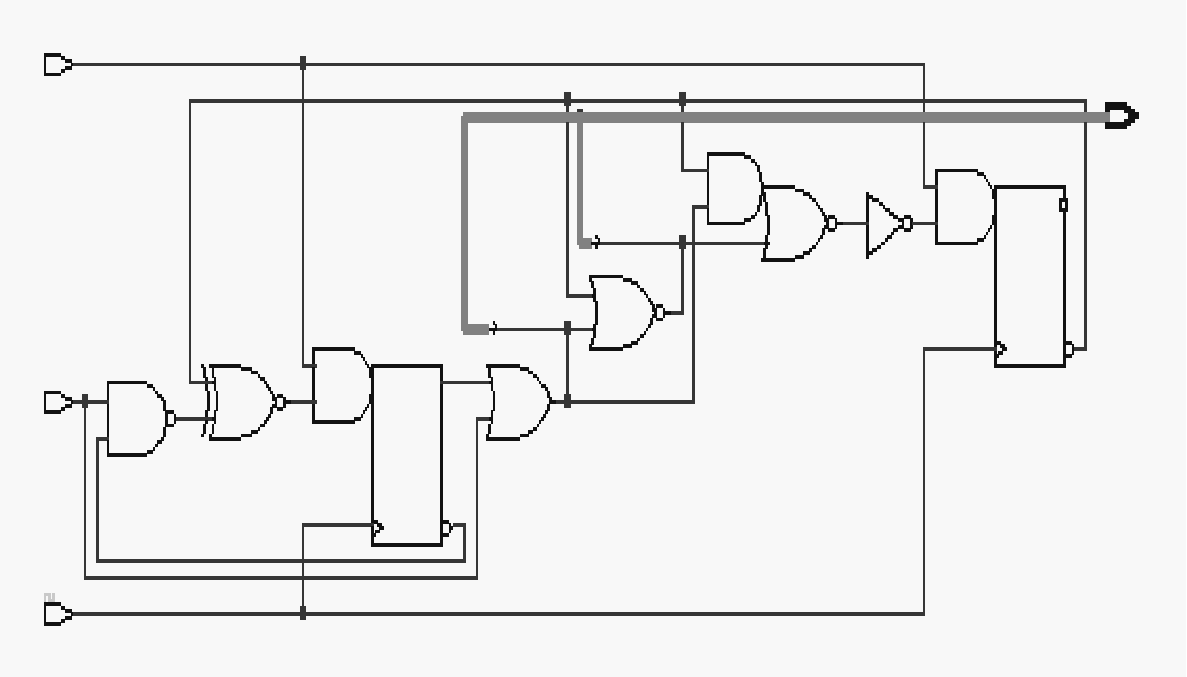
…Finite State Machine Synthesis…

```
library ieee;    use ieee.std_logic_1164.all;
entity state_ex is
     port (in1, clock, reset : in std_logic; out1 :
            out std_logic_vector (1 downto 0));
end state_ex;
architecture state_ex_a of state_ex is
    signal cur_state, next_state : std_logic_vector (1 downto 0);
begin
    process (clock, reset)
     begin
            if rising_edge(clock) then
                       if reset = '0' then cur_state <= "00";
                      else cur_state <= next_state;
                      end if;
           end if;
   end process;
```

…Finite State Machine Synthesis…

```
    process (in1, cur_state)
    begin
            case cur_state is
                        when "00" => if in1 = '0' then next_state <= "10"; out1 <=
    "00";
                                         else next_state <= "01"; out1 <= "10";
                                     end if;
                         when "01" => if in1 = '0' then next_state <= cur_state;
                                                   out1 <= "01";
                                         else next_state <= "10"; out1 <= "10";
                                      end if;
                        when "10" => next_state <= "11"; out1 <= "10";
                        when "11" => next_state <= "00"; out1 <= "10";
                        when others => null;
             end case;
    end process;
end Architecture state_ex_a;
```

…Finite State Machine Synthesis



Efficient Coding Styles

Key Synthesis Facts

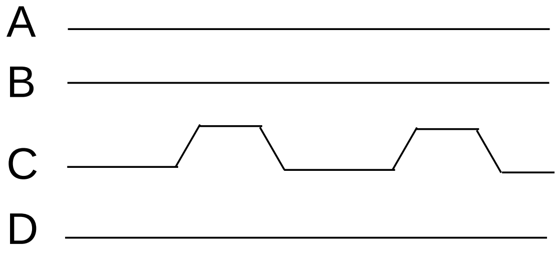
- Synthesis ignores the after clause in signal assignment
  - C <= A AND B after 10ns
  - May cause mismatch between pre-synthesis and post-synthesis simulation if a non-zero value used
  - The preferred coding style is to write signal assignments without the after clause.
- If the process has a static sensitivity list, it is ignored by the synthesis tool.
- Sensitivity list must contain all read signals
  - Synthesis tool will generate a warning if this condition is not satisfied
  - Results in mismatch between pre-synthesis and post-synthesis simulation

Synthesis Static Sensitivity Rule

Original VHDL Code

```
Process(A, B)
Begin
    D <= (A AND B) OR C;
End process;
```

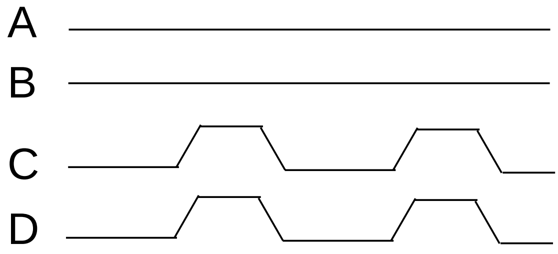
Pre-Synthesis Simulation



Synthesis View of Original VHDL Code

```
Process(A, B, C)
Begin
    D <= (A AND B) OR C;
End process;
```

Post-Synthesis Simulation



Impact of Coding Style on Synthesis Execution Time

Inefficient Synthesis Execution Time

```
Process(Sel, A, B, C, D)
Begin
    if Sel = “00 then  Out <= A;
    elsif Sel = “01” then Out<=B;
    elsif Sel = “10” then Out<=C;
    else  Out<=D;
    endif;
End process;
```

Efficient Synthesis Execution Time

```
Process (Sel, A, B, C, D)
Begin
    case Sel is
        when “00 =>  Out <= A;
        when “01” Out<=B;
        when “10” Out<=C;
        when “11”  Out<=D;
    end case;
End process;
```

- *Synthesis tool is capable of deducing that the if …elsif conditions are mutually exclusive but precious CPU time is required.*
- *In case statement, when conditions are mutually exclusive.*

Synthesis Efficiency Via Vector Operations

Inefficient Synthesis Execution Time

```
Process (Scalar_A, Vector_B)
Begin
    for k in Vector_B`Range loop
        Vector_C(k) <=Vector_B(k) and
                     Scalar_A;
    end loop;
End process;
```

Efficient Synthesis Execution Time

```
Process(Scalar_A, Vector_B)
    variable Temp:
    std_logic_vector(Vector_B`Range);
Begin
    Temp := (others => Scalar_A);
    Vector_C <=Vector_B and Temp;
End process;
```

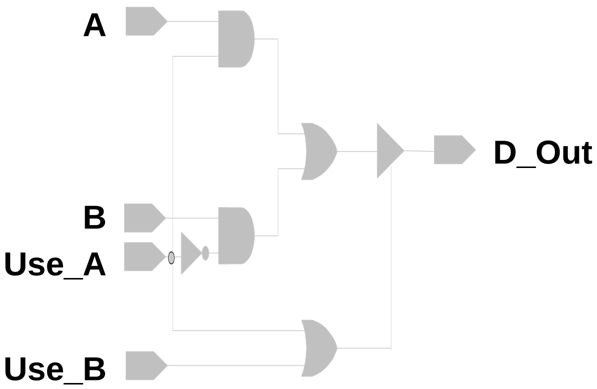
- *Loop will be unrolled and analyzed by the synthesis tool.*
- *Vector operation is understood by synthesis and will be efficiently synthesized.*

Three-State Synthesis

- A tri-state driver signal must be declared as an object of type std_logic.
- Assignment of ‘Z’ infers the usage of three-state drivers.
- The std_logic_1164 resolution function, resolved, is synthesized into a three-state driver.
- Synthesis does not check for or resolve possible data collisions on a synthesized three-state bus
  - It is the designer responsibility
- Only one three-state driver is synthesized per signal per process.

Example of the Three-State / Signal / Process Rule

```
Process(B, Use_B, A, Use_A)
Begin
    D_Out <= ‘Z’;
    if Use_B = ‘1’ then
          D_Out <= B;
    end if;
    if Use_A = ‘1’ then
          D_Out <= A;
    end if;
End process;
```



•*Last scheduled assignment has priority*

Latch Inference & Synthesis Rules…

- A latch is inferred to satisfy the VHDL fact that signals and process declared variables maintain their values until assigned new ones.
- Latches are synthesized from **if** statements if all the following conditions are satisfied
  - Conditional expressions are not completely specified
    - e.g. An else clause is omitted
  - Objects conditionally assigned in an if statement are not assigned a *default* value before entering this if statement
  - The VHDL attribute **`EVENT** is not present in the conditional if expression → **`EVENT** produces a FF
- If latches are not desired, then a value must be assigned to the target object under all conditions of an if statement (without the **`EVENT** attribute).

…Latch Inference & Synthesis Rules

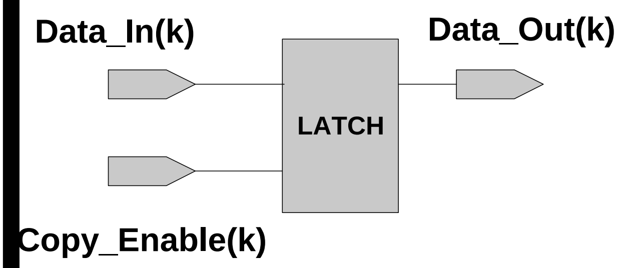
- For a **case** statement, latches are synthesized when all the following conditions are met:
  - One or more arms does not assigned value to a VHDL object.
  - No *default* value assigned to this object before the case statement is entered.

- Latches are synthesized whenever a **for…loop** statement satisfies all of the following conditions
  - **for…loop** contains a **next** statement
  - Objects assigned inside the **for…loop** are not assigned a *default* value before entering the enclosing **for…loop**

For…Loop Statement Latch Example

```
Process(Data_In, Copy_Enable)
Begin
 for k in 7 downto 0 loop
    next when  Copy_Enable(k)=‘0’ ;
    Data_Out(k) <= Data_in(k);
 end loop;
End process;
```

Seven latches will be synthesized



Flip-Flop Inference & Synthesis Rules…

- Flip-flops are inferred by either
  - **IF** statement containing **`EVENT**
  - **Wait until**….
    - **Wait on**… is not supported by synthesis
    - **Wait for**… is not supported by synthesis
- Synthesis accepts any of the following functionally equivalent statements for inferring a FF
  - **Wait until** Clock=‘1’;
  - **Wait until** Clock`Event and Clock=‘1’;
  - **Wait until** (not Clock`Stable) and Clock=‘1’;

…Flip-Flop Inference & Synthesis Rules

- Synthesis does not support the following Asynchronous description of set and reset signals
  - **Wait until** (clock=‘1’) or (Reset=‘1’) ;
  - **Wait on** Clock, Reset ;
- When using a synthesizable wait statement **only synchronous set and reset** can be used.
- **If** statement containing the VHDL attribute **`EVENT** cannot have an **else** or an **elsif** clause.

Alternative Efficient Coding Styles for Synchronous FSMs

- One process only
  - Handles both state transitions and outputs
- Two processes
  - A synchronous process for updating the state register
  - A combinational process for conditionally deriving the next state and updating the outputs
- Three processes
  - A synchronous process for updating the state register
  - A combinational process for conditionally deriving the next state
  - A combinational process for conditionally deriving the outputs

Conclusions

To ensure a **PROCESS** is synthesizable, it has to be one of following types:

- **Type 1:** Purely Combinational: all outputs are functions of only the current inputs (not the previous inputs)
- **Type 2:** Purely Synchronous: Each output changes only on the rising or falling edge of a single clock
- **Type 3:** Purely Synchronous with asynchronous set or reset

General rules for synthesis

- Every process must fall exactly into one of the above categories

Conclusions

Type 1: Purely Combinational Processes (RULES For purely combinational processes

Rule 1: Every input that can affect the output(s) must be in the sensitivity list.

Rule 2: Every output must be assigned a value for every possible combination of the inputs (binary values only)

Example: The following code is synthesizable

```
Process (sel, A, B)
    begin
    if (sel = ‘0’) then Y <= A; else Y <= B; end if;
end process;
```

Conclusions

Example of VHDL that will not be synthesized properly due to violation of rule 1

```
process (A, B)
begin
if (SEL = ‘0’) then  Y <= A; else Y<= B;
end if;
end process;
```

Example of VHDL that will not be synthesized properly due to violation of rule 2

```
process (SEL, A, B)
begin
if (SEL = ‘0’) then  Y <= A;
end if;
end process;
```

Conclusions

Type 2: Purely Synchronous Sequential Processes

Rules for purely sequential processes

- **Rule 1:** Only the clock should be in the sensitivity list
- **Rule 2:** Only signals that change on the same edge of the same clock should be part of the same process

```
process (CLK)
begin
if (CLK’event and CLK=‘1’) then
    Z <= A and B;
end if;
end process;
```

- Note that the same logic circuit will result even when the **if** statement is rewritten as

```
if (CLK=‘1’) then Z <= A and B;
```

Conclusions

Type 3: Purely Synchronous with Asynchronous Set/Reset

Rules:

- **Rule** 1: Sensitivity list includes clock and set/reset signal
- **Rule** 2: Must include the clk'event clause
- **Rule** 3: Must assign either 0 or 1 inside the first part of the **if** statement (i.e. the asynchronous condition is decided first)

CPU Modeling
Case Study: PARWAN

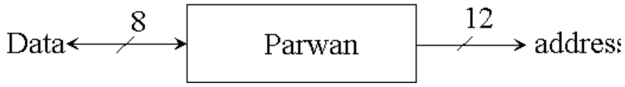
OUTLINE

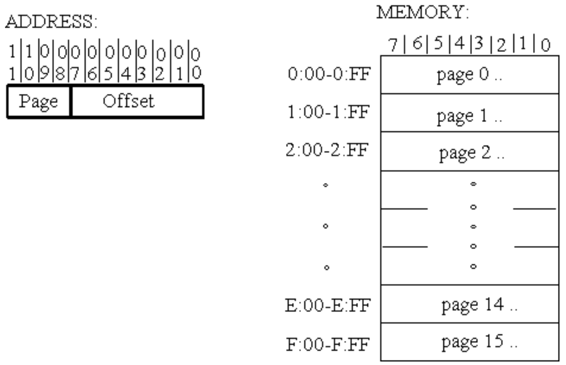
- The CPU
- Memory organization
- Instructions
- Addressing
- Utilities for VHDL description
- Interface
- Behavioral description
- Coding individual instructions
- Complete Behavioral Description

General description

- **PARWAN; *PAR_1*; A Reduced Processor**
- **Simple 8-bit CPU**
- **8-bit Data; 12-bit Address**
- **Primarily designed for educational purposes**
- **Includes most common instructions**

General CPU description





- 12-Bit Address Bus (4-Bit Page Address + 8-Bit Offset Within Page)
- Memory divided into pages (2^4 = 16 Page), Each Page has 2^8 = 256 bytes
- 12-Bit Address (*3 Hex Digits*) → One digit specifies the page and 2 digits for the offset → { X : YZ } (X = Page Number, YZ = Offeset Within Page)
- 8-Bit Data Bus → 16 Pages 256 BYTES Each
- **Uses memory mapped IO**

Instruction Set

1. FULL Address Instructions → (2-Byte Instructions →12 bit Address + Op-Code → Can be Direct or Indirect)

   LDA, AND, ADD, SUB, JMP, STA

2. PAGE Address Instructions ==>> (2-Byte Instructions → 8 bits offset + Op_Code → Only Direct Addressing)

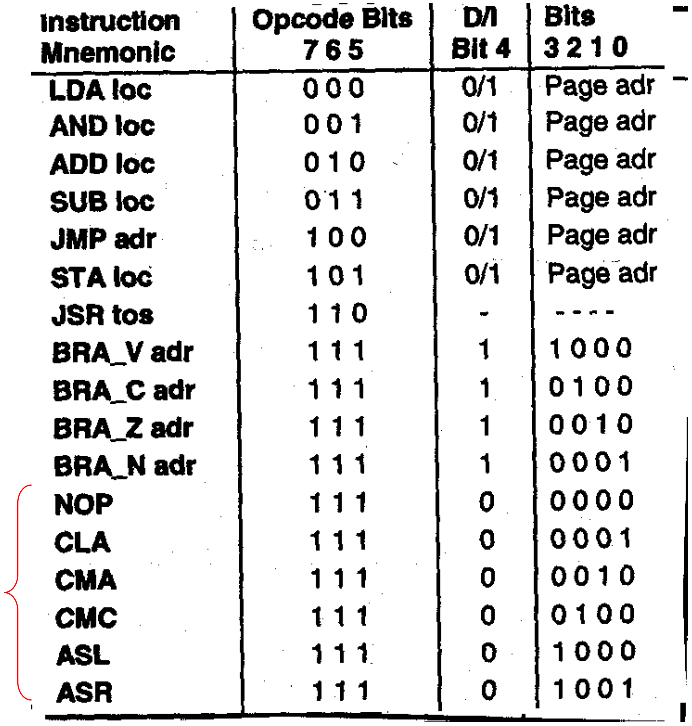
   JSR, BRA_V, BRA_C,
   BRA_Z, BRA_N

3. NO Address Instructions 1-Byte Instructions

   NOP, CLA, CMA, CMC,
   ASL, ASR

Instruction Set Description.

Instruction Mnemonic	Brief Description
LDA loc	Load AC w/(contents of loc)
A ND loc	AND AC w/(contents of loc)
ADD loc	Add (contents of loc) to AC
SUB loc	Sub (contents of loc) from AC
JMB adr	Jump to adr
STA loc	Store AC in contents of loc
JSR tos	Subroutine to tos
BRA-V adr	Branch to adr if V
BRA-C adr	Branch to adr if C
BRA-Z adr	Branch to adr if Z
BRA-N adr	Branch to adr if N
NOP	No operation
CLA	Clear AC
CMA	Complement AC
CMC	Complement carry
ASL	Arith shift left
ASR	Arith shift right

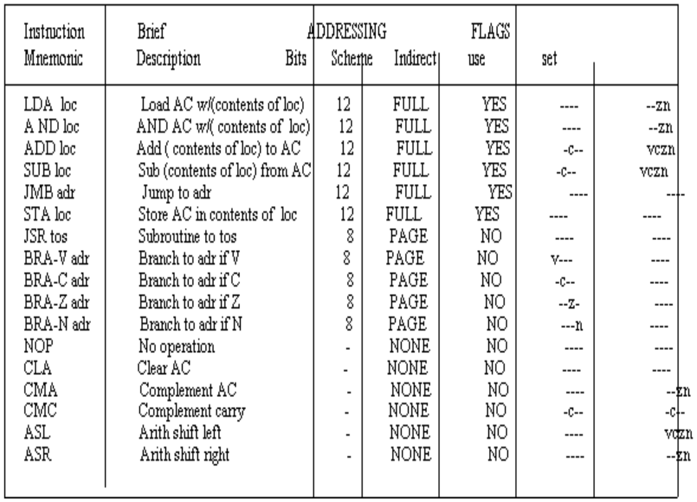


Instruction Mnemonic	Opcode Bits 7 6 5	D/I Bit 4	Bits 3 2 1 0
LDA loc	0 0 0	0/1	Page adr
AND loc	0 0 1	0/1	Page adr
ADD loc	0 1 0	0/1	Page adr
SUB loc	0 1 1	0/1	Page adr
JMP adr	1 0 0	0/1	Page adr
STA loc	1 0 1	0/1	Page adr
JSR tos	1 1 0	-	- - - -
BRA_V adr	1 1 1	1	1 0 0 0
BRA_C adr	1 1 1	1	0 1 0 0
BRA_Z adr	1 1 1	1	0 0 1 0
BRA_N adr	1 1 1	1	0 0 0 1
NOP	1 1 1	0	0 0 0 0
CLA	1 1 1	0	0 0 0 1
CMA	1 1 1	0	0 0 1 0
CMC	1 1 1	0	0 1 0 0
ASL	1 1 1	0	1 0 0 0
ASR	1 1 1	0	1 0 0 1

- Load and store operations
- Arithmetic & logical operations
- *jmp* and *branch* instructions

Single Byte Instructions

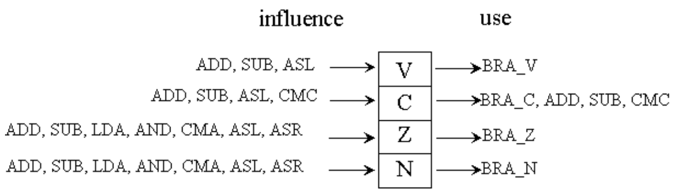
Instruction Set Description.



Instruction	Brief	ADDRESSING			FLAGS	
Mnemonic	Description	Bits	Scheme	Indirect	use	set
LDA loc	Load AC w/(contents of loc)	12	FULL	YES	----	--zn
A ND loc	AND AC w/(contents of loc)	12	FULL	YES	----	--zn
ADD loc	Add (contents of loc) to AC	12	FULL	YES	-c--	vczn
SUB loc	Sub (contents of loc) from AC	12	FULL	YES	-c--	vczn
JMB adr	Jump to adr	12	FULL	YES	----	----
STA loc	Store AC in contents of loc	12	FULL	YES	----	----
JSR tos	Subroutine to tos	8	PAGE	NO	----	----
BRA-V adr	Branch to adr if V	8	PAGE	NO	v---	----
BRA-C adr	Branch to adr if C	8	PAGE	NO	-c--	----
BRA-Z adr	Branch to adr if Z	8	PAGE	NO	--z-	----
BRA-N adr	Branch to adr if N	8	PAGE	NO	---n	----
NOP	No operation	-	NONE	NO	----	----
CLA	Clear AC	-	NONE	NO	----	----
CMA	Complement AC	-	NONE	NO	----	--zn
CMC	Complement carry	-	NONE	NO	-c--	-c--
ASL	Arith shift left	-	NONE	NO	----	vczn
ASR	Arith shift right	-	NONE	NO	----	--zn

- CPU contains *V C Z N* flags
  - V: Overflow
  - C: Carry out
  - Z: Zero
  - N: Negative
- Instructions use and/or influence these flags

Status Register (Flags)


influence

use

ADD, SUB, ASL → V → BRA_V

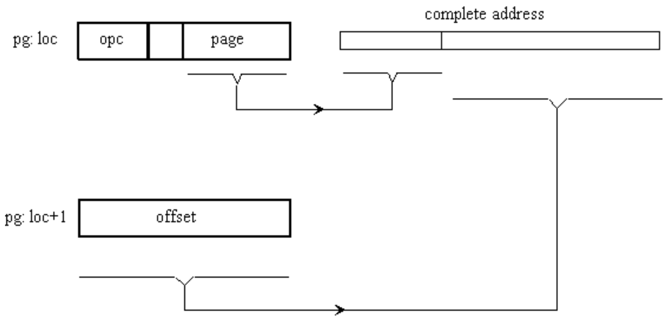
ADD, SUB, ASL, CMC → C → BRA_C, ADD, SUB, CMC

ADD, SUB, LDA, AND, CMA, ASL, ASR → Z → BRA_Z

ADD, SUB, LDA, AND, CMA, ASL, ASR → N → BRA_N

- Arithmetic instructions influence all flags
- Branch instructions use corresponding flags
- Shift instructions influence all flags

Full Addressing


pg: loc

opc

page

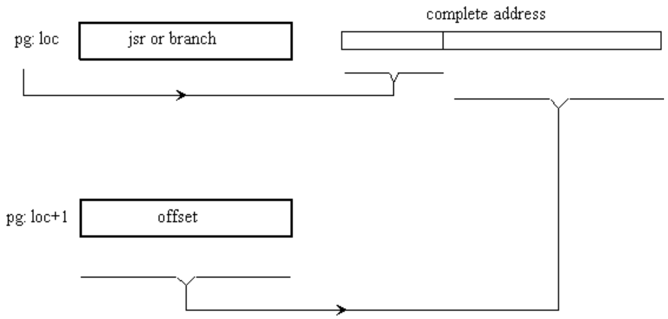
complete address

pg: loc+1

offset

- Full address instructions use two bytes
- Right hand side of first byte is Page #
- Second byte contains offset
- Bit 4 is Direct / Indirect indicator

Page Addressing



- Page address instructions use two bytes
- All of first byte is used by opcode
- Page part of address uses current page
- Second byte is the offset

Addressing Example

	Memory		
5:0D	11110100		BRA_C
5:0E	6A		6A
5:0F			

BRANCH TO 6A if Carry is set Else GoTo 5:0F

OpCode (Branch if C=1) = 1111_0100

c=0 : Next instruction from 5:0f
c=1 : Next instruction from 5:6A

- **Branching is done within current page only**

Addressing (JSR)

	MEMORY
	...
PC-> 5:11	JSR
5:12	3 3
5:13	INSTR AFTER JSR
...	...
5:33	0 0 0 0 0 0 0 0
5:34	SUBROUTINE CODE
...	...
5:55	JMP Indirect
5:56	3 3
5:57	...

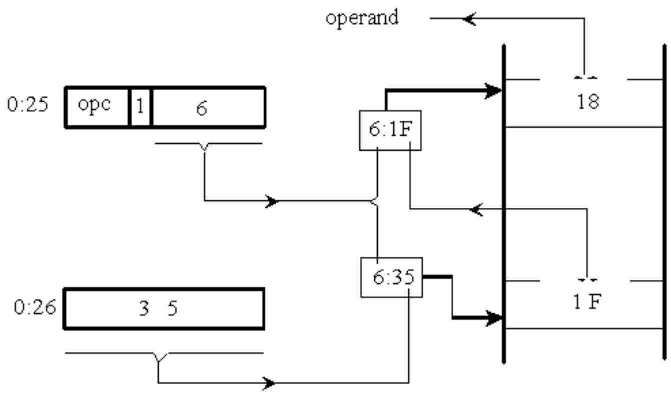
BEFORE JSR

	MEMORY
	...
5:11	JSR
5:12	3 3
5:13	INSTR AFTER JSR
...	...
5:33	1 3
PC-> 5:34	SUBROUTINE CODE
...	...
5:55	JMP Indirect
5:56	3 3
5:57	...

AFTER JSR

- **Store *jsr* return address at *tos***
- **Begin subroutine at *tos*+1**
- **Use indirect *jmp* to *tos* for return from subroutine**

Indirect Addressing



- Indirect addressing effects offset only
- Indirect Address = 6:35 → This Memory Byte Contains 1F
- Actual Address Accessed ==>> 6:1F

General CPU_Description Uilities

Basic Utilities (Developed Before)

```
PACKAGE basic_utilities IS
TYPE integers IS ARRAY (0 TO 12) OF INTEGER;
FUNCTION fgl (w, x, gl : BIT) RETURN BIT;
FUNCTION feq (w, x, eq : BIT) RETURN BIT;
PROCEDURE bin2int (bin : IN BIT_VECTOR; int :
OUT INTEGER);
PROCEDURE int2bin (int : IN INTEGER; bin :
OUT BIT_VECTOR);
PROCEDURE apply_data (SIGNAL target : OUT
BIT_VECTOR (3 DOWNTO 0);
CONSTANT values : IN integers; CONSTANT
period : IN TIME);
END basic_utilities;
```

Par- Utilities

```
LIBRARY cmos;
USE cmos.basic_utilities.ALL;
--
PACKAGE par_utilities IS
FUNCTION "XOR" (a, b : qit) RETURN qit ;
--
FUNCTION "AND" (a, b : qit_vector) RETURN
qit_vector;

FUNCTION "OR" (a, b : qit_vector) RETURN
qit_vector;
```

```
  FUNCTION "NOT" (a : qit_vector) RETURN
  qit_vector;
  --
  SUBTYPE nibble IS qit_vector (3 DOWNTO 0);
  SUBTYPE byte IS qit_vector (7 DOWNTO 0);
  SUBTYPE twelve IS qit_vector (11 DOWNTO
  0);
  --
  SUBTYPE wired_nibble IS wired_qit_vector (3
  DOWNTO 0);
  SUBTYPE wired_byte IS wired_qit_vector (7
  DOWNTO 0);
  SUBTYPE wired_twelve IS wired_qit_vector (11
  DOWNTO 0);
  --
  SUBTYPE ored_nibble IS ored_qit_vector (3
  DOWNTO 0);
  SUBTYPE ored_byte IS ored_qit_vector (7
  DOWNTO 0);
  SUBTYPE ored_twelve IS ored_qit_vector (11
  DOWNTO 0);
  --
CONSTANT zero_4 : nibble := "0000";
CONSTANT zero_8 : byte := "00000000";
CONSTANT zero_12 : twelve := "000000000000";
  --
  FUNCTION add_cv (a, b : qit_vector; cin : qit)
  RETURN qit_vector;
  FUNCTION sub_cv (a, b : qit_vector; cin : qit)
  RETURN qit_vector;
  --
  FUNCTION set_if_zero (a : qit_vector)
  RETURN qit;
  --
  END par_utilities;
```

```
PACKAGE body par_utilities IS
FUNCTION "XOR" (a, b : qit) RETURN qit IS
CONSTANT qit_or_table : qit_2d :=
                               ( ('0','1','1','X'),
                                 ('1','0','0','X'),
                                 ('1','0','0','X'),
                                ('X','X','X','X'));
BEGIN
RETURN qit_or_table (a, b);
END "XOR";

FUNCTION "AND" (a,b : qit_vector) RETURN
qit_vector IS
VARIABLE r : qit_vector (a'RANGE);
BEGIN
   loop1: FOR i IN a'RANGE LOOP
           r(i) := a(i) AND b(i);
   END LOOP loop1;
RETURN r;
END "AND";
--
FUNCTION "OR" (a,b: qit_vector) RETURN
qit_vector IS
VARIABLE r: qit_vector (a'RANGE);
BEGIN
   loop1: FOR i IN a'RANGE LOOP
      r(i) := a(i) OR b(i);
   END LOOP loop1;
RETURN r;
END "OR";
--
FUNCTION "NOT" (a: qit_vector) RETURN
qit_vector IS
VARIABLE r: qit_vector (a'RANGE);
```

```
BEGIN
      loop1: FOR i IN a'RANGE LOOP
         r(i) := NOT a(i);
      END LOOP loop1;
RETURN r;
END "NOT";
```

```
FUNCTION add_cv (a, b : qit_vector; cin : qit)
RETURN qit_vector IS
--left bits are sign bit
VARIABLE r, c: qit_vector (a'LEFT + 2
DOWNTO 0);
-- two extra bits in r are: msb for overflow, next
carry
VARIABLE a_sign, b_sign: qit;
BEGIN
a_sign := a(a'LEFT); b_sign := b(b'LEFT);
r(0) := a(0) XOR b(0) XOR cin;
c(0) := ((a(0) XOR b(0)) AND cin) OR (a(0) AND
b(0));
FOR i IN 1 TO (a'LEFT) LOOP
r(i) := a(i) XOR b(i) XOR c(i-1);
c(i) := ((a(i) XOR b(i)) AND c(i-1)) OR (a(i) AND
b(i));
END LOOP;
r(a'LEFT+1) := c(a'LEFT);
IF a_sign = b_sign AND r(a'LEFT) /= a_sign
THEN r(a'LEFT+2) := '1'; --overflow
ELSE r(a'LEFT+2) := '0'; END IF;

RETURN r;
END add_cv;
```

```
FUNCTION sub_cv (a, b : qit_vector; cin : qit)
RETURN qit_vector IS
VARIABLE not_b : qit_vector (b'LEFT
DOWNTO 0);
VARIABLE not_c : qit;
VARIABLE r : qit_vector (a'LEFT + 2
DOWNTO 0);
BEGIN
not_b := NOT b; not_c := NOT cin;
r := add_cv (a, not_b, not_c);

RETURN r;
END sub_cv;
```

```
FUNCTION set_if_zero (a : qit_vector)
RETURN qit IS
VARIABLE zero : qit := '1';
BEGIN
FOR i IN a'RANGE LOOP
IF a(i) /= '0' THEN zero := '0'; EXIT;
END IF;
END LOOP; RETURN zero;
END set_if_zero;
```

```
END par_utilities;
```

- *add_cv* adds its operands creates *c* and *v* bits → r = a + b + Cin
- Put overflow in leftmost result bit
- Put carry to the right of overflow

- Sub_Cv performs the Subtraction ==>> r = a - (b + Cin)

Op-Code Definitions

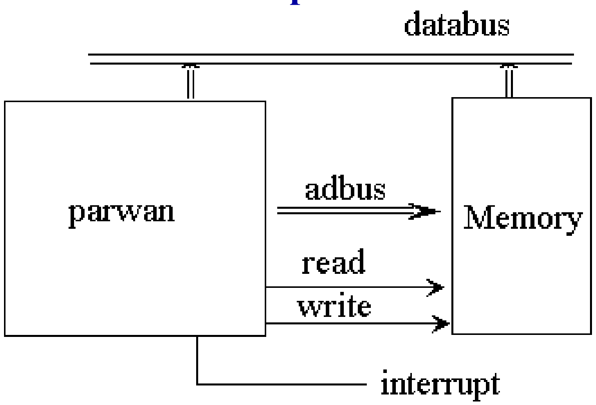
```
LIBRARY cmos;
   USE cmos.basic_utilities.ALL;
--
PACKAGE par_parameters IS
CONSTANT single_byte_instructions : qit_vector
   (3 DOWNTO 0) := "1110";
CONSTANT cla : qit_vector (3 DownTo 0) := "0001";
CONSTANT cma : qit_vector (3 DownTo 0) := "0010";
CONSTANT cmc : qit_vector (3 DownTo 0) := "0100";
CONSTANT asl : qit_vector (3 DownTo 0) := "1000";
CONSTANT asr : qit_vector (3 DownTo 0) := "1001";
CONSTANT jsr : qit_vector (2 DownTo 0) := "110";
CONSTANT bra : qit_vector (3 DownTo 0) := "1111";
CONSTANT indirect : qit := '1';
CONSTANT jmp : qit_vector (2 DownTo 0) := "100";
CONSTANT sta : qit_vector (2 DownTo 0) := "101";
CONSTANT lda : qit_vector (2 DownTo 0) := "000";
CONSTANT ann : qit_vector (2 DownTo 0) := "001";
CONSTANT add : qit_vector (2 DownTo 0) := "010";
CONSTANT sbb : qit_vector (2 DownTo 0) := "011";
END par_parameters;
```

- Assign appropriate names to opcodes
- *par_parameters* is used for readability

General CPU Description

1. Libraries
2. Interface
3. Architecture

Interface Description



- *Use a Single Process to Describe General Machine Operation*

```
LIBRARY cmos;
USE cmos.basic_utilities.ALL;
LIBRARY par_library;
USE par_library.par_utilities.ALL;
USE par_library.par_parameters.ALL;
--
```

```
ENTITY par_central_processing_unit IS
GENERIC (read_high_time, read_low_time,
                  write_high_time, write_low_time
                                    : TIME := 2 US;
                         cycle_time : TIME := 4 US;
                       run_time : TIME := 140 US);
PORT (  clk : IN qit;
           interrupt : IN qit;
           read_mem, write_mem : OUT qit;
           databus : INOUT wired_byte BUS :=
                                 "ZZZZZZZZ";
           adbus : OUT twelve );
END par_central_processing_unit;
```

- Make packages visible
- **Databus can be driven by Parwan and Memory**
- Use wiring resolution function
- Generic parameters specify relative read/write cycle time
- Pseudo Code is First Used To Behaviorally Describe Architecture

```
Architecture behavioral of par_central_processing_unit IS
 BEGIN
    PROCESS
       Declare necessary variables;
    BEGIN
          IF NOW > run_time THEN WAIT; END IF;
          IF interrupt = '1' THEN
                  Handle  interrupt;
          ELSE -- no interrupt  → Instruction Fetch
              Read first byte into byte1, increment pc;
              IF byte1 (7 Downto 4) =
                         single_byte_instructions THEN
                 Execute single_byte instruction;
              ELSE -- two-byte instructions
                 Read second byte into byte2,
                 increment pc;          -- 2nd Byte Fetch
             IF byte1 (7 Downto 5) = jsr THEN
                    Execute jsr instruction; --  byte2
                                               -- offset address
             ELSIF byte1 (7 DOWNTO 4) = bra
             THEN Execute bra instructions; --  byte2 has
                                                -- offset address
             ELSE -- all other two-byte instructions
               IF byte1 (4) = indirect THEN
                  Use byte1 and byte2 to get address;
               END IF; -- ends indirect
               IF byte1 (7 DOWNTO 5) = jmp THEN
                          Execute jmp instruction,
               ELSIF byte1 (7 DOWNTO 5) = sta THEN
                          Execute sta instruction, write ac;
               ELSE -- read operand for lda, and, add, sub
                          Read memory onto databus ;
                          Execute lda, and, add, and sub;
                          Disconnect memory from databus;
```

```
                END IF; -- jmp / sta / lda, and, add, sub
              END IF; -- jsr/bra /other double-byte instructions
              END IF; -- single-byte / double-byte
          END IF; -- interrupt / otherwise
     END PROCESS;
  END behavioral;
```

. Coding of Individual Instructions

```
-- Declaring necessary variables
VARIABLE pc : twelve;
VARIABLE ac, byte1, byte2 : byte;
VARIABLE v, c, z, n : qit;
VARIABLE temp : qit_vector (9 DOWNTO 0);
```

Handle interrupt

```
pc := zero_12;   -- Interrupt Handling Routine is
                 -- Located at Memory Address 0

WAIT FOR cycle_time;

-- Read first byte into byte1; increment pc ;

adbus <= pc;          -- Start A Memory Read Cycle

read_mem <= '1';

WAIT FOR read_high_time; -- Memory Access Delay

byte1 := byte (databus); -- databus is Type-Cast to byte

read_mem <= '0';

WAIT FOR read_low_time; -- Prevents OverWriting

pc := inc (pc);
```

Memory READ CYCLE

1. Put *address* on address bus
2. Wait half a clock cycle
3. Read data bus
4. Remove read request

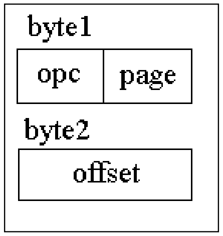
Execute single byte instructions

```
CASE byte1 (3 DOWNTO 0) IS

   When cla =>   ac := zero_8;  z := '1';

   When cma => ac := NOT ac;

                 IF ac = zero_8 Then z := '1'; End IF;
                 n := ac (7);

   When cmc =>  c := NOT c;

   When asl   =>  c := ac (7);

                ac := ac (6 DownTo 0) & '0';

                 n := ac (7);

                 IF c /= n THEN v := '1'; END IF;

   When asr =>   ac := ac (7) & ac (7 DOWNTO 1);

                 IF ac = zero_8 Then z := '1'; End IF;
                 n := ac (7);
   When OTHERS => NULL;
END CASE;
```

- Handing *single_byte instructions*, *cla*, *cma*, *cmc*, *asl* and *asr*
- Negative flag may be set for *cma*, *asl* and *asr*
- Zero flag may be set for *cma*, *asl* and *asr*
- For *asl*, overflow occurs it bits 6 & 7 differ

Execute Two_byte instructions
Read second byte into *byte2*, increment pc

```
adbus <= pc;
read_mem <= '1';
WAIT FOR read_high_time;
byte2 := byte (databus);
read_mem <= '0';
WAIT FOR read_low_time;
pc := inc (pc);
```


byte1

opc | page

byte2

offset

- **Reading byte from memory**
- ***Read_memory* stays high for half a clock**
- **Memory releases the bus in the second half**
- **Right half of *byte1* has page for full address**
- ***Byte2* now has the offset of address**

Execute *jsr* instruction, *byte2* has address

```
databus <= wired_byte (pc (7 DOWNTO 0) ); --
Offset Part of PC is
                                                  --
Written to DATA Bus
adbus (7 DOWNTO 0) <= byte2;  -- Offset Part of
Subroutine Address  is
                    -- Placed on Address Bus
(Page Address already there
Write_mem <= '1';
```

```
WAIT FOR write_high_time;
write_mem <= '0';

WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
pc (7 DOWNTO 0) := inc (byte2);
```

- **Handling *jsr***
- **Page part of *adbus* still points to the same instruction page**
- **Write *pc* to page location pointed by *byte2***
- **when writing is done, release the *databus***
- **Load *pc* to start from *byte2*+1 (*tos*)**

Execute *bra* instructions, address in *byte2*

```
IF
    ( byte1 (3) = '1' AND v = '1' ) OR
    ( byte1 (2) = '1' AND c = '1' ) OR
    ( byte1 (1) = '1' AND z = '1' ) OR
    ( byte1 (0) = '1' AND n = '1' )
THEN
    pc (7 DOWNTO 0) := byte2;
END IF;
```

- **Chock bits 3, 2, 1 and 0 against *v*, *c*, *z*, *n* flags**
- **Load *pc* with *byte2* if match is found**
- **Page part of *pc* still holds some page**

General.CPU_description.coding_individual_instructions
Use *byte1* and *byte2* to get address

```
adbus (11 DOWNTO 8) <= byte1 (3 Downto 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1';
Wait For read_high_time;
byte2 := byte (databus);
read_mem <= '0';
Wait For read_low_time;
```

- Use page of *byte1*, offset of *byte2*
- Form an address to fetch offset of operand
- Now *byte1* & *byte2* contain full operand address

General.CPU_description.coding_individual_instructions

Execute *jmp* instruction

```
pc := byte1 (3 Downto 0) & byte2;
```

- Load *pc* with full 12-bit address
- Could use two assignments instead of &

General.CPU_description.coding_individual_instructions

Execute *sta* instruction, write *ac*

```
adbus <= byte1 (3 DOWNTO 0) & byte2;
databus <= wired_byte (ac);
write_mem <= '1'; WAIT FOR write_high_time;
write_mem <= '0'; WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
```

- Put full address on *adbus*
- Put *ac* on *databus*
- Issue write, when done, release *databus*

General.CPU_description.coding_individual_instructions

Read memory onto *databus*

```
adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1';
WAIT FOR read_high_time;
CASE byte1 (7 DOWNTO 5) IS
    When lda => ac := byte (databus);
    When ann => ac := ac AND byte (databus);
    When add => temp := add_cv(ac, byte(databus), c);
                ac    := temp (7 DOWNTO 0);
                c     := temp (8);
                v     := temp (9);
    When sbb => temp := sub_cv(ac, byte(databus), c);
                ac    := temp (7 DOWNTO 0);
                c     := temp (8);
                v     := temp (9);
    When OTHERS => NULL;
END CASE;
IF ac = zero_8 THEN z := '1'; END IF;
n := ac (7);
read_mem <= '0';
WAIT FOR read_low_time;
```

- Full address on adbus
- Issue *read_mem*
- Perform *lda*, *ann*, *add* and *sbb*
- Arithmetic operations set *c* and *v* flags
- All operations set *z* and *n* flags

General.CPU_description .complete_behavioral

```
ARCHITECTURE behavioral OF
par_central_processing_unit IS
BEGIN
PROCESS
VARIABLE pc : twelve;
VARIABLE ac, byte1, byte2 : byte;
VARIABLE v, c, z, n : qit;
VARIABLE temp : qit_vector (9 DOWNTO 0);
VARIABLE pc : twelve;
VARIABLE ac, byte1, byte2 : byte;
VARIABLE v, c, z, n : qit;
VARIABLE temp : qit_vector (9 DOWNTO 0);
BEGIN
IF NOW > run_time THEN WAIT; END IF;
IF interrupt = '1' THEN
pc := zero_12;
WAIT FOR cycle_time;
ELSE -- no interrupt
adbus <= pc;
read_mem <= '1'; WAIT FOR read_high_time;
byte1 := byte (databus);
read_mem <= '0'; WAIT FOR read_low_time;
pc := inc (pc);
IF byte1 (7 DOWNTO 4) =
single_byte_instructions THEN
CASE byte1 (3 DOWNTO 0) IS
WHEN cla =>
ac := zero_8;
WHEN cma =>
ac := NOT ac;
IF ac = zero_8 THEN z := '1'; END IF;
```

```
n := ac (7);
WHEN cmc =>
c := NOT c;
WHEN asl =>
c := ac (7);
ac := ac (6 DOWNTO 0) & '0';
n := ac (7);
IF c /= n THEN v := '1'; END IF;
WHEN asr =>
ac := ac (7) & ac (7 DOWNTO 1);
IF ac = zero_8 THEN z := '1'; END IF;
n := ac (7);
WHEN OTHERS => NULL;
END CASE;
ELSE -- two-byte instructions
adbus <= pc;
read_mem <= '1'; WAIT FOR read_high_time;
byte2 := byte (databus);
read_mem <= '0'; WAIT FOR read_low_time;
pc := inc (pc);
```

Complete Behavioral Description

```
IF byte1 (7 DOWNTO 5) = jsr THEN
databus <= wired_byte (pc (7 DOWNTO 0) );
adbus (7 DOWNTO 0) <= byte2;
write_mem <= '1'; WAIT FOR write_high_time;
write_mem <= '0'; WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
pc (7 DOWNTO 0) := inc (byte2);
ELSIF byte1 (7 DOWNTO 4) = bra THEN
IF ( byte1 (3) = '1' AND v = '1' ) OR ( byte1 (2) =
'1' AND c = '1' ) OR
( byte1 (1) = '1' AND z = '1' ) OR ( byte1 (0) = '1'
AND n = '1' )
THEN
pc (7 DOWNTO 0) := byte2;
END IF;
ELSE -- all other two-byte instructions
IF byte1 (4) = indirect THEN
adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO
0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1'; WAIT FOR read_high_time;
byte2 := byte (databus);
read_mem <= '0'; WAIT FOR read_low_time;
END IF; -- ends indirect
IF byte1 (7 DOWNTO 5) = jmp THEN
pc := byte1 (3 DOWNTO 0) & byte2;
ELSIF byte1 (7 DOWNTO 5) = sta THEN
adbus <= byte1 (3 DOWNTO 0) & byte2;
databus <= wired_byte (ac);
write_mem <= '1'; WAIT FOR write_high_time;
```

```
write_mem <= '0'; WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
ELSE -- read operand for lda, and, add, sub
adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO
0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1'; WAIT FOR read_high_time;
CASE byte1 (7 DOWNTO 5) IS
WHEN lda =>
ac := byte (databus);
WHEN ann =>
ac := ac AND byte (databus);
WHEN add =>
temp := add_cv (ac, byte (databus), c);
ac := temp (7 DOWNTO 0); c := temp (8); v :=
temp (9);
WHEN sbb =>
temp := sub_cv (ac, byte (databus), c);
ac := temp (7 DOWNTO 0); c := temp (8); v :=
temp (9);
WHEN OTHERS => NULL;
END CASE;
IF ac = zero_8 THEN z := '1'; END IF;
n := ac (7);
read_mem <= '0'; WAIT FOR read_low_time;
END IF; -- jmp / sta / lda, and, add, sub
END IF; -- jsr / bra / other double-byte
instructions
END IF; -- single-byte / double-byte
END IF; -- interrupt / otherwise
END PROCESS;
END behavioral;
```

Entity Declaration:

```
ENTITY  Entity_Name  IS
    Generic (  Entity Parameters);
    PORT (  Definition of Input/Output Connectors );
END   [Entity]  Entity_Name;
```

Array Data Type Declaration Syntax

```
TYPE id Is Array ( Range_Constraint)   of   Type;
```

Conditionals:

```
1. IF condition  Then   statements  ; End IF;

2. IF condition Then statements; Else statements; End IF;

3. IF condition  Then  statements ; Elsif condition  Then
   statements ; Elsif condition  Then  statements ; ……
   ………… Elsif condition   Then  statements
   [Else statements ;] End IF;
```

CASE -Statement

```
CASE  Expression  is
      when  value => statements;
      when  value1 | value2| ...|valuen => statements ;
      when  discrete range of values   => statements ;
      when  others  => statements ;
End  CASE;
```

LOOPs

```
Loop_Label:  LOOP
                statements;
             End LOOP Loop_Label;
```

```
Loop_Label: FOR Loop_Variable  in  range  LOOP
               statements;
            End LOOP Loop_Label;
```

```
Loop_Label: WHILE  Condition  LOOP
               statements;
            End LOOP Loop_Label;
```

WAIT-Statement:

- WAIT;
- WAIT ON Signal_List;
- WAIT UNTIL Condition;
- WAIT FOR Time_Out_Expression;
- WAIT on Sig_List until Cond for Time_ Exp;

FUNCTIONS:

```
FUNCTION  function_Name(Input Parameter_List)
RETURN type  IS
         { Declarations}
Begin
         Function Algorithm;
    RETURN  Expression;
End function_Name;
```

Procedures:

```
PEOCEDURE  Proc_Name (Interface_List)   IS
         { Declarations}
Begin
         Procedure  Algorithm;
End Procedure_Name;
```

PROCESS:

```
[Process_Label:] PROCESS(Sensitivity_List)
                    {Process_Declarations;}
                 Begin
                    Statements;
                 END Process [Process_Label];
```

BLOCK: Required

```
Block_Label: Block (Guard_Condition)
                {Block_Declarations;}
             Begin
                Concurrent_Statements;
             END Block  Block_Label;
```

Concurrent Signal Assignment: {Unaffected may replace any Wave$_i$}

```
Label: target <= [Guarded] [Transport]
                Wave1  when  Cond1  Else
                Wave2  when  Cond2  Else
                ……………………………….
                Waven-1  when  Condn-1  Else
                Waven ;
```

```
With  Expression  Select
      target <= [Guarded] [Transport]
                Wave1  when  Choice1 ,
                Wave2  when  Choice2 ,
                ……………………………….
                Waven-1  when  Choicen-1 ,
                Waven when  OTHERS ;
```

Package:

```
Package Package_Name  IS
         Declarations{Constants, Signals, Types,
         Components & Subprograms};
End Package_Name;
```

```
Package Body Package_Name IS
              Subprogram_Bodies;
End Package_Name;
```

Use Clause:

```
use Library_name.Package_Name.<ids | ALL> ;
Ex              Use  work.utils.all ;
                Use Compass_Lib. Packag1.ALL ;
```

Component Declaration:

```
Component   Component_Name  [IS]
     Generic (  Component Parameters);
     PORT (  Definition of Input/Output Connectors );
END Component;
```

Configuration Specification:

```
For  <Instances_Specs> USE ENTITY  entity binding  ;
Ex: For u1,u7, u9 : AND2 Use Entity work.AND2(DF);
     For Others : AND2 Use Entity work.AND2(ALG);
     For ALL : OR2  Use  Entity   work.OR2(DF)
                    Generic Map(T_entity => T_Compt)
                     Port Map(entity_Port => Compt_Port) ;
```

Component Instantiation:

Required

```
Label :   <component_name>
                    Generic Map(association_list)
                    Port Map(association_list) ;
Note:  <association list> : either positional, or named
  named:        (compt_Name => Instance_Name)
```

Generate Statement:

Required

```
1.   Label   : For identifier in  Range Generate
                  Concurrent Statements;
              End Generate ;

2.   Label   : IF condition Generate
                  Concurrent Statements;
              End Generate;
```

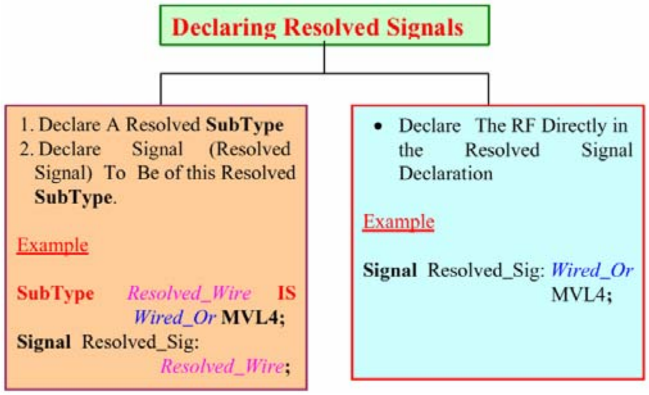
Assertion Statement:

```
Assert     a_Should_be_Valid_Condition
Report  " Message if Condition Not True "
Severity <severity_level ∊ {note warning, error, failure}>;
```

```
Entity entity_name Is
           Generic (Constant_Parameters);
           Port        (Interface_List);
End [entity] entity_name ;
```

```
Architecture Arch_Type  of  entity_name  Is
           Declarations( Types, Objects, Components, Subprograms)
Begin
           Concurrent Statements / Constructs
End [architecture] Arch_Type ;
```

Declaring Resolved Signals



Declaring Resolved Signals

1. Declare A Resolved **SubType**
2. Declare Signal (Resolved Signal) To Be of this Resolved **SubType**.

Example

```
SubType     Resolved_Wire     IS
                 Wired_Or MVL4;
Signal  Resolved_Sig:
                   Resolved_Wire;
```

- Declare The RF Directly in the Resolved Signal Declaration

Example

```
Signal  Resolved_Sig: Wired_Or
                     MVL4;
```

Alias Declaration:

```
Alias  <new_name>: Subtype  is  <existing name of same
    type and range>  ;
Ex: Alias opcode : Bit_Vector (3 to 0) is Instr(7 downto4);
     Alias Sign_Bit : Bit  is Accumulator(32) ;
```

Disconnect Statement:

```
Signal X : WX_Vector(7 downTo 0) BUS ;
DISCONNECT X : WX_Vector after 50 ns ;
```