Digital System Synthesis
Introduction to VHDL: Part I

Dr. Mahdi Abbasi

Computer Engineering Department

Bu Ali Sina University

Outline …

- Hardware description languages
- VHDL terms
- Design Entity
- Design Architecture
- VHDL model of full adder circuit
- VHDL model of 1’s count circuit
- Structural modeling of 4-bit comparator
- Design parameterization using Generic
- Test Bench example

… Outline

- Signal assignment
- VHDL Objects
- Behavioral modeling in VHDL

Hardware Description Languages

- **HDLs 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
- **Two widely-used HDLs today**
  - **VHDL:** VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
  - **Verilog** (from Cadence, now IEEE standard)

Styles in VHDL

- **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)

VHDL Terms …

- **Entity:**
  - All designs are expressed in terms of entities
  - Basic building block in a design
- **Ports:**
  - Provide the mechanism for a device to communication with its environment
  - Define the names, types, directions, and possible default values for the signals in a component's interface
- **Architecture:**
  - All entities have an architectural description
  - Describes the behavior of the entity
  - A single entity can have multiple architectures (behavioral, structural, …etc)
- **Configuration:**
  - A configuration statement is used to bind a component instance to an entity-architecture pair.
  - Describes which behavior to use for each entity

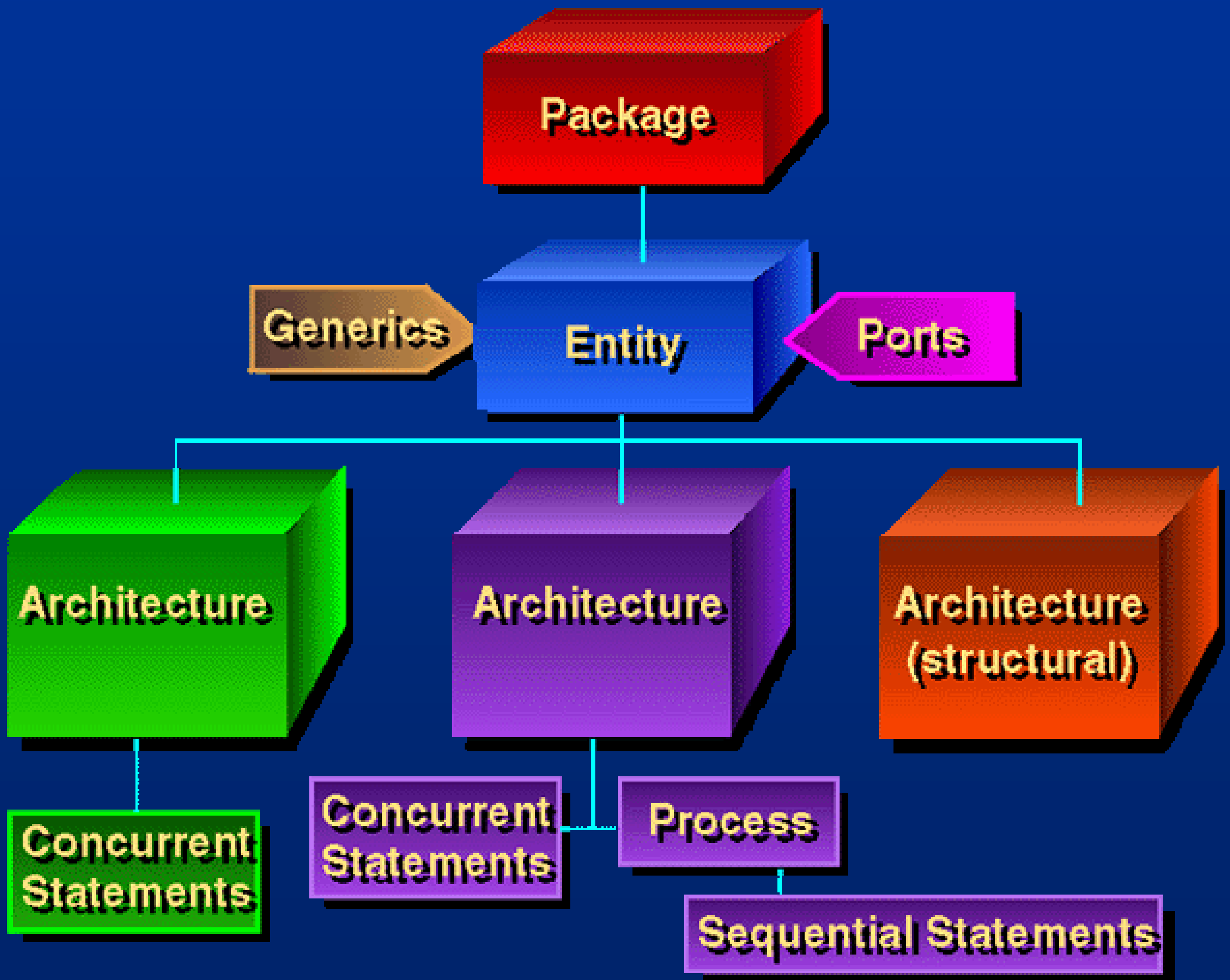
… VHDL Terms …

- **Generic:**
  - A parameter that passes information to an entity
  - Example: for a gate-level model with rise and fall delay, values for the rise and fall delays passed as generics
- **Process:**
  - Basic unit of execution in VHDL
  - All operations in a VHDL description are broken into single or multiple processes
  - Statements inside a process are processed sequentially
- **Package:**
  - A collection of common declarations, constants, and/or subprograms to entities and architectures.

VHDL Terms …

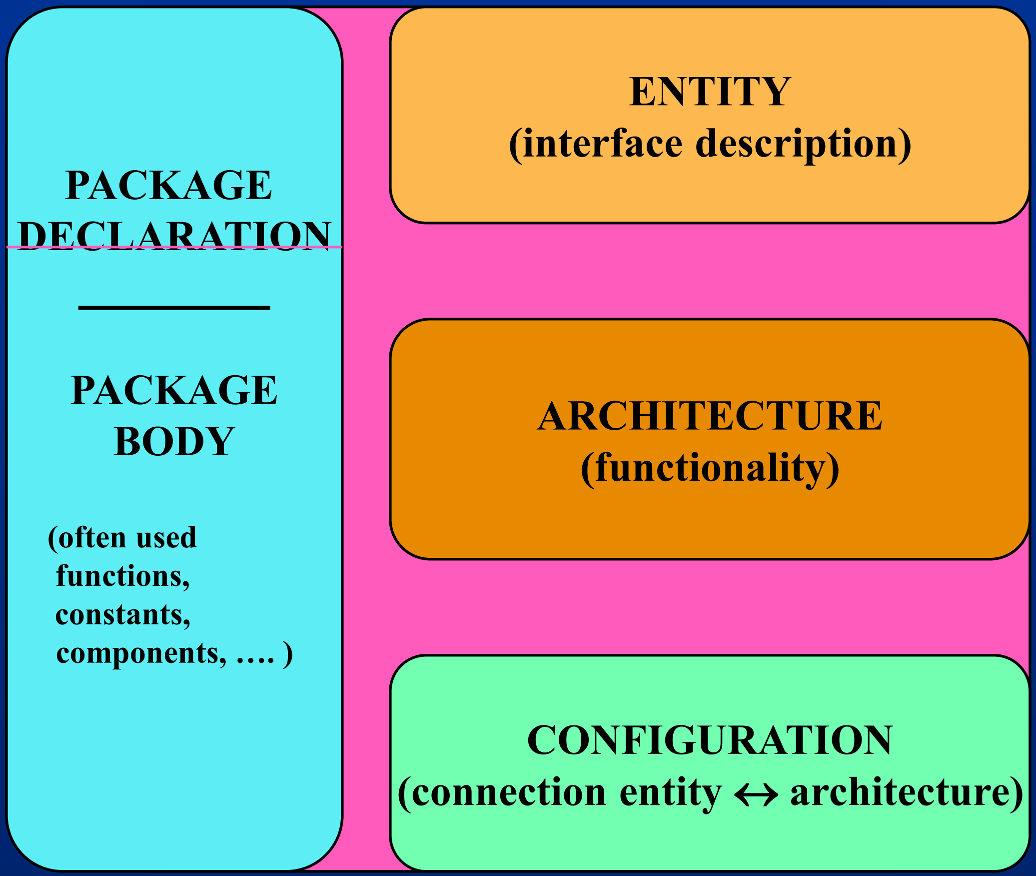
- **Attribute:**
  - Data attached to VHDL objects or predefined data about VHDL objects
  - Examples:
    - maximum operation temperature of a device
    - Current drive capability of a buffer
- **VHDL is NOT Case-Sensitive**
  - Begin = begin = beGiN
- **Semicolon " ; " terminates declarations or statements.**
- **After a double minus sign (--) the rest of the line is treated as a comment**

VHDL Models …



Package
Generics
Entity
Ports
Architecture
Architecture
Architecture (structural)
Concurrent Statements
Concurrent Statements
Process
Sequential Statements

… VHDL Models



PACKAGE DECLARATION
PACKAGE BODY
(often used functions, constants, components, ….)
ENTITY
(interface description)
ARCHITECTURE
(functionality)
CONFIGURATION
(connection entity ↔ architecture)

Design Entity …

- **In VHDL, the name of the system is the same as the name of its entity.**
- **Entity comprises two parts:**
  - *parameters* of the system as seen from outside such as bus-width of a processor or max clock frequency
  - *connections* which are transferring information to and from the system (system's inputs and outputs)
- **All parameters are declared as generics and are passed on to the body of the system**
- **Connections, which carry data to and from the system, are called *ports*. They form the second part of the entity.**

Entity Examples …

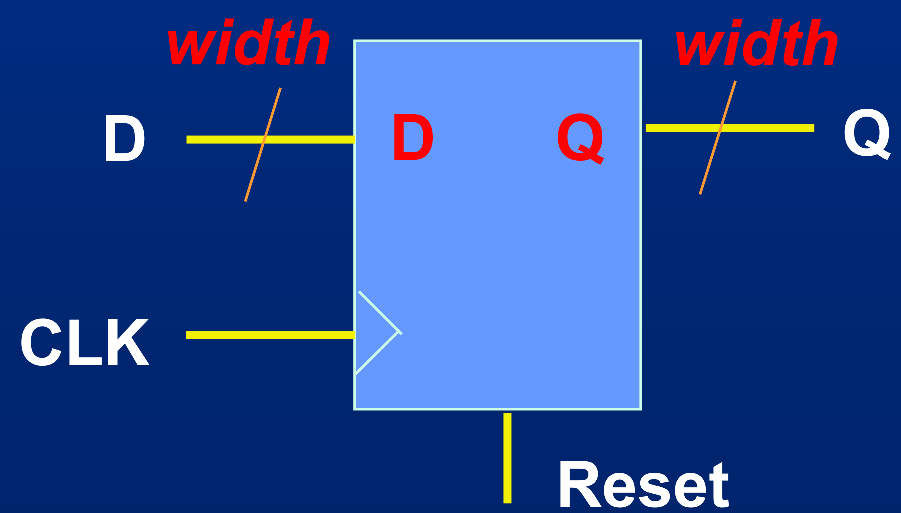
- Entity FULLADDER is

```
 -- Interface description of FULLADDER
port (            A, B, C: in bit;
             SUM, CARRY: out bit);
end FULLADDER;
```

… Entity Examples

- Entity Register is

```
  -- parameter: width of the register
  generic (width: integer);
  --input and output signals
  port (  CLK, Reset: in bit;
          D: in bit_vector(1 to width);
          Q: out bit_vector(1 to width));
end Register;
```

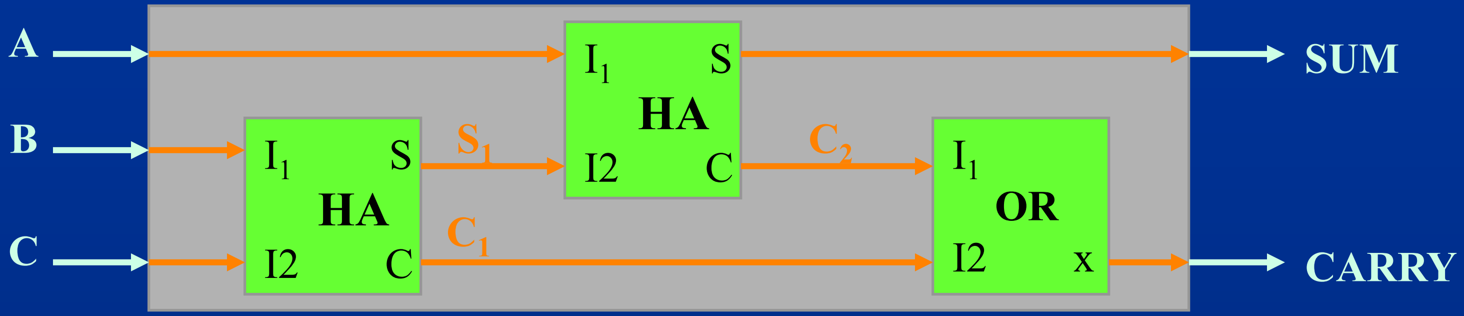


Architecture Examples: Behavioral Description

- Entity FULLADDER is
  port (A, B, C: in bit;
  SUM, CARRY: out bit);
  end FULLADDER;

- Architecture CONCURRENT of FULLADDER is
  begin
  SUM <= A xor B xor C after 5 ns;
  CARRY <= (A and B) or (B and C) or (A and C) after 3 ns;
  end CONCURRENT;

Architecture Examples: Structural Description …

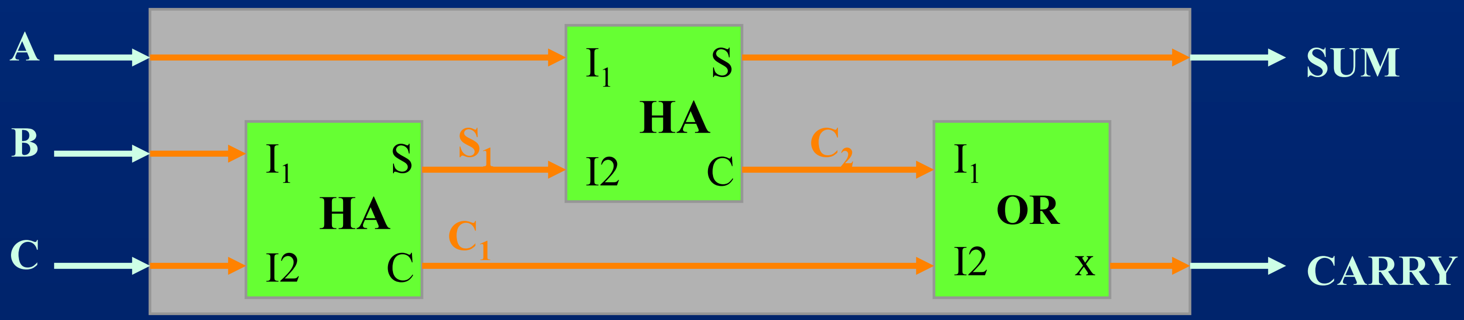



```
Entity HA is
PORT (I1, I2 : in bit; S, C : out bit);
end HA ;
Architecture behavior of HA is
begin
   S <= I1 xor I2;
   C <= I1 and I2;
end behavior;
```

```
Entity OR is
PORT (I1, I2 : in bit; X : out bit);
end OR ;
Architecture behavior of OR is
begin
   X <= I1 or I2;
end behavior;
```

Architecture Examples: Structural Description …

```
architecture STRUCTURAL of FULLADDER is
  signal S1, C1, C2 : bit;
  component HA
    port (I1, I2 : in bit; S, C : out bit);
  end component;
  component OR
    port (I1, I2 : in bit; X : out bit);
  end component;
begin
  INST_HA1 : HA     port map (I1 => B, I2 => C, S => S1, C => C1);
  INST_HA2 : HA     port map (I1 => A, I2 => S1, S => SUM, C => C2);
  INST_OR : OR  port map ( C2,  C1, CARRY);
end STRUCTURAL;
```



VHDL Predefined Operators

- **Logical Operators: NOT, AND, OR, NAND, NOR, XOR, XNOR**
  - Operand Type: Bit, Boolean, Bit_vector
  - Result Type: Bit, Boolean, Bit_vector
- **Relational Operators: =, /=, <, <=, >, >=**
  - Operand Type: Any type
  - Result Type: Boolean
- **Arithmetic Operators: +, -, *, /**
  - Operand Type: Integer, Real
  - Result Type: Integer, Real
- **Concatenation Operator: &**
  - Operand Type: Arrays or elements of same type
  - Result Type: Arrays
- **Shift Operators: SLL, SRL, SLA, SRA, ROL, ROR**
  - Operand Type: Bit or Boolean vector
  - Result Type: same type

VHDL Reserved Words

abs	disconnect	label	package	
access	downto	library	Poll	units
after	linkage		procedure	until
alias	else	loop	process	use
all	elsif			variable
and	end	map	range	
architecture	entity	mod	record	wait
array	exit	nand	register	when
assert	new	rem	while	
attribute	file	next	report	with
begin	for	nor	return	xor
block	function	not	select	
body	generate	null	severity	
buffer	generic	of	signal	
bus	guarded	on	subtype	
case	if	open	then	
component	in	or	to	
configuration	inout	others	transport	
constant	is	out	type	

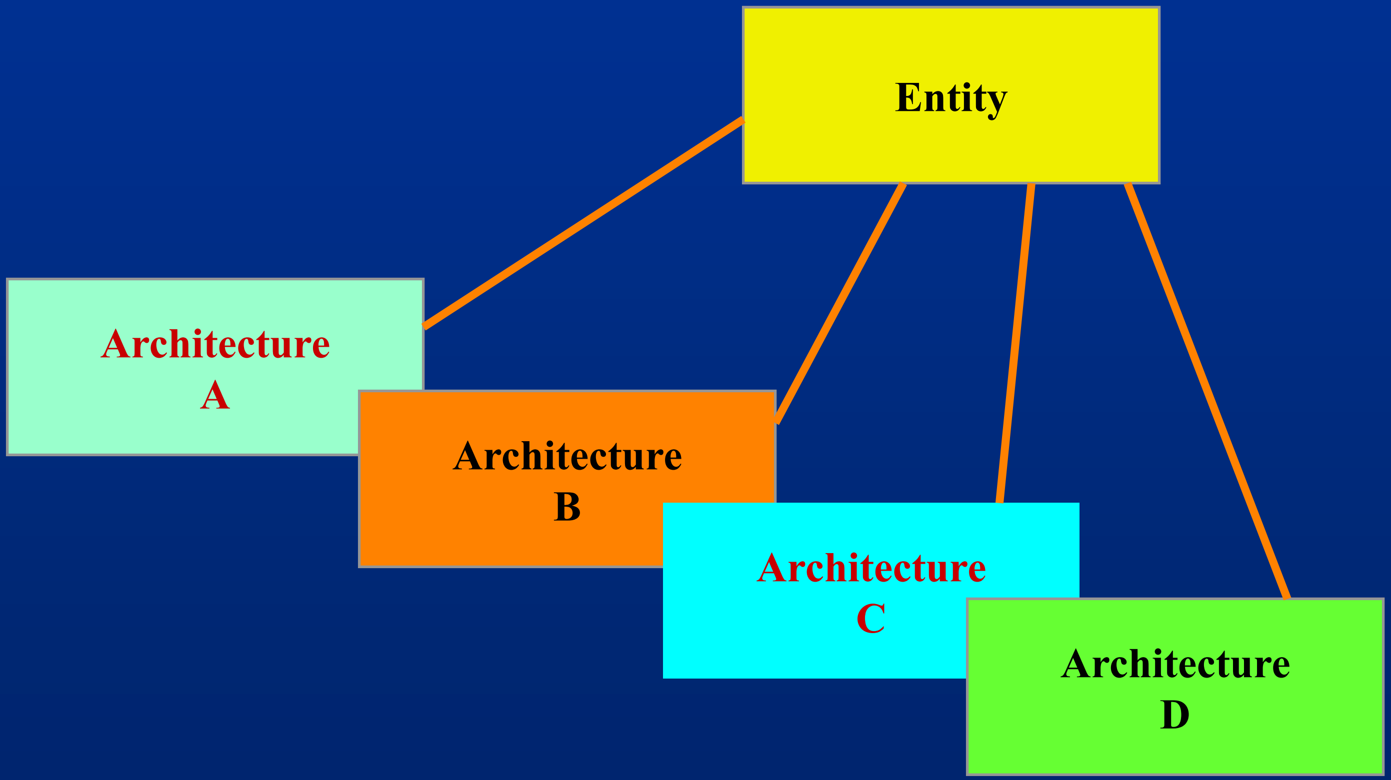
VHDL Language Grammar

- **Formal grammar of the IEEE Standard 1076-1993 VHDL language in BNF format**

  http://www.iis.ee.ethz.ch/~zimmi/download/vhdl93_syntax.html

One Entity Many Descriptions

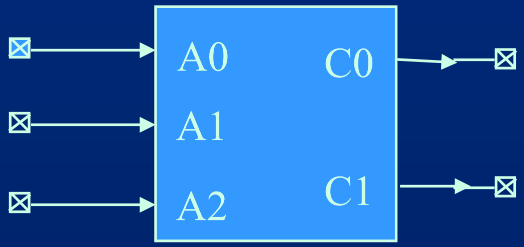
- A system (an *entity*) can be specified with different *architectures*



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

    $= A0 \oplus A1 \oplus A2$



Ones Count Circuit Interface Specification

1

2

DOCUMENTATION

3

```
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)
--   ____________________
--  | 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;
```

Ones Count Circuit Architectural Body: Data Flow

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

  $= A0 \oplus A1 \oplus A2$

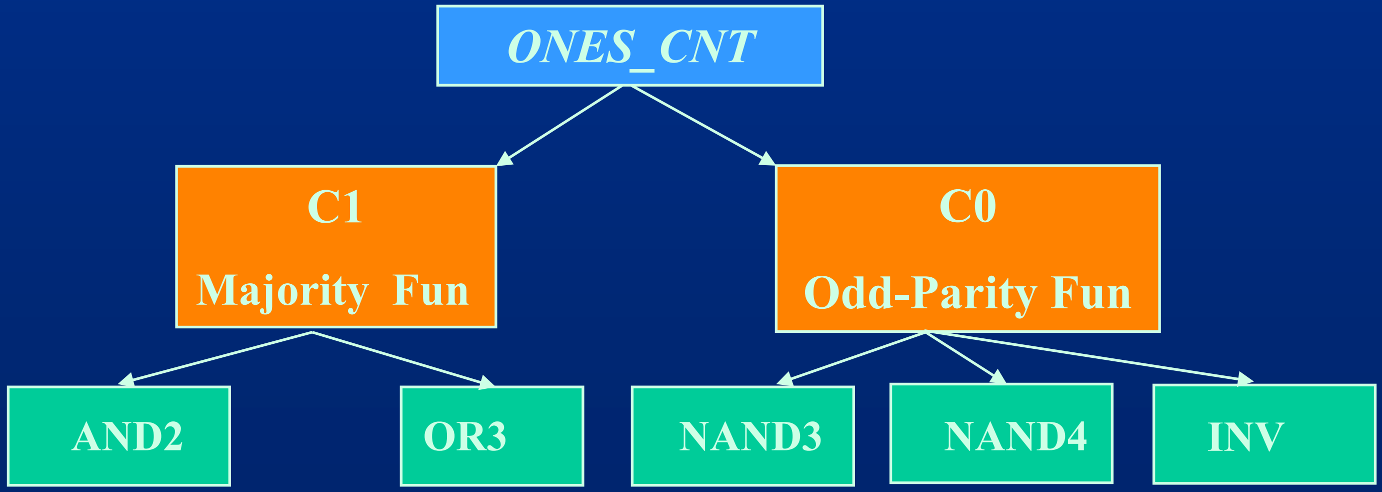
```
Architecture  Dataflow of  ONES_CNT  is
begin

   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 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));
--  C(0) <= A(2) xor A(1) xor A(0);

end Dataflow;
```

Ones Count Circuit Architectural Body: Structural …

- 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 Count Circuit Architectural Body: Structural …

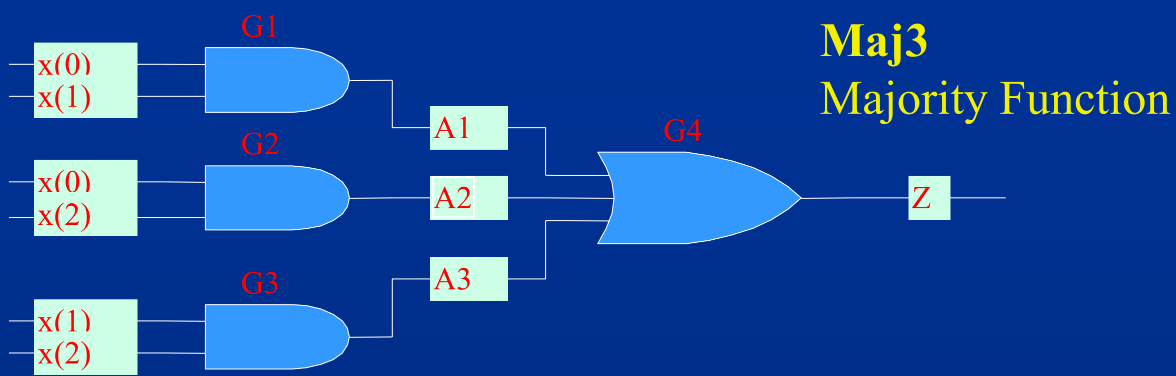
- Entity MAJ3 is

```
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 of 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

VHDL Structural Description of Majority Function

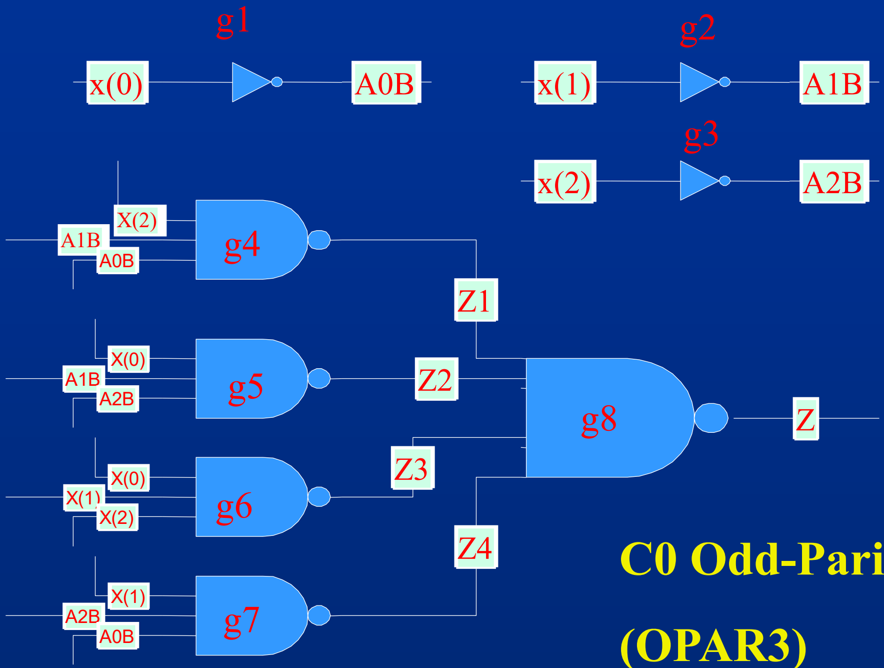
```
SIGNAL  A1, A2, A3:   BIT;
begin
-- Instantiate Gates
    g1: AND2 PORT MAP (X(0), X(1), A1);
    g2: AND2 PORT MAP (X(0), X(2), A2);
    g3: AND2 PORT MAP (X(1), X(2), A3);
    g4: OR3  PORT MAP (A1, A2, A3, Z);
end Structural;
```

Declare Maj3 Local Signals

Wiring of Maj3 Components

VHDL Structural Description of Odd Parity Function …

```
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 ;
```



C0 Odd-Parity

(OPAR3)

VHDL Structural Description of Odd Parity Function

```
SIGNAL A0B, A1B, A2B, 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;
```

VHDL Top Structural Level of Ones Count Circuit

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

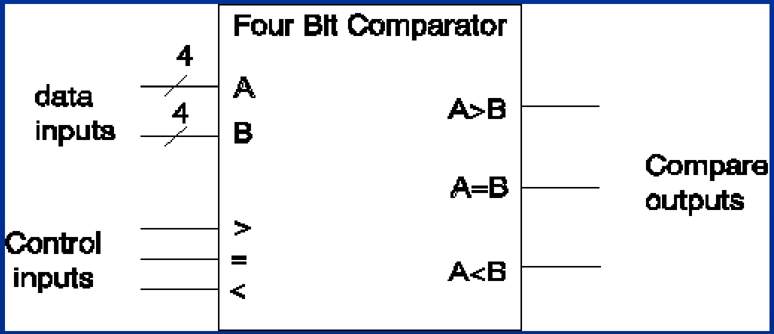
VHDL Behavioral Definition of Lower Level Components

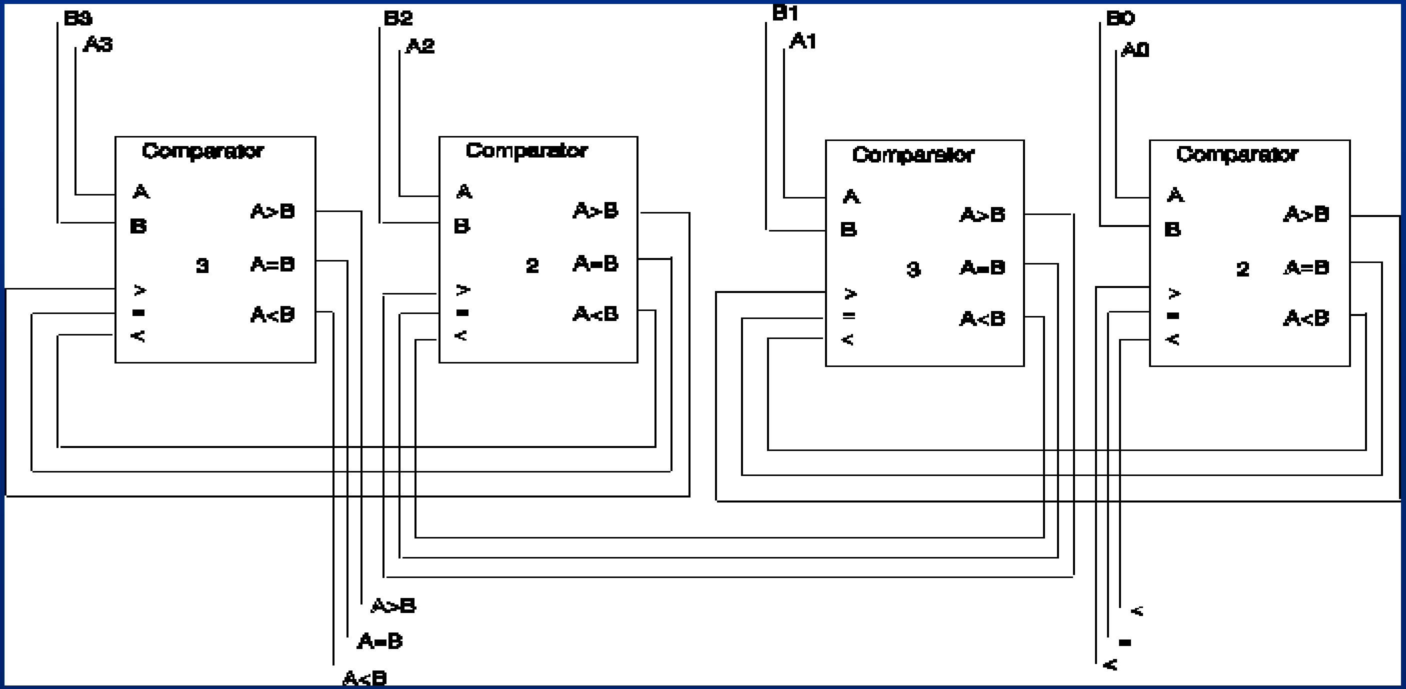
```
Entity INV is
        PORT( Ipt: in BIT;
         Opt: out BIT);
end INV;
Architecture behavior of INV is
begin
        Opt <= not Ipt;
end behavior;
```

```
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;
```

Other Lower Level Gates Are Defined Similarly

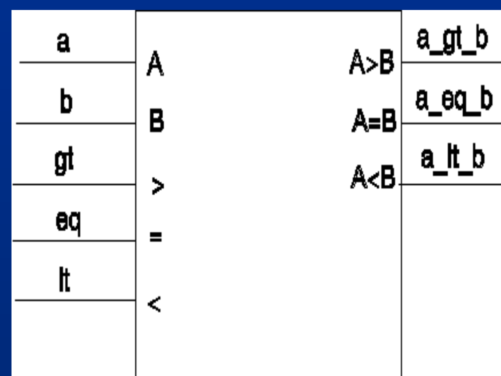
Structural 4-Bit Comparator





A 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



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

a>b

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

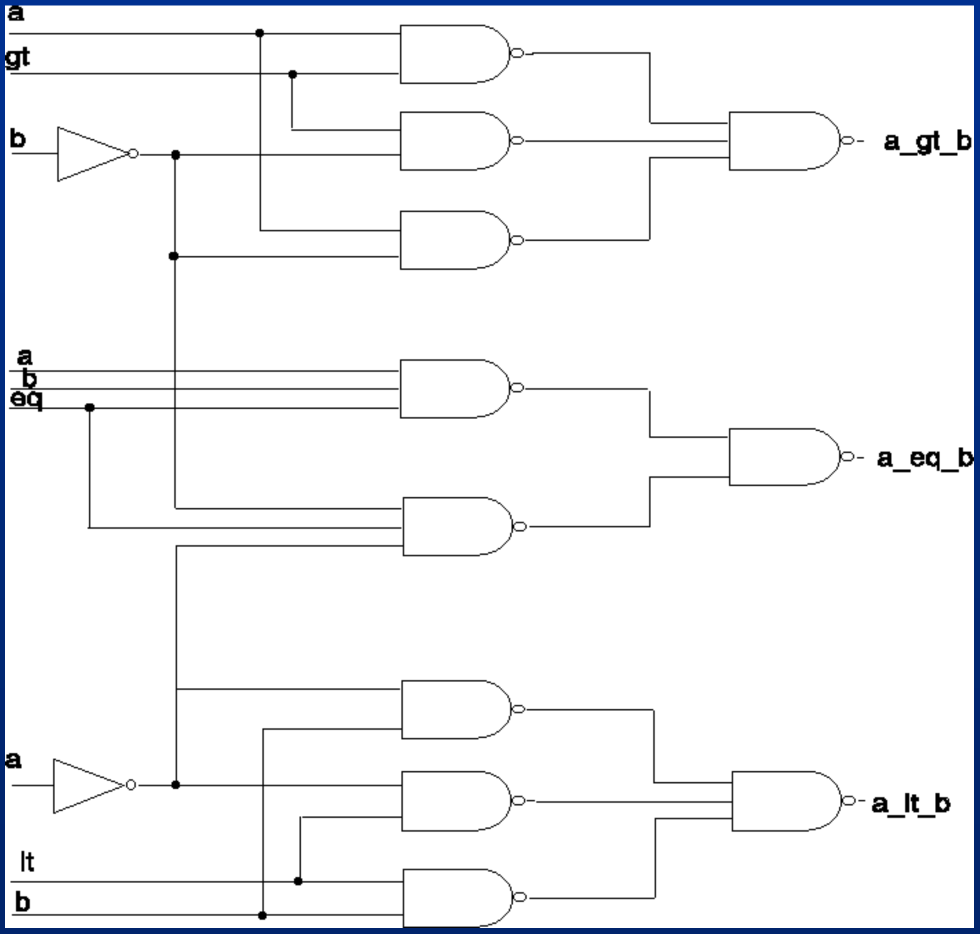
a=b

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

a<b

Structural Single-Bit Comparator

- Design uses basic components
- The less-than and greater-than outputs use the same logic



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 COMPONENT;
--  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;
```

… Structural Model of Single-Bit Comparator

```
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 gate_level;
```

Netlist Description of Single-Bit Comparator

```
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;
```

4-Bit Comparator Iterative Structural Wiring: "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
        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

    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;

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;
```

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

4-Bit Comparator: "IF …… Generate" Statement …

```
ARCHITECTURE iterative OF nibble_comparator IS
    --
    COMPONENT comp1
        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;
```

4-Bit Comparator: Alternative Architecture (Single Generate)

```
ARCHITECTURE Alt_iterative OF nibble_comparator IS
constant n: Positive :=4;
COMPONENT comp1
    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 ;
```

Design Parameterization …

- **GENERICs can pass design parameters**
- **GENERICs can include default values**
- **New versions of gate descriptions contain timing**

```
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;
```

… Design Parameterization …

```
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;
```

```
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;
```

Using Default values …

```
ARCHITECTURE default_delay 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 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);
```

No Generics Specified in Component Declarations

… Using Default values

```
-- 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*
- *If default values exist, they are used*

Assigning Fixed Values to Generic Parameters …

```
ARCHITECTURE fixed_delay OF bit_comparator IS
Component n1
Generic (tplh, tphl : Time); Port (i1: in Bit; o1: out Bit);
END Component;
Component n2
Generic (tplh, tphl : Time); Port (i1, i2: in Bit; o1: out Bit);
END Component;
Component n3
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);
```

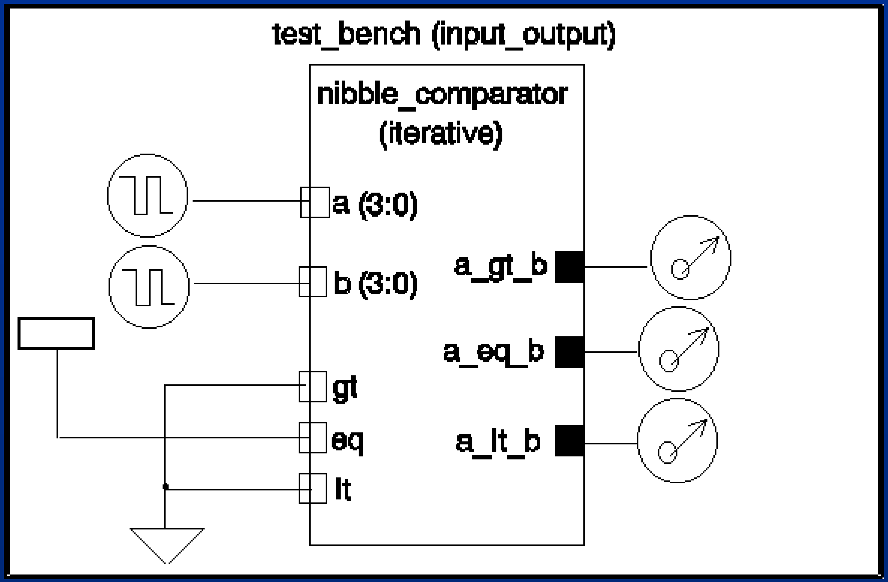
… Assigning Fixed Values to Generic Parameters

```
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);
g7 : n3 Generic Map (4 NS, 6 NS) PORT MAP (a, b, eq, im7);
g8 : n2 Generic Map (3 NS, 5 NS) PORT MAP (im6, im7, a_eq_b);
-- a_lt_b output
g9 : n2 Generic Map (3 NS, 5 NS) Port Map (im1, b, im8);
g10 : n2 Generic Map (3 NS, 5 NS) PORT MAP (im1, lt, im9);
g11 : n2 Generic Map (3 NS, 5 NS) PORT MAP (b, lt, im10);
g12 : n3 Generic Map (4 NS, 6 NS) PORT MAP (im8, im9, im10, a_lt_b);
END fixed_delay;
```

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

Structural Test Bench

- **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)



Testbench Example …

```
Entity nibble_comparator_test_bench IS
End nibble_comparator_test_bench ;
--
ARCHITECTURE input_output OF nibble_comparator_test_bench IS
--
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(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);
--
```

...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)
“0111" AFTER 6000 NS; -- a > b (need bit 3 only, best case)
END input_output;
```

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
```

… Signal Assignment

- **Selected Signal Assignment**

```
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.)

Signal Assignment Examples

Example: A 2x4 Decoder

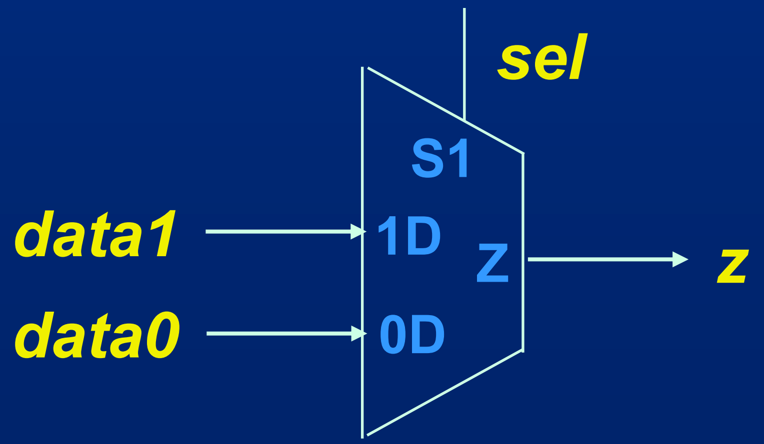
```
Signal  D  :  Bit_Vector(1 To 4) := "0000";
Signal  S0, S1  :  Bit;
.............................................
Decoder:   D <= "0001" after T  When S1='0' and S0='0' else
                "0010"  after T  When S1='0' else
                "0100" after T  When S0='0' else  "1000" ;
```

Example: 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
```

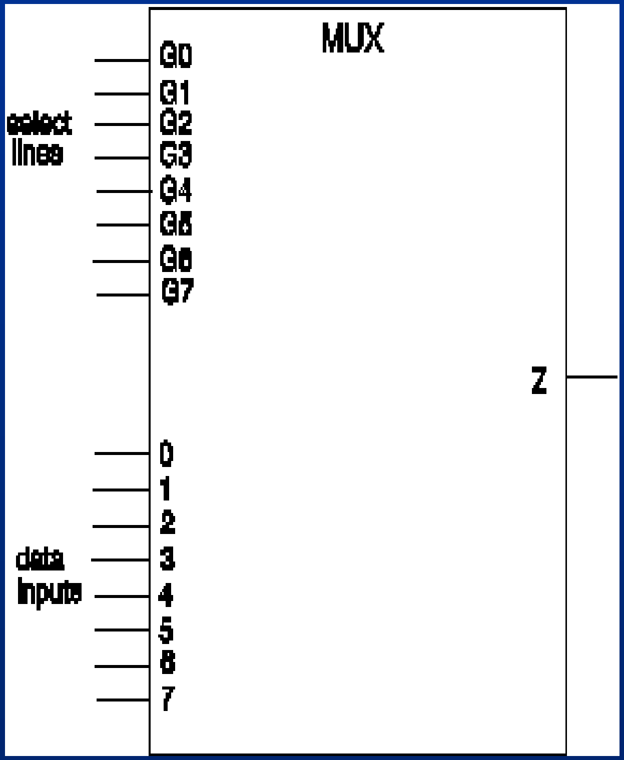
2x1 Multiplexer

```
Entity mux2_1 IS
    Generic (dz_delay: TIME := 6 NS);
    PORT (sel, data1, data0: IN BIT; z: OUT BIT);
END mux2_1;

Architecture dataflow OF mux2_1 IS
Begin
    z <= data1 AFTER dz_delay WHEN sel='1' ELSE
            data0 AFTER dz_delay;
END dataflow;
```



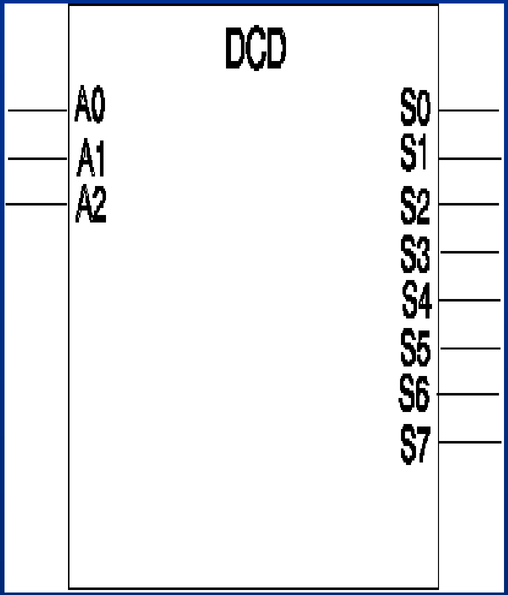
8x1 Multiplexer

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit, qit_vector
ENTITY mux_8_to_1 IS
PORT (i7, i6, i5, i4, i3, i2, i1, i0 : IN qit;
s7, s6, s5, s4, s3, s2, s1, s0 : IN qit; z : OUT qit );
END mux_8_to_1;
--
ARCHITECTURE dataflow OF mux_8_to_1 IS
SIGNAL sel_lines : qit_vector ( 7 DOWNTO 0);
BEGIN
sel_lines <= s7&s6&s5&s4&s3&s2&s1&s0;
WITH sel_lines SELECT
z <= '0' AFTER 3 NS WHEN "00000000",
i7 AFTER 3 NS WHEN "10000000" | "Z0000000",
i6 AFTER 3 NS WHEN "01000000" | "0Z000000",
i5 AFTER 3 NS WHEN "00100000" | "00Z00000",
i4 AFTER 3 NS WHEN "00010000" | "000Z0000",
i3 AFTER 3 NS WHEN "00001000" | "0000Z000",
i2 AFTER 3 NS WHEN "00000100" | "00000Z00",
i1 AFTER 3 NS WHEN "00000010" | "000000Z0",
i0 AFTER 3 NS WHEN "00000001" | "0000000Z",
'X' WHEN OTHERS;
END dataflow;
```



3-to-8 Decoder

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE : qit_vector
ENTITY dcd_3_to_8 IS
PORT (adr : IN qit_vector (2 DOWNTO 0);
so : OUT qit_vector (7 DOWNTO 0));
END dcd_3_to_8;
--
ARCHITECTURE dataflow OF dcd_3_to_8 IS
BEGIN
WITH adr SELECT
so <= "00000001" AFTER 2 NS WHEN "000",
"00000010" AFTER 2 NS WHEN "00Z" | "001",
"00000100" AFTER 2 NS WHEN "0Z0" | "010",
"00001000" AFTER 2 NS WHEN "0ZZ" | "0Z1" | "01Z" |
"011",
"00010000" AFTER 2 NS WHEN "100" | "Z00",
"00100000" AFTER 2 NS WHEN "Z0Z" | "Z01" | "10Z" |
"101",
"01000000" AFTER 2 NS WHEN "ZZ0" | "Z10" | "1Z0" |
"110",
"10000000" AFTER 2 NS WHEN "ZZZ" | "ZZ1" | "Z1Z" |
"Z11" | "1ZZ" | "1Z1" | "11Z" | "111",
"XXXXXXXX" WHEN OTHERS;
END dataflow;
```



VHDL Objects …

- **VHDL *OBJECT* : Something that can hold a value of a given *Data Type*.**
- **VHDL has 3 main 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*];

- *Obj_Class*: Constant, Variable, Signal, File
- <*id_list*>: ≥ 1 *identifier* (,)
- [*signal_kind*]: BUS, Register — Only for Signals
- [:= *expression*]: *Default Initial Value* (*not Optional for Constant Declarations*)

… 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	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 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.

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**

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**

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.

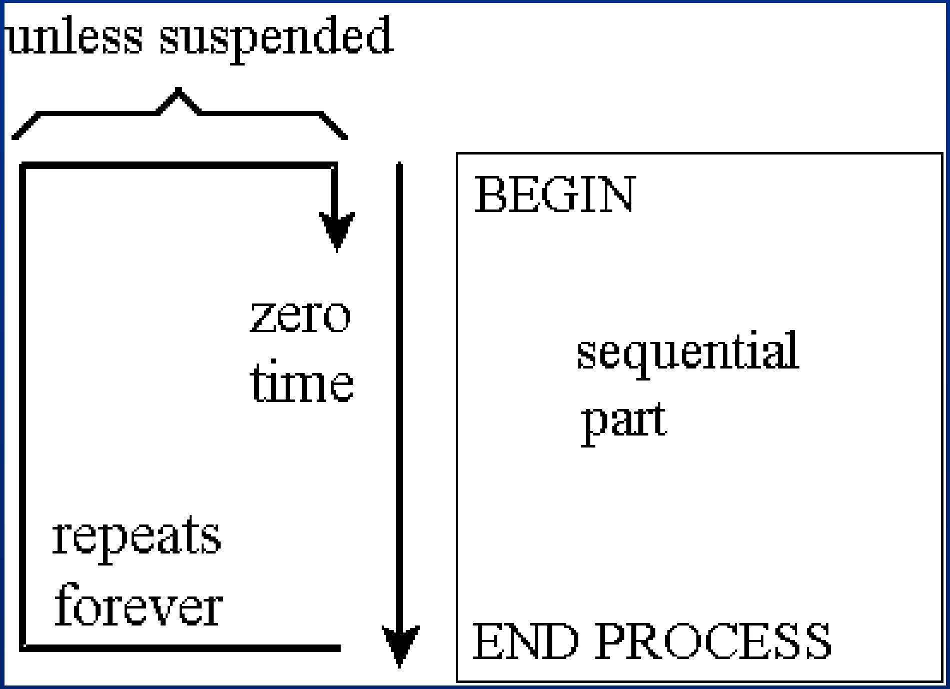
Optional *Optional*

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

Constant/Variables
No Signal Declarations Allowed

… Process Statement …

- **Unless sequential part is suspended**
  - It executes in zero real and delta time
  - It repeats itself forever



… Process Statement

- **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.**
- **Conditional and selective signal assignments are strictly concurrent and cannot be used in a process.**

Process Examples

```
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)

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

Assuming a `0` Initial Value of A,

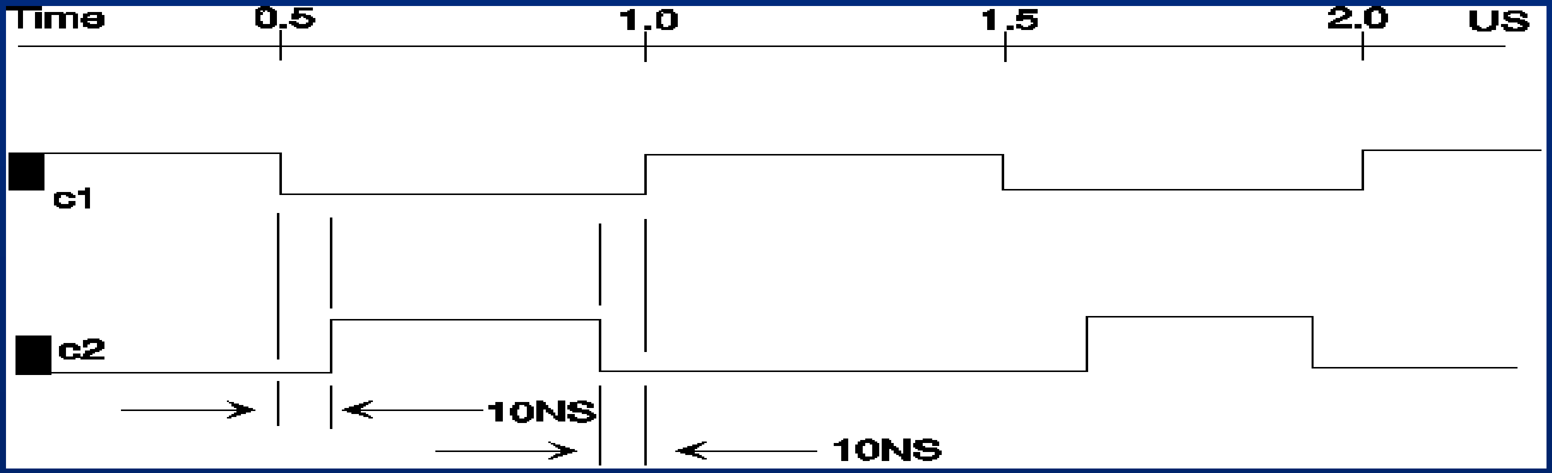
- First A is Scheduled to Have a Value `1` One Delta Time Later
- Thus, Upon Execution of IF_Statement, A Has a Value of `0` and *Action 2* will be Taken.
- If A *was Declared as a Process* Variable, *Action1* Would Have Been Taken

Wait Statement

- **Syntax of Wait Statement :**
  - WAIT; -- **Forever**
  - WAIT ON *Signal_List*; -- **On event on a signal**
  - WAIT UNTIL *Condition*; **-- until event makes condition true;**
  - WAIT FOR *Time_Out_Expression*;
  - WAIT FOR **0** *any_time_unit*; -- Process Suspended for 1 delta
- **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* **;**
  - **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***
  - Process Reactivated IF:
    - Event Occurred on the *Signal_List* while the *Condition* is True, *OR*
    - *Wait* Period Exceeds ``*Time_Expression* ``

Using Wait for Two-Phase Clocking

```
c1 <= not c1 after 500ns;
phase2: PROCESS
BEGIN
WAIT UNTIL c1 = '0';
WAIT FOR 10 NS;
c2 <= '1';
WAIT FOR 480 NS;
c2 <= '0';
END PROCESS phase2;
. . .
```



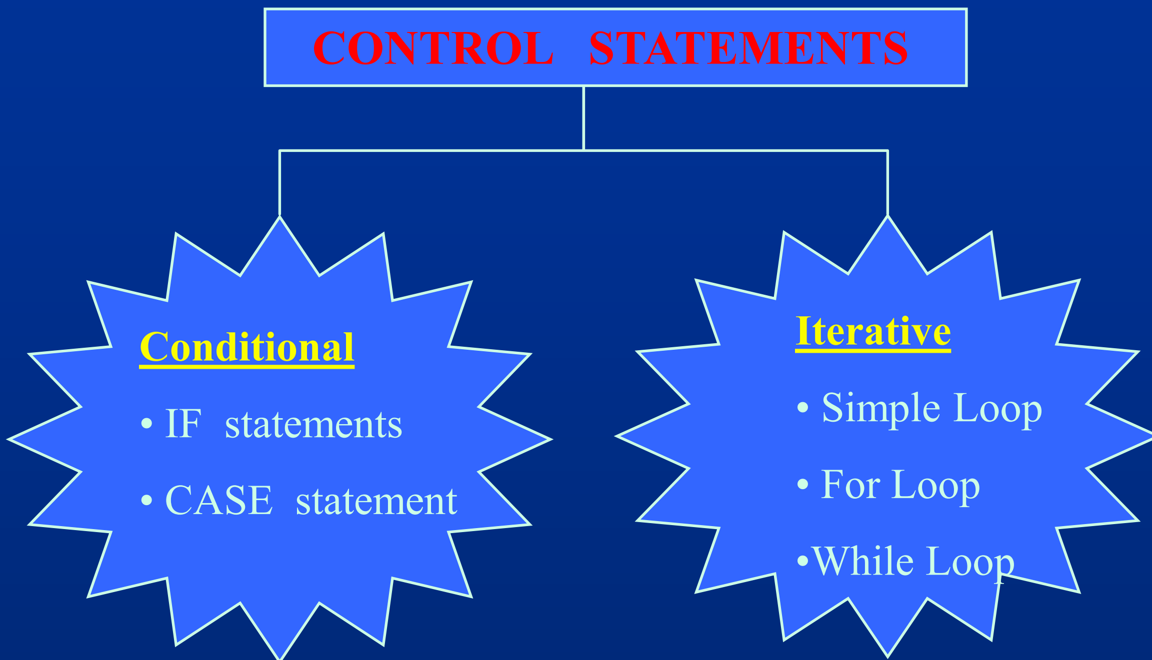
Positive Edge-Triggered D-FF Examples

```
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;
```

Sequential Statements



Conditional Control – IF Statement

- **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;
      End IF;
```

Conditional Control – Case Statement

- **Syntax**:

  (i) **CASE** *Expression* **is**

  when value **=>** *statements*;

  when value*1* | value*2* | ...| value*n* **=>** *statements*;

  when *discrete range of values* **=>** *statements*;

  when **others** **=>** *statements*;

  End CASE;

- 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.

Conditional Control – Case Statement

- 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*)

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

Ones Count Circuit Architectural Body: Behavioral (Truth Table)

```
Architecture  Truth_Table  of  ONES_CNT  is

begin
   Process(A)    -- Sensitivity List Contains only Vector A
    begin

       CASE A  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;
```

Loop Control …

- **Simple Loops**
- **Syntax:**

[Loop_Label:] LOOP

statements;

End LOOP [Loop_Label];

- **The Loop_Label is Optional**
- **The exit statement may be used to exit the Loop. It has two possible Forms:**
  - exit [Loop_Label]; -- This may be used in an if statement
  - exit [Loop_Label] when condition;

...Loop Control

```
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;
```

FOR Loop

- **Syntax**:

Need Not Be Declared

[Loop_Label]: **FOR** *Loop_Variable* **in** *range* **LOOP**

statements;

End LOOP *Loop_Label*;

```
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;
```

Ones Count Circuit Architectural Body: Behavioral (Algorithmic)

```
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;
```

WHILE Loop

- **Syntax**:

  [Loop_Label]: **WHILE** *condition* **LOOP**

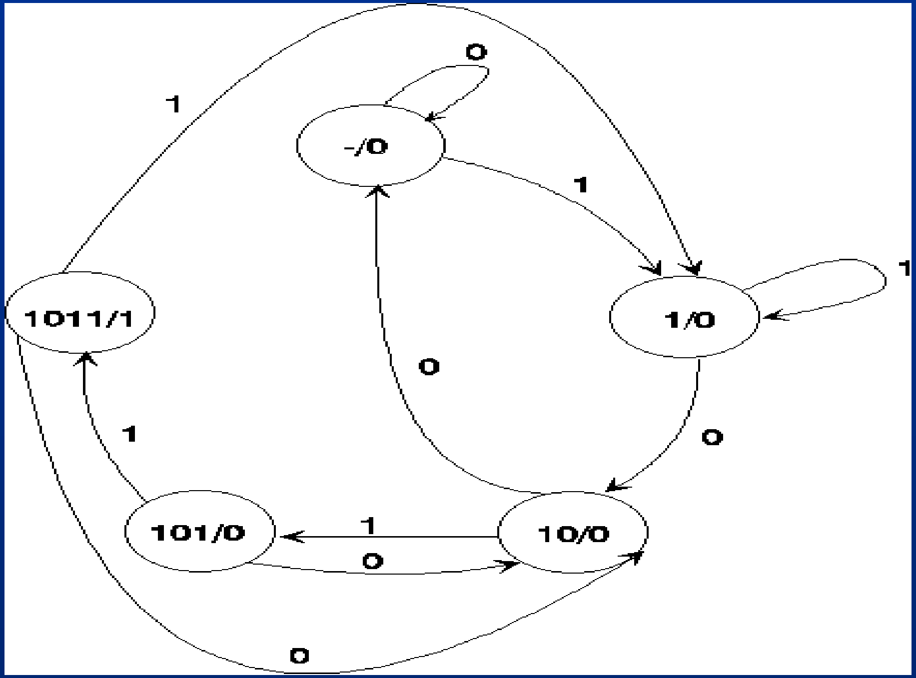
  statements;

  End LOOP *Loop_Label*;

```
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;
```

A Moore 1011 Detector using Wait

```
ENTITY moore_detector IS
PORT (x, clk : IN BIT;
z : OUT BIT);
END moore_detector;
```

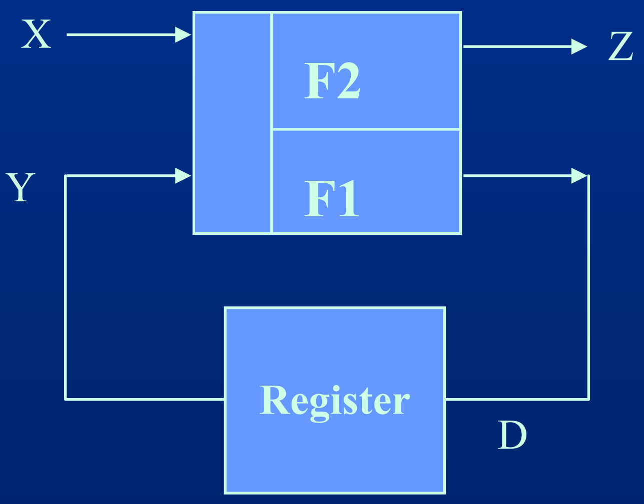


A Moore 1011 Detector

```
ARCHITECTURE most_behavioral_state_machine OF moore_detector IS
TYPE state IS (reset, got1, got10, got101, got1011);
SIGNAL current : state := reset;
BEGIN
PROCESS (clk)
BEGIN
IF (clk = '1' and CLK'Event) THEN
CASE current IS
WHEN reset =>
      IF x = '1' THEN current <= got1; ELSE current <= reset; END IF;
WHEN got1 =>
      IF x = '0' THEN current <= got10; ELSE current <= got1; END IF;
WHEN got10 =>
      IF x = '1' THEN current <= got101; ELSE current <= reset; END IF;
WHEN got101 =>
      IF x = '1' THEN current <= got1011; ELSE current <= got10; END IF;
WHEN got1011 =>
      IF x = '1' THEN current <= got1; ELSE current <= got10; END IF;
END CASE;
END IF;
END PROCESS;
z <= '1' WHEN current = got1011 ELSE '0';
END most_behavioral_state_machine;
```

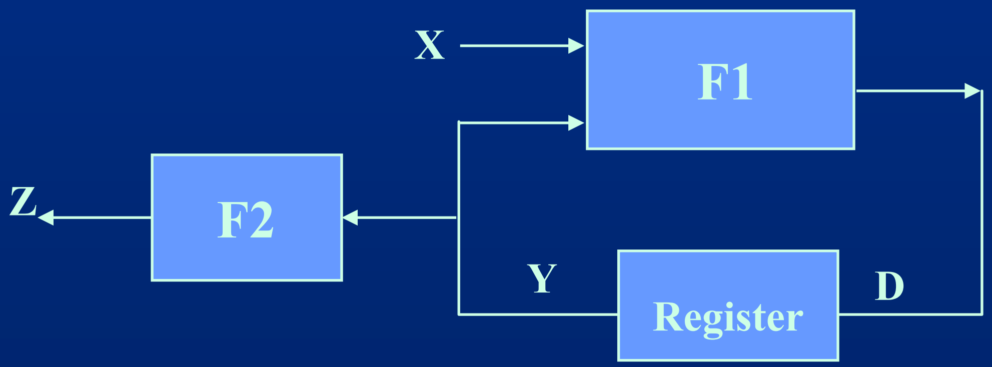
Generalized VHDL Mealy Model

```
Architecture  Mealy of  fsm  is
     Signal D, Y: Std_Logic_Vector( ...); -- Local Signals
Begin
Register: 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;
```



Generalized VHDL Moore Model

```
Architecture  Moore of  fsm  is
    Signal D, Y: Std_Logic_Vector( ...); -- Local Signals
Begin
Register: 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;
```

… FSM Example

```
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;
```

FSM Modeling Example

- It is required to design a sequential circuit using Mealy model that computes the equation Z=3*X-3, where X is an unsigned number that will be fed serially.

Present State	Next State, Z	
	X=0	X=1
S0 (B=3)	S1, 1	S3, 0
S1 (B=2)	S2, 0	S3, 1
S2 (B=1)	S2, 1	S4, 0
S3 (B=0)	S3, 0	S4, 1
S4 (C=1)	S3, 1	S5, 0
S5 (C=2)	S4, 0	S5, 1

FSM Modeling Example

```
entity Y3XM3SEQ is
port (Z: out bit;  X, Reset, CLK: in bit);
end Y3XM3SEQ;
Architecture Behavioral of Y3XM3SEQ is
Type state IS (S0, S1, S2, S3, S4, S5);
signal  CS, NS: state;
begin
process(CLK, Reset)
begin
  if (Reset='1') Then
     CS <= S0;
  elsif (Clk'EVENT and Clk = '1')  Then
    CS <= NS;
  end if;
end process;
```

FSM Modeling Example

```
process (X , CS)
begin
Z <= '0'; NS <= S0;
case CS is
when S0 =>  if (X='1') then  NS<=S3;   else  Z<='1'; NS<=S1; end if;
when S1 =>  if (X='1') then Z<='1'; NS<=S3; else NS<=S2; end if;
when S2 =>  if (X='1') then NS<=S4;  else  Z<='1'; NS<=S2; end if;
when S3 =>  if (X='1') then Z<='1'; NS<=S4;  else NS<=S3; end if;
when S4 =>  if (X='1') then NS<=S5;  else  Z<='1'; NS<=S3; end if;
when S5 =>  if (X='1') then Z<='1'; NS<=S5; else NS<=S4; end if;
end case;
end process;

end Behavioral;
```

FSM Test Bench

```
entity Y3XM3SEQ_TB is
end Y3XM3SEQ_TB;
Architecture Behavioral of Y3XM3_TB is
component Y3XM3SEQ is
port (Z: out bit;  X, Reset, CLK: in bit);
end component;
Signal CLK, Reset, X, Z: bit ;
begin

M1: Y3XM3SEQ port map (Z, X, Reset, CLK);

 CLK <= not CLK after 10 ps;
```

FSM Test Bench

```
process begin
 --Applying X=1
wait until (CLK='0'); Reset<='1';
wait until (CLK='0'); Reset<='0'; X<='1';
wait until (CLK='0'); X<='0';
wait until (CLK='0'); X<='0';
wait until (CLK='0'); X<='0';
--Applying X=5
wait until (CLK='0'); Reset<='1';
wait until (CLK='0'); Reset<='0'; X<='1';
wait until (CLK='0'); X<='0';
wait until (CLK='0'); X<='1';
wait until (CLK='0'); X<='0';
wait;
end process; end Behavioral;
```