EE595

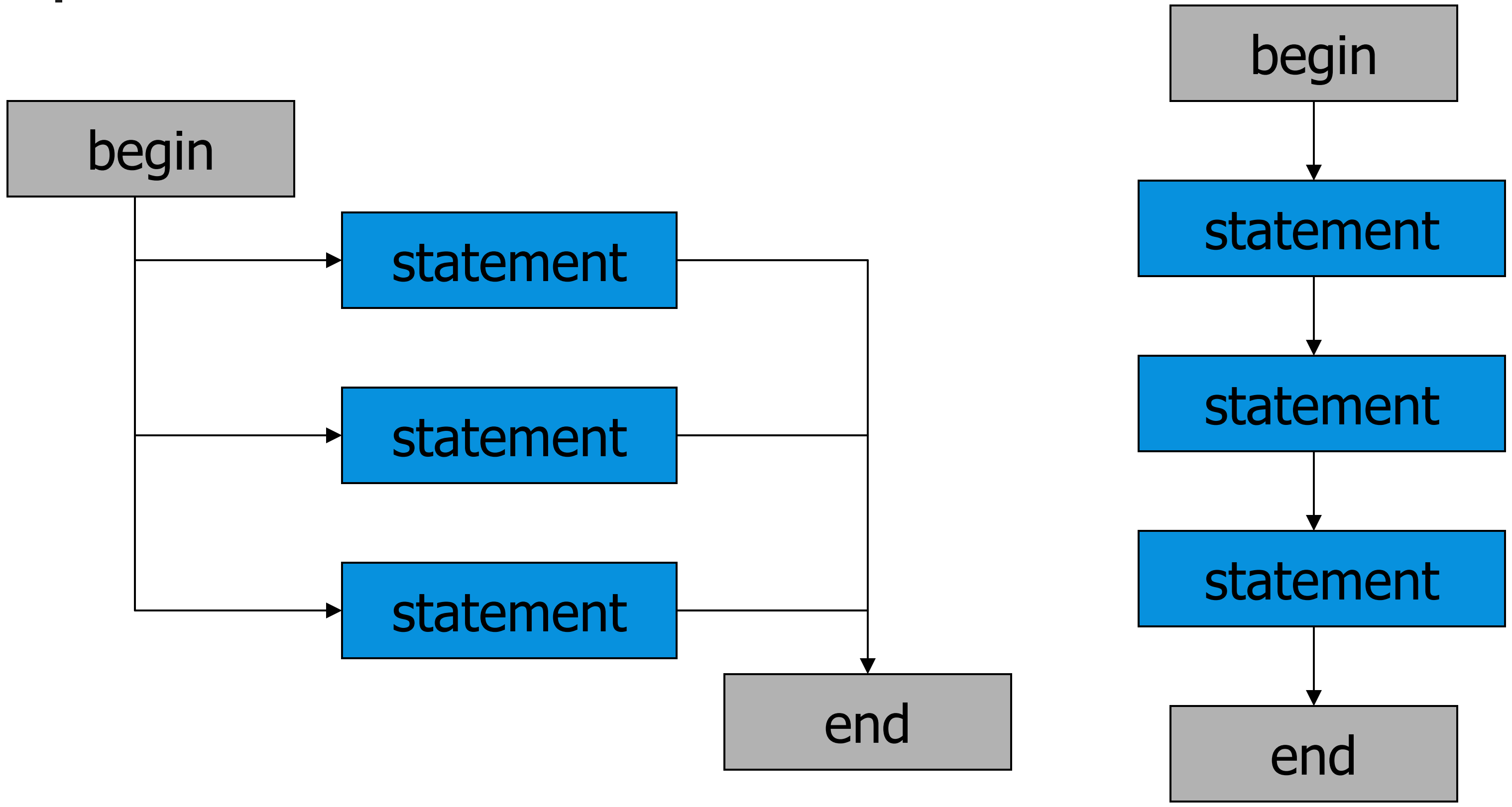
Part V
Behavioral Modeling in VHDL

California State University Northridge

Behavioral Modeling in VHDL

- VHDL behavior
  - Sequential Statements
  - Concurrent Statements

VHDL Behavior



VHDL Behavior (cont'd)

- In this section, some of the most commonly used *concurrent* and *sequential* statements will be introduced.
- VHDL provides ***concurrent statements*** for parallel operations or abstract models for a circuit in a behavioral manner. These statements can be executed by a simulator at the same simulation time.
- The ***process statement*** is the primary concurrent statement in VHDL.
- Within the process, ***sequential statements*** define the step-by-step behavior of the process.

VHDL Behavior (cont'd)

- The process contains sequential statements that describe the behavior of an architecture.
- Sequential statements, define algorithms for the execution within a process or
 a subprogram.
- Familiar Notations of
  - sequential flow
  - Control
  - Conditionals
  - Iterations
- Executed in order in which they appear in the process (like in a programming language.)

Sequential Statements

- Sequential Statements
  - Variable Assignment Statement
  - **PROCESS** Statement (*Concurrent and Sequential*)
  - **IF** Statement
  - **CASE** Statement
  - **LOOP** Statement
  - **NEXT** Statement
  - **EXIT** Statement
  - **WAIT** Statement
  - *Subprograms*
    - **Function***s*
    - **Procedure***s*
  - **ASSERT** Statement

Concurrent Statements

- Concurrent Statements
  - Concurrent Signal Assignments
  - Conditional Signal Assignment
  - Selected Signal Assignment
  - Concurrent Procedure Call
  - **BLOCK** Statements
    - Guarded Blocks

Sequential Statements

Variables

Variable assignment statement

replaces the current value of a variable with a new value which is Specified by an expression.

```
target_variable := expression;
```

Target_variable is a variable previously declared.
Expression is a statement using variables, signals, and literals.

IMPORTANT :

The named variable and the result of the expression must be same type
Statement executes in zero simulation time
Assignments can be made to a scalar or to an array
Variables declared within a process cannot pass values outside of a process.

Variables

Example

```
character := ‘A’  -- Character Assignment

x := 1.0          -- Real value Assignment
```

```
X := 1;           -- Integer Value Assignment
Y := 4;           -- Integer Value Assignment

Z := X + Y;       -- Variable Assignment specified
                  -- by an expression
```

Note: Recall that signal assignments use the operator “<=“

Processes

- Basic unit of Behavioral Description, found in architectures and occasionally entities. Entities can only contain passive process (processes that do not make assignment to signals)
- Process statements as a whole are concurrent statements. Concurrent signal assignment statements are an allowed shorthand method of writing processes.
- Processes have a declaration and statement section. The statement section contains only sequential statements. Sequential signal assignments have special considerations.

Processes (cont'd)

- Variables, constants, data types, and subprograms may be declared in processes, but not signals.
- Anything declared in the entity (e.g.. generics and ports) and the architectures are visible to processes.
- Processes must contain either a wait statement or sensitivity list (or would never quit running)
- During initialization the simulator executes each process up to its first statement
- All processes begin execution during simulator initialization. Variables are initialized once and thereafter retain their relative values.

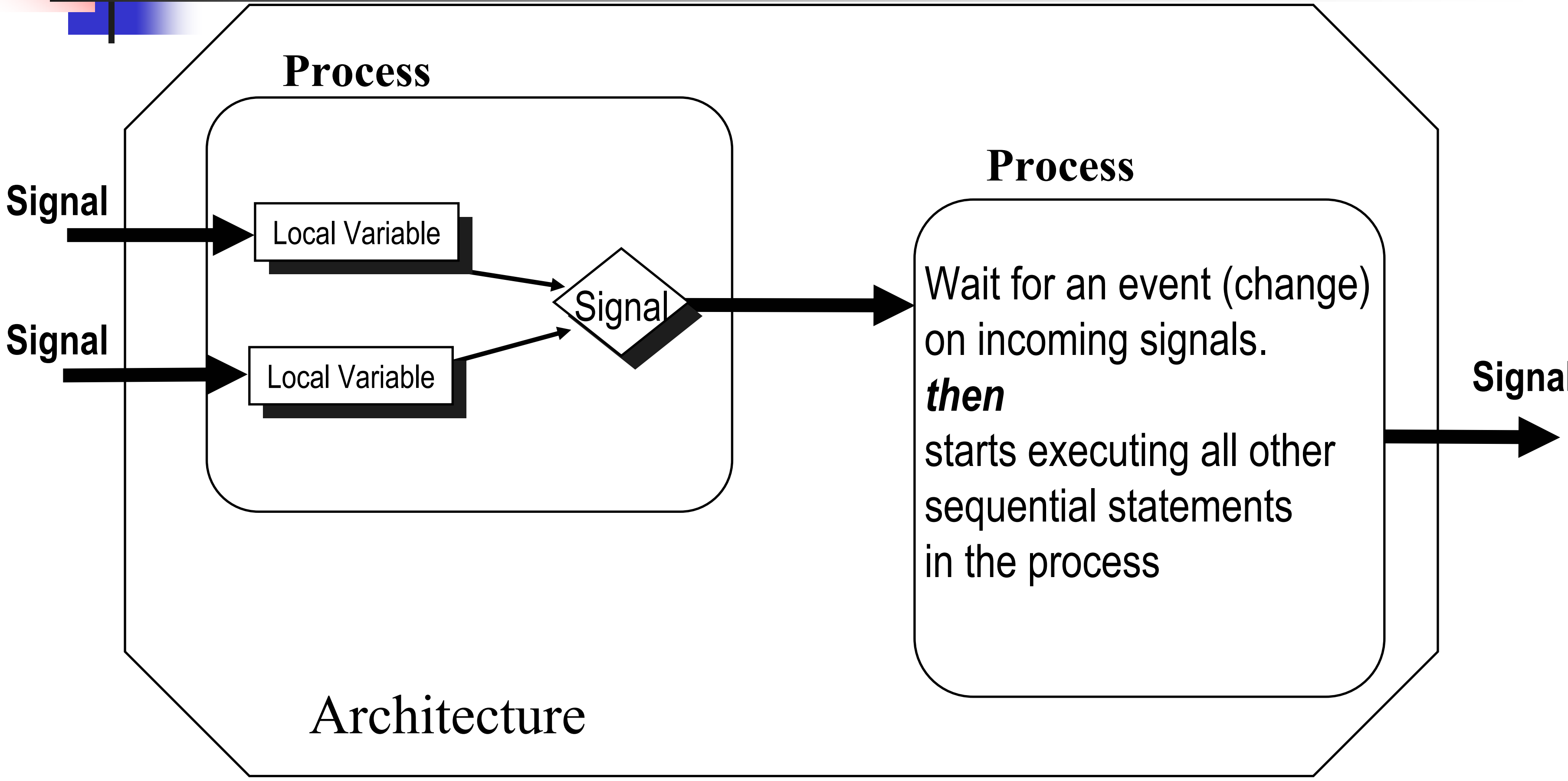
Parallel Process Execution

```
X: process begin
        if true then
          wait...
        case expression
        ........
        end process;
```

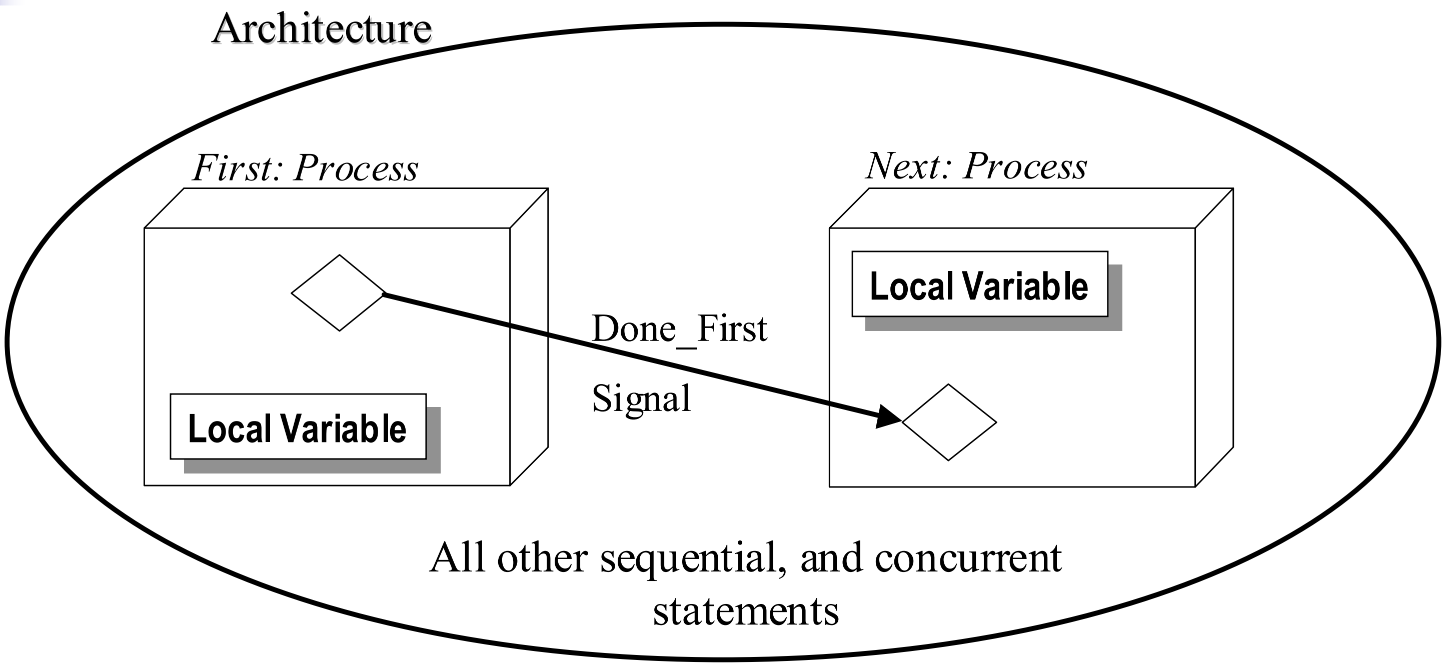
```
Y:process begin
   ...  --  Sequential Statements
      wait ...
   ....   --More statements
            end process;
```

- VHDL permits *concurrent process* operation to be simulated.
- Each process starts at **begin**, executes statements in sequence until a **wait** statement suspends the process. This allows other processes to run.
- When process hits **end process**; it continues from **beginning**.
- The process label (e.g: A:) helps during simulation and debugging

Process Model


Process
Signal
Signal
Local Variable
Local Variable
Signal
Process
Wait for an event (change) on incoming signals.
then
starts executing all other sequential statements in the process
Signal
Architecture

Process Model (cont'd)



1. ***Signals must be used to communicate between processes.***
2. ***Variables are local to the process in which they are declared.***
3. ***Signals can be used to synchronize processes.***

Communication Between Processes via Signals

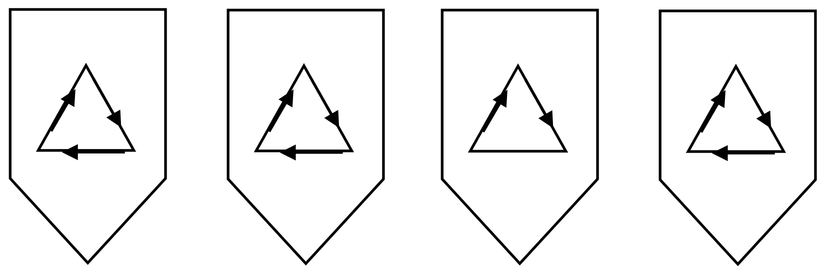
```
architecture EXAMPLE of COMMUNICATE is
        signal DONE_FIRST : Bit := '0';
begin
        FIRST: process
        begin
                ......          -- do first steps......
                DONE_FIRST <= '1';
                wait for 10 ns;
        end process;
        NEXT: process
        begin
                wait until DONE_FIRST = '1';
                .......         -- do the other steps
        end process;
end EXAMPLE;
```

Declaration of the signal

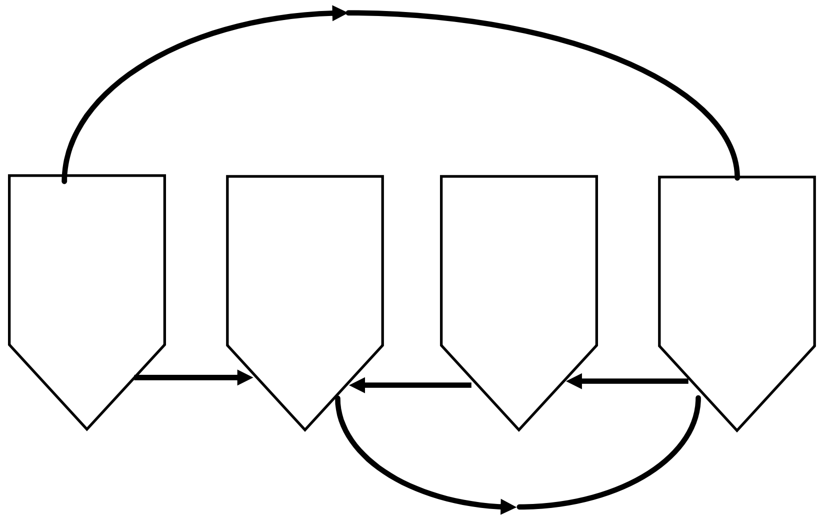
Assignment

Other process waits until '1' is assigned to the signal

Signals Communicate in Between the Processes

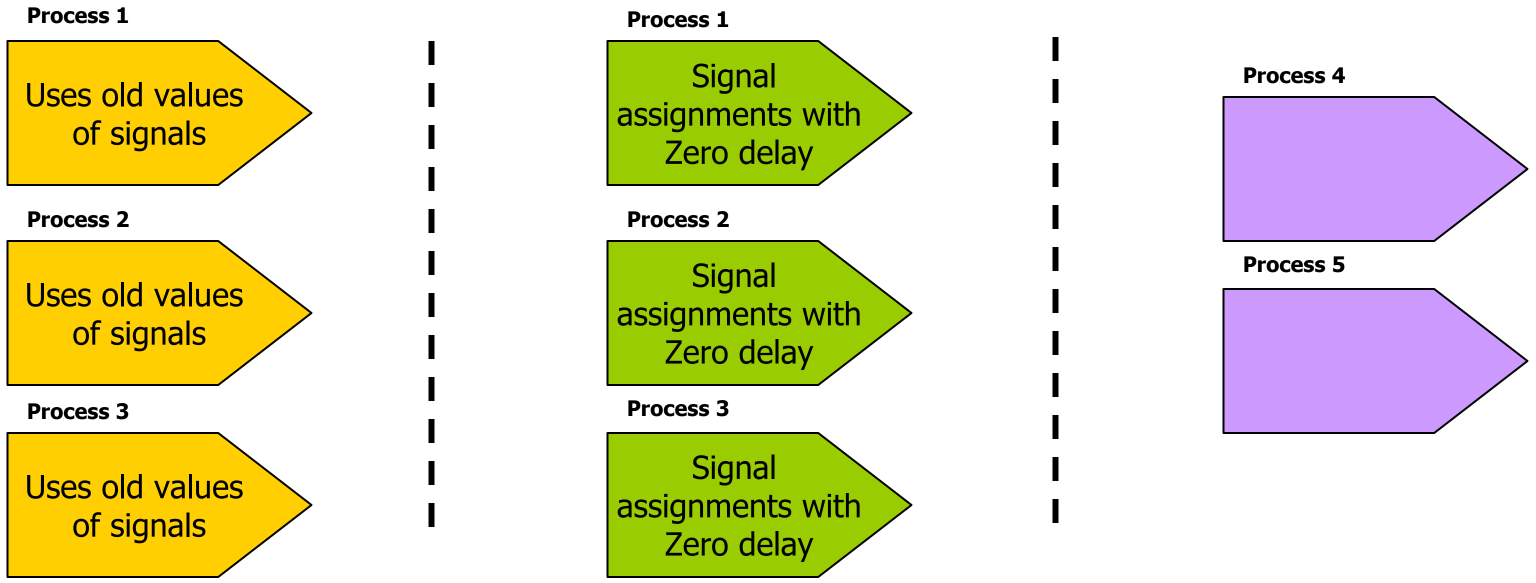


Concurrent Statements are evaluated first until to a “wait”



When all the processes are waiting, all the signals propagate.

Signals Assigned After Processes Run


Process 1
Uses old values of signals
Process 2
Uses old values of signals
Process 3
Uses old values of signals
Process 1
Signal assignments with Zero delay
Process 2
Signal assignments with Zero delay
Process 3
Signal assignments with Zero delay
Process 4
Process 5

All processes run until a "wait" executed

All processes are "waiting

Some other processes woken up

California State University Northridge

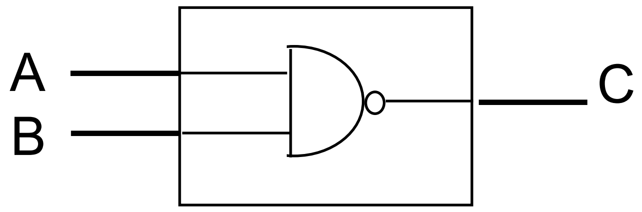
Process

Example

```
entity NAND2 is
Port (A, B: in Bit;
        C:     out Bit)
end NAND2;
```

Process will suspend here →

```
architecture BEHAVE of NAND2 is
                    PDQ : process
                       variable TEMP : Bit;
begin

                    wait on A, B;
                              TEMP := A nand B;
                    if (TEMP = '1') then
                    .......
                              sequential statements
                    ........
                    end process;
                    end BEHAVE;
```



California State University Northridge

Process with Signals

- Contains only sequential statements
- Can access signals defined in architecture and entity
- Must contain either an explicit sensitivity list, or a wait statement(s)
- Can have variable assignment and signal assignment statements.

Sensitivity List

- **PROCESS** (sig1, sig2,...sigN)
- Events (changes) on any signal in sensitivity list will cause process execution to resume, after *BEGIN* statement
- Process execution continues until *END* process statement is reached
- For synthesis, all signals which are read, typically must be included in the sensitivity list.

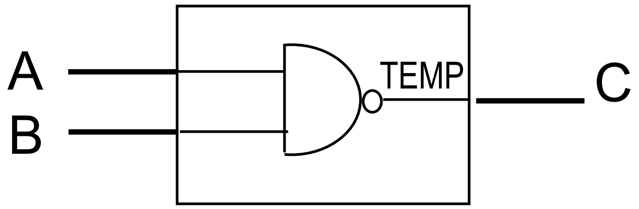
Sensitivity List

```
process (a,b)                          -- the process is only sensitive to a, b any
                                       -- changes on c does not initiates execution of
                                       -- process
begin
          y <= a AND b AND c;
end process;
```

Process with Sensitivity List

Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity NAND2 is
        port(      A,B : in          std_logic;
                     C: out          std_logic);
end NAND2;
architecture NAND2 of NAND2 is
begin
        process (a,b)                           -- signals a, b are the sensitivity list
            variable TEMP : std_logic;
        begin
                    TEMP := not (A and B);
                    if (TEMP = '1') then c <= TEMP after 6 ns; -- sequential signal assignment
                                elsif (TEMP = '0') then C <= TEMP after 5 ns;
                                else C <= TEMP after 6 ns;
                    end if;
        end process;
end NAND2;
```



California State University Northridge

IF statement

- **IF** statements
  - Represent hardware decoders in both abstract and detailed hardware models
  - Select for execution one or more of the enclosed sequential statements (if more than one sequential statement, use '**;**' at the end of each statement).
  - Can be nested.

```
if CONDITION1 then Sequential_Statement(s);
            {elsif CONDITION2 then Sequential_Statement(s);}
            {elsif CONDITION3 then Sequential_Statement(s);}
            {elsif CONDITION4 then Sequential_Statement(s);}
            {......
            }
                        [else Sequential_Statement(s);]
end if;
```

IF statement (cont'd)

```
if CONDITION then
              .....           --  Sequential_Statement(s)
end if;
```

```
if CONDITION then
              .....           -- Sequential_Statement(s)
              else
              .....           -- Sequential_Statement(s)
end if;
```

```
if CONDITION1 then
              .....           --  Sequential_Statements
              elsif CONDITION2 then
                            .....           --  Sequential_Statement(s)
              elsif CONDITION3 then
                            .....           --  Sequential_Statement(s)
              else
                            .....           --  Sequential_Statement(s)
end if;
```

IF statement
Example

```
library IEEE;
use IEEE.Std_uLogic_1164.all;

entity IF_STATEMENT is
port (          U, W, X, Y: in std_ulogic_vector (1 downto 0);
                Z              : out std_ulogic_vector (1 downto 0));
end IF_STATEMENT;
architecture IF_ARC of IF_STATEMENT is
begin
                process (U, W, X, Y)
                begin
                                if ( Y = "00") then
                                                Z <= U;
                                elsif (Y = "01") then
                                                Z <= W;
                                else
                                                Z <= X;
                                end if;
                end process;
end IF_ARC;
```

IF statement

Example (cont'd)

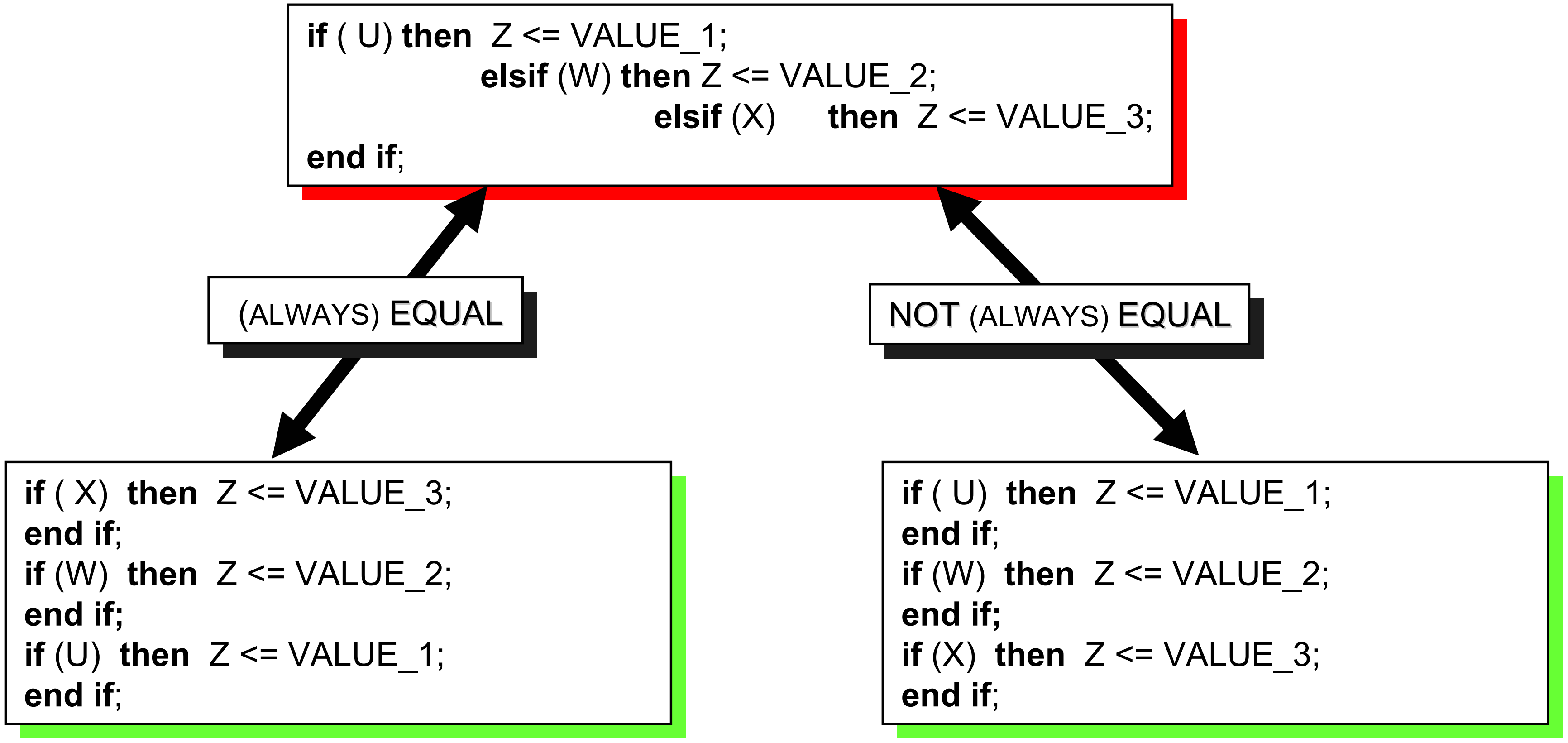
```
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity COUNT16 Is
   port (CLK,RST,LOAD: in std_ulogic;
        DATA: in unsigned (3 downto 0);
        COUNT: out unsigned (3 downto 0)
   );
end COUNT16;
```

```
architecture COUNT16_A of COUNT16 Is
begin
   process(RST,CLK)
      variable Q: unsigned (3 downto 0);
   begin
      if Rst = '1' then
         Q := "0000";
      elsif rising_edge(CLK) then
         if LOAD = '1' then
            Q := DATA;
         elsif Q = 15 then
            Q := "0000";
         else
            Q := Q + "0001";
         end if;
      end if;
      COUNT <= Q;
   end process;
end COUNT16_A;
```

Note:
Rising Edge is recognized by simulation tool only, not by synthesis tool.

IF statement

Example (cont'd)



```
if ( U) then  Z <= VALUE_1;
            elsif (W) then Z <= VALUE_2;
                        elsif (X)     then  Z <= VALUE_3;
end if;
```

```
if ( X)  then  Z <= VALUE_3;
end if;
if (W)  then  Z <= VALUE_2;
end if;
if (U)  then  Z <= VALUE_1;
end if;
```

```
if ( U)  then  Z <= VALUE_1;
end if;
if (W)  then  Z <= VALUE_2;
end if;
if (X)  then  Z <= VALUE_3;
end if;
```

CASE statement

- **CASE** statements
  - useful to describe decoding of busses and other codes
  - Select for execution one of a number of alternative sequential statements (if more than one sequential statement, use '**;**' at the end of each statement).

```
case (EXPRESSION) is
        when CHOISE1 => Sequential_Statement(s);
        when CHOISE2 | CHOISE3 | CHOISE4 => Sequential_Statement(s);
        when value1 to value2 => Sequential_Statement(s);
        when others => Sequential_Statement(s);
end case;
```

CASE statement

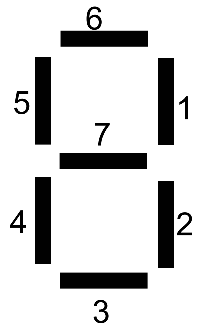
Example

```
library IEEE;
use IEEE.Std_Logic_1164.all;

entity CASE_STATEMENT is
port (          U, W, X, Y: in integer range 0 to 15;
                Z               : out integer range 0 to 15);
end CASE_STATEMENT;
architecture CASE_ARC of CASE_STATEMENT is
begin
                process (U, W, X, Y)
                begin
                        case Y is
                                when 0 to 9  => Z <= X;
                                when 13      => Z <= W;
                                when 11 | 15=> Z <= U;
                                when others  => Z <= 0;
                        end case;
                end process;
end IF_ARC;
```

CASE statement (cont'd)

- Rules
  - Use the CASE statement, when you have a complex decoding situation (It is more readable than nested if statements.)
  - CASE statement must enumerate all possible values of the expression or to have an OTHERS clause as a last choice of all of the choices.
  - None of the choices may overlap.

```
case BCD_DECODER is
            when "0000"   => SEGMENTS := "1111110";  -- zero
            when "0001"   => SEGMENTS := "1100000";  -- one
            when "0010"   => SEGMENTS := "1011011";  -- two
            when "0011"   => SEGMENTS := "1110011";  -- three
            when "0100"   => SEGMENTS := "1100101";  -- four
            when "0101"   => SEGMENTS := "0110111";  -- five
            when "0110"   => SEGMENTS := "0111111";  -- six
            when "0111"   => SEGMENTS := "1100010";  -- seven
            when "1000"   => SEGMENTS := "1111111";  -- eight
            when "1001"   => SEGMENTS := "1110111";  -- nine
            when OTHERS             => SEGMENTS := "- - - - - - -";  -- don't care
end case;
```



California State University Northridge

LOOP statement

- **LOOP** Statements
  - Provide a convenient way to describe bit-sliced logic or iterative circuit behavior.
  - Include sequential statements to execute repeatedly, zero or more times.
  - Can be nested.

```
                [LOOP_LABEL]: while CONDITION1 loop
                          .....          -- Sequential_Statement(s)
                end loop[LOOP_LABEL];
OR
                [LOOP_LABEL]: for IDENTIFIER in DISCRETE_RANGE loop
                          .....          -- Sequential_Statement(s)
                end loop[LOOP_LABEL];
```

LOOP statements

Example

FOR LOOP

```
LABEL_ONE:  for I in 1 to 20 loop
                Sequential_Statement(s);
end loop LABEL_ONE
```

WHILE LOOP

```
INITIAL := 0;
LABEL_TWO:  while (INITIAL < 10) loop
                Sequential_Statement(s);
                INITIAL := INITIAL + 1;
end loop LABEL_TWO
```

LOOP statements
Example (cont'd)

```
LABEL_THREE: for I in 1 to 10 loop
          VALUE := I * 2;
          SQUARE_I(I) := I * I;
end loop LABEL_THREE;
```

Loop index "I"
- *is not a variable*
- *Hides any other "I" not in the loop*
- *cannot be seen outside the loop*
- *must be a discrete type (not REAL)*
- *can be used as array reference*

```
LABEL_FOUR: while (TODAY <= 5) loop
          -- weekday Sequential_Statements
          TODAY := TODAY + 1;
end loop LABEL_FOUR;
```

NEXT statement

- The **NEXT** statement
  - skip the execution to the next iteration of an enclosing LOOP statement.
  - is convenient to use when you want to skip an iteration of a LOOP.

```
next [LABEL] [when CONDITION];
```

NEXT statement

Example

```
for I in 1 to 30 loop
            if (VALUE(I) = 0) then next;
            end if;
OUTPUT_VALUE(I) := VALUE(I);
end loop;
```

The **loop** statement has a range bounded by the **end loop**. Execution of the **next** statement causes iteration to skip to the next **loop** index value (e.g. **for** I.)

```
LABEL1: while I < 20 loop
LABEL2: while J < 20 loop
.....
.....            -- Sequential_Statement(s);
.....
                 next LABEL2 when I = J;
.....
.....            -- Sequential_Statement(s);
.....
end loop LABEL2;
end loop LABEL1;
```

Nested loops should be given unique labels.

EXIT statement

- The **EXIT** statement completes the execution of an enclosing LOOP statement.
- The LOOP label in the EXIT statement identifies the particular loop to be exited.

```
exit [LABEL] [when CONDITION];
```

EXIT statement

Example

```
for I in 1 to 30 loop
            if (VALUE(I) = 0) then exit;
            end if;
OUTPUT_VALUE(I) := VALUE(I);
end loop;
```

The **for** statement has a range for **loop**. When the I indexed variable has a value of zero, **exit** causes execution to exit the loop entirely.

WAIT statement

- The **WAIT** statement
  - provides for modeling signal-dependent activation
  - can be used to model a logic block that is activated by one or more signals.
  - causes a simulator to suspend execution of a process statement or a procedure, until some conditions are met, then execution resumes.
  - forces simulator signal propagation.

```
wait
wait on SIGNAL_NAME;
wait until TRUE_CONDITION;
wait for TIME_EXPRESSION;
```

Processes with no sensitivity list, no wait statement will loop forever at the initialization

WAIT statement

Example

```
wait on A,B;
```

suspends execution until a change occur on either signal A or B. Same as **Process (A,B)**

```
wait until T > 10;
```

suspends execution until condition T > 10 becomes satisfied. It requires an event on T to evaluate the expression.

```
wait for 20 ns;
```

suspends execution of process for 20 ns.

ASSERT statement

- **Assert** statement allows for testing a *condition* or *error* message. It checks to determine if a specified condition is true, and displays a message if a condition is false.

```
assert CONDITION [report MESSAGE_STRING] [severity MESSAGE_STRING];
```

Severity Levels are
Note, Warning, Error, Failure

ASSERT statement (cont'd)

- **Assert** statement
  - Cannot be formatted
  - Can be used as either concurrent or sequential statements
  - Prints out
    - simulation time
    - user written string
    - instance name
    - severity level (Defaults to ERROR)
  - Useful for timing checks, range checks, debugging.

ASSERT statement

Example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity SETUP_CHECK is
        port (      D, CLK: in std_logic;
                    Q: out std_logic);
begin
        SETUP_CHECK_PROCESS: process (CLK)
        begin
                    if (CLK='1') and (CLK'LAST_VALUE ='0') then
                              assert (D'LAST_EVENT >= 3 ns)
                                  report "Setup Error"
                                       severity WARNING;
        end if;
        end process;
end SETUP_CHECK;
```

Passive Process

Aliases

- Alternate designation for an object (signal, variable, constant)
- Assignments to the alias are assignments to the object. Assignments to the object are assignments to the alias.
- Example:

```
signal instruction : std_logic_vector (15 DOWNTO 0);
alias inst : std_logic_vector (15 DOWNTO 0) is instruction;
alias reverse_inst : std_logic_vector (0 TO 15) is instruction;
alias opcode : std_logic_vector (3 DOWNTO 0) is instruction
                              (15 DOWNTO 12);   --slice
alias opbit : std_logic is opcode(0);           --alias of an alias
```

Aliases (cont'd)

```
signal INST_FIELD : Bit_vector(31 downto 0);
alias OP_FIELD:Bit_vector(3 downto 0) is
            INST_FIELD(31 downto 28);
```

```
INST_FIELD<=01000101011001111000100110101011 0
            | | ||
OP_FIELD   <= 0100
```

Subprograms

- A subprogram has sequential statements contained inside, and is called from a process. Subprograms provide a convenient way of documenting frequently used operations.
- There are two different types of subprograms
  - A procedure (Return Multiple Values)
  - A function (Return a Single Value)
- Typical usage areas
  - Conversion Functions
  - Resolution Functions
- Subprograms are either built-in or user-designed.

Subprograms (cont'd)

- Contain sequential statements similar to processes
- May declare local variables, constants
- Executed when called from a sequential statement.
- Local Variables are re-initialized every time a subprogram is called.
- Parameters of calling routine are known as **actuals**, while the parameters of the declared subprogram are known as **formals**.
- Up level referencing to higher level variables and signals is allowed.
- Recursive calls by functions and procedures are allowed
- Attributes of signals cannot be accessed within subprograms

Functions

A user defined function needs to be declared if it is called. When a function is called, values are passed in through parameters just prior to its execution.

```
process
function C_TO_F (C: real) return real is
variable F: real;
begin
            F := C * 9.0 / 5.0 + 32.0;
            return (F);
end C_TO_F;
```

Function Declaration

```
variable NEW_TEMP: real;
begin
            NEW_TEMP := C_TO_F (5.0) + 20.0;
            .....           -- Sequantial_Statement(s);
end process;
```

Function Declaration

Functions (cont'd)

```
function NAME (PARAMETER) return TYPE is
.....  -- variable declarations
begin
.....
.....   --Sequential_ Statements
.....
return (  );
end NAME;
```

- Function contains sequential statements
- Converts one type to another
- Some built-in functions
- Some user supplies and reuse.
- No side effects

Functions (cont'd)

```
type FOURVAL is ('X', 'L', 'H', 'Z');                    -- Incoming values
type VALUE4 is ('X', '0', '1', 'Z');                     -- Outgoing values
function CONVER4VAL (S: FOURVAL) return VALUE4 is
begin
            case S is
                        when 'X' => return 'X';
                        when 'L' => return '0';
                        when 'H' => return '1';
                        when 'Z' => return 'Z';
            end case;
end CONVER4VAL;
```

TYPE CONVERSION

```
process
            variable PDQ: FOURVAL;
            variable XYZ: VALUE4;
begin
            XYZ := CONVER4VAL (PDQ)                          -- function call
end process;
```

Functions (cont'd)

- Called by expressions
- Produce a single return value
- Can not modify the parameters passed to it
- Requires a RETURN statement

```
function ADD_BITS (A, B : in bit) return Bit is
begin  -- functions can NOT return multiple values
                return (A XOR B);
end ADD_BITS;
```

```
function ADD_BITS2 (A, B : in Bit) return Bit is
 variable RESULT : bit;  -- variable is local to function
        begin
            RESULT := (A XOR B);
            return RESULT;  -- the two functions are equivalent
end ADD_BIT2;
```

Functions (cont'd)

```
architecture BEHAVIOR of ADDER is
begin
            process (ENABLE, X, Y)
            begin
            if (ENABLE = '1') then
               RESULT <= ADD_BITS(X, Y);
               CARRY  <= X and Y;
            else
               CARRY, RESULT <= '0';
            end process;
end BEHAVIOR;
```

```
function ADD_BITS

       (A, B : in Bit)
```

- Functions must be called by other statements
- Parameters use positional association

Functions

Example

```
-- Convert a std_ulogic_vector to an unsigned integer
--
function TO_UNSIGNED (A: std_ulogic_vector) return integer is
    alias AV: std_ ulogic_vector (1 to A'LENGTH) is A;
    variable RET,D: integer;
begin
    D := 1;
    RET := 0;

    for I in A'LENGTH downto 1 loop
        if (AV(I) = '1') then
            RET := RET + D;
        end if;
        D := D * 2;
    end loop;
    return RET;
END TO_UNSIGNED;
```

Functions

Example (cont'd)

```
-- Convert an integer to a std_ulogic_vector
--
function TO_VECTOR (SIZE: integer; NUM: integer) return std_ulogic_vector is
   variable RET: std_ulogic_vector (1 to SIZE);
   variable A: integer;
begin
   A := NUM;
   for I in SIZE downto 1 loop
      if ((A mod 2) = 1) then
         RET(I) := '1';
      else
         RET(I) := '0';
      end if;
      A := A / 2;
   end loop;
   return RET;
end TO_VECTOR;
```

Procedures

- Assignments can be made to as many as desired - assignments are made to parameters of modes OUT or INOUT
- Side effects - assignments can be made to signals which are not in the parameter, but only if the procedure is declared inside of the process that calls it.
- Static Up Level referencing - reference variable(s) from the process or other procedures that called them. If the variable(s) are not include in parameter list they may not be assigned to.
- Concurrent procedures calls are execute once at initialization and then execute only when the parameters of mode IN or INOUT have events on them.
- Invocation is a statement.

Procedures (cont'd)

```
function NAME (PARAMETERS) is
.....  -- [variable]
.....  -- [constants]
.....  -- [types]
.....  -- [declarations]
begin
.....
.....   --Sequential_Statements
.....
return (  );
end NAME;
```

PARAMETERS

variable NAMES [**in** | **out** | **inout**] **type** [:= EXPRESSION];

or

signal NAMES [**in** | **out** | **inout**] TYPE;

Procedures (cont'd)

- Procedure Declaration

```
procedure P;
procedure XOR2 (  signal IN1, IN2 : in Bit;
                  signal OUT1 : out Bit);
-- use word 'signal' is optional for parameters of mode in, but
-- required if out parameter is a signal
```

- Procedure Body

```
procedure DIV_16 (A : inout integer) is
begin
        A := A/16;
end DIV_16;
```

Procedures

Example

Actuals in Call

```
architecture BEHAVIOR of ADDER is
begin
            process (ENABLE, X, Y)
            begin
            ADD_BITS3(X, Y, ENABLE,
                        RESULT, CARRY);
            end process;
end BEHAVIOR;
```

- With parameter passing, it is possible to further simplify the architecture

Formals in Declaration

- The parameters must be compatible in terms of data flow and data type

```
procedure ADD_BITS3

(signal A, B, EN : in Bit;
signal TEMP_RESULT,
       TEMP_CARRY : out Bit);
```

Procedures

Example (cont'd)

Procedure for 8 bit parity generator

```
procedure  PARITY (A : in Bit_vector (0 to 7);
                         RESULT, RESULT_INV: out Bit) is
variable TEMP: Bit;
begin
      TEMP := '0';
      for I in 0 to 7 loop
      TEMP := TEMP xor A(I);
      end loop;
      RESULT := TEMP;
      RESULT_INV := not TEMP;
end PARITY;
```

Summary

- A process defines regions in architectures where sequential statements are executed (components are not permitted.)
- Process statements provide concurrent processing capability using local variables and global signals.
- VHDL Contains sequential statements, including IF THEN ELSE, CASE LOOP, etc.
- WAIT statements dynamically control process suspension/execution. In simulation, all processes are started and executed up to a WAIT.
- A process can call functions (that return a single value) and procedure (that return more than one value.)

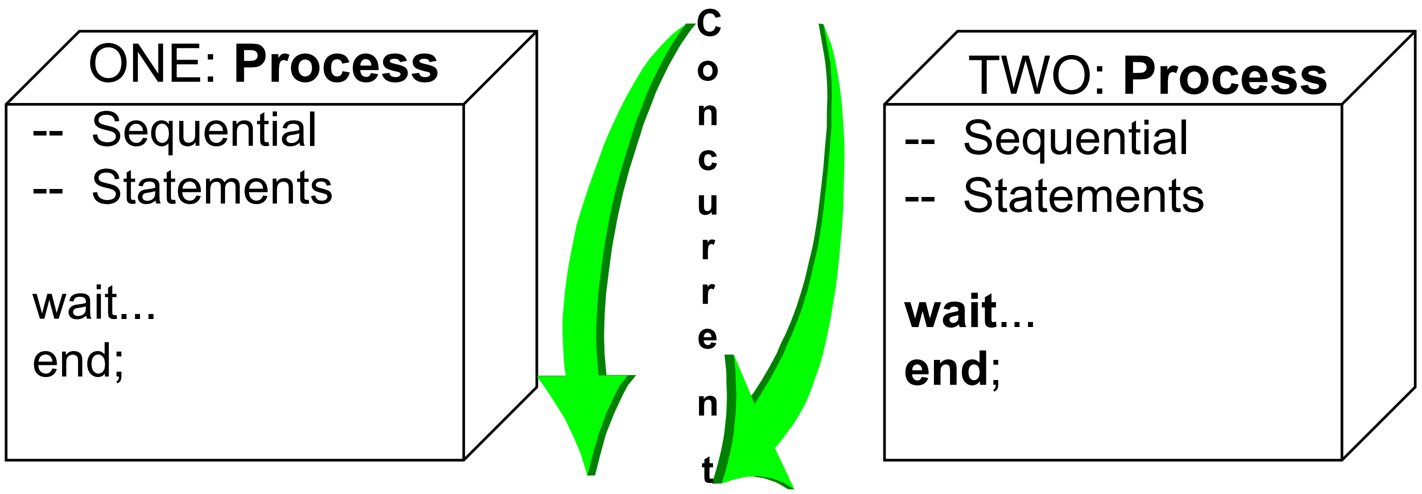
Concurrent Statements

Concurrent Statement

- Exists outside of a process but in an architecture
- The process is itself a concurrent statement all processes scheduled to run concurrently
- *Concurrent signal assignment* is a short hand form for a single statement process -- equivalent to process containing one statement, sensitive to changes on the right hand side.
- Used frequently in DATAFLOW style descriptions.

The Process

- A **Process**
  - Runs concurrently with other processes
  - Contains only sequential statements
  - Defines regions in architectures where statements execute sequentially
  - Must contain either an explicit sensitivity list or a WAIT statement
  - Can access signals defined in architecture and entity.



California State University Northridge

Concurrent Signal Assignment

- Execute asynchronously, with no defined relative order
- Used for dataflow style descriptions
- Syntax:

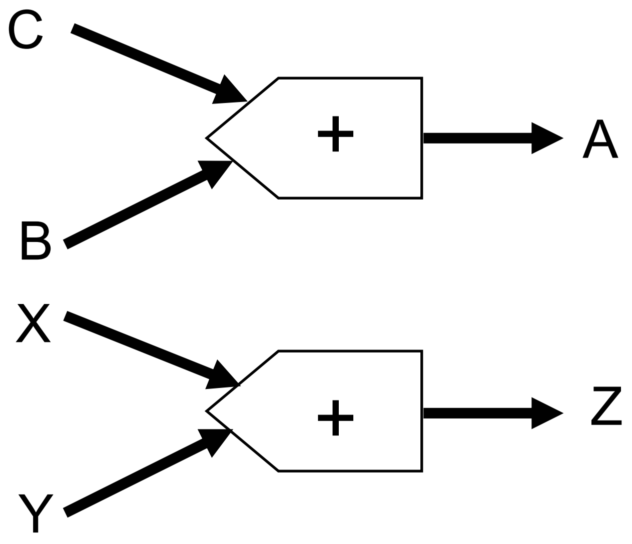
```
TARGET <= EXPRESSION;
```

EXPRESSION is logical, comparative, or arithmetic operation

- VHDL offers other forms of Concurrent Signal Assignment:
- **Conditional** - similar to if statements
- **Selected** - similar to case statement

Concurrent Assignment Statements

```
architecture EXAMPLE
begin
    A <= B + C;  -- Statement_one
    Z <= X + Y;   -- Statement_two
end;
```



- Builds combinational circuitry
- Note that   if B or C change then Statement _one is evaluated
                       if X or Y change then Statement_two is evaluated

California State University Northridge

Concurrent Signal Sensitivity

- Concurrent Statements are sensitive to all signals on the input side
- If a signal appears on both sides, the statement is sensitive to changes in its own output.

  A <= A+B; will be evaluated when B changes. This will change A and the statement will be evaluated again.
- Time will not be able to advance because the statement keeps executing.

Conditional Signal Statement

A conditional signal assignment is an concurrent statement and has one target, but can have more than one expression. Only one of the expressions can be used at a time.

```
TARGET <= {EXPRESSION when CONDITION else} EXPRESSION;
```

Example

```
Z <= A when (X > 3) else
       B when (X < 3) else
       C;
```

Note: *You cannot use conditional signal assignment in a process.*

Conditional Signal Statement

Example

```
entity EXAMPLE is
        port (A,B, C, SEL    : in integer range 0 to 7;
                   Z         : out integer range 0 to 7);
end EXAMPLE;
```

```
architecture IF_TYPE of EXAMPLE is
begin
   process (A, B, C, SEL)
   begin
            if (SEL > 5) then
              Z <= A;
            elsif (SEL < 5) then
              Z <= B;
            else
              Z <= C;
            end if;
   end process;
end IF_TYPE;
```

```
architecture COND_TYPE of EXAMPLE is
begin
   Z < =      A when SEL > 5 else
              B when SEL < 5 else
              C;
end COND_TYPE.
```

Selected Signal Statement

A selected signal assignment statement can have only one target, and can have only one WITH expressions. This value is tested for a match in a manner similar to the CASE statement.

```
with EXPRESSION select
   TARGET <= {EXPRESSION when CHOICES,};
```

Example

```
with SIGNAL select
      Z <=            A when 15,
                      B when 22,
                      C when OTHERS;
```

Note: *You cannot use Selected signal assignment in a process*.

Selected Signal Assignment

Example

```
entity EXAMPLE is
          port (A,B, C, SEL        : in integer range 0 to 7;
                            Z      : out integer range 0 to 7);
end EXAMPLE;
```

```
architecture CASE_TYPE of EXAMPLE is
begin
   process (A, B, C, SEL)
   begin
              case  SEL is
                when 0 to 4 =>
                          Z <= B;
                when 5 =>
                          Z <= C;
                when OTHERS =>
                          Z <= A;
              end case;
end process;
end IF_TYPE;
```

```
architecture SEL_TYPE of EXAMPLE is
begin
          with SEL select
             Z <=      B  when 0 to 4,
                       C  when 5,
                       A  when OTHERS;
end SEL_TYPE.
```

Concurrent Procedure Call

- IN, OUT and INOUT parameter modes
- Allows return of more than 1 value (unlike function call)
- Considered a statement
- Equivalent to a process containing the single procedure call followed by a wait on parameters of mode in or inout.

Concurrent Procedure Call

Example

```
architecture ...
    begin
           VECTOR_TO_INT(BITSTUFF, FLAG, NUMBER);
    end;
```

```
architecture ...
           begin
           process
              begin
                  VECTOR_TO_INT(BITSTUFF, FLAG, NUMBER);
                  wait on BITSTUFF, NUMBER;
              end process;
           end;
```

NUMBER was **inout**, procedure changed NUMBER so procedure run again

Sequential vs. Concurrent Statement in Simulation Cycle

- VHDL is inherently a concurrent language
  - All VHDL processes execute concurrently
  - Concurrent signal assignment statements are actually one-line processes
- VHDL statements execute sequentially *within a process*
- Concurrent processes with sequential execution within a process offers maximum flexibility
  - Supports various levels of abstraction
  - Supports modeling of concurrent and sequential events as observed in real systems

Blocks

- Blocks are concurrent statements and provide a mechanism to partition an architecture description
  - Items declared in declarative region of block are visible only inside the block, e.g. :
    - signals, subprograms
- Blocks may be nested to define a hierarchical partitioning of the architectural description
- Blocks may contain Guards for disabling drives.

Blocks

```
[LABEL:] block [(BOOLEAN EXPRESSION)]
         {DECLARATIONS}
begin
         .....      -- Concurrent Statements
end block;
```

Declarations declare objects local to the block and can be any of following:

- USE clause
- Subprogram declaration and body
- Type, constants, signal declarations
- Component declaration

Blocks

Must have a label

May have a label

```
B2: block
  begin
    C1 : E <= C + D;
    C2 : C <=  A + B;
  end  block;
```

- Blocks can be nested
- Objects declared in a BLOCK are visible to that block and all blocks nested within.
- When a *child* block inside a *parent* block declares an object with the same name as one in the parent block, the *child's* declaration overrides that of the *parent*.

Nested Blocks

```
B1: block
      signal S: Bit;                   -- -- Declaring first S in block B1
begin
      S <= A and B;                    -- -- using S in B1
      B2: block
            signal S: Bit;             -- -- Declaring second S in block B2
      begin
            S <= C and D;              -- -- using S in B2
            B3: block
            begin
                        Z <= S;        -- -- still using S in B2
            end block;                 -- -- B3   -- --(Z <= C and D)
      end block;                       -- -- B2
      Y <= S;                          -- -- using S from B1
end block;                             -- -- B1       -- -- (Y <= A and B)
```

Guarded Blocks

- Can be for modeling latches, clocked logic, bus controllers
- The Guard is a boolean expression
- Signal named “GUARD” is implicitly created in block and may be referenced
- Signal assignment statements with signal kind of BUS or REGISTER can use Guard as enable.
- When Guard is true, normal signal assignment activity may occur
- When Guard is false, drivers may be disconnected

Guard Expression

- A block can have a Boolean "*Guard expression*"

```
B2: block (SELECT = '1')
begin
        Z <= IN_1 when GUARD else IN_0;
end block;
```

- Equivalent to GUARD <= (SELECT = '1'); (sensitive to SELECT)
- The Boolean signal GUARD is created automatically when block has a "guard expression"
- As shown in the above example, you can test the signal GUARD
- Guarded Signal assignment are built-in to the language/simulator

Guarded Signal Assignments

- execute when block guard expression is true

```
architecture REG of LATCH is
signal Q: Std_Logic register := ‘U’;
begin
B2: block (ENABLE = ‘1’)
        begin
                Q <= guarded D;
        end block;
```

- The signal assignment to Q only occurs when GUARD is true

Guarded Signal Kinds

- guarded signals are of a signal kind; **bus** l **register**.

```
signal Q: Std_Logic register := 'U';
begin
B2: block (ENABLE = '1')
            begin
                        Q <= guarded D;
            end block;
```

- Q may be declared as kind register or bus
- Q must be of a resolved type (none of the types in Std.standard are resolved.)
- Signal kind register or bus can be *disconnect*ed.

Disconnects

- When the Guard expression is false, signals assigned with guarded assignments are said to be ***disconnected.***
- Signals of kind Register retain their value when disconnected.
- Signals of kind Bus change to their default value when disconnected. (the default value of std_logic is ‘U’)
- Signals of kind bus and register can only be assigned with a guarded signal assignment.

Drivers

- Drivers are created by signal assignment statements.
- Drivers contain present and future values.
- A driver is a contributor to a signal value.
- Value of a signal is the resolution of all of the driver values
- Multiple assignments in parallel may be error prone.

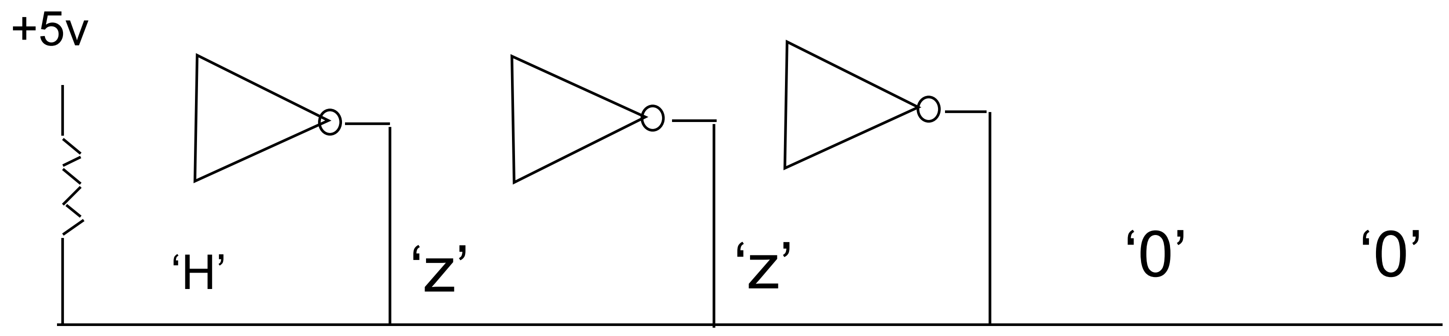
Resolution Function

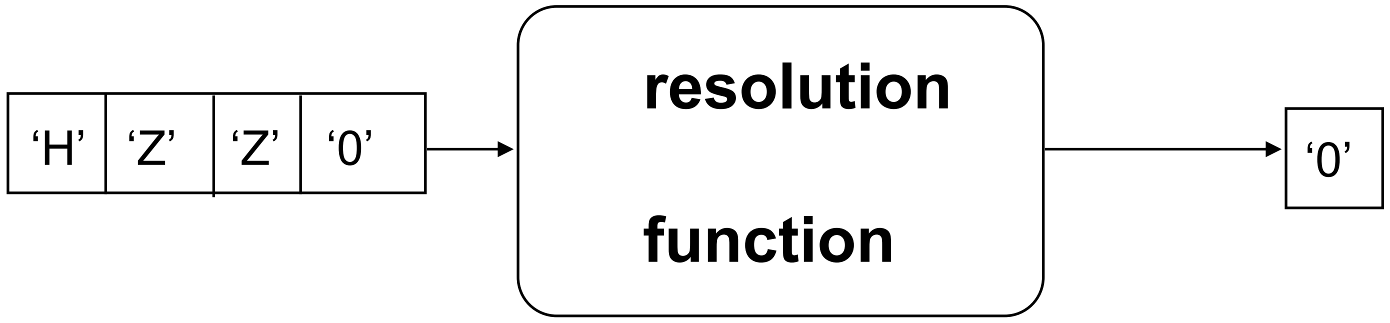
- No Resolution Functions are built into VHDL
- Vendor or User provided Functions
- Resolution Functions are called automatically
- Multiple Driver-values are passed in through parameters.

Resolution Function

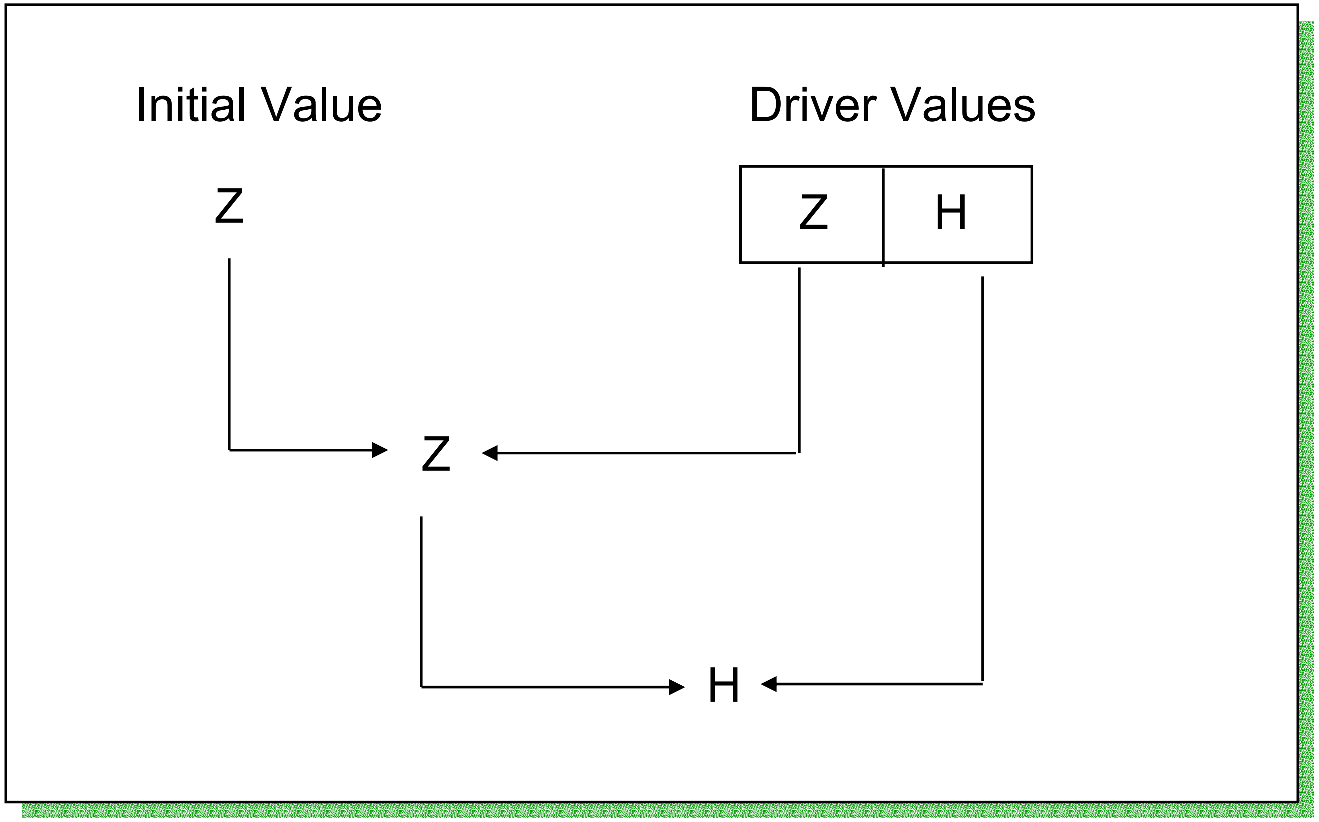
- Resolves value of signal with multiple drivers
- Required when signal has multiple drivers
- Function called after any assignment to signal
- Passed a variable length array of signal values to be resolved
- User definable: typically wired and, or 3-state
- Associates the resolution function with subtype
- (which becomes a resolved type)

Three State Bus

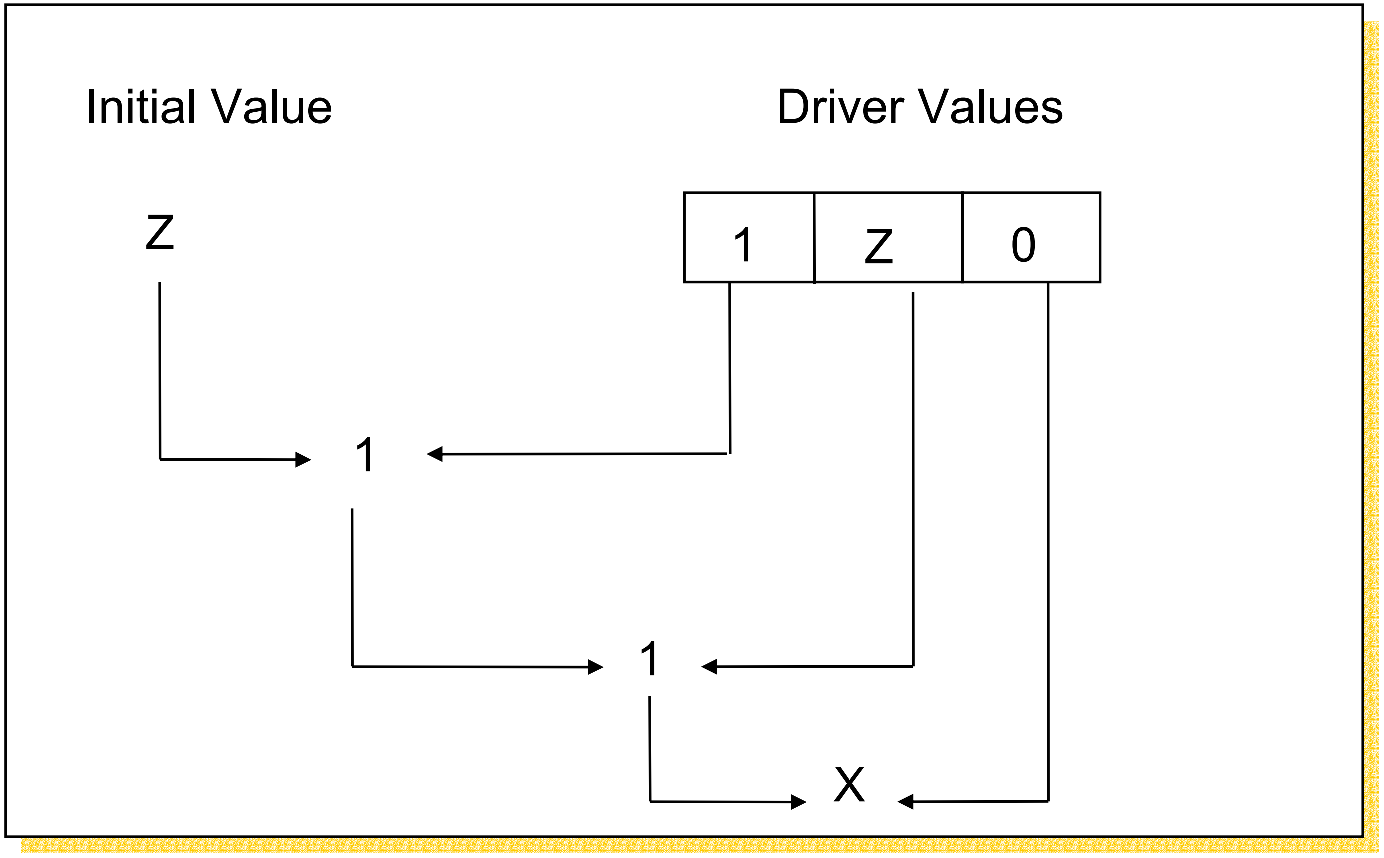




Driver Resolution



Driver Resolution Conflict



Bus Resolution Function

Example

```
package FOURPACK is
      type FOURVAL is (X, L, H, Z);
      type FOURVAL_VECTOR is array(natural range <>) of FOURVAL;
      function RESOLVE (S: FOURVAL_VECTOR) return FOURVAL;
end FOURPACK;

package body FOURPACK is
      function RESOLVE (S: FOURVAL_VECTOR) return is
               variable RESOLVE (S: FOURVAL_VECTOR) return FOURVAL is
                begin
                     for I in S'RANGE loop
                                case RESULT is
```

Bus Resolution Function

Example (cont'd)

```
    when Z =>
              case S(i) is
                when H =>
                   RESULT := H;
               when L =>
                   RESULT := L;
    when X =>
                   RESULT := X;
     when OTHERS =>
               null;
               end case;
              when L =>                                             ;
              case S(i) is
              when H =>
                                 RESULT := X;
           when X =>                                                        :
                                RESULT := X;
              when OTHERS =>
                         NULL;
```

California State University Northridge

Bus Resolution Function

Example (cont'd)

```
 end case;

       when H =>
                    case S(i) is
                      when L =>
                            RESULT := X;
                                          when X =>
                            RESULT := X;
                                 when OTHERS =>
                                end case;
                            when X =>
                      RESULT := X;
                                      end case;
                                      end loop;
                                      return RESULT;
             end RESOLVE;
         end FOURPACK;
```

Resolution Function for Subtype

```
package FOURPACK is
        type FOURVAL is (X, L, H, Z);
        type FOUR_VECTOR is array (Natural range <>)
                                   of FOURVAL;

        function RESOLVE (S: FOURVAL_VECTOR) return FOURVAL;

        subtype RESFOUR is RESOLVE FOURVAL,
end FOURPACK;
use Work.FOURPACK.all;
entity ...
architecture ...
signal A, B : RESFOUR;.
```

Summary

- Signals with Multiple Drivers require Resolution
- Resolution Functions are associated with subtype
- The resolved type and declaration and resolution function are usually declared in a Package

Predefined Attributes

- Data obtained from Blocks, Signals, Types and Subtypes
- Return values such as:
    - length of an array type
    - time since last signal change
    - range of values in a type

- Predefined attributes are useful for performing certain type of functions such as:
    - timing checks
    - bounds
    - clock edge detection
    - type conversion

Array Type Bound Example

```
process (...)
    type BIT_RANGE is array (31 DOWNTO 0) of Bit;
    variable LEFT_RANGE, RIGHT_RANGE,
           HIGH_RANGE, LOW_RANGE: integer;
begin
    LEFT_RANGE := BIT_RANGE'left;          -- returns 31
    RIGHT_RANGE := BIT_RANGE'right;        -- returns 0
    HIGH_RANGE := BIT_RANGE'high           -- returns 31
    LOW_RANGE := BIT_RANGE'low;            -- returns 0
end process;
```

Array Bound

Example

```
type T_RAM_DATA is array (0 TO 511) of integer;

variable RAM_DATA : T_RAM_DATA;
        .
        .
    for i in RAM_DATA'low TO RAM_DATA'high loop
        RAM_DATA(i) := 0;
    end loop;
```

Multi-Range Array Attributes

- 'left(N) -returns left bound of index range N
- 'right(N) -returns right bound of index range N
- 'high(N) -returns upper bound of index range N
- 'low(N) - returns lower bound of index range N

Example N= 1 2

```
variable: MEMORY (0 to 5, 0 to 7) of MEM_DATA;
          MEMORY 'right(2);
```

Array Length Attributes

- ‘length return of an array or array type

```
 process (...)
      type BIT4 is array (0 TO 3) of Bit;
      type BIT_STRANGE is array(10 TO 20) of Bit;
      variable LEN1, LEN2: integer;
begin
      LEN1 := BIT4’length;                    -- return 4
      LEN2 := BIT_STRANGE’length;             -- returns 11
end process;
```

Range Attributes

NAME ‘**range** - returns the declared range of a particular type

NAME’**reverse_range** - returns the declared range of a particular type in reverse order

Example:

```
function VECTOR_TO_INT(STUFF: Bit_Vector) return integer is
  variable RESULT: Integer := 0;
begin
          for i in STUFF ‘range loop
             ....
```

Type Attributes Position Function

TYPENAME ‘**succ(V)** - returns next value in type after input value

TYPENAME ’**pred(V)** - returns previous value in type before input value

TYPENAME ‘**leftof(V)** - returns value immediately to left of input value

TYPENAME ‘**rightof(V)** - returns value immediately to right of input value

TYPENAME ‘**pos(V)** - returns type position number from type value

TYPENAME ‘**val(P)** - returns type value from type position number

‘base - returns base type of type or subtype

Attributes Exercise

```
type COLOR is (RED, BLUE, GREEN, YELLOW, BLACK);
subtype COLOR_GUN is COLOR range RED to GREEN;
variable A: COLOR;
begin
    A:= COLOR'low;

    A:= COLOR 'succ(RED);

    A:= COLOR_GUN'base'right;

    A:= COLOR'base'left;

    A:= COLOR_GUN'base'succ(GREEN);
```

Attribute

Example

```
type CURRENT is range 0 TO 1000000
units uA:
      mA = 1000 uA;
      A = 1000 mA;
end units;
type VOLTAGE is range 0 TO 1000000
units uv
     mV = 1000 uV;
      V = 1000 mV;
end units;
type RESISTENCE is range 0 TO 1000000           ;
units  Ohm;
                KOhm = 1000 Ohm;
end units;
```

Attribute

Example (cont'd)

```
entity CALC_RES is
            port (I : in CURRENT; E : in VOLTAGE;
                 R : out RESISTANCE);
end CALC_RES;

architecture BEHAVE of CALC_RES is
begin
   process (I,E) begin
           R <= RESISTANCE'val(VOLTAGE'pos(I)/CURRENT'pos(E));
   end process;
end BEHAVE;
```

Signal Attributes

- SIGNAL’**event** - returns True if an event occurred on this signal during this delta
- SIGNAL’**active** - returns True if a transaction occurred this delta
- SIGNAL**’last_event** - returns the elapsed time since previous event
- SIGNAL**’last_value** - returns previous value of signal before last event
- SIGNAL**’last_active** - returns time elapsed since previous transaction

Signal Attribute

Example

```
library IEEE; use IEEE.std_logic_1164.all;
entity DFLOP is
      port (D, CLK : in std_logic; Q : out std_logic);
end DFLOP;
architecture DFF of DFLOP is
begin
      process (CLK)
      begin
                if (CLK = ‘1’) and (CLK’event) and (CLK’last_value = ‘0’) then
                   Q <= D;
                end if;
      end process;
end DFF;
```

Derived Signal Attribute

SIGNAL’**delayed** [(time)] - creates a signal that follows reference signal, delayed by time value

SIGNAL’**stable**[(time)] - creates a signal that is True when the reference signal has no events for time value.

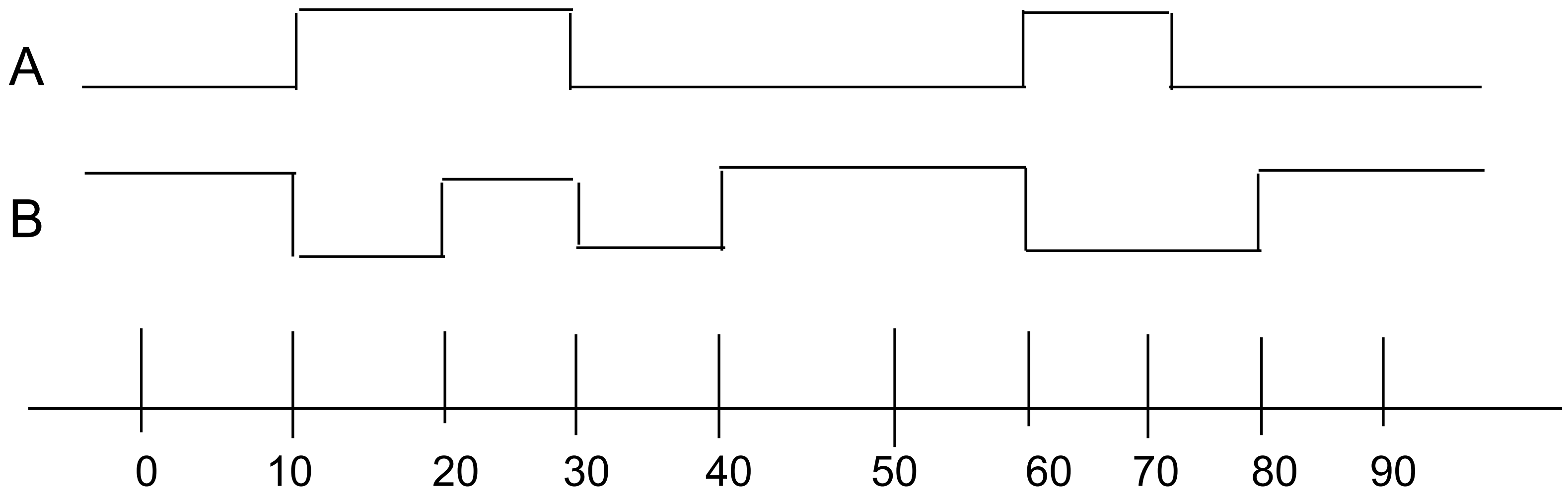
SIGNAL’**quiet**[(time)] - creates a signal that is True when the reference signal has no transactions for time value

SIGNAL ‘**transaction** - creates a signal of type Bit that toggles (has a transition) for every transaction on reference signal

Stable Attribute

Example

```
architecture BEHAVE of PULSE_GEN is
begin
        B <= A'stable(10 ns);
end PULSE_GEN;
```



User Defined Attributes

- Attach data to objects
- Data type user defined
- Data is constant
- Accessed with same syntax as predefined attributes

User Defined Attributes

Example

```
package BOARD_ATTRS is
type PACKAGE_TYPES is
    (LEADLESS, PGA, DIP);
attribute PACKAGE_TYPE :
    PACKAGE_TYPES;
attribute PACKAGE_LOC : Integer;
end BOARD_ATTRS;
use Work.BOARD_ATTRS.all;
entity BOARD is port(...); end BOARD;
architecture CPU_BOARD of BOARD is
```

User Defined Attributes

Example (cont'd)

```
component MC68040
generic (.......); port (........);
 end component;
 signal A: integer, B: PACKAGE_TYPE;
        attribute PACKAGE_TYPE of MC68040 : component is PGA
        attribute PACKAGE_LOC of MC68040 : component is 20;
        begin
                    A <= MC68040'PACKAGE_LOC;        -- returns 20
                    B <= MC68040'PACKAGE_TYPE        -- returns PGA
        end CPU_BOARD;
```

Recall: Three-State Logic

Type BIT3
- '0'
- '1'
- 'Z'

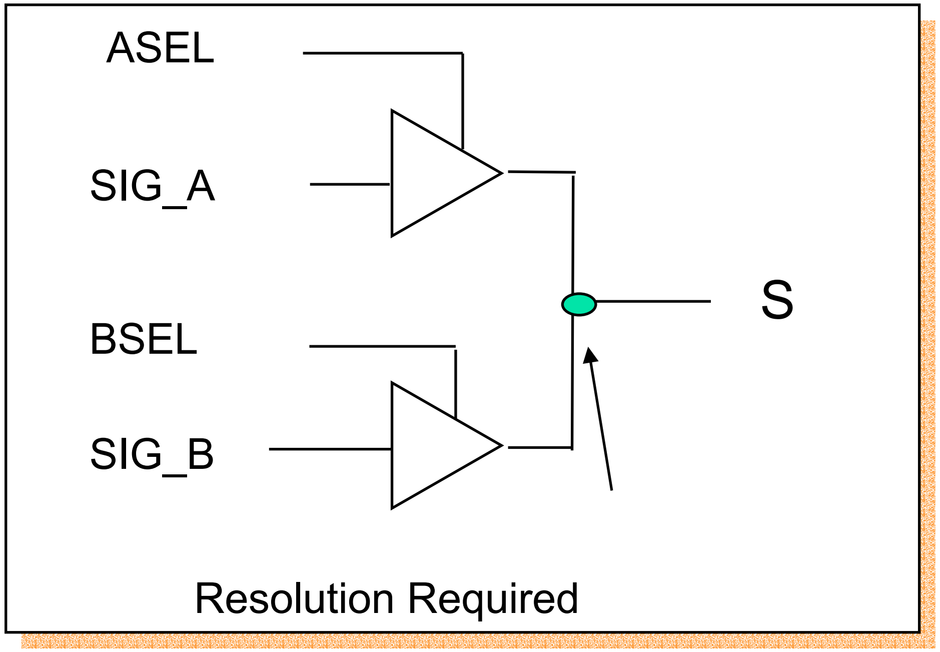
```
architecture …
begin
    C<= A;
    C<=B
 ...
```

How does a simulator resolve the logic values?

Three- State Multiple Drivers

Example

```
    architecture BEHAVE of STATE3 is
            signal S, SIG_A, SIG_B : BIT3;
    begin

       A: process (SIG_A, ASEL)
begin
      S <= 'Z';
    if (ASEL) then
      S <= SIG_A;
       end if;
       end process;
       B: process (SIG_B, BSEL)
begin
      S <= 'Z';
    if (BSEL) then
       S <= SIG_B;
      end if;
      end process;
  end BEHAVE;
```



Requires a resolution function for type BIT3

State Machines

To design a state machine in VHDL, each state can be translated to a case-when construct. The state transitions can then be specified in if-then-else statements.

For, example, to translate the state flow diagram into VHDL, we begin by defining an enumeration type, consisting of the state names, and declaring two signals of that type:

```
type STATETYPE is (IDLE, DECISION, READ, WRITE);

signal PRESENT_STATE, NEXT_STATE : STATETYPE;
```

State Machines (cont’d)

Next, we create a process. Next_state is determined by a function of the present_state and the inputs (read and read_write.)

```
STATE_COMB: process (PRESENT_STATE, READ_WRITE, READY)

begin
                ...

end process STATE_COMB;
```

Within the process we describe the state machine transitions. We open a case-when statement and specify the first case (when condition), which is for the idle state. For this case, we specify the outputs defined by the idle state and transitions from it :

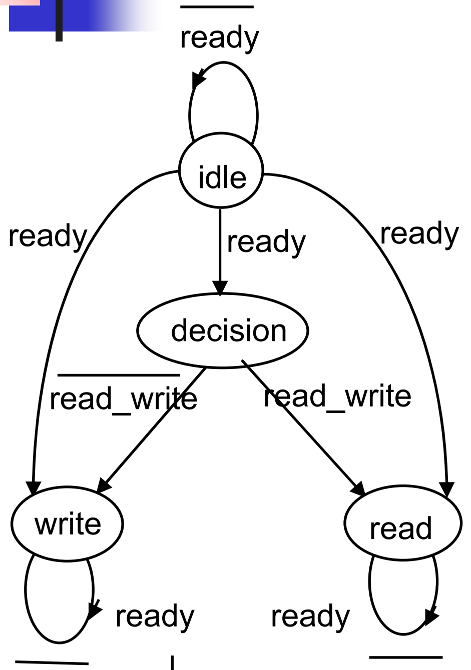
State Machines (cont'd)

```
STATE_COMB: process (PRESENT_STATE, READ_WRITE, READY)
          begin
                    case PRESENT_STATE is
                              WHEN idle =>
                              OE <= '0'; WE <= '0';
                              if READY = '1' then
                                 NEXT_STATE <= DECISION;
                              else                                  -- else not necessary
                                 NEXT_STATE => IDLE;      -- included for readability
                              end if;
```

The above process indicates what the next_state assignment will be based on present-state and present inputs, but it does not indicate when the next state becomes the present state .This happens synchronously, on the rising edge of a clock.

Two-Process FSM

Example



State	Outputs OE	WE
Idle	0	0
decision	0	0
write	0	1
read	1	0

```
entity EXAMPLE is
            port( READ_WRITE, READY, CLK              : in Bit;
                  OE, WE                              : out Bit);
end example;

architecture STATE_MACHINE of EXAMPLE is
            type STATETYPE is (IDLE, DECISION, READ, WRITE);
            signal PRESENT_STATE, NEXT_STATE : STATETYPE;
begin
STATE_COMB: PROCESS(PRESENT_STATE, READ_WRITE, READY)
            case PRESENT_STATE is
              when IDLE =>          OE  <=  '0'; WE  <=  '0';
                      if READY = '1' then
                        NEXT_STATE  <=  DECISION;
                      else
                        NEXT_STATE  <=  IDLE;
                      end if;
              when DECISION => OE  <=  '0'; WE  <=  '0';
                      if (READ_WRITE = '1') then
                        NEXT_STATE  <= READ;
                      else
                        NEXT_STATE  <=  WRITE;

                      end if;
```

Example- Two- Process FSM

```
when  READ  =>         OE   <=  ‘1’;  WE  <=  ‘0’;
      (if READY  = ‘1’ ) then
                 NEXT_STATE  <=  IDLE;
      else
                 NEXT_STATE  <=  READ;
      end if;
when WRITE  =>    OE  <=  ‘0’;  WE  <=  ‘1’;
      if  (READY  =  ‘1’)  then
                 NEXT_STATE  <=  IDLE;
      else
                 NEXT_STATE  <=  WRITE;
      end  if;
end case;
end  process STATE_COMB;
```

Two- Process FSM

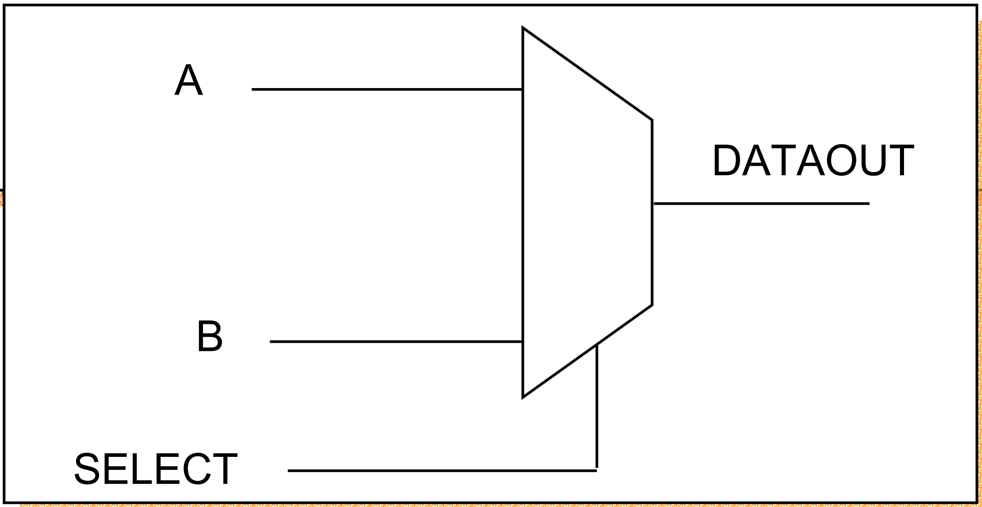
Example

```
STATE_CLOCKED : process(CLK)
begin
        if (CLK’EVENT AND CLK = ‘1’ ) then
          PRESENT_STATE  <=  NEXT_STATE;
        end if;
end process STATE_CLOCKED;

end architecture STATE_MACHINE;
                -- “architecture “ is optional; for clarity.
```

Combinational/Sequential VHDL Examples

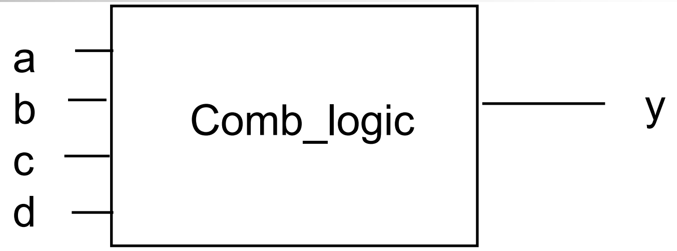
VHDL MUX

```
process (SELECT, A, B)
begin
    case SELECT is                                        -- no priority given but
        when '0' => DATAOUT <= A;         -- using if loop we
        when '1' => DATAOUT <= B;         -- can prioritize
        when others => null;
    end case;

end process;
```



Combinational Logic

Combinational Logic



Models representing a combinational block described by the truth table are given below:

CD \ AB	"00"	"01"	"11"	"10"
"00"	'0'	'1'	'1'	'0'
"01"	'1'	'1'	'1'	'0'
"11"	'0'	'1'	'1'	'1'
"10"	'0'	'0'	'0'	'0'

$y = C'b+db+cda+c'da'$

Combinational Logic

```
entity COMB_LOGIC is     -- y = c'd+db+cda+c'da'
       generic (P_DELAY : time);
       port (A, B, C, D : in Bit;
                  Y : out Bit);
end COMB_LOGIC;          -- behavior from the truth table
                                   -- process statement and sequential statements
architecture ARCH2 of COMB_LOGIC is
begin
  process (A, B, C, D)
       variable TEMP : Bit_Vector (3 DOWNTO 0);
       begin
           TEMP := A&B&C&D;
            case TEMP is
             when b"0000" | b"1000" | b"1001 | b"0011" |
             b"0010" | b"0110" | b"1110" | b"1010"  =>
             y <= '0';
             when OTHERS =>   y <= '1';
            end case;
   end process;
end ARCH2;
```

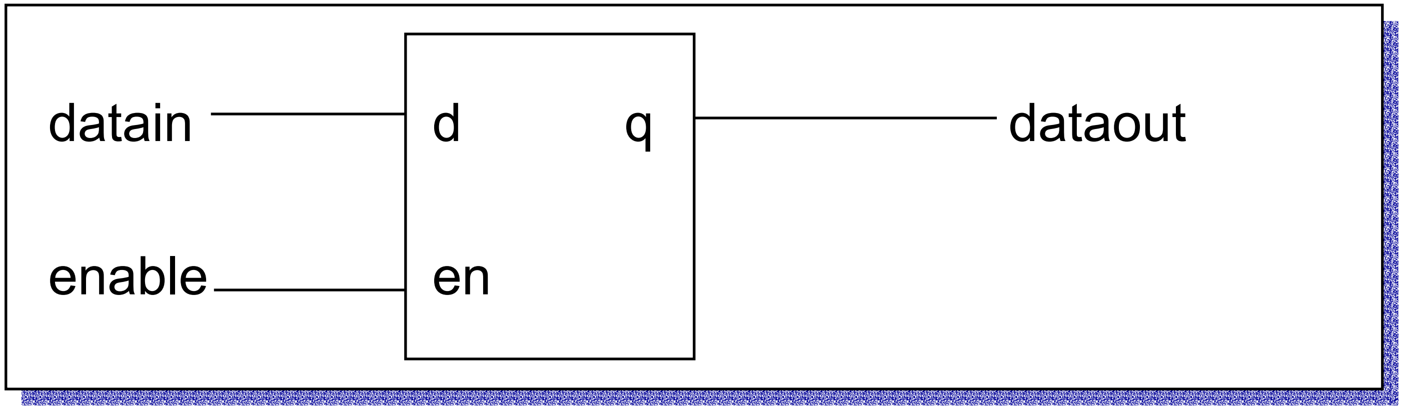
Combinational Logic

----- y = c'b+db+cda+c'da'

```
entity COMB_LOGIC is
          generic (P_DELAY : TIME);
          port (A, B, C, D: in Bit;
                    Y : out Bit);
end COMB_LOGIC;
                    -- simple signal assignment statement, RTL
architecture ARCH1 of COMB_LOGIC is
begin
          Y <= ((NOT C) AND B) OR (D AND B) OR
          (C AND D AND A) OR ((NOT C) AND D AND (NOT A));
end ARCH1;
```

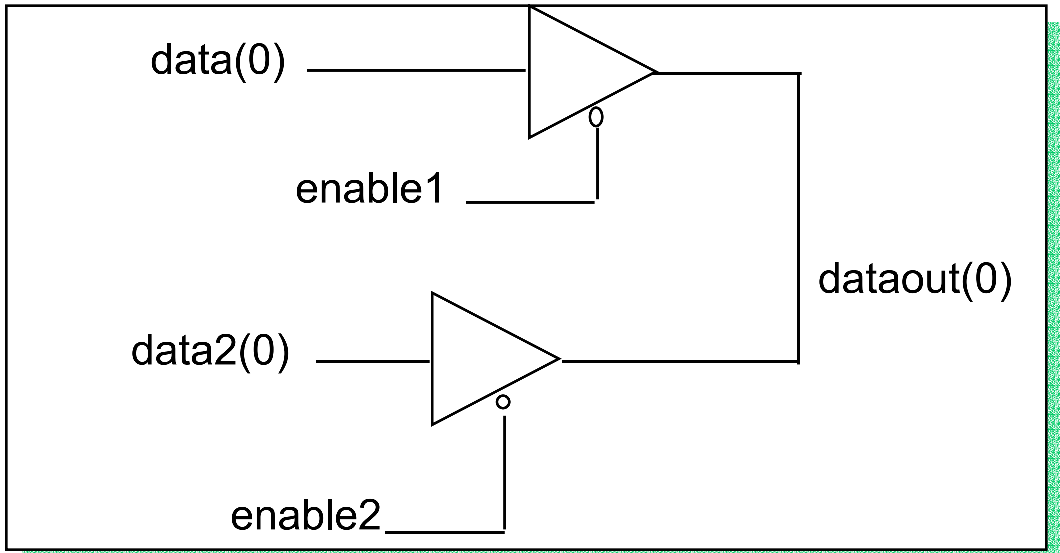
VHDL Latch

```
process (DATAIN, ENABLE)
begin
          if ENABLE = '1'                                -- it does not explicitly
          DATAOUT <= DATAIN;                             -- defines other
                                                         -- conditional values
 end if;
end process;
```



VHDL Tri-State Bus

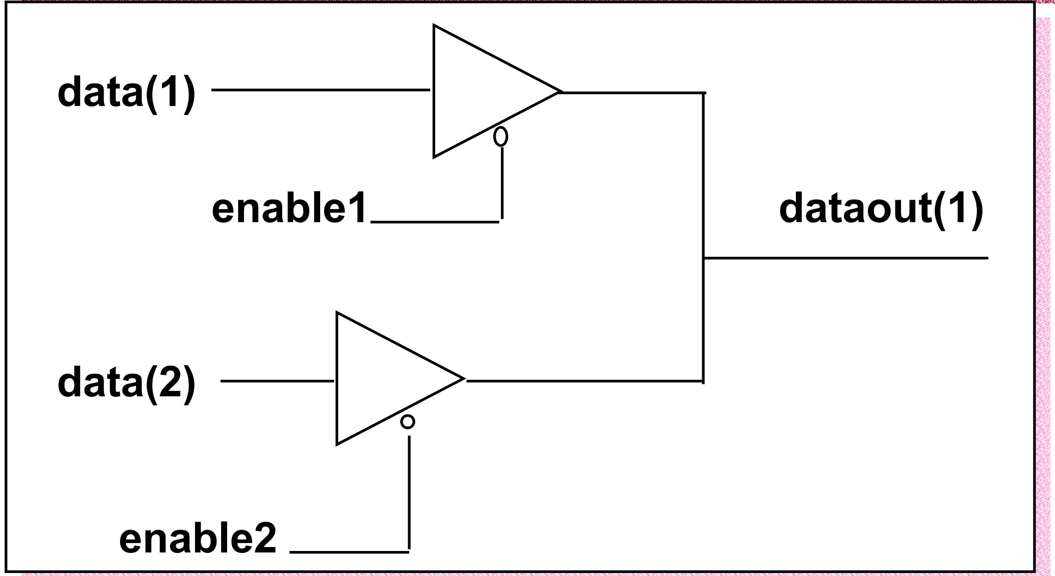
```
process (DATA1, ENABLE1)
begin
        if ENABLE = ‘0’ then
                DATA <= DATA1;          -- drive DATAOUT
        else
                DATAOUT <= “ZZ”;
        end if;
end process;
```



California State University Northridge

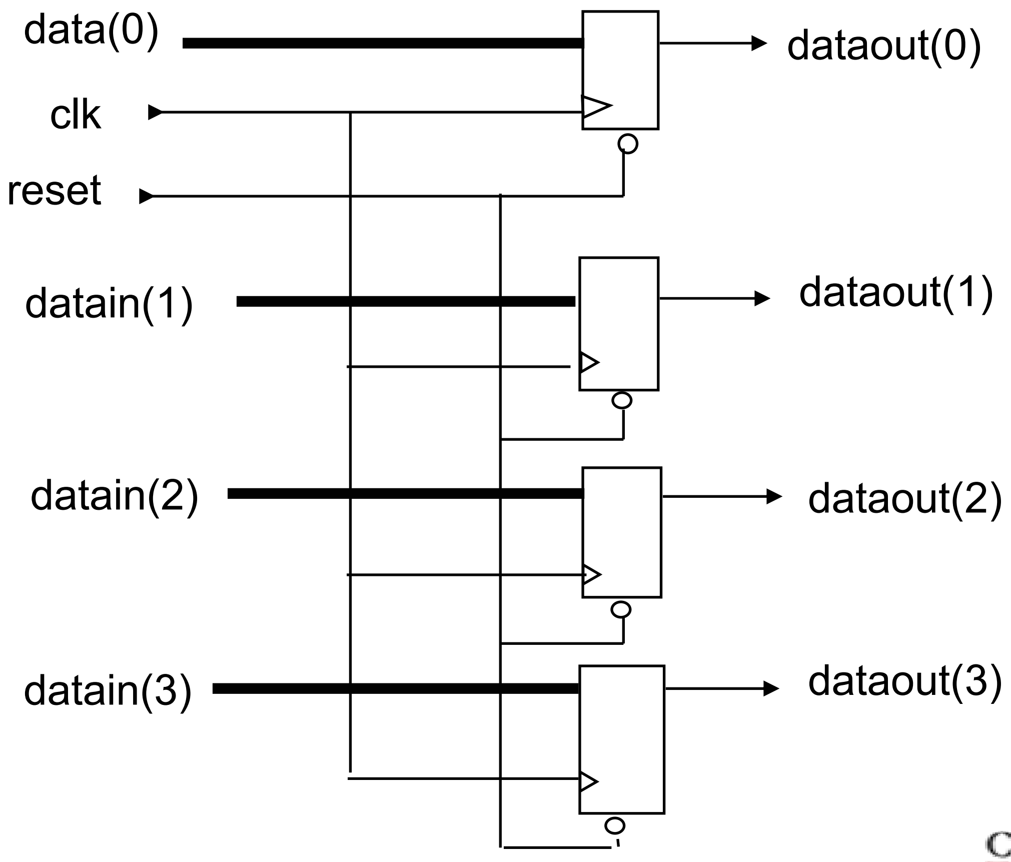
VHDL Tri-State Bus(cont)

```
process (DATA2, ENABLE2)
begin
        if ENABLE = '0' then
                DATAOUT <= DATA2        -- also drive's
        else
                DATAOUT <= "ZZ";
        end if;
end process;
```



VHDL Register

asynchronous reset

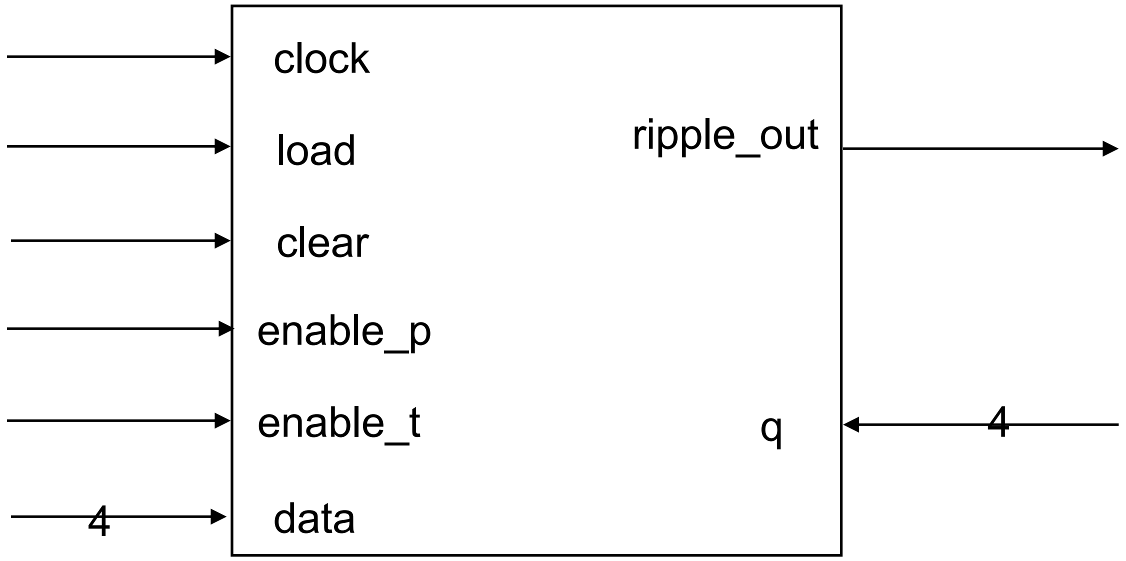


VHDL Register

```
process (RESET, CLK)
begin
        if RESET = '0' then
                DATAOUT <= "0000";
        elsif   CLK'event and CLK = '1' then -- rising edge
                DATAOUT <= DATAIN; -- of clock
        end if;
end process;
```

Asynchronous reset does not depend on clock.

Interface Description of 74LS161 4-bit Synchronous Counter



Interface Description of 74LS161 4-bit Synchronous Counter

Function Table of 74LS161 4 - bit synchronous counter

Clock	load	clear	enable_p	enable_t	Operation
X	X	low	X	X	asynchronous clear
rising	low	high	X	X	load input data
rising	high	high	high	high	increment court

* X means Don’t Care
* ripple_out = 1 when count = decimal 15 and enable_t = ‘1’