

# CS390: Microprocessors and Interfaces Lab

# VHDL

**18 January, 2005**

# What is VHDL

---

- **VHSIC Hardware Description Language**
  - a large and complicated programming language with many constructs and semantics meanings
  - to support the design and development of circuits of large scale (circuit complexity and project management)
  - a powerful modeling tool

# Background

---

- 1960's - 1980's
  - over 200 languages, either proprietary or academic
- 1983 VHSIC Program initiates definition of VHDL
- 1987 VHDL Standard (IEEE 1076) approved
- 1990 Verilog dominates the marketplace

# Background, continued

---

- 1992 IEEE 1164 (abstract data types for different signal characteristics, i.e. 3, 4, 9-valued logic standard)
- 1993 VHDL re-balloted
  - minor changes make it more user-friendly.
- 1994 Widespread acceptance of VHDL.

# Other HDLs

---

- **Verilog -- very popular, has been used for a long time**
  - C like syntax
  - wide support of simulation libraries of semiconductor devices
  - lack of higher-level management features -- such as VHDL's configuration, package, and library
- **PLD-oriented HDLs (ABEL, PALASM, etc.)**
  - simple and low cost
  - specialized for PLD device synthesis

# VHDL Design Example

---

- **Problem: Design a single bit half adder with carry and enable**
- **Specifications**
  - Inputs and outputs are each one bit
  - When enable is high, result gets  $x$  plus  $y$
  - When enable is high, carry gets any carry of  $x$  plus  $y$
  - Outputs are zero when enable input is low

# VHDL Design Example

## Entity Declaration

- As a first step, the entity declaration describes the interface of the component
  - input and output *ports* are declared

```
ENTITY half_adder IS  
  
    PORT( x, y, enable: IN BIT;  
          carry, result: OUT BIT);  
  
END half_adder;
```



# VHDL Design Example

## Behavioral Specification

- A high level description can be used to describe the function of the adder

```
ARCHITECTURE half_adder_a OF half_adder IS
    BEGIN
        PROCESS (x, y, enable)
            BEGIN
                IF enable = '1' THEN
                    result <= x XOR y;
                    carry <= x AND y;

                ELSE
                    carry <= '0';
                    result <= '0';

                END IF;
            END PROCESS;
        END half_adder_a;
```

- The model can then be simulated to verify correct functionality of the component



# VHDL Design Example

## Data Flow Specification

---

- A second method is to use logic equations to develop a data flow description

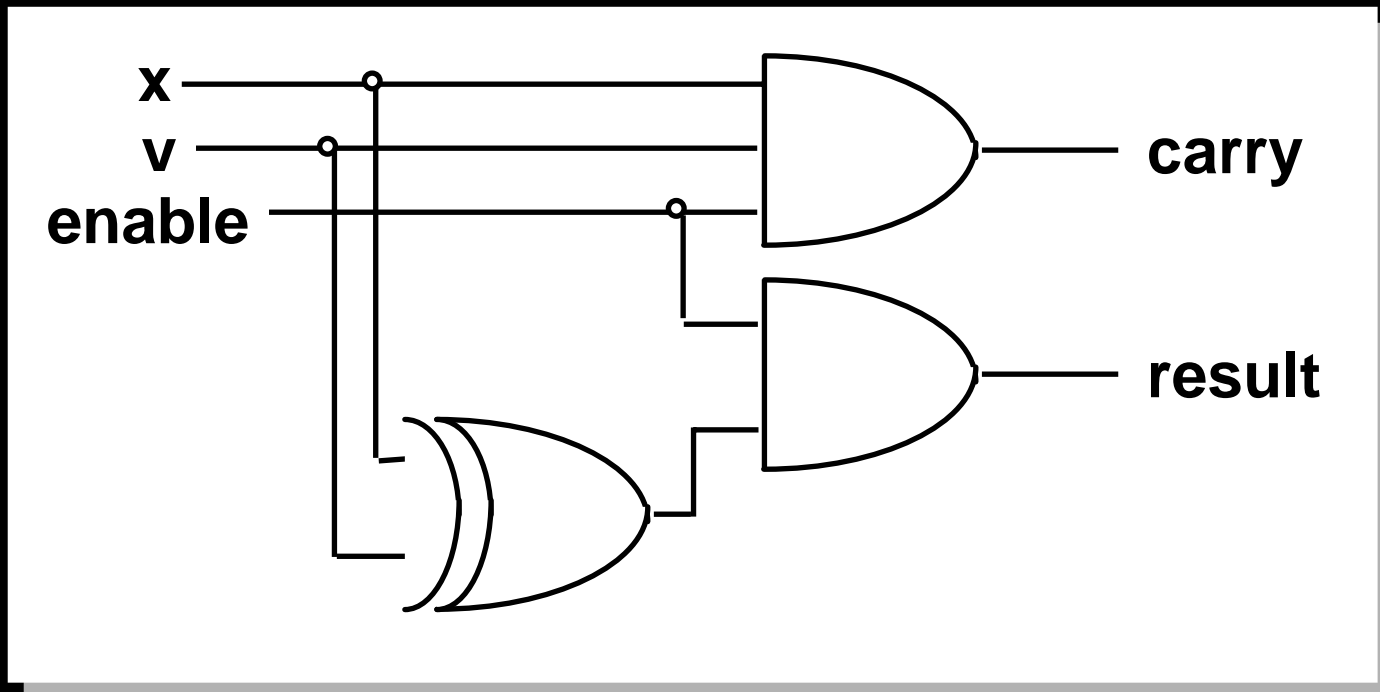
```
ARCHITECTURE half_adder_b OF half_adder
IS
    BEGIN
        carry <= enable AND (x AND y);
        result <= enable AND (x XOR y);
    END half_adder_b;
```

- Again, the model can be simulated at this level to confirm the logic equations

# VHDL Design Example

## Structural Specification

- As a third method, a structural description can be created from predescribed components



- These gates can be pulled from a library of parts

# VHDL Design Example

## Structural Specification (Cont.)

```
ARCHITECTURE half_adder_c OF half_adder IS

    COMPONENT and2
        PORT (in0, in1 : IN BIT;
              out0 : OUT BIT);
    END COMPONENT;

    COMPONENT and3
        PORT (in0, in1, in2 : IN BIT;
              out0 : OUT BIT);
    END COMPONENT;

    COMPONENT xor2
        PORT (in0, in1 : IN BIT;
              out0 : OUT BIT);
    END COMPONENT;

    FOR ALL : and2 USE ENTITY gate_lib.and2_Nty(and2_a);
    FOR ALL : and3 USE ENTITY gate_lib.and3_Nty(and3_a);
    FOR ALL : xor2 USE ENTITY gate_lib.xor2_Nty(xor2_a);

-- description is continued on next slide
```

# VHDL Design Example

## Structural Specification (cont.)

---

```
-- continuing half_adder_c description

SIGNAL xor_res : BIT; -- internal signal
-- Note that other signals are already declared in entity

BEGIN

    A0 : and2 PORT MAP (enable, xor_res, result);
    A1 : and3 PORT MAP (x, y, enable, carry);
    X0 : xor2 PORT MAP (x, y, xor_res);

END half_adder_c;
```

# Language Syntax

---

- **BNF format for syntax rule**
- **“<=“ ---- is defined to be**
  - `variable_assignment <= target;`
- **[ clause ] ---- optional**
- **{ clause } ---- optional and can be repeated**
- **| ---- alternatives**
  - `mode <= in | out | inout`
  - `constant_declaration <=`  
`constant identifier { , . . . } : subtype_indication [ := expression ] ;`
  - `constant number_of_byte : integer := 4;`

# Example

---

- **if\_statement**  $\Leftarrow$   
[ if\_label : ]  
if boolean\_expression then  
    { sequential\_statement }  
{ elsif boolean\_expression  
  then  
    { sequential\_statement }  
[ else  
  { sequential\_statement } ]  
end if [ if\_label ] ;
- **maximum**: if  $a > b$  then  
    if  $a > c$  then  
        result := a ;  
    else  
        result := c ;  
    end if ;  
  elsif  $b > c$  then  
    result := b ;  
  else  
    result := c ;  
  end if ;

# Primitive Objects in VHDL

---

- **Object** -- a named item that has a value of a specified type
- **4 classes of objects:**
  - constants, variables, signals, and files
- **Types** –
  - scalar type: individual values that are ordered
    - discrete, floating-point, and physical
  - composite type: array and record
  - file type
  - access type
- **VHDL is a strongly typed language**

# Primitive Objects in VHDL

---

- **Variable -- often no direct correspondence in hardware. Holds values that change over time.**
  - Example: `variable i_slice: integer range 0 to Reg_size-1;`
  - assignment (`:=`) – immediately overwrites the existing value
- **Signal -- analogous to wire or device output. Holds values which may change over time.**
  - used to establish connectivity and pass values between concurrently active design elements. A value for a signal is computed and added to a waveform for future application.
  - Examples            `signal preset, clear: bit;`  
                         `signal CS1: bit;`



# Primitive Data Types

---

- **Integer -- to count in discrete steps**
  - type *integer* is range  $-(2^{31}-1)$  to  $2^{31}-1$  (*at least*)
- **Bit -- to represent the most commonly occurring model of discrete digital circuit value**
  - type *bit* is ('0','1');
- **Boolean -- to represent decisions, outcomes of logical expression evaluation, especially used in procedural control flow**
  - type *boolean* is (false, true);
- **Enumeration types -- values are in a list of identifiers | character\_literal**
  - type *octal\_digital* is ('0', '1', '2', '3', '4', '5', '6', '7');

# Primitive Data Types

---

- **Character** -- to create models which work with textual data and communication / control symbols to provide textual information to designer-users through design tool windows to the model
  - type character is the set of ASCII symbols
- **Reals** to represent analog, continuously variable, measurable values in design space. (at least  $-1.0E+38$  to  $1.0E+38$  and with 6 decimal digits of precision)
- The predefined package *standard* (stored in the library *std*)
- **Type declaration**
  - type *byte\_integer* is integer range -128 to 127;

# STANDARD.VHD

---

```
-- This is Package STANDARD as defined in the VHDL 1992 Language
  Reference Manual.
package standard is
  type boolean is (false,true);
  type bit is ('0', '1');
  type character is (
    nul, soh, stx, etx, eot, ....
    '@', 'A', 'B', 'C', 'D', ....);
  type severity_level is (note, warning, error, failure);
  type integer is range -2147483647 to 2147483647;
  type real is range -1.0E308 to 1.0E308;
  type time is range -2147483647 to 2147483647
    units
      fs;
      ps = 1000 fs;
      ...
      hr = 60 min;
    end units;
```

# STANDARD.VHD (cont'd)

---

```
subtype delay_length is time range 0 fs to time'high;
impure function now return delay_length;
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
type string is array (positive range <>) of character;
type bit_vector is array (natural range <>) of bit;
type file_open_kind is (
    read_mode,
    write_mode,
    append_mode);
type file_open_status is (
    open_ok,
    status_error,
    name_error,
    mode_error);
attribute foreign : string;
end standard;
```



# Generics

---

- **Pass information from its environment into the design unit which is not time-varying**
- **Very useful for creating and using generalized designs.**

# Generics: Example

```
LIBRARY IEEE;  
USE IEEE.std_logic_1164.all;  
USE IEEE.std_logic_arith.all;
```

```
ENTITY Generic_Mux IS
```

```
    GENERIC (n: INTEGER);
```

```
    PORT (Y: OUT std_logic_vector(n-1 downto 0);
```

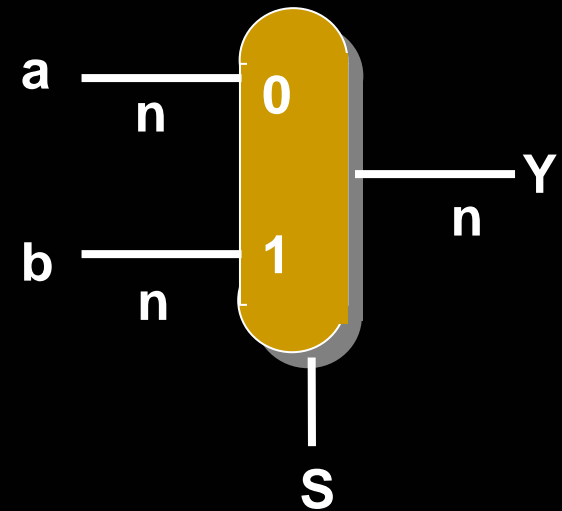
```
          a: IN  std_logic_vector(n-1 downto 0);
```

```
          b: IN  std_logic_vector(n-1 downto 0);
```

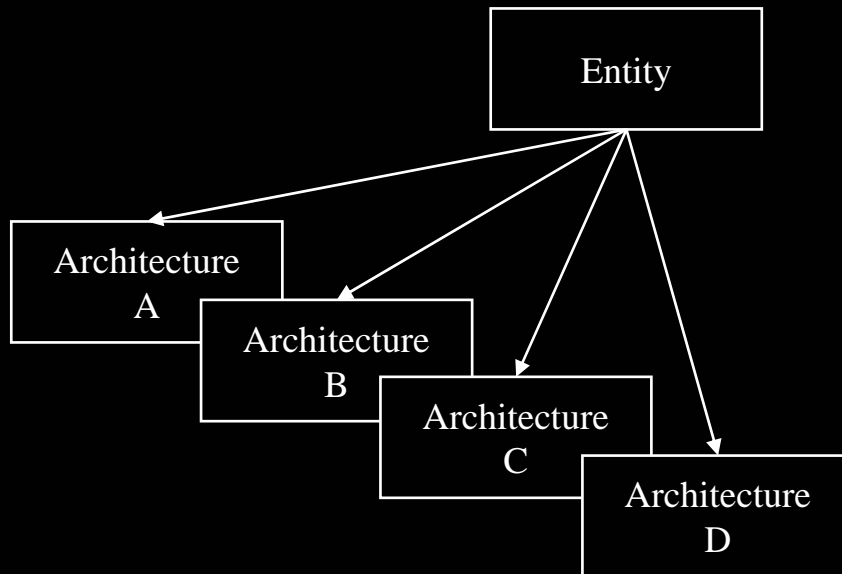
```
          S: IN  std_logic
```

```
);
```

```
END ENTITY;
```



# Entities and Architectures



- **Entities**
  - are Design Bodies
  - provide the Interface description
- **Architectures**
  - are concurrent
  - may be behavioral
  - may be structural



# Entities and Architecture

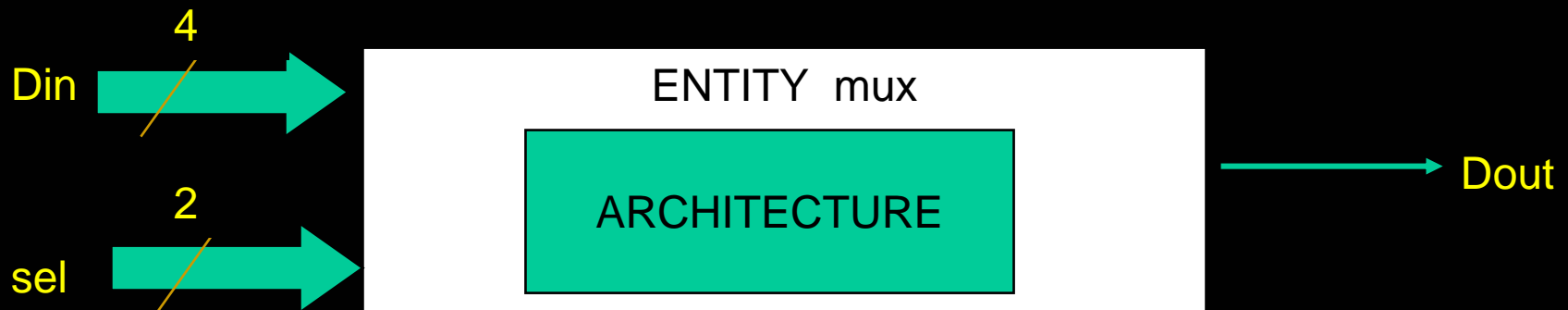
- **Entity**

- External view: Pin-out description, Interface description, I-O port definition etc

- **Architecture**

- Internal view

- ➔ Structural description: Gates, wires etc.
- ➔ Behavioral description: functions, procedures, RTL description



# Ports

---

- Pass information through the interface which is *time-varying*.
- Are signal objects
  - *connected together by signals*
  - *used to pass values between concurrently active units.*

# Interface Modes

---

- **Represent direction of value flow**
- **In entities, components, and blocks the modes may be:**
  - **IN**      within the design unit (both entity and body) the value may be read, but not written.
  - **OUT**     within the design unit (both entity and body) the value may be written, but not read.
  - **INOUT**   within the design unit (both entity and body) the value may be both read and written.

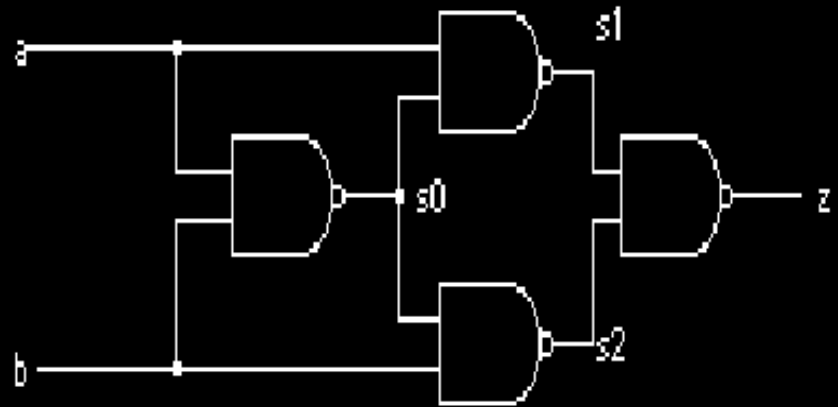
# Format of an Architecture

---

```
architecture identifier of entity_identifier is
  -- local declarations, typically signals
begin
  -- concurrent statements
end identifier ;
```

# Simple Example

```
entity XOR
  port (a,b:in bit;
        z : out bit );
end;
architecture nand_gates of
  XOR is
    signal s0, s1, s2:
      bit;
  begin
    s0 <= a nand b;
    s1 <= a nand s0;
    s2 <= b nand s0;
    z <= s1 nand s2;
  end nand_gates;
```



# Hierarchical Design Strategies

---

- **Bottom Up Strategy**

- create low level and auxiliary models first, entity and architecture
- once low level entities are present, they can be used as components in the next higher level architectural body.

- **Top Down Strategy**

- create highest level entity and architecture first, creating only the interface definitions (entity declarations) for lower level architectures

# Packages

---

- An important way of organizing the data.
- A collection of related declaration grouped to serve a common purpose.
- The external view of a package is specified in 'package declaration'.
- Its implementation is provided in the 'package body'.
- The packages can be shared among models.
- Several predefined packages exist, such as IEEE standard packages.

# Packages

---

- **Packages consist of two parts**
  - Package declaration -- contains declarations of objects defined in the package
  - Package body -- contains necessary definitions for certain objects in package declaration
    - e.g. subprogram descriptions
- **Examples of VHDL items included in packages :**
  - Basic declarations
    - Types, subtypes
    - Constants
    - Subprograms
    - Use clause



# Packages Declaration

- **An example of a package declaration :**

```
PACKAGE my_stuff IS
  TYPE binary IS ( ON, OFF );
  CONSTANT PI : REAL := 3.14;
  CONSTANT My_ID : INTEGER;
  PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
                     SIGNAL temp_result, temp_carry : OUT BIT);
END my_stuff;
```

- **Note some items only require declaration while others need further detail provided in subsequent package body**
  - for type and subtype definitions, declaration is sufficient
  - subprograms require declarations and descriptions

# Packages

## Package Body

- The package body includes the necessary functional descriptions needed for objects declared in the package declaration
  - e.g. subprogram descriptions, assignments to constants

```
PACKAGE BODY my_stuff IS
    CONSTANT My_ID : INTEGER := 2;

    PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
        SIGNAL temp_result, temp_carry : OUT BIT) IS
    BEGIN
        -- this function can return a carry
        temp_result <= (a XOR b) AND en;
        temp_carry <= a AND b AND en;
    END add_bits3;
END my_stuff;
```

# Packages

## Use Clause

- Packages must be made visible before their contents can be used
  - The USE clause makes packages visible to entities, architectures, and other packages

```
-- use only the binary and add_bits3 declarations
USE my_stuff.binary, my_stuff.add_bits3;

... ENTITY declaration...
... ARCHITECTURE declaration ...
```

```
-- use all of the declarations in package my_stuff
USE my_stuff.ALL;

... ENTITY declaration...
... ARCHITECTURE declaration
```

# Libraries

---

- **Analogous to directories of files**
  - VHDL libraries contain analyzed (i.e. *compiled*) VHDL entities, architectures, and packages
- **Facilitate administration of configuration and revision control**
  - E.g. libraries of previous designs
- **Libraries accessed via an assigned logical name**
  - Current design unit is compiled into the *Work* library
  - Both *Work* and *STD* libraries are always available
  - Many other libraries usually supplied by VHDL simulator vendor
    - ➔ E.g. proprietary libraries and IEEE standard libraries

# Packages: Example

```
package PKG is -- package declaration
  type T1 is ...
  type T2 is ...
  constant C : integer;
  procedure P1 (...);
end PKG;
```

```
package body PKG is -- package body
  type T3 is ...
  C := 17;
  procedure P1 (...) is
  ..
  end P1;
  procedure P2 (...) is
  ..
  end P2;
end PKG;
```

# Special Library: Work


---

- The identifier “work” is a special library that maps on to the present directory.
- All the design units in the present directory are visible to all models.
- Hence, an explicit declaration of “work” library is not required.
- However, one needs to specify the “work” when accessing declarations and design units in other files.

# Work Library: Example

```
entity test_bench is
end entity test_bench;

architecture test_reg4 of test_bench is
    signal d0, d1, d2, d3, en, clk, q0, q1, q2, q3 : bit;
begin
    dut : entity work.reg4(behav)
        port map ( d0, d1, d2, d3, en, clk, q0, q1, q2, q3 );
    stimulus : process is
    begin
        d0 <= '1'; d1 <= '1'; d2 <= '1'; d3 <= '1'; wait for 20 ns;
        en <= '0'; clk <= '0'; wait for 20 ns;
        en <= '1'; wait for 20 ns;
        clk <= '1'; wait for 20 ns;
        d0 <= '0'; d1 <= '0'; d2 <= '0'; d3 <= '0'; wait for 20 ns;
        en <= '0'; wait for 20 ns;
        ...
        wait;
    end process stimulus;
end architecture test_reg4;
```



# Processes



# Procedural Modeling USE: High level abstraction of behavior

---

```
entity traffic_light_controller
  generic (
    yellow_time : time;
    min_hwygreen : time;
    max_hwyred : time );
  port (
    farmroad_trip : in boolean;
    farmroad_light : out color;
    highway_light : out color );
end traffic_light_controller;

architecture specification of traffic_light_controller is begin
  . . .
```

# Procedural Modeling USE: High level abstraction of behavior

---

```
architecture specification of traffic_light_controller is begin
  cycle: process is
  begin
      highway_light <= green;
      farmroad_light <= red;
  wait for min_green;
  wait until farmroad_trip;
      highway_light <= yellow;
  wait for yellow_time;
      highway_light <= red;
      farmroad_light <= green;
  wait until not farmroad_trip for max_hwyred;
      farmroad_light <= yellow;
  wait for yellow_time;
  end process;
end specification;
```

# Procedural Modeling Use: Detailed Modeling of Behavior

---

Example: Timed Behavior of Primitive Elements

```
AND_n: process (x) is -- x is an array of bit
    variable Zvar : bit;

begin
    Zvar := '1';
    for i in x'range loop -- for every i in the range of x
        if x(i) = '0' then
            Zvar := '0' ;
            exit ;
        end if;
    end loop;
    Z <= Zvar after Tprop ;
end process AND_n;
```

# Process Statement

---

Is the “wrapper” around a sequential routine to compute the behavior desired for the design at a specific moment in time.

```
label: process [ (signal list) ] is  
  { declarations }  
begin  
  { sequential statements }  
  -- (typically ended by a wait statement)  
end process [ label ];
```

# Process Execution Model

---

- Executes once (at  $\text{TIME} = 0$ ) -- initialization, running till it hits a WAIT statement.
- Time advances until the wait condition is satisfied, then execution resumes.
- Executes in an endless loop,
  - **interrupted only by WAIT statements;**
  - **bottom of the process contains an implicit "go to the top."**
- **TIME DOES NOT ADVANCE** within a process; it advances during a WAIT statement.

# Process Statement

## - A Concurrent Statement

---

- A process is a kind of concurrent statement.
  - *includes declarations, sequential body, and all*
- Evaluation of a process is triggered when one of a list of signals in the wait statement changes value
- Note: Just because a process is sequential does NOT mean it is modeling the sequential behavior of a design.
  - *a description of functional behavior*
  - *For example: the AND\_n process example is the model of a combinational logic element.*

```
library IEEE;
use IEEE.std_logic_1164.all;

entity andcircuit is
    port(
        in1, in2, in3 : in std_ulogic;
        out1 : out std_ulogic
    );
end andcircuit;
```

# Initialization of Objects

---

- Signals, variables, constants can all be set to default values:

*signal enable : bit := 0;*

*variable Fval : std\_logic := '0';*

*constant Tplh : Time := Tprop + K \* load ;*

where Tprop, K, load are generics or constants,

- Ports can be initialized by

*entity xyz port ( a : in bit := '0'; . . . )*



# Signal assignment

---

- **Signals**

- Used to communicate between *concurrently executing processes*.
- Within a process they continue to have the form  
*sig <= waveform ;*
- Means that for the signal a sequence of value updating events is to be scheduled for the future.

# Variable assignment

---

- **Variables:**

- Exist within procedural bodies, like processes, functions, and procedures. Not visible to others.
- Variable assignment statements appear as follows:

***var := expression;***

- Used within the sequential body just as in other procedural languages.

$X \leftarrow Y;$

$X := Y;$

$Y \leftarrow X;$

$Y := X;$

# Misuse of Sequential Signal Assignments

---

- **Note a signal does not take on its new value *until time advances.***
- **Until the process hits a WAIT (hold, or suspend) statement, simulation time does not advance,**
- **Therefore, the signal will never be updated before the WAIT**
  - **and may not be updated even after the WAIT is complete if the WAIT completed faster than the signal update has delay associated with it.**

# Signals and Variables

- This example highlights the difference between signals and variables

```
ARCHITECTURE test1 OF mux IS
    SIGNAL x : BIT := '1';
    SIGNAL y : BIT := '0';
BEGIN
    PROCESS (in_sig, x, y)
    BEGIN
        x <= in_sig XOR y;
        y <= in_sig XOR x;
    END PROCESS;
END test1;
```

```
ARCHITECTURE test2 OF mux IS
    SIGNAL y : BIT := '0';
BEGIN
    PROCESS (in_sig, y)
        VARIABLE x : BIT := '1';
    BEGIN
        x := in_sig XOR y;
        y <= in_sig XOR x;
    END PROCESS;
END test2;
```

- Assuming a 1 to 0 transition on *in\_sig*, what are the resulting values for *y* in the both cases?

# VHDL Objects

## Signals vs Variables

- A key difference between variables and signals is the assignment delay

```
ARCHITECTURE sig_ex OF test IS
  PROCESS (a, b, c, out_1)
  BEGIN
    out_1 <= a NAND b;
    out_2 <= out_1 XOR c;
  END PROCESS;
END sig_ex;
```

Time	a	b	c	out_1	out_2
0	0	1	1	1	0
1	1	1	1	1	0
1+d	1	1	1	0	0
1+2d	1	1	1	0	1

# VHDL Objects

## Signals vs Variables (Cont.)

```
ARCHITECTURE var_ex OF test IS
BEGIN
    PROCESS (a, b, c)
    VARIABLE out_3 : BIT;
    BEGIN
        out_3 := a NAND b;
        out_4 <= out_3 XOR c;
    END PROCESS;
END var_ex;
```

Time	a	b	c	out_3	out_4
0	0	1	1	1	0
1	1	1	1	0	0
1+d	1	1	1	0	1

# Wait Statements

---

```
wait_stmt <=
    [ label : ] wait [ on signal_name{ , ... } ]
                    [ until boolean_expr ]
                    [ for time_expr ] ;
```

- wait;
- wait on a, b, c;
- wait until x = 1;
- wait for 100 ns;

# Wait on

---

- process being suspended until an event takes place on any one of the signals.
- The list of signals is also called a sensitivity list.

half\_adder: process is  
begin

s <= a xor b after 10 ns;

c <= a and b after 10 ns;

end process;

half\_adder: process (a, b) is  
begin

s <= a xor b after 10 ns;

c <= a and b after 10 ns;

wait on a, b;

end process;



# Wait until

---

wait on s1, s2, s3 until *condition*;

- The condition is evaluated only when an event occurs on a signal in the sensitivity list.
- The process is resumed when the condition evaluates to TRUE.
- Hence the process is resumed when
  - An event occurs in the sensitivity list and
  - The condition evaluates to TRUE.

## ***Example Use of Multiple Wait Statements: CPU and Memory Handshaking***

---

```
Memory: process is
begin
    DAV <= '0';
    wait until Mem_Req = '1';
    Data <= ROM_DATA(Address) after 50 ns;
    DAV <= '1' after 60 ns;
    wait until Mem_Req = '0';
end process;

CPU_Read: process is
begin
    Mem_Req <= '0';
    wait until ... the need for memory read ;
    Address <= . . . address value . . .
    Mem_Req <= '1' after 10 ns;
    wait until DAV = '1';
    MD_Reg <= Data;
end process;
```

# label: process ( a, b, c, d ) is

where signals a, b, c, d are the sensitivity list

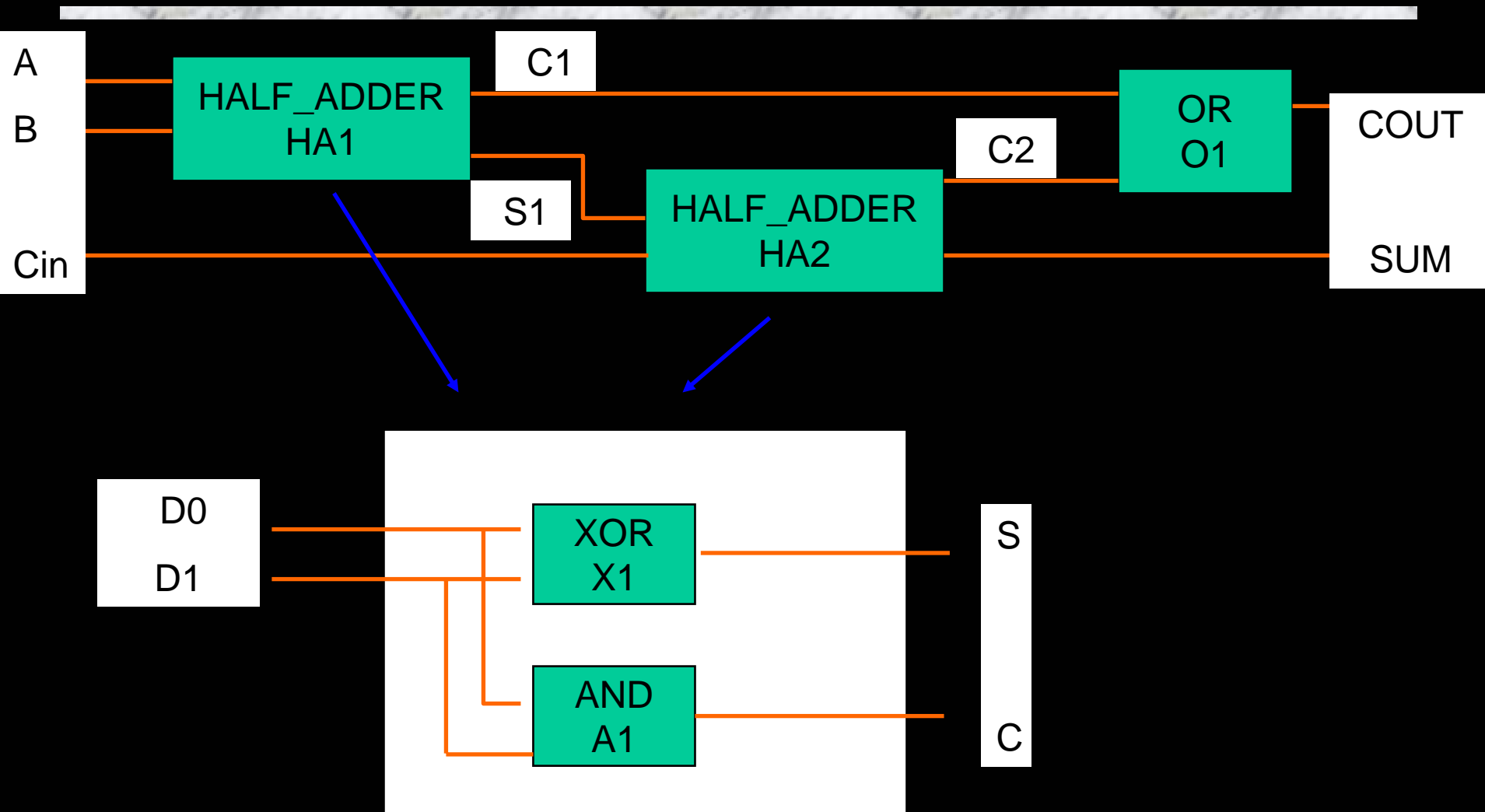
---

- is equivalent to a single WAIT with a sensitivity list at the bottom of the process:

```
process
begin
.....
wait on a, b, c, d;
end process;
```

- Whenever any of the signals in the sensitivity list change value, the process will be executed.
- *Note: Processes with a sensitivity list may not contain any wait statements, nor may they call procedures with wait statements.*

# Configuration & Component Instantiation



# Configuration & Component Instantiation

---

```
entity FULL_ADDER is
```

```
....
```

```
end FULL_ADDER;
```

```
architecture FA_WITH_HA of FULL_ADDER is
```

```
component HALF_ADDER
```

```
port (HA,HB: in BIT; HS,HC: out BIT);
```

```
end component;
```

```
component OR2
```

```
port (A,B: in BIT; Z: out BIT);
```

```
end component;
```

```
signal S1, C1, C2: BIT; (contd. on next slide)
```

# Configuration & Component Instantiation

---

```
begin
```

```
HA1: HALF_ADDER port map (A, B, S1, C1);
```

```
HA2: HALF_ADDER port map (S1, Cin, SUM, C2);
```

```
O1: OR2 port map (C1, C2, COUT);
```

```
end FA_WITH_HA;
```

- - similar declaration for entity HA and architecture HA\_STR
  - - HA\_STR has components XOR2 and AND2
- (contd. also on the next slide : ) )

# Configuration & Component Instantiation

---

```
library ECL;
configuration FA_HA_CON of FULL_ADDER is
  for FA_WITH_HA          - - Top-level configuration
    for HA1,HA2: HALF_ADDER
      use entity WORK.HA(HA_STR)
      port map (D0=> HA, D1, S, C);
    for HA_STR          - - Nested configuration
      for all: XOR2
        use entity WORK.XOR(XOR2);
      end for;
    for A1: AND2
      use configuration ECL.AND2CON;
    end for;
  end for;
end for;
```

# Sequential VHDL Statements



# Sequential Statements

---

These statements can appear inside a process description

- variable assignments
- if-then-else
- case
- loop
  - infinite loop
  - while loop
  - for loop
- assertion and report
- signal assignments
- function and procedure calls

# If statement: Examples

- Example 1:

```
if sel = 0 then
    result <= input_0; -- executed if sel = 0
else
    result <= input_1; -- executed if sel /= 0
end if;
```

- Example 2:

```
if sel = 0 then
    result <= input_0; -- executed if sel = 0
elseif sel = 1 then
    result <= input_1; -- executed if sel = 1
else
    result <= input_2; -- executed if sel /= 0, 1
end if;
```

# Case statement: Example

- Example: Suppose we are modeling an ALU with the following control input func declared as enumeration type.

```
type alu_func is (pass1, pass2, add, sub);
```

The behavior can be described as follows:

```
case func is
```

```
    when pass1 => results := operand1;
```

```
    when pass2 => results := operand2;
```

```
    when add => results := operand1 + operand2;
```

```
    when sub => results := operand1 - operand2;
```

```
end case;
```

# Null statement: Example

- To take care of conditions when no action is needed

```
case func is
```

```
  when pass1 => results := operand1;
```

```
  when pass2 => results := operand2;
```

```
  when add => results := operand1 + operand2;
```

```
  when sub => results := operand1 - operand2;
```

```
  when nop => null;
```

```
end case;
```

# Loop statements: Infinite Loop

---

- **Repeats a sequence of statements indefinitely.**
  - **avoid this situation in any high level programming language.**
  - **In digital systems this is useful as hardware devices repeatedly perform the same operation as long as power supply is on.**
- **Typical structure: Loop statement in process body with a wait statement.**

# Infinite Loop: Example

```
p1: process is
begin
    .....
    L1: loop
        .....
        L2 : loop
            .....
            -- nested loop
            .....
        end loop;
        .....
    end loop;
    .....
    wait;
end process;
```

# While Loop: Example

```
entity cos is
  port (theta: in real; result: out real;);
end entity;
architecture series of cos is
begin
  P1: process (theta) is
    variable sum, term, n: real;
  begin
    sum := 1.0; term := 1.0; n := 0.0;
    while abs term > abs (sum/1.0E6) loop
      n := n + 2.0;
      term := (-term) * (theta ** 2) / ((n-1) * n);
      sum := sum + term;
    end loop;
    result <= sum;
  end process;
end architecture;
```

$$\cos \theta = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \dots$$

# For loop

```
for_loop_stmt <=
  [ loop_label : ]
  for id in discrete_range loop
    { sequential_stmt }
  end loop [ loop_label ] ;
discrete_range <= expr ( to | downto ) expr
```

```
for count in 0 to 127 loop
  count_out <= count;
  wait for 5 ns;
end loop;
```



# For Loop: Rules

---

- Loop parameter's type is the base type of the discrete range.
- Loop parameter is a constant inside the loop body.
- It can be used in an expression but not written to.
- Loop parameter is not required to be explicitly declaration.
- Loop parameter's scope is defined by the loop body.
- Consequently, it hides any variable of the same name inside the loop body.

# For Loop: Example

```
P1: process is
  variable i, j: integer;
begin
  i := loop_param;           -- ERROR
  for loop_param in 1 to 10 loop
    ...
    loop_param := 5;        -- ERROR
    ...
  end loop;
  j := loop_param;          -- ERROR
end process;
```

# A Typical use of CASE: FSM

---

*A part of a process used to model the next state computation for a finite state machine.*

```
case machine_state is
  when sv0 => machine_state <= sv1
  when sv1 => machine_state <= sv2;
  when sv2 => machine_state <= sv3;
  when sv3 => machine_state <= sv0;
end case;
```

# Typical use of CASE: Multiplexer

---

*Model the output value generation for a finite state machine.*

```
case Current_state is
  when sv0 | sv1 | sv2 =>   Z_out <= '0';
  when sv3      =>         Z_out <= '1';
end case;
```

# While Loops

---

- *Form:*

```
while condition loop
    sequential statements
end loop;
```

- *Example:*

```
while bus_req = '1' loop
    wait until ad_valid_a = '1';
    bus_data <= data_src;
    msyn <= '1' after 50 ns;
    wait until ssyn = '1';
    msyn <= '0';
end loop;
```

# Next

---

- branches back to the beginning of the loop (like a Fortran CONTINUE statement).

```
loop
    sequential statements
next when condition
    sequential statements
end loop;
```

# Exit

---

- branches completely out of the loop to the first statement following the end loop;

```
loop
    sequential statements
    exit when condition
    sequential statements
end loop;
```

- Example: where x\_in is an array of inputs

```
for i := 2 to x_in'length loop
    new_val := new_val and x_in(i) ;
    exit when new_val = '0';
end loop;
```

# Concurrent VHDL



# Need for Concurrent VHDL

---

- **Intuitively closer to actual hardware than procedural descriptions**
- **More compact representation than procedural descriptions**
- **Provides natural way to represent the natural concurrency arising in hardware**

# Concurrent statements

---

- Signal assignment statements
  - (unconditional)
  - Conditional (when-else)
  - Selected (with-select)
- Process (interface to procedural descriptions)
- Component instantiation (interface to structural descriptions)
- Block statement

# Concurrent VHDL Statements

---

- Execution whenever an input (RHS) changes value.
- Execution order totally independent of order of appearance in source code.

*Example: Exchange of signal values*

```
X <= Y;
```

```
Y <= X;
```

# Simulation Cycle Revisited

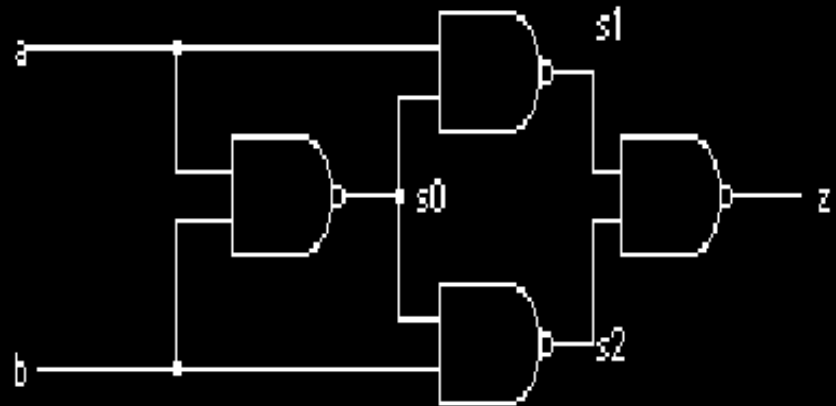
## Sequential vs Concurrent Statements

---

- **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

# Simple Example

```
entity XOR
  port (a,b:in bit;
        z : out bit );
end;
architecture nand_gates of
  XOR is
    signal s0, s1, s2:
      bit;
  begin
    s0 <= a nand b;
    s1 <= a nand s0;
    s2 <= b nand s0;
    z <= s1 nand s2;
  end nand_gates;
```



# Concurrent statements - more

---

*In the example*

```
begin
    s0 <= a nand b;
    s1 <= a nand s0;
    s2 <= b nand s0;
    z <= s1 nand s2;
end nand_gates;
```

*placing the first statement (s0 <= ...) after the last statement has absolutely no effect on execution or the result. The s1 <= ... and s2 <= ... statements would have used the old s0 value anyhow.*

# Conditional signal assignment (When-else statements)

---

- When-else statements imply priority encoding.

```
S <=
```

```
W0 after delay0 when c0 else
```

```
W1 after delay1 when c1 else
```

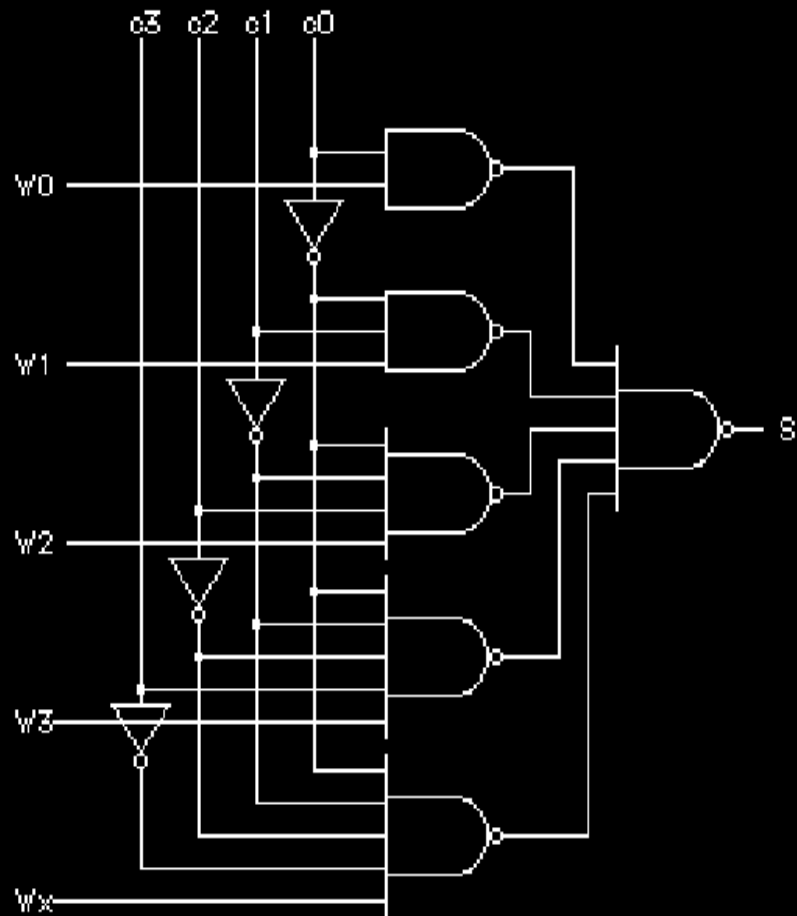
```
W2 after delay2 when c2 else
```

```
W3 after delay3 when c3 else
```

```
Wx after delayx ;
```

Note: Priority encoding is implied since more than one condition might be true at the same time. The condition appearing first in the statement has the priority.

# Circuit implementation





# Equivalent process

---

```
PROCESS (W0, W1, . . . , WX, C0, C1, . . . );  
BEGIN  
    IF c0 THEN  
        S <= W0 after delay0  
    ELSIF c1 THEN  
        S <= W1 after delay1  
    ELSIF c2 THEN  
        S <= W2 after delay2  
    ELSE S <= Wx after delayx ;  
END PROCESS;
```

# Simple when - else

```
S <= Wa after Ta when c0
else
    Wb after Tb;
```

- No ELSE implies Memory

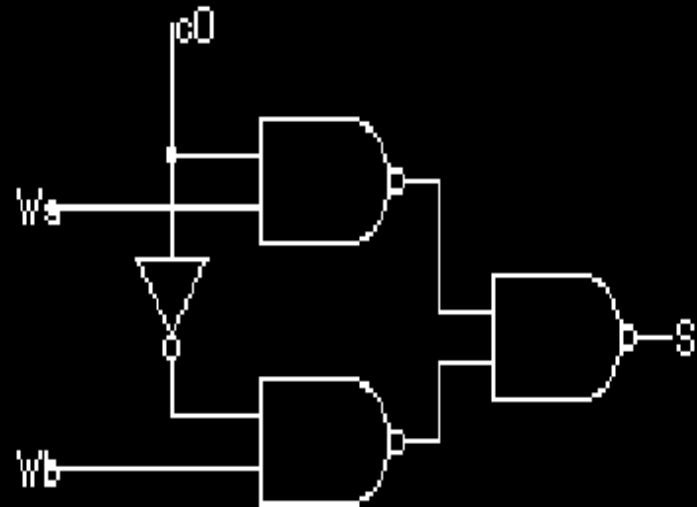
```
S <= X after t WHEN C;
```

- UNAFFECTED has the same effect as a null clause

```
S <= X after t WHEN C
```

```
ELSE
```

```
UNAFFECTED;
```



# No ELSE implies Memory

---

- Memory is implied where no else clause is provided.
- In the simulation model, no event is implied when the condition is false, so the signal retains its old value. I.e., it has memory.
- Examples:
  - Q <= D when rising\_edge (clock);
  - B <= A when phase\_a = '1' ;

# Simple enabling

---

-- for simple AND enabling

B <= A when en else '0';

B <= A when en else (others =>'0');

-- for tri-state driver output

B <= A when en else 'Z';

B <= A when en else (others => 'Z');

# Selected Signal Assignment

---

- Form

WITH *selector* SELECT

*signame* <= *W0* after *delay0* when *c0*,  
*W1* after *delay1* when *c1*,  
*W2* after *delay2* when *c2*,  
*Wx* after *delayx* when OTHERS;

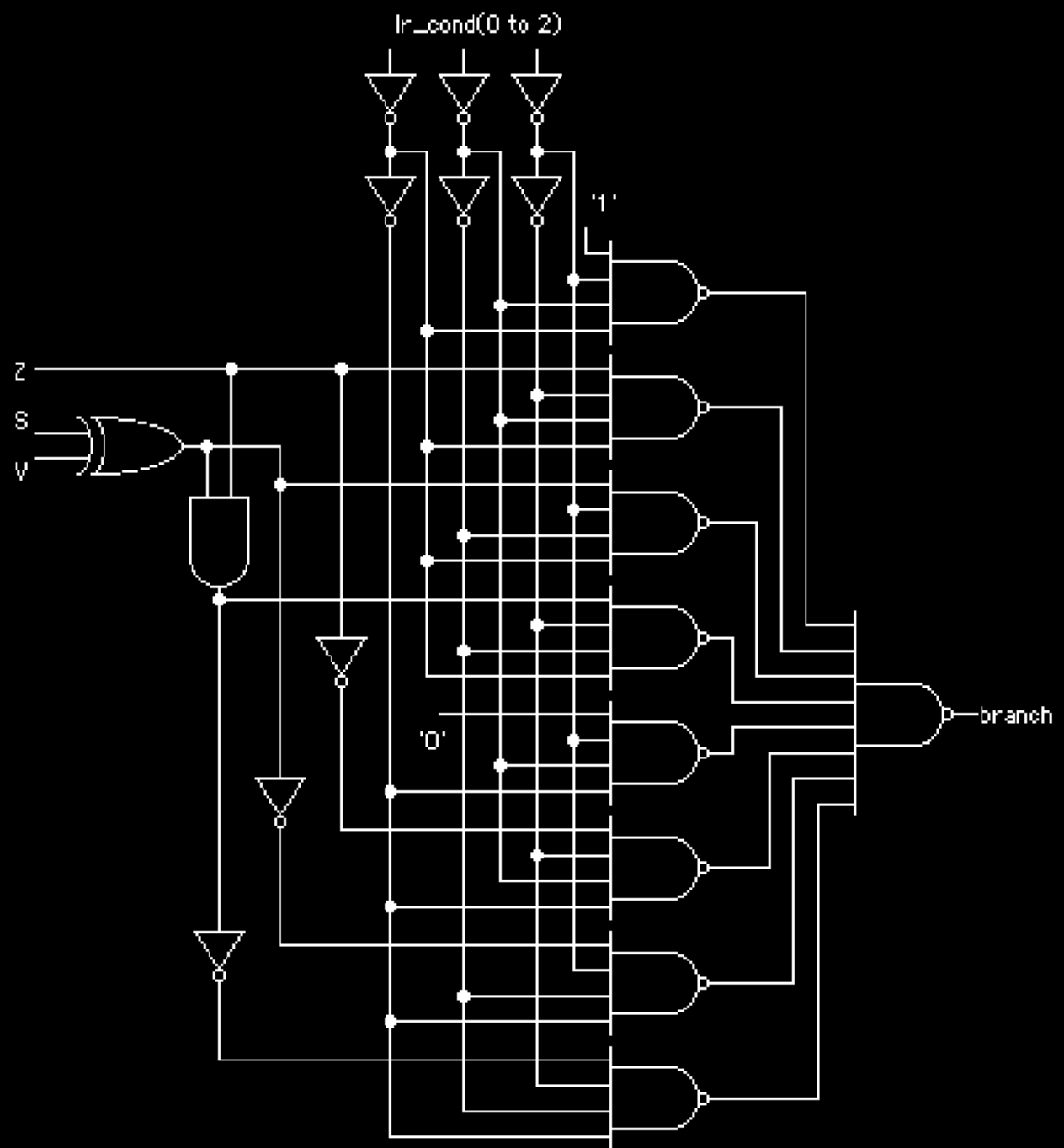
# Example: Branch Condition Selector

---

- Typically the *cond* field of the instruction specifies which of several logical expressions of status flipflops is to be used.

With IR.cond select

branch <=	'1'	when uncond,
	Z	when zero,
	S xor V	when less,
	(S xor V) and Z	when lteq,
	'0'	when never ,
	not Z	when nzero ,
	not (S xor V)	when gteq,
	not (S xor V) and Z	when grtr;



# Selected Signal Assignment

---

- All values must be included
  - the range of values for the selector should be restricted with some care.
- Otherwise, a declaration like “signal csel : integer;”  
calls for a  $2^{32}-1$  input multiplexer which we would not want to build! Even where address decoding is required the range should be 0 to 255 or so, the address partitioned to correspond to the decoding method being used.
- Since no more than one value can be selected at a time, no priority encoding is involved.



# OTHERS Clause

---

- handle 'X' -- synthesis standard
- unused inputs
- optimization possibilities
  
- Forces explicit consideration of what is to be done if unexpected input values occur.

# Example: An ALU

---

```
LIBRARY ieee; USE ieee.std_logic_1164.all;
USE work.ALU_funcs.add_w_carry;
ENTITY ALU is
  -- IO ports
  port (dout: out std_logic_vector;      -- Data Out(latched)
        a, b: in std_logic_vector;      -- A and B leg inputs
        cin: in std_logic;              -- Carry in, 1 bit
        cout: out std_logic;            -- Carry Out, 1 bit,unlatched
        func_sel: in std_logic_vector (0 to 1);
                                         -- Function select (2 bits) 00 => ADD
                                         -- 10 => AND   01=> OR 11=> XOR
        clk: in std_logic );-- Result register clock
end ALU;
```

# Example continued

---

ARCHITECTURE concurrent of ALU is

```
constant XOUT: std_logic_vector(a'range)
              := (others=>'X')
```

```
--ALU output: carry concatenated to left end makes an extra bit
signal ALUout: std_logic_vector(xt_reg'range);
```

```
-- Opcode interpretation
```

```
constant OPADD: std_logic_vector := "00";
constant OPOR:  std_logic_vector := "01";
constant OPAND: std_logic_vector := "10";
constant OPXOR: std_logic_vector := "11";
```

```
BEGIN
```

# Example continued

---

```
BEGIN
```

```
  with func_sel select
```

```
    (cout, ALUout) <= '0' & (a OR b)           when OPOR,  
                      '0' & (a AND b)         when OPAND,  
                      '0' & (a XOR b)         when OPXOR,  
                      add_w_carry(a, b, cin)  when OPADD,  
                      XOUT                    when OTHERS;
```

```
  -- where add_w_carry returns a value with its width  
  --       one greater than the size of its inputs,
```

```
  dout <= ALUout when rising_edge(clk);
```

```
END concurrent;
```

# Note use of OTHERS clause

---

- Only intended values are “00”, “01”, “10”, “11”.
- However!!!! The inputs are type STD\_LOGIC\_VECTOR (0 to 1). Other legal values that must be accounted for include: 0X, X0, X1, 1X, XX, --
- In fact, since STD\_LOGIC is a 9-valued system, there are 81-4 “other” selector values that must be handled.

# Procedures

---

- Procedure: Declared and then called
- Example:

```
procedure average_samples is
```

```
    variable total: real := 0.0;
```

```
    -----
```

```
    -----
```

```
end procedure average_samples;
```

- This can be called inside a process as:  
average\_samples;

# Functions

---

- Syntax is very similar to that of the procedures.
- Unlike procedure, function calculates and returns a result that can be used in an expression.
- Parameters of the function must be of 'in' mode and may not be of class variable.

- Example:

```
function bv_add (bv1, bv2 : in bit_vector) return bit_vector is
begin
    -----
end function bv_add;
```

```
signal source1, source2, sum: bit_vector (0 to 31);
adder: sum <= bv_add(source1, source2) after T_delay_adder;
```

# Assertion & Report statements

---

- A functionally correct model may need to satisfy certain conditions.
- Some of these can be specified by “assert” statements.
- Report Statement are useful for providing extra information from specific assertion statements (as there can be several assertion statements).
- Assert and report statements are particularly useful for de-bugging.



# Assertion & Report: Example

```
assert value <= max_value;
```

```
assert value <= max_value  
    report "Value too large";
```

```
type severity_level is (note,warning,error,failure);
```

```
assert clock_width >= 100 ns  
    report "clock width too small"  
    severity failure;
```

# Block Statement

---

- Three major purposes:
  - **Disable signal drivers by using guards**
  - **Limit scope of declarations (including signals)**
  - **Represent a portion of design**

```
B1: block (STROBE='1')  
begin  
    Z <= guarded (not A);  
end block B1;
```

# Signal Assignment, Delay and Attributes

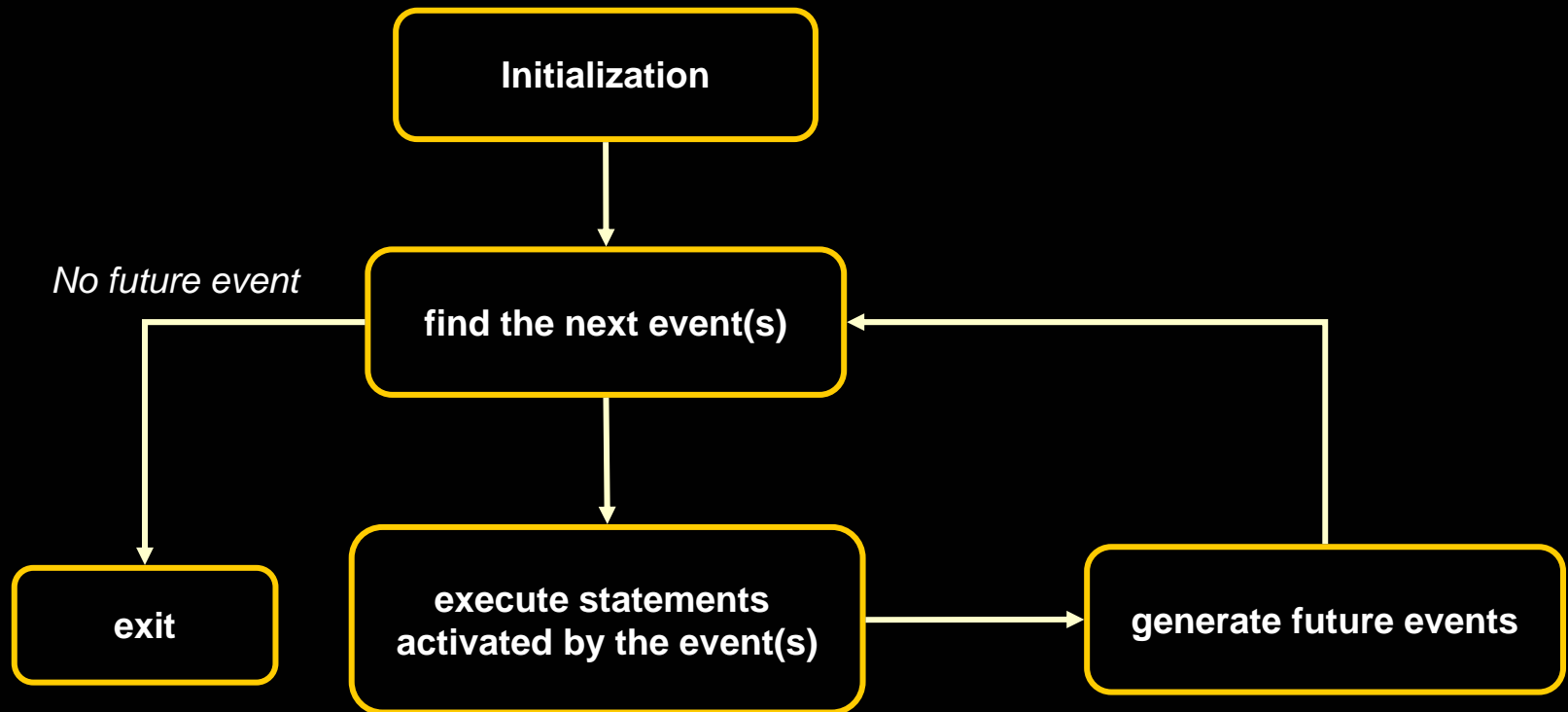
# Event-driven Simulation

---

- **Event: occurrence of an signal change (transaction)**
- **When an event occurs, need to handle it (execution)**
  - to execute any concurrent statements that are sensitive to the change
  - to activate a waiting process if the condition becomes true
- **The execution triggered by an event may generate future events**
  - `sig_a <= a_in after 5 ns;`

# Event-driven Simulation

---



# Event-driven Simulation

---

- **Events must be handled in sequence**
  - *current\_time* advances when the execution all current events is done and the next event is triggered
  - *event queue* – an internal data structure to manage the event sequence
- **System state**
  - represented by the states of all objects
  - thus, need a data structure to indicate that
    - the current state (value) of “*sig\_a*” is “0” and
    - it will be changed to “1” at *current\_time+5ns*

# Waveforms in Signal Assignment

---

- **Form**

Sig\_identifier <= val\_expr1 after delay1  
{ , val\_expr\_ after delay\_ } ;

- **Delay values must appear in ascending order**
- ***List of updating events (i.e., value / time) is called a *Driver* of the signal.***

# Simulation of VHDL

---

- **All components work concurrently**
- **If an event triggers 2 statements and the result of the 1<sup>st</sup> one is also the input of the 2<sup>nd</sup> one**
- **Sequential statements allow immediate updates**
  - for functional behavior
  - no notion of time
- **Concurrent statement – execute and update stages**
  - for circuit operation
  - only signal assignment and process statements



# Simulation of signal assignment statements

---

`sig_a <= a_input;` -- statement (1)

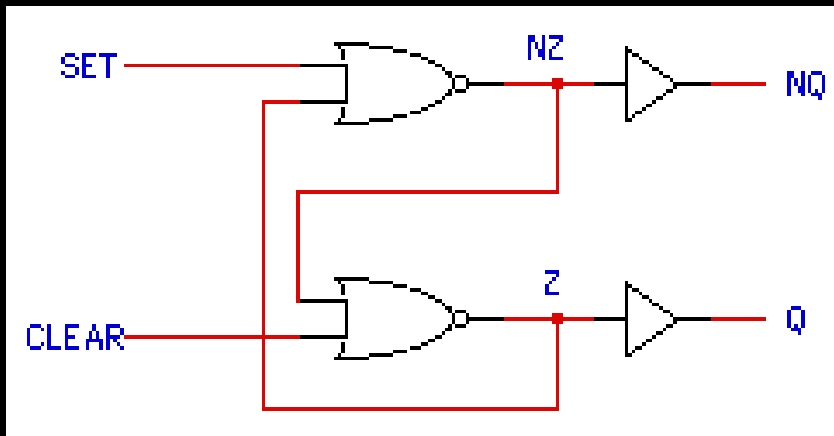
`add_a <= sig_a xor c_in xor a_input;` -- statement (2)

If *a\_input* changes at *current\_time* →

- execution stage -- *sig\_a* and *add\_a* will be computed in the statements (1) and (2)
- update stage -- the new values will be written to the objects *sig\_a* and *add\_a*, and new events can be generated due to the changes
- no change in *current\_time*
- *It seems there is a delay between execution and update (delta delay)*



# Concurrent VHDL Assignments



NQ <= NZ after tprop;  
NZ <= Z nor SET after tprop;  
Q <= Z after tprop;  
Z <= NZ nor CLEAR after  
tprop;

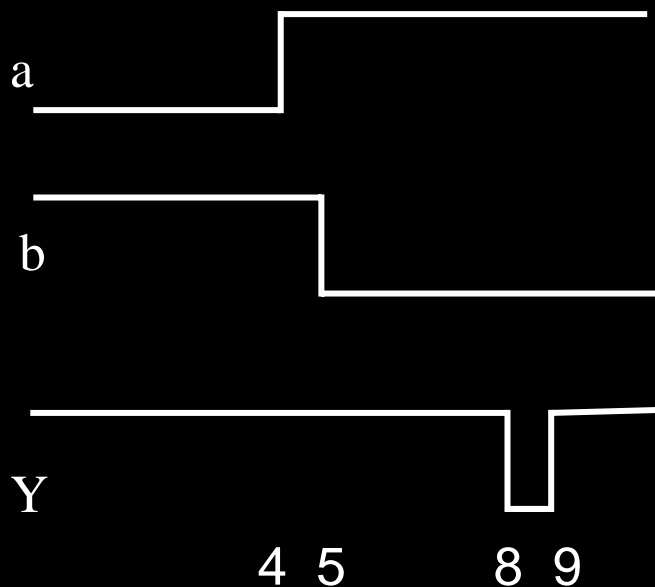
- ***Order of listing these implies nothing about execution order!***

# Inertial Delay



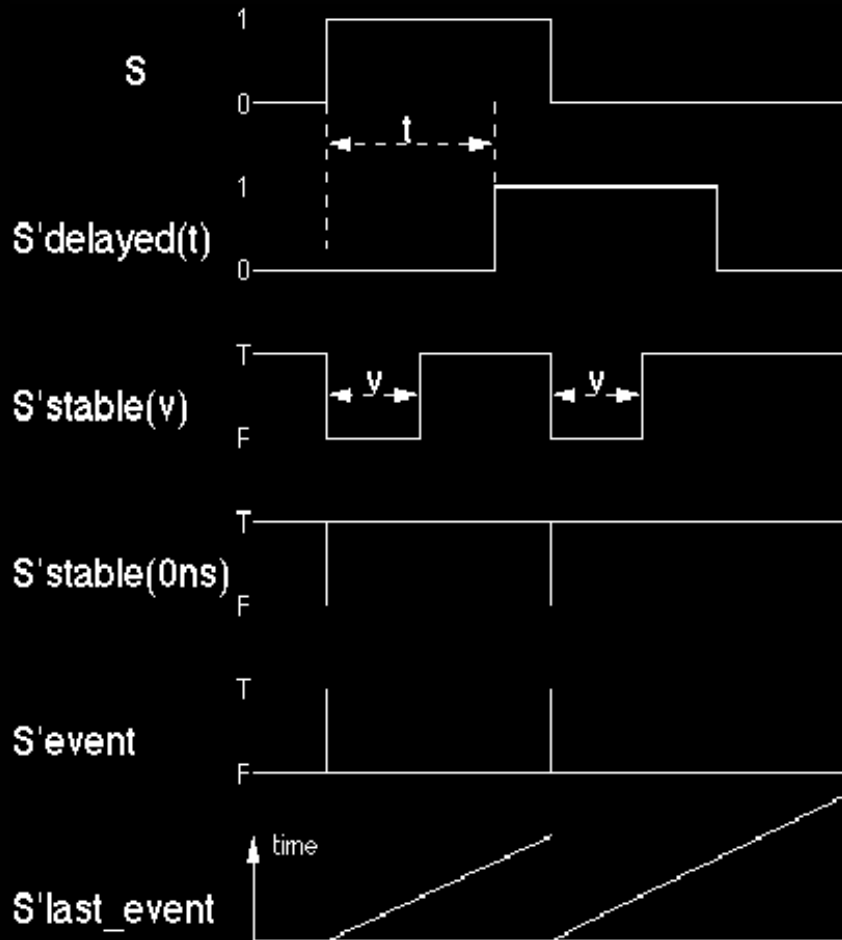
- Inputs:
  - a  $\leq$  '0', '1' after 4 ns;
  - b  $\leq$  '1', '0' after 5 ns;
  - y  $\leq$  a xor b after 4 ns;
- Output Y is initially '1' and will be scheduled:
  - '0' after 8 ns
  - '1' after 9 ns
    - 1st event is removed when 2nd is created and before it is done.

# Transport Delay



- Inputs: -- again
  - a <= '0', '1' after 4 ns;
  - b <= '1', '0' after 5 ns;
  - y <= a xor b after 4 ns;
- Output Y is initially '1' and will be scheduled:
  - '0' after 8 ns
  - '1' after 9ns
- 1st event is NOT removed when 2nd is created.

# Signal Attributes



- $S'$ delayed( $t$ )
  - a waveform all of its own, delayed by  $t$
- $S'$ stable( $t$ )
  - a waveform all of its own, type boolean
- $S'$ event
  - a function true only when  $S$  changes value
- $S'$ last\_event
  - time since last value change

# Attributes

- **Attributes provide information about certain items in VHDL**

- E.g. types, subtypes, procedures, functions, signals, variables
- General form of attribute use :

```
name'attribute_identifier -- read as "tick"
```

- **VHDL has several predefined, e.g :**

- X'EVENT -- TRUE when there is an event on signal X
- X'LAST\_VALUE -- returns the previous value of signal X
- Y'HIGH -- returns the highest value in the range of Y

# Operators

- chained to form complex expressions

```
res <= a AND NOT(B) OR NOT(a) AND b;
```

- Defined precedence levels in decreasing order :

- Miscellaneous operators -- \*\*, abs, not
- Multiplication operators -- \*, /, mod, rem
- Sign operator -- +, -
- Addition operators -- +, -, &
- Shift operators -- sll, srl, sla, sra, rol, ror
- Relational operators -- =, /=, <, <=, >, >=
- Logical operators -- AND, OR, NAND, NOR, XOR, XNOR



# **Array, Records, and Aggregated Constants**

# Composite structures

---

- **Arrays**
  - group elements of same types
- **Records**
  - group elements of different types
- **Access**
  - like pointers in “C”, may be useful in file I/O and creation of test environments.

# Declaring Arrays

---

- **FORM**

TYPE array\_name IS ARRAY (discrete\_range {,...})  
OF element\_subtype\_indication;

- **Examples**

TYPE carrier IS ARRAY (15 downto 0) of bit;

TYPE regs IS ARRAY ( 0 to 31) of byte;

(where we previously defined type byte by  
type byte is 0 to 255;)

# USE of Discrete Range

---

- **Discrete range is**
  - index\_value TO index\_value
  - index\_value DOWNTO index\_value
  - using a previously defined type
    - ➔ type\_name RANGE left\_value TO right\_value OR
    - ➔ type\_name RANGE right\_value DOWNTO left\_value

# Objects or Types

---

- **Can declare objects to be arrays directly**

SIGNAL AX : ARRAY ( 31 DOWNT0 0 ) of BIT;

- **HOWEVER! Almost always MULTIPLE arrays of the same dimensions and with the same element types are needed, so usually**

- declare array type

- declare objects

TYPE word IS ARRAY ( 31 DOWNT0 0 ) of BIT;

SIGNAL AX, BX, CX, DX : word;

VARIABLE temp : word;

# Array Attributes

---

- **left** -- left index defined in range
- **right** -- right index defined in range
- **low** -- smallest index value defined in range
- **high** -- largest index value defined in range
- **range** -- left index, direction, right index
- **reverserange** -- right index, opposite direction, left index
- **length** -- number of elements in array

# Reference to Elements of an Array

---

- Use parenthesis, not brackets for index

- Example:

Put a new example here

- Example

```
FOR j in AX'range LOOP
```

```
    IF j <> AX'right THEN AX ( j) <= AX(j-1);
```

```
    ELSE AX(j) <= new_right_bit;
```

```
END LOOP;
```

# Unconstrained Arrays

---

- **In a package,**
  - declare element types
  - declare array types
- **At the time the array type is declared, the actual size of the array objects is unknown**
- **Necessary because OPERATIONS on the array need to be written generally**
  - operate the objects regardless of the size they may have.



# How to Declare an Unconstrained Array

---

## FORM

```
TYPE arrayname IS ARRAY ( indextype RANGE <> ) OF element_type;
```

## Examples:

```
TYPE bit_vector IS ARRAY ( NATURAL RANGE <> ) OF BIT;
```

```
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <> ) OF  
STD_LOGIC;
```

# Bit\_Vectors and Strings

---

- Predefined for use in STD.STANDARD
- Both are unconstrained array declarations

Examples of use:

```
constant Error_Msg : string := "System is Unstable"; -- string is
-- an array of characters
```

```
constant zeroes : bit_vector := B"0000_0000";
```

```
constant empty: bit_vector := O"052";
```

```
constant restart: bit_vector := X"0FC";
```

- B, O, X indicate binary, octal, hex representation of bits

# Std\_Logic\_Vectors

---

- **Unconstrained array defined for us in**

*Library IEEE;*

*USE Package IEEE.STD\_LOGIC\_1164.all;*

Example of use:

```
entity alu ( left, right : in std_logic_vector;  
            result : out std_logic_vector;  
            cy_in : in std_logic;  
            cy_out : out std_logic;  
            control : in std_logic_vector (0 to 3) );
```

# Records

---

- **Declaration**

```
TYPE record_type IS
```

```
    RECORD
```

```
        field1 : type;
```

```
        field2 : type;
```

```
    END RECORD
```

- **Should be used to group elements of different types which logically belong together.**

# Records

---

- Definition --

```
TYPE time_stamp IS RECORD
```

```
    second : integer range 0 to 59;
```

```
    minute : integer range 0 to 59;
```

```
    hour : integer range 0 to 23;
```

```
END RECORD;
```

- VARIABLE current\_time : time\_stamp;

```
    current_time.second := 0;
```

- no variant field allowed in VHDL

# Example of Record

---

- Type declarations

TYPE Operations is (move, add, sub, adc, sbb, bra, call, inc, dec, push, pop, shf, rot flag);

TYPE Address\_mode is (reg, direct, indirect, displ, indexed);

TYPE Xreg is (ax, bx, cx, dx, sp, bp, si, di);

- A type describing the structure of instructions:

TYPE Instruction is RECORD

opcode : Operations;

src\_mode: Address\_mode;

src\_reg, dst\_reg: Xreg;

dst\_mode: Address\_mode;

END RECORD;

# References to records

---

- **Just as in other languages, name the object followed by period and the field**

FORM: objectname.fieldname

- **Example**

```
SIGNAL IR : Instruction;
```

```
...
```

```
case IR.opcode IS -- references opcode field
```

```
  where mov => ...
```

```
  where add => ...
```

# Aggregated constants

---

- The way to define constant values for composite data objects like arrays and records.

- **FORM**

( index\_or\_field\_name(s) => value,  
repeated as many times as needed. . . );

index\_or\_field\_name(s) can be a range of values for array indexing, several enumerated values separated by |, or the field names for records.



# Examples of Aggregated Constants

---

Assume

```
type clock_level is (low, rising, high, falling);
```

```
type ctable is array (clock_level) of bit;
```

```
constant conversion_table : ctable := ('0','1','1','0');
```

```
constant conversion_table : ctable := ("0110");
```

```
constant conversion_table : ctable :=  
    (low=>'0',rising=>'1',high=>'1',falling=>'0');
```

```
constant conversion_table : ctable := (low|falling => '0',  
    high|rising => '1');
```

All have the same meaning

# Referencing into tables

---

Suppose we declare

```
Variable clk_tran : clock_level;
```

```
Signal bit_action : bit;
```

Then we can write the following statement:

```
bit_action <= ctable ( clk_tran );
```

Note that the index values are NOT integers, but that's OK!

# Aggregated Constants

---

- **Strings and derived types on character can be placed in “ ... ”**

- **Examples**

**Constant X : bit\_vector := “0010”;**

**Constant Y : std\_logic\_vector := “001X”;**

**Constant Z : string := “ABCD”;**

# More examples of aggregated constants

---

- Record example

Constant CLEARAX : Instruction :=

```
(opcode => xor,  
src_reg => ax,  
dst_reg => ax,  
src_mode=> reg,  
dst_mode => reg);
```

- Array example

Constant ZZs : cpureg := (cpureg'range => 'Z');

# Multidimensional Arrays

---

**TYPE array\_name IS ARRAY(index\_type1,  
index\_type2, . . .) OF element\_type;**

- **Most useful in creating lookup tables for functions**

# Creating tables for lookup functions - example

```
TYPE stdlogic_table IS ARRAY(std_ulogic,std_ulogic) OF std_ulogic;
-- truth table for "and" function
CONSTANT and_table : stdlogic_table := (
-- -----
-- |  U    X    0    1    Z    W    L    H    -    |  |
-- -----
  ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
  ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
  ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )); -- | - |
```

# Another table lookup example

---

- Suppose inputs P and Q are `std_ulogic`, then the quickest way to get the new value for AND is a table lookup, handled by the AND operation in the Package `STD_LOGIC_1164` Body.

Result := and\_table (P,Q);

- Note: the values of P and Q are enumerated, not indexed.

# Use of OTHERS keyword

---

- Filling in all the default values in Arrays
- Extremely useful where unconstrained arrays need to be initialized.
- FORM:
  - use keyword OTHERS as the index\_name
  - Constant S\_ONE : std\_logic\_vector := (S\_ONE'right => '1', OTHERS => '0');
  - Constant S\_ZZZ : std\_logic\_vector := (OTHERS => 'Z');



# Design Processing

---

- Analysis
- Elaboration
- Simulation
- Synthesis

# Design Processing: Analysis

---

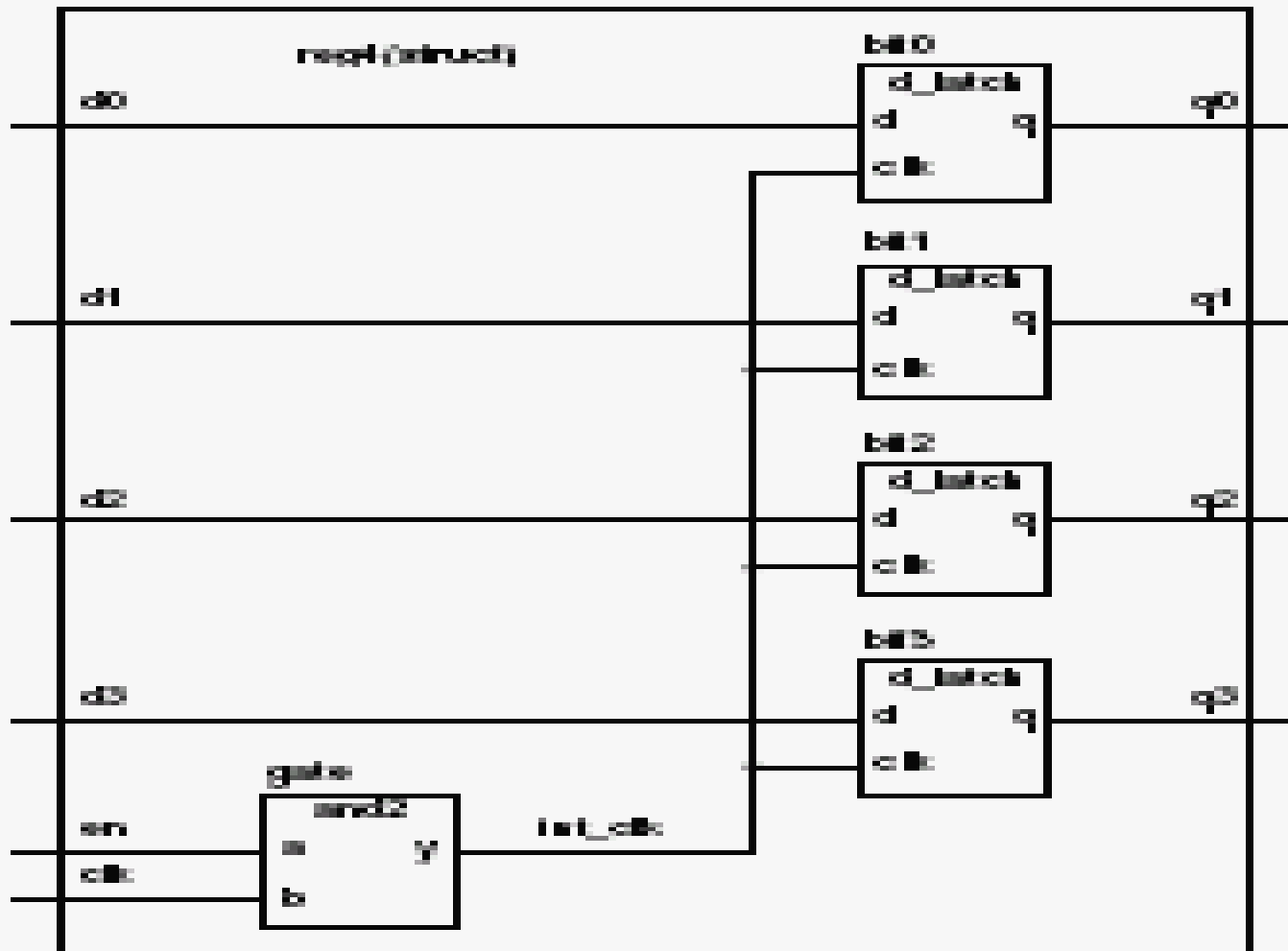
- Check for syntax and semantic errors
- Analyze each *design unit* separately
  - entity declaration
  - architecture body
  - ...
  - best if each design unit is in a separate file
- Analyzed design units are placed in a *library*
  - in an implementation dependent internal form
  - current library is called work

# Design Processing: Elaboration

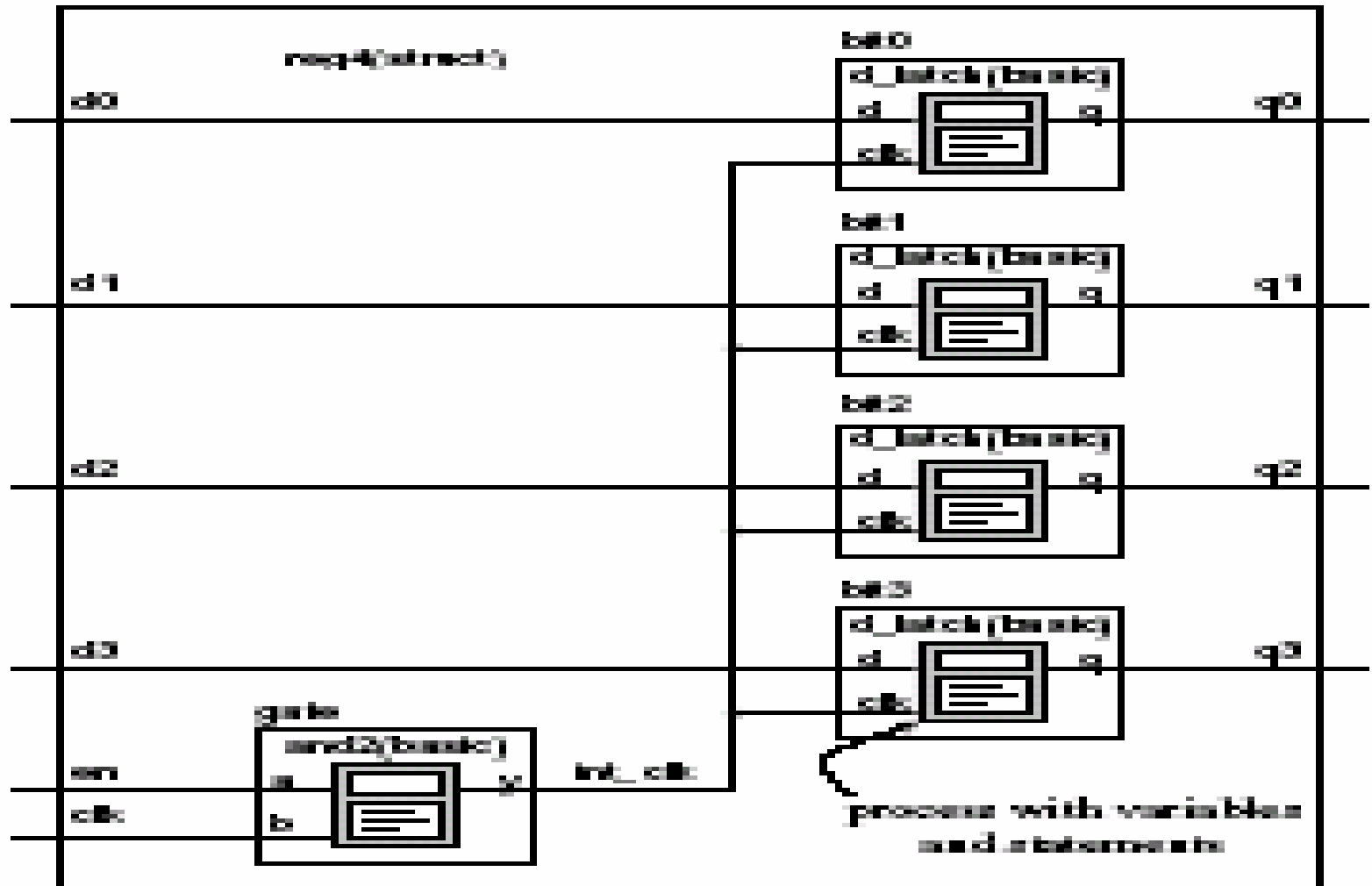
---

- “Flattening” the design hierarchy
  - create ports
  - create signals and processes within architecture body
  - for each component instance, copy instantiated entity and architecture body
  - repeat recursively
- Final result of elaboration
  - flat collection of signal nets and processes

# Design Processing: Elaboration Example



# Design Processing: Elaboration Example



# Design Processing: Simulation

---

- Execution of the processes in the elaborated model
  - Discrete event simulation
    - time advances in discrete steps
    - when signal values change—*events*
  - A processes is sensitive to events on input signals
    - specified in wait statements
    - resumes and schedules new values on output signals
      - schedules *transactions*
      - event on a signal if new value different from old value

# Design Processing: Simulation Algorithm

---

- Initialization phase
  - each signal is given its initial value
  - simulation time set to 0
  - for each process
    - activate
    - execute until a wait statement, then suspend
  - execution usually involves scheduling transactions on signals for later times

# Design Processing: Simulation Algorithm

---

- Simulation cycle
  - advance simulation time to time of next transaction
  - for each transaction at this time
    - update signal value
      - event if new value is different from old value
  - for each process sensitive to any of these events, or whose “wait for ...” time-out has expired
    - resume
    - execute until a wait statement, then suspend
- Simulation finishes when there are no further scheduled transactions



# Design Processing: Synthesis

---

- Translates register-transfer-level (RTL) design into gate-level netlist
  - Restrictions on coding style for RTL model
  - Tool dependent: A subset of RTL is synthesizable depending on the tool

---

# Questions?