

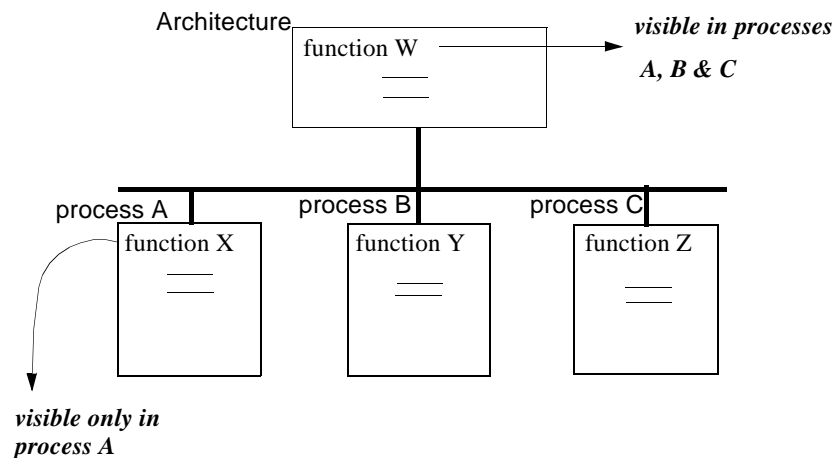
Essentials of Functions

```
function rising_edge (signal clock: std_logic) return boolean is  
--  
--declarative region: declare variables local to the function  
--  
begin  
-- body  
--  
return (expression)  
end rising_edge;
```

- Formal parameters and mode
- Functions cannot modify parameters
- Types of formals and actuals must match except for formals which are constants (default)
- Wait statements are not permitted in a function!

1

Placement of Functions



- When used by multiple design entities place functions in packages

2

Function: Example

```
architecture behavioral of dff is  
function rising_edge (signal clock : std_logic) return boolean is  
variable edge : boolean:= FALSE;  
begin  
edge := (clock = '1' and clock'event);  
return (edge);  
end rising_edge;  
  
begin  
output: process  
begin  
wait until (rising_edge(Clk));  
  
    Q <= D after 5 ns;  
    Qbar <= not D after 5 ns;  
  
end process output;  
end architecture behavioral;
```

Architecture
Declarative
Region

3

Function: Example

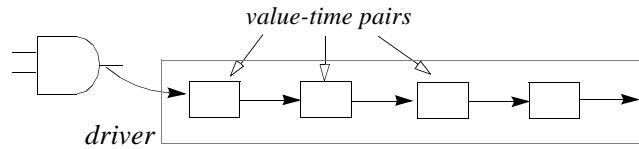
```
function to_bitvector (svalue : std_logic_vector) return bit_vector is  
variable outvalue : bit_vector (svalue'length-1 downto 0);  
begin  
for i in svalue'range loop -- scan all elements of the array  
  case svalue (i) is  
    when '0' => outvalue (i) := '0';  
    when '1' => outvalue (i) := '1';  
    when others => outvalue (i) := '0';  
  end case;  
end loop;  
return outvalue;  
end to_bitvector
```

- Type conversion functions
 - interoperability
- Use of attributes for flexible function definitions
- Browse the vendor supplied packages for many examples

4

Implementation of Signals

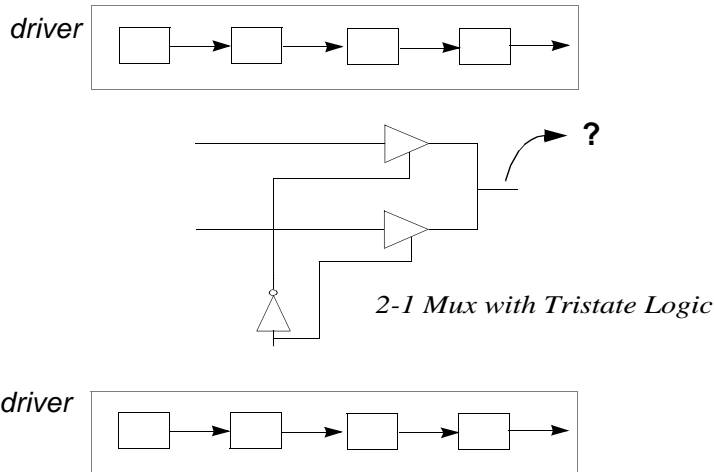
- The structure of a signal assignment statement
 - $signal \leq (value\ expression\ after\ time\ expression)$
 - RHS is referred to as a *waveform element*
- Every signal has associated with it a *driver*



- holds the current and future values of the signal - a *projected waveform*
- signal assignment statements modify the driver of a signal
- value of a signal is the value at the head of the driver
- note the hardware analogy

5

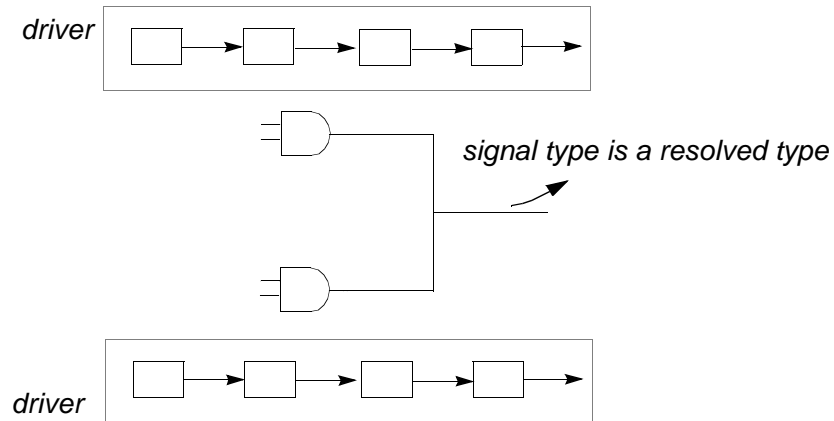
Shared Signals



- How do we model the state of a wire?
- Rules for determining the signal value is captured in the resolution function

6

Resolved Signals



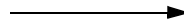
- Resolution function is invoked whenever an event occurs on this signal
- Resolution must be an associative operation

7

Resolved Types: std_logic

```
type std_logic is (  
  'U', -- Uninitialized  
  'X', -- Forcing Unknown  
  '0', -- Forcing 0  
  '1', -- Forcing 1  
  'Z', -- High Impedance  
  'W', -- Weak Unknown  
  'L', -- Weak 0  
  'H', -- Weak 1  
  '-' -- Don't care  
);
```

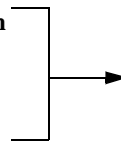
supports only single drivers



```
function resolved (s : std_logic_vector) return  
std_logic;
```

```
subtype std_logic is resolved std_logic;
```

supports multiple
drivers



8

Resolution Function: std_logic & resolved()

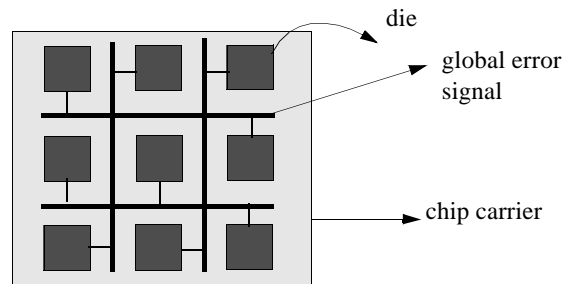
- resolving values for std_logic types

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

- Pairwise resolution of signal values from multiple drivers
- Resolution operation must be associative

9

Example



- Multiple components driving a shared error signal
- Signal value is the logical OR of the driver values

10

A Complete Example

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

entity mcm is
end entity mcm;

architecture behavioral of mcm is
function wire_or (sbus :std_ulogic_vector)
return std_ulogic;
begin
for i in sbus'range loop
if sbus(i) = '1' then
return '1';
end if;
end loop;
return '0';
end wire_or;

subtype wire_or_logic is wire_or
std_ulogic;
signal error_bus : wire_or_logic;
begin
Chip1: process
begin
--..
error_bus <= '1' after 2 ns;
--..
end process Chip1;
Chip2: process
begin
--..
error_bus <= '0' after 2 ns;
--..
end process Chip2;
end architecture behavioral;
```

- Use of unconstrained arrays and associative operations

11

Summary: Essentials of Functions

- Placement of functions
- Formal parameters
- Check the source listings of packages for examples of many different functions
- For example in the Aldec implementation look in
C:\Program Files\Aldec\Active-HDL 3.5 SE\Vlib
for a list of libraries
 - each library has a /src subdirectory

12

Essentials of Procedures

```
procedure read_v1d (variable f: in text; v :out std_logic_vector)
--declarative region: declare variables local to the procedure
--
begin
-- body
--
end read_v1d;
```

- parameters may be of mode **in** (read only) and **out** (write only)
- default class of input parameters is constant
- default class of output parameters is variable
- variables declared within procedure are initialized on each call

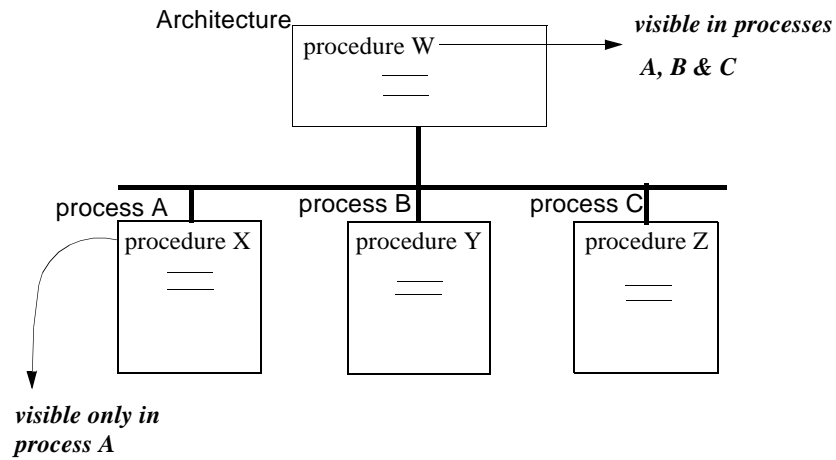
13

Procedures: Placement

```
architecture behavioral of cpu is
--
-- decalarative region
-- procedures can be placed in their entirety here → visible to all
-- processes
--
begin
  process_a: process
  -- declarative region of a process → visible only within
  -- procedures can be placed here process_a
  begin
  --
  -- process body
  --
  end process_a;
  process_b: process
  --declarative regions
  begin
  -- process body
  end process_b;
end behavioral;
```

14

Placement of Procedures



- placement of procedures determines visibility in its usage

15

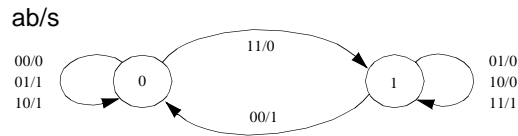
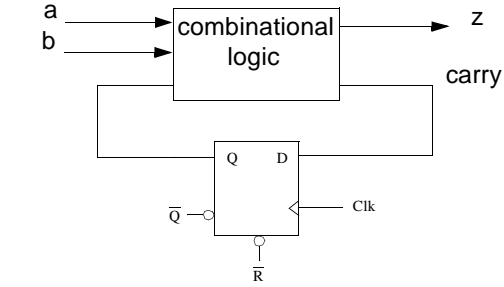
Procedures and Signals

```
procedure mread (address : in std_logic_vector (2 downto 0));  
signal R : out std_logic;  
signal S : in std_logic;  
signal ADDR : out std_logic_vector (2 downto 0);  
signal data : out std_logic_vector (31 downto 0)) is  
begin  
  ADDR <= address;  
  R <= '1';  
  wait until S = '1';  
  data <= DO;  
  R <= '0';  
end mread;
```

- procedures can make assignments to signals passed as input parameters
- procedures may not have a wait statement if the encompassing process has a sensitivity list
- procedures may modify signals not in the parameter list, e.g., ports

16

Concurrent vs. Sequential Procedure Calls



- Example: bit serial adder

17

Concurrent Procedure Calls

architecture structural of serial_adder is

component comb

port (a, b, c_in : in std_logic;
z, carry : out std_logic);

end component;

procedure dff(signal d, clk, reset : in std_logic;
signal q, qbar : out std_logic) is

begin

if (reset = '0') **then**

q <= '0' after 5 ns;

qbar <= '1' after 5 ns;

elsif (clk'event and clk = '1') **then**

q <= d after 5 ns;

qbar <= (not D) after 5 ns;

end if;

end dff;

signal s1, s2 : std_logic;

begin

C1: comb **port map** (a => a, b => b,
c_in => s1, z => z, carry => s2);

--

-- concurrent procedure call

--

dff(clk => clk, reset => reset, d => s2,
q => s1, qbar => open);

end structural;

18

Equivalent Sequential Procedure Call

```

architecture structural of serial_adder is
component comb
  port (a, b, c_in : in std_logic;
        z, carry : out std_logic);
end component;

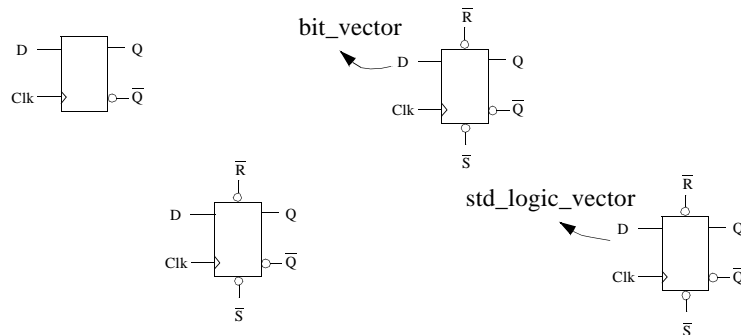
procedure dff(signal d, clk, reset : in std_logic;
              signal q, qbar : out std_logic) is
begin
  if (reset = '0') then
    q <= '0' after 5 ns;
    qbar <= '1' after 5 ns;
  elsif (clk'event and clk = '1') then
    q <= d after 5 ns;
    qbar <= (not D) after 5 ns;
  end if;
end dff;

signal s1, s2 : std_logic;
begin
  C1: comb port map (a => a, b => b,
                    c_in => s1, z => z, carry => s2);
  --
  -- sequential procedure call
  --
  process
  begin
    dff(clk => clk, reset => reset, d => s2,
        q => s1, qbar => open);
  wait on clk, reset, s2;
  end structural;

```

19

Subprogram Overloading



- hardware components differ in number of inputs and the type of input signals
- model each component by a distinct procedure
 - procedure naming becomes tedious

20

Subprogram Overloading

- consider the following procedures for the previous components
dff_bit (clk, d, q, qbar)
asynch_dff_bit (clk, d,q,qbar,reset,clear)
dff_std (clk,d,q,qbar)
asynch_dff_std (clk, d,q,qbar,reset,clear)
- all of the previous components can use the same name -->
subprogram overloading
- the proper procedure can be determined based on the
arguments of the call

21

Essentials of Packages

- Package Declaration
 - declaration of the functions, procedures, and types that are available in the package
 - serves as a package interface
 - only declared contents are visible for external use
- note the behavior of the **use** clause
- Package body
 - implementation of the functions and procedures declared in the package header
 - instantiation of constants provided in the package header

22

Example: Package Header std_logic_1164

```
package std_logic_1164 is
type std_ulogic is ('U', --Uninitialized
    'X', -- Forcing Unknown
    '0', -- Forcing 0
    '1', -- Forcing 1
    'Z', -- High Impedance
    'W', -- Weak Unknown
    'L', -- Weak 0
    'H', -- Weak 1
    '-' -- Don't care
);
type std_ulogic_vector is array (natural range <>) of std_ulogic;

function resolved (s : std_ulogic_vector) return std_ulogic;
subtype std_logic is resolved std_ulogic;

type std_logic_vector is array (natural range <>) of std_logic;

function "and" (l, r : std_logic_vector) return std_logic_vector;
--.<rest of the package definition>
end package std_logic_1164;
```

23

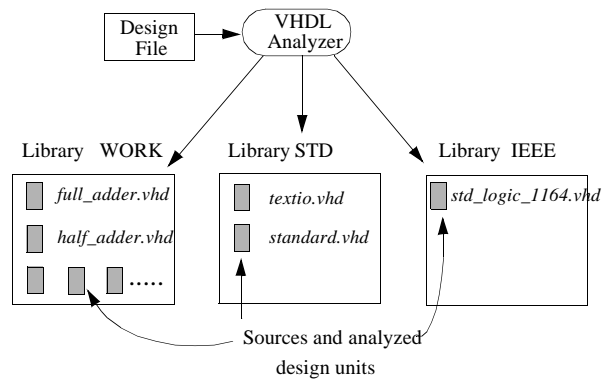
Example: Package Body

```
package body my_package is
--
-- type definitions, functions, and procedures
--
end my_package;
```

- Packages are typically compiled into libraries
- New types must have associated definitions for operations such as logical operations (e.g., and, or) and arithmetic operations (e.g., +, *)
- Examine the package std_logic_1164 stored in library IEEE

24

Essentials of Libraries

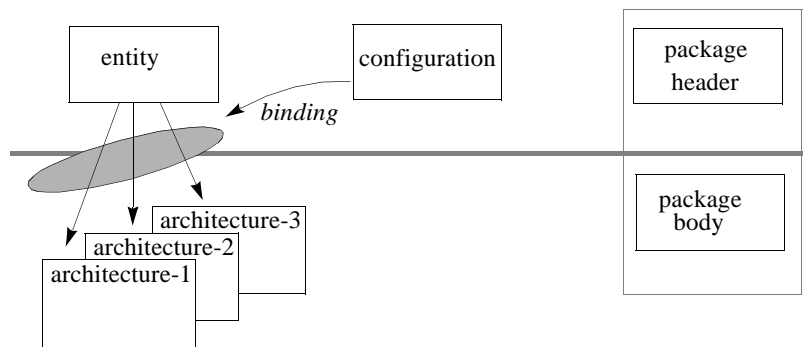


- Design units are analyzed (compiled) and placed in libraries
- Logical library names map to physical directories
- Libraries STD and WORK are implicitly declared

25

Design Units

Primary Design Units



- Distinguish the primary design units

26

Visibility Rules

file.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
entity design-1 is
.....

library IEEE;
use IEEE.std_logic_1164.rising_edge;
entity design-2 is
.....
```

- when multiple design units are in the same file visibility of libraries and packages must be established for each **primary** design unit (entity, package header, configuration) separately!
- The **use** clause may selectively establish visibility, e.g., only the function `rising_edge()` is visible within entity design-2

27

Summary

- Functions
 - resolution functions
- Procedures
 - concurrent and sequential procedure calls
- Subprogram overloading
- Packages
 - package declaration - primary design unit
 - package body
- Libraries
 - relationships between design units and libraries
- Visibility Rules

28