Subprograms, Packages, and Libraries

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;
```

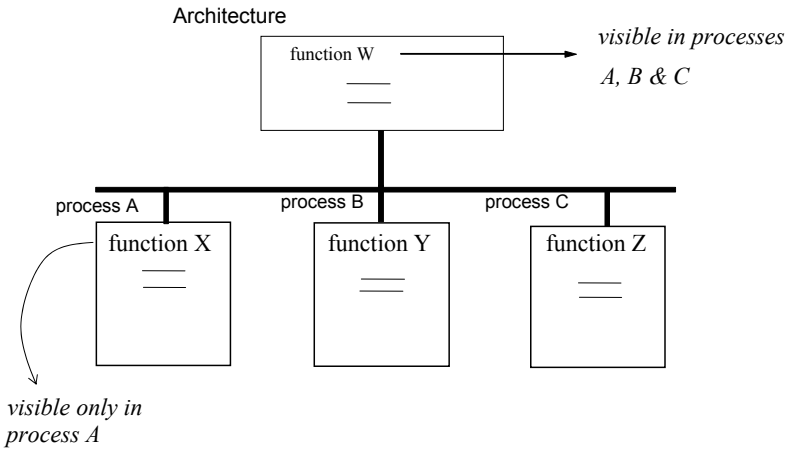
- Formal parameters and mode
  - Default mode is of type **in**
- Functions cannot modify parameters
  - Pure functions vs. impure functions
    - Latter occur because of visibility into signals that are not parameters
- Function variables initialized on each call

Essentials of Functions (cont.)

```
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;
```

- Types of formals and actuals must match except for formals which are constants (default)
  - Formals which are constant match actuals which are variable, constant or signal
- Wait statements are not permitted in a function!
  - And therefore not in any procedure called by a functions

Placement of Functions



- Place function code in the declarative region of the **architecture** or **process**

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;
```

Architecture Declarative Region

```
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;
```

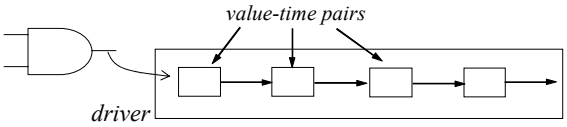

Function: Example

```
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
```

- A common use of functions: type conversion
- Use of attributes for flexible function definitions
  - Data size is determined at the time of the call
- Browse the vendor supplied packages for many examples

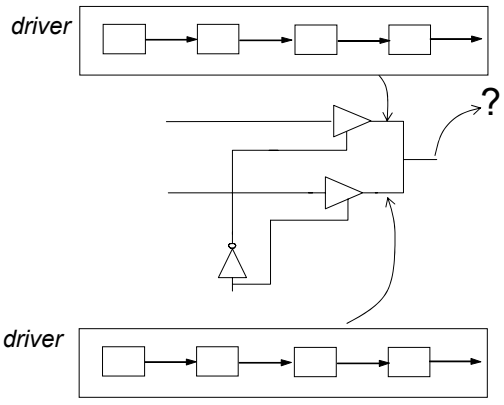
Implementation of Signals

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



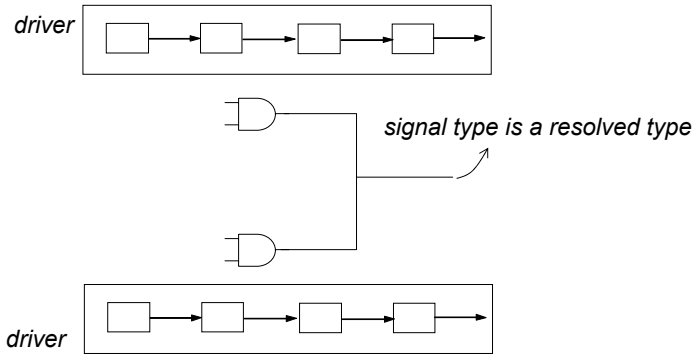
- Holds the current and future values of the signal - a projected waveform
- Signal assignment statements modify the driver of a signal
- Value of a signal is the value at the head of the driver

Shared Signals



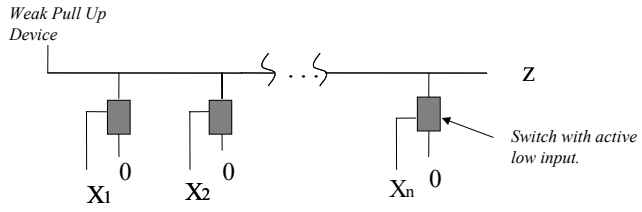
- How do we model the state of a wire?
- Rules for determining the signal value is captured in the *resolution function*

Resolved Signals



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

Resolution Function Behavior



- Physical operation
  - If any of the control signals activate the switch, the output signal is pulled low
- VHDL model
  - If any of the drivers attempt to drive the signal low (value at the head of the driver), the resolution functions returns a value of 0
  - Resolution function is invoked when any driver attempts to drive the output signal

Resolved Types: std_logic

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 only supports only single drivers

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

New subtype supports multiple drivers

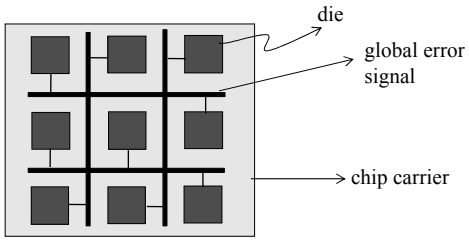
Resolution Function: std_logic & resolved()

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

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

Example



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

A Complete Example

Resolution function

```
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;
```

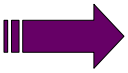
New resolved type

```
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
  - This is why the resolution function must be associative!

Synthesis Considerations: Functions

- Function in-lining model for synthesis
- All local variables initialized on each call and all outputs are computed



Combinational Logic

Summary: Essentials of Functions

- Placement of functions
  - Visibility
- Formal parameters
  - Actuals can have widths bound at the call time
- Check the source listings of packages for examples of many different functions

Essentials of Procedures

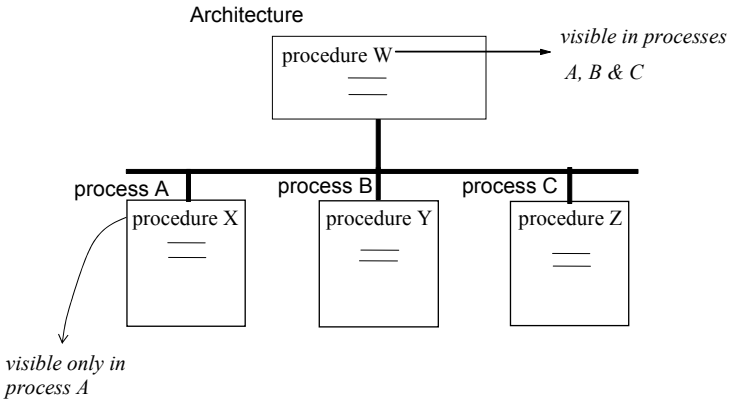
```
procedure read_v1d (variable f: in text; v :out std_logic_vector)
--declarative region: declare variables local to the procedure
--
begin
-- body
--
end read_v1d;
```

- 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

Procedures: Placement

```
architecture behavioral of cpu is
--
-- declarative 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 process_a
-- procedures can be placed here
begin
--
-- process body
--
end process_a;
process_b: process
--declarative regions                                   --> visible only within process_b
begin
-- process body
end process_b;
end architecture behavioral;
```

Placement of Procedures



- Placement of procedures determines visibility in its usage

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;
```

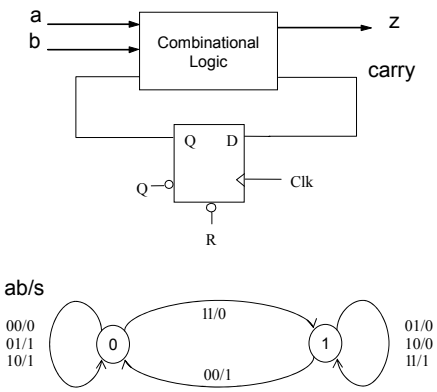
- Procedures can make assignments to signals passed as input parameters
- Procedures may not have a **wait** statement if the encompassing process has a sensitivity list

Procedures 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 may modify signals not in the parameter list, e.g., ports
- Signals may not be declared in a procedure
- Procedures may make assignments to signals not declared in the parameter list

Concurrent vs. Sequential Procedure Calls



- Example: bit serial adder

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 (rising_edge(clk)) 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 architectural structural;
```

- Variables cannot be passed into a concurrent procedure call
- Explicit vs. positional association of formal and actual parameters

Equivalent Sequential Procedure Call

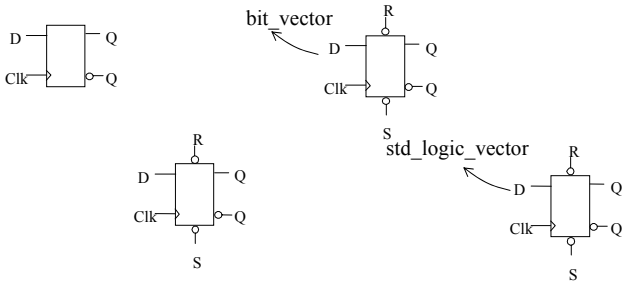
```
architecture structural of serial_adder is
component comb
port (a, b, c_in : in std_logic;
z, carry : out std_logic);
end component;
procedure dff(signal d, clk, reset : in std_logic;
signal q, qbar : out std_logic) is
begin
if (reset = '0') then
q <= '0' after 5 ns;
qbar <= '1' after 5 ns;
elsif (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 process;
end architecture structural;
```

Synthesis Considerations: Procedures

- Procedure in-lining model for synthesis
- Synthesis compilers generally restrict processes to one **wait** statement
  - Cannot predict how procedures will be used
    - Especially when using packages
  - Therefore **wait** statements in procedures are generally avoided
- Storage is inferred for parameters of mode **out** if encompassed in conditional blocks
  - Local variables are always initialized and hence will synthesize to wires

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

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
  - Example

```
function "*" (arg1, arg2: std_logic_vector) return std_logic_vector;
function "+" (arg1, arg2 :signed) return signed;
-- the following function is from std_logic_arith.vhd
--
```


Subprogram Overloading

- VHDL is a strongly typed language
- Overloading is a convenient means for handling user defined types
- We need a structuring mechanism to keep track of our overloaded implementations

Packages!

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

Example: Package Header std_logic_1164

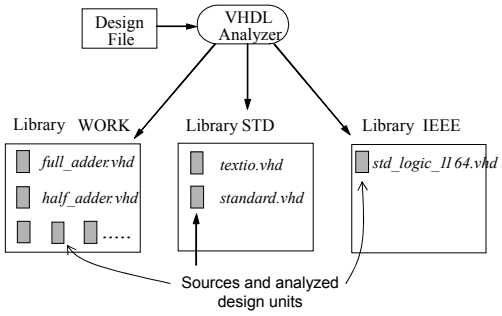
```
package std_logic_1164 is
type std_ulogic is ('U', --Unitialized
'X', -- Forcing Unknown
'0', -- Forcing 0
'1', -- Forcing 1
'Z', -- High Impedance
'W', -- Weak Unknown
'L', -- Weak 0
'H', -- Weak 1
'-' -- Don't care
);
type std_ulogic_vector is array (natural range <>) of std_ulogic;
function resolved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

Example: Package Body

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

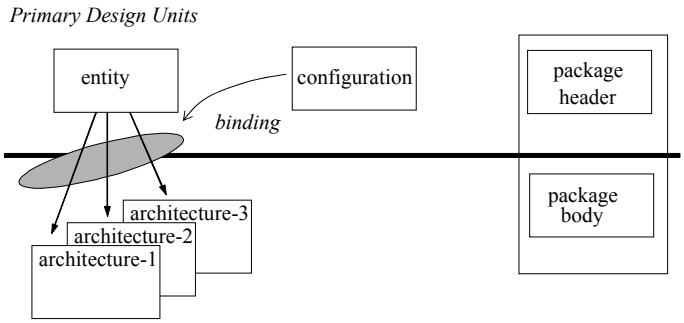
- Packages are typically compiled into libraries
- New types must have associated definitions for operations such as logical operations (e.g., and, or) and arithmetic operations (e.g., +, *)
- Examine the package std_logic_1164 stored in library IEEE

Essentials of Libraries



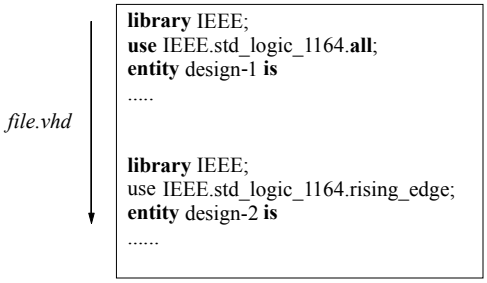
- Design units are analyzed (compiled) and placed in libraries
- Logical library names map to physical directories
- Libraries STD and WORK are implicitly declared

Design Units



- Distinguish the primary and secondary design units
- Compilation order

Visibility Rules



- 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!
  - Secondary design units derive library information from associated primary design unit
- The **use** clause may selectively establish visibility, e.g., only the function rising_edge() is visible within entity design-2
  - Secondary design inherit visibility
- Note design unit descriptions are decoupled from file unit boundaries

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