VHDL
Sub-Programs, Packages, & Libraries

EL 310

Erkay Savaş
Sabancı University

Motivation

- Structuring VHDL programs
  - modularity
- Design reuse
- Manage the complexity
- Available VHDL tools:
  - procedures,
  - functions,
  - packages,
  - libraries

Issues

- VHDL programs model physical systems
- There may have some issues we have to deal with such as:
- Can `wait` statements be used in a procedure?
- Can signals be passed to procedures and be modified within the procedure?
- How are procedures synthesized?
- Can functions operate on signals?

Packages & Libraries

- Groups of procedures and functions that are related can be aggregated into a module that is called package.
- A package can be shared across many VHDL models.
- A package can also contains user defined data types and constants.
- A library is a collection of related packages.
- Packages and libraries serve as repositories for functions, procedures, and data types.

Functions

- A function computes and returns a value of specified type using the input parameters.
- Function declaration:
  - **`function`** `rising_edge(`**`signal`** `clock:` **`in`** `std_logic)` **`return Boolean`**`;`
- Parameters are used but not modified within the function.
  - the mode of input parameters is **`in`**.
  - functions do not have parameters of mode **`out`**.
  - In fact, we do not have to specify the mode.

Function Definition

```
function rising_edge(signal clock: std_logic)
return Boolean is
```

formal parameters

```
--
-- declarative region: declare variables local to the function
--
begin
--
-- body
--
return (value);
end function rising_edge;
```

- The function is called with the <u>actual parameters</u>.
- Example: `rising_edge(enable);`
- Types of formal and actual parameters must match.
- Actual parameters could be variable, signal, constant or an expression.

Functions

- When no class is specified, parameters are by default constants.
- **wait** statements cannot be used in functions' body.
  - functions execute in zero simulation time.
- Functions cannot call a procedure that has a wait statement in it.
- Parameters have to be mode **in**.
  - signals passed as parameters cannot be assigned values in the function body.

Pure vs. Impure Functions

- VHDL’93 supports two distinct types of functions:
- Pure functions
  - always return the same value when called with the same parameter values.
- Impure functions
  - may return different values even if they are called with the same parameter values at different times.
  - All the signals in the architecture is visible within the function body
  - Those signals may not appear in the function parameter list (e.g. ports of the entity)

Example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is
  port(d, clock: in std_logic; q, qbar: out std_logic);
end entity dff;
architecture beh 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 function rising_edge;
begin
  output: process is
  begin
    wait until (rising_edge(clock));
    q <= d after 5 ns;
    qbar <= not d after 5 ns;
  end process output;
end architecture beh;
```

declarative region of architecture

Type Conversion Functions

```
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 function to_bitvector;
```

Unconstrained array

std_logic_arith

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;

type tbl_type is array (STD_ULOGIC) of STD_ULOGIC;
    constant tbl_BINARY : tbl_type :=
        ('X', 'X', '0', '1', 'X', 'X', '0', '1', 'X');

function CONV_INTEGER(ARG: UNSIGNED) return INTEGER is
  variable result: INTEGER;
  variable tmp: STD_ULOGIC;
  -- synopsys built_in SYN_UNSIGNED_TO_INTEGER
  -- synopsys subpgm_id 366
begin
  -- synopsys synthesis_off
  assert ARG'length <= 31
    report "ARG is too large in CONV_INTEGER"
    severity FAILURE;
  result := 0;
```

std_logic_arith

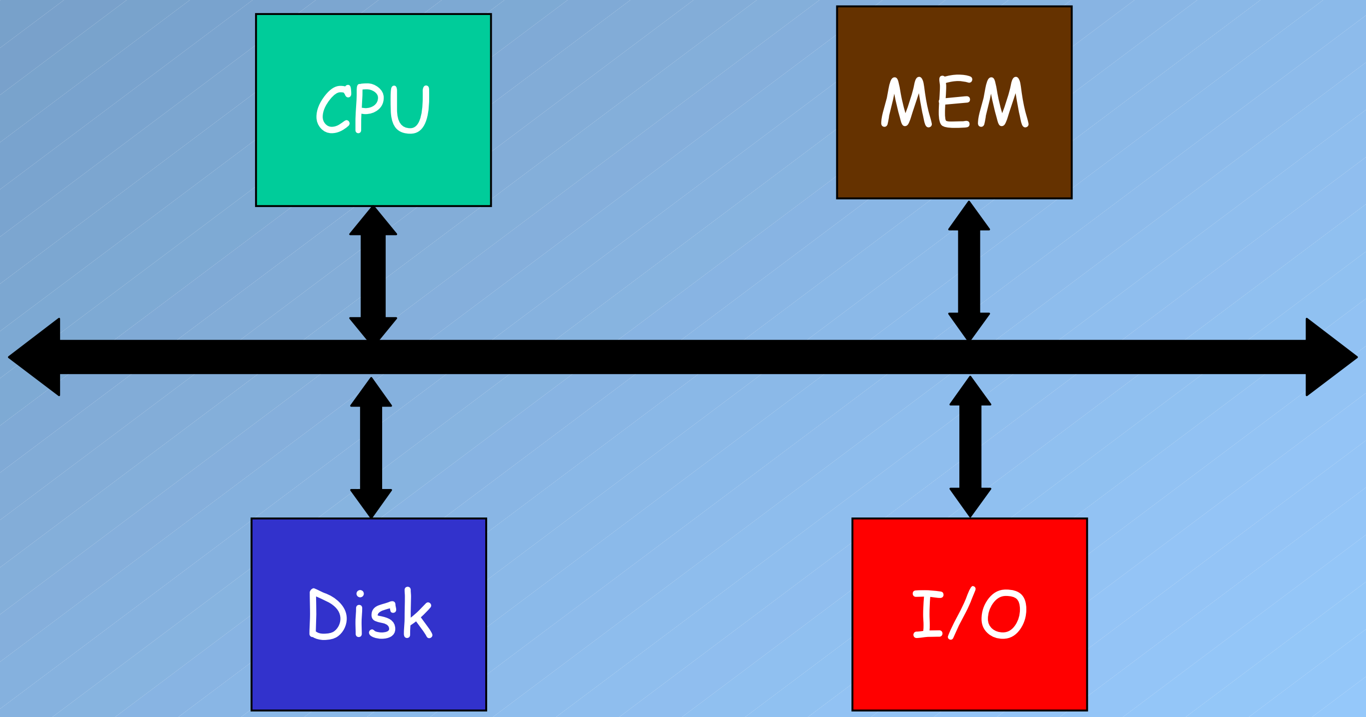
```
  for i in ARG'range loop
    result := result * 2;
    tmp := tbl_BINARY(ARG(i));
    if tmp = '1' then
      result := result + 1;
    elsif tmp = 'X' then
      assert false
         report "There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic
                operand, the result will be 'X'(es)."
        severity warning;
      assert false
         report "CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in
           an arithmetic operand, and it has been converted to 0."
         severity WARNING;
      return 0;
    end if;
  end loop;
  return result;
-- synopsys synthesis_on
end;
```

std_logic_arith

```
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT is
  variable tmp: STD_ULOGIC;
  -- synopsys built_in SYN_FEED_THRU
  -- synopsys subpgm_id 370
begin
  -- synopsys synthesis_off
  tmp := tbl_BINARY(ARG);
  if tmp = '1' then
    return 1;
  elsif tmp = 'X' then
    assert false
      report "CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an
           arithmetic operand, and it has been converted to 0."
      severity WARNING;
    return 0;
  else
    return 0;
  end if;
-- synopsys synthesis_on
end;
```

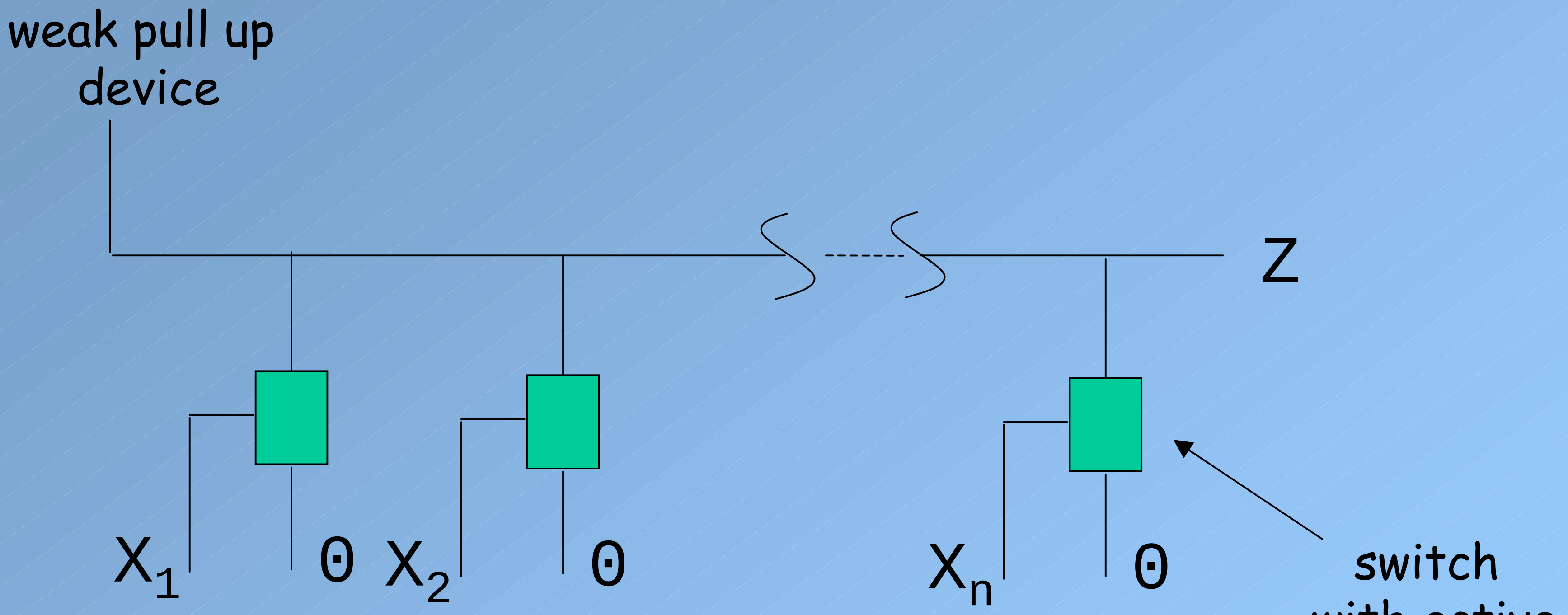
Resolution Functions

- For system busses, we use resolved signals



Resolution Functions

- Wired-logic implementations



Any $X_i = 0$ turns the switch on driving $Z = 0$

Resolution Functions

```
entity wired_and is
  port( X1, X2, X3: in bit; Z : out resolved_bit);
end entity;
architecture beh of wired_and is
begin
  x_process: process(X1) is
  begin
    Z <= X1;
  end process;
  y_process: process(X2) is
  begin
    Z <= X2;
  end process;
  z_process: process(X3) is
  begin
    Z <= X3;
  end process;
end architecture beh;
```

Resolved Types in the IEEE 1164 Standard

```
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
        );
```

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

Resolution function must perform an associative operations so that the order in which the multiple signal drivers are examined does not affect the resolved value of the signal.

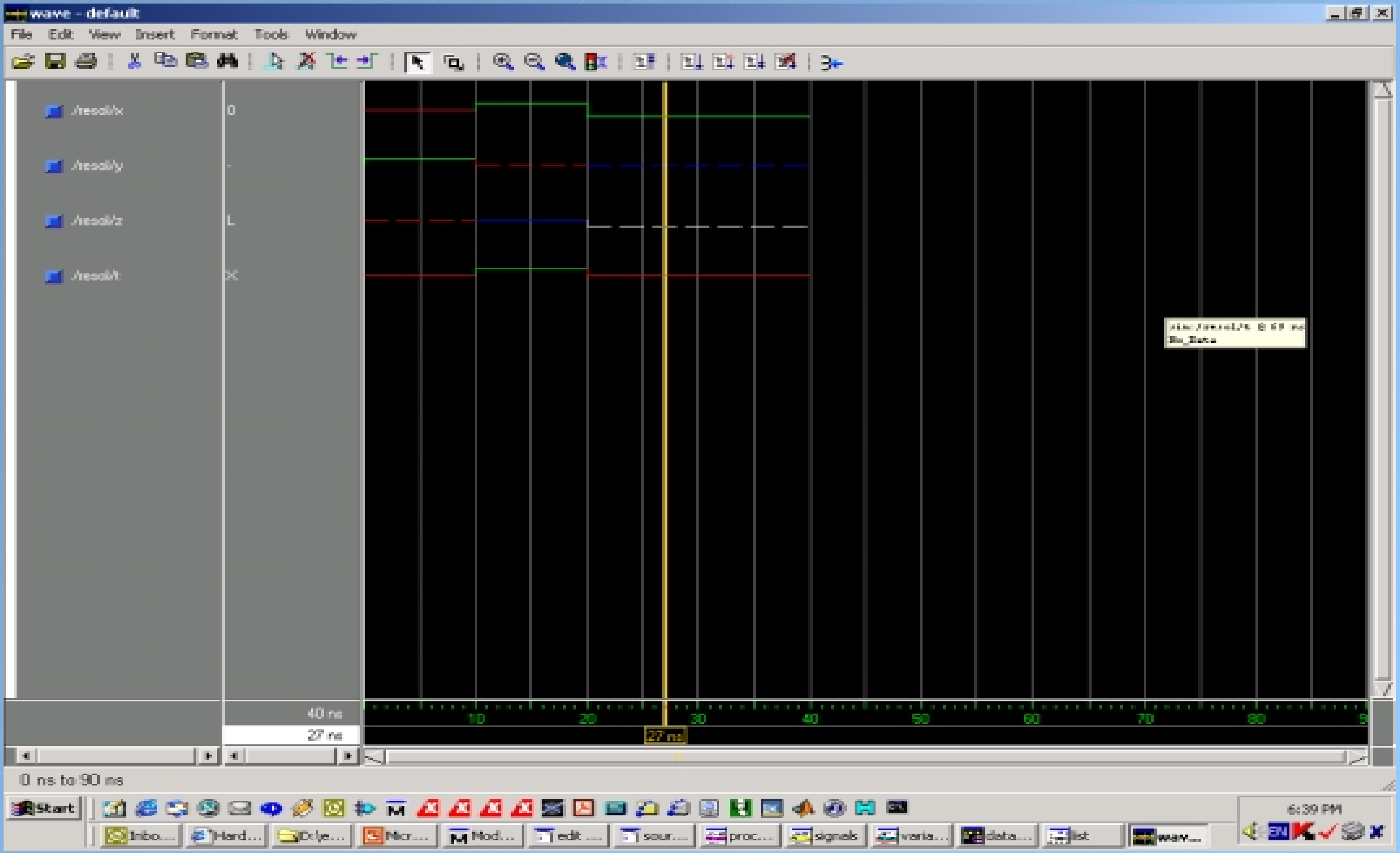
Resolution Table for `std_logic`

	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

Example: Multiple Drivers

```
entity resol is
end entity;
architecture beh of resol is
  signal X, Y, Z, T: std_logic;
begin
  X <= 'U', '1' after 10 ns, '0' after 20 ns;
  Y <= '1', 'W' after 10 ns, '-' after 20 ns;
  Z <= 'W', 'Z' after 10 ns, 'L' after 20 ns;
  x_process: process(X) is
  begin
    T <= X;
  end process;
  y_process: process(Y) is
  begin
    T <= Y;
  end process;
  z_process: process(Z) is
  begin
    T <= Z;
  end process;
end architecture beh;
```

Example: Waveforms

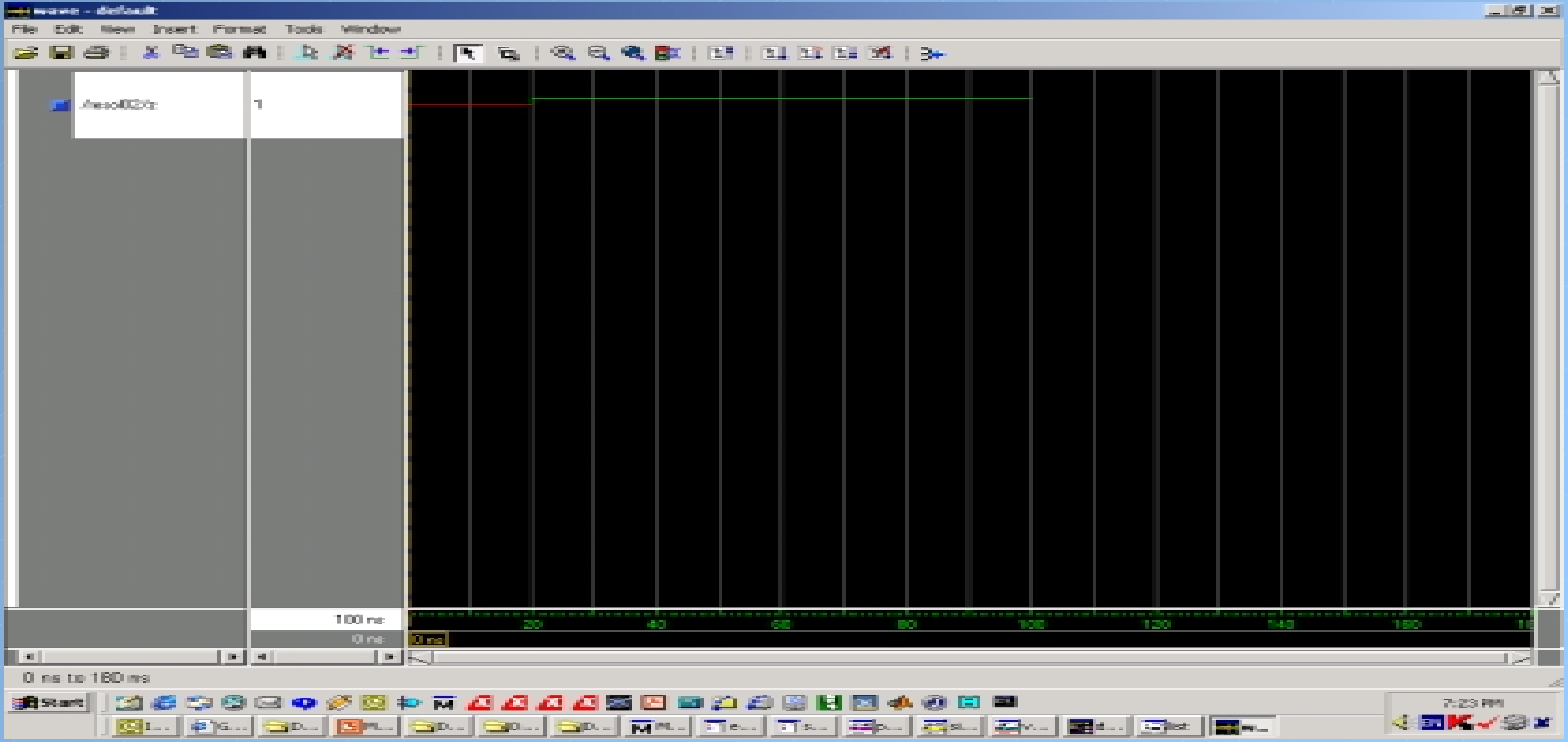


Resolution Function

```
architecture beh of resol02 is
  function resolution (drivers: std_ulogic_vector) return std_logic is
    type my_array_type is array (std_logic, std_logic) of STD_ULOGIC;
    constant my_array : my_array_type :=
         (('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'),
          ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'),
          ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'),
          ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'),
          ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'),
          ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'),
          ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'),
          ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'),
          ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'));
    variable tmp: std_ulogic;
  begin
    tmp := my_array(drivers(drivers'length-1),drivers(drivers'length-2));
    for i in drivers'length-3 downto 0 loop
      tmp := my_array(tmp, drivers(i));
    end loop;
    return tmp;
  end function;
```

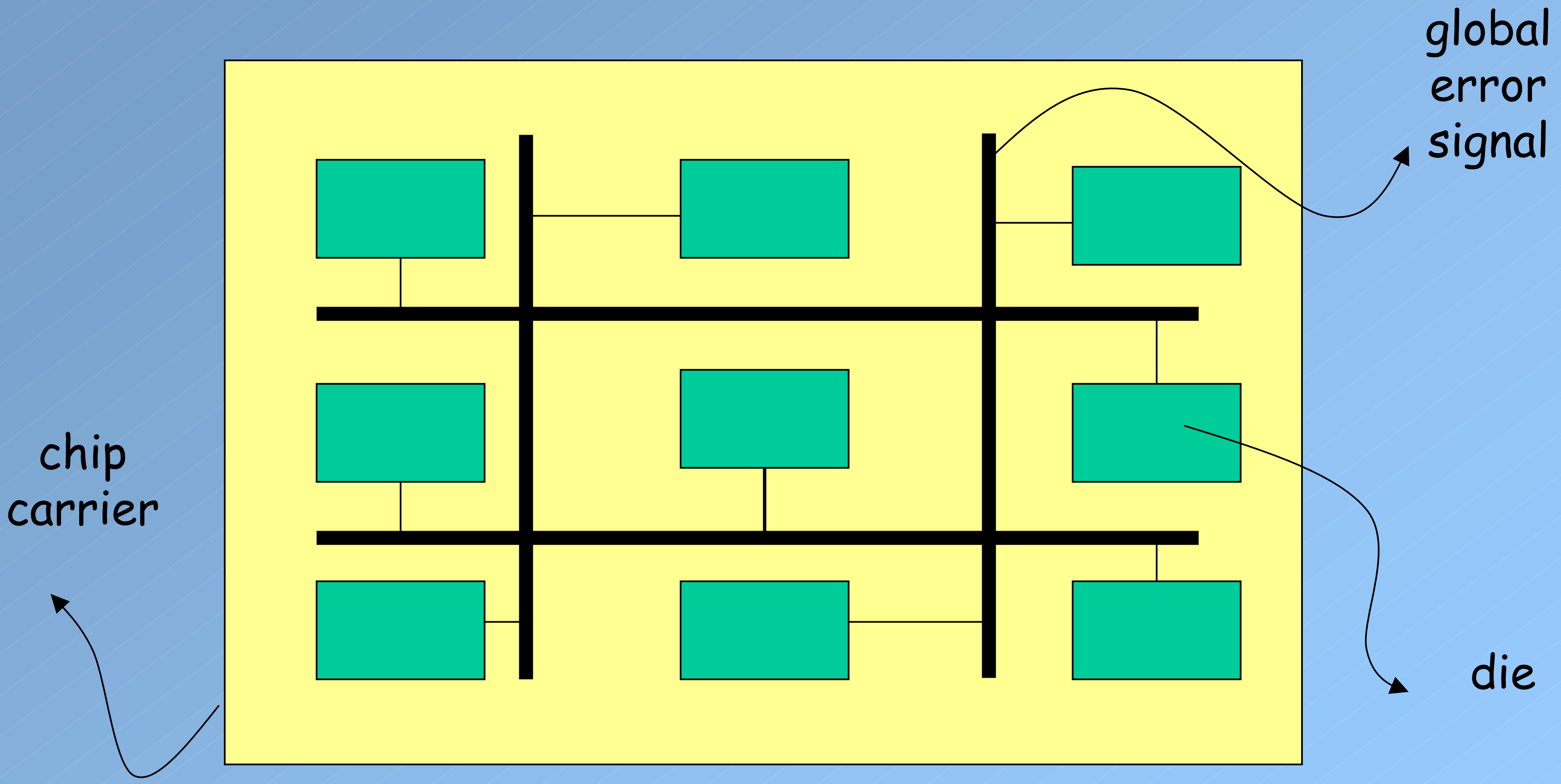
Resolution Function

```
signal Z: std_ulogic;
begin
  Z <= resolution("1HWW") after 20 ns;
end architecture beh;
```



Example: Resolution Functions

- Multi-chip module with multiple die



Example: Resolution Functions

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mcm is -- and empty entity declaration
end entity mcm;
architecture beh of mcm is
  function wire_or(sbus:std_ulogic_vector)
    return std_ulogic is
  begin
    for i in sbus'range loop
      if sbus(i) = '1' then return '1'; end if;
    end loop;
    return '0';
  end function wire_or;   resolution
                           function    base type

  subtype wire_or_logic is wire_or std_ulogic;
  signal error_bus: wire_or_logic;
...
```

Example: Resolution Functions

```
...
  subtype wire_or_logic is wire_or std_ulogic;
  signal error_bus: wire_or_logic;
begin
  chip1: process is
  begin
    ...
    error_bus <= ‘1’ after 2 ns;
    ...
  end process chip1;
  chip2: process is
  begin
    ...
    error_bus <= ‘0’ after 2 ns;
    ...
  end process chip2;
end architecture beh;
```

Synthesis Example

```
entity func is
  port(data: in std_logic_vector( 7 downto 0);
       count: out integer range 0 to 7);
end entity func;
architecture beh of func is
  function ones(signal data: std_logic_vector) return integer is
    variable count: integer range 0 to 7;
  begin
    for i in data’range loop
      if data(i) = ‘1’ then count:= count + 1; end if;
    end loop;
    return (count);
  end function ones;
begin
 check: process(data) is
 begin
   count <= ones(data);
 end process check;
end architecture beh;
```

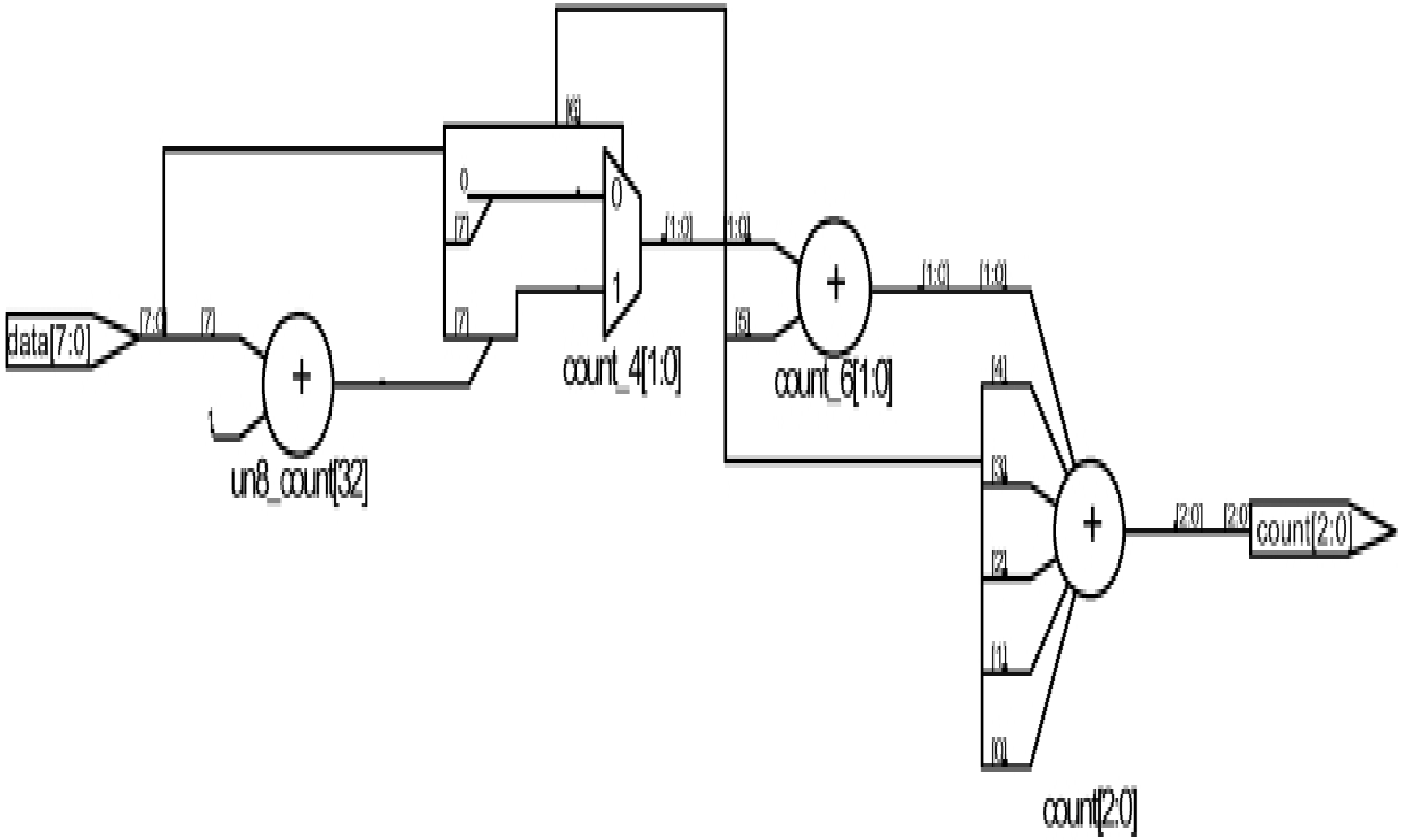
Synthesis Example (cont.)

- `count <= ones(data);`
- each time this function is called, internal variables are initialized, the function is executed, and output value is returned.
- Internal variables do not retain value across different invocations.
- The loop is unrolled

```
if data(0) = ‘1’ then count := count + 1; end if;
if data(1) = ‘1’ then count := count + 1; end if;
...
```

- When synthesized, it will produce a combinational circuit with a long dependency.

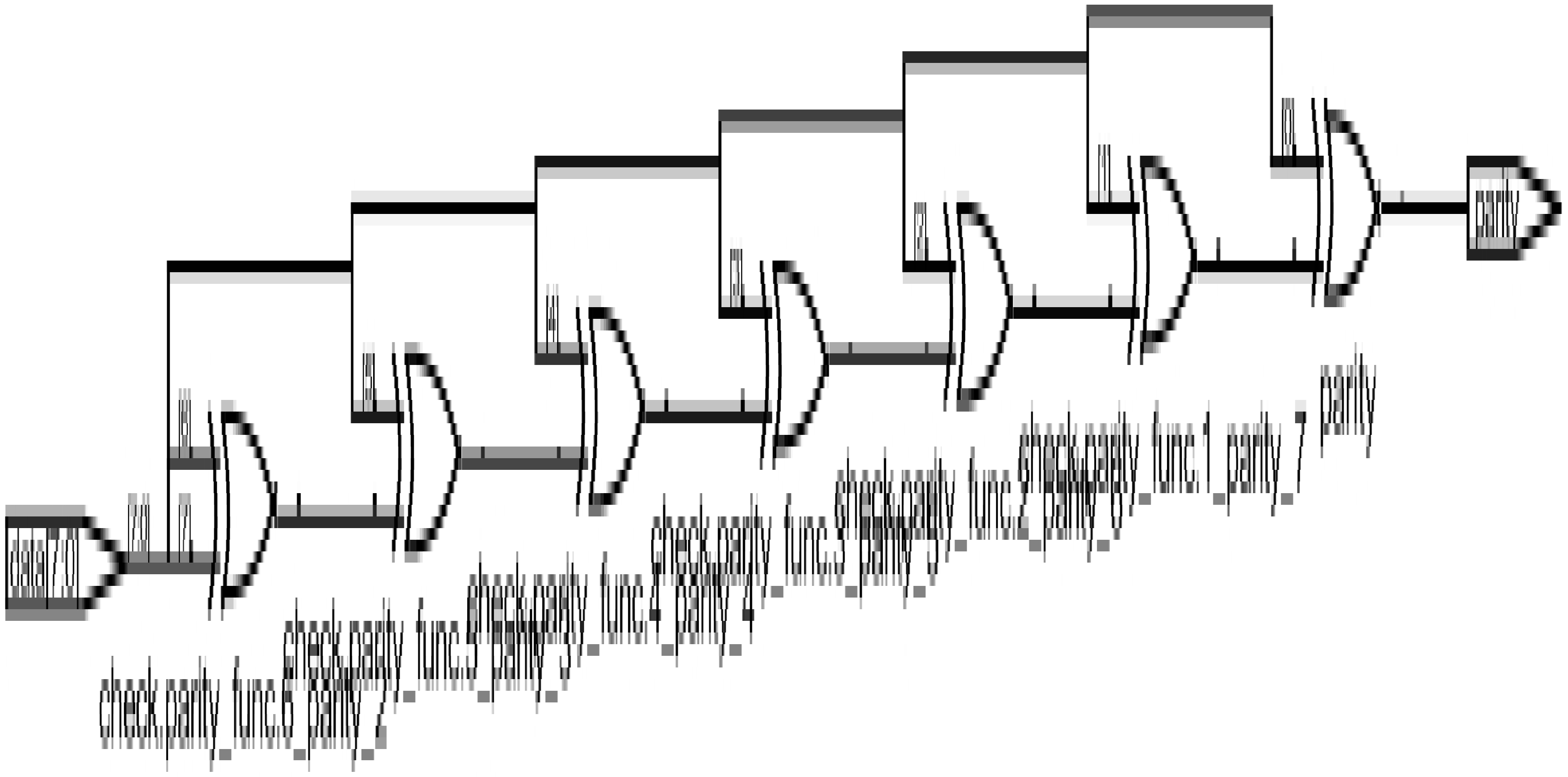
Synthesis Example (cont.)



Yet Another Example

```
entity func02 is
port(data: in std_logic_vector(7 downto 0); parity: out std_logic);
end entity func02;
architecture beh of func02 is
  function parity_func(signal data: std_logic_vector)
   return std_logic is
    variable parity: std_logic:='0';
  begin
  for i in data'range loop
    parity := parity xor data(i);
  end loop;
  return (parity);
  end function parity_func;
begin
 parity_process: process(data) is
 begin
   parity <= parity_func(data);
 end process check;
end architecture beh;
```

Yet Another Example



data[7:0]
[7:0]
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
parity
check_parity_func.6_parity_2
check_parity_func.5_parity_3
check_parity_func.4_parity_4
check_parity_func.3_parity_5
check_parity_func.2_parity_6
check_parity_func.1_parity_7
parity

Procedures

- Similar to functions
- distinguishing feature is that procedures can modify input parameters.
- Syntax:
  - **`procedure`** `read_v1d(`**`file`** `fname:` **`in`** `text; v:` **`out`** `std_logic_vector);`
  - a procedure that reads data from a file called `fname`.
- Parameters can be of `in`, `out`, `inout` modes.
  - default class is `constant` for input parameters.
  - default class is `variable` for `out` and `inout` mode parameters.

Procedures

- Variables declared within a procedure are initialized in each call
- Variables do not retain values across procedure calls.
- Example:

```
entity CPU is
  port(write_data: out std_logic_vector(31 downto 0);
       ADDR: out std_logic_vector(2 downto 0);
       MemRead, MemWrite: out std_logic;
       read_data: in std_logic_vector(31 downto 0);
       S: in std_logic);
end entity CPU;
```

Example: CPU + Memory

```
architecture beh of CPU is
  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 <= read_data;
     R <= ‘0’;
   end procedure mread;
  ...
end architecture beh;
```

Example: CPU + Memory

```
architecture beh of CPU is
...
  procedure mwrite(
    address: in std_logic_vector(2 downto 0);
    signal data: in std_logic_vector(31 downto 0);
    signal ADDR: out std_logic_vector(2 downto 0);
    signal W: out std_logic;
    signal DO: out std_logic_vector(31 downto 0)) is
  begin
    ADDR <= address;
    DO <= data;
    W <= ‘1’;
    wait until S = ‘1’;
    W <= ‘0’;
end procedure mwrite;
  ...
end architecture beh;
```

Example: CPU + Memory

```
architecture beh of CPU is
...
-- CPU description here
begin
  process is
  begin
  --
  -- behavioral description
  --
  end process;
process is
  begin
  --
  -- behavioral description
  --
  end process;
end architecture beh;
```

Using Procedures

- Signals can be passed to procedures and updated within procedures
- Signals cannot be declared within procedures
- Visibility rules apply here
  - procedure can update signals visible to it even if these signals do not appear in the parameter list.
  - This is sometimes called as side effect of procedure.
  - updating signals that do not appear in the parameter list is a poor programming practice.
- A process calling a procedure with a **`wait`** statement cannot have sensitivity list

Concurrent vs. Sequential Procedure Calls

- Concurrent procedure call
  - procedure calls can be made in concurrent signal assignments
  - execution of a procedure is concurrent with other concurrent procedures, CSA statements, or processes.
  - The procedure is invoked when there is an event on a signal that is a parameter to the procedure.
  - Parameter list cannot include a variable in a concurrent procedure call (shared variables can be included though).
- Sequential procedure call
  - executed within the body of a process.
  - executions is determined by the order.

Example: Concurrent Procedure Call

```
entity serial_adder is
  port(a, b, clk, reset: in std_logic; z: out std_logic);
end entity;
architecture structural of serial adder is
  component comb is
    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 procedure;
  signal s1, s2: std_logic;
begin
  C1: comb port(a=>a, b=>b, c_in=>s1, z=>z, carry=>s2);
  dff(clk=>clk, reset=>reset, d=>s2, q=>s1, qbar=>open);
end architecture structural;
```

behavioral description

explicitly associating formal and actual parameters

Example: Sequential Procedure Call

```
entity serial_adder is
  port(a, b, clk, reset: in std_logic; z: out std_logic);
end entity;
architecture structural of serial adder is
  component comb is
    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
  ...
  end procedure;
  signal s1, s2: std_logic;
begin
  C1: comb port(a=>a, b=>b, c_in=>s1, z=>z, carry=>s2);
  process
  begin
    dff(clk=>clk, reset=>reset, d=>s2, q=>s1, qbar=>open);
    wait on clk, reset, s2;
  end process;
end architecture structural;
```

Synthesis & Procedures

- In-lining approach
  - during synthesis, the compiler replaces the procedure calls with the corresponding code (flattening)
  - therefore, inferencing techniques are applicable
- Latch inference
  - local variables do not retain value across invocations; they will be synthesized into wires
  - signals of mode **out** may infer latches.
  - for example, if procedure resides in a conditional block of code, latches will be inferred for output signals of the procedure

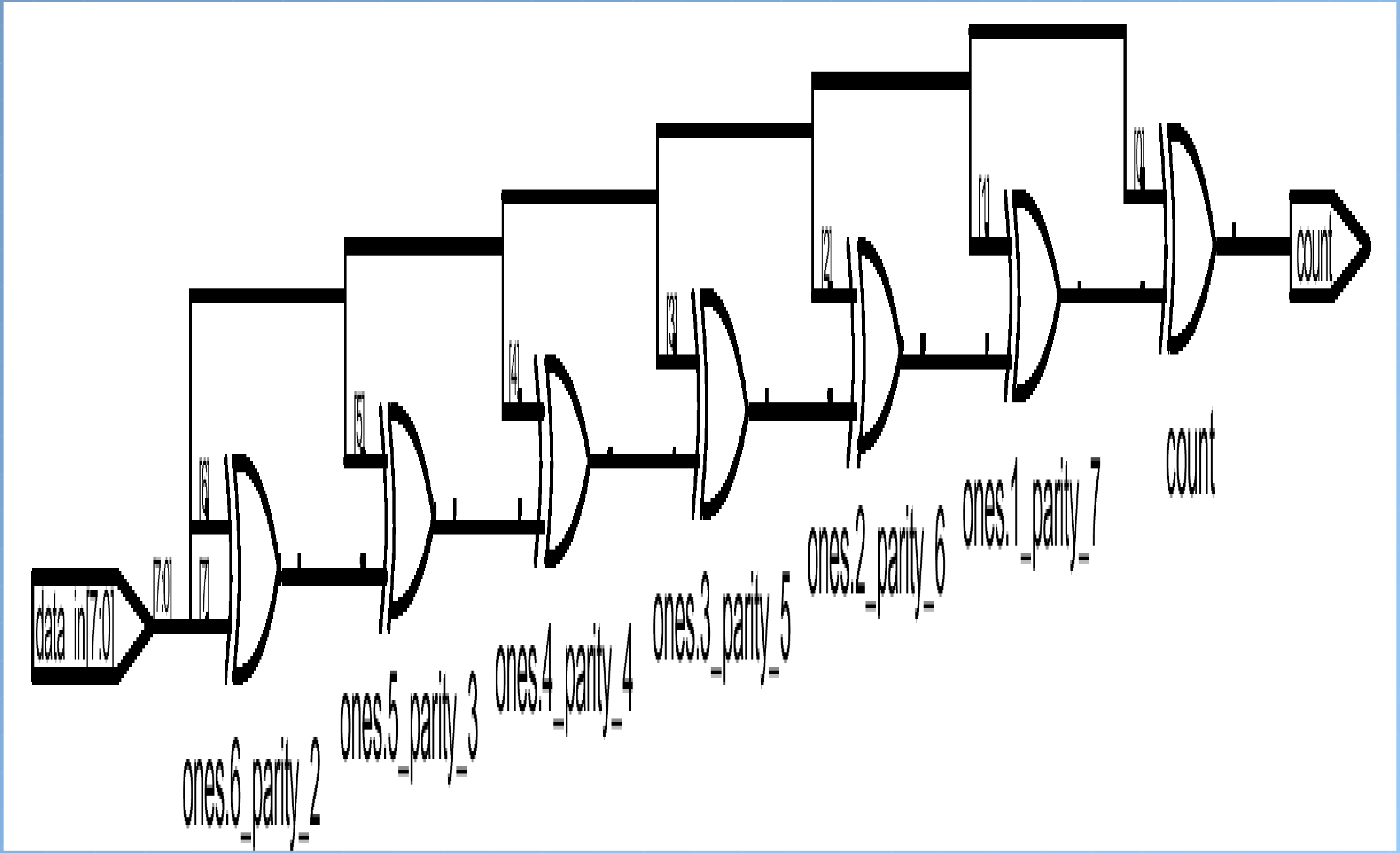
Synthesis & Procedures

- Wait statements
  - recall that synthesis compilers allows only one **wait** statement in a process.
  - procedures that are being called within a process that has already a **wait** statement cannot have a **wait** statement of its own.
  - Therefore, **wait** statements are generally not supported in procedures for synthesis.
- For synchronous logic
  - use **if** statements in the procedure

Example

```
entity proc_call is
  port(data:in std_logic_vector(7 downto 0); count: out std_logic);
end entity proc_call;
architecture beh of proc_call is
  procedure ones(signal data: in std_logic_vector;
                 signal count: out std_logic) is
    variable parity: std_logic:=‘0’;
  begin
    for i in data’range loop
      parity := parity xor data(i);
    end loop;
    count <= parity;
  end procedure ones;
begin
   ones(data, count); -- concurrent procedure call
end architecture beh;
```

Example

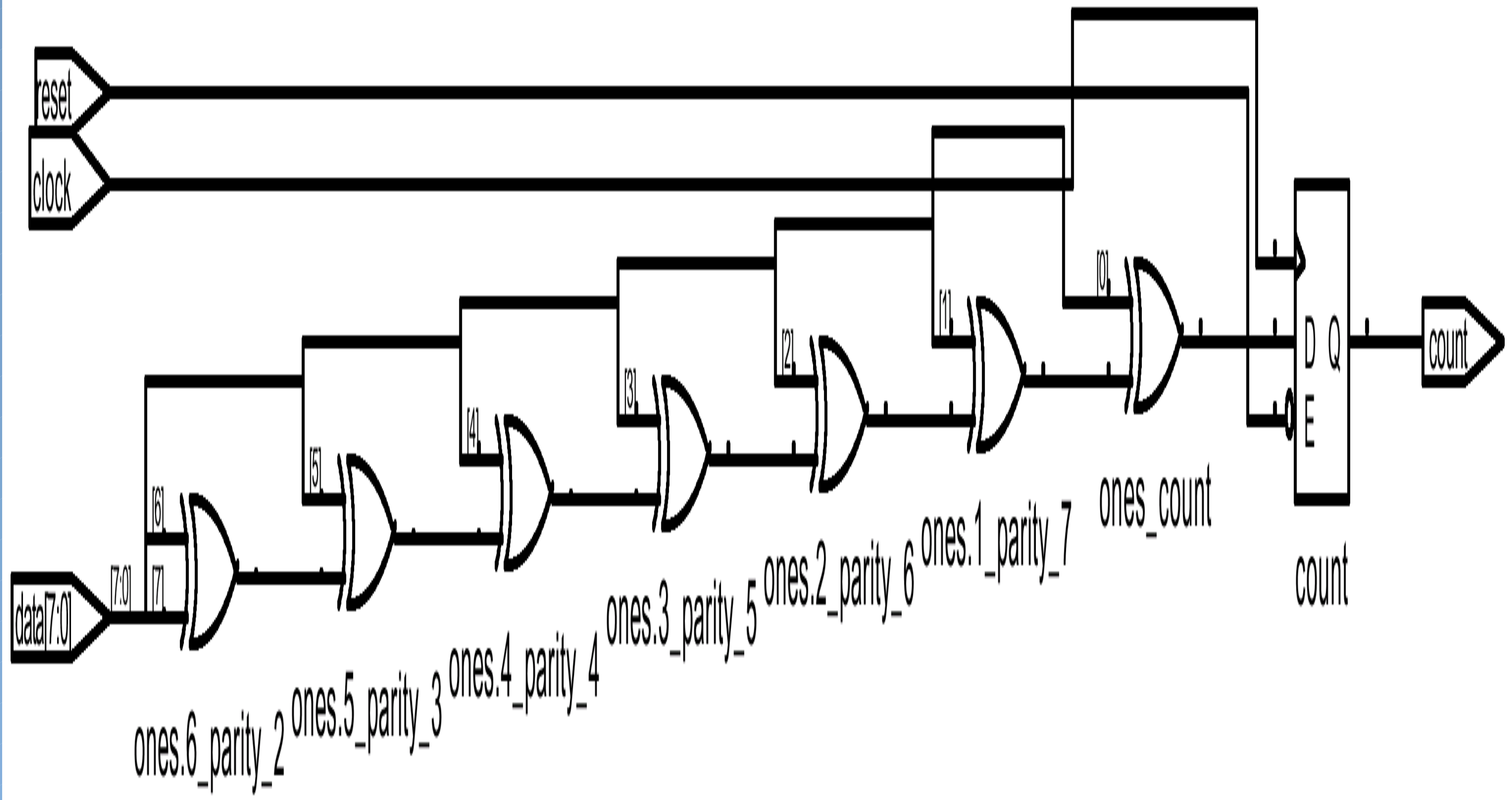


data_in[7:0]
[7:0]
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
ones.6_parity_2
ones.5_parity_3
ones.4_parity_4
ones.3_parity_5
ones.2_parity_6
ones.1_parity_7
count
count

Yet Another Example

```
entity proc_call is
  port(reset, clock: in std_logic;
       data:in std_logic_vector(7 downto 0); count: out std_logic);
end entity proc_call;
architecture beh of proc_call is
  procedure ones(signal data: in std_logic_vector;
                 signal count: out std_logic) is
    variable parity: std_logic:=‘0’;
  begin
    for i in data’range loop
      parity := parity xor data(i);
    end loop;
    count <= parity;
  end procedure ones;
begin
  process(reset, clock) is
  begin
   if reset = ‘0’ and rising_edge(clock) then
     ones(data, count);
    end if;
  end process;
end architecture beh;
```

Yet Another Example



Subprogram Overloading

- Subprograms
  - functions and procedures
- Sometimes, it is convenient to have two or more subprograms with the same name

  -
    ```
    function count (oranges: integer) return integer;
    function count (apples: bit) return integer;
    ```

  -
    ```
    dff(clk, d, q, qbar);
    dff(clk, d, q, qbar, reset, clear);
    ```
- In this case, subprograms are overloaded.
  - compiler will decide which subprogram to call based on the number and type of arguments.
  - and, or etc are overloaded in std_logic_1164 package.

Operator Overloading

- An operator that behaves different depending on the operands is said to be overloaded
  - For example, "*" and "+" are defined for certain predefined types of the language such as integers.
  - What if we want to use these operators to do multiplication and addition one data types we newly defined.
- Example
  - **function** `"*" (arg1, arg2: std_logic_vector)` **return** `std_logic_vector;`
  - **function** `"+" (arg1, arg2: signed)` **return** `signed;`
  - these examples are taken from `std_logic_arith.vhd` package.

Operator Overloading

- You can define a new multi-value logic and logic operations on the data of this new type.

```
type MVL is ('U', '0', '1', 'Z');
function "and" (L, R: MVL) return MVL;
function "or" (L, R: MVL) return MVL;
function "not" (R: MVL) return MVL;
```

- Example:

```
signal a, b, c: MVL;
a <= 'z' or '1';
b <= "or"('0', '1');
c <= (a or b) and (not c);
```

Packages

- A package provides a convenient mechanism to store items that can be shared across distinct VHDL programs:
  - Those items are type definitions, functions, procedures, etc.
  - We group logically related sets of functions and procedures into a package.
  - When working with a large design project consisting of many small VHDL programs, it is convenient to have common procedures and functions in separate packages.
  - For example, a package containing definition of new types for registers, instructions, memories, etc. will certainly be useful for microprocessor design.

Package Declaration

- Package declaration
  - contains information about what is available in the package that we can use in our VHDL programs.
  - In other words, it contains interface or specifications of the functions and procedures that are available in the package.
  - For example,
  - list of functions and procedures,
  - what are the parameters they take
  - the type of input parameters
  - What it returns,
  - what is the type of the returning value
  - etc.

Package Declaration: Syntax

```
package package-name is
  package-item-declarations → these may be:
  -- subprogram declarations
  -- type declarations
  -- subtype declarations
  -- constant declarations
  -- signal declarations
  -- variable declarations
  -- file declarations
  -- alias declarations
  -- component declarations
  -- attribute declarations
  -- attribute specifications
  -- disconnection specifications
  -- use clauses
end [package] [package-name];
```

Package Declaration: Example

```
package synthesis_pack is
  constant low2high: time := 20 ns;
  type alu_op is (add, sub, mul, div, eql);
  attribute pipeline: Boolean;
  type mvl is (‘U’, ‘0’, ‘1’, ‘Z’);
  type mvl_vector is array (natural range <>) of mvl;
  subtype my_alu_op is alu_op range add to div;
  component nand2
    port(a, b: in mvl; c: out mvl);
  end component;
end synthesis_pack;
```

- Items declared in a package declaration can be accessed by other design units by using `library` and `use` clauses.

```
use work.synthesis_pack.all;
```

Yet Another Example

```
use work.synthesis_pack.all; -- include all declarations
                             -- from package synthesis_pack

package program_pack is
  constant prop_delay: time;  -- deferred constant
  function “and” (l, r: mvl) return mvl;
  procedure load (signal array_name: inout mvl_vector;
                  start_bit, stop_bit, int_value: in
                  integer);
end_package program_pack;
```

Package Declaration: `std_logic_1164`

```
package std_logic_1164 is
  type std_ulogic is (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’,
                      ‘L’, ‘H’ , ‘-’);
  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_logic;
  type std_logic_vector is array (natural range <>)
      of std_logic;
  function “and”(l, r: std_logic_vector)
      return std_logic_vector;
  function “and”(l, r: std_ulogic_vector)
      return std_ulogic_vector;
...
end package std_logic_1164;
```

Package Body

- It basically contains the code that implements the subprograms
- Syntax:

```
package body package-name is
  package-body-item-declarations → These are:
  -- subprogram bodies
  -- complete constant declarations
  -- subprogram declarations
  -- type and subtype declarations
  -- file and alias declarations
  -- use clauses
end [package body] [package-name];
```

Package Body: Example

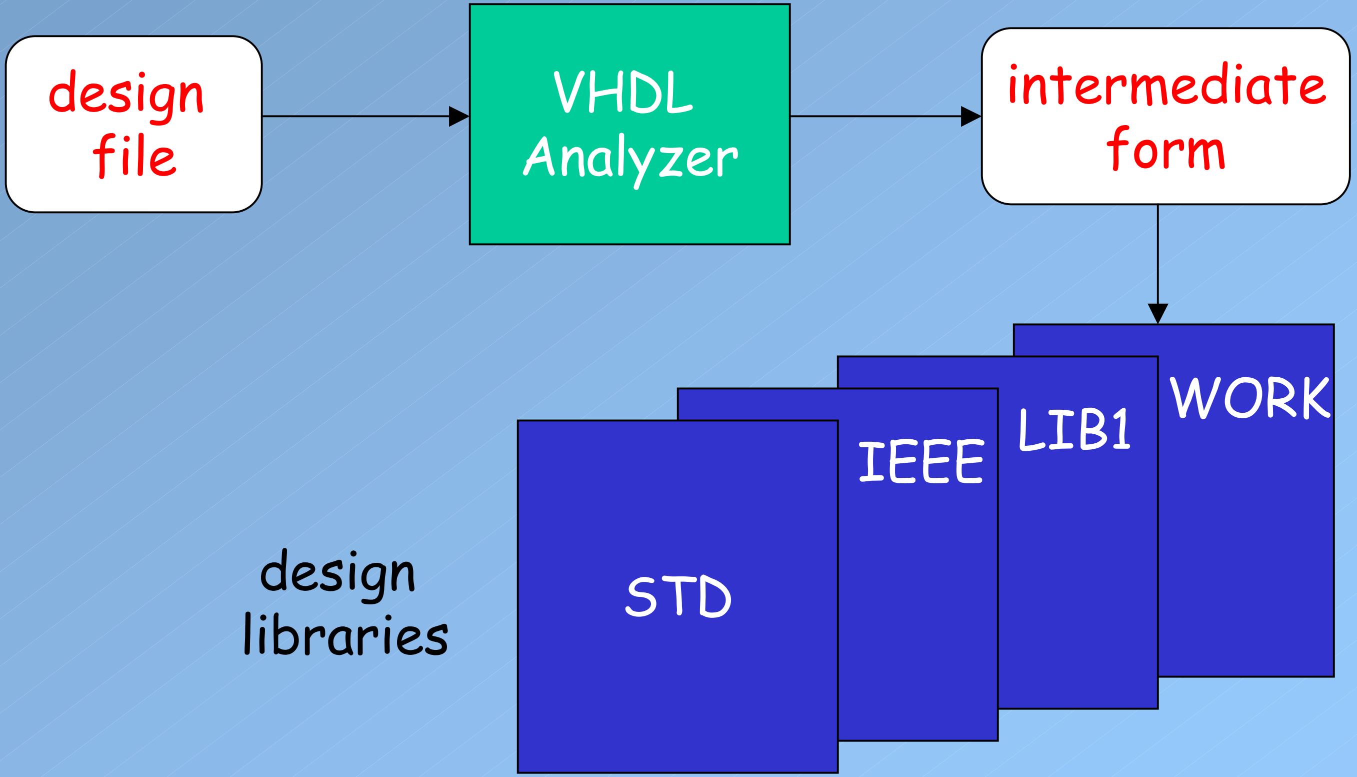
```
package body program_pack is
  use work.tables.all;
  constant prop_delay: time := 15 ns;
  function “and”(l, r: mvl) return mvl;
  begin
    return table_and(l, r);
    -- table_and is a 2-D constant defined in
    -- another package, called “tables” in the
    -- current working directory
  end “and”;
  procedure load(signal array_name: inout mvl_vector;
     start_bit, stop_bit, int_value: in integer) is
  begin
    -- procedure behavior here
  end load;
end program_pack;
```

Libraries

- Design units (design files)
  - entity declaration,
  - architecture body,
  - configuration declaration,
  - package declaration,
  - package body
- Each design unit is analyzed (compiled) and placed in a design library.
  - recall that libraries are generally implemented as directories and are referenced by a logical name.
  - this logical name corresponds to physical path to the corresponding directory.

Compilation Process

- VHDL analyzer verify the syntactic and semantic correctness of the source
- then compiles each design unit into an intermediate form.
- Each intermediate form is stored in a design library called working library.



Libraries: Visibility

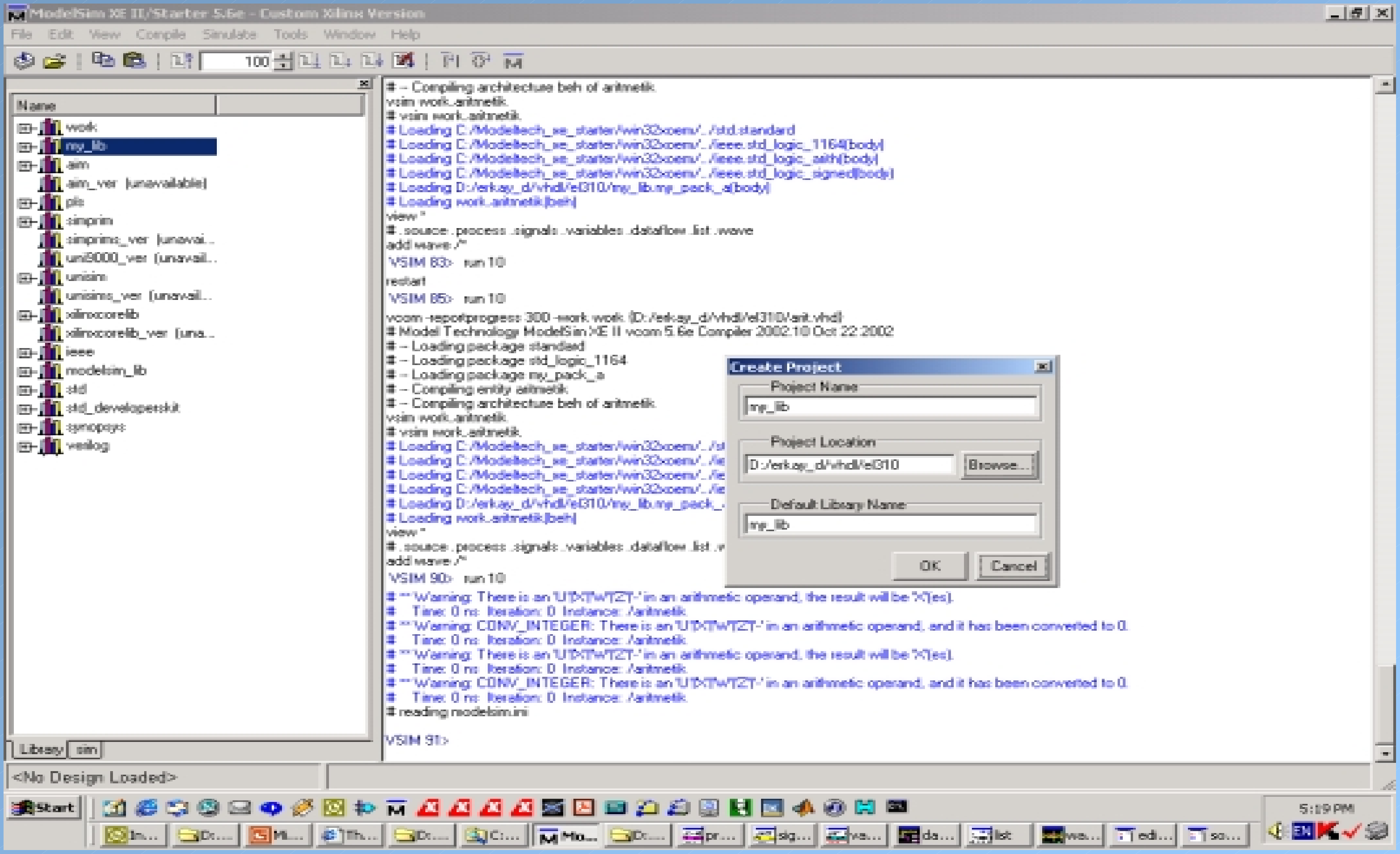
- Implicit visibility
  - In VHDL, the libraries STD and WORK are implicitly declared.
- Explicit visibility is achieved through
  - `library` clause
  - `use` clause
  - Example:

    ```
    library IEEE;
    use IEEE.std_logic_1164.all;
    ```
- Once a library is declared, all of the subprograms, type declarations in this library become visible to our programs through the use of `use` clause.

Context Clauses

- Examples for context clauses:
  - `library IEEE;`
    `use IEEE.std_logic_1164.all;`
- Context clauses only applies the following design entity
  - if a file contains more than one entity, context clauses must precede each of them to provide the appropriate visibility to each design entity.

How to Create a Library?



```
# -- Compiling architecture beh of aritmetik
vsim work.aritmetik
# vsim work.aritmetik
# Loading C:/Modeltech_xe_starter/win32xoem/../std.standard
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_1164(body)
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_arith(body)
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_signed(body)
# Loading D:/erkay_d/vhdl/el310/my_lib.my_pack_a(body)
# Loading work.aritmetik(beh)
view *
# .source .process .signals .variables .dataflow .list .wave
add wave /*
 VSIM 83>  run 10
restart
 VSIM 85>  run 10
vcom -reportprogress 300 -work work {D:/erkay_d/vhdl/el310/arit.vhd}
# Model Technology ModelSim XE II vcom 5.6e Compiler 2002.10 Oct 22 2002
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package my_pack_a
# -- Compiling entity aritmetik
# -- Compiling architecture beh of aritmetik
vsim work.aritmetik
# vsim work.aritmetik
# Loading C:/Modeltech_xe_starter/win32xoem/../st
# Loading C:/Modeltech_xe_starter/win32xoem/../ie
# Loading C:/Modeltech_xe_starter/win32xoem/../ie
# Loading C:/Modeltech_xe_starter/win32xoem/../ie
# Loading D:/erkay_d/vhdl/el310/my_lib.my_pack_
# Loading work.aritmetik(beh)
view *
# .source .process .signals .variables .dataflow .list .w
add wave /*
 VSIM 90>  run 10
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# reading modelsim.ini

VSIM 91>
```

Create Project

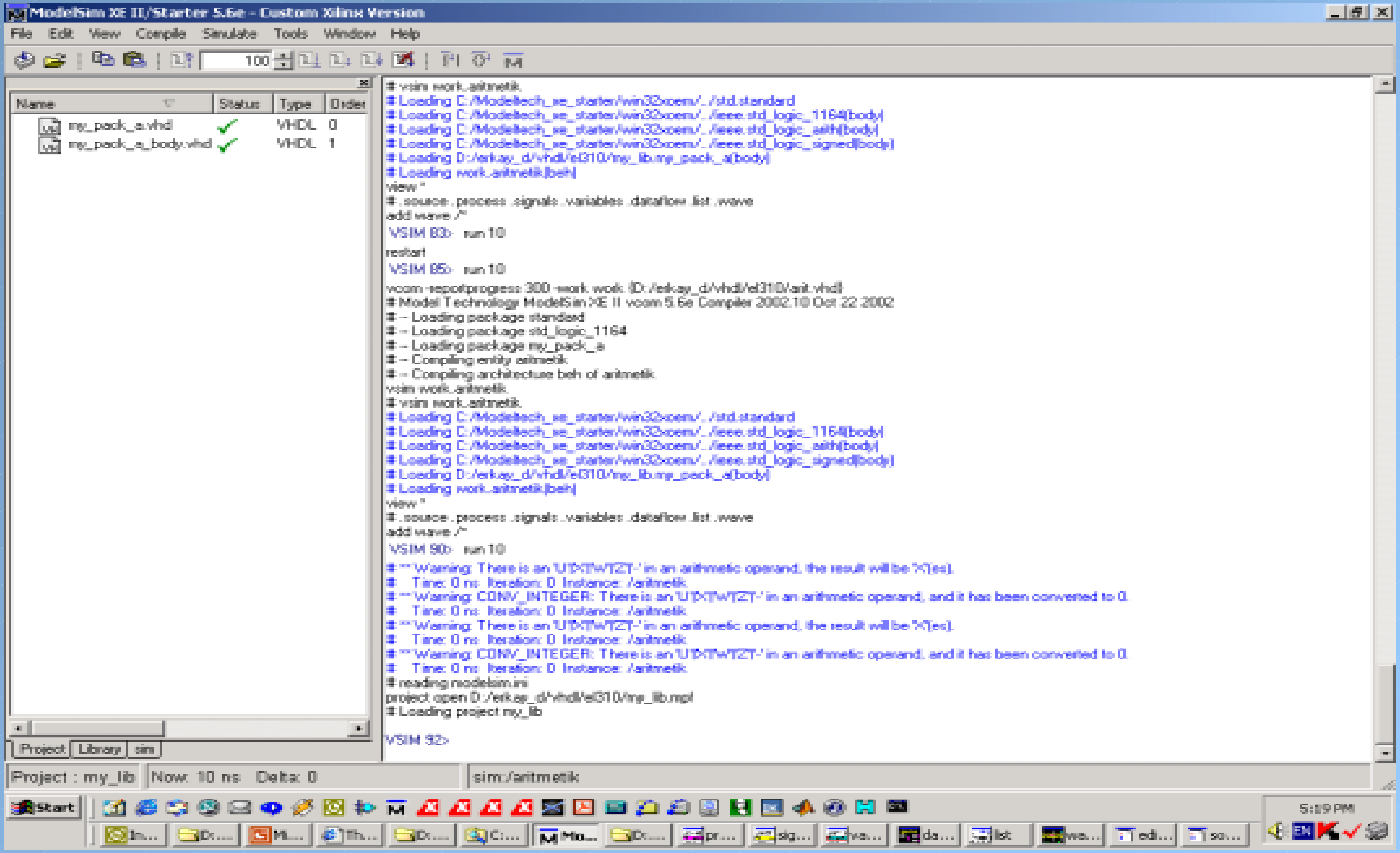
Project Name: my_lib

Project Location: D:/erkay_d/vhdl/el310 [Browse...]

Default Library Name: my_lib

[OK] [Cancel]

How to Add Packages to a Library?


ModelSim XE II/Starter 5.6e - Custom Xilinx Version
File Edit View Compile Simulate Tools Window Help
Project Library sim
Project : my_lib
Now: 10 ns Delta: 0
sim:/aritmetik

Name	Status	Type	Order
my_pack_a.vhd	✓	VHDL	0
my_pack_a_body.vhd	✓	VHDL	1

```
# vsim work.aritmetik
# Loading C:/Modeltech_xe_starter/win32xoem/../std.standard
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_1164(body)
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_arith(body)
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_signed(body)
# Loading D:/erkay_d/vhdl/el310/my_lib.my_pack_a(body)
# Loading work.aritmetik(beh)
view *
# .source .process .signals .variables .dataflow .list .wave
add wave /*
 VSIM 83> run 10
restart
 VSIM 85> run 10
vcom -reportprogress 300 -work work {D:/erkay_d/vhdl/el310/arit.vhd}
# Model Technology ModelSim XE II vcom 5.6e Compiler 2002.10 Oct 22 2002
# -- Loading package standard
# -- Loading package std_logic_1164
# -- Loading package my_pack_a
# -- Compiling entity aritmetik
# -- Compiling architecture beh of aritmetik
vsim work.aritmetik
# vsim work.aritmetik
# Loading C:/Modeltech_xe_starter/win32xoem/../std.standard
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_1164(body)
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_arith(body)
# Loading C:/Modeltech_xe_starter/win32xoem/../ieee.std_logic_signed(body)
# Loading D:/erkay_d/vhdl/el310/my_lib.my_pack_a(body)
# Loading work.aritmetik(beh)
view *
# .source .process .signals .variables .dataflow .list .wave
add wave /*
 VSIM 90> run 10
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# ** Warning: CONV_INTEGER: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, and it has been converted to 0.
#    Time: 0 ns  Iteration: 0  Instance: /aritmetik
# reading modelsim.ini
project open D:/erkay_d/vhdl/el310/my_lib.mpf
# Loading project my_lib

VSIM 92>
```

Package Declaration

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

package my_pack_a is
  subtype word is std_logic_vector(15 downto 0);
  function "+" (op1, op2: word) return word;
  function "-" (op1, op2: word) return word;
  function "*" (op1, op2: word) return word;
end package my_pack_a;
```

Package Body

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;
use IEEE.std_logic_arith.all;
package body my_pack_a is
  function "+" (op1, op2: word) return word is
    variable result: word;
    variable a, b, c: integer;
  begin
    a := conv_integer(op1);
    b := conv_integer(op2);
    c := a + b;
    result := conv_std_logic_vector(c, 16);
    return result;
  end function;
...
end package body my_pack_a;
```

Package Body

```
...
  function "-" (op1, op2: word) return word is
    variable result: word; variable a, b, c: integer;
  begin
    a := conv_integer(op1); b := conv_integer(op2);
    c := a - b;
    result := conv_std_logic_vector(c, 16);
    return result;
  end function;
  function "*" (op1, op2: word) return word is
    variable result: word; variable a, b, c: integer;
  begin
    a := conv_integer(op1);
    b := conv_integer(op2);
    c := a * b;
    result := conv_std_logic_vector(c, 16);
    return result;
  end function;
  end package body my_pack_a;
```

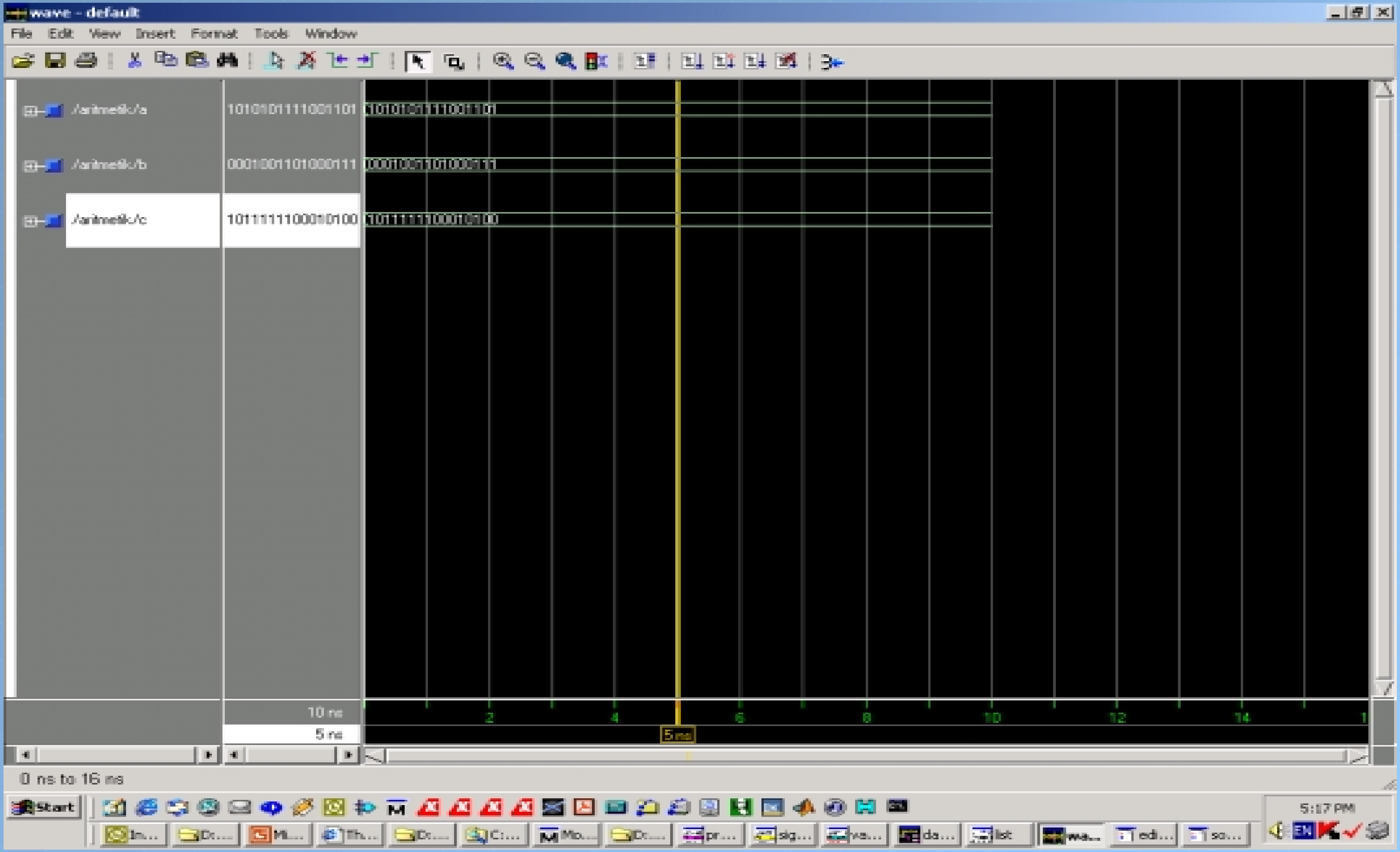
Example using this Package

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

entity aritmetik is
end entity;

architecture beh of aritmetik is
  signal a, b, c: word;
begin
 a <= x"abcd";
 b <= x"1347";
 c <= a + b;
end architecture beh;
```

Simulation Results



/aritmetik/a	1010101111001101	1010101111001101
/aritmetik/b	0001001101000111	0001001101000111
/aritmetik/c	1011111100010100	1011111100010100

Summary

- Hierarchy
- Functions
- Procedures
- subprogram overloading
- operator overloading
- packages
- libraries