

# VHDL Extended Identifiers

**VHDL-93 syntax:**

**SIGNAL** \A and B\ : std\_logic;

\A and B\ <= a AND b;

**COMPONENT** \74S405\ **IS**  
    **PORT** ( \rd#\ : IN std\_ulogic;  
    .....  
**END COMPONENT** ;

Very Important for VHDL & Verilog Co-Simulation

Presented by Abramov B.  
All right reserved

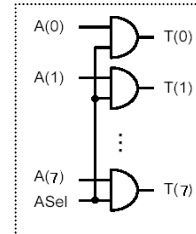
2

# VHDL 2006 useful features

The new standard of VHDL supports asymmetric and reduced bit operations:

```
signal a, t : std_logic_vector(7 downto 0);
signal asel : std_logic;
```

```
t <= a and asel; -- legal
t <= asel and a; -- legal
```



Example of new options – bitwise operations:

```
signal d : std_logic_vector(7 downto 0);
```

```
-- reduce and
c <= and d;
```



```
-- reduce or
c <= or d;
```



Presented by Abramov B.  
All right reserved

3

## Variables

- Variables are used for information storage and pass within process, function and procedure.
- Variable is given its value immediately, not at the delta end like signal.
- Assignments may be made from signals to variables and vice-versa, provided the types match.

```
process
  variable name : type [:= initial_value];
begin
  :
  :
end process;
```

Presented by Abramov B.  
All right reserved

4

# Variables Safe Usage

```
process(a,b) is
  variable temp : std_logic_vector (1 downto 0);
begin
  temp := a & b;
  case temp is
    when "00" => .....
    when "01" => .....
    when "10" => .....
    :
  end case;
end process;
```

Presented by Abramov B.  
All right reserved

5

# Variables Safe Usage

```
process(addr) is
  variable temp_addr : integer range 255
  downto 0;
begin
  temp_addr := conv_integer(addr);
  case temp_addr is
    :
  end case;
end process;
```

Presented by Abramov B.  
All right reserved

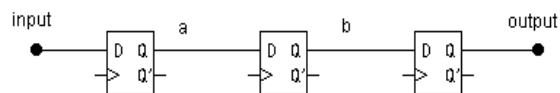
6

# Clocked Process with Variables

Complete the electronic schematic for each of the following processes:

```

signal a,b : std_logic;
----
process(clk) is
begin
if rising_edge (clk) then
    a <= input;
    b <= a;
    output <= b;
end if;
end process;
    
```



Presented by Abramov B.  
All right reserved

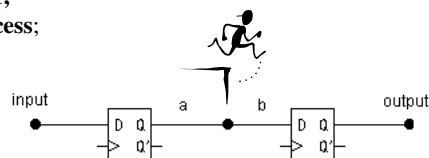
7

# Clocked Process with Variables

Complete the electronic schematic for each of the following processes:

```

signal a : std_logic;
process(clk) is
    variable b : std_logic;
begin
    if rising_edge (clk) then
        a <= input;
        b := a;
        output <= b;
    end if;
end process;
    
```



Presented by Abramov B.  
All right reserved

8

# Logic thought Variables

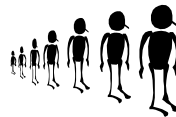
```
entity mul is
port(
    a : in std_logic_vector(3 downto 0);
    b : in std_logic_vector(3 downto 0);
    res : out std_logic_vector(7 downto 0)
);
end entity mul;
architecture arc_mul of mul is
begin
    process(a,b) is
        variable level_0,level_1 : std_logic_vector(7 downto 0);
        variable level_2,level_3 : std_logic_vector(7 downto 0);
        variable sum_01,sum_23 : std_logic_vector(7 downto 0);
    begin
```

Presented by Abramov B.  
All right reserved

9

## Logic thought Variables (cont)

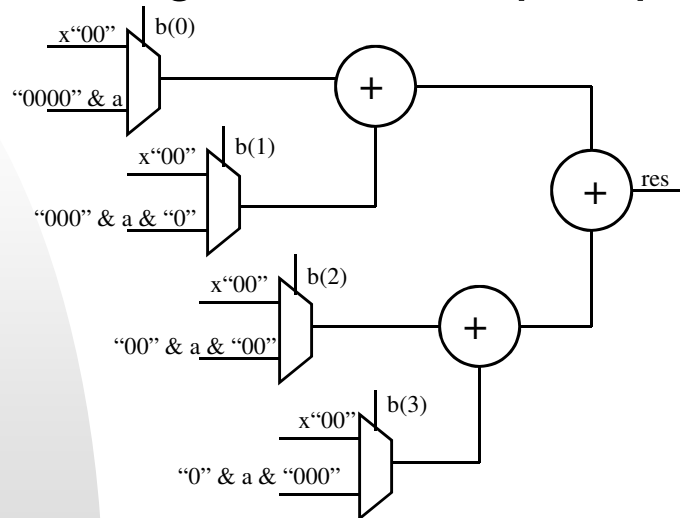
```
    if (b(0)='1') then
        level_0:= "0000" & a;
    else level_0:=(others=>'0');
    end if;
    if (b(1)='1') then
        level_1:= "000" & a & "0";
    else level_1:=(others=>'0');
    end if;
    if (b(2)='1') then
        level_2:= "00" & a & "00";
    else level_2:=(others=>'0');
    end if;
    if (b(3)='1') then
        level_3:= "0" & a & "000";
    else level_3:=(others=>'0');
    end if;
    sum_01:= level_0 + level_1; sum_23:= level_2 + level_3; res<=sum_01 + sum_23;
end process;
end architecture arc_mul;
```



Presented by Abramov B.  
All right reserved

10

## Logic thought Variables (cont)



Presented by Abramov B.  
All right reserved

11

## Variables Unsafe Usage

If a value of a variable is read before it is assigned in a clocked process  
(i.e. where operations are performed when an edge of clock signal is detected)  
then a register will be synthesized for this variable.  
A similar situation inside a combinational process may lead to generation of a latch.

```
process( clk) is
    variable q : std_logic;
begin
    if rising_edge(clk) then
        temp<= d and (not q);
        q:= d; -- temp and q act as registers
    end if;
end process;
```



Presented by Abramov B.  
All right reserved

12

# Variables Safe Usage

```
process( clk, rst) is
begin
  if (rst=active) then
    reg<=(others=>'0');
    temp<=(others=>'0');
  elsif rising_edge(clk) then
    case some_sig is
      when cond_A => temp<=.....
      when cond_B => temp<=.....
      .....
      when others => temp<= .....
    end case;
    reg<=temp + y;
  end if;
end process;
```

Signal temp  
represents  
register!!!

Presented by Abramov B.  
All right reserved

13

# Variables (cont)

```
process( clk, rst) is
  variable temp : std_logic_vector(7 downto 0);
begin
  if (rst=active) then
    reg<=(others=>'0');
  elsif rising_edge(clk) then
    case some_sig is
      when cond_A => temp:=.....
      when cond_B => temp:=.....
      .....
      when others => temp:= .....
    end case;
    reg<=temp + y;
  end if;
end process;
```

**temp** don't have reset!  
Case should be with all  
possible branches

Presented by Abramov B.  
All right reserved

14

# Variables read contradiction?

```

process (clk) is
  variable temp : some type;
begin
  if rising_edge(clk) then
    if (cond1) then
      temp:=a;
      .....
    if (cond2) then
      .....
      sig1<=temp;
      .....
    end if;
  end process;

```

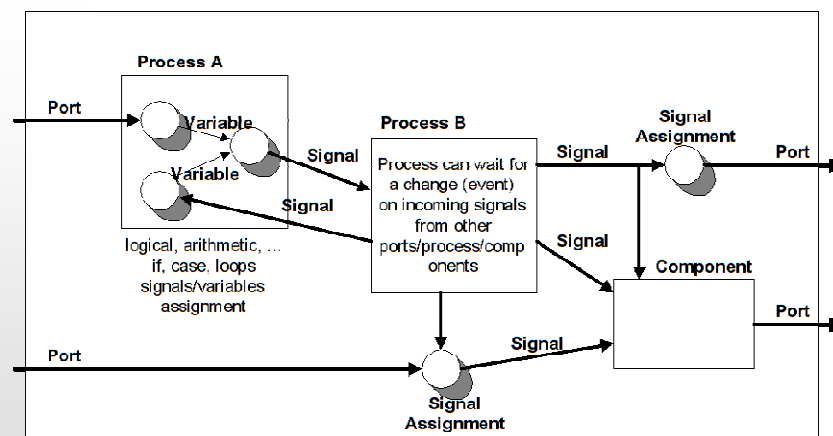
Parallel IF.  
cond1=cond2=true  
both with clock  
rise

Presented by Abramov B.  
All right reserved

15

# Variables Scope

## Ports / Signals / Variables Usage



Presented by Abramov B.  
All right reserved

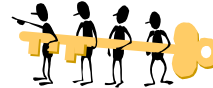
16



# Attributes

Attributes supply additional information an item:

- signal
- variable
- type
- component



Certain attributes are predefined for types, array objects and signals.

Syntax:

object'attribute name;

There are a number of attributes that are applicable to all scalar types and provide information about the range of values in the type.

Presented by Abramov B.  
All right reserved

17

## Attributes (cont)

**Scalar types attributes:**

- T'**left** - first (leftmost) value in T;
- T'**right** - last (rightmost) value in T;
- T'**low** - least value in T;
- T'**high** - greatest value in T;
- T'**ascending** - return boolean value, true if T is ascending range , false otherwise.
- T'**image**(x) - represent a scalar value of x into string form
- T'**value**(s) - return the scalar value of object s are represented by string form



Presented by Abramov B.  
All right reserved

18

## Attributes (cont)

There are attributes that applicable to just discrete and physical types.

For any such type T, a value x of that type and an integer n, the attributes are:

- T'**pos**(x) - position number of x in T
- T'**val**(n) - value in T at position n
- T'**succ**(x) - value in T at position one greater than that of x
- T'**pred**(x) - value in T at position one less than that of x
- T'**leftof**(x) - value in T at position one to the left of x
- T'**rightof**(x) - value in T at position one to the right of x

Presented by Abramov B.  
All right reserved

19

## Attributes (cont)

```
type my_type1 is ( unknown, low, medium, high);
type my_type2 is range 15 downto 0;
signal type1 : my_type1;
signal type2 : my_type2;
signal flag : boolean;
signal position : integer;
type1<= my_type1'left ; -- unknown
type2<= my_type2'left; -- 15
type1<= my_type1'right; -- high
type2<=my_type2'right; -- 0
type1<=my_type1'low; -- unknown
type2<=my_type2'low; -- 0
type1<= my_type1'high ;-- high
type2<=my_type2'high;-- 15
```

Presented by Abramov B.  
All right reserved

20

## Attributes (cont)

```
flag<=my_type1'ascending ; -- true;
flag<= my_type2'ascending; -- false
my_type1'image(low); -- "low"
type2<= my_type2'value("12"); -- 12
position<= my_type1'pos(low); -- 1
type2<= my_type2'val(4); -- 11
type2<=my_type2'succ(14);
-- the position of number 14 is 1, then returned result
   is number from position 2 -> 13
```

Presented by Abramov B.  
All right reserved

21

## Attributes (cont)

```
type1<=my_type1'succ(unknown); -- low
type2<=my_type1'pred(14) ; -- 15
type1<=my_type1'pred(medium); --low
type1<=my_type1'leftof(medium); -- low
type1<=my_type1'rightof(medium); --high
type2<=my_type2'leftof(10); -- 11
type2<=my_type2'rightof(10); --9
```

Presented by Abramov B.  
All right reserved

22

## Attributes (cont)

```
signal b : boolean;  
signal a : bit;  
signal i : integer;  
signal sl : std_logic;
```



```
a<=bit'val(boolean'pos(b));  
b<=boolean'val(bit'pos(a));  
i<=boolean'pos(b);  
i<=bit'pos(a);  
sl<=std_logic'val(boolean'pos(b) + std_logic'pos('0'));
```

Presented by Abramov B.  
All right reserved

23

## Attributes (cont)

### The array attributes:

- A'left - left bound of index range
- A'right - right bound of index range
- A'low - lower bound of index range
- A'high - upper bound of index range
- A'range - Index range
- A'reverse\_range - reverse of index range
- A'length - length of index range
- A'ascending - true if index range is an ascending,  
false otherwise

Presented by Abramov B.  
All right reserved

24

## Attributes (cont)

```
my_port : in std_logic_vector((N-1) downto 0);
signal my_sig : std_logic_vector(my_port'range); -- 7 downto 0
signal temp : std_logic_vector( (my_port'length-1) downto 0); -- 7 downto 0
signal rev_sig : std_logic_vector (my_port'reverse_range); -- 0 to 7
signal my_int : integer range 15 downto 0;
signal flag : boolean;
my_int<=my_sig'left; -- 7
my_int<=my_sig'low; -- 0
my_int<=my_sig'righ; -- 0
my_int<=my_sig'high; --7
flag<= my_sig'ascending; --false
flag<= rev_sig'ascending; -- true
my_sig<= x"aa";
rev_sig<=my_sig; -- x"55"
```



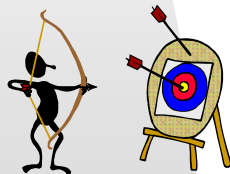
Presented by Abramov B.  
All right reserved

25

## Attributes (cont)

User defined attributes:

```
signal my_sig : std_logic_vector(8 downto 0);
attribute middle : integer;
attribute middle of my_sig: signal is my_sig'length/2; -- 4
signal target : std_logic_vector(my_sig'middle downto 0);
```



Presented by Abramov B.  
All right reserved

26

# Attributes (cont)

## Synthesis attributes

### Noopt:

**noopt** <instance name> <true or false>

Specifies that an instance should not be optimized or changed.

However, in contrast to don't\_touch, lower level hierarchy and leaf instances are not protected from optimization or change.

For example:

**attribute** noopt : **boolean**;

**attribute** noopt of <component\_name> : **component** is TRUE;

Presented by Abramov B.  
All right reserved

27

# Attributes (cont)

### Pin\_number:

**pin\_number** <pin number> <port name>

Assigns a device pin number to a certain port.

Note: Pin location corresponds to pin\_number attribute.

Syntax:

**attribute** pin\_number : **string**;

**attribute** pin\_number of din : **signal** is "P10";

Presented by Abramov B.  
All right reserved

28

## Attributes (cont)

```
entity attribute_example is
port (
  CLK_30M   : in  std_logic;
  PRESET    : in  std_logic;
  CLK_DIS   : in  std_logic;
  ID        : in  std_logic_vector (2 downto 0);
  Y_LED     : out std_logic;
  attribute altera_chip_pin_lc : string;
  attribute altera_chip_pin_lc of ID      : signal is "N3, L4, R1";
  attribute altera_chip_pin_lc of CLK_30M : signal is "G1";
  attribute altera_chip_pin_lc of CLK_DIS : signal is "M4";
  attribute altera_chip_pin_lc of PRESET  : signal is "E5";
  attribute altera_chip_pin_lc of Y_LED   : signal is "P2";
end entity attribute_example ;
```

Presented by Abramov B.  
All right reserved

29

## Attributes (cont)

### Pullup/pulldown:

Assign pullup or pulldown resistors (if possible) to your ports.

Syntax:

```
attribute pull:string;
attribute pull of inbus_a : signal is "pullup";
attribute pull of inbus_b : signal is "pulldn";
```

Presented by Abramov B.  
All right reserved

30

## Attributes (cont)

### Open drain:

Before using OPEN\_DRAIN, declare it with the following syntax:

**attribute OPEN\_DRAIN: *string*;**

After OPEN\_DRAIN has been declared, specify the VHDL constraint as follows:

**attribute OPEN\_DRAIN of *signal\_name* : signal is “TRUE”;**

Presented by Abramov B.  
All right reserved

31

## Attributes (cont)

### Preserve\_signal

When you apply preserve\_signal, synthesis tool preserves the specified signal and the driver in the design.

**preserve\_signal <signal name>**

Specifies that both a signal and the signal name must survive optimization.

Any parallel logic, such as a parallel inverters (gates), are optimized to a single instance. The attribute preserve\_signal can be applied on the parallel signals to tell synthesis tool to maintain the parallel structure.

**attribute preserve\_signal : *boolean*;**

**attribute preserve\_signal of nz1:signal is true ;**

**attribute preserve\_signal of nz2:signal is true ;**



Presented by Abramov B.  
All right reserved

32



## Attributes (cont)

### Clock\_cycle

clock\_cycle <clock period> <signal name>

Specifies the length (nanoseconds, real numbers) of the clock.

This is a clock control command.

**Note:** clock\_cycle is one of two basic clock commands.

The other : pulse\_width.

#### Syntax:

**attribute** clock\_cycle : **real**; -- or time type

**attribute** clock\_cycle **of** in\_clock:signal **is** 30.0;

Presented by Abramov B.  
All right reserved

33

## Attributes (cont)

### Pulse\_width

pulse\_width <clock width>

Specifies the width (nanoseconds) of the clock pulse.

This is a clock control command for duty\_cycle description.

#### Syntax:

**attribute** pulse\_width : **time**; -- or real type

**attribute** pulse\_width **of** clock:signal **is** 10 ns;

Presented by Abramov B.  
All right reserved

34

## Attributes (cont)

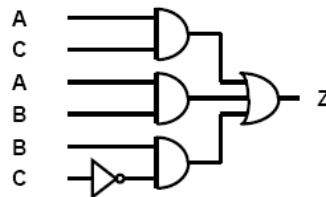
### Problem



### • Code

```
Z <=
  (A and C) or
  (A and B) or
  (B and not C) ;
```

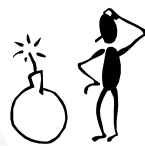
### • Required Circuit:



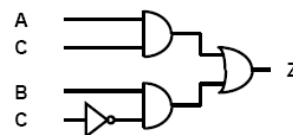
Presented by Abramov B.  
All right reserved

35

## Attributes (cont)



### • Resulting Circuit:



### • What happened?

- The term AB is reducible and was removed by the synthesis tool.

### ✦ Is it necessary to keep AB?

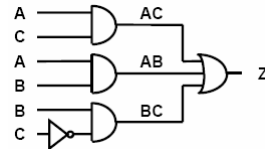
- ✦ For asynchronous logic, the AB term prevents glitches

Presented by Abramov B.  
All right reserved

36

## Attributes (cont)

Step 1 : Create intermediate signals



```
signal AC, AB, BC : signal;  
.  
.  
.  
AC <= A and C ;  
AB <= A and B ;  
BC <= B and not C ;  
  
Z <= AC or AB or BC ;
```

Presented by Abramov B.  
All right reserved

37

## Attributes (cont)

**attribute** keep : **boolean**;

**attribute** keep of signal\_name : signal is **true**;

- ✦ Step 2: Apply the keep attribute to a AC, AB, BC to preserve them through synthesis. This is equivalent to inserting a non-movable buffer on the signal.

```
attribute KEEP of AC, AB, BC : signal is TRUE;
```



Presented by Abramov B.  
All right reserved

38

## Attributes (cont)

### Attributes of signal:

Given a signal S, and a value T of type time, VHDL defines the following attributes:

- S'delayed(T)** - created the new signal of type S, delayed from S by T.
- S'stable(T)** - return Boolean value, true if there has been no event on s in the time interval T up to the current time, false otherwise.
- S'quiet(T)** - return Boolean value, true if there has been no transaction on s in the time interval T up to the current time, false otherwise.
- S'transaction** - return value of type bit, that changed value from '0' to '1' or vice-versa each time there is a transaction on S.
- S'event** - true if there is a event on S in the current cycle, false otherwise.
- S'active** - true if there is a transaction on S in the current cycle, false –otherwise.
- S'last\_event** - the time interval since the last event on S.
- S'last\_active** - the time interval since the last transaction on S.
- S'last\_value** - the value of S just before the last event on S.

Presented by Abramov B.  
All right reserved

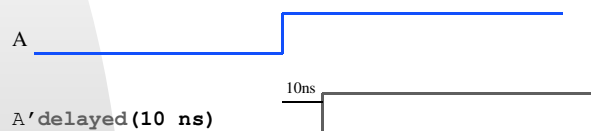
39

## Attributes (cont)

Generate a new signal delayed from the source signal.

**A'delayed**

**A'delayed(10 ns)**



Presented by Abramov B.  
All right reserved

40

## Attributes (cont)

✓ The clock Rising edge detecting.

`clk'event and clk='1';`

`not clk'stable and clk='1';`

`clk'event and clk'last_value='0';`

`not clk'stable and clk'last_value='0';`

✓ The clock falling edge detecting.

`clk'event and clk='0';`

`not clk'stable and clk='0';`

`clk'event and clk'last_value='1';`

`not clk'stable and clk'last_value='1';`

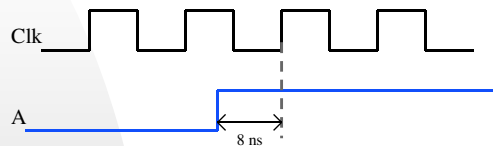


Presented by Abramov B.  
All right reserved

41

## Attributes (cont)

For Example (Set Up Violation Check):



**WAIT UNTIL** rising\_edge(Clk);

**ASSERT** (A'Last\_event >= 8 ns)

**REPORT** " SetUp Violation on Signal A ";



Presented by Abramov B.  
All right reserved

42

## Attributes (cont)

Setup time violation checker:

```
constant setup_time : time:= 2 ns;  
if clk'event and clk='1' then  
    if not d'stable( setup_time) then  
        report “ Setup time violation on  
        signal d”;  
    end if;  
end if;
```



Presented by Abramov B.  
All right reserved

43

## Attributes (cont)

Hold time violation checker:

```
constant hold_time : time:= 2 ns;  
if d'event then  
    if not clk'stable( hold_time) and clk'last_value= '0'  
    then  
        report “ Hold time violation on signal d”;  
    end if;  
end if;  
  
if clk'delayed(hold_time)'event and clk='1' then  
    if not d'stable( hold_time) then  
        report “ Hold time violation on signal d”;  
    end if;  
end if;
```

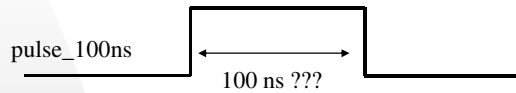


Presented by Abramov B.  
All right reserved

44

## Attributes (cont)

Pulse width measurement with attributes



```
wait until falling_edge(pulse_100ns);
if (pulse_100ns'delayed'last_event < 100 ns) then
    report "pulse_100ns length is less than
           expected time";
end if;
```

Presented by Abramov B.  
All right reserved

45

```
-- synthesis off
process is
    variable temp : std_logic_vector(cnt'range);
    variable ones_counter : integer:=0;
begin
    wait until rising_edge(clock);
    wait on cnt;
    temp:=cnt xor cnt'delayed'last_value;
    for i in temp'range loop
        ones_counter:= ones_counter + conv_integer(temp(i));
    end loop;
    assert (ones_counter=1 or now=0 or reset=reset_active)
        report "Counter error detected " & time'image(now) severity Warning;
end process;
-- synthesis on
```

XOR between  
current and previous  
values of Gray code  
counter should be  
one-hot value

Presented by Abramov B.  
All right reserved

46

# Loops



VHDL has a basic **loop** statement, which can be augmented to form the usual while and for loops seen in other programming languages. The loop statement contains a sequence of statements, which are supposed to be repeated many times. The statement also lists the conditions for repeating the sequence or specifies the number of iterations. A loop statement can have several different forms depending on the iteration scheme preceding the reserved word **loop**.

Presented by Abramov B.  
All right reserved

47

## Loops (cont)

In its simplest form, no iteration scheme is specified and the loop is repeated indefinitely.

Example :

```
signal clock : std_logic := '0';  
clk_gen: process is  
begin  
  L1: loop  
    wait for 5 ns;  
    clock <= '0';  
    wait for 5 ns;  
    clock <= '1';  
  end loop L1;  
end process clk_gen;
```

Presented by Abramov B.  
All right reserved

48



## Loops (cont)

In order to exit from an infinite loop, an **exit** statement has to be used. The **exit** statement is used to finish or exit the execution of an enclosing loop statement.

If the **exit** statement includes a condition, then the exit from the loop is conditional.

Syntax:

```
exit;  
exit loop_label;  
exit loop_label when condition;
```

Presented by Abramov B.  
All right reserved

49

## Loops (cont)

The **exit** statement terminates entirely the execution of the loop in which it is located.

The execution of the exit statement depends on a condition placed at the end of the statement, right after the when reserved word.

When the condition is **TRUE** (or if there is no condition at all) the **exit** statement is executed and the control is passed to the first statement after the end loop.



Presented by Abramov B.  
All right reserved

50

## Loops (cont)

The *loop label* in the **exit** statement is not obligatory and can be used only in case of labeled loops.

If no label is present then it is assumed that the exit statement relates to the innermost loop containing it.

If an exit from a loop on a higher level of hierarchy is needed then the loop has to be assigned a label, which will be used explicitly in the exit statement.



Presented by Abramov B.  
All right reserved

51

## Loops (cont)

```
signal a : integer:=0;
```

```
L2: loop
```

```
  a<= a+1;
```

```
  wait until rising_edge(clk);
```

```
  exit L2 when (a > 10);
```

```
end loop L2;
```

The infinite loop becomes in practice a finite, as the iterations will terminate as soon as the variable A becomes greater than 10.



Presented by Abramov B.  
All right reserved

52

## Loops (cont)

Instead of specifying an infinite loop with a conditional **exit** statement, a **while** loop can be used. In such a case the reserved word **while** with a condition precede the keyword **loop**.

The sequence of statements inside the **loop** will be executed if the condition of the iteration scheme is *true*.

The condition is evaluated before each execution of the sequence of statements.

When the condition is false, the loop is not entered and the control is passed to the **next** statement after the **end loop** clause

Presented by Abramov B.  
All right reserved

53

## Loops (cont)

```
shifter: process is
  variable i : positive := 1;
  begin
    L3: while i < 8 loop
      serial_out <= reg(i) ;
      i := i + 1;
      wait until rising_edge(clk);
    end loop L3;
  end process shifter;
```



Presented by Abramov B.  
All right reserved

54

## Loops (cont)

Another iteration scheme is useful when a discrete range can define the number of iterations.

In this case the keyword **for** with a **loop** parameter precede the keyword **loop**.

The header of the **loop** also specifies the discrete **range** for the loop parameter.

In each iteration the parameter takes one value from the specified range, starting from the leftmost value within the range.

Presented by Abramov B.  
All right reserved

55

## Loops (cont)

shifter: **process** is

**begin**

L4: **for** index **in** 0 **to** 7 **loop**

serial\_out <= din(index);

**wait until** rising\_edge(clk);

**end loop** L4;

**end process** shifter;

In the above example the loop statement parameter index will cause the loop to execute 8 times, with the value of index changing from 0 to 7.

Presented by Abramov B.  
All right reserved

56

## Loops (cont)

The **next** statement is used to complete execution of one of the iterations of an enclosing **loop** statement.

The completion is conditional if the statement includes a condition.

Syntax:

```
next;  
next loop_label;  
next loop_label when condition;
```



Presented by Abramov B.  
All right reserved

57

## Loops (cont)

The **next** statement allows to skip a part of an iteration loop.

If the condition specified after the **when** reserved word is **TRUE**, or if there is no condition at all, then the statement is executed.

This results in skipping all statements below it until the end of the **loop** and passing the control to the first statement in the **next** iteration.

```
Loop_Z: for index in 0 to 15 loop  
        next when((index rem 2) = 1);  
        bus_b(index/2) <= bus_a(index);  
end loop Loop_Z;
```

Presented by Abramov B.  
All right reserved

58

## Loops (cont)

Important notes:

- The **next** statement is often confused with the **exit** statement. The difference between the two is that the **exit** statement "exits" the loop entirely, while the **next** statement skips to the "next" loop iteration (in other words, it "exits" the current iteration of the loop).
- The parameter for a 'for' loop does not need to be specified – the loop declaration implicitly declares it.
- The **loop** parameter is a constant within a **loop**, which means that it may not be assigned any values inside the **loop**.

Presented by Abramov B.  
All right reserved

59

## Loops (cont)

The difference between next and exit

```
for I in 0 to max_no loop
  if (done(I)=true) then
    next;
  else
    Done(I):=true;
  end if;
  x(I) <= y(I) and z(I);
end loop;
```

```
for I in 0 to max_no loop
  if (done(I)=true) then
    exit;
  else
    Done(I):=true;
  end if;
  x(I) <= y(I) and z(I);
end loop;
```

Presented by Abramov B.  
All right reserved

60

## Loops (cont)

```

shifter: process(clk,rst) is
begin
  if (rst='0') then
    reg<=(others=>'0');
  elsif rising_edge(clk) then
    if (ld='1') then
      reg<=din;
    elsif (en='1') then
      for i in reg'low to (reg'high-1) loop
        reg(i+1)<=reg(i);
        -- reg(i) <='0';
      end loop;
    end if;
  end if;
end process shifter;

```

Shift left or shift right ?

What a difference within/out of commented line?



Presented by Abramov B.  
All right reserved

61

## Loops (cont)

```

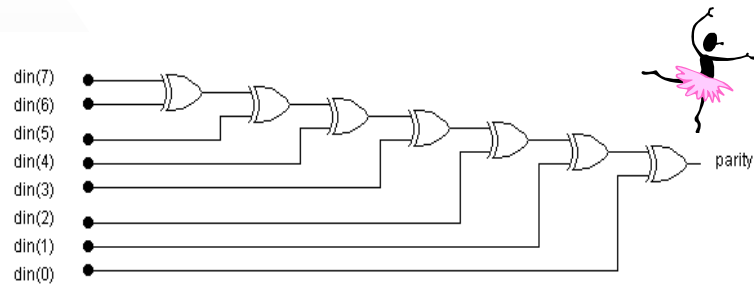
signal din : std_logic_vector(7 downto 0);
parity_check: process(din) is
variable p_even : std_logic;
begin
  for i in din'range loop
    if (i=din'left) then
      p_even:=din(din'left);
    else
      p_even:=p_even xor din(i);
    end if;
  end loop;
  parity<=p_even;
end process parity_check;

```

Presented by Abramov B.  
All right reserved

62

## Loops (cont)



Presented by Abramov B.  
All right reserved

63

## Loops (cont)

Loop is a very powerful instrument for description of similar operations or repeated structures.  
Example: *priority encoder implementation with If-else statement.*

```

process (din) is
  begin
    if (din(15)='1') then
      dout<="1111";
    elsif (din(14)='1') then
      dout<="1110";
    elsif (din(13)='1') then
      dout<="1101";
    .....
    else
      dout<="0000";
    end if;
  end process;
  
```

Current code style is very simple but should be used only for small encoders.

Presented by Abramov B.  
All right reserved

64



## Loops (cont)

*Priority encoder implementation with for loop and exit statements.*

**process** (din) **is**

**begin**

dout<=(others=>'0');

**for** i **in** din'**range** **loop**

dout<=conv\_std\_logic\_vector(i,dout'**length**);

**exit when** (din(i)='1');

**end loop**;

**end process**;

The given description is very short but nondeterministic for hardware compilers. The next description is better for synthesis.

Presented by Abramov B.  
All right reserved

65

## Loops (cont)

*Priority encoder implementation with for loop statement.*

**process** (din) **is**

**variable** first\_one : **boolean**;

**begin**

first\_one:= **false**;

dout<=(others=>'0');

**for** i **in** din'**range** **loop**

**if** ((**not** first\_one) **and** (din(i)='1')) **then**

first\_one:=**true**;

dout<=conv\_std\_logic\_vector(i,dout'**length**);

**end if**;

**end loop**;

**end process**;

Flag that indicates  
occurrence of the first bit  
which equals 1

The given description is completely generic.

Presented by Abramov B.  
All right reserved

66

## Loops (cont)

```

twos_complement: process (a) is
    variable temp : std_logic_vector(a'range);
    variable carry : std_logic;
begin
    carry := '1';
    for i in a'reverse_range loop
        temp(i) := (not a(i)) xor carry;
        carry := (not a(i)) and carry;
    end loop;
    b <= temp;
end process twos_complement;

```

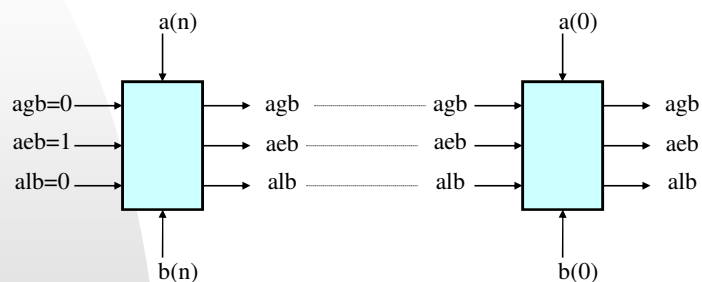


Presented by Abramov B.  
All right reserved

67

## Loops & Attributes Examples

Ripple comparator



Presented by Abramov B.  
All right reserved

68

# Loops & Attributes

```
process (a,b) is
    variable e,g,l : std_logic;
    begin
        (e,g,l):=std_logic_vector("4");
        for i in a'range loop
            g:=g or (a(i) and (not b(i)) and e);
            l:=l or (b(i) and (not a(i)) and e);
            e:=(a(i) xnor b(i)) and e;
        end loop;
        (eq,gr,ls)<=std_logic_vector(e & g & l);
    end process;
```

Presented by Abramov B.  
All right reserved

69

# Loops & Attributes

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity pipelined_mul is
generic ( din_width : natural:=16);
port(    clk : in std_logic;
        rst : in std_logic;
        a  : in std_logic_vector((din_width-1) downto 0);
        b  : in std_logic_vector((din_width-1) downto 0);
        res : out std_logic_vector((din_width*2 -1) downto 0));
end entity pipelined_mul;
```

Presented by Abramov B.  
All right reserved

70

# Loops & Attributes

```
architecture arc_pipelined_mul of pipelined_mul is
  type registers is array (natural range <>) of std_logic_vector(res'range);
  signal products : registers((din_width - 1) downto 0);
  signal products_p : registers((din_width - 1) downto 0);
  signal stage_0 : registers((din_width/2 - 1) downto 0);
  signal stage_1 : registers((din_width/4 - 1) downto 0);
  signal stage_2 : registers((din_width/8 - 1) downto 0);
begin
```

Presented by Abramov B.  
All right reserved

71

# Loops & Attributes

```
product_select:process(a,b) is
begin
  for i in b'reverse_range loop
    products(i)<=(others=>'0');
    if(b(i)='1') then
      products(i)(((a'length-1) + i) downto i)<=a;
    end if;
  end loop;
end process product_select;
```

Presented by Abramov B.  
All right reserved

72

# Loops & Attributes

```
adders:process(clk,rst) is
begin
    if (rst='1') then
        for f in 0 to (products'length - 1) loop
            products_p(f)<=(others =>'0');
        end loop;
        for j in 0 to (products_p'length/2 - 1) loop
            stage_0(j)<=(others =>'0');
        end loop;
```

Presented by Abramov B.  
All right reserved

73

# Loops & Attributes

```
        for k in 0 to (stage_0'length/2 - 1) loop
            stage_1(k)<=(others =>'0');
        end loop;
        for n in 0 to (stage_1'length/2 - 1) loop
            stage_2(n)<=(others =>'0');
        end loop;
        res<=(others=>'0');
```

Presented by Abramov B.  
All right reserved

74

# Loops & Attributes

```
elsif rising_edge(clk) then
    mux_pipeline:for g in 0 to (products'length - 1) loop
        products_p(g)<=products(g);
    end loop mux_pipeline;
    first_adders:for m in 0 to (products'length/2 - 1) loop
        stage_0(m)<=products_p(m*2) + products_p(m*2 + 1);
    end loop first_adders;
    second_adders:for l in 0 to (stage_0'length/2 - 1) loop
        stage_1(l)<=stage_0(l*2) + stage_0(l*2 + 1);
    end loop second_adders;
```

Presented by Abramov B.  
All right reserved

75

# Loops & Attributes

```
third_adders:for p in 0 to (stage_1'length/2 - 1) loop
    stage_2(p)<=stage_1(p*2) + stage_1(p*2 + 1);
end loop third_adders;
res<=stage_2(stage_2'left) + stage_2(stage_2'right);
end if;
end process adders;
end architecture arc_pipelined_mul;
```

Presented by Abramov B.  
All right reserved

76

# Assert statements

A statement that checks that a specified condition is true and reports an error if it is not.

Syntax:

```
assert condition  
    report string  
    severity severity_level;
```



Presented by Abramov B.  
All right reserved

77

## Assert statements (cont)

The **assertion** statement has three optional fields and usually all three are used.

The condition specified in an **assertion** statement must evaluate to a boolean value (true or false).

If it is false, it is said that an **assertion** violation occurred.

The expression specified in the report clause must be of predefined type **STRING** and is a message to be reported when assertion violation occurred.

If the severity clause is present, it must specify an expression of predefined type **SEVERITY\_LEVEL**, which determines the severity level of the assertion violation.

The **SEVERITY\_LEVEL** type is specified in the **STANDARD** package and contains following values: *NOTE*, *WARNING*, *ERROR*, and *FAILURE*.

**type** severity\_level **is** (*NOTE*, *WARNING*, *ERROR*, *FAILURE*);

Presented by Abramov B.  
All right reserved

78

## Assert statements (cont)

If the severity clause is omitted it is implicitly assumed to be ERROR.  
When an assertion violation occurs,  
the report is issued and displayed on the screen.  
The supported severity level supplies an information to the simulator.  
The severity level defines the degree to which the violation of the assertion  
affects operation of the process:

NOTE can be used to pass information messages from simulation.

```
assert false  
    report "The start of simulation :“ & time'image(now)  
severity NOTE;
```

Presented by Abramov B.  
All right reserved

79

## Assert statements (cont)

ERROR can be used when assertion violation makes continuation of the  
simulation not feasible.

Example:

```
assert not (s= '1' and r= '1')  
    report "Both values of signals S and R are equal to '1'"  
severity ERROR;
```

When the values of the signals S and R are equal to '1', the message is  
displayed and the simulation is stopped  
because the severity is set to ERROR.

Presented by Abramov B.  
All right reserved

80



## Assert statements (cont)

Assertion statements are not only sequential,  
but can be used as concurrent statements as well.

A concurrent assertion statement represents a passive process statement  
containing the specified assertion statement.

Important notes:

- The message is displayed when the condition is NOT met,  
therefore the message should be an opposite to the condition.
- Concurrent assertion statement is a passive process and as such can be  
specified in an entity.
- Concurrent assertion statement monitors specified condition  
continuously.

Presented by Abramov B.  
All right reserved

81

## Assert statements (cont)

Assertion statements are not only sequential,  
but can be used as concurrent statements as well.

A concurrent assertion statement represents a passive process statement  
containing the specified assertion statement.

Important notes:

- The message is displayed when the condition is NOT met,  
therefore the message should be an opposite to the condition.
- Concurrent assertion statement is a passive process and as such can be  
specified in an entity.
- Concurrent assertion statement monitors specified condition  
continuously.
- Synthesis tools generally ignore assertion statements.

Presented by Abramov B.  
All right reserved

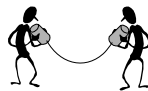
82

## Assert statements (cont)

VHDL-93 supports the report statement without the assert construction.

Example:

```
report "The current simulation time:" & time'image(now);
```



Presented by Abramov B.  
All right reserved

83

## Assert statements (cont)

```
process (clk, d) is
begin
  if clk'event and clk='1' then
    assert d'stable (tsu)
    report "d changed within setup interval"
    severity Warning;
  end if;
  if clk'event and clk='1' then
    assert d'stable (th)
    report "d changed within hold interval"
    severity Warning;
  end if;
end process;
```

Presented by Abramov B.  
All right reserved

84

# File I/O

VHDL defines the file object and includes some basic file IO procedures implicitly after a file type is defined.

A file type must be defined for each VHDL type that is to be input from or output to a file.

Example:

**TYPE** bit\_file **IS** **FILE** of bit;

In VHDL87, there are no routines to open or close a file, so both the mode of the file and its name must be specified in the file declaration.

The mode defaults to read if none is specified.



Examples:

**FILE** in\_file :bit\_file **IS** "my\_file.dat" -- opens a file for reading

**FILE** out\_file:bit\_file **IS** **OUT** "my\_other\_file.dat"; -- opens a file for writing

Presented by Abramov B.  
All right reserved

85

## File I/O (cont)

In VHDL93, a file can be named and opened in the declaration:

**FILE** in\_file:bit\_file **OPEN** **READ\_MODE** **IS** "my\_file.dat"; -- opens a file for reading

Or simply declared (and named and opened later):

**FILE** out\_file:bit\_file;

In VHDL87, the file is opened and closed when it come into and goes out of scope.

In VHDL93, there are two **FILE\_OPEN** procedures, one of which returns a value of the status (success) for opening the file, and one which doesn't.

There is also a **FILE\_CLOSE** procedure.

The values for **FILE\_OPEN\_KIND** are:

- ✓ **READ\_MODE**,
- ✓ **WRITE\_MODE**,
- ✓ **APPEND\_MODE**.

The values for **FILE\_OPEN\_STATUS** are:

- ✓ **OPEN\_OK**,
- ✓ **STATUS\_ERROR**,
- ✓ **NAME\_ERROR**,
- ✓ **MODE\_ERROR**.



Presented by Abramov B.  
All right reserved

86

## File I/O (cont)

The TEXTIO package provides additional declarations and subprograms for handling text (ASCII) files in VHDL. For example, the basic **READ** and **WRITE** operations of the **FILE** type are not very useful because they work with binary files. Therefore, the **TEXTIO** package provides subprograms for manipulating text more easily and efficiently. **TEXTIO** defines two new data types to assist in text handling. The first is the **LINE** data type. The **LINE** type is a text buffer used to interface VHDL I/O and the file. Only the **LINE** type may read from or written to a file. A new **FILE** type of **TEXT** is also defined. A file of type **TEXT** may only contain ASCII characters.

Presented by Abramov B.  
All right reserved

87

## File I/O (cont)

TEXTIO package declaration:

```
use std.textio.all;
```

STD\_LOGIC\_TEXTIO package provides subprograms for manipulating text with `std_ulogoc/std_logic` and `std_ulogic_vector/std_logic_vector` types.

STD\_LOGIC\_TEXTIO declaration:

```
library ieee;
```

```
use ieee.std_logic_textio.all;
```

For more efficiently STD\_LOGIC\_TEXTIO package, also provides the different representation of vector values: octal, hexadecimal and binary.

Presented by Abramov B.  
All right reserved

88

## File I/O (cont)

**PROCEDURE FILE\_OPEN ( ... )**

READ\_MODE  
WRITE\_MODE  
APPEND\_MODE

OPEN\_OK  
STATUS\_ERROR  
NAME\_ERROR  
MODE\_ERROR

**PROCEDURE FILE\_CLOSE ( ... )**

**FUNCTION ENDFILE ( ... )** return **BOOLEAN**;

Presented by Abramov B.  
All right reserved

89

## File I/O (cont)

```
procedure FILE_OPEN ( file f: ftype ;  
    External_Name: in string ;  
    Open_Kind: in File_Open_Kind:=READ_MODE );  
procedure FILE_OPEN ( Status: out FILE_OPEN_STATUS ;  
    file f: ftype ;  
    External_Name: in string ;  
    Open_Kind: in File_Open_Kind:=READ_MODE );
```

Preferred  
style

File\_Open\_Kind values :    READ\_MODE  
                              WRITE\_MODE  
                              APPEND\_MODE

Status values :  
    OPEN\_OK - File Opened successfully.  
    STATUS\_ERROR - File Object already has  
                    an external file associated with it.  
    NAME\_ERROR - External file does not exist.  
    MODE\_ERROR - External file can not open  
                    with requested File\_Open\_Kind.

Presented by Abramov B.  
All right reserved

90

## File I/O (cont)

```
procedure FILE_CLOSE ( file f : ftype )  
function ENDFILE ( file f : ftype ) return BOOLEAN;  
The simple loop:  
file_open_example: process is  
  file in_file : text;  
  constant file_name : string:= "my_file.txt"  
  constant file_path : string:= "c:\my_proj\";  
  -- others resources  
begin  
  file_open(in_file, file_path & file_name, read_mode);  
  -- optionally FILE_OPEN_STATUS check  
  while (not endfile(in_file)) loop  
    -- do something  
  end loop;  
  file_close(in_file);  
end process file_open_example;
```

Presented by Abramov B.  
All right reserved

91

## File I/O (cont)

```
procedure READLINE (file f: text; L: out line);  
procedure READ (L: inout line; VALUE : out bit; GOOD : out boolean);  
procedure READ (L: inout line; VALUE: out bit);  
procedure READ (L: inout line; VALUE: out bit_vector; GOOD: out boolean);  
procedure READ (L: inout line; VALUE: out bit_vector);  
procedure READ (L: inout line; VALUE: out boolean; GOOD: out boolean);  
procedure READ (L: inout line; VALUE: out boolean);  
procedure READ (L: inout line; VALUE: out character; GOOD: out boolean);  
procedure READ (L: inout line; VALUE: out character);  
procedure READ (L: inout line; VALUE: out integer; GOOD: out boolean);  
procedure READ (L: inout line; VALUE: out integer);
```

Presented by Abramov B.  
All right reserved

92

## File I/O (cont)

```
procedure READ (L: inout line; VALUE: out real; GOOD: out boolean);
procedure READ (L: inout line; VALUE: out real);

procedure READ (L: inout line; VALUE: out string; GOOD:out boolean);
procedure READ (L: inout line; VALUE: out string);

procedure READ (L: inout line; VALUE: out time; GOOD:out boolean);
procedure READ (L: inout line; VALUE: out time);
procedure WRITELINE (file f: text; L: inout line);
procedure WRITE (L: inout line; VALUE: in bit;
    JUSTIFIED: in SIDE := right;
    FIELD:in WIDTH := 0);
procedure WRITE (L: inout line; VALUE:in bit_vector;
    JUSTIFIED:in SIDE := right;
    FIELD:in WIDTH := 0);
procedure WRITE (L: inout line; VALUE:in boolean;
    JUSTIFIED:in SIDE := right;
    FIELD:in WIDTH := 0);
```

Presented by Abramov B.  
All right reserved

93

## File I/O (cont)

```
procedure WRITE(L: inout line; VALUE: in character;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L: inout line; VALUE: in integer;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L: inout line; VALUE: in real;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0;
    DIGITS: in NATURAL := 0);
procedure WRITE(L: inout line; VALUE: in string;
    JUSTIFIED : in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L: inout line; VALUE: in time;
    JUSTIFIED : in SIDE := right;
    FIELD: in WIDTH := 0;
    UNIT: in time := ns);
```

Presented by Abramov B.  
All right reserved

94

## File I/O (cont)

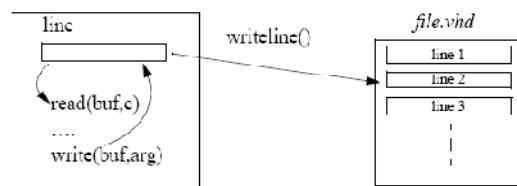
```
-- Hex
procedure HREAD(L: inout line; VALUE: out std_ulogic_vector);
procedure HREAD(L: inout line; VALUE: out std_ulogic_vector; GOOD: out boolean);
procedure HWRITE(L: inout line; VALUE: in std_ulogic_vector;
    JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure HREAD(L: inout line; VALUE: out std_logic_vector);
procedure HREAD(L: inout line; VALUE: out std_logic_vector; GOOD: out boolean);
procedure HWRITE(L: inout line; VALUE: in std_logic_vector;
    JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);

-- Octal
procedure OREAD(L: inout line; VALUE: out std_ulogic_vector);
procedure OREAD(L: inout line; VALUE: out std_ulogic_vector; GOOD: out boolean);
procedure OWRITE(L: inout line; VALUE: in std_ulogic_vector;
    JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
procedure OREAD(L: inout line; VALUE: out std_logic_vector);
procedure OREAD(L: inout line; VALUE: out std_logic_vector; GOOD: out boolean);
procedure OWRITE(L: inout line; VALUE: in std_logic_vector;
    JUSTIFIED: in SIDE := RIGHT; FIELD: in WIDTH := 0);
```

Presented by Abramov B.  
All right reserved

95

## File I/O (cont)



- A file is organized by *lines*
- Read and write procedures operate on line data structures
- Readline and writeline procedures transfer data to-from files

Presented by Abramov B.  
All right reserved

96



## File I/O (cont)

How to work with files in VHDL ?

Read from file:

- ✓ Open file in read\_mode(use file\_open procedure)
- ✓ Read the current line from file (use read\_line procedure)
- ✓ Current line pointer is updated automatically (on each call of read\_line procedure).
- ✓ Read from current line the relevant slice of data (use read procedure)
- ✓ Read data from line in accordance to types of objects stored in line (use read procedure with different types)
- ✓ Read the required number of lines or until EOF (use loop statement)
- ✓ Close file (use file\_close procedure)

Presented by Abramov B.  
All right reserved

97

## File I/O (cont)

How to works with files in VHDL ?

Write to file:

- ✓ Open the file in write\_mode(use file\_open procedure)
- ✓ Write data to line in accordance to types of objects that should be saved in line (use write procedure with different types)
- ✓ Write the current line to file (use write\_line procedure)
- ✓ Current line pointer updates automatically with each call to procedure write\_line.
- ✓ Write the required number of lines ( use loop statement)
- ✓ Close the file ( use file\_close procedure)

Presented by Abramov B.  
All right reserved

98

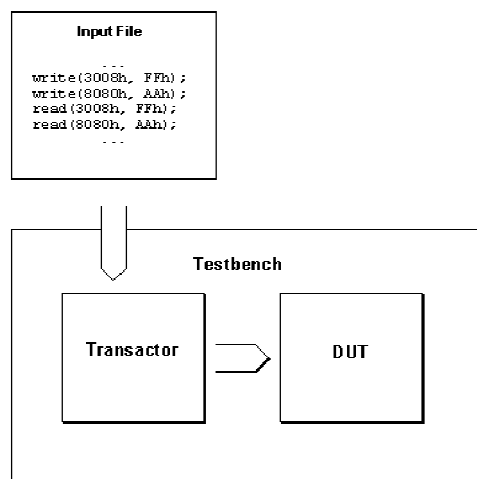
## File I/O (cont)

```
PROCESS(...)  
FILE DataFile : text;  
BEGIN  
:  
file_open(DataFile,"test.dat",write_mode);  
:  
:  
file_close(DataFile);  
:  
:  
file_open (DataFile,"test.dat",append_mode);  
:  
file_close(DataFile);  
:  
END PROCESS;
```

Presented by Abramov B.  
All right reserved

99

## From TVF to Stimulus



Presented by Abramov B.  
All right reserved

100

# From TVF to Stimulus

```

process is
file stim_file : text;
variable fopen_stat : FILE_OPEN_STATUS;
variable l : line; variable t : real; variable good : boolean;
variable cl,d1,d2,d3 : bit;
begin
file_open(fopen_stat,stim_file, "test_vector.txt", read_mode);
assert (fopen_stat=OPEN_OK) report "file open error"
severity FAILURE;
while not endfile(stim_file) loop
readline(stim_file,l); read(l,t,good);
next when not good;
read(l,cl); read(l,d1); read(l,d2); read(l,d3);
wait for ((t*1 ns) - now);
clk <= to_stdlogic(cl); data1 <= to_stdlogic(d1);
data2 <= to_stdlogic(d2); data3 <= to_stdlogic(d3);
end loop;
file_close(stim_file);
wait;
end process;

```

## TV FILE

```

%          D D D
%          a a a
%          C t t t
%          l a a a
%          k 1 2 3
%          - - -
%          I I I I

0.0 0 0 0 0
100.0 1 0 1 0
200.0 0 0 1 0
300.0 1 0 0 1
400.0 0 1 0 1
500.0 1 1 0 1
600.0 0 0 0 1
700.0 1 0 1 0
800.0 0 1 1 0
900.0 1 1 0 0
1000.0 0 0 0 0

```

Presented by Abramov B.  
All right reserved

101

# From TVF to Stimulus (cont)

```

read_from_file:process is
file in_file : text;
variable fopen_stat : file_open_status;
variable v_we,v_re : std_logic;
variable v_data : byte;
variable curr_line : line;
variable curr_t : time;
variable read_status : boolean;
procedure send_message(status_flag : in boolean; name: in string) is
begin
assert status_flag
report "missing " & name & " value" severity warning;
end procedure send_message;
begin
file_open(fopen_stat ,in_file,"c:\fifo_test\test_in.txt",read_mode);

```

Presented by Abramov B.  
All right reserved

102

## From TVF to Stimulus (cont)

```
if (fopen_stat=OPEN_OK) then
  while (not endfile(in_file)) loop
    readline(in_file,curr_line);
    read(curr_line,curr_t,read_status);
    next when not read_status;
    if (now < curr_t) then wait for (curr_t - now);
    end if;
    read(curr_line,v_we,read_status);
    send_message(read_status,"we");
    read(curr_line,v_re,read_status);
    send_message(read_status,"re");
    read(curr_line,v_data,read_status);
    send_message(read_status,"data");
    write_data<=v_data;we<=v_we; re<=v_re;
  end loop;
  file_close(in_file);
  wait;
else .....
end if;
end process read_from_file;
```



Presented by Abramov B.  
All right reserved

103

## Non Textual (Binary) File I/O

```
process is
type IntegerFileType is file of integer;
file dataout : IntegerFileType is out "output.bin";
variable check : integer :=0;
begin
for count in 1 to 10 loop
  check := check +1;
  write(dataout, check);
end loop;
wait;
end process;
```

- VHDL provides read(file,value), write(file, value)

Presented by Abramov B.  
All right reserved

104

## Non Textual (Binary) File I/O

```
process is
type IntegerFileType is file of integer;
file dataout : IntegerFileType is out "output.bin";
variable check : integer :=0;
begin
for count in 1 to 10 loop
    check := check +1;
    write(dataout, check);
end loop;
wait;
end process;
```

- VHDL provides read(file,value), write(file, value)

Presented by Abramov B.  
All right reserved

105

## Writing messages to shell

Work with files the powerful and flexible mechanism, but

however in many cases it possesses some redundancy.

In such cases it is more preferable to direct a stream to a window of a simulator:

```
process is
variable l, buff : line;
begin
samples := 1;
while (cpum_rd = '0') loop
if (cpum_cs_n = '1') then
report "##### data last sample";

l := new string("reading data => ");
hwrite(l, data);
writeln(output, l);
deallocate(l);
exit;
end if;

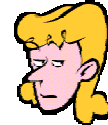
wait until rising_edge(cpu_clk);
if (cpum_cs_n = '0') then
data <= per_data;
wait for 0 ns;
buff := new string(" value=>");
hwrite(buff, data);
report "##### data sample N="
& integer'image(samples) & buff.all;
deallocate(buff);
samples := samples + 1;
end if;
end loop;
wait until falling_edge(cpum_cs_n);
end process;
```

Presented by Abramov B.  
All right reserved

106

# Writing messages to shell

```
--cpu read cycle data setup time violation checker
setup_check: process (cpu_clk) is
begin
  if rising_edge(cpu_clk) then
    if (cpum_rd='0' and cpum_cs_n='0' and samples>1) then
      if not per_data'stable(2 ns) then
        report "cpu data read setup time violation " & time'image(now)
        severity WARNING;
      end if;
    end if;
  end if;
end process setup_check;
```



Presented by Abramov B.  
All right reserved

107

# Writing messages to shell

```
--cpu read cycle hold time violation checker
hold_check: process (per_data) is
begin
  if per_data'event then
    if (cpum_rd='0' and cpum_cs_n='0' and samples>1) then
      if not cpu_clk'stable(2 ns) and cpu_clk'last_value='0' then
        report "cpu data read hold time violation " & time'image(now)
        severity WARNING;
      end if;
    end if;
  end if;
end process hold_check;
```



Presented by Abramov B.  
All right reserved

108

## Reading from command line

The opportunity dynamically to influence process of simulation by means of input from an environment of a simulator , it is the next step on a way of creation of more advanced test bench environment .

Example of setting several cycles for simulation:

```
number_of_cycles : process is
variable l : line;
variable num : integer;
begin
    readline (input,l);
    read (l,num);
    cycles_count <= num;
    wait;
end process;

stimulus: process is
variable loop_counter : integer;
begin
    wait on cycles_count;
    loop_counter := cycles_count;
    while loop_counter > 0 loop
        -----
        loop_counter:=loop_counter-1;
    end loop;
end process;
```

Presented by Abramov B.  
All right reserved

109

## Graphical Stimulus Representation

### ■ “Drawing” inputs in a VHDL test bench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_demo IS
END ENTITY tb_demo ;
ARCHITECTURE test OF tb_demo IS

    SIGNAL clock : std_logic := '0';
    SIGNAL resetN : std_logic := '0';
    SIGNAL in1 : std_logic := '0';
    SIGNAL in2 : std_logic := '0';
BEGIN
    clock <= NOT clock AFTER 100 ns;
    resetN <= '1' AFTER 200 ns;
```

Presented by Abramov B.  
All right reserved

110

# Graphical Stimulus Representation

```
process is
  variable vec1 :string(1 to 20):= "-----";
  variable vec2 :string(1 to 20):= "-----";
begin
  for i in vec1'range loop
    wait until rising_edge(clk);
    in1<='x';in2<='x';
    if (vec1(i) = '-') then
      in1 <= '1';
    elsif (vec1(i)='_') then
      in1 <= '0';
    end if;
    if (vec2(i) = '-') then
      in2 <= '1';
    elsif (vec2(i)='_') then
      in2 <= '0';
    end if;
  end loop;
end process;
end architecture test;
```



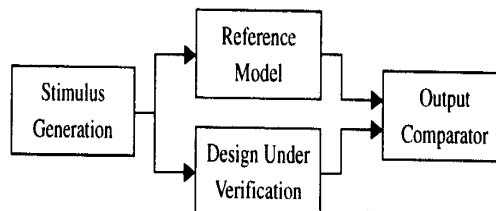
Presented by Abramov B.  
All right reserved

111

## Full Automatic Test Environment

The first step in automating verification is to include the expected outputs with the inputs stimulus for every clock cycle.

The next step toward automation of the output verification is the use of golden vectors.

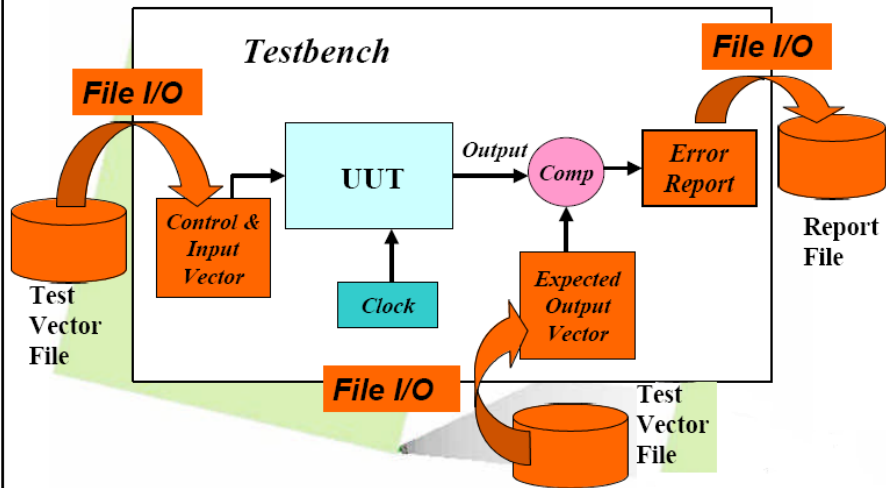


Presented by Abramov B.  
All right reserved

112



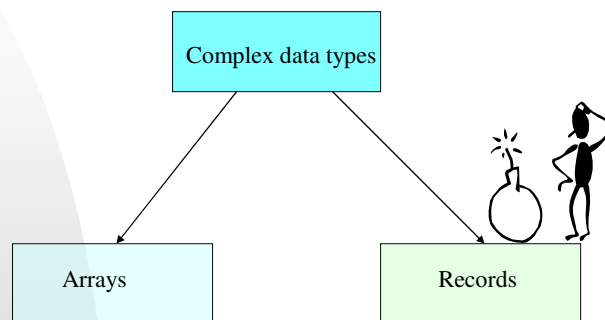
## Full Automatic Test Environment



Presented by Abramov B.  
All right reserved

113

## Complex Data Types



Presented by Abramov B.  
All right reserved

114

# Arrays

**Array** contains multiple elements of the same type.

Syntax:

```
type type_name is array ( range ) of element_type;
```



When an array object is declared, an existing array type must be used.

```
type nibble is array ( 3 downto 0 ) of std_logic;
```

```
signal my_bus : nibble;
```

Presented by Abramov B.  
All right reserved

115

# Arrays (cont)

An array type definition can be unconstrained, i.e. of undefined length.

**String**, **bit\_vector**, an **std\_logic\_vector** are defined in this way.

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

An object ( signal, variable or constant) of an unconstrained array type must have its index range defined when it is declared.

```
signal byte : std_logic_vector(7 downto 0);
```

Presented by Abramov B.  
All right reserved

116

## Arrays (cont)

### 1-D arrays:

```
type dword is array (31 downto 0) of integer;
```

```
constant depth : integer :=10;
```

```
type ram_model is array ((2**depth -1) downto 0) of dword; -- defined 1Kdword's
```

### 2-D arrays:

```
type arr_2d is array ((3 downto 0), (3 downto 0)) of dword;
```

Note: logic synthesis tools accept one-dimensional arrays of others supported types.



Presented by Abramov B.  
All right reserved

117

## Initializing Arrays

```
type point3d is array(1 to 3) of real;
```

```
signal P : point3d := (12.5, 2.4, 1.9);
```

```
signal P : point3d := (1=>12.5, 2=>2.4, 3=>1.9);
```

```
subtype addr is integer range 0 to 127;
```

```
type memory is array(addr) of std_logic_vector(7 downto 0);
```

```
signal mem : memory := (0=>X"FF", 3=>X"AB", others=>X"00");
```

```
signal mem : memory := (0 to 10=>X"AB", others=>X"00");
```

```
signal mem : memory := (others=>( others =>'0'));
```

```
subtype shortint is integer range 0 to 25;
```

```
type asciihex is array('a' to 'z') of shortint;
```

```
constant convtable : asciihex := ('a'=>0, 'b'=>1, ... 'z'=25);
```

Presented by Abramov B.  
All right reserved

118

## Initializing Arrays (cont)

```
type std2bit is array(std_logic) of bit;  
constant convtable : std2bit :=  
(‘U’=>‘0’, ‘X’=> ‘0’, ‘0’=> ‘0’, ‘1’ =>‘1’, ‘Z’ =>‘1’, ‘W’ =>‘0’, ‘L’ =>‘0’, ‘H’=>‘1’, ‘-’=>‘0’);  
  
-- initializing a multidimensional array  
type mod4table is array(0 to 3, 0 to 3) of integer;  
constant multable : mod4table := (0 => (others=> 0),  
                                  1 => (0=>0, 1=>1, 2=>2, 3=>3),  
                                  2 => (0=>0, 1=>2, 2=>1, 3=>2),  
                                  3 => (0=>0, 1=>3, 2=>2, 3=>1));
```

Presented by Abramov B.  
All right reserved

119

## Unconstrained Arrays

**array** type **range** < > **of** element type

```
type std_logic_vector is array (natural range<>) of std_logic;  
  
-- constraining by declaration  
signal word :std_logic_vector(31 downto 0);  
  
-- constraining by subtype definition  
subtype word is std_logic_vector(31 downto 0);  
  
-- constraining by initialization  
signal word :std_logic_vector := (1=>‘1’, 2=>‘Z’, 3=>‘1’);  
  
signal word :std_logic_vector := (‘1’, ‘Z’, ‘1’, ‘0’, ‘Z’);
```

Presented by Abramov B.  
All right reserved

120

# Records

The second type of composite types is **record** type.

**Record** type have named fields of different types:

```
type t_packet is record  
    byte_id : std_logic;  
    parity   : std_logic;  
    address  : integer range 3 downto 0;  
    data     : std_logic_vector( 3 downto 0);  
    valid    : boolean;  
end record;
```



Presented by Abramov B.  
All right reserved

121

# Records (cont)

## Records – A way of abstraction

- VHDL records support the abstraction of data into high level representation.
- Records are collection of elements that might be of different types (elements – filed).
- Records are ideal for representing packets of frames, standard and non-standard buses and interfaces.



Presented by Abramov B.  
All right reserved

122

## Records (cont)

Whole record values can be assigned using assignment statements, for example:

```
signal first_packet,last_packet : t_packet;  
  
last_packet<=first_packet;  
first_packet<=('1','1',2,x"a",true);  
last_packet.valid<=false;  
first_packet.data<=last_packet.data;
```

Presented by Abramov B.  
All right reserved

123

## CDT Example

```
entity fifo is  
  generic ( data_width : natural:=16 ; depth : natural :=256);  
  port (  
    clk      : in std_logic ;  
    rst      : in std_logic ;  
    din      : in std_logic_vector((data_width - 1) downto 0);  
    we       : in std_logic ;  
    re       : in std_logic ;  
    dout     : in std_logic_vector((data_width - 1) downto 0);  
    valid    : out std_logic ;  
    full     : buffer boolean ;  
    empty    : buffer boolean ;  
  );  
end entity fifo;
```

Presented by Abramov B.  
All right reserved

124

## CDT Example (cont)

```
architecture arc_fifo of fifo is
  constant active : std_logic := '0';
  type sram_type is array ((depth - 1) downto 0) of
    std_logic_vector((data_width - 1) downto 0);

  signal sram      : sram_type;
  signal wp        : integer range (depth - 1) downto 0;
  signal rp        : integer range (depth - 1) downto 0;
  signal data_in   : std_logic_vector((data_width - 1) downto 0);
  signal data_out  : std_logic_vector((data_width - 1) downto 0);
  -----
begin
```

Presented by Abramov B.  
All right reserved

125

## CDT Example (cont)

```
ram: process (we, wp, full, din) is
  begin
    if (we = '1' and (not full)) then
      sram(wp) <= din;
    end if;
  end process ram;

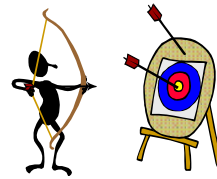
  dout <= sram(rp);
  full  <= (rp = (wp + 1));
  empty <= (wp = rp);
```

Presented by Abramov B.  
All right reserved

126

## CDT Example (cont)

```
write_pointer : process (clk,rst) is
begin
  if (rst = active) then
    wp <= 0;
  elsif rising_edge(clk) then
    if ((not full) and (we = '1')) then
      wp <= (wp + 1) mod depth;
    end if;
  end if;
end process write_pointer;
```



Presented by Abramov B.  
All right reserved

127

## CDT Example (cont)

```
read_pointer : process (clk,rst) is
begin
  if (rst = active) then
    rp <= 0;
    valid <= '0';
  elsif rising_edge(clk) then
    if ((not empty) and (re = '1')) then
      rp <= (rp + 1) mod depth;
      valid <= '1';
    else
      valid <= '0';
    end if;
  end if;
end process read_pointer;
end architecture arc_fifo;
```



Presented by Abramov B.  
All right reserved

128



# Aliases

An *alias* is an alternative name for existing object.

It does not define a new object.

VHDL provides the alias construct to enhance readability in VHDL descriptions.

Aliases are available in two varieties:

1. Object aliases rename objects
  - a. constant
  - b. signal
  - c. variable
  - d. file
2. Non-object aliases rename items that are not objects aliases
  - a. function names
  - b. literals
  - c. type names
  - d. attribute names

Presented by Abramov B.  
All right reserved

129

## Aliases (cont)

Syntax:

```
alias alias_name : alias_type is object_name;
```

Example:

```
signal my_bus : std_logic_vector(31 downto 0);  
alias upper_half_word : std_logic_vector(15 downto 0) is my_bus(31 downto 16);  
alias lower_half_word : std_logic_vector(15 downto 0) is my_bus(15 downto 0);  
alias reversed_bus : std_logic_vector(0 to 31) is my_bus;
```

In VHDL-93 standard all object may be “aliased”, all “non-object” can also be “aliased”.

Example:

```
alias main_logic is ieee.std_logic_1164.std_logic;
```

Presented by Abramov B.  
All right reserved

130

## Aliases (cont)

```
architecture arc_keyboard_receiver of keyboard_receiver is
  signal serial2parallel : std_logic_vector(10 downto 0);
  alias start_bit        : std_logic is serial2parallel(0);
  alias stop_bit         : std_logic is serial2parallel(10);
  alias odd_parity_bit   : std_logic is serial2parallel(9);
  alias scan_code        : std_logic_vector(7 downto 0) is serial2parallel(8 downto 1);
begin
  .....
  .....
  valid<='1' when (start_bit='0' and stop_bit='1' and parity=odd_parity_bit) else '0';
```



Presented by Abramov B.  
All right reserved

131

## Packages

A package is a **cluster of declarations and definitions** of objects, functions, procedures, components, attributes etc. that can be used in a VHDL description.

You cannot define an entity or architecture in a package,

so a package by itself does not represent a circuit.

A package consists of two parts:

- The package header, with declarations
- The package body, with definitions.

An example of a package is std\_logic\_1164, the IEEE 1164 logic types package.

It defines types and operations on types for 9-valued logic.

To include functionality from a package into a VHDL description, the use clause is used.

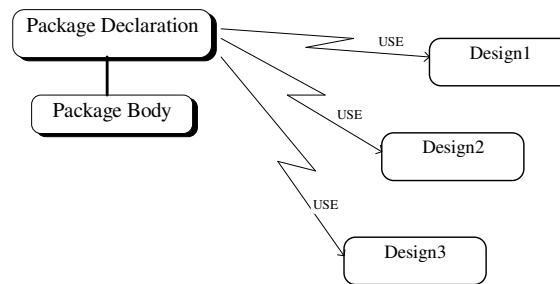


Presented by Abramov B.  
All right reserved

132

## Packages (cont)

To include functionality from a package into a VHDL description, the use clause is used.



Presented by Abramov B.  
All right reserved

133

## Packages (cont)

A Package is a common storage area used to hold data to be shared among a number of entities.

### Kind Of Packages:

- Standard Packages (IEEE, TextIO)
- Synthesizer Packages (exemplar\_1164)
- VITAL Packages (Gate Level Primitives)
- User Defined Project Packages

Presented by Abramov B.  
All right reserved

134

## Packages (cont)

Declaration part syntax :

```
package package_name is  
    declarations  
end package package_name;
```

Declarations may typically be any of the following:

type, subtype, constant, file, alias, component,  
attribute, function, procedure.

```
package demo_pack is  
    constant full : std_logic_vector;  
    type state is (idle, wait_cs, ack);  
    function parity ( data : std_logic_vector) return std_logic;  
end package demo_pack;
```

Presented by Abramov B.  
All right reserved

135

## Packages (cont)

When a procedure or function is declared in a package ,

Its body ( the algorithm part) must be placed in the **package body**.

Definition part syntax:

```
package body package_name is  
    declarations  
    deferred constant declaration  
    subprogram bodies  
end package body package_name;
```

Presented by Abramov B.  
All right reserved

136

## Packages (cont)

A constant declared in a package may be deferred.

This means its value is deferred in the package body.

**package body** demo\_pack **is**

**constant** full : **std\_logic\_vector** :=x"ff";

**function** parity ( data : **std\_logic\_vector**) **return** **std\_logic** **is**  
    **begin**

        -- function code

**end function** parity;

**end package body** demo\_pack ;

Presented by Abramov B.  
All right reserved

137

## Function

Function the powerful tool to implement functionality that is repeatedly used.

Functions take a number of arguments that are all inputs to the

function, and **return a single value.**

- All statements in functions and procedures are executed sequentially.
- May contain any sequential statement except signal assignment and wait.
- Variables that are local to the function can be declared .
- Local signals are not allowed.
- A type –conversion function may be called in a port map.
- Array-type parameters may be unconstrained.
- Can be called from the dataflow environment and from any sequential environment (processes or other sub-programs)
- Presents only combinatorial logic !!!

Presented by Abramov B.  
All right reserved

138

## Function (cont)

Syntax :

```
function function_name ( parameter_list) return type is  
    local declarations  
begin  
    sequential statements  
end function function_name ;
```



Presented by Abramov B.  
All right reserved

139

## Function (cont)

Function implementation example:

```
function parity ( data : std_logic_vector ) return std_logic is  
variable temp : std_logic;  
begin  
    for i in data'range loop  
        if (i=data'left) then  
            temp:=data(data'left);  
        else  
            temp:=temp xor data(i);  
        end if;  
    end loop;  
    return temp;  
end function parity;
```

Presented by Abramov B.  
All right reserved

140

## Function (cont)

Another implementation with variable initialization:

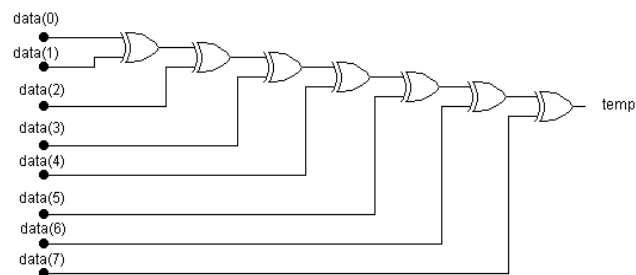
```
function parity ( data : std_logic_vector ) return std_logic is  
variable temp : std_logic = '0';  
begin  
  for i in data'range loop  
    temp := temp xor data(i);  
  end loop;  
  return temp;  
end function parity;
```

Presented by Abramov B.  
All right reserved

141

## Function (cont)

Hardware implementation of parity function with 8 bit data width:



Presented by Abramov B.  
All right reserved

142

# Aliases in functions

Aliases are often useful in unbound function calls.

For instance, if you want to make a function that takes the AND operation of the two left most bits of an arbitrary array parameter.

If you want to make the function general enough to handle arbitrary sized arrays, this function could look like this:

```
function left_and (arr: std_logic_vector) return std_logic is  
begin  
    return arr(arr'left) and arr(arr'left-1) ;  
end function left_and ;
```

Function does not work for ascending index ranges of arr!!!

Presented by Abramov B.  
All right reserved

143

# Aliases in functions(cont)

Instead, you could make an alias of arr,  
with a known index range, and operate on the alias:

```
function left_and (arr : std_logic_vector) return std_logic is  
    alias aliased_arr : std_logic_vector (0 to arr'length-1) is arr ;  
begin  
    return aliased_arr(0) and aliased_arr(1) ;  
end function left_and ;
```

Function works for both ascending and descending index ranges of arr!!!

Presented by Abramov B.  
All right reserved

144



# Function and recursion

The final function call shows how to implement the reduction operator (function call) using recursion to produce a tree.

In many cases a simple loop can produce the same results but the tree is guaranteed to give you the best synthesized result.



Presented by Abramov B.  
All right reserved

145

## Function (cont)

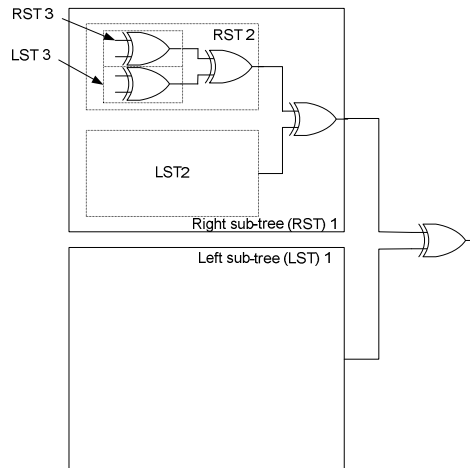
```
function recurse_xor (data: std_logic_vector) return std_logic is  
  alias t : std_logic_vector(0 to (data'length-1)) is data;;  
  variable left_tree , right_tree : std_logic;  
begin  
  if (t'length = 1) then  
    return t(0);  
  else  
    left_tree := recurse_xor (t(0 to (t'length / 2 - 1) ));  
    right_tree := recurse_xor (t(t'length / 2 to t'right));  
    return (left_tree xor right_tree);  
  end if;  
end function recurse_xor;
```

Presented by Abramov B.  
All right reserved

146

## Function (cont)

Hardware implementation  
of recursive parity function  
for 8 bit data width



Presented by Abramov B.  
All right reserved

147

## Function (cont)

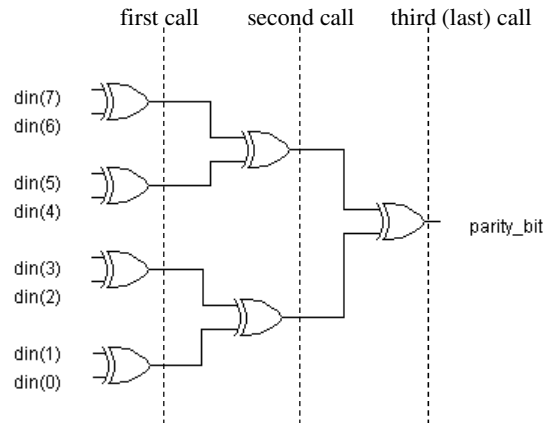
```
function recursive_parity ( vec : std_logic_vector) return std_logic is
variable temp : std_logic_vector((vec'length/2 - 1) downto 0);
begin
    if (vec'length = 2) then
        return (vec(vec'high) xor vec(vec'low));
    else
        for i in temp'range loop
            temp(i):=vec(i*2 + 1) xor vec(i*2);
        end loop;
        return recursive_parity(temp);
    end if;
end function recursive_parity;
```

Presented by Abramov B.  
All right reserved

148

## Function (cont)

Hardware implementation of recursive parity function with 8 bit data width:



Presented by Abramov B.  
All right reserved

149

## Function (cont)

```

type vec_arr is array (natural range <>) of std_logic_vector(15 downto 0);
function add (data : vec_arr) return std_logic_vector is
    alias t : vec_arr(0 to (data'length-1)) is data;
    variable res : std_logic_vector(t(0)'range);
begin
    if (t'length=1) then
        res:=t(0);
    else
        res:=add(t(0 to ((t'length/2)-1))) + add(t((t'length/2) to t'high));
    end if;
    return res;
end function add;
    
```

Presented by Abramov B.  
All right reserved

150

## Function (cont)

Functions that receive and return a constant values

**constant** MAX\_VALUE **integer** := some integer value;

**function** log2 (**constant** arg: **integer**) **return integer** **is**

**begin**

**for** i **in** 0 **to** 31 **loop**

**if** (2\*\*i>=arg) **then**

**return** i;

**end if**;

**return** 1;

**end function** log2;

**signal** cnt: **std\_logic\_vector** ( (log2(MAX\_VALUE)-1) **downto** 0);

### NOTICE!

This function has no hardware representation but only the calculation during compilation

Presented by Abramov B.  
All right reserved

151

## Function (cont)

**type** int\_arr **is** **array** (**natural range** <>) **of integer**;

**constant** C\_POLINOM : int\_arr:=(7,5,4,3);

**function** rand\_gen (buff : **std\_logic\_vector**; polinom : int\_arr) **return std\_logic\_vector** **is**

**variable** or\_gate, xor\_gate, sin : **std\_logic** := '0';

**begin**

**for** i **in** buff' **range** **loop**

        or\_gate:=or\_gate **or** buff(i);

**end loop**;

**for** j **in** polinom' **range** **loop**

        xor\_gate:=xor\_gate **xor** buff(polinom(j));

**end loop**;

    sin:=xor\_gate **or** (**not** or\_gate);

**return** (buff((buff' **high**-1) **downto** buff' **low**) & sin);

**end function** rand\_gen;

-- usage example

**elsif** rising\_edge(clk) **then**

    rand\_sig<=rand\_gen(rand\_sig,C\_POLINOM);

Presented by Abramov B.  
All right reserved

152

## Pure vs. Impure Functions

- A function may refer to signals or variables declared outside the function
- If it does, we call such functions **impure**.
- This means the result of the function may change from one call to another.
- Otherwise with the same inputs the function always returns the same value.
- Such functions are said to be **pure**.
- We may explicitly declare the type of a function.

Presented by Abramov B.  
All right reserved

153

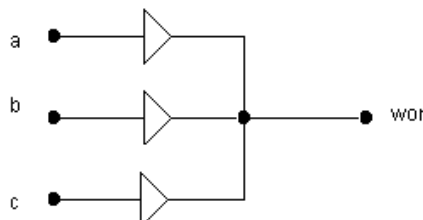
## Resolution function

In many digital systems, buses are used to connect a number of output drivers to a common signal.

For example, if open-collector or open-drain output drivers are used with a pull-up load on a signal, the signal can be pulled low by any driver, and is only pulled high by the load when all drivers are off.

This is called a *wired-or* or *wired-and* connection.

We can describe the following circuit with resolved type and resolution function



Presented by Abramov B.  
All right reserved

154

## Resolution function (cont)

```
library ieee;
use ieee.std_logic_1164.all;
package pack is
    function wor (din : std_logic_vector) return std_logic;
    subtype resolved_or is wor std_logic;
end package pack;
package body pack is
    function wor (din : std_logic_vector) return std_logic is
    begin
        for i in din'range loop
            if (din(i)='1') then
                return din(i);
            end if;
        end loop;
        return '0';
    end function wor;
end package body pack;
```



Presented by Abramov B.  
All right reserved

155

## Resolution function (cont)

```
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.pack.all;
-----
entity wired_or is
port(
    a : in std_logic;
    b : in std_logic;
    c : in std_logic;
    w_or: out resolved_or
);
end entity wired_or;
```



Presented by Abramov B.  
All right reserved

156

## Resolution function (cont)

```
architecture arc_wired_or of wired_or is  
begin  
    w_or<=a;  
    w_or<=b;  
    w_or<=c;  
end architecture arc_wired_or;
```



Presented by Abramov B.  
All right reserved

157

## Procedure

The other kind of subprograms is procedure.

Procedures take a number of arguments that can be inputs, outputs or inout, depending on the direction of the flow of information through the argument.

- All statements in procedures are executed sequentially
- May have **in**, **out** or **inout** parameters.
- Parameter may be signal, variable or constant.
- May be called concurrently or sequentially.
- A concurrent procedure call executes whenever any of its **in** or **inout** parameters change.
- May declare local variables.
- A procedure can contain wait statements, unless it is called from a process with a sensitivity list, or from within a function.
- May contain a return statement.

Presented by Abramov B.  
All right reserved

158

# Procedure (cont)

Procedures syntax :

```
procedure procedure_name (parameter_list) is  
    declaration  
  
begin  
    sequential statements  
  
end procedure procedure_name;
```



Presented by Abramov B.  
All right reserved

159

# Procedure (cont)

Procedure without parameters\_list:

```
procedure initialize is  
begin  
    address<=(others=>'-');  
    data<=(others=>'-');  
    wait until rst'event and rst='1';  
    wait for 10 ns;  
    cs<='0';  
    address<=x"00";  
    data<=x"ff";  
  
end procedure initialize;
```



Presented by Abramov B.  
All right reserved

160



## Procedure (cont)

Procedure with parameters\_list ( **in** by default)

```
procedure do_op (opcode : std_logic_vector) is  
    variable result : integer;  
begin  
    case opcode is  
        when add => result:=op1 + op2;  
        when sub => result:= op1 – op2;  
        .....  
    end case;  
    dest<= result after alu_tpd;  
end procedure do_op ;
```

Presented by Abramov B.  
All right reserved

161

## Procedure (cont)

Procedure with parameters\_list:

```
procedure adder ( a,b : in std_logic_vector ;  
                res: out std_logic_vector; overflow : out boolean) is  
    variable sum : std_logic_vector(a'range);  
    variable carry : std_logic:=‘0’;  
begin  
    for i in res'reverse_range loop  
        sum(i):= a(i) xor b(i) xor carry;  
        carry:=(a(i) and b(i)) or (a(i) and carry) or (b(i) and carry);  
    end loop;  
    res:= sum;  
    overflow:= (carry=‘1’);  
end procedure adder ;
```

Presented by Abramov B.  
All right reserved

162

## Procedure (cont)

Procedure with default value for a formal parameter of mode **in** :

```
procedure increment ( signal a : inout std_logic_vector;  
                      step : in integer:=1) is  
  
  begin  
    a<= a + step;  
  end procedure increment ;
```

We can call the procedure to increment by 1, as follows:

```
increment(count);  
  
or  
  
increment(count,open);
```

If we want to increment a signal count by other value ,we can call the procedure, as follows:

```
increment(count,5); or increment(count,new_step);
```

Presented by Abramov B.  
All right reserved

163

## Procedure (cont)

The procedures completed execution of the statements in their bodies before returning.

Sometimes it is useful to be able to return from the middle of a procedure.

We can do this using a return statement.

```
procedure read is  
  
  begin  
    addr<= addr_reg;  
    ram_read<='1';  
    wait until ram_ready='1' or rst='0';  
    if (rst='0') then  
      return;  
    end if;  
    wait until ram_ready='0';  
  end procedure read;
```

Presented by Abramov B.  
All right reserved

164

## Procedure (cont)

The procedures shift register as procedure with default parameters:

```

procedure shift_right_arst_ld_en( signal clk : in  std_logic; arst : in  std_logic;
ld : in  std_logic:= '0'; din : in std_logic_vector:= conv_std_logic_vector(0,din'length);
sin : in  std_logic_vector(15 downto 0):= x"0000";
en : in  std_logic; signal buf : inout std_logic_vector ) is
begin
    if (arst=RESET_INIT) then
        buf<=conv_std_logic_vector(0,buf'length);
    elsif rising_edge(clk) then
        if (ld='1') then
            buf(din'range)<=din;
        elsif (en='1') then
            buf<=sin & buf(buf'high downto sin'length);
        end if;
    end if;
end procedure shift_right_arst_ld_en;

```

Presented by Abramov B.  
All right reserved

165

## Procedure (cont)

Shift register shift right that provides shift right with serial in and shift enable by procedure *shift\_right\_arst\_ld\_en* :

```

shift_right_arst_ld_en
(
    clk => sys_clk,
    arst => sys_rst,
    sin => per_data,
    en  => ing_inbuf_en,
    buf => ing_search_inbuf
);

```

Not used inputs : ld -> parallel load and din -> parallel data input

Presented by Abramov B.  
All right reserved

166

## Procedure (cont)

Shift register that provides shift right with parallel load and shift enable by procedure *shift\_right\_arst\_ld\_en* :

```
shift_right_arst_ld_en
(
  clk => sys_clk,
  arst => sys_rst,
  ld  => cpu_table_rd_valid,
  din => cpu_table_data_rd,
  en  => ing_search_outbuf_re,
  buf => ing_search_outbuf
);
```

Not used inputs : sin -> serial data input (by default x"0000")

Presented by Abramov B.  
All right reserved

167

## Overloading

When we are writing subprograms, it is a good idea to chose names for out subprograms that indicate what operation they perform.

How to name two subprograms that perform the same kind of operation but on parameters of different types?

VHDL allows us to define subprograms in this way, using technique called **overloading** of subprogram names.

```
procedure increment(a : inout integer ; by : in integer:=1) is ...
procedure increment(a : inout std_logic_vector ; by : in integer:=1) is ...
procedure increment(a : inout std_logic_vector ;
  by : std_logic_vector :=b"1") is ...
```

```
-----
increment(count_1,2);
increment(count_v, "101")
```

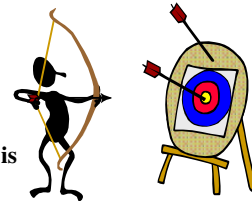
Presented by Abramov B.  
All right reserved

168

## Overloading (cont)

VHDL provides a way for us to define new subprograms using the operator symbols as names.

```
function "+" (left,right : std_logic) return std_logic is
begin
    return ( left or right);
end function "+";
function "*" (left,right : std_logic) return std_logic is
begin
    return ( left and right);
end function "*";
```



Now we can write your logical equation like a Boolean equation:

```
carry<=a*b + a*c + b*c;
```

Presented by Abramov B.  
All right reserved

169

## Overloading (cont)

```
package my_pack is
    function "not"(R: integer) return integer;
end package my_pack;

package body my_pack is
    function "not"(R: integer) return integer is
        variable vector : signed(31 downto 0);
    begin
        vector := conv_signed(R,32);
        for i in vector'range loop
            vector(i) :=not vector(i);
        end loop;
        return conv_integer(vector);
    end function "not";
end package body my_pack;
```

```
Int <= not(126);
Int <= not(integer(22.17))
```

Presented by Abramov B.  
All right reserved

170

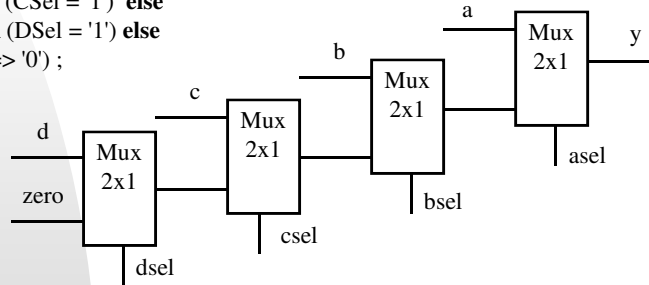
## Overloading (cont)

Simple design problem:

```

signal ASel, BSel, CSel, DSel : std_logic ;
signal Y, A, B, C, D : std_logic_vector(7 downto 0) ;
Y <= A when (ASel = '1') else
    B when (BSel = '1') else
    C when (CSel = '1') else
    D when (DSel = '1') else
    (others => '0') ;

```



This code defines priority select logic, which is inefficient from a hardware area and timing perspective.

Presented by Abramov B.  
All right reserved

171

## Overloading (cont)

When the select signals (ASel, ...) are mutually exclusive, overloading can be perfect solution for this problem.

```

function "and" (l : std_logic_vector ; R : std_logic)
    return std_logic_vector is
    variable res : std_logic_vector(l'range);
begin
    for i in l'range loop
        res(i) := l(i) and R;
    end loop;
    return res;
end function "and";

```

Now we can write :

```

Y <= (A and ASel) or (B and BSel) or (C and CSel) or (D and DSel) ;

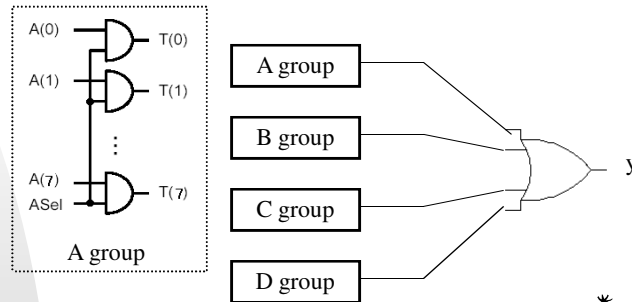
```

Presented by Abramov B.  
All right reserved

172

## Overloading (cont)

Here there is the hardware implication of A and ASEL:



Same functionality, but about 50 % faster and ~20 % smaller



Presented by Abramov B.  
All right reserved

173

## Abstract Data Types- ADT

An Abstract Data Types consists of two things:

- The custom VHDL data types and subtypes
- Operators that manipulate data of those custom types



ADTs are objects which can be used to represent an activity or component in behavioral modeling.

An ADT supports data hiding, encapsulation, and parameterized reuse.

As such they give VHDL some object-oriented capability.

An ADT is both a data structure (such as a stack, queue, tree, etc.)

and a set of functions (e.g. operators) that provide useful services of the data.

For example, a stack ADT would have functions for pushing an element onto the stack, retrieving an item from the stack, and perhaps several user-accessible attributes such as whether the stack is full or empty.

Presented by Abramov B.  
All right reserved

174

# Abstract Data Types- ADT

## Benefits

- ✓ Improve “readability” of VHDL
- ✓ Simulate with high level data types
- ✓ Rapid switch to synthesis – no change to architecture required
- ✓ Rapid Simulation
- ✓ Concurrent Development
- ✓ Single Architect/Designer

Presented by Abramov B.  
All right reserved

175

# Abstract Data Types- ADT

- Design Capture Paradigms
- Definition of ADT
- VHDL Support For ADT's
- ADT's In Our Design

Presented by Abramov B.  
All right reserved

176



# Abstract Data Types- ADT

## Design Capture Paradigms

### ▪Procedural

✓Focus solely on processing

✓Example: SQRT function

### ▪Modular

✓Focus on data organization

✓Example: FIFO module

### ▪Data abstraction

✓Focus on type of data

✓Set of operations

Example: Floating point, +, -, \*, /

Presented by Abramov B.  
All right reserved

177

## ADT (cont)

```
package complex_type is
  constant re : integer:=0;
  constant im : integer:=1;
  library ieee;
  use ieee.std_logic_1164.all;
  subtype dword is std_logic_vector(31 downto 0);
  type complex is array (re to im) of dword;

  function "+" (a,b: complex) return complex;
  function "-" (a,b: complex) return complex;
  function "*" (a,b: complex) return complex;
  function "/" (a,b: complex) return complex;
  function conjugate (a,b: complex) return complex;
end package complex_type;
```

This is a package declaration for a package that implements a complex number data type.  
Note that the data type is given as well as some standard operators on that type.

Presented by Abramov B.  
All right reserved

178

## ADT (cont)

```
package body complex_type is
  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_signed.all;
  function "+" (a,b: complex) return complex is
    variable t : complex;
  begin
    t(re):=a(re) + b(re);
    t(im):=a(im) + b(im);
    return t;
  end function "+";
  function "-" (a,b: complex) return complex is
    variable t : complex;
  begin
    t(re):=a(re) - b(re);
    t(im):=a(im) - b(im);
    return t;
  end function "-";
```

Presented by Abramov B.  
All right reserved

179

## ADT (cont)

```
function "*" (a,b: complex) return complex is
  variable t : complex;
begin
  t(re):=a(re) *b(re) - a(im)*b(im);
  t(im):=a(re) *b(im) + a(im)*b(re);
  return t;
end function "*";
function "/" (a,b: complex) return complex is
  variable t : complex;
  variable i : integer:=b(re)**2 + b(im)**2;
begin
  t(re):= a(re) *b(re) + a(im)*b(im);
  t(im):=a(im) *b(re) - a(re)*b(im);
  t(re):=t(re)/i;
  t(im):=t(im)/i;
  return t;
end function "/";
```

Presented by Abramov B.  
All right reserved

180

## ADT (cont)

```
function conjugate (a) return complex is  
  variable t : complex:=(0,0);  
  begin  
    t(re):=a(re) - b(im);  
    return t;  
  end function conjugate;  
end package body complex_type;
```

Presented by Abramov B.  
All right reserved

181

## ADT Benefits

- Reduced design communication overhead
- No functional boundaries to negotiate
- Easier to manage feature changes later
- Changes made with less consequence
- Early problem detection
- Minimal component structure - fewer files to edit
- Enhanced readability
- ✓ Type names add extra "descriptiveness"
- ✓ Readable by non-VHDL literate people
- ✓ Increased maintainability
- ✓ Type implementations easy to document
- Intellectual property development facilitated

Presented by Abramov B.  
All right reserved

182

# ADT Simulation Boosting

Video frame (640x480 pixel) rendering simulation:

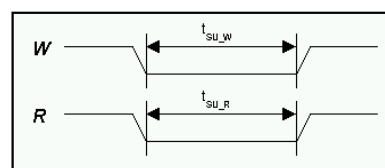
- ✓ With ADT ~ 15 min
- ✓ Without ADT (std\_logic type) ~ 2.0 hours

Presented by Abramov B.  
All right reserved

183

## Advanced Test Environment with BFM

- ❑ Simulation environment with SRAM behavioral model
- ❑ Below are the requirements for two output signals of a SRAM device



Parameter	Description	Min	Max	Unit
$t_{SU\_W}$	Setup time for W asserted	6	-	ns
$t_{SU\_R}$	Setup time for R asserted	5	-	ns

Presented by Abramov B.  
All right reserved

184

# Advanced Test Environment

```
timing_check: process is
  variable w_asserted, r_asserted: time;
begin
  wait until falling_edge (rst); -- wait for DUT to be reset
  wait until w = '0'; -- verify write access
  w_asserted := now;
  wait until w = '1';
  assert (now - w_asserted) >= tSU_W
    report "W setup time too short" severity Error;
  wait until r = '0'; -- verify read access
  r_asserted := now;
  wait until r = '1';
  assert (now - r_asserted) >= tSU_R
    report "R setup time too short" severity Error;
end process timing_check;
```

Presented by Abramov B.  
All right reserved

185

# Advanced Test Environment

```
stimulus: process is
begin
  w <= '1';
  r <= '1';
  wait until falling_edge (rst);
  wait for 10 ns;
  w <= '0', '1' after 8 ns; -- write access
  wait for 10 ns;
  r <= '0', '1' after 9 ns; -- read access
  wait for 10 ns;
  w <= '0', '1' after 7 ns; -- write access
  wait for 10 ns;
  -- read access
  r <= '0', '1' after 4 ns; -- this is a violation we want to detect
  wait for 10 ns;
  w <= '0', '1' after 8 ns; -- write access
  wait;
end process stimulus;
```

Presented by Abramov B.  
All right reserved

186

# Advanced Test Environment

```

process is -- Internal Memory
type mem_add_type is array (integer range <>) of std_logic_vector (a'range);
type mem_dat_type is array (integer range <>) of std_logic_vector (d'range);
variable mem_add: mem_add_type ( mem_words -1 downto 0);
variable mem_dat: mem_dat_type ( mem_words -1 downto 0);
variable used_pnt: integer := 0;
begin
    d <= (others => 'Z');
    wait until we_l'event or rd_l'event;
    assert (we_l='0' or we_l='1' or no_reset_yet ) report "invalid value" severity Error;
    assert (rd_l='0' or rd_l='1' or no_reset_yet) report " invalid value" severity Error;
    assert (to_X01(rd_l) /= '0' or to_X01(we_l) /= '0')
    report "both read and write are asserted" severity Error;
    if to_X01(we_l) = '0' then write;
    end if;
    if to_X01(rd_l) = '0' then read;
    end if;
end process;

```

Presented by Abramov B.  
All right reserved

187

# Advanced Test Environment

```

procedure write is
begin
    wait until to_X01(we_l) = '0'; -- Store written data
    for b in 0 to used_pnt loop
        if (i = used_pnt) then -- access to a new address
            mem_add(i) := a;
            mem_dat(i) := d;
            if (used_pnt < mem_words - 1) then
                used_pnt := used_pnt + 1;
            else report "Simulation model can't handle additional addresses";
            end if;
        end if;
        if mem_add(i) = a then -- access to an existing address
            mem_dat(i) := d;
            exit;
        end if;
    end loop;
end procedure write;

```

Presented by Abramov B.  
All right reserved

188

# Advanced Test Environment

```
procedure read is
begin -- Retrieve data from internal memory
wait until to_X01(re_l) = '0';
for i in 0 to used_pnt+1 loop
  if (i = used_pnt+1) then -- access to a new address
    report "Address has not been written to yet";
    d <= (others => 'X');
    exit;
  end if;
  if (mem_add(i) = a) then -- access to an existing address
    d <= mem_dat(i) after tRD;
    exit;
  end if;
end loop;
end procedure read;
```

Presented by Abramov B.  
All right reserved

189

## Bus Functional Model

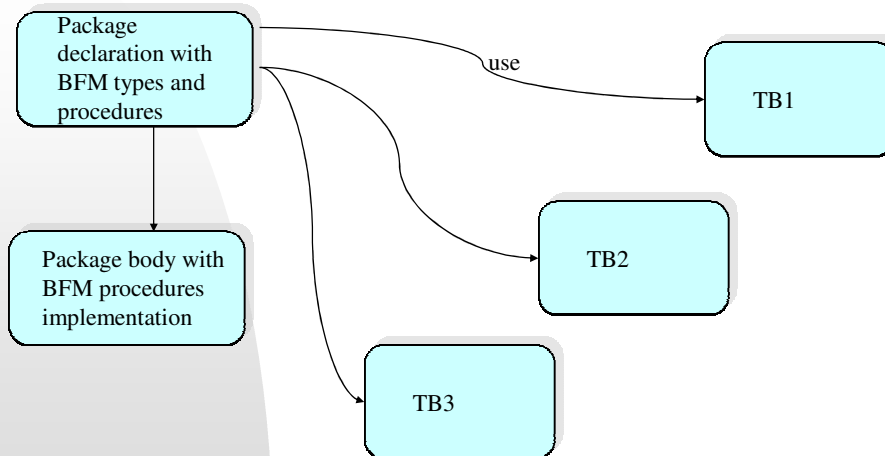
In this example, the record contains information fields that the BFM uses to drive stimulus and pass back to the process that contains the executed procedure:

- ✓ The cycle type
- ✓ Address of the cycle
- ✓ Write data if it's a write operation or expected read data if the cycle is a read operation
- ✓ Actual read data the BFM received
- ✓ Handshaking strobe signals used by the main procedure and the BFM for communication.

Presented by Abramov B.  
All right reserved

190

## Bus Functional Model



Presented by Abramov B.  
All right reserved

191

## Bus Functional Model

The stimulus driver; it utilizes the BFM procedure calls from the global package to drive the BFM stimulus.

```

wr("0000000f",test_vector,cmd); -- write cycle to address h0000000f
idle(cmd);                      -- idle
rd("00000000",test_vector,cmd); -- read cycle from address h00000000
  
```

writing 32-bit words to every eight-byte address memory elements in a given address range:

```

tv := to_stdlogicvector(x"a5a5e3e3");
for i in 16#00000100# to 16#00000140# loop
  if (i mod 8 = 0) then -- every other dword
    wr(conv_std_logic_vector(i,32),tv,cmd); -- write value to DUT
    tv := tv xor tv0; -- the next data
  end if;
end loop;
  
```

Presented by Abramov B.  
All right reserved

192



## Bus Functional Model

A classic read-modify-write cycle that reads a value from memory, modifies the read data, and writes out the newly modified piece of data back to memory:

```
av := to_stdlogicvector(x"00000020");
idle(cmd);
-- now get the actual data read
rd(av,tv,cmd);
tv(15 downto 0) := tv(15 downto 0) - 16#0ac1#;
tv(31 downto 16) := tv(31 downto 16) + 16#1fbe#;
-- write it back
wr(av,tv,cmd);
-- read it for verification
rd(av,tv,cmd);
```

Presented by Abramov B.  
All right reserved

193

## Generate

**Generate** statements are provided as a convenient way to create multiple instances of concurrent statements (including processes), most typically component instantiation statements.

All **generate** statements must start with label.

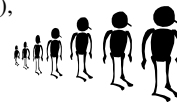
There are two basic varieties of **generate** statements:

- For generate

- ✓ All objects created are similar
- ✓ The generate parameter must be discrete and is undefined outside the generate statement
- ✓ Loop cannot be terminated early

- If generate

- ✓ Allows for conditional creation of logic
- ✓ Cannot use ELSE or ELSIF clauses with the IF generate



Presented by Abramov B.  
All right reserved

194

## Generate (cont)

The following example shows how you might use a **for-generate** statement to create four instances of a lower-level component (in this case a RAM block):

```
architecture generate_example of my_entity is
  component RAM16X1 is
    port(A0, A1, A2, A3, WE, D: in std_logic;
         O: out std_logic);
  end component RAM16X1;
begin
  ...
  RAMGEN: for i in 0 to 3 generate
    RAM: RAM16X1 port map ( ... );
  end generate RAMGEN;
end architecture generate_example;
```



Presented by Abramov B.  
All right reserved

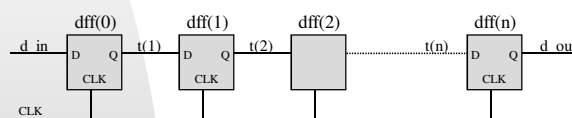
195

## Generate (cont)

The **if-generate** statement is most useful when you need to conditionally generate a concurrent statement.

A typical example of this occurs when you are generating a series of repetitive statements or components and need to supply different parameters, or generate different components, at the beginning or end of the series.

The following example shows how a combination of a **for-generate** statement and two **if-generate** statements can be used to describe a n-bit parity shift register constructed of cascaded DFF's:



architecture ...

signal t : std\_logic\_vector(1 to length-1);

begin

Presented by Abramov B.  
All right reserved

196

## Generate (cont)

```
g1: FOR i IN 0 TO (length-1) GENERATE
  g2: IF i=0 GENERATE
    dff : d_ff PORT MAP (d_in,clk,t(1));
  END GENERATE g2;
  g3: IF i>0 AND i<(length-1) GENERATE
    dff: d_ff PORT MAP (t(i),clk,t(i+1));
  END GENERATE g3;
  g4: IF i=(length-1) GENERATE
    dff : d_ff PORT MAP (t(i),clk,d_out);
  END GENERATE g4;
END GENERATE g1;
END ARCHITECTURE;
```

Presented by Abramov B.  
All right reserved

197

## Generate (cont)

The 16 bit adder with carry look ahead mechanism and  
speculative algorithm of group carry.

entity adder is

generic (width: natural:=16);

port(

a: in std\_logic\_vector((width-1) downto 0);

b: in std\_logic\_vector((width-1) downto 0);

s: out std\_logic\_vector(width downto 0)

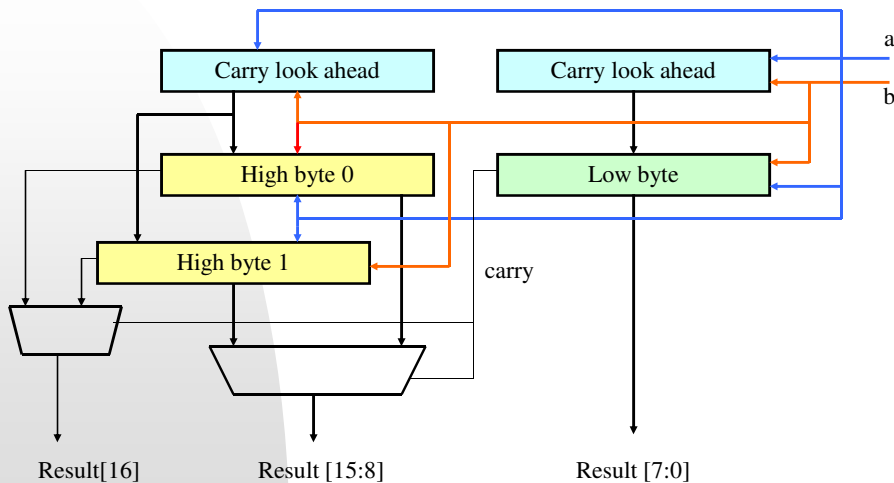
);

end entity adder;

Presented by Abramov B.  
All right reserved

198

## Generate (cont)



Presented by Abramov B.  
All right reserved

199

## Generate (cont)

```

architecture arc_adder of adder is
  attribute middle : integer;
  attribute middle of a : signal is a'length/2;
  function fa_sum( a_bit,b_bit,c_in: std_logic) return std_logic is
    variable sum :std_logic;
  begin
    sum:= a_bit xor b_bit xor c_in;
    return sum;
  end function fa_sum;
  function fa_carry( a_bit,b_bit,c_in: std_logic) return std_logic is
    variable carry : std_logic;
  begin
    carry:=(a_bit and b_bit) or (a_bit and c_in) or (b_bit and c_in);
    return carry;
  end function fa_carry;
  constant GND : std_logic:='0';
  constant VCC : std_logic:='1';

```

Presented by Abramov B.  
All right reserved

200

```

g1:for i in a'reverse_range generate
  lower_bit:if (i = a'low) generate
    temp(i)<=fa_sum(a(i),b(i),GND); g_l(i)<=a(i) and b(i);
  end generate lower_bit;
  others_bit: if (i > a'low) generate
    first_byte: if (i < a'middle) generate
      temp(i)<=fa_sum(a(i),b(i),g_l(i-1)); g_l(i)<=fa_carry(a(i),b(i),g_l(i-1));
    end generate first_byte;
    second_byte: if (i >= a'middle) generate
      second_byte_lower_bit: if (i = a'middle) generate
        temp_0(i - a'middle)<=fa_sum(a(i),b(i),GND);
        g_m_0(i - a'middle) <=fa_carry(a(i),b(i),GND);
        temp_1(i - a'middle)<=fa_sum(a(i),b(i),VCC);
        g_m_1(i - a'middle) <=fa_carry(a(i),b(i),VCC);
      end generate second_byte_lower_bit;
      second_byte_others_bit : if (i > a'middle) generate
        temp_0(i - a'middle)<=fa_sum(a(i),b(i),g_m_0(i- a'middle -1));
        g_m_0(i - a'middle) <=fa_carry(a(i),b(i),g_m_0(i- a'middle -1));
        temp_1(i - a'middle)<=fa_sum(a(i),b(i),g_m_1(i - a'middle -1));
        g_m_1(i - a'middle)<=fa_carry(a(i),b(i),g_m_1(i - a'middle -1));
      end generate second_byte_others_bit;
    end generate second_byte;
  end generate others_bit;
end generate g1;

```



Presented by Abramov B.  
All right reserved

201

## Generate (cont)

```

s(s'high)<= g_m_1(temp_1'high) when (g_l(g_l'high)=1') else g_m_0(temp_0'high);
s((s'high -1) downto a'middle)<=temp_1 when (g_l(g_l'high)=1') else temp_0;
s(temp'range)<=temp;
end architecture arc_adder;

```

Presented by Abramov B.  
All right reserved

202

# Generate & Recursion

```
entity mux_tree is
  generic (select_width : integer := 8);
  port
  (
    din  : in  std_logic_vector((2**select_width) -1) downto 0);
    sel  : in  std_logic_vector((select_width-1) downto 0);
    dout : out std_logic
  );
end entity mux_tree;
```

Step 1: Describe  
generic entity

Presented by Abramov B.  
All right reserved

203

```
architecture arc_mux_tree of mux_tree is
  component mux_tree is
    generic (select_width : integer := 5);
    port
    (
      din  : in  std_logic_vector((2**select_width) -1) downto 0);
      sel  : in  std_logic_vector((select_width-1) downto 0);
      dout : out std_logic
    );
  end component mux_tree;
begin
```

Step 2: Declare for  
component

Presented by Abramov B.  
All right reserved

204

```
-- mux 2x1 generation when select degrade until 1 bit
mux_2x1: if (sel'length=1) generate
    dout<=din(conv_integer(sel));
end generate mux_2x1;
```

Step 3: Denote the worst case condition

Presented by Abramov B.  
All right reserved

205

```
mux_tree_gen: if (sel'length/=1) generate
    signal temp : std_logic_vector(1 downto 0);
begin
    left_subtree: mux_tree
    generic map (select_width => (sel'length - 1))
    port map
    (
        din  => din(din'high downto (din'length/2)),
        sel  => sel((sel'high -1) downto sel'low),
        dout => temp(temp'high)
    );

    right_subtree: mux_tree
    generic map (select_width => (sel'length - 1))
    port map
    (
        din  => din(((din'length/2) - 1) downto din'low),
        sel  => sel((sel'high -1) downto sel'low),
        dout => temp(temp'low)
    );

    --mux 2x1 unites all the connected nodes
    dout<=temp(conv_integer(sel(sel'high)));
end generate mux_tree_gen;

end architecture arc_mux_tree;
```

Step 4: Build sub-trees by component calling

Presented by Abramov B.  
All right reserved

206

# Delay Mechanism

Delay is a mechanism allowing introducing timing parameters of specified systems.



Syntax:

`delay_mechanism ::= transport | [ reject time_expression ] inertial`

The delay mechanism allows introducing propagation times of described systems.

- Delays are specified in *signal assignment statements*.
- It is not allowed to specify delays in *variable assignments*.

Presented by Abramov B.  
All right reserved

207

## Delay Mechanism (cont)

There are two delay mechanism available in VHDL:

✓ inertial delay (default)

✓ transport delay.

Inertial delay is defined using the reserved word **inertial** and is used to model the devices, which are inherently inertial.

In practice this means, that impulses shorter than specified switching time are not transmitted.

Example:

`L_OUT <= inertial L_IN after 1 ns;`

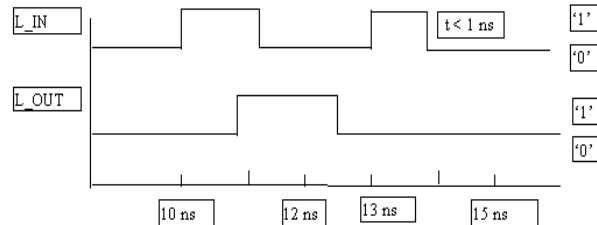
The signal value `L_IN` is assigned to the signal `L_OUT` with 1 ns delay. Not all changes of the signal `L_IN`, however, will be transmitted: if the width of an impulse is shorter than 1 ns then it will not be transmitted.

Presented by Abramov B.  
All right reserved

208



## Delay Mechanism (cont)



Presented by Abramov B.  
All right reserved

209

## Delay Mechanism (cont)

The transport delay is defined using the reserved word `transport` and is characteristic for transmission lines.

New signal value is assigned with specified delay independently from the width of the impulse in waveform.

Example:

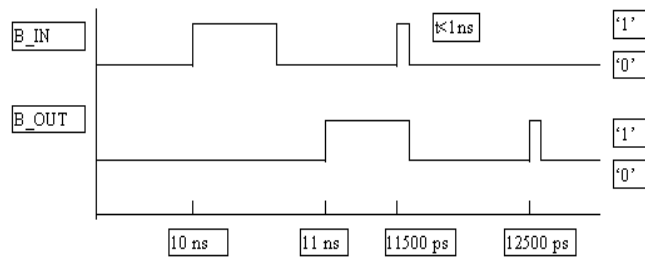
```
B_OUT <= transport B_IN after 1 ns;
```

The value of the signal B\_IN is assigned to the signal B\_OUT with 1 ns delay. The distance between subsequent changes of B\_IN is not important – all changes are transmitted to B\_OUT with specified delay.

Presented by Abramov B.  
All right reserved

210

## Delay Mechanism (cont)



Presented by Abramov B.  
All right reserved

211

## Delay Mechanism (cont)

Inertial delay specification may contain a reject clause.  
This clause can be used to specify the minimum impulse width that will be propagated, regardless of the switching time specified.

Example:

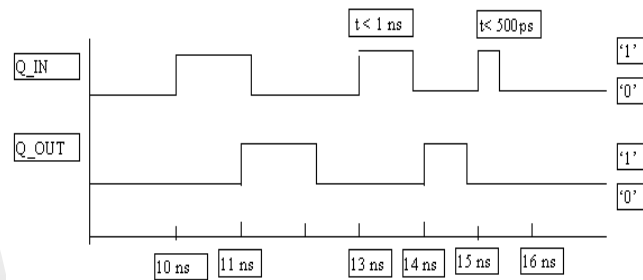
```
Q_OUT <= reject 500 ps inertial Q_IN after 1 ns;
```

The signal value Q\_IN is assigned to the signal Q\_OUT with 1 ns delay.  
Although it is an inertial delay with switching time equal to 1 ns,  
the reject time is specified to 500 ps  
and only impulses shorter than 500 ps will not be transmitted.

Presented by Abramov B.  
All right reserved

212

## Delay Mechanism (cont)



Presented by Abramov B.  
All right reserved

213

## Delay Mechanism (cont)

How this process works?

non\_deterministic: **process** (a) is

**constant** t\_01 : **time** := 800 ns;

**constant** t\_10 : **time** := 500 ns;

**begin**

**if** (a='1') **then**

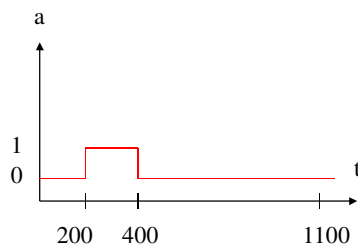
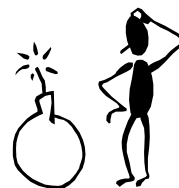
$z \leq \text{transport } a \text{ after } t_{01};$

**else**

$z \leq \text{transport } a \text{ after } t_{10};$

**end if;**

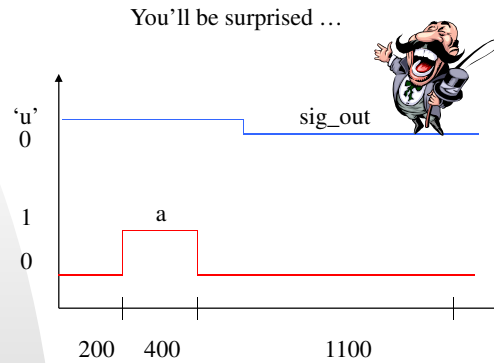
**end process** non\_deterministic;



Presented by Abramov B.  
All right reserved

214

## Delay Mechanism (cont)



Presented by Abramov B.  
All right reserved

215

## Blocks

A block statement (which is concurrent) contains a set of concurrent statements. The order of the concurrent statements does not matter, because all statements are always executing.

The syntax of a block statement is

```
label: block [ (expression) ]
        { block_declarative_item }
```

```
begin
        { concurrent_statement }
```

```
end block [ label ];
```

label => The label, which is required, names the block.

expression => The **guard** condition for the block.

Presented by Abramov B.  
All right reserved

216

## Blocks (cont)

Objects declared in a block are visible to that block and to all blocks nested within it.

When a child block (nested inside a parent block) declares an object with the same name as an object in the parent block, the child block's declaration overrides that of the parent.



Presented by Abramov B.  
All right reserved

217

## Blocks (cont)

```
B1: block
  signal S: bit; -- Declaration of "S" in block B1
begin
  S <= A and B; -- "S" from B1
  B2: block
    signal S: bit; -- Declaration of "S", block B2
    begin
      S <= C and D; -- "S" from B2
      B3: block
        begin
          Z <= S; -- "S" from B2
        end block B3;
      end block B2;
      Y <= S; -- "S" from B1
    end block B1;
```

Presented by Abramov B.  
All right reserved

218

## Blocks (cont)

A block can also have a **GUARD** expression.

In that case, an assignment inside the block that contains the keyword **GUARDED** will only be executed when the **GUARD** expression is **TRUE**.

**architecture** RTL of EG1 is

**begin**

    guarded\_block: **block** (a = '1')

**begin**

        z <= '1' **when guard** else '0';

**end block** guarded\_block;

**end architecture** RTL ;

Presented by Abramov B.  
All right reserved

219

## Blocks (cont)

Flip-flops and registers can also be generated with dataflow statements

(as opposed to from a process) using a **GUARDED** block.

**signal** input\_sig, output\_sig, clk : **bit** ;

b1 : **block** (clk'event and clk='1')

**begin**

    output\_sig <= **guarded** input\_sig ;

**end block** b1;

Presented by Abramov B.  
All right reserved

220

## Blocks (cont)

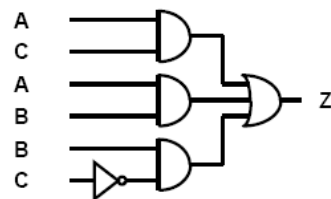
### Problem



### • Code

```
Z <=
  (A and C) or
  (A and B) or
  (B and not C) ;
```

### • Required Circuit:



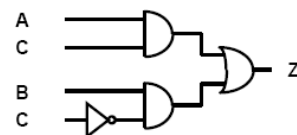
Presented by Abramov B.  
All right reserved

221

## Blocks (cont)



### • Resulting Circuit:



### • What happened?

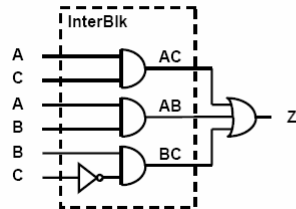
- The term AB is reducible and was removed by the synthesis tool.
- ✦ Is it necessary to keep AB?
- ✦ For asynchronous logic, the AB term prevents glitches

Presented by Abramov B.  
All right reserved

222

## Blocks (cont)

- Step 1: Create Hierarchy Using VHDL Block Statement



```
InterBlk : block
begin
    AC <= A and C ;
    AB <= A and B ;
    BC <= B and not C ;
end block ;

Z <= AC or AB or BC ;
```

Presented by Abramov B.  
All right reserved

223

## Blocks (cont)

- ♦ Step 2: Apply the CREATE\_HIERARCHY attribute to the block statement label to preserve it through synthesis.

```
attribute CREATE_HIERARCHY of InterBlk : label is TRUE;
```



Presented by Abramov B.  
All right reserved

224



# Bus Guarded Signal



```
-- this function uses an input bit_vector that is called anytime that an
-- assignment is made to the bus guarded signal en_bit (including null).
-- because en_bit is driven by multiple drivers, multiple bits are passed in.
function resolve_bit_bus (input_bits: bit_vector) return bit is
    variable result : bit := '0'; -- Default value
begin
    for i in input_bits'range loop
        if input_bits(i) = '1' then
            result := '1';
        end if;
    end loop; -- i
    return result;
end function resolve_bit_bus;

signal en_bit : resolve_bit_bus bit bus;
```

Presented by Abramov B.  
All right reserved

225

# Bus Guarded Signal

- The resolution function, also known as subtype\_indicator, will accept multiple calls from the guarded signal en\_bit.
- This resolution function will be called any time that an assignment is made to en\_bit.
- Thus, we are expecting multiple drivers, so we need an input of type bit\_vector.
- The resolution function then resolves the final output of en\_bit.
- Specifying the signal en\_bit as a bus, indicates that it is a guarded signal.
- Thus, the subtype\_indicator is used to resolve the output of the guarded signal.

Presented by Abramov B.  
All right reserved

226

## Bus Guarded Signal

```
process (a, b)
begin
  if a = '1' and b = '1' then
    en_bit <= '1';
  else
    en_bit <= null; -- disconnect driver
  end if;
end process;

process (c, d)
begin
  if c = '1' and d = '1' then
    en_bit <= '1';
  else
    en_bit <= null; -- disconnect driver
  end if;
end process;
```

Presented by Abramov B.  
All right reserved

227

## Bus Guarded Signal

- In a multiple process, the en\_bit is assigned to, creating multiple drivers.
- Thus, because more than one driver may be active at any time, the resolution function is called to resolve the final output.
- If no drivers are connected, the resolution function will assign '0'.
- Thus, the resolution function is called for all assignments to this signal, including assignments of null.
- **Key point to remember for bus guarded signals:**
- When all drivers are disconnected, the resolution function is called and returns a default value.
- For the following example, the default value is "0".
- This only happens for a bus guarded signal.
- A register signal would simply retain its previous value

Presented by Abramov B.  
All right reserved

228

# Register Guarded Signal

```
-- this function uses an input bit_vector that is called anytime that an
assignment is made to the register guarded-signal en_bit (excluding null).
-- because en_bit is driven by multiple drivers, multiple bits are passed in.
function resolve_bit_register (input_bits: bit_vector) return bit is
    variable result : bit := '0'; -- this is no longer a default value, but will
                                   be returned if all drivers are driving '0'
begin
    for i in input_bits'range loop
        if input_bits(i) = '1' then -- if any bit being passed in is equal to 1,
                                   the signal value will be resolved to 1
            result := '1';
        end if;
    end loop; -- i
    return result;
end function resolve_bit_register;

signal en_bit : resolve_bit_register bit register; . . .
```

- Key point to remember for register guarded signals : When all drivers are disconnected, the resolution function is NOT called, and therefore retains its previous value.

Presented by Abramov B.  
All right reserved

229

# Signal Assignment Block

```
-- resolution function (subtype_indicator)
function resolve_bit_bus (input_bits: bit_vector) return bit is
    . . .
signal en_bit : resolve_bit_bus bit bus; -- guarded signal
    . . .
block_ab: block (a = '1' and b = '1')
begin
    -- "guarded" indicates this driver is disconnected when the
    -- guard_expression (a.k.a. implicitly defined guard signal) is False
    en_bit <= guarded '1';
end block block_ab;

block_cd: block (c = '1' and d = '1')
begin
    -- "guarded" indicates this driver is disconnected when the
    -- guard_expression (a.k.a. implicitly defined guard signal) is False
    en_bit <= guarded '1';
end block block_cd;
```

Presented by Abramov B.  
All right reserved

230

## Signal Assignment Block

- Guarded signal assignment blocks are one of the most advantageous uses of a block.
- It is the only way to make a concurrent assignment to a guarded signal because null cannot be assigned in a normal concurrent assignment.
- This recreates the same guarded signal assignment as the previous bus guarded signal example.
- The keyword `guarded` is used to indicate that there are multiple drivers for this signal.
- If the signal being assigned to is a guarded signal, and the guard expression is `FALSE`, the driver is disconnected.

Presented by Abramov B.  
All right reserved

231

## Explicitly Defined Guard

```
block_ab: block
  signal guard : boolean;
begin
  guard_proc_ab: process (a, b)
  begin -- process guard_proc
    if a = '1' and b = '1' then
      guard <= true;
    else
      guard <= false;
    end if;
  end process guard_proc_ab;

  en_bit <= guarded '1';
end block block_ab;
```

Presented by Abramov B.  
All right reserved

232

## Guarded Signals Delay

- In a process you may assign “null” with a delay specification :  
`en_bit <= null after 4 ns;`
- **Null cannot be used in concurrent block assignments**, so VHDL provides a disconnect specification :  
`signal en_bit : resolve_bit_bus bit bus;`  
`-- when guard is FALSE, en_bit is disconnected after 4 ns`  
`disconnect en_bit : resolve_bit_bus bit after 4 ns;`
- Rather than specifying the delays for all signals separately (assuming they all have the same delays and are of the same type), you can use the keyword `all` :  
`disconnect all : resolve_bit_bus bit after 4 ns;`
- Or if some have the same delays (say, 4 ns), while other signals should have varying delays, you can assign individual disconnect specifications for some signals and use the keyword `others` to describe the delays for the remaining signals (again, assuming they are the same type).

Presented by Abramov B.  
All right reserved

233

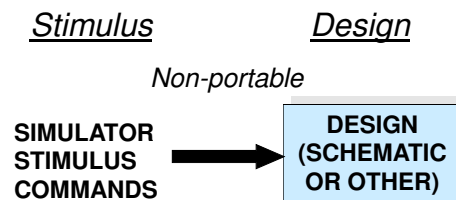
## Guarded Signals Delay

```
signal en_bit0, en_bit1, en_bit2, en_bit3, en_bit4, en_bit5 : resolve_bit_bus bit bus;
disconnect en_bit0 : resolve_bit_bus bit after 2 ns;
disconnect en_bit1 : resolve_bit_bus bit after 1 ns;
disconnect en_bit2 : resolve_bit_bus bit after 3 ns;
disconnect others : resolve_bit_bus bit after 4 ns;
```

Presented by Abramov B.  
All right reserved

234

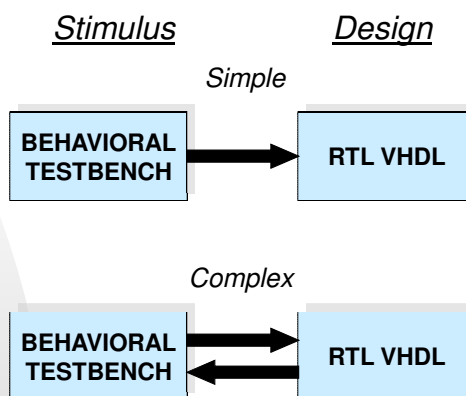
# Simulation Levels



Presented by Abramov B.  
All right reserved

235

# Simulation Levels



Presented by Abramov B.  
All right reserved

236

# Test Methodology

**Direct Test** – Designer knows the architecture of DUT (white box) and creates test stimulus adapted for DUT

Advantage: fast result, small set of test vectors.  
Disadvantage: poorly coverage of all possible scenarios.  
Usability: single module test cases.

**Random Test** – Test vectors generated in disregard to architecture requirements.

Advantage: large set of test vectors, easy to create erroneous scenarios.  
Disadvantage: poorly productivity, only small subset of generated test vectors may be used  
Usability: system level simulation.

Presented by Abramov B.  
All right reserved

237

# Test Methodology

**Semi Random Test** – Designer knows the architecture of DUT, but creates test stimulus adapted for DUT using random mechanism for generation of erroneous scenarios.

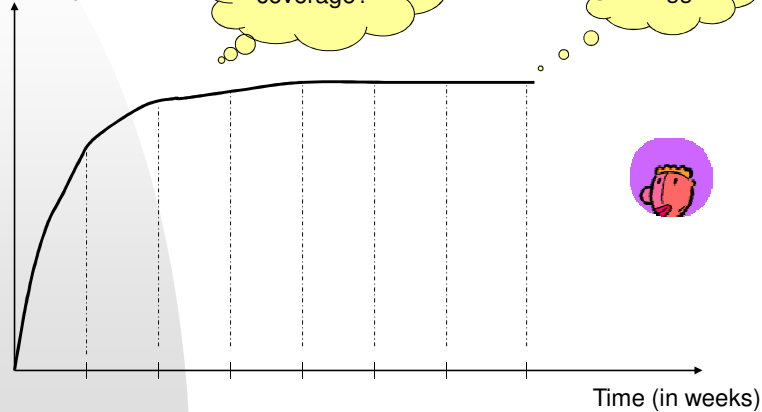
Advantage: Better productivity and reliability.  
Disadvantage: complex to specify and implementation  
Usability: single module test cases, the next step after direct test.

Presented by Abramov B.  
All right reserved

238

# Test Methodology

Number of detected bugs



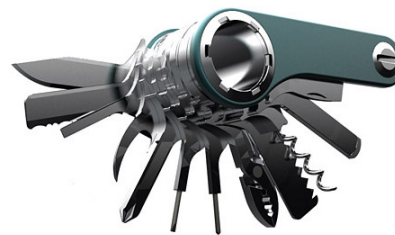
Presented by Abramov B.  
All right reserved

239

# Test Methodology

Useful “tools” for more testability and test productivity.

- ✓ Assert statements
- ✓ Interactive I/O to/from simulator shell
- ✓ Test vectors file and log files
- ✓ Vital library
- ✓ OVL library
- ✓ Bus Functional Models
- ✓ Third party models
- ✓ PLI interface with foreign languages models



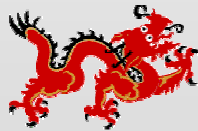
Presented by Abramov B.  
All right reserved

240



# Assertion Based Design

- ✓ The EDA industry has acknowledged that functional verification is causing a bottleneck in the design process and is inadequately equipped to deliver verification of future designs.
- ✓ Designs can no longer be sufficiently verified by ad-hoc testing and monitoring methodologies.
- ✓ More and more designs incorporate complex functionalities, employ reusable design components, and fully utilize the multi-million gate counts offered by chip vendors.



Presented by Abramov B.  
All right reserved

241

# Assertion Based Design (cont)

- ✓ The OVL is composed of a set of assertion monitors that verify specific properties of a design.
- ✓ These assertion monitors are instantiated in the design establishing a single interface for design validation.
- ✓ Vendors are expected to provide tool-specific libraries compliant with the terms of the OVL standard.



Presented by Abramov B.  
All right reserved

242

## Assertion Based Design (cont)

Assertion monitors are instances of modules whose purpose in the design is to guarantee that some conditions hold true.

Assertion monitors are modeled after VHDL assertions: they are composed of an event, message, and severity.

- Event is a property that is being verified by an assertion.  
An event can be classified as a temporal or static property.  
(A static property is a property that must be valid at all times, whereas a temporal property is a property that is valid during certain times.)
- Message is the string that is displayed in the case of an assertion failure.
- A severity represents whether the error captured by the assertion library is a major or minor problem.

Presented by Abramov B.  
All right reserved

243

## Assertion Based Design (cont)

The OVL library supports the following assertions.

assert_always	assert_no_underflow
assert_always_on_edge	assert_odd_parity
assert_change	assert_one_cold
assert_cycle_sequence	assert_one_hot
assert_decrement	assert_proposition
assert_delta	assert_quiescent_state
assert_even_parity	assert_range
assert_fifo_index	assert_time
assert_frame	assert_transition
assert_handshake	assert_unchange
assert_implication	assert_width
assert_increment	assert_win_change
assert_never	assert_win_unchange
assert_next	assert_window



Presented by Abramov B.  
All right reserved

244

## Assertion Based Design (cont)

Usage the assert\_decrement assertion should be used in circuits to ensure the proper change of values in structures such as counters and finite-state machines (FSM).



```
test_expr <= to_unsigned(count);
ok1: assert_decrement GENERIC MAP (1,4,1)
PORT MAP
(clk, resetn, test_expr);
PROCESS IS
BEGIN
    WAIT UNTIL (clk'EVENT AND clk = '1');
    IF resetn = '0' THEN
        count <= 0;
    ELSIF count = 0 THEN
        count <= 9;
    ELSE
        count <= count - 1;
    END IF;
END PROCESS;
```

Presented by Abramov B.  
All right reserved

245

## Assertion Based Design (cont)

The assert\_even\_parity assertion is most useful for control and datapath circuits.

This assertion ensures that

a variable or expression has an even number of bits asserted.

Some example uses of assert\_even\_parity are:

- address or data busses with error checking based on parity
- finite-state machines (FSM) with error detection mechanisms:

```
ok1: assert_even_parity
GENERIC MAP (0, 4)
PORT MAP
(
    clk      => clk,
    reset_n  => resetn,
    test_expr => stdv_to_unsigned (addr)
);
```

Presented by Abramov B.  
All right reserved

246

## Assertion Based Design (cont)

The assert\_handshake assertion continuously monitors the req and ack signals at every positive edge of the triggering event or clock clk.

It asserts that they follow the handshaking protocol specified by the parameters noted below.

Note that both req and ack must go inactive (0) prior to starting a new handshake validation sequence.

Optional checks can be performed as follows:

- ❑ min\_ack\_cycle: When this parameter is greater than zero, the check is activated. It checks whether an ack occurs at or after min\_ack\_cycle clocks.
- ❑ max\_ack\_cycle: When this parameter is greater than zero, the check is activated. It checks whether an ack occurs at or before max\_ack\_cycle clocks.
- ❑ req\_drop: When this parameter is greater than zero, the check is activated. It checks whether req remains active for the entire cycle until an ack occurs.

Presented by Abramov B.  
All right reserved

247

## Assertion Based Design (cont)

```
ok1: assert_handshake
GENERIC MAP (0, 0, 0, 1, 1)
PORT MAP (clk, resetn, req, ack_i);
p1: process (present_state , req) is
begin
  ack_i <= '0';
  case present_state is
    when s0 =>
      IF req = '1' THEN
        next_state <= s1;
      ELSE next_state <= s0;
      END IF;
    when s1 => next_state <= s2;
    when s2 => next_state <= s3;
    when s3 =>
      ack_i <= '1';
      next_state <= s0;
    end case;
  end process p1;
```



Presented by Abramov B.  
All right reserved

248

## Assertion Based Design (cont)

The `assert_fifo_index` assertion ensures that a FIFO-type structure will never overflow nor underflow.

This monitor can be configured to support multiple pushes (writes) into and pops (reads) from a fifo within a given cycle.

This example shows how `assert_fifo_index` can be used to verify that a FIFO will never overflow nor underflow.

This example uses the default parameter settings for the `push_width` and `pop_width` (that is, only one push and pop can occur on a given cycle).

Note: It is possible for a push and a pop to occur on the same cycle.

Ok1: `assert_fifo_index`

**GENERIC MAP** (0,16)

**PORT MAP** (clk, reset\_n, push, pop);

Presented by Abramov B.  
All right reserved

249

## Test Bench Boosting

- Use whenever possible *variables* instead of signals
- In models of memory use the *shared variables*
- Use *assertion* statements at a RTL code
- Try to use as small as possible *std\_logic* type for internal signals and variables
- Prefer to use *bit/bit\_vector*, *integer* and *boolean* types for internal signals and variables
- Use type conversion functions to reduce memory space for *std\_logic* type (*to\_bit*, *to\_bitvector*, *to\_X01*, *to\_X01Z*, *conv\_integer*).
- Use BFM to connects between internal blocks of model.
- Try to use the high level of abstraction ( records , enumeration types, disconnect models) instead of always dealing with low-level zeros and ones
- No zero delays !!!
- Minimize the number of processes with sensitivity list



Presented by Abramov B.  
All right reserved

250

# Test Bench Boosting

It is a common test benches practice to use a procedures with wait on clock to allow a specific amount of time to pass.

```
procedure wait_n ( constant number_of_clock_cycles : natural;  
                  signal clock : in std_logic) is  
  
begin  
    for I in 1 to number_of_clock_cycles loop  
        wait until rising_edge (clock);  
    end loop;  
end procedure wait_n;
```

## NOTE !!!

*While this loop is not complicated, the Performance Analyzer may identify the “wait” line as a bottleneck. The reason for this is the proliferation of processes waiting for signal events, even though the action taken by each process is minimal.*

Presented by Abramov B.  
All right reserved

251

# Test Bench Boosting

*Although slightly more obscure, the following fragment accomplishes the same behavior but the performance consequence is minimized.*

```
procedure wait_n ( constant number_of_clock_cycles : natural;  
                  signal clock : in std_logic;  
                  constant clock_period : time :=C_CLOCK_P) is  
  
begin  
    wait for ((number_of_clock_cycles * clock_period) - 1 ns);  
    wait until rising_edge (clock);  
end procedure wait_n;
```

Presented by Abramov B.  
All right reserved

252

# Shared Variables

VHDL87 limited the scope of the variable to the process in which it was declared. Signals were the only means of communication between processes, but signal assignments require an advance in either delta time or simulation time.

VHDL '93 introduced shared variables which are available to more than one process.

Like ordinary VHDL variables, their assignments take effect immediately. However, caution must be exercised when using shared variables because multiple processes making assignments to the same shared variable can lead to unpredictable behavior if the assignments are made concurrently.



The VHDL '93 standard does not define the value of a shared variable if two or more processes make assignments in the same simulation cycle.

Presented by Abramov B.  
All right reserved

253

# Write collision with ShV

```
architecture write_collision_example of write_collision is  
  shared variable count : integer;
```

```
begin
```

```
  p1: process is  
    begin
```

```
    .....
```

```
    .....
```

```
    count:=a;
```

```
    wait on b;
```

```
  end process p1;
```

```
  p2: process is
```

```
    begin
```

```
    .....
```

```
    .....
```

```
    count:=10;
```

```
    wait;
```

```
  end process p2;
```

```
end architecture write_collision_example;
```



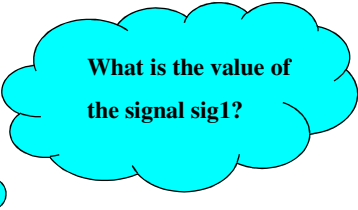
Presented by Abramov B.  
All right reserved

254

## Read contradiction with ShV?

```
architecture read_contradiction_example of read_contradiction is
  shared variable sv : integer;
begin
  p1: process (clk) is
  begin
    if rising_edge(clk) then
      sv:=a;
      .....
    end if;
  end process;

  p2: process (clk) is
  begin
    if rising_edge(clk) then
      .....
      sig1<=sv;
    end if;
  end process;
end architecture;
```



What is the value of  
the signal sig1?

Presented by Abramov B.  
All right reserved

255

## Shared Variables (cont)

A shared variable is best used for system level modeling  
and object-oriented programming !

The following package uses shared variable to make the stack  
available to more than one procedure.

The package declarative region is shown here declaring two  
procedures, push and pop.

```
package int_stack_pack is
  procedure pop (what : in integer);
  procedure push(what : out integer);
end package int_stack_pack;
```

Presented by Abramov B.  
All right reserved

256



## Shared Variables (cont)

As an example of where shared variables are useful,  
the `int_stack_pack` in this and the next slide uses a shared variable in two  
procedures used to maintain a stack.

The designer is responsible for ensuring that no two processes call  
these two procedures at any one time.

```
package body int_stack_pack is  
    type int_stack is array (0 to 100) of integer;  
    shared variable stack : int_stack;  
    shared variable index : natural:=0;
```

Presented by Abramov B.  
All right reserved

257

## Shared Variables (cont)

```
procedure pop (what : out integer) is  
begin  
    index:=index-1;  
    what:=stack(index);  
end procedure pop;  
procedure push (what : in integer) is  
begin  
    stack(index):=what;  
    index:=index+1;  
end procedure push;  
end package body int_stack_pack;
```

Presented by Abramov B.  
All right reserved

258

# Protected Type

We achieve mutual execution for a shared variable by declaring the Variable to be of a protected type.

There are two parts to the definition of a protected type:

- protected type declaration
- protected type body

Syntax: **type** t\_name **is protected**

declaration area

**end protected** t\_name;

**type** t\_name **is protected body**

implementation area

**end protected body** t\_name;

Presented by Abramov B.  
All right reserved

259

## Protected Type (cont)

A simple example of protected type:

```
type shared_counter is protected  
    procedure reset;  
    procedure increment (by : integer:=1);  
    impure function value return integer;  
end protected shared_counter ;
```



We can declare a shared variable to be of this type using a shared variable declaration:

**shared variable** counter : shared\_counter;

Now in process we can use the name of shared variable as a prefix to a method name :

```
counter.reset; counter.increment(3);  
if(counter.value>50) then  
    counter.reset;  
end if;
```

Presented by Abramov B.  
All right reserved

260

## Protected Type (cont)

We can implement the shared counter protected type as follows:

```
type shared_counter is protected body
    variable count: integer:=0;
    procedure reset is
    begin
        count:=0;
    end procedure reset;
    procedure increment (by : integer:=1) is
    begin
        count:=count+by;
    end procedure increment;
    impure function value return integer is
    begin
        return count;
    end function value;
end protected body shared_counter;
```

Presented by Abramov B.  
All right reserved

261

## Protected Type (cont)

- Only method names declared in the protected type declaration are visible outside the protected type definition.
- If a protected type is declared in a package declaration, the protected type body must be declared in the corresponding package body.
- Only variables and variable-class subprogram parameters can be of protected types.
- Shared variables must be of protected types.
- Protected types cannot be used as elements of files, as elements of composite types, or as types designated by access types.
- Variable assignment of one protected-type variable to another is not allowed.
- The equality (“=”) and inequality (“/=”) operators are not predefined for protected types.
- A protected type methods must not include or execute a wait statements.
- A function method must be declared as impure.

Presented by Abramov B.  
All right reserved

262

## Protected Type (cont)

```
entity dual_port_reg is
    generic (width : natural);
    port (    read_clk : in bit;
            write_clk : in bit;
            read_data : in bit_vector(width-1 downto 0);
            write_data : out bit_vector(width-1 downto 0)
    );
end entity dual_port_reg;
```

Presented by Abramov B.  
All right reserved

263

## Protected Type (cont)

```
architecture behavioral of dual_port_reg is
    subtype word is bit_vector((width-1) downto 0);
    type two_port_reg is protected
        impure function get return word;
        procedure set(new_value :in word);
    end protected two_port_reg ;
    type two_port_reg is protected body
        variable reg : word;
        impure function get return word is;
            return reg;
        end function get;
    end type two_port_reg;
```

Presented by Abramov B.  
All right reserved

264

## Protected Type (cont)

```
procedure set ( new_value : in word) is
begin
    reg:=new_value;
end procedure set;
end protected body two_port_reg ;
shared variable buff : two_port_reg ;
begin
    read: process(read_clk) is
    begin
        if read_clk'event and read_clk='1' then
            read_data<=buff.get;
        end if;
    end process read;
```

Presented by Abramov B.  
All right reserved

265

## Protected Type (cont)

```
write: process(write_clk) is
    begin
        if write_clk'event and write_clk='1' then
            buff.set(write_data);
        end if;
    end process write;
end architecture behavioral;
```



Presented by Abramov B.  
All right reserved

266

# Access Type

These are similar to pointer types found in many programming languages. In VHDL access types are used mainly in high-level behavioral models.

We can declare an access type using a new form of type definition, given by the syntax rule:



access\_type\_definition <= **access** [typesubtype indication];

For example :

```
type int_ptr is access integer;
```

This defines a new type, named int\_ptr, representing values that point to data objects of type integer.

Once we have declared an access type, we can declare a variable of that type within a process or subprogram:

```
variable count : int_ptr;
```

Presented by Abramov B.  
All rights reserved

267

## Access Type (cont)

- Application of access types is restricted to variables:
- ✓ Only variables can be of an access value.
- ✓ Also, only variables can be designated by an access value.
- Although, access types are very useful for modeling potentially large structures, like memories or FIFO, they are not supported by synthesis tools.
- The default value of an access type is **null**.
- To assign any other value to an object of an **access** type an *allocator* (**new**) has to be used.

Presented by Abramov B.  
All rights reserved

268

## Access Type (cont)

Next, we can create a new integer number data object and set count points to it:

```
count:=new integer;
```

This example shows that an *allocator*, written using the keyword **new**.

We access the object using the keyword **all** by this way:

```
count.all:=10;
```

We may use its value:

```
variable my_int : integer;  
my_int:=count.all;  
if (count.all>0) then .....
```

Presented by Abramov B.  
All right reserved

269

## Access Type (cont)

The second form of allocator, uses a *qualified expression*:

```
count:=new integer'(10);
```

thus, instead of writing the two statements:

```
count:=new integer;  
count.all:=10;
```

Example :

```
type stimulus is record  
    stimulus_time : time;  
    stimulus_value : bit_vector(3 downto 0);  
end record stimulus;  
type stimulus_ptr is access stimulus;  
variable bus_stimulus : stimulus_ptr;  
bus_stimulus:=new stimulus'(100 ns, x"d");
```

Presented by Abramov B.  
All right reserved

270

## Access Type (cont)

One very useful pointer comparison is the test for equality with **null**, the special pointer value that does not point to any object.

```
if (count/=null) then
    count.all:=count.all + 1;
end if;
```

One of the advantages of using access types that points to array objects is that we can deal with arrays of mixed length.

Canonical way :

```
type time_arr is array (positive range <>) of time;
variable activation_time : time_arr(1 to 10);
activation_time:=(1=>100 ns, 10 us, 15 us, others=> 25 us);
```

Dynamic way:

```
type time_arr_ptr is access time_arr;
variable activation_time : time_arr_ptr;
activation_time:=new time_arr'(100 ns, 10 us, 15 us);
activation_time:=new time_arr'(activation_time.all & time_arr'(70 us,150 ns));
```

Presented by Abramov B.  
All right reserved

271

## Access Type (cont)

**process is**

```
variable l : line; -- defined as string access
```

**begin**

```
wait until rising_edge(clk);
```

```
if (wr_n = '0') then
```

```
    l := new string'("writing data => ");
```

```
    hwrite(l, data);
```

```
    writeline(output, l);
```

```
    deallocate(l);
```

```
end if;
```

**end process;**

Presented by Abramov B.  
All right reserved

272



## Access Type (cont)

We wish to store a list of value to simulate a signal during a simulation.  
One possible approach would be to define an array variable of stimulus values, but if we make it too small ,we may run out of space,if we make it too large ,we may waste space in the host computer's memory.  
The alternative approach is to use access types and create value only as they are needed.

### Linked list:

```
type value_cell; --incomplete type declaration to solve "chicken and egg" problem
type vcell_ptr is access value_cell;
type value_cell is record -- complete type declaration
    value : bit_vector(7 downto 0);
    next_cell : vcell_ptr;
end record value_cell;
```

Now we can declare an access to point to the beginning of the list:

```
variable head,ccell,nnode : vcell_ptr;
```

Presented by Abramov B.  
All right reserved

273

## Access Type (cont)

We can add a cell to the list by allocating a new record.

### List creating:

```
head:=new vcell_ptr'(x"a3",null) ;
ccell:=head;
```

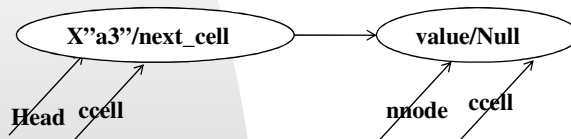
### **while (...)** loop

```
nnode:=new vcell_ptr'(new_val, null); OR ccell.next_cell:=new vcell_ptr'(new_val,null);
ccell.next_cell:=nnode;
ccell:=nnode;
```

### **end loop;**

### **while (...)** loop

```
ccell.next_cell:=new vcell_ptr'(new_val,null);
ccell:=ccell.next_cell;
end loop;
```



Presented by Abramov B.  
All right reserved

274

## Access Type (cont)

### Read list:

```
ccell:=head;  
while (ccell/=null) loop  
    s<=ccell.value;  
    wait until rising_edge(clk);  
    ccell:=ccell.next_cell;  
end loop;
```

### Search for value:

```
ccell:=head;  
while (ccell/=null) loop  
    ccell:=ccell.next_cell;  
    exit when ccell.value=search_value;  
end loop;  
assert (ccell/=null)  
    report "search for value failed";
```

Presented by Abramov B.  
All right reserved

275

## Access Type (cont)

The mechanism in VHDL provides for us to do memory free is the implicitly defined procedure **deallocate**.

Whenever we declare an access type, VHDL automatically provides an overloaded version of deallocate to handle pointers of that type.

```
type t_ptr is access t;
```

we automatically get a version of **deallocate** declared as

```
procedure deallocate (p : inout t_ptr);
```

If p is **null** to start with, the procedure has *no effect*.

Presented by Abramov B.  
All right reserved

276

## Access Type (cont)

For delete cell from list of stimulus values :

```
cell_to_be_deleted=value_list;  
value_list:=value_list.next_cell;  
deallocate(cell_to_be_deleted);
```

If we wish to delete the whole list, we can use a loop

```
while (value_list/=null) loop  
    cell_to_be_deleted=value_list;  
    value_list:=value_list.next_cell;  
    deallocate(cell_to_be_deleted);  
end loop;
```

Presented by Abramov B.  
All right reserved

277

## Finite State Machine

- A **finite state machine** (FSM) or **finite automaton** is a model of behavior composed of states and transitions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition.

Presented by Abramov B.  
All right reserved

278

# FSM Definition

Defined:

$x = \{x_1, x_2, \dots, x_n\}$  - set of input signals

$A = \{a_1, a_2, \dots, a_m\}$  - set of states

$y = \{y_1, y_2, \dots, y_k\}$  -set of output signals

The transition function -  $\delta$

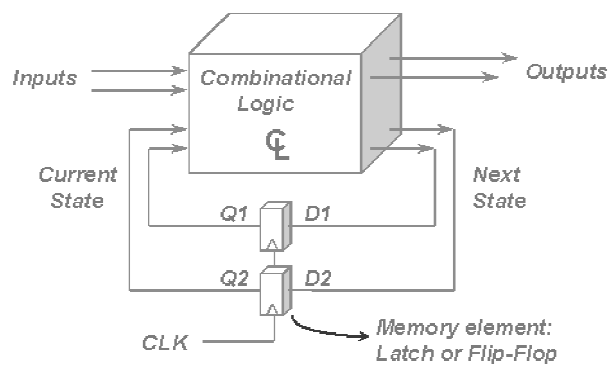
The output function -  $\lambda$

The initial state -  $a_1$

Presented by Abramov B.  
All right reserved

279

# FSM's Hardware Representation

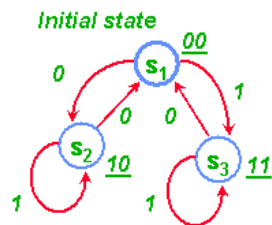


Presented by Abramov B.  
All right reserved

280

# FSM Functional Representation

State Transition Graph



t t+1 t+2  
 Inputs: 0 1 0  
 Outputs: 00 \_\_\_\_\_

State Transition Table

T(S, I)			O(S)	
	0	1		
s <sub>1</sub>	s <sub>1</sub>	s <sub>2</sub>	s <sub>1</sub>	00
s <sub>2</sub>	s <sub>1</sub>	s <sub>3</sub>	s <sub>2</sub>	10
s <sub>3</sub>	s <sub>2</sub>	s <sub>1</sub>	s <sub>3</sub>	11

Presented by Abramov B.  
All right reserved

281

## Berkley KISS format

```

.i 2
.o 2
.p 24      11 st2 st3 00
.s 6      00 st3 st4 00
00 st0 st0 00 01 st3 st3 01
01 st0 st1 00 10 st3 st3 10
10 st0 st1 00 11 st3 st3 11
11 st0 st1 00 00 st4 st5 00
00 st1 st0 00 01 st4 st4 00
01 st1 st2 00 10 st4 st4 00
10 st1 st2 00 11 st4 st4 00
11 st1 st2 00 00 st5 st0 00
00 st2 st1 00 01 st5 st5 00
01 st2 st3 00 10 st5 st5 00
10 st2 st3 00 11 st5 st5 00
  
```

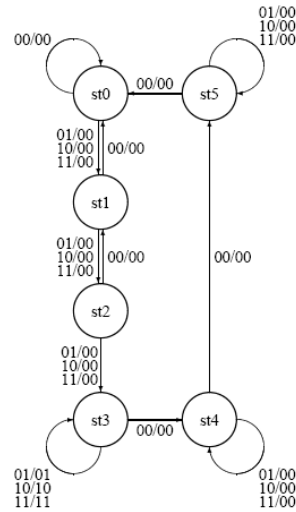
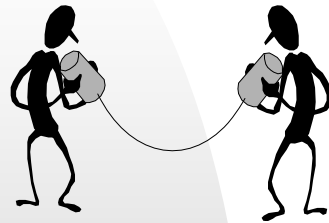
Where

*.i* <number of inputs>  
*.o* <number of outputs>  
 (zero, for automata)  
*.p* <number of product terms>  
*.s* <number of states>

Presented by Abramov B.  
All right reserved

282

# State Transition Graph

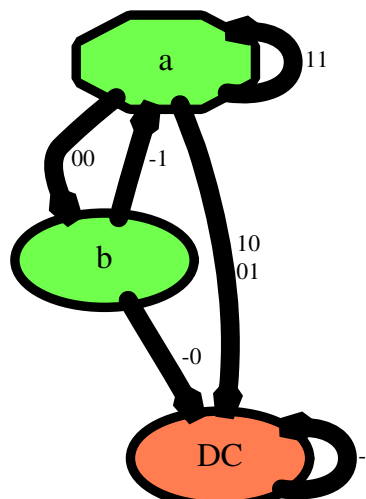


Presented by Abramov B.  
All right reserved

283

# KISS vs. Bubble Diagram

.i 2  
.o 0  
.s 3  
.p 7  
11 a a  
00 a b  
10 a dc  
01 a dc  
-1 b a  
-0 b dc  
-- dc dc



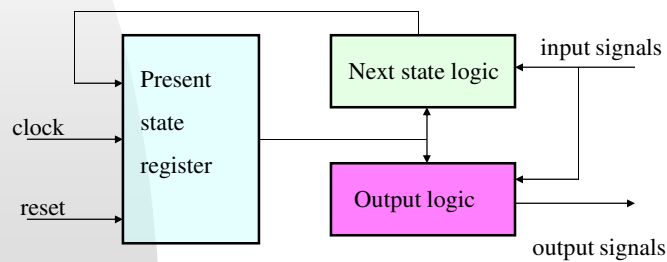
Presented by Abramov B.  
All right reserved

284

# FSM forms

Many designs expressed at the register-transfer level consist of combinatorial data paths controlled by *finite-state-machines* -> *FSM*. There are basically two forms of state machines, Mealy machines and Moore machines.

In a Mealy machine, the outputs depend directly on the present state and the inputs.

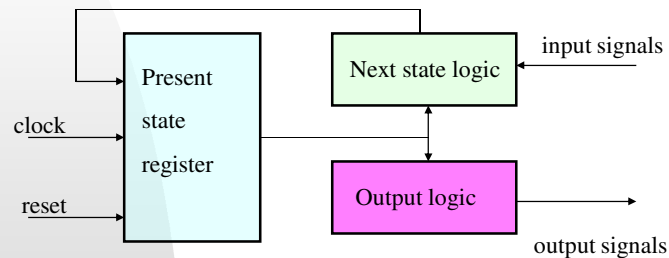


Presented by Abramov B.  
All right reserved

285

# FSM forms (cont)

In a Moore machine, the outputs depend directly on the present state through output logic.

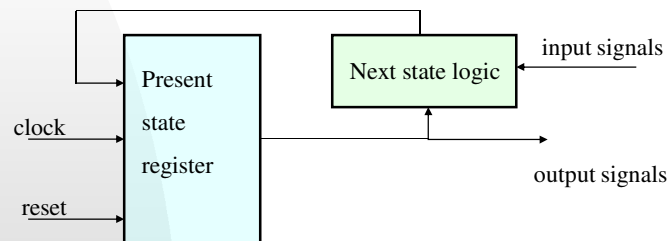


Presented by Abramov B.  
All right reserved

286

## FSM forms (cont)

In Medvedev machine (kind of Moore machine), the outputs depend directly on the present state without logic.



Presented by Abramov B.  
All right reserved

287

## FSM outputs

Moore FSM :

The output vector (Y) is a function of state vector (S) only

$Y = f(S)$  (synchronous to state, unregistered outputs)

Medvedev FSM (Full Moore) :

The output vector (Y) resembles the state vector (S) or the slice of state vector.

$Y = S$  (synchronous to state, registered output)

Good choice for latency dependent systems

Mealy FSM : The output vector (Y) is a function of state vector (S) and the input vector (X)

$Y = f(S, X)$  (asynchronous to state, unregistered output)

Full Mealy: The output vector (Y) resembles the next state vector (NS) or the slice of next state vector.

$Y = NS$



Presented by Abramov B.  
All right reserved

288



## How many processes ?

- One process – describes state register, state transaction and output logic.
  - ✓ Advantage : registered outputs
  - ✓ Disadvantage : verbose syntax, poorly debugging, 1 clock latency for outputs
- Two process – the first describes state register, the second combinatorial logic.
  - ✓ Advantage : easy to debugging, simply and readable code.
  - ✓ Disadvantage : non registered outputs, needs assignment to next state and outputs for all possible cases.
- Three processes – one for state register, one for next state logic, one for outputs
  - ✓ Advantage : easy to debugging, simply and readable code.
  - ✓ Disadvantage : non registered outputs, redundant code.
- Three processes – first for state register, second for next state logic, third for synchronous outputs.
  - ✓ Advantage : fully synchronous, readable code, easy for debugging.
  - ✓ Disadvantage : 1 clock cycle latency for output assertion

Presented by Abramov B.  
All right reserved

289

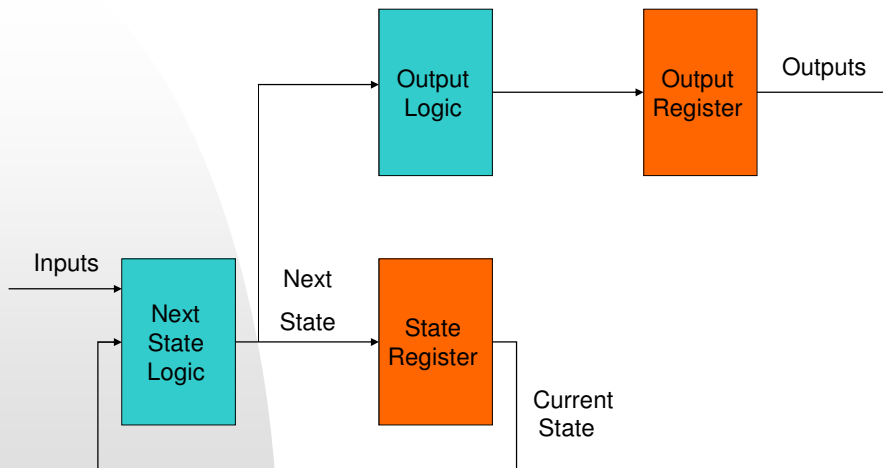
## Registered Output

- For performance and reliability, register the outputs of a state machine which conforms to synchronous design methodology and simplifies the timing with all blocks that the outputs drive and improves the overall performance.
- Registering outputs makes the FSM a better candidate for reuse and ensures that the outputs will not be optimized.
- To reduce the latency of the outputs, implement one of the following two strategies :
  1. Decode the next state signal rather than the current state (Moore output decoding), this option is available if you implement a combinatorial next state decoding block.
  2. Implement a “look ahead” Mealy scheme of decoding the current state and inputs.

Presented by Abramov B.  
All right reserved

290

# FSM with Output Registers



Presented by Abramov B.  
All right reserved

291

## Registered FSM without additional latency

```

type fsm_state is (idle, s1, s2, s3);
signal curr_st : fsm_state;
.....
fsm: process (clk, rst) is
variable next_st: fsm_state;
begin
    if (rst='0') then
        out1<='0'; out2<='0'; curr_st<=idle;
    elsif rising_edge(clk) then
        next_st:=curr_st;
        case (curr_st) is
            when idle => next_st:=....
            .....
        end case;
        case (next_st) is
            when idle=>out1<=... ; out2<=.....
            .....
        end case;
        curr_st<=next_st;
    end if;
end process fsm;
  
```

Presented by Abramov B.  
All right reserved

292

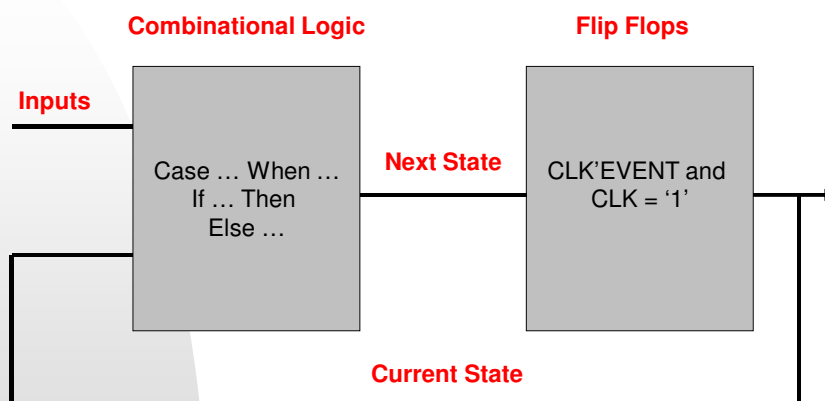
## Case vs. IF

- Case statements are the best implementation style, that's the reason we use them to define our state machines.
- If-then-else statements will generally infer priority encoded logic (cascaded logic adds delay), so we use case statement to achieve the most efficient use of our resources.
- We use case statements for next state decoding and output decoding.

Presented by Abramov B.  
All right reserved

293

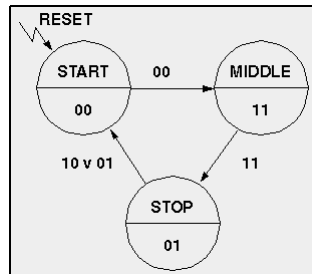
## Case vs. IF



Presented by Abramov B.  
All right reserved

294

# Medvedev FSM



```

subtype state_type is std_logic_vector(1 downto 0);
constant START : state_type:="00";
constant MIDDLE : state_type:="11";
constant STOP : state_type:="01";
  
```

Presented by Abramov B.  
All right reserved

295

# Medvedev FSM (cont)

Two processes style.

**architecture** rtl of medvedev\_fsm is

**signal** state,next\_st : state\_type;

**begin**

state\_reg: **process** (clk,rst) is

**begin**

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

state<=START;

**elsif** rising\_edge(clk) **then**

state<=next\_st;

**end if;**

**end process** state\_reg;;

May be:

```

state<= START when (rst='0') else
  next_st when rising_edge(clk);
  
```

Presented by Abramov B.  
All right reserved

296

## Medvedev FSM (cont)

```
comb_logic: process( a, b, state) begin
  next_st<=state;
  case state is
    when START =>
      if ((a or b)='0') then
        next_st<= MIDDLE;
      end if;
    when MIDDLE =>
      if ((a and b)='1') then
        next_st<=STOP;
      end if;
    when STOP =>
      if ((a xor b)='1') then
        next_st<=START ;
      end if;
    when others => next_st<=START ;
  end case;
end process comb_logic;
(y,z) <= std_logic_vector'(STATE);
end architecture rtl;
```

Presented by Abramov B.  
All right reserved

297

## FSM Performance

Factors affecting FSM performance are size, complexity, and state vector encoding. Size and complexity are determined by the design, and changes in performance cannot be affected without design changes.

Encoding style attributes allows exploration of different coding styles and their effect on performance without having to change the FSM design or VHDL code.

Presented by Abramov B.  
All right reserved

298

# Encoding vs. FSM Performance

Encoding	Area	Speed	State Vector Size
Binary (Sequential)	Better	Worse	$\text{ceiling}(\log_2(\text{number of states}))$
One-hot	Worse	Better	number of states
Gray	Better	Worse	$\text{ceiling}(\log_2(\text{number of states}))$

Presented by Abramov B.  
All right reserved

299

## FSM Performance

Many hardware compilers automatically encodes state vectors using Binary encoding when:

- the total number of states is 4 or less
- the total number of states is greater than 512
- the target device has a CPLD architecture (SOP array)
- the Safe FSM option is enabled

and automatically encodes state vectors using One-hot encoding when:

- the target device is not a CPLD (SOP array)
- the total number of states is 5 or greater
- the total number of states is 512 or less
- the Safe FSM option is disabled

Presented by Abramov B.  
All right reserved

300

# FSM Encoding

*TYPE\_ENCODING\_STYLE* attribute can be applied to the label of the process to control the encoding of the implicit state machine.

The *TYPE\_ENCODING\_STYLE* gives a hint to the compiler as to what kind of encoding style to choose.

There are five different styles to choose from:

*BINARY, GRAY, ONEHOT, RANDOM, AUTO.*

Here is an example of how to use the *TYPE\_ENCODING\_STYLE* attribute on a (imaginary) state enumerated type:

-- Declare the *TYPE\_ENCODING\_STYLE* attribute

**type** encoding\_style **is** (BINARY, ONEHOT, GRAY, RANDOM, AUTO) ;

**attribute** *TYPE\_ENCODING\_STYLE* : encoding\_style ;

-- Declare the (state-machine) enumerated type :

**type** my\_state\_type **is** (SEND, RECEIVE, IGNORE, HOLD, IDLE) ;

-- Set the *TYPE\_ENCODING\_STYLE* of the state type :

**attribute** *TYPE\_ENCODING\_STYLE* **of** my\_state\_type: **type** **is** ONEHOT ;

Presented by Abramov B.  
All right reserved

301

# FSM Encoding

To fully control the state encoding, use the *TYPE\_ENCODING* attribute. With the *TYPE\_ENCODING* attribute you can define the state table used.

Here is an example:

The *TYPE\_ENCODING* attribute takes an array of equal-length strings, where each string defines a row in the state table.

Declare the *TYPE\_ENCODING* attribute :

**type** string\_array **is** array (natural range <>, natural range <>) **of** character ;

**attribute** *TYPE\_ENCODING* : string\_array ;

-- Declare the (state-machine) enumerated type :

**type** my\_state\_type **is** (SEND, RECEIVE, IGNORE, HOLD, IDLE) ;

-- Set the type-encoding attribute :

**attribute** *TYPE\_ENCODING* **of** my\_state\_type:

**type** **is** ("0001", "0100", "0000", "1100", "0010") ;

Presented by Abramov B.  
All right reserved

302

## Enumerated type vs. constants

Enumerated type:

- Highest level of abstraction
- More flexibility
- Supports by synthesis tools
- Simply to implementation.
- Simply to reuse and changes

Constants

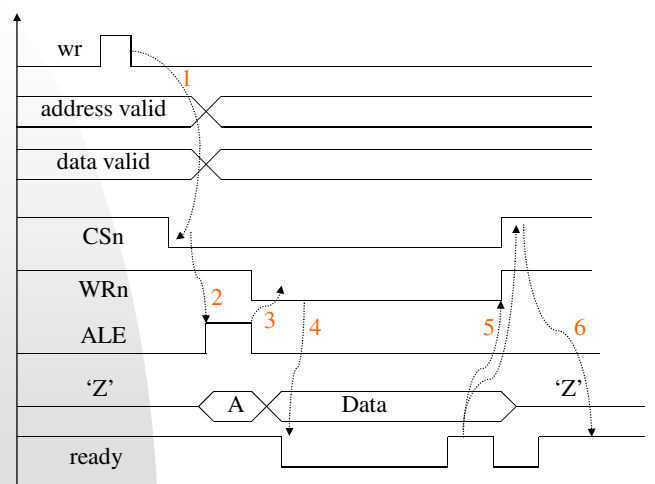
- For Medvedev or One-hot FSM
- For unique encoding (low power issues, hazard free encoding etc.)



Presented by Abramov B.  
All right reserved

303

## From wave diagram to FSM



Presented by Abramov B.  
All right reserved

304



# FSM with enumeration encoding

Canonical FSM style:

```

type fsm_type is (IDLE,START_READ,START_WRITE,ALE_READ,
    ALE_WRITE,WAIT_FOR_WREADY,WAIT_FOR_RREADY);
signal curr_state : fsm_type;
signal next_state : fsm_type;
begin
    current_state_register: process(clk,rst) is
    begin
        if (rst='0') then
            curr_state<=IDLE;
        elsif rising_edge(clk) then
            curr_state<= next_state;
        end if;
    end process current_state_register;

```



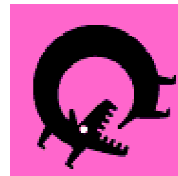
Presented by Abramov B.  
All right reserved

305

```

comb_logic: process(curr_state,wr,rd,ready) is
begin
    cs <= '1'; rdn <= '1'; wrn <= '1'; ale <= '0'; next_state<=curr_st;
    case curr_state is
        when IDLE => if (wr='1') then next_state <=START_WRITE;
            elsif (rd='1') then next_state <=START_READ;
        end if;
        when START_READ => cs <='0'; rdn <='0'; next_state<=ALE_READ;
        when START_WRITE => cs <='0'; wrn <='0'; next_state<=ALE_WRITE;
        when ALE_READ => cs <='0'; rdn <='0'; ale <='1'; next_state<=WAIT_FOR_RREADY;
        when ALE_WRITE => cs <='0'; wrn<='0'; ale<='1'; next_state<=WAIT_FOR_WREADY;
        when WAIT_FOR_WREADY => cs<='0'; wrn<='0';
            if (ready='1') then next_state<=IDLE;
        end if;
        when others => cs<='0'; rdn<='0';
            if (ready='1') then next_state<=IDLE;
        end if;
    end case;
end process comb_logic;

```



Presented by Abramov B.  
All right reserved

306

# One Hot Encoding

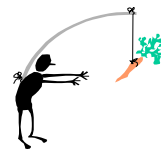
- One hot encoding is a technique that uses n flip flops to represent a state machine with n states.
- Each state has its own flip flop, and only one flip flop is “hot” (holds a 1) at any given time.
- Decoding the current state is as simple as finding the flip flop containing a 1.
- During a state transition exactly two flip-flops changes its values, which often translated into low power consumption.

Presented by Abramov B.  
All right reserved

307

# One-Hot FSM

```
constant IDLE : natural:=0;
constant START_READ : natural:=1;
constant START_WRITE : natural:=2;
constant ALE_READ : natural:=3;
constant ALE_WRITE : natural:=4;
constant WAIT_FOR_WREADY : natural:=5;
constant WAIT_FOR_RREADY : natural:=6;
signal cstate : std_logic_vector(WAIT_FOR_RREADY downto IDLE);
signal nstate : std_logic_vector(WAIT_FOR_RREADY downto IDLE);
begin
    current_state_register: process (clk,rst) is
    begin
        if (rst='0') then
            cstate<= (IDLE=>'1',others=>'0');
        elsif rising_edge(clk) then
            cstate<= nstate;
        end if;
    end process current_state_register;
```



Presented by Abramov B.  
All right reserved

308

# One-Hot FSM

```

comb_logic: process (cstate,wr,rd,ready) is
begin
  cs <= '1'; rdn <= '1'; wrn <= '1'; ale <= '0';
  nstate <= (others=>'0');
  if (cstate(IDLE)='1') then
    if (wr='1') then
      nstate(START_WRITE)<='1';
    elsif (rd='1') then
      nstate(START_READ)<='1';
    else nstate <= curr_state;
    end if;
  end if;
  if (cstate(START_READ)='1') then
    cs<='0';rdn<='0';
    nstate(ALE_READ)<='1';
  end if;
  if (cstate(START_WRITE)='1') then
    cs<='0';wrn<='0';
    nstate(ALE_WRITE)<='1';
  end if;
  .....

```

Presented by Abramov B.  
All right reserved

309

## Modified One Hot FSM Coding

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1

One Hot  
Coding

6	5	4	3	2	1	0
0	0	0	0	0	0	0
1	0	0	0	0	0	0
0	1	0	0	0	0	0
0	0	1	0	0	0	0
0	0	0	1	0	0	0
0	0	0	0	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	1

Modified One Hot  
Coding

Presented by Abramov B.  
All right reserved

310

# One-Hot FSM

```

constant START_READ      : natural:=0;
constant START_WRITE     : natural:=1;
constant ALE_READ        : natural:=2;
constant ALE_WRITE       : natural:=3;
constant WAIT_FOR_WREADY : natural:=4;
constant WAIT_FOR_RREADY : natural:=5;
signal curr_state : std_logic_vector(WAIT_FOR_RREADY downto START_READ);
signal next_state : std_logic_vector(WAIT_FOR_RREADY downto START_READ);
begin
    current_state_register: process(clk,rst) is
    begin
        if (rst='0') then
            curr_state<= (others=>'0');
        elsif rising_edge(clk) then
            curr_state<= next_state;
        end if;
    end process current_state_register;

```

Presented by Abramov B.  
All right reserved

311

# One-Hot FSM

```

comb_logic: process (curr_state,wr,rd,ready) is
begin
    cs <= '1';rdn <= '1';
    wrn <= '1';ale <= '0';
    next_state <= (others=>'0');
    if (curr_state=0) then
        if (wr='1') then
            next_state(START_WRITE)<='1';
        elsif (rd='1') then
            next_state(START_READ)<='1';
        end if;
    end if;
    if (curr_state(START_READ)='1') then
        cs<='0';rdn<='0';
        next_state(ALE_READ)<='1';
    end if;
    .....
end process comb_logic;

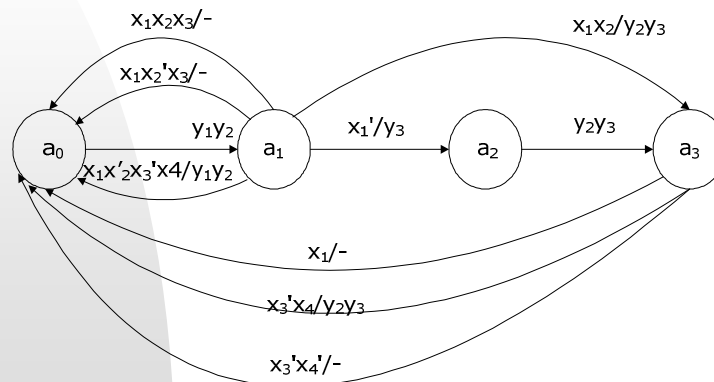
```

Presented by Abramov B.  
All right reserved

312

## FSM decomposition

- ✓ In this section, we will learn a simple technique of FSM decomposition.
- ✓ As an example, we use Mealy FSM with 4 states



Presented by Abramov B.  
All right reserved

313

## FSM decomposition

$a_m$	$C(a_m)$	$a_s$	$C(a_s)$	$X(a_m, a_s)$	$Y(a_m, a_s)$	$D(a_m, a_s)$
$a_0$	0001	$a_1$	0010	-	$y_1, y_2$	$D_1$
$a_1$	0010	$a_3$	0100	$x_1x_2$	$y_1, y_2$	$D_2$
		$a_2$	1000	$x_1$	$y_3$	$D_3$
		$a_0$	0001	$x_1x_2', x_3', x_4$	$y_1, y_2$	$D_0$
		$a_0$	0001	$x_1x_2', x_3$	$y_1, y_2$	$D_0$
		$a_0$	0001	$x_1x_2', x_3', x_4'$	-	$D_0$
$a_2$	0100	$a_3$	1000	-	$y_2, y_3$	$D_2$
$a_3$	1000	$a_0$	0001	$x_3', x_4'$	-	$D_0$
		$a_0$	0001	$x_3', x_4$	$y_1, y_2$	$D_0$
		$a_0$	0001	$x_3$	$y_1, y_2$	$D_0$

Presented by Abramov B.  
All right reserved

314

## FSM decomposition

Partition  $\pi$  is the set of states:

$$\pi = \{A_1, A_2, \dots\};$$

$$A_1 = \{a_0, a_1\}; A_2 = \{a_2, a_3\};$$

The number of component FSMs in the FSM network is equal to the number of blocks in partition  $\pi$ . Thus, in our example, we have three component FSMs  $S_1, S_2$ .

Let  $B_m$  is the set of states in the component FSM  $S_m$ .

$B_m$  contains the corresponding block of the partition  $\pi$  plus one additional state  $b_m$ . So, in our example:

$$S_1 \text{ has the set of states } B_1 = \{a_0, a_1, b_1\};$$

$$S_2 \text{ has the set of states } B_2 = \{a_2, a_3, b_2\};$$

Presented by Abramov B.  
All right reserved

315

## FSM decomposition

In Mealy FSM  $S_4$ , there is a transition between  $a_i$  and  $a_j$  (left) and both these states are in the same component FSM  $S_m$ .

In such a case, we have the same transition in this component FSM  $S_m$ .



Two states  $a_i$  and  $a_j$  are in different component FSMs .

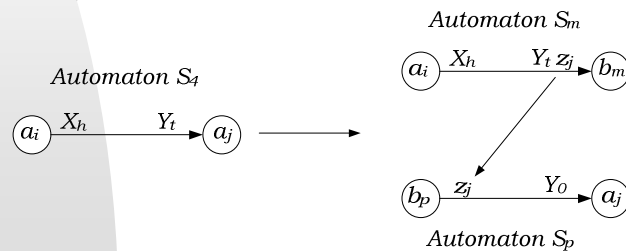
Let  $a_i$  is in the component FSM  $S_m$  ( $a_i \in B_m$ ) and  $a_j$  is in the component FSM  $S_p$  ( $a_j \in B_p$ ). In such a case, one transition of FSM  $S_4$  should be presented as two transitions – one in the component FSM  $S_m$  and one in the component FSM  $S_p$ .

Presented by Abramov B.  
All right reserved

316

## FSM decomposition

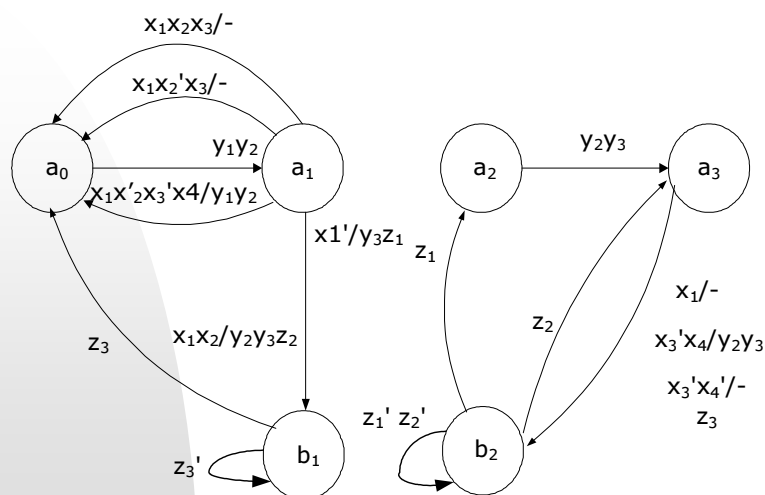
FSM  $S_m$  transits from  $a_i$  into its additional state  $b_m$  with the same input  $X_h$ . At its output, we have the same output variables from set  $Y_t$  plus one additional output variable  $z_j$ , where index  $j$  is the index of state  $a_j$  in the component FSM  $S_p$ . FSM  $S_p$  is in its additional state  $b_p$ . It transits from this state into state  $a_j$  with input signal  $z_j$ , that is an additional output variable in the component FSM  $S_m$ . The output at this transition is  $Y_0$  – the signal with all output variables equal to zero.



Presented by Abramov B.  
All right reserved

317

## FSM decomposition



Presented by Abramov B.  
All right reserved

318

## FSM decomposition

$$A_1 = \{a_0, a_1, b_1\};$$

$a_m$	$C(a_m)$	$a_s$	$C(a_s)$	$X(a_m, a_s)$	$Y(a_m, a_s)$	$D(a_m, a_s)$
$a_0$	001	$a_1$	010	-	$y_1, y_2$	$D_2$
$a_1$	010	$a_0$	001	$x_1 x_2 x_3$	-	$D_1$
		$a_0$	001	$x_1 x_2' x_3' x_4$	$y_1, y_2$	$D_1$
		$a_0$	001	$x_1 x_2' x_3$	-	$D_1$
		$b_1$	100	$x_1'$	$y_2, y_3, z_1$	$D_3$
$b_1$	100	$a_0$	001	$z_3$	-	$D_1$
		$b_1$	100	$z_3'$	-	$D_3$

Presented by Abramov B.  
All right reserved

319

## FSM decomposition

$$A_2 = \{a_2, a_3, b_2\};$$

$a_m$	$C(a_m)$	$a_s$	$C(a_s)$	$X(a_m, a_s)$	$Y(a_m, a_s)$	$D(a_m, a_s)$
$a_2$	001	$a_3$	010	-	$y_2, y_3$	$D_2$
$a_3$	010	$b_2$	100	$x_1$	-	$D_3$
		$b_2$	100	$x_3' x_4$	$y_2, y_3$	$D_3$
		$b_2$	100	$x_3' x_4'$	$z_3$	$D_3$
$b_2$	100	$a_2$	001	$z_1$	-	$D_1$
		$b_2$	100	$z_1' z_2'$	-	$D_3$

Presented by Abramov B.  
All right reserved

320



# FSM reuse

**FSM implementation as procedure.**

```

type ctrl_fsm is (write_word, check_num_of_wwords, write_to_ram,
                    clear_buf, check_block_select, wait_on_rw,
                    read_request, wait_on_mem_valid, send_word);
subtype wrctrl_fsm is ctrl_fsm range write_word to wait_on_rw;
subtype rdctrl_fsm is ctrl_fsm range check_block_select to send_word;
signal wrctrl_curr_st : wrctrl_fsm;
signal wrctrl_next_st : wrctrl_fsm;
signal rdctrl_curr_st : rdctrl_fsm;
signal rdctrl_next_st : rdctrl_fsm;

```

Presented by Abramov B.  
All right reserved

321

# FSM reuse

```

procedure ext_mem_write (
    signal curr_st      : in wrctrl_fsm;
    signal write_req    : in std_ulogic;
    signal is_addr      : in std_ulogic;
    signal buff_full    : in std_ulogic;
    signal buff_we      : out std_ulogic;
    signal ext_mem_we   : out std_ulogic;
    signal buff_cnt_clr : out std_ulogic;
    signal next_st      : out wrctrl_fsm ) is

    begin
        case curr_st is
            when .....
            when .....
        end case;
    end procedure ext_mem_write ;

```

Presented by Abramov B.  
All right reserved

322

## FSM reuse

```

procedure ext_mem_read (
    signal curr_st      : in   rdctrl_fsm;
    signal read_req    : in   std_ulogic;
    signal is_addr     : in   std_ulogic;
    signal buff_empty  : in   std_ulogic;
    signal read_valid  : in   std_ulogic;
    signal buff_re     : out  std_ulogic;
    signal ext_mem_re  : out  std_ulogic;
    signal next_st     : out  rdctrl_fsm ) is

begin
    case curr_st is
        when .....
    end case;
end procedure ext_mem_read ;

```

Presented by Abramov B.  
All right reserved

323

## FSM reuse

```

-- write control FSM from CPU to external RAM
wctrl_fsm_register : process(sys_clk, sys_rst) is
begin
    if (sys_rst = RESET_INIT) then
        wrctrl_curr_st <= wait_on_rw;
    elsif rising_edge(sys_clk) then
        wrctrl_curr_st <= ing_search_wrctrl_next_st;
    end if;
end process wctrl_fsm_register;
-- next state logic and output logic by ext_mem_write procedure
ext_mem_write ( wrctrl_curr_st, wdle, is_ingbuf_addr, is_inbuf_full, inbuf_en,
    cpu_table_wr_int, inbuf_count_clr, wrctrl_next_st );

```

Presented by Abramov B.  
All right reserved

324

# FSM rules

The possible states of the state machine are listed in an enumerated type.  
A signal of this type (present\_state) defines in which state the state machine appears.  
In a case statement of one process, a second signal (next\_state) is updated depending on present\_state and the inputs. In the same case statement, the outputs are also updated.  
Another process updates present\_state with next\_state on a clock edge, and takes care of the state machine reset.

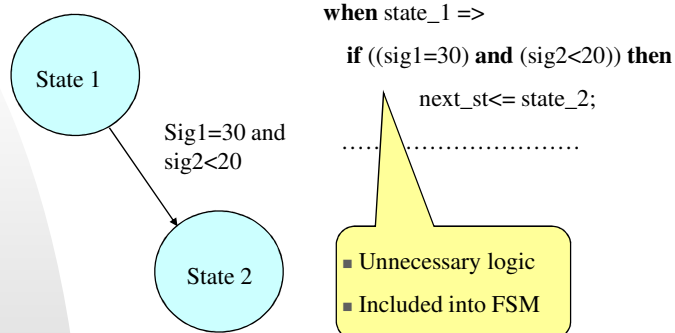
Here are a few general rules to follow:

- ✓ Only one state machine per module
- ✓ Keep extraneous logic at a minimum (try not to put other code in the same FSM)
- ✓ Instantiate state flip-flops separately from logic

Presented by Abramov B.  
All right reserved

325

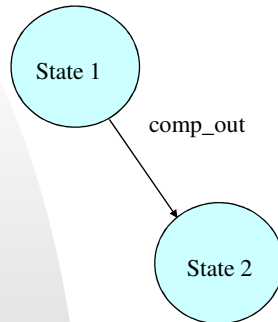
# FSM rules



Presented by Abramov B.  
All right reserved

326

# FSM rules



```

when state_1 =>
  if (comp_out) then
    next_st<= state_2;
    .....
  
```

- comp\_out calculated outside of FSM.
- Only input control signals !!!

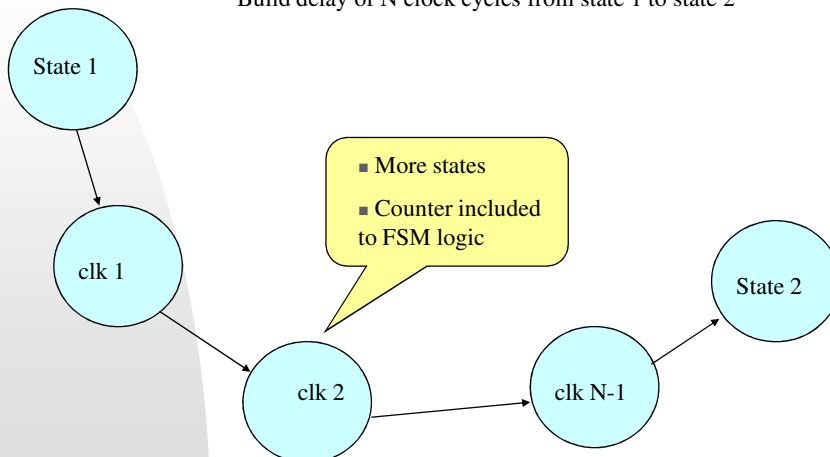
comp\_out <= (sig1=30) and (sig2<20);

Presented by Abramov B.  
All right reserved

327

# FSM rules

Build delay of N clock cycles from state 1 to state 2

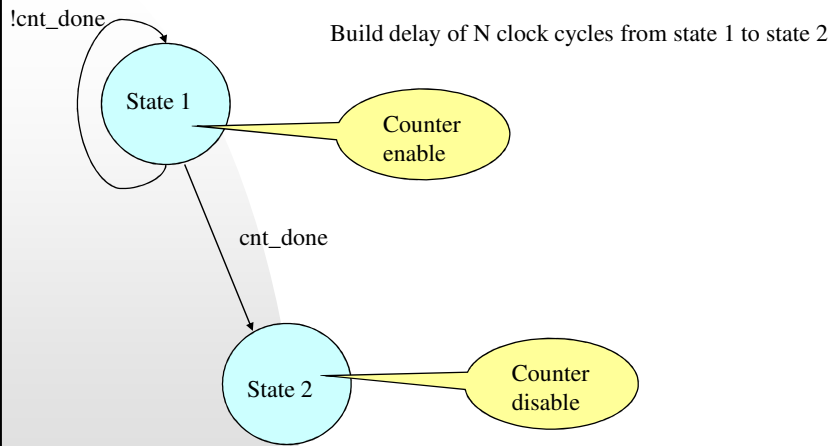


- More states
- Counter included to FSM logic

Presented by Abramov B.  
All right reserved

328

# FSM rules



Presented by Abramov B.  
All right reserved

329

# FSM rules

```

cnt_done<= ( cnt = (N-1));
delay_counter : process (clk ,rst) is
begin
  if (rst='0') then
    cnt<=(others=>'0');
  elsif rising_edge (clk) then
    if (cnt_en) then
      cnt <= cnt + 1;
    else cnt <= (others =>'0');
    end if;
  end if;
end process delay_counter;

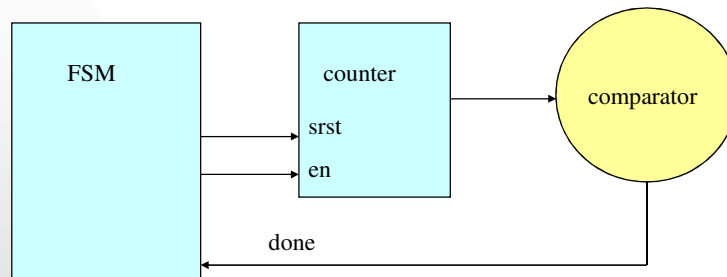
-- FSM logic
when state_1 =>
  cnt_en<=true;
  if (cnt_done) then
    next_st<= state_2;
  else next_st<= state_1;
  end if;
  .....
when state_2 =>
  cnt_en<=false;
  
```

Presented by Abramov B.  
All right reserved

330

# FSM rules

Try to minimize the number of control outputs signal of FSM:



Presented by Abramov B.  
All right reserved

331

# FSM rules

-- FSM logic

```

when state_1 =>
  srst <= false;
  cnt_en <= true;
  if (cnt_done) then
    next_st <= state_2;
  else next_st <= state_1;
  end if;
.....
when state_2 =>
  cnt_en <= false;
  srst <= true;
  
```

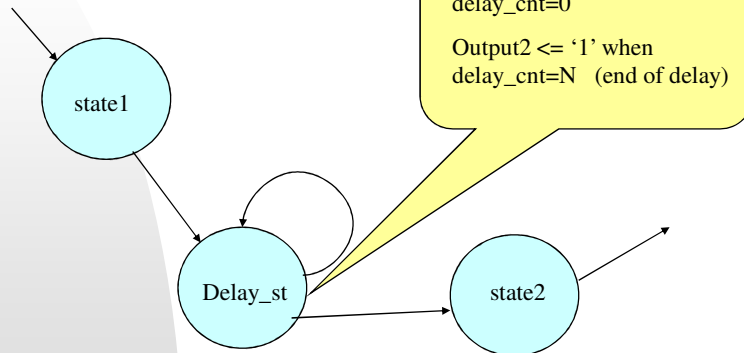
- Synchronous reset is not always needs.
- Try to write according to previous example of counter

Presented by Abramov B.  
All right reserved

332

# FSM rules

State signals utilization.



Presented by Abramov B.  
All right reserved

333

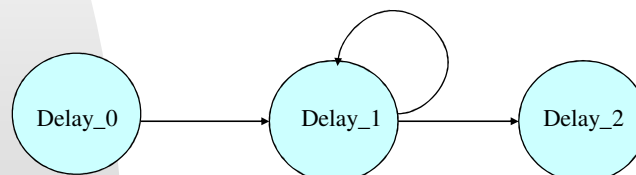
# FSM rules

First possible solutions:

Three states, full synchronous.

Split the delay state to three states

- delay\_0 with signal 1='1'
- delay\_2 with signal 2='1'
- delay\_1 that implements delay between delay\_0 and delay\_2



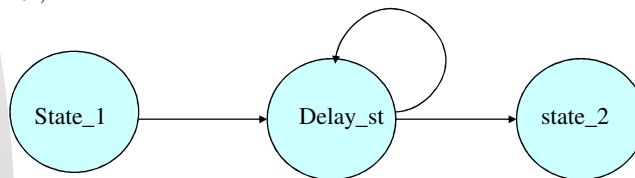
Presented by Abramov B.  
All right reserved

334

# FSM rules

Second possible solutions: one state , asynchronous inputs.

```
when delay_st =>
  if (cnt_eq_zero) then
    output1<='1';
  else output_1<='0';
  end if;
  if (cnt_done) then
    output2<='1';
  else output_2<='0';
  end if;
  .....
```



Presented by Abramov B.  
All right reserved

335

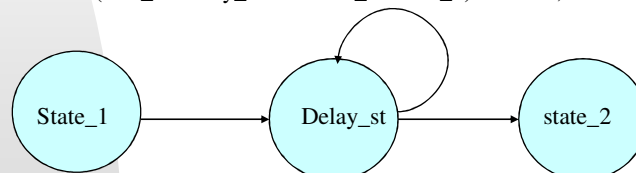
# FSM rules

Third solutions:

One state , outside from FSM logic.

Output\_1<= '1' when (curr\_st = state\_1 and next\_st=delay\_st ) else '0';

Output\_2<= '1' when (curr\_st=delay\_st and next\_st=state\_2) else '0';



Presented by Abramov B.  
All right reserved

336



# Safe FSM



## Safe/Unsafe State Machines

### Safe State Machines:

If the number of states  $N = 2^n$ , where  $n$  is state register width, and you use a binary or gray-code encoding, the state machine is “safe”, because all of the possible states are reachable.

### Unsafe State Machines:

If the number of states  $N < 2^n$ , or if you do not use binary or gray-code encoding, e.g. one-hot, the state machine is “unsafe”.

A “Safe” State Machine has been defined as one that:

- ✓ Has a set of defined states
- ✓ Can jump to a defined state if an illegal state has been reached .

Presented by Abramov B.  
All right reserved

337

# Safe FSM

**Designers Beware!!!** If using a CASE statement to implement the FSM, the others clause in your VHDL is ignored by the synthesis tools

This logic will not get synthesized unless you explicitly attribute your FSM as “Safe”

Synthesis tools offer a “Safe” option:

**type** states **is** (IDLE, GET\_DATA, SEND\_DATA, BAD\_DATA);

**signal** current\_state, next\_state : **states**;

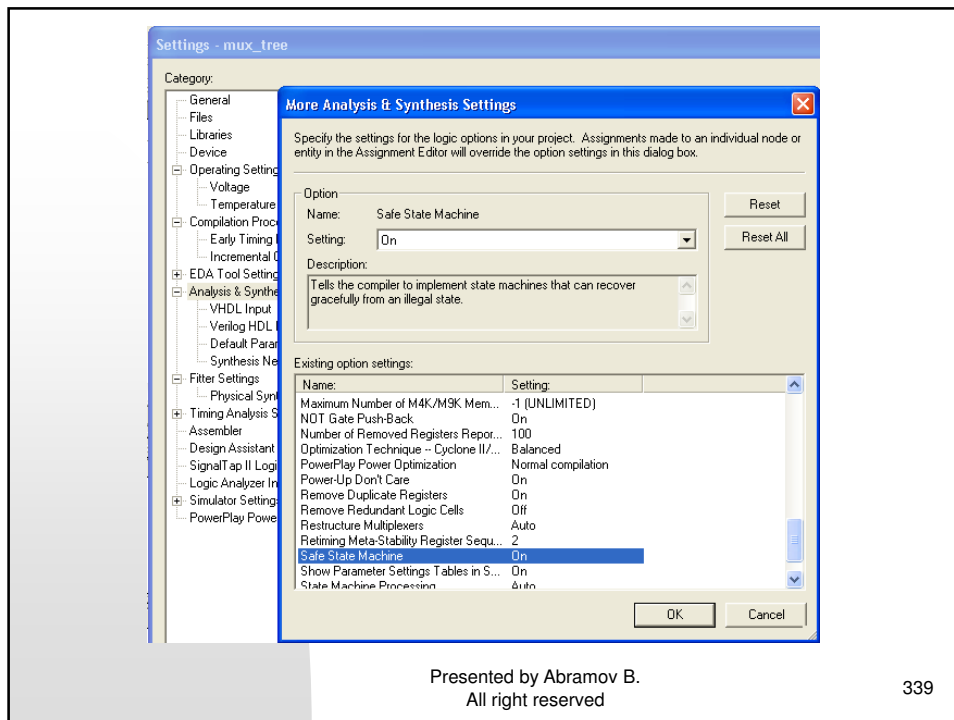
**attribute** SAFE\_FSM: **Boolean**;

**attribute** SAFE\_FSM of **states**: **type is true**;

Some versions of synthesis tools will not synthesize a “Safe” One-Hot FSM !!!

Presented by Abramov B.  
All right reserved

338



## Safe FSM

Setting the FSM\_COMPLETE attribute to true a synthesis tool to use transition specified by the VHDL default assignment to specify transition for unused states in the implementation.

**attribute** FSM\_COMPLETE : **boolean**;

**attribute** FSM\_COMPLETE of signal\_name : **signal is true**;

Presented by Abramov B.  
All right reserved

340

# Safe FSM

- If a Binary encoded FSM flips into an illegal (unmapped) state, the safe option will return the FSM into a known state that is defined by the others or default clause
- If a Binary encoded FSM flips into a good state, this error will go undetected.
  - ◆ If the FSM is controlling a critical output, this phenomena can be very detrimental!
  - ◆ How safe is this?

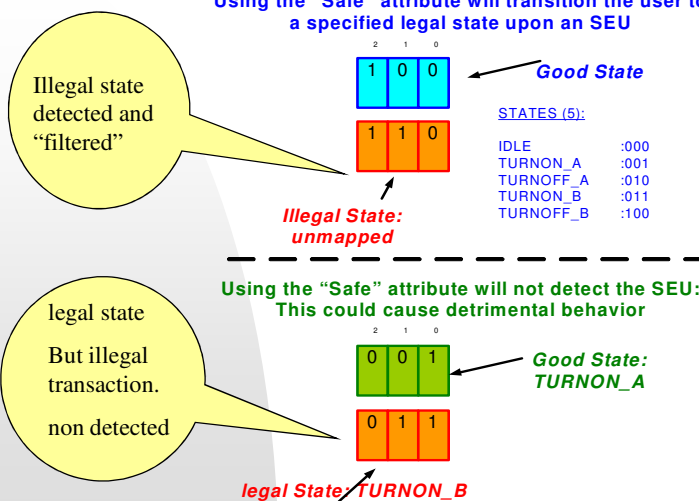
Presented by Abramov B.  
All right reserved

341

# Safe FSM

## State(1) Flips upon SEU:

Using the "Safe" attribute will transition the user to a specified legal state upon an SEU



Presented by Abramov B.  
All right reserved

342

# Safe FSM

- There used to be a consensus suggesting that Binary is “safer” than One-Hot
  - ◆ Based on the idea that One-Hot requires more DFFs to implement a FSM thus has a higher probability of incurring an error
- This theory has been changed!
  - ◆ The community now understands that although One-Hot requires more registers, it has the built-in detection that is necessary for safe design
  - ◆ Binary encoding can lead to a very “un-safe” design

Presented by Abramov B.  
All right reserved

343

# Safe FSM

- **One-Hot**
  - ◆ Synthesis “Safe” directive will generally not work – FSM becomes too large
  - ◆ Each state in a One-Hot encoding has a hamming distance of two.
  - ◆ Error Detection:
    - \* During normal operation, only one bit is turned on, thus It inherently has SEU error detection
    - \* SEU Error detection can be accomplished using combinational logic
    - \* SEUs occurring near a clock edge will always be detected
- **Binary**
  - ◆ Synthesis “Safe” directive generally will not meet requirements – can flip into a good state undetected
  - ◆ Binary Encoding has a hamming distance of One
  - ◆ Error Detection:
    - \* It needs extra DFFs in order to implement error detection
    - \* Utilizing an extra DFF can cause a SEU to go undetected due to glitches and routing differences between combinational logic and DFFs.

Presented by Abramov B.  
All right reserved

344

# Safe FSM

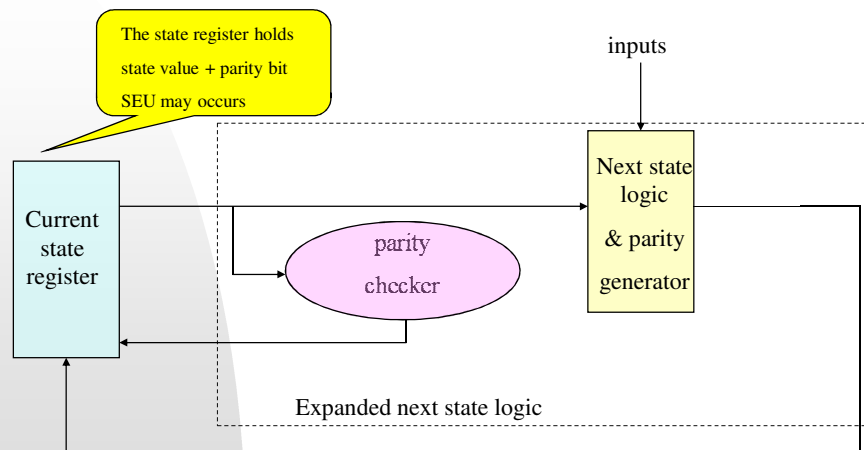
- The state machine needs to be tolerant of single event upsets
- State machine should not hang
- State machine should always be in a defined state
- No asynchronous inputs to state machine
- Default state must be specified

Presented by Abramov B.  
All right reserved

345

## Safe FSM architecture

This architecture is based on parity usage in accordance to diagram presented below.

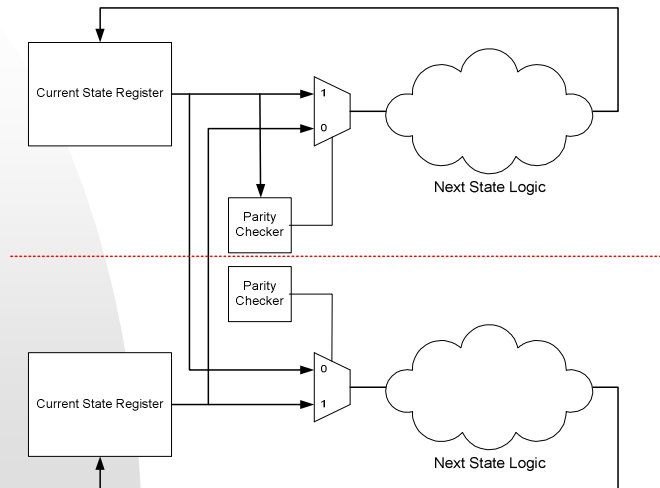


Presented by Abramov B.  
All right reserved

346

## Safe FSM architecture (cont)

This architecture is based on duplication and parity checking

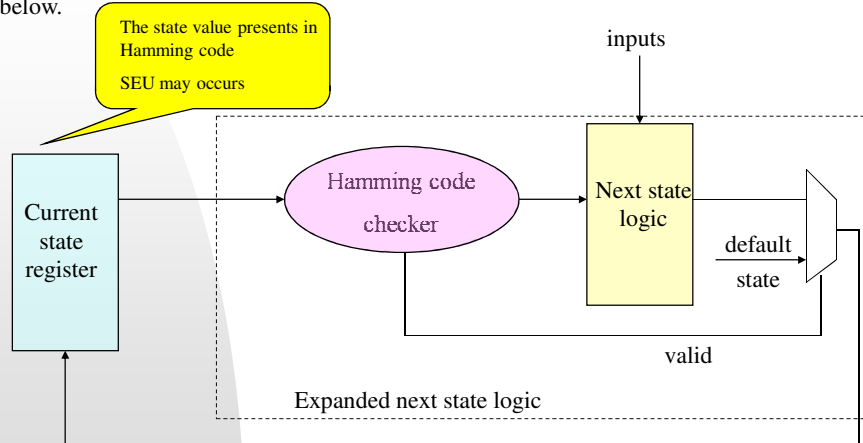


Presented by Abramov B.  
All right reserved

347

## Safe FSM architecture (cont)

This technique is based on Hamming code usage in accordance to diagram presented below.



Presented by Abramov B.  
All right reserved

348

# Safe FSM with FEC

The Hamming code encoder example:

```
procedure hamming_encoder
(signal din : inout std_logic_vector(7 downto 0);
signal dout: out std_logic_vector(11 downto 0)) is
  variable p : std_logic_vector(3 downto 0);
begin
  p(0):=din(6) xor din (4) xor din(3) xor din(1) xor din(0);
  p(1):=din(6) xor din (5) xor din(3) xor din(2) xor din(0);
  p(2):=din(7) xor din (3) xor din(2) xor din(1);
  p(3):=din(7) xor din (6) xor din(5) xor din(4);
  dout<=(din(7 downto 4),p(3),din(3 downto 1),p(2),din(0),p(1),p(0));
end procedure hamming_encoder;
```

Presented by Abramov B.  
All right reserved

349

# Safe FSM with FEC

The Hamming code decoder example:

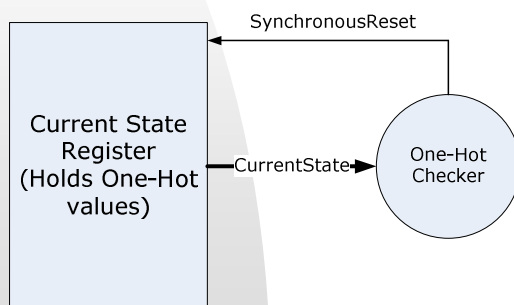
```
procedure hamming_decoder (signal din : in std_logic_vector(11 downto 0);
signal dout : out std_logic_vector(7 downto 0)) is
  variable checked : std_logic_vector(3 downto 0);
  variable dec : std_logic_vector(15 downto 0):=(others=>'0');
  variable fixed_buff : std_logic_vector(din'range);
begin
  checked(0):=din(10) xor din (8) xor din(6) xor din(4) xor din(2) xor din(0);
  checked(1):=din(10) xor din (9) xor din(6) xor din(5) xor din(2) xor din(1);
  checked(2):=din(11) xor din (6) xor din(5) xor din(4) xor din(3);
  checked(3):=din(11) xor din (10) xor din(9) xor din(8) xor din(7);
  dec(conv_integer(checked)):=1';
  fixed_buff:=din xor dec(12 downto 1);
  dout<=(fixed_buff(11 downto 8),fixed_buff(6 downto 4),fixed_buff(2));
end procedure hamming_decoder;
```

Presented by Abramov B.  
All right reserved

350

## Safe One-hot FSM

- You can include additional logic that detects more than one flip flop being asserted at a time.
- A collision signal can be generated to detect multiple flip flops asserted at the same time.
- Use the collision signal as synchronous reset, that returns the machine back to known state



You can implement the One-Hot checker as tree of one-hot dual-rail output analyzers. The one-hot analyser for 4 arguments implements the following functions:

$c1 \leq s1 \text{ or } s2 \text{ or } (s3 \text{ and } s4);$   
 $c2 \leq s3 \text{ or } s4 \text{ or } (s1 \text{ and } s2);$   
 $srst \leq c1 \text{ xor } c2;$

Where  $s$  - 4 bit state register,  
 $srst$  – synchronous reset.

Presented by Abramov B.  
 All right reserved

351

## Safe One-hot FSM

```

first(0) <= cstate(0) or cstate(1) or (cstate(2) and cstate(3));
first(1) <= cstate(2) or cstate(3) or (cstate(0) and cstate(1));
second(0) <= cstate(4) xor cstate(5); second(1) <= cstate(6);
third(0) <= first(0) or first(1) or (second(0) and second(1));
third(1) <= second(0) or second(1) or (first(0) and first(1));
srst <= third(0) xor third(1);
current_state_register: process (clk,rst) is
begin
  if (rst='0') then
    cstate <= (IDLE=>'1',others=>'0');
  elsif rising_edge(clk) then
    if (srst='1') then
      cstate <= (IDLE=>'1',others=>'0');
    else cstate <= nstate;
    end if;
  end if;
end process current_state_register;
  
```

For this we have 7 bit state register. The check logic consists from three one-hot dual-rail output analyzers

Presented by Abramov B.  
 All right reserved

352



# Safe One-hot FSM

## One-Hot code checking by recursive function

```
function is_one_hot (d : std_logic_vector) return std_logic_vector is
  variable d_rail : std_logic_vector(1 downto 0);
  variable t      : std_logic_vector(3 downto 0);
  variable m_bnd  : integer;
begin
  case d'length is
    when 2 => d_rail := d;
    when 3 => d_rail(d_rail'high) := (d(d'high) or d(d'high-1));
             d_rail(d_rail'low)  := (d(d'high) and d(d'high-1) or d(d'low));
    when 4 => d_rail(d_rail'high) := (d(d'high) or d(d'high-1) or d(d'low+1) and d(d'low));
             d_rail(d_rail'low)  := (d(d'high) and d(d'high-1) or (d(d'low+1) or d(d'low)));
    when others =>
      m_bnd := (d'length + 1)/2 + d'right;
      t(t'high downto t'length/2) := is_one_hot(d(d'high downto m_bnd));
      t(((t'length/2)-1) downto t'low) := is_one_hot(d(m_bnd-1 downto d'low));
      d_rail := is_one_hot(t);
  end case;
  return d_rail;
end function is one hot;
```

Presented by Abramov B.  
All right reserved

353

# Advanced FSM

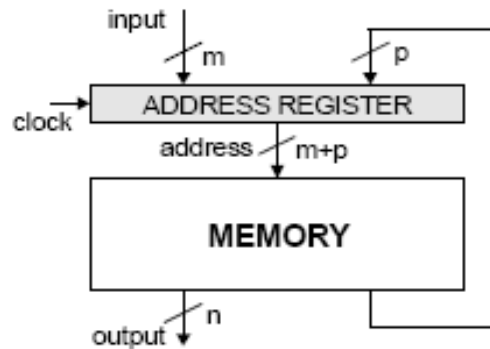
The primary architectures of FSMs usually have simple and regular structures at the top level. They are considered to be a composite of a combinational scheme and memory. The former calculates new states in state transitions and forms outputs, and the latter is used to store states. The top-level architecture can generally be reused for different applications. However, the combinational scheme is typically very irregular, and its implementation depends on the particular unique behavioral specification for a given FSM. Since such specifications vary from one implementation to another, we cannot construct a reusable FSM circuit.

Another approach to the design of FSMs is based on the use of RAM blocks. It eliminates many of the restrictions that apply to traditional FSM architectures and current methods for FSM synthesis. Since very fast EMB (Embedded Memory Blocks) are available within widely used FPGAs, we can apply the proposed technique directly to the design of FPGA-based circuits. Such an implementation is very efficient.

Presented by Abramov B.  
All right reserved

354

## Advanced RAM Based FSM



Presented by Abramov B.  
All right reserved

355

## Right Code Write

- VHDL file name is <entity\_name>.vhd  
A unit of the project .  
Includes components and/or FSM and/or  
logic processes
- Each file declares on only one entity.  
All architecture and configuration of the entity  
must be in the same file with the entity  
declaration.

Presented by Abramov B.  
All right reserved

356

## Right Code Write (cont)

- Line length is max up to 80 characters.
- No mix lower-upper case letters
- All the following names should be upper case letters:

Library name

Generic name

port name

- Others names should be lower case letters

Presented by Abramov B.  
All right reserved

357

## Right Code Write (cont)

- Architecture name : arc\_<entity\_name>
- Type name : <type\_name>\_type
- State name : st\_<state\_name>
- Constant name : c\_<constant\_name>
- Instantiation : u\_<instance\_name>
- Process : use label
- Variable : v\_<variable\_name>

Presented by Abramov B.  
All right reserved

358

## Right Code Write (cont)

- Active Low signal name: <name>\_n
- Clock signal name:< clock\_name>\_clk
- Enable signal name: <signal\_name>\_en
- Load signal name: <signal\_name>\_ld
- Valid signal name: <signal\_name>\_valid
- Write signal name: <signal\_name>\_wr
- Read signal name: <signal\_name>\_rd
- Transmit/receive : < <signal\_name>\_tx/rx

Presented by Abramov B.  
All right reserved

359

## Right Code Write (cont)

- Ranges of vectors/integers : value downto 0
- All of the outputs pins of the chip will be invoked from a FF stage and not from asynchronous process
- All of the inputs pins of the chip will be first sampled by a FF stage before being used or inspected
- Use a synchronizer when sampling an external control input pin

Presented by Abramov B.  
All right reserved

360

## Right Code Write (cont)

- Divide the chip into smaller units.  
each unit must have a specific role in the chip
- Each unit itself should be divided into smaller blocks with a specific functionality to each block
- Try to minimize the number of internal connections between internal blocks
- When 2 (or more) blocks have many common signals as their inputs or outputs ,then your should consider joining them together under one block

Presented by Abramov B.  
All right reserved

361

## Right Code Write (cont)

- Don't mix logic and instantiations in upper block level
- Instantiation by name is recommended
- If reset is used in a synchronous process it must appear in its sensitivity list
- Each FF must be assigned a default value in reset
- All control signals must be assigned a default value in reset

Presented by Abramov B.  
All right reserved

362

## Right Code Write (cont)

- Try to use as less number of different clocks as possible
- Try to avoid from dividing a clock inside the chip
- Don't put elsif/else branch after an elsif branch containing clock event.
- Don't ever use a wait statement in a code for synthesis
- All signals will be sampled on the rising edge of a clock.
- Consider sampling on a falling edge of a clock only on special cases.

Presented by Abramov B.  
All right reserved

363

## Right Code Write (cont)

- In a synchronous process the sensitivity list includes only reset and clock
- In an asynchronous process the sensitivity list includes all signals that are used or being inspected in the process
- No latches!!!
- Try to avoid a long "elsif" tree when possible
- Prefer to use a Case instead of a long "elsif" tree when there is no need for a priority to any of the case option

Presented by Abramov B.  
All right reserved

364

## Right Code Write (cont)

- Cover all the values of the case range with the “when others “ statement
- Prefer to use a Moore machine instead of a Mealy machine
- The signals created by the FSM must be control signals and etc.
- A default value must be assigned in each state to all of the outputs of the FSM

Presented by Abramov B.  
All right reserved

365