Module: Electronics & Telecomunication, 5rd year

Programmable Logical Devices

Advanced VHDL

Agenda

Structural description

map, generate

Lexical elements

objects: signals, variables, constants, generics

Sequential statements

process, wait, if, case, loop, next, exit, assert, function, procedure

Assertion based verification

introduction

Concurrent statements

assignments: unconditional, conditional and selected, subprograms, block

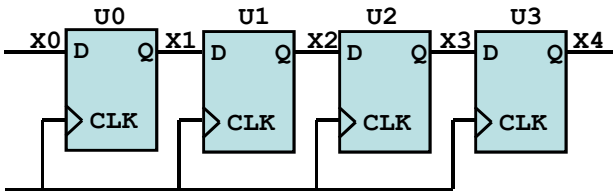
Structural description

generate statement

Syntax:

The most common usage of the generate statement is to create multiple copies of components, processes, or blocks.

label:
{[for *instruction* if *condition*]} generate
{*concurrent_statements*}
end generate;

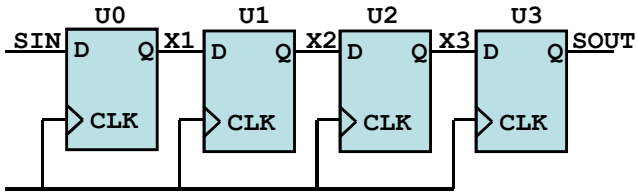


Example:

```
gen1: for i in 0 to 3 generate
      U: DFF port map (X(i), clk, X(i+1));
end generate;
```


Structural description

generate statement




```
for i in 0 to 3 generate
  if (i=0) generate
    UA: DFF port map (SIN, CLK, X(i+1)); end generate;
  if ((i>0) and (i<3)) generate
    UB: DFF port map (X(i), CLK, X(i+1)); end generate;
  if (i=3) generate
    UC: DFF port map (X(i), CLK, SOUT); end generate;
end generate;
```

Structural description

generate statement

```
gen_code_label:
 for index in 0 to 7 generate
    begin
      BUFR_inst : BUFR
    generic map (
       BUFR_DIVIDE => "BYPASS")
    port map (
     O => clk_o(index),
     CE => ce,
     CLR => clear,
     I => clk_i(index) );
  end generate;
```

The example shows a generate for loop that generates 8 regional clock buffers (BUFR) using the same chip enable (CE) and clear (CLR) signals but with their own clock input and output signals. The separate clock input and output signals are referenced to different bits of a signal vector using the variable called index.

http://www.fpgadeveloper.com/2011/07/code-templates-generate-for-loop.html

generate statement

```
entity example_generate is
  generic (
    g_DEBUG      : natural := 1
    );
end example_generate;

 -- 0 = no debug, 1 = print debug
-- Demonstrates Use Case #2: Turning on/off logic
g_KEEP_DEBUG : if g_DEBUG = 1 generate

  p_TEST: process (r_VECTOR) is
  begin
    w_VECTOR_TEST <= r_VECTOR;
  end process p_TEST;

end generate g_KEEP_DEBUG;

-- Demonstrates Use Case #2: Turning on/off logic
g_REMOVE_DEBUG : if g_DEBUG = 0 generate
  w_VECTOR_TEST <= (others => '0');
end generate g_REMOVE_DEBUG;
```

The second use case is very handy for debugging purposes, or for switching out different components without having to edit lots of code. The example below turns on an entire process just by switching g_DEBUG to 1. One interesting thing about generate statements used this way is that the same signal can be driven by multiple generate statements. The designer needs to ensure that these generate blocks are mutually exclusive, such that no two can be active at the same time. Otherwise there will be a problem with the same signal being driven by two sources

https://www.nandland.com/vhdl/examples/example-generate-statement.html

Constant declarations

Constant declarations

- **scalar:**

  `constant` *name*`:` *type* `:=` *expression*`;`

- **array:**

  `constant` *name*`:` *array_type* `[(`*index*`)]` `:=` *expression*`;`

eg:

```
constant Vcc: real := 5.0;
constant Cycle: time := 50 ns;
constant five: bit_vector := "0101";
constant SIX: std_logic_vector (8 to 11):= "0110";
constant H_clk_per : TIME := 10ns;
constant T_CSRS: TIME := H_clk_per/2 - 3ns;
```


Generic declarations

Generic declarations

`generic` `(`*generic_interface_list*`);`

`generic` `(` *name*`:` *type* `:=` *expression*`);`

Declares a static value similar to constant, but the value can be changed from the outside. May be declared in entity (available in all architectures associated with it), block and component. Used for parameterized models (bus width, delay time, etc.)

eg:

```
entity CPU is
  generic (BusWidth : integer := 16)
  port (
    DataIn  : in  bit_vector(BusWidth-1 downto 0);
    DataOut : out bit_vector(BusWidth-1 downto 0);
    ...
```

Generic declarations example – Altera-provided parameterized lpm_ff megafunctions

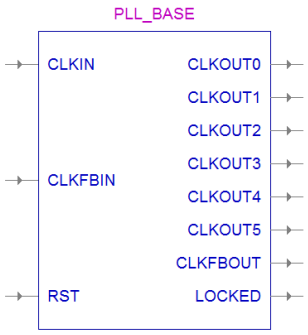
```
ENTITY reggen IS
GENERIC( REG_WIDTH : INTEGER);
PORT(
d : IN STD_LOGIC_VECTOR(REG_WIDTH - 1 DOWNTO 0);
clk : IN STD_LOGIC;
q : OUT STD_LOGIC_VECTOR(REG_WIDTH - 1 DOWNTO 0));
END reggen;

ARCHITECTURE a OF reggen IS
BEGIN
  PROCESS (clk)
     BEGIN
     IF (clk'event AND clk = '1') THEN
       q <= d;
      END IF ;
  END PROCESS;
END a;
```


Generic declarations example usage: generic initialisation via HDL

```
PACKAGE reg24gen_package IS                              -- package definition
CONSTANT TOP_WIDTH : INTEGER := 24;
CONSTANT HALF_WIDTH : INTEGER := TOP_WIDTH / 2;
END reg24gen_package;

USE work.reg24gen_package.ALL;                          -- component instantiation

ENTITY reg24gen IS
PORT( d : IN STD_LOGIC_VECTOR(23 DOWNTO 0);
    clk : IN STD_LOGIC;
      q : OUT STD_LOGIC_VECTOR(23 DOWNTO 0));
END reg24gen;

ARCHITECTURE a OF reg24gen IS
COMPONENT reggen                                        -- recall from package
GENERIC( REG_WIDTH : INTEGER);
PORT( d : IN STD_LOGIC_VECTOR(REG_WIDTH - 1 DOWNTO 0);
    clk : IN STD_LOGIC;
      q : OUT STD_LOGIC_VECTOR(REG_WIDTH - 1 DOWNTO 0));
END COMPONENT;
BEGIN
reg12a : reggen GENERIC MAP (REG_WIDTH => HALF_WIDTH)  -- upload generic data
PORT MAP
 (d => d(HALF_WIDTH - 1 DOWNTO 0),
clk => clk,
  q => q(HALF_WIDTH - 1 DOWNTO 0));
...
```

Lexical elements

Generic declarations example – Xilinx Spartan 6 PLL_BASE component




```
----- CELL PLL_BASE -----
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_SIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;
library SPARTAN6;
use SPARTAN6.vpkg.all;
use SPARTAN6.VCOMPONENTS.all;
entity PLL_BASE is
generic (
                    BANDWIDTH : string := "OPTIMIZED";
                    CLKFBOUT_MULT : integer := 1;
                    CLKFBOUT_PHASE : real := 0.0;
                    CLKIN_PERIOD : real := 0.000;
                    CLKOUT0_DIVIDE : integer := 1;
                    CLKOUT0_DUTY_CYCLE : real := 0.5;
                    CLKOUT0_PHASE : real := 0.0;
                    CLKOUT1_DIVIDE : integer := 1;
                    CLKOUT5_DUTY_CYCLE : real := 0.5;
                    CLKOUT5_PHASE : real := 0.0;
-- and many more parameters ...
```


Generic declarations example – Xilinx Spartan 6 PLL_BASE component usage generic initialisation via BDE

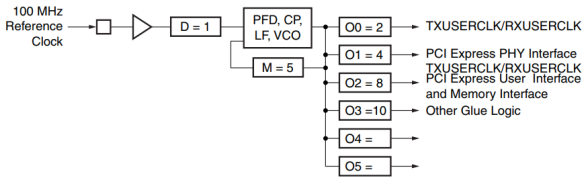
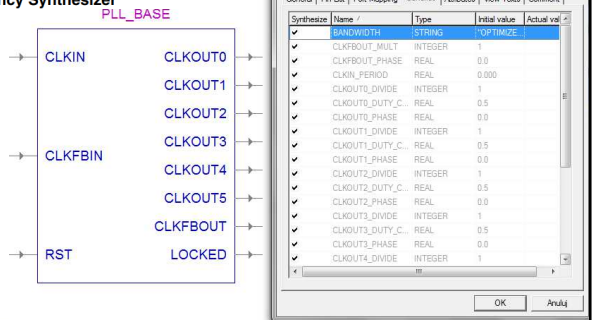
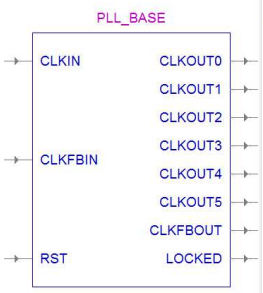


Figure 3-7: **PLL as a Frequency Synthesizer**



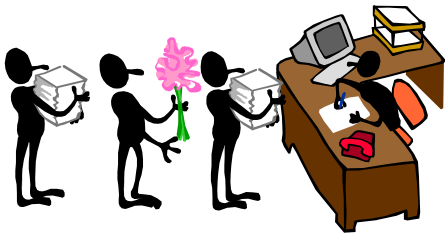
Generic declarations example – Xilinx Spartan 6 PLL_BASE component usage generic initialisation via BDE




```
Library UNISIM;
use UNISIM.vcomponents.all;

-- PLL_BASE: Phase-Lock Loop Clock Circuit
--           Virtex-5, Spartan-6
-- Xilinx HDL Libraries Guide, version 11.2

PLL_BASE_inst : PLL_BASE
generic map (
   BANDWIDTH => "OPTIMIZED",  -- "HIGH", "LOW" or "OPTIMIZED"
   CLKFBOUT_MULT => 1,        -- Multiplication factor for all output clocks
   CLKFBOUT_PHASE => 0.0,     -- Phase shift (degrees) of all output clocks
   CLKIN_PERIOD => 0.000,     -- Clock period (ns) of input clock on CLKIN
   CLKOUT0_DIVIDE => 1,       -- Division factor for CLKOUT0  (1 to 128)
   CLKOUT0_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT0 (0.01 to 0.99)
   CLKOUT0_PHASE => 0.0,      -- Phase shift (degrees) for CLKOUT0 (0.0 to 360.0)
   CLKOUT1_DIVIDE => 1,       -- Division factor for CLKOUT1 (1 to 128)
   CLKOUT1_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT1 (0.01 to 0.99)
   CLKOUT1_PHASE => 0.0,      -- Phase shift (degrees) for CLKOUT1 (0.0 to 360.0)
   CLKOUT2_DIVIDE => 1,       -- Division factor for CLKOUT2 (1 to 128)
   CLKOUT2_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT2 (0.01 to 0.99)
   CLKOUT2_PHASE => 0.0,      -- Phase shift (degrees) for CLKOUT2 (0.0 to 360.0)
   CLKOUT3_DIVIDE => 1,       -- Division factor for CLKOUT3 (1 to 128)
   CLKOUT3_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT3 (0.01 to 0.99)
   CLKOUT3_PHASE => 0.0,      -- Phase shift (degrees) for CLKOUT3 (0.0 to 360.0)
   CLKOUT4_DIVIDE => 1,       -- Division factor for CLKOUT4 (1 to 128)
   CLKOUT4_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT4 (0.01 to 0.99)
   CLKOUT4_PHASE => 0.0,      -- Phase shift (degrees) for CLKOUT4 (0.0 to 360.0)
   CLKOUT5_DIVIDE => 1,       -- Division factor for CLKOUT5 (1 to 128)
   CLKOUT5_DUTY_CYCLE => 0.5, -- Duty cycle for CLKOUT5 (0.01 to 0.99)
   CLKOUT5_PHASE => 0.0,      -- Phase shift (degrees) for CLKOUT5 (0.0 to 360.0)
   COMPENSATION => "SYSTEM_SYNCHRONOUS",  -- "SYSTEM_SYNCHRNOUS",
                                          -- "SOURCE_SYNCHRNOUS", "INTERNAL",
                                          -- "EXTERNAL", "DCM2PLL", "PLL2DCM"
   DIVCLK_DIVIDE => 1,      -- Division factor for all clocks (1 to 52)
   REF_JITTER => 0.100)     -- Input reference jitter (0.000 to 0.999 UI%)
   port map (
   CLKFBOUT => CLKFBOUT,      -- General output feedback signal
   CLKOUT0 => CLKOUT0,        -- One of six general clock output signals
   CLKOUT1 => CLKOUT1,        -- One of six general clock output signals
   CLKOUT2 => CLKOUT2,        -- One of six general clock output signals
   CLKOUT3 => CLKOUT3,        -- One of six general clock output signals
   CLKOUT4 => CLKOUT4,        -- One of six general clock output signals
   CLKOUT5 => CLKOUT5,        -- One of six general clock output signals
   LOCKED => LOCKED,          -- Active high PLL lock signal
   CLKFBIN => CLKFBIN,        -- Clock feedback input
   CLKIN => CLKIN,            -- Clock input
   RST => RST                 -- Asynchronous PLL reset
);
```


Sequential statements

- process (*concurrent!*)
- assignment statements
- wait
- if
- case
- null
- loop
- next
- exit
- assert
- subprograms



loop statement

Suitable for multiplication of logic in behavioral modeling.

Syntax:

```
[label:]
[while condition | for index in valA to valZ] loop
    sequential statements;
end loop;
```

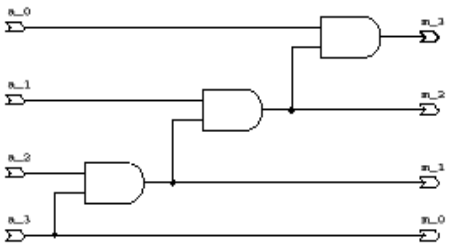
Examples:

```
L: for i in 1 to 10 loop
    sequential statements;
end loop;

M: while i < 11 loop
    equential statements;
    i := i + 1;
end loop;
```

loop statement

Example: multiplication of logic

```
entity multi_and is
   port (a: in bit_vector (0 to 3);
         m: out bit_vector (0 to 3));
end multi_and;

architecture example of multi_and is
begin
  process (a)
    variable b: bit;
    begin
      b := '1';
      for i in 0 to 3 loop
        b := a(3-i) and b;
        m(i) <= b;
      end loop;
  end process;
end example;
```



Sequential statements

next statement

Syntax:

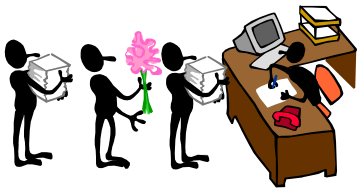
```
next [label] [when condition];
```

Examples:

```
for i in 0 to max_limit loop
   if a(i) = 0 then next;
   end if;
   g(i) := a(i);
end loop;
```

Immediate transition to the next iteration

```
L1: while i < 5 loop
L2: while j < 5 loop
     ...
        next L2 when i=j;
     ...
     end loop L2;
     end loop L1;
```



Sequential statements

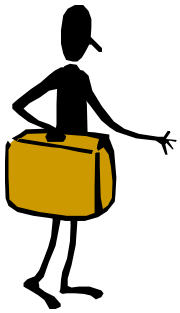
exit statement

Syntax :

```
exit [label] [when condition];
```

Example:

```
for i in 0 to max_limit loop
      if a(i) = 0 then exit;
      end if;
      g(i) := a(i);
end loop;
...
```



Immediate quit

Sequential statements

`assert` statement

Prints a message on the console during simulation and controls the simulation.

Syntax:

```
assert warunek [report string] [severity level];
level: FAILURE | ERROR | WARNING | NOTE
```

Example:

```
assert (Machine_Code /= "0000")
  report "Illegal Opcode"
  severity FAILURE;
```

`assert` statement may occure in the `entity` declaration.

VHDL'92 enables using `report` without `assert`.

```
report „Well done - simulation finished ☺"
```


Sequential statements

`assert` statement – *AHDL help explanation

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 (default)*
- *WARNING can be used in unusual situation in which the simulation can be continued, but the results may be unpredictable;*
- *ERROR can be used when assertion violation makes continuation of the simulation not feasible*
- *FAILURE can be used when the assertion violation is a fatal error and the simulation must be stopped at once.*

Summary

- ***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.***

Sequential statements

`assert` statement examples

```
CHECK_SETUP: process (CLK)
begin
  if (CLK'event and CLK = '1') then
     Q <= D;
     assert D'stable(SETUP_TIME) --ignored by synthesis
     report "Setup Violation..." severity warning;
  end if;
end process CHECK_SETUP;
--
assert packet_length /= 0
report "empty network packet received"
severity warning;
--
assert clock_pulse_width >= min_clock_width
severity error;
```


Assertion based verification

3

Why Assertions

- Assertions are already popular in ASIC design and the will appear in typical FPGA designs soon.
- Although relatively new, they are governed by IEEE standards (PSL, SystemVerilog, VHDL).
- Assertions are relatively simple (once you learn the basics).
- Assertions are based on design specification.
- Assertions create additional layer of safety in simulation (while you concentrate on better synthesis/implementation, they are reminding you about the original specs).
- Assertions create 'live documentation' (better documentation makes management of your design easier).

Assertions – A Practical Introduction for HDL Designers

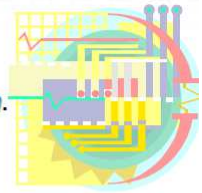
Webinar

www.aldec.com

Key Ideas

- *Assertion Based Verification* deals with significantly more ideas than just assertions:
  sequences → properties → assertions/covers
- The very basic idea of ABV is **property**:
  formalized description of certain behavior of your design
  e.g. "broken window triggers alarm",
  "security responds to alarm in 30 seconds".
- Properties can be used by verification tools in many ways:
  - `assert <property>` verifies that ***bad things do not happen***.
    e.g. "assert [that] broken window always triggers alarm".
  - `cover <property>` verifies that ***good things happen***.
    e.g. "cover response to triggered alarm in 30 seconds".

The World of Properties



- Typical digital design specification is full of ***design properties*** expressed in ***plain English***.
- Designer rewrites properties in ***HDL*** with correct ***hardware implementation*** in mind.
- ***Properties, assertions*** and ***covers*** represent ***pure behaviors*** (desired or undesired) of the design:
  - They can be very efficient ***documentation*** of the design.
  - They work as a ***reference*** during design verification.
  - They are accepted by variety of functional and formal ***verification tools*** .

Sequential statements

Subprograms – `function` statement

Functions return a single value. When the function is called the formal parameters are given the values of the actual parameters.

Syntax:

```
function name [(parameter: type;...)] return type is
declarations
begin
    sequential statements;
end [name];
```

Example: (from std_logic_1164 library, checks for *unknown* values)

```
function Is_X (s : STD_ULOGIC) return BOOLEAN is
begin
  case s is
    when 'U' | 'X' | 'Z' | 'W' | '-' => return true;
    when others                       => null;
  end case;
  return false;
end function Is_X;
```


Sequential statements

Subprograms – `procedure` statement

Procedures may return more then one value. A procedure itself does not return a value, but does formal parameters that are replaced by the values of the actual parameters

Syntax:

```
procedure name [(c;...)] is
declarations
begin
    sequential statements;
end [name]

parameters :
{[variable] names: [in | out | inout] type [:= expression]; |
 [signal] names: [in | out | inout] type;}
```

Passing the parameters to the procedure : `in inout`
Passing the parameters from the procedure : `out inout`

Sequential statements

Subprograms – `procedure` statement

Example:

```
procedure vect_to_int (z: in bit_vector (1 to 8);
                       zero_flag: out boolean;
                       q: inout integer) is
begin
   q := 0;
   zero_flag := true;
   for i in 1 to 8 loop
      q := q * 2;
      if z(i) = '1' then
         q := q + 1;
         zero_flag := false;
      end if;
   end loop;
   return;
end vect_to_int;
```

Call: `vect_to_int (s,t,u);`

Concurrent statements

Concurrent subprogram call

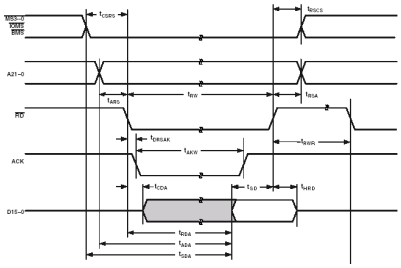
In both below examples, the effect is the same:

```
architecture concurrent of SUB_CALL is
begin
   vect_to_int (bitstuff, flag, number);
end concurrent ;


architecture sequential of SUB_CALL is
begin
   process (bitstuff, number)
   begin
      vect_to_int (bitstuff, flag, number);
   end process;
end sequential ;
```

Sequential statements

Subprograms – procedure example

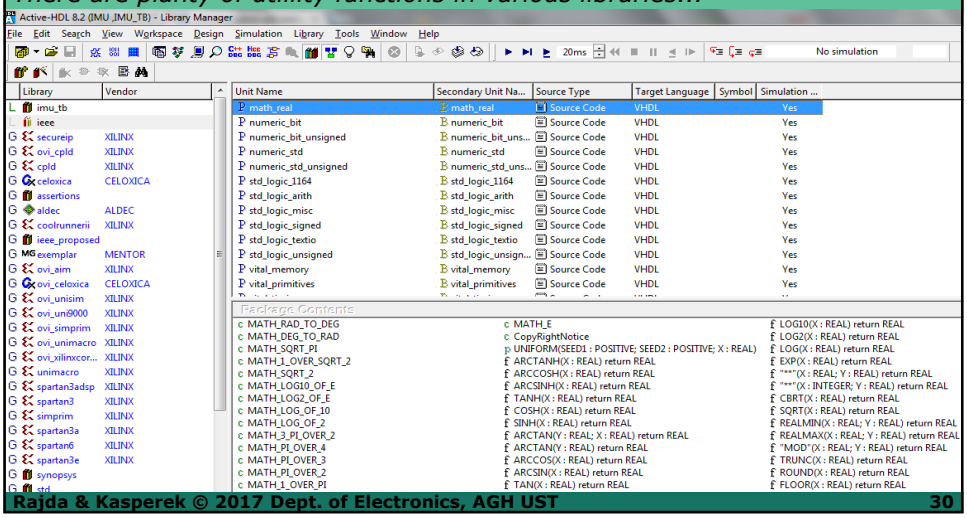
```
procedure read_cycle (
 address:      in  STD_LOGIC_VECTOR(15 downto 0); -- input parameter
 signal addr: out STD_LOGIC_VECTOR(15 downto 0); -- DSP addres bus
 signal data: in  STD_LOGIC_VECTOR(15 downto 0); -- DSP data
 data_read:    out STD_LOGIC_VECTOR(15 downto 0); -- reading result
 signal rd:    out STD_LOGIC;                     -- DSP signal
 signal ms3:   out STD_LOGIC                      -- DSP signal)
is

constant T_CSRS: TIME := H_clk_per/2 - 3ns;
constant T_ARS:  TIME := H_clk_per/2 - 3ns;
constant T_RW:   TIME := H_clk_per - 2ns + (wait_states*H_clk_per);
constant T_RSA:  TIME := H_clk_per/2 - 2ns;

begin
   ms3 <= '0';
   addr <= address;
   wait for T_ARS;
   rd <= '0';
   wait for T_RW;
   rd <= '1';
   data_read := data;
   wait for T_RSA;
   addr <= (others => 'Z');
   ms3 <= '1';
end procedure read_cycle;
```



Sequential statements

Subprograms – function/procedure statements

Subprograms are used mostly in testbenches.
There are planty of utlility functions in various libraries...



Concurrent statements

- signal assignment statements
  - unconditional
  - conditonal
  - selected
- **subprograms**
- **`block`**

Concurrent statements

`block` statement

Groups concurrent statements.
Blocks can be nested to form a hierarchy.

Syntax:

[*label* :] `block` [(*boolean_expression*)] [`is`];
[*declarations*]
`begin`
    concurrent statements ;
`end block` [*label*] ;

declarations:

- constants, types, signals
- subprograms
- `use` statements
- components

block statement

Boolean_expression automatically generates the `guard` signal, which may be used for conditioning the signal assignments through the `guarded` clause.

Example:

```
B1: block (control)
begin
    s <= guarded '1';
end block B1;
```

`s <= '1'` assignment will occure, if `control` equals `true`.

Note! Not all of the tools allow for the use of this option.