Performance Engineering of Real-Time and Embedded Systems

```
108  library ieee;
109  use ieee.std_logic_1164.all;
110  use ieee.std_logic_arith.all;
111
112  entity asyncFun is
113    port (
114      signal reset : in std_logic;
115      signal clk : in std_logic;
116      signal request : in std_logic;
117      signal i1 : in std_ulogic_vector(31 downto 0);
118      signal r_e_t_u_r_n : out std_ulogic_vector(31 downto 0);
119      signal acknowledge : out std_logic);
120  end;
121
122  architecture test of asyncFun is
123    signal val : std_ulogic_vector(31 downto 0);
124    signal count : unsigned(31 downto 0);
125    signal done : std_ulogic;
126  begin  -- test
127    process (clk)
128    begin  -- process
129      if clk'event and clk = '1' then  -- rising clock edge
130        if request = '1' then
131          val <= i1;
```

Introduction to VHDL

VHDL designs are decomposed into blocks.

- A block has an entity/architecture pair.
  - ***Entity describes the interface***
  - ***Architecture describes the behavior***

```
entity or2 is
    port(a, b : in bit;
         sw : in std_logic_vector( 7 downto 0 );
         y : out bit);
end entity or2;

architecture arch_or2 of or2 is
begin
    y <= a or b;
end architecture arch_or2;
```

List clocks, resets, input, and output in that order.

Port modes: in, out, inout, buffer

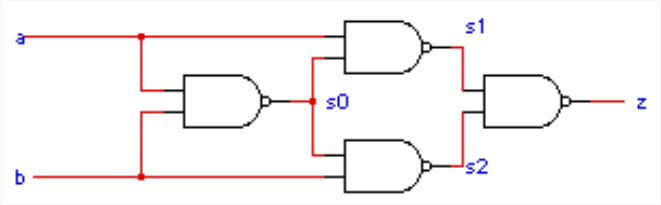
The architecture defines aspects of the behavior.

```
architecture_statement::=
architecture ARCHITECTURE_NAME of ENTITY_NAME is

[architecture-item-declaration]
-- here you can declare signals, constants, functions, procedures…
-- component declarations
-- no variable declarations !!!,

begin

   concurrent-statements:
      - concurrent simple signal assignment
      - process statement
         inside process you can use only sequential statements
      - conditional signal assignment
      - selected signal assignment
      - component instantiation statement
      - generate statement
            ……

end;
```

concurrent statements

- **A concurrent statement is "executed" when an event occurs on any signal that could effect the value being assigned.**

Components are used as building blocks and many can be instantiated.




```
entity XOR2 is
   port (a, b : in bit; z : out bit);
end XOR;

architecture nand2_gates of XOR2 is
signal s0, s1, s2: bit;

component  nand2
 port (a, b : in bit; z : out bit);
end;

begin
  U1: nand2  port map ( a, b, s0 );
  U2: nand2  port map ( a, s0, s1);
  U3: nand2  port map ( b, s0, s2);
  U4: nand2  port map ( s1, s2, z);
end  nand2_gates;
```

Describes physical connections between functional units

Constants can be declared for use in other places in the architecture.

constant *constant_name : type_name := constant_value;*

```
constant NUMBER_OF_BYTES : integer := 4;
constant NUMBER_OF_BITS : integer := 8 * NUMBER_OF_BYTES;
-- constant can be an expression

constant GATE_DELAY : time := 8 ns; -- time is predefined type
constant A, B : bit := '0';         -- both A and B are equal to '0'
constant C_VECTOR : bit_vector := "0001";
```

Signals hold values.

- Use signals as:
  - Input and output ports in entities
  - Like wires as interconnections between components inside architecture
  - A place holder for a value – model a flip-flop
- They can have history.

signal *list_of_signal_names : type_name [:= initial_value];*

```
signal SIG1 : integer := 5;
signal S1   : bit := '1';

signal DATA_BUS : bit_vector (0 to 7);
signal DATA_OUT : std_logic_vector (7 downto 0):= (0=>'1', others=>'U');
```

```
SIG_a <= '1';
SIG_b <= not SIG_a;
SIG_c <= a and b;
```

Subtypes allow you to create your own signal types.

subtype *subtype_name* **is** *type_description;*

```
subtype error_code is natural range 0 to 8;
constant NO_ERROR : error_code := 0;
constant TOO_HIGH : error_code := 1;
signal error : error_code := NO_ERROR;
```

Variables can hold values similar to standard programming languages.

- Use variables to:
  - ***Hold a single value of a given type.***
- Variables can be used and declared only inside processes.
- They do not have or cause events.

variable list_of_*variable_names : type_name [:= initial_value];*

```
v_a:='1';
v_a:='0';
```

You can declare arrays of values of the same data type.

- Arrays are very useful for buses or grouping of data.

```
type array_type_name is array (discrete range) of element_type;
```

```
type word is array (0 to 31) of bit;


SIGNAL a: BIT_VECTOR(0 TO 3);      -- ascending range
SIGNAL b: BIT_VECTOR(3 DOWNTO 0); -- descending range

   a <=  "0111"; -- double quotes used for vectors
   b <=  x"8";   -- indicates hex value
   switches <= sw7 & sw6 & sw5 & sw4; -- concatenate signals
   a(0) <= b(0); -- accessing bits in a vector
```

There is a full set of operators available for your use in expressions.

1. Binary logical operators : **and or nand nor xor xnor**
2. Relational operators : **= /= < <= > =>**
3. Shift operators : **sll srl sla sla sra rol ror**
4. Adding operators : **+ - & (concatenation)**
5. Unary sign operators : **+ -**
6. Multiplying operators : *** / mod rem**
7. Miscellaneous operators : **not abs ****

VHDL has two types of statements.

- Concurrent
  - ***All statements outside of a process***
  - ***A process block is considered one concurrent statement***
  - ***All concurrent statements execute in parallel; order does not matter***
- Sequential
  - ***Evaluated sequentially***
  - ***Statements grouped within a process***

You can make concurrent assignments to signals conditionally.

Conditional_signal_assignment ::=

target <= [**guarded**][**transport**] {*waveform* **when** *condition* **else**} waveform;

```
Y <=  A when SEL = "00" else
      B when SEL = "01" else
      C when SEL = "11" else
      D;
```

- First one found true sets value.
- Evaluated when an event occurs on signal on right side.

selected signal assignment::=

with *expression* **select**
target <= [**guarded**] [**transport**]
{*waveform* **when** choices ,}
waveform **when** choices;

```
with SEL select
   Y <=  A when "00",
         B when "01",
         C when "10",
         D when others;
```

- Good practice will cover all values, or have a default.
- Only one can be true to set value.
- Evaluated when an event occurs on signal on right side of select.

Process statement groups a set of sequential statements for execution.

```
process_statement::=
[process_label: ] process [ ( sensitivity_list ) ]
                        process_declaration_part -- define variables
                    begin
                        sequence_of_sequential_statements;
                    end process [process_label ] ;
```

- Process executes when a change occurs in a signal on the sensitivity list.
- Process sequentially executes statements until end of process or a wait statement.
- Process has either a wait statement or sensitivity list but not both.
  - ***Sensitivity list → resume when an event occurs on any signal in the sensitivity list***
  - ***Wait statement → resume when wait condition is met***

There are several types of statements allowed in a process.

- Here are some that may be of interest:
  - ***variable assignment ':='***
  - ***signal assignment '<='***
  - ***wait statement (if no sensitivity list)***
  - ***if statement***
  - ***case statement***

The if and case statements work as in standard programming languages.

```
If_Statement::=
if condition then
        sequence_of_sequential_statements;
{elsif condition then
        sequence_of_sequential_statements;}
[else
        sequence_of_sequential_statements;]
end if;
```

```
process (sel, a, b)

begin
  if sel = '1' then
   f <= a;
  else
   f <= b;
  end if;
end process;
```

```
Case_Statement::=
case expression is
  when choice(s) =>
      sequence_of_sequential_statements;
  {when choice(s) =>
      sequence_of_sequential_statements;}
  [when others =>
      sequence_of_sequential_statements;]
end case;
```

```
process (A, B, C, X)
   begin
        case  X  is
           when  0          =>   Z <= A;
           when  7 |  9     =>   Z <= B;
           when  1 to 5     =>   Z <= C;
           when others      =>   Z <= 0;
        end case;
end process;
```

- Again, be sure to cover all values of input, or have a default.

The wait statement causes suspension of the process execution.

There are several different types of wait statements:

wait for a specific time	**wait for** *specific_time;*
wait for a signal event	**wait on** *signal_event;*
wait for a true condition (requires an event)	**wait until** *condition;*
indefinite (process is never reactivated)	**wait;**

An event on a signal represents a change in that signal.

- Use events to trigger activity on signal edges, such as, a clock edge.

```
if rising_edge(clk) then
    cntr <= cntr + 1;
end if;
```

There are many VHDL practices to follow, but these are ones that are relevant to your work.

- Make sure you define an output for all combinations of inputs. (Select, case, if…elsif…else)
- Make sure all inputs to combinatorial logic in a process statement are listed on the sensitivity list.
- For good synchronous design, always use event checking on the clock, i.e. the clock in a sensitivity list is not enough.
- Never use events or edges in combinational logic.
- If you set a signal value in a process, do not use that signal later in that same process.
- Place purely combinational logic, which does not contribute to sequential operation, outside a process.

There are many tutorials available on the web.

- VHDL Mini-Reference
  - ***http://www.eng.auburn.edu/department/ee/mgc/vhdl.html***
- Links to several VHDL Tutorials
  - ***http://www.fpga4fun.com/HDL%20tutorials.html***
- Thanks go to Prof. Lukowiak for allowing some of his notes to be used here.

Software and hardware design are less different than software designers think, but more different than hardware designers think.

-- Fred Brooks, Keynote address, OOPSLA 2007

Father of IBM's OS360

Author of The Mythical Man-Month