Introduction to VHDL

VHSIC Hardware Description Language

VHDL

VHSIC Hardware Description Language

- **Language to specify digital systems**

  (VHSIC – Very-High-Speed Integrated Circuit)
- **Development started in 1983 sponsored by the US Department of Defense (DoD)**
- **IEEE standard in 1987 and 1993 and 2001.**

- **Other languages used for hardware description**
  - SystemC, **Verilog**, Esterel, SDL, . . .

VHDL vs Software

VHDL is not Software!

It is a "tool" for hardware specification.

- **Specific characteristics for hardware description**
  - **Implicit/explicit notion of time**
  - **Concurrency implied by the language itself**
  - **Capability to instantiate components and define structure**
- Developed for specification and simulation of digital circuits but currently used for synthesis, verification, etc.

Advantages / Disadvantages

Advantages

- Design specification independent of implementation technology
- Reduces design time and cost (specification uses higher levels of abstraction)
- Supported by “all” digital system design tools
  - Flexibility in tool choice and in design reuse
- The hardware can be specified at different levels of abstraction

Disadvantages

- The synthesized hardware may be less optimized than if manually designed (at the logic level)
- Simulation model ≠ synthesis model

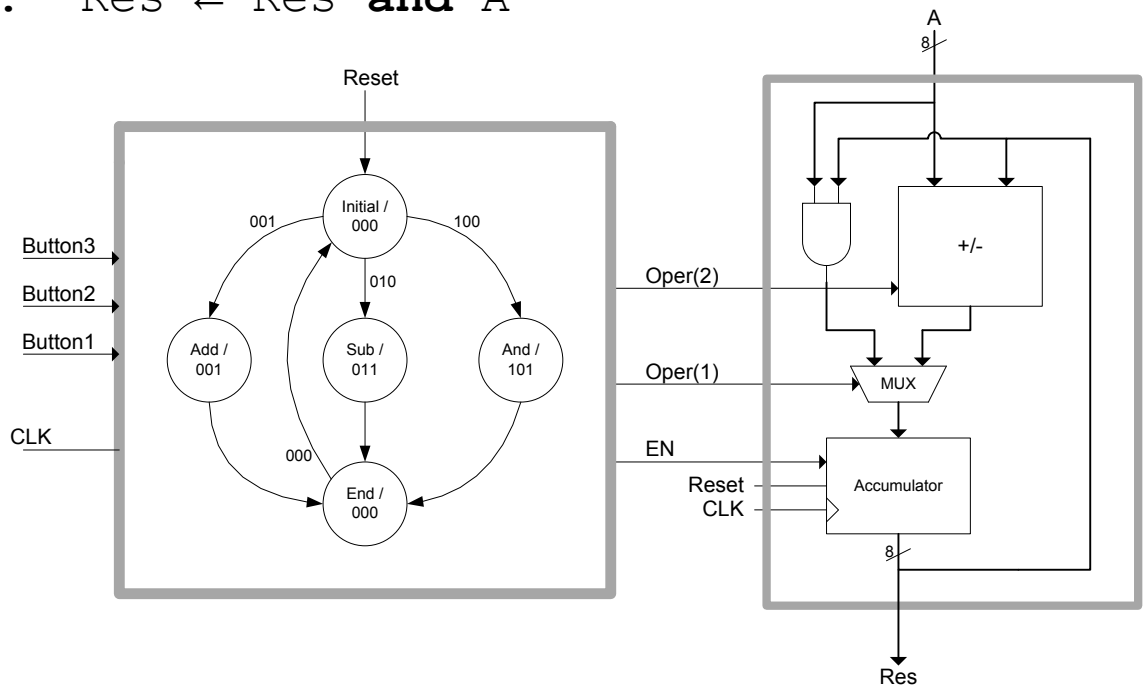
To be implemented in Introductory Lab

BASIC CIRCUIT EXAMPLE

VHDL Circuit Example

Circuit executes one of 3 operations (selected by push-button):

```
ADD:  Res ← Res + A
SUB:  Res ← Res - A
AND:  Res ← Res and A
```



Example Datapath Unit (1)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity datapath is
  port ( a    : in std_logic_vector (7 downto 0);
         oper : in std_logic_vector (1 downto 0);
         clk, en_accum, rst_accum : in std_logic;
         res : out std_logic_vector (7 downto 0));
end datapath;
```

Example Datapath Unit (2)

```
architecture behavioral of datapath is
  signal addsub, log_and, res_alu :
     std_logic_vector (7 downto 0);
  signal accum :
     std_logic_vector (7 downto 0) := (others => '0');
begin
  -- adder/subtractor
  addsub <= accum + a when oper(0)='0' else accum - a;
  -- logic unit
  log_and <= a and accum;
  -- multiplexer
  res_alu <= addsub when oper(1)='0' else log_and ;
  -- saida
  res <= accum;
```

Example Datapath Unit (3)

```
-- accumulator
  process (clk)
  begin
    if clk'event and clk='1' then
      if rst_accum='1' then
        accum <= X"00";
      elsif en_accum = '1' then
        accum <= res_alu;
      end if;
    end if;
  end process;

end behavioral;
```

Example Control Unit (1)

```
entity control is
  port (
    clk, rst : in std_logic;
    instr : in std_logic_vector (2 downto 0);
    enable : out std_logic;
    oper : out std_logic_vector (1 downto 0));
end control;

architecture Behavioral of control is
  type fsm_states is ( s_initial, s_end, s_down,
                       s_add, s_and );
  signal currstate, nextstate: fsm_states;
begin
  -- process 1
  -- process 2
end
```

Example Control Unit (2)

```
state_reg: process (clk, rst)
 begin
   if rst = '1' then
     currstate <= s_initial ;
   elsif clk'event and clk = '1' then
     currstate <= nextstate ;
   end if ;
 end process;
```

Example Control Unit (3)

```
state_comb: process (currstate, instr)
 begin
   nextstate <= currstate ;
   -- by default, does not change the state.

   case currstate is
     when s_initial =>
       if instr="001" then
         nextstate <= s_add ;
       elsif instr="010" then
         nextstate <= s_down ;
       elsif instr="100" then
         nextstate <= s_and;
       end if;
       oper <= "00";
       enable <= '0';
```

Example Control Unit (4)

```
when s_add =>
  nextstate <= s_end;
  oper <= "00";
  enable <= '1';

when s_down =>
  nextstate <= s_end;
  oper <= "01";
  enable <= '1';

when s_and =>
  nextstate <= s_end;
  oper<="10";
  enable<='1';
```

```
   when s_end =>
     if instr="000" then
       nextstate <= s_initial;
     end if;
     oper <= "00";
     enable<='0';

  end case;
end process;
```

Circuit = Control + Datapath (1)

```
entity circuito is
  port (
    clk, rst: in std_logic;
    instr: in std_logic_vector(2 downto 0);
    data_in: in std_logic_vector(7 downto 0);
    res: out std_logic_vector(7 downto 0)
    );
end circuito;
```

Circuit = Control + Datapath (2)

```
architecture Behavioral of circuito is
  component control
    port ( clk, rst : in std_logic;
           instr : in std_logic_vector(2 downto 0);
           enable : out std_logic;
           oper : out std_logic_vector(1 downto 0) );
  end component;
  component datapath
    port( a : in std_logic_vector(7 downto 0);
          oper : in std_logic_vector(1 downto 0);
          en_accum, rst_accum, clk : in std_logic;
          res : out std_logic_vector(7 downto 0) );
  end component;

  signal enable : std_logic;
  signal oper : std_logic_vector(1 downto 0);
```

Example Circuit = Control + Datapath (3)

```
begin
  inst_control: control port map(
    clk => clk,
    rst => rst,
    instr => instr,
    enable => enable,
    oper => oper
    );
  inst_datapath: datapath port map(
    a => data_in,
    rst_accum => rst,
    en_accum => enable,
    oper => oper,
    clk => clk,
    res => res
    );
end Behavioral;
```

Example TestBench (1)

```
ENTITY tb_circuito IS
END tb_circuito;

ARCHITECTURE behavior OF tb_circuito IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT circuito
      PORT( ... );
    END COMPONENT;

   -- Inputs
   signal clk : std_logic := '0';
   signal rst : std_logic := '0';
   signal instr : std_logic_vector(2 downto 0) := (others => '0');
   signal data_in : std_logic_vector(7 downto 0) := (others => '0');

   -- Outputs
   signal res : std_logic_vector(7 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;
```

Example TestBench (2)

```
BEGIN
  -- Instantiate the
  -- Unit Under Test (UUT)
  uut: circuito PORT MAP (
          clk => clk,
          rst => rst,
          instr => instr,
          data_in => data_in,
          res => res
        );

   -- Clock process definitions
   clk_process: process
   begin
     clk <= '0';
     wait for clk_period/2;
     clk <= '1';
     wait for clk_period/2;
   end process;
```

```
-- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns
      wait for 100 ns;
      wait for clk_period*10;

      -- insert stimulus here
      rst <= '1' after 20 ns,
             '0' after 40 ns;
      data_in <= X"67" after  40 ns,
                 X"12" after 120 ns,
                 X"C3" after 200 ns;
      instr <= "001" after  40 ns,
               "000" after  80 ns,
               "010" after 120 ns,
               "000" after 160 ns,
               "100" after 200 ns,
               "000" after 300 ns;
      wait;
   end process;
END;
```

Coding examples

VHDL LANGUAGE CONSTRUCTS

VHDL Design Units

Package

- Defines common data types, subprograms and components to be reused in multiple entities.

Entity

- Defines the external interface of the circuit or sub-circuit (input and output ports)

Architecture

- Defines the implementation of the circuit.
- Common use: one architecture per entity.

Configuration

- Multiple architectures may be developed for one entity
- The configuration indicates which architecture body is to be used with the given entity declaration

Circuit Specification

ENTITY *circuito* **IS**

definition of input and output ports

END *circuito*;

ARCHITECTURE *arqui1* **OF** *circuito* **IS**

declaration of types, constants, signals and components

BEGIN

concurrent and/or sequential statements

END *arqui1* ;

Basic VHDL data types

Standard:

bit
('0', '1')

boolean
(true, false)

bit_vector

integer

IEEE standard:

std_logic
('U','X','0','1','Z',
 'W','L','H','-')

std_logic_vector

Other standard types not supported for synthesis:

real, file, character, physical

STD_LOGIC

9-valued logic

'U' means uninitialized

'X' means unknown

'0' means low

'1' means high

'Z' means high impedance

'W' means weak unknown

'L' means weak low

'H' means weak high

'-' means don't care

For XST synthesis:

- The '0' and 'L' values are treated identically, as are '1' and 'H'.
- The 'X', and '-' values are treated as don't care.
- The 'U' and 'W' values are not accepted by XST.
- The 'Z' value is treated as high impedance.

IEEE libraries

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

These two first IEEE library statements are automatically included by the Xilinx ISE design editor.

This package defines the new data types:

std_logic

std_logic_vector

and a set of operations on them.

Std_logic signal types

Always use `std_logic` for simple signals,
and `std_logic_vector` for bus signals.

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

The signal has 8 bits, the leftmost bit (the MSB bit) has index 7, and the rightmost bit (LSB) has index 0.

```
a(5) -- accesses a single bit
a(4 downto 1) -- accesses part of the array (4 bits)
```

```
signal b : std_logic_vector (0 to 7);
```

The ascending range is seldom used because it may be confusing when the array represents a binary number.

Operators

Logical Operators: **and, or, nand, nor, xor, xnor, not**

Relational Operators: **=, /=, <, <=, >, >=**

& (concatenation)

Arithmetic Operators: **+ , - , ***

/, mod, rem

Supported if the right operand is a constant power of 2

** *Only supported if the left operand is 2*

Array aggregate

VHDL construct to assign a value to an array:

```
signal a, b, c : std_logic_vector (7 downto 0);

a <= "10110011";
a <= B"10110011";
a <= X"B3";

b <= "00000000";
b <= (others => '0');

c <= "10100000";
c <= (7 => '1', 5 => '1', others => '0');
```

Packages for Numeric Operations

Using Std_Logic_Arith - by Synopsys, a defacto industry standard

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
```

Most software tools store these packages in the ieee library.
Used in our examples (and in XST coding examples)

Using Numeric_Std - (Recent) IEEE standard

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

Use std_logic_arith or numeric_std, but not both

Std_Logic_Arith packages (1)

The **std_logic_unsigned** and **std_logic_signed** packages define overloaded arithmetic operators for the **std_logic_vector** data type.

- the std_logic_vector data type is interpreted as an **unsigned** or as a **signed** binary number, according to the package chosen.
- only one of the two packages can be chosen for implementation.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
...
signal x1, x2: std_logic_vector(3 downto 0);
signal x3: std_logic_vector(7 downto 0);
...
x3 <= x2 * x1;   -- Ex: 1000 * 0011 = 00011000
```

Std_Logic_Arith packages (2)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
...
signal x1, x2: std_logic_vector(3 downto 0);
signal x3: std_logic_vector(7 downto 0);
...
x3 <= x2 * x1;   -- Ex: 1000 * 0011 = 11101000
```

Numeric_Std package

Defines two new data types: **unsigned** and **signed**.
Both are arrays of **std_logic** elements. May require type conversions.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
...
signal u1, u2, u3: unsigned(3 downto 0);
signal s1, s2, s3: signed(3 downto 0);
signal x1, x2, x3: std_logic_vector(3 downto 0);
...
u3 <= u2 + u1;
s3 <= s2 + s1;
u2 <= unsigned(x2);          -- requires type casting
x3 <= std_logic_vector(u3); -- requires type casting
```

VHDL language constructs

- Concurrent statements (executed in parallel)
  - Concurrent signal assignment
    - simple, selected, conditional
  - For ... generate
  - Component instantiation
  - Processes

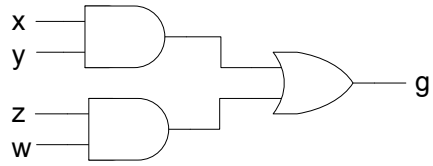
- Sequential Statements (executed sequentially in a process)
  - Assignments
  - If ... then ... else ...
  - Case
  - For ... loop
  - Wait

Concurrent VHDL statements

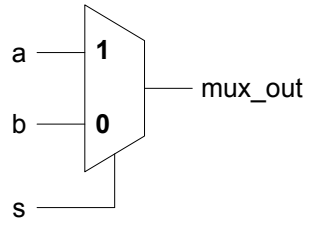
There is a clear mapping between the VHDL statements and the corresponding hardware components.

- **The designer must always be able to relate the block diagram of the circuit to the textual specification statements.**

```
g <= (x and y) or (z and w);
```




```
mux_out <= a when s='1' else b;
```



Sequential VHDL statements

Sequential statements look like those of a traditional programming language.

Not compatible with the concurrent execution model of VHDL
→ **must be enclosed inside a VHDL process**.

Inexperienced designers are lead to think that they can synthesize hardware directly from sequential descriptions.

May lead to **incomprehensible** or/and **erroneous** implementations.

Specifications using processes and sequential statements must be coded in a **very disciplined** way, such that the code can be **realistically synthesized** to the intended hardware.

Simple check: can you easily relate the hardware diagram from the set of sequential statements?

Behavioral Specification of a Full Adder (1)

```
entity fulladder is
  port ( x, y, cin : in std_logic ;
         sum, cout : out std_logic) ;
end fulladder;

architecture tabela1 of fulladder is
  signal aux : std_logic_vector(2 downto 0);
begin
  aux <= x & y & cin ;
  sum <= '1' when aux = b"100" or aux = b"010"
               or aux = b"001" or aux = b"111"
         else '0' ;
  cout <= '1' when aux = b"110" or aux = b"011"
                or aux = b"101" or aux = b"111"
          else '0' ;
end tabela1;
```

Behavioral Specification of a Full Adder (2)

```
entity fulladder is
  port (x, y, cin : in std_logic;
        sum, cout : out std_logic) ;
end fulladder;

architecture tabela2 of fulladder is
  signal aux : std_logic_vector(2 downto 0);
begin
  aux <= x & y & cin ;
  with aux select
    sum <= '1' when  b"100" | b"010" | b"001" | b"111",
           '0' when others ;
  with aux select
    cout <= '1' when b"110" | b"011" | b"101" | b"111",
            '0' when others ;
end tabela2;
```

Behavioral Specification of a Full Adder (3)

```
entity fulladder is
  port(x, y, cin : in std_logic;
       sum, cout : out std_logic) ;
end fulladder;

architecture equacoes of fulladder is
begin
  sum <= x xor y xor cin;
  cout <= (x and y) or (x and cin) or (y and cin);
end equacoes ;
```

Mixed model (structural and behavioral) of a Full Adder

```
entity fulladder is
  port (x, y, cin : in std_logic;
        sum, cout : out std_logic) ;
end fulladder;

architecture mista of fulladder is
  component half_adder
    port (x, y: in std_logic;
          cout, sum: out std_logic);
  end component;
  signal a, b, c : std_logic ;
begin
  u1: half_adder port map(x, y, a, b) ;
  u2: half_adder port map(b, cin, c, sum) ;
  cout <= a or c ;
end mista;
```

8-to-1 multiplexer

```
entity mux8 is
  port (
    sel: in std_logic_vector
                (2 downto 0);
    inp: in std_logic_vector
                (7 downto 0);
    z, z_bar: out std_logic);
end mux8;
```

XST complains!

"A value is missing in select."

```
architecture a of mux8 is
 signal temp : std_logic;
begin
 with sel select
  temp <= inp(0) when "000",
          inp(1) when "001",
          inp(2) when "010",
          inp(3) when "011",
          inp(4) when "100",
          inp(5) when "101",
          inp(6) when "110",
          inp(7) when "111";
  z <= temp ;
  z_bar <= not temp ;
end a;
```

8-to-1 multiplexer

```
entity mux8 is
  port (
    sel: in std_logic_vector
                (2 downto 0);
    inp: in std_logic_vector
                (7 downto 0);
    z, z_bar: out std_logic);
end mux8;
```

XST OK!

```
architecture a of mux8 is
 signal temp : std_logic;
begin
 with sel select
  temp <= inp(0) when "000",
          inp(1) when "001",
          inp(2) when "010",
          inp(3) when "011",
          inp(4) when "100",
          inp(5) when "101",
          inp(6) when "110",
          inp(7) when others;
  z <= temp ;
  z_bar <= not temp ;
end a;
```

Multiplexer (using sequential if)

```
entity mux4s is
  port (a, b, c, d : in std_logic;
        s : in std_logic_vector (1 downto 0);
        o : out std_logic);
end mux4s;
architecture archi of mux4s is
begin
  process (a, b, c, d, s)
  begin
    if (s = "00") then o <= a;
    elsif (s = "01") then o <= b;
    elsif (s = "10") then o <= c;
    else o <= d;
    end if;
  end process;
end archi;
```

Multiplexer (using sequential case)

```
entity mux4c is
 port (
  a, b, c, d : in std_logic;
  s : in std_logic_vector
                (1 downto 0);
  o : out std_logic);
end mux4c;
```

```
architecture archi of mux4c
is
begin
  process (a, b, c, d, s)
  begin
   case s is
     when "00" => o <= a;
     when "01" => o <= b;
     when "10" => o <= c;
     when others => o <= d;
   end case;
  end process;
end archi;
```

8-to-1 multiplexer (index conversion)

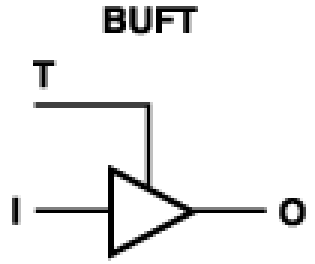
```
entity mux8i is
  port (
    sel: in std_logic_vector(2 downto 0);
    inp: in std_logic_vector(7 downto 0);
    z: out std_logic);
end mux8i;

architecture a of mux8i is
begin
  z <= inp(conv_integer(sel));
end a;
```

Decoder

```
entity dec is
  port (sel: in std_logic_vector (2 downto 0);
        res: out std_logic_vector (7 downto 0));
end dec;

architecture archi of dec is
begin
  res <= "00000001" when sel = "000" else
         "00000010" when sel = "001" else
         "00000100" when sel = "010" else
         "00001000" when sel = "011" else
         "00010000" when sel = "100" else
         "00100000" when sel = "101" else
         "01000000" when sel = "110" else
         "10000000";
end archi;
```

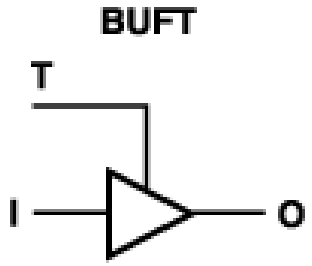
Buffer Tri-State (1)

```
entity three_st is
  port(T : in std_logic;
       I : in std_logic;
       O : out std_logic);
end three_st;

architecture archi of thre
begin
  process (I, T)
  begin
    if (T = '1') then
      O <= I;
    else
      O <= 'Z';
    end if;
  end process;
end archi;
```



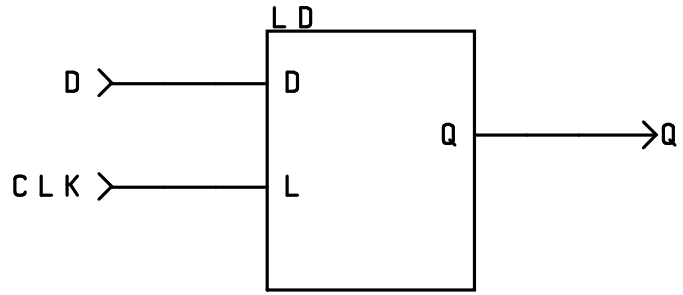
Buffer Tri-State (2)

```
entity three_st is
  port(T: in std_logic;
       I: in std_logic;
       O: out std_logic);
end three_st;

architecture archi of three_st is
begin
  O <= I when (T='1') else 'Z';
end archi;
```



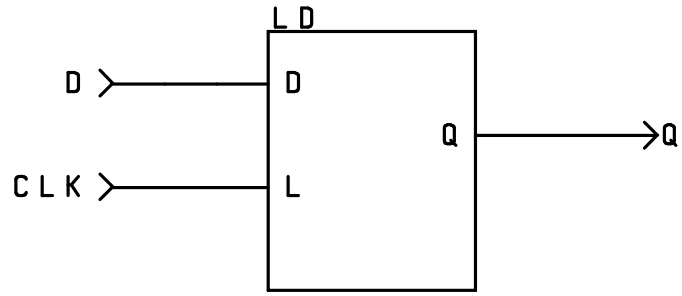
Latch D

```
entity latchd is
  port (d, clk: in std_logic;
        q: out std_logic);
end latchd;

architecture a of latchd is
begin
  process (clk, d)
  begin
    if (clk='1') then
      q <= d;
    end if;
  end process;
end a;
```



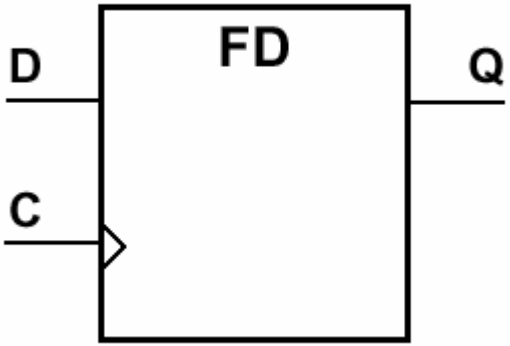
Latch D

```
entity latchd is
  port (d, clk: in std_logic;
        q: out std_logic);
end latchd;


architecture a of latchd is
begin
  q <= d when clk='1' ;
end a;
```



Flip-Flop D

```
entity flop is
  port(C, D : in std_logic;
       Q : out std_logic);
end flop;

architecture archi of flop is
begin
  process (C)
  begin
    if (C'event and C ='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

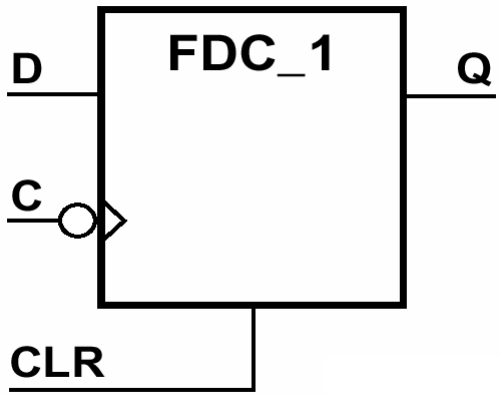


8-bit Register

```
entity reg8 is
  port(C : in std_logic;
       D : in std_logic_vector (7 downto 0);
       Q : out std_logic_vector (7 downto 0));
end reg8;

architecture archi of reg8 is
begin
  process (C)
  begin
    if (C'event and C = '1') then
      Q <= D;
    end if;
  end process;
end archi;
```

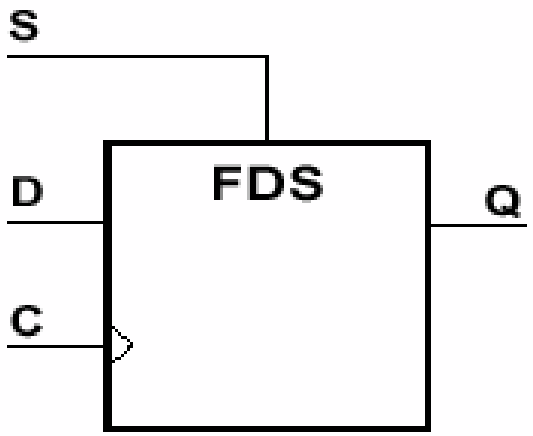
Flip-Flop D edge-triggered negative, asynchronous reset

```
entity flop is
  port(C, D, CLR : in std_logic;
       Q : out std_logic);
end flop;

architecture archi of flop is
begin
  process (C, CLR)
  begin
    if (CLR = '1')then
      Q <= '0';
    elsif (C'event and C='0') then
      Q <= D;
    end if;
  end process;
end archi;
```



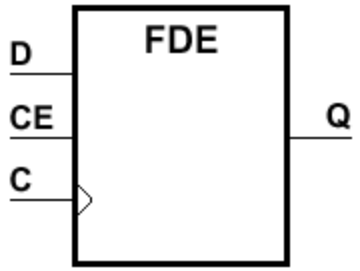
Flip Flop D edge-triggered positive, synchronous set

```
entity flop is
  port(C, D, S : in std_logic;
       Q : out std_logic);
end flop;
architecture archi of flop is
begin
  process (C)
  begin
    if (C'event and C='1') then
      if (S ='1') then
        Q <= '1';
      else
        Q <= D;
      end if;
    end if;
  end process;
end archi;
```



Flip-Flop D with “enable”

```
entity flop is
  port(C, D, CE : in std_logic;
       Q : out std_logic);
end flop;

architecture archi of flop is
begin
  process (C)
  begin
    if (C'event and C ='1') then
      if (CE ='1') then
        Q <= D;
      end if;
    end if;
  end process;
end archi;
```



4-bit Register, Enable and asynchronous Set

```
entity reg4 is
  port(C, CE, PRE : in std_logic;
       D : in std_logic_vector (3 downto 0);
       Q : out std_logic_vector (3 downto 0));
end reg4;
architecture archi of reg4 is
begin
  process (C, PRE)
  begin
    if (PRE ='1') then
      Q <= "1111";
    elsif (C'event and C ='1')then
      if (CE ='1') then
        Q <= D;
      end if;
    end if;   end process;    end archi;
```

Shift Register, serial input, serial output and synchronous reset

```
entity shift is
  port(
   clk,
   si,
   sync_reset:
       in std_logic;
   so: out std_logic);
end shift;
```

```
architecture archi of shift is
  signal tmp:
     std_logic_vector(7 downto 0);
begin
  process (clk, sync_reset)
  begin
    if (clk'event and clk='1') then
      if (sync_reset='1') then
        tmp <= (others => '0');
      else
        tmp <= tmp(6 downto 0) & si;
      end if;
    end if;
  end process;
  so <= tmp(7);
end archi;
```

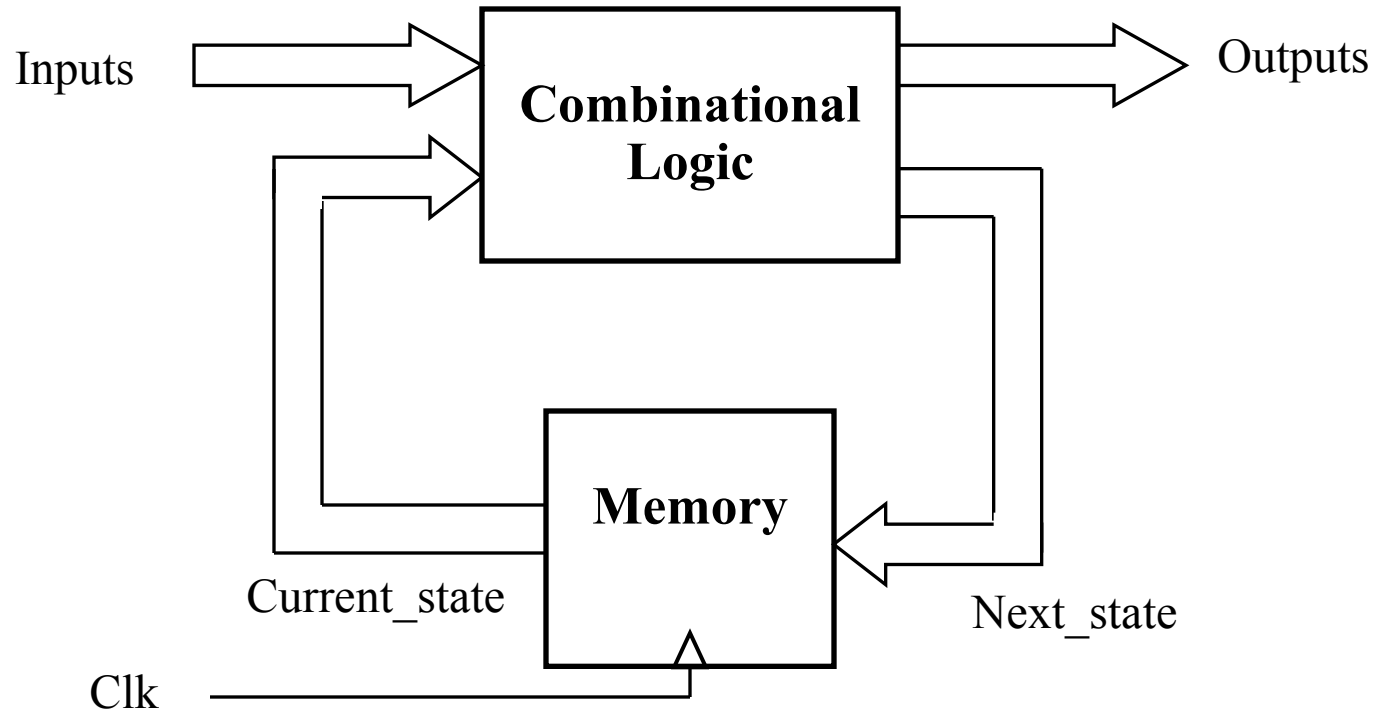
4-bit up counter with asynchronous clear

```
entity counter is
  port (clk, clr : in std_logic;
        q : out std_logic_vector (3 downto 0));
end counter;

architecture archi of counter is
  signal tmp: std_logic_vector (3 downto 0);
begin
  process (clk, clr)
  begin
    if (clr ='1') then
      tmp <= (others => '0');
    elsif (clk'event and clk = '1') then
      tmp <= tmp + 1;
    end if;
  end process;
  q <= tmp;
end archi;
```

4-bit updown counter

```
entity counter is
  port (clk, up_down : in std_logic;
        q : out std_logic_vector(3 downto 0));
end counter;

architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (up_down = '1') then
        tmp <= tmp + 1;
      else
        tmp <= tmp - 1;
      end if;
    end if;
  end process;
  q <= tmp;
end archi;
```

Finite State Machine Model



FSM Generic Model (2 processes)

```
entity fsm is
  port ( d_input : in din_type ;
         clk, rst: in std_logic;
         d_output: out dout_type ) ;
end fsm;
```

```
architecture proc2 of fsm is
  signal curr_state, next_state: states_type;
begin
  process (clk , rst )
    -- process 1 : synchronous state-register
  end process ;
  process (d_input, curr_state)
    -- process 2 : Combinational logic
  end process ;
end proc2 ;
```

FSM Generic Model (2 processes)

Definition of synchronous process

```
sync: process (clk, rst)
 begin
   if rst = '1' then
     curr_state <= state1 ;
   elsif clk'event and clk = '1' then
     curr_state <= next_state ;
   end if ;
 end process;
```

FSM Generic Model (2 processes)

Definition of combinational process

```
comb: process (d_input, curr_state);
begin
  next_state <= curr_state ;
  case curr_state is
    when state1 =>
      d_output <= "01" ;
      if d_input = '1' then
        next_state <= state2 ;
      end if ;
    when state2 =>    .   .   .
    when state3 =>    .   .   .
  end case ;
end process;
```

FSM Generic Model (3 processes)

```
architecture proc3 of fsm is
  signal state_actual, state_seguinte: tipo_state;
begin
 signal curr_state, next_state: states_type;
begin
  process (clk , rst )
    -- process 1 : synchronous state-register
  end process ;
  process (d_input, curr_state)
    -- process 2 : next_state logic
  end process ;
  process (d_input, curr_state)
    -- process 3 : output logic
  end process ;
end proc3;
```

Example: Sequence Detector “111”

```
entity fsm is
  port (serial_in, clk, rst: in std_logic;
        res: out std_logic);
end fsm;

architecture seqdet of fsm is
  type fsm_states is (STinitial, STone_1, STtwo_1s );
  signal curr_state, next_state: fsm_states;
begin
  -- process 1
  -- process 2
  -- process 3
end seqdet;
```

Sequence Detector “111” (process 1)

```
reg_state: process (clk, rst)
 begin
   if rst = '1' then
     curr_state <= STinitial ;
   elsif clk'event and clk = '1' then
     curr_state <= next_state ;
   end if ;
 end process;
```

Sequence Detector “111” (process 2a)

```
block_next_state: process (curr_state, serial_in)
begin
  next_state <= curr_state ;
  -- by default, the fsm stays in the same state.
  case curr_state is
    when STinitial =>
      if (serial_in = '1')
        then next_state <= STone_1; end if;
    when STone_1 =>
      if (serial_in = '1')
        then next_state <= STtwo_1s;
        else next_state <= STinitial; end if;
    when STtwo_1s =>
      if (serial_in = '0')
        then next_state <= STinitial; end if;
    end case;
  end process;
```

Sequence Detector “111” (process 2b)

```
block_next_state: process (curr_state, serial_in)
 begin
   next_state <= curr_state ;
   -- by default, the fsm stays in the same state.
   if (serial_in = '1') then
     case curr_state is
       when STinitial => next_state <= STone_1 ;
       when STone_1   => next_state <= STtwo_1s ;
       when STtwo_1s  => next_state <= STtwo_1s ;
     end case;
   else
     next_state <= STinitial ;
   end if;
 end process;
```

Sequence Detector “111” (process 3)

```
block_dout: process (curr_state, serial_in)
 begin
   res <= 'X' ;
   -- by default, output is don’t care
   case curr_state is
     when STinitial => res <= '0' ;
     when STone_1 => res <= '0' ;
     when STtwo_1s =>
       if serial_in = '1' then
         res <= '1' ;
       else
         res <= '0' ;
       end if;
   end case;
 end process;
```

8-bit Adder (concurrent model)

```
entity adder is
  port (x, y : in std_logic_vector(7 downto 0);
        sum : out std_logic_vector(8 downto 0));
end adder;
```

```
architecture equations of adder is
  signal carry : std_logic_vector(8 downto 0);
begin
  carry(0) <= '0';
  sum(8) <= carry(8);
  sum(7 downto 0) <= x xor y xor carry(7 downto 0);
  carry(8 downto 1) <= (x and y)
                       or (x and carry(7 downto 0))
                       or (y and carry(7 downto 0));
end equations;
```

8-bit Adder (concurrent model) with array attributes

```
entity adder is
  port (x, y : in std_logic_vector(7 downto 0);
        sum : out std_logic_vector(8 downto 0));
end adder;
```

```
architecture equations of adder is
  signal carry : std_logic_vector(sum'range);
begin
  carry(carry'low) <= '0';
  sum(sum'high) <= carry(carry'high) ;
  sum(x'range) <= x xor y xor carry(x'range) ;
  carry(carry'high downto 1) <= (x and y)
                              or (x and carry(x'range))
                              or (y and carry(x'range));
end equations ;
```

8-bit Adder (concurrent model) with for … generate

```
entity adder is
  port (x, y : in std_logic_vector(7 downto 0);
        sum : out std_logic_vector(8 downto 0));
end adder;
```

```
architecture equations of adder is
  signal carry : std_logic_vector(8 downto 0);
begin
  carry(0) <= '0';
  sum(8) <= carry(8) ;

  ADD8: for i in 0 to 7 generate
    sum(i) <= x(i) xor y(i) xor carry(i);
    carry(i+1) <= (x(i) and y(i))
                  or (x(i) and carry(i))
                  or (y(i) and carry(i));
  end generate;
end equations ;
```

8-bit Adder (structural model) with for … generate

```
architecture archi of adder is
  component fulladder
    port (x, y, cin : in std_logic;
          sum, cout : out std_logic);
  end component;
  signal carry : std_logic_vector(8 downto 0);
begin
  carry(0) <= '0';
  sum(8) <= carry(8) ;

  ADD8: for i in 0 to 7 generate
    u: fulladder
       port map (x => x(i), y => y(i), cin => carry(i),
                 sum => sum(i), cout => carry(i+1));
  end generate;
end archi ;
```

8-bit Adder (structural model) with parameterizable for … generate

```
entity adder is
  generic (N : integer := 8);  -- number of bits of adder
  port (x, y : in std_logic_vector(N-1 downto 0);
        sum : out std_logic_vector(N downto 0)) ;
end adder;
```

```
architecture archi of adder is
  component fulladder
    port(x, y, cin: in std_logic; sum, cout: out std_logic);
  end component;
  signal carry : std_logic_vector(sum'range);
begin
  carry(carry'low)<='0';
  sum(sum'high)<= carry(carry'high);
  add8 : for i in x'range generate
   u: fulladder port map (x => x(i), y => y(i), cin => carry(i),
                          sum => sum(i), cout => carry(i+1));
  end generate;
end archi;
```

Use of Generics for Component Parameterization

```
-- generics are first declared in the entity/component declaration
entity adder is
  generic (N : integer := 8);  -- number of bits of adder
  port (x, y : in std_logic_vector(N-1 downto 0);
        sum : out std_logic_vector(N downto 0));
end adder;
```

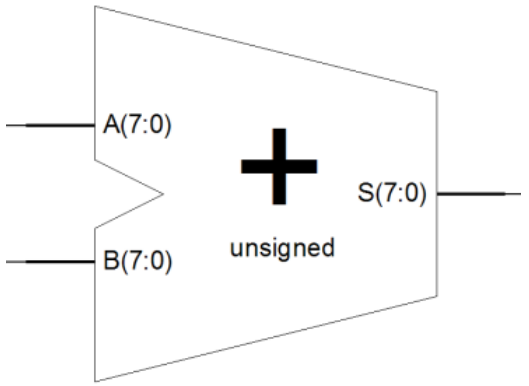
```
signal a, b: std_logic_vector(7 downto 0)) ;
signal c:    std_logic_vector(8 downto 0)) ;
signal d, e: std_logic_vector(31 downto 0)) ;
signal f:    std_logic_vector(32 downto 0)) ;

-- generics are assigned when the component is instantiated
add8:  adder
       generic map (N => 8)
       port map (x => a, y => b, sum => c);
add32: adder
       generic map (N => 32)
       port map (x => d, y => e, sum => f);
```

8-bit Adder (sequential model) with for … loop

```
entity adder is
  port (x, y : in std_logic_vector(7 downto 0);
        sum : out std_logic_vector(8 downto 0)) ;
end adder;

architecture archi of adder is
begin
  soma: process (x, y)
    variable carry : std_logic ;
    -- the use of variables is NOT recommended.
  begin
    carry := '0';
    for I in 0 to 7 loop
      sum(I) <= x(I) xor y(I) xor carry;
      carry  := (x(I) and y(I))
                or (x(I) and carry) or (y(I) and carry);
    end loop;
    sum(8) <= carry ;
  end process;
end archi ;
```

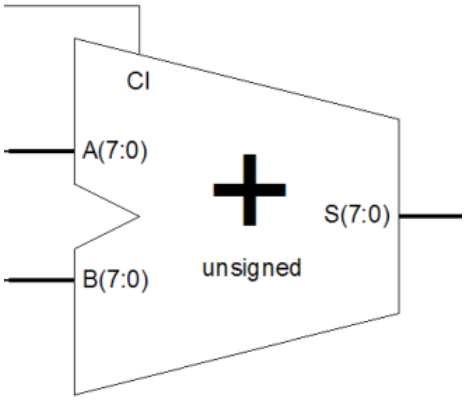
8-bit Adder (arithmetic model)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adder is
  port (a, b : in std_logic_vector(7 downto 0);
        sum : out std_logic_vector(7 downto 0)) ;
end adder;

architecture archi of adder is
begin
  sum <= a + b;
end archi ;
```



8-bit Adder with Carry-in

```
entity adder is
  port (a, b : in std_logic_vector(7 downto 0);
        ci : in std_logic;
        sum : out std_logic_vector(7 downto 0));
end adder;

architecture archi of adder is
begin
  sum <= a + b + ci;
end archi;
```



8-bit Adder with Carry-out

"std_logic_unsigned" does not allow to write "+" as
Res(9-bit) = A(8-bit) + B(8-bit)
to obtain Carry Out.

```
entity adder is
  port (a, b : in std_logic_vector(7 downto 0);
        co : out std_logic;
        sum : out std_logic_vector(7 downto 0));
end adder;
architecture archi of adder is
  signal res : std_logic_vector(8 downto 0);
begin
  res <= ('0' & a) + ('0' & b);
  sum <= res(7 downto 0);
  co <= res(8) ;
end archi;
```

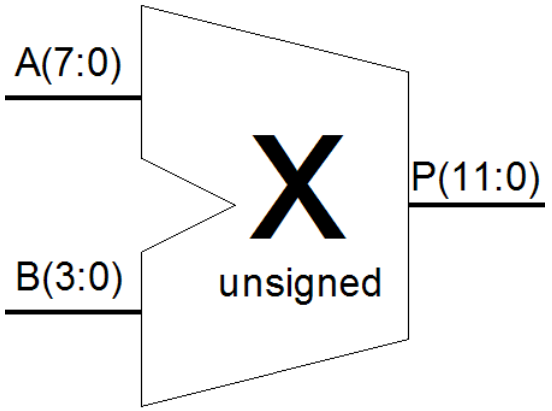
XST recognizes that the 9-bit adder can be implemented as an 8-bit adder with carry out.

8-bit Adder with Carry-in and Carry-out

```
entity adder is
  port (a, b : in std_logic_vector(7 downto 0);
        ci : in std_logic;
        co : out std_logic;
        sum : out std_logic_vector(7 downto 0));
end adder;

architecture archi of adder is
  signal res : std_logic_vector(8 downto 0);
begin
  res <= ('0' & a) + ('0' & b) + ci ;
  sum <= res(7 downto 0) ;
  co <= res(8) ;
end archi;
```

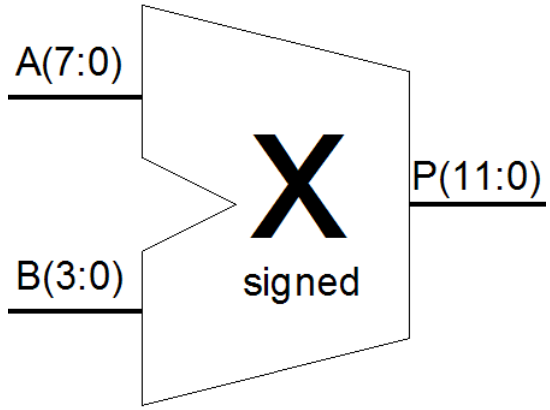
8x4-bit multiplier – unsigned arithmetic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_UNSIGNED.all;

entity mult is
  port (a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(3 downto 0);
        res : out std_logic_vector(11 downto 0));
end mult;

architecture archi of mult is
begin
  res <= a * b;
end archi;
```



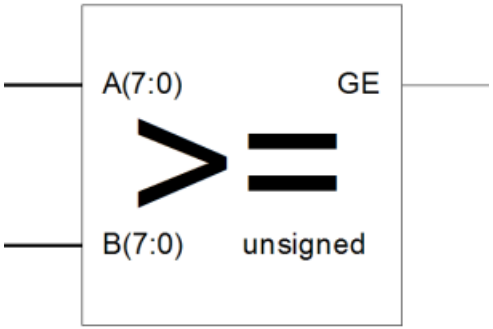
8x4-bit multiplier - signed arithmetic

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_SIGNED.all;

entity mult is
  port (a : in std_logic_vector(7 downto 0);
        b : in std_logic_vector(3 downto 0);
        res : out std_logic_vector(11 downto 0));
end mult;

architecture archi of mult is
begin
  res <= a * b;
end archi;
```



Comparator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity compar is
  port (a, b : in std_logic_vector(7 downto 0);
        cmp : out std_logic);
end compar;

architecture archi of compar is
begin
  cmp <= '1' when a >= b
         else '0';
end archi;
```



Simple ALU (entity)

Executes 4 operations:
1. F = A shifted_left 1
2. F = not(A)
3. F = A + B
4. F = A and B

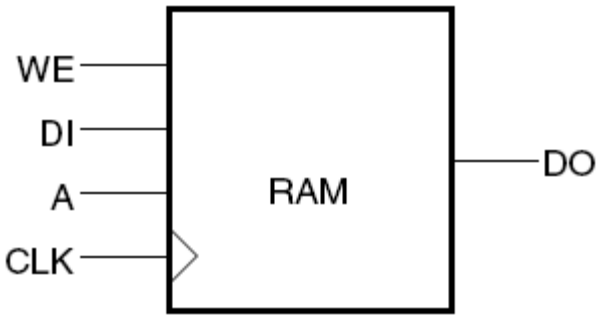
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity alu is
  port ( a, b : in std_logic_vector(7 downto 0) ;
         cin : in std_logic;
         sel : in std_logic_vector(1 downto 0);
         f : out std_logic_vector(7 downto 0);
         cout : out std_logic) ;
end alu;
```

Simple ALU (architecture)

```
architecture a of alu is
  signal sum : std_logic_vector(8 downto 0);
begin
  with sel select
    f <=  a(6 downto 0) & '0'  when "00",
          not(a)               when "01",
          sum(7 downto 0)      when "10",
          a and b              when others ;

  cout <= sum(8) when sel = "10"
          else a(7) when sel = "00"
          else '0' ;

  sum <= ('0' & a) + ('0' & b) + cin ;
end a ;
```

32×4-bit single-port RAM - Async. read (1)

```
entity raminfr is
  port (clk : in std_logic;
        we : in std_logic;
        address : in std_logic_vector (4 downto 0);
        di : in std_logic_vector (3 downto 0);
        do : out std_logic_vector (3 downto 0));
end raminfr;
```



32×4-bit single-port RAM - Async. read (2)

```
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of
                   std_logic_vector (3 downto 0);
  signal RAM : ram_type;

begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM (conv_integer(address)) <= di;
      end if;
    end if;
  end process;
  do <= RAM (conv_integer(address));
end syn;
```

32×4-bit ROM

```
entity rom_asyn is
  port (addr : in std_logic_vector(4 downto 0);
        data : out std_logic_vector(3 downto 0));
end rom_asyn;
architecture asyn of rom_asyn is
  type rom_type is array (31 downto 0)
                   of std_logic_vector (3 downto 0);
  constant ROM : rom_type
    :=("0001","0010","0011","0100","0101",
       "0110","0111","1000","1001","1010",
       "1011","1100","1101","1110","1111",
       "0001","0010","0011","0100","0101",
       "0110","0111","1000","1001","1010",
       "1011","1100","1101","1110","1111",
       "1111","1111");
begin
  data <= ROM(conv_integer(addr));
end asyn;
```

Propagation delays in VHDL simulation

```
Y <= X;                 -- delta delay
Y <= X after 10 ns;     -- standard time unit delay
```

The simulation time is expressed in *standard* time units
(the addition of delta delays does not cause the simulation time to advance).

Simulation steps:

1. Advance the simulation time until the first event in the queue.
2. Activate all processes scheduled for this simulation time.
3. Execute the activated processes (no order of execution is implied) and (eventually) schedule new events (some of the new events may correspond to delta delays).
4. If there are new transactions on signals at the current simulation time (due to signal assignments with delta delays), go to step 2, otherwise go to step 1.

Inertial and Transport Delays

```
Y <= X after 10 ns ;                -- inertial delay
Y <= transport X after 10 ns ; -- transport delay
```

Inertial delay: the signal is propagated **only if** the input maintains the same value during the time specified.

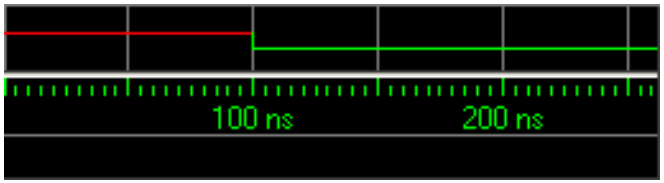
Transport delay: all changes of the input signal are propagated.

```
Z <= '1' after 50 ns;
Z <= '0' after 100 ns;
```

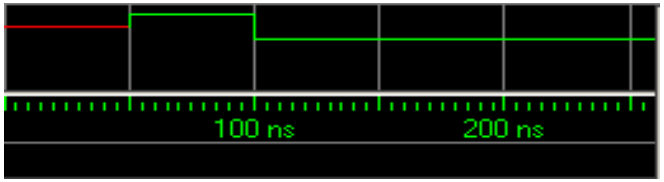
What happens?

```
Z <= '1' after 50 ns,
     '0' after 100 ns ;
```

The first delay is inertial and all the following are transport delays.



Not supported by ISE simulator.



Recommended.

Simple VHDL simulation example

```
entity x is
 Port (a : in std_logic;
       b : in std_logic;
       c : out std_logic);
end x;

architecture Behavioral of x is
  begin

    c <= a xor b;

end Behavioral;
```

XST: testbench (1)

```
-- VHDL Test Bench Created by ISE for module: x
--
-- This testbench has been automatically generated using
-- types std_logic and std_logic_vector for the ports of
-- the unit under test.  Xilinx recommends that these types
-- always be used for the top-level I/O of a design in
-- order to guarantee that the testbench will bind
-- correctly to the post-implementation simulation model.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY t_vhd IS
END t_vhd;
```

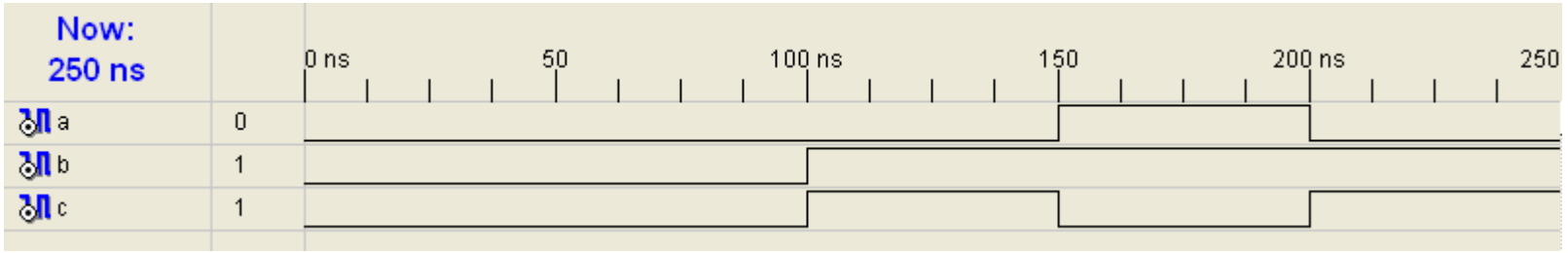
XST: testbench (2)

```
ARCHITECTURE behavior
OF t_vhd IS

-- Component Declaration
-- for the Unit Under
-- Test (UUT)
COMPONENT x
 PORT (a : IN std_logic;
       b : IN std_logic;
       c : OUT std_logic);
END COMPONENT;

--Inputs
SIGNAL a: std_logic := '0';
SIGNAL b: std_logic := '0';

--Outputs
SIGNAL c :  std_logic;
```

```
BEGIN
-- Instantiate the Unit Under
-- Test (UUT)
 uut: x PORT MAP(a => a,
                 b => b, c => c);
 tb : PROCESS
 BEGIN
 -- Wait 100 ns for global
 -- reset to finish
 wait for 100 ns;
 -- Place stimulus here
 a <= '1' after 50 ns,
      '0' after 100 ns;
 b <= '1';
 wait; -- will wait forever
 END PROCESS;
END;
```

Simulation Results

Behavior Simulation: gate output switches instantaneously



Post-Route Simulation: output switches after gate delay