VHDL - Combinatorial PROCESS Logic Synthesis

©Hanan Ribo

Introduction

- The next step after HDL code Simulation is HDL code Synthesis.
- Synthesis contains the next three steps:
  - ✓ conversion of the high-level VHDL (or Verilog) language, which describes the circuit at the Register Transfer Level (RTL), into a netlist at the gate level.
  - ✓ Optimization of the gate-level netlist for speed (minimize critical path) and for area (minimize Logic function).
  - ✓ Implementation of the optimized gate-level netlist based on **MUXs+LUTs, Latches, FFs** (in case of FPGA as a target Hardware).
- The last step is a place and route (fitter), software will generate the physical layout for a FPGA chip or will generate the masks for an ASIC chip.

Synthesis coding approach

- Synthesis tools and Simulation tools translate PROCESS based HDL code in a different way (concurrent code translated in the same way).
- Synthesis tools search for adjustment of VHDL code to one of the next three template kinds (*ieee-1076.6 standard*):

  Combinational Logic, Synchronous Logic, Latch based Logic.
- **Our goal:**
  - ✓ writing of HDL code which will be translated in the same exact way by all Synthesis and Simulation tools.
  - ✓ Avoid of HDL code which synthesized with hardware errors in the required design (must avoid from Sick Hardware).
  - ✓ **Important rule:** when you write HDL code, think Hardware!

Note: Unusual and Unsupported design approach

With a guarded BLOCK or with WHEN statements (using concurrent code) even very simple sequential circuits can be constructed. This, however, is Unusual and Unsupported design approach.

In conclusion: Synchronous design will be described using PROCESS only!

DFF implementation example using concurrent code

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY dff IS
  PORT ( d, clk, rst: IN STD_LOGIC;
         q: OUT STD_LOGIC);
END dff;
-------------------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
  b1: BLOCK (clk'EVENT AND clk='1')
  BEGIN
     q <= GUARDED '0' WHEN rst='1' ELSE d;
  END BLOCK b1;
END dff;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
------------------------------
ENTITY dff IS
  PORT ( d, clk : IN STD_LOGIC;
             q : BUFFER STD_LOGIC);
END dff;
------------------------------
ARCHITECTURE dff OF dff IS
BEGIN
    q <= d WHEN clk'event and clk='1' ELSE q;
END dff;
```

PROCESS Logic Synthesis

The way we write a PROCESS affects its synthesis and is associated with one of the following two synthesis types:

- **Combinatorial PROCESS (Combinational Logic Circuit):**
  - ✓ PROCESS that its sensitivity list contains all its internal input SIGNALS and the PROCESS doesn't contain IF-THEN statement which its condition on SIGNAL event.
  - ✓ This kind of PROCESS describes a combinational logic circuit.
  - ✓ If we write a partial sensitivity list, the compiler completes it, differ from simulation environment.
- **Sequential PROCESS (Synchronous / Asynchronous Logic Circuit):**
  - ✓ PROCESS that its sensitivity list contains a input SIGNAL and the PROCESS contains IF-THEN statement which its condition on SIGNAL event.
  - ✓ This kind of PROCESS describes FFs based sequential logic circuit triggered by a SIGNAL event.

Combinatorial PROCESS - Rules

In order the compiler will synthesize the a PROCESS as a Combinational Logic Circuit we must obey the next three rules (Combinational Logic template):

Rule 1: Make sure that all the input SIGNALS of the required combinational circuit appear in the PROCESS sensitivity list.

Rule 2: Don’t use IF-THEN statement which its condition on SIGNAL event (SIGNAL transition).

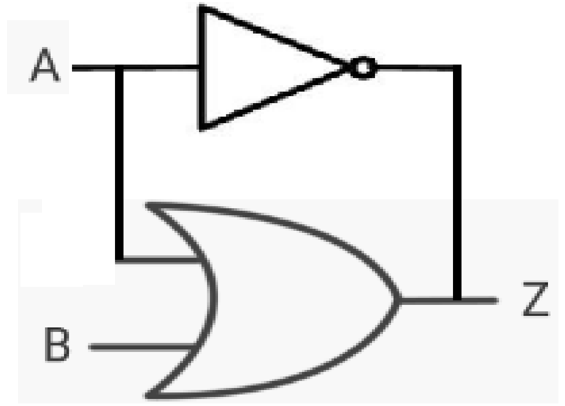
Rule 3: PROCESS must cover all the permutations (full truth table description) of the input and output SIGNALS of the combinational circuit.

- ✓ Use ELSE option in IF-THEN statement.
- ✓ For a complex Combinational Circuit description, use default SIGNALS assignments (at the PROCESS beginning we write all the default output SIGNALS assignment and later we use IF-THEN without covering all permutations).

General rules for Combinational Logic Circuit

- **General Rule 1**: Use SIGNALS assignment without feedback (a SIGNAL must appear only in one assignment side).

- **General Rule 2**: Avoid from multiple assignment to the same SIGNAL from different PROCESSES (cause multiple driven).



Combinatorial PROCESS – MUX 3-1 example

- The PROCESS must cover all the permutations (full truth table description) of the input and output SIGNALS of the combinational circuit.

```
entity mux is port(
    I0, I1, I2 : in std_logic_vector(2 downto 0);
             S : in std_logic_vector(1 downto 0);
             O : out std_logic_vector(2 downto 0));
end mux;
```

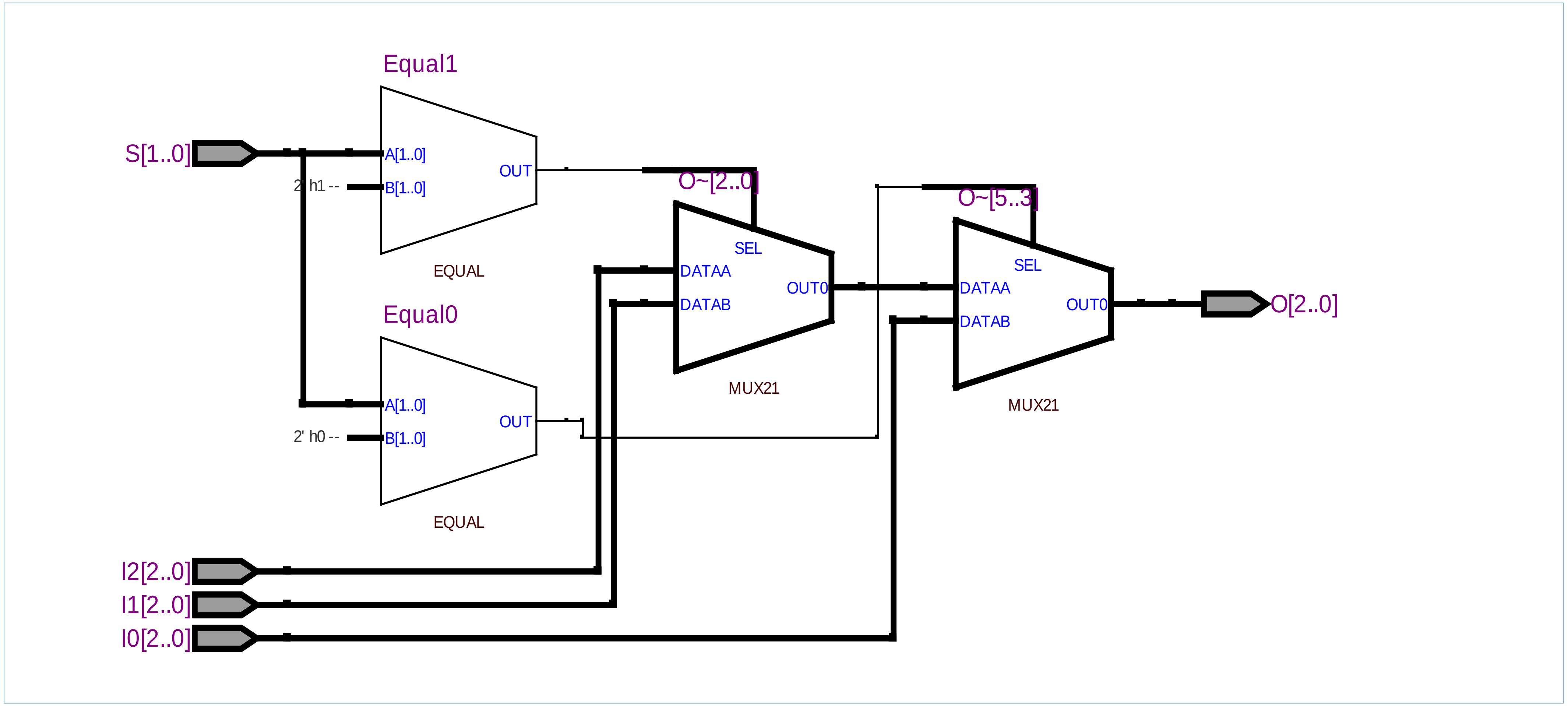
- **Attitude 1**: use ELSE option in IF-THEN statement.
- **Attitude 2**: use default SIGNALS assignments

```
architecture rtl of mux is
begin
    process(I2,I1,I0,S)
    begin
        if      S = "00" then  O <= I0;
        elsif   S = "01" then  O <= I1;
        else                   O <= I2;
        end if;
    end process;
end rtl;
```

```
architecture rtl of mux is
begin
    PROCESS (I2, I1, I0, S)
    BEGIN
        O <= I2 ;
        IF    S="00" THEN O <= I0 ;
        ELSIF S="01" THEN O <= I1 ;
        END IF;
    END PROCESS ;
end architecture rtl;
```

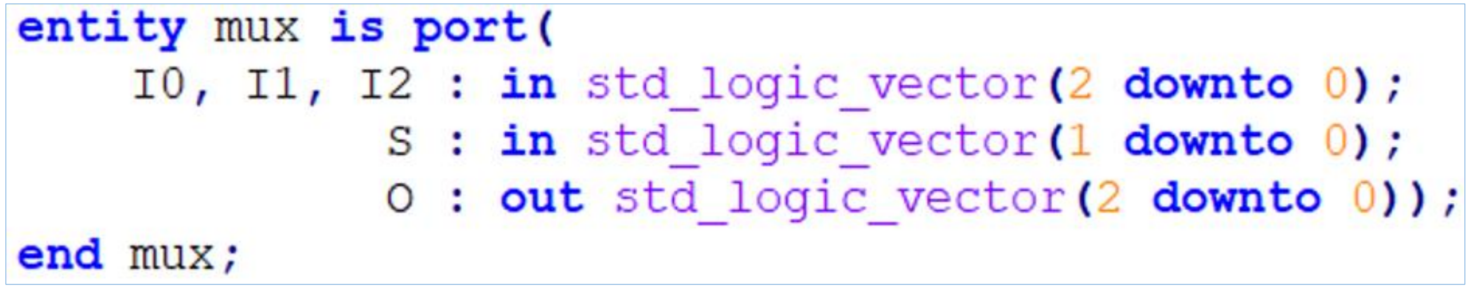
Combinatorial PROCESS – MUX 3-1 example

Synthesis result:

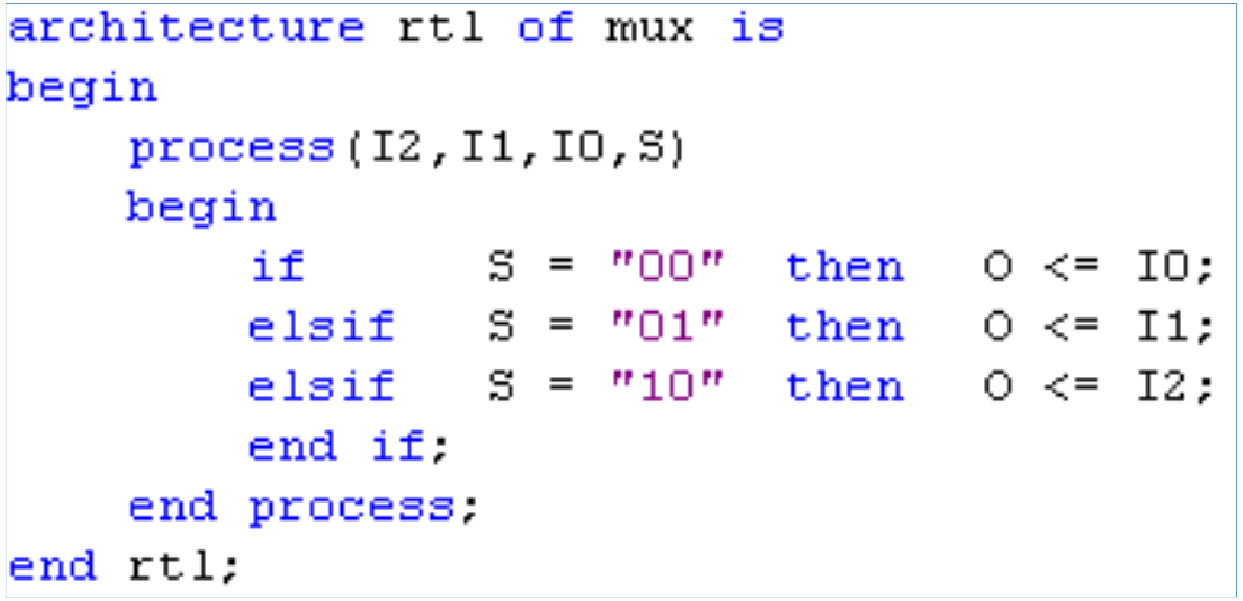


Wrong Combinatorial PROCESS (Sick HW)

- The next PROCESS doesn't cover all the input S permutations of the combinational circuit (permutation S="11" is missing, in order to hold the output O last value, tree Latches will be inferred) .



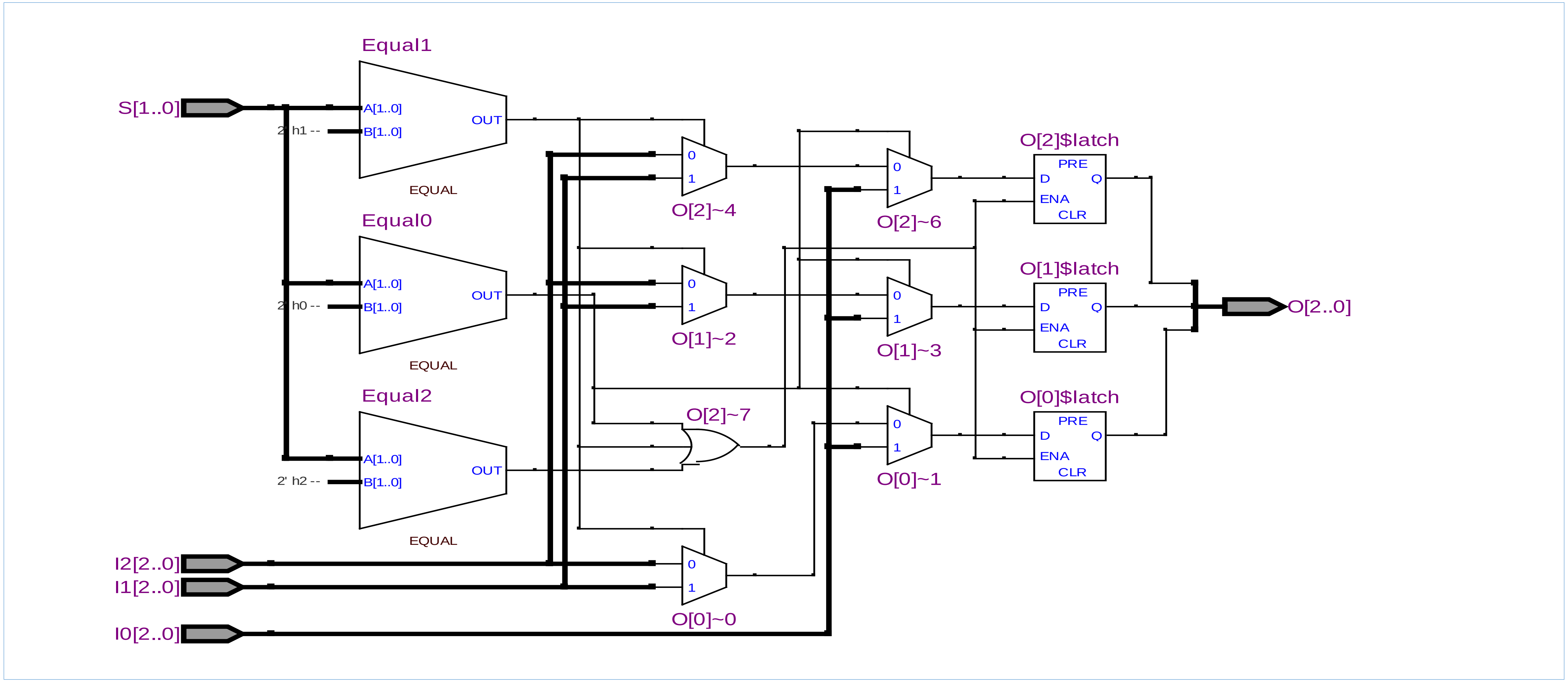
```
entity mux is port(
    I0, I1, I2 : in std_logic_vector(2 downto 0);
             S : in std_logic_vector(1 downto 0);
             O : out std_logic_vector(2 downto 0));
end mux;
```



```
architecture rtl of mux is
begin
    process(I2,I1,I0,S)
    begin
        if      S = "00"  then   O <= I0;
        elsif   S = "01"  then   O <= I1;
        elsif   S = "10"  then   O <= I2;
        end if;
    end process;
end rtl;
```

Wrong Combinatorial PROCESS (Sick HW)

Synthesis result (we meant for MUX 3-1, the result is a Sick HW):



Simulation vs Synthesis

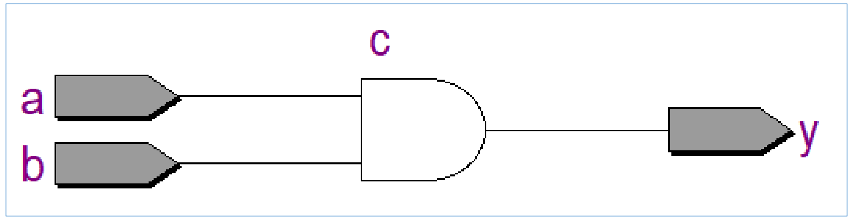
```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
-----------------------------------
entity case3 is port(
    a,b : in std_logic;
    y   : out std_logic);
end case3;
-----------------------------------
architecture arc of case3 is
  signal c : std_logic;
begin
  process (a, b)
    begin
     c <= a and b ;
     y <= c ;
  end process ;
end architecture arc;
```

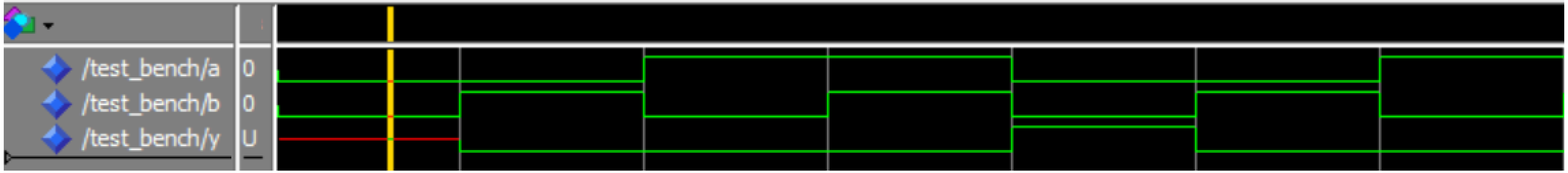
case3 code simulation doesn't describe **AND** gate

Don't use **SIGNALS** for Intermediate calculations

Combinatorial PROCESS: The PROCESS covers all the permutations (full truth table description) of the input and output SIGNALS of the combinational circuit.

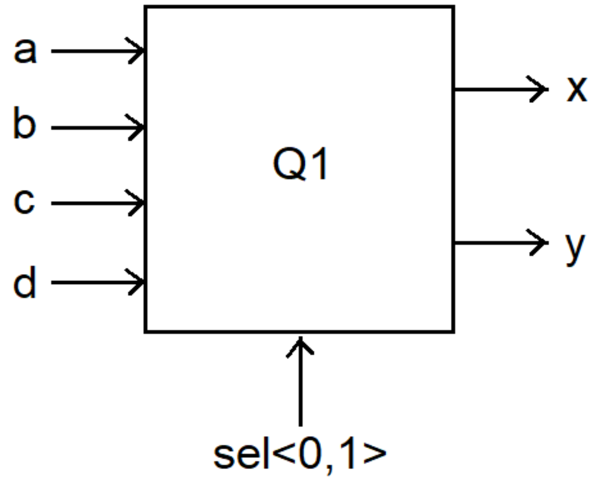
Synthesis result:





Wrong Combinatorial PROCESS (Sick HW)

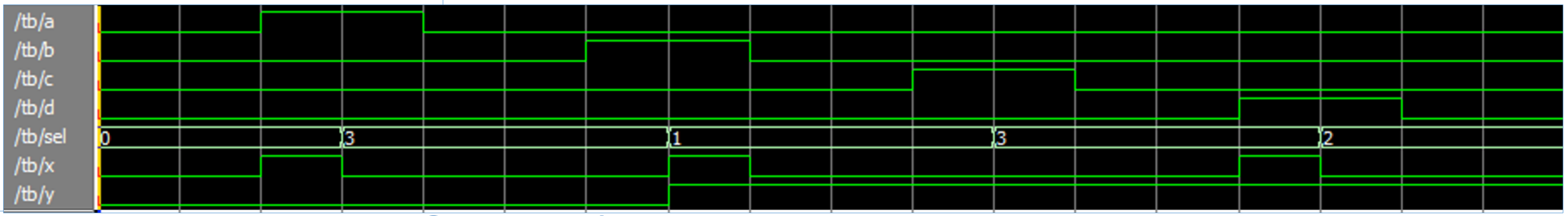
```
entity selector is
   port(
      a,b,c,d : in std_logic;
          sel : in integer range 0 to 3;
          x,y : out std_logic
    );
end selector;
------------------------------------------
architecture arc of selector is
begin
    process(a,b,c,d,sel)
    begin
        if (sel=0) then
            x<=a;
            y<='0';
        elsif (sel=1) then
            x<=b;
            y<='1';
        elsif (sel=2) then
            x<=c;
        else
            x<=d;
        end if;
    end process;
end arc;
```



sel	x	y
00	a	0
01	b	1
10	c	
11	d	

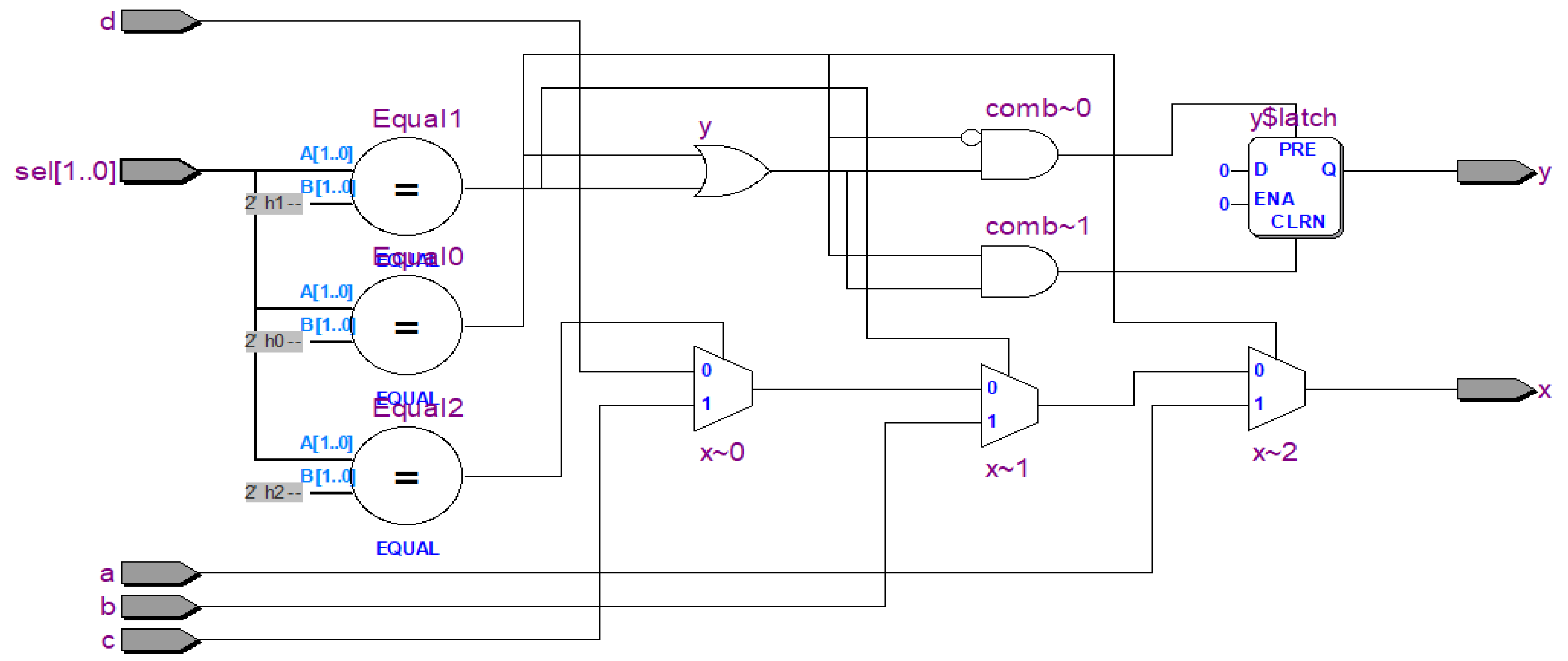
sel	x	y
00	a	0
01	b	1
10	c	y
11	d	y

the specifications provided for y are incomplete, as can be observed in the truth-table. Therefore, a latch will be implemented, which renders the previous y value.



Wrong Combinatorial PROCESS (Sick HW)

Synthesis result (we meant for selector, the result is a Sick HW):



$$y = 0 \; \boldsymbol{when} \; \boldsymbol{CLRN} = 1: \quad CLRN = ((sel == 1) or (sel == 0)) and (sel == 0)$$

$$y = 1 \; \boldsymbol{when} \; \boldsymbol{PRESET} = 1: \quad PRESET = ((sel == 1) or (sel == 0)) and \; not((sel == 0))$$

$$y = (sel(0) \; AND \; sel(1)) \; OR \; (sel(0) \; AND \; y) \; OR \; (sel(1) \; AND \; y)$$