



**Universidade Federal de Santa Catarina
Centro Tecnológico – CTC
Departamento de Engenharia Elétrica**



“EEL7020 – Sistemas Digitais”

Prof. Eduardo Augusto Bezerra

Eduardo.Bezerra@eel.ufsc.br

Florianópolis, agosto de 2012.

Sistemas Digitais

Processos implícitos e explícitos

Processos implícitos e explícitos

Processos implícitos:

- Atribuições com sinais em paralelo (exemplo: `LEDR <= SW`);
- Atribuições com seleção (*with / select*) e comparação (*when*);
- Componentes instanciados (*component / port map*);

Processos explícitos:

- Definidos com a palavra reservada ***process***.

Processos implícitos e explícitos

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY implicito IS
PORT (
    a, b : IN STD_LOGIC;
    y    : OUT STD_LOGIC
);
END implicito;
ARCHITECTURE logic OF implicito IS
    SIGNAL c : STD_LOGIC;
BEGIN
    c <= a AND b;
    y <= c;
END logic;
```

- **Soluções equivalentes;**
- **Exemplo adaptado de material educacional da Altera.**

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY explicito IS
PORT (
    a,b : IN STD_LOGIC;
    y    : OUT STD_LOGIC
);
END explicito;
ARCHITECTURE logic OF explicito IS
    SIGNAL c : STD_LOGIC;
BEGIN
    PROCESS (a, b)
    BEGIN
        c <= a AND b;
    END PROCESS;
    PROCESS (c)
    BEGIN
        y <= c;
    END PROCESS;
END logic;
```

Processos implícitos e explícitos

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY implicito IS
PORT (
    a, b : IN STD_LOGIC;
    y    : OUT STD_LOGIC
);
END implicito;
ARCHITECTURE logic OF implicito IS
    SIGNAL c : STD_LOGIC;
BEGIN
    c <= a AND b;
    y <= c;
END logic;
```

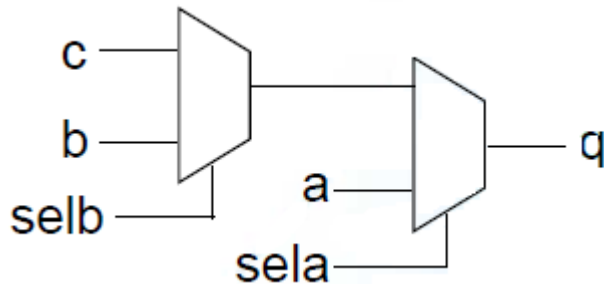
- **Soluções NÃO SÃO equivalentes;**
- **Exemplo adaptado de material educacional da Altera.**

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY explicito IS
PORT (
    a,b : IN STD_LOGIC;
    y : OUT STD_LOGIC
);
END explicito;
ARCHITECTURE logic OF explicito IS
    SIGNAL c : STD_LOGIC;
BEGIN
    PROCESS (a, b)
    BEGIN
        c <= a AND b;
        y <= c;
    END PROCESS;
END logic;
```

Processos **implícitos** – revisão de atribuição condicional

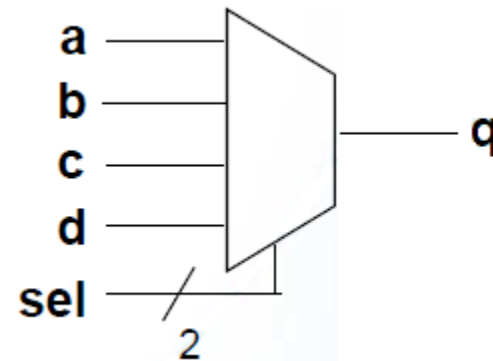
architecture x of y is
begin

```
q <= a when sela = '1'  
else  
    b when selb = '1'  
else  
    c;  
end x;
```



architecture x of y is
begin

```
with sel select  
q <= a when "00",  
    b when "01",  
    c when "10",  
    d when others;  
end x;
```

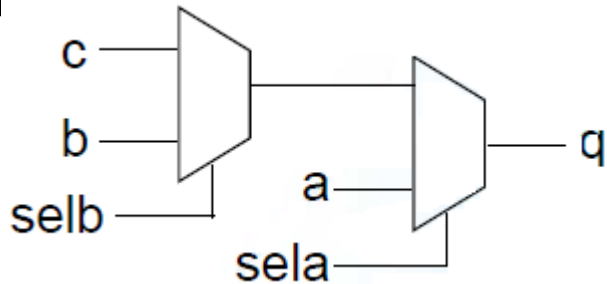


Processos **explícitos** – comparação e seleção

architecture **implicito** of y is
begin

```
q <= a when sela = '1'  
else  
    b when selb = '1'  
else  
    c;
```

end implicito;



architecture **explicito** of y is
begin

```
process (sela, selb, a, b, c)  
begin
```

```
    if sela = '1' then  
        q <= a;  
    elsif selb = '1' then  
        q <= b;  
    else  
        q <= c;  
    end if;  
end process;
```

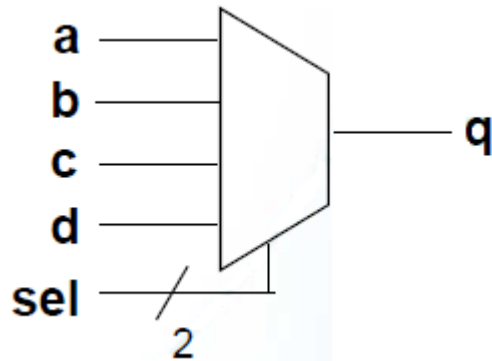
```
end explicito;
```

Processos **explícitos** – comparação e seleção

architecture **implicito** of y is
begin

```
with sel select  
q <= a when "00",  
  b when "01",  
  c when "10",  
  d when others;
```

end implicito;



architecture **explicito** of y is
begin

```
process (sel, a, b, c, d)  
begin
```

```
  case sel is  
    when "00" =>  
      q <= a;
```

```
    when "01" =>  
      q <= b;
```

```
    when "10" =>  
      q <= c;
```

```
    when others =>  
      q <= d;
```

```
  end case;
```

```
end process;
```

```
end explicito;
```


Estudo de caso: uso de processo explícito para implementar registrador com reset assíncrono, clock e sinal de enable

Deslocamento de vetor de entrada 1 bit à esquerda (1/2)

- sr_in recebe palavra de N bits (vetor de N bits). A cada
- pulso de clock, a palavra em sr_in é deslocada 1 bit para a
- esquerda, e copiada para sr_out, também de N bits.

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
entity desloc_1_bit_esq is
```

```
    generic ( N : natural := 64 );
```

```
    port ( clk      : in std_logic;
```

```
          enable   : in std_logic;
```

```
          reset    : in std_logic;
```

```
          sr_in    : in std_logic_vector((N - 1) downto 0);
```

```
          sr_out   : out std_logic_vector((N - 1) downto 0)
```

```
    );
```

```
end entity;
```

Deslocamento de vetor de entrada 1 bit à esquerda (2/2)

```
architecture rtl of desloc_1_bit_esq is
```

```
    signal sr: std_logic_vector ((N - 1) downto 0); -- Registrador de N bits
```

```
begin
```

```
    process (clk, reset)
```

```
    begin
```

```
        if (reset = '0') then -- Reset assíncrono do registrador
```

```
            sr <= (others => '0');
```

```
        elsif (rising_edge(clk)) then -- Sinal de clock do registrador (subida)
```

```
            if (enable = '1') then -- Sinal de enable do registrador
```

```
                -- Desloca 1 bit para a esquerda. Bit mais significativo é perdido.
```

```
                sr((N - 1) downto 1) <= sr_in((N - 2) downto 0);
```

```
                sr(0) <= '0';
```

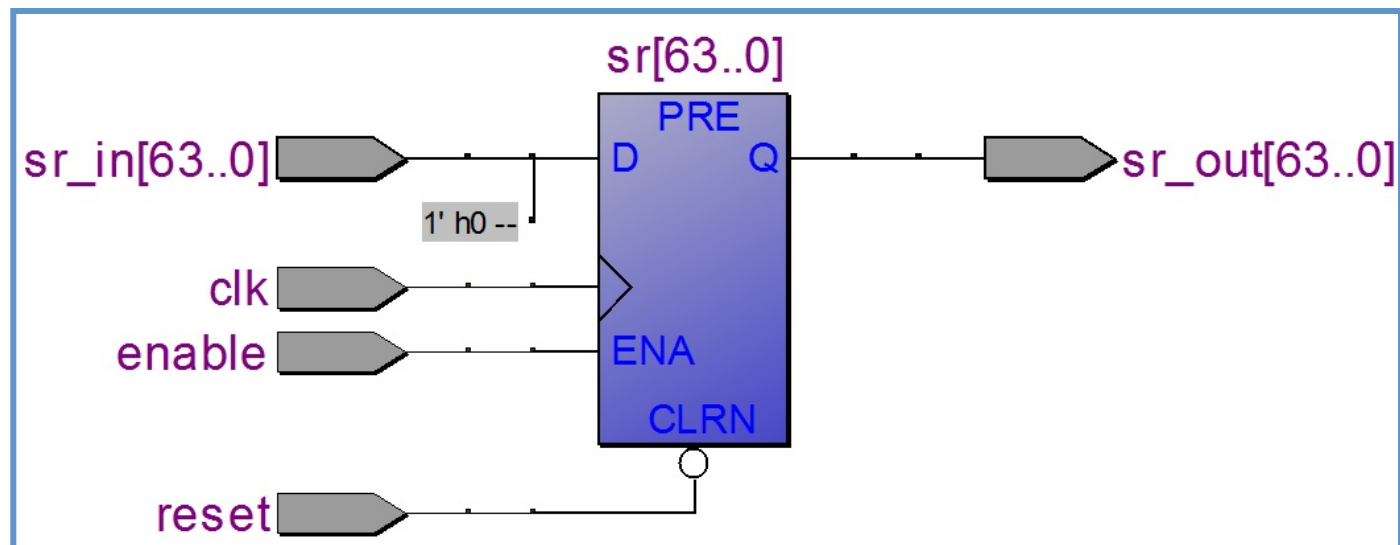
```
            end if;
```

```
        end if;
```

```
    end process;
```

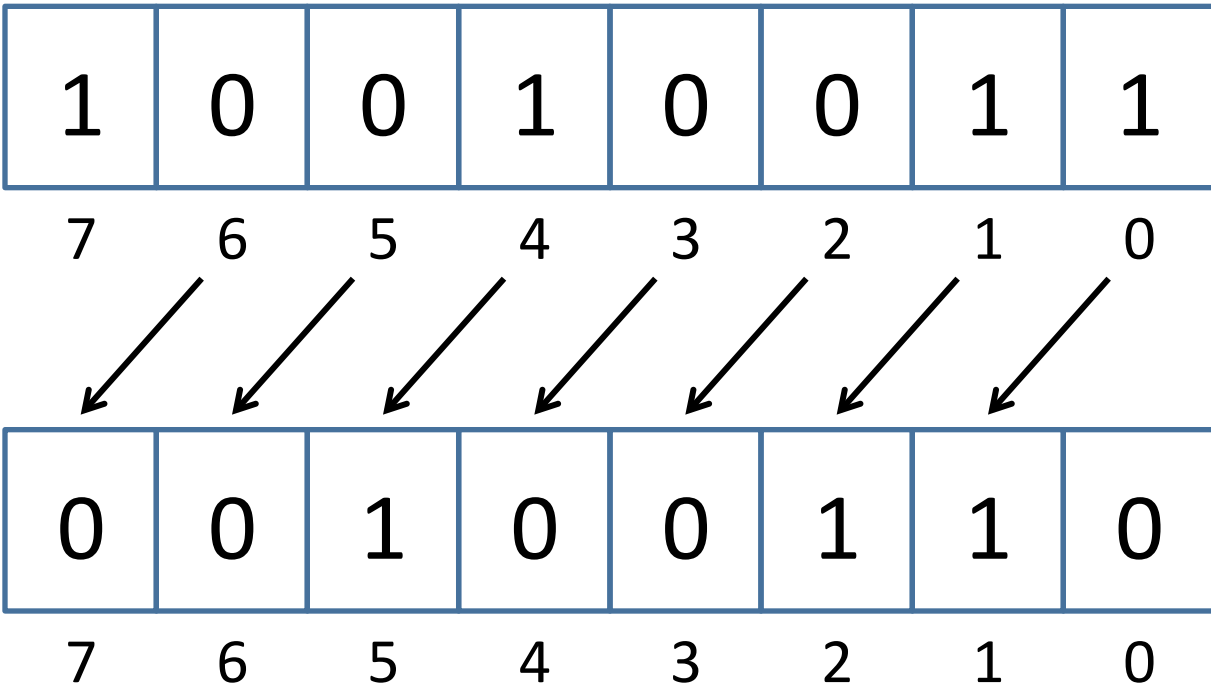
```
    sr_out <= sr;
```

```
end rtl;
```



Deslocamento de vetor de entrada 1 bit à esquerda

Valor de entrada em *sr_in* = 93H



Valor de saída em *sr_out* = 26H

~~1~~ ←

← 0

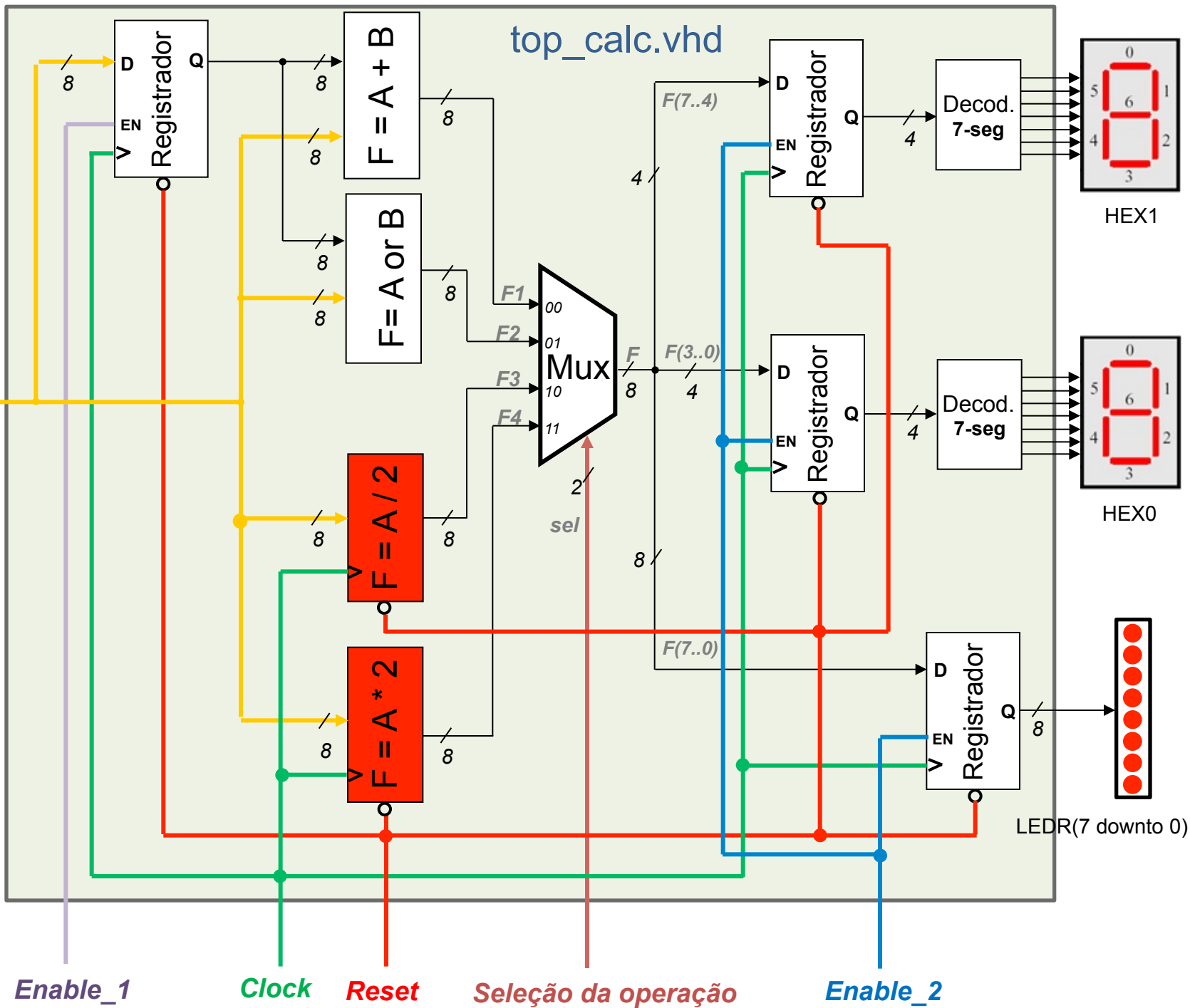
Tarefa:
mini-calculadora com multiplicação e divisão por 2

Tarefa – Descrição Textual

- Alterar a mini-calculadora desenvolvida nas aulas anteriores, de forma a realizar a “multiplicação por 2” e a “divisão por 2”.
- Utilizando um **registrador de deslocamento**, projetar um circuito para realizar operações de **divisão por 2** (shift right).
- Utilizando um **registrador de deslocamento**, projetar um circuito para realizar operações de **multiplicação por 2** (shift left).
- Substituir o componente que realiza a função **$F = \text{not } A$** pelo novo componente que realiza a multiplicação por 2.
- Substituir o componente que realiza a função **$F = A \text{ xor } B$** pelo novo componente que realiza a divisão por 2.

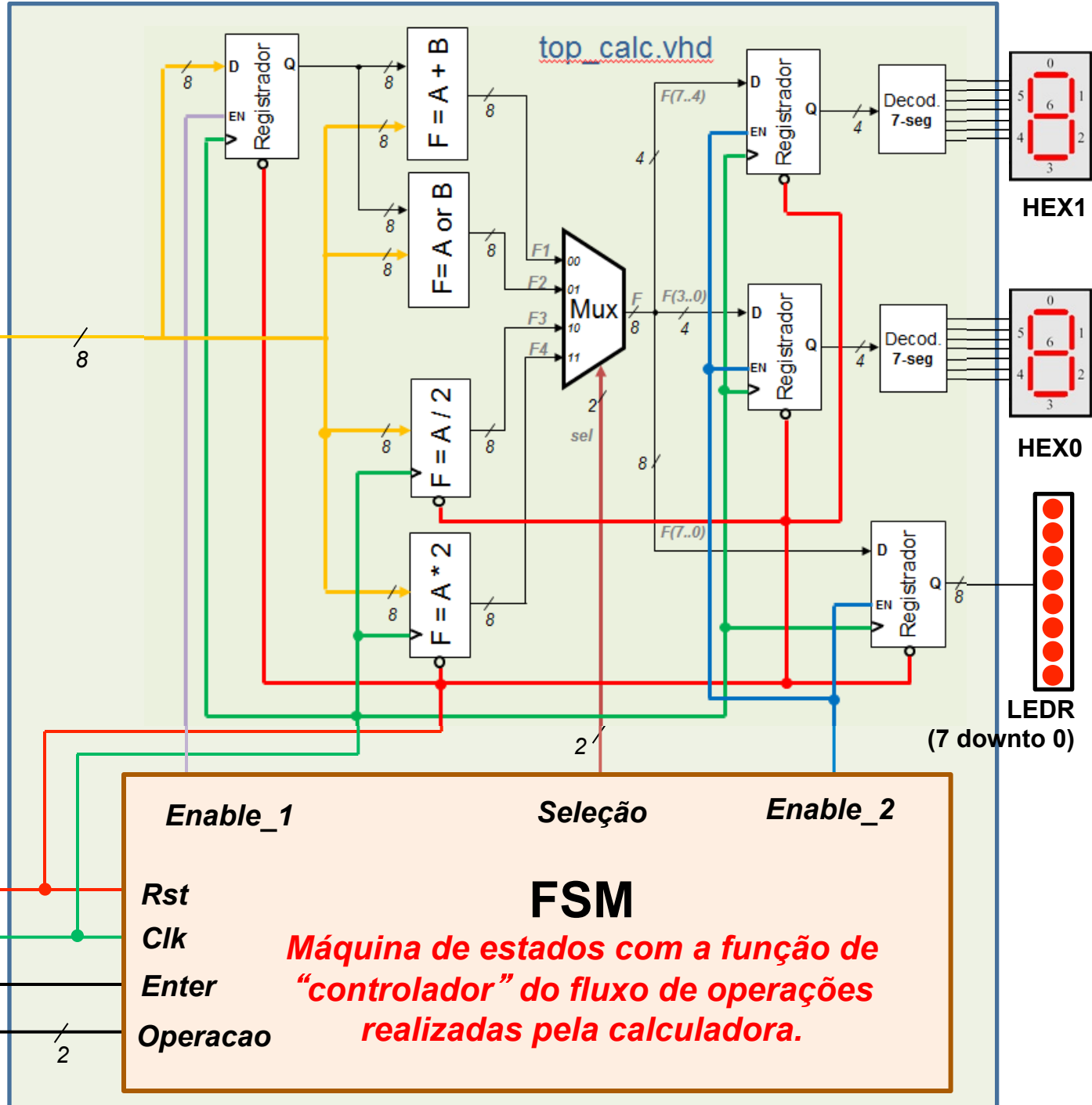
Atenção!! Nesse novo circuito existem dois componentes que utilizam apenas um operando, e dois componentes que utilizam dois operandos. **É preciso alterar a FSM para se adequar a essa situação!!**

Operandos
SW(7 downto 0)



TOPO

Operandos
SW(7 downto 0)



Reset KEY(0)

Clock CLOCK_50

Enter KEY(1)

Operação SW(17..16)

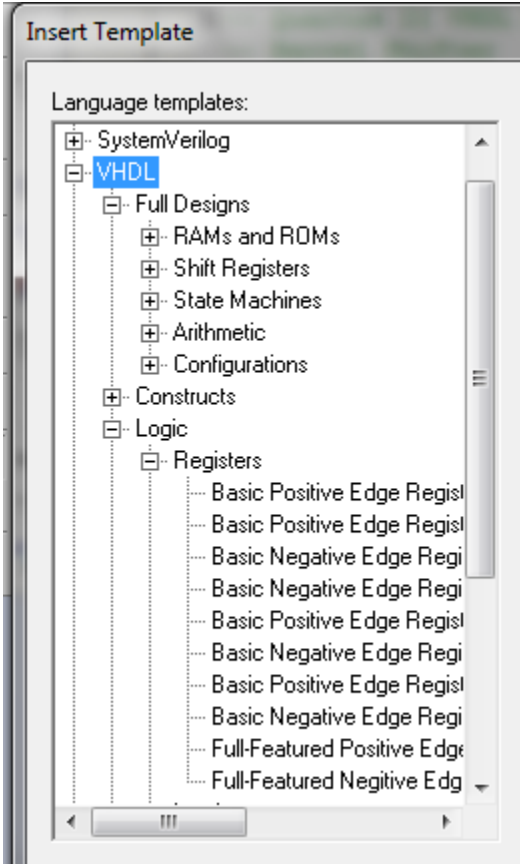
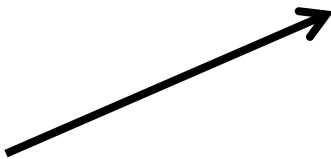
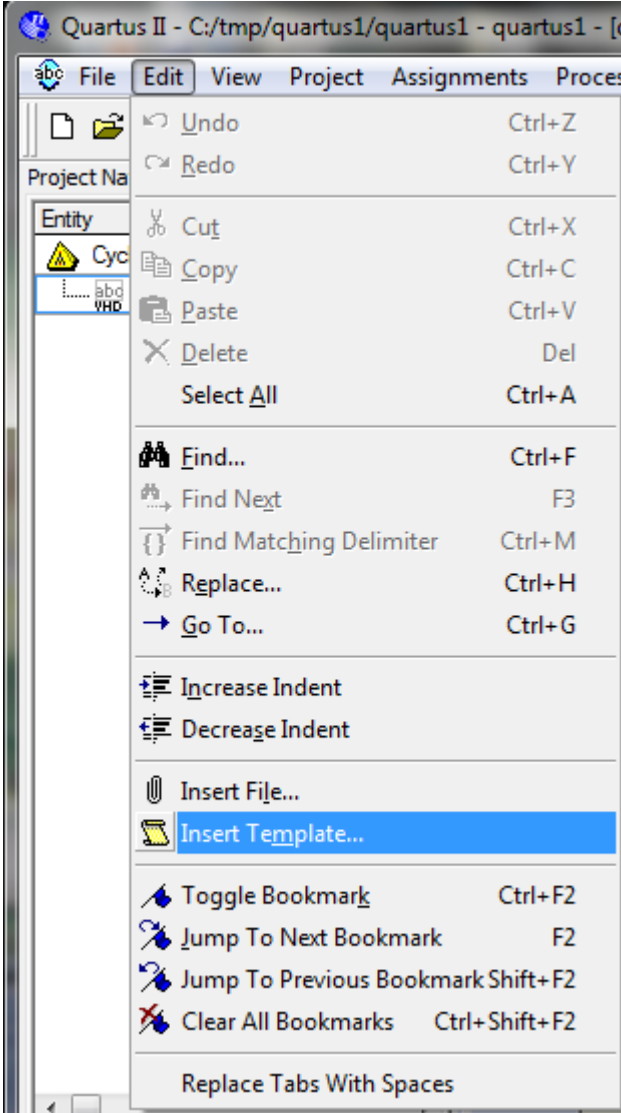
Enable_1 *Seleção* *Enable_2*

FSM

Máquina de estados com a função de "controlador" do fluxo de operações realizadas pela calculadora.

***Outros templates com
processos explícitos do Quartus II***

Quartus II – Templates de descrições VHDL com processos



Basic Positive Edge Register with Asynchronous Reset and Clock Enable

```
process (<clock_signal>, <reset>)
begin
    -- Reset whenever the reset signal goes low, regardless
    -- of the clock or the clock enable
    if (<reset> = '0') then
        <register_variable> <= '0';
    -- If not resetting, and the clock signal is enabled,
    -- update the register output on the clock's rising edge
    elsif (rising_edge(<clock_signal>)) then
        if (<clock_enable> = '1') then
            <register_variable> <= <data>;
        end if;
    end if;
end process;
```

Basic Positive Edge Register with Asynchronous Reset

```
process (CLK, RST)
begin
```

```
-- Reset whenever the reset signal goes low, regardless
-- of the clock
```

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

```
    Q <= '0';
```

```
-- If not resetting update the register output
```

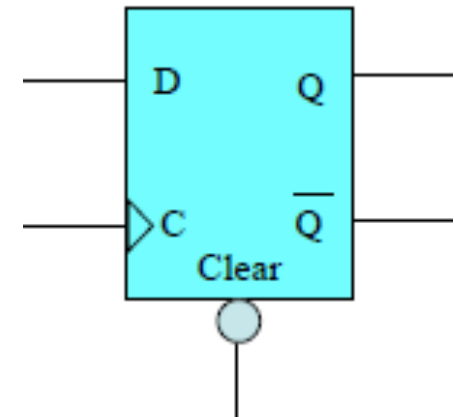
```
-- on the clock's rising edge
```

```
elsif (CLK'event and CLK = '1') then
```

```
    Q <= D;
```

```
end if;
```

```
end process;
```



Binary counter (1/2)

-- Binary counter

library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;

entity binary_counter is

 generic (MIN_COUNT : natural := 0;
 MAX_COUNT : natural := 255);

 port (

 clk : in std_logic;

 reset : in std_logic;

 enable : in std_logic;

 q: out integer range MIN_COUNT to MAX_COUNT

);

end entity;

Binary counter (2/2)

architecture rtl of binary_cunter is
begin

 process (clk)

 variable cnt: integer range MIN_COUNT to MAX_COUNT;

 begin

 if (rising_edge(clk)) then

 if (reset = '1') then

 -- Reset counter to 0

 cnt := 0;

 elsif enable = '1' then

 cnt := cnt + 1;

 end if;

 end if;

 -- Output the current count

 q <= cnt;

 end process;

end rtl;

Basic Shift Register with Asynchronous Reset (1/2)

-- One-bit wide, N-bit long shift register with asynchronous reset

library ieee;

use ieee.std_logic_1164.all;

entity basic_shift_register_asynchronous_reset is

generic (NUM_STAGES : natural := 256);

port (clk : in std_logic;
enable : in std_logic;
reset : in std_logic;
sr_in : in std_logic;
sr_out : out std_logic
);

end entity;

Basic Shift Register with Asynchronous Reset (2/2)

```
architecture rtl of basic_shift_register_asynchronous_reset is
    type sr_length is array ((NUM_STAGES-1) downto 0) of std_logic;
    signal sr: sr_length;          -- Declare the shift register
begin
    process (clk, reset)
    begin
        if (reset = '1') then
            sr <= (others => '0');
        elsif (rising_edge(clk)) then
            if (enable = '1') then
                -- Shift data by one stage; data from last stage is lost
                sr((NUM_STAGES-1) downto 1) <= sr((NUM_STAGES-2) downto 0);
                sr(0) <= sr_in;
            end if;
        end if;
    end process;
    sr_out <= sr(NUM_STAGES-1);
end rtl;
```


Four-State Mealy State Machine (1/5)

- A Mealy machine has outputs that depend on both the state and the
- inputs. When the inputs change, the outputs are updated immediately,
- without waiting for a clock edge. The outputs can be written more than
- once per state or per clock cycle.

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
entity four_state_mealy_state_machine is
```

```
port (
```

```
    clk      : in std_logic;
```

```
    input    : in std_logic;
```

```
    reset    : in std_logic;
```

```
    output   : out std_logic_vector (1 downto 0)
```

```
);
```

```
end entity;
```

Four-State Mealy State Machine (2/5)

architecture rtl **of** four_state_mealy_state_machine **is**

-- Build an enumerated type for the state machine

type state_type **is** (s0, s1, s2, s3);

signal state : state_type; -- Register to hold the current state

begin

process (clk, reset)

begin

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

 state <= s0;

elsif (rising_edge(clk)) **then**

-- Determine the next state synchronously, based on

-- the current state and the input

case state **is**

when s0 =>

if input = '1' **then** state <= s1;

else state <= s0;

end if;

Four-State Mealy State Machine (3/5)

```
when s1 =>
    if input = '1' then state <= s2;
    else                 state <= s1;
    end if;
when s2 =>
    if input = '1' then state <= s3;
    else                 state <= s2;
    end if;
when s3 =>
    if input = '1' then state <= s3;
    else                 state <= s1;
    end if;
end case;
end if;
end process;
```

Four-State Mealy State Machine (4/5)

- Determine the output based only on the current state
- and the input (do not wait for a clock edge).

```
process (state, input)
begin
    case state is
        when s0 =>
            if input = '1' then
                output <= "00";
            else
                output <= "01";
            end if;
```

Four-State Mealy State Machine (5/5)

```
when s1 =>
    if input = '1' then output <= "01";
    else output <= "11";
    end if;
when s2 =>
    if input = '1' then output <= "10";
    else output <= "10";
    end if;
when s3 =>
    if input = '1' then output <= "11";
    else output <= "10";
    end if;
end case;
end process;
end rtl;
```

Four-State Moore State Machine (1/4)

- A Moore machine's outputs are dependent only on the current state.
- The output is written only when the state changes. (State transitions are synchronous.)

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
entity four_state_moore_state_machine is
```

```
port (
```

```
    clk      : in std_logic;
```

```
    input    : in std_logic;
```

```
    reset    : in std_logic;
```

```
    output   : out std_logic_vector (1 downto 0)
```

```
);
```

```
end entity;
```

Four-State Moore State Machine (2/4)

architecture rtl **of** four_state_moore_state_machine **is**

-- Build an enumerated type for the state machine

type state_type **is** (s0, s1, s2, s3);

signal state : state_type; -- Register to hold the current state

begin

process (clk, reset)

begin

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

 state <= s0;

elsif (rising_edge(clk)) **then**

-- Determine the next state synchronously, based on

-- the current state and the input

case state **is**

when s0 =>

if input = '1' **then** state <= s1;

else state <= s0;

end if;

Four-State Moore State Machine (3/4)

```
when s1 =>
    if input = '1' then state <= s2;
    else                 state <= s1;
    end if;
when s2 =>
    if input = '1' then state <= s3;
    else                 state <= s2;
    end if;
when s3 =>
    if input = '1' then state <= s3;
    else                 state <= s1;
    end if;
end case;
end if;
end process;
```


Four-State Moore State Machine (4/4)

-- Output depends solely on the current state.

```
process (state, input)
```

```
begin
```

```
    case state is
```

```
        when s0 =>
```

```
            output <= "00";
```

```
        when s1 =>
```

```
            output <= "01";
```

```
        when s2 =>
```

```
            output <= "10";
```

```
        when s3 =>
```

```
            output <= "11";
```

```
    end case;
```

```
    end process;
```

```
end rtl;
```

Single-port RAM with initial contents (1/4)

-- Single-port RAM with single read/write address and initial contents

library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all ;

entity single_port_ram_with_init **is**

generic (DATA_WIDTH : **natural** := 8;
 ADDR_WIDTH : **natural** := 6);

port (

clk : **in std_logic**;

addr: **in natural range 0 to 2**ADDR_WIDTH - 1**;

data: **in std_logic_vector((DATA_WIDTH-1) downto 0)**;

we : **in std_logic := '1'**;

q : **out std_logic_vector((DATA_WIDTH -1) downto 0)**

);

end single_port_ram_with_init;

Single-port RAM with initial contents (2/4)

architecture rtl of single_port_ram_with_init is

-- Build a 2-D array type for the RAM

```
subtype word_t is std_logic_vector((DATA_WIDTH-1) downto 0);  
type memory_t is array(2**ADDR_WIDTH-1 downto 0) of word_t;
```

function init_ram return memory_t is

```
    variable tmp : memory_t := (others => (others => '0'));
```

begin

```
    for addr_pos in 0 to 2**ADDR_WIDTH - 1 loop
```

-- Initialize each address with the address itself

```
        tmp(addr_pos) := std_logic_vector(to_unsigned(addr_pos,  
                                                    DATA_WIDTH));
```

```
    end loop;
```

```
    return tmp;
```

```
end init_ram;
```

Single-port RAM with initial contents (3/4)

- Declare the RAM signal and specify a default value.
- Quartus II will create a memory initialization file (.mif) based on
- the default value.

```
signal ram : memory_t := init_ram;
```

- Register to hold the address

```
signal addr_reg : natural range 0 to 2**ADDR_WIDTH-1;
```

Single-port RAM with initial contents (4/4)

```
begin
    process(clk)
    begin
        if (rising_edge(clk)) then
            if (we = '1') then
                ram(addr) <= data;
            end if;
            -- Register the address for reading
            addr_reg <= addr;
        end if;
    end process;
    q <= ram(addr_reg);
end rtl;
```