More VHDL

by Erol Seke

For the course “**Introduction to VHDL**”

ESKİŞEHİR OSMANGAZI UNIVERSITY

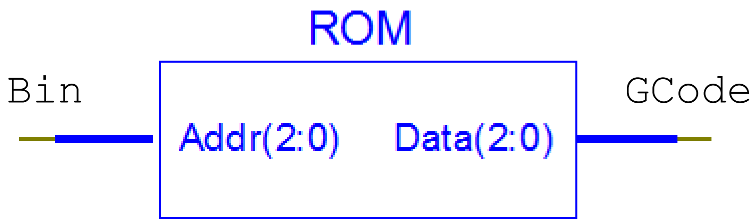
CASE / WHEN

The syntax of `CASE / WHEN` sequential statement is

```
CASE [signal] is
  WHEN [constant] => [Statements]
  WHEN [constant] => [Statements]
  ...
  <WHEN others => [Statements]>
END CASE;
```

Example : 3 Bit Binary to Gray-Code Converter

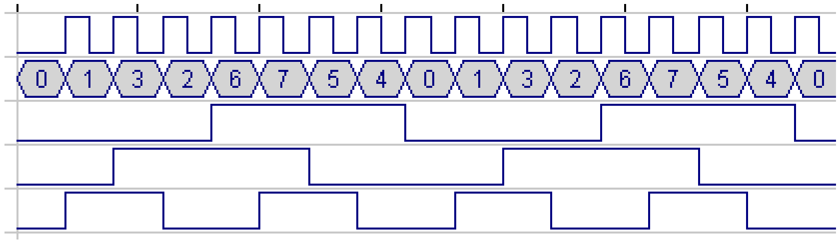
```
process(Bin) is begin
  case Bin is
    when "000" => GCode <="000";
    when "001" => GCode <="001";
    when "010" => GCode <="011";
    when "011" => GCode <="010";
    when "100" => GCode <="110";
    when "101" => GCode <="111";
    when "110" => GCode <="101";
    when "111" => GCode <="100";
    when others => null;
  end case;
end process;
```



`case/when` is similar to `when/else` and `with/select/when` but sequential (used in processes like `if/else/end if`)

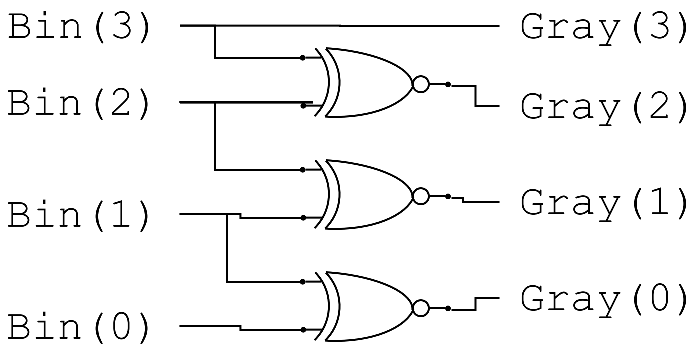
Example

```
entity Gray is Port (  -- Gray Counter
  clk   : in   STD_LOGIC;
  GCode : inout  STD_LOGIC_VECTOR (2 downto 0):="000");
end Gray;

process(clk) is begin
  if(rising_edge(clk)) then
    case GCode is
      when "000" => GCode <="001"; -- such an approach can
      when "001" => GCode <="011"; -- be used for any
      when "011" => GCode <="010"; -- conversion
      when "010" => GCode <="110";
      when "110" => GCode <="111";
      when "111" => GCode <="101";
      when "101" => GCode <="100";
      when "100" => GCode <="000";
      when others => null; -- required for Z,X,U...
    end case;
  end if;
end process;
```



Another Binary To Gray Converter

```
entity Bin2Gray is Port (
  Bin   : in  STD_LOGIC;
  Gray  : inout  STD_LOGIC_VECTOR (3 downto 0):="000");
end Bin2Gray;


Architecture Bin2Gray of Bin2Gray is begin

  Gray(Bin'LEFT) <= Bin(Bin'LEFT);
  Gray(Bin'LEFT-1 downto 0) <= Bin(Bin'LEFT   downto 1) xor
                               Bin(Bin'LEFT-1 downto 0);
end Bin2Gray;
```



How about Gray2Bin ?

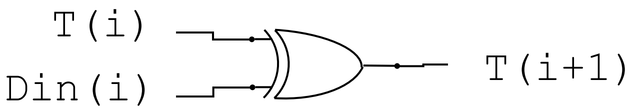
FOR / GENERATE

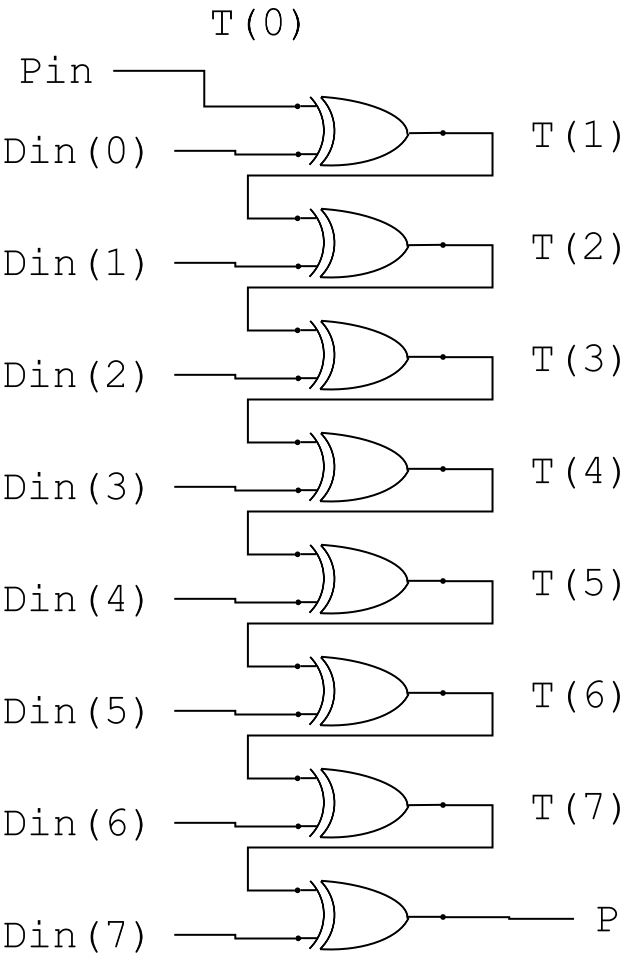
Design : Parity generator for 8 bits input, 1 parity selector input (0=even, 1=odd) and a parity bit output

```
entity Parity is
    Port ( Din : in  STD_LOGIC_VECTOR (7 downto 0);
           Pin : in  STD_LOGIC;   -- parity selection
           P   : out STD_LOGIC); -- parity bit
end Parity;

architecture Parity of Parity is
  signal T : STD_LOGIC_VECTOR(8 downto 0);
begin
  T(0) <= Pin;
  P <= T(8);

  L1: for i in 0 to 7 generate
    T(i+1) <= T(i) xor Din(i); -- this circuit is
  end generate;                  -- generated 8 times

end Parity;
```

This is an entirely combinatorial circuit

This circuit that is instantiated several times





Note : Although we meant to generate the 8-xor gate circuit on the right, it is highly probable that vendor specific synthesizers realize the circuit using LUTs. (having the same net result, of course)

Reason : LUT propagation times are independent of the complexity of the circuit and FPGAs are full of LUTs.

Simulation result

We Want a GENERIC Parity Bit Generator

so that we do not have write another module each time we need a parity bit generator with different size

```
entity Parity is
    Generic ( Nbits : integer := 8 ); -- can use a default size
    Port ( Din : in  STD_LOGIC_VECTOR (Nbits-1 downto 0);
           Pin : in  STD_LOGIC;
           P   : out STD_LOGIC);
end Parity;

architecture Parity of Parity is
  signal T : STD_LOGIC_VECTOR(Nbits downto 0);
begin
  T(0) <= Pin;
  P <= T(Nbits);

  L1: for i in 0 to Nbits-1 generate
    T(i+1) <= T(i) xor Din(i); -- this circuit is
  end generate;                 -- generated Nbits times

end Parity;
```

Hmw : Design a GENERIC Gray2Bin converter using for-loop

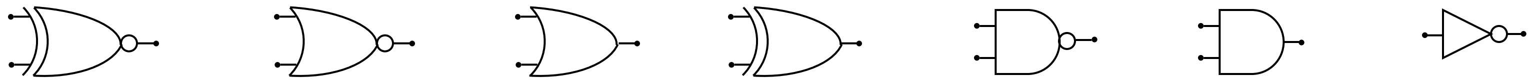
Instantiation of a Generic Module

```
entity EvenParity is
    Port ( Din : in  STD_LOGIC_VECTOR (15 downto 0);
           P   : out STD_LOGIC);
end EvenParity ;

architecture EvenParity of EvenParity is
 component Parity is
    Generic ( Nbits : integer := 8);
    Port ( Din : in  STD_LOGIC_VECTOR (Nbits-1 downto 0);
           Pin : in  STD_LOGIC;
           P   : out STD_LOGIC);
  end component;  -- a generic component ready to be used
begin
  EvenPrt: Parity
    generic map ( -- if not used, default values are assumed
      Nbits => Din'LENGTH -- force to generate 16 bit PG
    )
    port map (
      Din => Din,
      Pin => '0',  -- means even-parity
      P   => P
    );
end EvenParity ;
```

Logical (Bitwise) Operators

```
signal A, B, C : STD_LOGIC_VECTOR(3 downto 0);
signal D, E, F : STD_LOGIC_VECTOR(3 downto 0);
signal G, H, I : STD_LOGIC_VECTOR(3 downto 0);
...
A <= "1010";   B <= "1100";

 C <= NOT A;     -- C is 0101 now
 D <= A AND B;   -- D is 1000 now
 E <= A OR B;    -- E is 1110 now
 F <= A NAND B;  -- F is 0111 now
 G <= A NOR B;   -- G is 0001 now
 H <= A XOR B;   -- H is 0110 now
 I <= A XNOR B;  -- I is 1001 now

-- will work for bit, bit_vector, std_ulogic, std_ulogic_vector too --
```

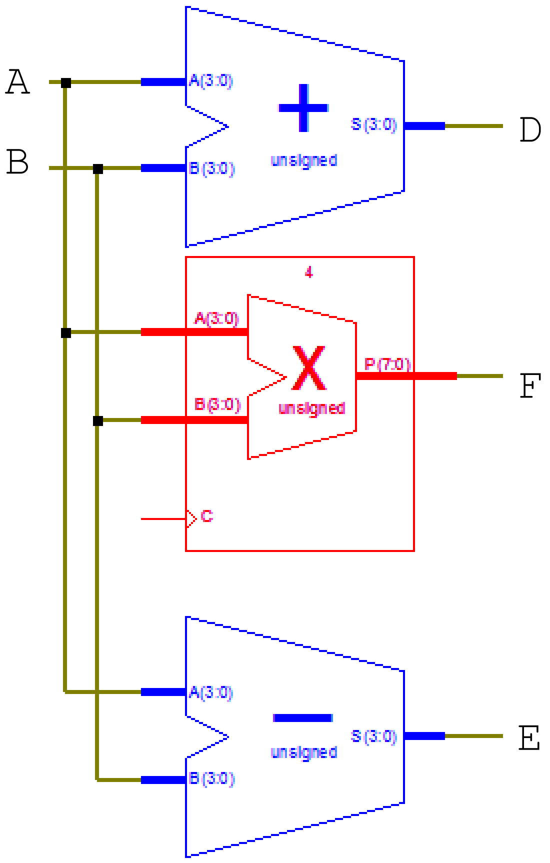
refresh : match these



Arithmetic Operators (via inference)

```
signal A, B, D, E : STD_LOGIC_VECTOR(3 downto 0);
signal C, G : integer;
signal F : STD_LOGIC_VECTOR(7 downto 0);
...
A <= "0110";   B <= "0011";   C <= 6;
```

```
D <= A + B; -- D is 1001 now
E <= A - B; -- E is 0011 now
F <= A * B; -- F is 00010010 now
G <= C / 2; -- G is 3 now
```



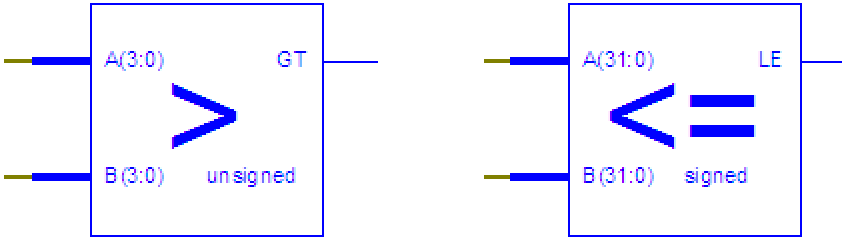
Adders and multipliers are generally implemented with ready-to-use hardware in FPGA.

Homework : how does division circuitry get implemented?

Conditional Operators

```
signal A, B: STD_LOGIC_VECTOR(3 downto 0);
signal C, G : integer;
...
```

```
if(A>B) then ... end if; -- greater than
if(C<G) ...              -- less than
if(A=B) ...              -- equal to
if(A/=B) ...             -- not equal to
if(C<=G) ...    -- less than or equal to
if(A>=B) ... -- greater than or equal to
```



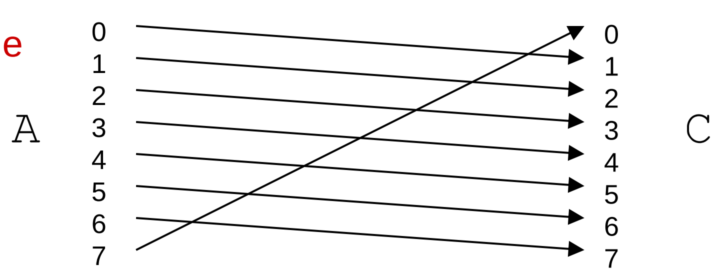
we expect that comparators are also ready-to-use in FPGAs

Shift, Rotate and Concatenate Operators

```
signal A, B, C: BIT_VECTOR(7 downto 0);
...
A <= "01001011";  B <= "11010010";

C <= B sll 1;   -- C is 10100100 now
C <= A sla 2;   -- C is 00101111 now (rmb replicated)
C <= A srl 3;   -- C is 00001001 now
C <= B sra 2;   -- C is 11110100 now (lmb replicated)
C <= A rol 2;   -- C is 00101101 now
C <= A ror 2;   -- C is 11010010 now
C <= B(6 downto 2) & A(7 downto 4);  -- C is 01010100
C <= ('1','1','1','1', A(1), '0','0','0');
```

These operators themselves do not require any logic.

example



A rol 1

Inference in VHDL (as opposed to Instantiation)

```
P <= A * B; -- inference with <18 bit signals
```

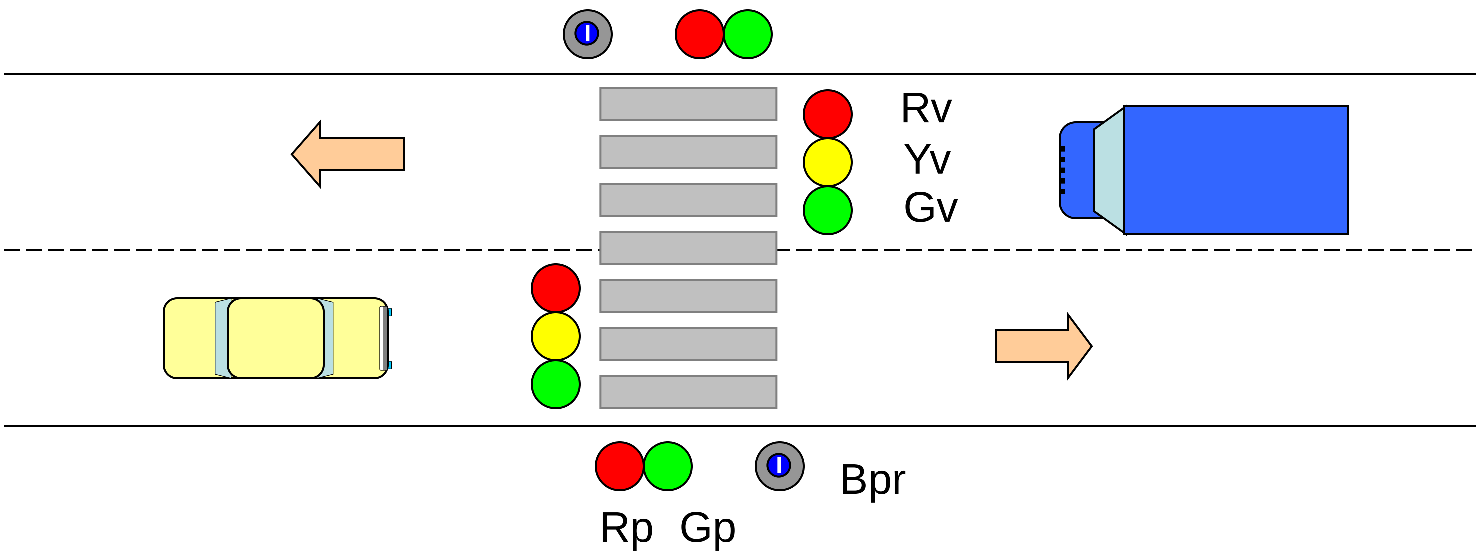
```
MULT18X18_inst : MULT18X18 -- direct instantiation
 port map (
  P => P,      -- 36-bit multiplier output
  A => A,      -- 18-bit multiplier input
  B => B       -- 18-bit multiplier input
);
```

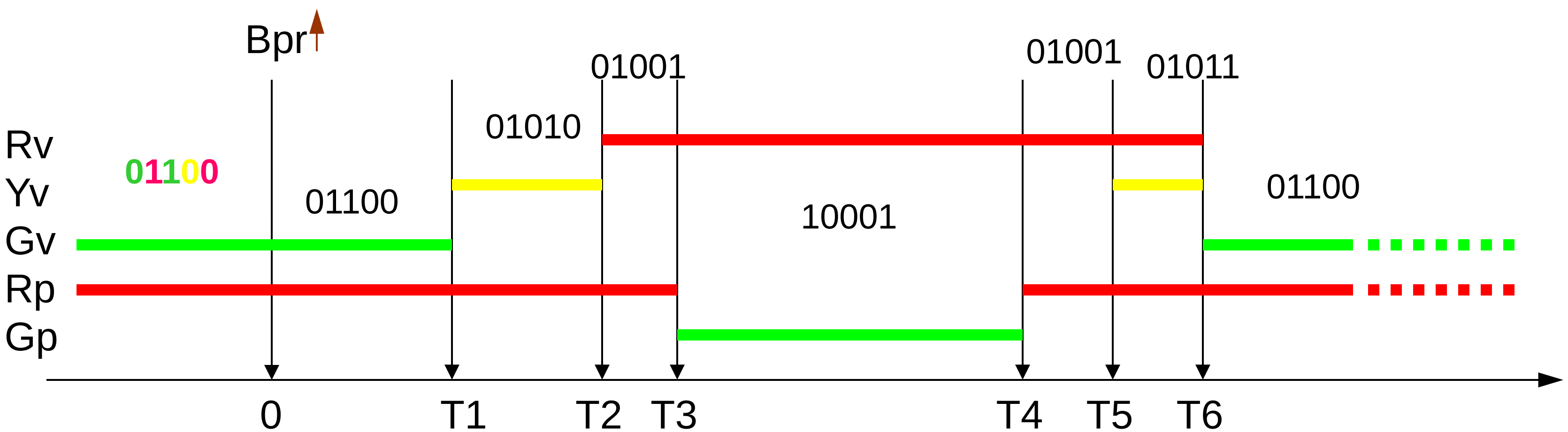
```
O <= I1 when S='1' else I0;
```

```
MUXF8_inst : MUXF8
 port map (
  O  => O,     -- Output of MUX to general routing
  I0 => I0,    -- Input (tie to MUXF7 LO out)
  I1 => I1,    -- Input (tie to MUXF7 LO out)
  S  => S      -- Input select to MUX
);
```

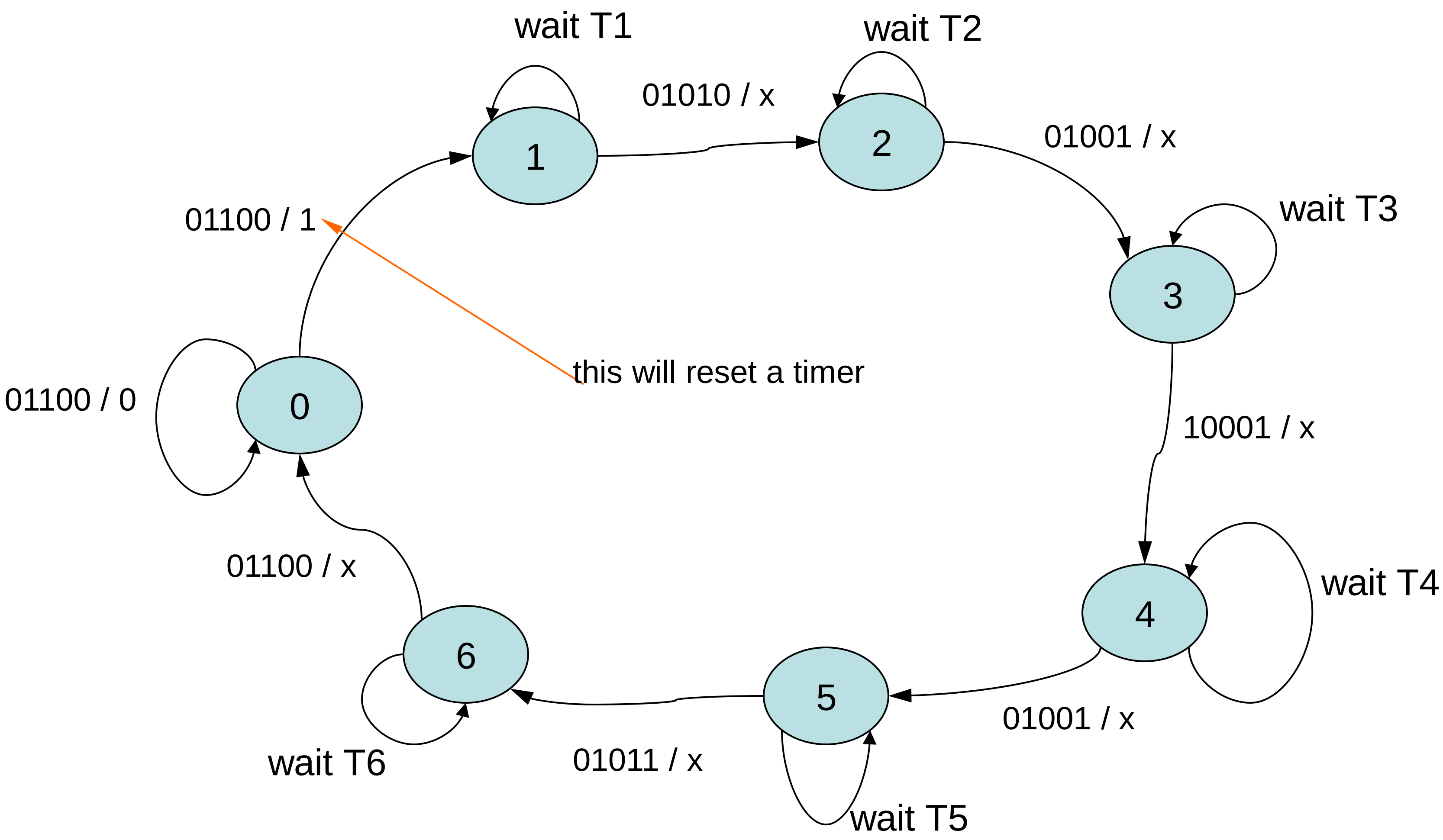
Example (Traffic Lights)

Traffic lights on a *pedestrian crossing* is controlled by a pedestrian request button (Bpr) via a digital controller. Normally green lights for vehicles lit all the time until Bpr signals. The timing of the lights is illustrated with the time-diagram below.

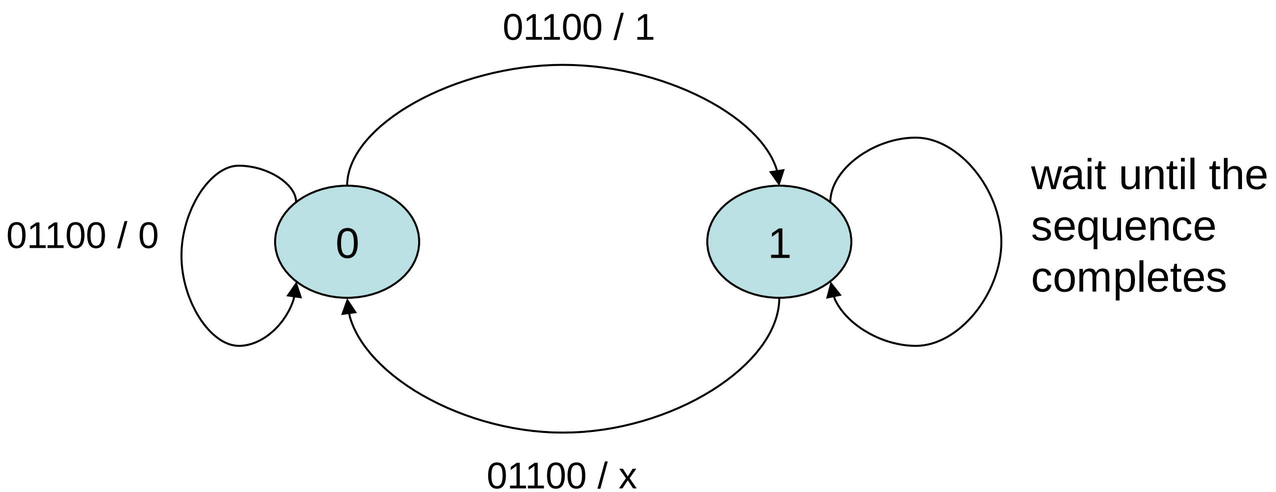


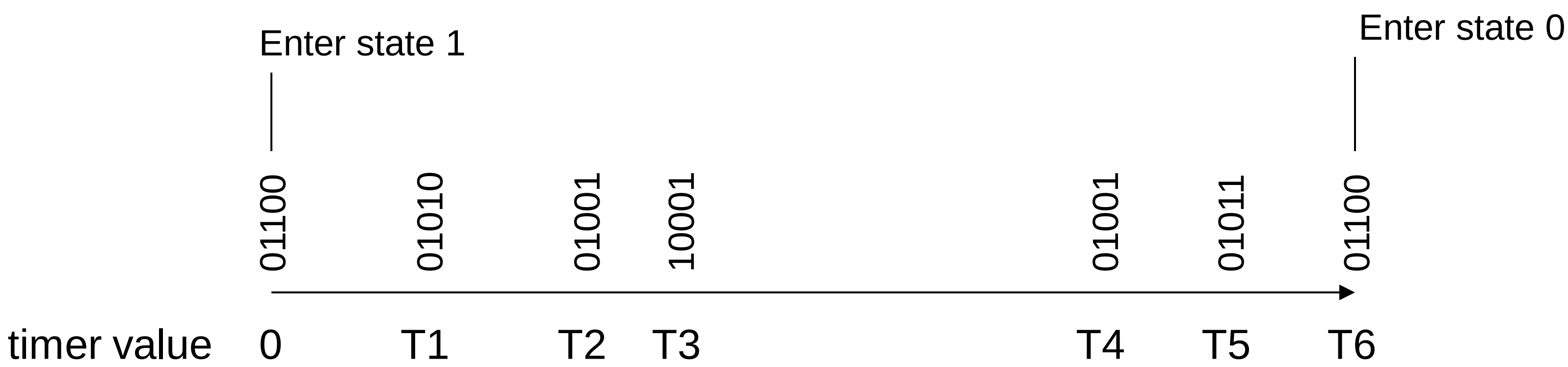


State Diagram



Simplified States





Counter / Timer Component

```
architecture Timer of Timer is
  signal clk1 : STD_LOGIC;
  signal cntr1s : integer range 0 to CLOCK_FREQ;
begin
  CNT1: process(clk) is begin
    if(rising_edge(clk)) then
      if(cntr1s=CLOCK_FREQ) then
        cntr1s <= 0;
        clk1 <= '1';
      else
        cntr1s <= cntr1s +1;
        clk1 <= '0';
      end if;
    end if;
  end process;

  CNT: process(clk1, Rst) is begin
    if(Rst='1') then
      Tout <= 0;
    elsif(rising_edge(clk1)) then
      Tout <= Tout +1;
    end if;
  end process;
end Timer;
```

so that every tick is 1 second

```
entity Timer is
  Generic ( CLOCK_FREQ : integer := 49999999 );
  Port ( clk  : in  STD_LOGIC;
         Rst  : in  STD_LOGIC;
         Tout : inout integer);
end Timer;
```

Keep in mind that every clock signal reserves a dedicated clock route in FPGA (if exist). One should seriously consider fully synchronous designs when this poses a problem in terms of clock resources. Such problems usually arouse when the designs are complex and require multiple clock signals.

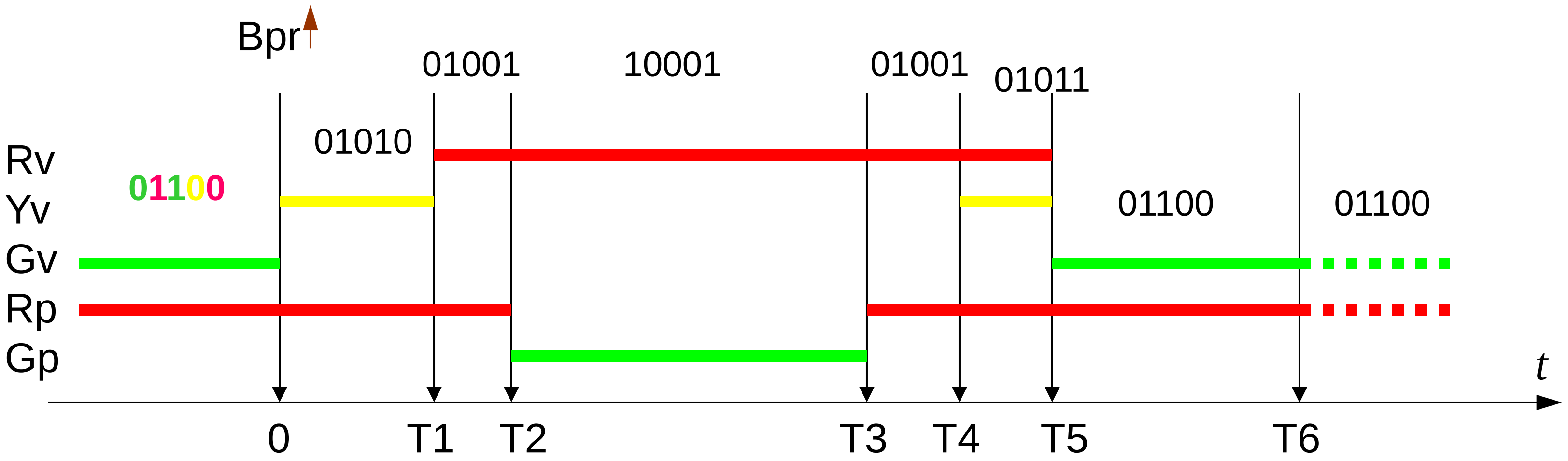
```vhdl
signal Rst : STD_LOGIC;
signal Tout : integer;
  -- Timer action values
constant T1 : integer := 10;
constant T2 : integer := T1 + 10;
constant T3 : integer := T2 + 5;
constant T4 : integer := T3 + 20;
constant T5 : integer := T4 + 5;
constant T6 : integer := T5 + 3;
signal state : STD_LOGIC;
signal Lights: STD_LOGIC_VECTOR(4 downto 0);
```

```vhdl
TMR: Timer port map (
   clk => clk,
   Rst => Rst,
   Tout => Tout
 );

 LGTS: process(clk) is begin
   if(rising_edge(clk)) then
     if(state='0') then
       if(Bpr='1') then
         Rst <='1'; state <= '1';
       end if;
     else
       Rst <= '0';
       case Tout is
         when T1 => Lights <= "01010";
         when T2 => Lights <= "01001";
         when T3 => Lights <= "10001";
         when T4 => Lights <= "01001";
         when T5 => Lights <= "01011";
         when T6 => Lights <= "01100"; state <= '0';
         when others => Null;
       end case;
     end if;
   end if;
 end process;
```

Homework :

In previous traffic lights problem, the intentional delay T1 before the lights start the sequence is there to prevent a pedestrian keep pressing the Bpr button and continuously blocking the vehicles.

Change the design so that action starts right after the Bpr signal by having this delay at the end of the sequence and after the Gv and Rp lights.



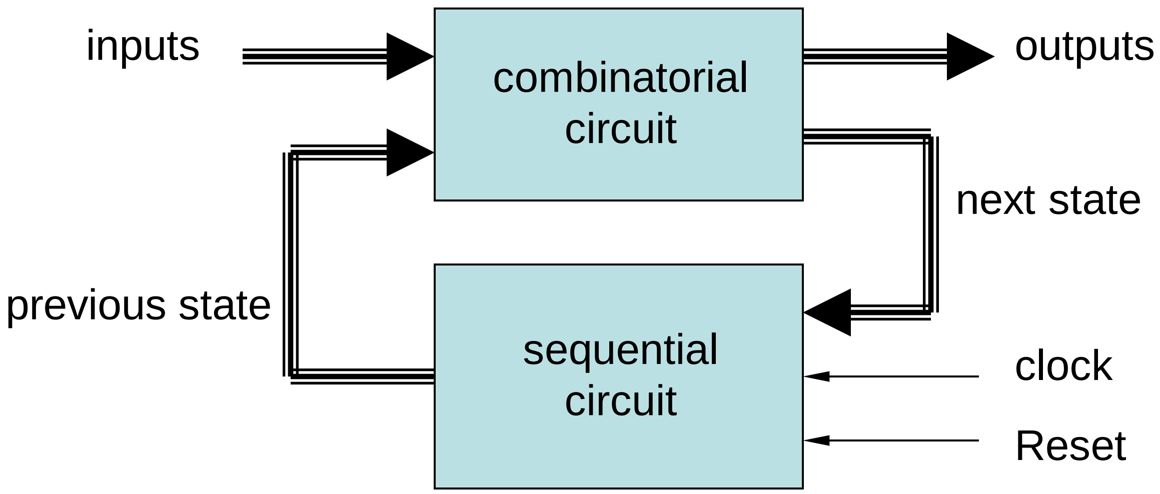
Enumerated Types and Subtypes

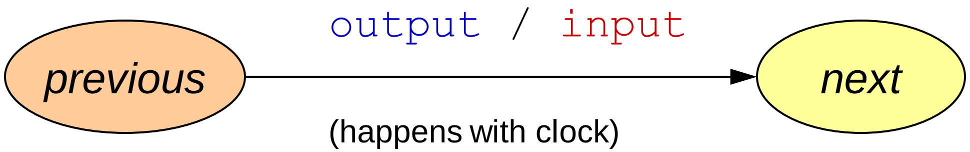
```
TYPE color is (red, yellow, green, blue, white, black);
...
SUBTYPE traffic_colors is color range red to green;
...
type colorset is array (0 to 2) of color;
...
signal tr_light : traffic_colors;
signal forecolor, backcolor : color;
signal myset: colorset := (blue, white, black);


tr_light <= red;  -- OK

tr_light <= blue;  -- error

forecolor <= tr_lights;  -- OK

backcolor <= myset(1);  -- OK
```

Records

```
TYPE RGBcolor is record
 red    : STD_LOGIC_VECTOR(7 downto 0);
 green  : STD_LOGIC_VECTOR(7 downto 0);
 blue   : STD_LOGIC_VECTOR(7 downto 0);
 intensity : integer;
end record;
...
signal R : STD_LOGIC_VECTOR(7 downto 0);
signal RGB, RGB2 : RGBcolor;


 RGB <= RGB2;
 RGB.red <= R;
 ...

 variable MyColor, X : RGBcolor;
 ...
 MyColor := ("01010101",x"5B",x"24",14);
 X := (red => R, others => '0');
```

State Machines





What the state diagram means:

When we are in *previous* state move to the *next* state on the clock if the `input` is right.
Set the `output` as given.

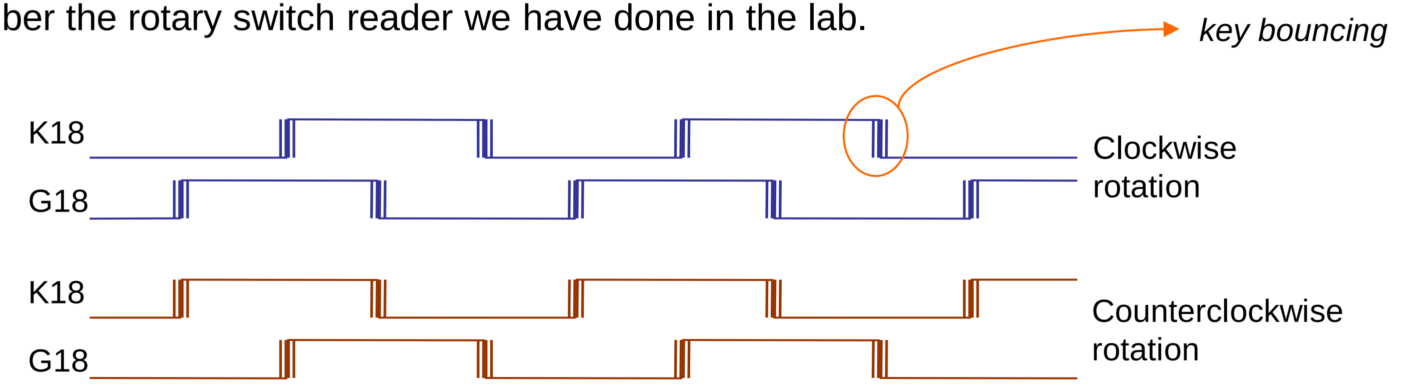
A Template

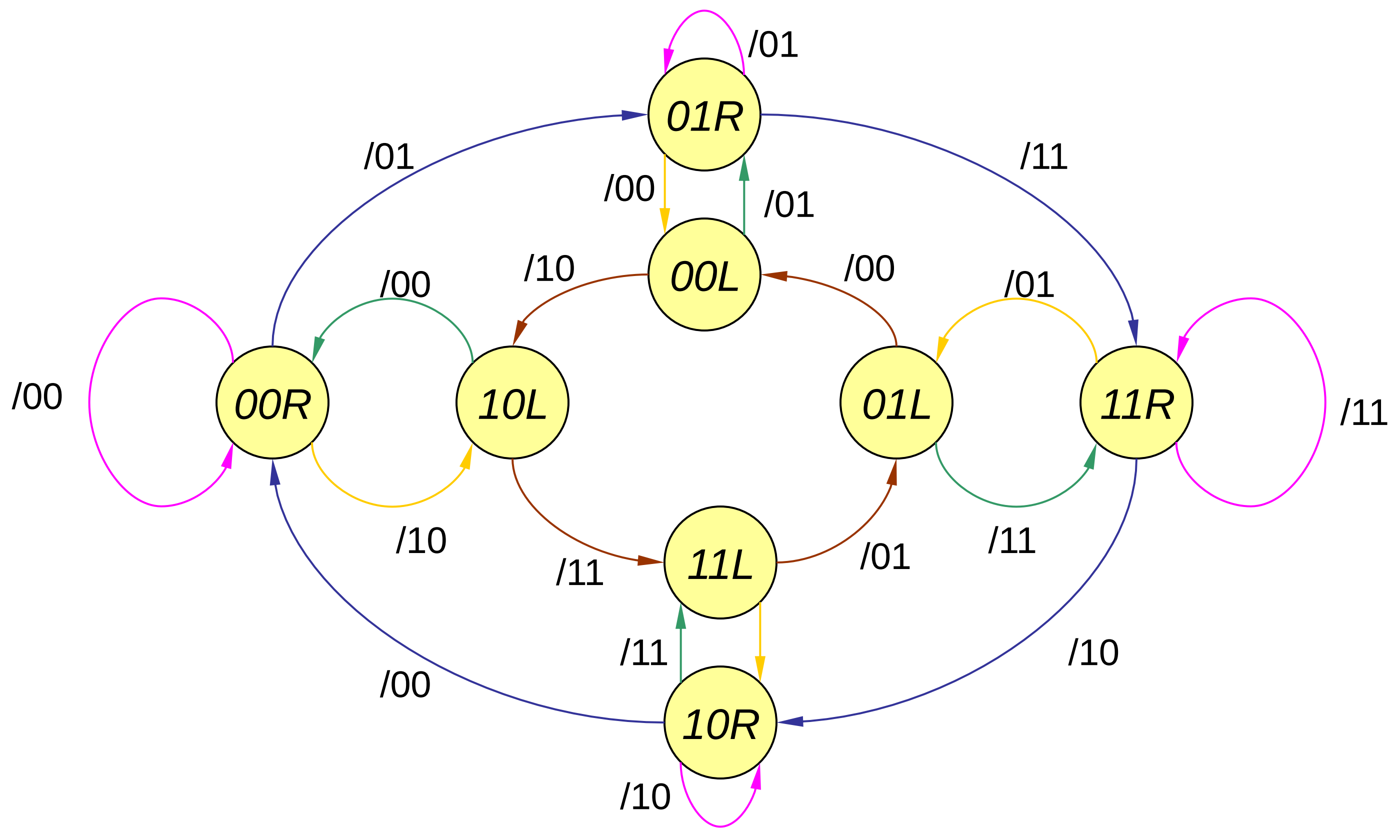
```
  type state is (state0, state1, state2, ...
...
-- Combinatorial Part
process(prev_state) is
begin
  case prev_state is
    when state0 => precalculated_value <= ...
         if(condition) then next_state <= stateX;
         elsif(condition) then next_state <= stateY;
         ...
    when state1 =>
    ...
end process;

-- Sequential Part
process(clock, Reset) is
begin
  if(Reset = '1') then
    prev_state <= state0;
  elsif(RISING_EDGE(clock)) then
    prev_state <= next_state;
    output <= precalculated_value;
  end if;
end process;
```

Homework: Do the traffic lights according to the template

Rotary Switch Reader with State Machine

Remember the rotary switch reader we have done in the lab.





Homework :

Read sections 4.5, 5.4, 8

Do problems 8.2, 8.3

Implement the rotary switch state machine.

END