Generic size 2-1 MUX example

```
entity mux21 is
       generic(size:natural:=16);                    --default value 16
       port(i0,i1 : in  std_logic_vector(size-1 downto 0); --based on "size"
            c     : in std_logic;
            z     : out std_logic_vector(size-1 downto 0));
end mux21;

architecture beh of mux21 is
begin
       update: process(c,i0,i1) is
       begin
              if c='0' then
                  z<=i0;
              else
                  z<=i1;
              end if;
       end process update;
end beh;
```

Instantiation of Generic mux

1. If you have given a default value for the generic (above the default is sixteen) and you want to instantiate the component with the default generics, the instantiation looks like a normal instantiation:

```
dut1 : entity work.mux21(beh)
   port map(i0,i1,c,z);
```

2. If you have not specified a default value, or you want to instantiate a component with a value other than the default, the instantiation looks like this:

```
dut2 : entity work.mux21(beh)
   generic map(8)            -- note: no ; at end of generic map
   port map(i0,i1,c,z);
```

Note about instantiation: In these examples, the signals i0, i1, and z have been declared as internal signals with a set width (as in a number). So for example 1, the internal signal declaration inside the testbench would look like this:

```
architecture test of testMux21 is
    signal i0,i1,z  : std_logic_vector(15 downto 0); -- note 15 not "size"
    signal c        : std_logic;
begin
    dut1 : entity work.mux21(beh)
        port map(i0,i1,c,z);
…
end test;
```

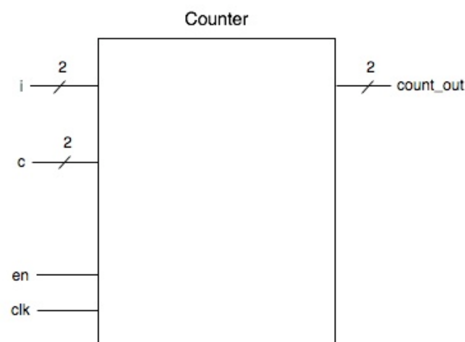
And for example 2, would have looked like this:

```
architecture test of testMux21 is
    signal i0,i1,z  : std_logic_vector(7 downto 0); -- note 7 (not "size")
    signal c        : std_logic;
begin
    dut1 : entity work.mux21(beh)
       generic map(8)
        port map(i0,i1,c,z);
…
end test;
```

Definition Review: Asynchronous means independent of the clock. So asynchronous set and reset means that if set or reset are asserted, the set/reset should happen immediately, no matter what the clock is

Control Signals to determine function.

We often want to have signals that will control what function our device should perform. This is in essence a state machine. Let's look at a 2-bit counter, with the function HOLD, COUNT UP, COUNT DOWN, and LOAD. Hold means that the output does not change. Count up means the output gets the next value in the sequence (00,01,10,11,00…). Count down means the output gets the previous in that same series, meaning it will count down (00,11,10,01,00,11…). Load means the output will get the value of the input (note: this is just like a normal register).



We need a 2-bit (because we have 4 functions) control signal c where we will assign the meaning of c to be:

c	**Function**
00	Hold
01	Count Up
10	Count Down
11	Load

We will look at the current state of the counter as well as what function we're performing to determine what the next state should be.

Let's set up a Current State/Next State table (what should the blanks be?):

	c(1)	c(0)	Current(1)	Current(0)	count_out(1)	count_out(0)
hold	0	0	0	0	**0**	**0**
	0	0	0	1	**0**	**1**
	0	0	1	0	**1**	**0**
	0	0	1	1	**1**	**1**
count up	0	1	0	0	**0**	**1**
	0	1	0	1	**1**	**0**
	0	1	1	0	**1**	**1**
	0	1	1	1	**0**	**0**
count down	1	0	0	0		
	1	0	0	1		
	1	0	1	0		
	1	0	1	1		
load	1	1	0	0	**i(1)**	**i(0)**
	1	1	0	1	**i(1)**	**i(0)**
	1	1	1	0	**i(1)**	**i(0)**
	1	1	1	1	**i(1)**	**i(0)**

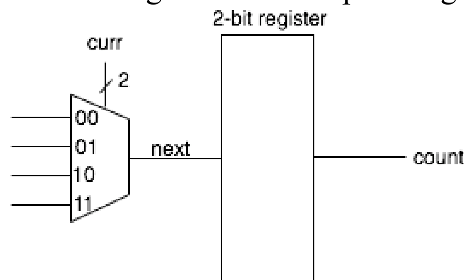
We will now use this table to implement this counter 2 different ways. The first way is behaviorally, and the second is structurally using only MUXes and a register.

```
entity counter is
    port(i,c       : in std_logic_vector(1 downto 0);
         en, clk   : in std_logic;
         count_out : out std_logic_vector(1 downto 0));
end counter;

architecture beh of counter is
begin
    update: process (c,i,en,clk) is
        variable hold_count: std_logic_vector(1 downto 0):="00";
    begin
        if en='1' and clk='1' and clk'event then
            case c is
                -- HOLD
                when "00" => hold_count:=hold_count;

                -- LOAD
                when "11" => hold_count:=i;

                -- COUNT UP
                when "01" =>
                                case hold_count is
                                    when "00" => hold_count:="01";
                                    when "01" => hold_count:="10";
                                    when "10" => hold_count:="11";
                                    when "11" => hold_count:="00";
                                    when others => hold_count:="00";
                                 end case;

                -- COUNT DOWN
                when "10" =>
                                case hold_count is
                                    when "00" => hold_count:="11";
                                    when "01" => hold_count:="00";
                                    when "10" => hold_count:="01";
                                    when "11" => hold_count:="10";
                                    when others => hold_count:="00";
                                 end case;

                -- default case
                when others => report "INVALID";
            end case;
        end if;
        count_out<=hold_count;
end process;
end beh;
```

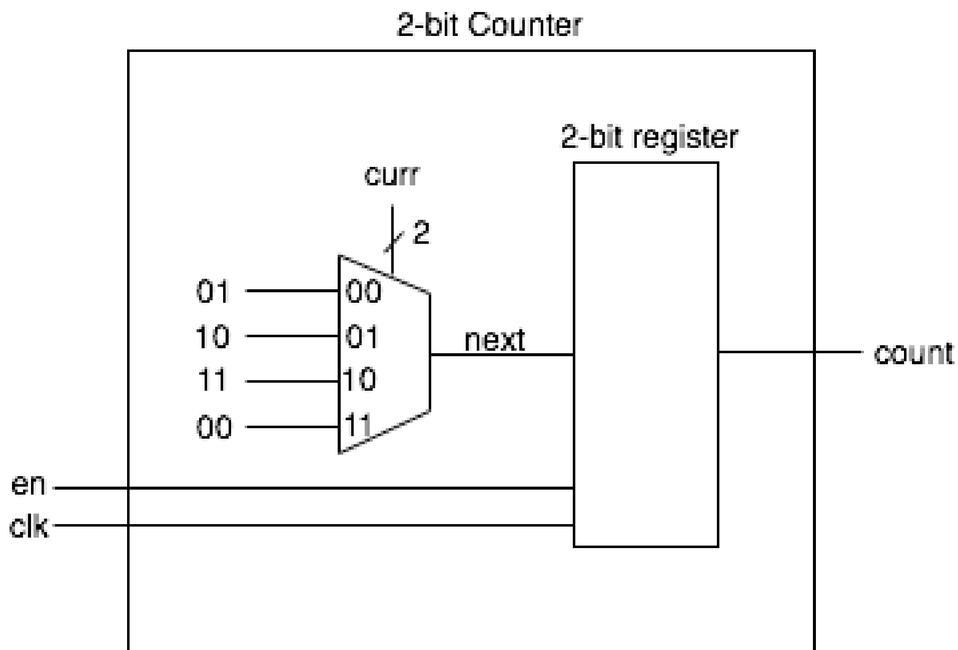
If we want to do a structural design using MUXes and registers, we go to the truth table. The basic concept is that we use the current state along with the controls as the control for a MUX. Let's look at a smaller example to develop the concept first. Let's just look at a 2-bit counter that only counts up. The state transition truth table is:

curr	count
00	01
01	10
10	11
11	00

We can use a 4-1 MUX and a 2-bit register to develop a design for this counter.



So, when curr = "00", what should next be? Here is where we look at the truth table. Look up the correct output value when curr= "00" and we see that it is supposed to be "01". So, when curr = "00" the MUX will choose the first input which we can set to be "01". So, the next value to get loaded into the register would be "01".



Here's the truth table again.

	c	Current	count_out
hold	00	00	**00**
	00	01	**01**
	00	10	**10**
	00	11	**11**
count up	01	00	**01**
	01	01	**10**
	01	10	**11**
	01	11	**00**
count down	10	00	**11**
	10	01	**00**
	10	10	**01**
	10	11	**10**
load	11	00	**i**
	11	01	**i**
	11	10	**i**
	11	11	**i**