

**ECSE-323**

# Digital System Design

**VHDL Lecture #4**

## Example: 4-bit Counter with Control Signals

```
signal count : integer range 0 to 15;
```

```
begin -- begin architecture block
```

```
counter1 : process(reset, Clk)
```

```
begin
```

```
  if reset = '1' then
```

```
    count <= 0;
```

```
  elsif Clk = '1' and Clk'event then
```

```
    if enable = '1' then
```

```
      if load = '1' then count <= ldata;
```

```
      elsif updown = '1' then count <= count + 1;
```

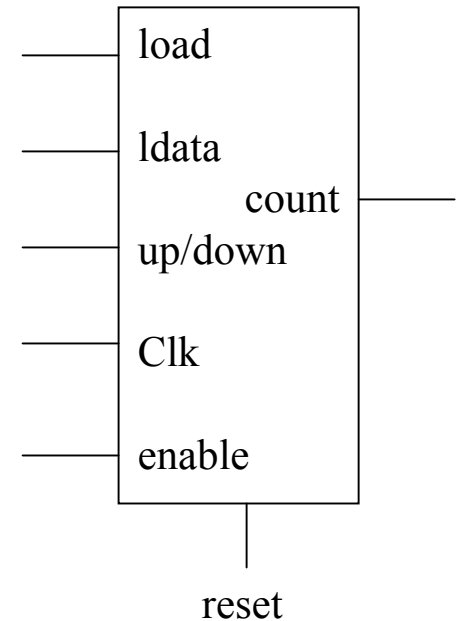
```
      else count <= count - 1;
```

```
      end if; -- if load
```

```
    end if; -- if enable
```

```
  end if; -- if reset
```


```
end process;
```



# How to Enable a Clocked Operation:


```
elsif Clk = '1' and Clk'event then
    if enable = '1' then
        -- do stuff when enabled
```

This is the right way to enable a clocked operation



```
elsif Clk = '1' and enable = '1' and Clk'event then
    -- do stuff when enabled
```

This is a bad way to enable a clocked operation



**Why is this type of VHDL construct bad?**

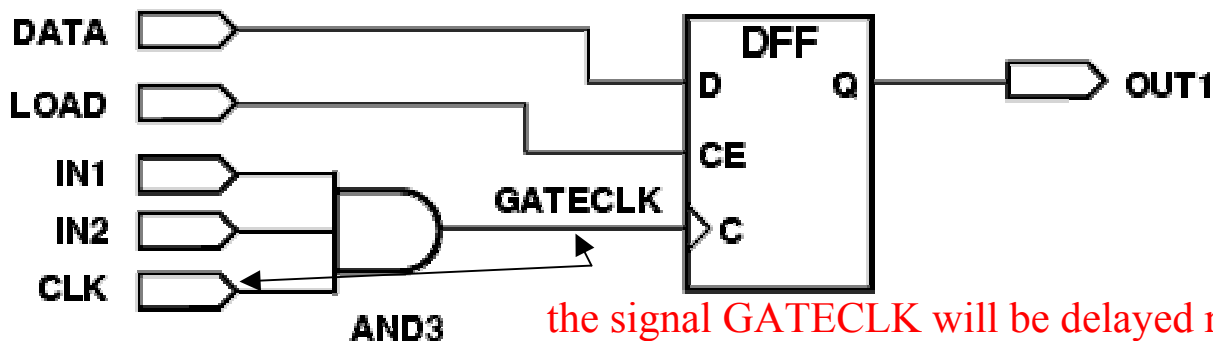
**It may confuse the compiler. It may create a *gated clock* (causing clock skew and glitches and slowing down the system)**

```

architecture BEHAVIORAL of gate_clock is
    signal GATECLK: STD_LOGIC;
begin
    GATECLK <= (IN1 and IN2 and CLK);
    GATE_PR: process (GATECLK,DATA,LOAD)
    begin
        if (GATECLK'event and GATECLK='1') then
            if (LOAD='1') then
                OUT1 <= DATA;
            end if;
        end if;
    end process;
end BEHAVIORAL;

```

## Implementation of a Gated Clock



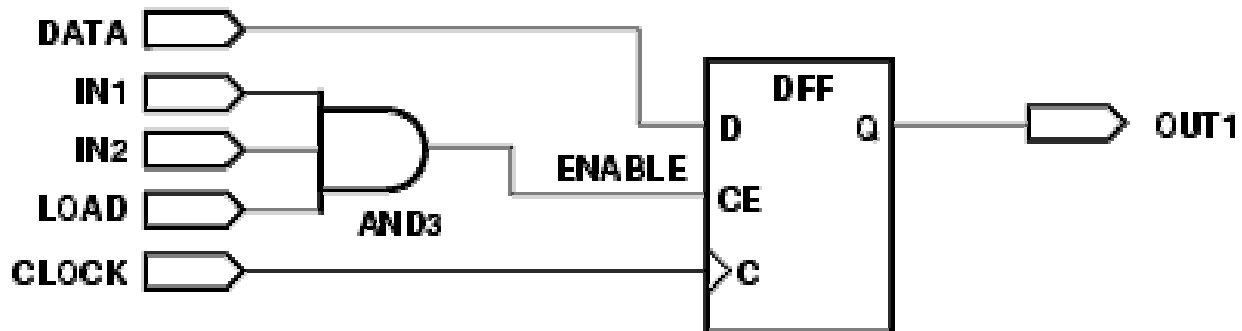
example is from the Xilinx Synthesis and Simulation Design Guide -  
<http://toolbox.xilinx.com/docsan/xilinx4/data/docs/sim/sim.html>

```

architecture BEHAVIORAL of gate_clock is
    signal ENABLE: STD_LOGIC;
begin
    ENABLE <= IN1 and IN2 and LOAD;
    EN_PR: process (ENABLE, DATA, CLOCK)
    begin
        if (CLOCK'event and CLOCK='1') then
            if (ENABLE='1') then
                DOUT <= DATA;
            end if;
        end if;
    end process;
end BEHAVIORAL;

```

## Implementation of an Enabled Clock



example is from the Xilinx Synthesis and Simulation Design Guide -  
<http://toolbox.xilinx.com/docsan/xilinx4/data/docs/sim/sim.html>

## Arithmetic Signal Types and Operations

`std_logic_vector(N downto 0)`

defines an  $(N+1)$ -bit unsigned binary number whose LSB is bit 0 and MSB is bit  $N$ .

**$S \leftarrow X + Y;$**

This expression describes an  $(N+1)$ -bit adder *without* carry in or carry out.

The signal  $S$  has  $N+1$  bits (and not  $N+2$ ).

## Note:

VHDL requires that at least one of the operands on the RHS have the same bit length as the LHS.

To add in a Carry input and output we can write:

```
S2 <= ('0' & X) + Y + Cin;
```

where S2 has one more bit than X and Y

**&** is the VHDL *concatenation* operator

It combines two vectors into a single longer vector.

for example:

```
signal X, Y : std_logic_vector(1 downto 0);
signal Z : std_logic_vector(3 downto 0);

  .   .   .
X <= "11";
Y <= "10";
Z <= X & Y; -- Z = "1110"
```



**Question:** How can we represent *signed* numbers in VHDL?

**Answer:** define a new signal type, along with associated arithmetic operators

## Signed vs. Unsigned Signal Types

**USE** `ieee.std_logic_unsigned.all`  
interprets all *std\_logic\_vector* signals as unsigned binary numbers.

**USE** `ieee.std_logic_signed.all`  
interprets all *std\_logic\_vector* signals as signed binary numbers (2's complement).

If you want to mix *both* signed and unsigned signals then include

**USE** `ieee.std_logic_arith.all`

You *must* then explicitly say what types are intended:

```
signal    X : SIGNED;
```

```
signal    X : UNSIGNED;
```

The `use ieee.std_logic_arith.all` statement should be included *after* the `use ieee.std_logic_1164.all` statement.

## Integer Signal Type

The *integer* signal type is very useful for implementing counters.

```
signal X : INTEGER range -16 to 15;
```

If the range is left out, the default range is used:

$$-(2^{31} - 1) \text{ to } (2^{31} - 1)$$

The number of bits in an integer signal is not specified explicitly. The compiler determines how many bits to allocate to the signal.

The **integer** data type is built in to the VHDL standard, hence *no library needs to be included* to use this type.

```
signal X : integer range -32768 to 32767;
```

In the above example, 16 bits would be allocated to the signal **X**.

There are *conversion functions* to convert between the various signal types.

These conversions are often necessary because in VHDL the types of *signals* on either side of an assignment statement must be the *same*.

## Commonly used conversion routines are:

```
CONV_INTEGER (operand) ;  
CONV_UNSIGNED (operand) ;  
CONV_SIGNED (operand) ;  
CONV_STD_LOGIC_VECTOR (operand, size) ;
```

**operand** can be of type *integer*, *unsigned*, *signed*, or *std\_logic*.

### Example

```
signal Y : integer range 0 to 100 ;  
signal X : std_logic_vector (6 downto 0) ;  
  
X <= CONV_STD_LOGIC_VECTOR (Y, 7) ;  
Y <= CONV_INTEGER (X) ;
```



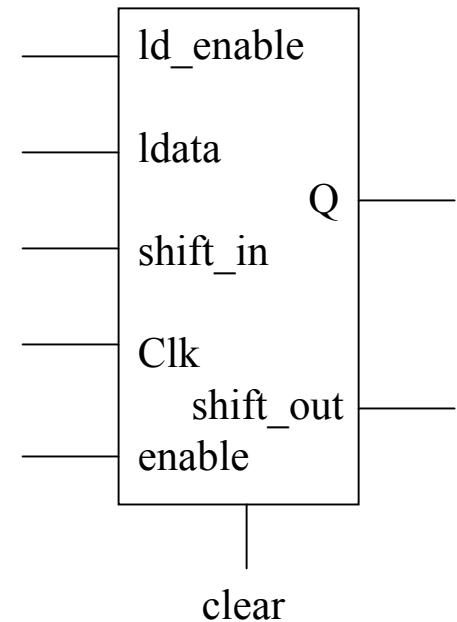
OK, let's look at some examples



```

signal Q : std_logic_vector(7 downto 0);
begin -- begin architecture block
sreg1 : process(clear, Clk)
begin
    if clear = '1' then
        Q <= 0;
    elsif Clk = '1' and Clk'event then
        if ld_enable = '1' then
            Q <= ldata;
        else
            Q <= shift_in & Q(7 downto 1);
            shift_out <= Q(0);
        end if; -- if enable
    end if; -- if clear
end process;

```



what is the  
value of shift\_out?

**SHIFT REGISTER**

```
Q <= shift_in & Q(7 downto 1);  
shift_out <= Q(0);
```

Suppose that  $Q = "11010110"$  going into the process block, and that  $shift\_in = '0'$ .

After evaluation of the process block we have that  $Q = "01101011"$

$shift\_out$  will be assigned the value of  $Q(0)$  *before* the process block is evaluated, i.e.  
 $shift\_out = 0$ , not  $shift\_out = 1$

This is because events on signals created as a result of assignments within a process block actually don't take effect (or aren't *scheduled*) ***until the end of the process block is reached.***

**The values of *all signals* inside of a process block are the values they have at the beginning of the process block.**

```
signal count : integer range 0 to 639;
signal pulse : std_logic;
begin -- begin architecture block
div1 : process(clear,Clk)
begin
    pulse <= '0'; -- give a default value
    if clear = '1' then
        count <= 639;
    elsif Clk = '1' and Clk'event then
        if count_enable = '1' then
            if count = 0 then
                pulse <= '1'; count <= 639;
            else count <= count - 1; -- count down
            end if; -- if count
        end if; -- if count_enable
    end if; -- if clear
end process;
```

**This circuit provides a pulse that is one clock period wide every 640 clock pulses.**

## **FREQUENCY DIVIDER**

Why do we count down instead of counting up?

**Because it often takes *less circuitry* to detect when *count = 0* than to detect when *count = 639*.**

A common use for frequency divider circuits is to effectively slow down clocking of a circuit.

**But, you should not use the pulse signal as a clock!**

Instead, use it as a control signal, such as a counter enable.

For example, suppose we want to make a circuit that generates a *pulse* once every *640 clock pulses* and another one every *640\*640 clock pulses*.

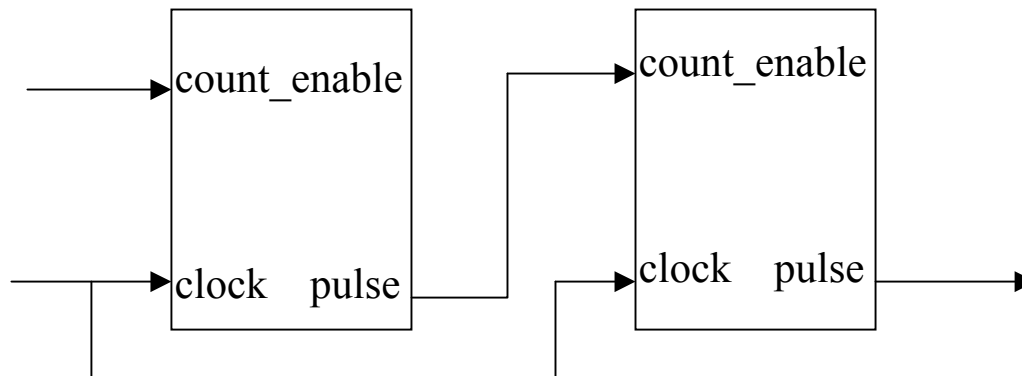
**We can do this by having one frequency divider circuit drive another.**





**This is a ripple counter, and should be avoided!**

**Instead, use a fully synchronous design:**



# Why are ripple counters bad?

- 1. Because they are slow.**
- 2. Outputs do not all change at the same time.**

# Why are they slow?

- 1. Because the clock input to a circuit element is delayed relative to the clock input of the preceding element. Thus any feedback to the preceding element (such as in an FSM) will be delayed, requiring a reduction in clock frequency.**
- 2. Ripple clock signals must be routed through logic cells (slow) rather than via dedicated (fast) clock wiring.**

Figure 5. FLEX 10K LAB

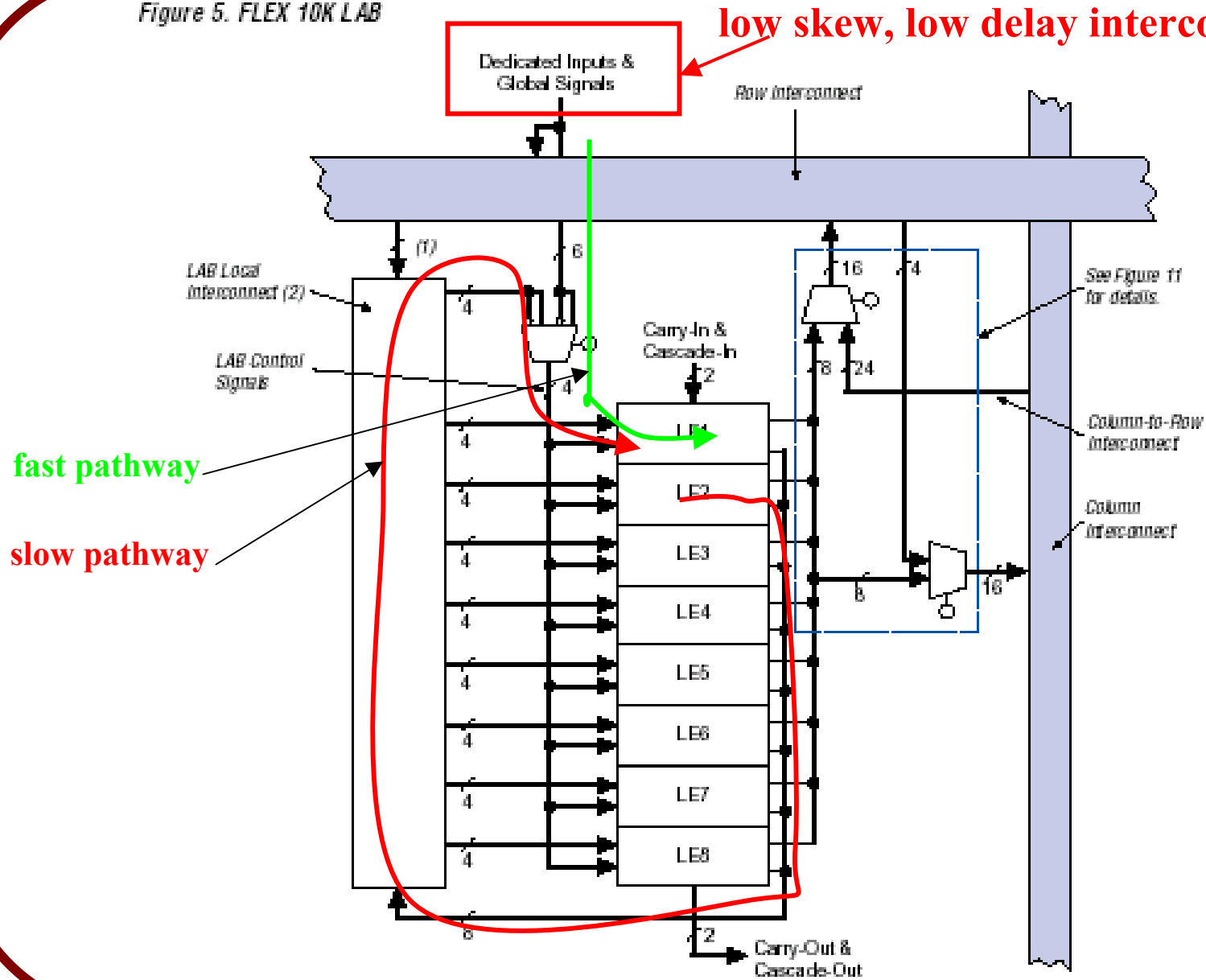
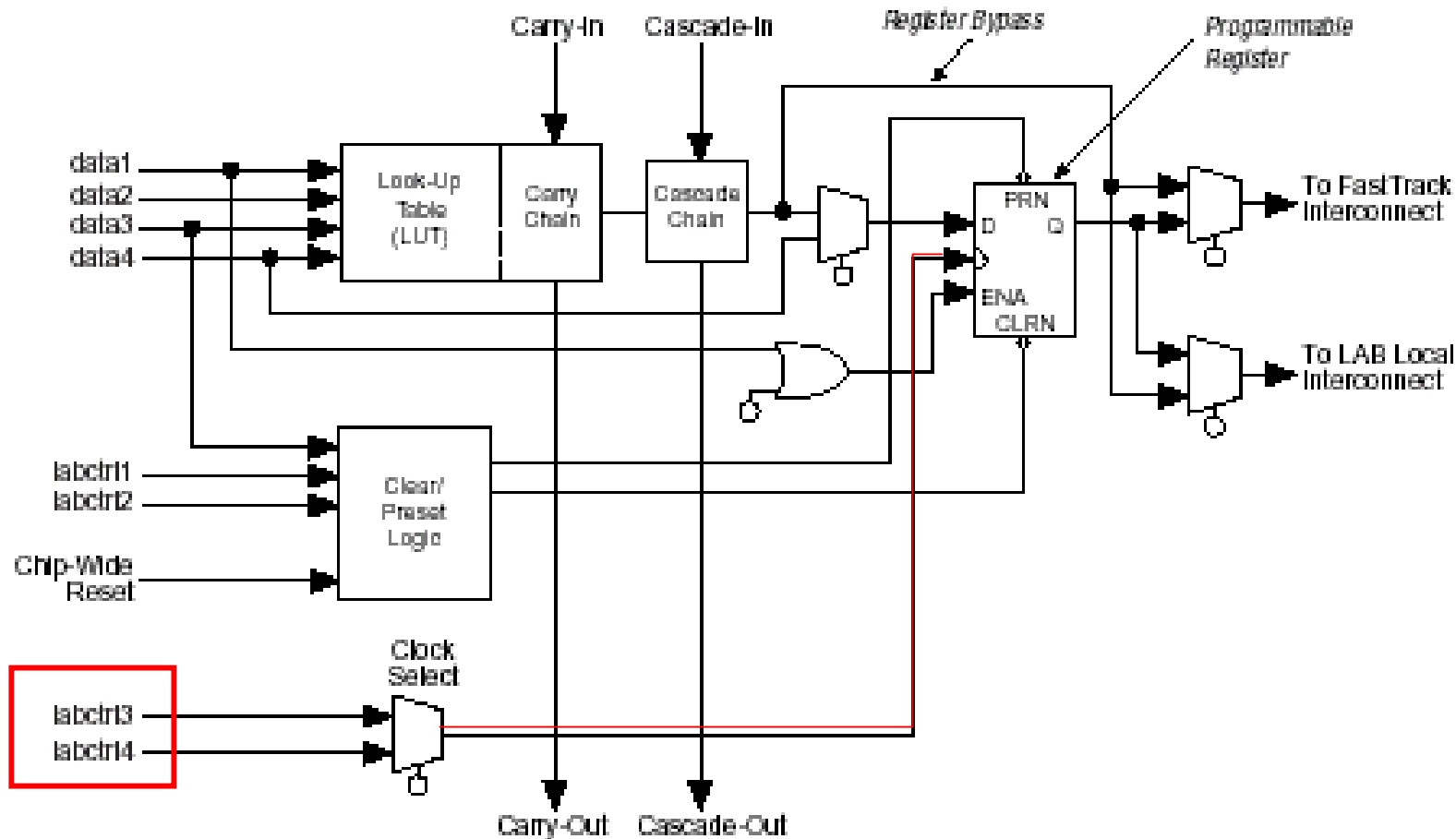


Figure 6. FLEX 10K Logic Element



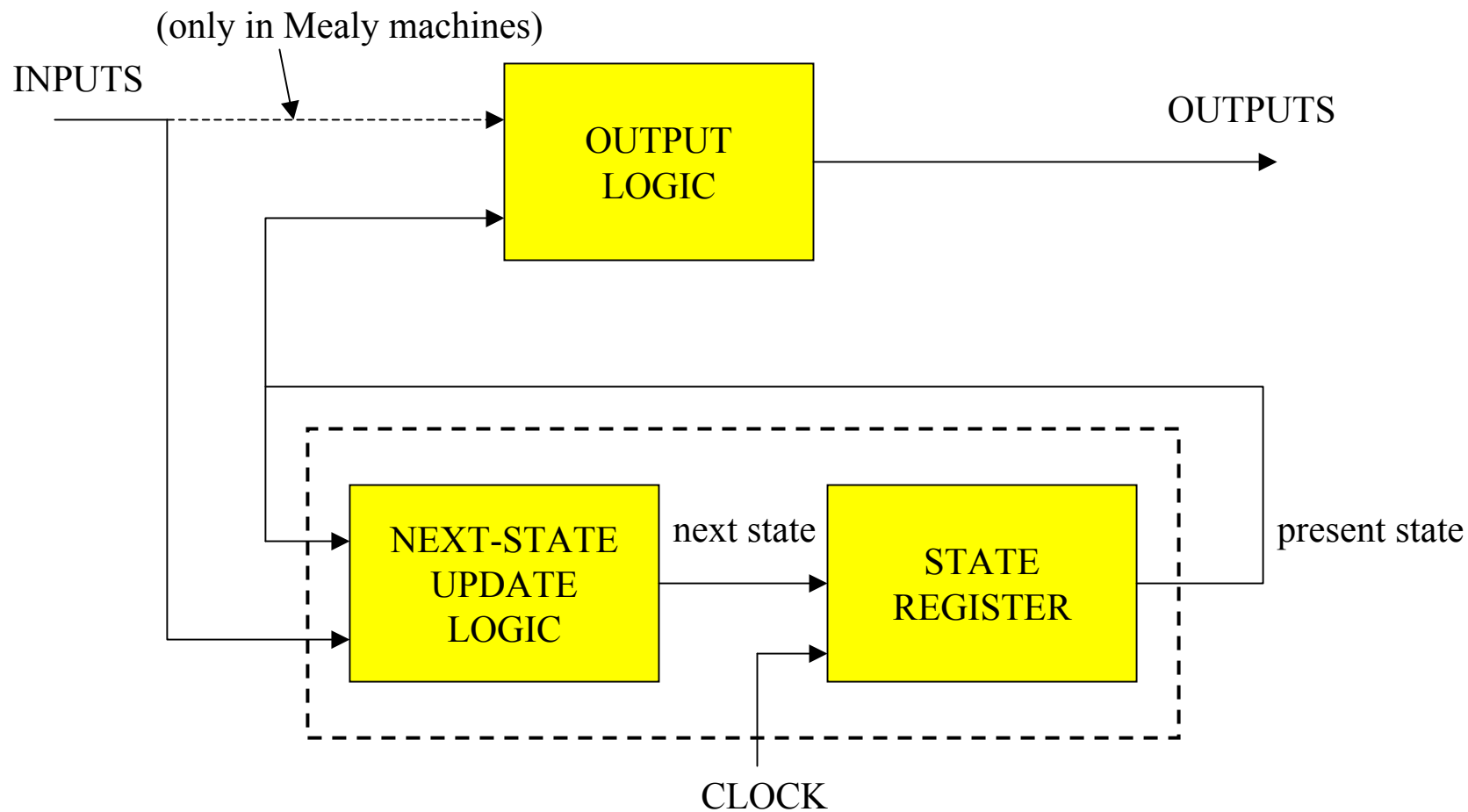
## A complex counter example.

Suppose we want to count in the following sequence:  $\{0, 1, 4, 5, 8, 9, 12, 13, 0, \dots\}$

A keen observer will note that this sequence can be obtained by adding  $1$  if the count is even, and adding  $3$  if the count is odd.

```
signal count : integer range 0 to 15;
signal odd : std_logic;
begin -- begin architecture block
count1 : process(clear, Clk)
    begin
        if clear = '1' then
            count <= 0; odd <= '0';
        elsif Clk = '1' and Clk'event then
            if count_enable = '1' then
                odd <= not odd;
                if odd = '0' then count <= count + 1;
                else count <= count + 3;
            end if; -- if odd
        end if; -- if count_enable
    end if; -- if clear
end process;
```

# VHDL Description of Finite State Machines





FSMs should be described using 2 process blocks

1. - one for the state update (and state storage)
2. - one for the output logic

For some *Moore machines* it might be more readable to combine these *two* process blocks into a *single* process block.


## Define a "state" signal type to hold the state

```
TYPE State_type IS (list of signal values);
```

The name for the new signal type



The list of *values* that the new signal type can have. These can be numerical values, or symbolic state names, for example.



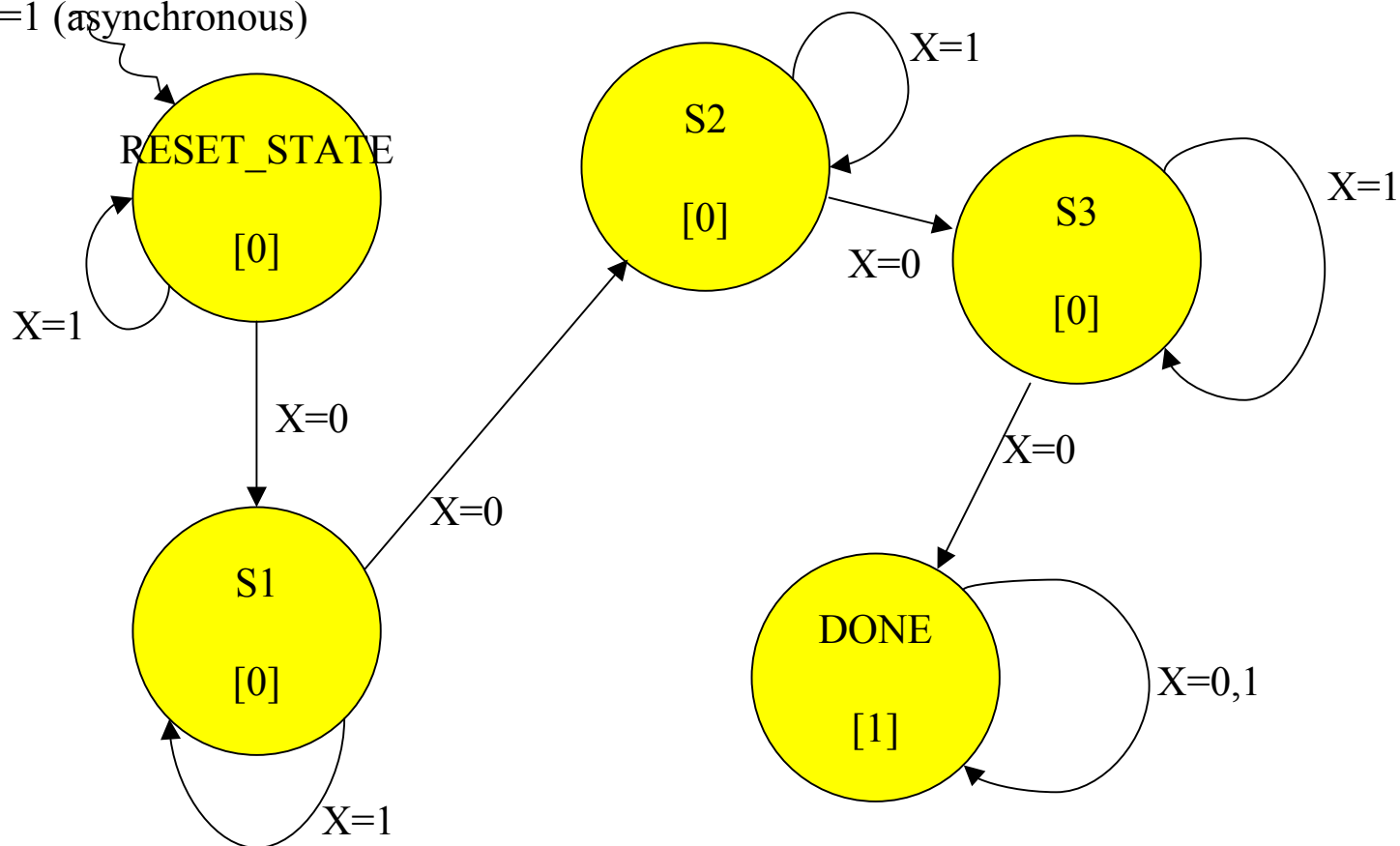
## Example

```
architecture behavioural of FSM is
  TYPE state_signal IS
    (RESET_STATE, S1, S2, S3, DONE);
  SIGNAL state : state_signal;
begin
  ...
```

**You must place the signal TYPE declaration in the declarations area of the architecture.**

# Example of a Moore Machine

RESET=1 (asynchronous)



*Case statements* are very useful for describing *finite state machines*.

## Description using 2 process blocks

```
architecture behavioural of FSM is
TYPE state_signal IS (RESET_STATE, S1, S2, S3, DONE);
SIGNAL state : state_signal;
begin
state_update : process (clk,reset)
begin
if reset = '1' then
state <= RESET_STATE;
elsif clk'EVENT and clk='1' then
case state is
when RESET_STATE =>
if x = '0' then state <= S1; end if;
when S1 =>
if x = '0' then state <= S2; end if;
when S2 =>
if x = '0' then state <= S3; end if;
```

```

        when S3 =>
            if x = '0' then state <= DONE; end if;
        when DONE =>
            state <= DONE;
    end case;
end if; -- if reset
end process;

```

```
output_logic : process(state)
```

```
begin
```

```
case state is
```

```
when RESET_STATE => Z <= '0';
```

```
when S1 => Z <= '0';
```

```
when S2 => Z <= '0';
```

```
when S3 => Z <= '0';
```

```
when DONE => Z <= '1';
```

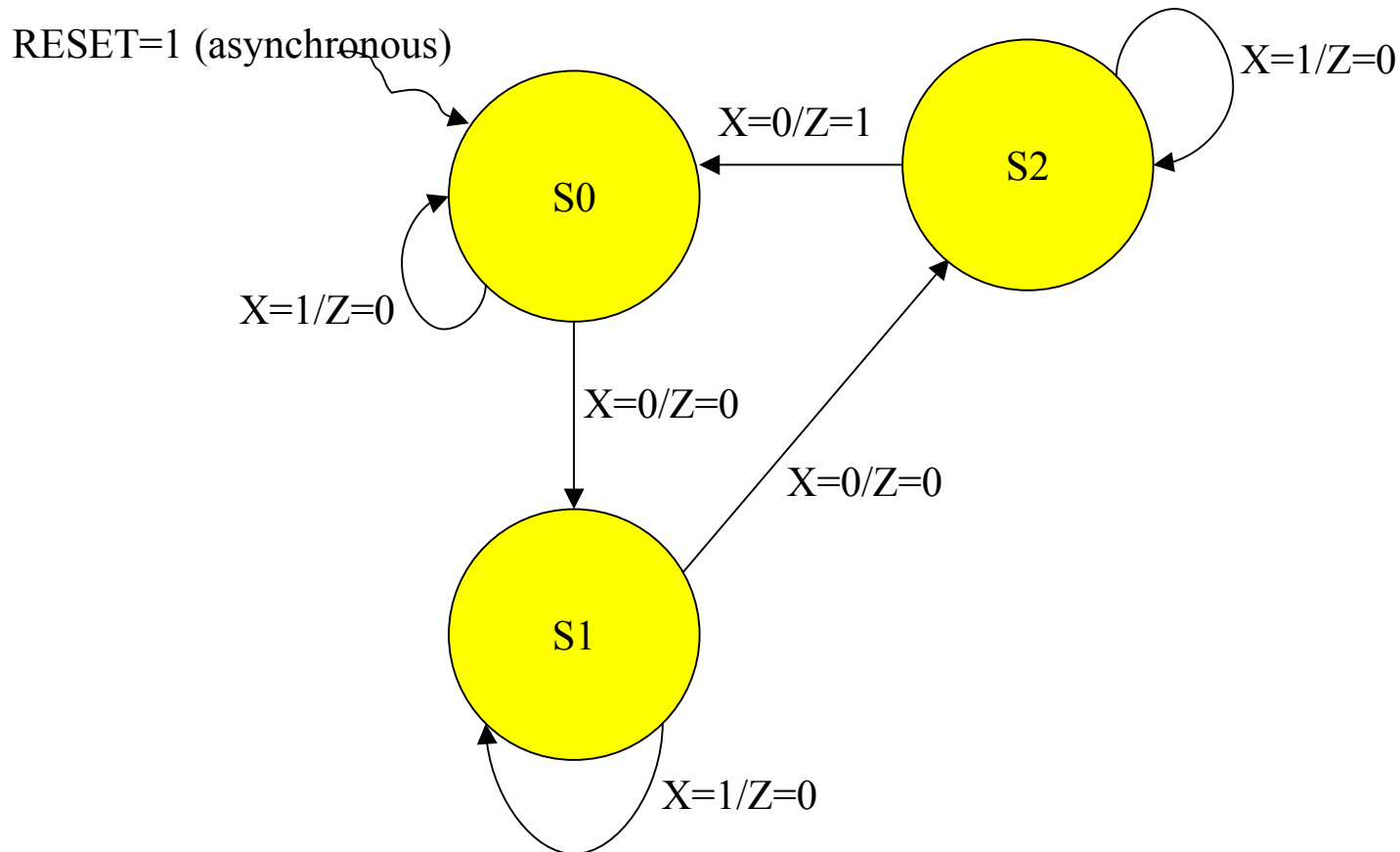
```
end case;
```

```
end process;
```

```
end behavioural;
```

Note: all cases are accounted for,  
so we do not need a **when others**

## Example of a Mealy Machine





# VHDL Description of a Mealy Machine

architecture behavioural of FSM is

```
TYPE state_signal IS (S0, S1, S2);
```

```
SIGNAL state : state_signal;
```

```
begin
```

```
state_update : process (clk,reset)
```

```
begin
```

```
if reset = '1' then
```

```
state <= RESET_STATE;
```

```
elsif clk'EVENT and clk='1' then
```

```
case state is
```

```
when S0 =>
```

```
if x = '0' then state <= S1; end if;
```

```
when S1 =>
```

```
if x = '0' then state <= S2; end if;
```

```
when S2 =>
```

```
if x = '0' then state <= S0; end if;
```

```
end case;
```

```
end if; -- if reset
```

```
end process;
```

## VHDL Description of a Mealy Machine (cont.)

```
output_logic : process(state, X)
  begin
    case state is
      when S0 => Z <= '0';
      when S1 => Z <= '0';
      when S2 => Z <= not X;
    end case;
  end process;
end behavioural;
```

Note that the sensitivity list contains both the signals **state** and **X**

## State Assignment Methods

The Quartus compiler assigns the state bits for each state, based on some state optimization procedure.

The very first state in the list in your state signal type declaration will be assigned the state bit values of *all zeroes*.

**This is useful for *power-on reset* where upon powering on of an FPGA all flipflops get set to zero, and will therefore go to the first state in the list.**

Therefore, make the first state in the state type list be the ***RESET*** state!  
(or some initial state)

## State Assignment Methods

If you don't like the state assignment that Quartus gives you, or you want to use your own assignment (to implement *one-hot state encoding*, for example) you can tell the Quartus program what to use.

This is done by specifying a user-defined **ATTRIBUTE**

```
architecture behaviour of fsm is
  type state_type is (S0, S2, S3);
  attribute ENUM_ENCODING : STRING;
  attribute ENUM_ENCODING of state_type :
    TYPE IS "00 01 11";
  signal states : state_type;
```