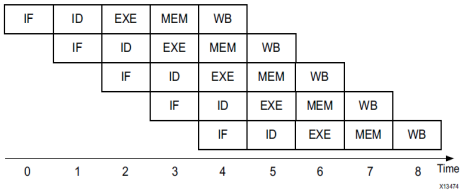
Lecture 3:
Sequential circuits: latches, flip flops, and counters.
VHDL Signals and variables.

Emad Samuel Malki Ebeid

Summary of lecture 2



Processor with Multiple Instruction Execution Units

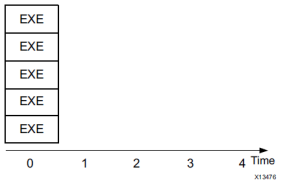
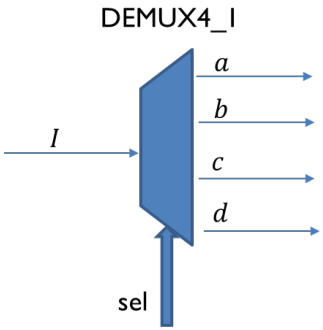
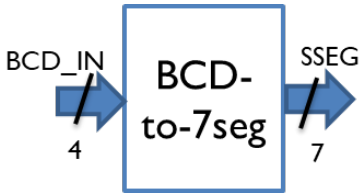
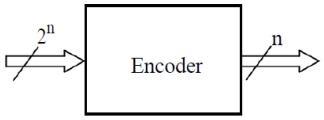


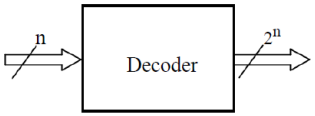
Figure 3-4: **FPGA with Multiple Instruction Execution Units**





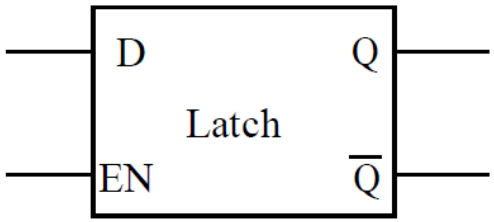
- Concurrent and Sequential statements
  - Conditional signal assignment (when)
  - Selected Signal Assignment (with select)
  - Process statement (if/else)
  - Process statement (case)

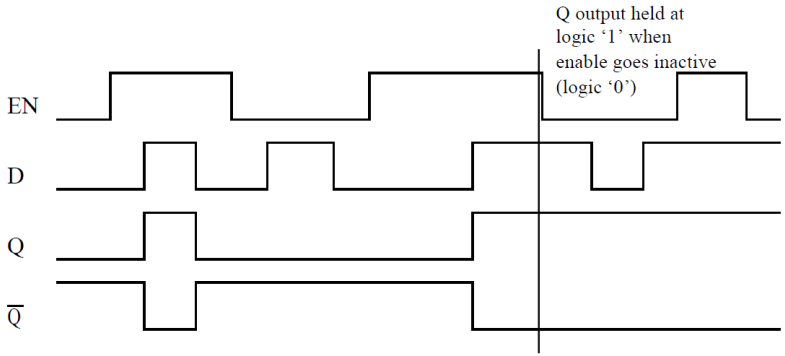




Latches

- A latch is a level sensitive memory cell that is transparent to signals passing from the D input to Q output when enabled, and holds the value of D on Q at the time when it becomes disabled. The Q-bar output signal is always the inverse of the Q output signal,





inputs		outputs	
D	EN	Q_+	$\overline{Q}_+$
–	0	Q_-	$\overline{Q}_-$
0	1	0	1
1	1	1	0

– =don't care

Characteristic table of a D-type transparent latch

Modeling of a D-type transparent latch

- The **if** statement without **else** clause is the simplest implementation of D-type transparent latch.

```
entity d_latch is
  port (d, en: in std_logic;
          q: out std_logic);
  end entity d_latch;

  architecture rtl of d_latch is
  begin
    process (d, en)
    begin
      if (en = '1') then
        q <= d;
      end if;
    end process;
  end architecture rtl;
```

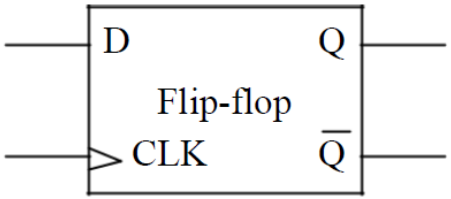
Modeling latches with clear input

- Latches with clear will set the output to '0' whenever the clear signal goes to '1'. Latches with preset will preset the output to '1' instead to '0'.
- Clear and preset inputs to a latch are always asynchronous with the enable.
- The example code models a latch with a clear input:

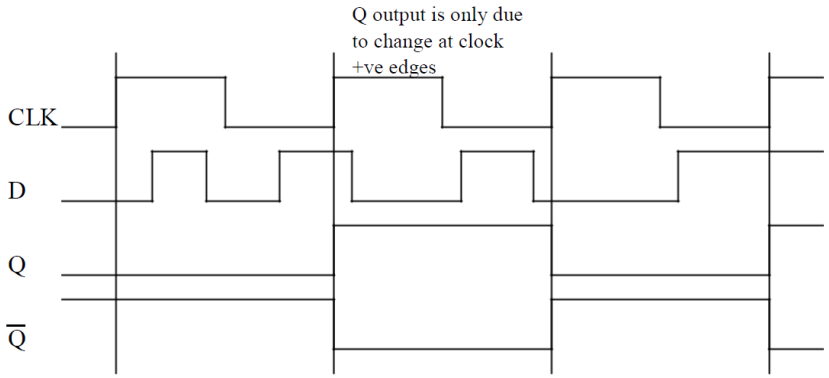
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity d_latch is
  port (d, en, clear: in std_logic;
        q: out std_logic);
end entity d_latch;

architecture rtl of d_latch is
begin
  process (d, en, clear)
  begin
    if (clear = '1') then
      q <= '0';
    elsif (en = '1') then
      q <= d;
    end if;
  end process;
end architecture rtl;
```

D-Type Flip-Flop

- The D-type flip-flop is an edge triggered memory device that transfers a signal's value on its D input, to its Q output, when an active edge transition occurs on its clock input.
- The output value is held until the next active clock edge. The Q-bar output signal is always the inverse of the Q output signal.



inputs		outputs	
D	CLK	Q_+	$\overline{Q}_+$
0	↑	Q_-	$\overline{Q}_-$
1	↑		
–	0	Q_-	$\overline{Q}_-$
–	1	Q_-	$\overline{Q}_-$



Modeling of a D-type flip-flop

- Different models of flip-flops with +ve and –ve edge triggered are shown in architectures below. The first and the second architectures use a VHDL attributes to detect the clk signal edge, while the third and fourth architectures use a function call for the same purpose.

```
entity d_flip_flop is
  port (d, clk: in std_logic;
        q: out std_logic);
end entity d_flip_flop;

architecture first of d_flip_flop
is begin
  process (clk)
  begin
    if (clk'event and clk = '1')
      then q <= d;
    end if;
  end process;
end architecture first;
```

1

```
architecture second of d_flip_flop
is begin
  process (clk)
  begin
    if (clk'event and clk = '0')
      then q <= d;
    end if;
  end process;
end architecture second;
```

2

```
architecture third of d_flip_flop
is begin
  process (clk)
  begin
    if (rising_edge(clk)) then
      q <= d;
    end if;
  end process;
end architecture third;
```

3

```
architecture fourth of d_flip_flop
is begin
  process (clk)
  begin
    if (falling_edge(clk)) then
      q <= d;
    end if;
  end process;
end architecture fourth;
```

4

Which one is more robust in a noisy env.?

'*event* means a change in a digital signal

Modeling flip-flops with reset

- The first architecture models a D-type flip-flop with an asynchronous reset input, this is the commonly used D-type flip-flop, while the second architecture models D-type flip-flop with a synchronous reset as appears from the process sensitivity list and the **if** statements sequence.

```
entity d_flip_flop is
  port (d, clk, rst: in std_logic;
        q: out std_logic);
end entity d_flip_flop;

architecture first of d_flip_flop
is begin
  process (rst, clk)
  begin
    if (rst = '1') then
      q <= '0';
    elsif (rising_edge(clk)) then
      q <= d;
    end if;
  end process;
end architecture first;
```

```
architecture second of d_flip_flop
is begin
  process (clk)
  begin
    if (rising_edge(clk)) then
      if (rst = '1') then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end architecture second;
```

Modeling flip-flop with enable and asynchronous reset

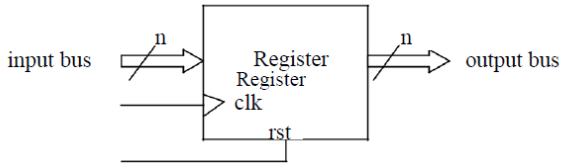
- Flip-flop with enable input and asynchronous reset is modeled to illustrate the addition of the enable signal control, which is synchronous, accordingly with the asynchronous reset.

```
entity d_flip_flop is
  port (d, clk, rst, en: in std_logic;
        q: out std_logic);
end entity d_flip_flop;

architecture rtl of d_flip_flop is
begin
  process (rst, clk)
  begin
    if (rst = '1') then
      q <= '0';
    elsif (rising_edge(clk)) then
      if (en = '1') then
        q <= d;
      end if;
    end if;
  end process;
end architecture rtl;
```

Registers

- A parallel register is simply a bank of D-type flip-flops, its implementation has the same structure as for one flip-flop but with extended inputs and outputs.

- A typical example for a parallel 8 bits register is modeled, based on the flip-flop model shown previously but with inputs and outputs are extended as buses.




```
entity reg_par is
  port (clk, rst: in std_logic;
        reg_in: in std_logic_vector(7 downto 0);
        reg_out: out std_logic_vector(7 downto
0)); end entity reg_par;

architecture rtl of reg_par is
begin
  process (rst, clk)
  begin
    if (rst = '1') then
      reg_out <= "00000000";
    elsif (rising_edge(clk)) then
      reg_out <= reg_in;
    end if;
  end process;
end architecture rtl;
```

Modeling of a parallel 8 bits register with enable

- The same as the previous example but with an added enable signal to provide control over the register, so that the register only register the value at its inputs when enable is active, otherwise remaining the contents unchanged.
- Also, an "**others**" statement in the reset action line is added to fill registers with any size with zeros.

```
entity reg_par is
  port (clk, rst, en: in std_logic;
        reg_in: in std_logic_vector(7 downto 0);
        reg_out: out std_logic_vector(7 downto
0)); end entity reg_par;

architecture rtl of reg_par is
begin
  process (rst, clk)
  begin
    if (rst = '1') then
      reg_out <= (others => '0');
    elsif (rising_edge(clk)) then
      if (en = '1') then
        reg_out <= reg_in;
      end if;
    end if;
  end process;
end architecture rtl;
```

Signal and Variables

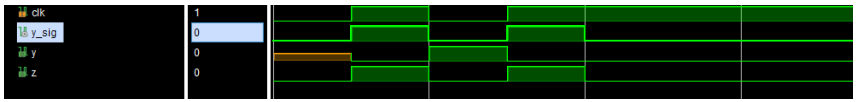
Signals

- Can be used inside or outside processes.
- They take the value depend on if the signal is used in combinational or sequential code
  - In combinational code, signals immediately take the value of their assignment.
  - In sequential code, signals do not immediately take the value of their assignment.
- They are defined in the architecture before the *begin* statement.
- They are assigned using the **<=** assignment symbol.

Variables

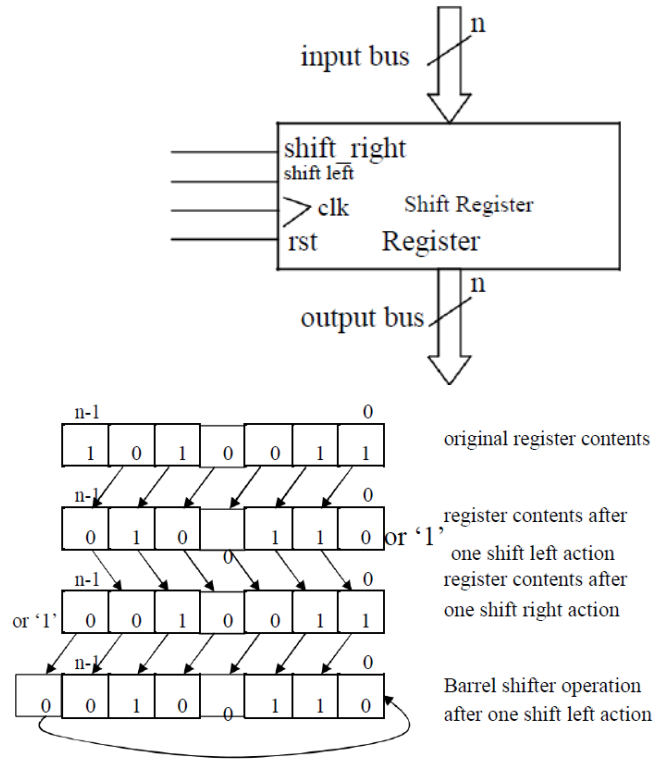
- Only be used inside processes.
- They immediately take the value of their assignment.
- They need to be defined after the keyword *process* but before the keyword *begin*.
- They are assigned using the **:=** assignment symbol.

```
entity signalvsVariable is
    Port ( clk : in STD_LOGIC;
           y : out STD_LOGIC;
           z : out STD_LOGIC);
end signalvsVariable;

architecture Behavioral of signalvsVariable is
    signal y_sig: std_logic;
begin
    process(clk)
        variable z_var: std_logic;
        begin
        if(rising_edge(clk)) then
        y_sig <='1';
        z_var :='1';
        else
        y_sig <='0';
        z_var :='0';
        end if;
        y <= y_sig;
        z <= z_var;
    end process;

end Behavioral;
```



Shift Registers

- Shift registers are registers with added features that enable them to shift their contents in either directions.
- A typical parallel shift register that has added controls to achieve these features; signals shift_left and shift_ right.
- These signals control whether the register is to shift its contents to left or right.
- Depending on the usage of the register, the empty place appears after shifting is filled whether with '0', or '1', or the shifted bit from the other side of the register (barrel shifters)
- Signals are simply a bank of D-type flip-flops, its implementation has the same structure as for one flip-flop but with extended inputs and outputs



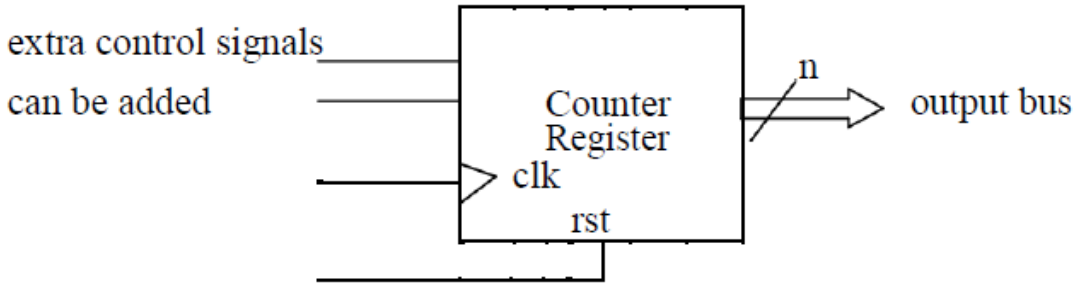
Modeling of a parallel 8 bits shift register

- A typical example for a parallel 8 bits shift register contains controls shift_left, and shift_right signals and combined them together into one bus. That also facilitates the use of **case** statements afterward.
- An intermediate signal definition is made to define a new bus (reg_ temp) to hold the register contents which is necessary because we cannot assign value to output and read it in the same time; (we cannot do something like that: reg_out <= reg_out because reg_out is defined as output), so we cannot read it inside (can not appear on the right-hand side of any assignment). Before the end of the architecture, the value of the intermediate bus is then passed to the output bus.

```
entity reg_par_shift is
  port (clk, rst: in std_logic;
        shift_right, shift_left: in std_logic;
        reg_in: in std_logic_vector(7 downto 0);
        reg_out: out std_logic_vector(7 downto
0)); end entity reg_par_shift;

architecture rtl of reg_par_shift is
  signal shift_control: std_logic_vector(1 downto 0);
  signal reg_temp: std_logic_vector(7 downto
0); begin
  shift_control <= shift_left & shift_right;
  process (rst, clk)
  begin
    if (rst = '1') then
      reg_temp <= (others => '0');
    elsif (rising_edge(clk)) then
      case shift_control is
        when "00" => reg_temp <= reg_in;
        when "01" => reg_temp <= '0' & reg_temp(7 downto
1);
        when "10" => reg_temp <= reg_temp(6 downto 0) &
'0';
        when others => reg_temp <= reg_temp;  --can be
omitted
      end case;
    end if;
  end process;
  reg_out <= reg_temp;
end architecture rtl;
```

Counters

- A register that goes through a predetermined sequence of binary values (states), upon the application of input pulses on one or more inputs, is called a counter. Counters count the number of occurrences of an event, that is, input pulses that occur either randomly or at uniform intervals of time. Counters are used extensively in digital design for all kinds of applications. Apart from general purpose counting, counters can be used as clock dividers and for generating timing control signals



Modeling of a 4 bits binary counter

- The simplest and most common way of modeling a binary counter that can increment or decrement is by adding or subtracting a constant 1 using the "+" or "–" arithmetic operators in assignments residing in a section of code inferring synchronous logic.
- An intermediate signal definition is made to define a new bus (count_temp) to hold the counter state which is necessary because we cannot assign value to output and read it in the same time; (we cannot do something like that: count_out <= count_out + 1 because count _out is defined as output), so we cannot read it inside (can not appear on the right hand side of any assignment), also the "+ 1" operation cannot by applied over std_logic_vector types as they do not implicitly have an equivalent value, instead an unsigned or signed types has to be used.
- Before the end of the architecture the value of the intermediate bus is then passed to the output bus after doing a type conversion to convert the type unsigned to std_logic_vector type used in the output bus definition in the entity declaration.

```
entity binary_count is
  port (clk, rst: in std_logic;
        count_out: out std_logic_vector(3 downto
0)); end entity binary_count;

architecture rtl of binary_count is
  signal count_temp:
begin

  process (rst, clk)
  begin
    if (rst = '1') then
      count_temp <= (others => '0');
    elsif (rising_edge(clk)) then
      count_temp <= count_temp + 1;
    end if;
  end process;
  count_out <= std_logic_vector(count_temp);
end architecture rtl;
```

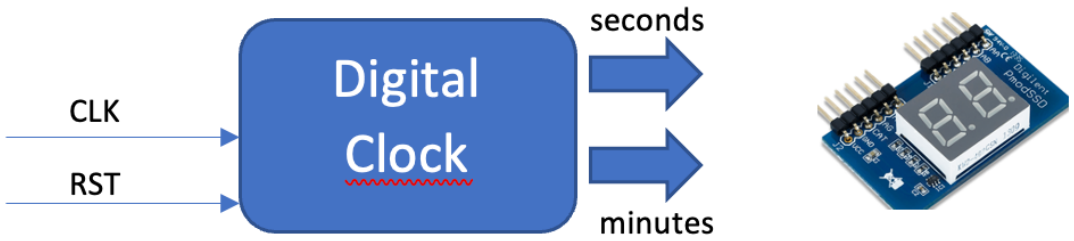
Modeling of a 4 bits binary counter with enable

- The same counter implemented before is used but with adding extra control over the counting event, so this counter only counts when the enable signal is active, otherwise the counter holds its state.

```
entity binary_count is
  port (clk, rst, enable: in std_logic;
        count_out: out std_logic_vector(3 downto
0)); end entity binary_count;

architecture rtl of binary_count is
  signal count_temp: unsigned(3 downto 0);
begin
  process (rst, clk)
  begin
    if (rst = '1') then
      count_temp <= (others => '0');
    elsif (rising_edge(clk)) then
      if (enable = '1') then
        count_temp <= count_temp + 1;
      else
        count_temp <= count_temp; --can be omitted
      end if;
    end if;
  end process;
  count_out <= std_logic_vector(count_temp);
end architecture rtl;
```

Assignment 1

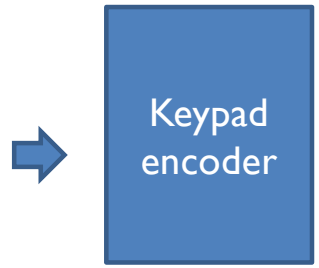
- Design a digital clock circuit to count seconds and minutes using counters.
- The assignment is in itsLearning (Assignments).



- Bonus point, present the output (seconds and minutes) in a format that matches 7-segment displays.

Keypad Encoder

- It is a device that acts as the link between a keypad and a digital device.
- Its primary function is to provide a digital output equal to the key is pressed on the keypad.



Key pressed (e.g., 0001)