

Lecture 3 – Sequential Logic Circuits

Reference: Roth/John Text: Chapter 2

VHDL “Process” Construct

- Allows conventional programming language structures to describe circuit behavior – especially sequential behavior
 - Process statements are executed in sequence
 - Process statements are executed once at start of simulation
 - Process is suspended at “end process” until an event occurs on a signal in the “sensitivity list”

```
[label:] process (sensitivity list)
    declarations
begin
    sequential statements
end process;
```

Modeling combinational logic as a process

-- All signals referenced in process must be in the sensitivity list.

entity And_Good is

port (a, b: in std_logic; c: out std_logic);

end And_Good;

architecture Synthesis_Good of And_Good is

begin

process (a,b) -- gate sensitive to events on signals a and/or b

begin

c <= a and b; -- c updated (after delay on a or b “events”

end process;

end;

-- Above process is equivalent to simple signal assignment statement:

-- c <= a and b;

Bad example of combinational logic

-- This example produces unexpected results.

entity And_Bad is

port (a, b: in std_logic; c: out std_logic);

end And_Bad;

architecture Synthesis_Bad of And_Bad is

begin

process (a) -- sensitivity list should be (a, b)

begin

c <= a and b; -- will not react to changes in b

end process;

end Synthesis_Bad;

-- synthesis may generate a flip flop, triggered by signal a

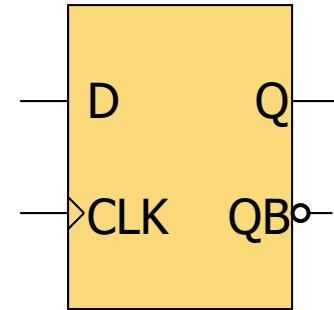
Modeling sequential behavior

-- Edge-triggered flip flop/register

```
entity DFF is
  port (D,CLK: in bit;
        Q: out bit);
end DFF;
```

```
architecture behave of DFF is
```

```
begin
  process(clk) -- "process sensitivity list"
  begin
    if (clk'event and clk='1') then -- rising edge of clk
      Q <= D; -- optional "after x" for delay
      QB <= not D;
    end if;
  end process;
end;
```



- **clk'event** is an "attribute" of signal clk (signals have several attributes)
 - **clk'event = TRUE** if an event has occurred on clk at the current simulation time
 - **FALSE** if no event on clk at the current simulation time
 - **clk'stable** is a complementary attribute (TRUE of no event at this time)

Edge-triggered flip-flop

- ❑ Special functions in package `std_logic_1164` for `std_logic` types
 - `rising_edge(clk)` = TRUE for 0->1, L->H and several other “rising-edge” conditions
 - `falling_edge(clk)` = TRUE for 1->0, H->L and several other “falling-edge” conditions

Example:

```
signal clk: std_logic;
```

```
begin
```

```
    process (clk)
```

```
        -- trigger process on clk event
```

```
    begin
```

```
        if rising_edge(clk) then
```

```
            -- detect rising edge of clk
```

```
            Q <= D ;
```

```
            -- Q and QB change on rising edge
```

```
            QB <= not D;
```

```
        end if;
```

```
    end process;
```

Common error in processes

- ❑ Process statements are evaluated only at time instant T , at which an event occurs on a signal in the sensitivity list
 - Statements in the process use signal values that exist at time T .
 - Signal assignment statements “schedule” future events.

Example:

```
process (clk)                -- trigger process on clk event
begin
    if rising_edge(clk) then -- detect rising edge of clk
        Q  <= D ;           -- Q and QB change  $\delta$  time after rising edge
        QB <= not Q;       -- Timing error here!!
    end if;                -- Desired QB appears one clock period late!
end process;               -- Should be: QB <= not D;
```

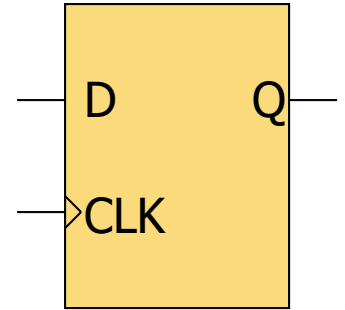
As written above, if clk edge occurs at time T :

Q will change at time $T+\delta$, to $D(T)$

QB will change at time $T+\delta$, to “not $Q(T)$ ” – using $Q(T)$ rather than new $Q(T+\delta)$

Alternative to sensitivity list

```
process -- no "sensitivity list"
begin
    wait on clk; -- suspend process until event on clk
    if (clk='1') then
        Q <= D after 1 ns;
    end if;
end process;
```



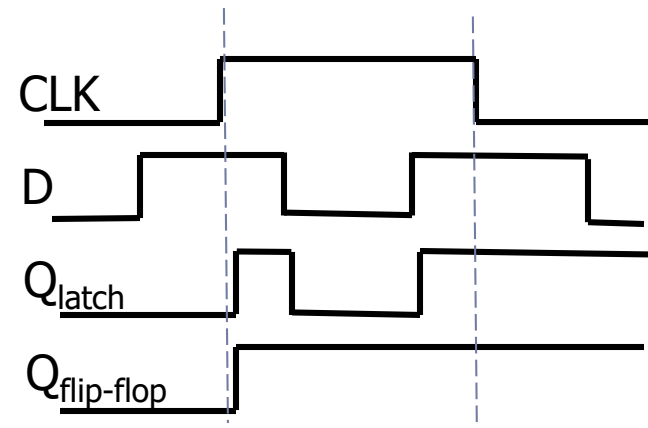
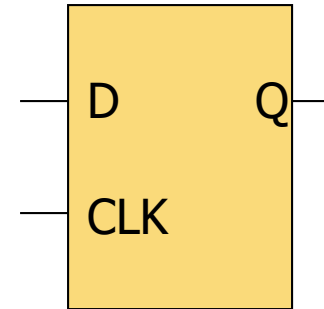
- ▶ **BUT - sensitivity list is preferred for sequential circuits!**
- ▶ Other “wait” formats: wait until (clk'event and clk='1')
 wait for 20 ns;
- ▶ This format does not allow for asynchronous controls
- ▶ Cannot have both sensitivity list and wait statement
- ▶ **Process executes endlessly if neither sensitivity list nor wait statement provided!**

Level-Sensitive D latch vs. D flip-flop

```
entity Dlatch is
  port (D,CLK: in bit;
        Q: out bit);
end Dlatch;
```

```
architecture behave of Dlatch is
begin
```

```
  process(D, clk)
  begin
    if (clk='1') then
      Q <= D after 1 ns;
    end if;
  end process;
end;
```



Q_{latch} can change when CLK becomes '1' and/or when D changes while CLK='1' (rather than changing only at a clock edge)

RTL “register” model (not gate-level)

entity Reg8 is

```
port (D: in std_logic_vector(0 to 7);  
      Q: out std_logic_vector(0 to 7);  
      LD: in std_logic);
```

end Reg8;

architecture behave of Reg8 is

begin

```
process(LD)
```

```
begin
```

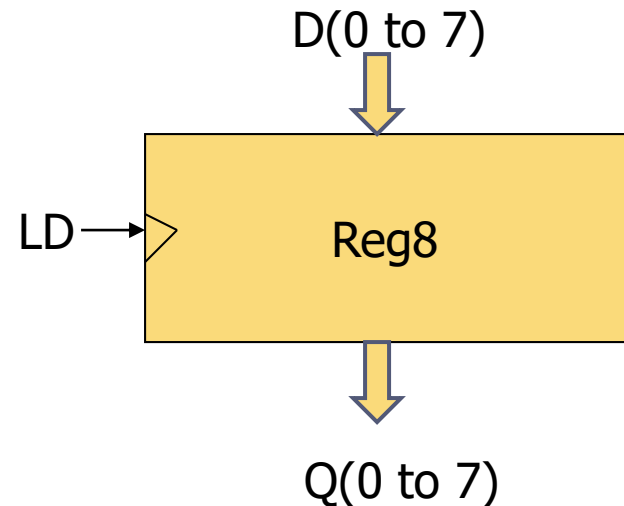
```
if rising_edge(LD) then
```

```
    Q <= D;
```

```
end if;
```

```
end process;
```

```
end;
```



D and Q can be any abstract data type

RTL “register” with clock enable

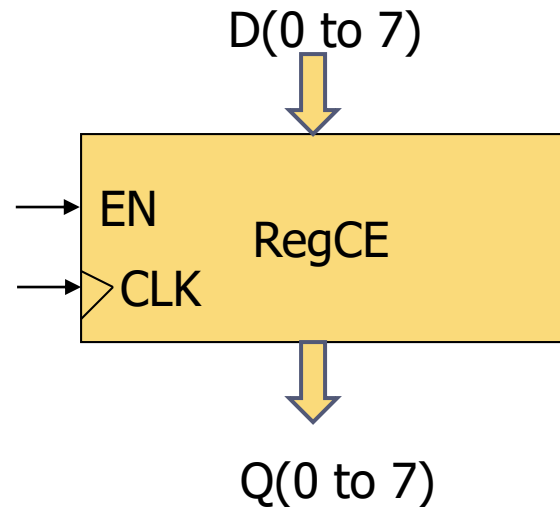
- Connect all system registers to a common clock
- Select specific registers to be loaded

entity RegCE is

```
port (D: in std_logic_vector(0 to 7);
      Q: out std_logic_vector(0 to 7);
      EN: in std_logic;           --clock enable
      CLK: in std_logic);
end RegCE;
```

architecture behave of RegCE is
begin

```
  process(CLK)
  begin
    if rising_edge(CLK) then
      if EN = '1' then
        Q <= D;  --load only if EN=1 at the clock transition
      end if;
    end if;
  end process;
end;
```



Synchronous vs asynchronous inputs

- ❑ Synchronous inputs are synchronized to the clock.
- ❑ Asynchronous inputs are not and cause immediate change.
 - Asynchronous inputs normally have precedence over sync. inputs

```
process (clock, asynchronous_signals )
begin
    if (boolean_expression) then
        asynchronous_signal_assignments
    elsif (boolean_expression) then
        asynchronous_signal_assignments
    elsif (clock'event and clock = constant) then
        synchronous_signal_assignments
    end if ;
end process;
```

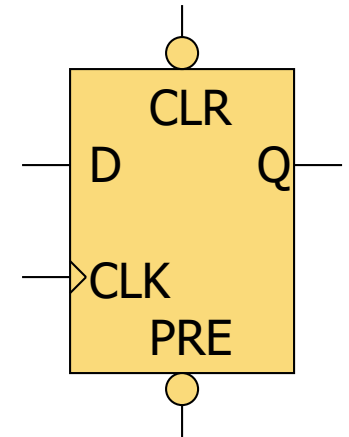
Synchronous vs. Asynchronous Flip-Flop Inputs

entity DFF is

```
port (D,CLK: in std_logic;    --D is a sync input
      PRE,CLR: in std_logic;  --PRE/CLR are async inputs
      Q: out std_logic);
end DFF;
```

architecture behave of DFF is

```
begin
  process(clk, PRE, CLR)
  begin
    if (CLR='0') then          -- async CLR has precedence
      Q <= '0';
    elsif (PRE='0') then      -- then async PRE has precedence
      Q <= '1';
    elsif rising_edge(clk) then -- sync operation only if CLR=PRE='1'
      Q <= D;
    end if;
  end process;
end;
```



What happens if CLR = PRE = 0 ??

Sequential Constructs: if-then-else

General format:

```
if (condition) then
    do stuff
elsif (condition) then
    do more stuff
else
    do other stuff
end if;
```

Example:

```
if (S = "00") then
    Z <= A;
elsif (S = "11") then
    Z <= B;
else
    Z <= C;
end if;
```

elsif and else clauses are optional, BUT incompletely specified if-then-else (no else) implies memory element

Sequential Constructs: case-when

General format:

```
case expression is
    when value =>
        do stuff
    when value =>
        do more stuff
    when others =>
        do other stuff
end case;
```

Example:

```
case S is
    when "00" =>
        Z <= A;
    when "11" =>
        Z <= B;
    when others =>
        Z <= C;
end case;
```

Sequential Constructs: for loop

General format:

```
[label:] for identifier in range loop  
    do a bunch of junk  
end loop [label];
```

Example:

```
init: for k in N-1 downto 0 loop  
    Q(k) <= '0';  
end loop init;
```

Note: variable k is “implied” in the for-loop and does not need to be declared

Sequential Constructs: while loop

General format:

```
[label:] while condition loop
    do some stuff
end loop [label];
```

Example:

```
init: while (k > 0) loop
    Q(k) <= '0'
    k := k - 1;
end loop init;
```

Note: Variable k must be declared as a process “variable”, between sensitivity list and begin, with format:

- variable variable_name : type := initial_value;
- variable k: integer := N-1;

Verilog: Abstract Modeling with Cyclic Behaviors

- ❑ Abstract
 - Do not use hardware to specify values
- ❑ Cyclic behaviors
 - Verilog keyword **always**, followed by an **event-control expression**
 - ▶ e.g. **always @ (posedge clk)**
 - Execute **procedural statements** to generate values of variables
 - Assign values to register variables to describe the behavior of hardware
 - Do not expire after the last procedural statement
 - ▶ re-execute after executing the last procedural statement executes (subject to timing controls)
 - Model both level-sensitive and edge-sensitive behaviors
 - ▶ Depending on the event-control expression
 - Synthesis tool selects the hardware

Cyclic Behavior Ex: DFF with Sync. Set/Reset

- ❑ Edge-triggered
- ❑ Synchronous set/reset
 - Signals **set/reset** not in the event-control expression
 - No influence until posedge clk
- ❑ Non-blocking assignments (`<=`) within CB

```
module df_behav (q, q_bar, data, set_n, reset_n, clk);
  input          data, set_n, clk, reset_n;
  output        q, q_bar;
  reg           q;

  assign q_bar = ~ q;

  always @ (posedge clk) // Flip-flop with synchronous set/reset
  begin
    if (reset_n == 0) q <= 0;    // <= is the nonblocking assignment operator
    else if (set_n == 0) q <= 1;
    else q <= data;
  end
endmodule
```

Cyclic Behavior Example: DFF (cont.)

- ❑ A variable that is assigned values **by a procedural assignment operator** in a single-pass (i.e., init) or cyclic behavior (i.e. always) must be declared as a **register type variable** to store information during simulation
 - Not necessarily imply a hardware register after synthesis
 - Such as the variable q
 - ▶ Happen to be a DFF in the example
- ❑ Procedural statement is executed sequentially
- ❑ Event-control expression is re-evaluated after all procedural statements are executed

Example: DFF with Asynchronous Set/Reset

- Signals set/reset are in the event-control expression

```
module asynch_df_behav (q, q_bar, data, set_n, clk, reset_n );
    input          data, set_n, reset_n, clk;
    output         q, q_bar;
    reg            q;

    assign q_bar = ~q;

    always @ (negedge set_n or negedge reset_n or posedge clk)
    begin
        if (reset_n == 0) q <= 0;
        else if (set_n == 0) q <= 1;
        else q <= data;          // synchronized activity
    end
endmodule
```

Ex: DFF with Asynchronous Set/Reset (Cont.)

- ❑ Good practice to place the synchronous signal (i.e. clock) of the asynchronous behavior in the last conditional clause in the event control expression
 - Made easy to identify the synchronous signal
 - ▶ Either by human for readability or by synthesis tool
 - Made easy to infer the need of a flip-flop to hold the value between two active edges of the synchronous signal
- ❑ Verilog allows mixture of level-sensitive and edge-qualified variables in the same event-control expression
 - BUT, synthesis tools do not support such models of behavior
 - Hence, **event-control expression must be**
 - ▶ **Entirely level-sensitive, or**
 - ▶ **Entirely edge-sensitive**

Example: Transparent-Latch Using Cyclic Behavior

```
module t_latch (q_out, enable, data);  
  output q_out;  
  input enable, data;  
  reg q_out;
```

```
always @ (enable or data)
```

```
  begin
```

```
    if (enable) q_out = data;
```

```
    // Note: no "else" assignment for q_out
```

```
    // hence, the value of q_out is implied to be kept, i.e. latched
```

```
  end
```

```
endmodule
```

