Structural & behavioral modeling styles

IAS0600 Digital Systems Design with VHDL

Natalia Cherezova, Peeter Ellervee

Modeling styles

- Dataflow
  - Concurrent signal assignments
- Structural
  - Interconnected components
- Behavioral
  - Sequential statements (in a process)

Structural design

- A design entity in VHDL can be used as an individual module or as a component definition
- Large systems can be broken into smaller pieces
  - Smaller pieces are easier to design and easier to debug
- Simple components can be reused
- Can create hierarchical designs

Direct component instantiation

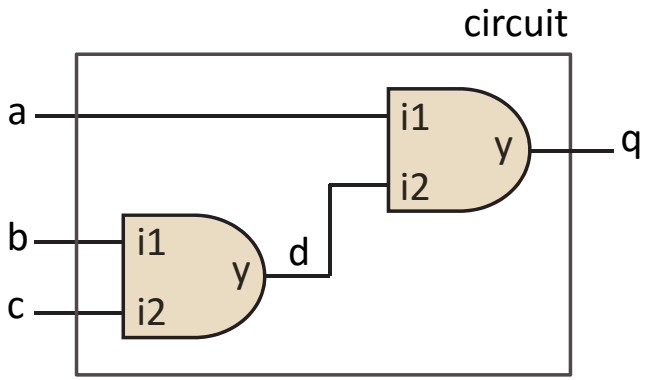
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity circuit is
  Port ( a, b, c : in STD_LOGIC;
         q : out STD_LOGIC);
end circuit;

architecture a_struct of circuit is
  signal d : STD_LOGIC;
begin

  U1: entity work.and_gate
        port map (i1 => b, i2 => c, y => d);
  U2: entity work.and_gate
        port map (i1 => a, i2 => d, y => q);

end a_struct;
```



Connect two components by mapping signal *d* to the input of one component and to the output of the other component.

Instance label

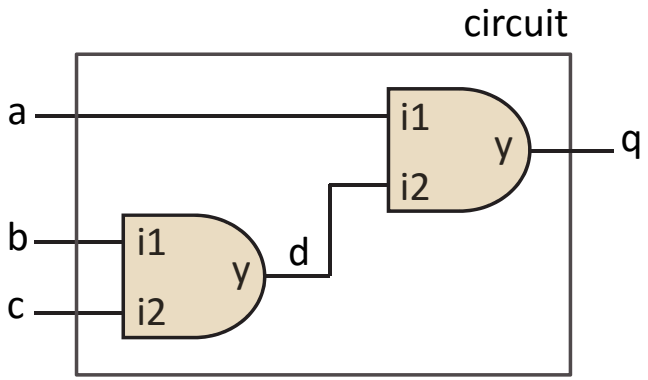
Indirect component instantiation

Preferable way for design sources

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity circuit is
  Port ( a, b, c : in STD_LOGIC;
         q : out STD_LOGIC);
end circuit;

architecture a_struct of circuit is
  component and_gate is
    port (i1, i2 : in std_logic;
          y : out std_logic);
  end component;
  signal d : STD_LOGIC;
begin
  U1: and_gate port map (i1 => b, i2 => c, y => d);
  U2: and_gate port map (i1 => a, i2 => d, y => q);
end a_struct;
```



Component declaration

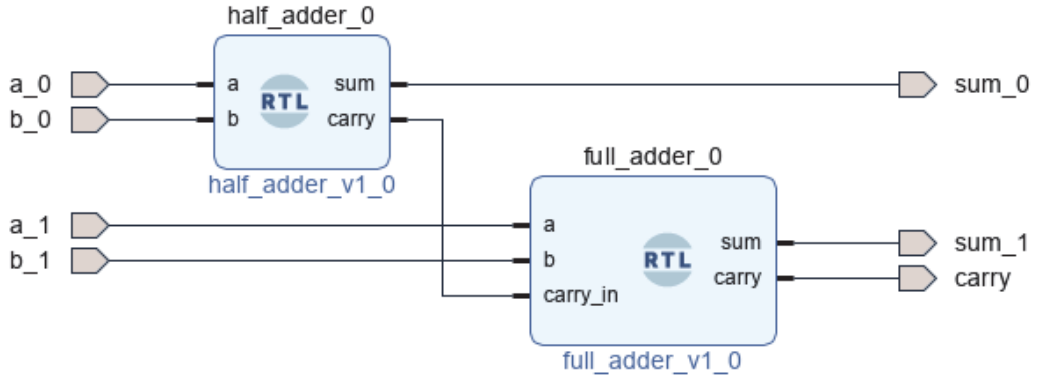
Component instantiation

Port mapping

- Named association
  - Association is stated explicitly
  - Order does not matter
  - `port => local_signal`
  - `U1: and_gate port map (i1 => b, i2 => c, y => d);`
- Positional association
  - Only the mapped signals are listed
  - Order is important: signals should be ordered exactly as in component declaration
  - More compact but also more prone to errors
  - `U1: and_gate port map (b, c, d);`

Vivado Block Design

- Allows creating designs using IP (Intellectual Property) blocks or RTL modules
- Visual representation of structural modeling
- Finished block diagram should be translated into VHDL description for synthesis and simulation



Modeling styles

- Dataflow
  - Concurrent signal assignments
- Structural
  - Interconnected components
- Behavioral
  - Sequential statements (in a process)

Processes

- A process can be viewed as a subcircuit within an architecture
- Within the process, each statement is sequential
- Processes allow you to describe the system or a function in an algorithmic way
- The process statement by itself is concurrent. Two processes represent two functional blocks that are executed concurrently

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

Sensitivity list vs. wait statement

- The sensitivity list is mandatory, except there is a WAIT statement in the process
- Sensitivity list specifies which signals will trigger the process
- Whenever a signal in a sensitivity list changes, the process is evaluated

Sensitivity list

```
process (a, b)
begin
  if (a = '1' and b = '1') then
    y <= '1';
  else
    y <= '0';
  end if;
end process;
```

```
process
begin
  if (a = '1' and b = '1') then
    y <= '1';
  else
    y <= '0';
  end if;
  wait on a, b;
end process;
```

Wait statement

The WAIT statement

- **`wait until`** `<condition>;`
  - Causes process to hold until the condition is fulfilled
- **`wait on`** `<sensitivity_list>;`
  - Causes process to hold until any listed signal changes
- **`wait for`** `<time_expression>;`
  - Hold process for a specified period
  - Not synthesizable, for simulations only
- **`wait`**`;`
  - Suspend the process indefinitely

Signals update in a process

- Signals are evaluated when the process is suspended
- Process with the sensitivity list is suspended at the end
- Process with wait statements is additionally suspended at every wait statement

```
process (a, b)
begin
  y <= a OR b;
  y <= a NOR b;
  y <= a AND b;
end process;
```

y always equals (a AND b)

Process is suspended here

```
process
begin
  y <= a OR b;
  wait for 10 ns;

  y <= a NOR b;
  wait for 10 ns;

  y <= a AND b;
  wait for 10 ns;
end process;
```

Process is suspended here

And here

And also here

Signals are not updated immediately

```
process (a, b)
begin
  y <= a OR b;
  g <= y;

  y <= a NOR b;
  h <= y;

  y <= a AND b;
  f <= y;
end process;
```

The initial value of **y** is '0'.
The process is triggered: **a** = '1' and **b** = '1'.
What are the values of **g**, **h**, **f**, and **y** at the end of the process?

g, **h**, and **f** are assigned the previous value of **y**.
y takes the value of the last assignment.
Therefore, g = '0', h = '0', f = '0', y = '1'

Variables

- Variable can be used only in a sequential code (e.g., inside a process)
- Should be declared in a process, and is local to that process
- Updated immediately
- Assignment operator for variables is :=

```
process (a, b)
  variable y : std_logic;
begin
  y := a OR b;
  g <= y;

  y := a NOR b;
  h <= y;

  y := a AND b;
  f <= y;
end process;
```

If a = '1' and b = '1', then at the end of the process, g = '1', h = '0', and f = '1'.

Sequential statements

Concurrent statements cannot be used in a process

- IF statement

```
if <condition> then
  <assignments>;
elsif <condition> then
  <assignments>;
else
  <assignments>;
end if;
```

- CASE statement

```
case <expression> is
  when <value> => <assignments>;
  when <value> => <assignments>;
  ...
  when others => <assignments>;
end case;
```

Can be skipped when combinations are covered

Types of processes

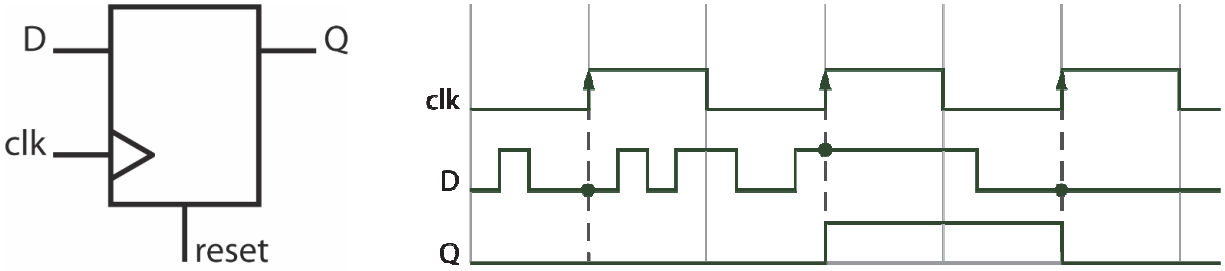
- Combinational process
  - Models combinational logic
  - Sensitive to all inputs used in the combinational logic
  - The sensitivity list includes **all** signals that are **read** in the process
- Sequential process
  - Models sequential logic
  - Sensitive to a clock and control signals (e.g., reset)
  - Sensitivity list does not include all inputs, **only** the clock and control signals

Sequential logic

- The output of sequential logic depends not only on the current input values but also on the previous input history
- This means that sequential logic possesses an internal state (memory) that changes in accordance with the applied sequence of input values

Flip-flops

- The basic element of sequential logic is a flip-flop
- The most commonly used type is a D-type flip-flop
- Flip-flop is an edge-sensitive element, meaning it is active on clock transition from LOW to HIGH



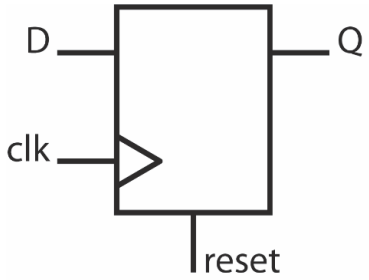
D-type flip-flop in VHDL

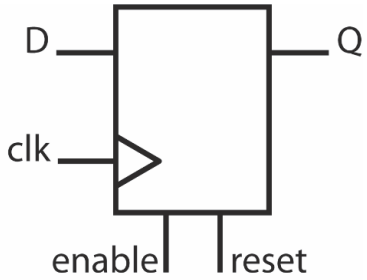
```
process (clock, reset)
begin
  if reset = '1' then
    Q <= '0';
  elsif clock'event and clock = '1' then
    Q <= D;
  end if;
end process;
```

Asynchronous reset

```
process (clock)
begin
  if rising_edge(clock) then
    if reset = '1' then
      Q <= '0';
    elsif enable = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

Synchronous reset and enable

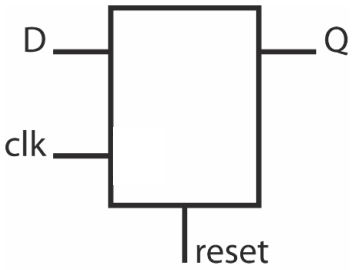


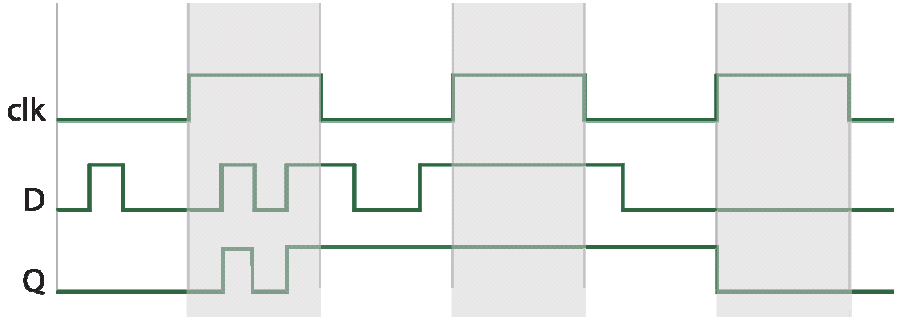


Enable signal allows us to keep the flip-flop value unchanged for several clock cycles

Latches

- Unlike a flip-flop, a latch is a level-sensitive element
- It is active during the whole time when the clock is HIGH
- When the clock is HIGH, change in the input is immediately propagated to the output



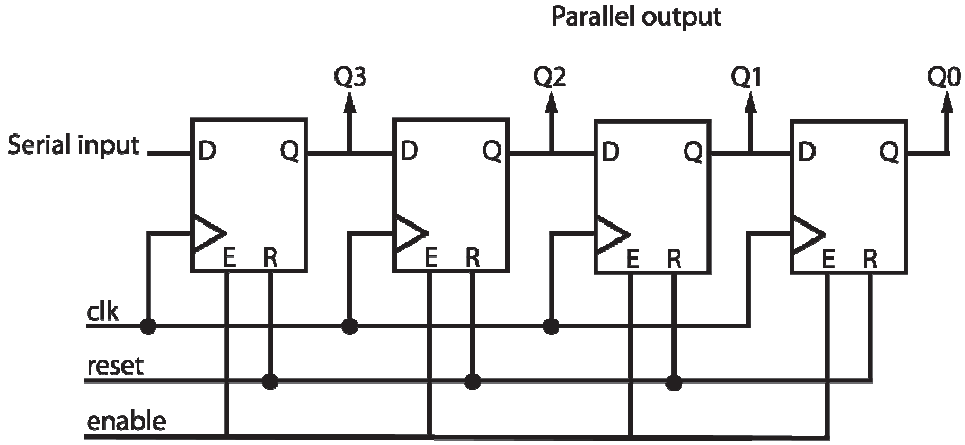


Shift register

- Shift register consists of serially connected DFFs
- Its value is shifted one position to the right when enable signal is HIGH (in the example)

```
process (clock)
begin
  if clock'event and clock = '1' then
    if reset = '1' then
      Q <= (others => '0');
    elsif enable = '1' then
      Q(3) <= D;
      Q(2 downto 0) <= Q(3 downto 1);
    end if;
  end if;
end process;
```

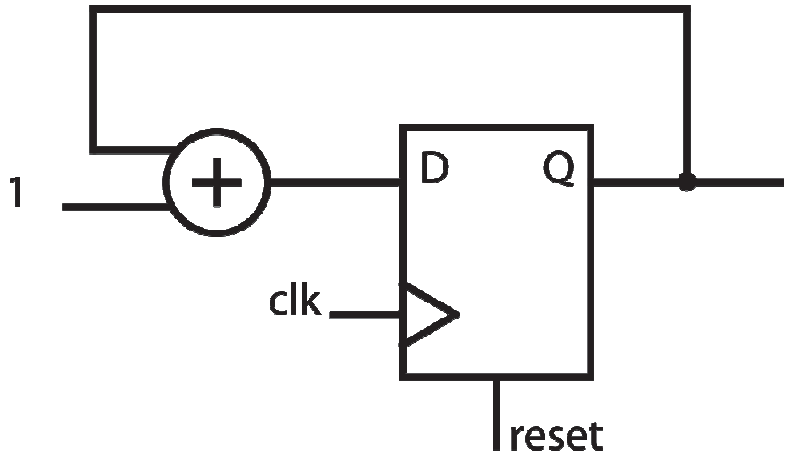
A convenient way to assign the whole vector to '0'



Counter

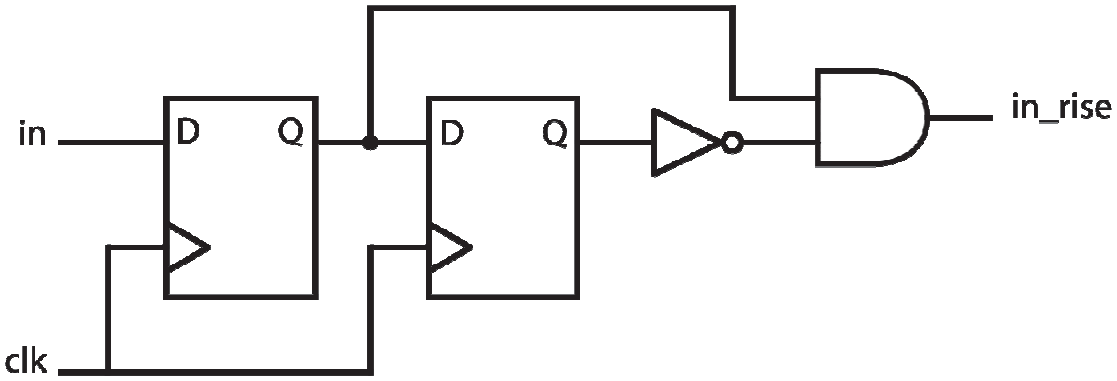
- Counter is a register that increments or decrements when enabled
- Its next state is calculated as an addition of 1 to its current state

```
process (clock)
begin
  if clock'event and clock = '1' then
    if reset = '1' then
      Q <= (others => '0');
    elsif enable = '1' then
      Q <= std_logic_vector(unsigned(Q) + 1);
    end if;
  end if;
end process;
```



Detecting the edge of data input

- `rising_edge()` function can be used **only** with the **clock** signal
- In order to detect the rising or falling edge of the input, sample the input's value every clock cycle



Synchronous design

- All sequential logic elements in lab 3 should be synchronized on the 100 MHz clock
- Do not use generated signals as a clock, use enable signals