

SYSTEM MODELLING AND SYNTHESIS WITH HDL

DTEK0078

2023 Lecture 6



Our research website: <https://tiers.utu.fi/>

IMPORTANT

- In a process, a signal is never updated immediately on assignment
- In synthesizable code, multiple assignments to the same signal in a process behave as if only the last assignment were effective



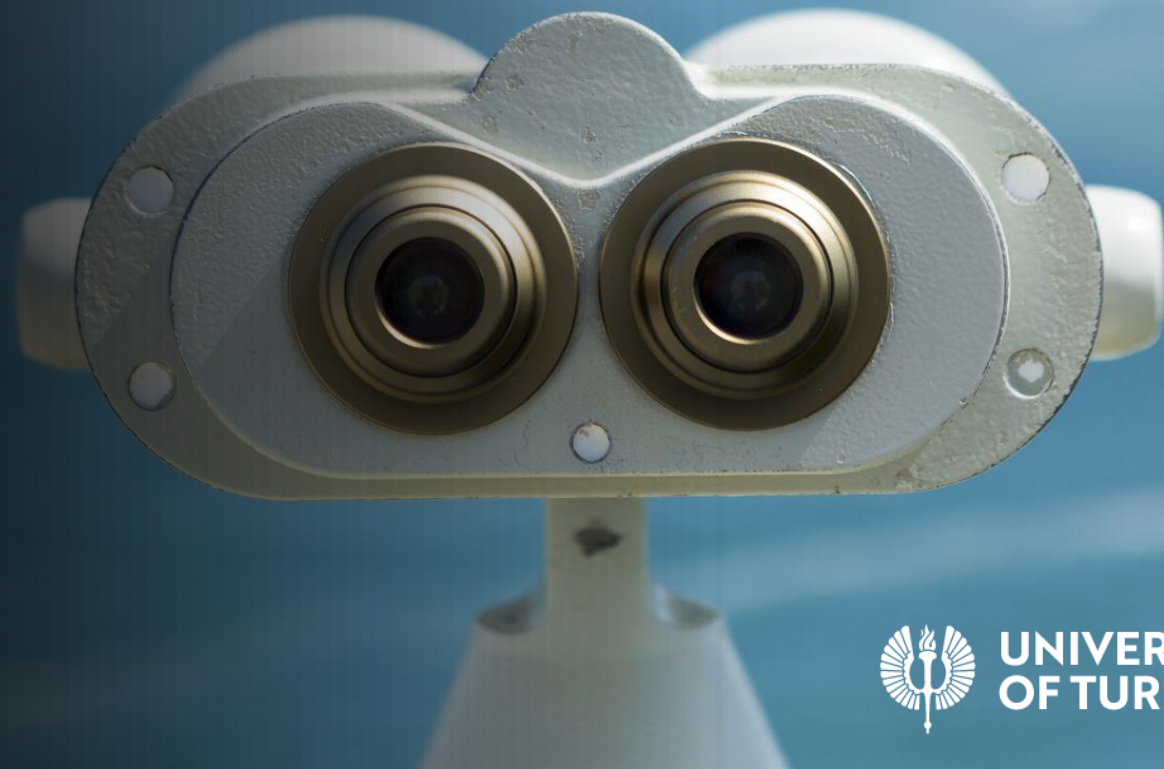
Facts: Signals and Processes

- Signal values do not change while a process is running
- If a process makes multiple assignments to a signal, only the last assignment before the process suspends is effective

```
process
begin
    integer_signal <= 1;
    integer_signal <= 2;
    integer_signal <= 3;
    if some_condition then
        integer_signal <= 4;
    end if;
    wait for 1 ms;
end process;
```



Simulation



input

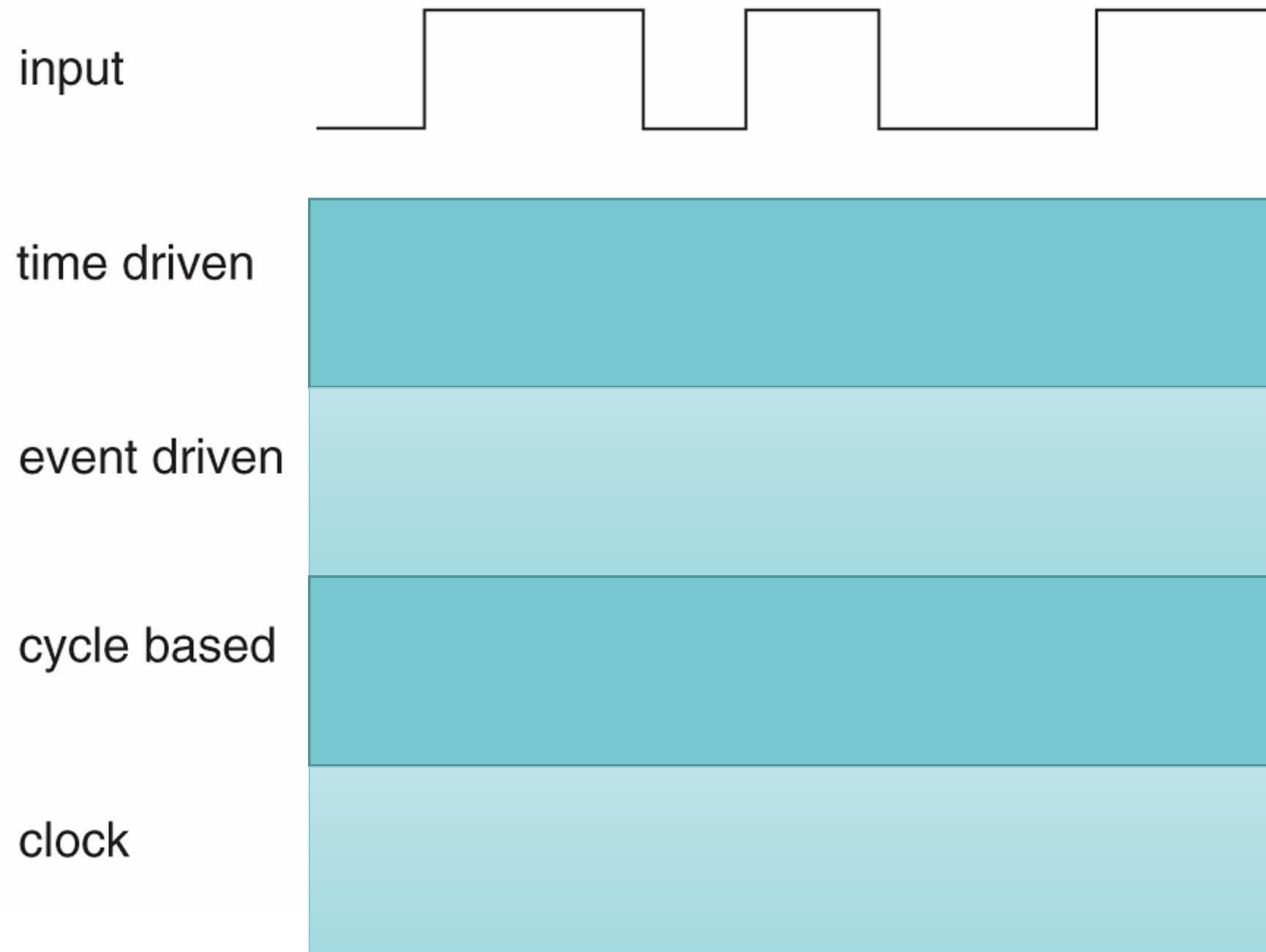
time driven

event driven

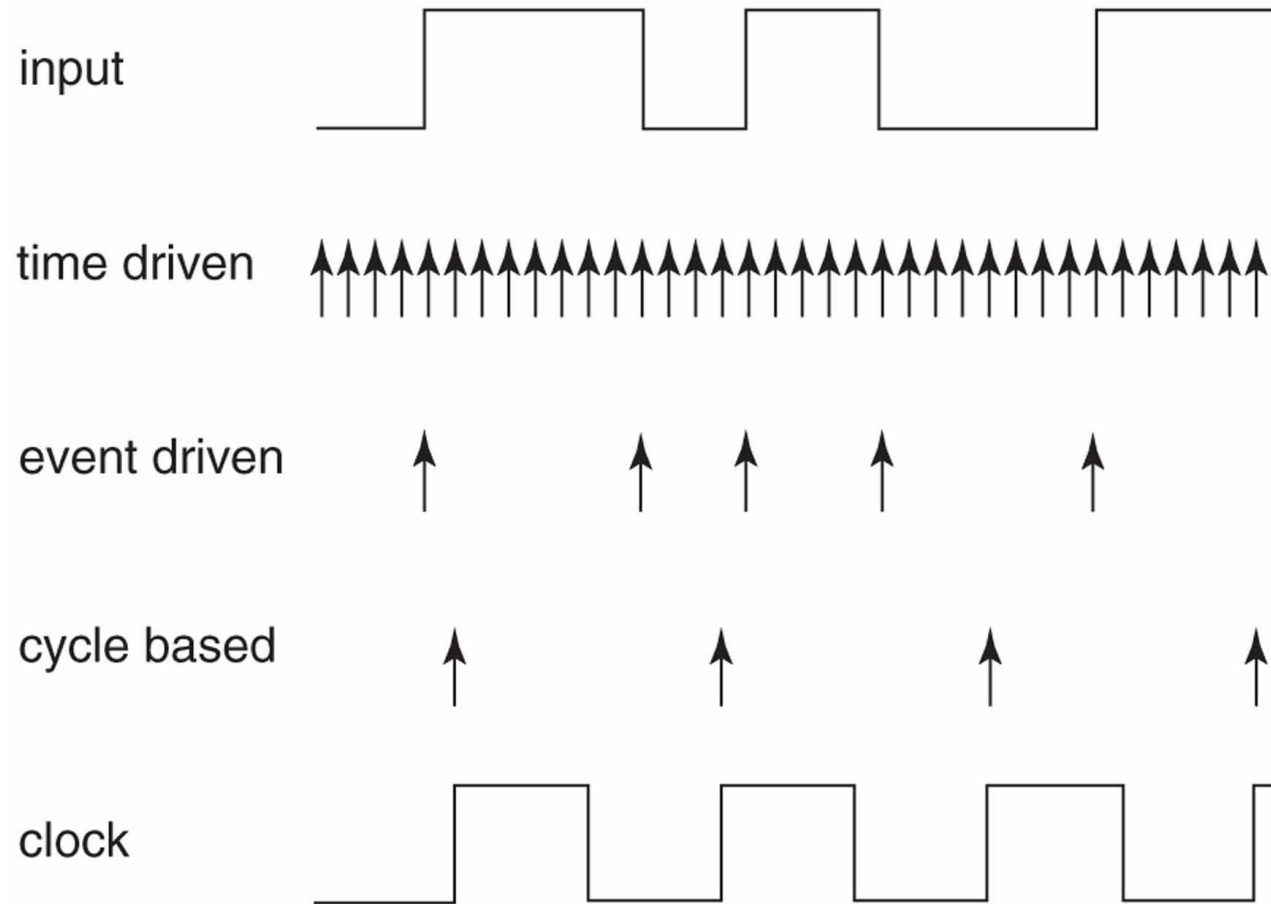
cycle based

clock

Simulation Types



Simulation Types

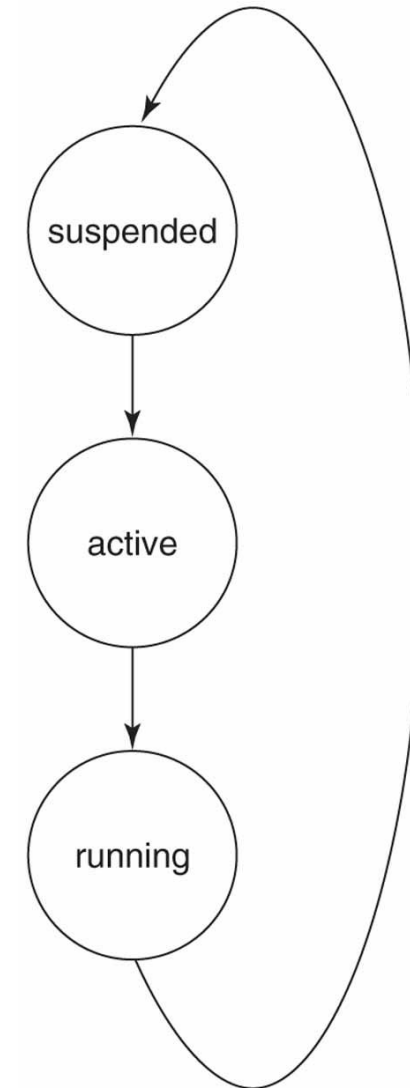


VHDL is event
drivent

Simulation Types

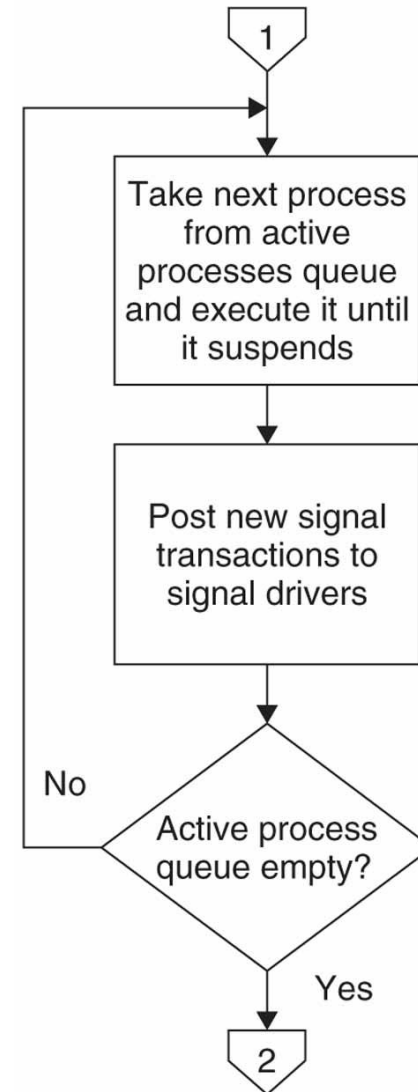
States of a Simulation Process

- **Suspended:** simulation process is not running or active
- **Active:** Simulation process is in the active processes queue waiting to be executed
- **Running:** Simulator is executing the simulation process



Simulation

- Simulation cycle consists of *signal update* and *process execution* phases
- The order in which the active processes are executed is not important



Simulation Cycle – in detail

1. The simulation time is first advanced to the next time at which a transaction on a signal has been scheduled
2. All the transactions scheduled for that time are performed
 - This may cause some events to occur on some signals
3. All processes that are sensitive to those events are resumed and are allowed to continue until they reach a wait statement and suspend
 - Again, the processes usually execute signal assignments to schedule further transactions on signals
4. When all the processes have suspended again, the simulation cycle is repeated
 - When there are no further scheduled transactions, the simulation ends

Signals versus Variables

- Variable's value is updated immediately during the current simulation cycle
- Signal's value is not updated during the current simulation cycle
 - The new value will not take effect until the update phase of the next simulation cycle
 - Without any delay information the new value is updated after *a delta delay*

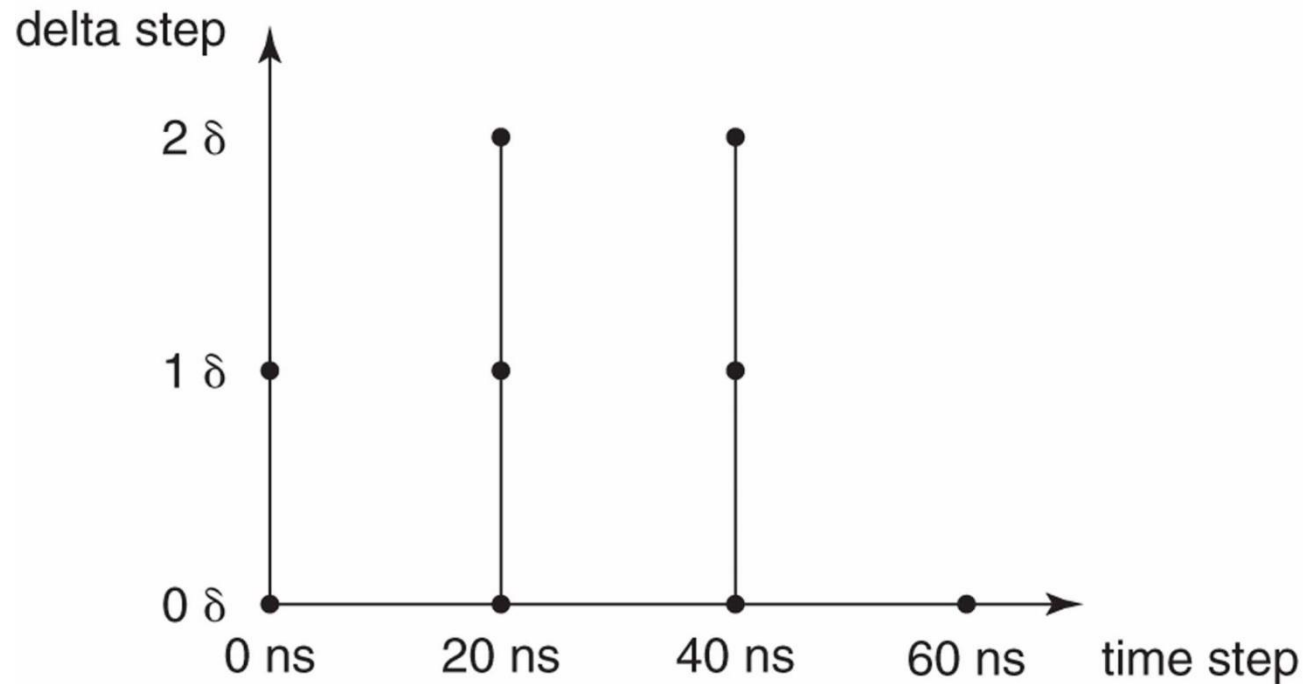
```
S <= '1';
```

```
...
```

```
if S then ...
```

- The signal value does not change as soon as the signal assignment statement is executed
- The process does not see the effect of the assignment until the next time it resumes, even if this is at the same simulation time

Delta Delay



- The signal value does not change as soon as the signal assignment statement is executed
- The process does not see the effect of the assignment until the next time it resumes, even if this is at the same simulation time

Delta Delay

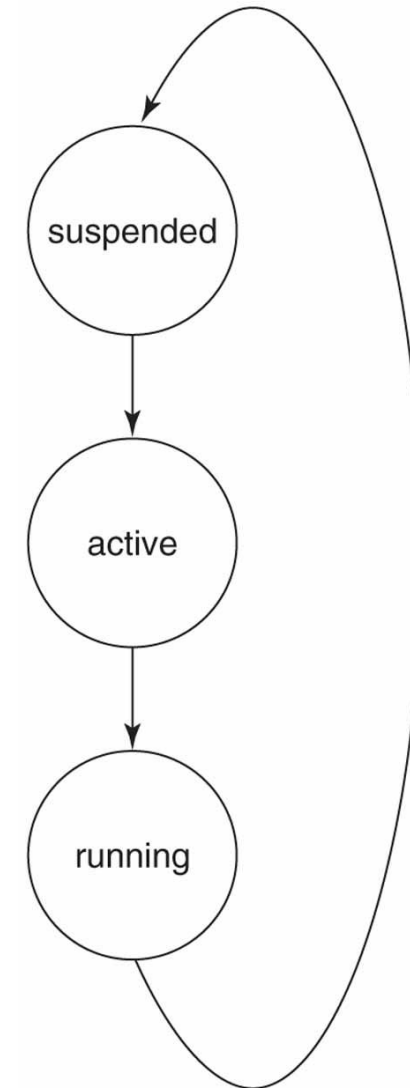
**To understand Delta Cycle,
you should understand the
simulation cycle in VHDL**

Signal update phase is
followed by a process
execution phase



States of a Simulation Process

- **Suspended:** simulation process is not running or active
- **Active:** Simulation process is in the active processes queue waiting to be executed
- **Running:** Simulator is executing the simulation process



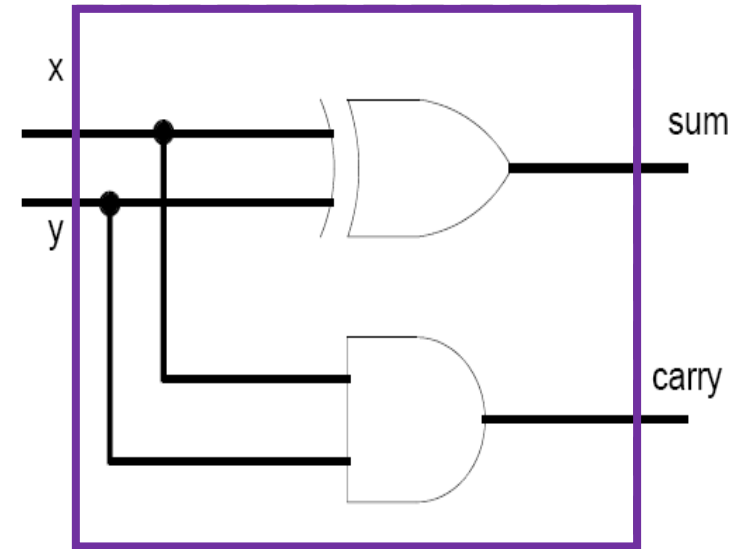


Timing

An Example Module

```
entity half_adder is  
  port(x,y: in std_logic;  
        sum, carry: out std_logic);  
end half_adder;
```

```
architecture myadder of half_adder is  
begin  
  sum <= x xor y;  
  carry <= x and y;  
end myadder;
```



Timing

- To postpone the update of a signal value, use an **after** clause to define the time period

```
Dout <= not Din after 10 ns;
```

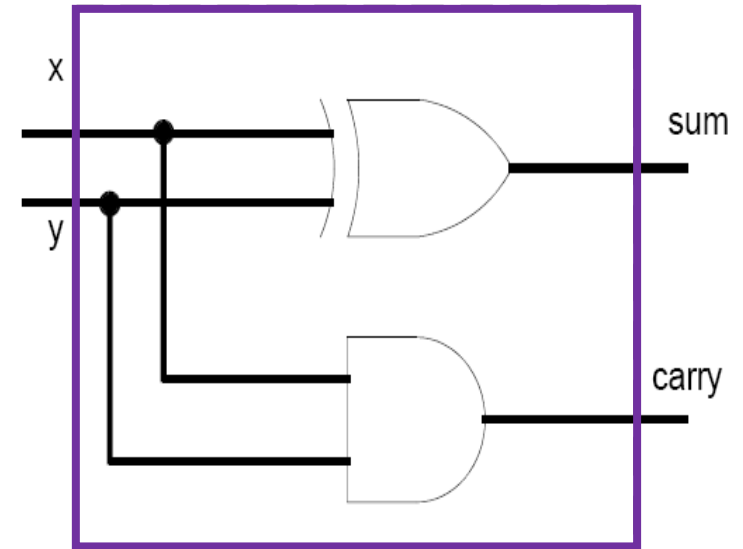
- The time period is added to current simulation time
- You can create a wave form using the **after** clause:

```
Dout <= '0' after 10 ns, '1' after 20 ns, '0' after 30 ns;
```

An Example Module

```
entity half_adder is
port(x,y: in std_logic;
      sum, carry: out std_logic);
end half_adder;
```

```
architecture myadder of half_adder is
begin
    sum <= x xor y after 10 ns;
    carry <= x and y after 10 ns;
end myadder;
```



Waiting

- Wait statement is used in processes to wait on signal changes or a time period

```
wait [sensitivity_clause] [condition_clause] [timeout_clause] ;
```

```
sensitivity_clause ::= on signal_name {, signal_name}
```

```
condition_clause ::= until boolean_expression
```

```
timeout_clause ::= for time_expression.
```

- With **on**, you can specify signal to which the process responds
- With **until**, you can stall the process until a condition is met
- With **for**, you can stall the process a time period

Waiting

- With **on**, you can specify signal to which the process responds
- With **until**, you can stall the process until a condition is met
- With **for**, you can stall the process a time period

Waiting

- With **on**, you can specify signal to which the process responds
- With **until**, you can stall the process until a condition is met
- With **for**, you can stall the process a time period

```
wait on clk until reset = '1';
```

```
wait until x for 10 ns;
```

```
wait;
```

Waiting

- With **on**, you can specify signal to which responds
- With **until**, you can stall the process until
- With **for**, you can stall the process a time

```
wait on clk until reset = '1';
```

```
wait until x for 10 ns;
```

```
wait;
```

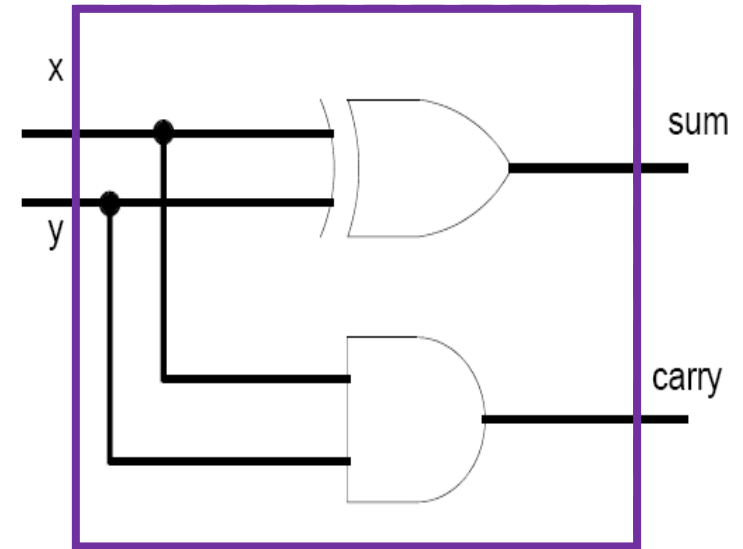
Wait on x, y until z='1' for 100 ns;

The execution of the process is suspended for a maximum of 100 ns. **If an event occurs on x or y prior to 100 ns, the condition z is evaluated.** If z is true (that is z='1') when the event on x or y occurs, the process will resume at that time; otherwise, the suspension continues

An Example Module

```
entity half_adder is
port(x,y: in std_logic;
      sum, carry: out std_logic);
end half_adder;

architecture myadder of half_adder is
begin
  process is
  begin
    sum <= x xor y after 10 ns;
    carry <= x and y after 10 ns;
    wait on x, y;
  end process;
end myadder;
```

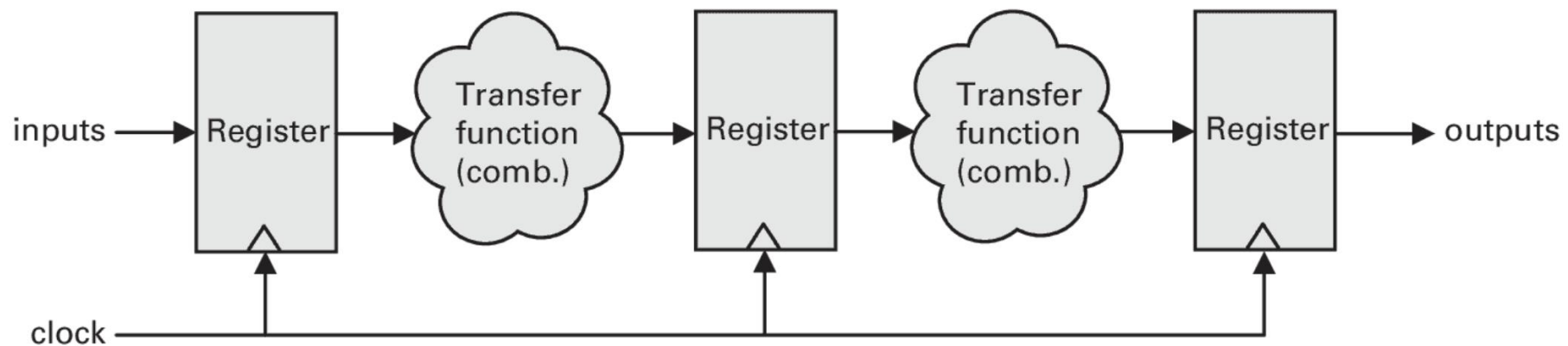




Synchronous Design

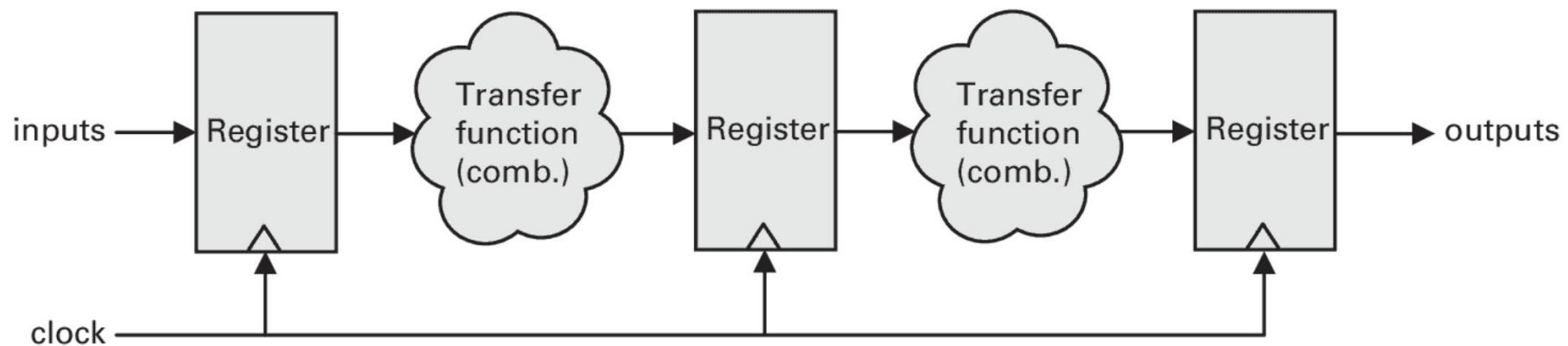
RTL Design Methodology

- RTL design is great for modeling circuits that perform computations in steps
- RTL provides a systematic approach to transforming an algorithm into a hardware circuit



RTL Design Methodology

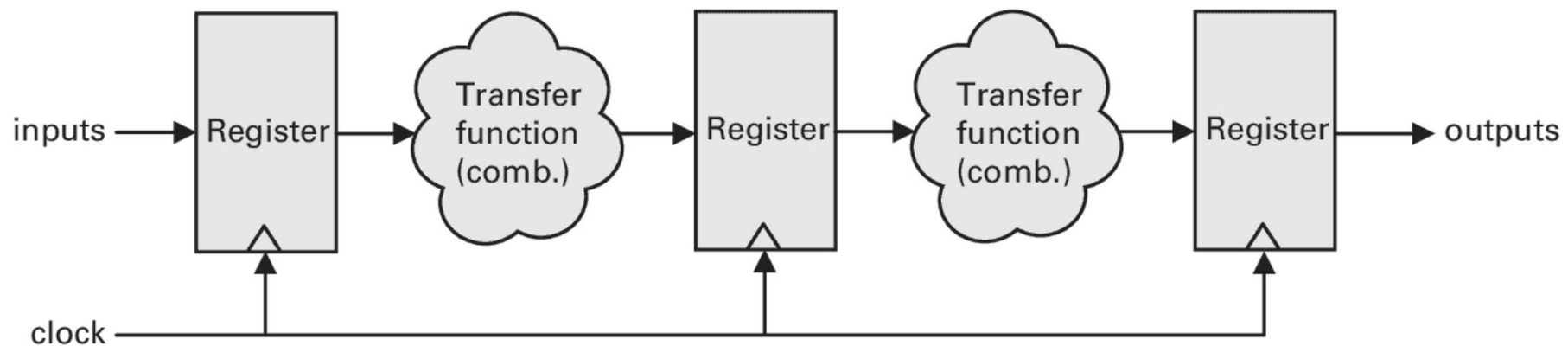
A typical RTL design is divided into two parts: a datapath that performs the data transformations and keeps calculation results, and a controller that orchestrates the entire process



RTL Design Methodology

A typical RTL design is divided into two parts: a datapath that performs the data transformations and keeps calculation results, and a controller that orchestrates the entire process

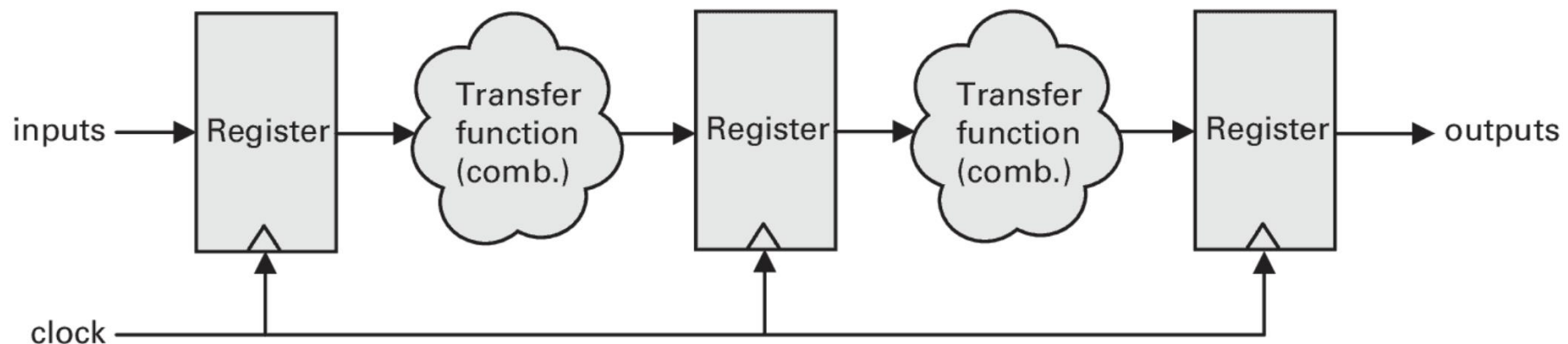
- Combinational logic blocks can be implemented as concurrent statements or combinational processes




RTL Design Methodology

A typical RTL design is divided into two parts: a datapath that performs the data transformations and keeps calculation results, and a controller that orchestrates the entire process

- Registers can be inferred from clocked processes that assign values to signals of any synthesizable data type





Synchronous Circuits Guidelines

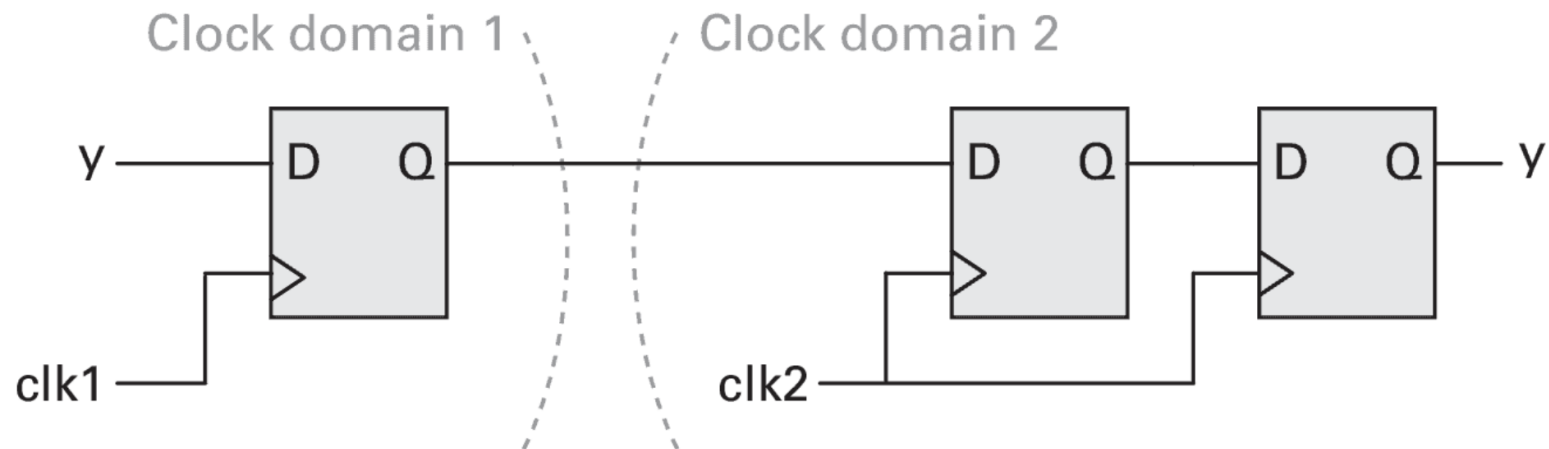


Synchronous Circuits Guidelines

- Use a single clock for the entire design
 - not always possible in more complex designs, it is a goal worth pursuing
- Use only edge-based storage elements (flip-flops)
 - Do not use latches
- Clock all flip-flops at the same edge
- Provide a reset value for each flip-flop

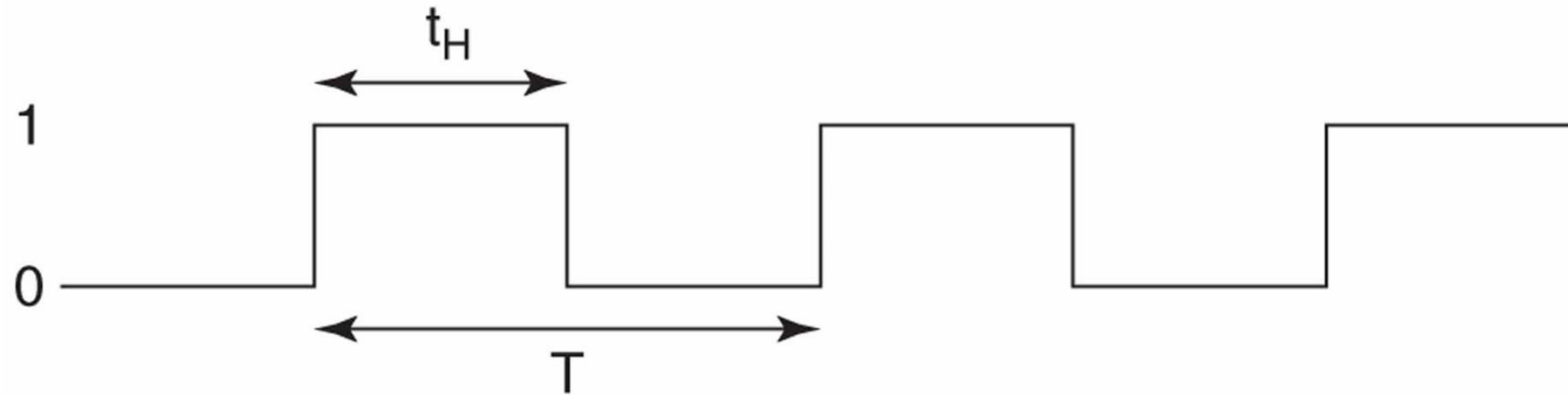
Synchronous Circuits Guidelines

- Avoid complicated and conditional resets
- Avoid combinational feedback
 - If feedback must be used, then make sure the signal passes through a register
- Synchronize all signals that cross clock domains

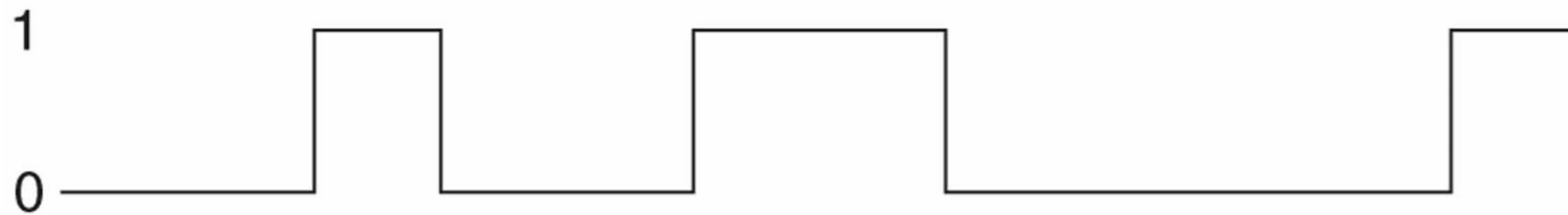


Clock Signal

- Clock signal is a sequence of pulses



(a)



(b)

Clocks

- Free running clock definitions:

```
signal clock : std_ulogic;  
...  
clock_gen : process is  
begin  
    clk <= '1';  
    wait for 10 ns;  
    clk <= '0';  
    wait for 10 ns;  
end process clock_gen;
```

```
signal clock : std_ulogic := '1';  
...  
clock <= not clock after 5ns;
```

- Limited number of clock cycles:

```
signal clock : std_ulogic;  
...  
process  
begin  
    for i in 1 to num_clockcycles loop  
        clock <= not clock;  
        wait for 10 ns;  
        clock <= not clock;  
        wait for 10 ns;  
    end loop;  
    wait;  
end process;
```

Reset Signal

- Reset signal sets the system in a predefined state

```
signal clock : std_ulogic := '1';  
signal rst   : std_logic;  
.  
.  
.  
rst <= '0', '1' after 10 ns, '0' after 100 ns;  
clock <= not clock after 5ns;
```



Edge Sensitive Storage





Edge Sensitive Storage

- In VHDL code, clock signals are no different than any other signal
- A register is inferred when a signal or variable is updated on a clock edge
 - in other words, when we do an assignment inside a region of a process that is subject to a clock edge

Edge Sensitive Storage

The edge condition can be modeled in two basic forms:

A process with a sensitivity list and then test for the clock edge using an if statement

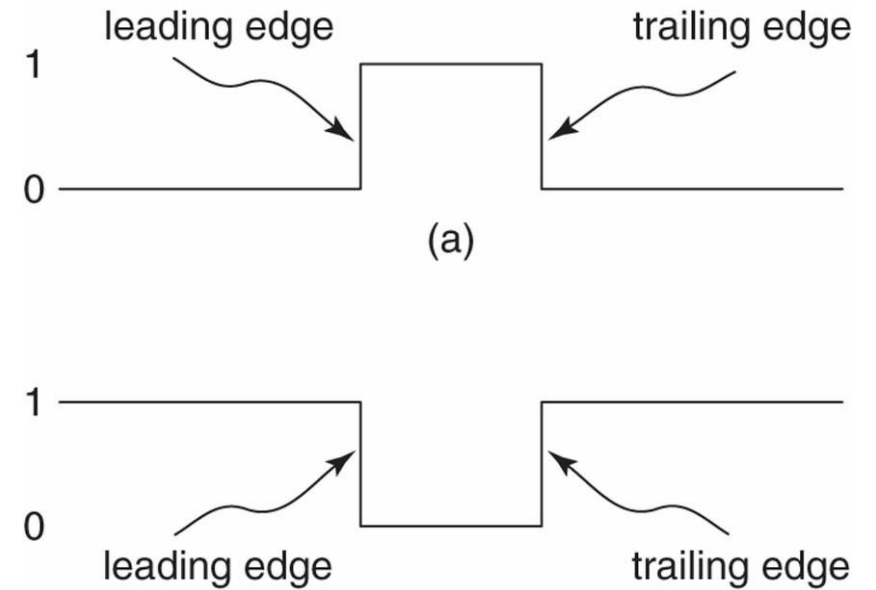
A wait statement that suspends until a clock edge occurs

- In this case, the other statements inside the process are implicitly subject to this condition
- The wait statement should be the first or last statement in the process

Edge Sensitive Storage

A process with a sensitivity list and then test for the clock edge using an if statement

A wait statement that suspends until a clock edge occurs



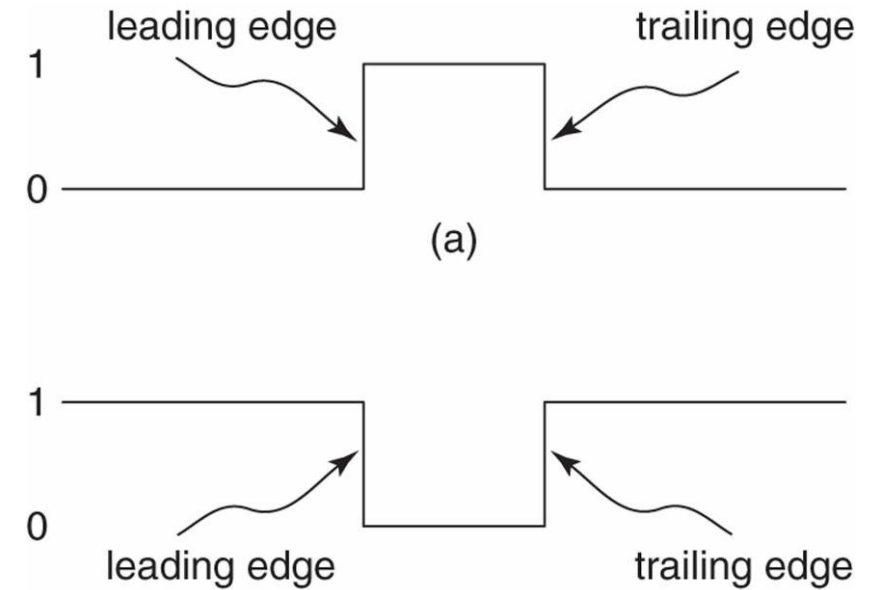
Edge Sensitive Storage

Function table for a positive-level D latch.

D	CLK	Q_{t+1}
0	0	Q_t
0	1	0
1	0	Q_t
1	1	1

Function table for a positive-edge-triggered D flip-flop.

D	CLK	Q_{t+1}
0	\uparrow	0
1	\uparrow	1
X	0	Q_t
X	1	Q_t



D Latch and Flip-flop

```
[process_label:] process (<clock_signal>)  
  <declarations>  
begin  
  if <clock_edge> then  
    <sequence_of_statements>  
  end if;  
end process [process_label];
```

Function table for a positive-edge-triggered D flip-flop.

D	CLK	Q_{t+1}
0	↑	0
1	↑	1
X	0	Q_t
X	1	Q_t

Function table for a positive-level D latch.

D	CLK	Q_{t+1}
0	0	Q_t
0	1	0
1	0	Q_t
1	1	1

```
[process_label:] process (<clock_signal>, <input_signals>)  
  <declarations>  
begin  
  if <clock_level> then  
    <sequence_of_statements>  
  end if;  
end process [process_label];
```



Synchronous Process: Flip-Flop

```
process (clk)
begin
    if rising_edge(clk) then
        q <= d ;
    end if;
end process;
```

```
process
begin
    wait until rising_edge(clk);
    q <= d ;
end process;
```



Sync and Async Reset Signal

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

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



In a synchronous design, only edge-sensitive storage elements should be used

Level-sensitive storage

- Level-sensitive storage may happen unintentionally, sometimes because of coding styles that lead to the inference of latches
- It is important to know how to model latches to avoid them
- Latches can only be inferred in assignments that are not under control of a clock edge



**UNIVERSITY
OF TURKU**