

FFs and Registers

In this lecture, we show how the **process** block is used to create FFs and registers

Flip-flops (FFs) and **registers** are both derived using our standard data types, *std_logic*, *std_logic_vector*, *signed* and *unsigned*

Storage elements are critical to emulating *variables* in programming languages

They play a central role in allowing C programs to be converted into hardware implementations

VHDL (and verilog) allow complex hardware to be described in either **single-segment** style to **two-segment** style

Proponents of single-segment style argue that such descriptions are

- More efficient from a simulation perspective (the sensitivity list consists of *clk* only)
- More concise, i.e., requiring fewer VHDL statements to describe the circuit

Neither of these are compelling reasons, and neither offset the benefits of two-segment style (in my opinion)

One-Segment vs. Two-Segment Style

We will use **two-segment** style exclusively throughout the rest of this lecture series for several reasons:

- Two-segment provides a *conceptual advantage* by cleanly separating storage elements from the combinational logic portion of the design

After many years of experience, the biggest challenge of writing VHDL is being able to **quickly** craft a description that has the fewest **bugs**

The advantage afforded by partitioning the circuit into combinational and sequential components is difficult to over-state

- Two-segment style will provide opportunities to guide the synthesis tool to produce a more efficient hardware implementation (in my opinion)

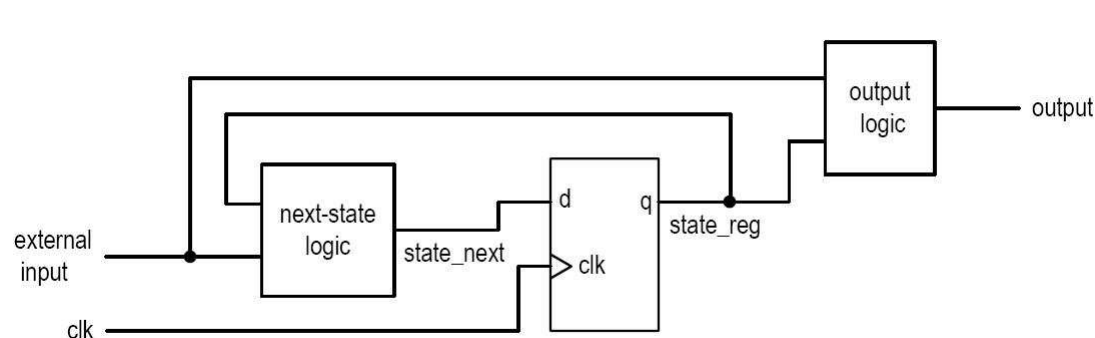
Two-segment style provides **easy access** to **both** the inputs and outputs of FFs and registers

Two-segment style will also allow the designer to easily specify **signals** (wires) in combinational circuit descriptions, without being forced to create FFs

FFs and Finite State Machines

We will focus on creating designs with a *single-clock domain* and a globally distributed *clk* signal (*globally synchronous*)

The end goal of our learning will be to create a **finite state machine (FSM)**



State registers (state_reg)
represent the storage
elements

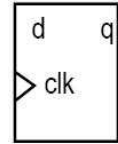
Next state logic
represent the combinational
circuit that determines
state_next

From previous courses, you probably remember the following about FSM operation:

- At the rising edge of the clock, *state_next* is sampled and stored into the register (and becomes the new value of *state_reg*)
- The external *inputs* and *state_reg* signals propagate through **next-state** and **output** logic to determine the new values of the *state_next* and *output* signals
- This sequence of operations repeats indefinitely

FFs

DFFs are the workhorse of modern digital circuit design, and they play a central role in **FSM** and **datapath** implementations



clk	q*
0	q
1	q
\downarrow	d

(b) pos-edge triggered D FF

The truth table of a DFF specifies that the state of the FF remains *unchanged* until a **rising edge** of the clock arrives

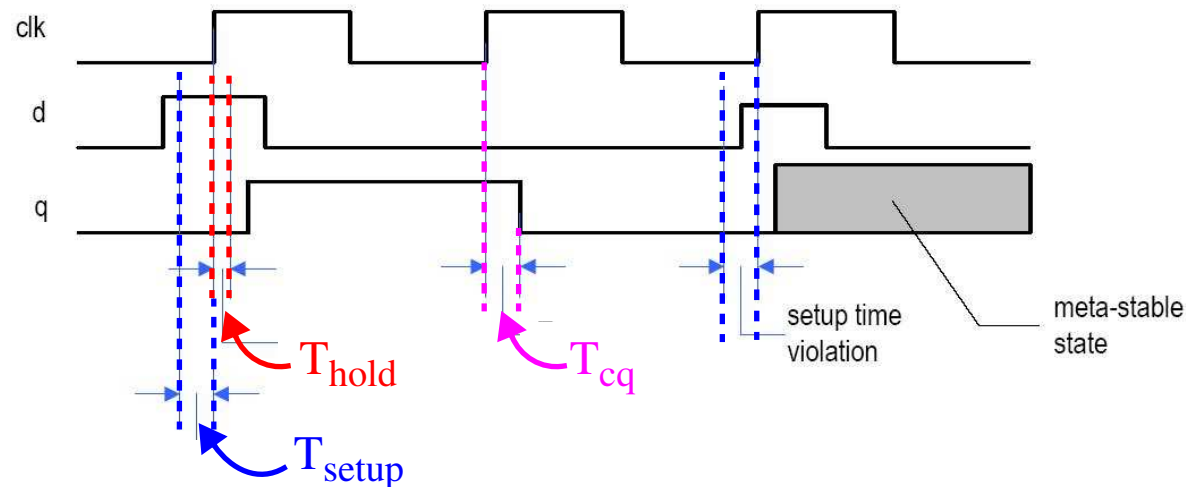
This type of FF is referred to a **rising-edge-triggered DFF**, or **FF** for short (we will NEVER use falling-edge-triggered FFs)

Most FFs that you will create will also have a *set* or *reset* signal

FFs

It is important to have a solid understanding of the timing diagram for a FF

Sooner or later, you will encounter timing violations in your design and will need to either 1) fix them or 2) slow the clock down



Every storage element has *setup* and *hold* time requirements

- Setup time is the amount of time a signal (driving the *d* input) needs to be stable **before** the rising edge of the *clk*
- Hold time is the amount of time this signal needs to be maintained on *d* **after** the rising edge of the *clk*

FFs

Timing violations are almost always *setup-time* violations

When this happens, the length of the path through the combinational circuit is TOO long

This can happen if you have too many **when-else** clauses in a conditional signal assignment, which creates a critical path that is longer than the *clk* cycle time

When you set the *clk* frequency during synthesis to, say, 50 MHz, ALL combinational paths in your design MUST be less than (20 ns - *setup-time*)

The synthesis tool will work very hard to create implementations from your VHDL descriptions that meet the timing requirements

If it fails, the onus is on you to fix the timing violations

We will discuss simple strategies that you can use to deal with timing violations when we get to FSM design

FFs

Use the following **process** block construct to create a FF with an **asynchronous reset**

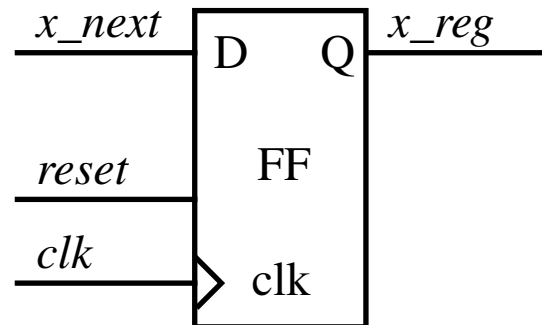
```
architecture beh of example_design is  
  signal x_reg, x_next: std_logic;  
  begin  
    process(clk, reset)  
      begin  
        if ( reset = '1' ) then  
          x_reg <= '0';  
        elsif ( rising_edge(clk) ) then  
          x_reg <= x_next;  
        end if;  
      end process;  
  
    ...  
  end beh;
```

Memorize this syntax! You will use this same structure over-and-over again, in every VHDL module you create

FFs

The FF has two names, which specify the input, x_next and the output, x_reg

The synthesis tool creates a FF as follows from this **process** block description:



If you attempt to synthesize this by itself, the synthesis tool will delete the FF b/c the input and output are not connected to anything (they float)

In most design scenarios, you will want to 'control' updates to your FFs

In other words, your FFs will maintain their contents *most* of the time, and only occasionally will be updated with new values (on one of the rising clk edges)

There are two ways of doing this:

- Add a MUX before the input x_next
- Gate the clock, i.e., add an AND gate in series with the clk connection

FFs

Golden Rule of Digital Design: Never insert any logic in series with the *clk*

Doing so makes it difficult for the synthesis tool to do a proper timing analysis

Experienced circuit designers will violate this rule to reduce the switching frequency of the FFs and save energy

So unless you have a really good excuse, DON'T DO IT.

The alternative of using a MUX is by far the most common method

This is easily specified using a **with-select** or **when-else** statement

with en **select**

```
x_next <= new_val when en = '1',  
        x_reg when others;
```

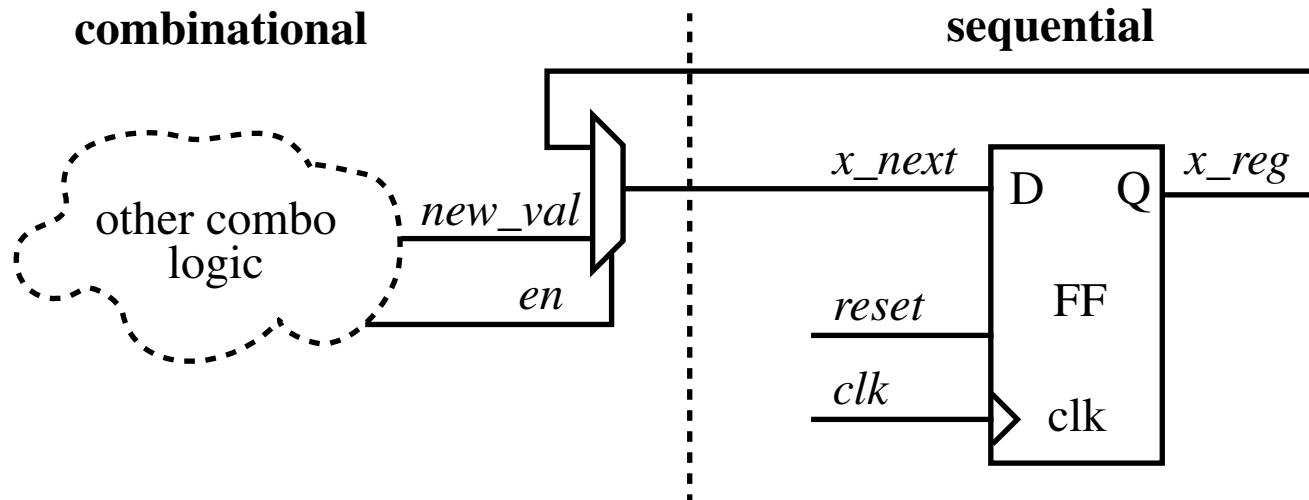
OR

```
x_next <= new_val when en = '1' else  
        x_reg;
```

As we discussed, **with-select** is a better match since we are describing a MUX, but many times the Boolean expression may be more complex, with multiple signals

FFs

In either case, the following schematic is created by the synthesis tool



Note that I've separated the design into *combinational* and *sequential* components, and placed the MUX in the combinational component

As I mentioned, two-segment style creates this conceptual representation where:

- Combo: *_reg* signals are INPUTS and *_next* signals are OUTPUTS
- Seq: *_reg* signals are OUTPUTS and *_next* signals are INPUTS

The *other combo logic* portion must be specified, otherwise the synthesis tool will eliminate the FF and MUX -- more on this soon...

Golden Rules of FF Process Blocks

We will 'close the loop' and create a valid design soon, but let's first cover my **golden rules** on **process** blocks which describe FFs

NEVER VIOLATE THESE RULES

- **Rule 1:** Never include anything except *clk* and *reset* in the sensitivity list of FF **process** blocks

```
process (clk, reset)
begin
  if ( reset = '1' ) then
    x_reg <= '0';
  elsif (rising_edge(clk)) then
    x_reg <= x_next;
  end if;
end process;
```

The synthesis tools looks for 'clues' in your VHDL code for constructs that describe FFs, and then *infers* them during synthesis

More importantly, YOU always want to be sure where these FFs are inferred!!!

Golden Rules of FF Process Blocks

- **Rule 2:** Always use this template: *if (reset = '1')* ... *elsif (rising_edge(clk))* ...

```
process (clk, reset)
begin
  if ( reset = '1' ) then
    x_reg <= '0';
  elsif ( rising_edge(clk) ) then
    x_reg <= x_next;
  end if;
end process;
```

The ONLY exception is when you want a **synchronous** reset, i.e., when reset of the FFs ONLY occurs on the rising edge of *clk* (NOTE: NEVER USE BOTH TYPES)

```
process (clk, reset)
begin
  if ( rising_edge(clk) ) then
    if ( reset = '1' ) then
      x_reg <= '0';
    else
      x_reg <= x_next; ...
```

Golden Rules of FF Process Blocks

- **Rule 3:** Never include anything except **signal assignment** in the *if-elsif* statements

```
process (clk, reset)
begin
  if ( reset = '1' ) then
    x_reg <= '0';
  elsif ( rising_edge(clk) ) then
    x_reg <= x_next;
  end if;
end process;
```

Define ALL combinational functions **OUTSIDE** the FF **process** block

VHDL allows a lot of flexibility w.r.t. describing circuits, and there are many, many ways that you can write code that will result in unexpected behavior

Although the labs will task you on writing VHDL and then inspecting the schematics produced by the synthesis tool, you will not be able to do this very often in practice
Even moderately complex designs make this type of inspection untenable

Consequences of Violating the Golden Rules of FF Process Blocks

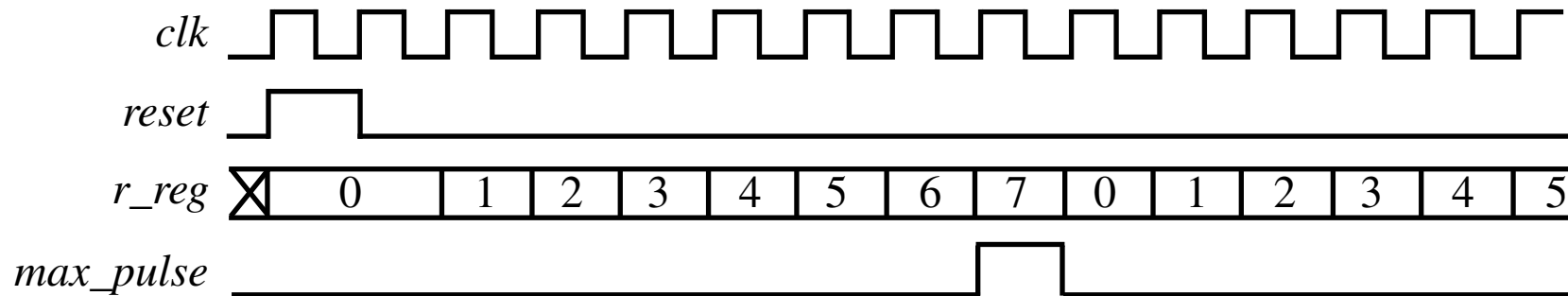
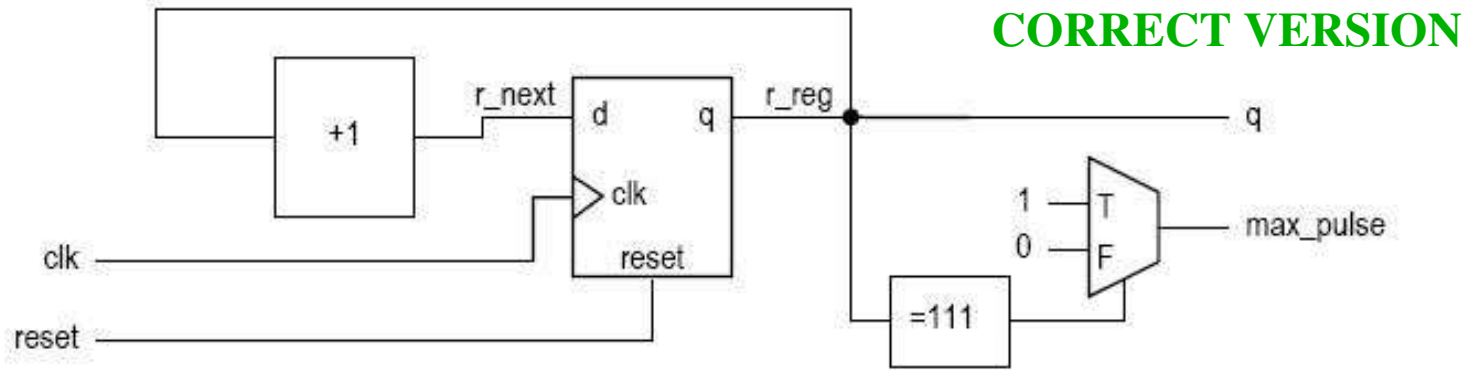
Violating **golden rule 3** is the most common

Here's the **correct** description of a circuit that generates a pulse every 8 clock cycles, which assumes *max_pulse* is defined in the **entity** as a std_logic **out** signal

```
architecture beh of pulse_cir is
  signal r_reg, r_next: unsigned(2 downto 0);
begin
  process(clk, reset)
    begin
      if ( reset = '1' ) then
        r_reg <= (others=>'0');
      elsif ( rising_edge(clk) ) then
        r_reg <= r_next;
      end if;
    end process;
    r_next <= r_reg + 1;
    max_pulse <= '1' when r_reg = "111" else '0';
end beh;
```

Consequences of Violating the Golden Rules of FF Process Blocks

This synthesizes to the following correct version of the circuit



The synthesis tool predictably (and correctly) interprets both the sequential and combinational components of the design

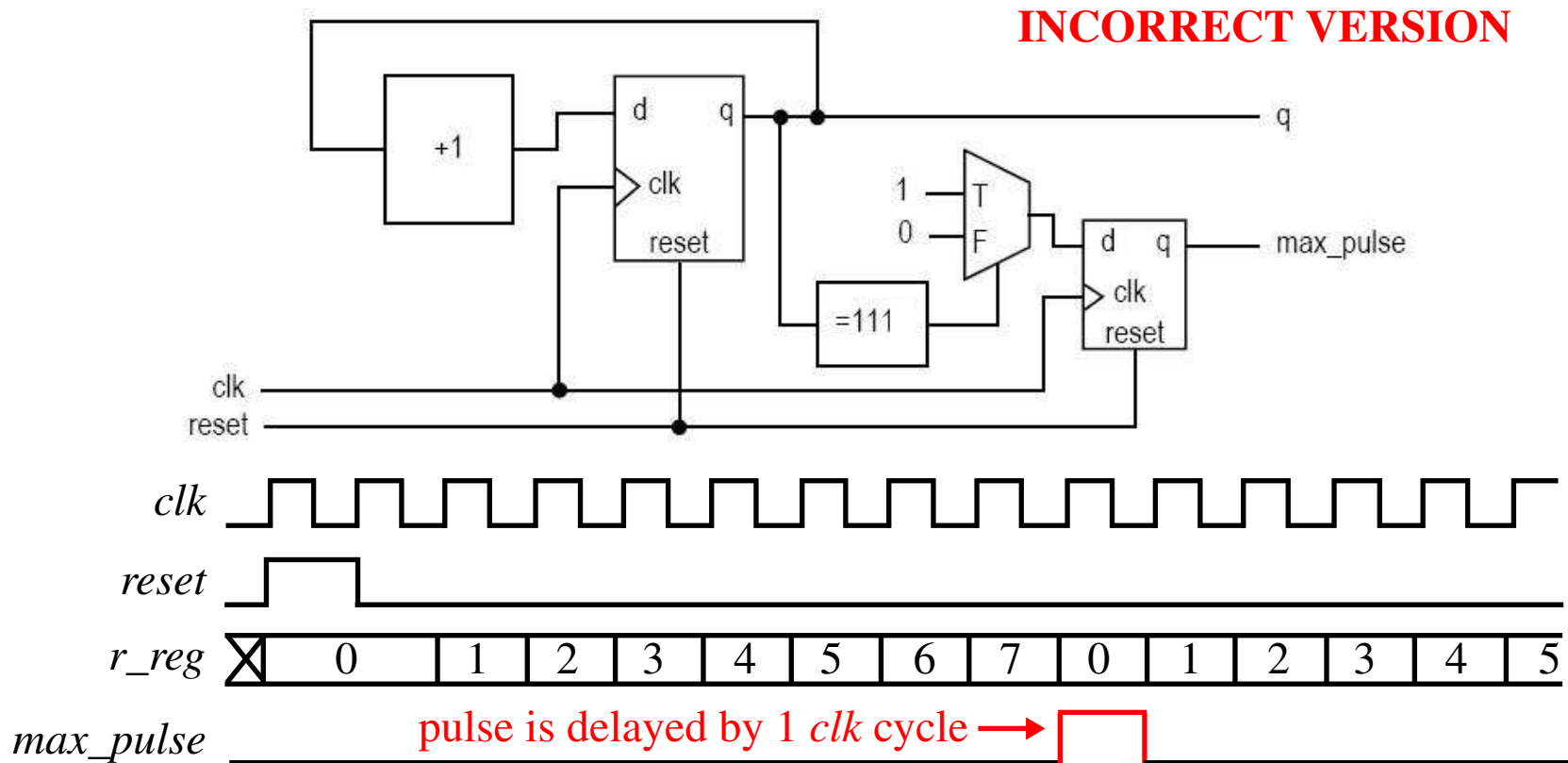
Consequences of Violating the Golden Rules of FF Process Blocks

If, on the other hand, you try to place the *max_pulse* code inside the **process** block:

```
architecture beh of pulse_cir_INCORRECT is
  signal r_reg, r_next: unsigned(2 downto 0);
begin
  process(clk, reset)
    begin
      if ( reset = '1' ) then
        r_reg <= (others=>'0');
      elsif ( rising_edge(clk) ) then
        r_reg <= r_next;
        if (r_reg = "111") then
          max_pulse <= '1';
        else
          max_pulse <= '0';
        end if;
      end if;
    end process;
  r_next <= r_reg + 1; end pulse_cir_INCORRECT;
```


Consequences of Violating the Golden Rules of FF Process Blocks

The synthesis tool synthesizes this INCORRECT version of the circuit:



An additional FF for *max_pulse* is inferred because *max_pulse* is not assigned to **under all possible conditions** (it is in the *elsif* branch), and the pulse is delayed!

There are ways to fix this problem, but the best solution is 'DON'T DO THIS'