



LUND
UNIVERSITY

EITF35: Introduction to Structured VLSI Design

Part 2.2.2: VHDL-3

Liang Liu
liang.liu@eit.lth.se



Outline

□ Inference of Basic Storage Element

□ Some Design Examples

- DFF with enable
- Register shifter
- Counter

□ Coding Style: Segment

□ Variables in Sequential Circuit

□ Poor Design Examples



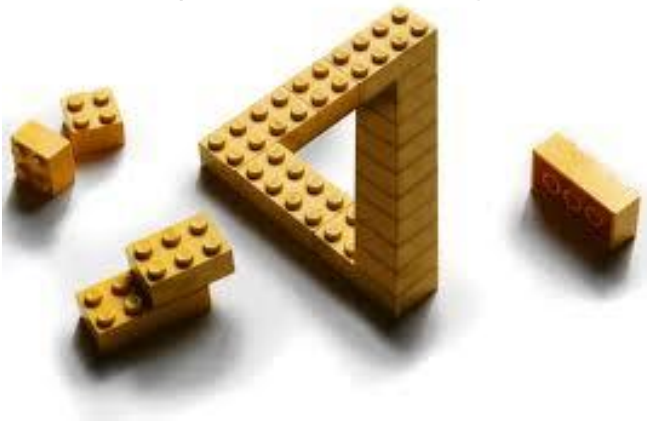
Inference of Basic Storage Elements

□ VHDL code should be clear so that the pre-designed cells can be inferred

- *Again, as an architecture designer, you need to be very familiar with the available elements*

□ VHDL code of storage elements

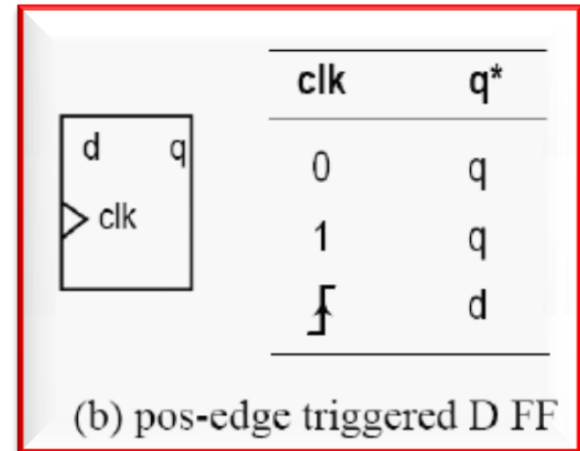
- Positive edge-triggered D FF
- Negative edge-triggered D FF
- D FF with asynchronous reset
- D Latch (***DON'T USE***)



Positive edge-Triggered D FF

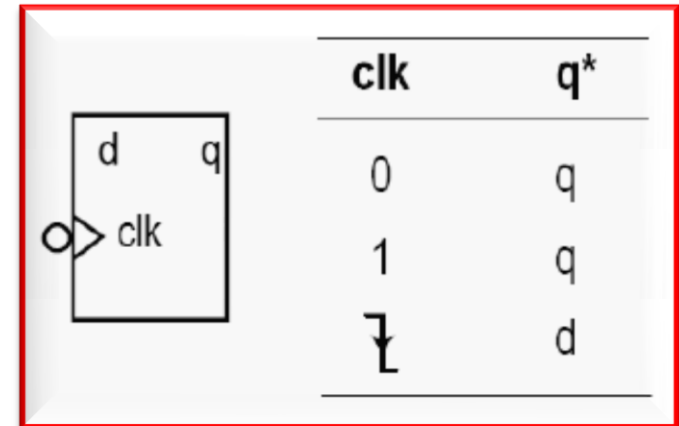
- No else branch
- Note the sensitivity list (*only clk*)

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port(
    clk: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end dff;
architecture arch of dff is
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      q <= d;
    end if;
  end process;
end arch;
```



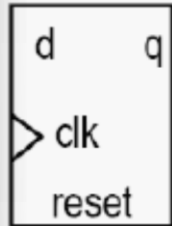
Negative edge-Triggered D FF


```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port(
    clk: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end dff;
architecture arch of dff is
begin
  process (clk)
  begin
    if (clk'event and clk='0') then
      q <= d;
    end if;
  end process;
end arch;
```



D FF with Async. Reset

```
entity dffr is
  port(
    clk: in std_logic;
    reset: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end dffr;
architecture arch of dffr is
begin
  process (clk, reset)
  begin
    if (reset='1') then
      q <= '0';
    elsif (clk'event and clk='1') then
      q <= d;
    end if;
  end process;
end arch;
```



	reset	clk	q*
	1	-	0
	0	0	q
	0	1	q
	0		d



D Latch (Learn How to Avoid)

Bad1:

```

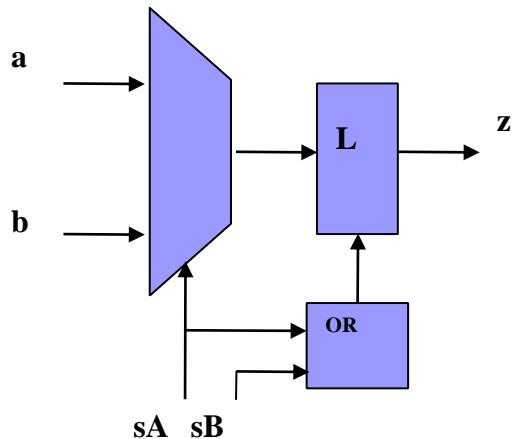
process (sA, sB, a, b)
begin
  if sA='1' then
    z<=a;
  elsif sB='1' then
    z<=b;
  end if;
end process Bad1;
  
```

		c	q*
d	q	0	q
c		1	d

Bad2:

```

process (sA, a, b)
begin
  if sA='1' then
    f<=a;
  end if;
end process Bad2;
  
```



Bad3:

```

process (I3, I2, I1, I0, S)
begin -- use case statement
  case S is
    when "00" => O <= I0;
    when "01" => O <= I1;
    when "10" => O <= I2;
  end case;
end process Bad3;
  
```



Exercise

```

c1: process (clk)
begin
    if (clk 'event
and clk = '1') then
        q <= '1';
    end if;
end process c1;

```

```

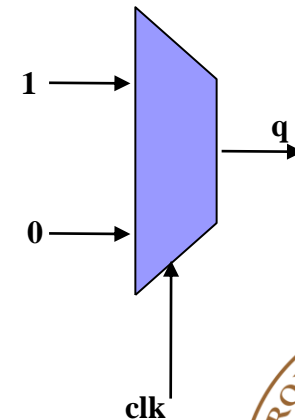
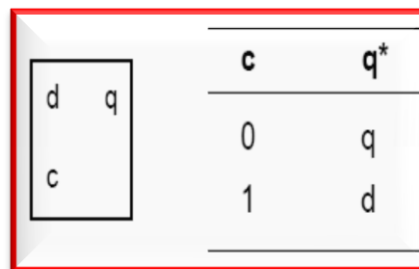
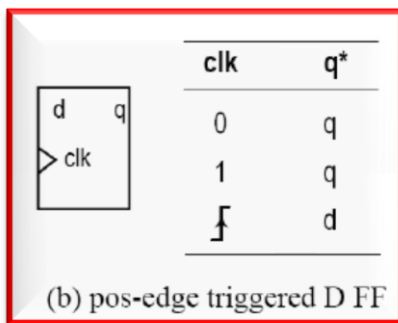
c2: process (clk)
begin
    if (clk
= '1') then
        q <= '1';
    end if;
end process c2;

```

```

c3: process (clk)
begin
    if (clk
= '1') then
        q <= '1';
    else
        q <= '0';
    end if;
end process c3;

```



Outline

□ Inference of Basic Storage Element

□ **Some Design Examples**

- DFF with enable
- Register shifter
- Counter

□ Coding Style: Segment

□ Variables in Sequential Circuit

□ Poor Design Examples

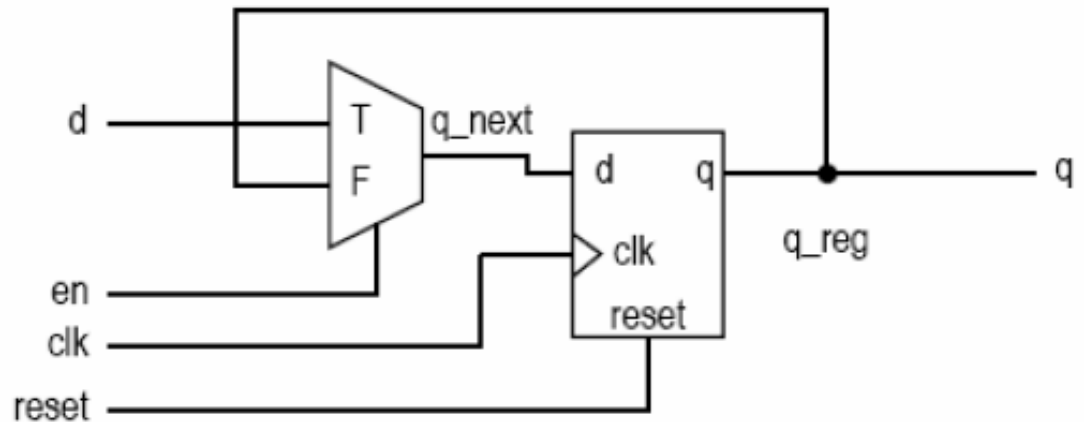


Design Examples: D FF with sync enable

□ Sync Enable

- Means the enable signal is controlled by clock

reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	f	0	q
0	f	1	d



Design Examples: D FF with sync enable

architecture two_seg_arch of dff_en is

```
signal q_reg:std_logic;  
signal q_next:std_logic;
```

```
begin
```

```
-- D FF
```

```
  process (clk, reset)
```

```
  begin
```

```
    if (reset='1') then
```

```
      q_reg <= '0';
```

```
    elsif (clk'event and clk='1') then
```

```
      q_reg <= q_next;
```

```
    end if ;
```

```
  end process;
```

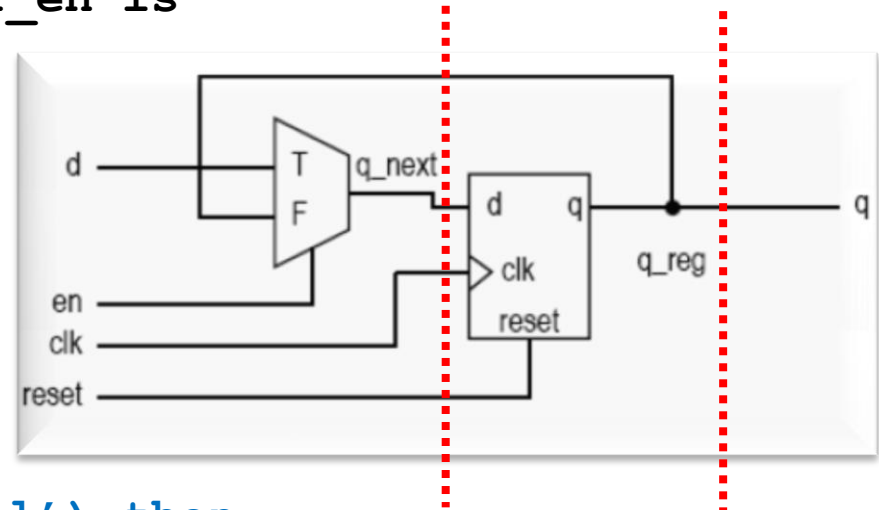
```
-- next-state logic
```

```
  q_next <= d when en ='1' else  
    q_reg;
```

```
-- output logic
```

```
  q <= q_reg;
```

```
end two_seg_arch;
```



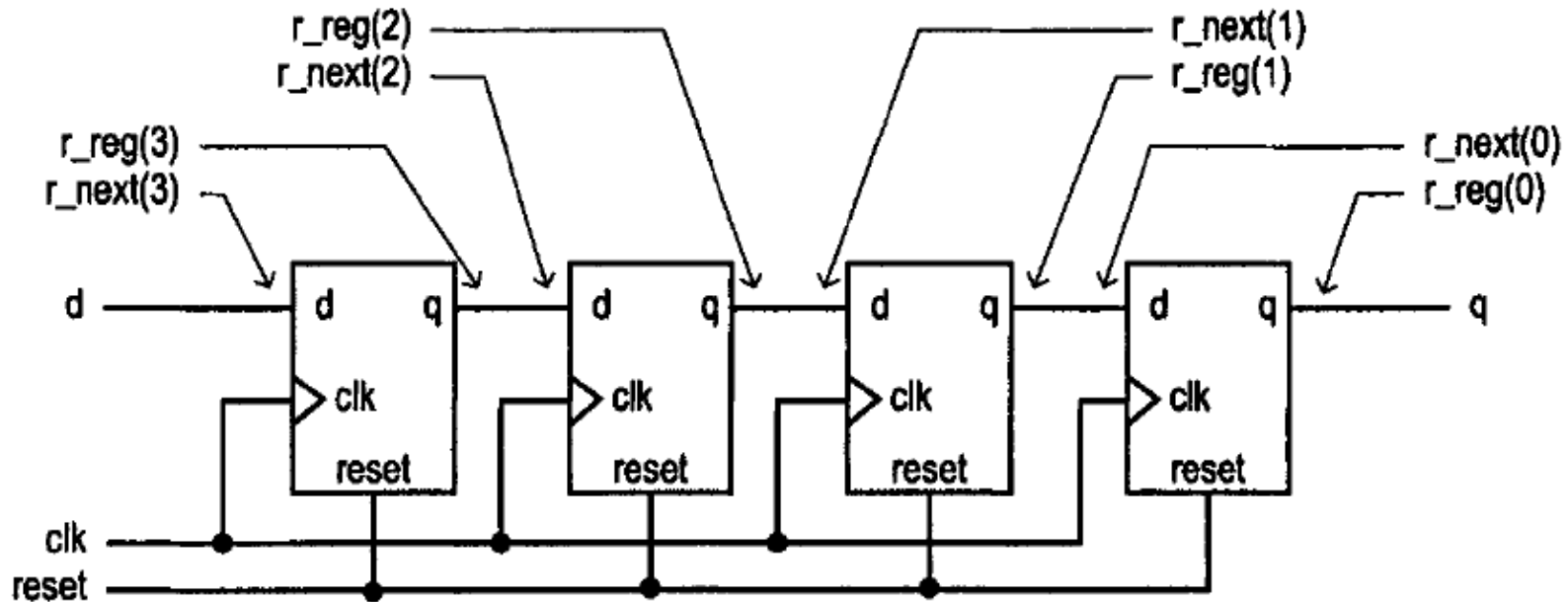
**Multi-Segment
Recommended!**



Design Examples: Shift-right registers

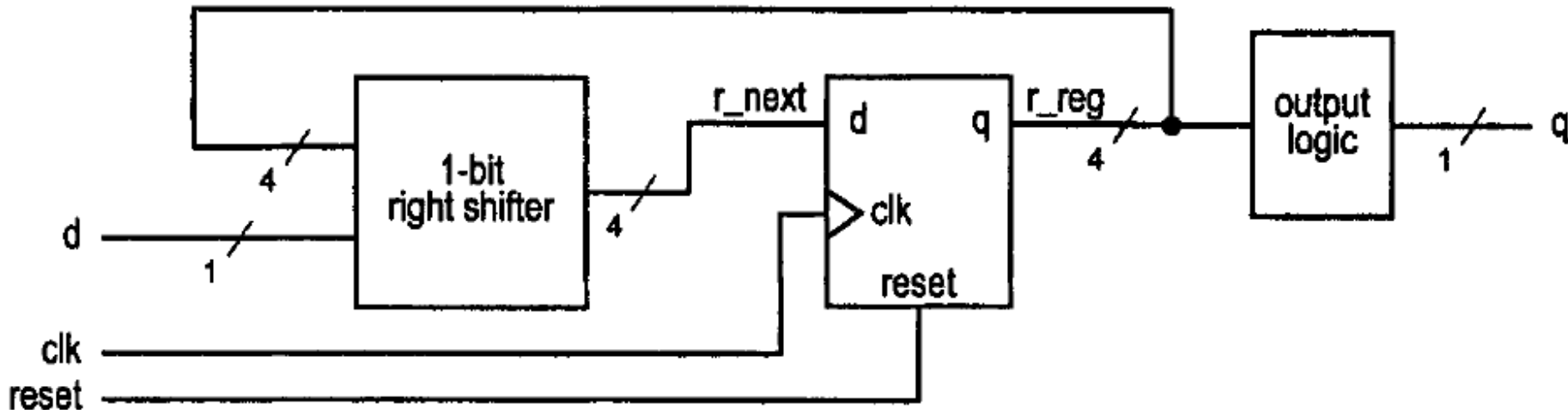
□ Shift Registers

- Shifts the content of the register left or right 1 bit in each clock cycle
- Major application: send parallel data through a serial line



Design Examples: Shift-right registers

- If we group the registers together: **multi-segment form**



Design Examples: Shift-right registers

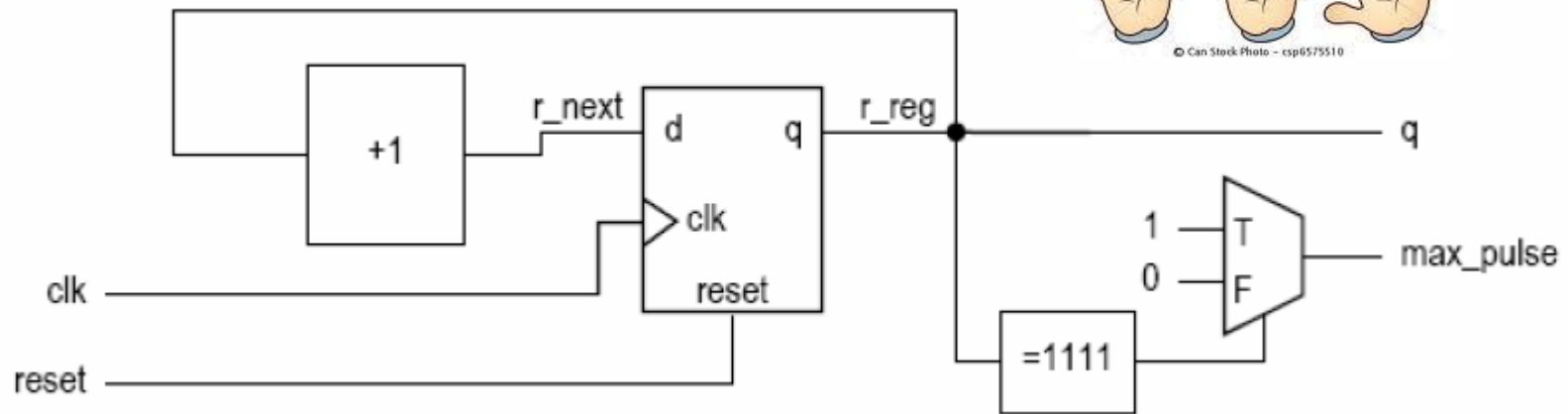
```
architecture two_seg_arch of shift_right_register is
signal r_reg : std_logic_vector (3 downto 0);
signal r_next : std_logic_vector (3 downto 0);
begin
    process (clk,reset)
        begin
            if(reset= '1') then
                r_reg <= (others=>'0');
            elsif (clk 'event and clk= '1') then
                r_reg <= r_next;
            end if ;
        end process;
        r_next <= d & r_reg(3 downto 1);
        q <= r_reg(0);
    end two_seg_arch;
```



Design Examples: Binary Counter

Binary Counter

- Circulates through a sequence that resembles the unsigned binary number
- An n-bit binary counter has a register with n FFs
- Increments the content of the register every clock cycle, counting from 0 to $2^n - 1$ and then repeating.



Design Examples: Binary Counter

```
entity binary_counter4_pulse is
port( clk, reset: in std_logic;
max_pulse: out std_logic;
q: out std_logic_vector (3 downto 0);
end binary_counter4_pulse ;
architecture two_seg_arch of binary_counter4_pulse is
signal r_reg : unsigned (3 downto 0) ;
signal r_next : unsigned (3 downto 0) ;
    process (clk, reset)
        begin
            if (reset='1') then r_reg <= ( others=> '0') ;
            elsif (clk'event and clk='1') then r_reg <= r_next;
            end if;
        end process;
    r_next <= r_reg + 1; -- incrementor
    q <= std_logic_vector(r_reg);
    max_pulse <= '1' when r_reg= "1111" else '0'; -- output
end two_seg_arch;
```



Design Examples: Binary Counter

□ How to wrap around: 1111->0000

- Poor code ('Wrong' code)

```
bad:r_next <= (r_reg + 1) mod 16
```

□ In the IEEE `numeric_std` package, "+" on the **unsigned** data type is modeled after a **hardware adder**

□ Wrap around **automatically** when the addition result exceeds the range.

□ **Mod** operation may cannot be synthesized

```
Good:r_next <= (r_reg + 1)
```



Outline

□ Inference of Basic Storage Element

□ Some Design Examples

- DFF with enable
- Register shifter
- Counter

□ **Coding Style: Segment**

□ Variables in Sequential Circuit

□ Poor Design Examples



Coding Style: Segment

□ One-segment

- Describe storage and combinational logic in one process
- May appear compact for certain simple circuit
- But it can be error-prone

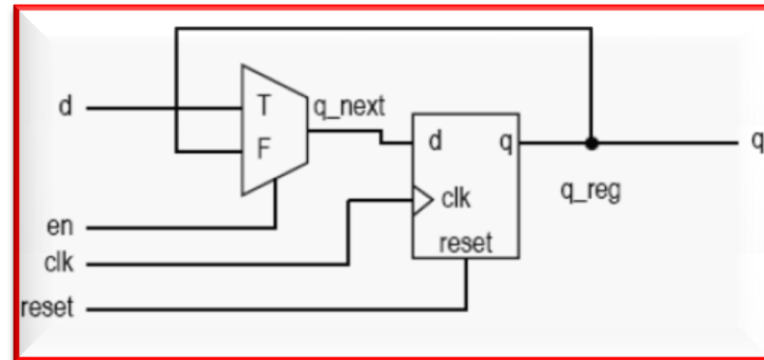


*Is integration
always better???*



Segment: D FF with sync enable

```
architecture one_seg_arch of dff_en is
begin
  process (clk,reset)
  begin
    if (reset='1') then
      q <= '0' ;
    elsif (clk'event and clk='1') then
      if (en='1') then
        q <= d;
      end if;
    end if ;
  end process;
end one_seg_arch;
```



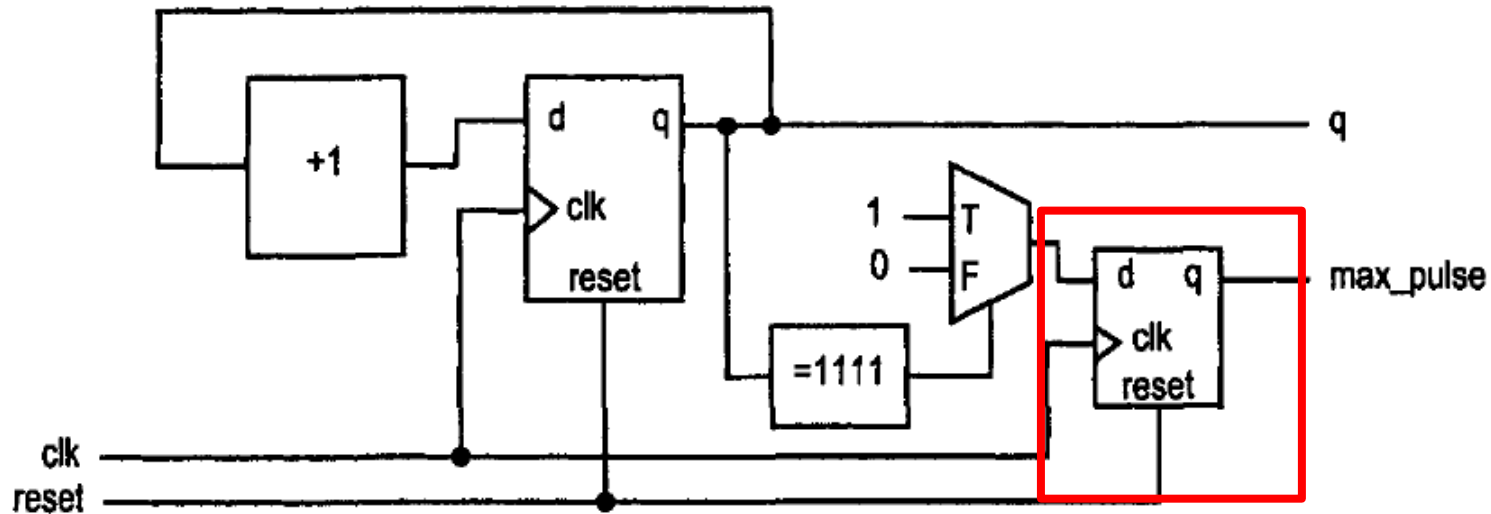
Segment: Binary Counter

```
architecture not_work_one_seg_glitch_arch
    of binary_counter4_pulse is
    signal r_reg: unsigned(3 downto 0);
begin
    process (clk, reset)
    begin
        if (reset='1') then
            r_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            r_reg <= r_reg + 1;
            if r_reg="1111" then
                max_pulse <= '1';
            else
                max_pulse <= '0';
            end if;
        end if;
    end process;
```

*What will be
the circuit?*



Segment: Binary Counter



- A 1-bit register is inferred for the `max_pulse` signal.
- The register works as a buffer and *delays the output by one clock cycle*,
- and thus the `max_pulse` signal will be asserted when `r_reg="0000"`.



Segment: Summary

□ Two-segment code

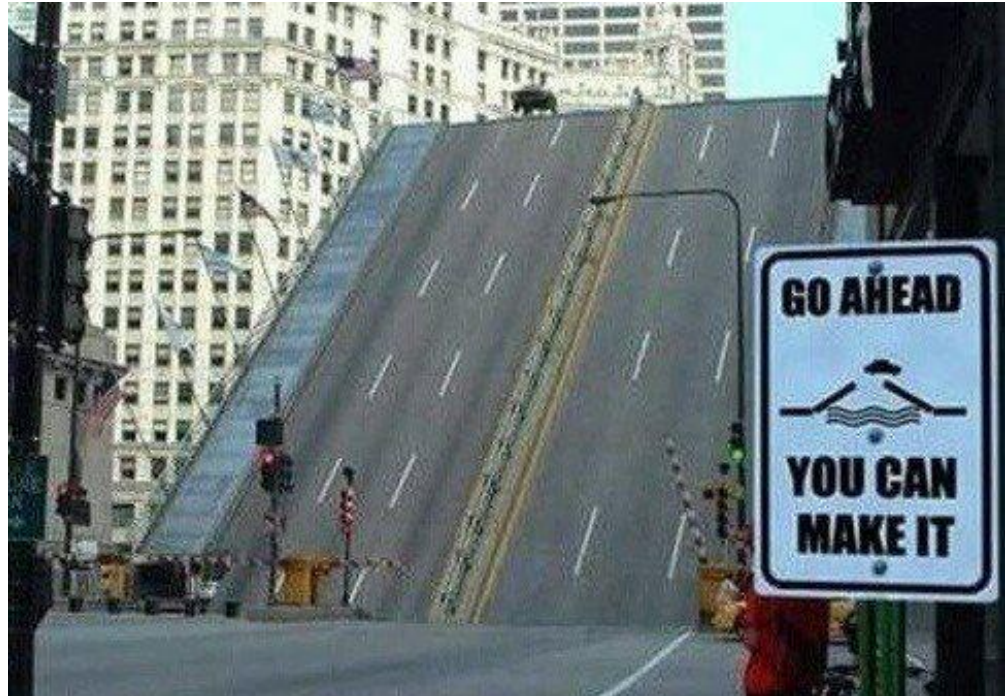
- Separate memory segment from the rest
- Has a clear mapping to hardware component
- *Is preferred and recommended*

□ One-segment code

- Mix memory segment and next-state logic/output logic
- Can sometimes be more compact
- No clear hardware mapping
- Error prone



Segment: Summary



□ Keep the *hardware and the corresponding coding rule* in mind and then go ahead!

- Signals inside the `clk'event` and `clk='1'` branch are referred as registers



Outline

□ Inference of Basic Storage Element

□ Some Design Examples

- DFF with enable
- Register shifter
- Counter

□ Coding Style: Segment

□ **Variables in Sequential Circuit**

□ Poor Design Examples



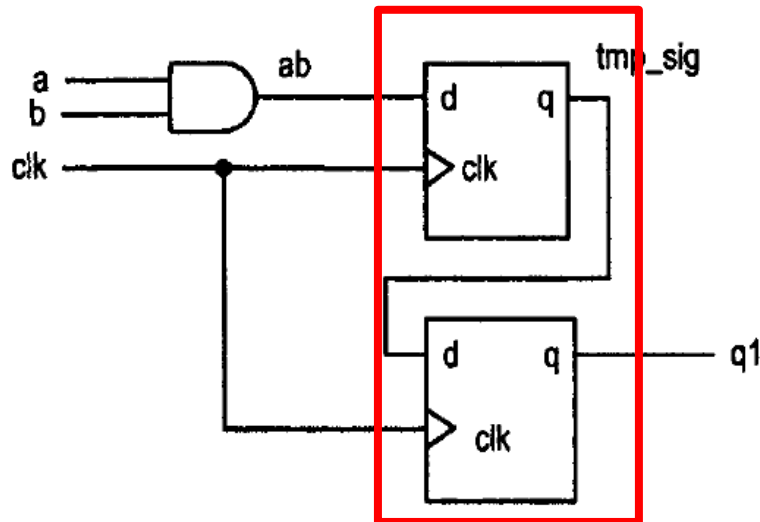
Variables in Sequential Circuit

- Signals **always imply an FF** under `clk'event` and `clk='1'` condition
- When you don't want to infer an **FF** in a **one-segment** process
- Variable is **local** in a process and is not needed outside
- Variable may imply differently
 - Variable is used **after** it is assigned: get a value every time when the process is invoked
 - ***no register is inferred***
 - Variable is used **before** it is assigned: use the value from the previous process execution
 - ***FF or register need to be inferred***



Variables in Sequential Circuit: Example

```
architecture arch of variable_ff_demo is
  signal tmp_sig1: std_logic;
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      tmp_sig1 <= a and b;
      q1 <= tmp_sig1;
    end if ;
  end process;
end architecture;
```



- Registers are inferred
- $q1$ is one clock later than tmp_sig1



Variables in Sequential Circuit: Example

architecture arch of variable_ff_demo is

begin

```
process (clk)
```

```
variable tmp_var2: std_logic; -- declare in process
```

```
begin
```

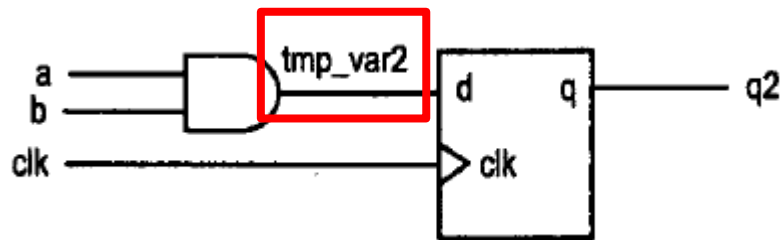
```
if (clk'event and clk='1') then
```

```
tmp_var2 := a and b; -- notice assignment format
```

```
q1 <= tmp_var2;
```

```
end if ;
```

```
end process;
```



□ Use variable

□ *tmp_sig2* is used after it is assigned

□ Just a hard wire, no Reg. is inferred



Variables in Sequential Circuit: Example

architecture arch of variable_ff_demo is

begin

 process (clk)

 variable tmp_var2: std_logic; -- declare in process

 begin

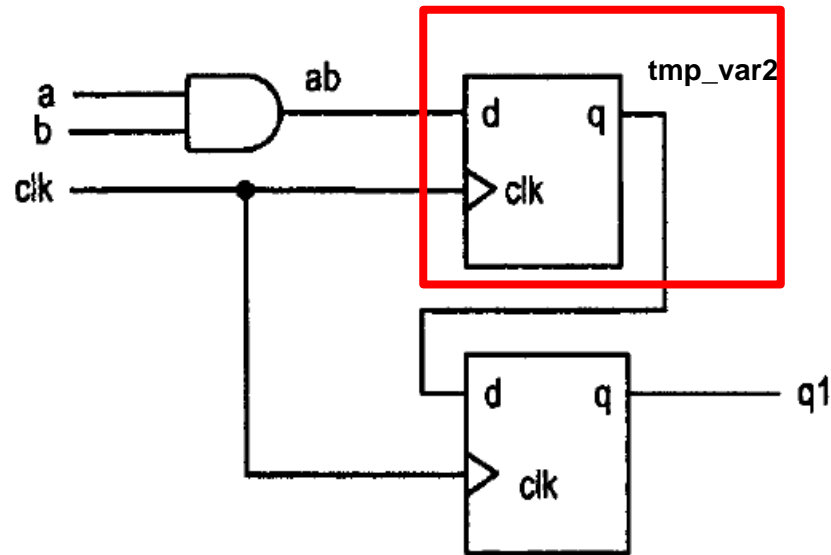
 if (clk'event and clk='1') then

 q1 <= tmp_var2;

 tmp_var2 := a and b; -- change the assignment order

 end if ;

 end process;



2min: Draw the corresponding circuits



Outline

□ Inference of Basic Storage Element

□ Some Design Examples

- DFF with enable
- Register shifter
- Counter

□ Coding Style: Segment

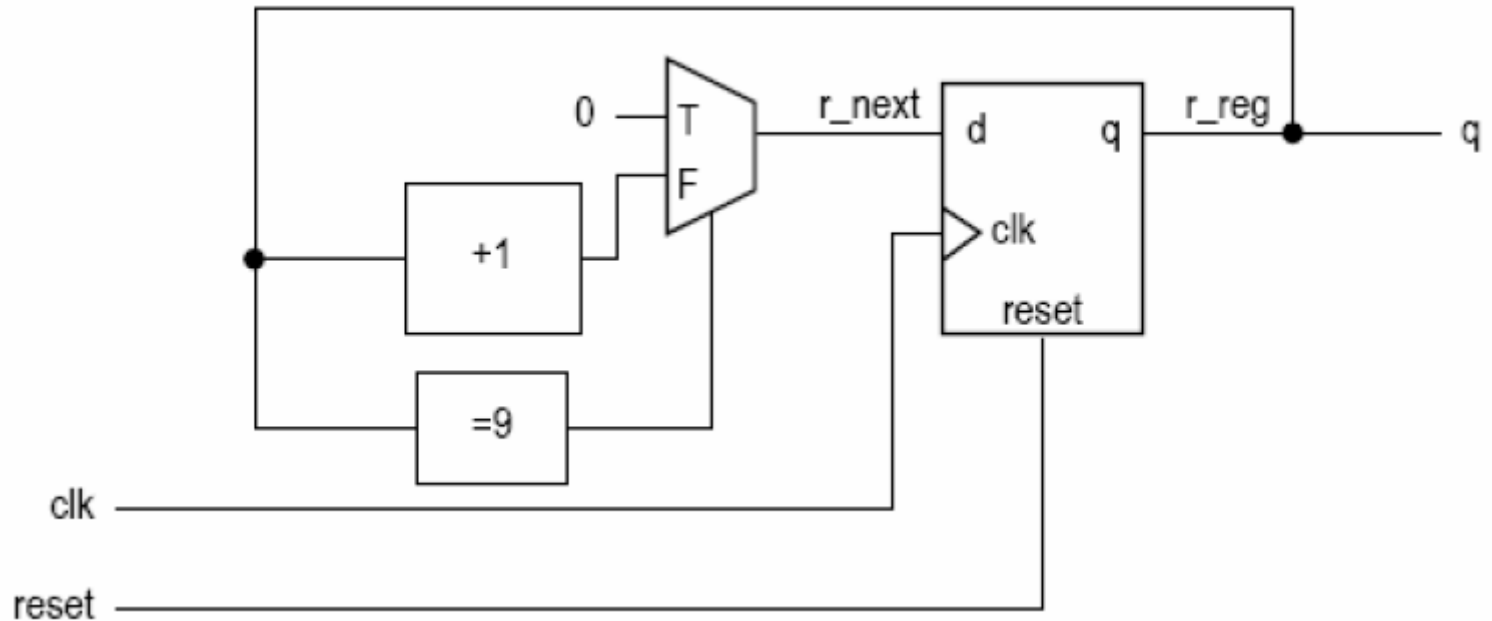
□ Variables in Sequential Circuit

□ Poor Design Examples



Poor Design 1: Misuse of asynchronous reset

Example: a mod-10 counter: 0,1,2 ...,7,8,9, 0,1,2..., 7,8,9,0



Poor Design 1: Misuse of asynchronous reset

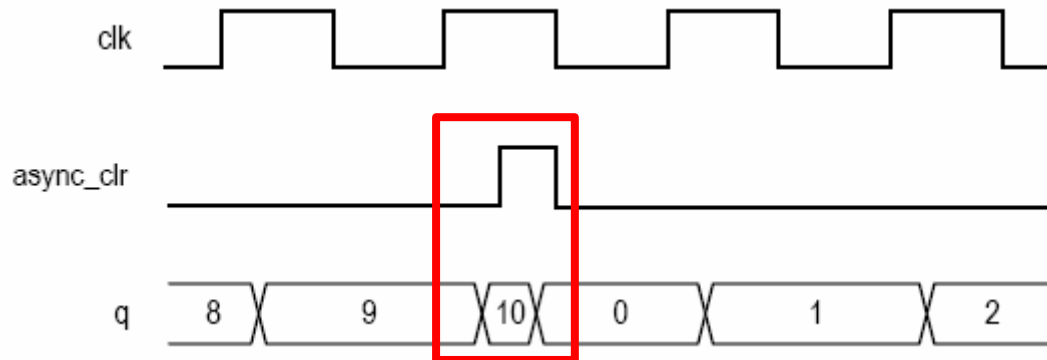
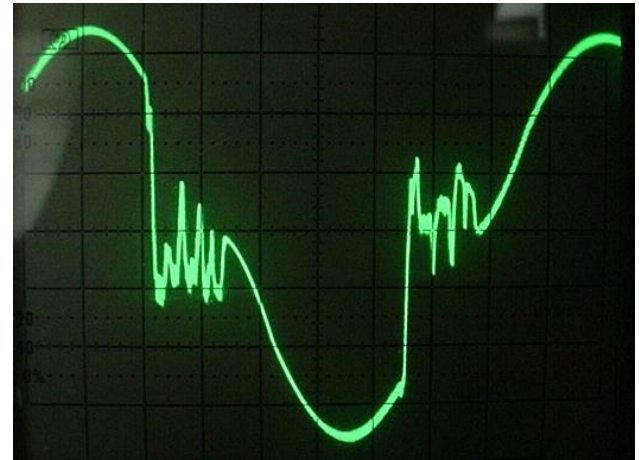
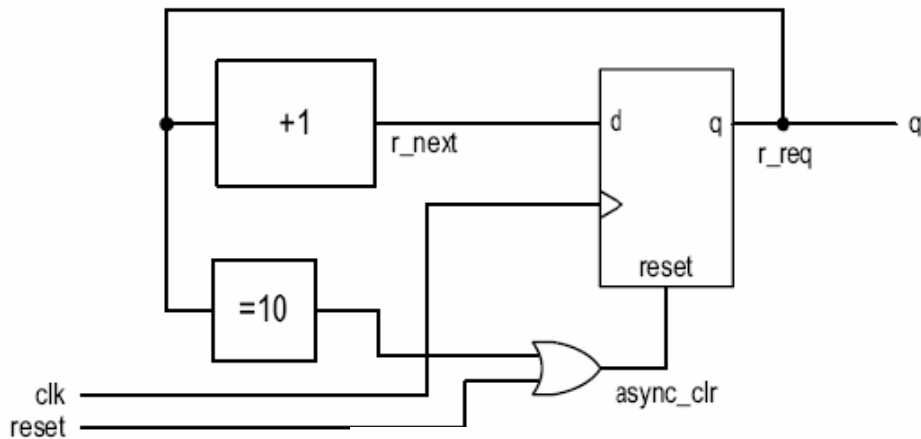
```
library ieee;use ieee.std_logic_1164.all;use ieee.numeric_std.all;
entity mod10_counter is
  port(clk,reset: in std_logic;
        q:out std_logic_vector (3 downto 0));
end mod10_counter;
architecture poor_async_arch of mod10_counter is
  signal r_reg: unsigned (3 downto 0) ;
  signal r_next: unsigned (3 downto 0) ;
  signal async_clr: std_logic;
begin
  process (clk,async_clr)
    begin
      if (async_clr='1') then r_reg <= (others=>'0');
      elsif(clk'event and clk='1') then r_reg<=r_next;
    end if ;
  end process;
  async_clr <='1' when (reset='1'or r_reg="1010") else '0';
  r_next <= r_reg + 1;
  q <= std_logic_vector(r_reg);
end poor_async_arch;
```



Poor Design 1: Misuse of asynchronous reset

Problems

- Glitch in counter: r_reg goes to 10 and then reset, due to the delay of comparator
- **Glitches** in aync_clr can reset the counter
- Timing analysis?



Poor Design 1: Misuse of asynchronous reset

□ Remedy

architecture two_seg of mod10_counter is

```
signal r_reg: unsigned (3 downto 0) ;
```

```
signal r_next: unsigned (3 downto 0) ;
```

```
begin
```

```
  process (clk, reset)
```

```
  begin
```

```
    if (reset = '1') then r_reg <= (others=>'0');
```

```
    elsif (clk'event and clk='1') then r_reg <= r_next;
```

```
    end if ;
```

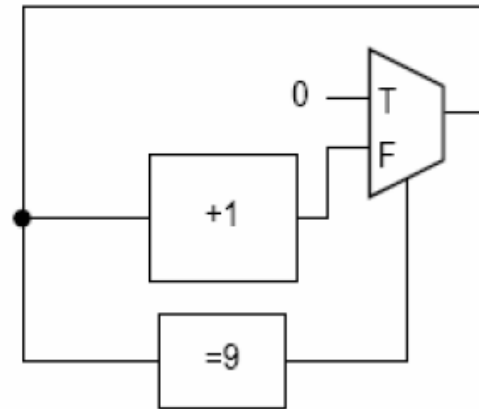
```
  end process;
```

```
  r_next <= (others=>'0') when (r_reg=9) else r_reg+1;
```

```
  q <= std_logic_vector(r_reg);
```

```
end two_seg;
```

Suggestion: Asynchronous reset should only be used for power-on initialization!



Something to remember

- ❑ Strictly follow the synchronous design methodology; i.e., all registers in a system should be ***synchronized by a common global clock signal***.
- ❑ The memory components should be coded clearly so that a predesigned cell can be ***inferred from the device library***.
- ❑ Isolate the memory components from the VHDL description and code them in a ***separate segment***. One-segment coding style is not advisable.
- ❑ ***Asynchronous reset***, if used, should be only for system initialization. It should not be used to clear the registers during regular operation
- ❑ Unless there is a compelling reason, a ***variable should not be used*** to infer a memory component.



Reading advice

RTL Hardware Design Using VHDL: P213-P254



Thanks

