Sequential Circuit Design: Principle

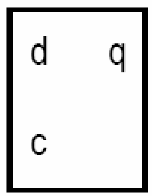
Outline

1. Overview on sequential circuits
2. Synchronous circuits
3. Danger of synthesizing asynchronous circuit
4. Inference of basic memory elements
5. Simple design examples
6. Timing analysis
7. Alternative one-segment coding style
8. Use of variable for sequential circuit

1. Overview on sequential circuit

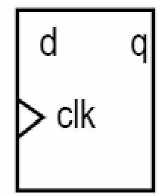
- Combinational vs sequential circuit
  - Sequential circuit: output is a function of current input and state (memory)
- Basic memory elements
  - D latch
  - D FF (Flip-Flop)
  - RAM
- Synchronous vs asynchronous circuit

- D latch: level sensitive
- D FF: edge sensitive



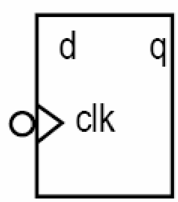
c	q*
0	q
1	d

(a) D latch



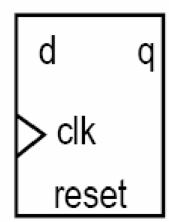
clk	q*
0	q
1	q
↑	d

(b) pos-edge triggered D FF



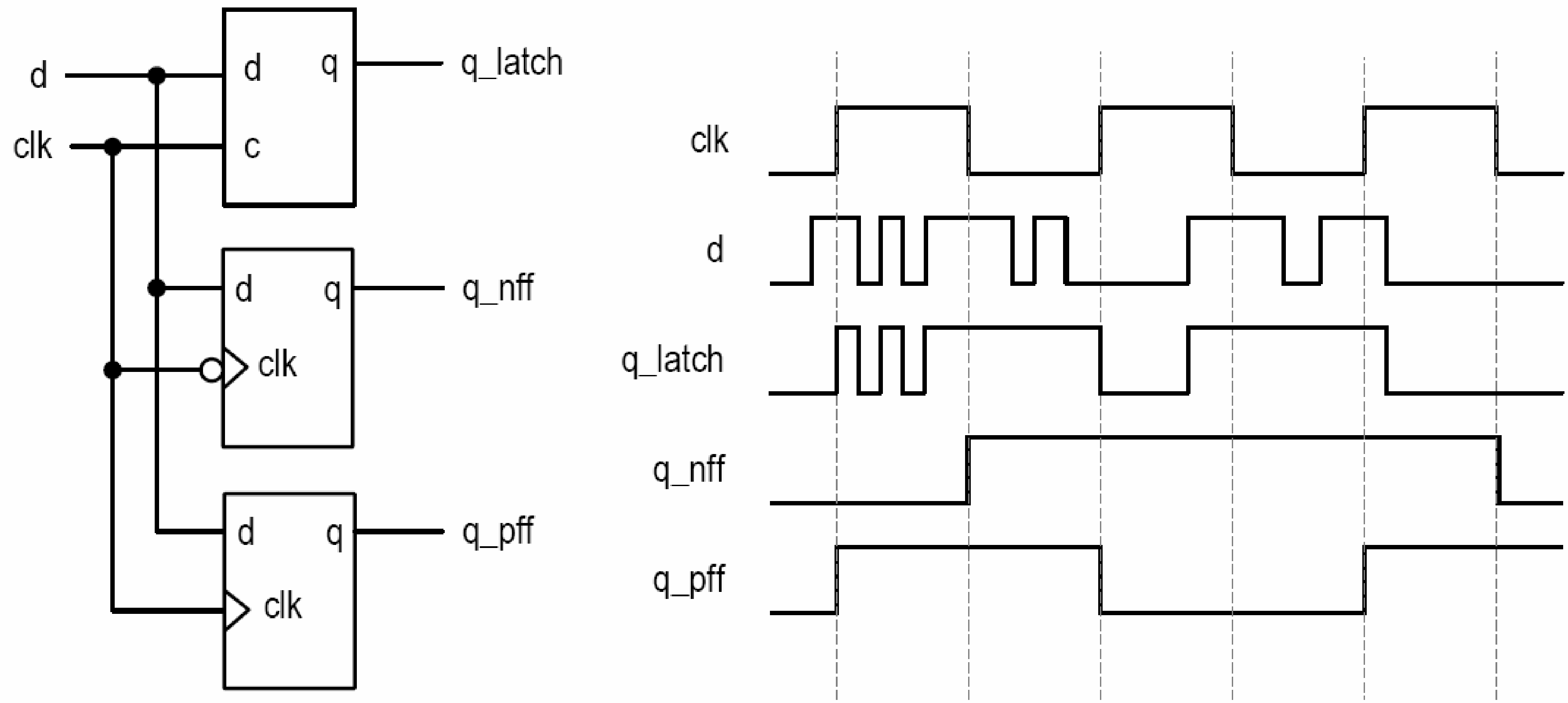
clk	q*
0	q
1	q
↓	d

(c) neg-edge triggered D FF

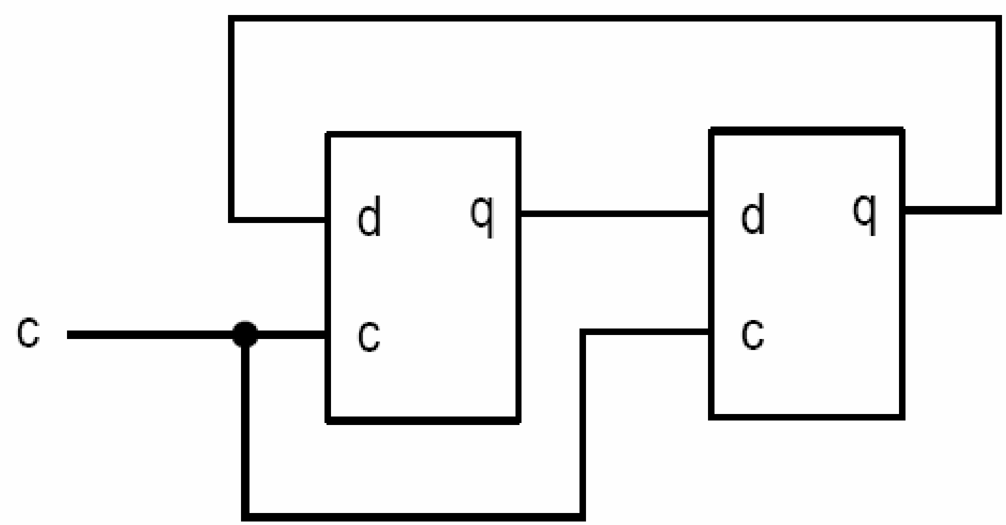


reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↑	d

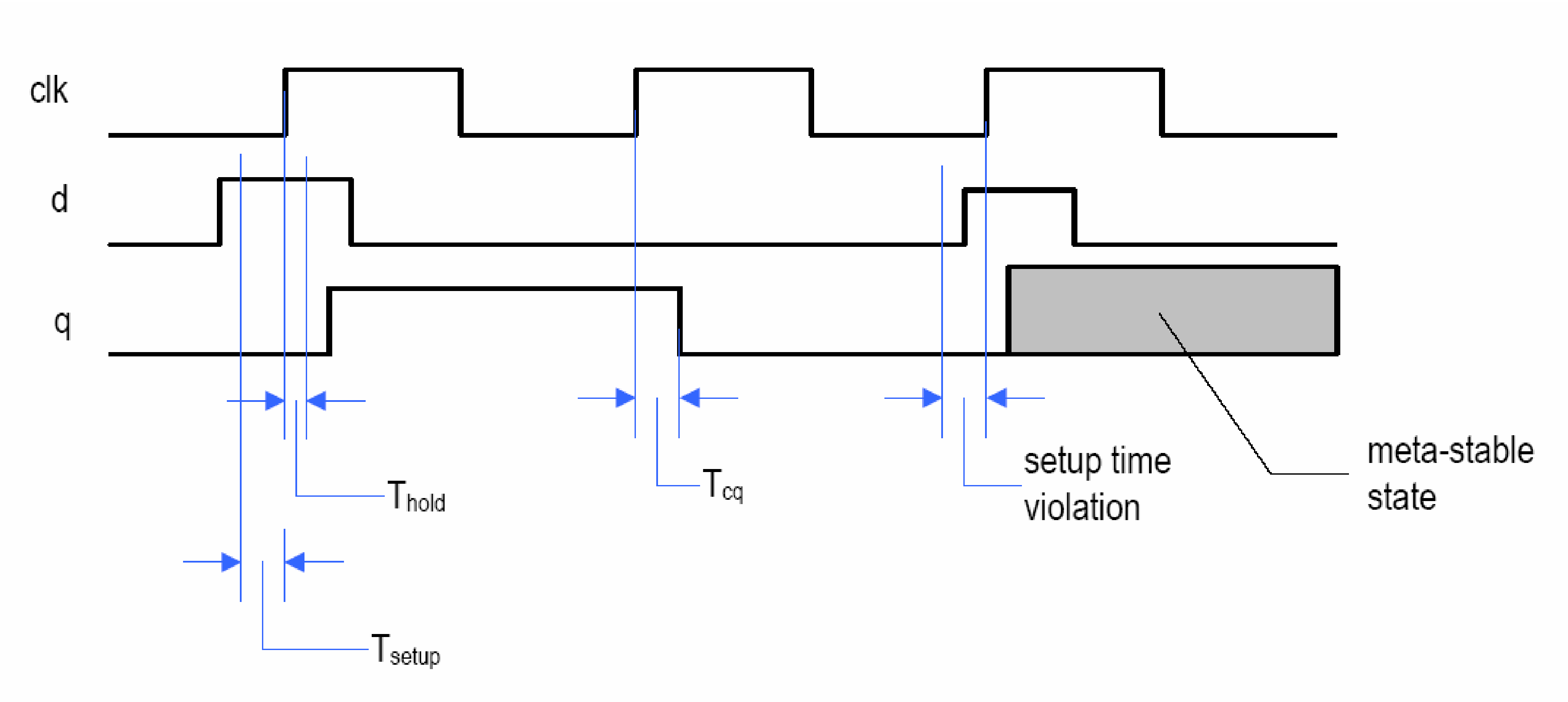
(d) D FF with asynchronous reset


d
clk
d
q
c
q_latch
d
q
clk
q_nff
d
q
clk
q_pff
clk
d
q_latch
q_nff
q_pff

- Problem wit D latch:
  Can the two D latches swap data?



- Timing of a D FF:
  - Clock-to-q delay
  - Constraint: setup time and hold time



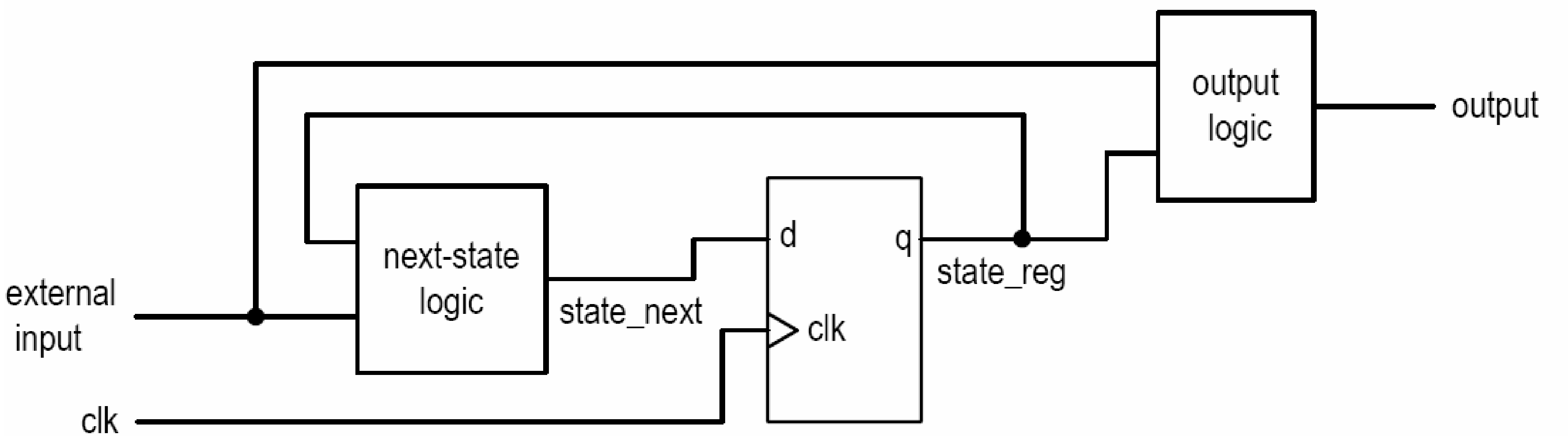
Synch vs asynch circuits

- Globally synchronous circuit: all memory elements (D FFs) controlled (synchronized) by a common global clock signal
- Globally asynchronous but locally synchronous circuit (GALS).
- Globally asynchronous circuit
  - Use D FF but not a global clock
  - Use no clock signal

2. Synchronous circuit

- One of the most difficult design aspects of a sequential circuit:
  How to satisfy the timing constraints
- The Big idea: Synchronous methodology
  - Group all D FFs together with a single clock: Synchronous methodology
  - Only need to deal with the timing constraint of <u>one</u> memory element

- Basic block diagram
  - State register (memory elements)
  - Next-state logic (combinational circuit)
  - Output logic (combinational circuit)
- Operation
  - At the rising edge of the clock, state_next sampled and stored into the register (and becomes the new value of state_reg
  - The next-state logic determines the new value (new state_next) and the output logic generates the output
  - At the rising edge of the clock, the new value of state_next sampled and stored into the register
- Glitches has no effects as long as the state_next is stabled at the sampling edge


output
logic
output
next-state
logic
d
q
state_reg
external
input
state_next
clk
clk

Sync circuit and EDA

- Synthesis: reduce to combinational circuit synthesis
- Timing analysis: involve only a single closed feedback loop (others reduce to combinational circuit analysis)
- Simulation: support “cycle-based simulation”
- Testing: can facilitate scan-chain

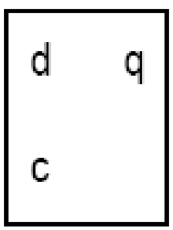
Types of sync circuits

- Not formally defined, Just for coding
- Three types:
  - “Regular” sequential circuit
  - “Random” sequential circuit (FSM)
  - “Combined” sequential circuit (FSM with a Data path, FSMD)

3. Danger of synthesizing asynchronous circuit

- D Latch/DFF
  - Are combinational circuits with feedback loop
  - Design is different from normal combinational circuits (it is delay-sensitive)
  - Should not be synthesized from scratch
  - Should use pre-designed cells from device library

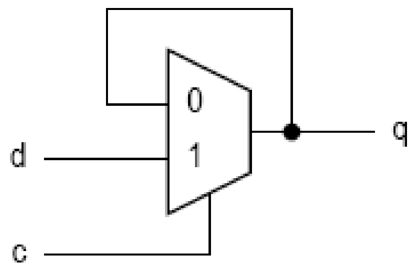
E.g., a D latch from scratch

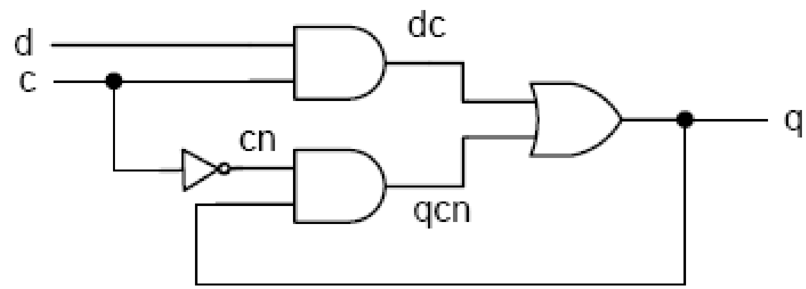


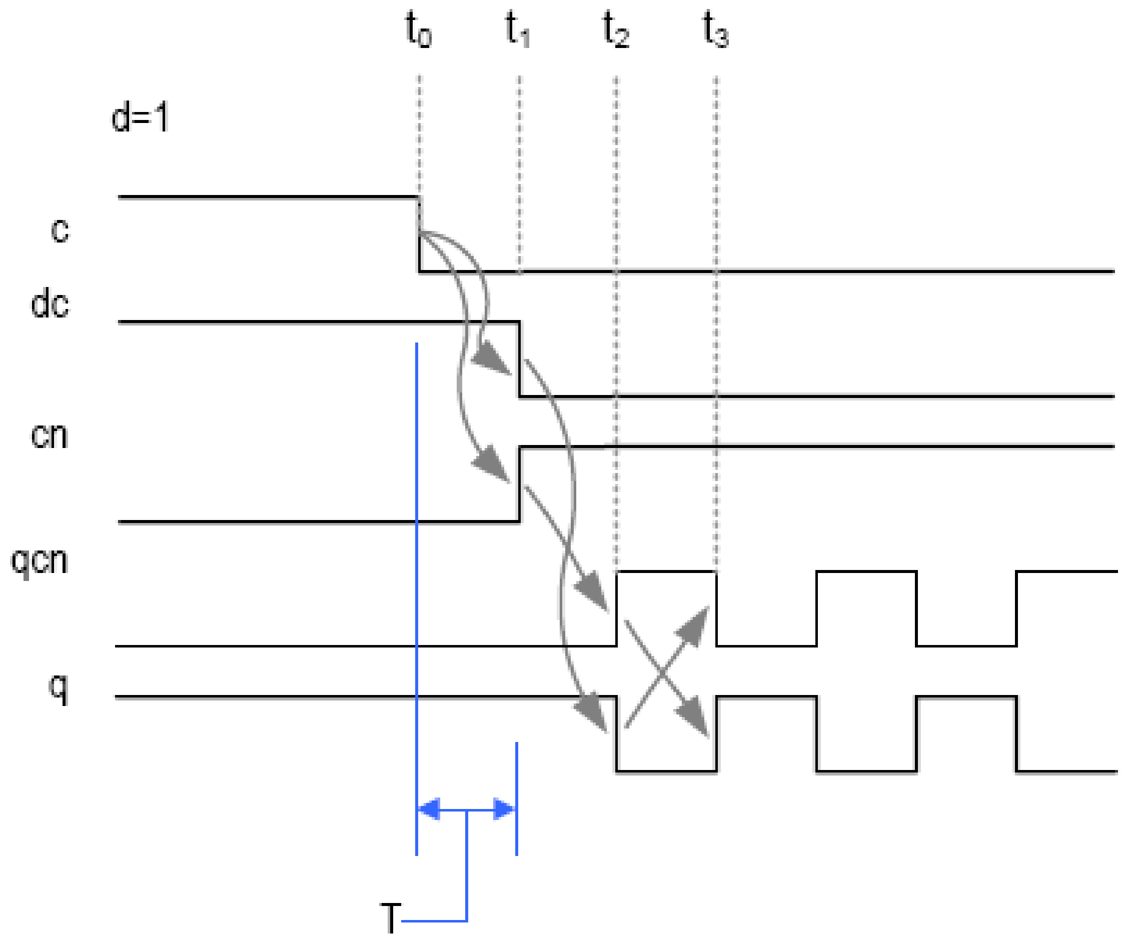
c	q*
0	q
1	d

(a) D latch

```
library ieee;
use ieee.std_logic_1164.all;
entity dlatch is
   port(
      c: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dlatch;

architecture demo_arch of dlatch is
   signal q_latch: std_logic;
begin
   process (c, d, q_latch)
   begin
      if (c='1') then
         q_latch <= d;
      else
         q_latch <= q_latch;
      end if;
   end process;
   q <= q_latch;
end demo_arch;
```


0
1
d
c
q


d
c
dc
cn
qcn
q


t_0 t_1 t_2 t_3
d=1
c
dc
cn
qcn
q
T

4. Inference of basic memory elements

- VHDL code should be clear so that the pre-designed cells can be inferred
- VHDL code
  - D Latch
  - Positive edge-triggered D FF
  - Negative edge-triggered D FF
  - D FF with asynchronous reset

Notes from Xilinx Synthesis and Simuation Design Guide

Xilinx® FPGA devices have abundant flip-flops. FPGA architectures support flip-flops with the following control signals:

- Clock Enable
- Asynchronous Set/Reset
- Synchronous Set/Reset

Notes from Xilinx Synthesis and Simuation Design Guide

Xilinx® FPGA devices have abundant flip-flops. FPGA architectures support flip-flops with the following control signals:

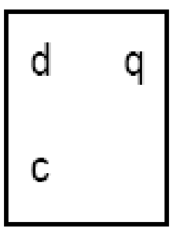
- Clock Enable
- Asynchronous Set/Reset
- Synchronous Set/Reset

All synthesis tools targeting Xilinx FPGA devices are capable to infer registers with all mentioned above control signals. For more information on control signal usage in FPGA design, see Control Signals.

In addition, the value of a flip-flop at device start-up can be set to a logical value **`0`** or **`1`**. This is known as the initialization state, or **`INIT`**.

D Latch

- No else branch
- D latch will be inferred



c	q*
0	q
1	d

(a) D latch

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

From Xilinx:

Synthesizers infer latches from incomplete conditional expressions, such as:

- An **`if`** statement without an **`else`** clause
- An intended register without a rising edge or falling edge construct

If **Statement Without an** ***else*** **Clause VHDL Coding Example**

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

If Statement Without an *else* Clause VHDL Coding Example

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

Many times this is done by mistake. The design may still appear to function properly in simulation. This can be problematic for FPGA designs, since timing for paths containing latches can be difficult to analyze. Synthesis tools usually report in the log files when a latch is inferred to alert you to this occurrence.

Xilinx® recommends that you avoid using latches in FPGA designs, due to the more difficult timing analyses that take place when latches are used.

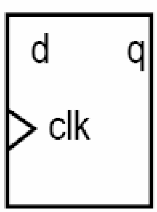
Some synthesis tools can determine the number of latches in your design.

For more information, see your synthesis tool documentation.

You should convert all **`if`** statements without corresponding **`else`** statements and without a clock edge to registers or logic gates. Use the recommended coding styles in the synthesis tool documentation to complete this conversion.

Pos edge-triggered D FF

- No else branch
- Note the sensitivity list

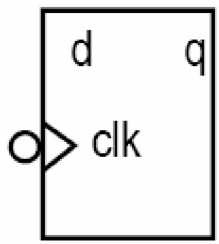


clk	q*
0	q
1	q
↑	d

(b) pos-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='1') then
         q <= d;
      end if;
   end process;
end arch;
```

- Neg edge-triggered D FF

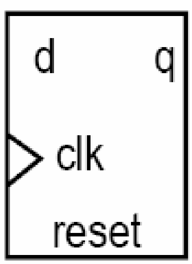


clk	q*
0	q
1	q
↓	d

```
if (clk'event and clk='0') then
```

D FF with async reset

- No else branch
- Note the sensitivity list



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	$\uparrow$	d

```
library ieee;
use ieee.std_logic_1164.all;
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') th
         q <= d;
      end if;
   end process;
end arch;
```

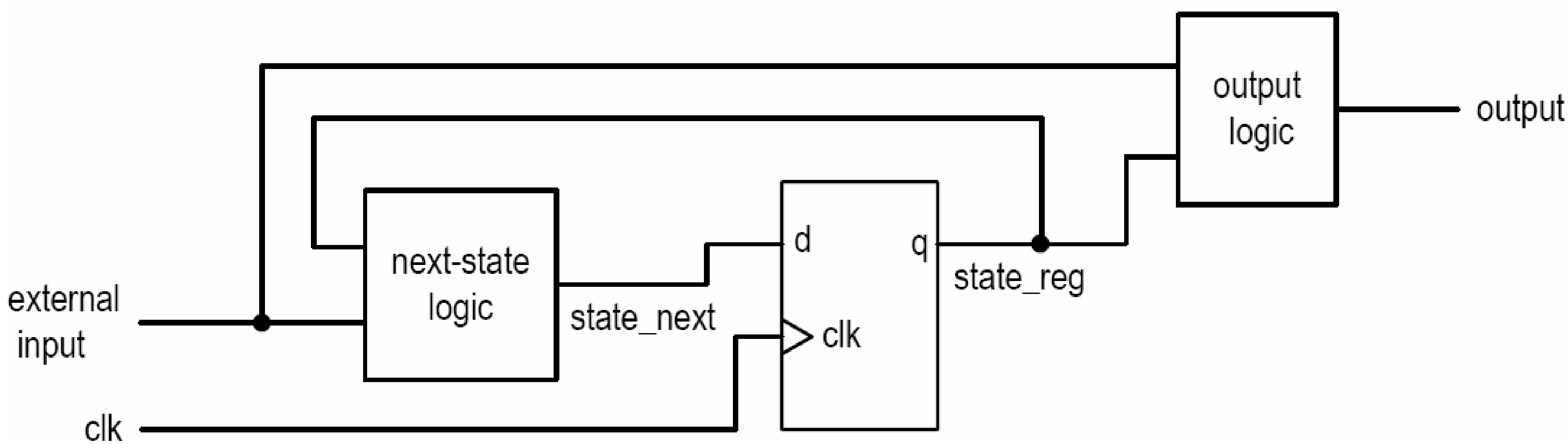
Register

- Multiple D FFs with same clock and reset

```
library ieee;
use ieee.std_logic_1164.all;
entity reg8 is
   port(
      clk: in std_logic;
      reset: in std_logic;
      d: in std_logic_vector(7 downto 0);
      q: out std_logic_vector(7 downto 0)
   );
end reg8;
architecture arch of reg8 is
begin
   process (clk,reset)
   begin
      if (reset='1') then
         q <=(others=>'0');
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

5. Simple design examples

- Follow the block diagram
  - Register
  - Next-state logic (combinational circuit)
  - Output logic (combinational circuit)

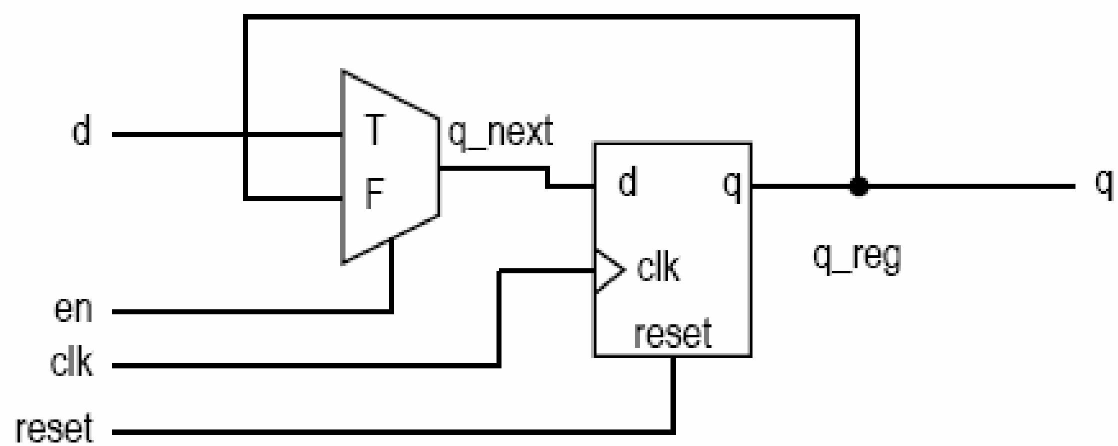


D FF with sync enable

- Note that the en is controlled by clock
- Note the sensitivity list

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

(a) Function table



(b) Conceptual diagram

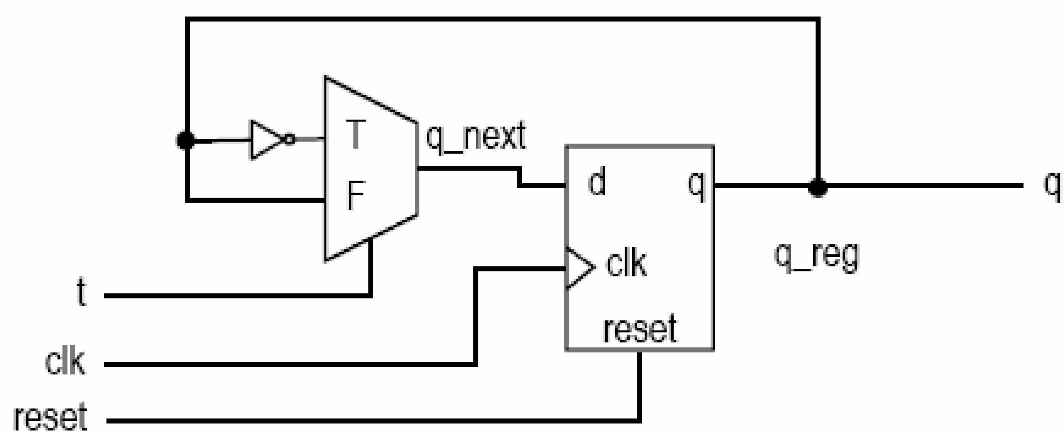
```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity dff_en is
   port(
      clk: in std_logic;
      reset: in std_logic;
      en: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end dff_en;
```

```
architecture two_seg_arch of dff_en is
   signal q_reg: std_logic;
   signal q_next: std_logic;
begin
   -- a 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;
```

T FF

reset	clk	t	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↑	0	q
0	↑	1	q'

(a) Function table

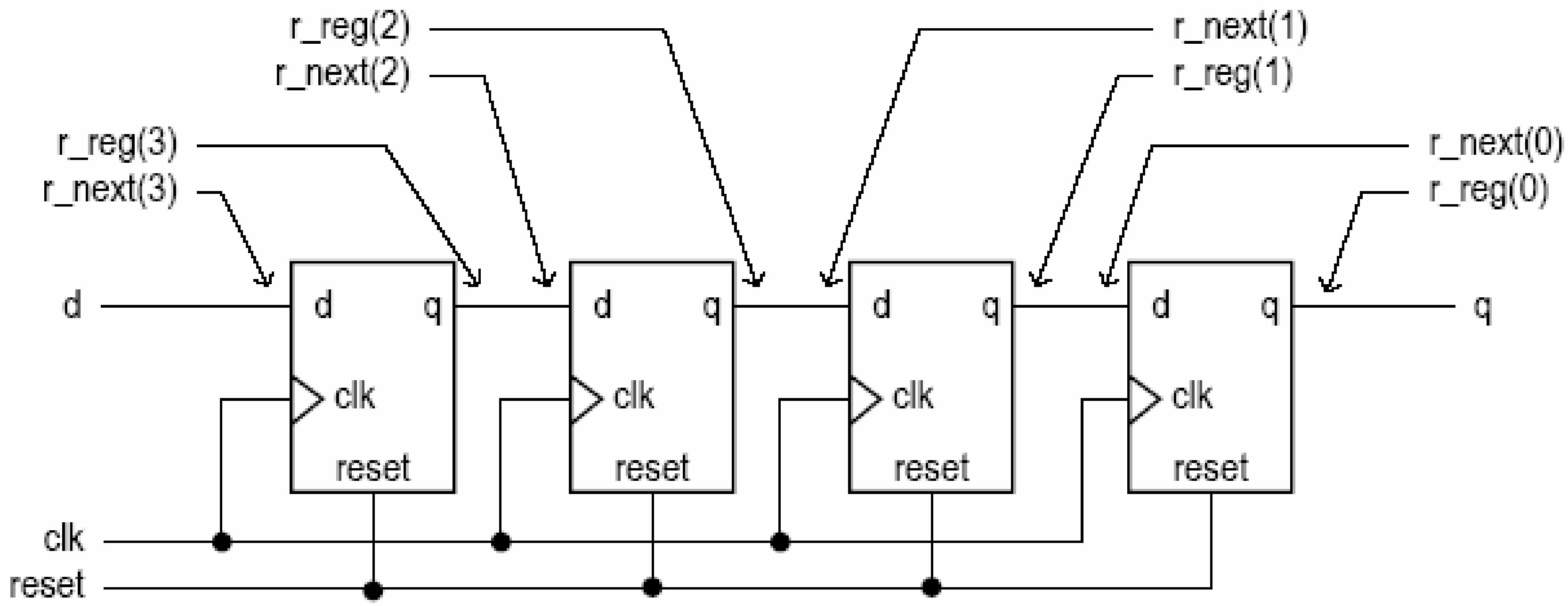


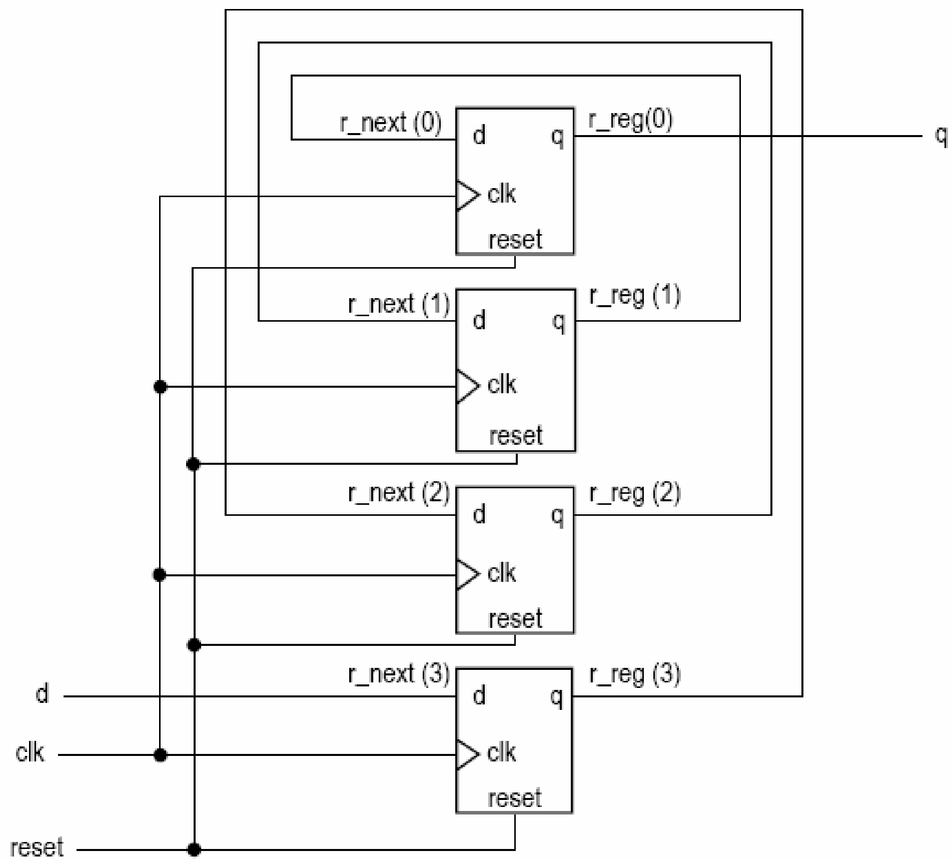
(b) Conceptual diagram

```
library ieee;
use ieee.std_logic_1164.all;
entity tff is
   port(
      clk: in std_logic;
      reset: in std_logic;
      t: in std_logic;
      q: out std_logic
   );
end tff;

architecture two_seg_arch of tff is
   signal q_reg: std_logic;
   signal q_next: std_logic;
```

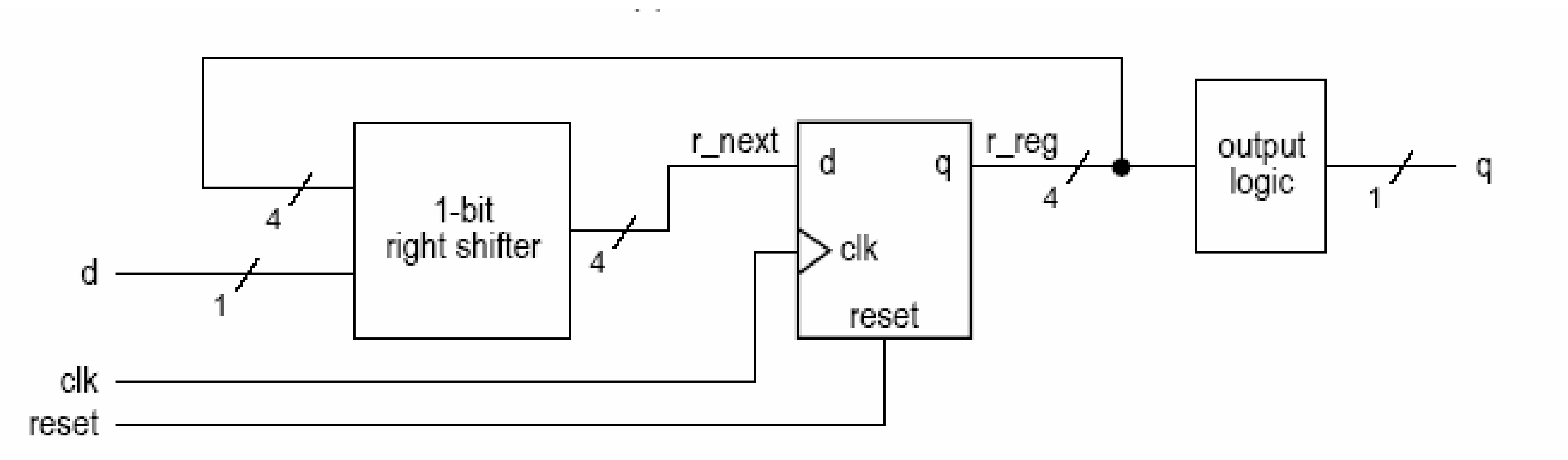
```
architecture two_seg_arch of tff is
   signal q_reg: std_logic;
   signal q_next: std_logic;
begin
   -- a D FF
   process (clk,reset)
   begin
      if (reset='1') then
         q_reg <= '0';
      elsif (clk'event and clk='1') then
         q_reg <= q_next;
      end if;
   end process;
   -- next-state logic
   q_next <= q_reg when t='0' else
             not(q_reg);
   -- output logic
   q <= q_reg;
end two_seg_arch;
```

Free-running shift register





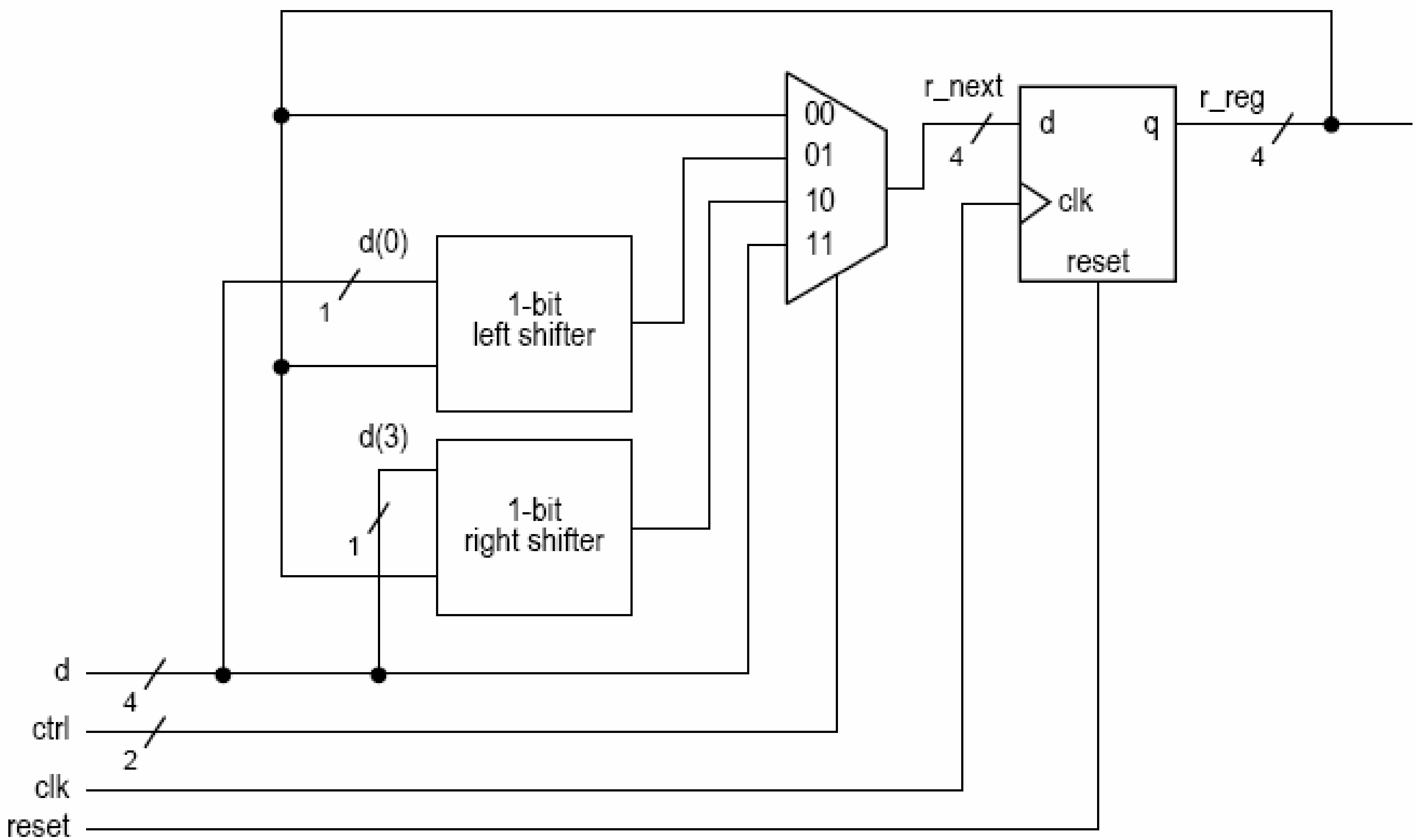
(a) Vertical form


1-bit
right shifter
r_next
d
q
clk
reset
r_reg
output
logic
q
d
clk
reset
4
1
4
4
1

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity shift_right_register is
   port(
      clk, reset: in std_logic;
      d: in std_logic;
      q: out std_logic;
end shift_right_register;

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
   -- register
   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;
   -- next-state logic (shift right 1 bit)
   r_next <= d & r_reg(3 downto 1);
   -- output
   q <= r_reg(0);
end two_seg_arch;
```

Universal shift register

- 4 ops: parallel load, shift right, shift left, pause



by P. Chu

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_register is
   port(
      clk, reset: in std_logic;
      ctrl: in std_logic_vector(1 downto 0);
      d: in std_logic_vector(3 downto 0);
      q: out std_logic_vector(3 downto 0));
end shift_register;

architecture two_seg_arch of shift_register is
   signal r_reg: std_logic_vector(3 downto 0);
   signal r_next: std_logic_vector(3 downto 0);
```

```vhdl
begin
   -- register
   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;
   -- next-state logic
   with ctrl select
      r_next <=
         r_reg                          when "00", --no op
         r_reg(2 downto 0) & d(0) when "01", --shift left;
         d(3) & r_reg(3 downto 1) when "10", --shift right;
         d                              when others;
   -- output
   q <= r_reg;
end two_seg_arch;
```

Arbitrary sequence counter

input pattern	output pattern
000	011
011	110
110	101
101	111
111	000

```
entity arbi_seq_counter4 is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(2 downto 0)
      );
end arbi_seq_counter4;

architecture two_seg_arch of arbi_seq_counter4 is
   signal r_reg: std_logic_vector(2 downto 0);
   signal r_next: std_logic_vector(2 downto 0);
begin
```

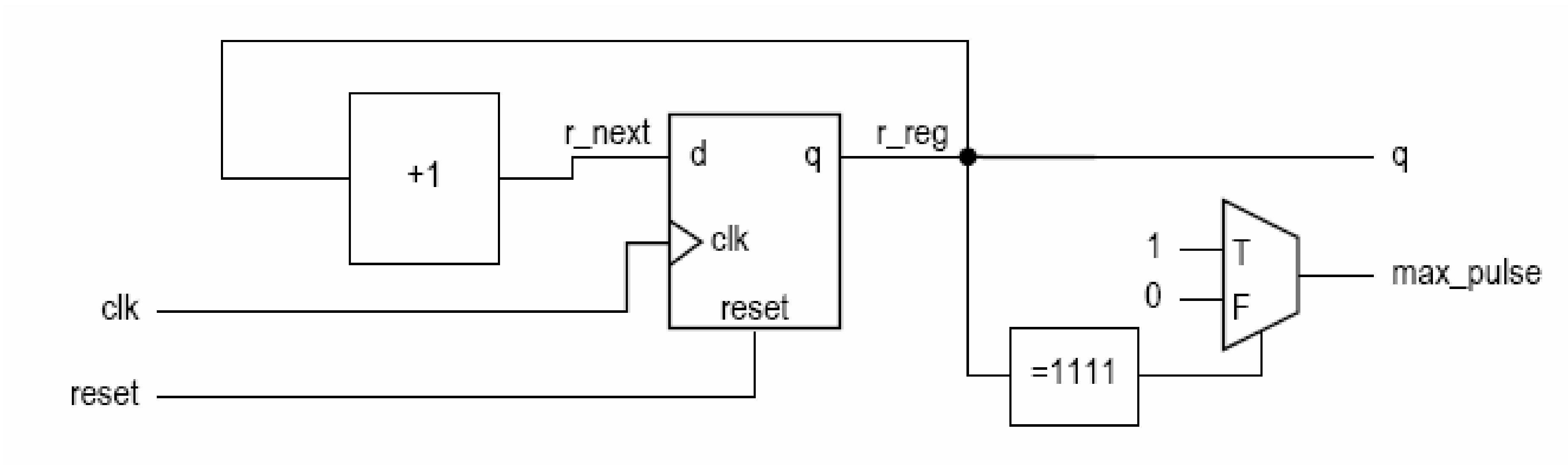
```
begin
   -- register
   process (clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= "011" when r_reg="000" else
             "110" when r_reg="011" else
             "101" when r_reg="110" else
             "111" when r_reg="101" else
             "000";  -- r_reg="111"
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

Free-running binary counter

- Count in binary sequence
- With a max_pulse output: asserted when counter is in “11…11” state

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
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;
```

```vhdl
architecture two_seg_arch of binary_counter4_pulse is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   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;
   -- next-state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_pulse <= '1' when r_reg="1111" else
                '0';
end two_seg_arch;
```



- Wrapped around automatically
- Poor practice:

Binary counter with bells & whistles

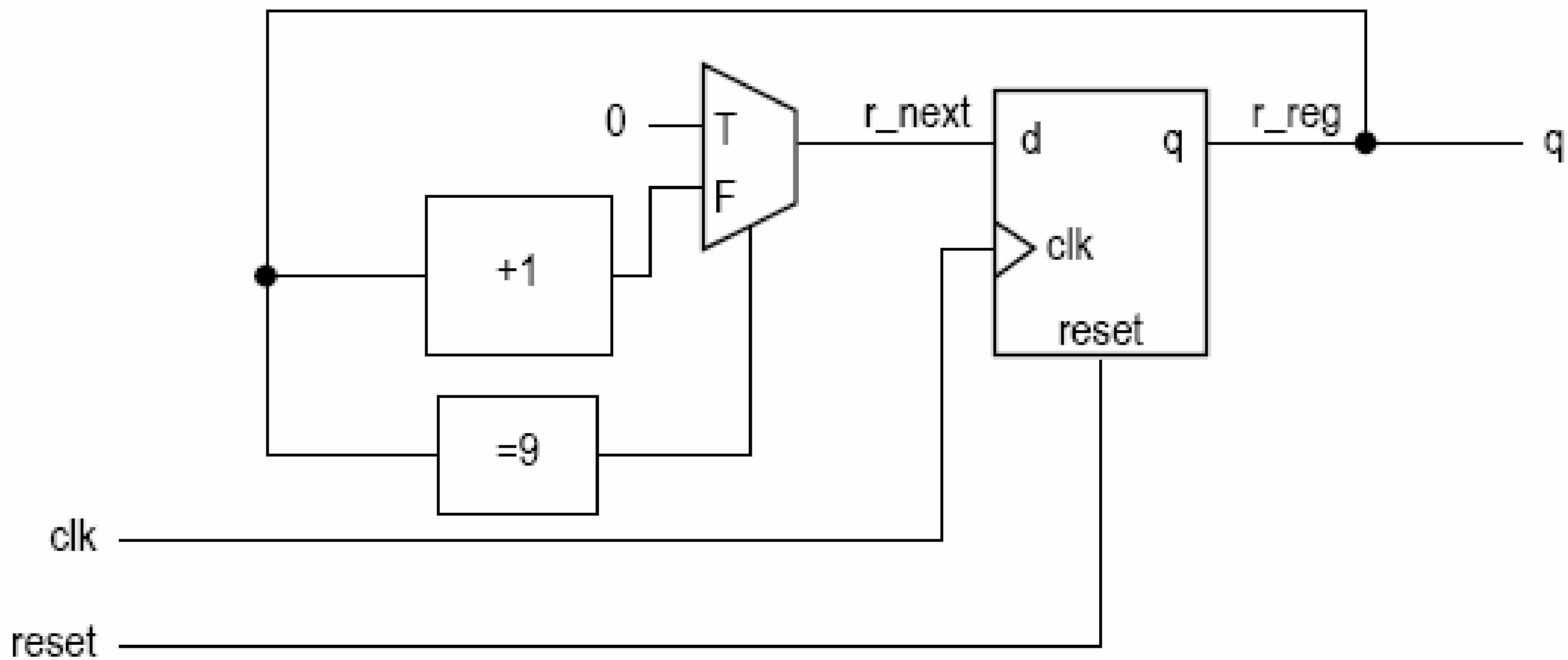
syn_clr	load	en	q*	operation
1	–	–	00 ··· 00	synchronous clear
0	1	–	d	parallel load
0	0	1	q+1	count
0	0	0	q	pause

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter4_feature is
   port(
      clk, reset: in std_logic;
      syn_clr, en, load: std_logic;
      d: std_logic_vector(3 downto 0);
      q: out std_logic_vector(3 downto 0)
      );
end binary_counter4_feature;
```

```vhdl
architecture two_seg_arch of binary_counter4_feature is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   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;
   -- next-state logic
   r_next <= (others=>'0') when syn_clr='1' else
             unsigned(d)   when load='1' else
             r_reg + 1     when en ='1' else
             r_reg;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

Decade (mod-10) counter

```vhdl
architecture two_seg_arch of mod10_counter is
   constant TEN: integer := 10;
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   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;
   -- next-state logic
   r_next <= (others=>'0') when r_reg=(TEN-1) else
             r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

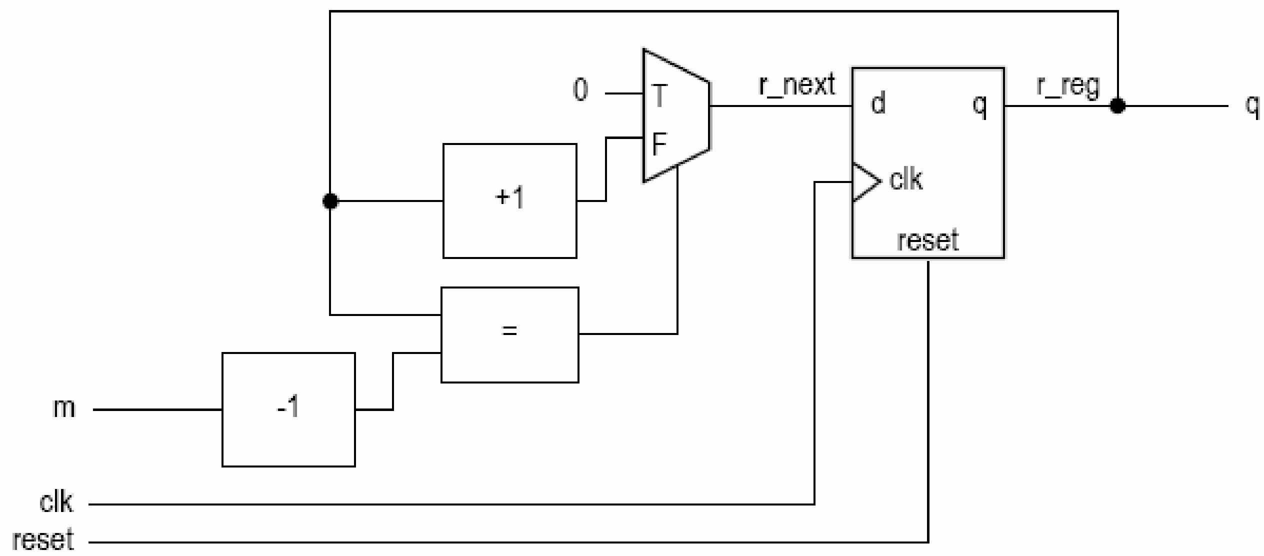

0
T
F
+1
=9
r_next
d
q
clk
reset
r_reg
q
clk
reset

Programmable mod-m counter

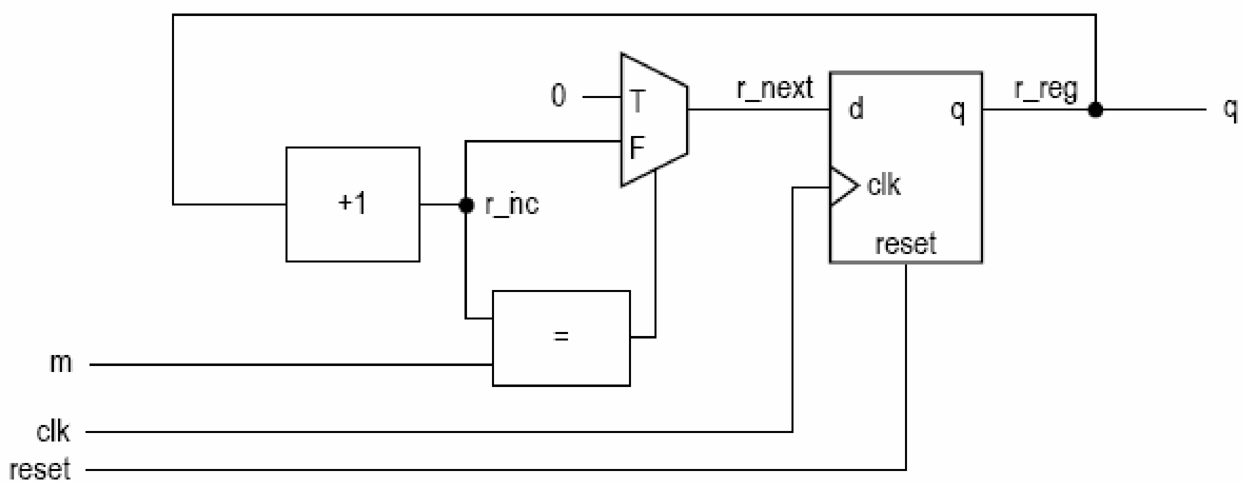
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity prog_counter is
   port(
      clk, reset: in std_logic;
      m: in std_logic_vector(3 downto 0);
      q: out std_logic_vector(3 downto 0)
      );
end prog_counter;

architecture two_seg_clear_arch of prog_counter is
```

```
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   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;
   -- next-state logic
   r_next <= (others=>'0') when r_reg=(unsigned(m)-1) else
             r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_clear_arch;
```

```
architecture two_seg_effi_arch of prog_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next, r_inc: unsigned(3 downto 0);
begin
   -- register
   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;
   -- next-state logic
   r_inc <= r_reg + 1;
   r_next <= (others=>'0') when r_inc=unsigned(m) else
             r_inc;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_effi_arch;
```



(a) Block diagram of initial design

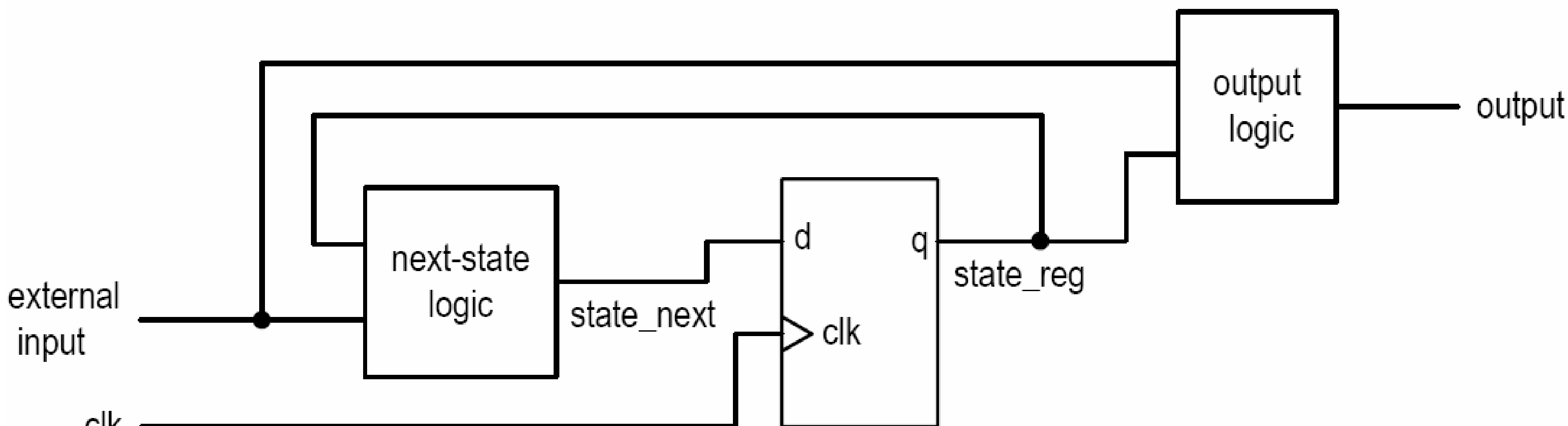


(b) Block diagram of more efficient design

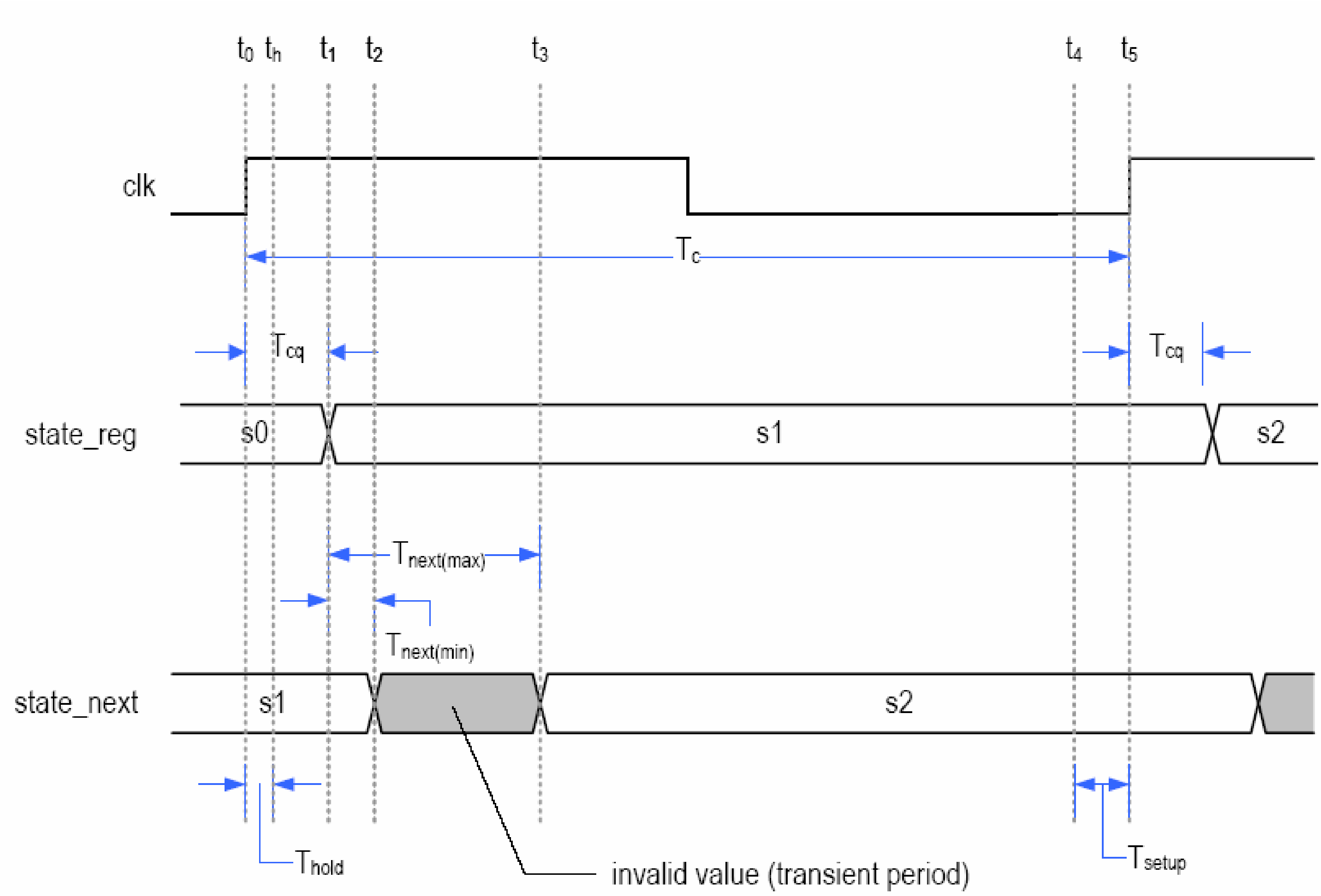
6. Timing analysis

- Combinational circuit:
  - characterized by propagation delay
- Sequential circuit:
  - Has to satisfy setup/hold time constraint
  - Characterized by maximal clock rate (e.g., 200 MHz counter, 2.4 GHz Pentium II)
  - Setup time and clock-to-q delay of register and the propagation delay of next-state logic are embedded in clock rate

- state_next must satisfy the constraint
- Must consider effect of
  - state_reg: can be controlled
  - synchronized external input (from a subsystem of same clock)
  - unsynchronized external input
- Approach
  - First 2: adjust clock rate to prevent violation
  - Last: use “synchronization circuit” to resolve violation



- Setup time violation and maximal clock rate



$$t_3 = t_0 + T_{cq} + T_{next(max)}$$

$$t_4 = t_5 - T_{setup} = t_0 + T_c - T_{setup}$$

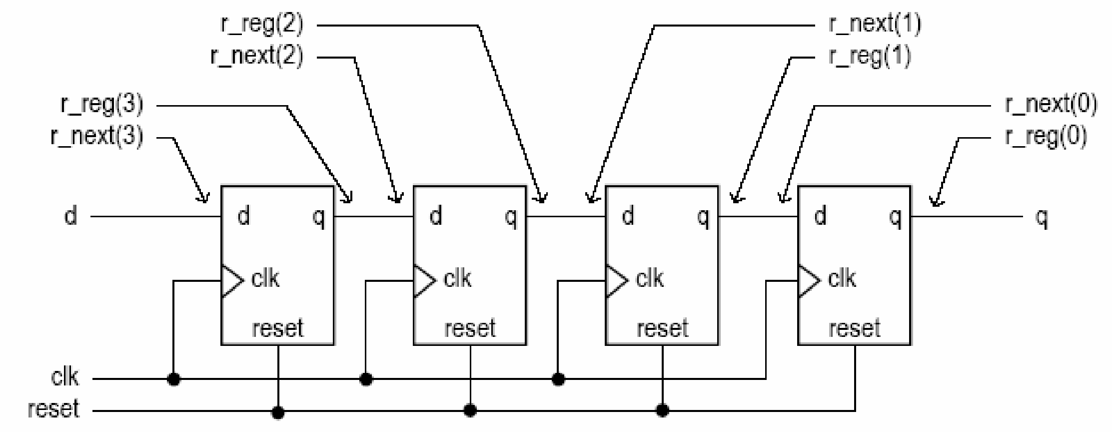
$$t_3 < t_4$$

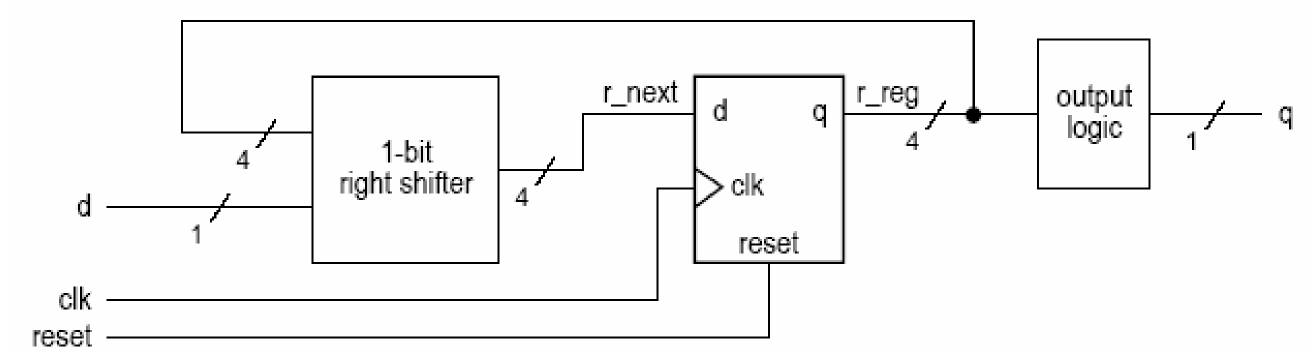
$$t_0 + T_{cq} + T_{next(max)} < t_0 + T_c - T_{setup}$$

$$T_{cq} + T_{next(max)} + T_{setup} < T_c$$

$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup}$$

- E.g., shift register; let Tcq=1.0ns Tsetup=0.5ns

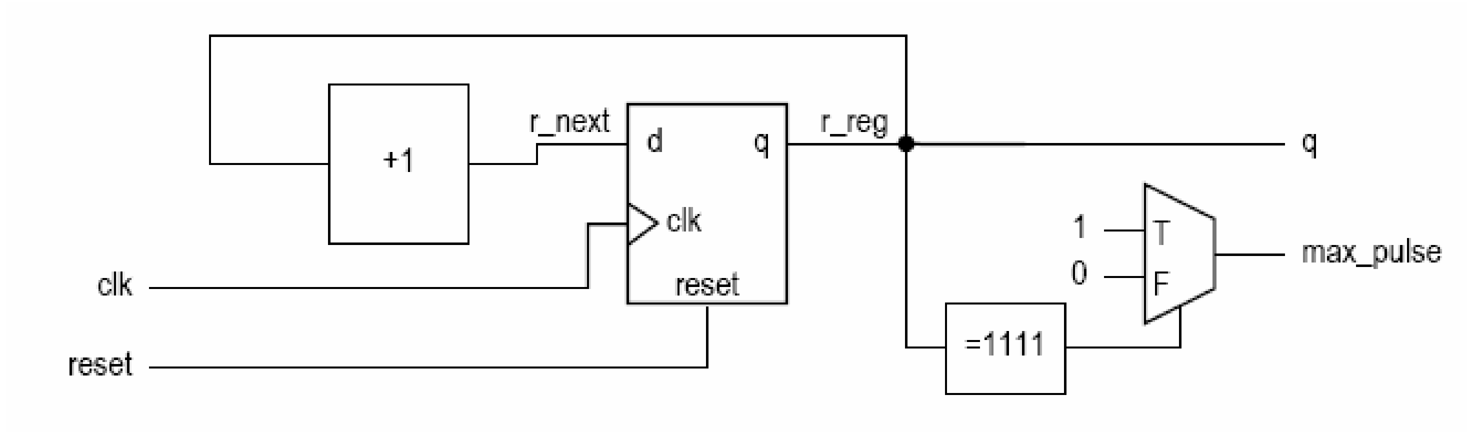




$$T_{c(min)} = T_{cq} + T_{setup} = 1.5 \text{ ns}$$

$$f_{max} = \frac{1}{T_{cq} + T_{setup}} = \frac{1}{1.5 \text{ ns}} \approx 666.7 \text{ MHz}$$

- E.g., Binary counter; let Tcq=1.0ns Tsetup=0.5ns



width	nand	xor	$>_a$	$>_d$	=	$+1_a$	$+1_d$	$+_a$	$+_d$	mux
					VHDL operator					
					area (gate count)					
8	8	22	25	68	26	27	33	51	118	21
16	16	44	52	102	51	55	73	101	265	42
32	32	85	105	211	102	113	153	203	437	85
64	64	171	212	398	204	227	313	405	755	171
					delay (ns)					
8	0.1	0.4	4.0	1.9	1.0	2.4	1.5	4.2	3.2	0.3
16	0.1	0.4	8.6	3.7	1.7	5.5	3.3	8.2	5.5	0.3
32	0.1	0.4	17.6	6.7	1.8	11.6	7.5	16.2	11.1	0.3
64	0.1	0.4	35.7	14.3	2.2	24.0	15.7	32.2	22.9	0.3

$$f_{max} = \frac{1}{T_{cq} + T_{8\_bit\_inc(area)} + T_{setup}} = \frac{1}{1 \text{ ns} + 2.4 \text{ ns} + 0.5 \text{ ns}} \approx 256.4 \text{ MHz}$$

$$f_{max} = \frac{1}{T_{cq} + T_{16\_bit\_inc(area)} + T_{setup}} = \frac{1}{1 \text{ ns} + 5.5 \text{ ns} + 0.5 \text{ ns}} \approx 142.9 \text{ MHz}$$

$$f_{max} = \frac{1}{T_{cq} + T_{32\_bit\_inc(area)} + T_{setup}} = \frac{1}{1 \text{ ns} + 11.6 \text{ ns} + 0.5 \text{ ns}} \approx 76.3 \text{ MHz}$$

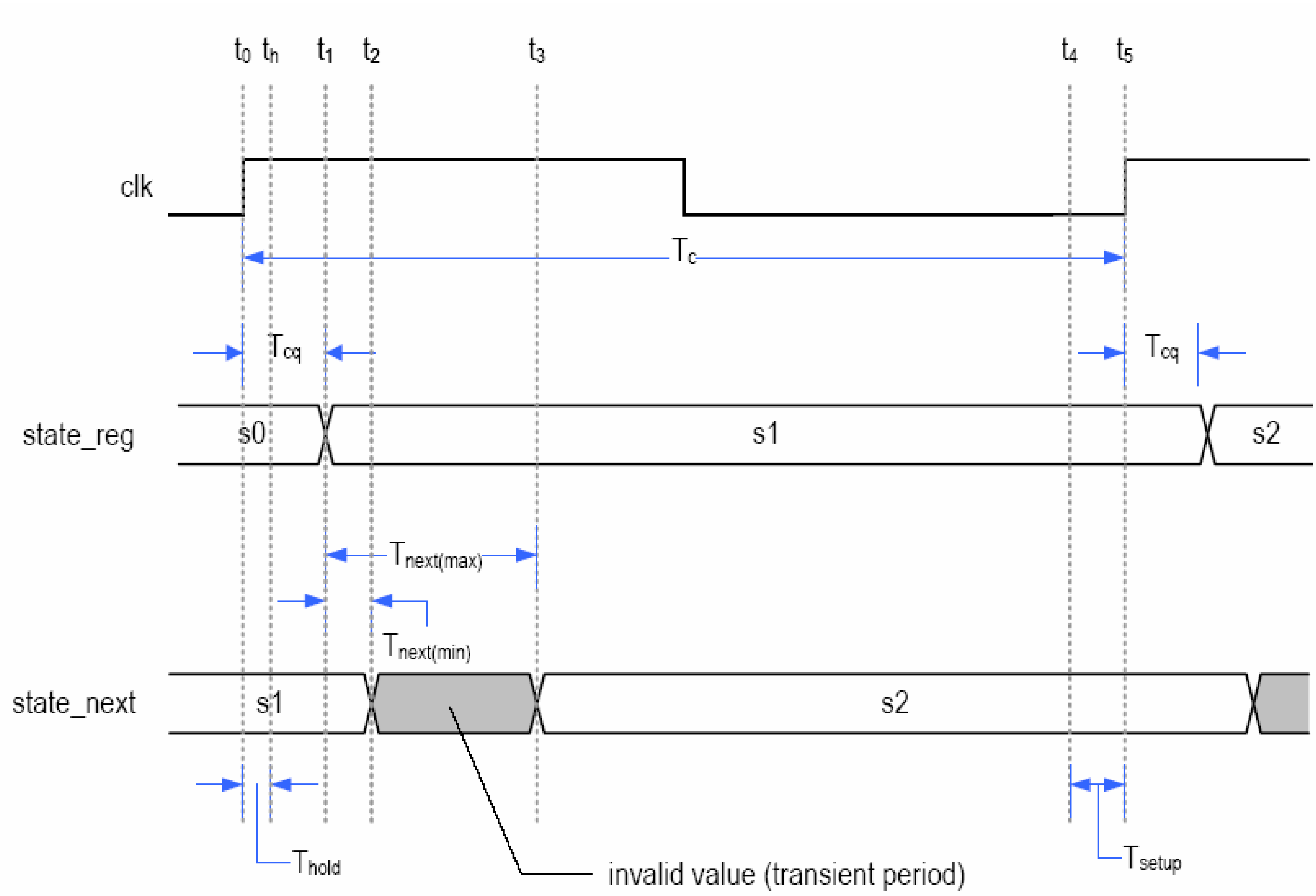
$$f_{max} = \frac{1}{T_{cq} + T_{8\_bit\_inc(delay)} + T_{setup}} = \frac{1}{1 \text{ ns} + 1.5 \text{ ns} + 0.5 \text{ ns}} \approx 333.3 \text{ MHz}$$

$$f_{max} = \frac{1}{T_{cq} + T_{16\_bit\_inc(delay)} + T_{setup}} = \frac{1}{1 \text{ ns} + 3.3 \text{ ns} + 0.5 \text{ ns}} \approx 208.3 \text{ MHz}$$

and

$$f_{max} = \frac{1}{T_{cq} + T_{32\_bit\_inc(delay)} + T_{setup}} = \frac{1}{1 \text{ ns} + 7.5 \text{ ns} + 0.5 \text{ ns}} \approx 111.1 \text{ MHz}$$

- Hold time violation



$$t_2 = t_0 + T_{cq} + T_{next(min)}$$

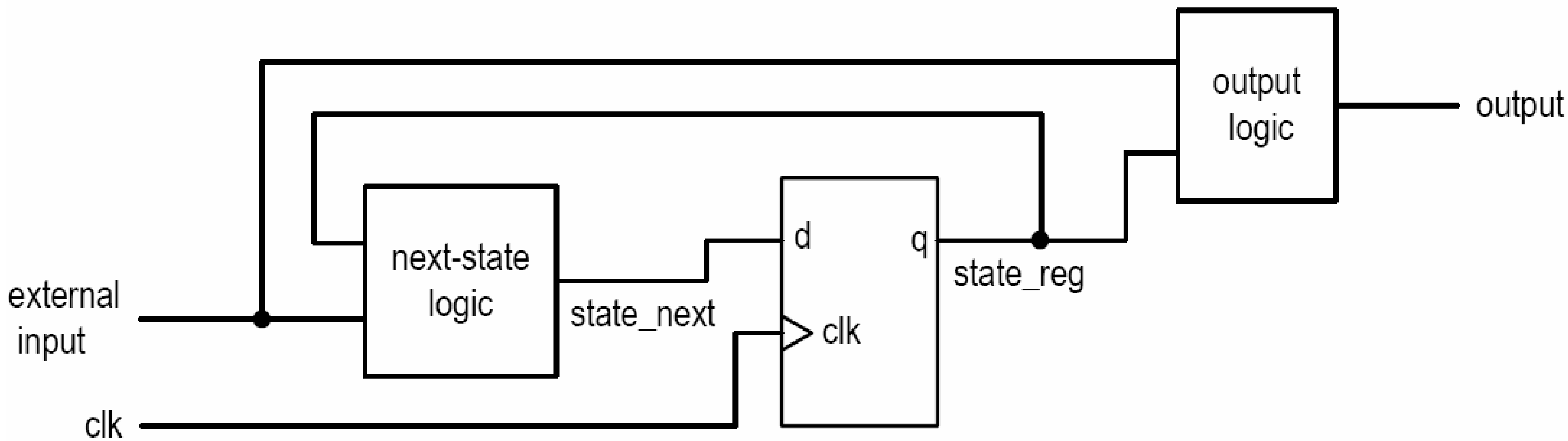
$$t_h = t_0 + T_{hold}$$

$$t_h < t_2$$

$$T_{hold} < T_{cq} + T_{next(min)}$$

$$T_{hold} < T_{cq}$$

Output delay



$$T_{co} = T_{cq} + T_{output}$$

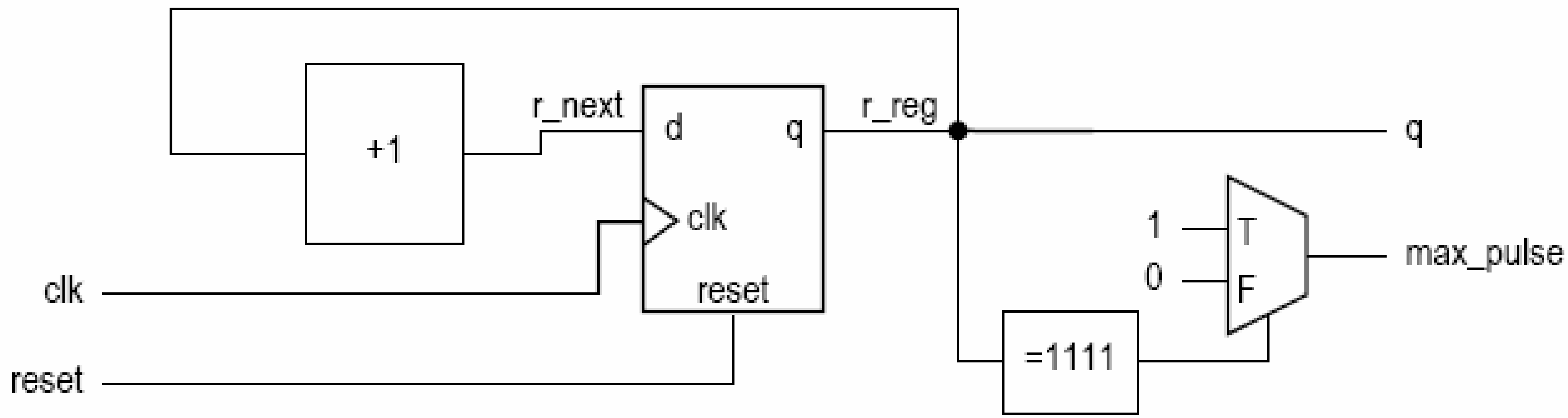
Consider two segment vs one segment counter description. First 2-segment:

```
architecture two_seg_arch of binary_counter4_pulse is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
   -- register
   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;
   -- next-state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_pulse <= '1' when r_reg="1111" else
                '0';
end two_seg_arch;
```

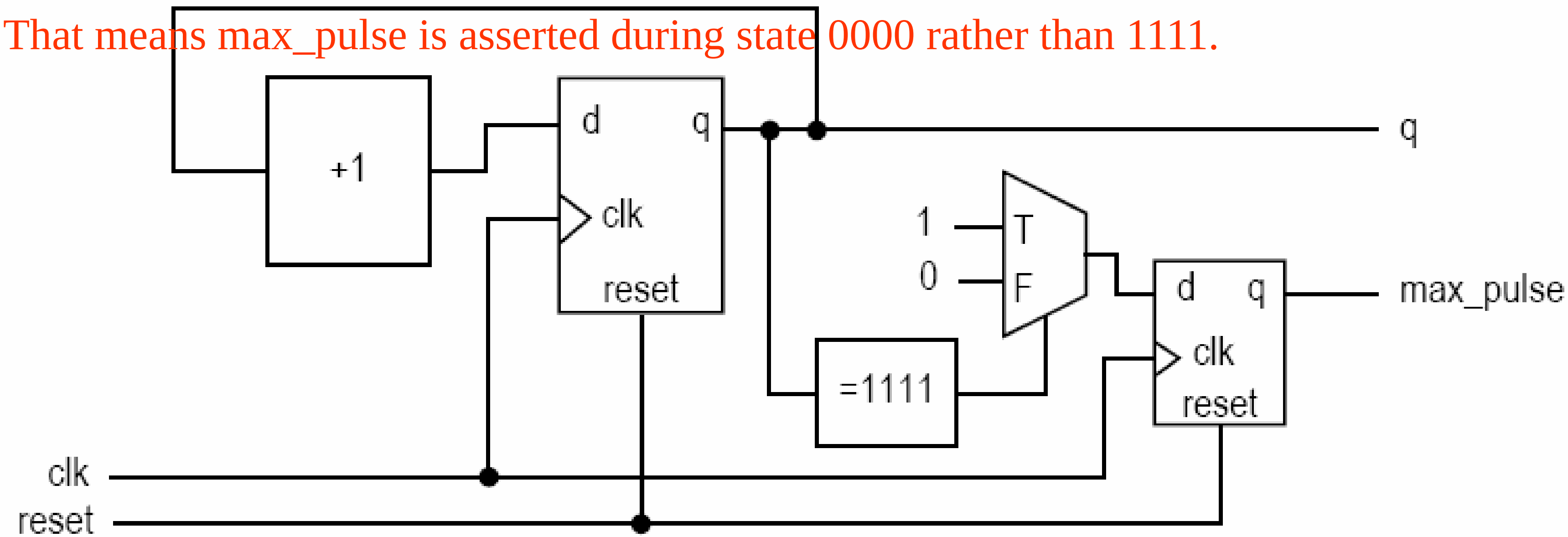
Here is a one segment version that creates an unintended one-clock delay in max_pulse output (see circuit in next slide)

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

Intended circuit with max_pulse asserted during state 1111. This circuit created with the two-segment description.



Circuit created with the one-segment example description. Note unintended flip-flop That means max_pulse is asserted during state 0000 rather than 1111.



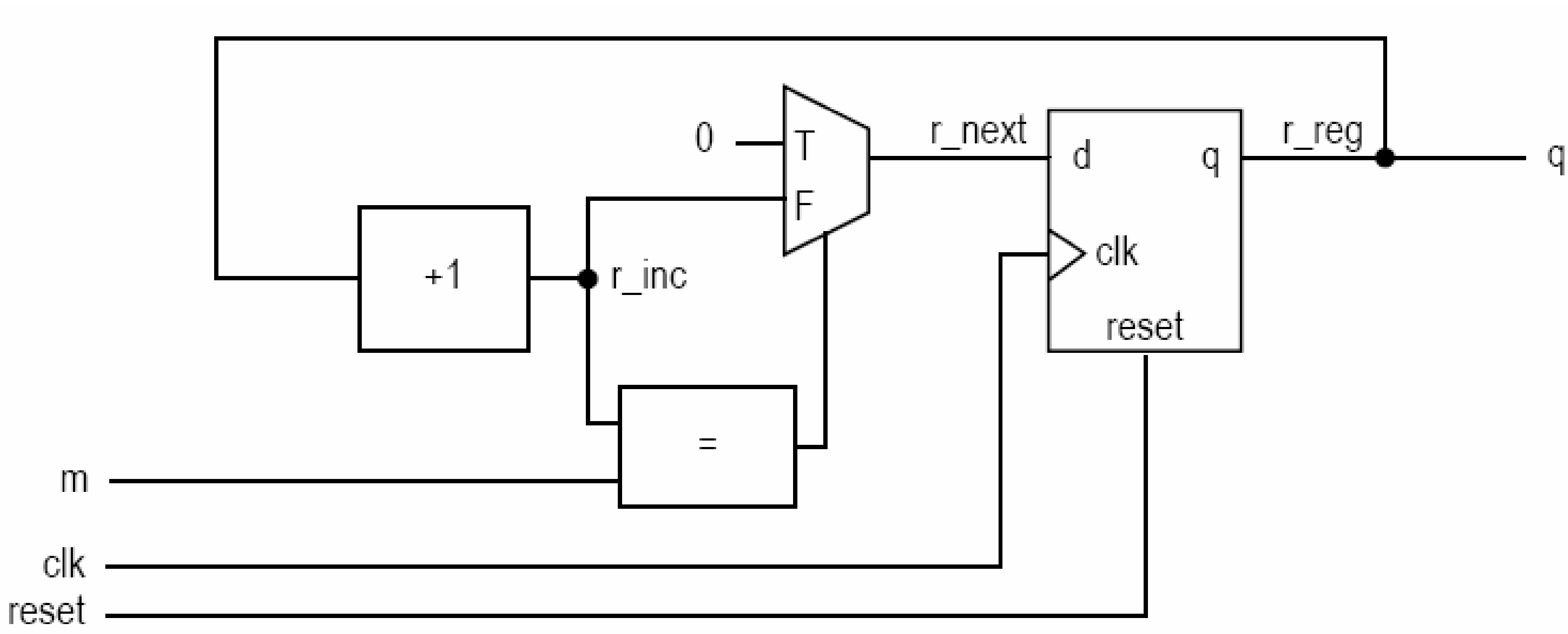
Revised “one-segment” counter description

```
architecture 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;
      end if;
   end process;
   q <= std_logic_vector(r_reg);
   max_pulse <= '1' when r_reg="1111" else
                '0';
end work_one_seg_glitch_arch;
```

Programmable mod-m counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity prog_counter is
   port(
      clk, reset: in std_logic;
      m: in std_logic_vector(3 downto 0);
      q: out std_logic_vector(3 downto 0)
      );
end prog_counter;

architecture two_seg_clear_arch of prog_counter is
```

```
architecture two_seg_effi_arch of prog_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next, r_inc: unsigned(3 downto 0);
begin
   -- register
   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;
   -- next-state logic
   r_inc <= r_reg + 1;
   r_next <= (others=>'0') when r_inc=unsigned(m) else
             r_inc;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_effi_arch;
```


0
T
F
r_next
d
q
r_reg
q
+1
r_inc
clk
reset
=
m
clk
reset

```
architecture not_work_one_arch of prog_counter 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=unsigned(m)) then
            r_reg<= (others=>'0');
         end if;
      end if;
   end process;
   q <= std_logic_vector(r_reg);
end not_work_one_arch;
```

```vhdl
architecture work_one_arch of prog_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_inc: unsigned(3 downto 0);
begin
   process (clk,reset)
   begin
      if reset='1' then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         if (r_inc=unsigned(m)) then
            r_reg <= (others=>'0');
         else
            r_reg <= r_inc;
         end if;
      end if;
   end process;
   r_inc <= r_reg + 1;
   q <= std_logic_vector(r_reg);
end work_one_arch;
```

A failed attempt to create an up/down counter. This will not synthesize. Two edge sensing statements in one if-else construct will cause a synthesis error.

```
signal r_reg : unsigned(3 downto 0);
process(clk,reset,ctrl)
begin
   if reset = '1' then
      r_reg <= (others=>'0');
   elsif (clk'event and clk='1' and ctrl='1') then
      r_reg <= r_reg + 1;
   elsif (clk'event and clk='1' and ctrl='0') then
      r_reg <= r_reg - 1;
   end if;
end process;
```

- Two-segment code
  - Separate memory segment from the rest
  - Can be little cumbersome
  - Has a clear mapping to hardware component
- One-segment code
  - Mix memory segment and next-state logic / output logic
  - Can sometimes be more compact
  - No clear hardware mapping
  - Error prone
- Two-segment code is preferred