Chapter 8
Sequential Circuit Design: Principle

Dr. Curt Nelson
Engr433 – Digital Design

Outline

• Overview of sequential circuits;
• Synchronous circuits;
• Danger of synthesizing asynchronous circuits;
• Inference of basic memory elements;
• Simple design examples;
• Timing analysis;
• Alternative one-segment coding style;
• Use of variable for sequential circuits.
Overview of Sequential Circuits

- 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 circuits

D Latch and D Flip-Flops

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

(a) D latch

<table>
<thead>
<tr>
<th>c</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

(b) pos-edge triggered D FF

<table>
<thead>
<tr>
<th>clk</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 q</td>
<td></td>
</tr>
<tr>
<td>1 q</td>
<td></td>
</tr>
<tr>
<td>f d</td>
<td></td>
</tr>
</tbody>
</table>

(c) neg-edge triggered D FF

<table>
<thead>
<tr>
<th>clk</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 q</td>
<td></td>
</tr>
<tr>
<td>1 q</td>
<td></td>
</tr>
<tr>
<td>f d</td>
<td></td>
</tr>
</tbody>
</table>

(d) D FF with asynchronous reset

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 f d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
D Latch and D Flip-Flops Example

- Problem with D latch - Can the two latches swap data?

D Latch Example
D Flip-Flop Timing

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

Synchronous vs Asynchronous Circuits

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

- One of the most difficult design aspects of a sequential circuits is how to satisfy the timing constraints.
- A Big, Useless idea
  - Group all D FFs together with a single clock.
  - Only need to deal with the timing constraint of one memory element.

Basic Block Diagram

- 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 have no effect as long as the state_next is stable at the sampling edge.
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)
Danger of Synthesizing Asynchronous Circuits

- 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

<table>
<thead>
<tr>
<th>d</th>
<th>q</th>
<th>c</th>
<th>q^*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
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
begin
  signal q_latch: std_logic;
  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;
```
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
D Latch

- No else branch
- D latch will be inferred

<table>
<thead>
<tr>
<th>d</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

(a) D latch

```vhdl
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;
```

Pos edge-triggered D FF

- No else branch
- Note the sensitivity list

```vhdl
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;
```

(b) pos-edge triggered D FF
• Neg edge-triggered D FF

<table>
<thead>
<tr>
<th>clk</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>1</td>
<td>q</td>
</tr>
<tr>
<td>'t'</td>
<td>d</td>
</tr>
</tbody>
</table>

![](image)

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

D FF with Async Reset

• No else branch
• Note the sensitivity list

```vhdl
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') then
            q <= d;
        end if;
    end process;
end arch;
```
Multiple D FFs with same clock and reset

```vhdl
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;
```

Simple Design Examples

- Follow the block diagram
  - Register
  - Next-state logic (combinational circuit)
  - Output logic (combinational circuit)
D FF With Synchronous Enable

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

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>en</th>
<th>q*</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

(a) Function table  (b) Conceptual diagram

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

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>t</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>1</td>
<td>q'</td>
</tr>
</tbody>
</table>

(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;
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 <= 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
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, 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

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);
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;
end

entity arb_seq_counter4 is
  port(
    clk, reset: in std_logic;
    q: out std_logic_vector(2 downto 0)
  );
end arb_seq_counter4;

architecture two_seq_arch of arb_seq_counter4 is
  signal r_reg: std_logic_vector(2 downto 0);
  signal r_next: std_logic_vector(2 downto 0);
  output
end two_seq_arch;

Arbitrary Sequence Counter

<table>
<thead>
<tr>
<th>input pattern</th>
<th>output pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>011</td>
</tr>
<tr>
<td>011</td>
<td>110</td>
</tr>
<tr>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

---

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;
end
Free-Running Binary Counter

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

```vhdl
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;
```
architecture two_seg_arch of binary_counter4_pulse is
signal r_reg: unsigned(3 downto 0);
signal r_next: unsigned(3 downto 0);
begins
-- 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:

\[ r_{next} \leftarrow (r_{reg} + 1) \mod 16; \]
Binary counter with bells & whistles

<table>
<thead>
<tr>
<th>syn_clr</th>
<th>load</th>
<th>en</th>
<th>q*</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>synchronous clear</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>d</td>
<td>parallel load</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>q+1</td>
<td>count</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>q</td>
<td>pause</td>
</tr>
</tbody>
</table>

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;

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

architecture two_seg_arch of mod10_counter is
constant TEN: integer := 10;
signal r_reg: unsigned(2 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;
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
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 $T_{cq}=1.0\text{ns}$ $T_{setup}=0.5\text{ns}$

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

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

- E.g., Binary counter; let $T_{cq}=1.0\text{ns}$ $T_{setup}=0.5\text{ns}$

<table>
<thead>
<tr>
<th>width</th>
<th>nand</th>
<th>xor</th>
<th>&gt;a</th>
<th>&gt;d</th>
<th>+1a</th>
<th>+1d</th>
<th>+a</th>
<th>+d</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>22</td>
<td>25</td>
<td>68</td>
<td>26</td>
<td>27</td>
<td>33</td>
<td>51</td>
<td>118</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>44</td>
<td>52</td>
<td>102</td>
<td>51</td>
<td>55</td>
<td>73</td>
<td>101</td>
<td>265</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>85</td>
<td>105</td>
<td>211</td>
<td>102</td>
<td>113</td>
<td>153</td>
<td>203</td>
<td>437</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>171</td>
<td>212</td>
<td>398</td>
<td>204</td>
<td>227</td>
<td>313</td>
<td>405</td>
<td>755</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>
\[ f_{\text{max}} = \frac{1}{T_{\text{cq}} + T_{\text{8-bit inc(area)}} + T_{\text{setup}}} = \frac{1}{1 \text{ ns} + 2.4 \text{ ns} + 0.5 \text{ ns}} \approx 256.4 \text{ MHz} \]

\[ f_{\text{max}} = \frac{1}{T_{\text{cq}} + T_{\text{16-bit inc(area)}} + T_{\text{setup}}} = \frac{1}{1 \text{ ns} + 5.5 \text{ ns} + 0.5 \text{ ns}} \approx 142.9 \text{ MHz} \]

\[ f_{\text{max}} = \frac{1}{T_{\text{cq}} + T_{\text{32-bit inc(area)}} + T_{\text{setup}}} = \frac{1}{1 \text{ ns} + 11.6 \text{ ns} + 0.5 \text{ ns}} \approx 76.3 \text{ MHz} \]

\[ f_{\text{max}} = \frac{1}{T_{\text{cq}} + T_{\text{8-bit inc(delay)}} + T_{\text{setup}}} = \frac{1}{1 \text{ ns} + 1.5 \text{ ns} + 0.5 \text{ ns}} \approx 333.3 \text{ MHz} \]

\[ f_{\text{max}} = \frac{1}{T_{\text{cq}} + T_{\text{16-bit inc(delay)}} + T_{\text{setup}}} = \frac{1}{1 \text{ ns} + 3.3 \text{ ns} + 0.5 \text{ ns}} \approx 208.3 \text{ MHz} \]

and

\[ f_{\text{max}} = \frac{1}{T_{\text{cq}} + T_{\text{32-bit inc(delay)}} + T_{\text{setup}}} = \frac{1}{1 \text{ ns} + 7.5 \text{ ns} + 0.5 \text{ ns}} \approx 111.1 \text{ MHz} \]

- **Hold time violation**

```

<table>
<thead>
<tr>
<th>clk</th>
<th>T_{cq}</th>
<th>T_{eq}</th>
<th>T_{z}</th>
</tr>
</thead>
<tbody>
<tr>
<td>state_reg</td>
<td>s0</td>
<td>s1</td>
<td>s2</td>
</tr>
<tr>
<td>state_next</td>
<td>s1</td>
<td>s2</td>
<td></td>
</tr>
</tbody>
</table>
```

invalid value (transient period)
\[ t_2 = t_0 + T_{cq} + T_{next(min)} \]

\[ t_h = t_0 + T_{hold} \]

\[ t_h < t_2 \]

\[ T_{hold} < T_{cq} + T_{next(min)} \]

\[ T_{hold} < T_{cq} \]

---

Output delay

\[ T_{co} = T_{cq} + T_{output} \]
7. Alternative one-segment coding style

- Combine register and next-state logic/output logic in the same process
- May appear compact for certain simple circuit
- But it can be error-prone

D FF with sync enable

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>en</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>f</td>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>f</td>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

(a) Function table

(b) Conceptual diagram
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;
• Interpretation: any left-hand-side signal within the clk’event and clk='1' branch infers a D FF

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;

T FF

<table>
<thead>
<tr>
<th>reset</th>
<th>clk</th>
<th>t</th>
<th>q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>q'</td>
</tr>
</tbody>
</table>

(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;
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;
architecture one_seg_arch of tff is
  signal q_reg: std_logic;
begin
  process (clk, reset)
  begin
    if reset='1' then
      q_reg <= '0';
    elsif (clk'event and clk='1') then
      if (t='1') then
        q_reg <= not q_reg;
      end if;
    end if;
  end process;
  q <= q_reg;
end one_seg_arch;

Binary counter with bells & whistles


<table>
<thead>
<tr>
<th>synclr</th>
<th>load</th>
<th>en</th>
<th>q*</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>00</td>
<td>synchronous clear</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>d</td>
<td>parallel load</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>q+1</td>
<td>count</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>q</td>
<td>pause</td>
</tr>
</tbody>
</table>

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;
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;

architecture one_seg_arch of binary_counter4_feature is
signal r_reg: unsigned(3 downto 0);
signal r_next: unsigned(3 downto 0);
begin
  -- register & next-state logic
  process (clk, reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      if syn_clr='1' then
        r_reg <= (others=>'0');
      elsif load='1' then
        r_reg <= unsigned(d);
      elsif en='1' then
        r_reg <= r_reg + 1;
      end if;
    end if;
  end process;
  -- output logic
  q <= std_logic_vector(r_reg);
end one_seg_arch;
Free-running binary counter

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

```vhdl
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;
```
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;
**Programmable mod-m counter**

```vhdl
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;
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;

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;
• 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