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

• Problem with D latch: Can the two D latches swap data?
• Timing of a D FF:
  – Clock-to-q delay
  – Constraint: setup time and hold time

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 one 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

Sync 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

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 scratch

E.g., a D latch from scratch

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
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;
```

D FF with async reset

- No else branch
- Note the sensitivity list

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port(
    clk: in std_logic;
    reset: in std_logic;
    d: in std_logic;
    q: out std_logic
  );
end dff;
architecture arch of dff 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;
```

5. Simple design examples

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

Neg edge-triggered D FF

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port(
    clk: in std_logic;
    q: out std_logic
  );
end dff;
architecture arch of dff is
begin
  if (clk'event and clk='0') then
    q <= '1';
  end if;
end arch;
```

Register

- Multiple D FFs with same clock and reset

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity reg is
  port(
    clk: in std_logic;
    reset: in std_logic;
    en: in std_logic;
    q: out std_logic_vector(7 downto 0)
  );
end reg;
architecture arch of reg 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;
```

D FF with sync enable

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

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port(
    clk: in std_logic;
    en: in std_logic;
    q: out std_logic
  );
end dff;
architecture arch of dff is
begin
  if (en='1') then
    q <= d;
  end if;
end arch;
```
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 tff_arch of tff is
  signal q_reg: std_logic;
  signal q_next: std_logic;
begin
  -- a DFF
  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 tff_arch;

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

Free-running shift register
Universal shift register

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

```vhdl
library ieee;
namespace std_logic_1164.all;

entity shift_right_register is
  port(
    clk, reset: in std_logic;
    d: in std_logic;
    q: out std_logic;
  );
end entity shift_right_register;

architecture two_reg_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');
    else
      if (clk='1' and clk='1') then
        r_reg <= r_next;
      end if;
    end process;
    -- next-state logic (shift right 2 bits)
    r_next <= d & r_reg(3 downto 1);
    -- output
    q <= r_reg(0);
  end two_reg_arch;
```

```vhdl
library ieee;
namespace std_logic_1164.all;

entity shift_register is
  port(
    clk, reset: in std_logic;
    d: in std_logic_vector(3 downto 0);
    q: out std_logic_vector(3 downto 0);
  );
end entity shift_register;

architecture two_reg_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');
    else
      if (clk='1' and clk='1') then
        r_reg <= r_next;
      end if;
    end process;
    -- next-state logic
    with d select
      r_next <=
        r_reg(2 downto 0) & d(0) when "00",
        r_reg(3 downto 0) & d(0) when "01",
        r_reg(0) & r_reg(2 downto 1) when "10",
        r_reg when others;
    -- output
    q <= r_reg;
  end two_reg_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>011</td>
</tr>
<tr>
<td>111</td>
<td>000</td>
</tr>
</tbody>
</table>

entity arbi_seq_counter is
process
  cin, reset: in std_logic;
begin
  if (reset = '1') then
    x_reg <= (others => '0');
    else
      if (cin'event and cin = '1') then
        r_reg <= r_reg + 1;
      end if;
  end if;
end process;
architecture two_seq_arch of arbi_seq_counter is
  signal r_reg: std_logic_vector(2 downto 0);
  signal r_next: std_logic_vector(2 downto 0);
end two_seq_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 is
  -- Pin out
  port(
    clk, reset: in std_logic;
    max_pulse: out std_logic;
    q: out std_logic_vector(3 downto 0);
  );
end binary_counter4;

architecture two_seq_arch of binary_counter4 is
  signal r_reg: std_logic_vector(3 downto 0);
  signal r_next: std_logic_vector(3 downto 0);
begin
  if (reset = '1') then
    r_reg <= (others => '0');
    else
      if (clk'event and clk = '1') then
        r_reg <= r_reg + 1;
      end if;
  end if;
end process;
architecture two_seq_arch of binary_counter4 is
process
  cin, reset: in std_logic;
begin
  if (reset = '1') then
    x_reg <= (others => '0');
    else
      if (cin'event and cin = '1') then
        r_reg <= r_reg + 1;
      end if;
  end if;
end process;
architecture two_seq_arch of binary_counter4 is
begin
  register
  process (clk, reset)
  begin
    if (reset = '1') then
      r_reg <= (others => '0');
    else
      if (cin'event and cin = '1') then
        r_reg <= r_reg + 1;
      end if;
    end if;
  end process;
end two_seq_arch;

Binary counter with bells & whistles

- Wrapped around automatically
- Poor practice:

\[ r_{next} = (r_{reg} + 1) \mod 16; \]
Decade (mod-10) counter

```vhdl
circuit decade_counter
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;
  -- next-state logic
  T_next <= (others=>'0') when syn_clk='1' else
            r_reg + 1 when cc = '1' else
            r_reg;
  -- output logic
  q <= std_logic_vector(r_reg);
end decade_counter;
```

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 prog_counter_arch of prog_counter is
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;
  -- next-state logic
  r_next <= (others=>'0') when r_reg+(m-1) else
            r_reg + 1;
  -- output logic
  q <= std_logic_vector(r_reg);
end prog_counter_arch;
```
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

\[ t_3 = t_0 + T_{cq} + T_{next(max)} \]
\[ t_4 = t_5 - T_{setup} = t_0 + T_c - T_{setup} \]
\[ t_3 < t_4 \]
\[ t_0 + T_{cq} + T_{next(max)} < t_0 + T_c - T_{setup} \]
\[ T_{cq} + T_{next(max)} + T_{setup} < T_c \]
\[ T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup} \]

- **Setup time violation and maximal clock rate**

\[ T_{c(min)} = T_{cq} + T_{setup} = 1.5 \text{ ns} \]
\[ f_{max} = \frac{1}{T_{cq} + T_{setup}} = \frac{1}{1.5 \text{ ns}} \approx 666.7 \text{ MHz} \]

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

<table>
<thead>
<tr>
<th>nMB</th>
<th>x</th>
<th>y</th>
<th>$n_1$</th>
<th>$n_2$</th>
<th>$n_3$</th>
<th>$n_4$</th>
<th>$n_5$</th>
<th>REMARKS</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>55</td>
<td>51</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>41</td>
<td>32</td>
<td>182</td>
<td>31</td>
<td>55</td>
<td>73</td>
<td>104</td>
</tr>
<tr>
<td>32</td>
<td>85</td>
<td>162</td>
<td>183</td>
<td>211</td>
<td>142</td>
<td>145</td>
<td>205</td>
<td>137</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>171</td>
<td>212</td>
<td>396</td>
<td>201</td>
<td>227</td>
<td>515</td>
<td>405</td>
</tr>
</tbody>
</table>

- Hold time violation

- 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>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>-</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>0 1</td>
<td>-</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>0 f</td>
<td>0</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>0 f</td>
<td>1</td>
<td>q</td>
<td>q</td>
</tr>
</tbody>
</table>

- Interpretation: any left-hand-side signal within the clk'event and clk='1' branch infers a D FF.

```
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;
```

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>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>-</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>0 1</td>
<td>-</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>0 f</td>
<td>0</td>
<td>q</td>
<td>q</td>
</tr>
<tr>
<td>0 f</td>
<td>1</td>
<td>q</td>
<td>q</td>
</tr>
</tbody>
</table>

- Interpretation: any left-hand-side signal within the clk'event and clk='1' branch infers a T FF.

```
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 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';
    elseif (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;
```

```
architecture one_seg_arch of dff_en is
begin
  process (clk, reset)
  begin
    if (reset='1') then
      q <= '0';
    elseif (clk'event and clk='1') then
      if (en='1') then
        q <= d;
      end if;
    end if;
  end process;
end one_seg_arch;
```
Binary counter with bells & whistles

<table>
<thead>
<tr>
<th>sym</th>
<th>clr</th>
<th>load</th>
<th>on</th>
<th>q</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>00...00</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>-</td>
<td>d</td>
<td>q</td>
<td>count</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>q</td>
<td>q</td>
<td>q = 1</td>
</tr>
<tr>
<td>0</td>
<td>q</td>
<td>0</td>
<td>0</td>
<td>q</td>
<td>prev</td>
</tr>
</tbody>
</table>

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

Free-running binary counter

- Count in binary sequence
- With a max_pulse output: asserted when counter is in “11...11” state
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_arch of prog_counter is
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 <= 't' when r_reg='1111' else '0';
end two_seg_arch;

architecture work_one_seg_glitch_arch of binary_counter_pass_pulse is
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 <= 't' when r_reg='1111' else '0';
end work_one_seg_glitch_arch;

architecture two_seg_effi_arch of prog_counter is
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);
end two_seg_effi_arch;

architecture two_seg_arch of binary_counter_pass_pulse is
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;

  -- output logic
  q <= std_logic_vector(r_reg);
end two_seg_arch;

architecture work_one_seg_glitch_arch of binary_counter_pass_pulse is
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);
end work_one_seg_glitch_arch;

architecture two_seg_effi_arch of prog_counter is
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);
end two_seg_effi_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