Overview of Sequential Circuits

- **Storage Elements**
  - Sequential circuits
  - Storage elements: Latches & Flip-flops
  - Registers and counters
- **Circuit and System Timing**
- **Sequential circuit description and analysis:**
  State tables & State diagrams
Sequential Circuits

- A Sequential circuit contains:
  - Storage elements: Latches or Flip-Flops
  - Combinational Logic: implements a multiple-output switching function
    - Next state function: Next State = f(Inputs, State)
    - Output function: two types
      - Mealy: Outputs = g(Inputs, State)
      - Moore: Outputs = h(State)

![Diagram of Sequential Circuits]

Latches

- Basic S-R latch
- Basic S̅-R latch
- Clocked S-R latch
- D latch
**Edge-Triggered D Flip-Flop**

- The edge-triggered D flip-flop is the master-slave D flip-flop.
- The change of Q is associated with the negative edge at the end of the pulse - *negative-edge triggered flip-flop*.

**Flip-flop Behavior Example**

- Use the characteristic tables to find the output waveforms for the flip-flops shown:

```
Clock
D
Q_D
```

```
D
C
```

```
D
?
```
Modeling of Flip-Flops

Library IEEE;
use IEEE.Std_Logic_1164.all;

entity FLOP is
  port (D, CLK : in std_logic;
        Q : out std_logic);
end FLOP;

architecture A of FLOP is
begin
  process
  begin
    wait until CLK'event and CLK='0';
    Q <= D;
  end process;
end A;

- D Flip-flop controlled by a clock pulse edge.

Or:

process (clk)
begin
  if (clk='0') then
    Q <= D;
  end if;
end process;

Direct Inputs

- At power up or at reset, sequential circuit usually is initialized to a known state before it begins operation
  - Done outside of the clocked behavior of the circuit, i.e., asynchronously.
  - Direct R/S inputs

- For the example flip-flop shown
  - 0 applied to \( \overline{R} \) resets the flip-flop to the 0 state
  - 0 applied to \( \overline{S} \) sets the flip-flop to the 1 state
Positive Edge-Triggered D Flip-flop with Asynchronous Set/Reset

library IEEE;
use IEEE.std_logic_1164.all;

entity ASYNC_FF is
  port (D, CLK, SETN, RSTN : in std_logic;
        Q : out std_logic);
end ASYNC_FF;

architecture RTL of ASYNC_FF is
begin
  process (CLK, RSTN, SETN)
  begin
    if (RSTN = '1') then
      Q <= '0';
    elsif SETN = '1' then
      Q <= '1';
    elsif (CLK'event and CLK = '1') then
      Q <= D;
    end if;
  end process;
end RTL;

- It has a sensitivity list
- If / elsif – structure
  - The last elsif has an edge
  - No unconditional else branch

Registers

- Register: the simplest storage component in a computer, a bit-wise extension of a flip-flop.

- Registers can be classified into
  - Simple Registers
  - Parallel-Load Registers
  - Shift Registers
Simple Registers

Fig. 1 A 4-bit Register: top – schematic, bottom – graphical symbol.

- A simple register consists of \( N \) flip-flops driven by a common clock signal.
- Has \( N \) inputs and \( N \) outputs in addition to the clock signal.

Fig. 2 A 4-bit register with asynchronous preset and clear
Library ieee;
USE ieee.std_logic_1164.all;
ENTITY simple_register IS
  GENERIC ( N : INTEGER := 4);
  PORT ( I : IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
         Clock, Clear, Preset : IN STD_LOGIC;
         Q : OUT STD_LOGIC_VECTOR (N-1 DOWNTO 0));
END simple_register;

ARCHITECTURE simple_memory OF simple_register IS
BEGIN
  PROCESS (Preset, Clear, Clock)
  BEGIN
    IF Preset = '0' THEN
      Q <= (OTHERS => '1');
    ELSIF Clear = '0' THEN
      Q <= (OTHERS => '0');
    ELSIF (Clock'EVENT AND Clock = '1') THEN
      Q <= I;
    END IF;
  END PROCESS;
END simple_memory;

---

**Parallel Load Registers**

- In the previous registers, new data is stored automatically on every rising edge of the clock.

- In most digital systems, the data is stored for several clock cycles before it is rewritten. For this reason it is useful to be able to control WHEN the data will be entered into a register.
  - Use a control signal called *Load* or *Enable*. This allows loading into a register known as a **parallel-load register**.
**VHDL Code for Parallel-load Register**

```vhdl
Library ieee;
USE ieee.std_logic_1164.all;
ENTITY load_enable IS
    GENERIC ( N : INTEGER := 4);
    PORT ( D : IN STD_LOGIC_VECTOR (N-1 DOWNTO 0);
        Clock, Resetn, load : IN STD_LOGIC;
        Q : BUFFER STD_LOGIC_VECTOR(N-1 DOWNTO 0));
END load_enable;
ARCHITECTURE rtl OF load_enable IS
    SIGNAL state : std_logic_vector(N-1 DOWNTO 0);
BEGIN
    PROCESS (Resetn, Clock) IS
    BEGIN
        IF Resetn = '0' THEN
            state <= (OTHERS => '0');
        ELSIF (Clock'EVENT AND Clock = '1') THEN
            IF load = '1' THEN
                state <= D;
            ELSE
                state <= state;
            END IF;
        END IF;
    END PROCESS;
    Q <= state;
END rtl;
```

**Timing diagram for parallel-load register**

[Timing diagram showing the behavior of the parallel-load register with input D and output Q over time]
Serial-in/Parallel-out Shift Register

(a) Graphic symbol
(b) Operation table
(c) Register schematic

Parallel Load Shift Register

(a) Graphic symbol
(b) Operation table
(c) Register schematic
Counters

- Counter: a special type of register that incorporates an incrementer or decrementer, which allows it to count upward or downward.

- We shall examine VHDL code for the following counters:
  - Up counter
  - Down counter

VHDL for Up-Counter

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY upcount IS
  GENERIC ( N : INTEGER := 4 );
  PORT ( Clock, Resetn, Enable : IN STD_LOGIC;
         Q: BUFFER STD_LOGIC_VECTOR (N-1 DOWNTO 0));
END upcount;

ARCHITECTURE cnt OF upcount IS
  SIGNAL count : STD_LOGIC_VECTOR (N-1 DOWNTO 0);
BEGIN
  PROCESS (Resetn, Clock)
  BEGIN
    IF Resetn = '0' THEN
      count <= ( OTHERS => '0' );
    ELSIF (Clock'EVENT AND Clock = '1') THEN
      IF Enable = '1' THEN
        count <= count +1;
      ELSE
        count <= count;
      END IF;
    END IF;
  END PROCESS;
  Q <= count;
END cnt;
```
### Timing Diagram for Up-Counter

<table>
<thead>
<tr>
<th></th>
<th>250 ns</th>
<th>500 ns</th>
<th>750 ns</th>
<th>1.0 μs</th>
<th>1.25 μs</th>
<th>1.5 μs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clock</strong></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reset</strong></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Enable</strong></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Count</strong></td>
<td>H0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>H0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

### Synchronous Counters

- **Internal Logic**
  - Incrementer: Q+0 or Q+1
- **Contraction of a ripple carry adder with one operand fixed at 000X**
- **Symbol for Synchronous Counter**

![Symbol for Synchronous Counter](image)
Synchronous Counters (Contd.)

- Contraction of carry-lookahead adder
  - Reduce path delays
  - Called parallel gating
  - Lookahead can be used on COs and ENs to prevent long paths in large counters

Counter with Parallel Load

<table>
<thead>
<tr>
<th>Load</th>
<th>Count</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Hold Stored Value</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Count Up Stored Value</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>Load D</td>
</tr>
</tbody>
</table>

- Add path for input data D
  - enabled for Load = 1

- Add logic to:
  - When Load = 1 disable count logic (feedback from outputs)
  - When Load = 0 and Count = 1 enable count logic
architecture Behavioral of bcd_counter is
    signal regcnt : std_logic_vector(3 downto 0);
begin
    count: process(reset,clk) is
        begin
            if ( reset='1' ) then
                regcnt <= "0000";
            elsif ( clk'event and clk='1') then
                regcnt <= regcnt+1;
                if (regcnt = "1001") then
                    regcnt <= "0000";
                end if;
            end if;
        end process;
end Behavioral;