Counters and Shift Registers

---

**Counter**
- A Sequential Circuit whose output progresses in a predictable repeating pattern
- It advances one state for each clock pulse
- Uses:
  - Event Counting
  - Frequency Division
  - Timing and Control Operations

**Terms**

- **Shift Register**
  - A Sequential Circuit that moves stored data bits in a specific direction by chaining FFs
- **Used in**
  - Serial data transfers
    - Serial In, Parallel Out (SIPO / PISO) conversions
  - Arithmetic
  - Delays

- **State Diagram**
  - A graphical diagram showing the progression of states in a sequential circuit such as a counter

- **Count Sequence**
  - The specific series of output states through which a counter progresses

---

**Counter Modulus**

- **Modulus**
  - The number of states a counter sequences through before repeating \((\text{modulo-N or mod-N})\)
- **Up / Down Counter**
  - Indicates the direction of the counter sequence
    - higher to lower \(\Rightarrow\) down
    - lower to higher \(\Rightarrow\) up

- **Modulus of a counter is the number of states through which a counter progresses**
  - A Mod-12 Up Counter counts 12 states from 0000 to 1011 (0 to 11) and then repeats
  - A Mod-12 Down Counter counts 12 states from 1011 to 0000 (11 to 0) and then repeats
Counter State Diagram

- Each Counter State is represented by a binary value (0000 to 1011) in a bubble.
- The progression to the next state is shown by an arrow (0000 $\Rightarrow$ 0001 $\Rightarrow$ 0010).
- Each state transition is caused by a pulse on the clock input to the counter.

Counter State Diagram

Figure 9.2

Counters

- Full Sequence Counter
  - An N-Bit Counter that counts the maximum modulus ($2^N$).
  - Mod2, Mod4, Mod8 etc.
- Truncated Counter
  - A N-Bit Counter whose modulus is less than the maximum possible.
  - Mod3 (N=2), Mod12 (N=4).

Counts

- Mod12 Counter counts up from 0000 to 1011 while a Mod16 counts up from 0000 to 1111.
- Both counters are 4-Bit (N=4), so 4 flip-flops are required to implement each.
- This is illustrated in the counter state tables (See Textbook p. 506).

Timing Diagrams

- A Timing Diagram for a Mod16 Up Counter is illustrated in Figure 9.4.
- The diagram shows the output waveforms for Q0, Q1, Q2, and Q3 in time.
- The output Q0 changes for every clock pulse, Q1 changes on every two clock pulses, Q2 on four and Q3 on 8 clocks.

Timing Diagrams

Figure 9.4
Timing Diagrams

- The outputs (Q0 - Q3) of the counter can be used as frequency dividers
  - Q0 = Clock \( \div 2 \)
  - Q1 = Clock \( \div 4 \)
  - Q2 = Clock \( \div 8 \)
  - Q3 = Clock \( \div 16 \)
- Keep in mind that frequency is based on T of the output not a transition on the output
- The same is true for a Mod12 except Q3 = Clock \( \div 12 \)

Simplest Counter Design

- Uses only NC (No Change) or Toggle modes of JK FFs
- \( J_0 = K_0 = 1 \)
- \( J_1 = K_1 = Q_0 \)
- \( J_2 = K_2 = Q_0Q_0 \)
- \( J_3 = K_3 = Q_0Q_0Q_0 \)
- \( J_4 = K_4 = Q_0Q_0Q_0Q_0 \)
- etc

Synchronous Counter

- A counter whose flip-flops are all clocked by the same source and change state in synchronization
- 2 Basic Sections:
  - Memory Section (flip-flops) to keep track of present state
  - Control section to direct the counter to the next state using Command and Status Lines

Synchronous State Table (Analysis)

- Set equations for the inputs (JK, D, T) in terms of the Q outputs for the counter
- Set up a table similar to the one in Table 9.5 and place the first Initial state in the present state column (usually all 000)
- Use the initial state to fill in the inputs that will result from this state before the next clock pulse

Synchronous Counter Block Diagram

- Input
- Command Lines
- Status Lines
- Output Decoder (optional)
- Output Lines
- CLK
- Memory Section (Flip-flops)
- Stores present state
- Determines next state

Synchronous State Table (Analysis)

- Determine the result on each FF in the counter and place this in next state
- Enter next state on the present state line 2 and repeat the process until you cycle back to the first initial state (See example 9.3, p 512)
Synchronous Counter (Design)

- Basic approach is to derive the Boolean equations to control the change to next state from the present state by a clock
- State transitions are defined in a state “bubble” diagram
- Same approach is used in state machine design (Chapter 10)

Basic Design Approach

- Draw a state diagram showing state changes, and inputs and outputs
- Create a Present/Next State Table
- List Present States in Binary Order and Next States based on the State Diagram
- Use FF Excitation Tables to determine FF (JK,D,T) inputs for each Present ⇒ Next State Transition
  * Excitation Table shows the required input for every possible transition of FF output

JK Flip-Flop Excitation Table

<table>
<thead>
<tr>
<th>Transition</th>
<th>Mode Required</th>
<th>JK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 → 0</td>
<td>No Change or Reset</td>
<td>00</td>
</tr>
<tr>
<td>0 → 1</td>
<td>Toggle or Set</td>
<td>11</td>
</tr>
<tr>
<td>1 → 0</td>
<td>Toggle or Reset</td>
<td>10</td>
</tr>
<tr>
<td>1 → 1</td>
<td>No Change or Set</td>
<td>01</td>
</tr>
</tbody>
</table>

Basic Design Approach (cont’d)

- Specify Inputs Equations for each input and simplify using Boolean Reductions
- This can be a long tedious process
  * See example of mod-12 counter, page 515
- A VHDL design for counters is more easily done and not as time consuming

Unused States

- Can be don’t cares
  * If we enter an unused state, it will transition to the main sequence eventually
  * Usually, this is OK
- Can be designed to transition to initial count state
  * Redesign our next-state expressions
  * More complex

Additional Counter Features

- Load
  * Parallel
  * Synchronous or Asynchronous
- Clear
- Enable
- Direction
  * Up or Down
- Terminal Count
  * and Ripple Carry Output (RCO)
- Output Decode
A preset counter (Parallel Load) has an additional input (Load) that can be synchronous or asynchronous and [four] parallel data inputs (Figure 9.33).

The load pulse selects whether the synchronous counter outputs are generated by count logic or parallel load data.

The Synchronous Load Counter is shown in Figure 9.36.

An Asynchronous Load Counter uses an Asynchronous Clear or Preset to force counter to a known state (usually 0000 or 1111) (Figure 9.41).

Shown in Figure 9.46 by adding another AND Gate to each FF Input to Inhibit the Count Function.

This has the effect of inhibiting the Clock to the counter (a clock pulse has no effect).

Outputs remain at last state until counter is enabled again.
Bidirectional Counter

- Adds a Direction Input (DIR) to the counter and the Control Logic for Up or Down Counting
- Basic counter element (TFF) is shown in Figure 9.50
- The Control Logic selects the Up or Down Count Logic depending on the state of DIR (Figure 9.51)

Terminal Count Decoding

- Uses a combinational decoder to detect when the last state of a counter is reached (Terminal Count)
- Determines a Maximum Count for an Up Counter and a Minimum Count for a Down Counter

Terminal Count Decode Logic

- The TC decoder generates a RCO (Ripple Carry Out) when the terminal count is reached (a high pulse for 1 clock period)
- May be used to asynchronously clock a further counter stage to extend counter width
Shift Registers

Terms

- Shift Register
  - A Synchronous Sequential circuit that will store and move N-bit data either serially or in parallel in a N-Bit Register (N flip-flops)

Serial Shift Register

- A 4-Bit Left Shift Register
  - Din is shifted into the LSB FF and shifted towards the MSB

Bidirectional Shift Register

- Uses a Control Input signal called DIRECTION to change circuit function from Shift Right to Shift Left
  - 4-Bit Bidirectional SR shown in Figure 9.91
  - When DIR = 0 the path of Left_Shift_In ⇒ Q0 ⇒ Q1 ⇒ Q2 ⇒ Q3 is selected
  - When DIR =1 then it selects the Right Shift In Path
Bidirectional Shift Register

SR with Parallel Load
- Similar to a Parallel Load Counter and is shown in Figure 9.93
- Uses a 2 to 1 Mux (AND/OR) to control inputs to the FF in the SR. The input choice is from previous FF Output or the Parallel Input
- When Load=1 Parallel Data is loaded in on the next clock pulse

SR with Parallel Load

Universal SR
- Combines the basic functions of a Parallel Load SR with a Bidirectional SR
- Uses two control inputs (S1,S0) to select the function

Universal SR Truth Table

<table>
<thead>
<tr>
<th>S0</th>
<th>S1</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Hold (Q_i = Q_i)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>RSR (Q_i = Q_{i+1})</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>LSR (Q_i = Q_{i-1})</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Load (Q_i = P_i)</td>
</tr>
</tbody>
</table>

Shift Register Counters
- Ring Counter
  - A serial Shift Register with feedback from the output of the last FF to the input of the first FF
- Johnson Counter
  - A serial shift register with the complemented feedback from the output of the last FF to the input of the first FF
- Counter sequences are based on a continuous rotation of data through the SR
Ring Counters (or Barrel Shifters)

- A basic Ring Counter (Figure 9.77) is constructed of D-FF with a Feedback Loop
- Data is initially loaded into the SR by using either Resets or Presets
- The counter can circulate a 0 or 1 by loading a 0111 or 1000

The Modulus of a Ring Counter is defined as the maximum number of unique states

- Modulus is dependent on the initial load value {1000, 0100, 0010, 0001} = Mod4 while {1010, 0101} = Mod2
- Typically an N-FF Ring Counter has N States not $2^N$ like a binary counter

Johnson Counters

- Same as a Ring (Figure 9.106) except that $!Q0$ (Complement) is fed back to D3 not Q0
- Adds a complement or “twist” to the data and is called a Twisted Ring Counter
- Usually Initialized with 0000 by a Clear

Typically has more states than a ring counter

- Sequence of states = {0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001}
- Maximum modulus is 2N for an N-bit Johnson Counter
Chapter 9 (VHDL)

VHDL Constructs for Counters and Shift Registers

VHDL Process Statements
- Sequential Counters use a Process Statement to control transitions to the next count state
- A VHDL Attribute is used with an identifier (signal) to define clock edges
- Clock uses an Attribute called EVENT such as (clk'EVENT AND clk='1) to define a Rising Edge Clock Event

VHDL Up Counter ENTITY
- Basic Entity for Binary Counter with Asynchronous Clear

ENTITY ct_simp IS
PORT( clk, clear : IN BIT;
q : OUT INTEGER RANGE 0 to 255);
END ct_simp;

-- Counter Outputs (8-Bit) are given as an integer with a range of 0 to 255 (Full Sequence Counter)

VHDL Up Counter ARCHITECTURE
ARCHITECTURE behavioral OF ct_simp IS
BEGIN
PROCESS(clk, clear)
VARIABLE count : INTEGER RANGE 0 to 255;
BEGIN
IF(clear = '0') THEN
count := 0;
ELSE
IF(clk'EVENT AND clk = '1') THEN
count := count + 1;
END IF;
END IF;
END IF;
q <= count;
END PROCESS;
END behavioral;

VHDL Up Counter
- The process was set to execute on changes in clk or clear
- The asynchronous clear is placed before the clk EVENT
- The clk attribute (EVENT) was set for rising edge
- The next state transition was controlled by a simple integer addition (count := count + 1)

VHDL Up Counter
- This same approach could be used by other hardware (multipliers, dividers etc.)
- In the example a Variable count was used for the integer operations (the increment)
- It uses a Variable Assignment Operator (:=) instead of a Signal Assignment (<=)
LPM Counters

- The Altera LPM (Library of Parameterized Modules) Counter can be used to create counter designs in VHDL.
- This is a Structured Design Approach that uses the LPM-Counter as a Component in a Hierarchy.
- The LPM Counter is instantiated in the Structured Design.

LPM Library Declaration

- In this case we add the Altera LPM Library to the usual STD_LOGIC.
- The ieee library must be declared before the LPM library.

VHDL LPM Entity

- Entity for an 8-Bit Mod256 Counter.
- LPM requires the use of STD_LOGIC data types.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY LPM
USE lpm.lpm_components.ALL;

ENTITY lpm_simple IS
PORT(clk, clear : IN STD_LOGIC;
     q          : OUT STD_LOGIC_VECTOR(7 downto 0));
END lpm_simple;
```

VHDL LPM Architecture

- The Architecture uses a Generalized LPM counter with a width set to 8 Bits.
- IO of the LPM are mapped to the clk, clear and q pins of the ENTITY.
- We do NOT need to create:
  - An integer expression for count
  - Clocked process or asynchronous clear
- They are mapped to a programmable width counter.

```vhdl
ARCHITECTURE count OF lpm_simple IS
  SIGNAL clrn :STD_LOGIC; -- Clear for LPM
BEGIN
  count8 : lpm_counter
    GENERIC MAP(LPM_WIDTH => 8);
    PORT MAP (clock => clk,
               aclr => clrn,
               q => q(7 DOWNTO 0));
  clrn <= NOT clear;
END count;
```
Other LPM Counter Features

- **Parallel Load**
  - A function (syn/asyn) that allows loading of a binary value into the counter FF

- **Clear**
  - Asynchronous or Synchronous Reset

- **Preset**
  - A Set (Sync or Async)

- **Counter Enable**
  - A control function that allows a counter to execute its sequence or hold its present value

LPM Counter Features

- **Bidirectional**
  - A control input to switch counter from a Count Up to Count Down

- There are other features for LPM Counters that are given in Altera Reference Data Sheets (See table in chapter 9)

  The same holds true for other LPM Functions such as Arithmetic and Memory

VHDL Counter (Bidirectional, 4-Bit) Entity

```vhdl
ENTITY pre_ct8a IS
PORT(clk, count_ena, clear, load : IN BIT;
     direction : IN BIT;
     p       : IN INTEGER RANGE 0 to 255;
     max_min : OUT BIT;
     q       : OUT INTEGER RANGE 0 to 255);
END pre_ct8a;
```

VHDL Counter (Bidirectional, 4-Bit) Architecture

```vhdl
ARCHITECTURE a OF pre_ct8a IS
BEGIN
    PROCESS (clk, clear, load)
        VARIABLE  cnt: INTEGER RANGE 0 to 255;
    BEGIN
        IF (clear = '0') THEN cnt := 0;                       -- asynchronous clear
        ELSIF (load = '1' AND clear = '1) THEN                -- asynchronous load
            cnt := p;
        ELSE
            IF (clk'EVENT AND clk = '1') THEN
                IF (count_ena = '1' AND direction = '0') THEN     -- down count
                    cnt := cnt - 1;
                ELSIF (count_ena = '1' AND direction = '1') THEN   -- up count
                    cnt := cnt + 1;
                END IF;
            END IF;
        END IF;
        max_min <= '0';
        q <= cnt;
    END PROCESS;
END a;
```

Terminal Count Architecture

```vhdl
IF (cnt = 0 AND direction = '0') THEN
    max_min <= '1';
ELSIF (cnt = 255 AND direction = '1') THEN
    max_min <= '1';
ELSE
    max_min <= '0';
END IF;
```

VHDL SR Entity

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY altera;
USE altera.maxplus.all;
-- Note IEEE is before altera declarations
-- Maxplus is for the primitive DFF Design
ENTITY srg4strc IS
PORT( serial_in, clk : IN STD_LOGIC;
     q              : BUFFER STD_LOGIC_VECTOR(3 downto 0));
END srg4strc;
-- The 4 Bit Register is given a type Buffer to allow
-- q(0) ⇒ q(3) to be used as Input or Output
```
**VHDL SR Structured Architecture**

```vhdl
ARCHITECTURE right_shift OF srg4strc IS
  COMPONENT dff -- DFF Component
    PORT( d, clk : IN STD_LOGIC;
        q      : OUT STD_LOGIC );
  END COMPONENT;
  BEGIN
    flipflop3: dff
      PORT MAP(serial_in, clk, q(3));
    dffs: FOR I IN 2 downto 0 GENERATE
      flipflops 2 to 0: dff
        PORT MAP(q(I + 1), clk, q(I));
    END GENERATE;
  END right_shift;
```

**Structured Architecture**

- Four dff components are mapped to create a RSR
- `serial_in` is to Q3 and shift is towards Q0
- The DFF for Q3 is mapped individually
- Uses a FOR GENERATE Loop to create and map the remaining DFFs

**Dataflow Design Approach**

- **Dataflow Design**
  - A VHDL design approach that uses Boolean equations to define relationships between inputs and outputs
  - The Entity is the same as the structured approach except the Altera library is not needed (no primitives)

**VHDL Dataflow RSR Design**

```vhdl
ARCHITECTURE right_shift OF srg4dflw IS
  SIGNAL d : STD_LOGIC_VECTOR(3 downto 0);
  BEGIN
    PROCESS(clk)
      BEGIN
        IF clk'EVENT AND clk = '1' THEN
            q <= d; -- Still declared as a buffer in the entity
        END IF;
    END PROCESS;
    d <= serial_in & q(3 downto 1);
  END right_shift;
```

-- The actual data flows on `d(0 - 3)` outside the process. -- `d(0-3)` uses the Concatenate Operator (`&`) to create -- the four bit RSR. The process and d assignment are -- both executed concurrently.

**VHDL Bidirectional SR**

```vhdl
PROCESS(clk)
BEGIN
  IF(clk'EVENT AND clk = '1') THEN
    CASE direction IS
      WHEN '0' => q <= q(2 downto 0) & lsi; -- Left Shift
      WHEN '1' => q <= rsi & q(3 downto 1); -- Right Shift
      WHEN OTHERS => Null;
    END CASE;
  END IF;
END PROCESS;
END bidirectional_shift;
```

**Generic Width SR**

- Uses a VHDL Generic Clause in the Entity to specify a Width Variable
- General form is `GENERIC (Clause := Value)`
- For a 4 Bit SR we use
  ```vhdl
  GENERIC(Width : Positive := 4)
  ```
**VHDL Generic Entity**

Width set to 4 bits

```vhdl
ENTITY srt_bhv IS
    GENERIC (Width: POSITIVE := 4);
    PORT(
        serial_in : IN STD_LOGIC;
        clk : IN STD_LOGIC;
        q : BUFFER STD_LOGIC_VECTOR (Width-1 downto 0));
END srt_bhv;
```

**VHDL Generic Architecture**

```vhdl
PROCESS(clk)
BEGIN
    IF (clk'EVENT AND clk = '1') THEN
        q(width-1 downto 0) <= serial_in & q(width-1 downto 1);
    END IF;
END PROCESS;
```

**LPM Shift Registers**

- Configureable LPM shift register called \texttt{lpm\_shiftreg}
- Includes required and optional (GENERIC) parameters that are defined in such as LPM\_WIDTH, LPM\_DIRECTION (See Table 9.15)
- Design approach is the same as for Counters using Structured VHDL

**LPM SR Entity**

Remember to declare LPM library for use

```vhdl
ENTITY srg8_lpm IS
    PORT(
        clk : IN STD_LOGIC;
        serial_in : IN STD_LOGIC;
        serial_out : OUT STD_LOGIC);
END srg8_lpm;
```

**LPM SR Architecture (Component Statement)**

```vhdl
ARCHITECTURE lpm_shift OF srg8_lpm IS
    COMPONENT lpm_shiftreg
        GENERIC (LPM_WIDTH : POSITIVE);
        PORT (clock, shiftin : IN STD_LOGIC;
              shiftout : OUT STD_LOGIC);
    END COMPONENT;
    BEGIN
        shift_8 : lpm_shiftreg
            GENERIC MAP(LPM_WIDTH => 8);
            PORT MAP(clk, serial_in, serial_out);
    END lpm_shift;
    -- The generic map assigns a specific width of 8
    -- and the Port Map maps Entity Signals to LPM Signals
```