SEQUENCE GENERATOR

A sequence generator is a digital circuit that produces a predefined sequence of binary or numerical outputs. Sequence generators are commonly used in testing, encryption, communication systems, and state machines.

DESIGNING STEPS

- Find number of flip flops using L ≤ (2n − 1) where L is length of sequence, n indicate number of flip flop needed.
- Write truth table for this sequence using n flip-flops.
- Third step is the K-map optimization problem.

Binary Sequence Generator (1-0-1-1-0)

L ≤ (2n) here, L=5 hence , n=4

```
module sequence_generator_10110 (
   input clk,       // Clock signal
   input reset,     // Reset signal
   output reg seq_out // Sequence output
);
   // State encoding
   reg [2:0] state; // 3-bit state register to represent 5
states

   // State Definitions
   parameter S0 = 3'b000, // State for '1'
          S1 = 3'b001, // State for '0'
          S2 = 3'b010, // State for '1'
          S3 = 3'b011, // State for '1'
          S4 = 3'b100; // State for '0'

   // Sequential Logic for State Transition
   always @(posedge clk or posedge reset) begin
      if (reset)
         state <= S0; // Reset to the initial state
      else begin
         case (state)
            S0: state <= S1; // 1 -> 0
            S1: state <= S2; // 0 -> 1
            S2: state <= S3; // 1 -> 1
            S3: state <= S4; // 1 -> 0
            S4: state <= S0;
            default: state <= S0; // Default to initial state
         endcase
      end
   end

   // Output Logic
   always @(*) begin
      case (state)
         S0: seq_out = 1;
         S1: seq_out = 0;
         S2: seq_out = 1;
         S3: seq_out = 1;
         S4: seq_out = 0;
         default: seq_out = 0;
      endcase
   end
endmodule
```

Fibonacci Sequence Generator

Generates a Fibonacci sequence

$$F_n=F_{n-1}+F_{n-2}$$

```
module fibonacci_generator (
   input clk,
   input rst,
   output reg [3:0] fib
);
   reg [3:0] prev1, prev2;
 always @(posedge clk or posedge rst)
begin
      if (reset) begin
        prev1 <= 1; // Initialize first two no
          prev2 <= 0;
          fib <= 0;
      end else begin
          fib <= prev1 + prev2; // Fibonacci
          prev2 <= prev1;
          prev1 <= fib;
      end
   end
endmodule
```

Pseudo-Random seq

Uses Linear Feedback Shift Registers (LFSRs) to produce pseudo-random sequences.

```
module lfsr_sequence (
   input clk,
   input rst,
   output reg [3:0] lfsr
);
 always @(posedge clk or posedge rst)
begin
      if (reset)
         lfsr <= 4'b1001;
 // Initialize LFSR with non-zero value
      else
         lfsr <= {lfsr[2:0], lfsr[3] ^ lfsr[2]};
 // Feedback for randomness
   end
endmodule
```

Gray Code Sequence

Produces a sequence where only one bit changes between consecutive states.

```
module gray_code_generator (
   input clk,
   input reset,
   output reg [2:0] gray
);
   reg [2:0] binary;
   always @(posedge clk or posedge
reset) begin
      if (reset)
         binary <= 0;
      else
         binary <= binary + 1; // Increment in
binary
   end
   always @(*) begin
      gray = binary ^ (binary >> 1); // Binary
to Gray conversion
   end
endmodule
```

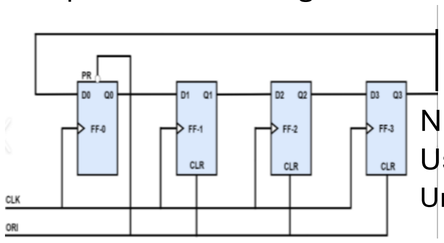
1. **Reset Behavior:** Ensure the generator starts with a known state.
2. **Timing and Clock:** Ensure timing constraints are met, especially in high-frequency applications.
3. **State Encoding:** Optimize the state representation for efficient hardware usage.
4. **Modularity:** Use parameters or constants to modify the sequence dynamically.

COUNTER

Synchronous Counter: synchronous counter has one global clock which drives each flip flop so output changes in parallel. $T_{delay} = t_{pd}$

Ring Counter

Formed by connecting the output of last stage back to input of the shift register.



N bit- MOD N

Used state – N

Unused state– 2^N -1

CLOCK FREQUENCY - F/2N

Drawback:

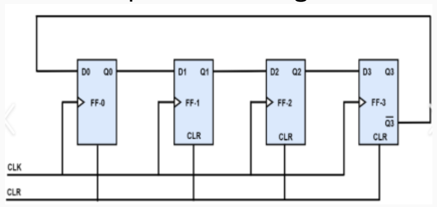
- Need to start the counter or need extra gate to make itself start.
- Can generate on N state

```
module ring_counter
(
   input wire clk,      // Clock input
   input wire reset,    // Active high reset
   output reg [3:0] q   // 4-bit ring counter output
);
   always @(posedge clk or posedge reset)
 begin
      if (reset)  begin
         q <= 4'b0001;  // Initialize the ring counter
         end
   else begin
         q <= {q[2:0], q[3]};  // Rotate left
      end
  end
endmodule
```

Johson counter

Also called twisted ring counter/switched tail / Mobile walking

Form by connecting complement of output of last stage back to input of shift register.



N bit- MOD N

Used state – 2N

Unused state – 2^N -2N

CLOCK FREQUENCY - F/2N

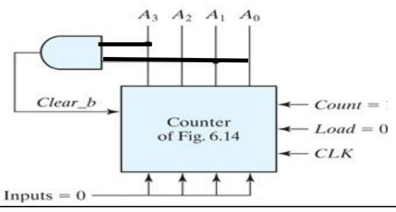
Drawback:

- High number of unused states.
- Enter to lock up state.

```
module johnson_counter(
   input wire clk,      // Clock input
   input wire reset,    // Active high reset
   output reg [3:0] q   // 4-bit Johnson counter output
);
   always @(posedge clk or posedge reset)
    begin
      if (reset)
      begin
         q <= 4'b0000;  // Initialize the counter
      end
 else
      begin
         q <= {q[2:0], ~q[3]};  // Shift left and invert the MSB
      end
   end
endmodule
```

BCD COUNTER

```
module bcd_counter(
   input wire clk,      // Clock input
   input wire reset,    // Active high reset
   output reg [3:0] bcd   // 4-bit BCD output    );
always @(posedge clk or posedge reset) begin
 if (reset) begin
  bcd <= 4'b0000;  // Reset BCD counter to 0000
 end else if (bcd == 4'b1001) begin
  bcd <= 4'b0000;  // Reset to 0000 when it reaches 9
 end else begin
   bcd <= bcd + 1;
    end
   end
endmodule
```



DECODER COUNTER

```
module decade_counter(
   input wire clk,      // Clock input
   input wire reset,    // Active high reset
   output reg [3:0] count   // 4-bit counter output);
 always @(posedge clk or posedge reset) begin
      if (reset) begin
         count <= 4'b0000;  // Reset counter to 0000
      end else if (count == 4'b1001) begin
         count <= 4'b0000;  // Reset to 0000 WHEN 9
      end else begin
         count <= count + 1;  // Increment counter by 1
      end
   end
endmodule
```

Asynchronous Counter: counter is a cascaded arrangement of flip-flops where the output of one flip-flop drives the clock input of the following flip-flop. $T_{delay} = t_{pd}$

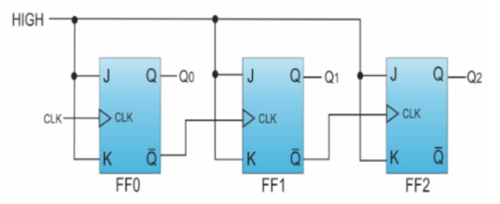
Ripple counter

Clk	edge	nature
Q		DOWN
Q ●	○	UP
Q’	○	UP
Q’ ●	○	DOWN

- All flip flop works in Toggle mode.
- Full state = 2n, Full state = UP +DOWN
- For DOWN counter use PRESET and for UP counter use CLEAR
- ACTIVE HIGH- AND / OR & ACTIVE LOW- NAND /NOR
- DUTY TIME = (ON TIME) / (ON TIME + OFF TIME)

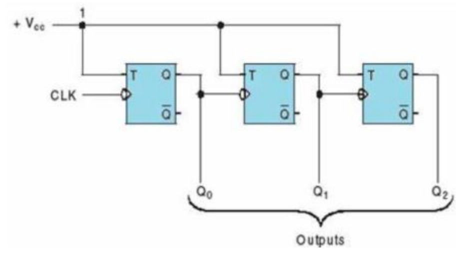
UP COUNTER

```
module up_counter(
   input wire clk,       // Clock input
   input wire reset,     // Active high reset
   output reg [2:0] count // 3-bit counter
output
);
   // Always block to handle counting on
positive clock edge or reset
   always @(posedge clk or posedge reset)
begin
      if (reset)
 begin
 count <= 3'b000; // Reset counter 000
      end
 else begin
 count <= count + 1;  // Increment by 1
      end
   end
endmodule
```



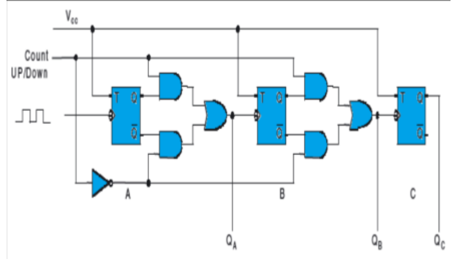
DOWN COUNTER

```
module down_counter(
   input wire clk,       // Clock input
   input wire reset,     // Active high
reset
   output reg [3:0] count // 4-bit counter
output
);
   // Always block to handle counting
on positive clock edge or reset
   always @(posedge clk or posedge
reset)
 begin
      if (reset) begin
  count <= 4'b1111;  // Reset to 1111
      end else begin
count <= count - 1;  // Decrement by 1
      end
   end
endmodule
```



UP/DOWN COUNTER

```
module up_down_counter(
   input wire clk,        // Clock input
   input wire rst,      // Active high
   input wire up_down,    // Control
   output reg [3:0] count
);
always @(posedge clk or posedge rst)
begin
      if (reset) begin
         count <= 4'b0000;  // Resst 0000
  end else
if (up_down) begin
   count <= count + 1;  // Increment (up)
      end else begin
count <= count - 1;
//Decrement(down)
      end
   end
endmodule
```



SYNCHRONOUS COUNTER	ASYNCRONOUS COUNTER
All flip flop are clocked at a time .	Only the first flip-flop is driven by the clock. The other flip-flops are triggered by the output of the preceding flip-flop.
Faster, since all flip-flops change state simultaneously on the clock edge.	Slower, due to the propagation delay between flip-flops.
More complex due to the requirement for synchronizing all flip-flops.	Simpler in design as only the first flip-flop is clocked.
Consumes more power, as all flip-flops are clocked simultaneously.	Generally, consumes less power as only one flip-flop receives the clock signal.
Errors are easier to control or detect due to synchronized state changes.	Difficult to control or detect errors due to the asynchronous nature.
Suitable for high-speed, reliable counting applications like in CPUs or complex digital systems.	Suitable for simple, low-frequency counters or applications where speed is not critical.