# Chapter 9: Sequential Logic Modules

Prof. Soo-Ik Chae

# Objectives

After completing this chapter, you will be able to:

- Describe how to model asynchronous and synchronous *D*-type flip-flops
- Describe how to model registers (data register, register file, and synchronous RAM)
- Describe how to model shift registers
- Describe how to model counters (ripple/synchronous counters and modulo *r* counters)
- Describe how to model sequence generators
- Describe how to model timing generators

# Basic Sequential Logic Modules

- Commonly used sequential logic modules:
  - Synchronizer
  - Finite state machine
  - Sequence detector
  - Data register
  - Shift register
  - CRC generator
  - Register file
  - Counters (binary, BCD, Johnson)
  - Timing generator
  - Clock generator
  - Pulse generator

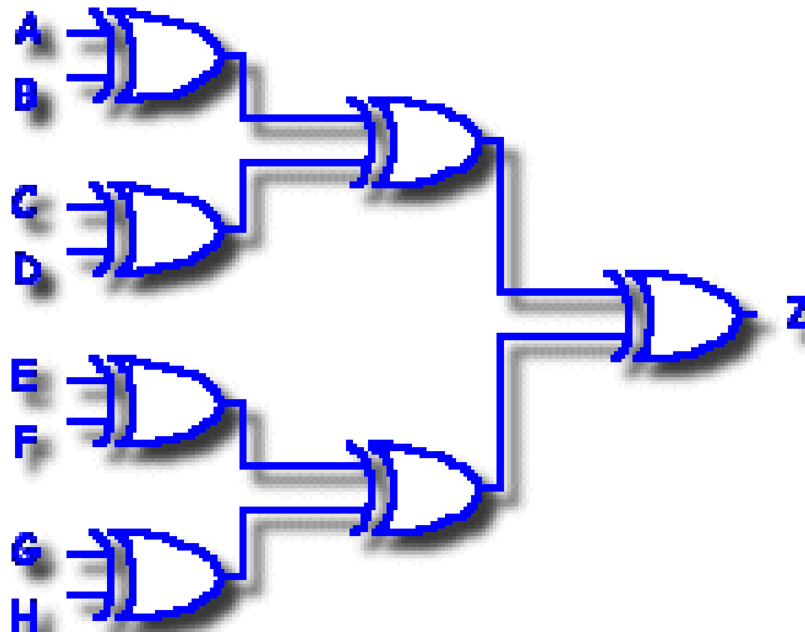
# Options for Modeling Sequential Logic

- Options for modeling sequential logic:
  - Behavioral statement
  - Task with delay or event control
  - Sequential UDP
  - Instantiated library register cell
  - Instantiated modules

# Libraries

- Library files contain all of the encapsulated and reusable modules for a design.
- These modules include
  - All flip-flops,
  - Memories,
  - Input, output, tri-state drivers
  - Clock generators
  - Parity trees
  - Muxes
  - Error-correcting encode/decode logic
  - Queues

# Parity tree



# Asynchronous Reset *D*-Type Flip-Flops

- Asynchronous reset *D*-type flip-flop
  - The reset signal is embodied into the event-sensitivity list of always statement.

```
// asynchronous reset D-type flip-flop
module DFF_async_reset (clk, reset_n, d, q);
input   clk, reset_n, d;
output reg q;
// the body of flip flop
always @(posedge clk or negedge reset_n)
   if (!reset_n) q <= 0; // reset_n signal is active-low.
   else          q <= d;
endmodule
```

  - How would you model a *D*-type latch?

# Inferred Latch

```
module latch(dout, din, le);
   input din;
   output dout;
   input  le; //  latch enable
   reg dout;
   always @(din or le)
     if (le == 1'b1)
       dout = din;
endmodule
```

- Three cases for inferred
  1. if statement with no else
  2. case conditions are not complete
  3. when sensitive list is not complete  (only in simulation)

Simulation and synthesis results can be inconsistent?!

# Synchronous Reset *D*-Type Flip-Flops

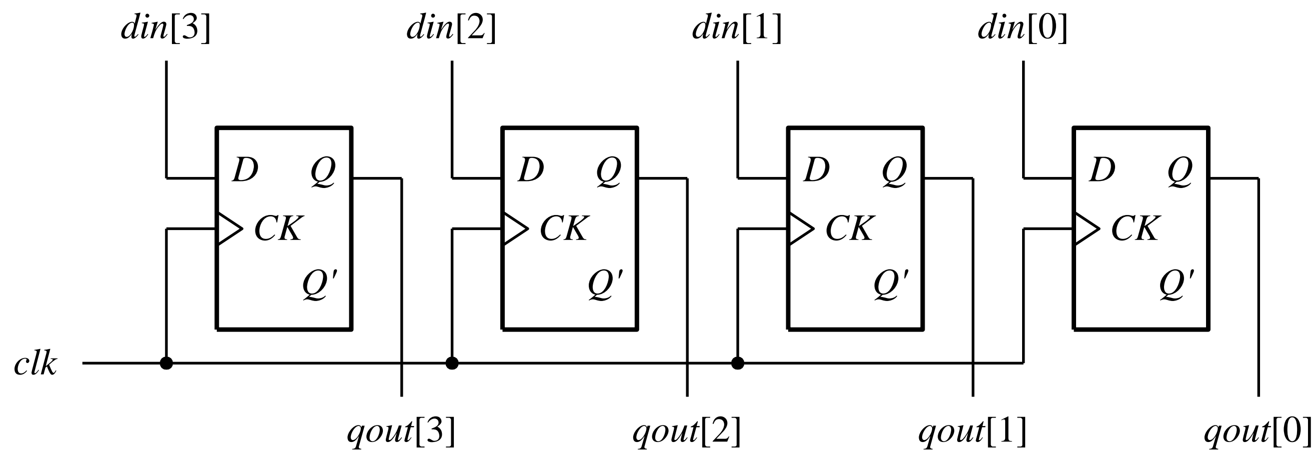
- Synchronous reset *D*-type flip-flop
  - The reset signal is not embodied into the event-sensitivity list of always statement.

```
// synchronous reset D-type flip-flop
module DFF_sync_reset (clk, reset, d, q);
input   clk, reset, d;
output reg q;
// the body of flip flop
always @(posedge clk)
   if (reset)  q <= 0; // reset signal is active-high.
   else        q <= d;
endmodule
```

# Registers

- Registers (or data registers) provide a means for storing information in any digital system.
- Types of registers used in digital systems:
  - Data register using flip-flops or latches.
  - Register file being used in data paths.
  - Synchronous RAM (random access memory) being used for storing moderate amount of information.
- A flip-flop may take up 10 to 20 times the area of a 6-transistor static RAM cell.
- In Xilinx FPGA, both register file and synchronous RAM are synthesized into static RAM structures consisting of LUTs or block RAMs.

# Data Registers




```
// an n-bit data register
module register(clk, din, qout);
parameter N = 4;   // number of bits
input   clk;
input   [N-1:0] din;
output reg [N-1:0] qout;
// the body of an n-bit data register
always @(posedge clk) qout <= din;
endmodule
```

# Data Registers

```
// an n-bit data register with asynchronous reset
module register_reset (clk, reset_n, din, qout);
parameter N = 4;   // number of bits
input   clk, reset_n;
input   [N-1:0] din;
output reg [N-1:0] qout;

// The body of an n-bit data register
always @(posedge clk or negedge reset_n)
   if (!reset_n) qout <= {N{1'b0}};
   else          qout <= din;
endmodule
```

Coding style:

- For active-low signal, end the signal name with an underscore followed by a lowercase letter b or n, such as reset_n.

# Data Registers

```
// an N-bit data register with synchronous load and
// asynchronous reset
module register_load_reset (clk, load, reset_n, din, qout);
parameter N = 4;   // number of bits
input  clk, load, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;

// the body of an N-bit data register
always @(posedge clk or negedge reset_n)
   if (!reset_n)    qout <= {N{1'b0}};
   else  if (load)  qout <= din;
   else qout <= qout;                    // a redundant expression
endmodule
```

Notice that: the else part is a redundant expression for sequential circuit. Why? Try to explain it.

# A Register File

```
// an N-word register file with one-write and two-read ports
module register_file(clk, rd_addra, rd_addrb, wr_addr, wr_enable, din, douta, doutb);
parameter M = 4;   // number of address bits
parameter N = 16;  // number of words, N = 2**M
parameter W = 8;   // number of bits in a word
input  clk, wr_enable;
input   [W-1:0]   din;
output [W-1:0]   douta, doutb;
input   [M-1:0]   rd_addra, rd_addrb, wr_addr;
reg      [W-1:0]   reg_file [N-1:0];

// the body of the N-word register file
assign douta = reg_file[rd_addra],
          doutb = reg_file[rd_addrb];
always @(posedge clk)
  if (wr_enable) reg_file[wr_addr] <= din;
endmodule
```

# An Synchronous RAM

```
// a synchronous RAM module example
module syn_ram (addr, cs, din, clk, wr, dout);
parameter N = 16;   // number of words
parameter A = 4;    // number of address bits
parameter W = 4;    // number of wordsize in bits
input  [A-1:0]  addr;
input  [W-1:0] din;
input   cs, wr, clk;   // chip select, read-write control, and clock signals
output reg [W-1:0] dout;
reg    [W-1:0] ram [N-1:0];  // declare an N * W memory array

// the body of synchronous RAM
always @(posedge clk)
   if (cs)
         if (wr) ram[addr] <= din;
         else        dout <= ram[addr];
endmodule
```

# Confusion!

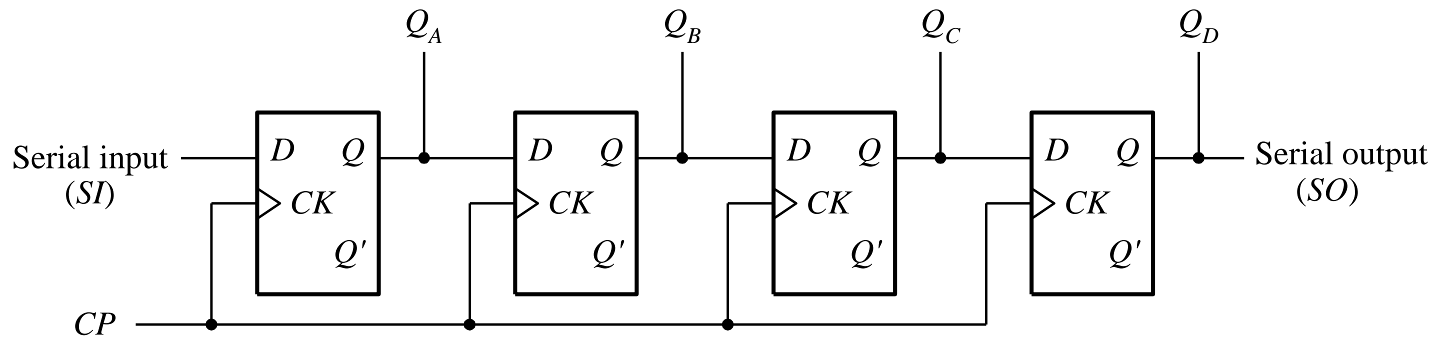
```
if (cs)
        if (wr) ram[addr] <= din;
        else        dout <= ram[addr];
//always associating the else with the closest previous if that lacks an else.
```

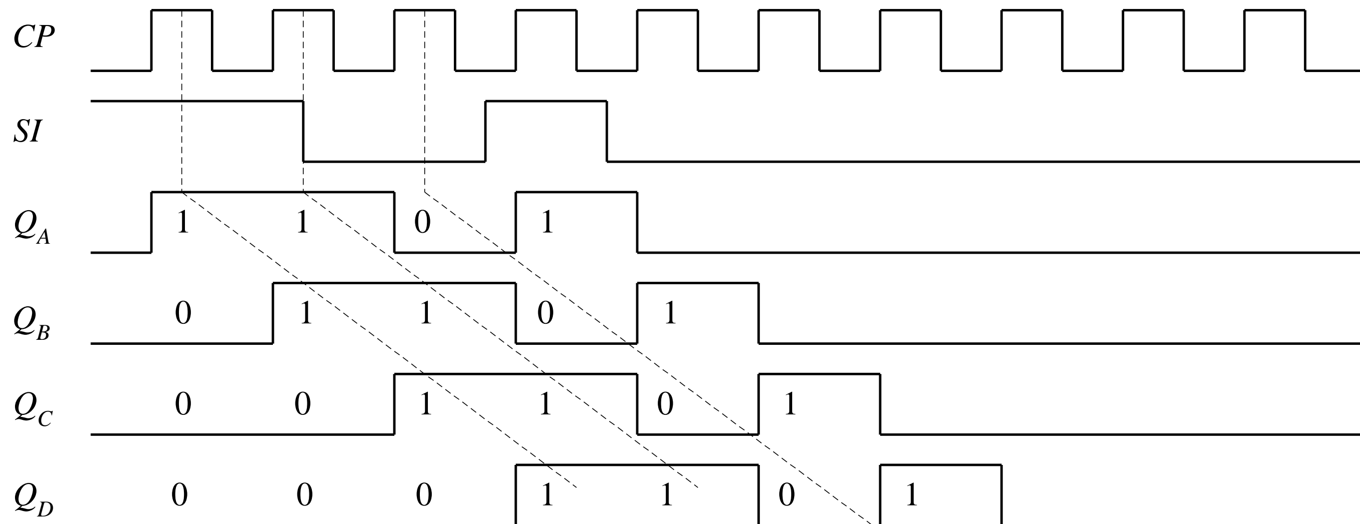
```
if (cs)   begin
        if (wr) ram[addr] <= din;
end
else        dout <= ram[addr];
// a begin-end block statement shall be used to force the proper assoication
```

# Shift Registers

- Shift registers perform left or right shift operation.
- Parallel/serial format conversion:
  - SISO (serial in serial out)
  - SIPO (serial in parallel out)
  - PISO (parallel in serial out)
  - PIPO (parallel in parallel out)

# Shift Registers



# Shift Registers

```
// a shift register module example
module shift_register(clk, reset_n, din, qout);
parameter N = 4;   // number of bits
input  clk, reset_n;
input  din;
output reg [N-1:0] qout;

// the body of an N-bit shift register
always @(posedge clk or negedge reset_n)
   if (!reset_n) qout <= {N{1'b0}};
   else          qout <= {din, qout[N-1:1]};
endmodule
```

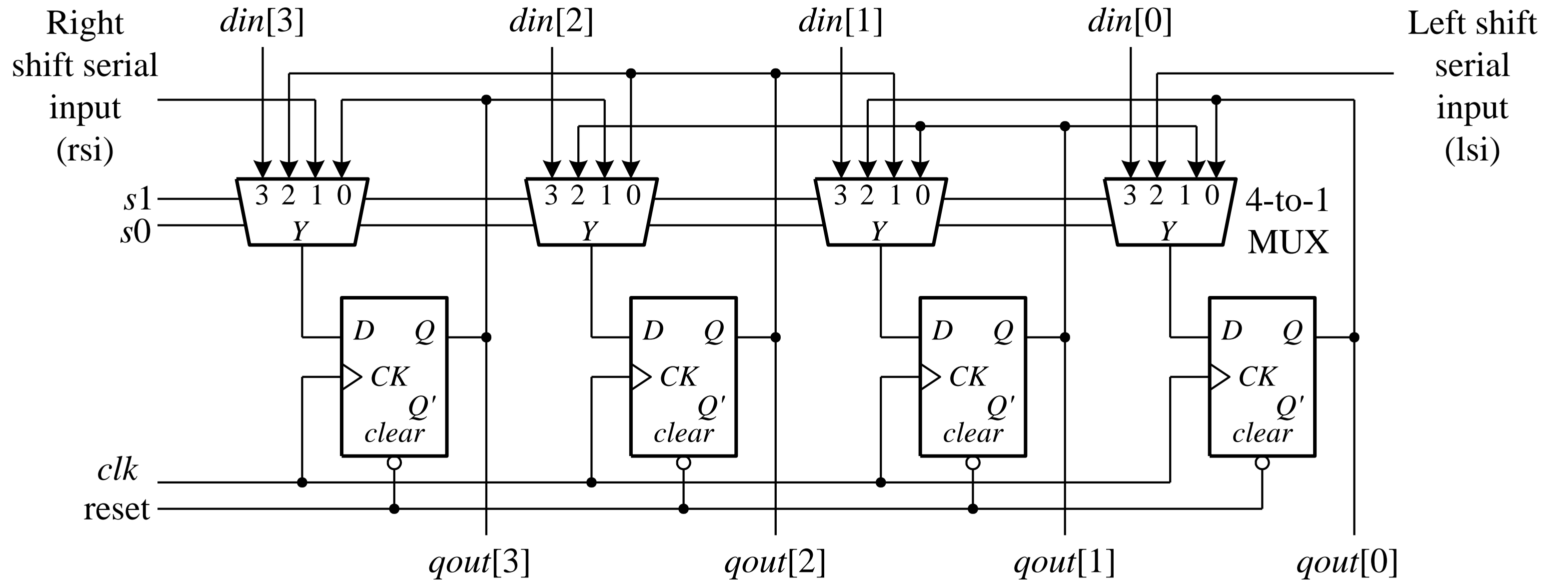
# A Shift Register with Parallel Load

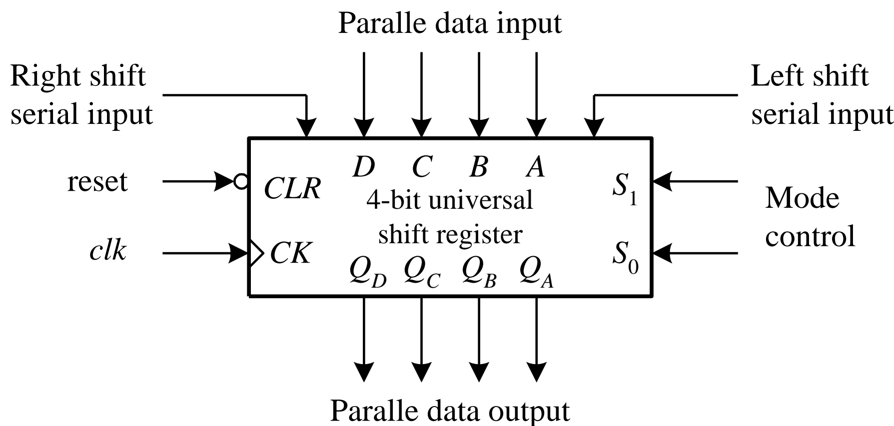
```
// a shift register with parallel load module example
module shift_register_parallel_load(clk, load, reset_n, din, sin, qout);
parameter N = 8;  // number of bits
input  sin, clk, load, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;

// the body of an N-bit shift register
always @(posedge clk or negedge reset_n)
   if (!reset_n)  qout <= {N{1'b0}};
   else if (load) qout <= din;
   else           qout <= {sin, qout[N-1:1]};
endmodule
```

# Universal Shift Registers

- A universal shift register can carry out
  - SISO (serial in serial out)
  - SIPO (serial in parallel out)
  - PISO (parallel in serial out)
  - PIPO (parallel in parallel out)
- The register must have the following capabilities:
  - Parallel load
  - Serial in and serial out
  - Shift left and shift right

# Universal Shift Registers




| $s1$ | $s0$ | Function |
|---|---|---|
| 0 | 0 | No change |
| 0 | 1 | Right shift |
| 1 | 0 | Left shift |
| 1 | 1 | Load data |

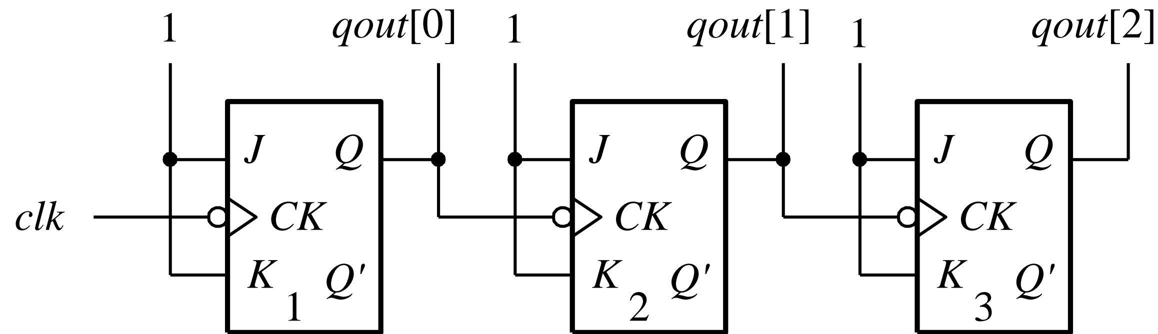
# Universal Shift Registers

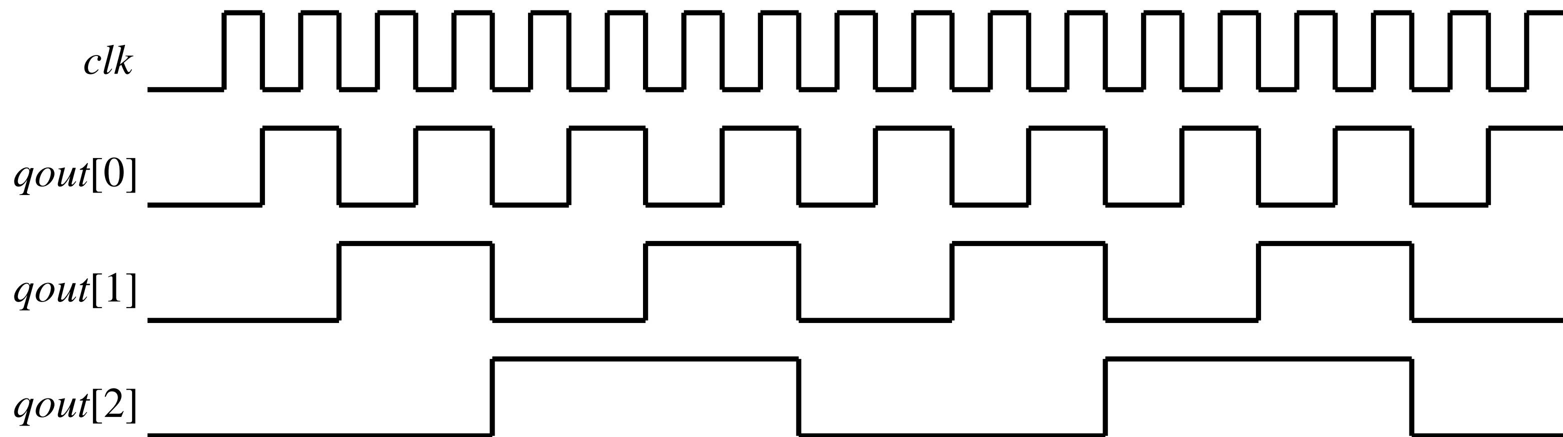
```
// a universal shift register module
module universal_shift_register (clk, reset_n, s1, s0, lsi, rsi, din, qout);
parameter N = 4;  // define the size of the universal shift register
input  s1, s0, lsi, rsi, clk, reset_n;
input  [N-1:0] din;
output reg [N-1:0] qout;
// the shift register body
always @(posedge clk or negedge reset_n)
   if (!reset_n) qout <= {N{1'b0}};
   else case ({s1,s0})
        2'b00: ; // qout <= qout;                 // No change
        2'b01: qout <= {lsi, qout[N-1:1]};   // Shift right
        2'b10: qout <= {qout[N-2:0], rsi};   // Shift left
        2'b11: qout <= din;                       // Parallel load
   endcase
endmodule
```

# Counters

- Counter is a device that counts the input events such as input pulses or clock pulses.
- Types of counters:
  - Asynchronous
  - Synchronous
- Asynchronous (ripple) counters:
  - Binary counter (up/down counters)
- Synchronous counters:
  - Binary counter (up/down counters)
  - BCD counter (up/down counters)
  - Gray counters (up/down counters)

# Binary Ripple Counters



# Binary Ripple Counters

```
// a 3-bit ripple counter module example
module ripple_counter(clk, qout);
input  clk;
output reg [2:0] qout;
wire   c0, c1;
// the body of the 3-bit ripple counter
assign c0 = qout[0], c1 = qout[1];
always @(negedge clk)
    qout[0] <= ~qout[0];
always @(negedge c0)
    qout[1] <= ~qout[1];
always @(negedge c1)
    qout[2] <= ~qout[2];
endmodule
```

- The output cannot be observed from simulators due to lacking initial values of qout. => reset is required!

# Binary Ripple Counters

```
// a 3-bit ripple counter with enable control
module ripple_counter_enable(clk, enable, reset_n, qout);
input  clk, enable, reset_n;
output reg [2:0] qout;
wire   c0, c1;
// the body of the 3-bit ripple counter
assign c0 = qout[0], c1 = qout[1];
always @(posedge clk or negedge reset_n)
   if (!reset_n) qout[0] <= 1'b0;
   else if (enable)   qout[0] <= ~qout[0];
always @(posedge c0 or negedge reset_n)
   if (!reset_n) qout[1] <= 1'b0;
   else if (enable)   qout[1] <= ~qout[1];
always @(posedge c1 or negedge reset_n)
   if (!reset_n) qout[2] <= 1'b0;
   else if (enable)   qout[2] <= ~qout[2];
endmodule
```

# A Binary Counter Example

```
// an n-bit binary counter with synchronous reset  and enable control.
module binary_counter(clk, enable, reset, qout, cout);
parameter N = 4;
input  clk, enable, reset;
output reg [N-1:0] qout;
output rco;

wire cout;       // carry output

// the body of the N-bit binary counter
always @(posedge clk)
   if (reset)            qout <= {N{1'b0}};
   else if (enable)  qout <= qout + 1;
// generate carry output
assign  #1 cout = &qout;
endmodule
```

Q: What is the difference?

```
{cout, qout} <= qout + 1;
```

Q: What if without the delay timing control?

```
assign  cout = &qout;
```

# A Binary Counter Example

```
// an n-bit binary counter with synchronous reset  and enable control.
module binary_counter(clk, enable, reset, qout, rco);
parameter N = 4;
input  clk, enable, reset;
output reg [N-1:0] qout;
output rco;

wire cout;      // carry output

// the body of the N-bit binary counter
always @(posedge clk)
  if (reset)          qout <= {N{1'b0}};
  else if (enable)  qout <= qout + 1;
// generate carry output
assign   cout = &qout;
always @(negedge clk)
  rco <= cout;
endmodule
```

# Binary Up/Down Counters --- version 1

```
// an N-bit binary up/down counter with synchronous reset and enable control.
module binary_up_down_counter_reset(clk, enable, reset, upcnt, qout, cout, bout);
parameter N = 4;
input  clk, enable, reset, upcnt;
output reg [N-1:0] qout;
output cout, bout;  // carry and borrow outputs
// the body of N-bit up/down binary counter
always @(posedge clk)
   if (reset) qout <= {N{1'b0}};
   else if (enable) begin
              if (upcnt) qout <= qout + 1;
              else         qout <= qout - 1;  end
 //       else  qout <= qout;  // a redundant expression
 // Generate carry and borrow outputs
 assign #1 cout = &qout;  // Why #1 is required ?
 assign #1 bout = |qout;
endmodule
```

# Binary Up/Down Counters --- version 2

```
// an N-bit up/down binary counter with synchronous reset and enable control.
module up_dn_bin_counter(clk, reset, eup, edn, qout, cout, bout);
Parameter N = 4;
// Enable up count (eup) and enable down count (edn)
// cannot be set to one simultaneously.
input   clk, reset, eup, edn;
output reg [N-1:0] qout;
output cout, bout;
// the body of the N-bit binary counter
always @(posedge clk)
   if (reset)       qout <= {N{1'b0}};    // synchronous reset
   else if (eup)  qout <= qout + 1;
   else if (edn)  qout <= qout - 1;
assign #1 cout = (&qout)& eup; // generate carry out
assign #1 bout = (~|qout)& edn; // generate borrow out
endmodule
```

# Overriding Parameters

- Using the defparam statement

```
module counter_nbits (clock, clear, qout);
parameter N = 4;  // define counter size
input   clock, clear;
output reg [N-1:0] qout;
always @( posedge clear or negedge clock)
begin                    // qout <= (qout + 1) % 2^n;
     if (clear) qout <= {N{1'b0}};
     else         qout <= (qout + 1) ;
end
endmodule
```

```
// define top level module
module two_counters(clock, clear, qout4b, qout8b);
input   clock, clear;
output [3:0] qout4b;
output [7:0] qout8b;
// instantiate two counter modules
defparam cnt_4b.N = 4, cnt_8b.N = 8; // using a hierarchical name of the parameter
counter_nbits cnt_4b (clock, clear, qout4b);
counter_nbits cnt_8b (clock, clear, qout8b);
endmodule
```

# Overriding Parameters

- Using module instance parameter value assignment--- one parameter

```
module counter_nbits (clock, clear, qout);
parameter N = 4;  // define counter size
input   clock, clear;
output reg [N-1:0] qout;
always @( posedge clear or negedge clock)
begin                    // qout <= (qout + 1) % 2^n;
      if (clear) qout <= {N{1'b0}};
      else         qout <= (qout + 1) ;
end
endmodule
```

```
// define top level module
module two_counters(clock, clear, qout4b, qout8b);
input   clock, clear;
output [3:0] qout4b;
output [7:0] qout8b;
// instantiate two counter modules
counter_nbits #(4) cnt_4b (clock, clear, qout4b);
counter_nbits #(8) cnt_8b (clock, clear, qout8b);
endmodule
```

# Overriding Parameters

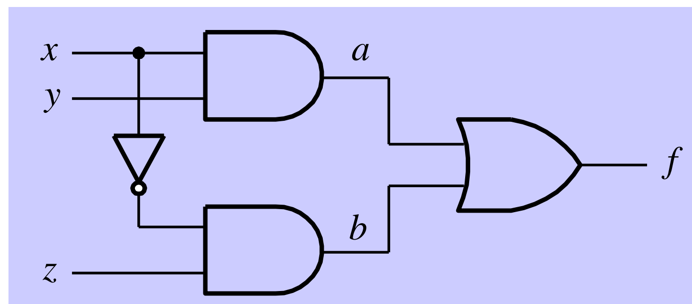
❖ Using module instance parameter value assignment --- two parameters

```
// define top level module
module parameter_overriding_example(x, y, z, f);
input  x, y, z;
output f;
hazard_static #(4, 8) example (x, y, z, f);
endmodule
```

```
module hazard_static (x, y, z, f);
parameter delay1 = 2, delay2 = 5;
input x, y, z;
output f;
wire  a, b, c;      // internal net
// logic circuit body
   and  #delay2 a1 (b, x, y);
   not  #delay1  n1 (a, x);
   and  #delay2 a2 (c, a, z);
   or    #delay2 o2 (f, b, c);
endmodule
```

```
hazard_static #(.delay2(4), .delay1(6)) example (x, y, z, f);
```

- parameter value assignment by name --- minimize the chance of error!



# Binary Up/Down Counters --- version 2

```
// an example to illustrating the cascade of two up/down counters.
module up_dn_bin_counter_cascaded(clk, reset,eup, edn, qout, cout, bout);
parameter N = 4;
input  clk, reset, eup, edn;
output [2*N-1:0] qout;
output cout, bout;

// declare internal nets for cascading both counters
wire   cout1, bout1;

// The body of the cascaded up/down counter
   up_dn_bin_counter #(4) up_dn_cnt1 (clk, reset,eup, edn, qout[3:0], cout1, bout1);
   up_dn_bin_counter #(4) up_dn_cnt2 (clk, reset,cout1, bout1, qout[7:4], cout, bout);
endmodule
```

# A Modulo r Binary Counter (1)

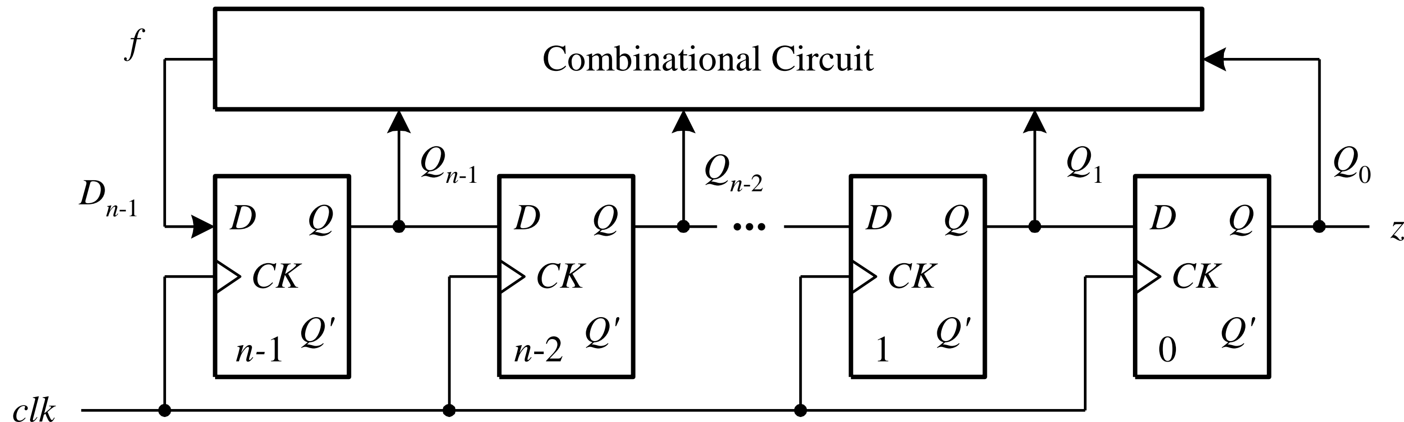
```
// the body of modulo r binary counter with synchronous reset and enable control.
module modulo_r_counter(clk, enable, reset, qout, cout);
parameter N = 4;
parameter R= 10;  // BCD counter
input  clk, enable, reset;
output reg [N-1:0] qout;
output cout;  // carry output
// the body of modulo r binary counter.
assign #1 cout = (qout == R - 1);
always @(posedge clk)
    if (reset)  qout <= {N{1'b0}};
    else begin
        if (enable) if (cout) qout <= 0;
                    else        qout <= qout + 1;
    end
endmodule
```

# A Modulo r Binary Counter (2)

```
// the body of modulo r binary counter with synchronous reset and enable control.
module modulo_r_counter(clk, enable, reset, qout, cout);
parameter N = 4;
parameter R= 10;  // BCD counter
input  clk, enable, reset;
output reg [N-1:0] qout;
output cout;  // carry output
wire cx;
// the body of modulo r binary counter.
assign cx = (qout == R - 1);
always @(negedge clk)
  cout <= cx;
always @(posedge clk)
    if (reset)  qout <= {N{1'b0}};
    else begin
        if (enable) if (cout) qout <= 0;
                    else       qout <= qout + 1;
    end
endmodule
```

# Sequence Generators

- In this lecture, we focus on the following three circuits:
  - PR (pseudo random)-sequence generator
  - Ring counter
  - Johnson counter




The general paradigm of sequence generator

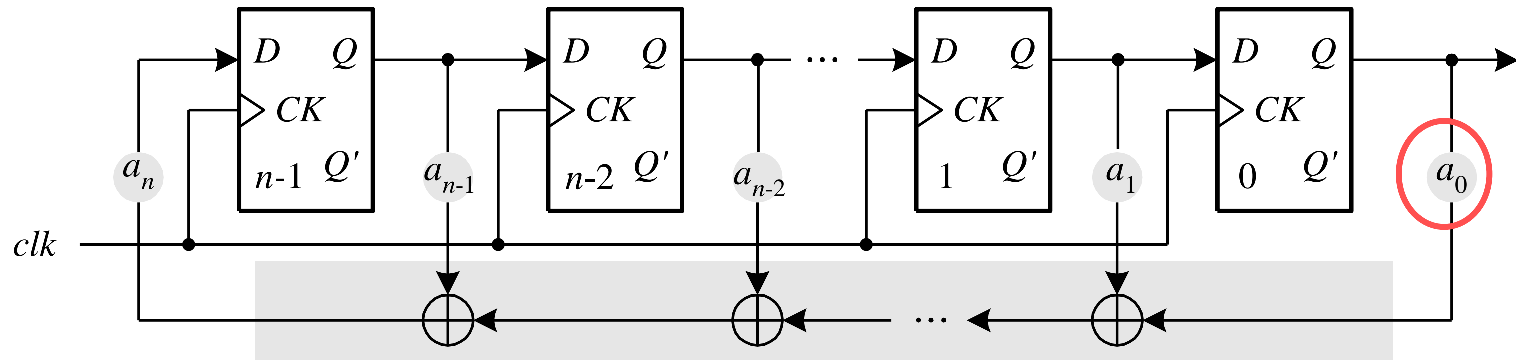
# Primitive Polynomials

| $n$ | $f(x)$ | $n$ | $f(x)$ | $n$ | $f(x)$ |
|---|---|---|---|---|---|
| 1, 2, 3, 4, 6, 7, 15, 22, 60 | $1+x+x^n$ | 24 | $1+x+x^2+x^7+x^n$ | 43 | $1+x+x^5+x^6+x^n$ |
| | | 26 | $1+x+x^2+x^6+x^n$ | 44, 50 | $1+x+x^{26}+x^{27}+x^n$ |
| 5, 11, 21, 29 | $1+x^2+x^n$ | 30 | $1+x+x^2+x^{23}+x^n$ | 45 | $1+x+x^3+x^4+x^n$ |
| 10, 17, 20, 25, 28, 31, 41, 52 | $1+x^3+x^n$ | 32 | $1+x+x^2+x^{22}+x^n$ | 46 | $1+x+x^{20}+x^{21}+x^n$ |
| | | 33 | $1+x^{13}+x^n$ | 48 | $1+x+x^{27}+x^{28}+x^n$ |
| 9 | $1+x^4+x^n$ | 34 | $1+x+x^{14}+x^{15}+x^n$ | 49 | $1+x^9+x^n$ |
| 23, 47 | $1+x^5+x^n$ | 35 | $1+x^2+x^n$ | 51, 53 | $1+x+x^{15}+x^{16}+x^n$ |
| 18 | $1+x^7+x^n$ | 36 | $1+x^{11}+x^n$ | 54 | $1+x+x^{36}+x^{37}+x^n$ |
| 8 | $1+x^2+x^3+x^4+x^n$ | 37 | $1+x^2+x^{10}+x^{12}+x^n$ | 55 | $1+x^{24}+x^n$ |
| 12 | $1+x+x^4+x^6+x^n$ | 38 | $1+x+x^5+x^6+x^n$ | 56,59 | $1+x+x^{21}+x^{22}+x^n$ |
| 13 | $1+x+x^3+x^4+x^n$ | 39 | $1+x^4+x^n$ | 57 | $1+x^7+x^n$ |
| 14, 16 | $1+x^3+x^4+x^5+x^n$ | 40 | $1+x^2+x^{19}+x^{21}+x^n$ | 58 | $1+x^{19}+x^n$ |
| 19, 27 | $1+x+x^2+x^5+x^n$ | 42 | $1+x+x^{22}+x^{23}+x^n$ | | |

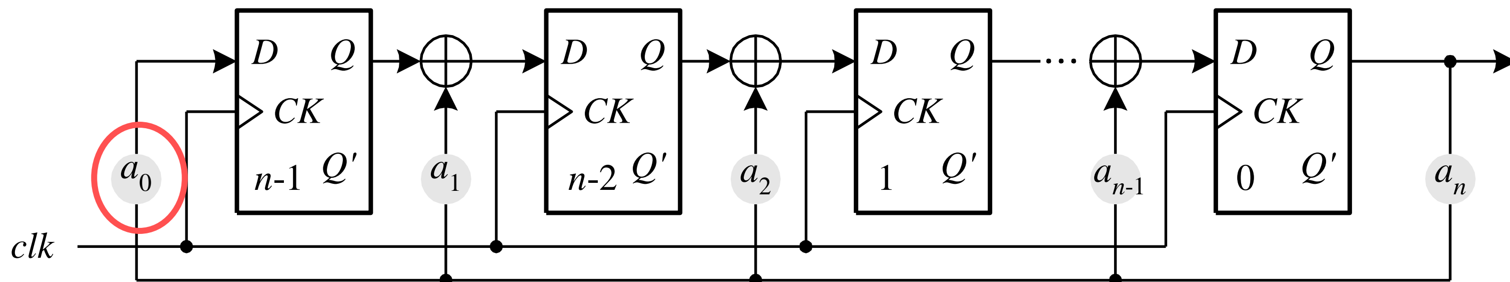
$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

# Maximal Length Sequence Generators

$$f(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$



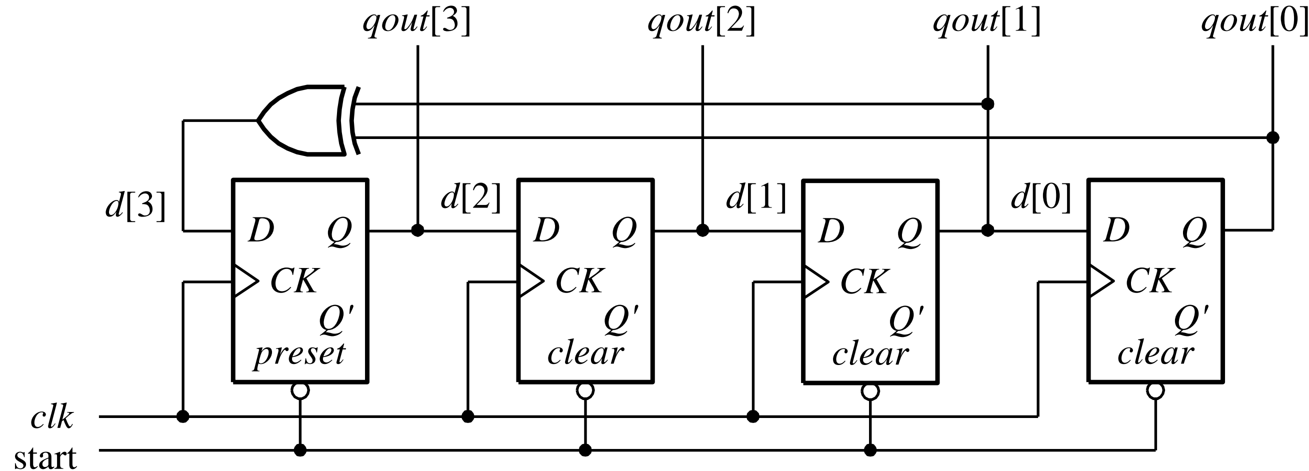

(a) Standard format




(b) Modular format

# A PR-Sequence Generator Example

- An example of 4-bit pr-sequence generator:
  - primitive polynomial: $1 + x + x^4$




- Using start to set the initial to 1000, the circuit will start from state 4'b1000 after reset signal is applied.

# A PR-Sequence Generator Example

```
// an N-bit pr_sequence generator module --- in standard form
module pr_sequence_generate (clk, qout);
parameter N = 4;  // define the default size
parameter [N:0] tap = 5'b10011;
input  clk;
output reg [N-1:0] qout = 4'b0100;
wire   d;

// pseudo-random sequence generator body
assign d = ^(tap[N-1:0] & qout[N-1:0]);
always @(posedge clk)
     qout <= {d, qout[N-1:1]};
endmodule
```

qout = 4’b0100 for simulation only. Without the initial value, simulators cannot calculate qout and hence we could not observe the qout values.

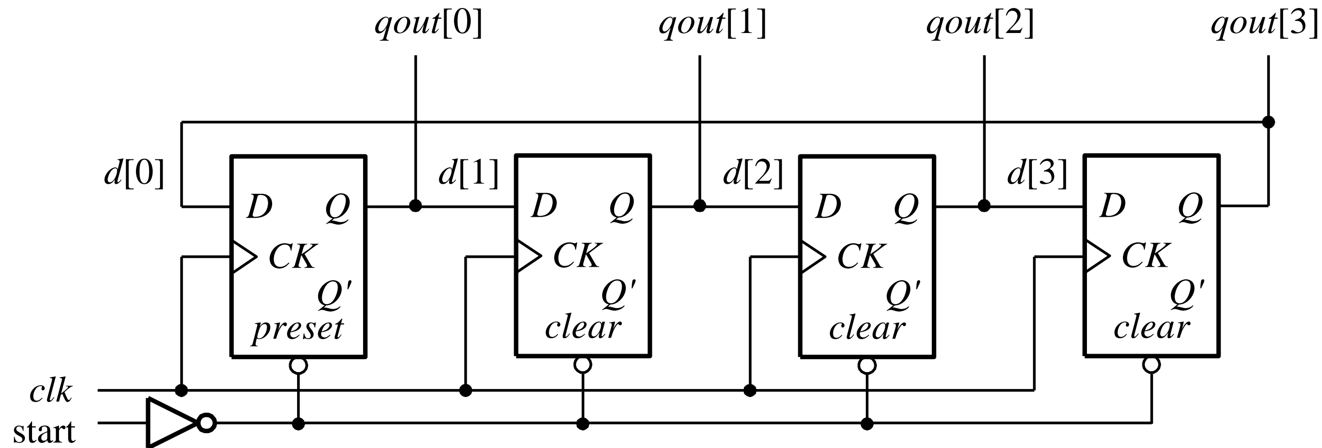
Q: Write an N-bit pr_sequence generator in modular form.

# A PR-Sequence Generator Example

```
// an N-bit pr_sequence generator module --- in standard form
module pr_sequence_generate (clk, start, qout);
parameter N = 4;  // define the default size
parameter [N:0] tap = 5'b10011;
input  clk, start;
output reg [N-1:0] qout;
wire   d;
// pseudo-random sequence generator body
assign d = ^(tap[N-1:0] & qout[N-1:0]);
always @(posedge clk or posedge start)
   if (start)  qout <= {1'b1, {N-1{1'b0}}};
   else        qout <= {d, qout[N-1:1]};
endmodule
```

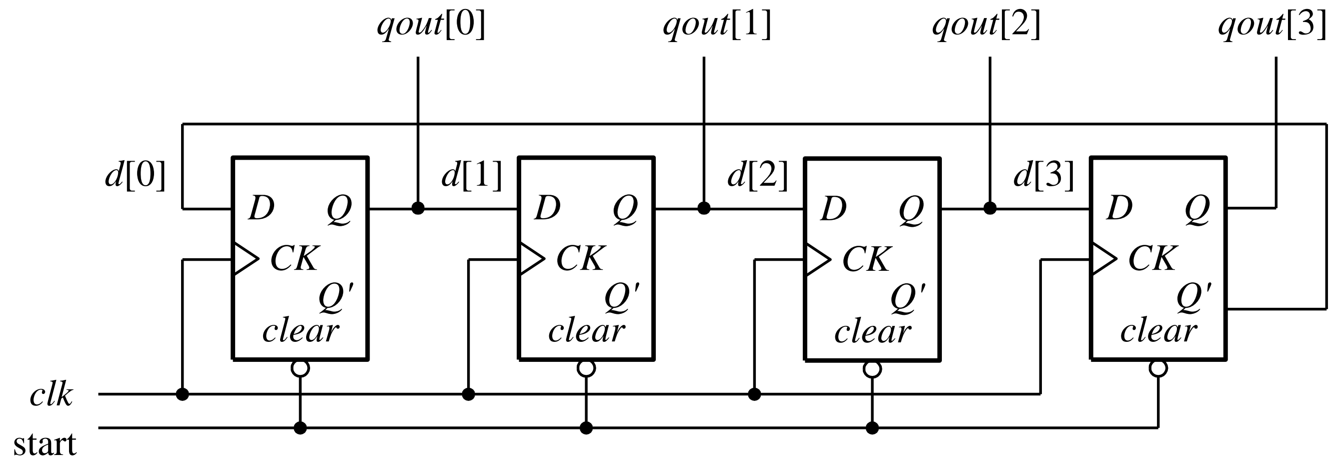
- Using start to set the initial value to 4’b1000, hence simulators can calculate qout and hence we could observe the qout values.
- Of course, the circuit will start from state 4’b1000 after reset signal is applied.

# Ring Counters




```
// a ring counter with initial value
module ring_counter(clk, start, qout);
parameter N = 4;
input  clk, start;
output reg [0:N-1] qout;
// the body of ring counter
always @(posedge clk or posedge start)
  if (start)  qout <= {1'b1,{N-1{1'b0}}};
  else        qout <= {qout[N-1], qout[0:N-2]};
endmodule
```

# Johnson Counters

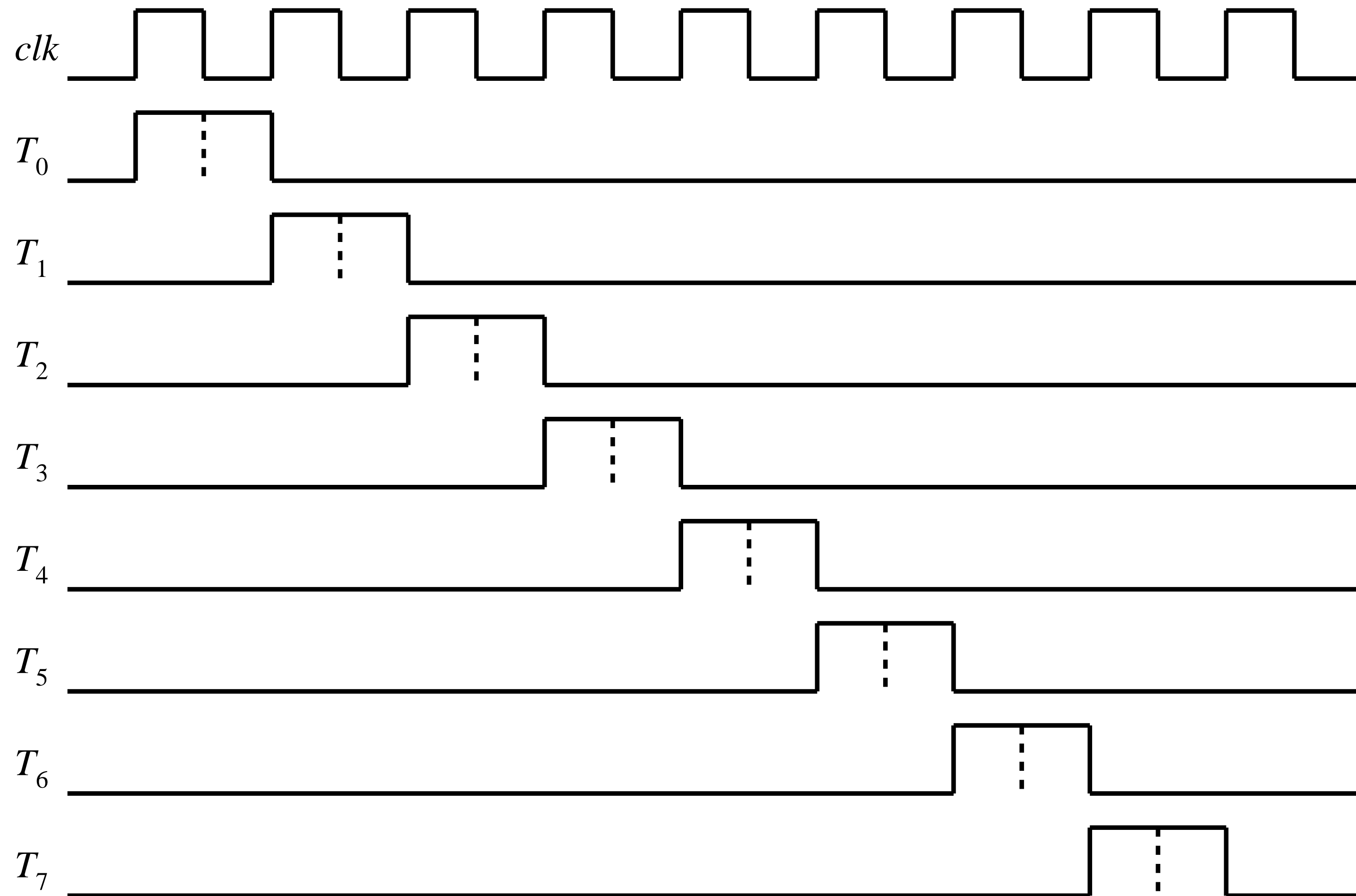



```
// Johnson counter with initial value
module ring_counter(clk, start, qout);
parameter N = 4; // define the default size
input  clk, start;
output reg [0:N-1] qout;
// the body of Johnson counter
always @(posedge clk or posedge start)
    if (start)  qout <= {N{1'b0}};
    else        qout <= {~qout[N-1], qout[0:N-2]};
endmodule
```

# Timing Generators

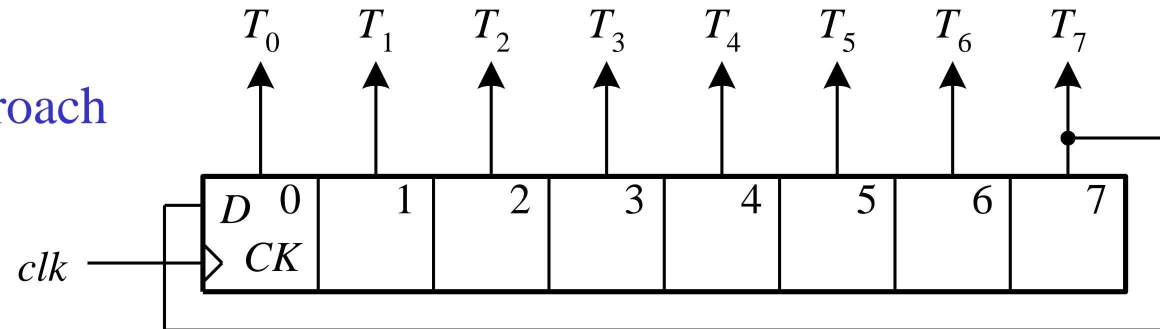
- A timing generator is a device that generates timings required for specific application.
- Multiphase clock signals:
  - Ring counter
  - Binary counter with decoder
- Digital monostable circuits:
  - Retriggerable
  - Nonretriggerable

# A Multiphase Clock Signal



# Multiphase Clock Generators

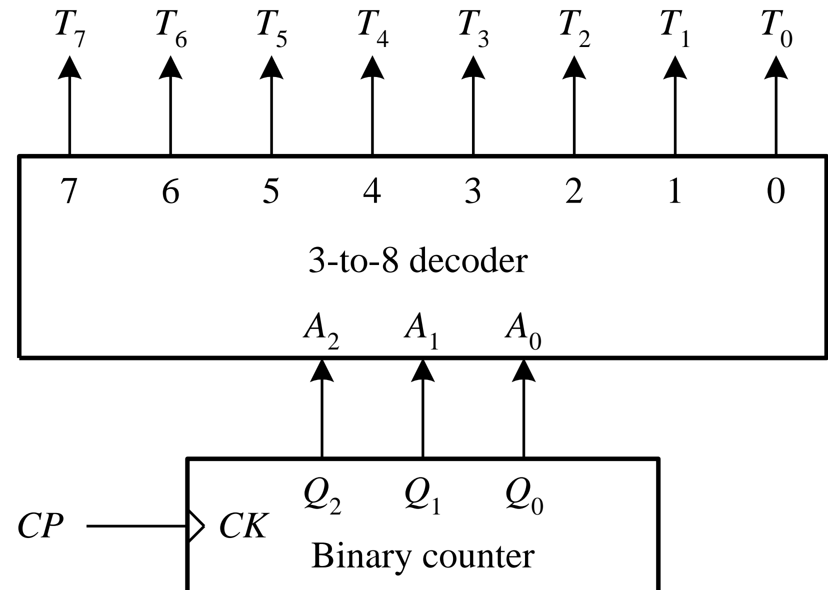
(a) Ring counter approach



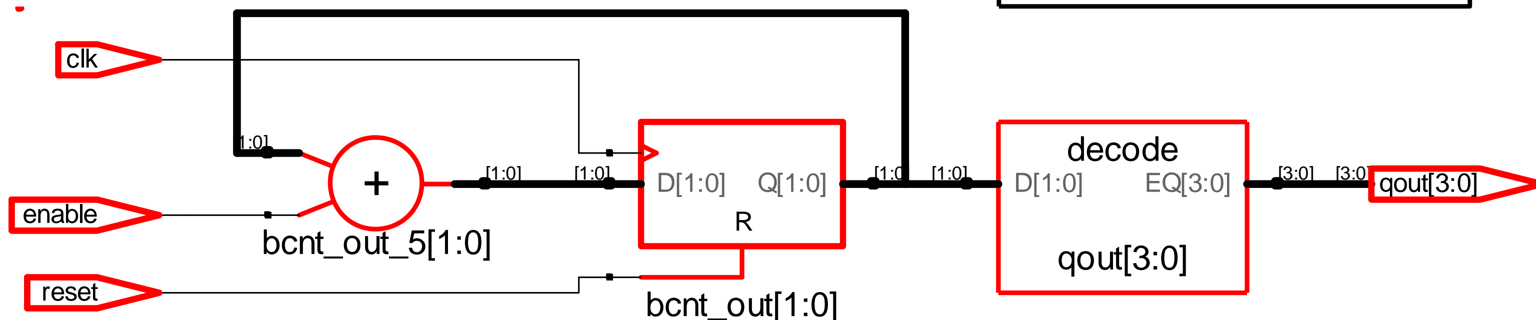

```
// a ring counter with initial value serve as a timing generator
module ring_counter_timing_generator(clk, reset, qout);
parameter n = 4;  // define the counter size
input  clk, reset;
output reg [0:n-1] qout;
// the body of ring counter
always @(posedge clk or posedge reset)
  if (reset)  qout <= {1'b1,{n-1{1'b0}}};
  else        qout <= {qout[n-1], qout[0:n-2]};
endmodule
```

# Multiphase Clock Generators

❖ Binary counter with decoder approach




Synthesized output from SynplifyPro with n = 4 and m =2. (Program is listed on the next page.)



# Multiphase Clock Generators

```
// a binary counter with decoder serve as a timing generator
module binary_counter_timing_generator(clk, reset, enable, qout);
parameter N = 8;   // define the number of phases
parameter M = 3;  // define the bit number of binary counter
input  clk, reset, enable;
output reg [N-1:0] qout;
reg    [M-1:0] bcnt_out;
// the body of binary counter
always @(posedge clk or posedge reset)
  if (reset)  bcnt_out <= {M{1'b0}};
  else if (enable) bcnt_out <= bcnt_out + 1;
// decode the output of the binary counter
always @(bcnt_out)
   qout = {N-1{1'b0},1'b1} << bcnt_out;
endmodule
```