Module 3

- **Sequential Basics**: Storage elements Counters[book 2], Sequential circuit timing : Propagation Delays, Setup, and Hold Times, Timing Conditions for Proper Operations, Glitches In Sequential Circuits, Synchronous Design. Tristate Logic and Busses [book 1].
- Design flow of ASIC and FPGA based systems. Logic Synthesis, Synthesis Design Flow Coding guidelines[book1,2,3,6]

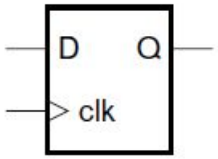
- Sequential circuits are the mainstay of digital systems.
- Sequential circuit elements that are widely used in digital systems for storing information and for counting events. We then see how a system can be built from two main sections: a data- path and a control section.
- Discussion of a clocked synchronous timing methodology based on the abstraction of discrete time. This methodology is central to design of complex digital systems.

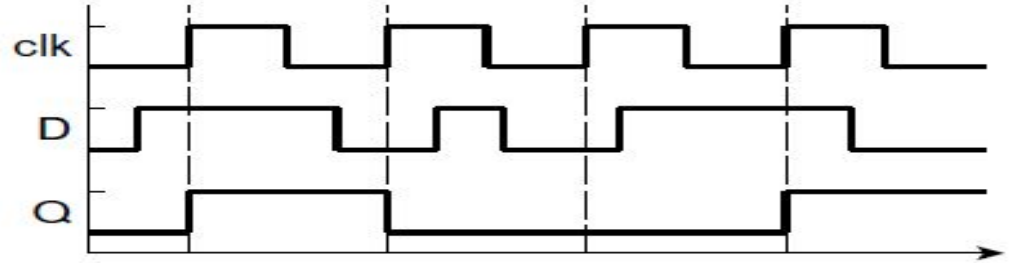
S TO R A GE EL EMEN TS

- Sequential circuit as one whose outputs depend not only on the current values of inputs, but also on the previous values of inputs.
- Sequential circuits are commonly regu- lated by a periodic clock signal that divides the passage of time into dis- crete clock cycles.

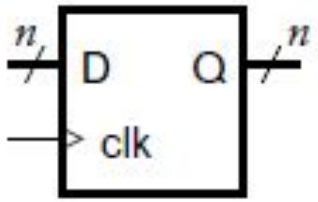
D Flip Flop

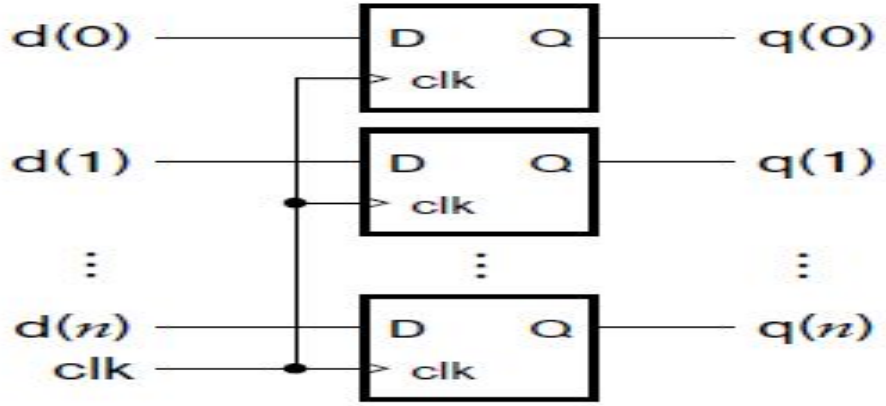
- The flip-flop is edge-triggered, meaning that on each rising edge of the clk input, the current value of the D input is stored within the flip-flop and reflected on the Q output. We illustrated use of D flip-flops in sequential circuits in where we stored the previous two values of an input signal on successive clock edges so that we could detect a given sequence of input values.
- While it is possible to implement a flip-flop as a combination of gates, it is not very instructive to do so. Moreover, flip-flops are provided as
-



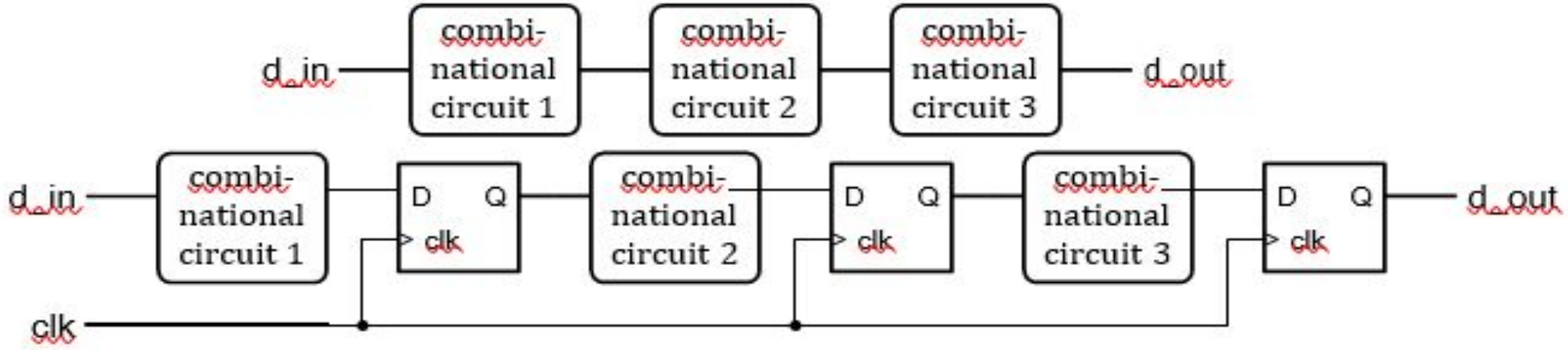


- A group of flip-flops used in this way is called a *register*. Each flip-flop in the register stores one bit of the code word of the stored value, as shown in Figure . The circuit at the top of the figure shows that each bit of an input and an output signal is connected to the input and output, respectively, of one of the flip-flops, and that the clock signal is connected in common to the clock input of all of the flip-flops. When there is a rising edge on the clock input, each flip- flop in the register updates its stored bit from the signal connected to its data input and drives the new value on its data output. The symbol for the register is shown





Circuit composed of combinational subcircuits (top), and a pipeline containing the same subcircuits.

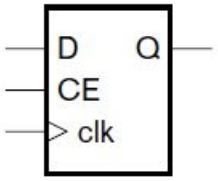


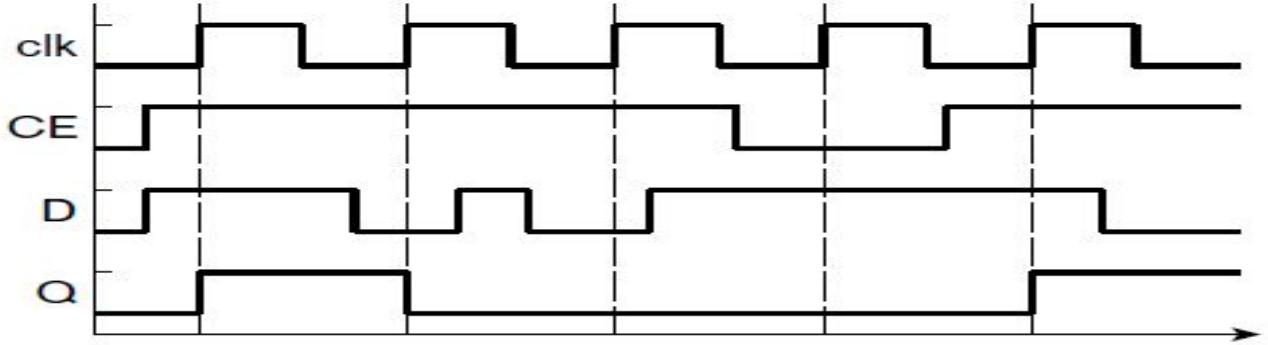
One use for a register constructed from simple D flip-flops is as a *pipeline register* in a sequential design., focusing on the use of pipelining as a technique for improving performance of a digital system. For now, consider the circuit outlined at the top of Figure . Successive values of data arriving at the input are processed by a number of combinational subcircuits, for example, by arithmetic subcircuits built from components described in Chapter 3. The total propagation delay of the circuit is the sum of the propagation delays of the individual subcircuits. This total delay must be less than the interval between arriving data values, otherwise data values may be lost. If the total delay is too long, we can divide the circuit into segments by inserting a register after each subcircuit, as shown

EXAMPLE 4.1 Develop a Verilog model for a pipelined circuit that computes the average of corresponding values in three streams of input values, a, b and c. The pipeline consists of three stages: the first stage sums values of a and b and saves the value of c; the second stage adds on the saved value of c; and the third stage divides by three. The inputs and output are all signed fixed-point numbers indexed from 5 down to −8.

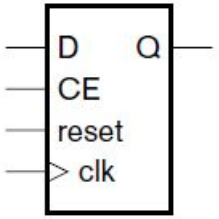
```
module average_pipeline ( output reg signed [5:-8] avg,
                          input      signed [5:-8] a, b, c,
                          input                    clk );

  wire signed [5:-8] a_plus_b, sum, sum_div_3;
  reg  signed [5:-8] saved_a_plus_b, saved_c, saved_sum;

  assign a_plus_b = a + b;

  always @(posedge clk) begin  // Pipeline register 1
    saved_a_plus_b <= a_plus_b;
    saved_c        <= c;
  end

  assign sum = saved_a_plus_b + saved_c;

  always @(posedge clk)  // Pipeline register 2
    saved_sum <= sum;

  assign sum_div_3 = saved_sum * 14'b00000001010101;

  always @(posedge clk)  // Pipeline register 3
    avg <= sum_div_3;

endmodule
```

A D flip-flop with clock-enable input. With Timing Diagram





A D flip-flop with clock-enable and reset inputs.




```
always @(posedge clk)
  if (ce) q <= d;
```

EXAMPLE 4.2 Develop a Verilog model for an accumulator that calculates the sum of a sequence of fixed-point numbers. Each input number is signed with 4 pre-binary-point and 12 post-binary-point bits. The accumulated sum has 8 pre-binary-point and 12 post-binary-point bits. A new number arrives at the input during a clock cycle when the data_en control input is 1. The accumulated sum is cleared to 0 when the reset control input is 1. Both control inputs are synchronous.

SOLUTION The module requires a clock input, two control inputs, a data input and a data output, as follows:

```
module accumulator
  ( output reg signed [7:-12] data_out,
    input      signed [3:-12] data_in,
    input                     data_en, clk, reset );

  wire signed [7:-12] new_sum;

  assign new_sum = data_out + data_in;

  always @(posedge clk)
    if      (reset)   data_out <= 20'b0;
    else if (data_en) data_out <= new_sum;

endmodule
```

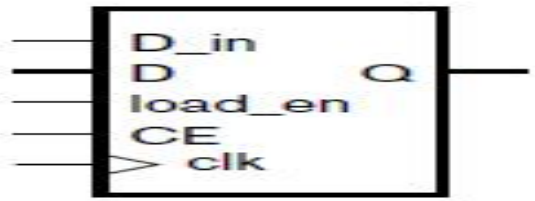
EXAMPLE 4.3 The symbol in Figure 4.10 shows a negative-edge-triggered flip-flop with clock enable, negative-logic asynchronous preset and clear, and both active-high and active-low outputs. It is illegal for both preset and clear to be active together. Develop a Verilog model for this flip-flop.

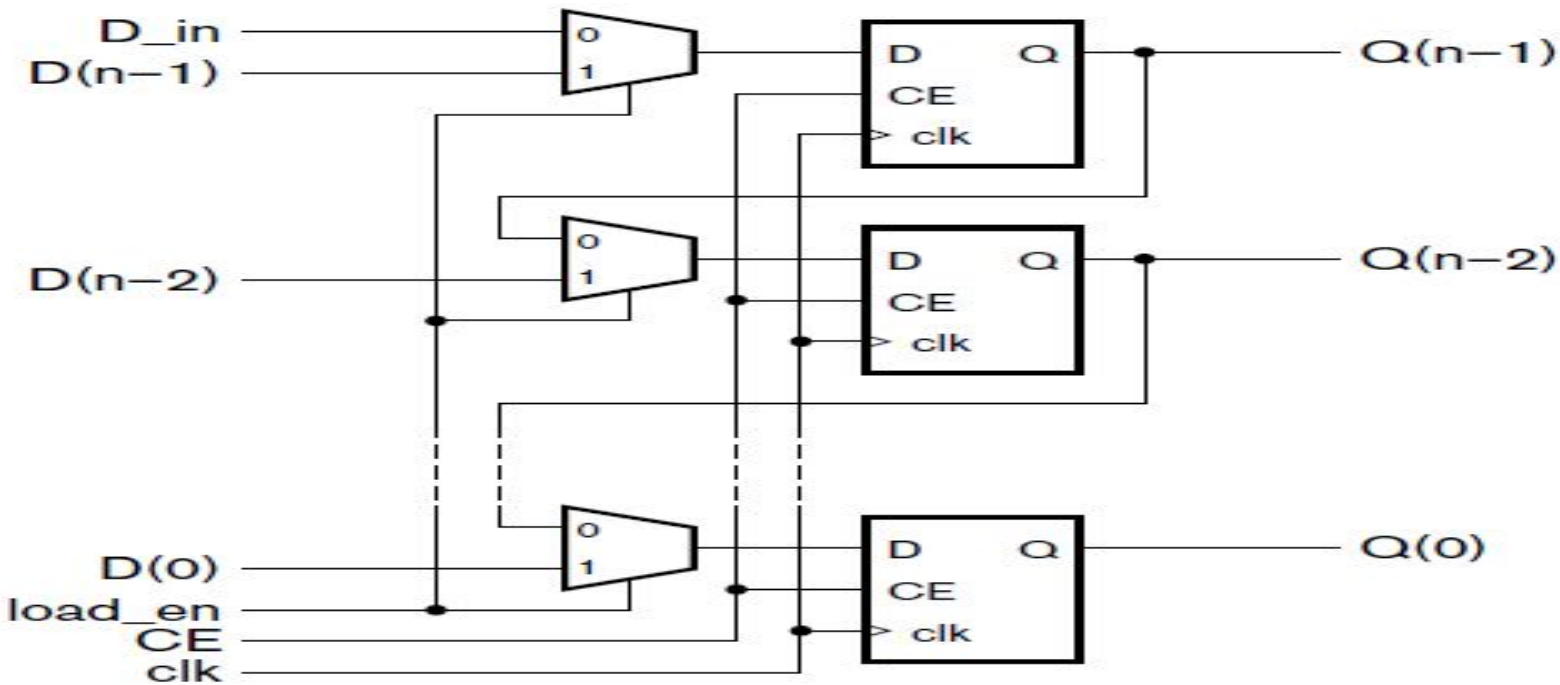
SOLUTION The module definition is

```
module flip_flop_n ( output reg Q,
                     output     Q_n,
                     input      pre_n, clr_n, D,
                     input      clk_n, CE );

  always @( negedge clk_n or
            negedge pre_n or negedge clr_n ) begin
    if (!pre_n && !clr_n)
      $display("Illegal inputs: pre_n and clr_n both 0");
    if      (!pre_n) Q <= 1'b1;
    else if (!clr_n) Q <= 1'b0;
    else if (CE)     Q <= D;
  end

  assign Q_n = ~Q;

endmodule
```

Shift Registers

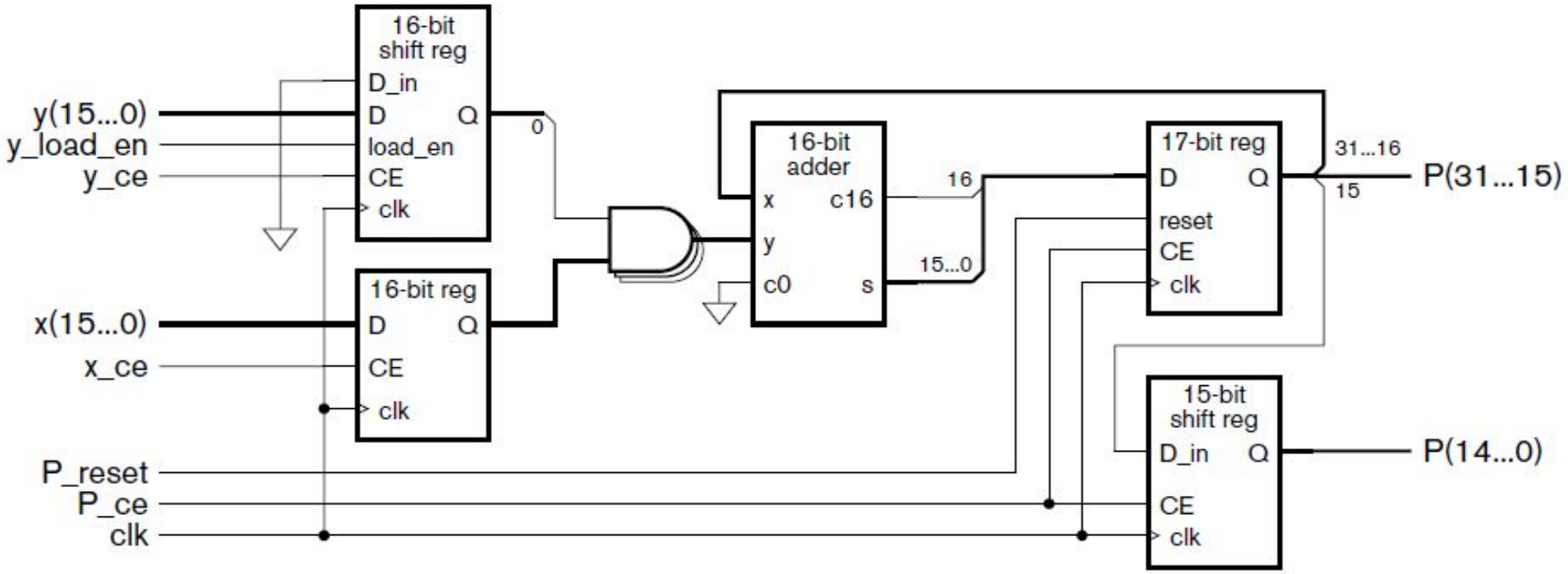
- . A *shift register*, on the other hand, can perform a shift operation on the stored data.
- shift operation has the effect of scaling a numeric value by a power of 2.
- shift operations are also used to implement serial transfer of data, that is, transfer one bit at a time over a single wire, instead of using separate wires for each of the bits of data
- and combine arithmetic scaling with storage functions.


D_in
D
load_en
CE
clk
Q


D_in
D(n−1)
D(n−2)
D(0)
load_en
CE
clk
0
1
D
CE
clk
Q
Q(n−1)
Q(n−2)
Q(0)

- shows how it can be implemented with D flip-flops and multiplexers.
- The shift register is updated on a rising clock edge when CE is 1. In that case, when the load_en signal is 1, the multiplexers select new data on the D(n–1) through D(0) inputs for updating the register.
- Alternatively, when CE is 1 and load_en is 0, the multiplexers select the existing data, shifted right by one place. The least significant bit is discarded, and the most significant bit is updated with the value of the D_in signal.
- If we tie D_in to 0, the shift register performs a logical shift right operation on the stored data. Alternatively, if we connect the most significant output bit back to D_in, the shift register performs an arithmetic shift right operation.

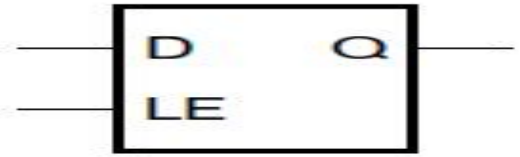
How to perform multiplication of unsigned integers by addition of partial products. Construct a multiplier for two 16-bit operands containing just one adder that adds successive partial products over successive clock cycles. The final product is 32 bits.

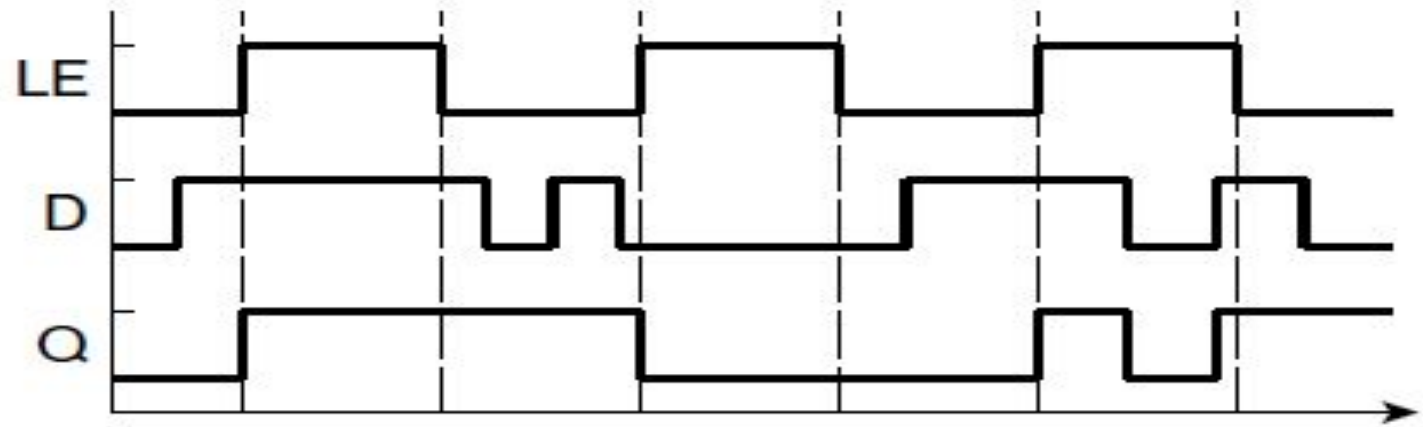


- SOLUTION In order to perform the operation over multiple cycles, we need a number of registers to hold intermediate results, as shown in Figure
- The x operand is stored in an ordinary register whose output connects to an array of 16 AND gates that form a partial product. The y operand is stored in a shift register whose least significant bit, Q(0), controls the AND gates. The y operandis shifted on successive cycles, thus giving the 16 successive partial products. The sum of the partial products are accumulated in a 17-bit ordinary register and a 15-bit shift register. Since the shift register is never required to load data other than through the D_in connection, the data and load_en inputs are absent. On each clock cycle, the least significant bit of the ordinary register is shifted into the shift register, and the remaining bits of the ordinary register are added with the next partial product. By shifting the accumulated sum in this way, partial products are added at successively more significant positions of the result.

L ATC H E S

- As we have seen, a flip-flop is a basic sequential circuit element that stores one bit. Most digital circuits use edge-triggered flip-flops that store a new data value when the clock signal changes from 0 to 1. No further values are stored while the clock remains at 1, nor when the clock returns to 0.
- Some systems, however, use sequential elements called latches, with slightly different timing for storage of values. Figure shows a symbol for a latch, and Figure shows the timing behavior




LE
D
Q

EXAMPLE 4.5 The following always block is intended to model multi- plexer circuitry that selects between a number of inputs to assign to outputs z1 and z2. Identify the error in the block and describe the behavior that results.

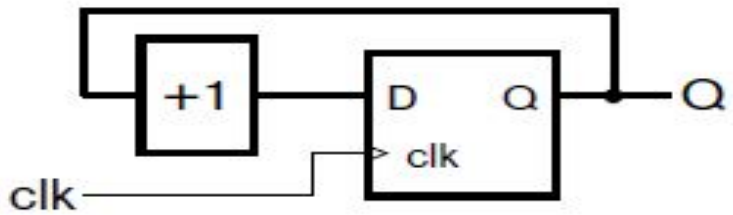
```
always @*
if (~sel) begin
z1 <= a1; z2 <= b1;
End
 else begin
z1 <= a2; z3 <= b2;
end
```

SOLUTION The assignment to z3 in the "else" part of the if statement should assign to z2. As a consequence, z2 is not updated on that execution path and z3 is not updated on the execution path in which sel is 0. Thus, the block implies transparent latches for z2 and z3. The latch for z2 is transparent when sel is 0 and stores a value when sel is 1. The latch for z3 is transparent when sel is 1 and stores a value when sel is 0. This unintended behavior can be corrected simply by changing the target of the assignment from z3 to z2, as it should be.

Counters

A counter is a sequential component that increments or decrements a stored value. Counters occur in many digital circuit applications. For example, if an application requires a given operation to be performed on

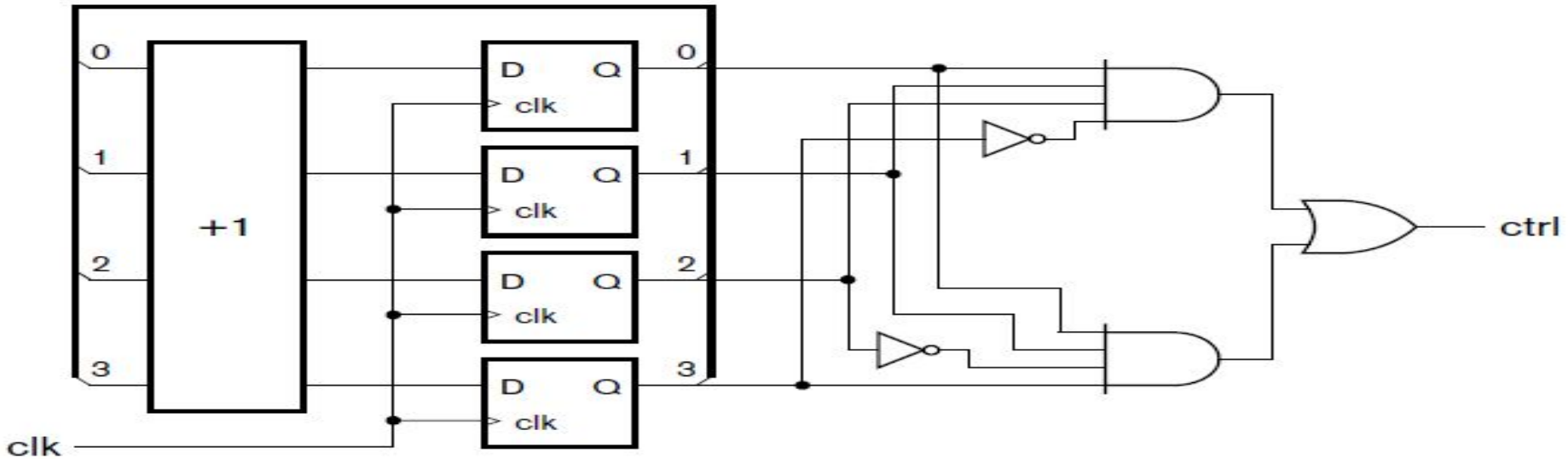
- a number of items of data or to be repeated a number of times, a counter can be used to keep track of how many items have been processed or how many times the operation has been performed. Counters are also used as timers, by counting the number of intervals of a fixed duration that have passed.
- A simple form of counter is composed of an edge-triggered register counter composed of a register and an incrementer and an incrementer, as shown in Figure .



- The value stored in the reg ister is interpreted as an unsigned binary integer. The incrementer can be implemented using the circuit we described for an unsigned incrementer. The counter increments the stored value on every clock edge. When the stored count value reaches its maximum value ($2n - 1$, for an n-bit counter), the incrementer yields a result of all zeros,with the carry out being ignored. This result value is stored on the next clock edge.
- Thus, the counter acts like the odometer in a car, rolling over to zeros after reaching its maximum value. Mathematically speaking, the counter increments modulo $2n$. The counter goes through all $2n$ unsigned binary integer values in order every $2n$ clock cycles. One use for such a counter is in conjunction with a decoder to produce periodic control signals.

Design a circuit that counts 16 clock cycles and produces a control signal, ctrl, that is 1 during every eighth and twelfth cycle.

SOLUTION We need a 4-bit counter, since $16 = 2^4$. The counter counts from 0 to 15 and then wraps back to 0. During the eighth cycle, the counter value is 7 (0111_2), and during the twelfth cycle, the counter value is 11 (1011_2). We can generate the control signal by decoding the two required counter values and forming the logical OR of the decoded signals. The required circuit is shown in Figure 4.21.

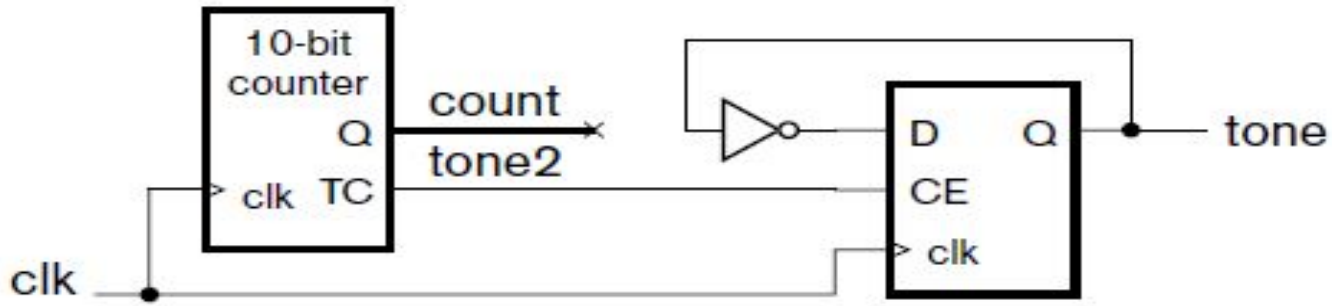


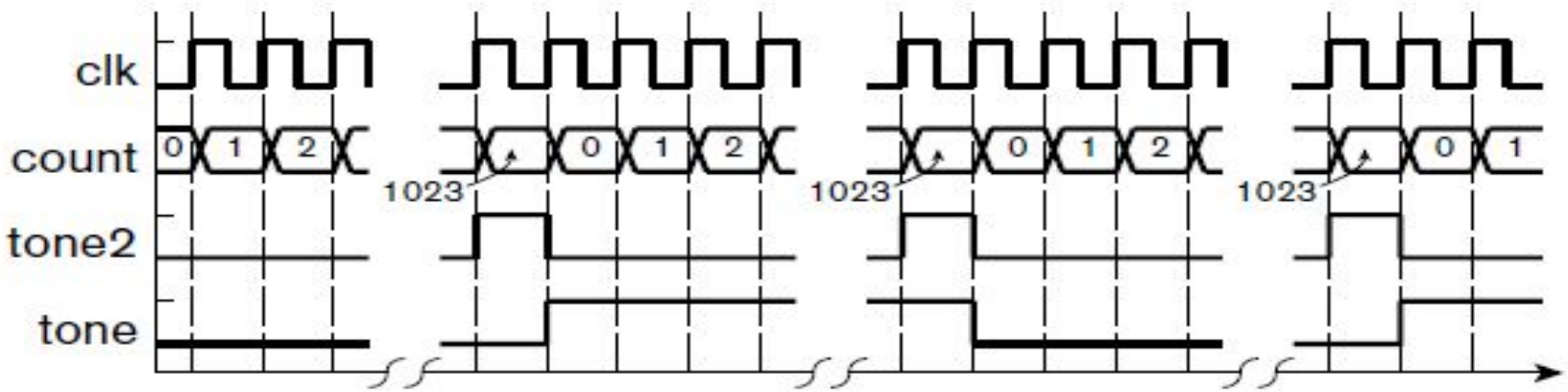
Develop a Verilog model of the circuit for previous example

```
module decoded_counter ( output ctrl,
                         input  clk );

  reg [3:0] count_value;

  always @(posedge clk)
    count_value <= count_value + 1;

  assign ctrl = count_value == 4'b0111 ||
                count_value == 4'b1011;

endmodule
```

A digital alarm clock needs to generate a periodic signal at a frequency of approximately 500Hz to drive the speaker for the alarm tone. Use a counter to divide the system's master clock signal, with a frequency of
1 MHz, to derive the alarm tone.

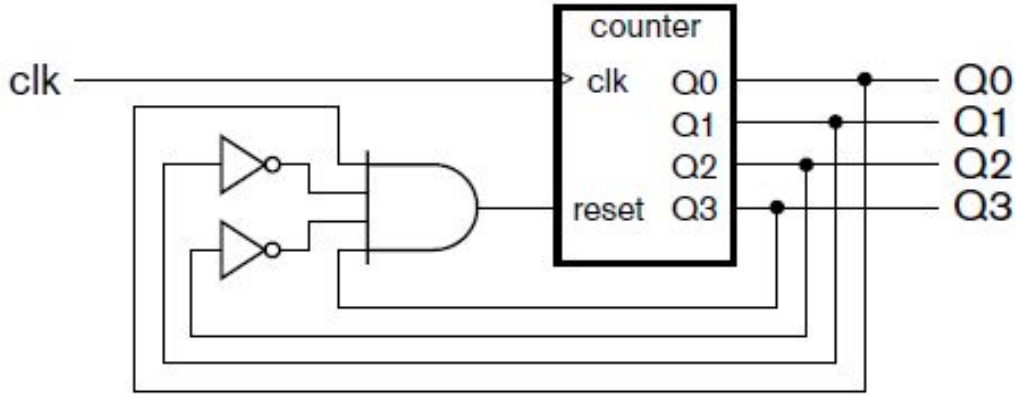
SOLUTION We need to divide the master clock signal by approximately 2000. We can use a divisor of $2^{11} = 2048$, which gives us an alarm tone frequency of 488Hz, which is close enough to 500 Hz. Thus, we could use the terminal-count output of an 11-bit counter for the tone signal. However, the duty cycle (the ratio of time for which the signal is 1 to the time for which it is 0) would only be 1/2048, which would have very low AC energy. We can rectify this by dividing the master clock by 2^{10} with a 10-bit counter, and using the terminal-count output as the count-enable input to a divide-by-2 counter. A circuit is shown in Figure 4.24, and a timing diagram in Figure 4.25. The output of the divide-by-2 counter alternates between 0 and 1 for every pulse on its clock-enable input. The output thus has a 50% duty cycle, which will drive a speaker much more efficiently.


10-bit
counter
Q
clk
TC
count
tone2
clk
D
Q
CE
clk
tone


clk
count
tone2
tone
0
1
2
1023
0
1
2
1023
0
1
2
1023
0
1

Design a circuit for a modulo 10 counter, otherwise known as a *decade counter*.

SOLUTION The maximum count value is 9, so we need 4 bits for the counter. The unsigned binary code word for 9 is 1001_2. We can decode this value and use it to reset to counter to 0 on the next clock cycle. The circuit is shown in Figure 4.26.

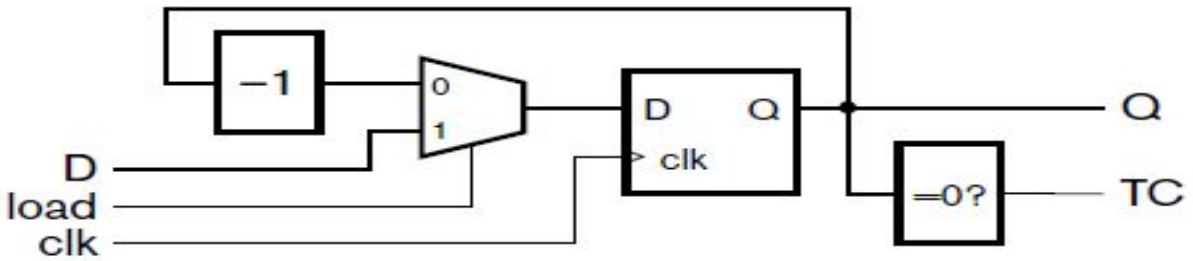


Develop a Verilog model for the decade counter of previous Example

SOLUTION The module definition is

```
module decade_counter ( output reg [3:0] q,
                        input            clk );

  always @(posedge clk)
    q <= q == 9 ? 0 : q + 1;

endmodule
```

- Another form of counter that is useful in timing applications is a *down counter with load*. This counter is loaded with an input value, and then decrements the count value. The terminal count output is activated when the count value reaches zero. A circuit for the counter is shown in Figure



- If the clock input to the counter is a periodic signal with period t and the counter is loaded with a value k, the terminal count is reached after an interval of $k \times t$. Thus, this form of counter can be used as an *interval timer*, where the terminal-count output signal is used to trigger an activity after expiration of a given time interval.

EXAMPLE 4.11 Develop a Verilog model for an interval timer that has clock, load and data input ports and a terminal-count output port. The timer must be able to count intervals of up to 1000 clock cycles.

SOLUTION The data input and counter need to be 10 bits wide, since that is the minimum number of bits needed to represent 1000. The module definition is

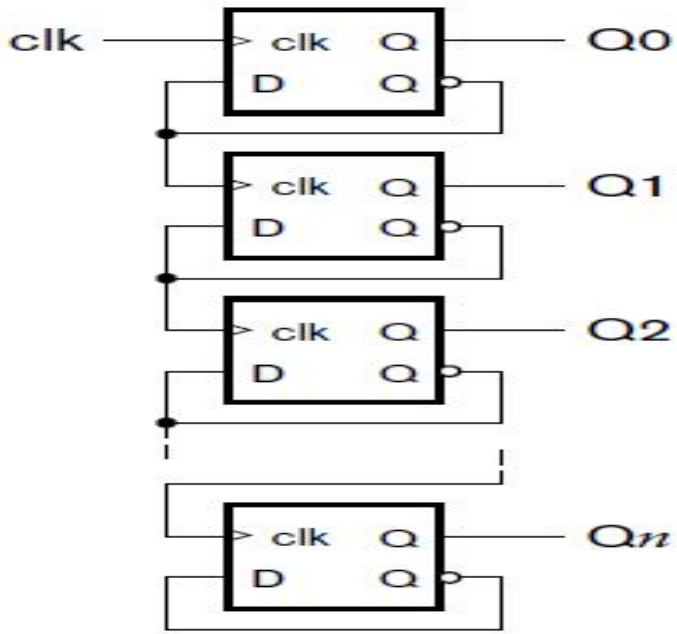
```
module interval_timer_rtl ( output       tc,
                            input [9:0] data,
                            input       load, clk );

  reg [9:0] count_value;

  always @(posedge clk)
    if (load) count_value <= data;
    else      count_value <= count_value - 1;

  assign tc = count_value == 0;

endmodule
```

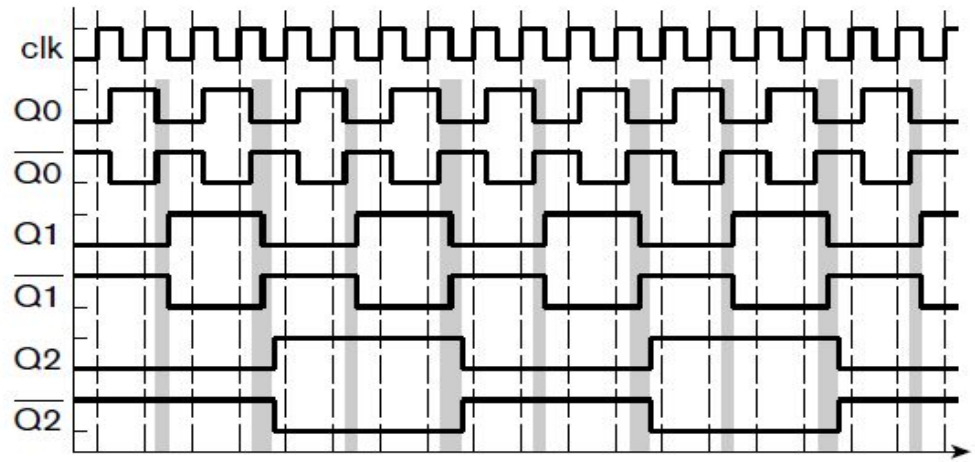
Modify the interval timer so that, when it reaches zero, it reloads the previously loaded value rather than wrapping around to the largest count value.

SOLUTION We need to use a separate register to store the data value to load into the counter. When the load input is activated, a new data value is loaded into the storage register as well as into the counter. When the terminal count is reached, the counter should be loaded from the storage register. The inputs and outputs of the revised interval timer are the same, so we don't need to change the ports of the module definition. The revised module is

```
module interval_timer_repetitive ( output       tc,
                                   input [9:0] data,
                                   input       load, clk );

  reg [9:0] load_value, count_value;

  always @(posedge clk)
    if (load) begin
      load_value <= data;
      count_value <= data;
    end
    else if (count_value == 0)
      count_value <= load_value;
    else
      count_value <= count_value - 1;

  assign tc = count_value == 0;

endmodule
```

Ripple Counter





- The last kind of counter that we will describe in this section is a *ripple counter* (distinct from ripple carry used in an incrementer of a counter), shown in Figure. It is somewhat different in structure from the syn- chronous counters we have previously examined. Like those counters, it has a collection of flip-flops for storing the count value. However, unlike them, the clock signal is not connected in common to all of the flip-flop clock inputs. Rather, the clock input just triggers the flip-flop for the least significant bit, causing it to toggle between 0 and 1 on each rising clock edge. When the Q output changes to 0, the Q output changes to 1, triggering the next flip-flop to toggle between 0 and 1. This flip-flop behaves similarly….

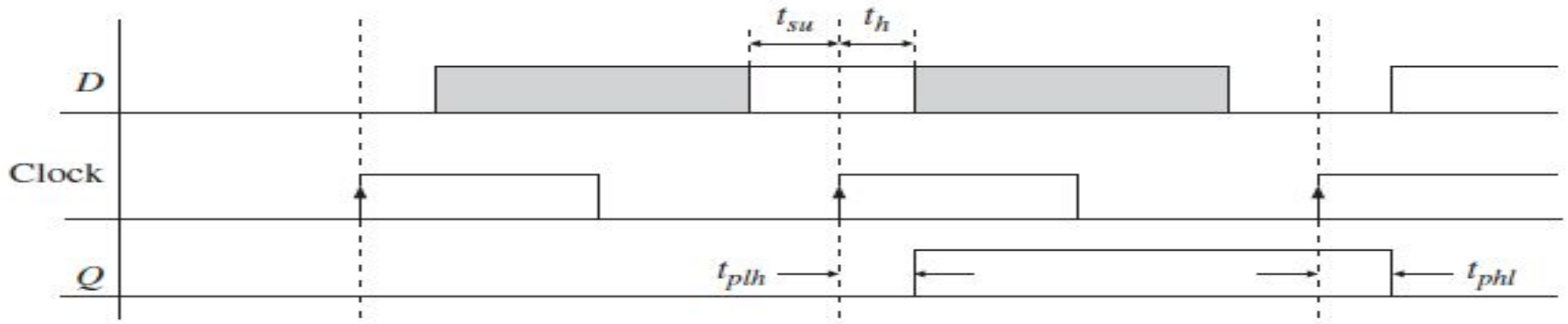
Sequential Circuit Timing

- The correct functioning of sequential circuits involves several timing issues. Propagation delays of flip‑flops,
- gates and wires;
- setup times and
- hold times of flip‑flops;
- clock synchronization;
- clock skew;

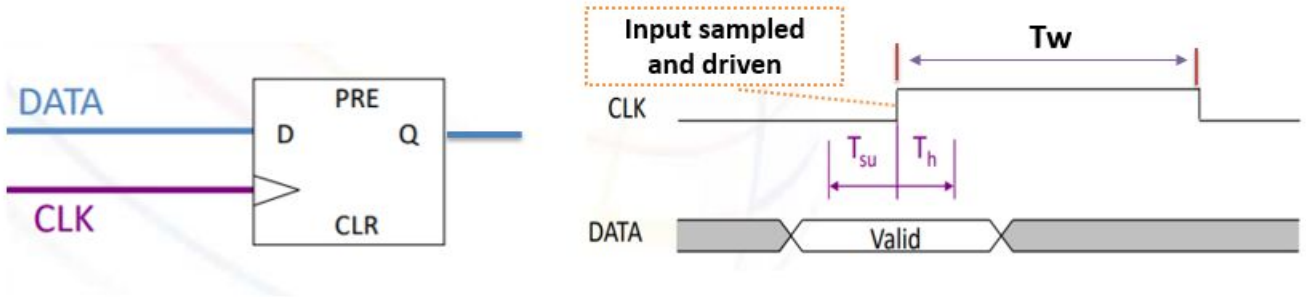
Propagation Delays, Setup, and Hold Times

- There is a certain amount of time, albeit small, that elapses from the time the clock changes to the time the *Q* output changes. This time, called *propagation delay* or ***clock-to-Q delay*** of the flip-flop is indicated in Figure 1-34. The propagation delay can depend on whether the output is changing from high to low or vice versa. In the figure, the propagation delay for a low-to-high change in *Q* is denoted by t_{plh}, and for a high-to-low change it is denoted by t_{phl}.
- For an ideal *D* flip-flop, if the *D* input changed at exactly the same time as the active edge of the clock, the flip-flop would operate correctly. However, for a real flip- flop, the *D* input must be stable for a certain amount of time before the active edge of the clock. This interval is called the *setup time* (t_{su}). Furthermore, *D* must be stable for a certain amount of time after the active edge of the clock. This interval is called the *hold time* (t_h). Figure 1-34 illustrates setup and hold times for a *D* flip-flop that changes state on the rising edge of the clock. *D* can change at any time during the shaded region on the diagram, but it must be stable during the time interval t_{su} before the active edge and for t_h after the active edge. If *D* changes at any time during the forbidden interval, it cannot be determined whether the flip-flop will change state. Even worse, the flip- flop may malfunction and output a short pulse or even go into oscillation.

- Flip-flops typically have a setup time about 3–10x of the propagation delay of an inverter (NOT) gate. The hold times are typically 1–2x of the delay of an inverter. Minimum values for *tsu* and *th* and maximum values for *tplh* and *tphl* can be obtained from manufacturers' data sheets or ASIC (Application Specific Integrated Circuit) libraries accompanying design tools.
-

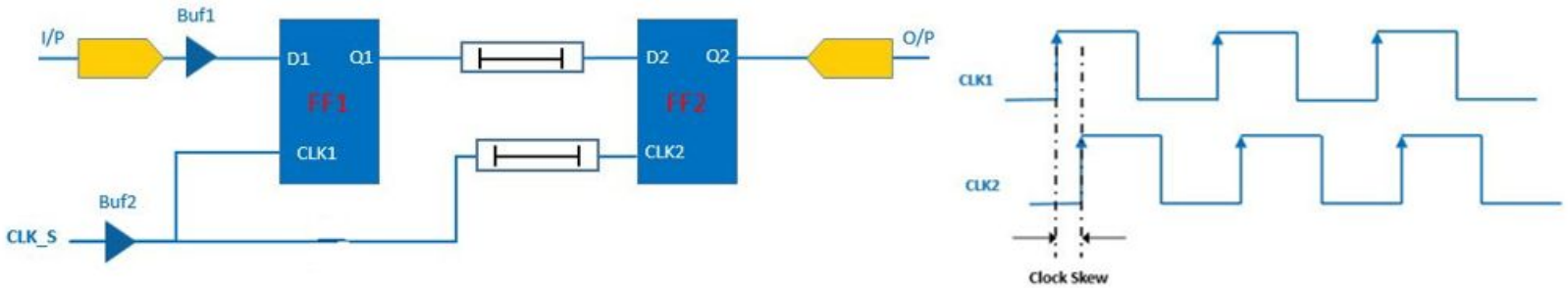


Set up and Hold Time



- In digital circuits, always data is either sampled or driven.
- Sampling is done with respect to reference clock and driving as well.
- **Pulse width (Tw)**
  - It is the time between the active and inactive states of the same signal
- **Setup time (T_{su})**
  - For an edge triggered sequential element, the setup time is the time interval before the active clock edge during which the data should remain unchanged
- **Hold time (T_h)**
  - Time interval after the active clock edge during which the data should remain unchanged

Clock skew



- Clock Skew is the difference in the arrival time of the clock at clock pin of different flip flops
- Correct behavior assumes that all storage elements sample at exactly the same time
- Not possible in real systems:
  - clock driven from some central location
  - different wire delay to different points in the circuit
- Problems arise if skew is of the same order as FF contamination delay
- Gets worse as systems get faster (wires don't improve as fast)
  - Distribute clock signals in general direction of data flow
  - Wire carrying the clock between two communicating components should be as short as possible
  - Try to make all wires from the clock generator be the same length - clock tree

Timing Conditions for Proper Operation

- In a synchronous sequential circuit, state changes occur immediately following the active edge of the clock.
- The maximum clock frequency for a sequential cir- cuit depends on several factors.
- The clock period must be long enough so that all flip-flop and register inputs will have time to stabilize before the next active edge of the clock.
- Propagation delays and setup and hold times create complications in sequential circuit timing.

- **Static timing analysis** (STA) is a method of validating the timing performance of a design by checking all possible paths for timing violations under worst-case conditions. A static analysis path starts at a source flip-flop (or at a primary input) and terminates at a destination flip-flop (or primary output). A static timing path between two flip-flops starts at the input to the source flip-flop and terminates at the input of the destination flip-flop. It does not go through the destination flip- flop. The path terminates when it encounters a clocked device. If a signal goes from register (flip-flop) A to register B and then to register C, the signal contains two paths. The timing paths in a synchronous digital system can be classified into 4 types:
    - Register to register paths (i.e., flip-flop to flip-flop)
    - Primary input to register paths (i.e., input to flip-flop)
    - Register to primary output paths (i.e., flip-flop to output)
    - Input to output paths (i.e., no flip-flop)