DIGITAL SYSTEM DESIGN USING VERILOG

COURSE CODE: 19EC5DCDSV (3 CREDITS)

MODULE 3A

Faculty In-Charge: Dr. Dinesha P

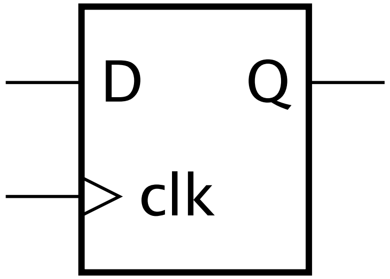
drdinesh-ece@dayanandasagar.edu

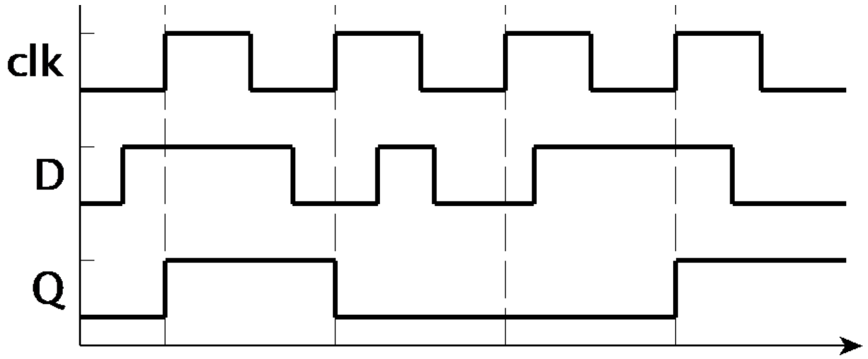
Sequential Basics

- Sequential circuits
  - Outputs depend on current inputs and previous inputs
  - Store *state*: an abstraction of the history of inputs
- Usually governed by a periodic clock signal

D-Flip flops

- 1-bit storage element
  - We will treat it as a basic component
  - 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





- Other kinds of flipflops
  - SR (set/reset), JK, T (toggle)

Registers:

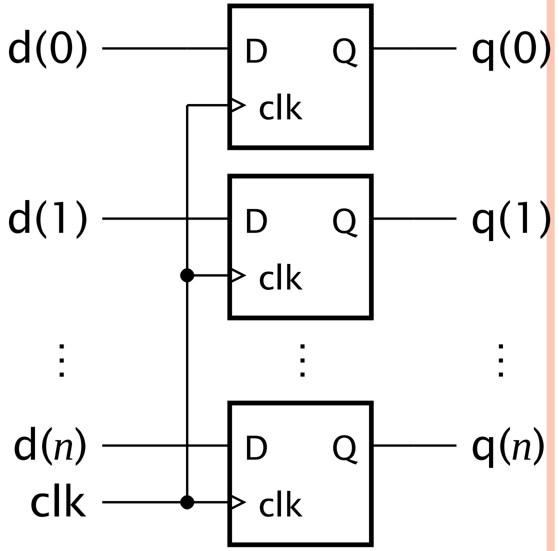
- A group of flip-flops used in this way is called a *register*. *Each flip-flop in the register stores one bit of the* code word of the stored value,

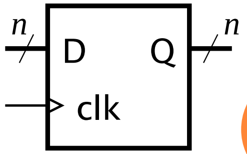
- Store a multi-bit encoded value
  - One D-flipflop per bit
  - Stores a new value on each clock cycle

```
wire [n:0] d;
reg  [n:0] q;
...

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

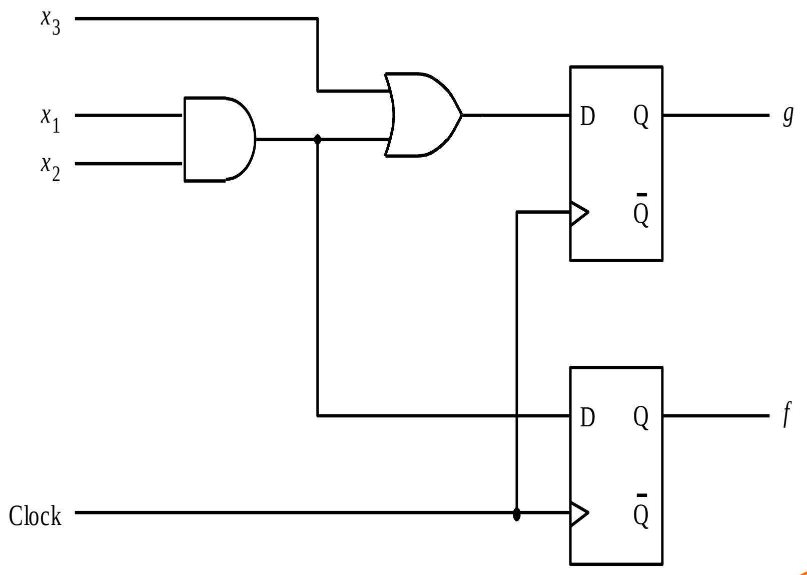
event list

nonblocking asignment





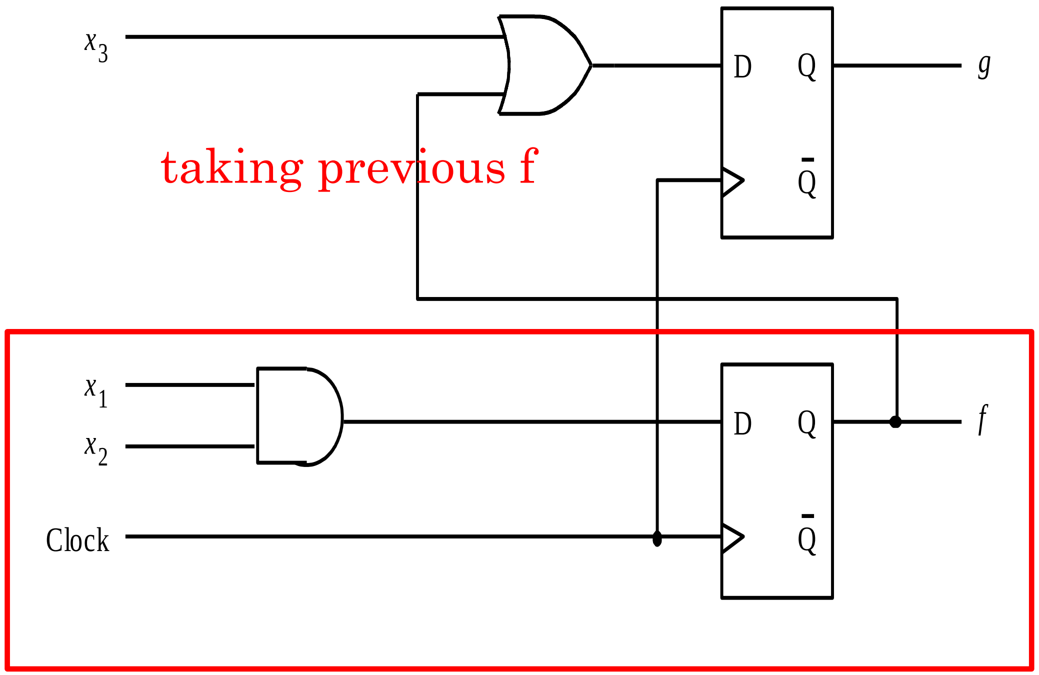
A Simple Circuit Using Blocking Assignment



```
module simple (x1, x2, x3,
Clock, f, g);
        input x1, x2, x3, Clock;
        output reg f, g;

        always @(posedge Clock)
        begin
                f = x1 & x2;
                g = f | x3;
        end

endmodule
```

A rising edge, latch f = x1&X2, and
latch g = (x1&x2) | x3.

Non-Blocking Assignment



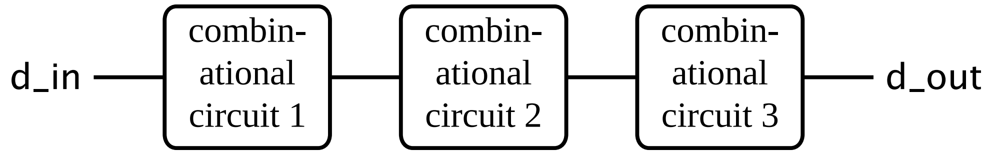
```
module simple (x1, x2, x3,
Clock, f, g);
        input x1, x2, x3, Clock;
        output reg f, g;

        always @(posedge
Clock)
        begin
                f <= x1 & x2;
                g <= f | x3;
        end
                        // reversing f and g
endmodule               statement makes n
                        difference in result.
```

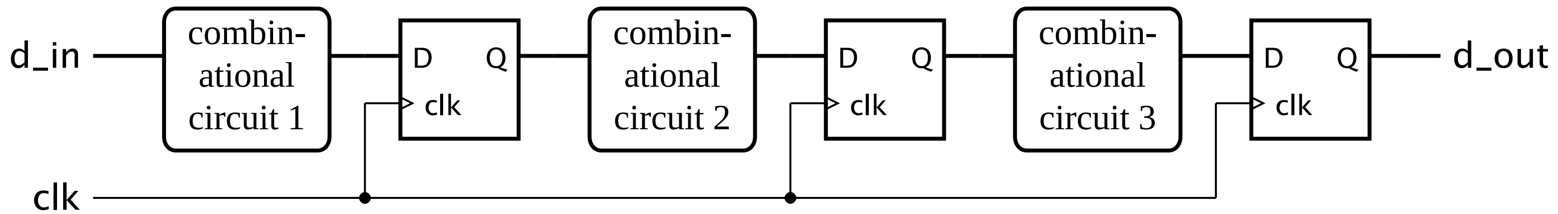
// Non-blocking assignment, f and g are updated at the same time.
Meaning that, g must take a previous f.

Pipelines Using Registers

Total delay = $\text{Delay}_1 + \text{Delay}_2 + \text{Delay}_3$

Interval between outputs > Total delay





Clock period = $\max(\text{Delay}_1, \text{Delay}_2, \text{Delay}_3)$

Total delay = 3 × clock period

Interval between outputs = 1 clock period

Pipeline Example

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.

- Compute the average of corresponding numbers in three input streams
  - New values arrive on each clock edge

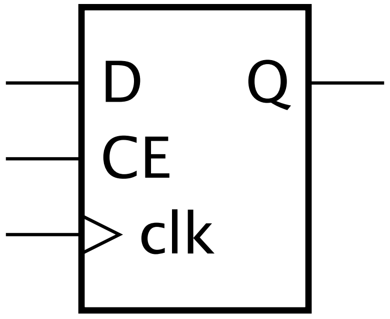
```
module average_pipeline ( output reg signed [5:-8] avg,
                          input      signed [5:-8] a, b, c,
                          input                    clk );


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

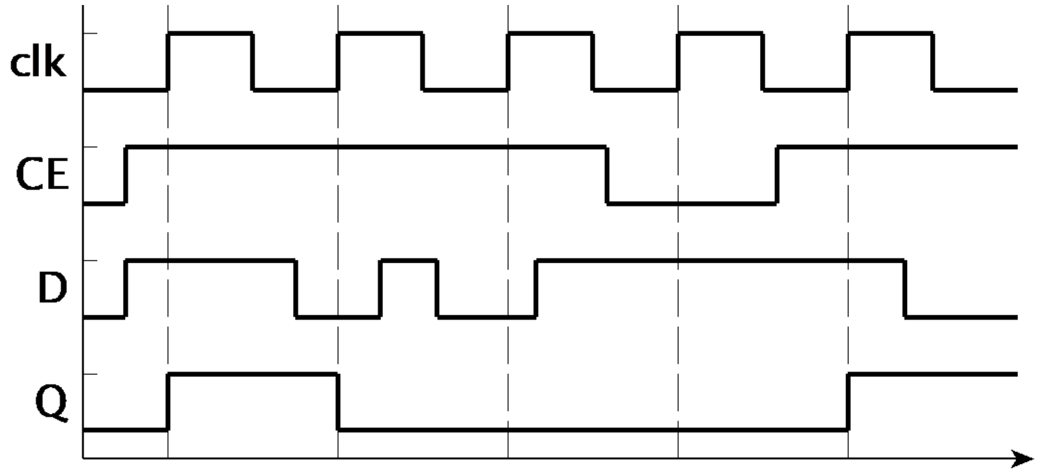
Pipeline Example

```
  ...
  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
```

D-Flipflop with Enable

- Storage controlled by a clock-enable
  - stores only when CE = 1 on a rising edge of the clock





- CE is a *synchronous control* input

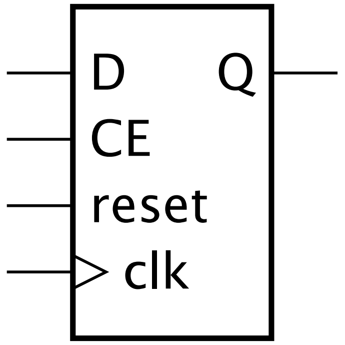
Register with Enable

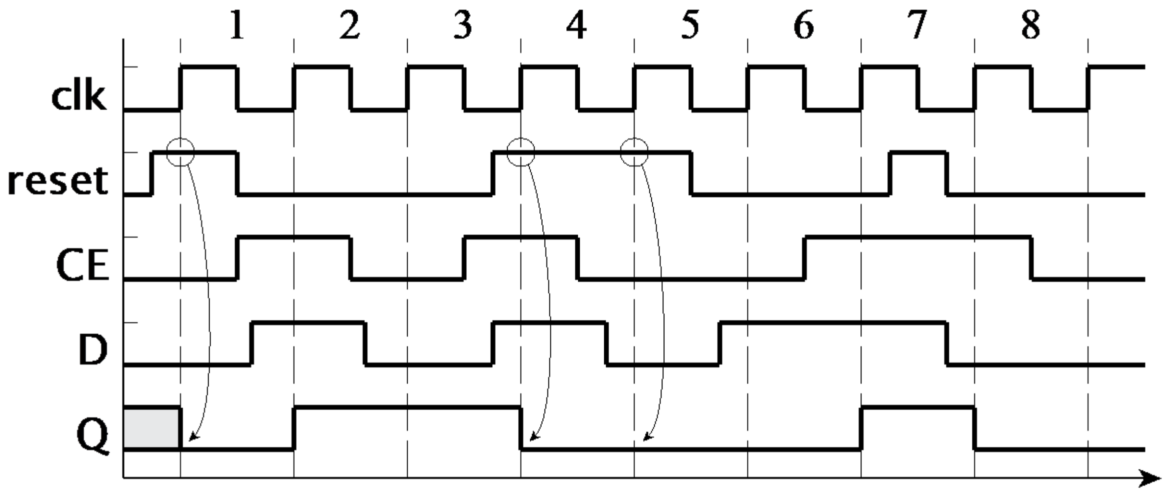
- One flipflop per bit
  - clk and CE wired in common

```
wire [n:0] d;
wire       ce;
reg  [n:0] q;
...

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

Register with Synchronous Reset

- Reset input forces stored value to 0
  - reset input must be stable around rising edge of clk

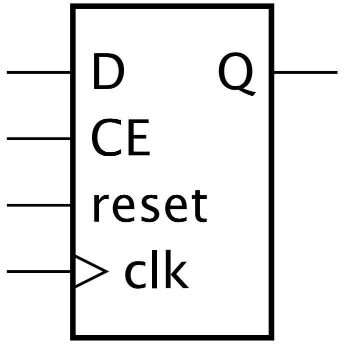


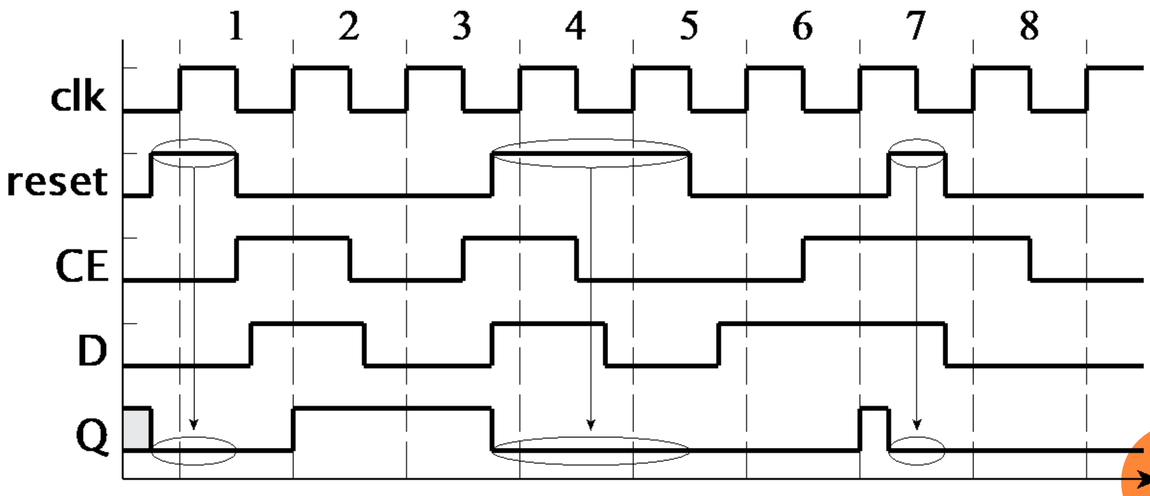


```
always @(posedge clk)
  if      (reset) q <= 0;
  else if (ce)    q <= d;
```

Register with Asynchronous Reset

- Reset input forces stored value to 0
  - reset can become 1 at any time, and effect is immediate
  - reset should return to 0 synchronously





Asynch Reset in Verilog

```
always @(posedge clk or posedge reset)
  if      (reset) q <= 0;
  else if (ce)    q <= d;
```

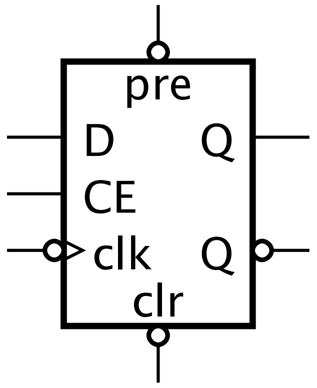
- reset is an *asynchronous control* input here
  - include it in the event list so that the process responds to changes immediately

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.

- Sum a sequence of signed numbers
  - A new number arrives when data_en = 1
  - Clear sum to 0 on synch reset

```
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
```

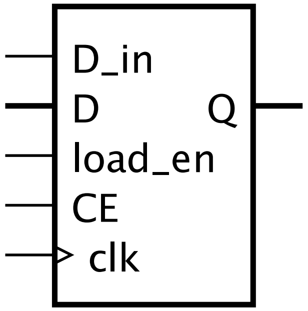
The symbol in Figure shows a negative-edge-triggered flip-flop with clock enable, negative-logic asynchronous preset and clear, and both active-high and active-low outputs. It is illegal for both preset and clear to be active together. Develop a Verilog model for this flip-flop.

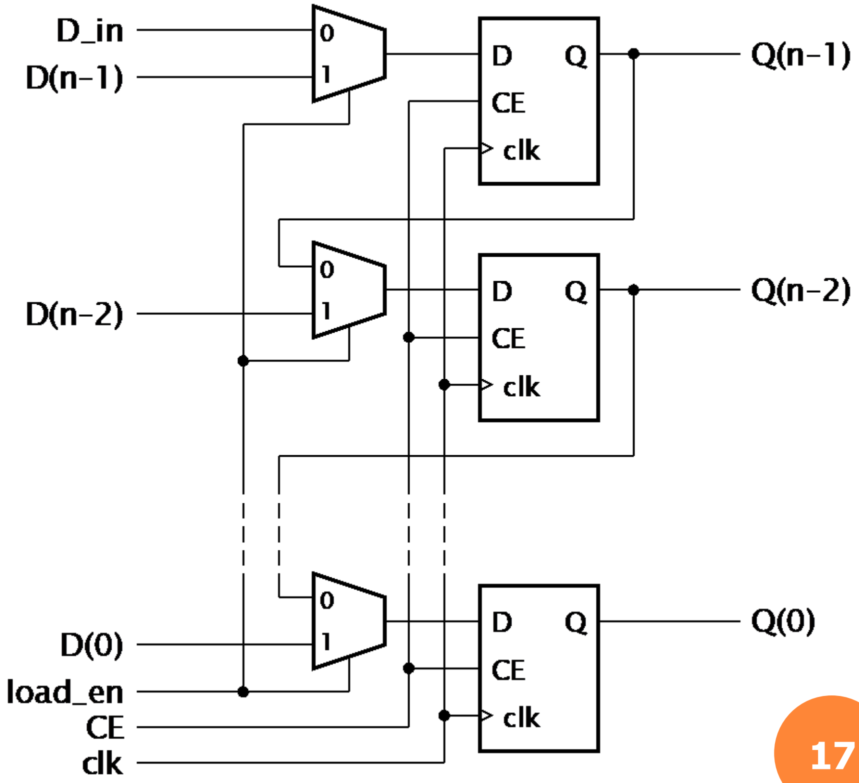


```
module flip_flop_n ( output reg Q,
                     output     Q_n,
                     input      pre_n, clr_n, D,
                     input      clk_n, CE );

  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

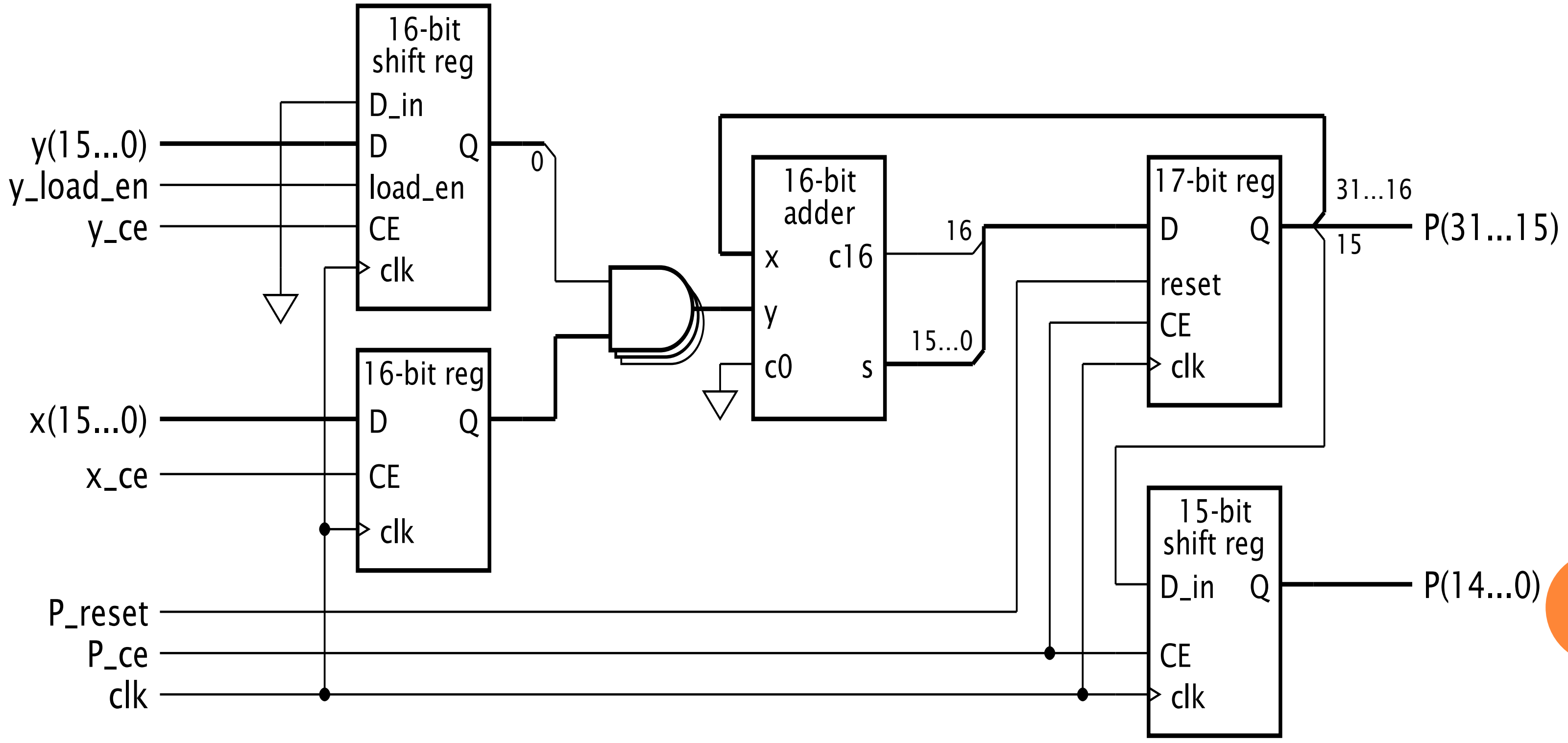
- Performs shift operation on stored data
  - Arithmetic scaling
  - Serial transfer of data





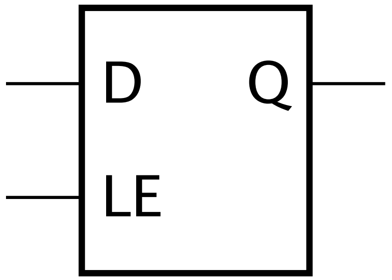
we showed how to perform multiplication of unsigned integers by addition of partial products. Construct a multiplier for two 16-bit operands containing just one adder that adds successive partial products over successive clock cycles. The final product is 32 bits.

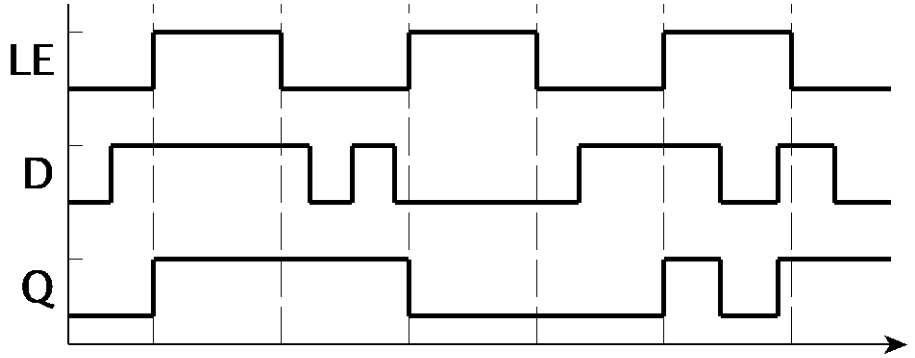
- 16×16 multiply over 16 clock cycles, using one adder
  - Shift register for multiplier bits
  - Shift register for LCB's of accumulated product

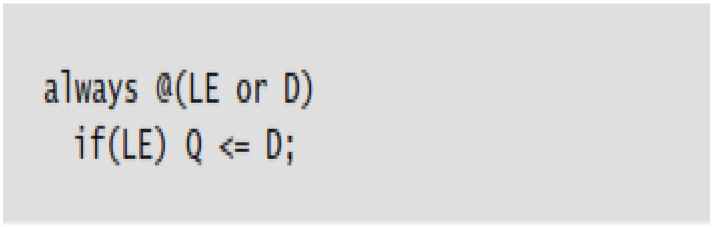


Latches

- Level-sensitive storage
  - Data transmitted while enable is '1'
    - *transparent* latch
  - Data stored while enable is '0'





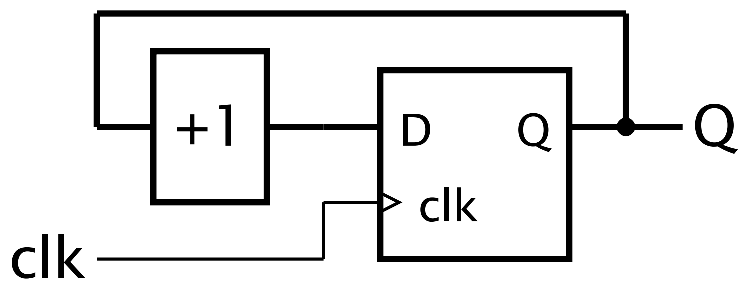


```
always @(LE or D)
  if(LE) Q <= D;
```

Counters

- Stores an unsigned integer value
  - increments or decrements the value
- Used to count occurrences of
  - events
  - repetitions of a processing step
- Used as timers
  - count elapsed time intervals by incrementing periodically

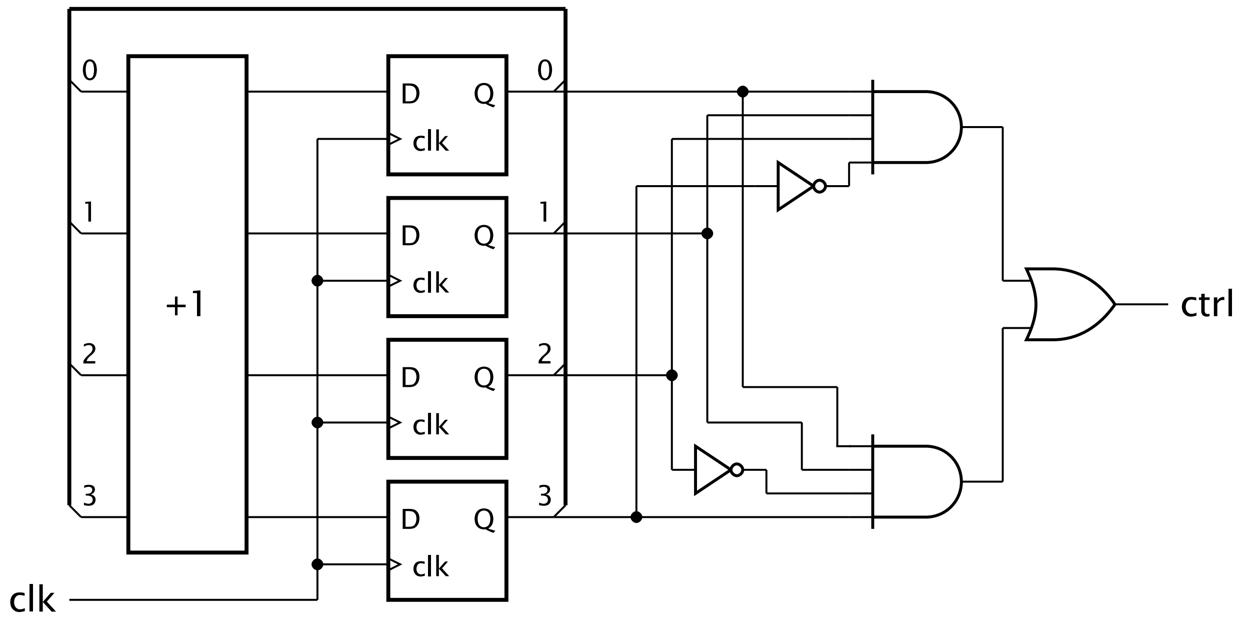
Free-Running Counter



- Increments every rising edge of clk
  - up to 2^n–1, then wraps back to 0
  - i.e., counts modulo 2^n
- This counter is *synchronous*
  - all outputs governed by clock edge

EXAMPLE 4.6 Design a circuit that counts 16 clock cycles and produces a control signal, ctrl, that is 1 during every eighth and twelfth cycle.

- Count modulo 16 clock cycles
  - Control output = 1 every 8th and 12th cycle
  - decode count values 0111 and 1011

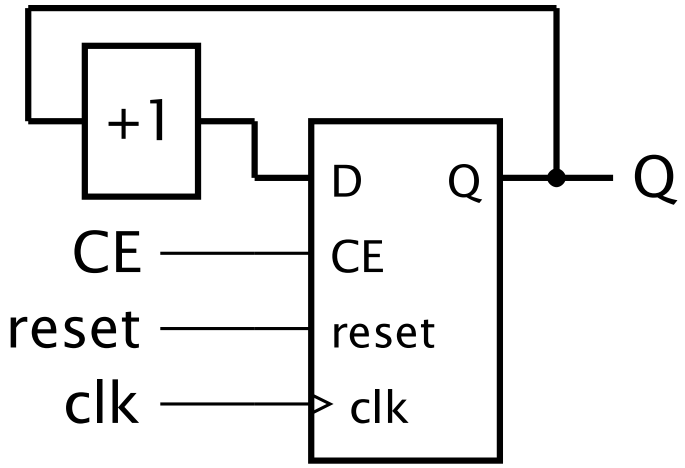


DEVELOP A VERILOG MODEL OF THE CIRCUIT SHOWN ABOVE

```
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
```

Count Enable and Reset

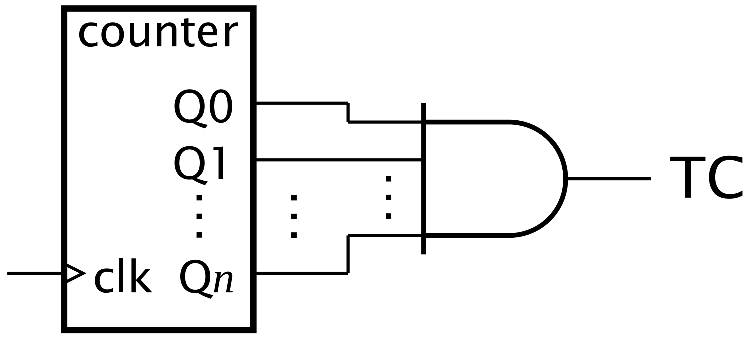
- Use a register with control inputs



  - Increments when CE = 1 on rising clock edge
  - Reset: synch or asynch

Terminal Count

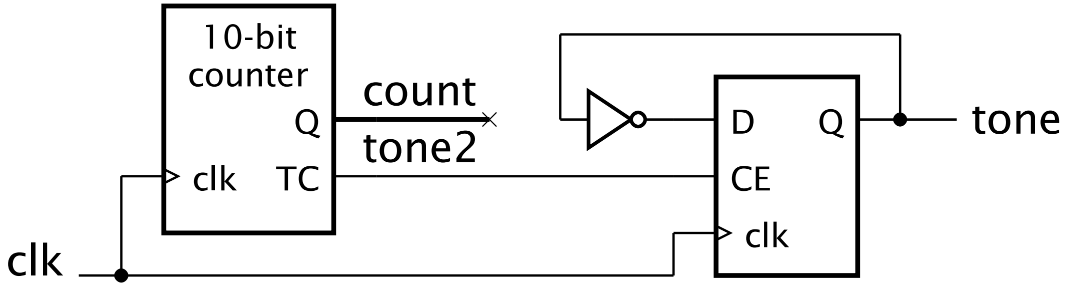
- Status signal indicating final count value

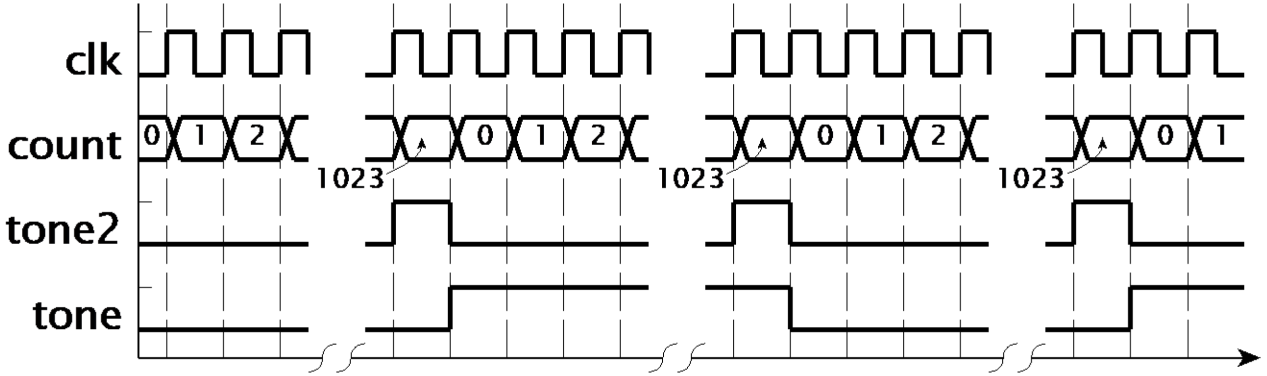


- TC is 1 for one cycle in every 2^n cycles
  - frequency = clock frequency / 2^n
- Called a *clock divider*

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.

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.

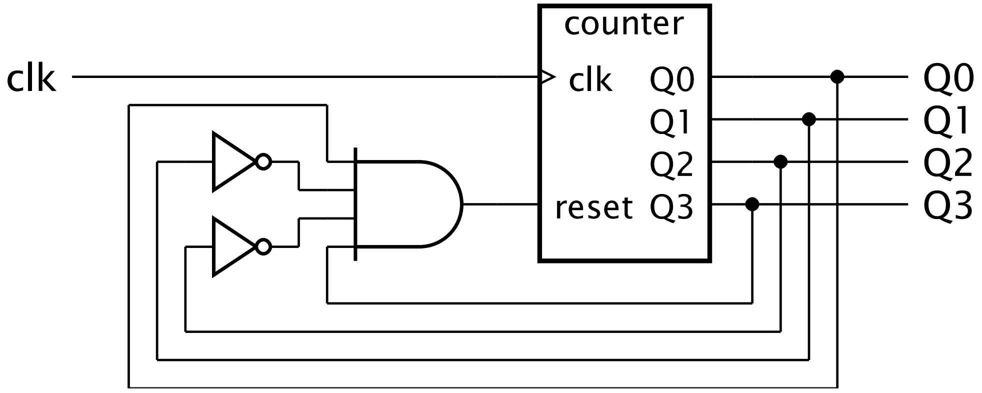




Divide by *k*:
Design a circuit for a modulo 10 counter, otherwise known as a *decade counter*.

- Decode *k*–1 as terminal count and reset counter register
  - Counter increments modulo *k*

    Example: decade counter
  - Terminal count = 9



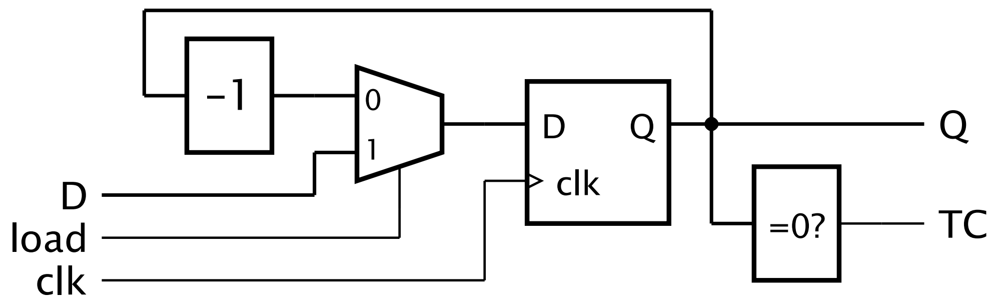
Decade Counter in Verilog

Develop a Verilog model for the decade counter shown in above example Example

```
module decade_counter ( output reg [3:0] q,
                        input            clk );
  always @(posedge clk)
    q <= q == 9 ? 0 : q + 1;
endmodule
```

Down Counter with Load

- Load a starting value, then decrement
  - Terminal count = 0
- Useful for interval timer



LOADABLE COUNTER IN VERILOG

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.

```
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
```

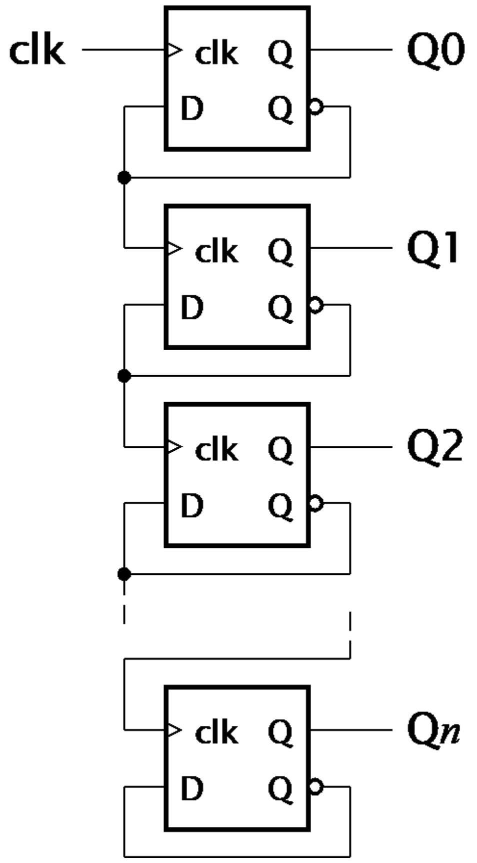
Reloading Counter in Verilog

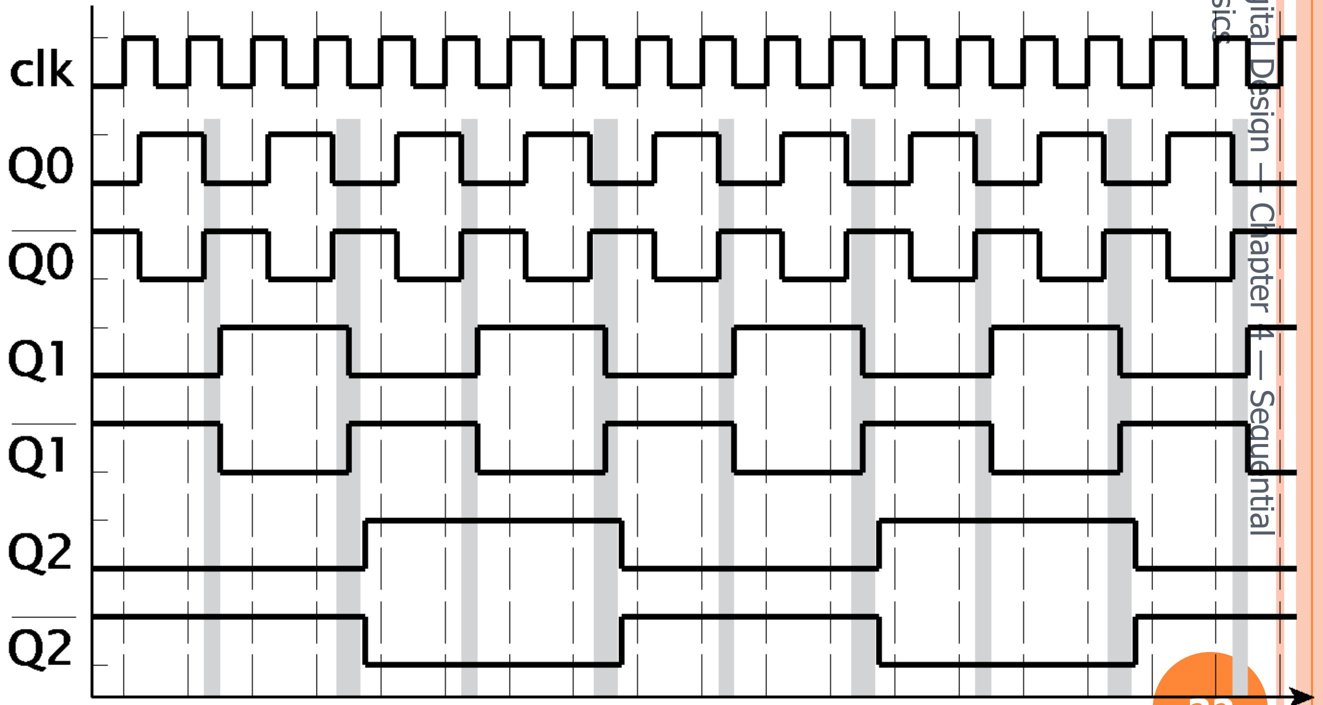
Modify the interval timer so that, when it reaches zero, it reloads the previously loaded value rather than wrapping around to the largest count value.

```
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

- Each bit toggles between 0 and 1
  - when previous bit changes from 1 to 0





Ripple or Synch Counter?

- Ripple counter is ok if
  - length is short
  - clock period long relative to flipflop delay
  - transient wrong values can be tolerated
  - area must be minimal
- E.g., alarm clock
- Otherwise use a synchronous counter

TEST BENCH:

A testbench is simply a Verilog module.

A design under test, abbreviated as DUT, is a synthesizable module of the functionality we want to test. In other words, it is the circuit design that we would like to test. We can describe our DUT using one of the three modeling styles in Verilog – Gate-level, Dataflow, or Behavioral.

```
module and_gate(c,a,b);

input a,b;
output c;

assign c = a & b;

endmodule
```

```
module and_tb;
reg A,B;
wire C;
and_gate dut(.a(A), .b(B), .c(C));
initial
begin
#5 A =0; B=0;
#5 A =0; B=1;
#5 A =1; B=0;
#5 A =1; B=1;
end
initial
begin
$monitor("simtime = %g, A =%b, B =%b, C =%b", $time,A,B,C);
end
endmodule
```

Steps:

- Start with the module declaration
- Declare reg and wire

The reg data type will hold the value until a new value is assigned to it. This is used to apply a stimulus to the inputs of DUT

The wire data type is similar to that of a physical connection. It will hold the value that is driven by a port, assign statement, or reg. This is used to check the output signals *from* the DUT.

- DUT Instantiation

<dut_module><instance_name>(.<dut_signal>(test_module_signal),…….)

Instance_name is user defined name

Initial and Always blocks

There are two sequential blocks in Verilog, initial and always. It is in these blocks that we apply the stimulus.

The following is the stimulus for and_gate in the initial block

```
initial
begin
A = 0; B = 0; // starts execution at t=0
#10 A = 0; B = 1; // execution at t = 10 time units
#10 A = 1; B = 0; // execution at t = 20 time units
#10 A = 1; B = 1; //execution at t = 30 time units
end
```

an always block repeatedly executes, although the execution starts at time t=0.

The following is the stimulus for and_gate in the always block

```
module always_block_example;
reg clk;

initial
begin
clk = 0;
end

always
 #10 clk = ~clk;
endmodule
```

- **Simulation**
- During simulation, the designer should know the status of the current simulation. Hence, a printout of the simulation result is essential, which will inform the designer. The value of all the signals should be displayed as it helps for debugging purposes. Therefore, while writing testbench, we can use two system tasks to print the simulation results.

$display

This is an important system task available in Verilog. It is used for displaying values of variables or strings or expressions. This inserts a newline by default at the end of the string. Let's see how we can use a $display to print signals in a test bench:

```
$display( "time = %g, A = %b, B = %b, C =%b", $time, A,B,C);
```

- The characters mentioned in the quotes will be printed as it is. The letter along with % denotes the string format. We use %b to represent binary data. We can use %d, %h, %o for representing decimal, hexadecimal, and octal, respectively. The %g is used for expressing real numbers.

- As the name states, it is clear that it will monitor the data or variable for which it is written, and whenever the variable changes, it will print the changed value.

- https://technobyte.org/testbench-in-verilog/