Loops :

* for loop
* while loop
* Repeat

- using parameters (make your code reusable for different sizes of the inputs)
- assigning a name for a begin – end block
- disable
- forever (runs always)

```
module Majority_4b ( output reg Y,
                     input A, B, C, D);

always@( A, B, C, D)
  begin
    case ( {A, B, C, D})
      7, 11, 13, 14, 15 : y = 1;
                        : y = 0;
      default
    endcase
  end

endmodule
```

```
module Majority #( parameter size = 8 ,
                   parameter MSB = 3 ,
                   parameter majority = 5 )
                 ( input [(size-1) : 0] Data,
                   output reg y );

reg [ MSB : 0 ] count ;
integer k ;

always @ (Data) begin
    count = 0 ;
    for ( k=0 ; k < size ; k = k+1) begin
        if ( Data [k] == 1)
            count = count + 1 ;
    end
    y = ( count >= majority ) ;
end

endmodule
```

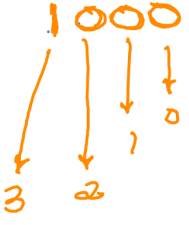
8 ≡ 1000

3 bits

4 ≡ 100

2 bits

eight ones

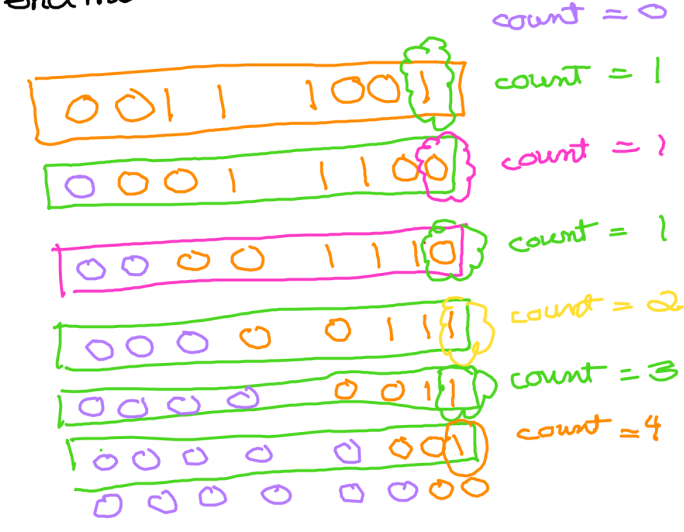


k will take values from 0 to 7

controlling the index of the bit in the word.

```
module Count_ones #( parameter word_size = 8 ,
                     parameter count_size = 4 )
( input  [(word_size-1) : 0]  word_in ,
  output [(count_size-1) : 0] count_out ) ;

reg [(word_size-1) : 0] temp_reg ;
reg [(counter_size-1) : 0] count ;

always @ (word_in)
  begin : ones_count
    count = 0 ;
    temp_reg = word_in ;
    while ( temp_reg )
      begin
        count = count + temp_reg [0] ;
        temp_reg = temp_reg >> 1 ;
      end
  end

assign count_out = count ;

end module
```

if (temp_reg[0])
count = count + 1 ;



Different approach for the count_is module using "repeat".

```
        i = 0 ;
temp_reg = word_in ;
count    = 0 ;

repeat ( word_size )
  [ begin
        count = count + temp_reg [ i ]
            i = i+1 ;
    end
```

```
parameter   half_cycle = 50 ;
parameter   stop_time  = 350 ;

initial
  begin : clock_loop
    clock = 0 ;
    forever
      # half_cycle  clock = ~ clock ;
  end

initial
  # stop_time  disable clock_loop ;
```

forever

1. computational activity within sequential flow
2. can be nested
3. Becomes active following a specific sequence

always (& initial)

concurrent behavior

cannot be nested

Becomes active when the simulation starts

```
module find_first_one ( output reg [3:0] index_value ,
                        input [15:0] A_word ,
                        input trigger ) ;

always @ ( posedge trigger )
  begin : search_for_1
    for ( index_value = 0 ; index_value < 15 ; index_value = index_value + 1 )
      if ( A_word [index_value] == 1 )
        disable search_for_1 ;
  end
endmodule
```