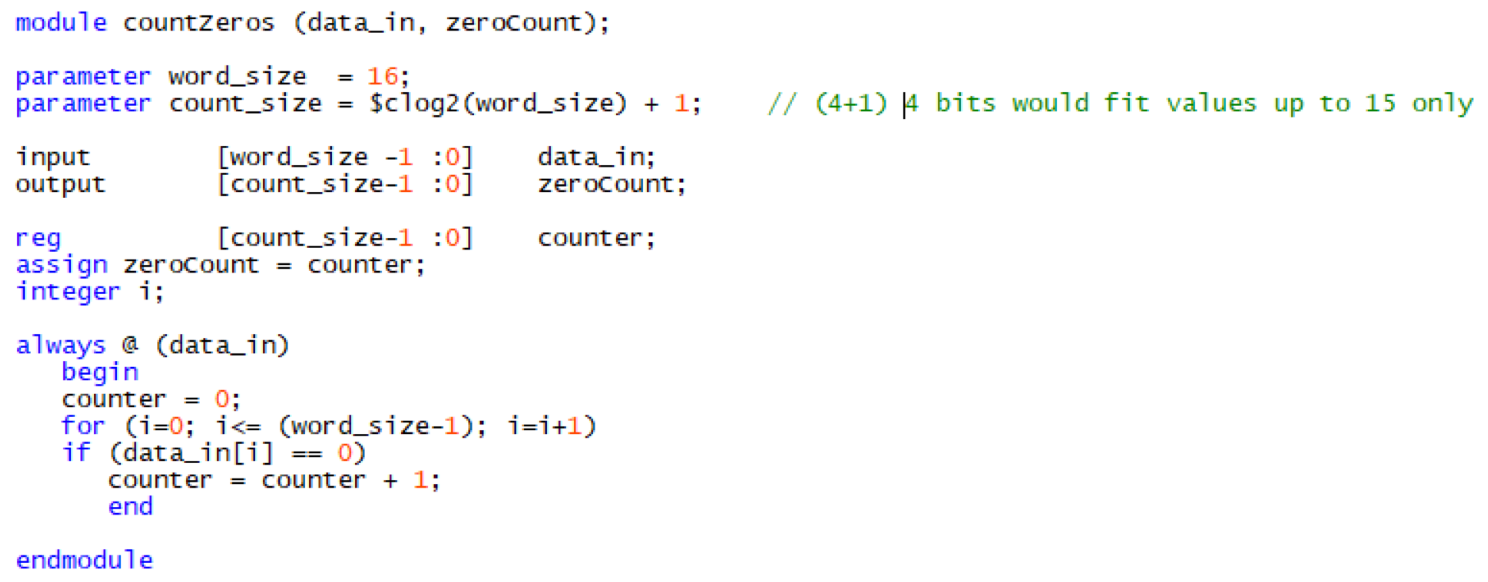
Lesson 6: Loops in Verilog

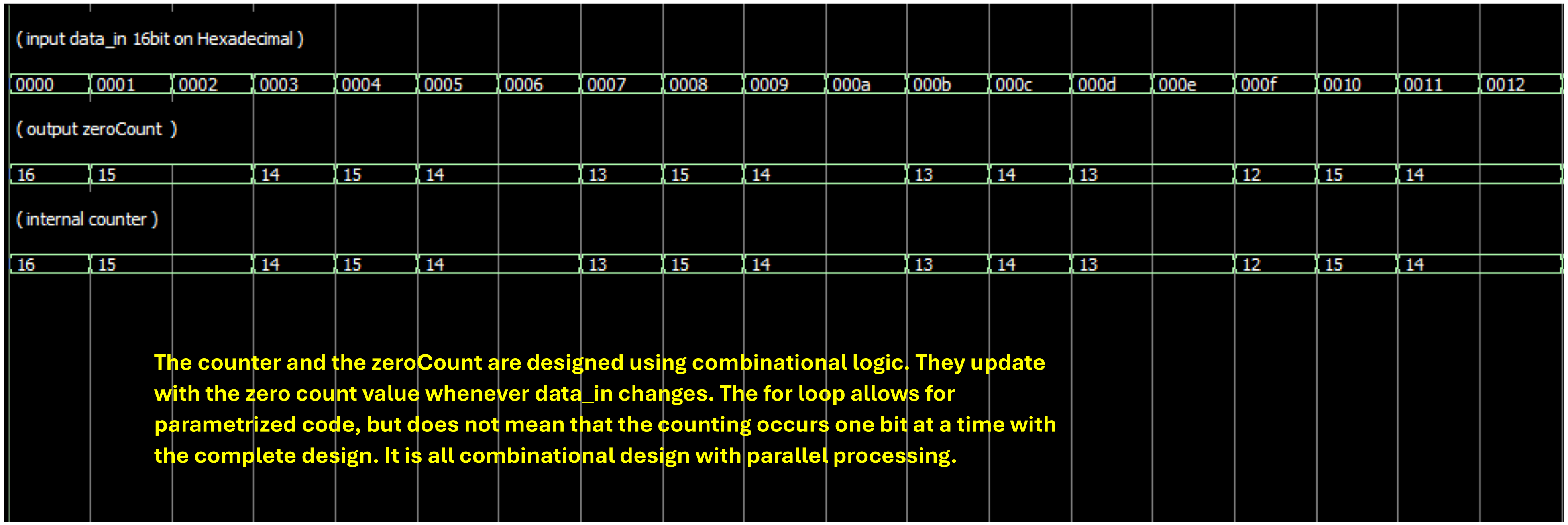
Assignment 1:

Design a module that counts the number of 0's in an input word. The code should be **parametrizable** with an initial value of word-size = 16. The word size should be a parameter that can be changed without updating anything else in the code, and yet get no errors while compiling the code and obtaining correct functionality. The module should represent combinational logic that would update the count whenever the input word changes.
Add any additional parameters you need and use any code style or loop (***for*** loop, ***while*** loop, or ***repeat***) you prefer.
Test the design using a testbench and verify that the design works for 16-bit words as well as 8-bit words. Indicate what parameters had to be updated for each case.
Share the Verilog code, the testbench code, and the simulation results, and how you verified the correct functionality of the code. Combine your screenshots and conclusion in one pdf file.



```
module countZeros (data_in, zeroCount);

parameter word_size  = 16;
parameter count_size = $clog2(word_size) + 1;    // (4+1) 4 bits would fit values up to 15 only

input        [word_size -1 :0]    data_in;
output       [count_size-1 :0]    zeroCount;

reg          [count_size-1 :0]    counter;
assign zeroCount = counter;
integer i;

always @ (data_in)
   begin
   counter = 0;
   for (i=0; i<= (word_size-1); i=i+1)
   if (data_in[i] == 0)
      counter = counter + 1;
      end

endmodule
```

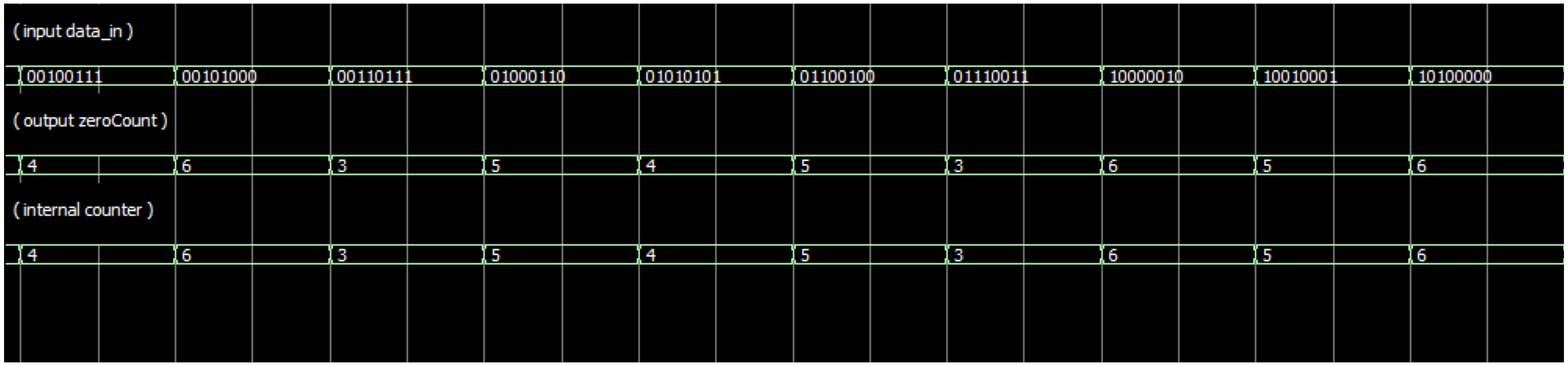
```verilog
module countZeros_tb ();

parameter word_size  = 16;
parameter count_size = $clog2(word_size) + 1;     // to be able to accomodate the maximum of 16

reg             [word_size -1 :0]     data_in;
wire            [count_size-1 :0]     zeroCount;
wire            [count_size-1 :0]     counter;

countZeros  UUT (data_in, zeroCount);

assign counter = UUT.counter;

initial
begin
     data_in = 0;
  repeat (40) begin
  #10  data_in = data_in + 1;            end
  repeat (40) begin
  #10   data_in = data_in + 16'hFF0F;   end
  end

 initial
 #800 $stop;

 endmodule
```

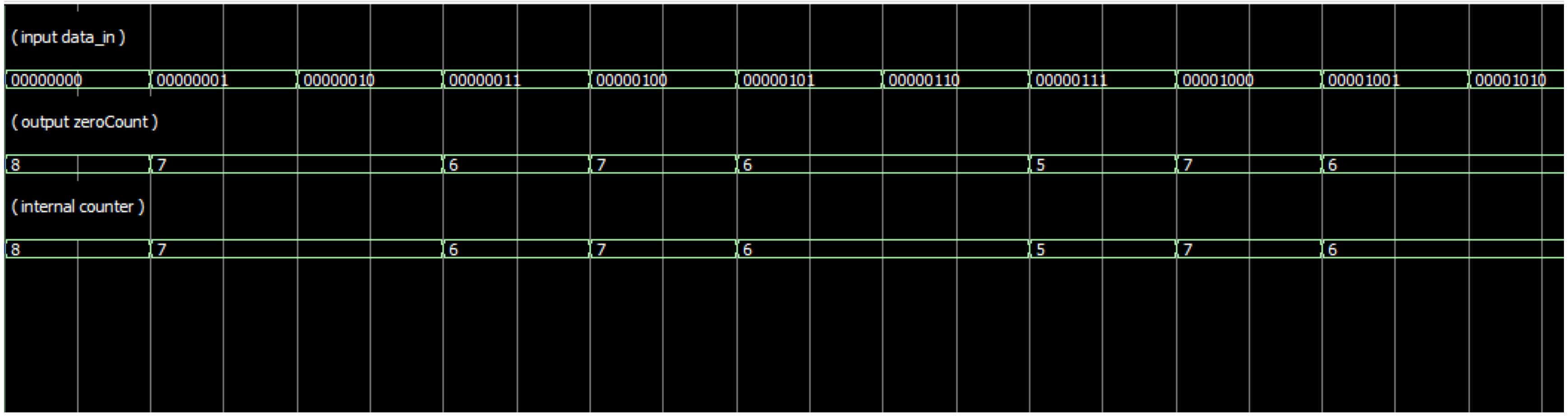


```
( input data_in 16bit on Hexadecimal )
0000 0001 0002 0003 0004 0005 0006 0007 0008 0009 000a 000b 000c 000d 000e 000f 0010 0011 0012
( output zeroCount )
16   15        14   15   14        13   15   14        13   14   13        12   15   14
( internal counter )
16   15        14   15   14        13   15   14        13   14   13        12   15   14
```

The counter and the zeroCount are designed using combinational logic. They update with the zero count value whenever data_in changes. The for loop allows for parametrized code, but does not mean that the counting occurs one bit at a time with the complete design. It is all combinational design with parallel processing.

```
module countZeros (data_in, zeroCount);

parameter word_size  = 8;
parameter count_size = $clog2(word_size) + 1;    // (3+1) 4 bits would fit values up to 7 only

input         [word_size -1 :0]    data_in;
output        [count_size-1 :0]    zeroCount;

reg           [count_size-1 :0]    counter;
assign zeroCount = counter;
integer i;

always @ (data_in)
   begin
   counter = 0;
   for (i=0; i<= (word_size-1); i=i+1)
   if (data_in[i] == 0)
      counter = counter + 1;
      end

endmodule


module countZeros_tb ();

parameter word_size  = 8;
parameter count_size = $clog2(word_size) + 1;    // to be able to accomodate the maximum of 8

reg           [word_size -1 :0]    data_in;
wire          [count_size-1 :0]    zeroCount;
wire          [count_size-1 :0]    counter;

countZeros  UUT (data_in, zeroCount);

assign counter = UUT.counter;

initial
begin
     data_in = 0;
  repeat (40) begin
  #10  data_in = data_in + 1;           end
  repeat (40) begin
  #10   data_in = data_in + 16'hFF0F;   end
  end

 initial
 #800 $stop;

 endmodule
```





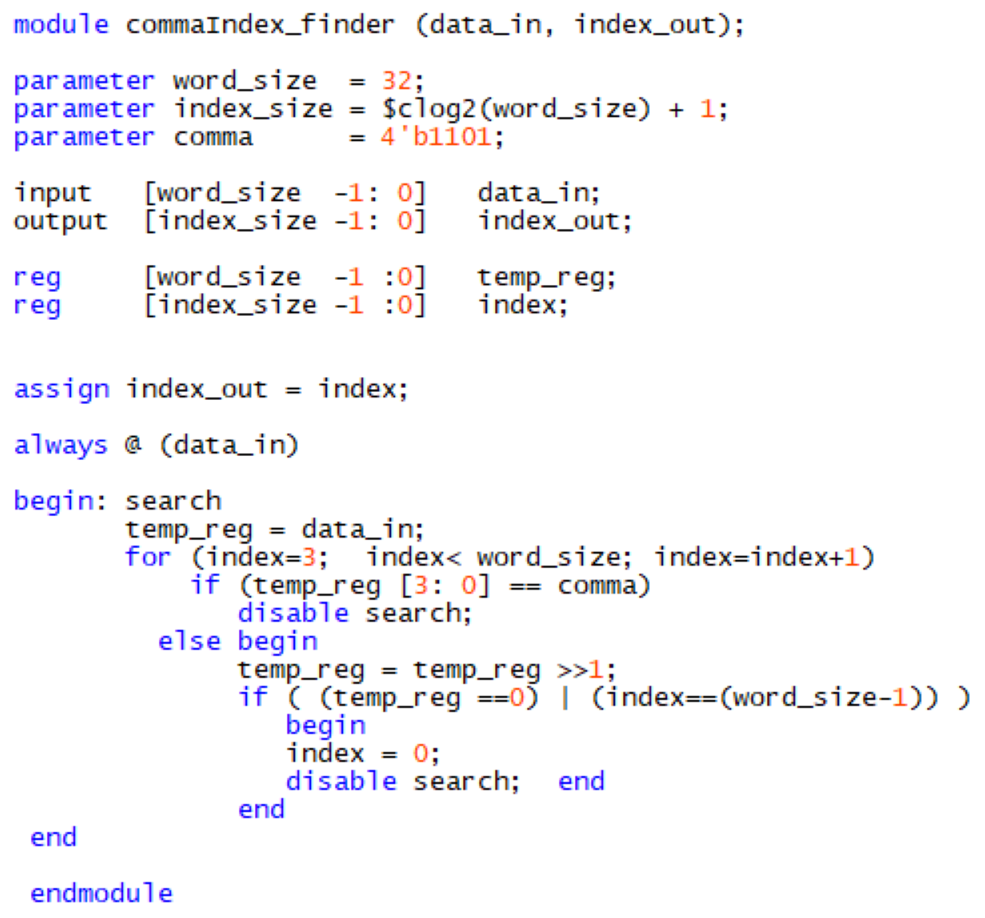
When changing the parameter value, the word size is updated and the functionality is verified.

Assignment 2:

In communication links comma symbols are used to correctly align words or mark the beginning of valid data. It is required to design a module that can detect a comma symbol of 4’b1101 in an input word starting from the LSB side. If the symbol is found, then the index (bit order) of the MSB of the comma symbol should be given at the output. If the code is not found then an index of zero, should appear at the output. The input word has a size of 32 bits.

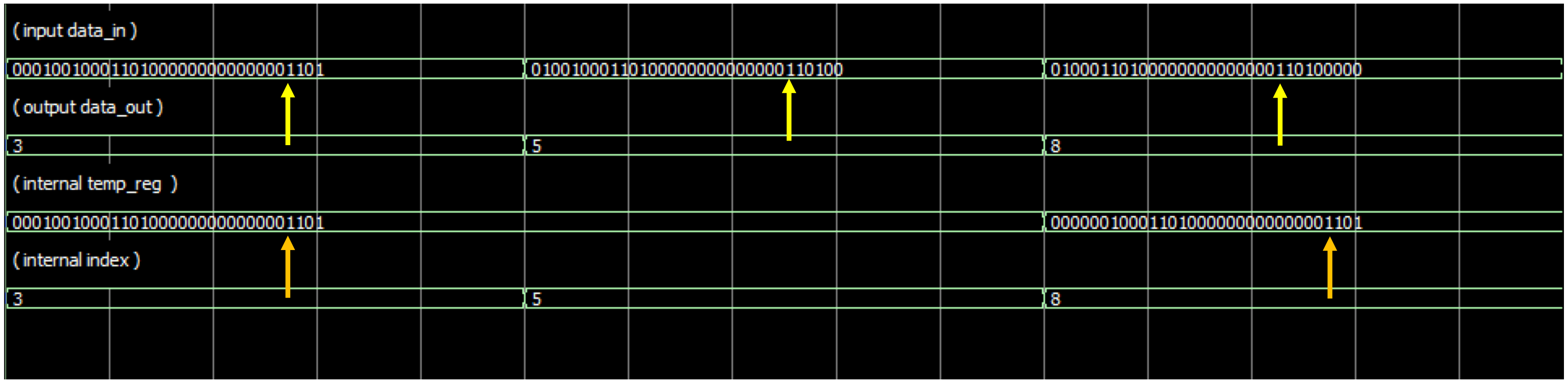
It is required to keep the design data independent and synthesizable. Use if statements and loops in your design.

Test the design using a test-bench and verify its functionality.
Share the Verilog code, the test-bench code, and the simulation results, and how you verified the correct functionality of the code. Combine your screenshots and conclusion in one pdf file.

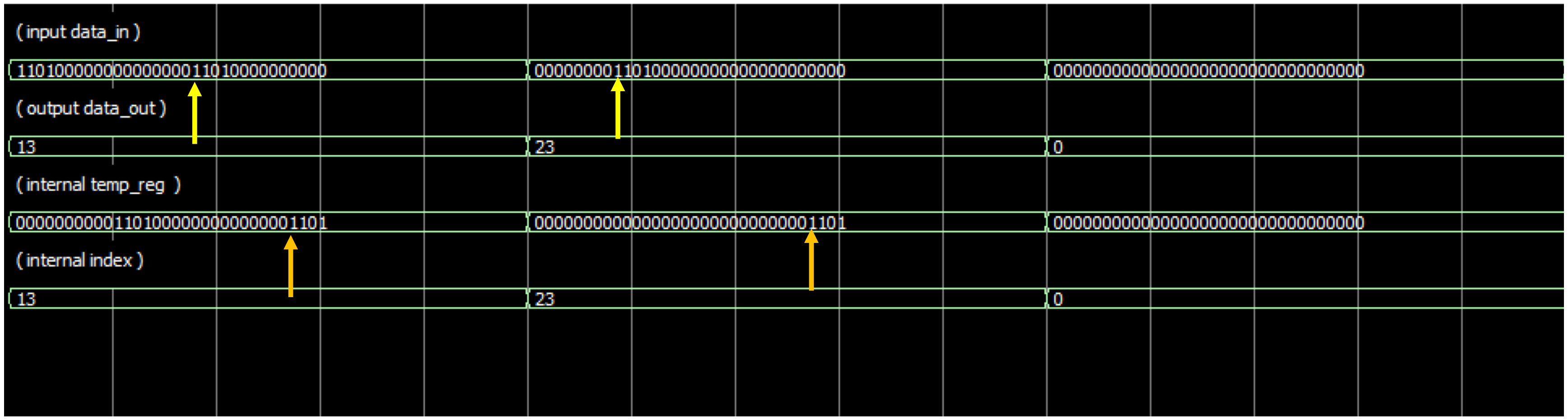


```
module commaIndex_finder (data_in, index_out);

parameter word_size  = 32;
parameter index_size = $clog2(word_size) + 1;
parameter comma      = 4'b1101;

input   [word_size  -1: 0]   data_in;
output  [index_size -1: 0]   index_out;

reg     [word_size  -1 :0]   temp_reg;
reg     [index_size -1 :0]   index;


assign index_out = index;

always @ (data_in)

begin: search
       temp_reg = data_in;
       for (index=3;  index< word_size; index=index+1)
           if (temp_reg [3: 0] == comma)
              disable search;
         else begin
              temp_reg = temp_reg >>1;
              if ( (temp_reg ==0) | (index==(word_size-1)) )
                 begin
                 index = 0;
                 disable search;   end
              end
 end

 endmodule
```

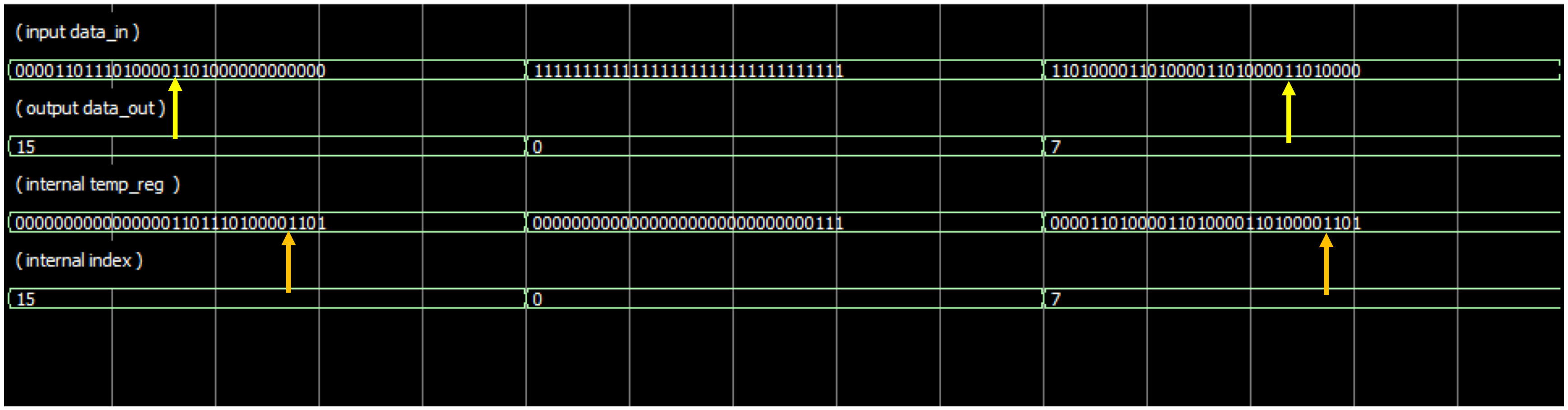
```
module index_tb ();

parameter word_size  = 32;
parameter index_size = $clog2(word_size) + 1;
parameter comma      = 4'b1101;                    // HEX = d

reg     [word_size  -1: 0]    data_in;
wire    [index_size -1: 0]    index_out;

wire    [word_size  -1 :0]    temp_reg;
wire    [index_size -1 :0]    index;

commaIndex_finder    UUT   (data_in, index_out);

assign temp_reg   = UUT.temp_reg;
assign index      = UUT.index;

initial
    begin
             data_in = 32'h_1234_000d;
             forever begin
      #10    data_in = data_in << 2 ;              // correct index of comma MSB selcted
      #10    data_in = data_in << 3 ;
      #10    data_in = data_in << 5 ;              // temp_reg should always align with d at LSB
      #10    data_in = data_in << 10;
      #10    data_in = 0;                          // zero value
      #10    data_in = 32'h_0dd0_d000;             // 3 commas, design should pick first
      #10    data_in = 32'h_FFFF_FFFF;             // non_zero valu without comma
      #10    data_in = 32'h_d0d0_d0d0;     end     // several commas, design should pick first
      end

initial
#200   $stop;

endmodule
```



```
( input data_in )
00010010001101000000000000001101    01001000110100000000000000110100    01000110100000000000000110100000
( output data_out )
3                                   5                                   8
( internal temp_reg )
00010010001101000000000000001101                                        00000010001101000000000000001101
( internal index )
3                                   5                                   8
```



```
( input data_in )
11010000000000000110100000000000    00000000110100000000000000000000    00000000000000000000000000000000
( output data_out )
13                                  23                                  0
( internal temp_reg )
00000000001101000000000000001101    00000000000000000000000000001101    00000000000000000000000000000000
( internal index )
13                                  23                                  0
```



```
( input data_in )
00001101111010000110100000000000    11111111111111111111111111111111    11010000110100001101000011010000
( output data_out )
15                                  0                                   7
( internal temp_reg )
00000000000000001101110100001101    00000000000000000000000000000111    00001101000011010000110100001101
( internal index )
15                                  0                                   7
```

From the simulation results it is noticed that:

1. The index holds the order of the MSB of the detected comma 1101. If the comma does not exist the index reports a zero.
2. The temp_reg holds a final value with the first right-most comma shifted all the way to the Least Significant 4bits.
3. If the comma does not exist, the temp_reg has the data_in MSB shifted to the LSB 4 bits and the rest of the bits are filled with zeros. This is the for loop condition when index reached (word_size-1) and the comma was still not detected. In this case the index_out becomes a zero.
4. If the data_in is a zero, the condition of (temp_reg==0) is reached and the index output becomes again a zero.