PROJECT EDA

Final delivery

CESS
OMAR ASHRAF MOSTAFA SAMY
20p1542

Introduction:

Cyclic Redundancy Checks (CRCs) are derived from the theory of cyclic error-correcting codes, which are designed to detect errors in communication networks. The concept of using systematic cyclic codes for error detection was first introduced by W. Wesley Peterson in 1961. Cyclic codes are advantageous due to their simplicity in implementation and their effectiveness in detecting burst errors, which are sequences of erroneous data symbols. This is particularly important as burst errors are common in various communication channels, including magnetic and optical storage devices.

An n-bit CRC applied to a data block of any length will reliably detect any single burst error up to n bits long. For longer burst errors, the probability of detection is approximately

(1 - 2^-n).

Defining a CRC code involves specifying a generator polynomial. This polynomial acts as the divisor in a polynomial long division, where the message is the dividend. The quotient from this division is discarded, and the remainder becomes the CRC value. It's important to note that the polynomial coefficients are calculated using the arithmetic of a finite field, allowing the addition operation to be performed bitwise-parallel, without carries between digits.

In practical applications, CRCs typically use the finite field of two elements, GF(2), where the elements are 0 and 1, aligning with computer architecture. An n-bit CRC produces a check value that is n bits long. Multiple CRCs can be defined for a given n, each with a unique polynomial of degree n, which requires n + 1 bits for encoding. Usually, either the most significant bit (MSB) or the least significant bit (LSB) of the polynomial is omitted in specifications, as they are always 1. CRCs and their polynomials are often named in the form CRC-n-XXX.

For example, the simplest error-detection system is the parity bit, which is actually a

1-bit CRC using the generator polynomial x + 1. This system is referred to as CRC-1.

A Cyclic Redundancy Check (CRC) is a widely used error-detecting code in digital networks and storage devices, designed to identify accidental changes to digital data. When data blocks enter these systems, a short check value is appended, derived from the remainder of a polynomial division of the data's contents. During data retrieval, this calculation is repeated, and if the check values don't match, corrective measures can be taken to address potential data corruption. While primarily used for error detection, CRCs can also assist in error correction.

The term "cyclic redundancy check" reflects that the verification value is a form of redundancy (it extends the message without adding new information) and the underlying algorithm is based on cyclic codes. CRCs are favored because they are straightforward to implement in binary hardware, easy to analyze mathematically, and highly effective at detecting common errors caused by transmission noise. The fixed length of the check value also allows the CRC function to be used as a hash function in some cases.

Calculating a Cyclic Redundancy Check (CRC) involves a few key steps: defining the generator polynomial, performing binary division, and appending the resulting remainder to the data. Here's a step-by-step guide on how to calculate a CRC:

1. **Define the Generator Polynomial:**
    - Choose a generator polynomial (also known as the divisor). This polynomial is a key part of the CRC calculation and is defined in binary form. For example, a common 3-bit CRC polynomial is x3+x+1x^3 + x + 1x3+x+1, which translates to the binary number 1011.
2. **Append Zeros to the Data:**
    - Append a number of zeros to the end of the data equivalent to the degree of the generator polynomial. For example, if the data is 110101 and the polynomial is of degree 3, append three zeros: 000110101.
3. **Perform Binary Division:**
    - Divide the data (with appended zeros) by the generator polynomial using binary (modulo-2) division. This division is similar to long division but uses XOR instead of subtraction.
    - Here's how to perform the division:
        - Align the leftmost bit of the divisor with the leftmost bit of the dividend.
        - If the leftmost bit of the current dividend portion is 1, perform XOR between the divisor and the current portion of the dividend, then bring down the next bit of the dividend.
        - If the leftmost bit is 0, bring down the next bit without performing XOR.
        - Repeat until all bits of the dividend have been processed.

4. **Obtain the Remainder:**
    - The remainder after the division is the CRC value. This remainder will have a length equal to the degree of the polynomial.
5. **Append the CRC to the Data:**
    - Append the CRC remainder to the original data. This combination forms the transmitted message.

In the case of CRC, the divisor is a polynomial upon which the sender and the receiver of the data packet agree. It's calculated by treating the polynomial as having binary coefficients, like in the following:

$x^5 + x^2 + 3$						
Exponent	5	4	3	2	1	0
Polynomial	1	0	0	1	0	1
		100101				

Cyclic Redundancy Checks (CRCs) are widely used in digital designs for several important reasons:

1. Error Detection:

- **High Detection Rate:** CRCs are highly effective at detecting common types of errors, especially burst errors, which are sequences of errors in contiguous bits. This makes them particularly useful in environments where data integrity is critical.
- **Single-Bit Errors:** CRCs can detect all single-bit errors, ensuring that any alteration of a single bit in the data can be identified.
- **Multiple Errors:** They can detect errors involving multiple bits with a high probability, depending on the polynomial used.

2. Efficiency:

- **Hardware Implementation:** CRCs are simple to implement in hardware using shift registers and XOR gates, making them fast and efficient. This simplicity allows CRC calculations to be performed in real-time, which is crucial for high-speed data communication.
- **Low Computational Overhead:** The algorithm for calculating CRCs is computationally efficient, requiring only bitwise operations, which are less

demanding than other error-checking methods like checksums or cryptographic hashes.

3. **Versatility:**

- **Adaptability:** CRCs can be adapted for different data lengths and error-detection requirements by selecting appropriate generator polynomials. This makes them versatile for various applications, from small embedded systems to large data networks.
- **Compatibility with Binary Data:** CRCs work seamlessly with binary data, which is the native format for digital systems, ensuring minimal conversion or processing overhead.

4. **Integrity Verification:**

- **Data Integrity:** CRCs provide a straightforward way to verify the integrity of data during transmission and storage. By appending a CRC to the data, the receiver can check if the data has been altered during transmission or storage.
- **Error Correction Assistance:** While CRCs are primarily used for error detection, they can also assist in error correction when combined with other error-correcting codes and algorithms.

5. **Widespread Use:**

- **Standardization:** CRCs are standardized and widely adopted in various communication protocols and storage formats. This standardization ensures interoperability and reliability across different systems and devices.
- **Broad Application Range:** CRCs are used in numerous applications, including network communications (Ethernet, Wi-Fi), storage devices (hard drives, SSDs), file formats (ZIP, PNG), and embedded systems.

6. **Reliability:**

- **Robustness:** CRCs add a layer of robustness to digital systems by ensuring that even minor errors can be detected and addressed, enhancing the overall reliability of the system.
- **Noise Resilience:** They are particularly effective in environments with high noise levels, such as wireless communication and industrial settings, where data integrity is often compromised.

Literature Review:

Cyclic Redundancy Checks (CRCs) are essential in ensuring data integrity in digital communication and storage systems. Various approaches have been developed to implement CRC modules using VHDL (VHSIC Hardware Description Language), each with unique advantages and trade-offs. This literature review surveys different methods for implementing CRC modules, highlighting key techniques and their performance implications.

1. Basic Linear Feedback Shift Register (LFSR) Approach

- **Overview:** The most fundamental approach involves using a Linear Feedback Shift Register (LFSR) to implement the CRC algorithm. This method leverages shift registers and XOR gates to compute the CRC value.
- **Implementation:**
    - The input data is fed into a series of shift registers.
    - XOR operations are performed according to the generator polynomial.
- **Advantages:** Simple and straightforward to implement; efficient in terms of hardware resources.
- **Disadvantages:** Limited processing speed due to the bit-by-bit processing nature.

2. Parallel CRC Calculation

- **Overview:** Parallel CRC computation techniques aim to increase the processing speed by handling multiple bits of the input data simultaneously.
- **Implementation:**
    - The input data is divided into smaller chunks that are processed in parallel.
    - Specialized circuits are designed to perform parallel XOR operations.
- **Advantages:** Significantly faster than the basic LFSR approach; suitable for high-speed applications.
- **Disadvantages:** More complex hardware design; increased resource utilization.

3. Pipelined CRC Architecture

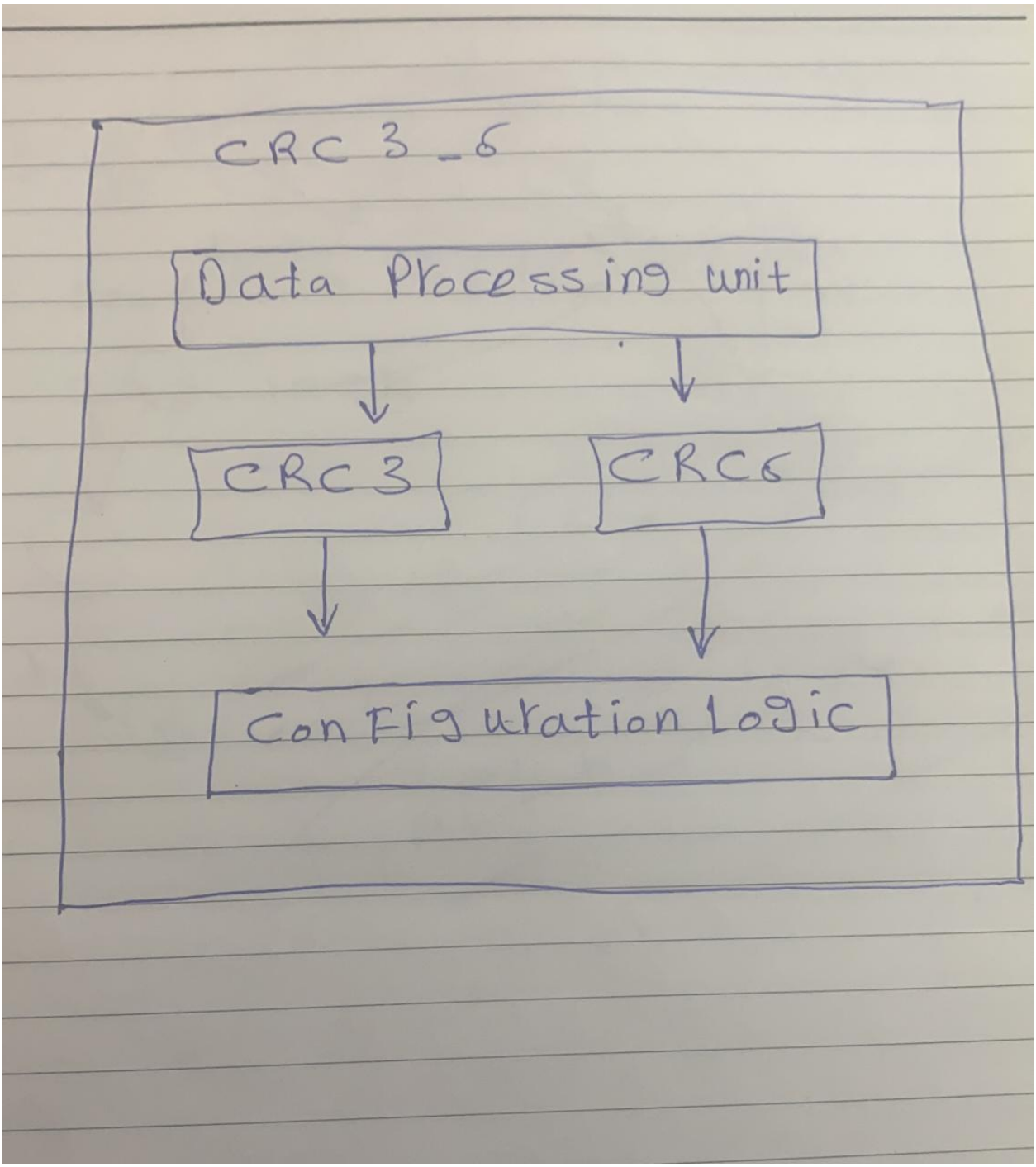
- **Overview:** Pipelined architectures further enhance the speed of CRC computation by introducing pipeline stages.
- **Implementation:**
    - The input data is divided and processed in pipeline stages.
    - Each stage performs part of the CRC calculation, allowing multiple data chunks to be processed concurrently.
- **Advantages:** High throughput; reduces latency by overlapping operations.
- **Disadvantages:** Increased design complexity; requires careful management of pipeline stages to avoid data hazards.

4. Look-Up Table (LUT) Based Approach

- **Overview:** This method uses precomputed values stored in look-up tables to accelerate the CRC computation.
- **Implementation:**
    - Precompute CRC values for all possible input byte values.
    - During computation, use the input data to index into the LUT and retrieve precomputed CRC values.
- **Advantages:** Extremely fast; minimal logic required during runtime.
- **Disadvantages:** Requires significant memory for the LUT; less flexible for different polynomials.

System Design:

Block diagram



Data Flow

The data flow involves the movement of data through different stages of processing within the module.

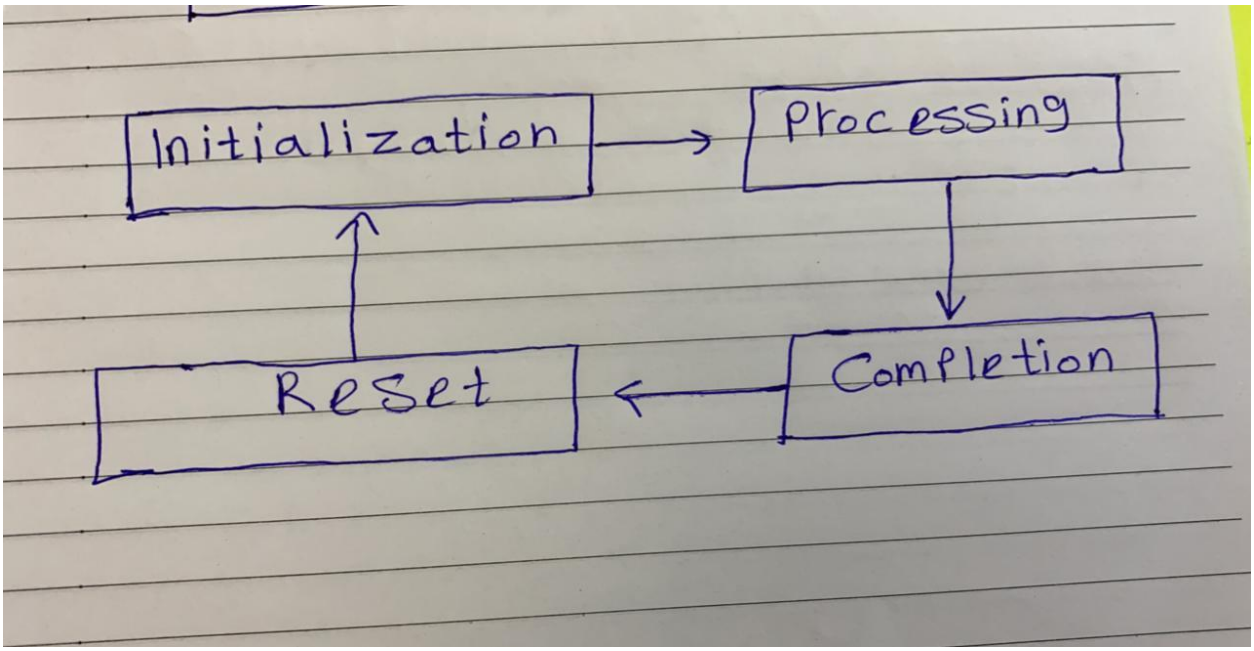
- Input Stage: Data is received via Data_in, config, Clk, Rst, and Data_valid signals.
- Processing Stage: Data is processed based on the CRC configuration (3-bit or 6-bit).
- Output Stage: The computed CRC value is sent to the Crc_out port, and the Crc_valid signal indicates the validity of the CRC output.

state machine implementation

The module operates as a finite state machine with the following states:

1. Initialization: Resets CRC values based on the configuration.
2. Processing: Computes the CRC based on the incoming data.
3. Completion: Outputs the final CRC value.

State Diagram:



Detailed Implementation

Input and Output Ports

- Inputs:
    - Clk: Clock signal.
    - Rst: Reset signal.
    - Data_valid: Validity of input data.
    - Data_in: Input data vector (38 bits).
    - config: Configuration for CRC (3 bits).
- Outputs:
    - Crc_out: Computed CRC output (6 bits).
    - Crc_valid: Validity of CRC output.

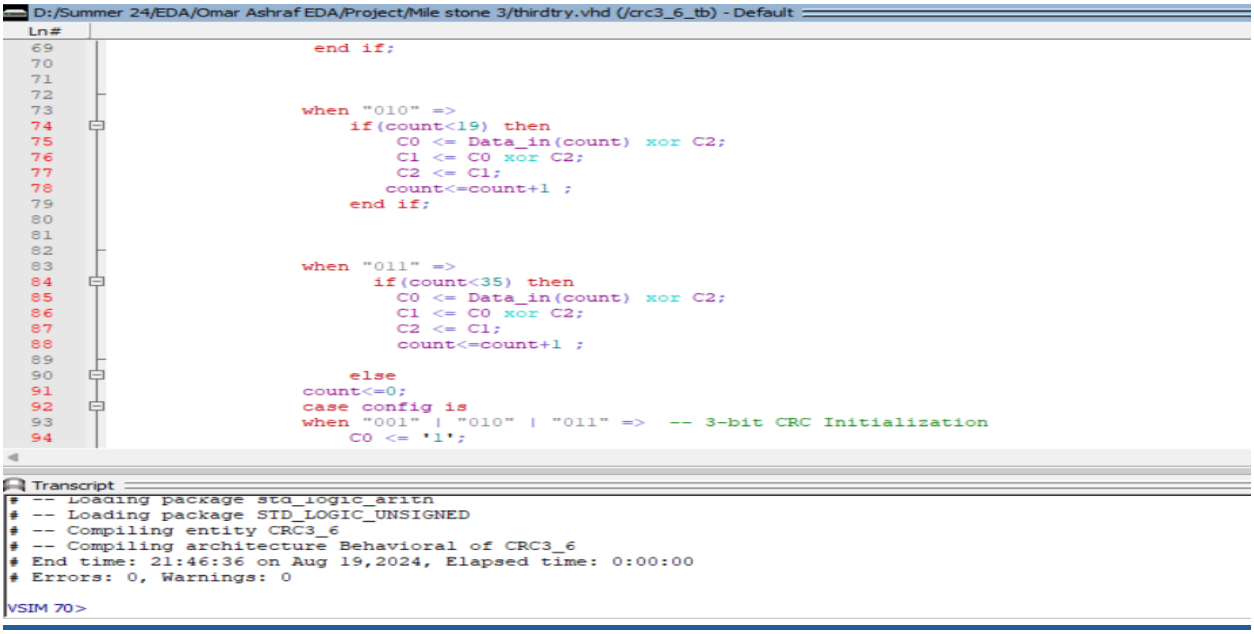
Internal Signals

- C0, C1, C2, C3, C4, C5: Internal signals representing CRC bits.
- count: Counter for tracking processing steps.

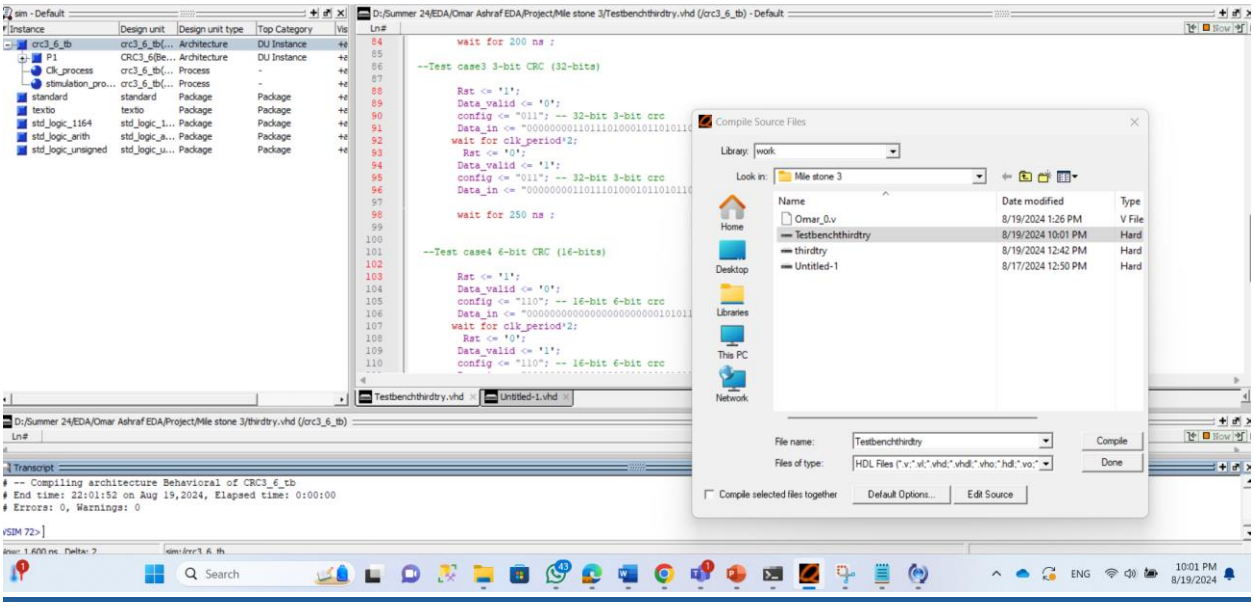
Finally: The CRC3_6 module is designed to compute either a 3-bit or 6-bit CRC based on the provided configuration. It processes the input data iteratively and outputs the computed CRC value when the computation is valid. The design includes initialization, processing, and output assignment stages, implemented using VHDL.

System Simulation:

This is the compilation of the VHDL code showing that has Zero Errors and Warnings.



This is the compilation of the Test bench code showing that has Zero Errors and Warnings.



All test cases are running without any errors. I work on 6 test cases, 3 test cases for the 3-bit CRC and the rest for 6-bit CRC.

Steps for simulation:

You first must compile the code then start simulation the add wave if you test manually you have to force the reset =1 firstly and the clock=0 then get reset=0 again and force the config and the data in you want to enter

First test case data input =AD2C output=4→ (16 bits 3-bit CRC)

Second test case data input = ox75 output=0x1→→(8 bits 3-bit CRC)

Third test case data input = ox3745adfc output=0x4→(32 bits 3-bit CRC)

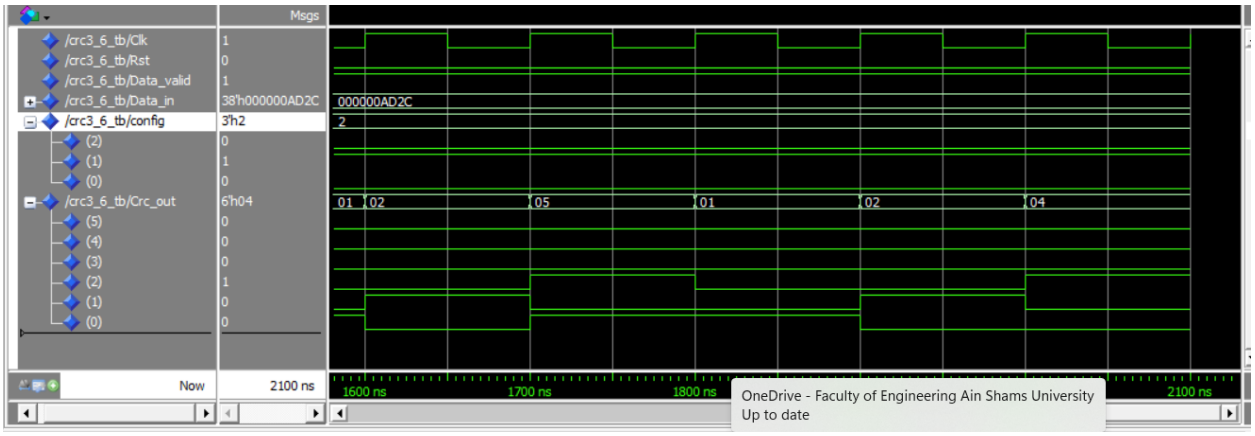
Fourth test case data input = ox57CB output=0x19 → (16 bits 6-bit CRC)

Fifth test case data input = ox35output=0x25→(8 bits 6-bit CRC)

Sixth test case data input =ox345ABCD output=0x2f→ (32 bits 6-bit CRC)

In the part of simulation of the wave form all these tests showed that they are valid there strategy to test the code that it work effectively with the 3-bit CRC and the 6-bit CRC with variety of the number of bits you chose they can be 8or16or32 for each type of the CRC.

Example of valid test case I test it in the lecture today AD2C



VHDL Code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity CRC3_6 is
    Port (
        Clk, Rst, Data_valid : in  std_logic;
        Data_in   : in  std_logic_vector(37 downto 0);
        config    : in  std_logic_vector(2 downto 0);
        Crc_out   : out std_logic_vector (5 downto 0);
        Crc_valid : out std_logic
    );
end CRC3_6;

architecture Behavioral of CRC3_6 is
    signal C0, C1, C2, C3, C4, C5 : std_logic;
    signal count : integer := 0;
begin

process(Clk, Rst)
begin
    if rising_edge(Clk) then
        if Rst = '1' then
            case config is
                when "001" | "010" | "011" =>  -- 3-bit CRC Initialization
```

```
            C0 <= '1';
            C1 <= '1';
            C2 <= '1';
            Crc_valid <= '0';
        when "101" | "110" | "111" =>  -- 6-bit CRC Initialization
            C0 <= '1';
            C1 <= '0';
            C2 <= '1';
            C3 <= '0';
            C4 <= '1';
            C5 <= '0';
            Crc_valid <= '0';
        when others =>
    end case;
elsif Data_valid = '1' then
    case config is
        when "001" =>
            if(count<11) then
                C0 <= Data_in(count) xor C2;
                C1 <= C0 xor C2;
                C2 <= C1;
                 count<=count+1;

              else
              count<=0;
              case config is
        when "001" | "010" | "011" =>  -- 3-bit CRC Initialization
            C0 <= '1';
```

```
        C1 <= '1';
        C2 <= '1';
        Crc_valid <= '0';
    when "101" | "110" | "111" =>  -- 6-bit CRC Initialization
        C0 <= '1';
        C1 <= '0';
        C2 <= '1';
        C3 <= '0';
        C4 <= '1';
        C5 <= '0';
        Crc_valid <= '0';
    when others =>
end case;


          end if;
    when "010" =>
        if(count<19) then
            C0 <= Data_in(count) xor C2;
            C1 <= C0 xor C2;
            C2 <= C1;
           count<=count+1 ;
        end if;
    when "011" =>
          if(count<35) then
            C0 <= Data_in(count) xor C2;
            C1 <= C0 xor C2;
            C2 <= C1;
            count<=count+1 ;
```

```
        else
         count<=0;
         case config is
    when "001" | "010" | "011" =>  -- 3-bit CRC Initialization
       C0 <= '1';
       C1 <= '1';
       C2 <= '1';
       Crc_valid <= '0';
    when "101" | "110" | "111" =>  -- 6-bit CRC Initialization
       C0 <= '1';
       C1 <= '0';
       C2 <= '1';
       C3 <= '0';
       C4 <= '1';
       C5 <= '0';
       Crc_valid <= '0';
    when others =>
end case;


          end if;
    when "101" =>
       if(count<14) then
          C0 <= Data_in(count) xor C5;
          C1 <= C0;
          C2 <= C1;
          C3 <= C2 xor C5;
          C4 <= C3 xor C5;
```

```
            C5 <= C4;
            count<=count+1 ;

          else
            count<=0;
            case config is
      when "001" | "010" | "011" =>  -- 3-bit CRC Initialization
         C0 <= '1';
         C1 <= '1';
         C2 <= '1';
         Crc_valid <= '0';
      when "101" | "110" | "111" =>  -- 6-bit CRC Initialization
         C0 <= '1';
         C1 <= '0';
         C2 <= '1';
         C3 <= '0';
         C4 <= '1';
         C5 <= '0';
         Crc_valid <= '0';
      when others =>
   end case;
        end if;
      when "110" =>
            if(count<22) then
             C0 <= Data_in(count) xor C5;
             C1 <= C0;
             C2 <= C1;
             C3 <= C2 xor C5;
```

```
            C4 <= C3 xor C5;
            C5 <= C4;
             count<=count+1 ;

        else
             count<=0;
             case config is
        when "001" | "010" | "011" =>  -- 3-bit CRC Initialization
            C0 <= '1';
            C1 <= '1';
            C2 <= '1';
            Crc_valid <= '0';
        when "101" | "110" | "111" =>  -- 6-bit CRC Initialization
            C0 <= '1';
            C1 <= '0';
            C2 <= '1';
            C3 <= '0';
            C4 <= '1';
            C5 <= '0';
            Crc_valid <= '0';
        when others =>
    end case;

            end if;


        when "111" =>
            if(count<38) then
```

```
            C0 <= Data_in(count) xor C5;
            C1 <= C0;
            C2 <= C1;
            C3 <= C2 xor C5;
            C4 <= C3 xor C5;
            C5 <= C4;
           count<=count+1 ;

        else
          count<=0;
          case config is
      when "001" | "010" | "011" =>  -- 3-bit CRC Initialization
         C0 <= '1';
         C1 <= '1';
         C2 <= '1';
         Crc_valid <= '0';
      when "101" | "110" | "111" =>  -- 6-bit CRC Initialization
         C0 <= '1';
         C1 <= '0';
         C2 <= '1';
         C3 <= '0';
         C4 <= '1';
         C5 <= '0';
         Crc_valid <= '0';
      when others =>
   end case;

          end if;
```

```
            when others =>
            end case;
            Crc_valid <= '1';
        else
            Crc_valid <= '0';
        end if;
    end if;
end process;

Crc_out <=
    C5 & C4 & C3 & C2 & C1 & C0 when (config = "101" or config = "110" or config =
"111") else
    "000" & C2 & C1 & C0 when (config = "001" or config = "010" or config = "011") else
    (others => '0');  -- Default case to avoid latching



end Behavioral;
```

Test bench code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CRC3_6_tb is
end CRC3_6_tb;

architecture Behavioral of CRC3_6_tb is
  component CRC3_6
   Port (
      Clk        : in  std_logic;
      Rst        : in  std_logic;
      Data_valid : in  std_logic;
      Data_in    : in  std_logic_vector(37 downto 0);
      config     : in  std_logic_vector(2 downto 0);
      Crc_out    : out std_logic_vector(5 downto 0);
      Crc_valid  : out std_logic
   );
   end component;


   signal Clk        : std_logic;
   signal Rst        : std_logic;
   signal Data_valid : std_logic;
   signal Data_in    : std_logic_vector(37 downto 0);
   signal config     : std_logic_vector(2 downto 0);
```

```
    signal Crc_out    : std_logic_vector(5 downto 0);
    signal Crc_valid  : std_logic;


    constant clk_period : time := 10 ns;
begin


    P1: CRC3_6
        Port map (
            Clk        => Clk,
            Rst        => Rst,
            Data_valid => Data_valid,
            Data_in    => Data_in,
            config     => config,
            Crc_out    => Crc_out,
            Crc_valid  => Crc_valid
        );


    Clk_process :process
    begin
        Clk <= '0';
        wait for clk_period/2;
        Clk <= '1';
        wait for clk_period/2;
    end process;
```

```
    stimulation_process: process
    begin
        --Test case1 3-bit CRC 16-bits
        Rst <= '1';
          Data_valid <= '0';
          config <= "010"; -- 16-bit 3-bit crc
        Data_in <= "00000000000000000000001010110100101100"; --AD2C
       wait for clk_period*2;
Rst <= '0';
        Data_valid <= '1';
          config <= "010"; -- 16-bit 3-bit crc
        Data_in <= "00000000000000000000001010110100101100"; --AD2C
        --Data_in <= "00000001110010100101001100001001101000"
        wait for 200 ns ;

     --Test case2 3-bit CRC (8-bits)

        Rst <= '1';
          Data_valid <= '0';
          config <= "001"; -- 8-bit 3-bit crc
        Data_in <= "00000000000000000000000000000001110101"; --ox75
       wait for clk_period*2;
         Rst <= '0';
        Data_valid <= '1';
          config <= "001"; -- 8-bit 3-bit crc
        Data_in <= "00000000000000000000000000000001110101"; --ox75
```

```
        wait for 200 ns ;

 --Test case3 3-bit CRC (32-bits)

        Rst <= '1';
          Data_valid <= '0';
          config <= "011"; -- 32-bit 3-bit crc
        Data_in <= "00000000110111010001011010110111111100"; --ox3745adfc
       wait for clk_period*2;
         Rst <= '0';
        Data_valid <= '1';
          config <= "011"; -- 32-bit 3-bit crc
        Data_in <= "00000000110111010001011010110111111100"; --ox3745adfc

        wait for 350 ns ;


  --Test case4 6-bit CRC (16-bits)

        Rst <= '1';
          Data_valid <= '0';
          config <= "110"; -- 16-bit 6-bit crc
        Data_in <= "00000000000000000000000101011111001011"; --ox57CB
       wait for clk_period*2;
         Rst <= '0';
        Data_valid <= '1';
          config <= "110"; -- 16-bit 6-bit crc
        Data_in <= "00000000000000000000000101011111001011"; --ox57CB
```

```
  wait for 200 ns ;

 --Test case5 6-bit CRC (8-bits)

   Rst <= '1';
      Data_valid <= '0';
      config <= "101"; -- 8-bit 6-bit crc
   Data_in <= "00000000000000000000000000000000110101"; --ox35
  wait for clk_period*2;
    Rst <= '0';
   Data_valid <= '1';
      config <= "101"; -- 8-bit 3-bit crc
   Data_in <= "00000000000000000000000000000000110101"; --ox35

   wait for 200 ns ;

 --Test case6 6-bit CRC (32-bits)

   Rst <= '1';
      Data_valid <= '0';
      config <= "111"; -- 32-bit 6-bit crc
   Data_in <= "00000000000011010001011010101111001101"; --ox345ABCD
  wait for clk_period*2;
    Rst <= '0';
   Data_valid <= '1';
      config <= "111"; -- 32-bit 3-bit crc
   Data_in <= "00000000000011010001011010101111001101"; --ox345ABCD
```

```
        wait for 250 ns ;

      wait;
    end process;

end Behavioral ;
```

Implementation Reports:

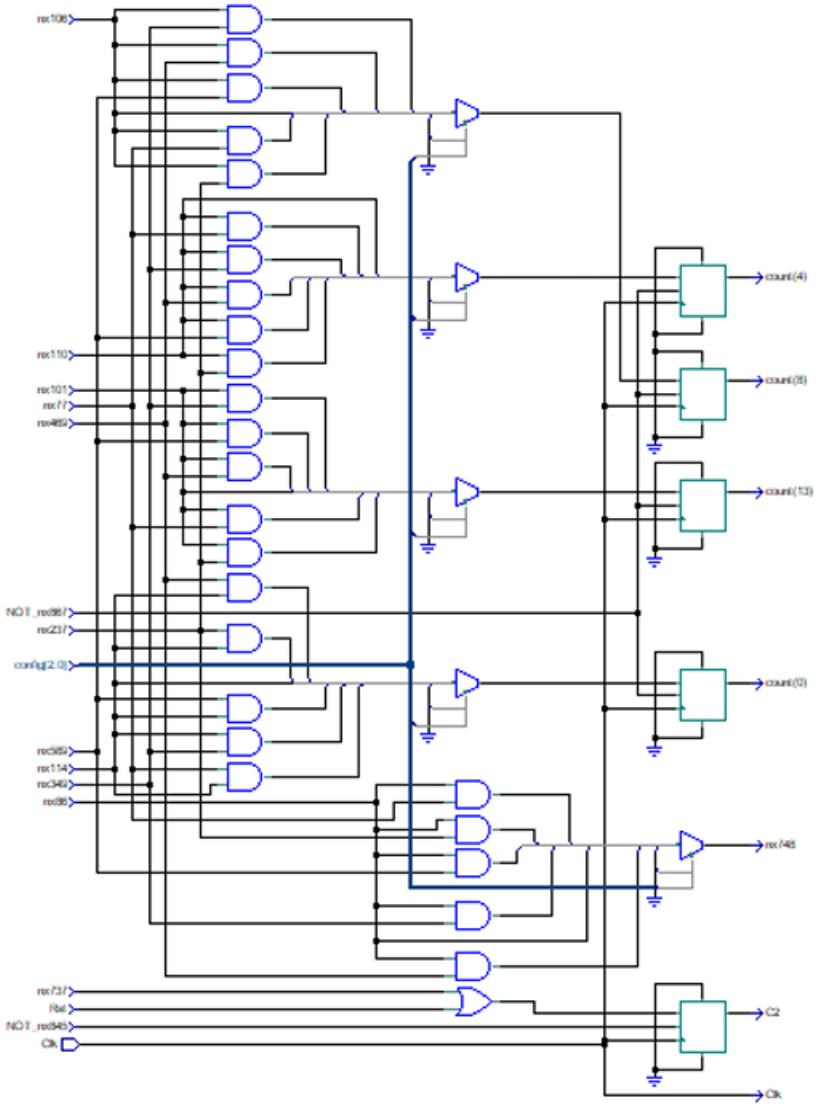
Report Area:

```
AN1V1      scl05u      1 x       5       5 gates
AN2T0      scl05u     33 x       5     155 gates
AN3T0      scl05u      2 x       6      12 gates
AN4T0      scl05u      1 x       8       8 gates
AO1A0      scl05u      4 x       6      25 gates
AO1I0      scl05u      2 x       6      12 gates
AO1I1      scl05u      1 x       6       6 gates
AO2I0      scl05u      1 x       8       8 gates
AO2L0      scl05u      1 x       8       8 gates
AO2L1      scl05u      1 x       8       8 gates
AO6L0      scl05u      3 x      11      32 gates
AO6L2      scl05u      1 x      11      11 gates
AO8L0      scl05u      8 x      14     114 gates
AOA4I0     scl05u      7 x       8      53 gates
FD1H0      scl05u     34 x       9     306 gates
FD1H1      scl05u      5 x      10      50 gates
HA1A0      scl05u     30 x       7     198 gates
IV1N0      scl05u     17 x       3      53 gates
IV1N2      scl05u      1 x       3       3 gates
MX2L0      scl05u      1 x       6       6 gates
MX2T0      scl05u      1 x       6       6 gates
ND2N0      scl05u     14 x       5      63 gates
ND3N0      scl05u      2 x       6      12 gates
NR2R0      scl05u     18 x       5      81 gates
NR2R1      scl05u      2 x       5       9 gates
NR3R1      scl05u      1 x       6       6 gates
NR3R2      scl05u      3 x       7      20 gates
NR5R0      scl05u      1 x       9       9 gates
OA1R0      scl05u      1 x       6       6 gates
OAI1A0     scl05u      1 x       6       6 gates
OAI1A1     scl05u      2 x       6      13 gates
OAI2N0     scl05u      1 x       8       8 gates
OAI3N0     scl05u      4 x       8      30 gates
OAI5R0     scl05u      1 x      11      11 gates
OAOI0      scl05u      3 x       8      23 gates
OR2T0      scl05u      1 x       5       5 gates
OR3T0      scl05u      1 x       6       6 gates
OR4T0      scl05u      1 x       8       8 gates
OR6T0      scl05u      3 x      11      33 gates
OR8T0      scl05u      1 x      14      14 gates
XN2R0      scl05u      2 x       5      10 gates
XR2T0      scl05u      4 x       5      20 gates

 Number of ports :                         51
 Number of nets :                         304
 Number of instances :                    222
 Number of references to this view :        0

Total accumulated area :
 Number of gates :                       1473
 Number of accumulated instances :        222
Info, Command 'report_area' finished successfully
```

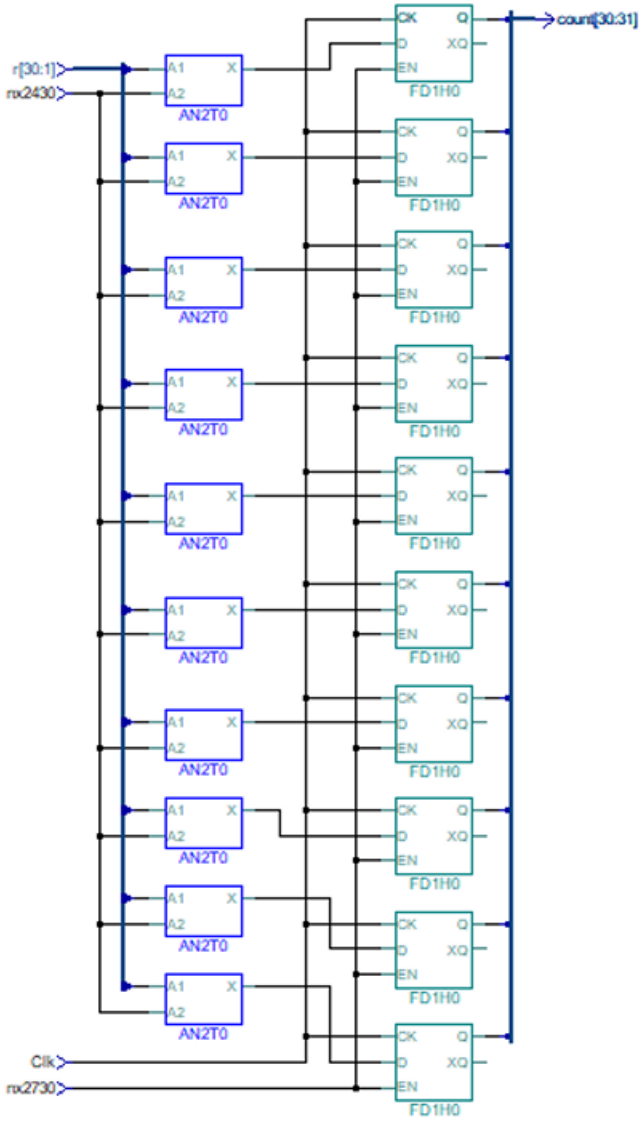
Report Delay:

```
Critical path #1, (path slack =  5.0):

NAME                                    GATE              ARRIVAL              LOAD
-------------------------------------------------------------------------------------
clock information not specified
delay thru clock network                                  0.00 (ideal)


reg_count(0)/Q                          FD1H1       0.00  0.60 up              0.60
count_inc_9_D0_I0_dup_135/CO            HA1A0       0.56  1.16 up              0.17
count_inc_9_D0_I0_dup_136/CO            HA1A0       0.43  1.58 up              0.17
count_inc_9_D0_I0_dup_137/CO            HA1A0       0.43  2.01 up              0.17
count_inc_9_D0_I0_dup_138/CO            HA1A0       0.43  2.44 up              0.17
count_inc_9_D0_I0_dup_139/CO            HA1A0       0.43  2.87 up              0.17
count_inc_9_D0_I0_dup_140/CO            HA1A0       0.43  3.29 up              0.17
count_inc_9_D0_I0_dup_141/CO            HA1A0       0.43  3.72 up              0.17
count_inc_9_D0_I0_dup_142/CO            HA1A0       0.43  4.15 up              0.17
count_inc_9_D0_I0_dup_143/CO            HA1A0       0.43  4.58 up              0.17
count_inc_9_D0_I0_dup_144/CO            HA1A0       0.43  5.00 up              0.17
count_inc_9_D0_I0_dup_145/CO            HA1A0       0.43  5.43 up              0.17
count_inc_9_D0_I0_dup_146/CO            HA1A0       0.43  5.86 up              0.17
count_inc_9_D0_I0_dup_147/CO            HA1A0       0.43  6.29 up              0.17
count_inc_9_D0_I0_dup_148/CO            HA1A0       0.43  6.72 up              0.17
count_inc_9_D0_I0_dup_149/CO            HA1A0       0.43  7.14 up              0.17
count_inc_9_D0_I0_dup_150/CO            HA1A0       0.43  7.57 up              0.17
count_inc_9_D0_I0_dup_151/CO            HA1A0       0.43  8.00 up              0.17
count_inc_9_D0_I0_dup_152/CO            HA1A0       0.43  8.43 up              0.17
count_inc_9_D0_I0_dup_153/CO            HA1A0       0.43  8.85 up              0.17
count_inc_9_D0_I0_dup_154/CO            HA1A0       0.43  9.28 up              0.17
count_inc_9_D0_I0_dup_155/CO            HA1A0       0.43  9.71 up              0.17
count_inc_9_D0_I0_dup_156/CO            HA1A0       0.43 10.14 up              0.17
count_inc_9_D0_I0_dup_157/CO            HA1A0       0.43 10.56 up              0.17
count_inc_9_D0_I0_dup_158/CO            HA1A0       0.43 10.99 up              0.17
count_inc_9_D0_I0_dup_159/CO            HA1A0       0.43 11.42 up              0.17
count_inc_9_D0_I0_dup_160/CO            HA1A0       0.43 11.85 up              0.17
count_inc_9_D0_I0_dup_161/CO            HA1A0       0.43 12.27 up              0.17
count_inc_9_D0_I0_dup_162/CO            HA1A0       0.43 12.70 up              0.17
count_inc_9_D0_I0_dup_163/CO            HA1A0       0.43 13.13 up              0.17
count_inc_9_D0_I0_dup_164/CO            HA1A0       0.41 13.54 up              0.14
ix2598/X                                XN2R0       0.36 14.03 dn              0.06
ix443/X                                 OAOI0       0.32 14.35 up              0.06
reg_count(31)/D                         FD1H0       0.00 14.35 up              0.00
data arrival time                                        14.35


data required time  (default specified - setup time)       19.30
-------------------------------------------------------------------------------------
data required time                                         19.30
data arrival time                                          14.35
                                                        ----------
slack                                                    4.95
-------------------------------------------------------------------------------------
```

Schematic:



Technology Schematic:



Netlist:

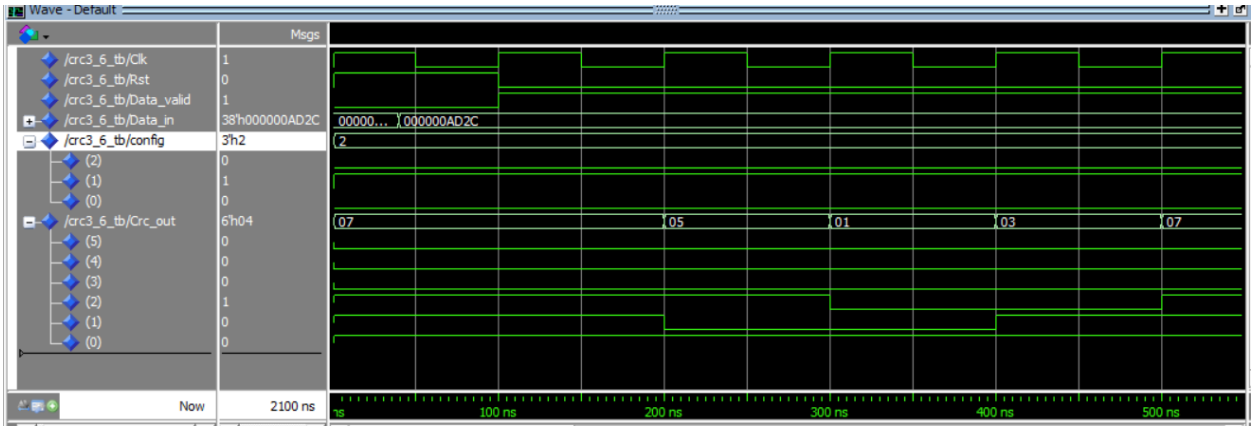
```
// Verilog description for cell CRC3_6,
// 08/19/24 13:22:25
//
// LeonardoSpectrum Level 3, 2020a.2
//


module CRC3_6 ( Clk, Rst, Data_valid, Data_in, config, Crc_out, Crc_valid ) ;

    input Clk ;
    input Rst ;
    input Data_valid ;
    input [37:0]Data_in ;
    input [2:0]config ;
    output [5:0]Crc_out ;
    output Crc_valid ;

    wire count_30, count_29, count_28, count_27, count_26, count_25, count_24,
         count_23, count_22, count_21, count_20, count_19, count_18, count_17,
         count_16, count_15, count_14, count_13, count_12, count_11, count_10,
         count_9, count_8, count_7, count_6, count_5, count_4, count_3, count_2,
         count_1, count_0, r_30, r_29, r_28, r_27, r_26, r_25, r_24, r_23, r_22,
         r_21, r_20, r_19, r_18, r_17, r_16, r_15, r_14, r_13, r_12, r_11, r_10,
         r_9, r_8, r_7, r_6, r_5, r_4, r_3, r_2, r_1, count_inc_9_carry_31,
         count_inc_9_carry_30, count_inc_9_carry_29, count_inc_9_carry_28,
         count_inc_9_carry_27, count_inc_9_carry_26, count_inc_9_carry_25,
         count_inc_9_carry_24, count_inc_9_carry_23, count_inc_9_carry_22,
         count_inc_9_carry_21, count_inc_9_carry_20, count_inc_9_carry_19,
         count_inc_9_carry_18, count_inc_9_carry_17, count_inc_9_carry_16,
         count_inc_9_carry_15, count_inc_9_carry_14, count_inc_9_carry_13,
         count_inc_9_carry_12, count_inc_9_carry_11, count_inc_9_carry_10,
         count_inc_9_carry_9, count_inc_9_carry_8, count_inc_9_carry_7,
         count_inc_9_carry_6, count_inc_9_carry_5, count_inc_9_carry_4,
         count_inc_9_carry_3, count_inc_9_carry_2, C0, nx4, nx16, nx24, nx34,
         nx36, nx46, nx2427, count_31, nx2428, nx2430, nx50, nx58, nx70, nx72,
         nx80, nx90, nx102, nx106, nx108, nx116, nx130, nx138, nx154, nx162,
         nx170, nx190, nx198, nx206, nx224, nx232, nx240, nx262, nx270, nx278,
         nx296, nx304, nx312, nx332, nx340, nx348, nx366, nx374, nx382, nx404,
         nx412, nx424, nx430, nx442, nx454, nx460, nx468, nx482, nx486, nx502,
         nx526, nx530, nx538, nx558, nx572, nx580, nx584, nx588, nx596, nx604,

         nx608, nx624, nx634, nx646, nx654, nx700, nx770, nx774, nx780, nx800,
         nx810, C4, nx832, nx838, nx848, nx872, nx876, nx884, nx892, nx906,
         nx908, nx916, nx924, nx942, nx950, nx958, nx970, nx974, nx994, nx2439,
         nx2442, nx2449, nx2452, nx2455, nx2461, nx2463, nx2465, nx2467, nx2471,
         nx2473, nx2475, nx2477, nx2479, nx2483, nx2488, nx2493, nx2497, nx2520,
         nx2540, nx2560, nx2580, nx2583, nx2585, nx2587, nx2591, nx2593, nx2595,
         nx2597, nx2600, nx2602, nx2604, nx2607, nx2609, nx2615, nx2621, nx2624,
         nx2627, nx2629, nx2632, nx2638, nx2644, nx2646, nx2649, nx2654, nx2656,
         nx2658, nx2662, nx2664, nx2666, nx2668, nx2670, nx2675, nx2678, nx2681,
         nx2686, nx2689, nx2690, nx2697, nx2699, nx2707, nx2709, nx2714, nx2715,
         nx2716, nx2717, nx2721, nx2730;
    wire [30:0] \$dummy ;



    HA1A0 count_inc_9_D0_I0_dup_135 (.S (r_1), .CO (count_inc_9_carry_2), .A (
          count_1), .B (count_0)) ;
    HA1A0 count_inc_9_D0_I0_dup_136 (.S (r_2), .CO (count_inc_9_carry_3), .A (
          count_2), .B (count_inc_9_carry_2)) ;
    HA1A0 count_inc_9_D0_I0_dup_137 (.S (r_3), .CO (count_inc_9_carry_4), .A (
          count_3), .B (count_inc_9_carry_3)) ;
    HA1A0 count_inc_9_D0_I0_dup_138 (.S (r_4), .CO (count_inc_9_carry_5), .A (
          count_4), .B (count_inc_9_carry_4)) ;
    HA1A0 count_inc_9_D0_I0_dup_139 (.S (r_5), .CO (count_inc_9_carry_6), .A (
          count_5), .B (count_inc_9_carry_5)) ;
    HA1A0 count_inc_9_D0_I0_dup_140 (.S (r_6), .CO (count_inc_9_carry_7), .A (
          count_6), .B (count_inc_9_carry_6)) ;
    HA1A0 count_inc_9_D0_I0_dup_141 (.S (r_7), .CO (count_inc_9_carry_8), .A (
          count_7), .B (count_inc_9_carry_7)) ;
    HA1A0 count_inc_9_D0_I0_dup_142 (.S (r_8), .CO (count_inc_9_carry_9), .A (
          count_8), .B (count_inc_9_carry_8)) ;
    HA1A0 count_inc_9_D0_I0_dup_143 (.S (r_9), .CO (count_inc_9_carry_10), .A (
          count_9), .B (count_inc_9_carry_9)) ;
    HA1A0 count_inc_9_D0_I0_dup_144 (.S (r_10), .CO (count_inc_9_carry_11), .A (
          count_10), .B (count_inc_9_carry_10)) ;
```

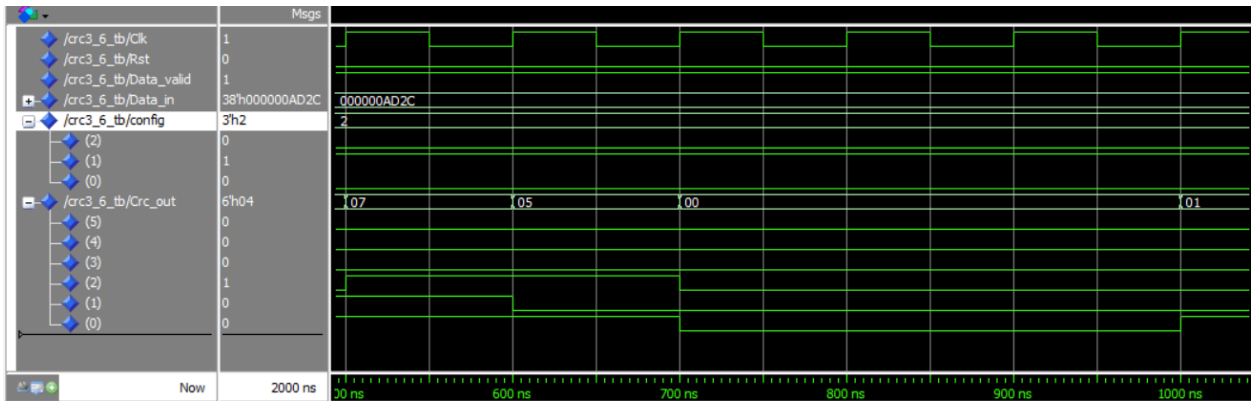
```
    AO6L0 ix2708 (.X (nx2707), .A1 (nx468), .A2 (nx2709), .B1 (nx502), .B2 (
          nx892), .C1 (nx486), .C2 (nx2621)) ;
    NR2R0 ix893 (.X (nx892), .A1 (config[1]), .A2 (nx2461)) ;
    OA1R0 ix811 (.X (nx810), .A1 (Data_valid), .A2 (Rst), .B (nx34)) ;
    AO1I0 ix2718 (.X (nx2717), .A1 (nx2675), .A2 (nx2587), .B (Rst)) ;
    NR2R0 ix985 (.X (Crc_out[2]), .A1 (nx2689), .A2 (nx2473)) ;
    NR2R0 ix987 (.X (Crc_out[3]), .A1 (nx2714), .A2 (nx2721)) ;
    OAI1A1 ix2722 (.X (nx2721), .A1 (config[0]), .A2 (config[1]), .B (config[2])
           ) ;
    NR2R0 ix989 (.X (Crc_out[4]), .A1 (nx2715), .A2 (nx2721)) ;
    NR2R0 ix991 (.X (Crc_out[5]), .A1 (nx2716), .A2 (nx2721)) ;
    IV1N0 ix585 (.X (nx584), .A (nx2649)) ;
    IV1N0 ix581 (.X (nx580), .A (nx2717)) ;
    IV1N0 ix2700 (.X (nx2699), .A (nx502)) ;
    IV1N0 ix487 (.X (nx486), .A (nx2591)) ;
    IV1N0 ix2676 (.X (nx2675), .A (nx468)) ;
    IV1N0 ix2584 (.X (nx2583), .A (nx460)) ;
    IV1N0 ix2601 (.X (nx2600), .A (nx106)) ;
    IV1N2 ix537 (.X (nx2430), .A (nx2439)) ;
    IV1N0 ix2478 (.X (nx2477), .A (nx2428)) ;
    IV1N0 ix47 (.X (nx46), .A (nx2463)) ;
    IV1N0 ix37 (.X (nx36), .A (nx2473)) ;
    IV1N0 ix35 (.X (nx34), .A (nx2721)) ;
    IV1N0 ix2596 (.X (nx2595), .A (nx16)) ;
    IV1N0 ix2710 (.X (nx2709), .A (nx4)) ;
    AN1V1 ix2729 (.X (nx2730), .A (nx2427)) ;
    AN2T0 ix71 (.X (nx70), .A1 (nx2607), .A2 (nx2488)) ;
    AN2T0 ix455 (.X (nx454), .A1 (nx2607), .A2 (nx2488)) ;
    MX2L0 ix2594 (.X (nx2593), .A (nx2607), .B (nx2488), .S (nx460)) ;
    XR2T0 ix2598 (.X (nx2597), .A1 (nx2580), .A2 (count_inc_9_carry_31)) ;
    AN2T0 ix425 (.X (nx424), .A1 (nx2493), .A2 (nx2607)) ;
    XN2R0 ix2628 (.X (nx2627), .A1 (nx2629), .A2 (nx2689)) ;
    OR2T0 ix103 (.X (nx102), .A1 (nx2609), .A2 (nx2604)) ;
    AN2T0 ix775 (.X (nx774), .A1 (nx2493), .A2 (count_4)) ;
    XN2R0 ix2682 (.X (nx2681), .A1 (nx2615), .A2 (nx2689)) ;
    XR2T0 ix959 (.X (nx958), .A1 (nx2629), .A2 (nx2716)) ;
    XR2T0 ix917 (.X (nx916), .A1 (nx2714), .A2 (nx2716)) ;
    XR2T0 ix885 (.X (nx884), .A1 (nx2689), .A2 (nx2716)) ;
endmodule
```

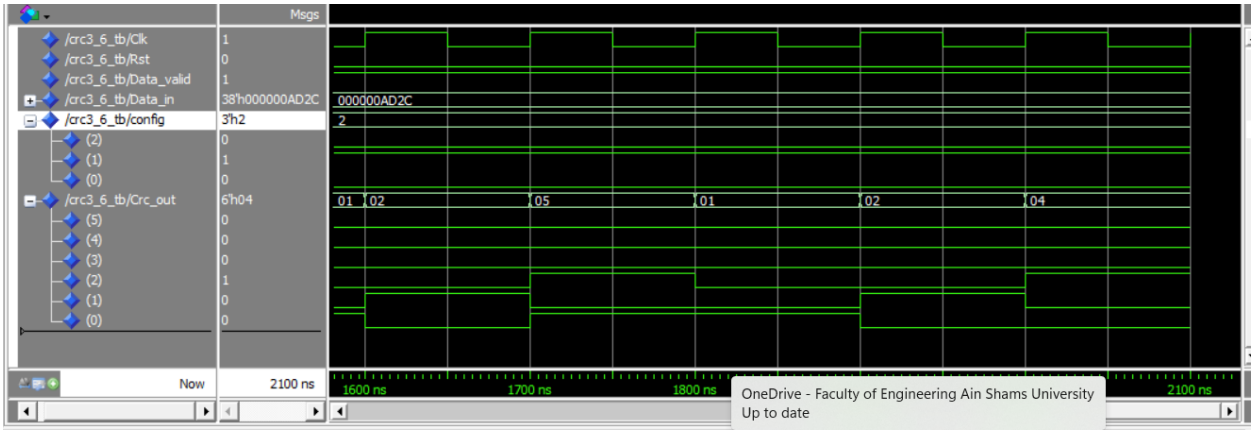
Simulation Waveforms:

This is the first test case I do it, it tests 3-bit CRC (16bits) Input data =AD2C

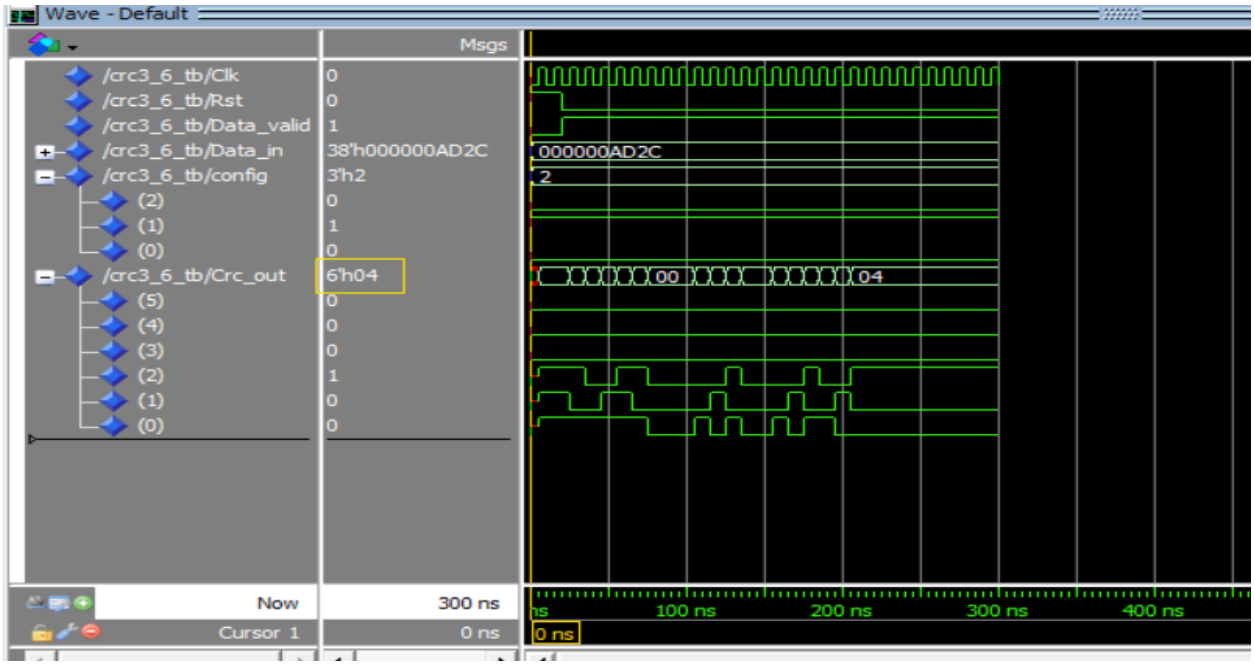
The expected output=4 and it gives valid output as we see in the next images.



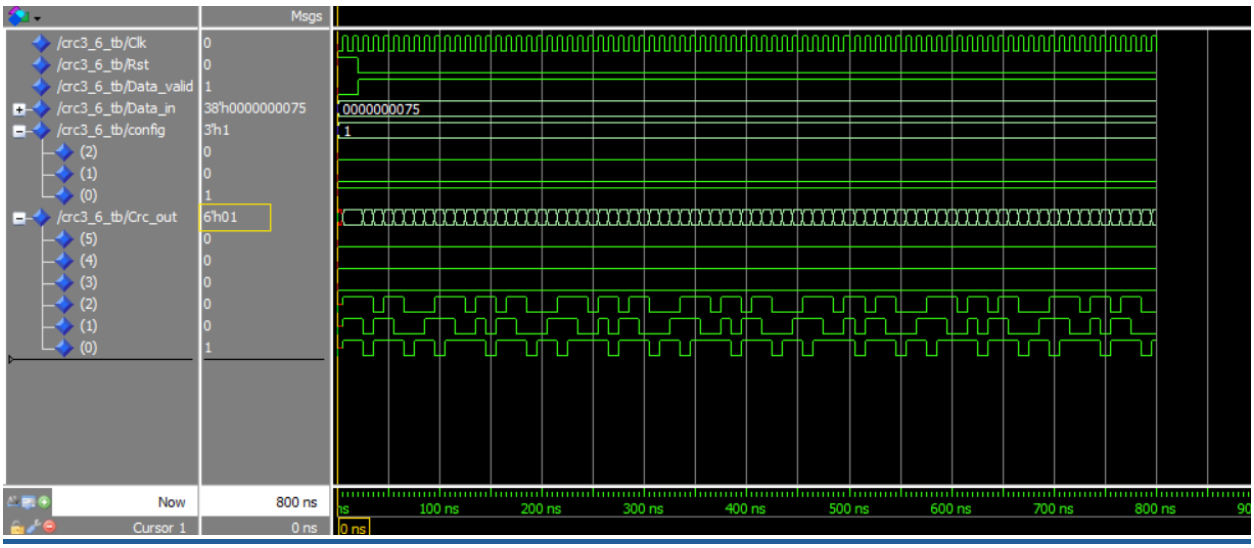




Same testcase but by testbench not Manuel it gives the expected output means that the testbench is valid and it is valid test case for 3-bit CRC (16bits)



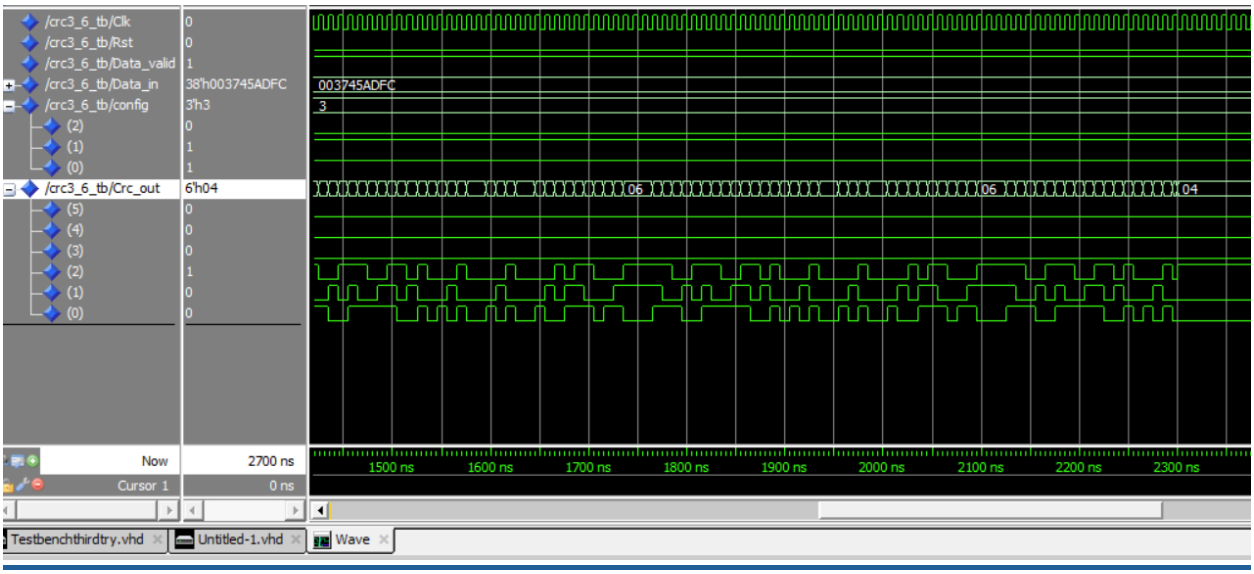
This the second test case by the test bench it tests 3-bits CRC (8bits) and it gives the expected output right means that the testbench is valid and it is valid test case for 3-bit CRC (8bits)



This the output of test case from CRC app which supports that test case in test bench is valid and right.

```
C:\Users\Dell\Downloads>crc.exe 0x75 8 0

3-bit CRC of [8 bits data] 0x75 = 0x1
```

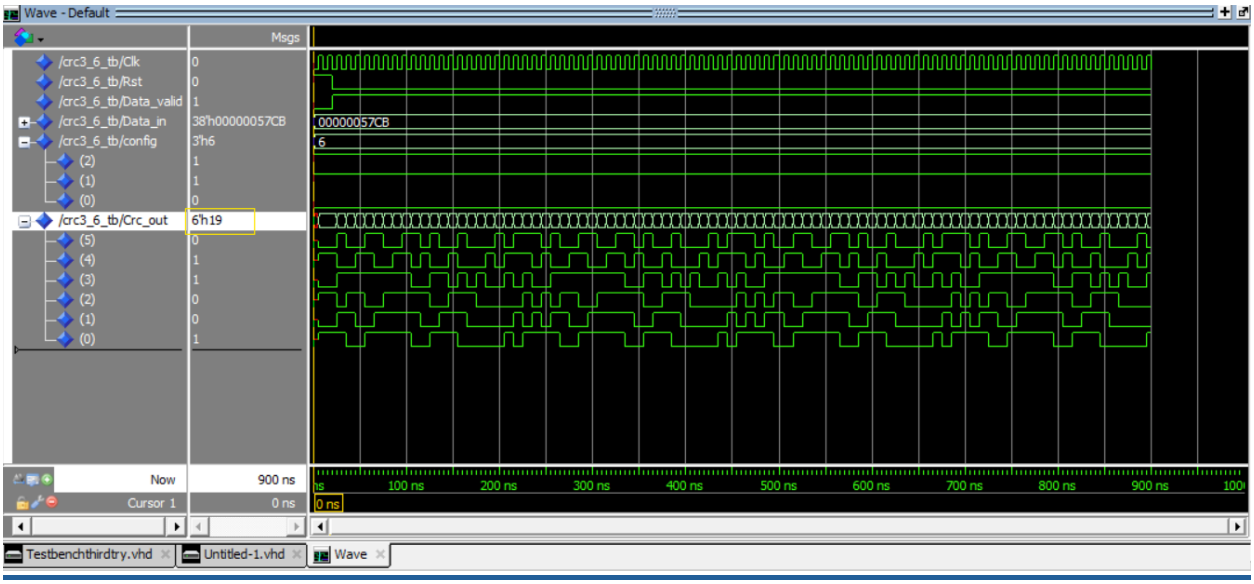
This the third test case by the test bench it tests 3-bits CRC (32bits) and it gives the expected output right means that the testbench is valid and it is valid test case for 3-bit CRC (32bits)



This the output of test case from CRC app which supports that test case in test bench is valid and right.

```
C:\Users\Dell\Downloads>crc.exe 0x3745adfc 32 0

3-bit CRC of [32 bits data] 0x3745adfc = 0x4
```

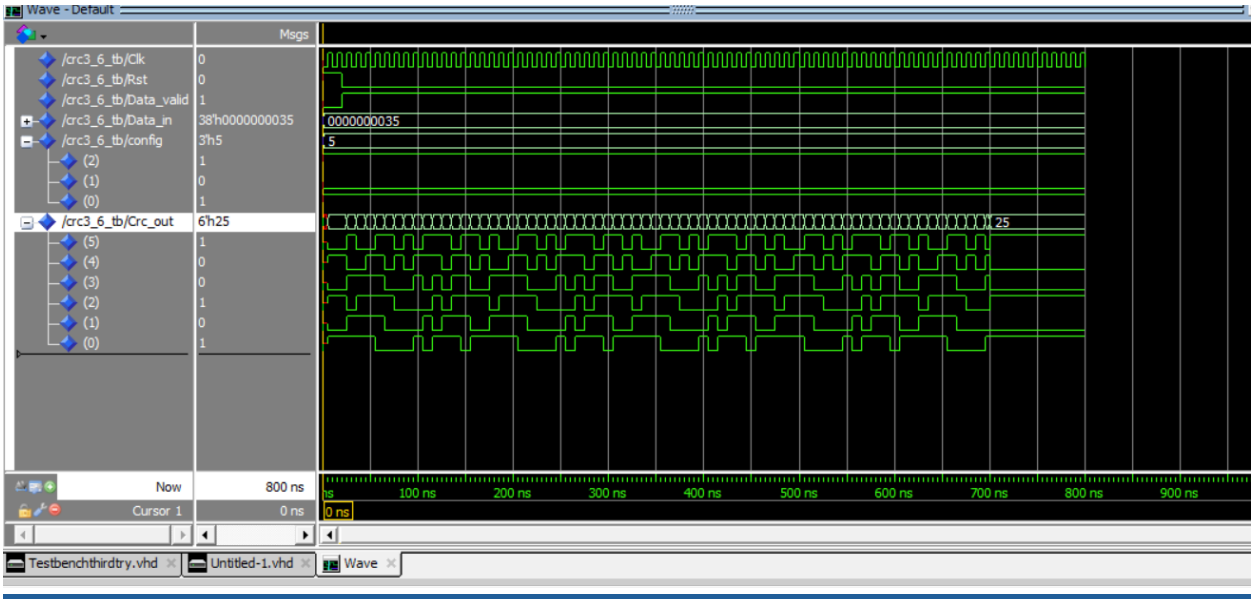
This the Fourth test case by the test bench it tests 6-bits CRC (16bits) and it gives the expected output right means that the testbench is valid and it is valid test case for 6-bit CRC (16bits)



This the output of test case from CRC app which supports that test case in test bench is valid and right.

```
C:\Users\Dell\Downloads>crc.exe ox57CB 16 1

6-bit CRC of [16 bits data] 0x0 = 0x19
```

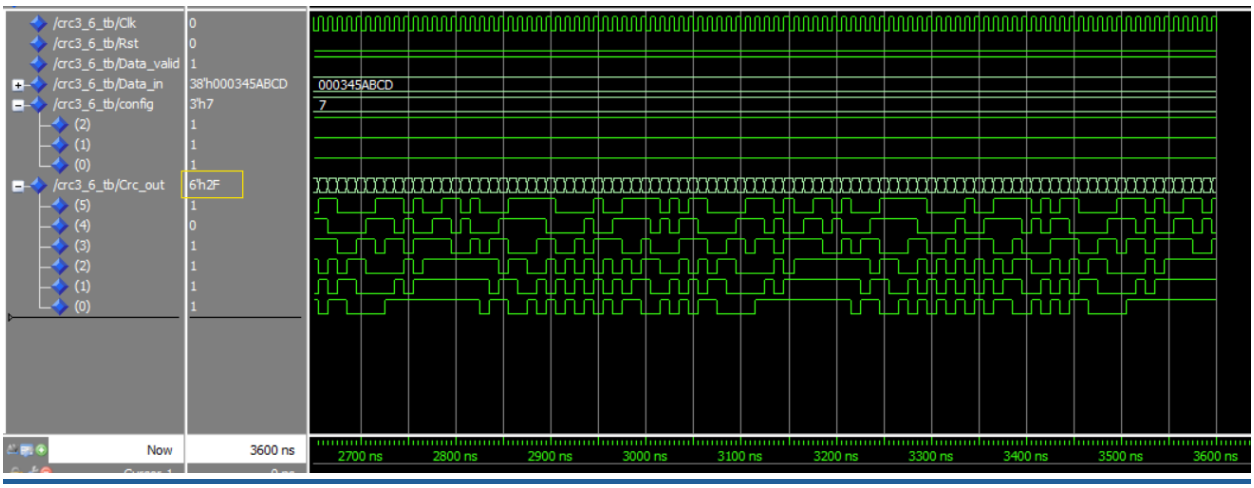
This the Fifth test case by the test bench it tests 6-bits CRC (8bits) and it gives the expected output right means that the testbench is valid and it is valid test case for 6-bit CRC (8bits)



This the output of test case from CRC app which supports that test case in test bench is valid and right.

```
C:\Users\Dell\Downloads>crc.exe 0x35 8 1

6-bit CRC of [8 bits data] 0x35 = 0x25
```

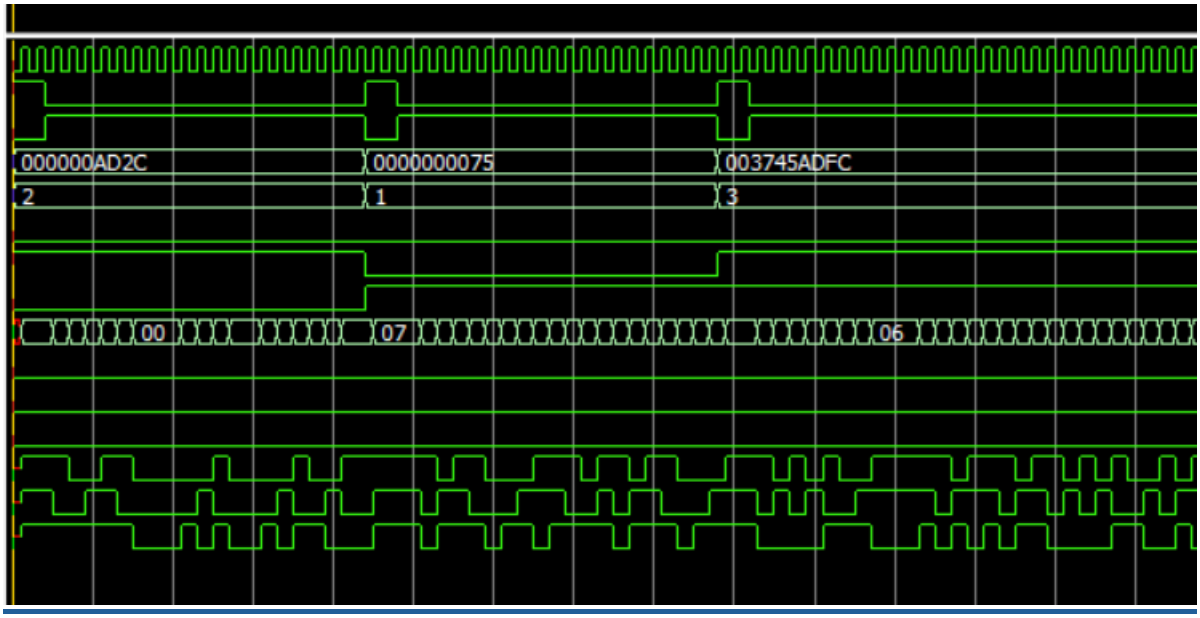
This the sixth test case by the test bench it tests 6-bits CRC (32bits) and it gives the expected output right means that the testbench is valid and it is valid test case for 6-bit CRC (32bits)



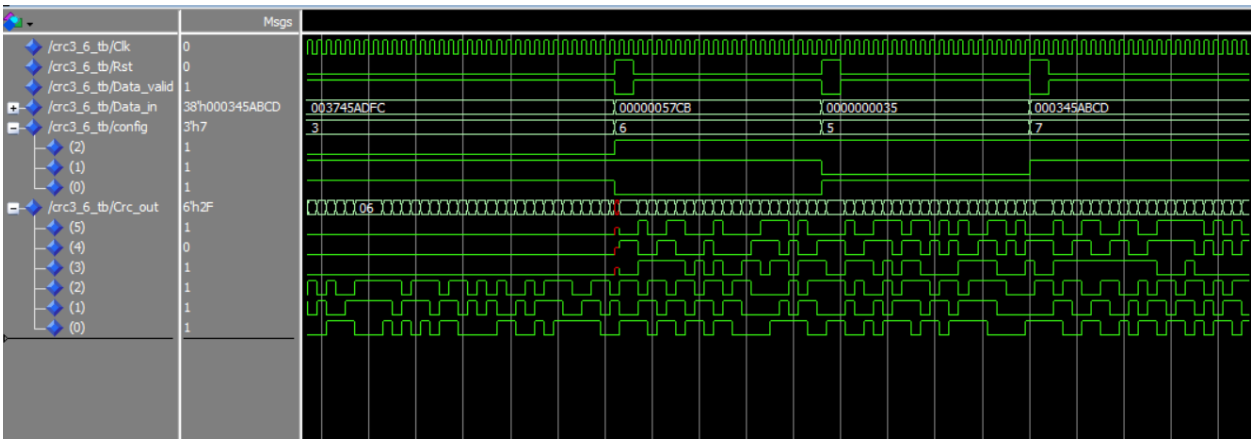
This the output of test case from CRC app which supports that test case in test bench is valid and right.

```
C:\Users\Dell\Downloads>crc.exe ox345abcd 32 1

6-bit CRC of [32 bits data] 0x0 = 0x2f
```

3 first test cases together



3 last test cases together



Challenges:

Configurable Initialization

Challenge:

- The need to initialize different CRC states based on the configuration (config) input. This involves setting initial values correctly for both 3-bit and 6-bit CRC calculations.

Implementation:

- The code handles this by using a case statement to set the initial values of C0, C1, C2, C3, C4, and C5 based on the config input.
- However, ensuring that these initializations occur correctly and that the design doesn't miss any specific cases can be complex.

Data Processing for Different Configurations

Challenge:

- Implementing the correct CRC computation logic for both 3-bit and 6-bit CRCs within the same module. Each type of CRC has different polynomial and data width requirements.
- Managing the count of processed bits correctly to ensure the entire input is processed as needed for different configurations.

Implementation:

- The code uses nested case statements to handle different CRC configurations and bit-count conditions.
- Ensuring that each bit of the input data is processed correctly, and the computation moves from one state to another seamlessly is critical and can be error prone.

Ensuring Timing and Synchronization

Challenge:

- Ensuring that the CRC computation is synchronized with the clock signal (Clk) and that the reset signal (Rst) correctly reinitializes the module.
- Proper handling of the Data_valid signal to start and stop the CRC computation process.

Implementation:

- The code checks for the rising edge of the clock and the status of the reset signal to manage the timing of the operations.