

HLS-Assignment 9 (PART-1)

July 10, 2023

Sampath Govardhan
FWC22071

VITIS-HLS

1 Problem Statement

Problem Statemt

Design a digital circuit using RTL and HLS for a 5G NR CRC bits generator.

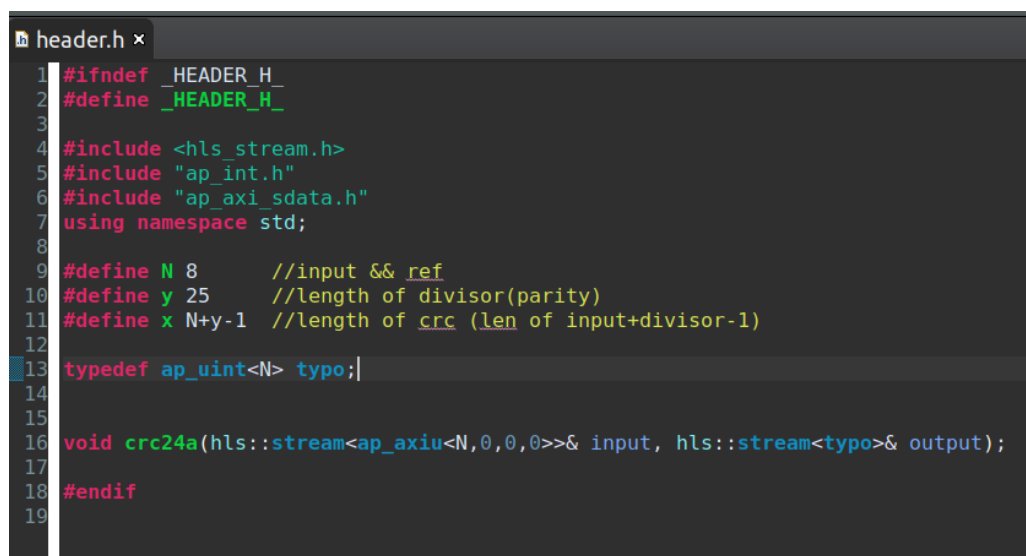
Requirements:

1. The module should process 8 bits per clock cycle.
2. The input and output buses should use the AXIS interface.
3. Implement a pipelined design.
4. The implementation should use as minimum resources as possible.

Considerations:

1. Implement the module only for CRC24A
2. A last signal should be used to indicate the end of the input bit stream.
3. The design implementation should be targeted on the ZCU111 FPGA board.

2 Header File



```
1 #ifndef HEADER_H
2 #define HEADER_H
3
4 #include <hls_stream.h>
5 #include "ap_int.h"
6 #include "ap_axi_sdata.h"
7 using namespace std;
8
9 #define N 8      //input && ref
10 #define y 25     //length of divisor(parity)
11 #define x N+y-1 //length of crc (len of input+divisor-1)
12
13 typedef ap_uint<N> typo;|
14
15
16 void crc24a(hls::stream<ap_axiu<N,0,0,0>>& input, hls::stream<typo>& output);
17
18 #endif
19
```

Figure 1: header.h

3 Crc bits Generator Code

```
crc.cpp x
1 #include "header.h"
2 void crc24a(hls::stream<ap_axiu<N,0,0,0>>& input, hls::stream<typo>& output) {
3
4 #pragma HLS INTERFACE mode=axis register_mode=both port=input register
5 #pragma HLS INTERFACE mode=axis register_mode=both port=output register
6
7     ap_uint<1> crc[x], oput[x];
8     ap_uint<1> divisor[y] = {1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1};
9     typo o;
10
11 // Read input stream and do padding
12     ap_axiu<N,0,0,0> d = input.read();
13     loop1: for (int i = 0; i < x; i++) {
14 #pragma HLS UNROLL
15         crc[i] = (i < N) ? d.data(i,i) : 0;
16         oput[i] = (i < N) ? d.data(i,i) : 0;
17     }
18
19 // Division is performed only when last is high
20     loop2: for (int i = 0; i <= x - y; i++) {
21 #pragma HLS PIPELINE II=1
22         if (crc[i] == 1 && d.last==1) {
23             loop3: for (int j = 0; j < y; j++) {
24                 int k=i+j;
25 #pragma HLS UNROLL
26                 crc[k] = crc[k] ^ divisor[j];
27             }
28         }
29     }
30
31 // Write the result to output stream c
32     loop4: for (int i = 0; i < x; i++) {
33 #pragma HLS UNROLL
34         oput[i] = crc[i] ^ oput[i];
35         o(i%N, i%N) = oput[i];
36         if (i%N==7){
37             output.write(o);
38         } }
39 }
```

Figure 2: crc.cpp

4 Design Choices

1. I Considered input message of length 8 and I designed the module accordingly.
2. The function takes two parameters: input and output.

input is an AXIS input stream of ap_axiu type with a width of 8 bits and with a last signal.

output is an AXIS output stream of ap_uint<8> type, since last signal is not requested.

3. The pragmas specify the interface properties for the function. mode=axis indicates that the function uses an AXI-Stream interface. register_mode=both specifies that both the input and output streams should be implemented using

registers.

4. divisor is an array of `ap_uint<1>` type with a size of 25(crc24a).

crc and oput are arrays of `ap_uint<1>` type with a size of $x(\text{input length} + \text{divisor length} - 1)$. crc array is used for crc computation and oput array is used to stream output data along with input message.

o is a variable of type `ap_uint<8>` which is used for storing of output data and then streaming accordingly.

d is a variable of type `ap_uint<8>` which reads input data. Since I had only 8 bits input data I read it only once.

5. loop1 initializes values to the crc and oput arrays based on the data present in d(from the input stream). It copies the i-th bit of d.data to `crc[i]` and `oput[i]` and it assigns all other indexes to zero.

The loop is unrolled to improve the efficiency of copying bits from d.data to crc and oput arrays. Unrolling eliminates loop control overhead and allows multiple assignments to be performed in parallel.

6. loop2 and loop3 performs the division operation for the CRC algorithm. It checks if the i-th bit of crc is 1 and if d.last is also 1 (indicating it is the last element of the input stream). If both conditions are satisfied, it performs the XOR operation between `crc[k]` and `divisor[j]` and stores the result in `crc[k]`, where k iterates from i to $i+y-1$. At the end of loop3 crc array is loaded with crc computed values(remainder) of input message and divisor.

For the loop2 the `#pragma HLS PIPELINE II=1` directive is used to pipeline the loop iterations. By applying pipelining, the tool can overlap the execution of consecutive iterations, allowing the operations within the loop to be executed concurrently. This can result in increased throughput.

Additionally, the `#pragma HLS UNROLL` directive is used for loop3. It instructs the tool to unroll the inner loop completely and multiple loop iterations are executed in parallel, which can further improve parallelism and increase performance.

7. loop4 performs the final XOR operation between crc(has crc computation output) and oput(has input message) arrays(since we need the output in form of input+crcoutput) and updates the o object with the XOR result.

The line `o(i%N, i%N) = oput[i];` assigns the i-th bit of oput to the corresponding position in o. Finally, if the current index i is a multiple of 8 (i.e., $i\%N == 7$), it writes the o object to the output stream.

This loop performs the final XOR operation between crc and oput arrays and updates the variable o. Unrolling the loop helps increase parallelism and pipeline the operations for better performance.

8. Overall, this code reads an input stream of AXI-Stream elements and performs the CRC-24 algorithm on the data, and writes the result to the output stream.

5 Testbench Code

```
crc_tb.cpp x
1 #include "header.h"
2 #include <vector>
3
4 int main() {
5     hls::stream<ap_axiu<N,0,0,0>> a;
6     hls::stream<typo> b;
7     ap_axiu<N,0,0,0> w;
8
9     w.data=0b00010110;           //msbto1sb
10
11     w.last=1;
12     a.write(w);
13
14 // Perform binary division
15     crc24a(a, b);
16
17 // Read the result from the output stream
18     vector<ap_uint<1>> p;
19     cout << "CRC generator output : ";
20     while(!b.empty()){
21         typo d = b.read();
22         for (int i = 0; i < N ; i++) {
23             cout<< d(i,i);
24             p.push_back(d(i,i));
25         }
26     }
27     cout<<endl;
28
29 // Checking if output is valid or not
30     bool flag=0;
31     ap_uint<1> divisor[y] = {1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1};
32
33
34     //Output is valid only when remainder division of output with divisor is 0
35     for (int i = 0; i <= x - y; i++) {
36         if (p[i] == 1) {
37             for (int j = 0; j < y; j++) {
38                 p[i + j] = p[i+j] ^ divisor[j];
39             }
40         }
41     }
42     cout<<"CRC detector output : ";
43     for (int i = 0; i < 32; i++) {
44         cout<<p[i];
45         if (p[i]==1){
46             flag=1;
47         }
48     }
49     cout<<endl;
50     if ( flag==0) {
51         cout << "!PASS!CRC Check at detector is Success" << endl;
52     }
53     else {
54         cout << "ERROR!CRC Check at detector has Failed" << endl;
55     }
56     return 0;
57 }
58
```

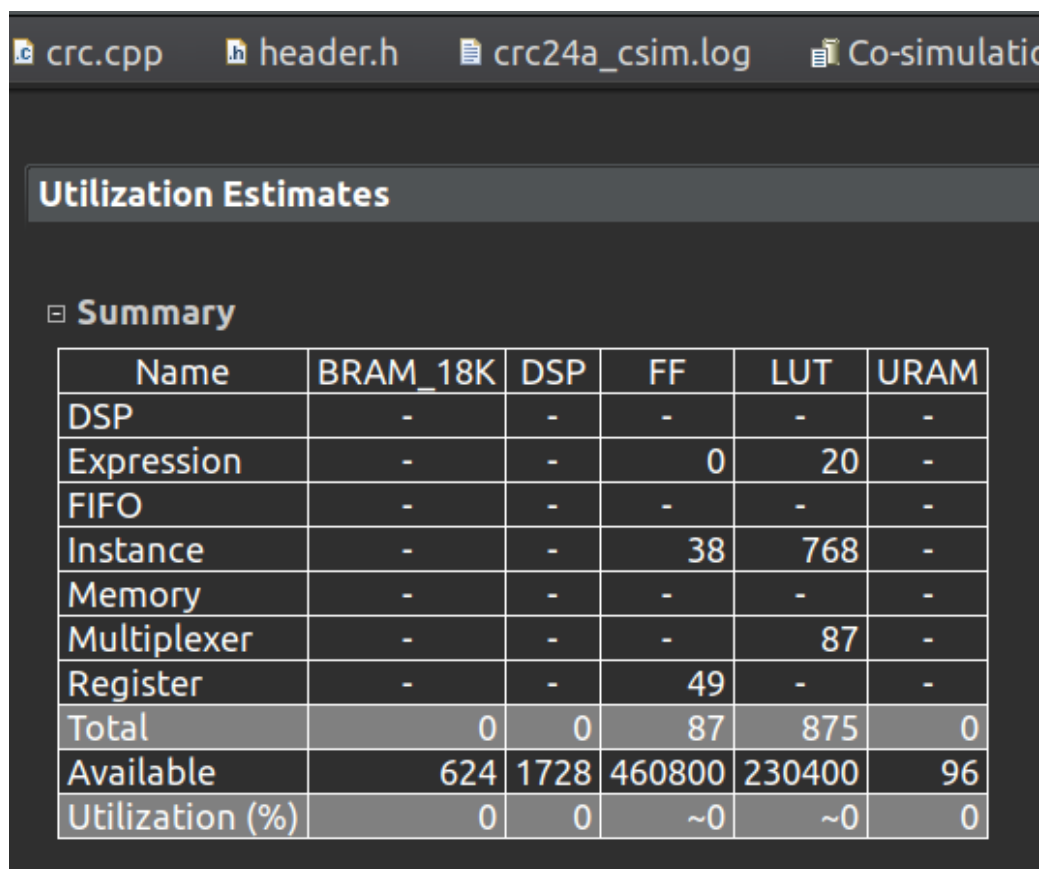
(a) crc_tb.cpp

6 C simulation Output

```
crc24a_csim.log x
1 INFO: [SIM 2] ***** CSIM start *****
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3   Compiling ../../../../codes/crc_tb.cpp in debug mode
4   Compiling ../../../../codes/crc.cpp in debug mode
5   Generating csim.exe
6 CRC generator output : 01101000101101001111001100000010
7 CRC detector output : 00000000000000000000000000000000
8 !PASS!CRC Check at detector is Success
9 INFO [HLS SIM]: The maximum depth reached by any hls::stream() instance in the design is 4
10 INFO: [SIM 1] CSim done with 0 errors.
11 INFO: [SIM 3] ***** CSIM finish *****
12
```

Figure 4: C Simulation Output

7 HLS Resource Consumption Report



The screenshot shows the Co-simulation window with tabs for crc.cpp, header.h, crc24a_csim.log, and Co-simulation. The Utilization Estimates section is expanded, showing a summary table of resource consumption.

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	20	-
FIFO	-	-	-	-	-
Instance	-	-	38	768	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	87	-
Register	-	-	49	-	-
Total	0	0	87	875	0
Available	624	1728	460800	230400	96
Utilization (%)	0	0	~0	~0	0

Figure 5: Resource Consumption

8 HLS Timing and Fmax Report

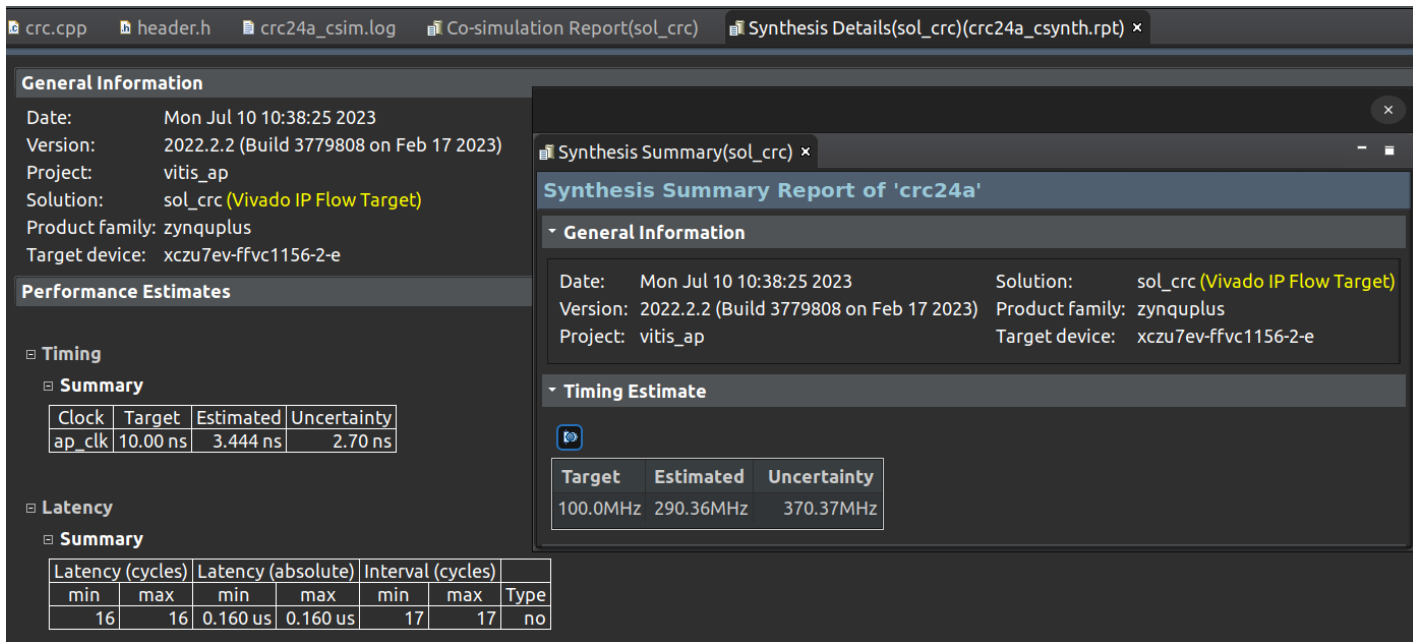


Figure 6: Timing and Fmax

9 CoSimulation Report

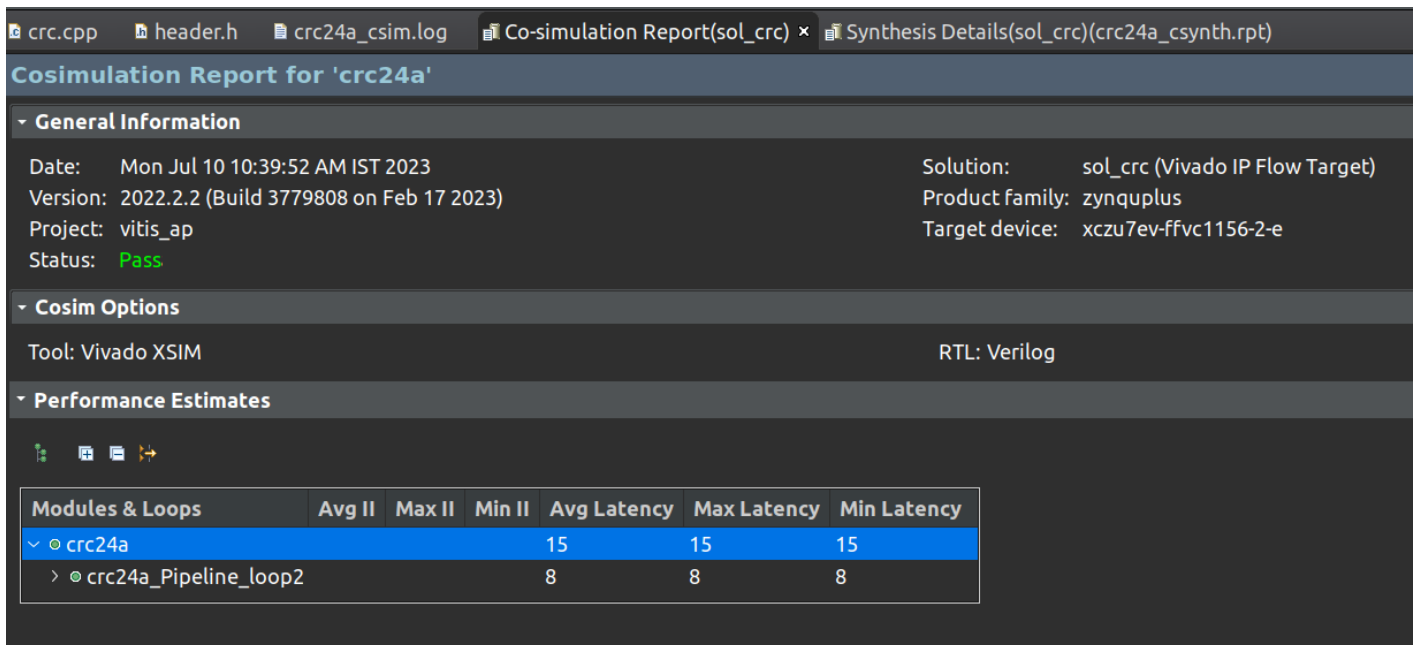


Figure 7: Cosimulation

10 Performance Evaluation

After synthesis source code has :

1. Execution Time: 12.65 seconds
2. Latency: 16
3. Resource Utilization: 87 FF & 875 LUT
4. Memory Usage: 76.434 MB
5. Fmax: 290.36 MHz
6. Iteration Interval: 17
7. Throughput: 0.632

VIVADO

11 Block Design

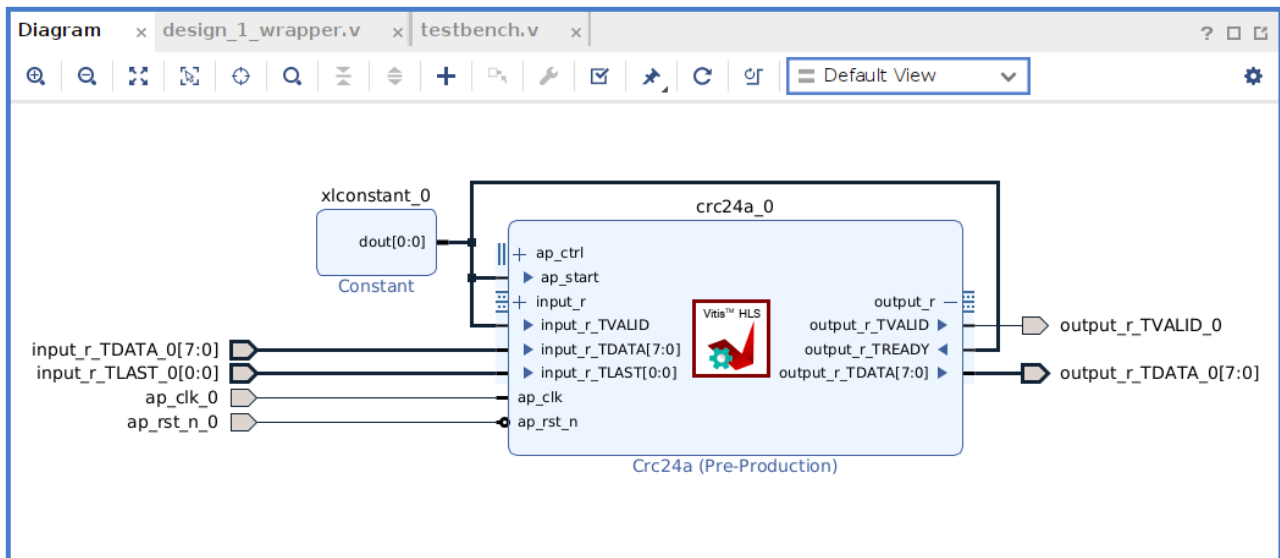


Figure 8: Block Diagram

12 Verilog Testbench

```
//testbench.v
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 06/26/2023 11:35:30 AM
// Design Name:
// Module Name: testbench
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 -- File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////

module testbench();
    reg ap_clk_0;
    reg ap_rst_n_0;
    reg [7:0] ip;
    reg input_r_TLAST_0;
    wire [7:0] op;
    wire output_r_TVALID_0;

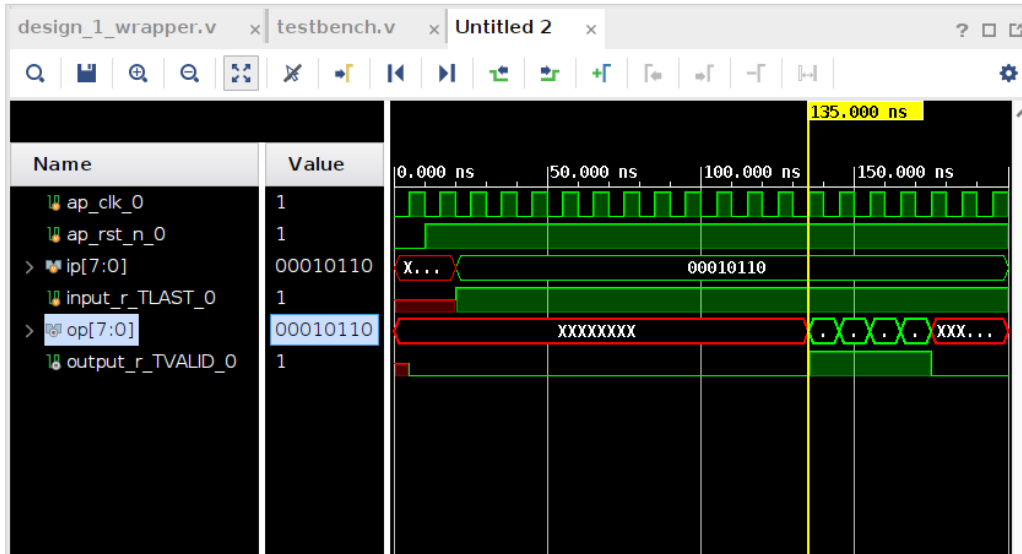
    design_1_wrapper uut(.ap_clk_0(ap_clk_0),.input_r_TLAST_0(input_r_TLAST_0),
        .ap_rst_n_0(ap_rst_n_0),.input_r_TDATA_0(ip),.output_r_TDATA_0(op),
        .output_r_TVALID_0(output_r_TVALID_0));

    always #5 ap_clk_0=~ap_clk_0;

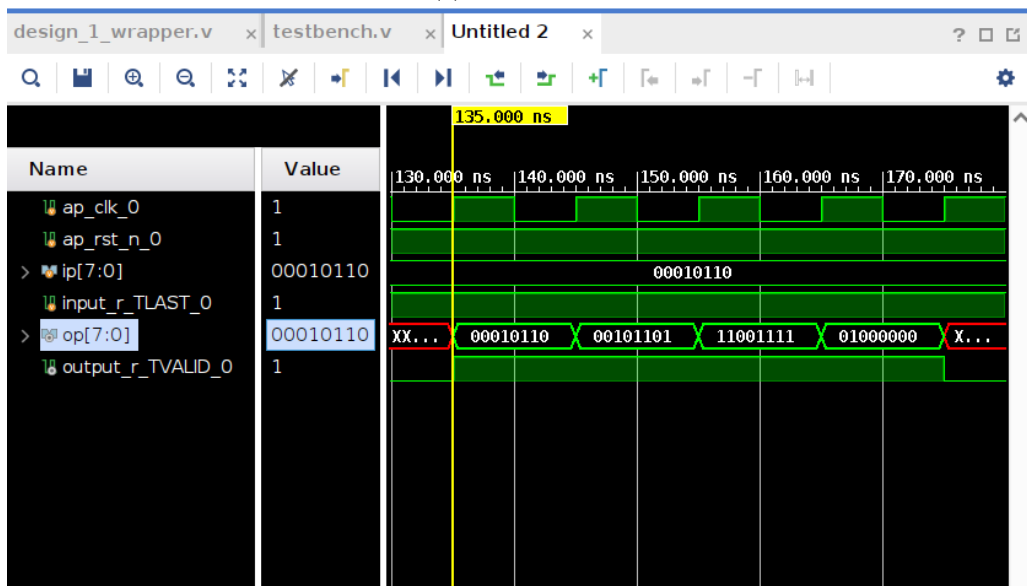
    initial begin
        ap_clk_0=0;ap_rst_n_0=0;
        #10
        ap_rst_n_0=1;
        #10
        ip=8'b00010110;// ascii "h"
        input_r_TLAST_0=1;
        #180
        $finish;
    end

endmodule
```


13 Output Waveform



(a) Output of IP



(b) Zoomed format of above figure