International Institute of Information Technology, Bangalore

May 2024

A PROJECT REPORT ON

Coverage Driven Verification (CDV) of Elliptic Curve Cryptography (ECC) Crypto Processor using QuestaSim

Guided by

“Prof. Subir Kumar Roy”

Submitted by

MT2023**533** Sushma R
MT2023**508** Alwin Shaju

Coverage Driven Verification (CDV) of Elliptic Curve Cryptography (ECC) Crypto Processor

Introduction and Specifications

Introduction:

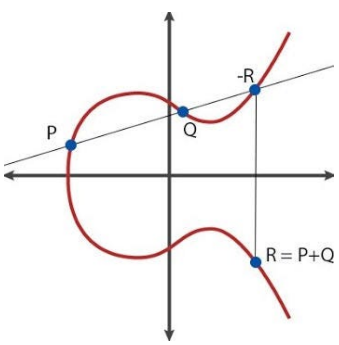
1. Overall Functionality: To produce the output public key coordinates xq and yq, the Montgomery Algorithm is utilized by the ECC_all_Parts (also known as the ECC Crypto Processor) module to perform the necessary calculations and operations. To accomplish the required functionality, it manages clock synchronization, initialization, conversion from affine to projective coordinates, multiplications, additions, and inversions.

2. Purpose: The Montgomery Algorithm serves as the foundation for Elliptic Curve Cryptography (ECC), which is the method used by the ECC Crypto Processor to create public keys.

3. Key Generation: The GRAIN-128 technique is used by the ECC Crypto Processor to produce a private key. This private key is an essential input for the processes that follow.

4. Algorithm: The Montgomery Algorithm, a mathematical formula created for Elliptic Curve Cryptography, is what the CPU uses. The public key computation process is made secure and efficient by this approach.Here they do Multiplication,addition and inversion.

5. Public Key Generation: By performing scalar multiplication on the input base point and the private key produced by the GRAIN-128 algorithm, the processor creates a public key with a length of 163 bits.



Note:

Verifying SHA-256 module, GRAIN–128 module is not part of this project because this comes under digital signature which will make this ECDSA while the project is on doing coverage driven verification on ECC processor.

Specifications:

1. ECC Crypto Processor is defined with the following input and output ports:
    a. Inputs: clk, rst, a, b, k, xp, yp
    b. Outputs: xq, yq
2. Input and Output Sizes: The module expects input signals a, b, k, xp, yp, and produces output signals xq, yq, all of size 163 bits. **i.e. Key length support 163 bit.**
3. Registers and Wires:
    Registers: X1, X2, Z1, Z2, and count are defined as registers to store intermediate values.
    Wires: V1, V2, V3, R1, R2, R3, temp_X1, temp_X2, temp_Z1, temp_Z2, t_Z2, init_X2, and init_Z2 are defined as wires to facilitate internal signal connections and computations.

4. Initialization: Upon a positive edge of the clock signal, the module performs the initialization process based on the rst (reset) input. If rst is asserted, the count is set to 0, and the initial values of X1, X2, Z1, and Z2 are set to xp, init_X2, 163'b1, and init_Z2, respectively.

    Montgomery Algorithm Implementation:
    Multiplier Modules: The module instantiates multiple instances of the multiplier_163b module to perform various multiplications required by the Montgomery Algorithm. These modules include m13, m14, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, and m12.
    CS Adder Modules: The module uses instances of the cs_adder_163b module, namely cs4, cs1, cs2, cs3, cs5, cs6, cs7, cs8, and cs9, to perform addition operations in the Montgomery Algorithm

5. Affine to Projective Coordinates Conversion: The module implements the conversion from affine coordinates to projective coordinates. This conversion is performed based on the values of k and count. Depending on the value of k[count], the module updates the values of X1, X2, Z1, and Z2 accordingly.
6. Inversion Operation: The module utilizes instances of the inversion_op module, namely i1, i2, and i3, to compute the inverses of Z1, Z2, and xp, respectively.
7. Output Generation: The module assigns the values of R1 and R2 to the output ports xq and yq, respectively, using the assign statement

Algorithm 1: Montgomery Point Multiplication

Input: $k = (k_{n-1}, ..., k_2, k_1, k_0)$ where $k_{n-1} = 1$
$P = (x_p, y_p) \in GF(2^m)$
Output: $Q = (x_q, y_q) = k \cdot P$
Initialize: $X_1 = x_p$, $Z_1 = 1$, $Z_2 = x_p^2$, $X_2 = x_p^4 + b$;
for *(i from* $m-2$ *down to* 0*)* **do**
  if $(k_i = 1)$ **then**
    $Z_1 = X_2 \times Z_1$, $X_1 = X_1 \times Z_2$
    $T_1 = X_1 + Z_1$, $X_1 = X_1 \times Z_1$
    $Z_1 = T_1^2$, $T_1 = X_p \times Z_1$
    $X_1 = X_1 + T_1$, $Z_2 = Z_2^2$
    $T_1 = Z_2^2$, $T_1 = b \times T_1$
    $X_2 = X_2^2$, $Z_2 = X_2 \times Z_2$
    $X_2 = X_2^2$, $X_2 = X_2 + T_1$
  end
  else
    $Z_2 = X_1 \times Z_2$, $X_2 = X_2 \times Z_1$
    $T_1 = X_2 + Z_2$, $X_2 = X_2 \times Z_2$;
    $Z_2 = T_1^2$, $T_1 = X_p \times Z_2$;
    $X_2 = X_2 + T_1$, $Z_1 = Z_1^2$;
    $T_1 = Z_1^2$, $T_1 = b \times T_1$;
    $X_1 = X_1^2$, $Z_1 = X_1 \times Z_1$;
    $X_1 = X_1^2$, $X_1 = X_1 + T_1$;
  end
end
$X_q = \frac{X_1}{Z_1}$;
$Y_q = X_p + (\frac{X_1}{Z_1})[(X_1 + x_p \times Z_1)(X_2 + x_p \times Z_2) + (x_p^2 + y_p)(Z_1 \times Z_2)] \times (x_p \times Z_1 \times Z_2)^{-1} + y_p$

Analysis of the design hierarchy in Verilog

ECC_Crypto_Processor :

ECC_all_Parts{x1}

- **cs_adder_163b{x9}**
- **multiplier_163b{x20}**
  - **red{x1}**
  - **ks163{x1}**

    Overlap_243bit{x1}

    ks81{x6}

    Overlap_81bit{x1}

    ks27{x6}

    Overlap_27bit{x1}

    ks9{x6}

    Overlap_9bit{x1}

    ks3{x6}
- **inversion_op{x3}**
  - **circ_leftshift_163b{x1}**
  - **multiplier_163b{x2}**

    red{x1}

    ks163{x1}

    Overlap_243bit{x1}

    ks81{x6}

    Overlap_81bit{x1}

    ks27{x6}

    Overlap_27bit{x1}

    ks9{x6}

    Overlap_9bit{x1}

    ks3{x6}

The above module hierarchy represents a design structure for an ECC (Elliptic Curve Cryptography) Crypto Processor: {xN} represents the number of instances present for each module. It is written in that format to reduce the reading complexity.

ECC_Crypto_Processor:

ECC_all_Parts: This is the top-level module or entity representing the ECC Crypto Processor. This module contains multiple sub-modules or instances of other modules. It is represented by the label "x1" indicating that there is one instance of this module.

- cs_adder_163b: This module represents an adder with a width of 163 bits.
- multiplier_163b: This module represents a multiplier with a width of 163 bits.
- red: This module performs reduction operations.
- ks163: This module represents a Karatsuba multiplier with a width of 163 bits.
- overlap_243:This module represents an overlap module which is used to remove the critical path from the Karatsuba multiplier with a width of 243.
- ks81: This module represents a Karatsuba multiplier with a width of 81 bits.
- overlap_81:This module represents an overlap module which is used to remove the critical path from the Karatsuba multiplier with a width of 81.
- ks27: This module represents a Karatsuba multiplier with a width of 27 bits.
- overlap_27:This module represents an overlap module which is used to remove the critical path from the Karatsuba multiplier with a width of 27.
- ks9: This module represents a Karatsuba multiplier with a width of 9 bits.
- overlap_9:This module represents an overlap module which is used to remove the critical path from the Karatsuba multiplier with a width of 9 bits.
- ks3: This module represents a Karatsuba multiplier with a width of 3 bits.
- inversion_op: This module represents an inversion operation.
- circ_leftshift_163b: This module represents a circular left shift operation with a width of 163 bits.

Verification plan for CDV/simulation of ECC crypto Processor

1.Description of Verification Levels:

For our project there are many module hierarchies which are present so to verify it we are individually first checking the modules then doing the functionality check of the entire design.

ECC_Crypto_Processor :

ECC_all_Parts{x1}

- **cs_adder_163b{x9}**
- **multiplier_163b{x20}**
  - **red{x1}**
  - **ks163{x1}**

    Overlap_243bit{x1}

    ks81{x6}

    Overlap_81bit{x1}

    ks27{x6}

    Overlap_27bit{x1}

    ks9{x6}

    Overlap_9bit{x1}

    ks3{x6}

- **inversion_op{x3}**
  - **circ_leftshift_163b{x1}**
  - **multiplier_163b{x2}**

    red{x1}

    ks163{x1}

    Overlap_243bit{x1}

    ks81{x6}

    Overlap_81bit{x1}

    ks27{x6}

    Overlap_27bit{x1}

    ks9{x6}

    Overlap_9bit{x1}

    ks3{x6}

The above module hierarchy represents a design structure for an ECC (Elliptic Curve Cryptography) Crypto Processor:

{xN} represents number of instances present for each module. It is written in that format to reduce the reading complexity.

2.Required Tools:

1. Questasim
2. Vivado
3. VS Code

3.Functions to be verified:

- Reset functionality
- Clk functionality
- Module relations between each other
- GF(2) multiplication output
- Adder Functionality
- Inversion Functionality
- Reduction Functionality

4.Specific tests and methods:

- Directed Testbench to check the functionality of some modules.
- Layered Testbench with random inputs to check the corner cases for every module.
- So we have used White Box method as we already know what is happening inside each module .

5.Coverage Requirements:

- **Functional Coverage Goals:**
    - Make that the design function properly for every possible combination of input.
    - Check if certain functional blocks or algorithms in the design are

correct by directed testbenches.
    - Verify that crucial output signals or statuses are accurate,for eg:reset signals.

- **Code Coverage Goals:**
    - To guarantee that every line of code is tested, attain high statement coverage.
    - To ensure that conditional statement branches are covered and to strive for excellent branch coverage.
    - To test both true and false situations, strive for high toggle coverage.

- **Verification Strategies:**
    - **Functional Coverage:**Establish functional coverage points so that you can monitor the goals of coverage and modify test cases as necessary.

- **Coverage Metrics:**To make sure the verification tests cover all major issues of the design functionality, keep an eye for the functional coverage points so that we can check the bins that haven't been touched and what are the reasons for it.

6.Test Case Scenarios:

- Construct a stimulus generator that produces arbitrary inputs for the design and tracks the output signals to see if they correspond with the desired outcomes.
- Create test cases that address boundary conditions and edge cases for the input signals, including zero and the lowest and greatest values.

ECC_all_Parts

Input: clk , rst
Input:
[162:0]a,b,k,xp,yp
Output: [162:0]
xq,yq

S.No	Condition	Condition Explanation	Expected behaviour	Description of test scenario
1	rst=1	Reset is asserted high	Outputs xq,yq should result in xq = (X1 * V1) / Z1 yq = (X2 * V2) + yp + (V1 * V2) + (V1 * V2) Where: X1 <= xp Z1 <= 163'b1 X2 <= xp^4+b Z2 <= xp^2	This scenario describes the active high reset of the ECC Crypto processor. Validate the behavior of the design during reset conditions. Check if the reset signal resets the design to the expected initial state.
2	rst=0	Reset is not asserted high	Outputs xq,yq should be the corresponding outputs to the respected inputs (a,b,k,xp,yp)	This scenario describes proper functioning of ECC Crypto processor.
3	rst= 0 ⇢ 1	Reset is asserted high in middle of operation	Eventually in next clock cycle, Outputs xq,yq should result in xq = (X1 * V1) / Z1 yq = (X2 * V2) + yp + (V1 * V2) + (V1 * V2) Where: X1 <= xp Z1 <= 163'b1 X2 <= xp^4+b Z2 <= xp^2	This scenario describes how the ECC Crypto processor behaves when reset is asserted in the middle of an operation.
4	rst = 1 ⇢ 0	Reset is de-asserted high in middle of operation	Outputs xq,yq should be the corresponding outputs to the respected inputs (a,b,k,xp,yp)	This scenario describes how the ECC Crypto processor gets back to normal behavior when the reset signal is de-asserted.
5	Constraint : When rst = 1 is asserted we should not consider coverage for			

	inputs a,b,k,xp,yp
6	Consider cross coverage between a,b,k, xp,yp

7	Consider transition bins for a,b,k,xp,yp,xq,yq
8	Code coverage done using Questasim and verified the design

cs_adder_163b

Input: [162:0]adder_in1 ,
adder_in2 Output: [162:0]
adder_out

1	adder_in1=p adder_in2=q	Output adder_out should be the corresponding to the XOR operation respected inputs (p xor q)	This Scenario describes Proper functioning of **cs_adder_163b**.
2	Consider cross coverage between **adder_in1, adder_in2**		
3	Consider transition bins for **adder_in1, adder_in2,adder_out**		
4	Code coverage done using Questasim and verified the design		

inversion_op

input clk,reset
input [162:0] inv_inp,
output [162:0] inv_out

1	inv_inp	Output inv_out should be the corresponding to the input inv_inp inverse	This Scenario describes Proper functioning of inversion_op.
2	Consider transition bins for **inv_imp**		
3	Code coverage done using Questasim and verified the design		

circ_leftshift_163b

input [6:0] circ_shft_val,
input [162:0] circ_shft_inp,
output [162:0] circ_shft_out

1	circ_shft_val =p circ_shft_inp =q	Output circ_shft_out should be the (p<<q) or (p>>(163-q))	This Scenario describes Proper functioning of **circ_leftshift_163b**.
2	Consider cross coverage between **circ_shft_val, circ_shft_inp.**		
3	Consider transition bins for **circ_shft_val, circ_shft_inp,circ_shft_out**		
4	Code coverage done using Questasim and verified the design		

multiplier_163b

input
[162:0]mult_in1,m
ult_in2 output
[162:0] mult_out

1	mult_in1 mult_in2	Output mult_out should be the multiplication of inputs	This Scenario describes Proper functioning of **multiplier_163b**.
2	Consider cross coverage between **mult_in1, mult_in2.**		
3	Consider transition bins for **mult_in1, mult_in2,mult_out**		
4	Code coverage done using Questasim and verified the design		

red

Input: [325:0] D
Output: reg [162:0] r

1	D	Output r should be the corresponding to the input D reduced version	This Scenario describes Proper functioning of **red**.
2	Consider transition bins for **D,r**		
3	Code coverage done using Questasim and verified the design		

ks163

```
input wire [162:0] a;
input wire [162:0] b;
output wire [324:0]y;
```

1	a , b	Output d should be the (a*b) in $GF2^n$	This Scenario describes Proper functioning of **ks163**.
2	Consider cross coverage between **a,b**.		
3	Consider transition bins for **a,b,y**		
4	Code coverage done using Questasim and verified the design		

ks81

```
input wire [80:0] a;
input wire [80:0] b;
output wire  [160:0]y;
```

1	a , b	Output d should be the (a*b) in $GF2^n$	This Scenario describes Proper functioning of **ks81**.
2	Consider cross coverage between **a,b**.		
3	Consider transition bins for **a,b,y**		
4	Code coverage done using Questasim and verified the design		

ks27

```
 input wire [26:0] a;
input wire [26:0] b;
output wire  [52:0]y;
```

1	a , b	Output d should be the (a*b) in $GF2^n$	This Scenario describes Proper functioning of **ks27**.
2	Consider cross coverage between **a,b.**		
3	Consider transition bins for **a,b,y**		
4	Code coverage done using Questasim and verified the design		

ks9

input wire [8:0] a;
input wire [8:0] b;
output wire [16:0]y;

1	a , b	Output d should be the (a*b) in $GF2^n$	This Scenario describes Proper functioning of **ks9**.
2	Consider cross coverage between **a,b**.		
3	Consider transition bins for **a,b,y**		
4	Code coverage done using Questasim and verified the design		

ks3

input wire [2:0] a;
input wire [2:0] b;
output wire [4:0]y;

1	a , b	Output d should be the (a*b) in $GF2^n$	This Scenario describes Proper functioning of **ks3**.
2	Consider cross coverage between **a,b.**		
3	Consider transition bins for **a,b,y**		
4	Code coverage done using Questasim and verified the design		

Overlap_9

input wire [4:0] p0,p1,p2,p3,p4,p5;
output wire [16:0]y;

1	p0,p1,p2,p3,p4,p5	Output y will efficiently combine the partial products generated by the Karatsuba algorithm to form the final product.	This Scenario describes Proper functioning of **overlap_9**
2	Consider transition bins for **p0,p1,p2,p3,p4,p5,y**		
3	Code coverage done using Questasim and verified the design		

Overlap_27

input wire [16:0] p0,p1,p2,p3,p4,p5;
output wire [52:0]y;

1	p0,p1,p2,p3,p4,p5	Output y will efficiently combine the partial products generated by the Karatsuba algorithm to form the final product.	This Scenario describes Proper functioning of **overlap_27**
2	Consider transition bins for **p0,p1,p2,p3,p4,p5,y**		
3	Code coverage done using Questasim and verified the design		

Overlap_81

input wire [52:0] p0,p1,p2,p3,p4,p5;
output wire [160:0]y;

1	p0,p1,p2,p3,p4,p5	Output y will efficiently	This Scenario

		combine the partial products generated by the Karatsuba algorithm to form the final product.	describes Proper functioning of **overlap_81**
2	Consider transition bins for **p0,p1,p2,p3,p4,p5,y**		
3	Code coverage done using Questasim and verified the design		

Overlap_243

input wire [160:0] p0,p1,p2,p3,p4,p5;
output wire [484:0]y;

1	p0,p1,p2,p3,p4,p5	Output y will efficiently combine the partial products generated by the Karatsuba algorithm to form the final product.	This Scenario describes Proper functioning of **overlap_243**
2	Consider transition bins for **p0,p1,p2,p3,p4,p5,y**		
3	Code coverage done using Questasim and verified the design		

Contribution of each team member – "Task Division"

☆ Prime Contributor(s) for performing particular Task

✓ Helped Performing Task

Task	Sushma R	Alwin Shaju
Reading Documentation	☆	☆
Figuring out what to verify	☆	☆
Understanding **ECC_all_Parts Module**	✓	☆
Verification Plan for **ECC_all_Parts Module**	✓	☆
Understanding **cs_adder_163b Module**	☆	✓
Verification Plan for **cs_adder_163b**	☆	✓
Understanding **multiplier_163b**	☆	✓
Verification Plan for **multiplier_163b**	☆	✓
Understanding **inversion_op**	☆	✓
Verification Plan for **inversion_op**	☆	✓
Understanding **circ_leftshift_163b**	☆	✓
Verification Plan for **circ_leftshift_163b**	☆	✓
Understanding **ks163, ks81, ks27, ks9, ks3 Modules**	✓	☆
Verification Plan for **ks163, ks81, ks27, ks9, ks3 Modules**	✓	☆
Understanding **Overlap243,Overlap160,Overlap81,Overlap 27,Overlap9**	✓	☆
Verification Plan for **Overlap243,Overlap160,Overlap81,Overlap 27,Overlap9**	✓	☆

Layered testbench implementation - **Generator**	☆	✓

Layered testbench implementation - **Driver**	☆	✓
Layered testbench implementation - **Monitor**	☆	✓
Layered testbench implementation – **Scoreboard**	✓	☆
Layered testbench implementation - **Transaction**	☆	☆
Layered testbench implementation - **Interface**	✓	☆
Writing Golden reference model in System Verilog for **ECC_all_Parts Module**	✓	☆
Writing Golden reference model in System Verilog for **Inversion_Op**	✓	☆
Writing Golden reference model in System Verilog for **cs_adder_163b Module**	☆	✓
Writing Golden reference model in System Verilog for **multiplier_163b**	✓	☆
Writing Golden reference model in System Verilog for **circ_leftshift_163b**	☆	✓
Writing Golden reference model in System Verilog for **ks163, ks81, ks27, ks9, ks3 Modules**	☆	☆
Analyzing errors for **ks163, ks81, ks41, ks40, ks21, ks20, ks11, ks10 Modules**	✓	☆
Analyzing error in **Inversion_OP**	☆	✓
Reading "**Power Attack Resistant Efficient FPGA Architecture for Karatsuba Multiplier"** to fix Karatsuba multiplier	✓	☆
Analyzing errors for **ECC_all_Parts Module**	☆	☆
Reading "Hardware design and implementation of ECC based crypto processor for low-area-applications on FPGA" to ECC crypto processor	☆	☆

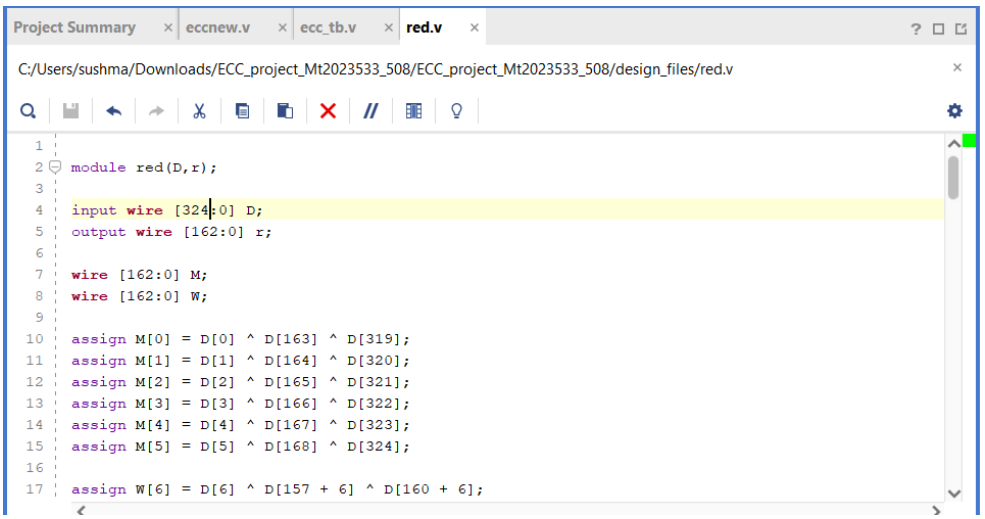
Reading “A fast algorithm for computing multiplicative inverses in GF(2*m*) using normal base” for Inversion_module	✓	☆
Integration on code and running Questasim	☆	✓
Analyzing results and writing more test cases and more randomization constraints to cover corner cases.	☆	☆
Documentation	☆	☆
Conclusion	☆	✓

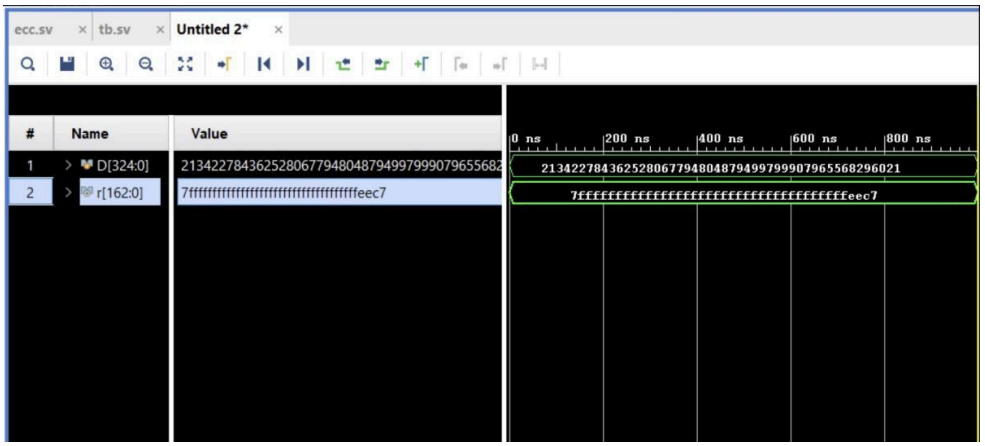
List of problems/issues that you faced during the verification process using any of the System Verilog simulators.

Missing Module Definition:

In the ECC Crypto Processor, it has been noted that the definition for the "Red" module, which is used within the Karatsuba 163 multiplier module, is absent. This "Red" module is essential for carrying out modular reduction operations on the output from the Karatsuba multiplier. It processes the multiplier's product by applying modular arithmetic to ensure the result stays within a specified range.

This missing module has been written and added to the existing design files and hence is considered for verification.





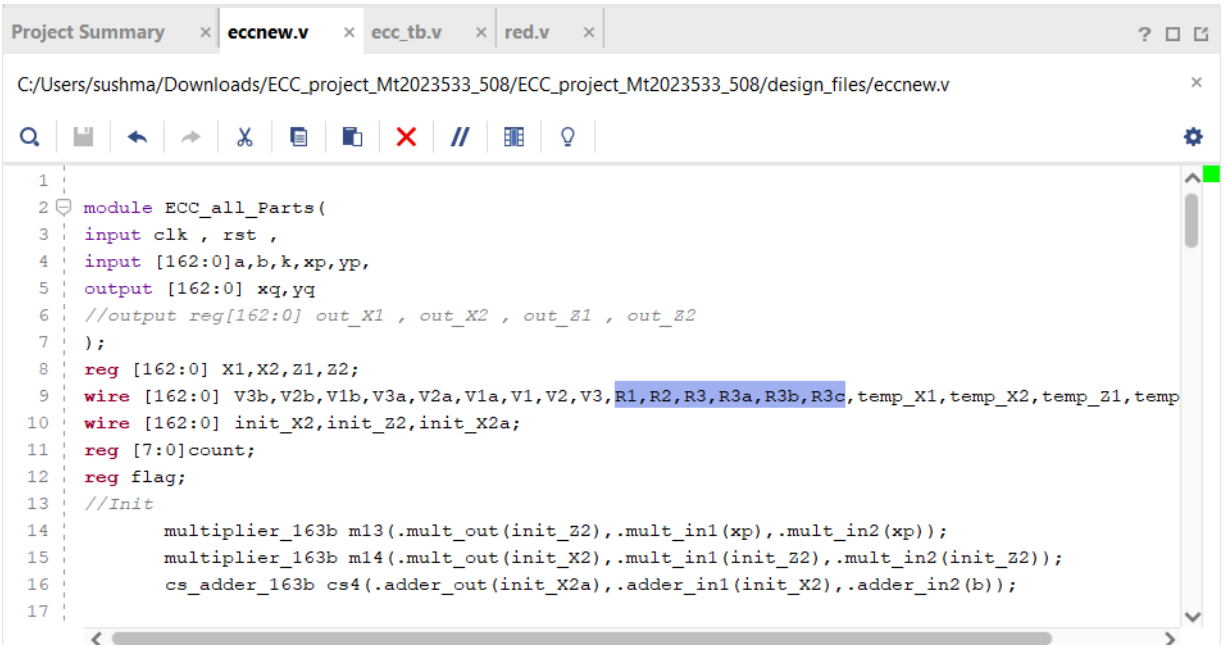
Inconsistent Simulation Results:

During the simulation of the entire ECC Crypto Processor with directed test vectors, a discrepancy was noted where the actual outputs did not match the expected outcomes as detailed in the documentation. This inconsistency has raised concerns regarding the design's accuracy and reliability.

Further investigation revealed that certain nets within the design are concurrently driven by multiple modules. This concurrent driving leads to signal conflicts, resulting in unknown or "X" values during simulation. As a result, the overall output of the ECC Crypto Processor becomes unreliable and fails to align with the anticipated results.

To achieve consistent simulation outcomes, new wires have been declared and the error has been resolved.

```
multiplier_163b m13(.mult_out(init_Z2),.mult_in1(xp),.mult_in2(xp));
multiplier_163b m14(.mult_out(init_X2),.mult_in1(init_Z2),.mult_in2(init_Z2));
cs_adder_163b cs4(.adder_out(init_X2),.adder_in1(init_X2),.adder_in2(b));

multiplier_163b m1(.mult_out(V1),.mult_in1(X1),.mult_in2(Z2));
multiplier_163b m2(.mult_out(V2),.mult_in1(X2),.mult_in2(Z1));
multiplier_163b m3(.mult_out(V3),.mult_in1(X1),.mult_in2(Z1));
multiplier_163b m4(.mult_out(R3),.mult_in1(Z1),.mult_in2(Z1));
multiplier_163b m5(.mult_out(R3),.mult_in1(R3),.mult_in2(R3));
cs_adder_163b cs1(.adder_out(t_Z2),.adder_in1(V1),.adder_in2(V2));
multiplier_163b m6(.mult_out(temp_Z2),.mult_in1(t_Z2),.mult_in2(t_Z2));
multiplier_163b m7(.mult_out(temp_Z1),.mult_in1(V3),.mult_in2(V3));
multiplier_163b m8(.mult_out(V1),.mult_in1(V1),.mult_in2(V2));
multiplier_163b m9(.mult_out(V2),.mult_in1(xp),.mult_in2(temp_Z2));
multiplier_163b m10(.mult_out(V3),.mult_in1(b),.mult_in2(R3));
multiplier_163b m11(.mult_out(R3),.mult_in1(X1),.mult_in2(X1));
multiplier_163b m12(.mult_out(R3),.mult_in1(R3),.mult_in2(R3));
cs_adder_163b cs2(.adder_out(temp_X2),.adder_in1(V1),.adder_in2(V2));
cs_adder_163b cs3(.adder_out(temp_X1),.adder_in1(V3),.adder_in2(R3));
```



Low-Level Module Issues:

During our examination of the Karatsuba multiplier module, a key component of the ECC Crypto Processor, we encountered output discrepancies similar to the previously mentioned inconsistent simulation results. Upon further investigation, we discovered that these discrepancies were the result of low-level module issues stemming from inadvertent errors made during the previous design phase.

With the aim to resolve these errors, new karatsuba multiplier modules were used which were supposed to be a part of the updated design folders but unfortunately remained missing. So, we rectified the error ourselves. These mentioned discrepancies were resolved with the karatsuba multipliers being used as of now.

Solutions Implemented:

1. Implementation of missing red module: Upon deep investigation, we were able to identify the algorithm(Ref - "*Hardware Design and Implementation of ECC based Crypto Processor for Low-area-applications on FPGA*") that implements the red module. After thorough analysis and understanding of the algorithm, we successfully translated it into a Verilog design.

Algorithm 2: FF Reduction [11] and [12]

Input: polynomial D(x) with 325 bits
Output: r(x) with 163 bits wide length
M ← D[i] ⊕ D[i+163] ⊕ D[i+319]
W ← D[i] ⊕ D[i+157] ⊕ D[i+160]

1) for $0 \leq i \leq 1$
r[i] ← M ⊕ D[i+320] ⊕ D[i+323]

2) for $i = 2$
r[i] ← M ⊕ D[i+320]

3) for $3 \leq i \leq 5$
r[i] ← M ⊕ D[i+160] ⊕ D[i+316] r[i] ⊕ D[i+317]

4) for $i = 6$
r[i] ← W ⊕ D[i+163] ⊕ D[i+313] ⊕ D[i+314] ⊕ D[i+316]

5) for $7 \leq i \leq 10$
r[i] ← W ⊕ D[i+156] ⊕ D[i+163] ⊕ D[i+312] ⊕ D[i+314]

6) for $11 \leq i \leq 12$
r[i] ← W ⊕ D[i+156] ⊕ D[i+163] ⊕ D[i+312]

7) for $13 \leq i \leq 161$
r[i] ← W ⊕ D[i+156] ⊕ D[i+163]

8) for $i = 162$
r[i] ← W ⊕ D[i+156]

2. Implementation of Karatsuba: As stated in "problems faced in verification" section there is no proper implementation for Karatsuba present in design. To fix this, we implemented the Karatsuba multiplier with the following equations. (Ref – "*An Optimized M- Term Karatsuba-Like Binary Polynomial Multiplier for Finite Field Arithmetic"*).

$$\begin{cases} X = (X_0)b^0 + (X_1)b^1 + (X_2)b^2 + ... + (X_{m-1})b^{m-1} \\ Y = (Y_0)b^0 + (Y_1)b^1 + (Y_2)b^2 + ... + (Y_{m-1})b^{m-1} \\ X \times Y = (X_0.Y_0)b^0 + (X_0.Y_1 + X_1.Y_0)b^1 + \\ \qquad (X_0.Y_2 + X_1.Y_1 + X_2.Y_0)b^2 + ... \\ \quad +(X_{m-1}.Y_{m-2} + X_{m-2}.Y_{m-1})b^{2m-3} + (X_{m-1}Y_{m-1})b^{2m-2} \end{cases} \quad (1)$$

$$\begin{cases} X = (X_0)b^0 + (X_1)b^1 \\ Y = (Y_0)b^0 + (Y_1)b^1 \\ X \times Y = [X_0.Y_0]b^0 + [(X_0 + X_1)(Y_0 + Y_1) - X_0.Y_0 \\ \qquad -X_1.Y_1]b^1 + [X_1Y_1]b^2 \end{cases} \quad (2)$$

3. Fixing ECC Crypto processor: As stated in "problems faced in verification" section ECC Crypto processor is giving "XXX" due to multi driven error, we fixed this by understanding the "Lopez Dahab Algorithm" (Ref - "*Hardware Design and Implementation of ECC based Crypto Processor for Low-area-applications on FPGA")*

Inputs: $P = (x_p, y_p) \in GF(2^m)$,
$K \leftarrow (k_{i-j}, \ldots, k_1, k_0)$ where, K is an j bit integer
Output: $k.p = (x_q, y_q)$

Step 1: Affine to LD Conversion/Initializations

1)	$X_1 \leftarrow (x_p)$	2)	$Z_1 \leftarrow 1$	*3)*	$Z_2 \leftarrow (X_1)^2$
4)	$X_2 \leftarrow (Z_2)^2$	*5)*	$X_2 \leftarrow (X_2+b)$		

Step 2: Scalar Multiplication (SM) Loop Process

for int i = j-2 down to 0 do

1)	$V_1 \leftarrow (X_1 Z_2)$	2)	$V_2 \leftarrow (X_2 Z_1)$	*3)*	$V_3 \leftarrow (X_1 Z_1)$
4)	$R_3 \leftarrow (Z_1)^2$	5)	$R_3 \leftarrow (R_3)^2$	6)	$Z_2 \leftarrow (V_1 + V_2)$
7)	$Z_2 \leftarrow (Z_2)^2$	8)	$Z_1 \leftarrow (V_3)^2$	9)	$V_1 \leftarrow (V_1 V_2)$
10)	$V_2 \leftarrow (x_p Z_2)$	11)	$V_3 \leftarrow (b R_3)$	12)	$R_3 \leftarrow (X_1)^2$
13)	$R_3 \leftarrow (R_3)^2$	14)	$X_2 \leftarrow (V_1 + V_2)$	15)	$X_1 \leftarrow (V_3 + R_3)$

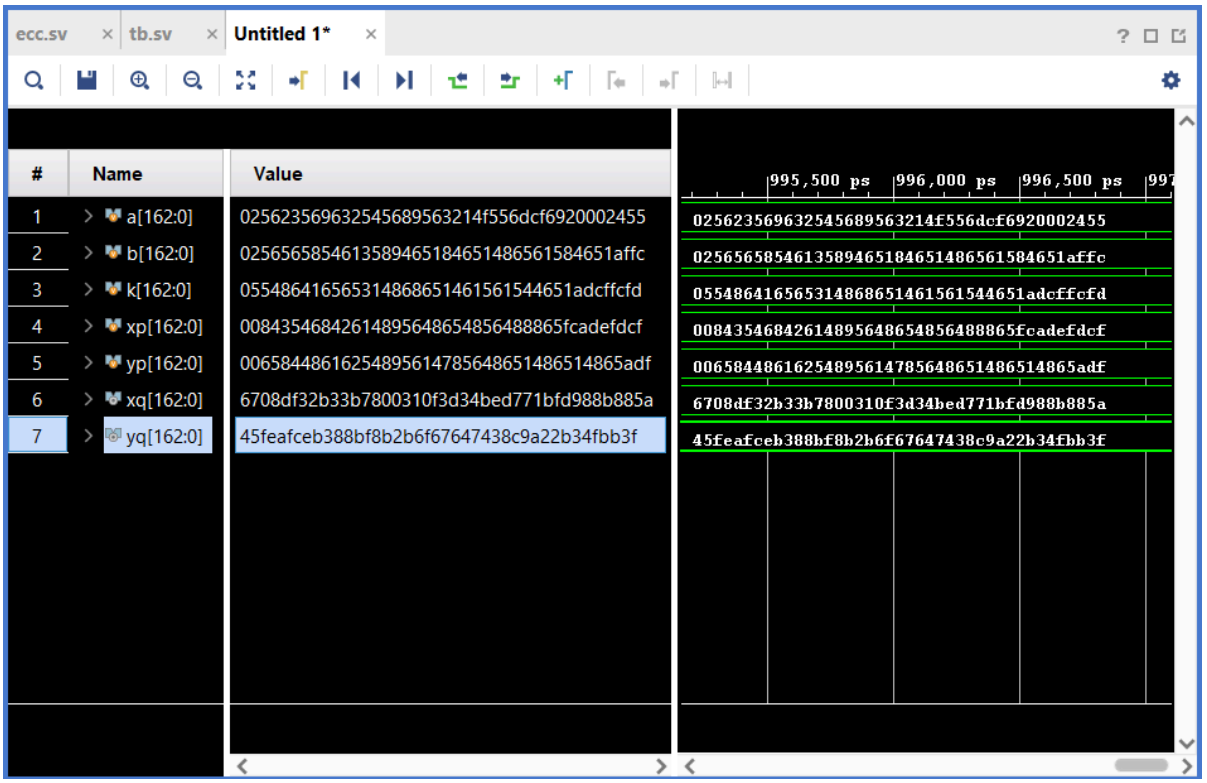
If (i = 0 and K[i] = 1)
swap *(X_1, X_2)*, swap *(Z_1, Z_2)*
end if
end for

Step 3: LD to Affine Conversion/Reconversion

1)	$V_1 \leftarrow Inv(Z_1)$	2)	$V_2 \leftarrow Inv(Z_2)$	3)	$V_3 \leftarrow Inv(x_p)$
4)	$R_1 \leftarrow (X_1 V_1)$	5)	$V_2 \leftarrow (X_2 V_2)$	6)	$R_3 \leftarrow (x_p)^2$
7)	$R_3 \leftarrow (R_3 + y_p)$	8)	$V_1 \leftarrow (x_p + R_1)$	9)	$V_2 \leftarrow (x_p + V_2)$
10)	$V_1 \leftarrow (V_1 V_3)$	11)	$V_2 \leftarrow (V_1 V_2)$	12)	$V_2 \leftarrow (V_2 + R_3)$
13)	$V_2 \leftarrow (V_1 V_2)$	14)	$R_2 \leftarrow (V_2 + y_p)$		

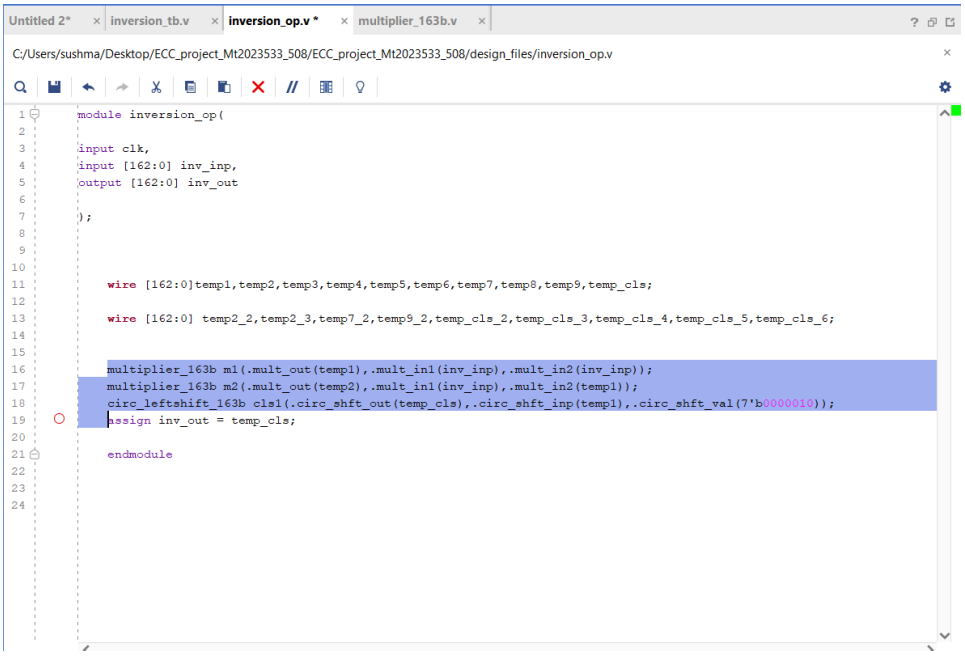
return KP = (x_q, y_q) = (R_1, R_2)

Simulation:



4. Improper design of Inversion module

The inversion module within the ECC crypto processor was not designed as per the algorithm for obtaining the multiplicative inverse.
The code which was present in the design files initially, was not functionally correct, it was not producing the expected multiplicative inverse of the input.



As indicated in the diagram above, the written code does not actually give the inverse of the input.

The algorithm to find the inverse of the input is shown below:

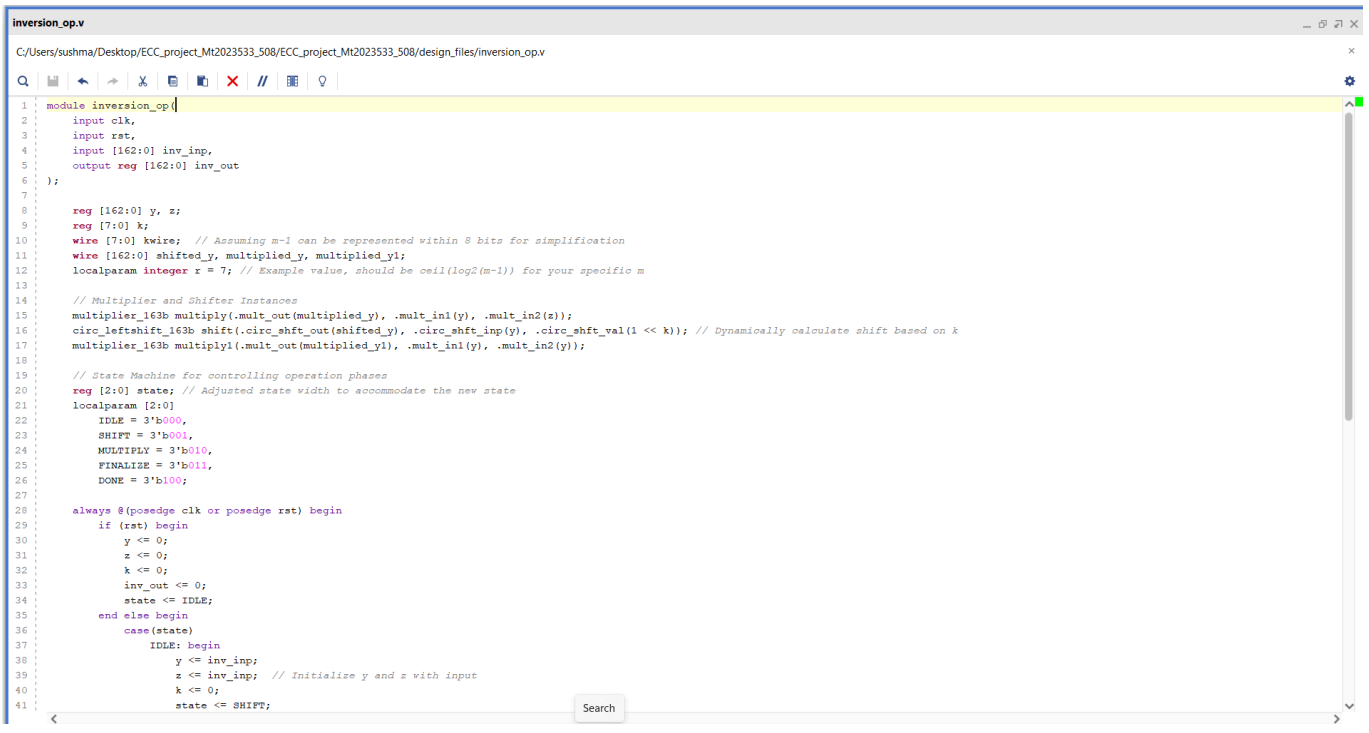
Theorem 1 can be described by

ALGORITHM 2.

S1. $y := x$
S2. for $k := 0$ to $r - 1$ do
S3. begin
S4. $z := y^{2^k}$ (2^k cyclic shifts)
S5. $y := yz$ (multiplication in GF(2^m))
S6. end
S7. $y := y^2$ (multiplicaton in GF(2^m))
S8. write y.

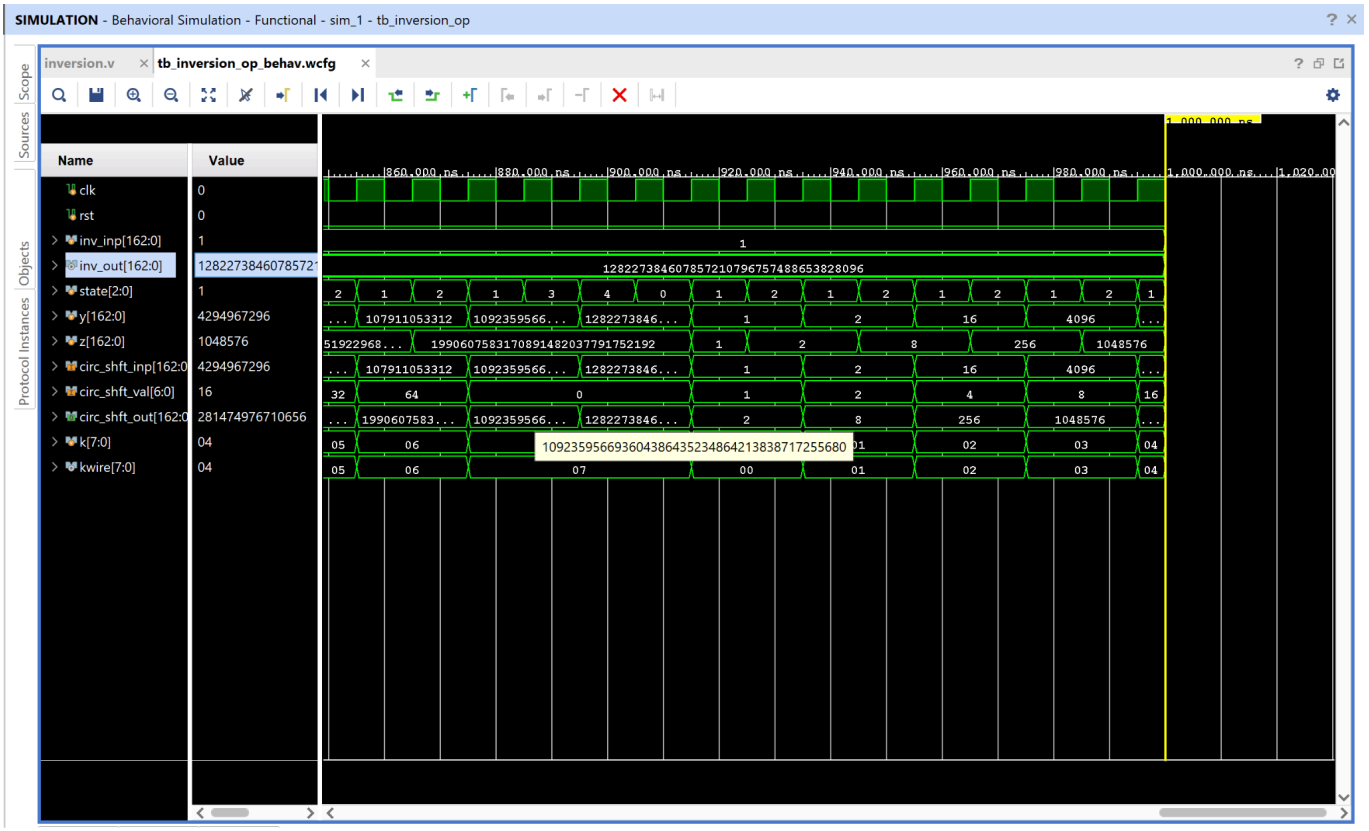
The following theorem is the generalization of Theorem 1.

So, based on the above inversion algorithm a new Verilog code was written as per the exact functionality. The new module is shown below:



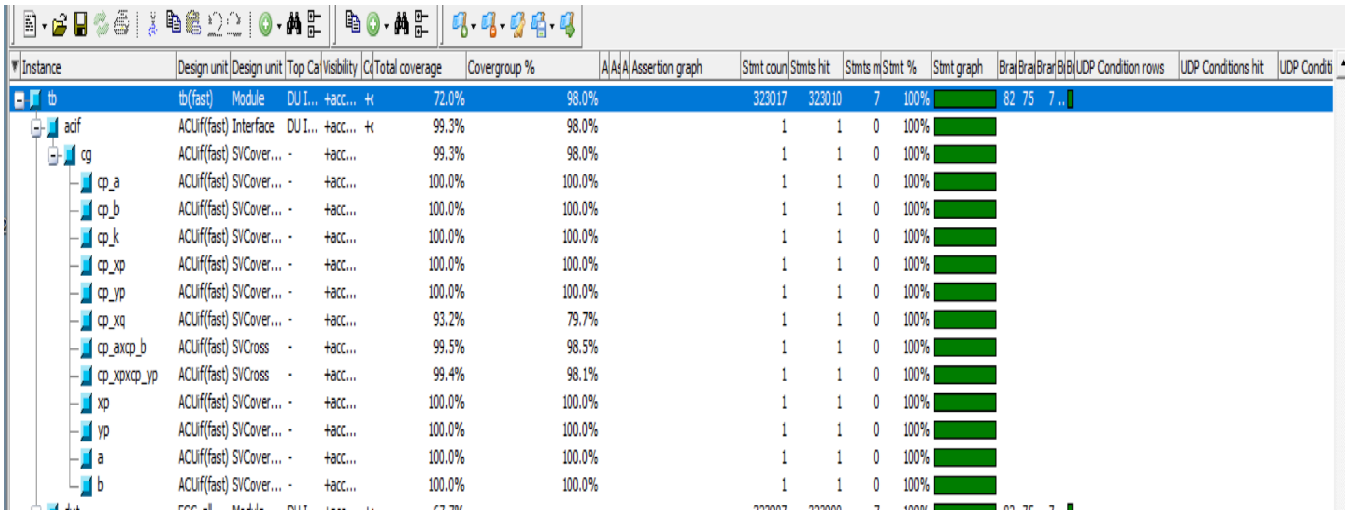

```
module inversion_op(
    input clk,
    input rst,
    input [162:0] inv_inp,
    output reg [162:0] inv_out
);

    reg [162:0] y, z;
    reg [7:0] k;
    wire [7:0] kwire;  // Assuming m-1 can be represented within 8 bits for simplification
    wire [162:0] shifted_y, multiplied_y, multiplied_y1;
    localparam integer r = 7; // Example value, should be ceil(log2(m-1)) for your specific m

    // Multiplier and Shifter Instances
    multiplier_163b multiply(.mult_out(multiplied_y), .mult_in1(y), .mult_in2(z));
    circ_leftshift_163b shift(.circ_shft_out(shifted_y), .circ_shft_inp(y), .circ_shft_val(1 << k)); // Dynamically calculate shift based on k
    multiplier_163b multiply1(.mult_out(multiplied_y1), .mult_in1(y), .mult_in2(y));

    // State Machine for controlling operation phases
    reg [2:0] state; // Adjusted state width to accommodate the new state
    localparam [2:0]
        IDLE = 3'b000,
        SHIFT = 3'b001,
        MULTIPLY = 3'b010,
        FINALIZE = 3'b011,
        DONE = 3'b100;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            y <= 0;
            z <= 0;
            k <= 0;
            inv_out <= 0;
            state <= IDLE;
        end else begin
            case(state)
                IDLE: begin
                    y <= inv_inp;
                    z <= inv_inp;  // Initialize y and z with input
                    k <= 0;
                    state <= SHIFT;
```

The input output functionality is also verified wrt to the above design.

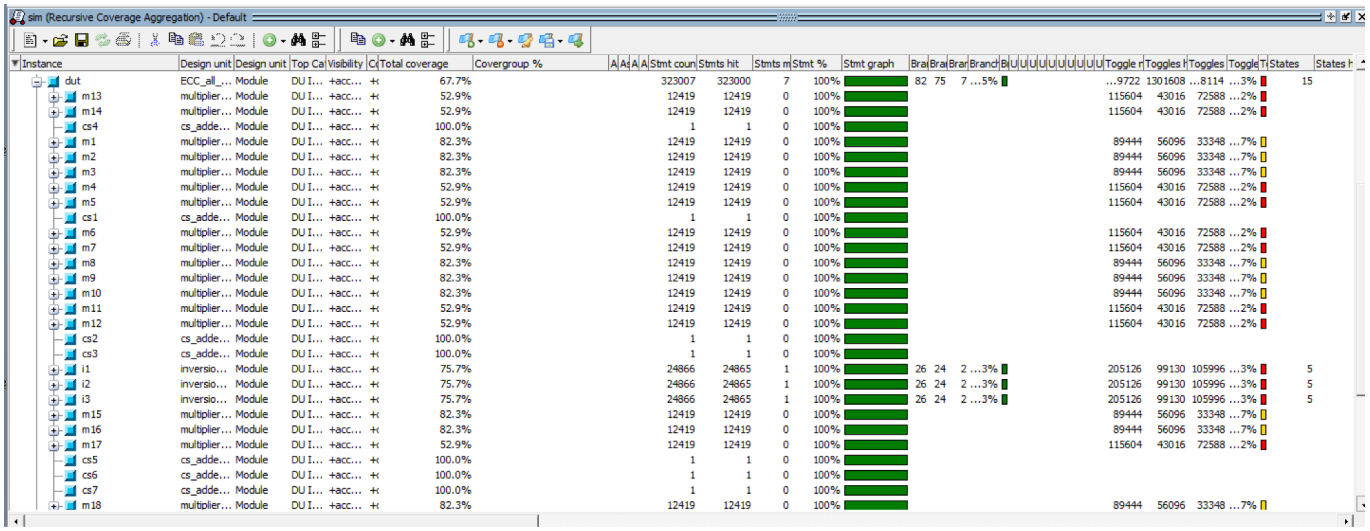


Reports

1.ECC_all_Parts Coverage:



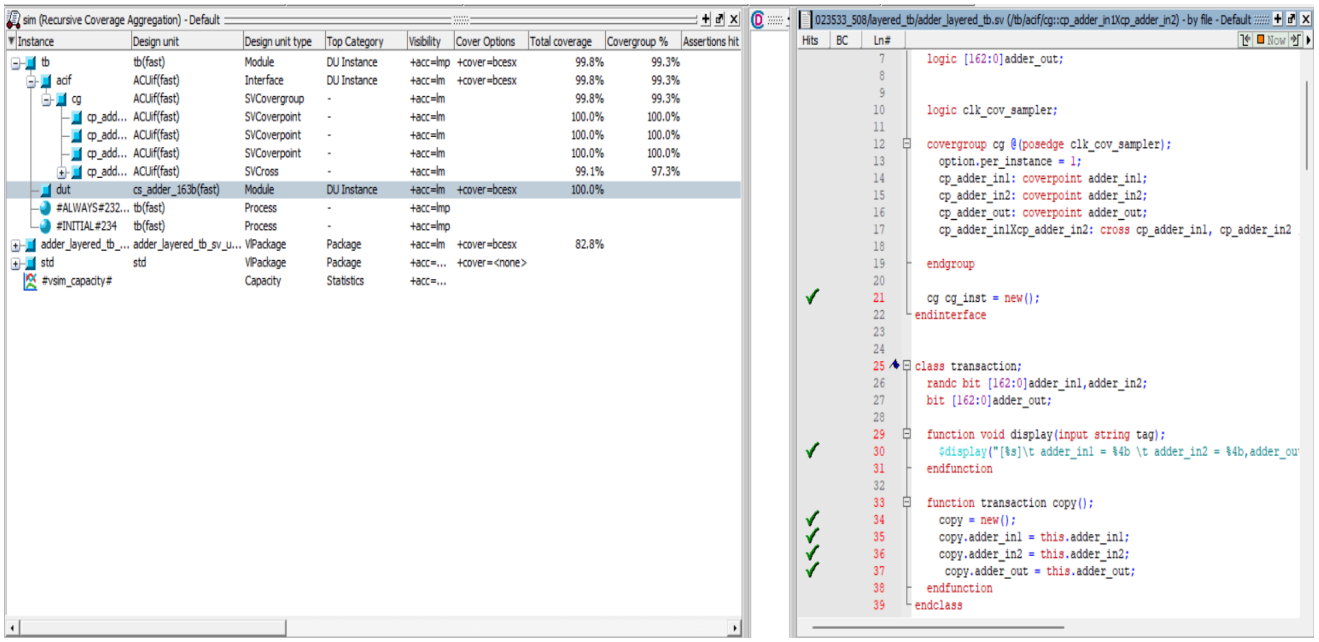
Instance	Design unit	Design unit	Top Ca	Visibility	C	Total coverage	Covergroup %	A	A	A	Assertion graph	Stmt coun	Stmts hit	Stmts m	Stmt %	Stmt graph	Bra	Bra	Brar	Bi	Bi	UDP Condition rows	UDP Conditions hit	UDP Conditi
tb	tb(fast)	Module	DU I...	+acc...	+	72.0%	98.0%					323017	323010	7	100%		82	75	7	..				
acif	ACUif(fast)	Interface	DU I...	+acc...	+	99.3%	98.0%					1	1	0	100%									
cg	ACUif(fast)	SVCover...	-	+acc...		99.3%	98.0%					1	1	0	100%									
cp_a	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
cp_b	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
cp_k	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
cp_xp	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
cp_yp	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
cp_xq	ACUif(fast)	SVCover...	-	+acc...		93.2%	79.7%					1	1	0	100%									
cp_axcp_b	ACUif(fast)	SVCross	-	+acc...		99.5%	98.5%					1	1	0	100%									
cp_xpxcp_yp	ACUif(fast)	SVCross	-	+acc...		99.4%	98.1%					1	1	0	100%									
xp	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
yp	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
a	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									
b	ACUif(fast)	SVCover...	-	+acc...		100.0%	100.0%					1	1	0	100%									

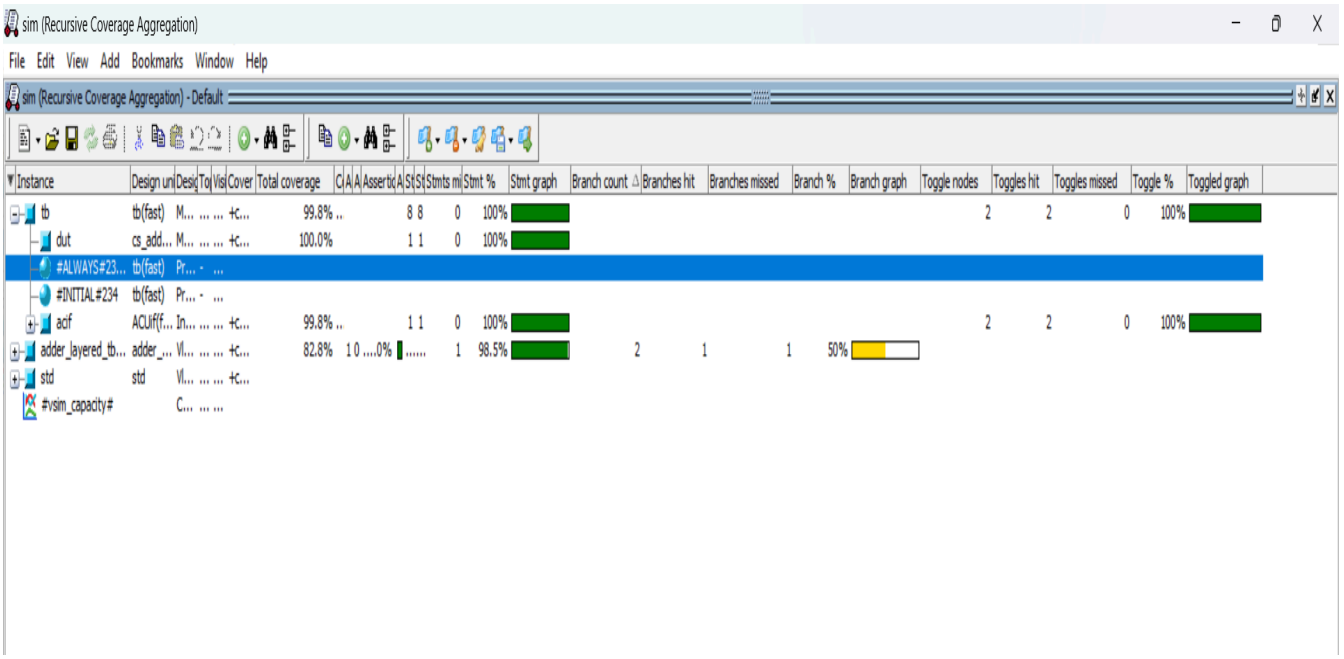


sim (Recursive Coverage Aggregation) - Default

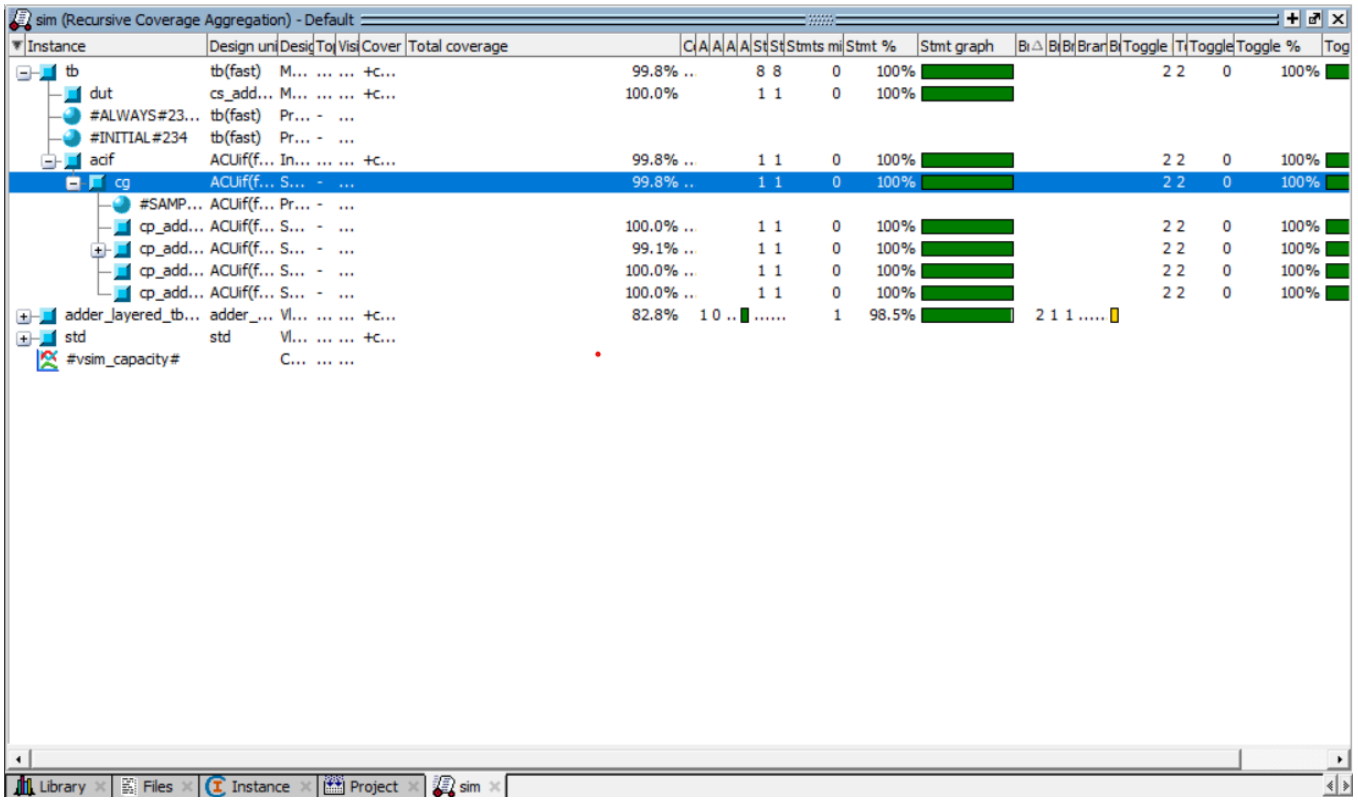
Instance	Design unit	Design unit	Top Ca	Visibility	C	Total coverage	Covergroup %	A	A	A	Stmt coun	Stmts hit	Stmts m	Stmt %	Stmt graph	Bra	Bra	Brar	Branch	B	U	U	U	U	U	U	U	U	Toggle r	Toggles h	Toggles	Toggle	T	States	States h
dut	ECC_all_...	Module	DU I...	+acc...	+	67.7%					323007	323000	7	100%		82	75	7	...5%										...9722	1301608	...8114	...3%		15	
m13	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
m14	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
cs4	cs_adde...	Module	DU I...	+acc...	+	100.0%					1	1	0	100%																					
m1	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m2	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m3	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m4	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
m5	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
cs1	cs_adde...	Module	DU I...	+acc...	+	100.0%					1	1	0	100%																					
m6	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
m7	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
m8	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m9	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m10	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m11	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
m12	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
cs2	cs_adde...	Module	DU I...	+acc...	+	100.0%					1	1	0	100%																					
cs3	cs_adde...	Module	DU I...	+acc...	+	100.0%					1	1	0	100%																					
i1	inversio...	Module	DU I...	+acc...	+	75.7%					24866	24865	1	100%		26	24	2	...3%										205126	99130	105996	...3%		5	
i2	inversio...	Module	DU I...	+acc...	+	75.7%					24866	24865	1	100%		26	24	2	...3%										205126	99130	105996	...3%		5	
i3	inversio...	Module	DU I...	+acc...	+	75.7%					24866	24865	1	100%		26	24	2	...3%										205126	99130	105996	...3%		5	
m15	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m16	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			
m17	multiplier...	Module	DU I...	+acc...	+	52.9%					12419	12419	0	100%															115604	43016	72588	...2%			
cs5	cs_adde...	Module	DU I...	+acc...	+	100.0%					1	1	0	100%																					
cs6	cs_adde...	Module	DU I...	+acc...	+	100.0%					1	1	0	100%																					
cs7	cs_adde...	Module	DU I...	+acc...	+	100.0%					1	1	0	100%																					
m18	multiplier...	Module	DU I...	+acc...	+	82.3%					12419	12419	0	100%															89444	56096	33348	...7%			

2. Cs_adder_163b

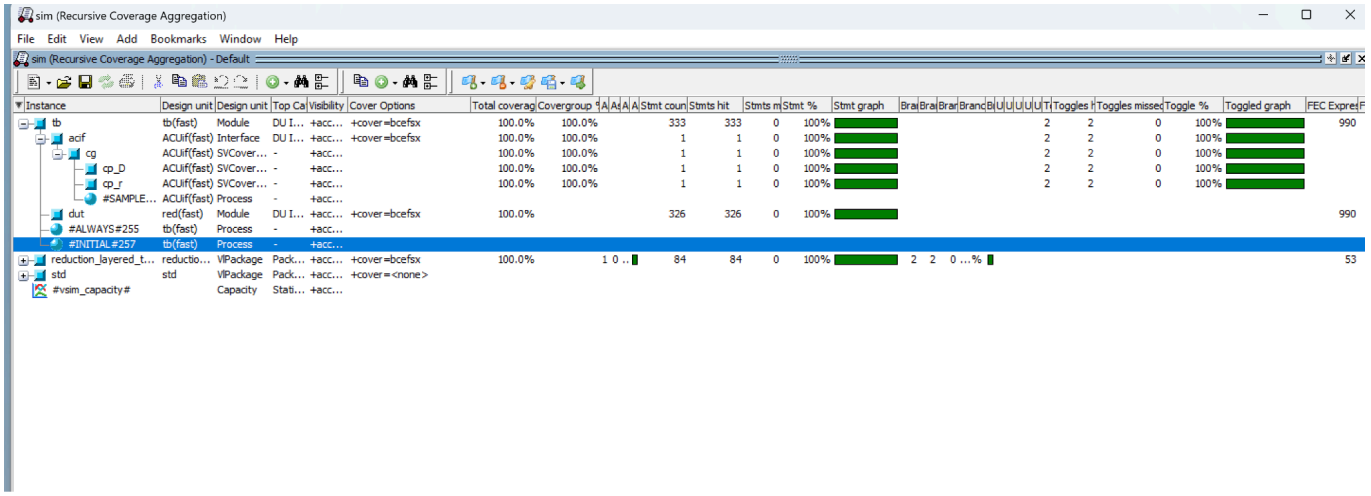




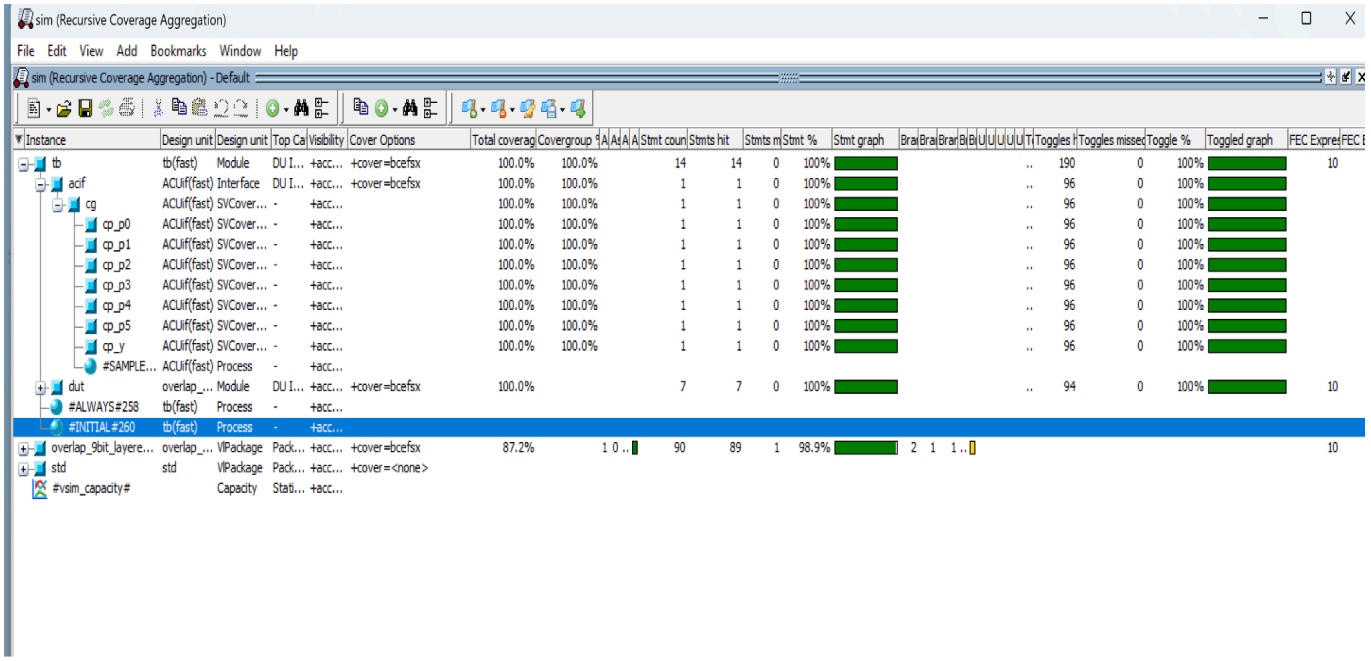
3.Circ_leftshift_163b



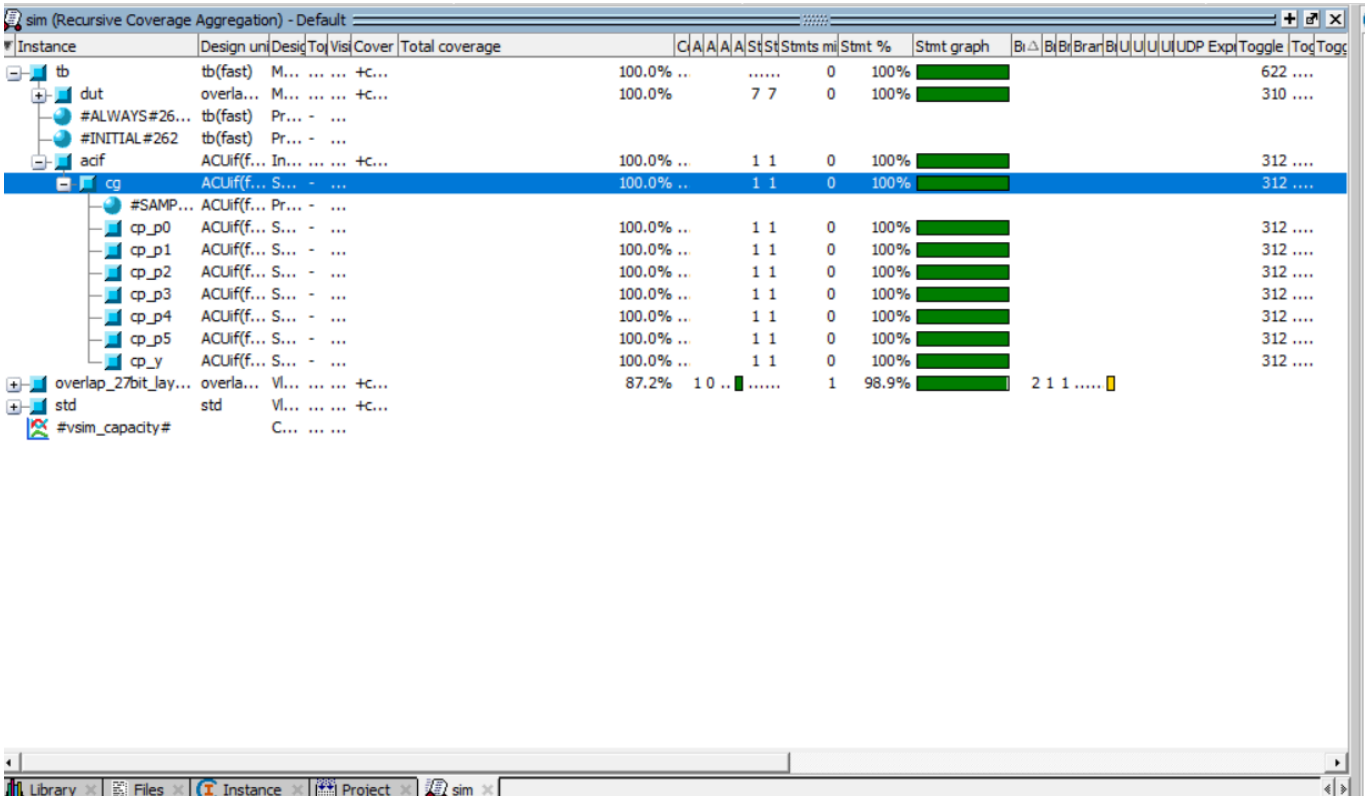
4.Reduction Module



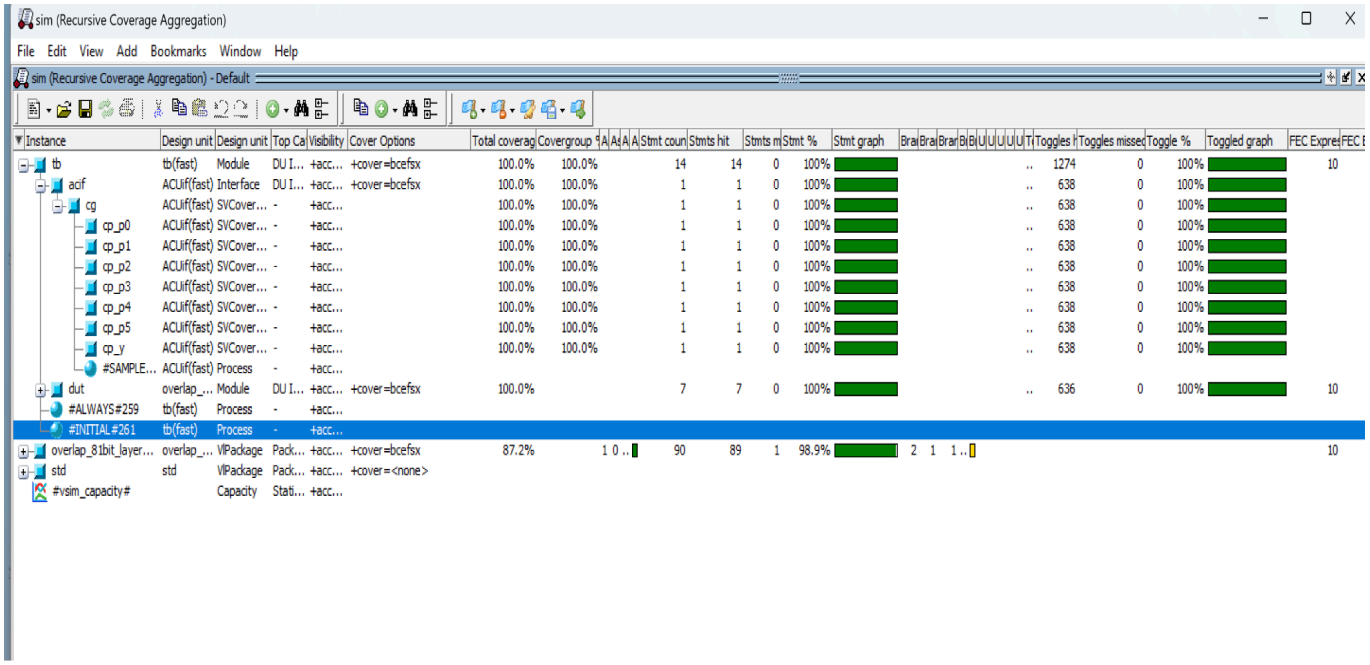
5. Overlap_9



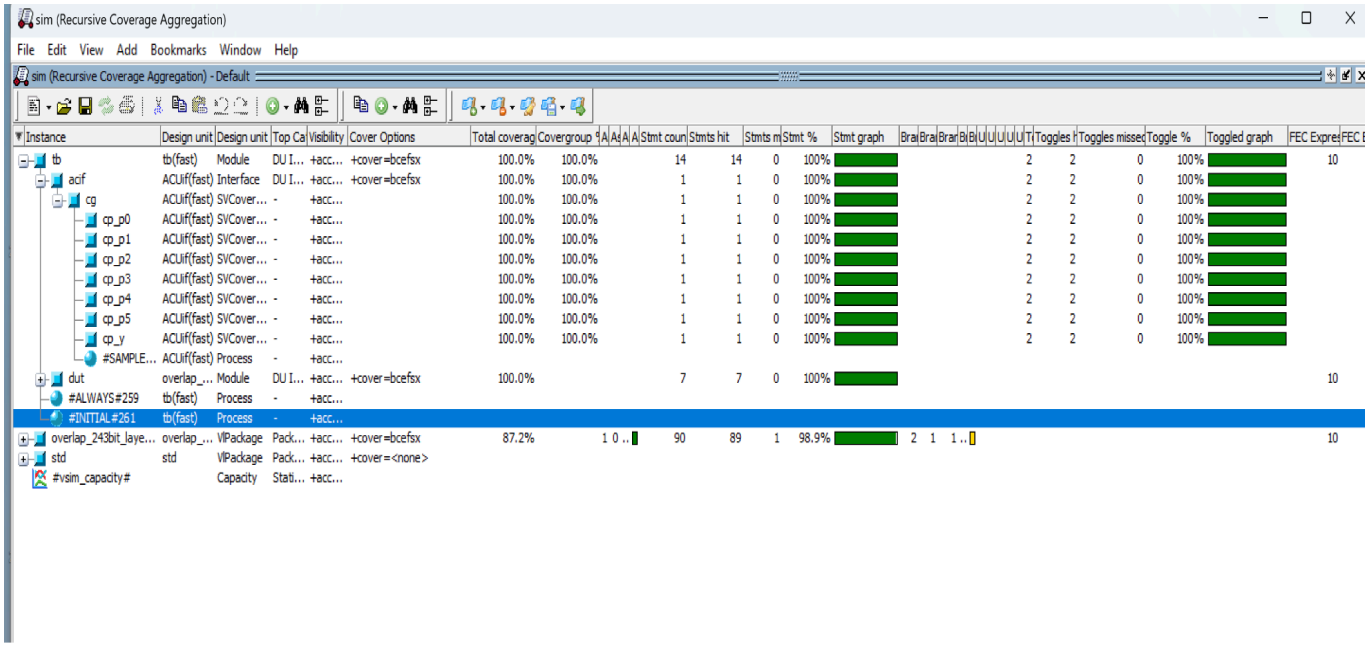
6. Overlap_27


sim (Recursive Coverage Aggregation) - Default
Instance
Design uni
Cover
Total coverage
Stmts mi
Stmt %
Stmt graph
UDP Exp
Toggle
tb
tb(fast)
100.0%
100%
622
dut
overla...
100.0%
7 7
0
100%
310
#ALWAYS#26...
tb(fast)
#INITIAL#262
tb(fast)
acif
ACUif(f...
100.0%
1 1
0
100%
312
cg
ACUif(f...
100.0%
1 1
0
100%
312
#SAMP...
ACUif(f...
cp_p0
ACUif(f...
100.0%
1 1
0
100%
312
cp_p1
ACUif(f...
100.0%
1 1
0
100%
312
cp_p2
ACUif(f...
100.0%
1 1
0
100%
312
cp_p3
ACUif(f...
100.0%
1 1
0
100%
312
cp_p4
ACUif(f...
100.0%
1 1
0
100%
312
cp_p5
ACUif(f...
100.0%
1 1
0
100%
312
cp_y
ACUif(f...
100.0%
1 1
0
100%
312
overlap_27bit_lay...
overla...
87.2%
1
98.9%
2 1 1
std
std
#vsim_capacity#
Library
Files
Instance
Project
sim

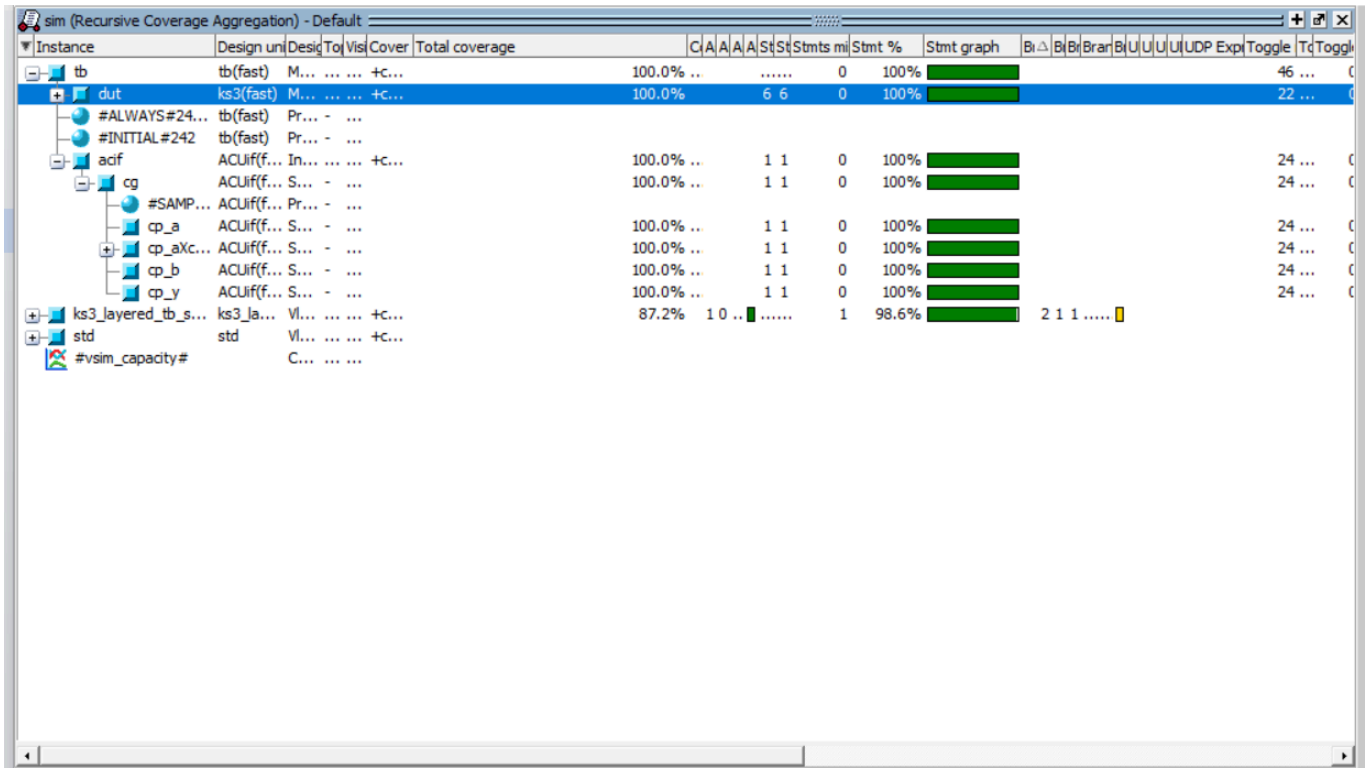
7. Overlap_81



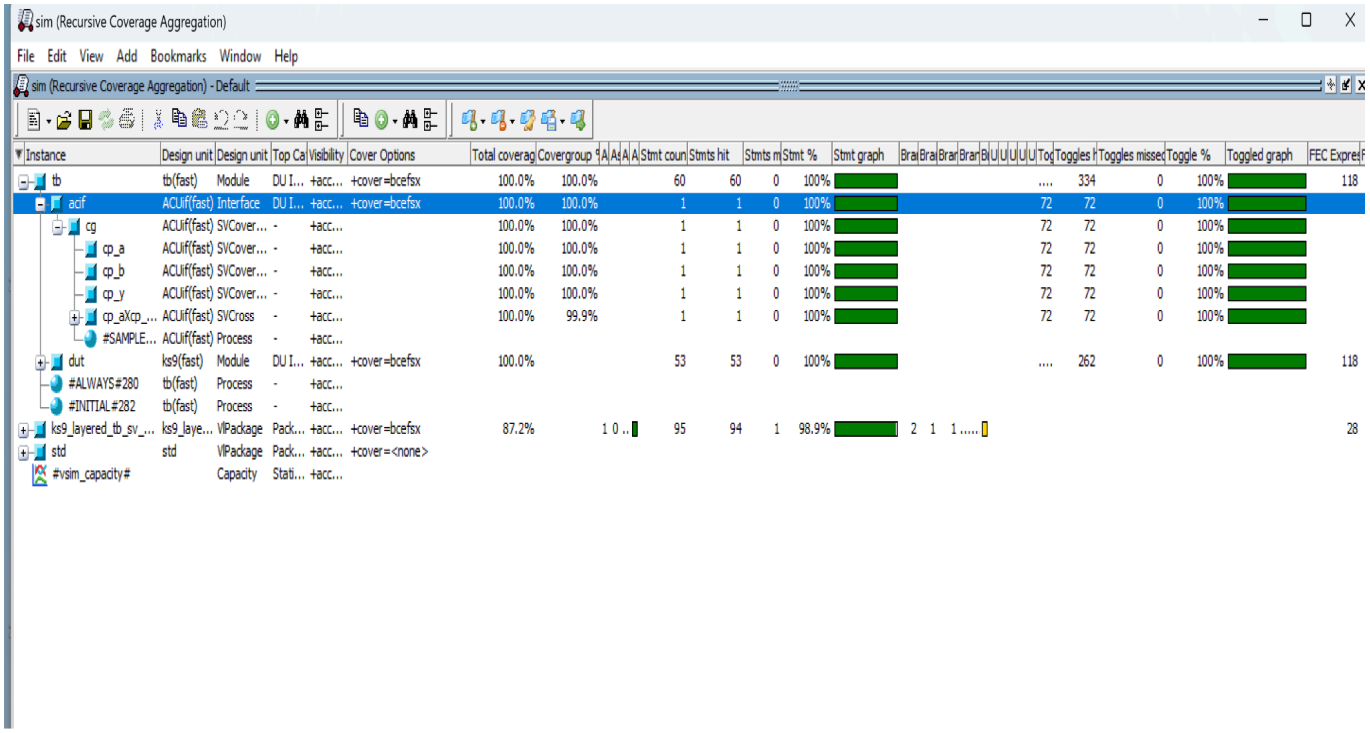
8. Overlap_243



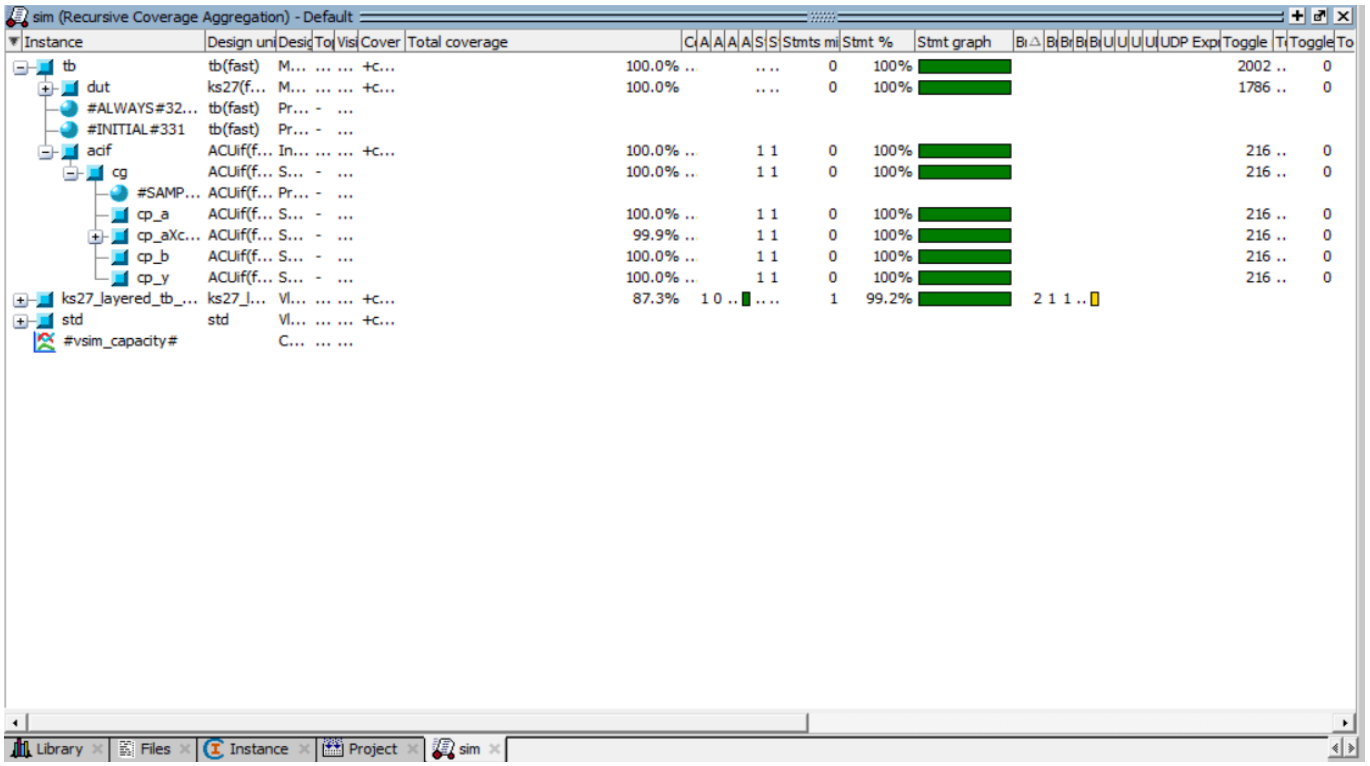
9. KS3


sim (Recursive Coverage Aggregation) - Default
Instance
Design unit
Cover
Total coverage
Stmts mi
Stmt %
Stmt graph
Toggle
tb
tb(fast)
100.0%
100%
46
dut
ks3(fast)
100.0%
6 6
0
100%
22
#ALWAYS#24...
tb(fast)
#INITIAL#242
tb(fast)
acif
ACUif(f...
100.0%
1 1
0
100%
24
cg
ACUif(f...
100.0%
1 1
0
100%
24
#SAMP...
ACUif(f...
cp_a
ACUif(f...
100.0%
1 1
0
100%
24
cp_aXc...
ACUif(f...
100.0%
1 1
0
100%
24
cp_b
ACUif(f...
100.0%
1 1
0
100%
24
cp_y
ACUif(f...
100.0%
1 1
0
100%
24
ks3_layered_tb_s...
ks3_la...
87.2%
1 0 ..
1
98.6%
2 1 1 .....
std
std
#vsim_capacity#

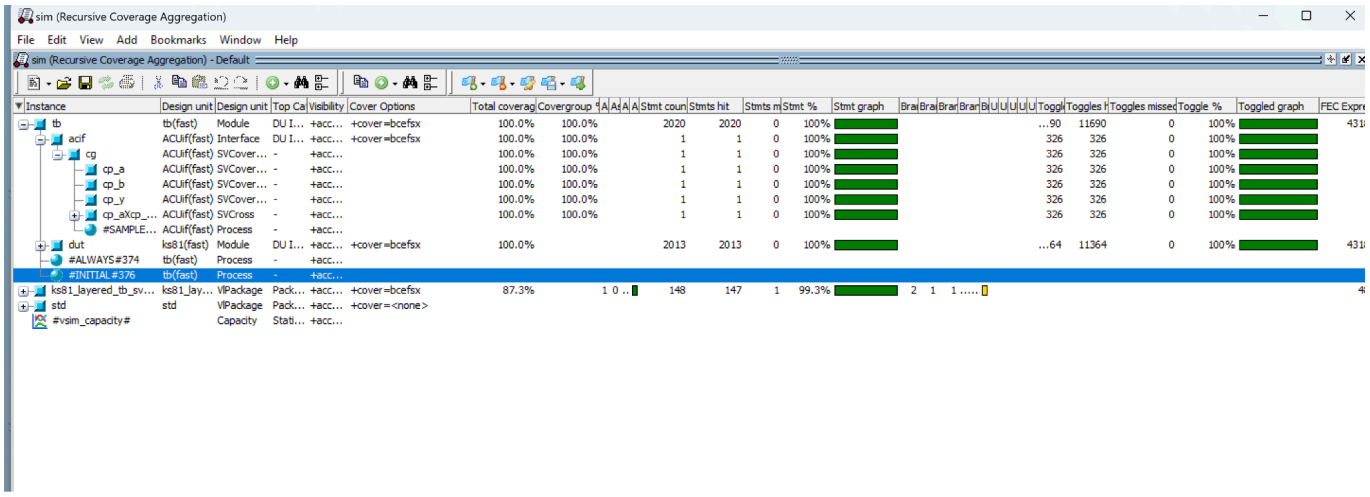
10. KS_9


sim (Recursive Coverage Aggregation)
File Edit View Add Bookmarks Window Help
sim (Recursive Coverage Aggregation) - Default
Instance
Design unit
Design unit
Visibility
Cover Options
Total coverag
Covergroup %
Stmt coun
Stmts hit
Stmts m
Stmt %
Stmt graph
Toggles
Toggles missed
Toggle %
Toggled graph
tb
tb(fast)
Module
+acc...
+cover=bcefsx
100.0%
100.0%
60
60
0
100%
334
0
100%
118
acif
ACUif(fast)
Interface
+acc...
+cover=bcefsx
100.0%
100.0%
1
1
0
100%
72
72
0
100%
cg
ACUif(fast)
SVCover...
+acc...
100.0%
100.0%
1
1
0
100%
72
72
0
100%
cp_a
ACUif(fast)
SVCover...
+acc...
100.0%
100.0%
1
1
0
100%
72
72
0
100%
cp_b
ACUif(fast)
SVCover...
+acc...
100.0%
100.0%
1
1
0
100%
72
72
0
100%
cp_y
ACUif(fast)
SVCover...
+acc...
100.0%
100.0%
1
1
0
100%
72
72
0
100%
cp_aXcp_...
ACUif(fast)
SVCross
+acc...
100.0%
99.9%
1
1
0
100%
72
72
0
100%
#SAMPLE...
ACUif(fast)
Process
+acc...
dut
ks9(fast)
Module
+acc...
+cover=bcefsx
100.0%
53
53
0
100%
262
0
100%
118
#ALWAYS#280
tb(fast)
Process
+acc...
#INITIAL#282
tb(fast)
Process
+acc...
ks9_layered_tb_sv_...
ks9_laye...
VPackage
+acc...
+cover=bcefsx
87.2%
1 0 ..
95
94
1
98.9%
2 1 1 .....
28
std
std
VPackage
+acc...
+cover=<none>
#vsim_capacity#
Capacity
+acc...

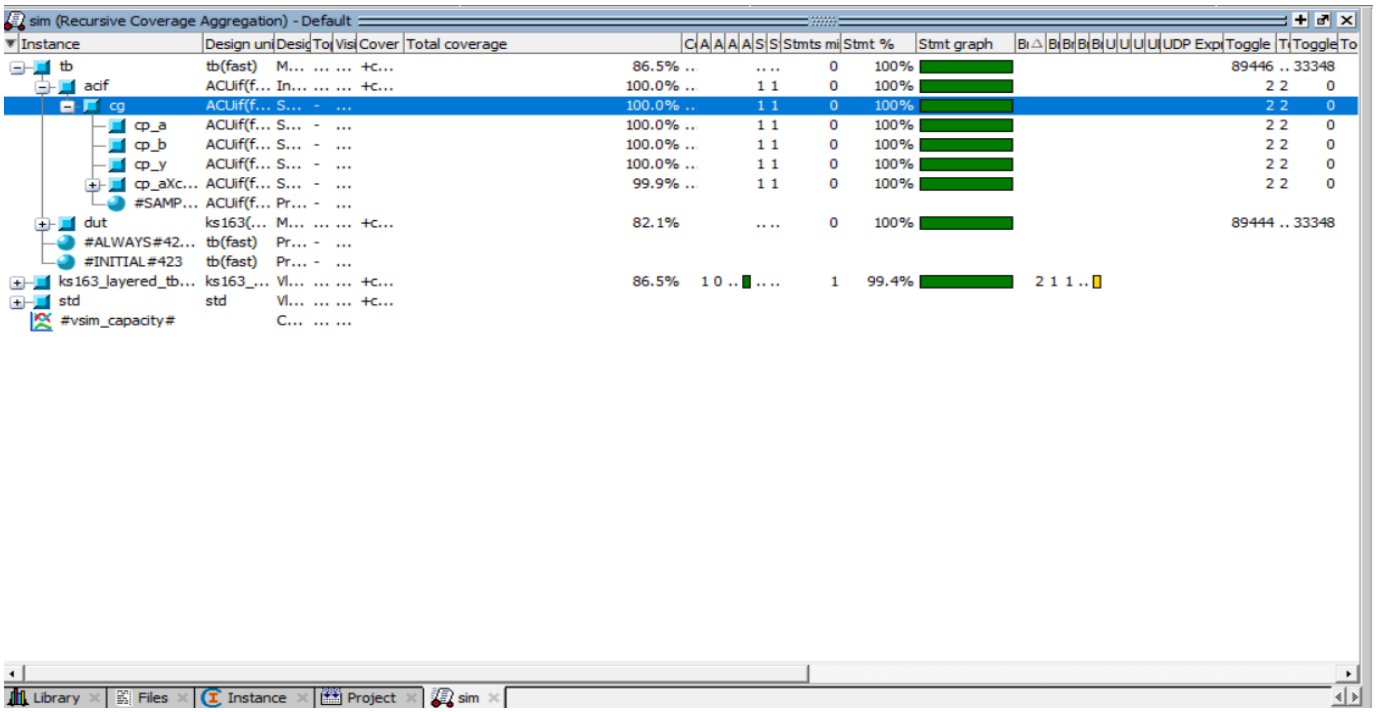
11. KS_27



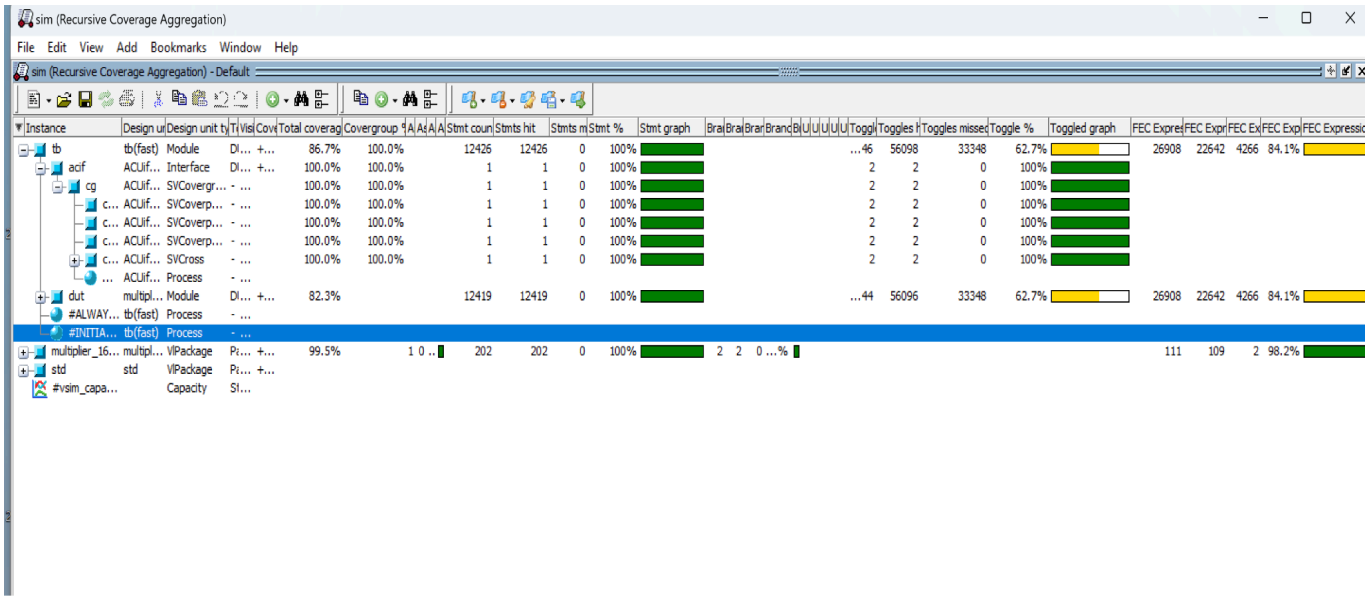
12. KS_81



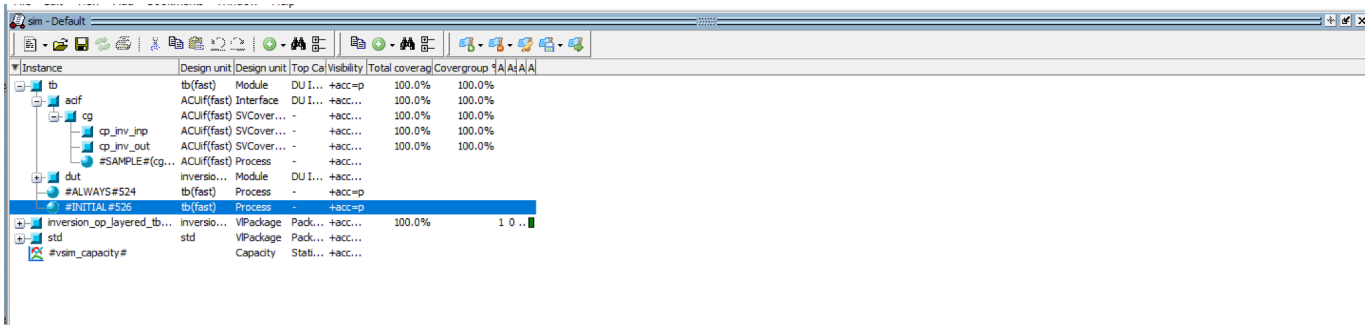
13. KS_163

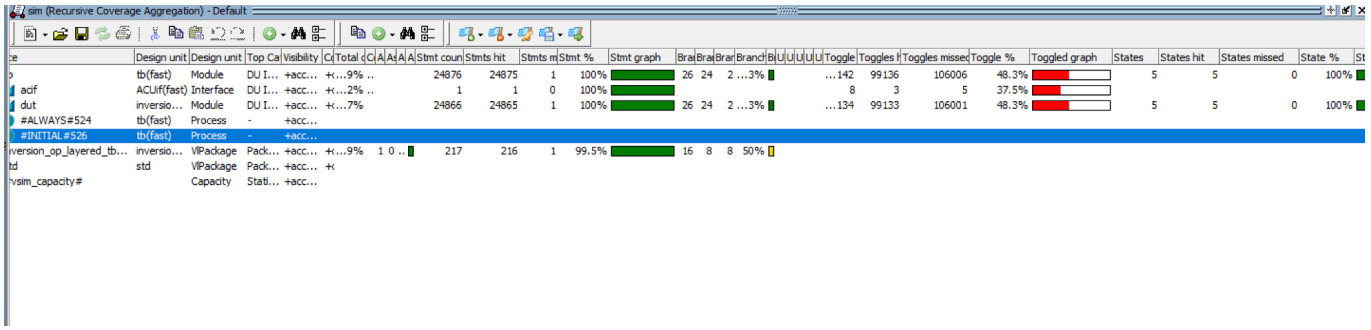


14. Mult_163b



15. Inversion_op





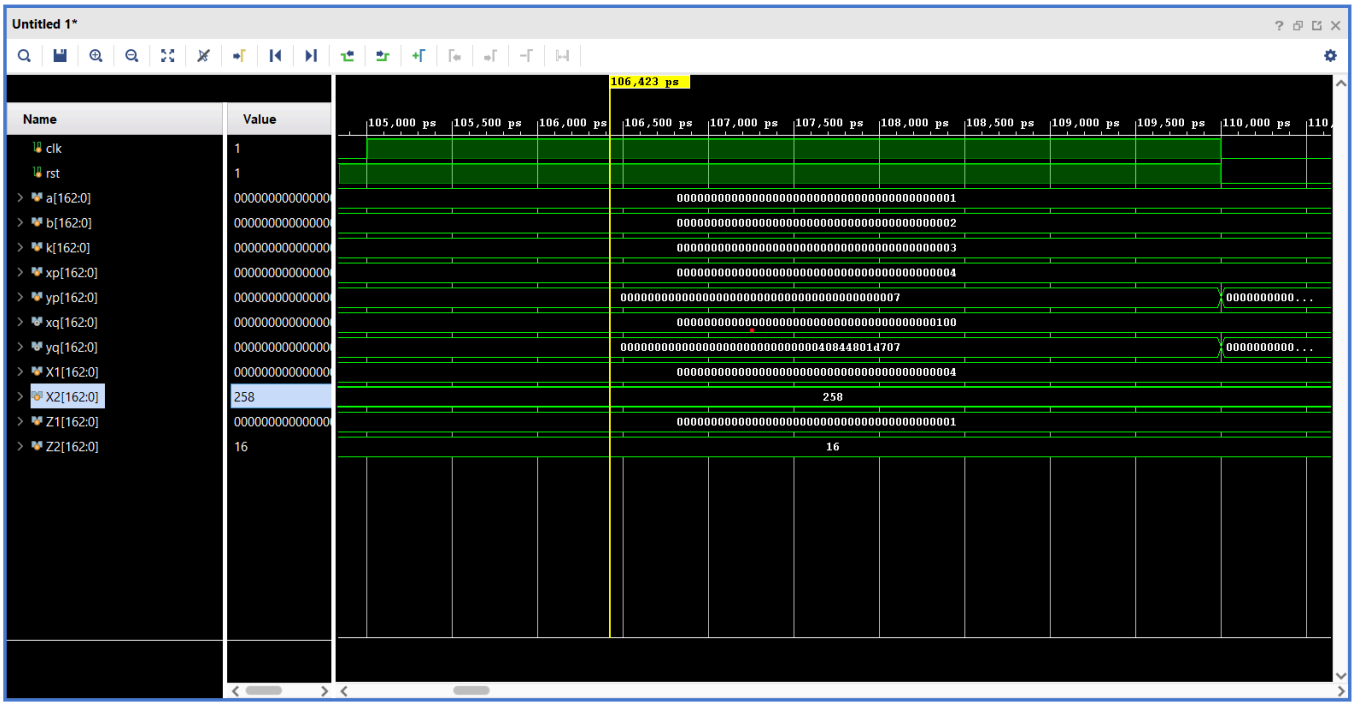
FUNCTIONALITY CHECKS

1. ECC_all_parts

In the ECC all parts module, we have checked the reset functionality of the design and verified the same.
The above mentioned reset functionality is tested and verified to be working correctly in the design module.

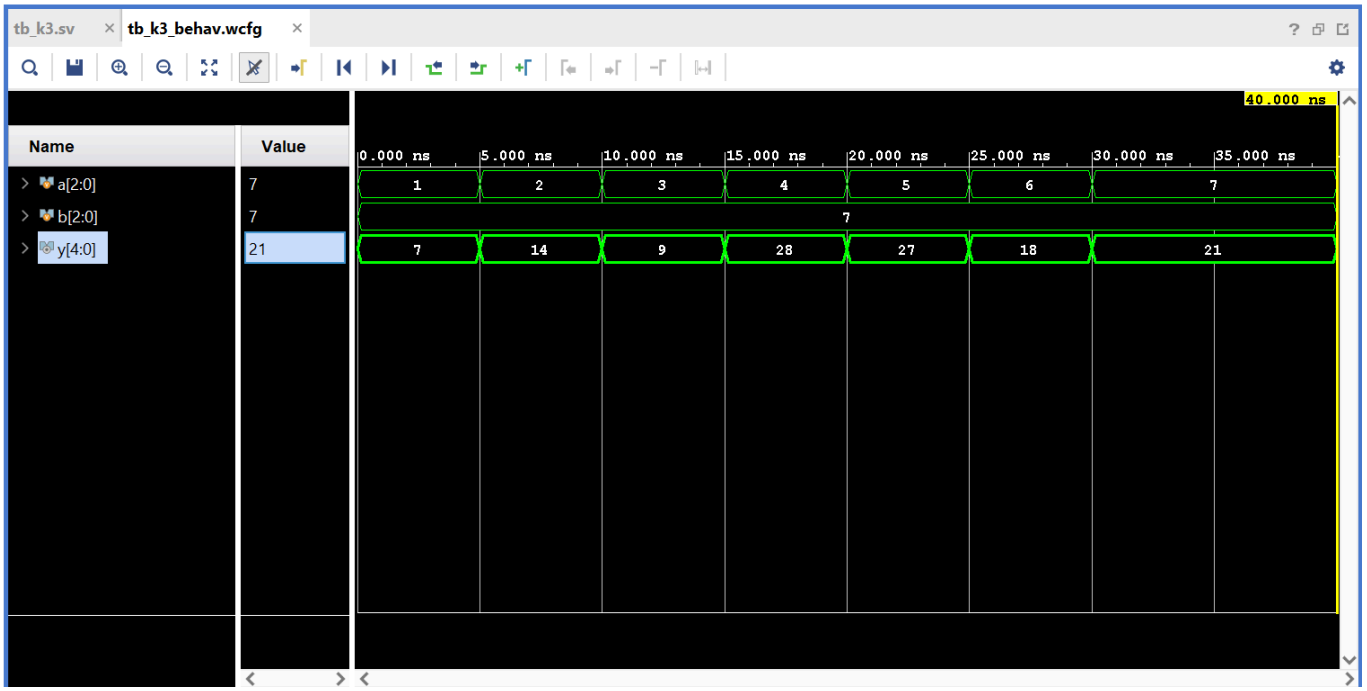
The expected reset behavior is:
Outputs xq,yq should result in
xq = (X1 * V1) / Z1
yq = (X2 * V2) + yp + (V1* V2) + (V1 * V2)
Where:
X1 <= xp
Z1 <= 163'b1
X2 <= xp^4+b Z2 <= xp^2
Z2 <= xp^2

The above mentioned reset functionality is tested and verified to be working correctly in the design module.



2. Constraints on output coverage

Consider the KS_3 multiplier module; while dealing with the output coverage of the design, certain corresponding output bins are declared as ignore bins.
This implementation stems from the fact that certain values are not obtained in the 3 bit Karatsuba multiplier as output or to be more precise, certain values of output will not be generated at all for the given input combinations.
This was thoroughly tested using a directed testbench and hence deducing what bins to ignore at the output side.
The example of one such simulation result is shown below:



Therefore, so as to ensure that the coverage is not impacted unnecessarily, certain ignore bins are essential.

Conclusions

After carefully studying the features and specifications of the Elliptic Curve Crypto (ECC) Processor, we successfully designed a layered testbench and a directed testbench for each module. To ensure the comprehensiveness of our testing approach, we conducted extensive verification using a variety of test case scenarios, including corner cases.

However, during the verification phase, we encountered numerous challenges across multiple test scenarios. These were due to implementation errors and design deficiencies in the ECC Crypto Processor. We meticulously identified and documented these errors with the goal of promptly addressing them. We have tried to resolve the issues to the best of our abilities.

Throughout the verification stage, we placed a strong emphasis on error analysis, providing detailed information on the problems discovered. This analysis offered valuable insights into the design's weaknesses and pinpointed specific areas needing improvement. We aimed to ensure that any future updates to the ECC Crypto Processor would incorporate the necessary corrections and enhancements.

In a nutshell, we have performed a thorough CDV using industry standard tools on the ECC processor considering the functional and the code coverage aspects of the design on Questa Sim. This process required modification of some design modules to ensure the proper functionality. With these changes in place, the coverage results obtained were satisfactory as intended in our verification plan that aimed at reaching a good coverage value.

The inversion module which was missing was designed by us, so further batches can verify that and hence try to improve overall coverage.

REFERENCES

1. A fast algorithm for computing multiplicative inverses in GF(2*m*) using normal base Toshiya Itoh, Shigeo Tsujii Department of Electrical and Electronic Engineering, Faculty of Engineering, Tokyo Institute of Technology, Tokyo 152, Japan

2. Power Attack Resistant Efficient FPGA Architecture for Karatsuba Multiplier Chester Rebeiro1 and Debdeep Mukhopadhyay2 1 MSScholar, Dept. of Computer Science and Engineering 2 Assistant Professor, Dept. of Computer Science and Engineering Indian Institute of Technology Madras, India {rebeiro, debdeep}@cse.iitm.ernet.in

3. An Optimized M-Term Karatsuba-Like Binary Polynomial Multiplier for Finite Field Arithmetic Madhan Thirumoorthi Mitra Mirhassani , Graduate Student Member, IEEE, Moslem Heidarpur , Senior Member, IEEE, and Mohammed Khalid , Member, IEEE, , Senior Member, IEEE

4. A high performance ECC hardware implementation with instruction-level parallelism over GF(2163), Yu Zhang, Dongdong Chen, Younhee Choi, Li Chen, Seok-Bum Ko