Verification of 16 Bit Pipelined Adder

Ameya Chhatre

Prashis Raghuwanshi

Overview

- Introduction
- Block Diagram & Structure
- Verification
- Block Diagrams
- Code Snippet
- Implementation
- Challenges

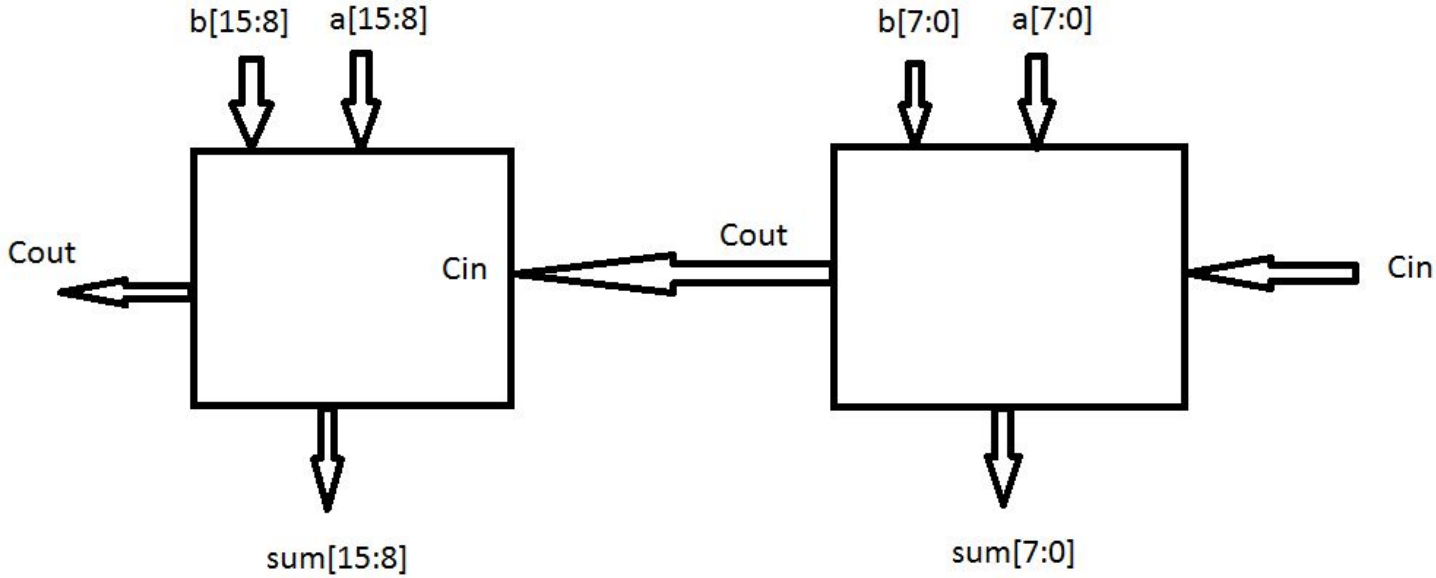
16 Bit Pipelined Adder

Input / Output Signals:

- Input Signals: 16 bit a, 16 bit b, Carry In
- Output Signals: 17 bit Output (16 bit data + 1 bit Carry Out)

Registers:

- Internal Register: 33 bit (16 bit a + 16 bit b + Carry In)
- Pipelined Register: 25 bit (upper 8 bit a&b + sum of lower bits of a&b+ carry)
- Output Register: 17 bit (16 bit output + final Carry Out)



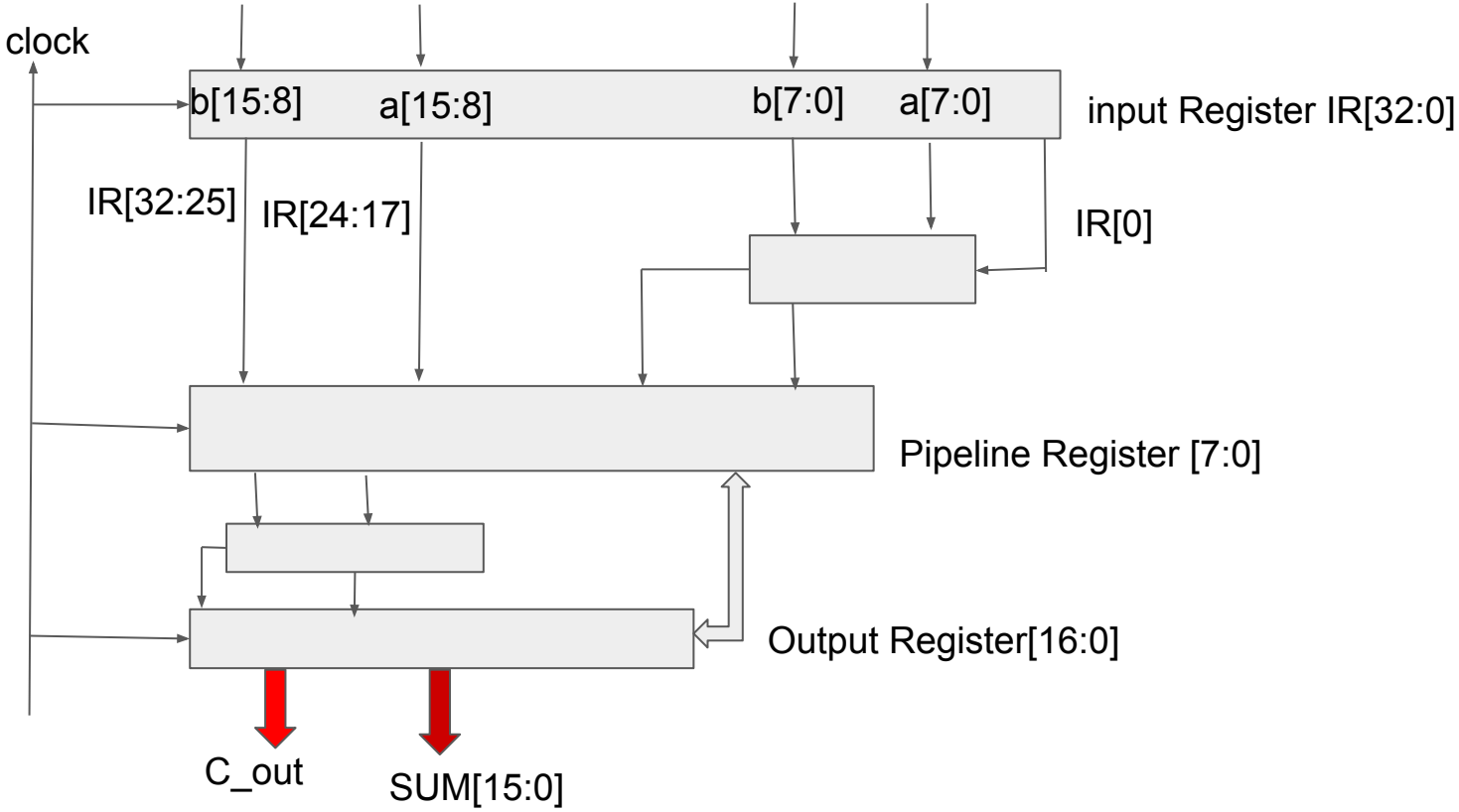
16 bit sum + Cout = 17 bit Result

A Pipelined 16 bit adder

Pipelining Approach

- Pipelining has been done to operate at higher throughput by distributing the processing over multiple cycles of the clock
- Trade - off between speed and physical resources warrant this approach
- More registers needs to be used
- The architecture contains an additional register (PR) between the data input register (IR) and the data output registers.

Structure


clock
b[15:8]
a[15:8]
b[7:0]
a[7:0]
input Register IR[32:0]
IR[32:25]
IR[24:17]
IR[0]
Pipeline Register [7:0]
Output Register[16:0]
C_out
SUM[15:0]

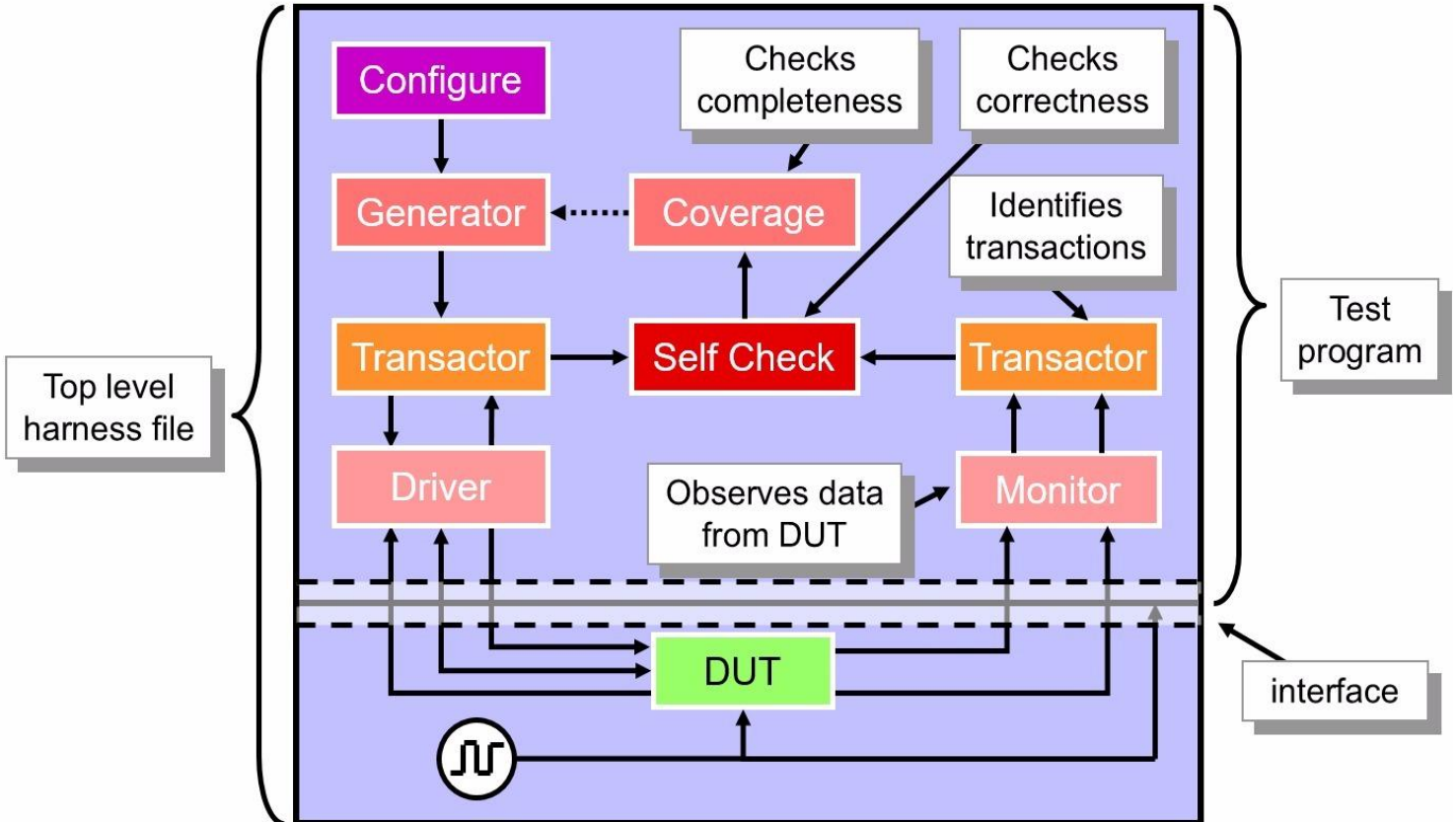
Design Code

```
module adder #(parameter

size    = 16,

half    = size/2,           //8
double  = 2*size,           //32
triple  = 3*half,           //24

size1 = half - 1,           //7
size2 = size - 1,           //15
size3 = half + 1,           //9
R1 = 1,                     //R1 = 1

L1 = half,                  //L1 = 8
R2 = size3,                 //R2 = 9
L2 = size,                  //L2 = 16
R3 = size + 1,              //R3 = 17
L3 = size + half,           //L3 = 24
R4 = double - half + 1,     //R4 = 25
L4 = double                 //L4 = 32

                            //L1 = 8; L2 = 16; L3 = 24; L4 = 32
                            //R1 = 1; R2 = 9;  R3 = 17; R4 = 25
)
```

```
(

input [15:0]     a, b,         //16 bit inputs
input c_in, clock,
output [size2:0]     sum,      // 16 bit output
output c_out

);

reg [double:0]          IR;    //33 bit IR
reg [triple:0]          PR;    //25 bit PR
reg [size:0]            OR;    //17 bit OR
assign {c_out, sum} = OR;    // 1 bit carry out + 16 bit sum = 17 bit OR
```

```
always @ (posedge clock) begin

//Load Input Register

IR[0] <= c_in;              // We are loading the 16 bit input data in a & b + carry in signal in the Input Register = 33 bit data
IR[L1:R1] <= a[size1:0];    //IR[8:1]   <= a[7:0]
IR[L2:R2] <= b[size1:0];    //IR[16:9]  <= b[7:0]
IR[L3:R3] <= a[size2: half]; //IR[24:17] <= a[15:8]
IR[L4:R4] <= b[size2: half]; //IR[32:25] <= b[15:8]

//Load Pipeline Register    //Piperline Register is 25 bit. We are loading the upper (upper 8 bit a, upper 8 bit b) bits of IR +
                            //8 bit a + 8 bit b + Cin bit = 9 bit data

PR[L3:R3] <= IR[L4:R4];     //PR[24:17] <= IR[32:25]
PR[L2:R2] <= IR[L3:R3];     //PR[16:9]  <= IR[24:17]
PR[half:0] <= IR[L2:R2] + IR[L1:R1] + IR[0];            //PR[8:0]  <= IR[16:9] + IR [8:1] + IR[0] //PR[8] is the Cout of 1st stage

// Load Output Register     //Output Register is 17 bit. addition of upper 8 bits of a, upper 8 bits of b + 9 bits from lower addition
OR <= {{1'b0,PR[L3:R3]}} + {{1'b0,PR[L2:R2]}} + PR[half];  //OR <= (0, 8) + (0, 8) + 1bit = 17 bits  //OR[16] is the Cout of last Stage

end
endmodule
```

The SystemVerilog Test Environment



Interface

```
interface adder_io(input logic clk);
  parameter simulation_cycle = 100;


  logic [15:0]  a;
  logic [15:0]  b;
  logic         c_in;

  logic [size2:0]   sum;
  logic         c_out;

  clocking cb @ (posedge clk);
    default input #1 output #1;
    output a;
    output b;
    output cin;
    input sum;
    input c_out;
  endclocking

  modport TB(output a,output b,output c_in,clocking cb,input sum,input c_out);

endinterface
```

```
module adder_test_top;

bit clk;

adder_io top_if(clk);
adder_tb t(top_if);

adder dut(
.a(top_if.a),
.b(top_if.b),
.c_in(top_if.c_in),
.clk(top_if.clk),
.sum(top_if.sum),
.c_out(top_if.c_out)
);

initial begin

forever begin
    #5
    clk = ~clk;
    end
  end
endmodule
```

Top Level

TESTBENCH

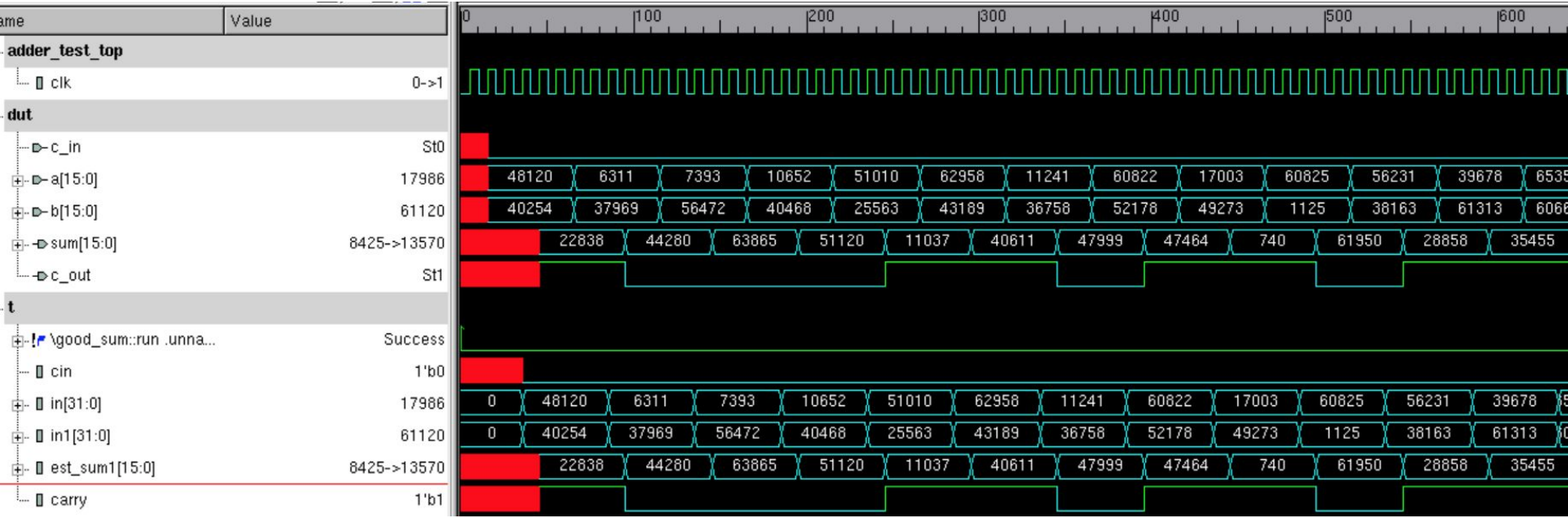
```
program automatic adder_tb(adder_io.TB adder_io);

int in=0;                                   //Value of input a from the design goes into in
int in1=0;                                  //Value of input b from the design goes into in1
logic cin;                                  //Carry In

logic [16:0]est_sum;                        //17 bit output
logic [15:0]est_sum1;                       //17 bit output seperated into 16 bit sum & 1 bit carry
logic carry;                                //Carry out

int j = 0;                                  // To repeat the operation 50 times

covergroup funct;                           //Functional Coverage for input a
    coverpoint adder_io.a
        {option.auto_bin_max = 4;}
endgroup

covergroup funct2;                           //Functional Coverage for input b
   coverpoint adder_io.b
        {option.auto_bin_max = 4;}
endgroup
```

```
class good_sum;
    rand int len[];                                  // Define Constraints for input a
    constraint c_len {foreach (len[i])
    len[i] inside {[1:65535]};
    len.sum() > 65535;
    len.size() inside {[10:1000]};}  ;

    rand int len1[];                                 // Define Constraints for input b
    constraint c_len1 {foreach (len1[k])
    len1[k] inside {[1:65535]};
    len1.sum() > 65535;
    len1.size() inside {[10:1000]};}

task run();                                         //Generate Random Values based on Constraints.
    assert(randomize());
endtask

endclass
```

```
task driveinput();                          // Drive the input signal @ clock edge of Clocking block
    begin
    @adder_io.cb;
    begin
    @adder_io.cb;
    adder_io.a = ua.len[j];                 // Random values into input a
    adder_io.b = ua.len1[j];                // Random values into input b
    adder_io.c_in = 1'b0;                   // Carry_In initialized to 0

    functa.sample();                        // Functional Covererage for input a
    functb.sample();                        // Functional Covererage for input b

    @adder_io.cb;
    @adder_io.cb;
    in = adder_io.a;
    in1 = adder_io.b;
    cin = adder_io.c_in;

    @adder_io.cb;
    est_sum = in+in1+cin;                   // Estimate the sum
    est_sum1 = est_sum[15:0];               //Seperate the sum & carry
    carry = est_sum[16];

    end
    end
endtask
```

```
task selftest();                                    // Self Check the Correctness of the design
                                                    // Compare the estimated sum with the Acquired Sum
    begin

    if (adder_io.sum==est_sum1 && adder_io.c_out==carry)
        $display("design correct");
    else $display("design incorrect");

    end
endtask

good_sum ua = new();                                // new instance of class
funct functa = new();                               // new instance of functional coverage for a
funct2 functb = new();                              // new instance of functional coverage for b

initial begin

ua.run();                                           // Generate the random values
repeat(50)                                          // repeat the below operation 50 times to get 100% functional Coverage
begin
driveinput();                                       // Drive the inputs with the constrained random stimulus & estimate the sum
selftest();
overflow();                                         // Check for overflow condition

j = j+1;                                            // increment the counter for repeating 50 times
end
end
endprogram
```

WAVEFORM


ame
Value
adder_test_top
clk
0->1
dut
c_in
St0
a[15:0]
17986
b[15:0]
61120
sum[15:0]
8425->13570
c_out
St1
t
\good_sum::run .unna...
Success
cin
1'b0
in[31:0]
17986
in1[31:0]
61120
est_sum1[15:0]
8425->13570
carry
1'b1
0
100
200
300
400
500
600
48120 6311 7393 10652 51010 62958 11241 60822 17003 60825 56231 39678
40254 37969 56472 40468 25563 43189 36758 52178 49273 1125 38163 61313
22838 44280 63865 51120 11037 40611 47999 47464 740 61950 28858 35455
0 48120 6311 7393 10652 51010 62958 11241 60822 17003 60825 56231 39678
0 40254 37969 56472 40468 25563 43189 36758 52178 49273 1125 38163 61313
22838 44280 63865 51120 11037 40611 47999 47464 740 61950 28858 35455

SELF CHECK

```
Running simv -V
Chronologic VCS simulator copyright 1991-2012
Contains Synopsys proprietary information.
Compiler version G-2012.09-SP1; Runtime version G-2012.09-SP1;  Dec 13 10:08 2015
VCS Build Date = Feb 25 2013 20:36:30
Start run at Dec 13 10:08 2015

design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
design correct
```

Functionality Coverage

Total Groups Coverage Summary

SCORE	WEIGHT
100.00	1

Total groups in report: 2

SCORE	WEIGHT	GOAL	NAME
100.00	1	100	adder_test_top.t::funct
100.00	1	100	adder_test_top.t::funct2

Summary for Group adder_test_top.t::funct

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	4	0	4	100.00

Variables for Group adder_test_top.t::funct

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
adder_io.a	4	0	4	100.00	100	1

Go to top

Summary for Variable adder_io.a

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	4	0	4	100.00

Automatically Generated Bins for adder_io.a

Bins

NAME	COUNT	AT LEAST
auto[0:16383]	13	1
auto[16384:32767]	11	1
auto[32768:49151]	11	1
auto[49152:65535]	15	1

Summary for Group adder_test_top.t::funct2

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Variables	4	0	4	100.00

Variables for Group adder_test_top.t::funct2

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL	WEIGHT
adder_io.b	4	0	4	100.00	100	1

Go to top

Summary for Variable adder_io.b

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	4	0	4	100.00

Automatically Generated Bins for adder_io.b

Bins

NAME	COUNT	AT LEAST
auto[0:16383]	11	1
auto[16384:32767]	9	1
auto[32768:49151]	16	1
auto[49152:65535]	14	1

DESIGN FLOW AND SPECIFICATIONS

- In our Design , Carry Signal becomes high (1) if the addition of integers a and b exceeds 16’hFFFF or 65535.
- No carry is Generated if sum of a and b is less than 65535.
- The Functionality Cover points are the Inputs that is a and b.
- The Constraints are such that we get a fairly large number such that the addition overflows & generates a carry

FUNCTIONALITY COVERAGE

- A and B are set as Functional Coverage Points
- Auto bins were defined. The entire range(0:65535) is divided into 4 bins & ensured complete coverage of the driving inputs.
- 100% Functional Coverage achieved.

Project Timeline and Completion

- **Verification Planning**
- **Interface Module**
- **Systemverilog Testbench**
- **Task for Randomization**
- **Constraint Random Stimulus Generation**
- **Building Top Module**
- **Task for Driving Inputs to DUT**
- **Task for Self Checking**
- **Functional Coverage**