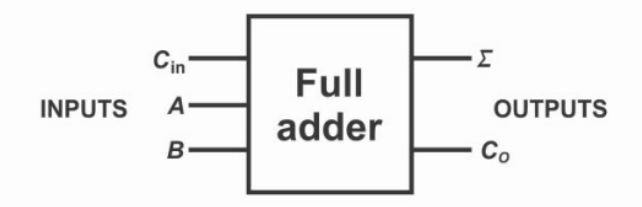
1 Half Adder

2 Full Adder

Bộ Full Adder là một mạch logic số dùng để cộng 3-bit nhị phân, thường đầu vào được biểu diễn dưới dạng A, B, và C_{in} (bit nhớ), đầu ra được biểu diễn dưới dạng SUM(s) và C_{out} (bit nhớ). Với SUM(s) $= A \oplus B \oplus C_{in}$ và $C_{out} = (A\&B)\|(C_{in}\&(A \oplus B))$.



Hình 1: Bộ Full Adder

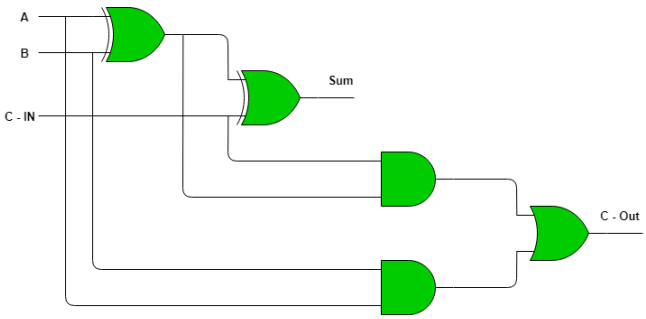
Bảng sự thật:

Input			Output	
A	B	C_{in}	SUM(s)	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Bảng 1: Bảng sự thật của bộ Full Adder

Thực hiện bộ Full Adder:

- Bộ Full Adder thực hiện bằng cổng logic:



Hình 2: Bộ Full Adder thực hiện bằng cổng logic

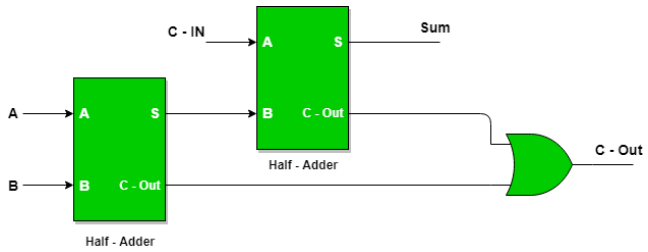
```
1 module full_adder_circuit(
2     input logic      i_data_a,
3     input logic      i_data_b,
```

```
    input logic      i_carry,

    output logic     o_data,
    output logic     o_carry
);

wire xor1;

assign xor1 = i_data_a ^ i_data_b;
assign o_data = xor1 ^ i_carry;
assign o_carry = (i_data_a & i_data_b) | (i_carry & xor1);

endmodule
```

Listing 1: Bộ Full Adder thực hiện bằng cổng logic

logic

- Bộ Full Adder thực hiện bằng bộ Half Adder:



Hình 3: Bộ Full Adder thực hiện bằng các bộ Half Adder

```
module full_adder_with_half_adder(
    input logic      i_data_a,
    input logic      i_data_b,
    input logic      i_carry,

    output logic     o_data,
    output logic     o_carry
);

wire w1, w2, w3;
half_adder adder_1(
    .i_data_a(i_data_a),
    .i_data_b(i_data_b),

    .o_data(w1),
    .o_carry(w2)
);

half_adder adder_2(
    .i_data_a(i_carry),
    .i_data_b(w1),

    .o_data(o_data),
    .o_carry(w3)
);

assign o_carry = w3 | w2;

```

```
endmodule

module half_adder(
    input logic     i_data_a,
    input logic     i_data_b,

    output logic    o_data,
    output logic    o_carry
);

assign o_data = i_data_a ^ i_data_b;    // S = A XOR B
assign o_carry = i_data_a & i_data_b;   // C = A AND B

endmodule
```

Listing 2: Bộ Full Adder thực hiện bằng bộ Half Adder

Sử dụng các test case sau để kiểm tra hệ thống:

```
#include <iostream>
#include <verilated.h>
#include "VName_moduel.h" // Verilator-generated header file

int main(int argc, char **argv) {
  // Initialize Verilator
  Verilated::commandArgs(argc, argv);

  // Create an instance of the module
  VName_moduel* dut = new VName_moduel;

  // Input test vectors
  int test_cases[8][3] = {
    {0, 0, 0}, // {i_data_a, i_data_b, i_carry}
    {0, 0, 1},
    {0, 1, 0},
    {0, 1, 1},
    {1, 0, 0},
    {1, 0, 1},
    {1, 1, 0},
    {1, 1, 1},
  };

  // Print table header
  std::cout << "A B Cin | Sum Cou" << std::endl;
  std::cout << "--------|--------" << std::endl;

  // Run test cases
  for (int i = 0; i < 8; ++i) {
    // Apply inputs
    dut->i_data_a = test_cases[i][0];
    dut->i_data_b = test_cases[i][1];
    dut->i_carry  = test_cases[i][2];

    // Evaluate the circuit
    dut->eval();

    // Display inputs and outputs
    std::cout << test_cases[i][0] << " "
    << test_cases[i][1] << "   "
    << test_cases[i][2] << "  |   "
    << (int)dut->o_data << "   "
```

```
43        << (int)dut->o_carry << std::endl;
44    }
45
46    // Cleanup
47    delete dut;
48
49    return 0;
50  }
```

Listing 3: Test bench của bộ Full Adder

Kết quả:

```
1  $ ./obj_dir/Vfull_adder_circuit
2  A B Cin | Sum Cou
3  --------|--------
4  0 0   0  |   0   0
5  0 0   1  |   1   0
6  0 1   0  |   1   0
7  0 1   1  |   0   1
8  1 0   0  |   1   0
9  1 0   1  |   0   1
10 1 1   0  |   0   1
11 1 1   1  |   1   1
```

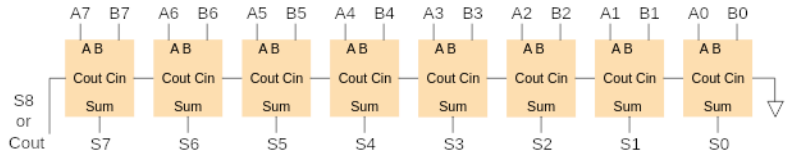
Listing 4: Kết quả của test bench của bộ Full Adder sử dụng cổng logic

```
1  $ ./obj_dir/Vfull_adder_with_half_adder
2  A B Cin | Sum Cou
3  --------|--------
4  0 0   0  |   0   0
5  0 0   1  |   1   0
6  0 1   0  |   1   0
7  0 1   1  |   0   1
8  1 0   0  |   1   0
9  1 0   1  |   0   1
10 1 1   0  |   0   1
11 1 1   1  |   1   1
```

Listing 5: Kết quả của test bench của bộ Full Adder sử dụng bộ Half Adder

3 Ripple Carry Adder (RCA)

Ripple Carry Adder (RCA) là một bộ cộng số học trong thiết kế mạch số, được sử dụng để cộng hai số nhị phân. RCA hoạt động dựa trên nguyên tắc tính toán carry (bit nhớ) theo kiểu tuần tự (ripple), tức là carry của mỗi bit phụ thuộc vào carry từ bit trước đó. Bộ Ripple Carry Adder được cấu tạo từ nhiều bộ Full Adder (FA) kết nối tuần tự với nhau.



Hình 4: Cấu trúc bộ Ripple Carry Adder

Trong đó:

- Tổng (S_i): $S_i = A_i \oplus B_i \oplus C_i$.
- Carry (C_{i+1}): $C_{i+1} = (A_i \& B_i) + (C_i \& (A_i \oplus B_i))$
- Ưu điểm:
  + Thiết kế đơn giản, mỗi bit được xử lý bằng bộ Full Adder được kết nối theo chuỗi.
  + Tiết kiệm tài nguyên phần cứng, phù hợp cho các hoạt động không yêu cầu quá cao về hiệu suất.
  + Dễ dàng mở rộng mà không cần thay đổi quá nhiều vào logic của mạch.
  + Tiết kiệm chi phí vì sử dụng ít cổng logic.
- Nhược điểm:
  + Độ trễ cao do phải xử lý tuần tự, độ trễ tăng tuyến tính theo n-bit, với độ trễ được tính bằng
  $$T_{delay} = n \times T_{unit}$$
  với, T_{unit} là độ trễ của từng khối trong RCA.
  + Hiệu suất thấp đối với số lượng bit lớn.
  + Không tối ưu khi hệ thống yêu cầu thời gian thực vì có độ trễ lớn.
  + Giới hạn hoạt động do tính tuần tự nên hệ thống RCA bị giới hạn tần số hoạt động, không đáp ứng ở hệ thống hoạt động ở tần số cao.

```
module rca (
input logic [31:0]  i_data_a,  // Operand A
input logic [31:0]  i_data_b,  // Operand B
output logic [31:0] o_data     // Sum output
);

logic [31:0] carry; // Carry signals

// Instance of the first full adder (LSB)
full_adder FA0 (
.i_data_a(i_data_a[0]),
.i_data_b(i_data_b[0]),
.i_carry(1'b0),           // Initial carry-in is 0
.o_data(o_data[0]),
.o_carry(carry[0])
```

```
16  );
17
18  // Instances of remaining 31 full adders
19  full_adder FA1  (.i_data_a(i_data_a[1]),  .i_data_b(i_data_b[1]),  .
        i_carry(carry[0]), .o_data(o_data[1]),  .o_carry(carry[1]));
20  full_adder FA2  (.i_data_a(i_data_a[2]),  .i_data_b(i_data_b[2]),  .
        i_carry(carry[1]), .o_data(o_data[2]),  .o_carry(carry[2]));
21  full_adder FA3  (.i_data_a(i_data_a[3]),  .i_data_b(i_data_b[3]),  .
        i_carry(carry[2]), .o_data(o_data[3]),  .o_carry(carry[3]));
22  full_adder FA4  (.i_data_a(i_data_a[4]),  .i_data_b(i_data_b[4]),  .
        i_carry(carry[3]), .o_data(o_data[4]),  .o_carry(carry[4]));
23  full_adder FA5  (.i_data_a(i_data_a[5]),  .i_data_b(i_data_b[5]),  .
        i_carry(carry[4]), .o_data(o_data[5]),  .o_carry(carry[5]));
24  full_adder FA6  (.i_data_a(i_data_a[6]),  .i_data_b(i_data_b[6]),  .
        i_carry(carry[5]), .o_data(o_data[6]),  .o_carry(carry[6]));
25  full_adder FA7  (.i_data_a(i_data_a[7]),  .i_data_b(i_data_b[7]),  .
        i_carry(carry[6]), .o_data(o_data[7]),  .o_carry(carry[7]));
26  full_adder FA8  (.i_data_a(i_data_a[8]),  .i_data_b(i_data_b[8]),  .
        i_carry(carry[7]), .o_data(o_data[8]),  .o_carry(carry[8]));
27  full_adder FA9  (.i_data_a(i_data_a[9]),  .i_data_b(i_data_b[9]),  .
        i_carry(carry[8]), .o_data(o_data[9]),  .o_carry(carry[9]));
28  full_adder FA10 (.i_data_a(i_data_a[10]), .i_data_b(i_data_b[10]), .
        i_carry(carry[9]), .o_data(o_data[10]), .o_carry(carry[10]));
29  full_adder FA11 (.i_data_a(i_data_a[11]), .i_data_b(i_data_b[11]), .
        i_carry(carry[10]), .o_data(o_data[11]), .o_carry(carry[11]));
30  full_adder FA12 (.i_data_a(i_data_a[12]), .i_data_b(i_data_b[12]), .
        i_carry(carry[11]), .o_data(o_data[12]), .o_carry(carry[12]));
31  full_adder FA13 (.i_data_a(i_data_a[13]), .i_data_b(i_data_b[13]), .
        i_carry(carry[12]), .o_data(o_data[13]), .o_carry(carry[13]));
32  full_adder FA14 (.i_data_a(i_data_a[14]), .i_data_b(i_data_b[14]), .
        i_carry(carry[13]), .o_data(o_data[14]), .o_carry(carry[14]));
33  full_adder FA15 (.i_data_a(i_data_a[15]), .i_data_b(i_data_b[15]), .
        i_carry(carry[14]), .o_data(o_data[15]), .o_carry(carry[15]));
34  full_adder FA16 (.i_data_a(i_data_a[16]), .i_data_b(i_data_b[16]), .
        i_carry(carry[15]), .o_data(o_data[16]), .o_carry(carry[16]));
35  full_adder FA17 (.i_data_a(i_data_a[17]), .i_data_b(i_data_b[17]), .
        i_carry(carry[16]), .o_data(o_data[17]), .o_carry(carry[17]));
36  full_adder FA18 (.i_data_a(i_data_a[18]), .i_data_b(i_data_b[18]), .
        i_carry(carry[17]), .o_data(o_data[18]), .o_carry(carry[18]));
37  full_adder FA19 (.i_data_a(i_data_a[19]), .i_data_b(i_data_b[19]), .
        i_carry(carry[18]), .o_data(o_data[19]), .o_carry(carry[19]));
38  full_adder FA20 (.i_data_a(i_data_a[20]), .i_data_b(i_data_b[20]), .
        i_carry(carry[19]), .o_data(o_data[20]), .o_carry(carry[20]));
39  full_adder FA21 (.i_data_a(i_data_a[21]), .i_data_b(i_data_b[21]), .
        i_carry(carry[20]), .o_data(o_data[21]), .o_carry(carry[21]));
40  full_adder FA22 (.i_data_a(i_data_a[22]), .i_data_b(i_data_b[22]), .
        i_carry(carry[21]), .o_data(o_data[22]), .o_carry(carry[22]));
41  full_adder FA23 (.i_data_a(i_data_a[23]), .i_data_b(i_data_b[23]), .
        i_carry(carry[22]), .o_data(o_data[23]), .o_carry(carry[23]));
42  full_adder FA24 (.i_data_a(i_data_a[24]), .i_data_b(i_data_b[24]), .
        i_carry(carry[23]), .o_data(o_data[24]), .o_carry(carry[24]));
43  full_adder FA25 (.i_data_a(i_data_a[25]), .i_data_b(i_data_b[25]), .
        i_carry(carry[24]), .o_data(o_data[25]), .o_carry(carry[25]));
44  full_adder FA26 (.i_data_a(i_data_a[26]), .i_data_b(i_data_b[26]), .
        i_carry(carry[25]), .o_data(o_data[26]), .o_carry(carry[26]));
45  full_adder FA27 (.i_data_a(i_data_a[27]), .i_data_b(i_data_b[27]), .
        i_carry(carry[26]), .o_data(o_data[27]), .o_carry(carry[27]));
46  full_adder FA28 (.i_data_a(i_data_a[28]), .i_data_b(i_data_b[28]), .
        i_carry(carry[27]), .o_data(o_data[28]), .o_carry(carry[28]));
47  full_adder FA29 (.i_data_a(i_data_a[29]), .i_data_b(i_data_b[29]), .
```

```
    i_carry(carry[28]), .o_data(o_data[29]), .o_carry(carry[29]));
48  full_adder FA30 (.i_data_a(i_data_a[30]), .i_data_b(i_data_b[30]), .
    i_carry(carry[29]), .o_data(o_data[30]), .o_carry(carry[30]));
49  full_adder FA31 (.i_data_a(i_data_a[31]), .i_data_b(i_data_b[31]), .
    i_carry(carry[30]), .o_data(o_data[31]), .o_carry()); // Final carry
    ignored
50
51  endmodule
52
53  module full_adder(
54  input logic     i_data_a,
55  input logic     i_data_b,
56  input logic     i_carry,
57
58  output logic    o_data,
59  output logic    o_carry
60  );
61
62  wire xor1;
63
64  assign xor1 = i_data_a ^ i_data_b;
65  assign o_data = xor1 ^ i_carry;
66  assign o_carry = (i_data_a & i_data_b) | (i_carry & xor1);
67
68  endmodule
```

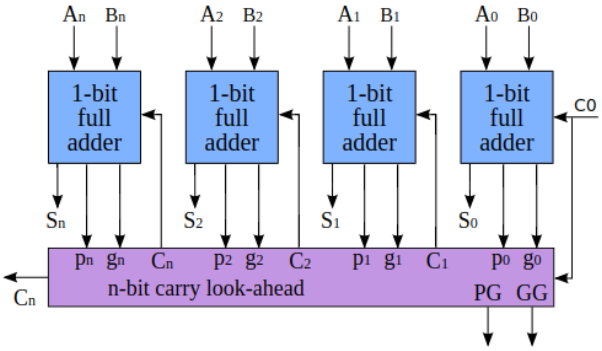
Listing 6: RCA

```
1   $ ./obj_dir/Vrca
2   === Test bench for Ripple Carry Adder (RCA) ===
3   [PASS] Test case 1: A = 149, B = 186, Expected = 335, Actual = 335
4   [PASS] Test case 2: A = 143, B = 254, Expected = 397, Actual = 397
5   [PASS] Test case 3: A = 4, B = 0, Expected = 4, Actual = 4
6   ...
7   [PASS] Test case 97: A = 165, B = 214, Expected = 379, Actual = 379
8   [PASS] Test case 98: A = 120, B = 128, Expected = 248, Actual = 248
9   [PASS] Test case 99: A = 116, B = 42, Expected = 158, Actual = 158
10  [PASS] Test case 100: A = 36, B = 2, Expected = 38, Actual = 38
11
12  === Test Summary ===
13  Total Test Cases: 100
14  PASS: 100
15  FAIL: 0
```

Listing 7: Kết quả của test

4 Carry Look-Ahead Adder (CLA)

Carry Look-Ahead Adder (CLA) là một loại bộ cộng số học trong kỹ thuật số, được thiết kế để thực hiện phép cộng hai số nhị phân nhanh hơn so với bộ cộng thông thường (như Ripple Carry Adder - RCA). CLA đạt được tốc độ cao bằng cách tính toán các tín hiệu carry đồng thời (song song) thay vì tuần tự, nhờ vào các tín hiệu Generate (G) và Propagate (P).



Hình 5: Cấu trúc của bộ Carry Look-Ahead

Trong đó,

- Generate (G_i): $G_i = A_i \cdot B_i$i.
- Propagate (P_i): $P_i = A_i + B_i$.
- Tổng (S_i): $S_i = G_i \cdot P_i$.
- Carry (C_{i+1}): $C_{i+1} = G_i + P_i \cdot C_i$.
- Ưu điểm:
  + Tốc độ xử lý cao do tín hiệu carry được xử lý song song.
  + Giảm độ trễ với độ trễ tăng theo $O(\log_2(n))$, thay vì $O(n)$ như RCA.
- Nhược điểm:
  + Phức tạp về thiết kế do yêu cầu xử dụng nhiều cổng logic để tính toán carry đồng thời, làm tăng độ phức tạp của hệ thống.
  + Tốn tài nguyên phần cứng do số lượng cổng logic tăng nhanh khi số bit tăng, làm tăng chi phí phần cứng.
  + Khó mở rộng do khi số bit lớn thì yêu cầu cần thiết kế phức tạp hơn, và việc tính toán carry đòi hỏi nhiều tài nguyên.

```
1  module cla (
2  input  logic [31:0] i_data_a,   // Operand A (32-bit input)
3  input  logic [31:0] i_data_b,   // Operand B (32-bit input)
4  output logic [31:0] o_data,     // Sum output (32-bit)
5  output logic        o_carry     // Carry-out (1-bit output)
6  );
7  logic [31:0] G, P;              // Bitwise Generate (G) and Propagate (P)
       signals
8  logic [32:0] C;                 // Carry signals for each bit (including
       carry-out)
9
10 // Generate and Propagate logic
11 // G[i] = i_data_a[i] & i_data_b[i]: A carry is generated when both bits
       are 1.
12 // P[i] = i_data_a[i] | i_data_b[i]: A carry is propagated if at least one
        of the bits is 1.
13 assign G = i_data_a & i_data_b; // Generate signals
14 assign P = i_data_a | i_data_b; // Propagate signals
```

```
15
16  // Carry logic
17  // C[0] is initialized to 0 because there is no carry-in for the least
        significant bit.
18  // C[i+1] = G[i] | (P[i] & C[i]): Carry for the next bit depends on the
        current bit's generate or propagate conditions.
19  assign C[0] = 0; // Initial carry-in is 0
20  generate
21  genvar i;
22  for (i = 0; i < 32; i++) begin : carry_logic_block
23  // Compute carry-out for each bit position
24  assign C[i+1] = G[i] | (P[i] & C[i]);
25  end
26  endgenerate
27
28  // Sum logic
29  // o_data[i] = i_data_a[i] ^ i_data_b[i] ^ C[i]: The sum is computed using
        the XOR of the two operands and the carry-in for each bit.
30  assign o_data = i_data_a ^ i_data_b ^ C[31:0]; // Sum computation
31  assign o_carry = C[32]; // The final carry-out from the most significant
        bit
32  endmodule
```

Listing 8: CLA

```
1   $ ./obj_dir/Vcla
2   === Test bench for Ripple Carry Adder (RCA) ===
3   [PASS] Test case 1: A = 234, B = 25, Expected = 259, Actual = 259
4   [PASS] Test case 2: A = 30, B = 255, Expected = 285, Actual = 285
5   [PASS] Test case 3: A = 120, B = 28, Expected = 148, Actual = 148
6   [PASS] Test case 4: A = 194, B = 123, Expected = 317, Actual = 317
7   ...
8   [PASS] Test case 99: A = 85, B = 175, Expected = 260, Actual = 260
9   [PASS] Test case 100: A = 101, B = 30, Expected = 131, Actual = 131
10
11  === Test Summary ===
12  Total Test Cases: 100
13  PASS: 100
14  FAIL: 0
```

Listing 9: Kết quả test

5 Kogge-Stone Adder (KSA)

Kogge-Stone Adder (KSA) là một bộ cộng song song hiệu suất cao được thiết kế để tính toán tín hiệu carry một cách nhanh chóng bằng cách sử dụng cấu trúc prefix network. Đây là một trong những loại Parallel Prefix Adders được sử dụng phổ biến nhất trong các bộ xử lý hiện đại, nhờ khả năng giảm độ trễ tính toán của tín hiệu carry xuống mức tối thiểu.

- Ưu điểm:
  + Tốc độ xử lý nhanh, độ trễ tính toán carry giảm xuống $O(\log_2(n))$, rất nhanh đối với số bit lớn.
  + Khả năng tính toán trong từng cấp có thể thực hiện đồng thời, cải thiện hiệu suất.

+ Thích hợp cho tính toán số bit lớn như 32-bit, 64-bit, 128-bit hoặc lớn.

- Nhược điểm:

+ Tài nguyên phần cứng lớn do cần nhiều cổng logic rất lớn, đặc biết đối với các số bit lớn.
+ Độ phức tạp thiết kế cao do cấu trúc mạng prèĩ đòi hỏi thiết kế phức tạp, khó tối ưu hóa cho chi phí.
+ Tiêu thụ năng lượng cao do sử dụng lượng lớn cổng logic và phép tính đồng thời.

6 Index

```cpp
#include <iostream>
#include <verilated.h>
#include <cstdlib>  // For rand() and srand()
#include <ctime>    // For time()

#include "Vrca.h"

int main(int argc, char** argv) {
  // Initialize Verilator
  Verilated::commandArgs(argc, argv);
  std::cout << "=== Test bench for Ripple Carry Adder (RCA) ===" << std::
      endl;

  // Create an instance of the RCA module
  Vrca* dut = new Vrca;

  // Initialize random seed
  srand(static_cast<unsigned>(time(0)));

  // Variables to track results
  int pass_count = 0;
  int fail_count = 0;

  // Run 100 test cases
  for (int i = 0; i < 100; ++i) {
    // Generate random inputs
    int a = rand() % 256;  // Random 8-bit value (0 to 255)
    int b = rand() % 256;  // Random 8-bit value (0 to 255)
    int expected_sum = a + b;  // Expected result

    // Apply inputs to the DUT
    dut->i_data_a = a;
    dut->i_data_b = b;

    // Evaluate the DUT
    dut->eval();

    // Capture output
    int actual_sum = dut->o_data;

    // Check for correctness
    if (actual_sum == expected_sum) {
      ++pass_count;
```

```
43          std::cout << "[PASS] Test case " << i + 1 << ": A = " << a << ", B =
                " << b
44          << ", Expected = " << expected_sum << ", Actual = " << actual_sum <<
                std::endl;
45        } else {
46          ++fail_count;
47          std::cout << "[FAIL] Test case " << i + 1 << ": A = " << a << ", B =
                " << b
48          << ", Expected = " << expected_sum << ", Actual = " << actual_sum <<
                std::endl;
49        }
50      }
51
52      // Display summary
53      std::cout << "\n=== Test Summary ===" << std::endl;
54      std::cout << "Total Test Cases: 100" << std::endl;
55      std::cout << "PASS: " << pass_count << std::endl;
56      std::cout << "FAIL: " << fail_count << std::endl;
57
58      // Cleanup
59      delete dut;
60      return 0;
61  }
```

Listing 10: Test case cho Bộ cộng