Project report

On

16-bit Parallel Prefix Adder – Implementation in Verilog

submitted by

T V S Sweta

EC032

&

Abinav Vadaliya

EC035

under the supervision of

Mr. Nikhil Kothari Sir

Department of Electronics and Communication

Dharmsinh Desai University

Nadiad

Dharmsinh Desai University

Faculty of Technology

College Road, Nadiad – 387001. (Gujarat)

Certificate

This is to certify that the practical / term work carried out in the subject of Summer Internship Report (SIP) and recorded in this journal is the bonafide work of Miss **T V S Sweta,** Roll No:**EC032** Identity No: **21ECUOG041** of B. Tech. Semester-VII in the branch of Electronics & Communication during the academic year 2024 - 2025.

(Prof. Nikhil Kothari) (Dr. Purvang Dalal)

Staff In-Charge Head of the Department

Date: Date:

Dharmsinh Desai University

Faculty of Technology

College Road, Nadiad – 387001. (Gujarat)

Certificate

This is to certify that the practical / term work carried out in the subject of Summer Internship Report (SIP) and recorded in this journal is the bonafide work of Mr **Abhinav Vadaliya,** Roll No:**EC035** Identity No: **21ECUSG074** of B. Tech. Semester-VII in the branch of Electronics & Communication during the academic year 2024 - 2025.

(Prof. Nikhil Kothari) (Dr. Purvang Dalal)

Staff In-Charge Head of the Department

Date: Date:

Abstract

In digital design, efficient arithmetic operations are crucial for high-performance computing systems, with adders being fundamental components in arithmetic units. Traditional adders like Ripple Carry Adders (RCA) and Carry Look-ahead Adders (CLA) present trade-offs between simplicity, speed, and complexity. To address these limitations, the Parallel Prefix Adder (PPA) offers a balanced approach, minimizing both time and area complexities. This project report explores the design and implementation of PPAs, which perform fast binary addition through parallel computation of carry bits. By leveraging prefix computation, intermediate results are pre-computed and combined hierarchically, significantly reducing propagation delay compared to traditional adders. The report covers an introduction to basic adder types, detailed PPA architecture, design variants such as Kogge-Stone and Brent-Kung adders, implementation and simulation steps, and a comparative analysis of PPAs against RCAs and CLAs in terms of speed, area, and power consumption. Additionally, it discusses practical applications of PPAs in modern processors and potential areas for further research and optimization. The findings highlight the advantages of PPAs in achieving high-speed arithmetic operations with manageable complexity, making them valuable in advanced digital systems.

Index

Chapter 1: Background

1.1: Motivation

The motivation behind this project stems from the ever-increasing demand for faster and more efficient computing systems. As digital technology advances, the need for high-speed arithmetic operations becomes paramount, particularly in processors and other critical components of modern computing devices. Traditional adders, such as Ripple Carry Adders (RCA) and Carry Look-ahead Adders (CLA), while effective, present significant trade-offs between simplicity, speed, and complexity. These limitations necessitate the exploration of more advanced adder designs.

Parallel Prefix Adders (PPAs) offer a promising solution by addressing the inefficiencies of traditional adders. PPAs leverage parallel computation of carry bits, significantly reducing propagation delay and enhancing overall performance. This project aims to delve into the design and implementation of PPAs, providing a comprehensive understanding of their architecture and benefits. By exploring various PPA designs, such as Kogge-Stone and Brent-Kung adders, and comparing them with traditional adders, this project seeks to highlight the advantages of PPAs in achieving high-speed arithmetic operations with manageable complexity.

1.2 : Literature Review

Parallel Prefix Circuits are essential for optimizing digital arithmetic operations by reducing the number of components while maintaining speed. These circuits implement downsizing techniques, transforming complex sequences into more efficient forms, which significantly reduces the number of components required without compromising speed.

Parallel Prefix Adders (PPAs) leverage this principle by using the sum generated from previous bits to determine the carry-in for the current bits. In a full adder, the sum aligns perfectly with the prefix circuit methodology. Over the years, various architectures have been developed to calculate carry bits efficiently. Notable designs include the J. Sklansky Conditional Adder (1960), Kogge-Stone Adder (1973), Ladner-Fischer Adder (1980), Brent-Kung Adder (1982), Han-Carlson Adder (1987), and S. Knowles Adder (1999).

Among these, the J. Sklansky Conditional Adder and the Ladner-Fischer Adder stand out for their modular design, low depth, and high fan-out nodes, making them particularly suitable for implementation. These architectures offer a balanced approach to achieving minimal time complexity and reduced component usage, making them ideal for high-performance digital systems. This literature review underscores the evolution and significance of PPAs in digital design, highlighting their role in enhancing computational efficiency and performance.

1.3: Basic Theory

The propagate and generate signals are first computed using the input bits as:

gi = ai & bi

pi = ai | bi.

Henceforth, for every 'circle' in the below diagram, the propagate and generate signals

are:

pi = pi&pj

gi= pi&gj | gi.

And the final sum is calculated as: si = g(i-1) ^ ai ^ bi**.**

Chapter 2: Problem Definition and Design

The objective of this project is to design and implement a 16-bit Parallel Prefix Adder (PPA) that optimizes both speed and area complexity. The 16-bit PPA aims to overcome these limitations by leveraging the principles of parallel prefix computation, which allows for the simultaneous calculation of carry bits, thereby reducing the overall addition time.

The specific goals of this project include:

- Design Efficiency: Develop a 16-bit PPA that minimizes the number of logic gates and interconnections required, ensuring a compact and efficient design.
- Speed Optimization: Achieve faster computation times by reducing the propagation delay through parallel processing of carry bits.
- Comparative Analysis: Evaluate the performance of the 16-bit PPA against traditional adder designs (RCA and CLA) in terms of speed, area, and power consumption.
- Implementation and Testing: Implement the 16-bit PPA using hardware description languages (HDLs) and validate its functionality through simulation and testing.

2.1 : Design

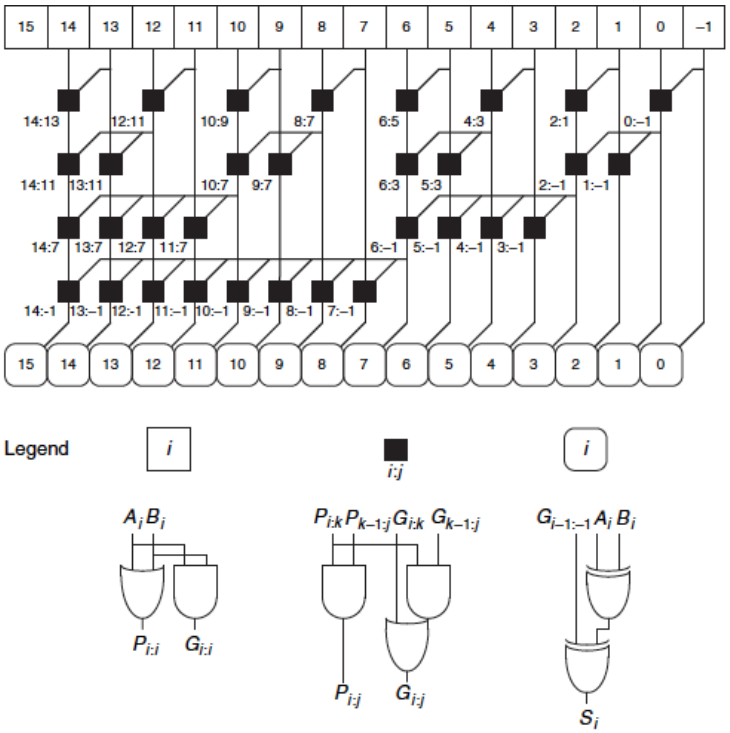


Fig 2.1. 16 bit Parallel Prefix Adder circuit diagram

Chapter 3: Test Setup & Methodology

To validate the design and performance of the 16-bit Parallel Prefix Adder (PPA), a comprehensive test setup and methodology are essential. This section outlines the steps and procedures used to ensure the accuracy, efficiency, and reliability of the PPA.

3.1 : Test Setup

3.1.1 : Simulation Environment

- HDL Modeling:

  Usede a hardware description language (HDL) like Verilog to create a model of the 16-bit Parallel Prefix Adder (PPA). This involves writing code that describes the behavior and structure of the PPA at a hardware level.

- Simulation Tools:

  Employed simulation tools such as ModelSim or Xilinx Vivado. These tools allow you to simulate the PPA design, providing a virtual environment to test and verify its functionality before actual hardware implementation.

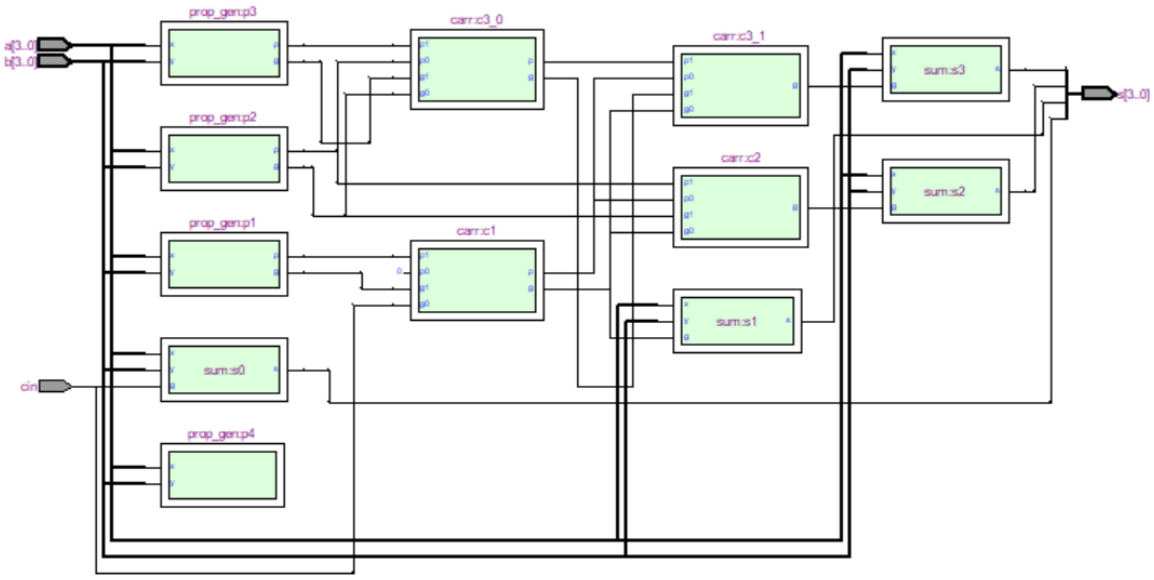


Fig 3.1.1 4 bit circuit diagram

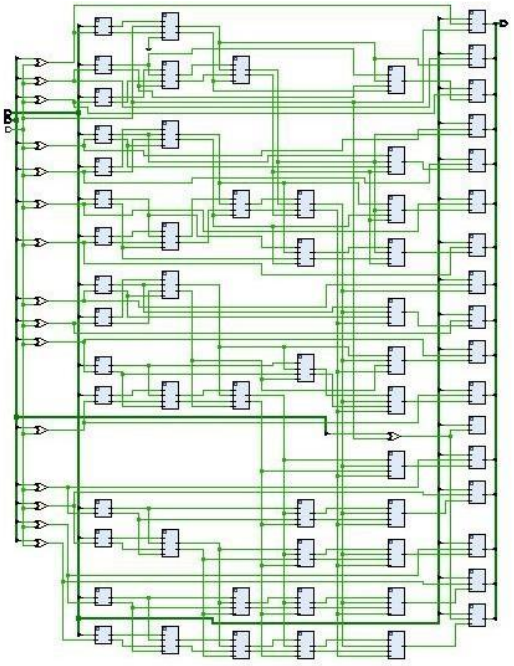


Fig 3.1.2 16 bit netlist viewer

3.1.2 : Test Bench Development

- Creating a Test Bench:

  Developed a test bench, which is a piece of code that generates various input vectors to test the PPA. This test bench was applied to different sets of inputs of the PPA and observed the outputs to ensure it behaves as expected.

- Edge Cases:

  Included specific test cases such as:

  All Zeros: Input vectors where all bits are zero.

  All Ones: Input vectors where all bits are one.

  Alternating Bits: Input vectors with alternating 0s and 1s (e.g., 01010101).

  Random Bit Patterns: Input vectors with random combinations of 0s and 1s.

  These edge cases help ensure that the PPA is thoroughly tested under various conditions.

3.1.3 : Reference Models

- Traditional Adders:

  Implemented reference models for traditional adders like Ripple Carry Adders (RCA). This model served as a baseline to compare the performance and accuracy of the PPA.

3.2: Methodology

- Functional Verification:

  Simulated the PPA using the test bench to verify its functionality. This involves running the simulation and checking if the PPA produces correct outputs for all input combinations.

  Compared the outputs of the PPA with the expected results from the reference models (RCA) to ensure the PPA is functioning correctly.

- Performance Analysis:

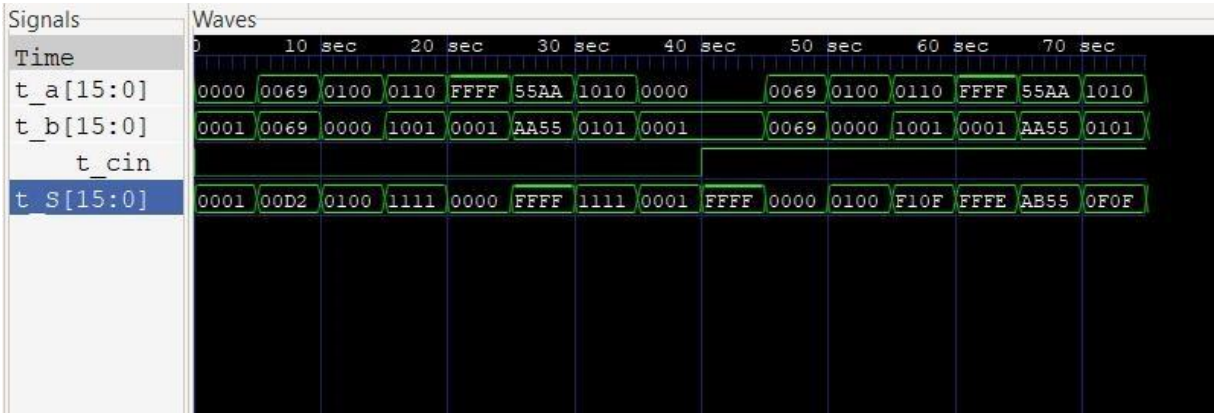
  Measured the propagation delay of the PPA, which is the time it takes for an input change to affect the output.

  Compared this delay with the propagation delay of the RCA using Xilinx Vivado, measured in nanoseconds (nsec).

  Calculated the percentage difference in delay between the PPA and RCA to evaluate the performance improvement.

Chapter 4: Result and discussion

4.1: Testbench verification



- **PARALLEL PREFIX ADDERS :**

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	6	7	5	a[3]	S[11]	7.751	4.133	3.617	∞	input port clock
Path 2	∞	6	7	5	a[3]	S[12]	7.751	4.133	3.617	∞	input port clock
Path 3	∞	6	7	5	a[3]	S[13]	7.751	4.133	3.617	∞	input port clock
Path 4	∞	6	7	5	a[3]	S[14]	7.751	4.133	3.617	∞	input port clock
Path 5	∞	6	7	5	a[3]	S[15]	7.751	4.133	3.617	∞	input port clock
Path 6	∞	6	7	4	a[3]	S[10]	7.558	4.105	3.452	∞	input port clock
Path 7	∞	6	7	4	a[3]	S[8]	7.558	4.105	3.452	∞	input port clock

- **RIPPLE CARRY ADDERS :**

Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	∞	10	11	3	a[0]	cout	9.468	4.599	4.868	∞	input port clock
Path 2	∞	10	11	3	a[0]	sum[14]	9.468	4.599	4.868	∞	input port clock
Path 3	∞	10	11	3	a[0]	sum[15]	9.462	4.593	4.868	∞	input port clock
Path 4	∞	9	10	3	a[0]	sum[12]	8.883	4.481	4.401	∞	input port clock
Path 5	∞	9	10	3	a[0]	sum[13]	8.883	4.481	4.401	∞	input port clock
Path 6	∞	8	9	3	a[0]	sum[10]	8.298	4.363	3.934	∞	input port clock
Path 7	∞	8	9	3	a[0]	sum[11]	8.298	4.363	3.934	∞	input port clock

Total Time (Path Delay):It includes all the delays involved in the path, including logic, routing, and other components.

Logic Time: This represents the delay introduced by the combinational logic elements (such as LUTs, AND gates, OR gates) in the design.

Net Time (Routing Delay): It measures the time it takes for signals to travel through the physical wires or nets between components. The larger the design, the more significant net delays may become.

PPA is approximately 22% faster than conventional adders.

The circuit of an RCA is simpler (fewer logic gates), the sequential nature of carry propagation means that each bit addition depends on the completion of the previous bit's carry, causing significant delays.

A PPA, on the other hand, uses more complex logic and a larger number of gates, but it performs the carry computation in parallel across multiple bits, improving the overall speed despite the higher complexity. As the number of bits increases, the delay increases linearly, making it unsuitable for high-speed, wide-bit-width adders.

PPA: The logarithmic delay growth with respect to the bit width makes PPAs much more scalable, particularly in large word-width designs (e.g., 16-bit, 32-bit, or 64-bit adders).

Chapter 5: Conclusion

Based on practical observations, the 16-bit Parallel Prefix Adder (PPA) demonstrates significant improvements in speed and efficiency compared to traditional adders like Ripple Carry Adders (RCA) and Carry Look-ahead Adders (CLA). The PPA's ability to compute carry bits in parallel reduces propagation delay, resulting in faster addition operations. Additionally, the optimized design maintains manageable area and power consumption, making it a highly effective solution for high-performance digital systems. This project confirms that PPAs are a valuable advancement in digital arithmetic, offering a balanced approach to achieving both speed and efficiency.

Chapter 7: Annexure

```
module triangle (input wire a, b, output wire p, g);
   or or_1 (p,a,b);
   and and_1 (g,a,b);
endmodule

module box (input wire pi, gi, pj, gj, output wire P, G);
   wire temp;
   and and_1 (P,pi,pj);
   and and_2 (temp,pi, gj);
   or or_0 (G,gi, temp );
endmodule

module circle (input wire a, b, gi, output wire s);
   xor xor3_1 (s,a,b,gi);
endmodule

module test1 (input wire [15:0] a, b, input wire cin, output wire [15:0] S);
   wire [15:0] p, g;
   wire [15:0] b_res;
   xor xor_0 (b_res[0],b[0], cin);
   xor xor_1 (b_res[1],b[1], cin);
   xor xor_2 (b_res[2],b[2], cin);
   xor xor_3 (b_res[3],b[3], cin);
   xor xor_4 (b_res[4],b[4], cin);
   xor xor_5 (b_res[5],b[5], cin);
   xor xor_6 (b_res[6],b[6], cin);
   xor xor_7 (b_res[7],b[7], cin);
   xor xor_8 (b_res[8],b[8], cin);
   xor xor_9 (b_res[9],b[9], cin);
   xor xor_10 (b_res[10],b[10], cin);
   xor xor_11 (b_res[11],b[11], cin);
   xor xor_12 (b_res[12],b[12], cin);
   xor xor_13 (b_res[13],b[13], cin);
   xor xor_14 (b_res[14],b[14], cin);
   xor xor_15 (b_res[15],b[15], cin);
   triangle triangle_0 (a[0], b_res[0], p[0], g[0]);
   triangle triangle_1 (a[1], b_res[1], p[1], g[1]);
   triangle triangle_2 (a[2], b_res[2], p[2], g[2]);
   triangle triangle_3 (a[3], b_res[3], p[3], g[3]);
   triangle triangle_4 (a[4], b_res[4], p[4], g[4]);
   triangle triangle_5 (a[5], b_res[5], p[5], g[5]);
   triangle triangle_6 (a[6], b_res[6], p[6], g[6]);
   triangle triangle_7 (a[7], b_res[7], p[7], g[7]);
   triangle triangle_8 (a[8], b_res[8], p[8], g[8]);
   triangle triangle_9 (a[9], b_res[9], p[9], g[9]);
   triangle triangle_10 (a[10], b_res[10], p[10], g[10]);
   triangle triangle_11 (a[11], b_res[11], p[11], g[11]);
   triangle triangle_12 (a[12], b_res[12], p[12], g[12]);
   triangle triangle_13 (a[13], b_res[13], p[13], g[13]);
```

```
triangle triangle_14 (a[14], b_res[14], p[14], g[14]);
triangle triangle_15 (a[15], b_res[15], p[15], g[15]);
wire [7:0] lvl1_P, lvl1_G;
box box_lvl1_0 (p[0], g[0], 1'b0, cin, lvl1_P[0], lvl1_G[0]);
box box_lvl1_1 (p[2], g[2], p[1], g[1], lvl1_P[1], lvl1_G[1]);
box box_lvl1_2 (p[4], g[4], p[3], g[3], lvl1_P[2], lvl1_G[2]);
box box_lvl1_3 (p[6], g[6], p[5], g[5], lvl1_P[3], lvl1_G[3]);
box box_lvl1_4 (p[8], g[8], p[7], g[7], lvl1_P[4], lvl1_G[4]);
box box_lvl1_5 (p[10], g[10], p[9], g[9], lvl1_P[5], lvl1_G[5]);
box box_lvl1_6 (p[12], g[12], p[11], g[11], lvl1_P[6], lvl1_G[6]);
box box_lvl1_7 (p[14], g[14], p[13], g[13], lvl1_P[7], lvl1_G[7]);
wire [7:0] lvl2_P, lvl2_G;
box box_lvl2_0 (p[1], g[1], lvl1_P[0], lvl1_G[0], lvl2_P[0], lvl2_G[0]);
box box_lvl2_1 (lvl1_P[1], lvl1_G[1], lvl1_P[0], lvl1_G[0], lvl2_P[1], lvl2_G[1]);
box box_lvl2_2 (p[5], g[5], lvl1_P[2], lvl1_G[2], lvl2_P[2], lvl2_G[2]);
box box_lvl2_3 (lvl1_P[3], lvl1_G[3], lvl1_P[2], lvl1_G[2], lvl2_P[3], lvl2_G[3]);
box box_lvl2_4 (p[9], g[9], lvl1_P[4], lvl1_G[4], lvl2_P[4], lvl2_G[4]);
box box_lvl2_5 (lvl1_P[5], lvl1_G[5], lvl1_P[4], lvl1_G[4], lvl2_P[5], lvl2_G[5]);
box box_lvl2_6 (p[13], g[13], lvl1_P[6], lvl1_G[6], lvl2_P[6], lvl2_G[6]);
box box_lvl2_7 (lvl1_P[7], lvl1_G[7], lvl1_P[6], lvl1_G[6], lvl2_P[7], lvl2_G[7]);
wire [7:0] lvl3_P, lvl3_G;
box box_lvl3_0 (p[3], g[3], lvl2_P[1], lvl2_G[1], lvl3_P[0], lvl3_G[0]);
box box_lvl3_1 (lvl1_P[2], lvl1_G[2], lvl2_P[1], lvl2_G[1], lvl3_P[1], lvl3_G[1]);
box box_lvl3_2 (lvl2_P[2], lvl2_G[2], lvl2_P[1], lvl2_G[1], lvl3_P[2], lvl3_G[2]);
box box_lvl3_3 (lvl2_P[3], lvl2_G[3], lvl2_P[1], lvl2_G[1], lvl3_P[3], lvl3_G[3]);
box box_lvl3_4 (p[11], g[11], lvl2_P[5], lvl2_G[5], lvl3_P[4], lvl3_G[4]);
box box_lvl3_5 (lvl1_P[6], lvl1_G[6], lvl2_P[5], lvl2_G[5], lvl3_P[5], lvl3_G[5]);
box box_lvl3_6 (lvl2_P[6], lvl2_G[6], lvl2_P[5], lvl2_G[5], lvl3_P[6], lvl3_G[6]);
box box_lvl3_7 (lvl2_P[7], lvl2_G[7], lvl2_P[5], lvl2_G[5], lvl3_P[7], lvl3_G[7]);
wire [7:0] lvl4_P, lvl4_G;
box box_lvl4_0 (p[7], g[7], lvl3_P[3], lvl3_G[3], lvl4_P[0], lvl4_G[0]);
box box_lvl4_1 (lvl1_P[4], lvl1_G[4], lvl3_P[3], lvl3_G[3], lvl4_P[1], lvl4_G[1]);
box box_lvl4_2 (lvl2_P[4], lvl2_G[4], lvl3_P[3], lvl3_G[3], lvl4_P[2], lvl4_G[2]);
box box_lvl4_3 (lvl2_P[5], lvl2_G[5], lvl3_P[3], lvl3_G[3], lvl4_P[3], lvl4_G[3]);
box box_lvl4_4 (lvl3_P[4], lvl3_G[4], lvl3_P[3], lvl3_G[3], lvl4_P[4], lvl4_G[4]);
box box_lvl4_5 (lvl3_P[5], lvl3_G[5], lvl3_P[3], lvl3_G[3], lvl4_P[5], lvl4_G[5]);
box box_lvl4_6 (lvl3_P[6], lvl3_G[6], lvl3_P[3], lvl3_G[3], lvl4_P[6], lvl4_G[6]);
box box_lvl4_7 (lvl3_P[7], lvl3_G[7], lvl3_P[3], lvl3_G[3], lvl4_P[7], lvl4_G[7]);
circle circle_0 (a[0], b_res[0], cin, S[0]);
circle circle_1 (a[1], b_res[1], lvl1_G[0], S[1]);
circle circle_2 (a[2], b_res[2], lvl2_G[0], S[2]);
circle circle_3 (a[3], b_res[3], lvl2_G[1], S[3]);
circle circle_4 (a[4], b_res[4], lvl3_G[0], S[4]);
circle circle_5 (a[5], b_res[5], lvl3_G[1], S[5]);
circle circle_6 (a[6], b_res[6], lvl3_G[2], S[6]);
circle circle_7 (a[7], b_res[7], lvl3_G[3], S[7]);
circle circle_8 (a[8], b_res[8], lvl4_G[0], S[8]);
circle circle_9 (a[9], b_res[9], lvl4_G[1], S[9]);
circle circle_10 (a[10], b_res[10], lvl4_G[2], S[10]);
circle circle_11 (a[11], b_res[11], lvl4_G[3], S[11]);
```

```
    circle circle_12 (a[12], b_res[12], lvl4_G[4], S[12]);
    circle circle_13 (a[13], b_res[13], lvl4_G[5], S[13]);
    circle circle_14 (a[14], b_res[14], lvl4_G[6], S[14]);
    circle circle_15 (a[15], b_res[15], lvl4_G[7], S[15]);
endmodule

// TESTBENCH
module prefix_tb;
    reg [15:0] t_a, t_b;
    reg t_cin;
    wire [15:0] t_S;

    initial begin
        $dumpfile("testbench.vcd");
        $dumpvars(0, prefix_tb);

    end

    // Instantiate the test1 module
    test1 as16 (.a(t_a), .b(t_b), .cin(t_cin), .S(t_S));

    initial
    begin
        t_a = 16'h0000; t_b = 16'h0001; t_cin = 1'b0;
        #5
        t_a = 16'h0069; t_b = 16'h0069; t_cin = 1'b0;
        #5
        t_a = 16'h0100; t_b = 16'h0000; t_cin = 1'b0;
        #5
        t_a = 16'h0110; t_b = 16'h1001; t_cin = 1'b0;
        #5
        t_a = 16'hffff; t_b = 16'h0001; t_cin = 1'b0;
        #5
        t_a = 16'h55aa; t_b = 16'haa55; t_cin = 1'b0;
        #5
        t_a = 16'h1010; t_b = 16'h0101; t_cin = 1'b0;
        #5
        t_a = 16'h0000; t_b = 16'h0001; t_cin = 1'b0;
        #5
        t_a = 16'h0000; t_b = 16'h0001; t_cin = 1'b1;
        #5
        t_a = 16'h0069; t_b = 16'h0069; t_cin = 1'b1;
        #5
        t_a = 16'h0100; t_b = 16'h0000; t_cin = 1'b1;
        #5
        t_a = 16'h0110; t_b = 16'h1001; t_cin = 1'b1;
        #5
        t_a = 16'hffff; t_b = 16'h0001; t_cin = 1'b1;
        #5
        t_a = 16'h55aa; t_b = 16'haa55; t_cin = 1'b1;
```

```
        #5
        t_a = 16'h1010; t_b = 16'h0101; t_cin = 1'b1;
        #5
        t_a = 16'h0000; t_b = 16'h0001; t_cin = 1'b1;
    end

    initial
    begin
        $monitor($time, "a = %h, b = %h, cin = %b, sum = %h", t_a, t_b, t_cin, t_S);
    end
endmodule
```