Carry Skip Adder

A FPGA Project

Team Members

Dutt Panchal

Yagnesh Hirpara

Nayan Satasiya

Project Mentor

Prabhas Barman,

Verification Engineer at

Scaledge Technology

Date

20-06-2025

Table of Contents

1. Abstract

This project aims to design and implement an efficient Carry Skip Adder (CSKA) using Verilog HDL and deploy it on the Zybo Z7 FPGA board. Carry Skip Adders are optimized adders that improve the delay performance of ripple carry adders by allowing the carry to "skip" across blocks of bits. The final design was synthesized, simulated, and implemented using Xilinx Vivado Design Suite and tested successfully on hardware.

2. Introduction

2.1 Objective

To design a fast 8-bit binary adder based on the Carry Skip Adder architecture and implement it on an FPGA using Verilog HDL and the Vivado toolchain.

2.2 Motivation

While Ripple Carry Adders are simple to design, they are slow due to the sequential carry propagation. CSKA offers a middle ground between speed and complexity by skipping blocks when all bits propagate.

2.3 Applications

- High-speed arithmetic units
- DSP processors
- Embedded systems with constrained area and power budgets

3. Theoretical Background

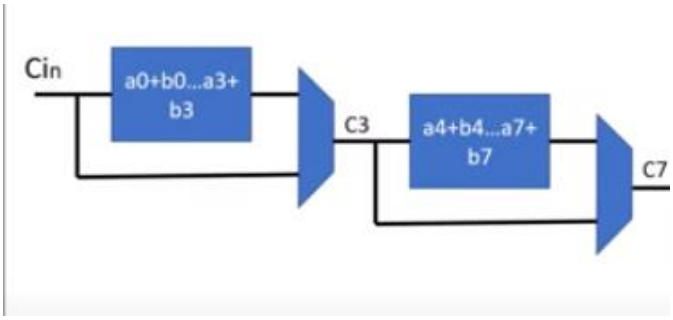
3.1 Carry Skip Adder Overview

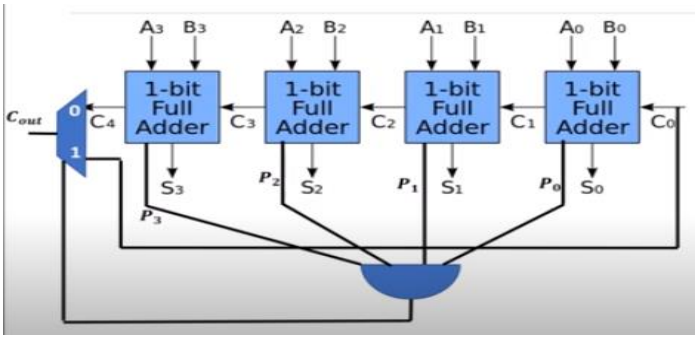
A Carry Skip Adder improves performance by dividing the adder into blocks and allowing the carry to skip over blocks under certain conditions.

Block Architecture:

- Each 4-bit block computes partial sums and a propagate signal.
- Carry-out is determined either from the last full adder or directly from the carry-in if the propagate signal is high.

3.2 Logic Diagram





4. Design Specification

4.1 Design Parameters

- Number of bits: 8
- Number of blocks: 2 (4-bit each)
- Input: Two 8-bit binary numbers, Carry-in
- Output: 8-bit sum, Carry-out

4.2 HDL Language

Verilog (IEEE Std 1364-2001)

4.3 Specification Overview

A Carry Skip Adder (CSKA) is an improved type of digital adder that reduces the propagation delay of carries in multi-bit binary addition by allowing the carry to "skip" over certain blocks of bits when no carry propagation is needed.

It divides the full adder into blocks, and each block generates a block propagate signal that determines whether the carry can bypass the entire block, thus enhancing speed compared to a ripple carry adder.

Consider an 8-bit Carry Skip Adder divided into two 4-bit blocks:

Block 1 (bits 0–3) and Block 2 (bits 4–7) each compute:

Sum bits using full adders.
Blocks propagate signal: true if all bits in the block propagate the carry.

If the block propagate of Block 1 is high, and a carry-in exists, the carry can skip directly to Block 2, bypassing internal delays of Block 1.

➡️ Advantage: Faster addition than ripple carry adders, especially in wider bit-width adders, with relatively lower complexity than carry-lookahead adders.

5. Hardware Platform

5.1 Board: Zybo Z7-10 / Z7-20

- Zynq-7000 SoC (FPGA + ARM Cortex-A9)
- USB-JTAG programmer
- 1GB DDR3, HDMI, Pmod, switches, LEDs
- Supported by Vivado WebPACK

5.2 FPGA Resources Used (Post Implementation)

We are using 5 input switches (4 for two 2-bits inputs and 1 for Carry-in) and three output LED pins (2 LEDs for 2 bit Sum output and 1 for Carry Output).

6. Design Implementation

6.1 Verilog Code

Below is the Verilog design code for 2-bit carry skip adder that we have actually implemented on FPGA.

```
// carry look ahead adder
module cla_block #(parameter WIDTH = 2)
(
	input [WIDTH-1:0] A, B, input Cin,
	output [WIDTH-1:0] Sum, output Cout, output P_block // Propagate block signal );
wire [WIDTH-1:0] P, G, C;
assign P = A ^ B; assign G = A & B;
assign C[0] = Cin;
genvar i;
generate for (i = 1; i < WIDTH; i = i + 1) begin
	assign C[i] = G[i-1] | (P[i-1] & C[i-1]);
end
endgenerate
assign Sum = P ^ C;
assign Cout = G[WIDTH-1] | (P[WIDTH-1] & C[WIDTH-1-1]);
```

```
assign P_block = &P;

endmodule

// carry skip addder

module cska_top #(parameter N = 2, BLOCK_SIZE = 2)

(

        input [N-1:0] A, B, input Cin,

        output [N-1:0] Sum, output Cout

);

localparam BLOCKS = N / BLOCK_SIZE;

wire [BLOCKS:0] carry;

wire [BLOCKS-1:0] propagate;

assign carry[0] = Cin;

genvar i;

generate for (i = 0; i < BLOCKS; i = i + 1) begin : cska_block

        wire [BLOCK_SIZE-1:0] sum_temp;

        wire block_carry, block_p;

 cla_block #(.WIDTH(BLOCK_SIZE)) cla_inst (
    .A    (A[i*BLOCK_SIZE +: BLOCK_SIZE]),
    .B    (B[i*BLOCK_SIZE +: BLOCK_SIZE]),
    .Cin  (carry[i]),
    .Sum  (sum_temp),
    .Cout (block_carry),
    .P_block (block_p)
  );

  assign Sum[i*BLOCK_SIZE +: BLOCK_SIZE] = sum_temp;
  assign propagate[i] = block_p;

  assign carry[i+1] = block_p ? carry[i] : block_carry;
```

```
end


endgenerate

assign Cout = carry[BLOCKS];

endmodule
```

6.2 Code Explanation

The design consists of two modules:

1. cla_block – Carry Lookahead Adder Block

- **Function**: Adds two inputs A and B of parameterized width (WIDTH), along with a carry-in (Cin).
- **Internal Logic**:
  - Generates **propagate (P = A ^ B)** and **generate (G = A & B)** signals.
  - Computes carry signals (C) using lookahead logic:
    *C[i]=G[i−1]+(P[i−1]·C[i−1])C[i] = G[i-1] + (P[i-1] \cdot C[i-1])*C[i]=G[i−1]+(P[i−1]·C[i−1])
  - Outputs the sum: *Sum=P⊕CSum = P \oplus C*Sum=P⊕C
  - Outputs block propagate signal P_block = &P (high if all bits propagate).

2. cska_top – Carry Skip Adder Top Module

- **Function**: Builds a multi-block carry skip adder using instances of cla_block.
- **Parameters**:
  - N: Total number of bits (e.g., 2)
  - BLOCK_SIZE: Bits per block (e.g., 2 → 1 block only)
- **Working**:
  - Each cla_block computes sum and local carry-out.
  - If all bits propagate (P_block = 1), the incoming carry is directly skipped to the next block.
  - Otherwise, the internal carry is used.
  - Outputs: final Sum[N-1:0] and Cout.

7. Simulation and Functional Verification

7.1 Testbench Code

```
`timescale 1ns / 1ps

module tb_cska;

// 2-bit Carry Skip Adder

parameter N = 2;

parameter BLOCK_SIZE = 2;

reg [N-1:0] A, B; reg Cin; wire [N-1:0] Sum;

wire Cout;

// DUT (Design Under Test)

cska_top #(.N(N), .BLOCK_SIZE(BLOCK_SIZE)) dut
( .A(A), .B(B), .Cin(Cin), .Sum(Sum), .Cout(Cout) );

// Waveform generation

initial begin

        $dumpfile("cska_2bit.vcd");

        $dumpvars(0, tb_cska);

end

// Display task

task show_result;

        begin

                $display("Time = %0t | A = %b | B = %b | Cin = %b => Sum = %b | Cout
        = %b", $time, A, B, Cin, Sum, Cout);

        end

endtask
```

```
initial begin

// All zeros
if ($test$plusargs("ALL_ZERO")) begin
  A = 2'b00; B = 2'b00; Cin = 0;
  #10; show_result();
end

// All ones
if ($test$plusargs("ALL_ONES")) begin
  A = 2'b11; B = 2'b11; Cin = 1;
  #10; show_result();
end

// One operand zero
if ($test$plusargs("ONE_OPER_ZERO")) begin
  A = 2'b00; B = 2'b10; Cin = 0;
  #10; show_result();
end

// One operand zero with Cin = 1
if ($test$plusargs("ONE_OPER_ZERO_CIN")) begin
  A = 2'b00; B = 2'b01; Cin = 1;
  #10; show_result();
end

// Only Cin high
if ($test$plusargs("CIN_ONLY")) begin
  A = 2'b00; B = 2'b00; Cin = 1;
  #10; show_result();
end

// Max sum with carry out
if ($test$plusargs("MAX_SUM")) begin
  A = 2'b11; B = 2'b00; Cin = 1;
  #10; show_result();
end

// Carry skip behavior
if ($test$plusargs("SKIP_CASE")) begin
  A = 2'b11; B = 2'b00; Cin = 1;
```

```
  #10; show_result();
end

// No propagation
if ($test$plusargs("NO_PROP")) begin
  A = 2'b10; B = 2'b01; Cin = 0;
  #10; show_result();
end

// Overflow-like behavior
if ($test$plusargs("OVRFLW")) begin
  A = 2'b10; B = 2'b10; Cin = 1;
  #10; show_result();
end

// Random test (repeat 5 times)
if ($test$plusargs("RANDOM")) begin
  repeat (5) begin
    A = $random; B = $random; Cin = $random;
    #10; show_result();
  end
end

$finish;


end

endmodule
```

7.2 List of Implemented Testcases

1. All Zero and Ones Inputs
2. One Input Operand is Zero
3. One Input Operand is Zero + Carry In
4. Maximum Sum without Carry Out
5. Carry Generation in Lower Bits – Skip Enabled
6. All Propagation Bits High
7. No Propagation
8. Overflow Condition
9. Random Inputs and Cin

10. Alternating Bits Inputs
11. Reverse Bits Pattern Inputs

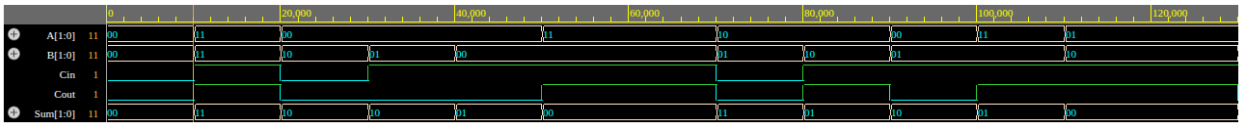
7.3 Simulation Results

1. Include waveform screenshots
2. Show sample inputs and expected vs. actual outputs
3. Verify block propagate and skip conditions

Output Log from EDA Playground: (Through Monitor)

```
Compiler version U-2023.03-SP2_Full64; Runtime version U-2023.03-SP2_Full64;  Jun 17 11:57 2025
Time = 10000 | A = 00 | B = 00 | Cin = 0 => Sum = 00 | Cout = 0
Time = 20000 | A = 11 | B = 11 | Cin = 1 => Sum = 11 | Cout = 1
Time = 30000 | A = 00 | B = 10 | Cin = 0 => Sum = 10 | Cout = 0
Time = 40000 | A = 00 | B = 01 | Cin = 1 => Sum = 10 | Cout = 0
Time = 50000 | A = 00 | B = 00 | Cin = 1 => Sum = 01 | Cout = 0
Time = 60000 | A = 11 | B = 00 | Cin = 1 => Sum = 00 | Cout = 1
Time = 70000 | A = 11 | B = 00 | Cin = 1 => Sum = 00 | Cout = 1
Time = 80000 | A = 10 | B = 01 | Cin = 0 => Sum = 11 | Cout = 0
Time = 90000 | A = 10 | B = 10 | Cin = 1 => Sum = 01 | Cout = 1
Time = 100000 | A = 00 | B = 01 | Cin = 1 => Sum = 10 | Cout = 0
Time = 110000 | A = 11 | B = 01 | Cin = 1 => Sum = 01 | Cout = 1
Time = 120000 | A = 01 | B = 10 | Cin = 1 => Sum = 00 | Cout = 1
Time = 130000 | A = 01 | B = 10 | Cin = 1 => Sum = 00 | Cout = 1
Time = 140000 | A = 01 | B = 00 | Cin = 1 => Sum = 10 | Cout = 0
```

Output Waveform from EDA Playground



EDA Playground Link: https://www.edaplayground.com/x/bMCH

8. Implementation on FPGA

Step-by-Step Guide to Implement Using Vivado

1. Create Project

- Open Vivado → Create New Project
- Name: Carry_Skip_Adder_8bit

- Choose “RTL Project” → Check “Do not specify sources at this time”

2. Add Design Files

- Add your Verilog design and testbench files
- Set top module as your main CSKA file

3. Add Constraints

- Use Zybo Z7 master XDC file
- Map switches to inputs and LEDs to outputs

4. Run Simulation

- Go to Flow Navigator → Simulation → Run Simulation
- Verify output matches expected results

5. Synthesize Design

- Run Synthesis → Check for warnings/errors

6. Implement Design

- Run Implementation → Analyze Timing Summary

7. Generate Bitstream

- Generate Bitstream for loading to board

8. Program FPGA

- Connect Zybo Z7 via USB
- Go to "Hardware Manager" → Open Target → Auto Connect
- Program Device → Select Bitstream → Done!

9. Result and Discussion

9.1 Hardware Verification

- Observed LED patterns for sum output
- Test different combinations using board switches
- Compare functional behavior with simulation

FPGA I/Os Used:

(We have implemented 2 bits input carry-skip-adder on FPGA as it is difficult to implement 8 bit carry skip adder)

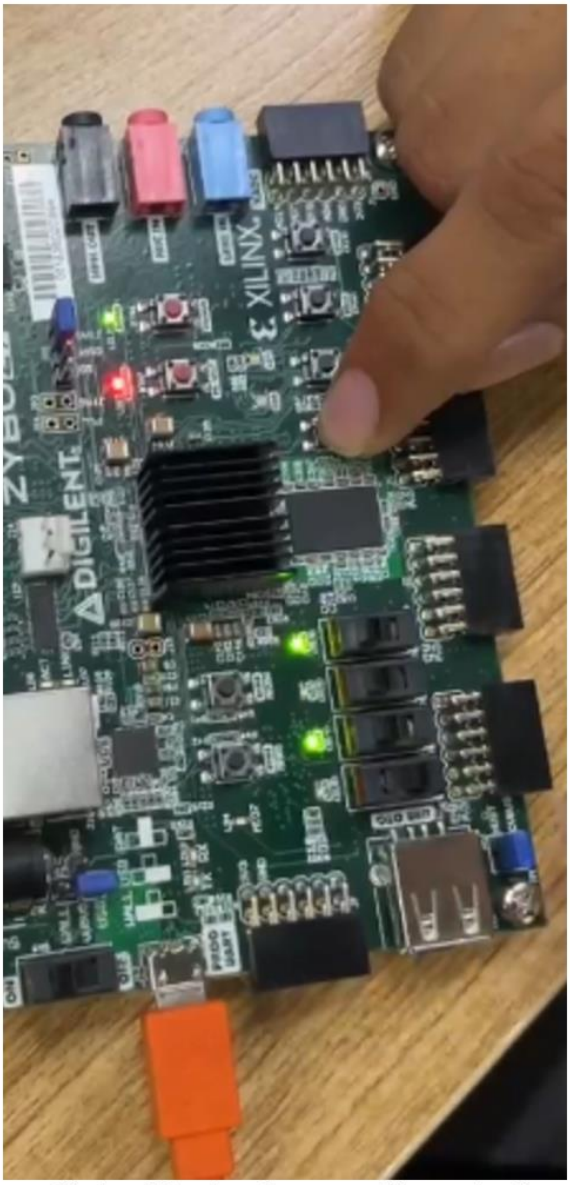
1. 4 Input Switches + 1 Push Button Switch

   - 2 Input Switches for input A (2 bit)
   - 2 Input Switches for input B (2 bit)
   - 1 Input Push Button Switch For Carry Input (1 Bit)

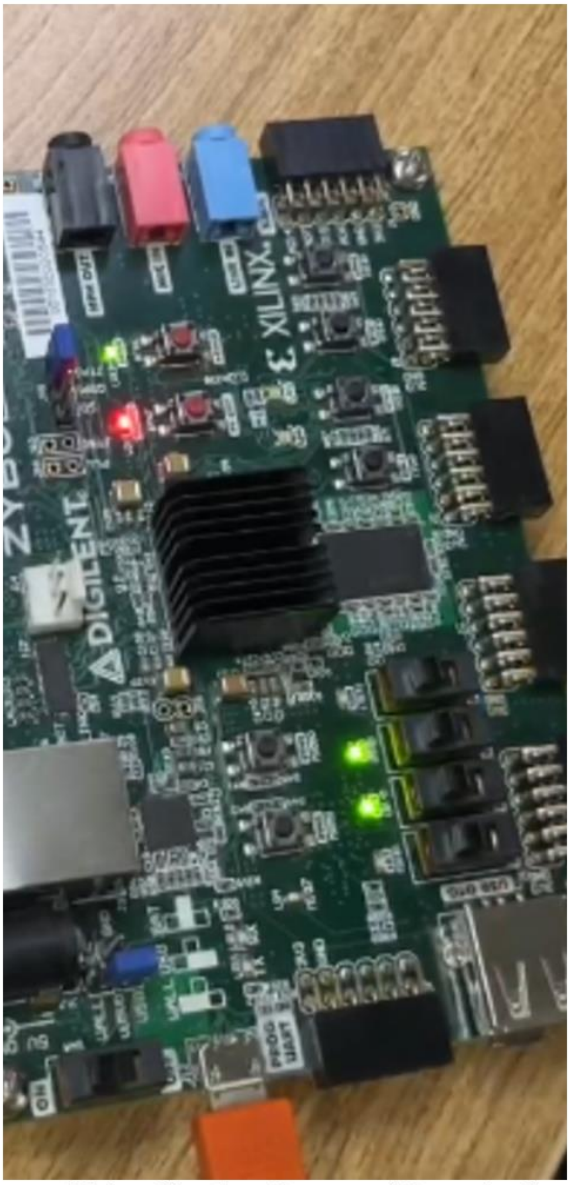
2. 3 In-built Output LEDs

   - 2 for Sum (2 bit) output
   - 1 for Carry output (1 bit)

9.2 FPGA Output Snapshots



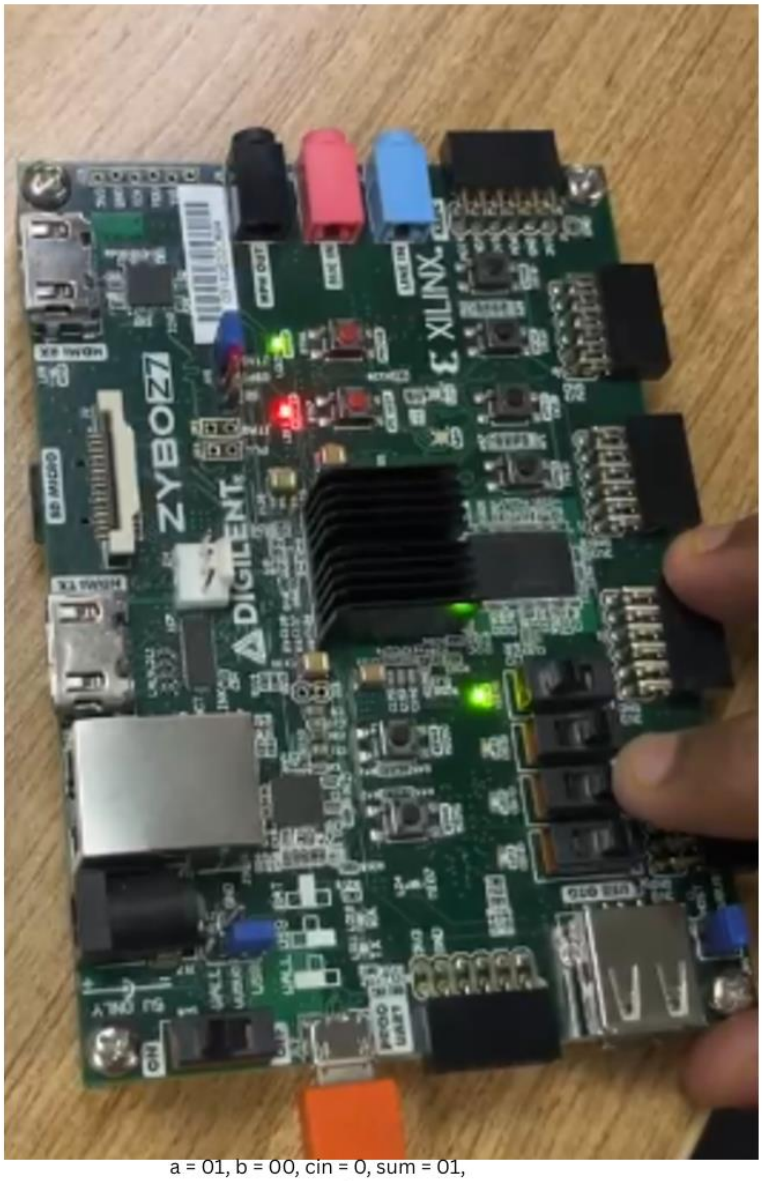
a = 11, b = 01, cin = 1, sum = 01, cout = 1



a = 11, b = 11, cin = 0, sum = 10, cout = 1

a = 01, b = 01, cin = 0, sum = 10, cout = 0

a = 11, b = 11, cin = 0, sum = 00, cout = 1



a = 01, b = 00, cin = 0, sum = 01, cout = 0

9.3 Live FPGA Implementation Simulation Video

Check out these links to watch live working and simulation of this design on FPGA Board with some explanation:

- Video No. 1
- Video No. 1

10. Conclusion

The 8-bit Carry Skip Adder was successfully implemented and verified on the Zybo Z7 FPGA. The design achieved better performance than a basic Ripple Carry Adder and demonstrated the advantages of block-based carry propagation in hardware-efficient addition circuits.

11. References

- Zybo Z7 Reference Manual, Digilent
- IEEE Std 1364-2001 Verilog HDL
- FPGA Prototyping by VHDL/Verilog Examples – Pong P. Chu
- Xilinx Vivado User Guide
- Carry-Skip-Adder YouTube Guide

Thank you.