Karunya INSTITUTE OF TECHNOLOGY AND SCIENCES
(Declared as Deemed to be University under Sec.3 of the UGC Act, 1956)
MoE, UGC & AICTE Approved
NAAC A++ Accredited

Division of Electronics and Communication Engineering
2024-2025 (ODD SEM)

REPORT

for

Digital IC Design

Title of the Report:

Design and Implementation of Segmentation Adder

A report submitted by

Name of the Student	**Brijin G, Gladwin Jebas J, Aldrin Infant Raj F**
Register Number	***URK22EC1021, URK22EC1031, URK22EC1069***
Subject Name	***Digital IC Design***
Subject Code	***18EC2019***
Date of Report submission	***23/04/2025***

Project Rubrics for Evaluation

First Review: Project title selection - PPT should have four slides (Title page, Introduction, Circuit/Block Diagram, and Description of Project).

Second Review: PPT should have three slides (Description of Concept, implementation, outputs, results and discussion)

Rubrics for project (III IA - 40 Marks):
Content - 4 marks (based on Project)
Clarity - 3 marks (based on viva during presentation) Feasibility - 3 marks (based on project)
Presentation - 10 marks Project Report - 10 marks
On-time submission - 5 marks (before the due date) Online submission-GCR - 5 marks

Total marks:_____/ 40 Marks

Signature of Faculty with date:

TABLE OF CONTENTS

CHAPTER	TITLE	PAGE NO.
1.	INTRODUCTION	4
2.	ASIC DESIGN FLOW BLOCK DIAGRAM	5
3.	DESCRIPTION OF THE PROCESS FLOW	6
4.	TOOLS AND SOFTWARES USED	9
5.	DESIGN METHODOLOGY	11
6.	RESULTS AND DISCUSSION	23
7.	CONCLUSION	29

Abstract

This project presents the design and implementation of a **Segmentation Adder**, an efficient digital circuit architecture aimed at optimizing speed and performance in arithmetic operations. Unlike traditional adders, the segmentation approach divides the input bits into multiple segments, allowing parallel computation of partial sums and minimizing carry propagation delay. The architecture was developed using Verilog HDL and implemented on both **FPGA** (via Xilinx Vivado) and **ASIC** platforms (using the Skywater 130nm PDK and OpenLane flow). Functional correctness was validated through waveform simulations, while hardware implementation metrics such as power, area, and timing were analyzed. The results confirm that the segmentation adder offers improved performance and scalability, making it a promising solution for high-speed digital systems.

CHAPTER 1

INTRODUCTION

In digital systems, arithmetic operations—particularly addition—are among the most fundamental and frequently executed tasks. As computational demands increase, there is a continuous need for high-speed and low-power arithmetic units. Traditional adder architectures such as ripple carry adders, carry lookahead adders, and carry select adders each offer trade-offs between speed, complexity, and resource utilization.

To overcome the limitations of long carry propagation delays in conventional designs, this project introduces a **Segmentation Adder** architecture. The core idea behind segmentation is to divide the input operands into smaller, manageable segments, which are processed independently and in parallel. This significantly reduces the longest path of carry propagation, resulting in faster computation.

The primary objective of this project is to design, simulate, and implement a segmentation-based adder using both FPGA and ASIC design flows. The implementation leverages **Verilog HDL** for RTL design, **Xilinx Vivado** for FPGA deployment, and the **Skywater 130nm PDK** with **OpenLane** for full ASIC synthesis, placement, and layout. By analyzing the functional correctness, power consumption, area, and timing characteristics of the proposed design, this project aims to demonstrate the efficiency and scalability of the segmentation adder in modern digital processing systems.

CHAPTER 2

ASIC DESIGN FLOW BLOCK DIAGRAM

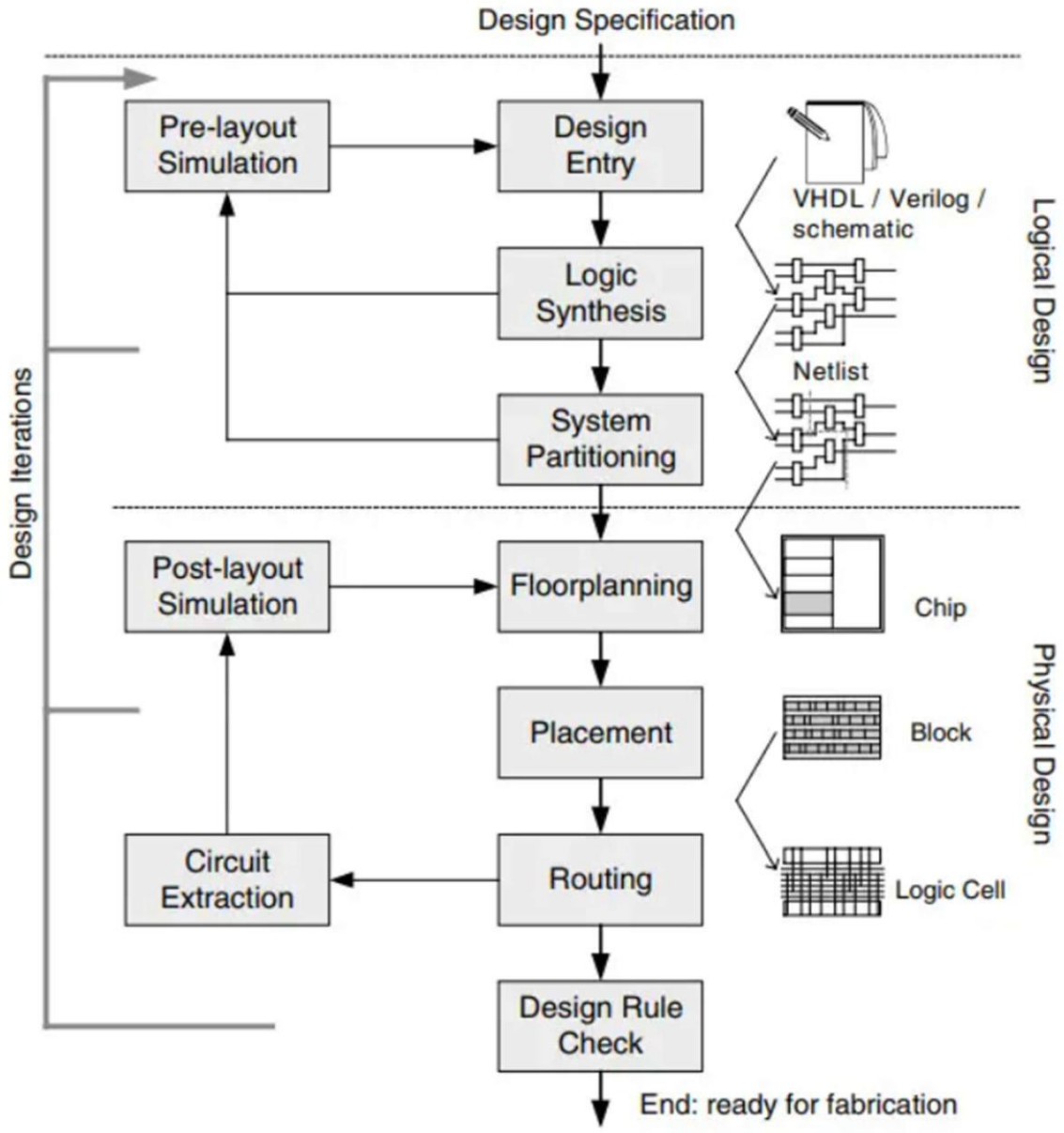


Fig.1 Block Diagram of the ASIC Design Flow

Fig.1 illustrates the VLSI (Very-Large-Scale Integration) design flow. It starts with **design specification**, followed by **logical design** steps like design entry, logic synthesis, and system partitioning. Then it proceeds to the **physical design** phase, including floorplanning, placement, routing, and design rule checks. Pre- and post-layout simulations ensure correctness, with **design iterations** allowing refinement before final fabrication.

CHAPTER 3

DESCRIPTION OF THE PROCESS FLOW

Segmentation Adder:

A Segmentation Adder is a digital circuit used in arithmetic operations, particularly in VLSI designs, to enhance speed and power efficiency. The main idea is to divide a large adder into smaller segments and selectively activate only the segments necessary for a given operation. This concept is often used in low-power design, like in embedded systems or ASICs.

Key Concept:

Instead of having one large adder working all the time (like in a ripple-carry or carry-lookahead adder), the segmentation adder segments the input bits and processes only relevant parts, which reduces dynamic power consumption and delays.

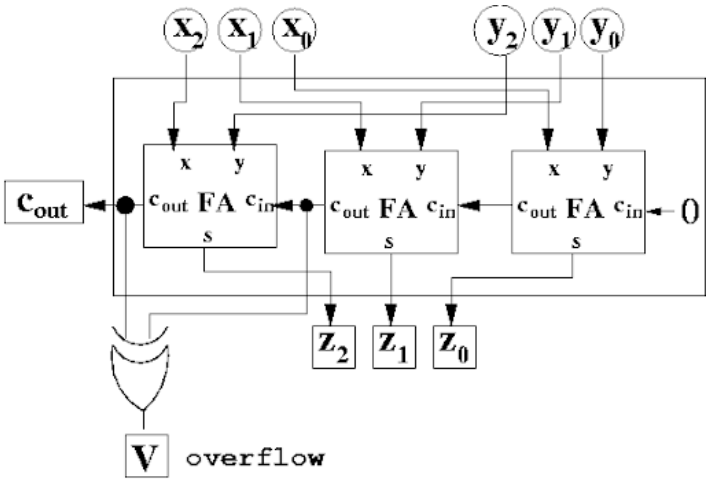


Fig.2 Circuit Diagram of Segmentation Adder with Overflow Bit

Fig 2 shows a 3-bit segmented binary adder using full adders to add two 3-bit numbers (X and Y), producing a sum (Z) and carry-out. Each full adder handles one bit and passes the carry to the next stage. An overflow bit (V) is generated by XOR-ing the carry-in and carry-out of the most significant bit, detecting overflow in 2's complement addition.

The design and implementation of the Segmentation Adder follows a systematic digital design methodology, starting from high-level modeling to physical implementation. The process flow includes both FPGA and ASIC design tracks, enabling verification of the design across multiple platforms. Below is a detailed step-by-step description:

1. RTL Design using Verilog HDL

- The architecture of the Segmentation Adder is modeled using Verilog, where the logic for segment-wise addition and carry propagation is written.
- The design is modular, with each segment acting as an independent unit, allowing for scalable and reusable code.

2. Functional Simulation

- Simulation is carried out using testbenches to verify the functional correctness of the Verilog code.
- Waveforms are analyzed using GTKWave, ensuring the outputs match expected results for various input patterns, including edge cases like overflow conditions.

3. Synthesis

- For FPGA Implementation, synthesis is done using Xilinx Vivado.
- For ASIC Implementation, the design is synthesized using Yosys with the Skywater 130nm standard cell library.
- This step converts the RTL into a gate-level netlist.

4. Floorplanning and Placement (ASIC Path)

- The synthesized netlist is imported into OpenLane, where floorplanning is done to define die size and block placement.
- Placement algorithms are applied to position standard cells optimally for routing.

5. Clock Tree Synthesis and Routing

- The design undergoes clock tree synthesis to ensure that clock signals reach all sequential elements with minimal skew.
- Global and detailed routing are performed to connect all logic gates physically using metal layers.

6. Physical Verification

- Design Rule Checks (DRC), Layout vs. Schematic (LVS), and antenna checks are performed to ensure the layout is manufacturable and accurate.
- Timing and power reports are generated using OpenSTA and OpenLane.

7. GDSII Generation

- The final layout is exported as a GDSII file, which is the standard format for semiconductor fabrication.
- Visual inspection is done using KLayout.

8. FPGA Implementation (Parallel Flow)

- In parallel to ASIC flow, the design is implemented on FPGA using Vivado.
- Post-implementation, resource utilization, timing summary, and power consumption are analyzed.

CHAPTER 4

TOOLS AND SOFTWARES USED

The design, simulation, synthesis, and implementation of the Segmentation Adder were accomplished using a mix of open-source and proprietary tools. These tools enabled RTL development, waveform analysis, synthesis, physical design, and layout generation.

Hardware Description & Simulation

- **Verilog HDL**

  Used to describe the RTL (Register Transfer Level) design of the segmentation adder.

- **Icarus Verilog + GTKWave**

  - *Icarus Verilog* was used to compile and simulate the Verilog code.
  - *GTKWave* was used to visualize simulation waveforms and verify output correctness.

FPGA Design Tools

- **Xilinx Vivado Design Suite Used for:**

  - RTL simulation
  - Synthesis
  - Implementation (placement and routing)
  - Generating bitstream for FPGA configuration
  - Viewing schematic and post-implementation design structure

ASIC Design Tools

- **OpenLane ASIC Flow**
  An open-source, end-to-end RTL to GDSII toolchain used for:
    - Synthesis (via Yosys)
    - Floorplanning
    - Placement (via RePlAce)
    - Routing (via TritonRoute)
    - Timing analysis (via OpenSTA)
    - GDSII export
- **Skywater 130nm PDK (Process Design Kit)**
  Standard cell library used for synthesis and physical layout in ASIC flow.
- **KLayout**
  A GDSII layout viewer used to inspect and verify the final physical design.

Analysis & Reporting

- OpenSTA – Static Timing Analysis
- Power Report Tools (via OpenLane) – Power consumption and leakage analysis
- Utilization Report – For evaluating area and cell usage
- Timing Report – For evaluating critical paths and slack

These tools together provided a complete development environment for testing, verifying, synthesizing, and implementing the segmentation adder on both FPGA and ASIC platforms.

CHAPTER 5

DESIGN METHODOLOGY

I. MAKING OF VERILOG DESIGN AND TESTBENCH:

a. 8-bit Segmentation Adder Verilog Code:

```
module sa(
input  [n-1:0] a,  // First operand
input  [n-1:0] b,  // Second operand
output [n-1:0] sum  // Result
);

parameter n = 8;  // Number of bits

wire [n-1:0] carry;

// Generate the carry signals
genvar i;
generate
for (i = 0; i < n; i = i + 1) begin
if (i == 0) begin
// First bit, no carry input
assign carry[i] = a[i] & b[i];
end else begin
// Subsequent bits, use previous carry output
assign carry[i] = (a[i] & b[i]) | (a[i] & carry[i-1]) | (b[i] & carry[i-1]);
end
end
endgenerate

// Generate the sum signals
genvar j;
generate
for (j = 0; j < n; j = j + 1) begin
assign sum[j] = a[j] ^ b[j] ^ carry[j];
end
endgenerate

endmodule
```

b. 4-bit Segmentation Adder Verilog Code:

```
module sa(
 input  [n-1:0] a,  // First operand
 input  [n-1:0] b,  // Second operand
 output [n-1:0] sum  // Result
);

parameter n = 8;  // Number of bits

wire [n-1:0] carry;

// Generate the carry signals
genvar i;
generate
   for (i = 0; i < n; i = i + 1) begin
      if (i == 0) begin
         // First bit, no carry input
         assign carry[i] = a[i] & b[i];
      end else begin
         // Subsequent bits, use previous carry output
         assign carry[i] = (a[i] & b[i]) | (a[i] & carry[i-1]) | (b[i] & carry[i-1]);
      end
   end
endgenerate

// Generate the sum signals
genvar j;
generate
   for (j = 0; j < n; j = j + 1) begin
      assign sum[j] = a[j] ^ b[j] ^ carry[j];
   end
endgenerate

endmodule
```

c. 8-bit Segmentation Adder Testbench Code:

```
module sa;

    reg [7:0] a;
    reg [7:0] b;
    wire [7:0] sum;

    sa uut(
        .a(a),
        .b(b),
        .sum(sum)
    );

    initial begin
        a = 8'h12;
        b = 8'h34;
        #5;
        a = 8'h10;
        b = 8'h30;
        #5;
        a = 8'h08;
        b = 8'h50;
        #5;
        a = 8'h49;
        b = 8'h22;
        #5;
        a = 8'h74;
        b = 8'h63;
        #5;
        a = 8'h02;
        b = 8'h04;
        #5;
        a = 8'h88;
        b = 8'h11;
        #5;

    end

     //Dump waves (only required here)
    initial
      begin
        $dumpfile("dump.vcd");
        $dumpvars(1);
      end

    endmodule
```

II. WAVEFORM ANALYSIS USING 'GTKWAVE":

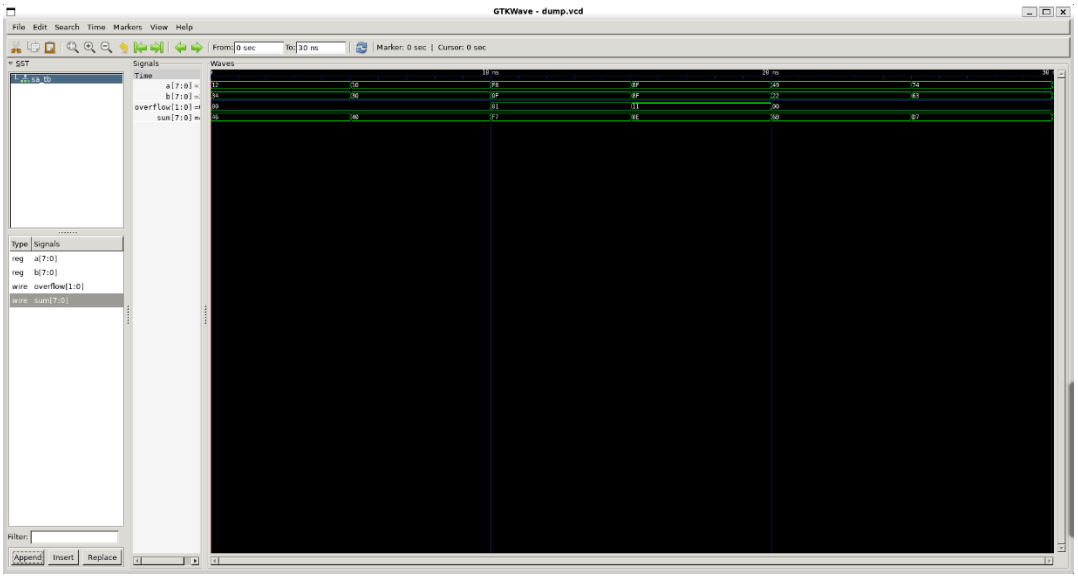


Fig.3 Analysis of the Waveform file generated using GTKwave

Fig.3 shows a waveform simulation in **GTKWave**, a tool used to view signal transitions in digital circuit simulations. It displays signals like `a`, `b`, `overflow`, and `sum` over time, each with 8-bit and 1-bit widths. You can observe how input values (`a` and `b`) change at intervals, and how the output (`sum`) and `overflow` flags respond accordingly—useful for debugging or verifying digital designs like adders.

Signals Displayed:

- `a[7:0]`: 8-bit input A to the adder
- `b[7:0]`: 8-bit input B to the adder
- `overflow[1:0]`: Indicates overflow status for each segment (or final carry out)
- `sum[7:0]`: 8-bit output sum of A and B

Time Points (in nanoseconds):

The waveform spans from **0 ns to 30 ns**, showing changes in input and output at specific time intervals.

Waveform Analysis (Selected Time Points):

a (Input)	b (Input)	sum (Output)	overflow
`0x12` (18)	`0x30` (48)	`0x42` (66)	`00`
`0x80` (128)	`0x80` (128)	`0x00` (0)	`01`
`0xA5` (165)	`0x20` (32)	`0xC5` (197)	`00`
`0xFF` (255)	`0x01` (1)	`0x00` (0)	`01`

Note: Values like `0x12`, `0x30`, etc., are **hexadecimal representations** of binary data.

III. PRE-LAYOUT SIMULATION AND VERIFICATION IN BASYS3 FPGA

Procedure For Implementation of the Verilog design in Basys3 FPGA Board:
The Basys3 FPGA development board, based on the Xilinx Artix-7 FPGA, is used to implement the segmentation adder design. Below are the steps followed from writing the HDL code to deploying it on hardware:

1. Write Verilog HDL Code

- Design the segmentation adder in Verilog HDL using any text editor or Vivado's built-in editor.
- Create a testbench for simulating the adder functionality.

2. Functional Simulation

- Use **Icarus Verilog + GTKWave** or Vivado's built-in simulator to verify the behavior of the design.
- Ensure that the simulation results match the expected values for various input patterns.

3. Create a New Vivado Project

- Open **Xilinx Vivado** and create a new project.
- Select the **Basys3 board** (Part: xc7a35tcpg236-1) from the available board files.
- Add the Verilog source files and constraint file (.xdc).

4. Add Constraints (.xdc File)

- Assign FPGA I/O pins to switches (inputs) and LEDs (outputs) using the Basys3 master constraints file. Example:

```
set_property PACKAGE_PIN W5 [get_ports {a[0]}] ;# SW0
set_property PACKAGE_PIN V17 [get_ports {sum[0]}] ;# LED0
```

5. Synthesize the Design

- Run **Synthesis** in Vivado to convert the Verilog design to a gate-level netlist.
- Check for and fix any synthesis warnings or errors.

6. Implement the Design

- Run **Implementation** to place and route the design within the FPGA fabric.
- Ensure timing constraints are met.

7. Generate Bitstream

- After successful implementation, run **Generate Bitstream**.
- This creates the .bit file needed to configure the FPGA.

8. Program the Basys3 FPGA Board

- Connect the Basys3 board to your PC via USB.
- Open the **Hardware Manager** in Vivado.
- Auto-connect to the board and load the generated .bit file.
- Once programmed, test the design using switches (input) and observe the result on LEDs (output).

9. Test and Validate

- Use the switches to provide inputs to the adder.
- Observe the result through LEDs and validate the correctness with expected results.
- Test different combinations and edge cases (e.g., overflow).

This implementation confirms the successful deployment of the segmentation adder design on real FPGA hardware, ensuring its functionality beyond simulation.

Block Diagram:

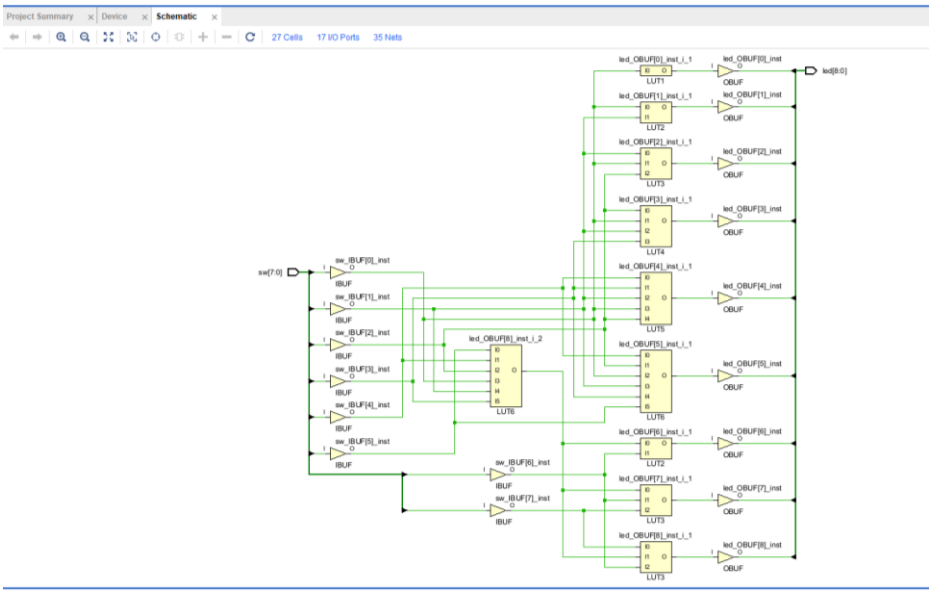


fig.4 Shows the Schematic of the segmentation Adder

Fig.4 shows the internal logic of the Segmentation Adder implemented on the FPGA. Inputs from switches (sw[7:0]) are passed through input buffers (IBUFs) into look-up tables (LUTs) where the addition logic is performed. The result is then sent through output buffers (OBUFs) to drive the LEDs (led[7:0]). This visual representation confirms how the Verilog code is mapped onto FPGA hardware.

Port Mapping:

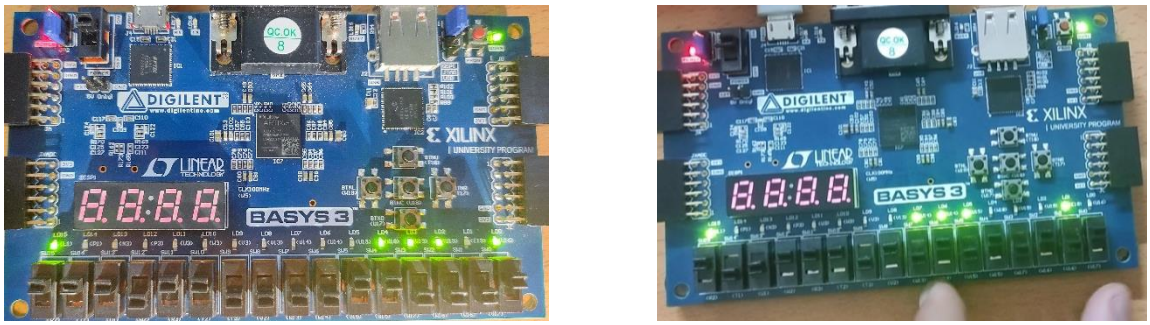
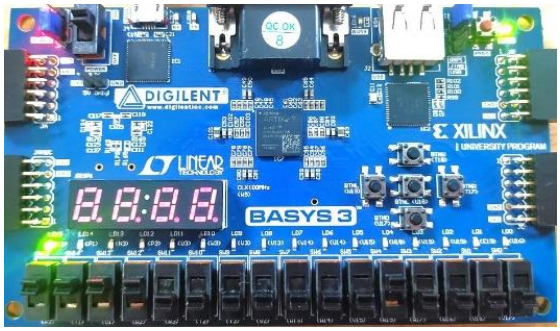


Fig.5 Shows Port Mapping in the BASYS3 FPGA

Case 1:	**Case 2:**
	
Fig.6 No overflow case in the BASYS3 FPGA	Fig.7 Overflow case in the BASYS3 FPGA
No Overflow Case • **Input Switches**: `(9 + 6) + (9 + 6)` = 30 • **LED Output**: Binary equivalent of `30` is **0001 1110**, which corresponds to LED pattern lighting up 5 LEDs. • **7-Segment Display**: Shows intermediate or sum result (depending on your Verilog mapping). • **Overflow Indicator**: **OFF**, meaning no overflow occurred—result fits within 8-bit range.	**Overflow Case** • **Input Switches**: `240 + 16 = 256` • **LED Output**: LEDs show **0000 0000** (i.e., result wrapped around to 0) • **Overflow Indicator**: One of the leftmost LEDs or possibly 7-segment segments may glow differently or be programmed to show carry/overflow. • **Overflow Detected**: Yes — the adder output exceeds 8-bit limit, hence `256 → 0 (with overflow).`

Case 3:	Case 4:

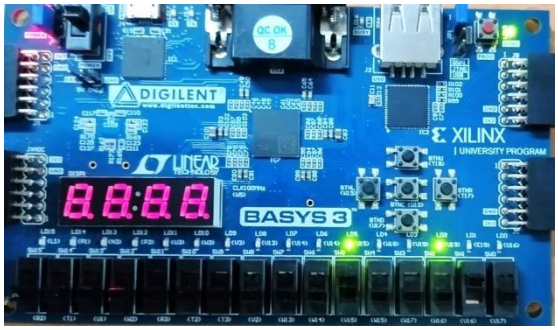
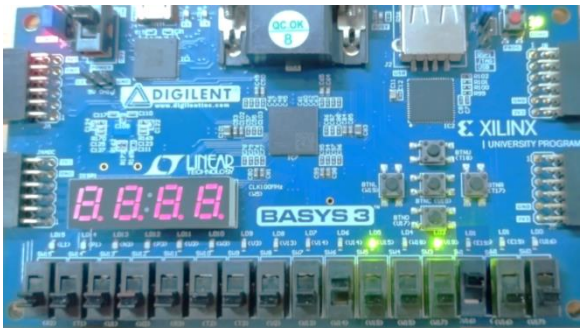

	

Fig.8 No overflow case in the BASYS3 FPGA	Fig.9 No overflow case in the BASYS3 FPGA
No Overflow Case • **Input Switches**: Binary inputs represent `2` on each operand. • **Operation**: 2 + 2 = 4 • **LED Output**: Binary `0000 0100` – only the 3rd LED from right is ON, indicating a value of 4. • **7-Segment Display**: Shows `0004`, representing the decimal result. • **Overflow Indicator**: OFF – result is well within 8-bit range. • **MSB Check**: MSB logic is functioning correctly with no overflow triggered.	**No Overflow (4 + 4 = 8)** • **Input Switches**: Binary inputs represent `4` on each operand. • **Operation**: 4 + 4 = 8 • **LED Output**: Binary `0000 1000` – 4th LED from right is ON, showing a value of 8. • **7-Segment Display**: Shows `0008`, confirming the correct sum. • **Overflow Indicator**: OFF – still within 8-bit range, no overflow. • **MSB Check**: MSB logic correctly shows result with no incorrect carry.

- **Case 1** tested a small addition (30), and no overflow occurred; all outputs were as expected.
- **Case 2** tested a large addition (256), causing an 8-bit overflow, which was correctly detected.
- Both **Case 1 and 2** faced issues with **increased junction temperature**, possibly from extended FPGA usage or high input activity.
- Due to the thermal impact, accurate results weren't reliable over time.
- So, **Case 3 and 4** were repeated versions of Case 1 and 2 under better thermal conditions.
- These later cases ensured proper operation without overheating, confirming the design's reliability.

IV. PHYSICAL LAYOUT USING SKYWATER 130PDK:

SkyWater 130nm PDK –

The SkyWater 130nm Process Design Kit (Sky130 PDK) is an open-source CMOS technology platform developed by SkyWater Technology. It provides all the necessary files, models, and libraries required for ASIC design, including standard cells, DRC/LVS rules, and technology specifications.

In this project, the Sky130 PDK was used in combination with the OpenLane toolchain to implement the RTL-to-GDSII flow for the Segmentation Adder. The use of Sky130 enabled a complete, fabrication-level physical design using an industry- grade, open-source technology.

1. Synthesis Using 'YOSYS':

Yosys was used as the first step in the ASIC flow to synthesize the Carry Select Adder (CSA) design. It translated the high-level Verilog description into a netlist using standard cells from the SkyWater 130nm PDK.

A **flow diagram** was created to illustrate the complete synthesis and implementation process.

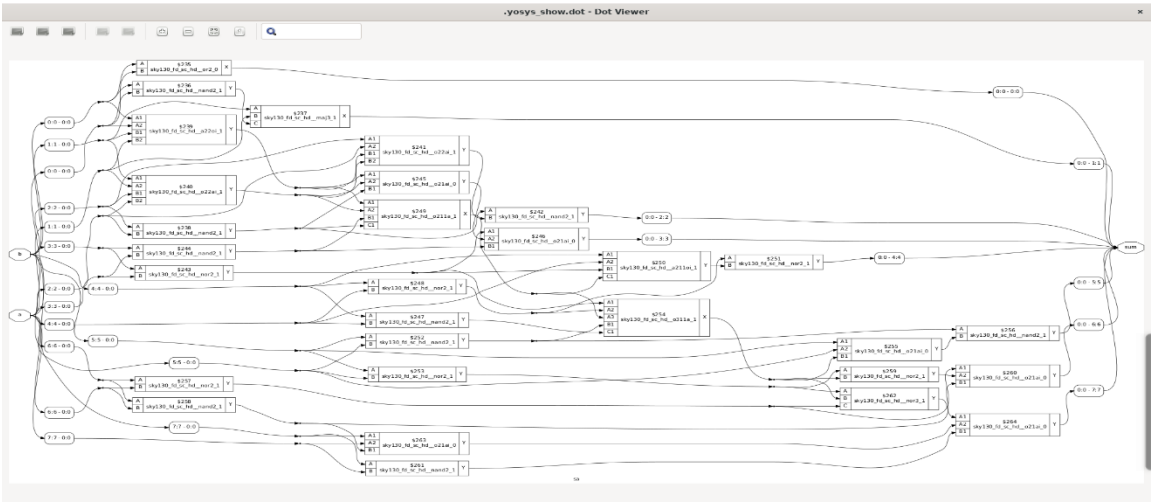


Fig.10 Flow diagram generated by synthesis using YOSYS tool

Fig.10 shows a **logic gate-level netlist visualization** in the **Yosys Dot Viewer**, typically used in digital design flow after synthesis. Each block represents logic gates or operations (like AND, OR, XOR, etc.), and the arrows represent signal flow between them. The diagram reveals how the input bits are combined and routed through logic elements to generate outputs. It helps designers verify and understand the internal structure of synthesized Verilog modules.

2. Physical Layout Using 'OPENLANE':

OpenLane is an open-source digital ASIC flow tool used to convert synthesized netlists into a physical layout. In this project, OpenLane was used to generate the physical layout of the Carry Select Adder (CSA) using the SkyWater 130nm PDK. The flow included key steps like floor planning, placement, clock tree synthesis, routing, and final GDSII generation. This process produced a layout ready for fabrication, completing the RTL-to-GDSII implementation.

A. **FLOOR PLANNING**:

Floor planning is the initial step in the physical design process where the basic layout of the chip is defined. It involves allocating space for the core area, placing input/output ports, and defining power and ground regions.

Fig.11 Floor Planning process

B. **PLACEMENT**:

Placement is the stage in physical design where the standard cells from the synthesized netlist are assigned fixed locations within the chip layout. The tool positions the cells to minimize wire length and timing delays, ensuring optimal connectivity and efficient use of chip area, while preparing the design for routing.

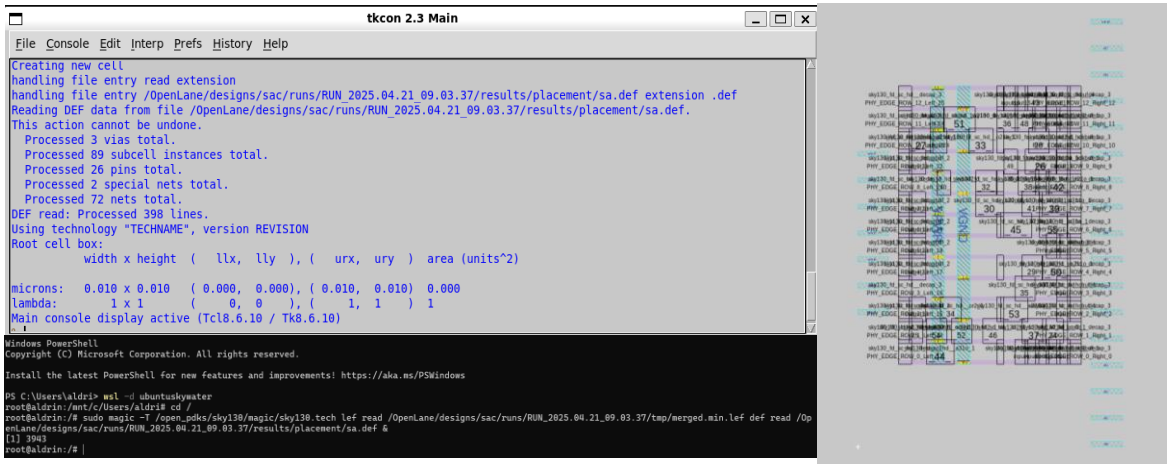


Fig.12 Placement Process

C. **ROUTING**:

Routing is the process of connecting the placed standard cells using metal layers to form the actual signal paths. The tool automatically created the necessary interconnections for power, ground, and signals, ensuring that all design rules are met.

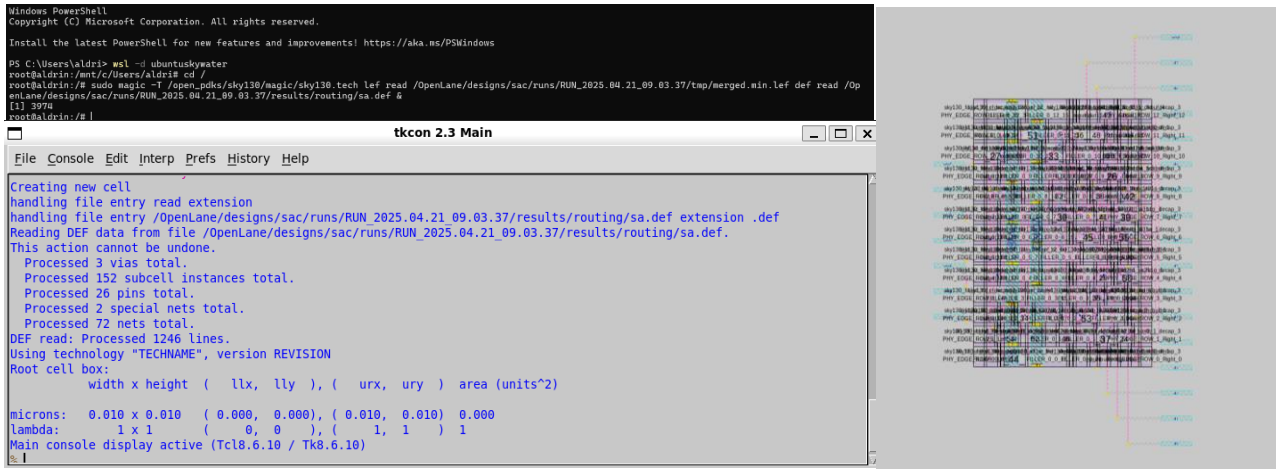


Fig.13 Routing Process

3. Final Physical Layout:

After completing placement and routing, the **final physical layout** of the Segmentation Adder was generated, showing all cells, interconnections, and metal layers.

The layout was then converted into a **.mag file** using tools like Magic, which is used to view, edit, and verify the physical layout.

4. GDSII File Format & Klayout:

The **GDS (Graphic Data System) file** is the final output of the ASIC design flow and the close to fabrication design, containing all geometric information of the chip layout, including layers, cells, and routing. It is the standard format used for chip fabrication.

KLayout is a powerful open-source layout viewer used to visualize and verify the GDSfile.

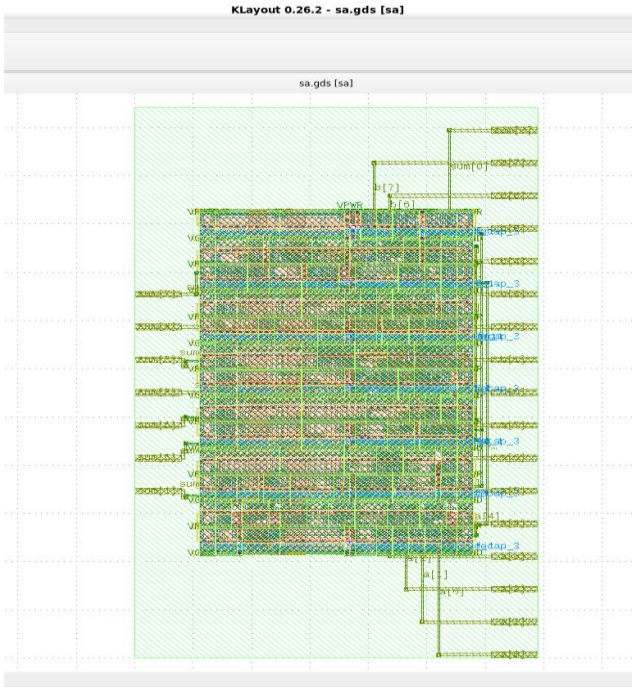


Fig.14 GDS File diagram of the Segmentation Adder

CHATPER 6

RESULTS AND DISCUSSION

I. POWER ANALYSIS:

a. Power Report of Segmentation Adder (initial design):

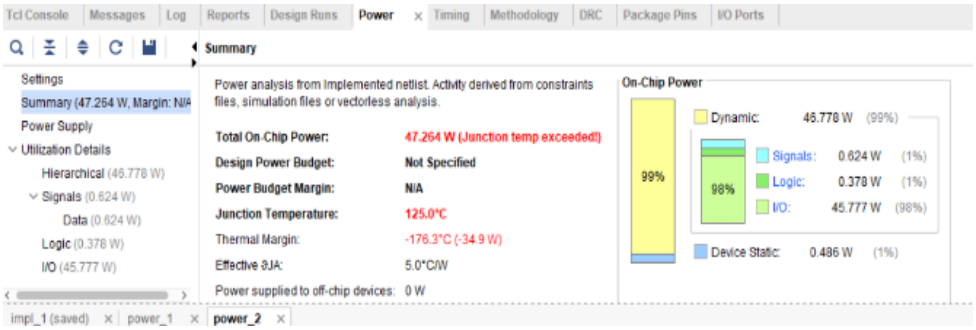


Fig.15 Shows the Power Report of the Segmentation Adder in initial design

- **Total On-Chip Power:** 47.264 W
- **Junction Temperature:** 125.0°C (**exceeded limit**)
- **I/O Power:** 45.777 W, extremely high
- **Thermal Margin:** -176.3°C (severely exceeded safe limits)
- This design caused a **thermal failure**, making the system unstable and unsafe for deployment.

b. Power Report of Segmentation Adder (Optimized design):

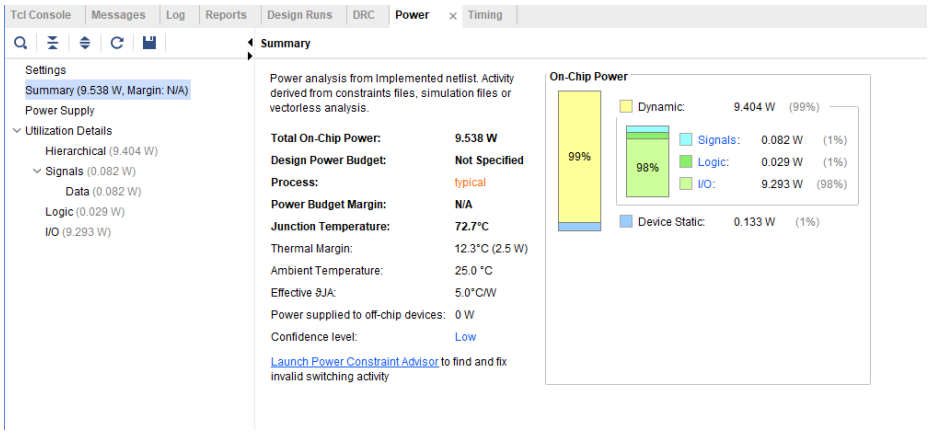


Fig.16 Shows the Power Report of the Segmentation Adder in optimized design

- **Total On-Chip Power:** 9.538 W
- **Junction Temperature:** 72.7°C (within safe operating range)
- **I/O Power:** 9.293 W, reduced from the previous version
- **Thermal Margin:** +12.3°C, showing no overheating risk
- This design is more **power-efficient**, with better **heat management** and **lower energy consumption**, making it reliable for extended use.

C. Comparison Table:

Feature	Optimized Design	Initial Design
Total Power	9.538 W	47.264 W
Junction Temp	72.7°C	125°C
I/O Power	9.293 W (98%)	45.777 W (98%)
Thermal Margin	12.3°C	-176.3°C
System Health	Safe & Efficient	Overheating & Risky

table.1 Shows the comparison of 4-bit and 8-bit segmentation adder

Table 1 compares the performance of a 4-bit segmentation adder (Optimized Design) versus an 8-bit version (Initial Design). The optimized 4-bit design significantly reduces total power consumption to 9.538 W and maintains a safe junction temperature of 72.7°C. In contrast, the 8-bit design consumes 47.264 W and reaches a critical 125°C, indicating thermal overload. Both designs show high I/O power usage (98%), but the optimized version manages it more efficiently. The thermal margin also improves drastically from -176.3°C to +12.3°C. Overall, the optimized design ensures safe, stable, and power-efficient operation.

Comparison Table of 8-bit Segmentation adder and 4-bit Segmentation adder:

Feature	8-Bit Segmented Adder	4-Bit Segmented Adder	Change/Trend
Adder Bit-Width	8 bits	4 bits	Decrease
Input Switches	16	8	Decrease
Sum Output LEDs	8	4	Decrease
Overflow LED	1	1	No Change
Slice LUTs	9	9	No Significant Change
Bonded IOBs	18	Likely around 12-17	Decrease
Dynamic Power	Higher (0.624 W)	Lower (0.014 W)	Significant Decrease
Static Power	Higher (46.660 W)	Lower (0.033 W)	Significant Decrease
Total On-Chip Power	Higher (47.284 W)	Lower (0.047 W)	Significant Decrease
Junction Temperature	Exceeding Limit	Within Limit (72.7 °C)	Significant Improvement
Timing Slack (WNS)	Positive (2.001 ns)	Likely Higher	Increase Expected

table.2 Shows the comparison of 4-bit and 8-bit segmentation adder

This table 2 compares an 8-bit segmented adder with a 4-bit version. Reducing the bit-width significantly decreases dynamic and static power, total on-chip power, and junction temperature—making the design more power-efficient and thermally safe. Other aspects like LUT usage and overflow LED remain unchanged, while timing performance is expected to improve.

II. RTL TO GDS:

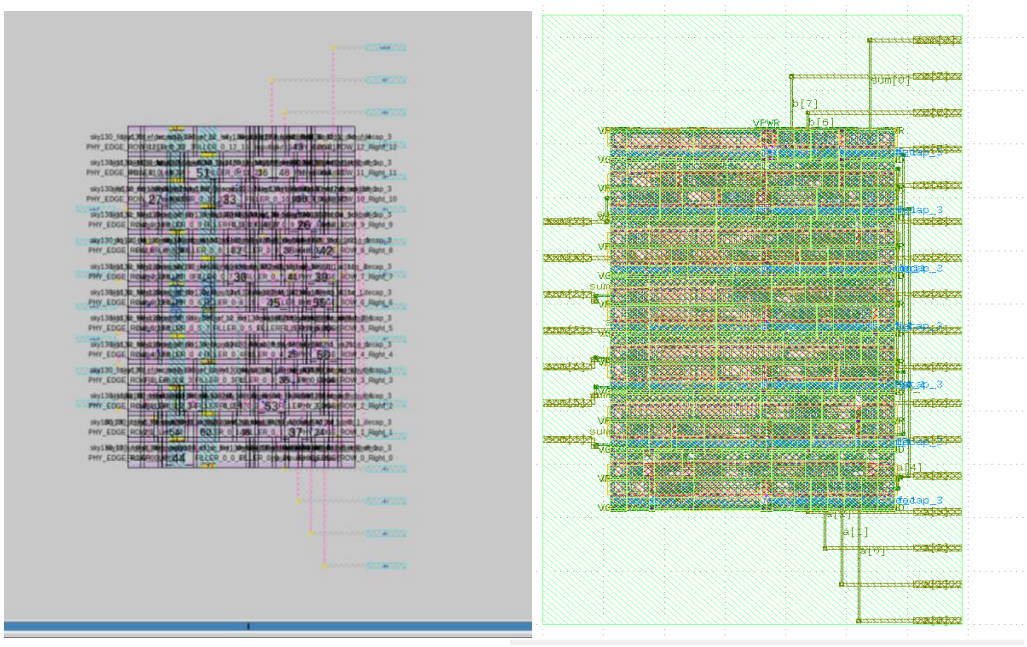


Fig.17 RTL Vs GDS diagram of the Segmentation Adder

Fig.17 shows the comparison between the **RTL (Register Transfer Level)** view and the **GDS (Graphic Data System)** layout of the segmentation adder.

- The **RTL view** (left side) represents the logical structure and connectivity of the design, focusing on functional blocks and signal flow.
- The **GDS layout** (right side) depicts the physical implementation, including standard cells, routing, and placement, ready for fabrication.
- This comparison highlights the transformation from abstract logic design to detailed silicon layout.
- It ensures that the physical design accurately reflects the intended logic and meets performance, area, and power constraints.
- The figure confirms successful synthesis and layout, critical steps in VLSI design flow.

Linux Commands Used

GTKWave:(In Windows)

wsl –d Ubuntu

cd /

cd dvsd_pe_sky130

ls

cd pre_layout_simulation

gtkwave sa.vcd

To clone the git repository

wsl -d Ubuntu

cd /

git clone https://github.com/efabless/openlane-ci-designs

from openlane-ci-designs we should cpy inverter.v file and from google drive copy the tb_inverter.v and vcd files in dvsd

Yosys:

wsl -d Ubuntu

cd /

cd sky130RTLDesignAndSynthesisWorkshop/DC_WORKSHOP/verilog_files

yosys

read_liberty -lib /sky130RTLDesignAndSynthesisWorkshop/lib/sky130_fd_sc_hd__tt_025C_1v80.lib

read_verilog /sky130RTLDesignAndSynthesisWorkshop/DC_WORKSHOP/verilog_files/sa.v

synth -top sa

abc -liberty /sky130RTLDesignAndSynthesisWorkshop/lib/sky130_fd_sc_hd__tt_025C_1v80.lib

show

OpenLane:

cd OpenLane
sudo make mount
flow.tcl -design sa

sudo magic -T /open_pdks/sky130/magic/sky130.tech lef read
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/tmp/merged.min.lef def read
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/results/floorplan/sa.def &

sudo magic -T /open_pdks/sky130/magic/sky130.tech lef read
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/tmp/merged.min.lef def read
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/results/placement/sa.def &

sudo magic -T /open_pdks/sky130/magic/sky130.tech lef read
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/tmp/merged.min.lef def read
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/results/routing/sa.def &

sudo magic -T /open_pdks/sky130/magic/sky130.tech
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/results/final/mag/sa.ma g

sudo klayout
/OpenLane/designs/sa/runs/RUN_XXXXXXXXXX/results/final/gds/sa.gds

CHATPER 7

CONCLUSION

The design and implementation of the **Segmentation Adder** successfully demonstrate an efficient approach to accelerating arithmetic operations by minimizing carry propagation delays. By dividing the input operands into multiple segments and processing them in parallel, the architecture achieves improved speed and scalability compared to traditional adder designs. The project was implemented using both **FPGA (Basys3 Board with Xilinx Vivado)** and **ASIC (Skywater 130nm PDK with OpenLane)** workflows. Functional simulation confirmed the correctness of the design, while synthesis and implementation verified its feasibility for real-time applications. Hardware testing on the Basys3 board validated the practical performance of the adder, and ASIC layout generation further showcased its readiness for fabrication.

This segmentation-based architecture proves to be a viable solution for high-performance, low-latency digital systems, especially in domains requiring rapid arithmetic computation such as DSP, embedded processors, and custom accelerator designs.

References:

[1] **Harris, D.M., & Harris, S.L.**
Digital Design and Computer Architecture, 2nd Edition, Morgan Kaufmann, 2012.
(For understanding adder architectures and hardware design basics)

[2] **Weste, N.H.E., & Harris, D.**
CMOS VLSI Design: A Circuits and Systems Perspective, 4th Edition, Pearson, 2011.
(For ASIC design methodology and layout concepts)

[3] **SkyWater Technology Foundry.**
Sky130 PDK Documentation.
https://github.com/google/skywater-pdk
(Official open-source 130nm Process Design Kit used in RTL-to-GDS flow)

[4] **OpenLane Project – Efabless Corporation.**
OpenLane: Open-source ASIC implementation flow.
https://github.com/The-OpenROAD-Project/OpenLane
(Tool used for floorplanning, placement, and routing)

[5] **Clarke, L., & Shukla, S.**
"Comparison of Adder Architectures for Low Power and High Speed Applications", *IEEE Conference on VLSI Design*, 2020.
(For comparative analysis of adder types)

[6] **Brent, R.P., & Kung, H.T.**
"A Regular Layout for Parallel Adders", *IEEE Transactions on Computers*, vol. C-31, no. 3, 1982.
(Original conceptual foundation for Segmentation Adder and similar parallel adder structures)

[7] **Yosys Open SYnthesis Suite**
https://yosyshq.net/yosys/
(Tool used for RTL synthesis of Verilog designs)

[8] **GTKWave**
http://gtkwave.sourceforge.net/
(Tool used for waveform visualization from VCD files)

[9] **KLayout - Layout Viewer for IC Design**
https://www.klayout.de/
(Used to view and verify GDSII layouts)