# ENHANCED FIR FILTER DESIGN USING HYBRID ADDERS AND MULTIPLIERS

***A Project report submitted in partial fulfillment of the requirements for the award of the degree of***

**BACHELOR OF TECHNOLOGY**

**IN**

**ELECTRONICS AND COMMUNICATION ENGINEERING**


*Submitted by*

**PALURU THARUN**

**(A21126512164)**


Under the guidance of

**Mr. N. Srinivasa Naidu**

**Assistant Professor**


**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES

(UGC AUTONOMOUS)

*(Permanently Affiliated to AU, Approved by AICTE and Accredited by NBA & NAAC with 'A' Grade)*

Sangivalasa, Bheemunipatnam mandal, Visakhapatnam district, Andhra Pradesh


2024-2025

**DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING**

**ANIL NEERUKONDA INSTITUTE OF TECHNOLOGY AND SCIENCES**

**(UGC AUTONOMOUS)**

**(Permanently Affiliated to AU, Approved by AICTE, and Accredited by NBA & NAAC with 'A+' Grade)**

**Sangivalasa - 531162, Bheemunipatnam Mandal, Visakhapatnam District (A. P)**


# CERTIFICATE

This is to certify that the project report entitled "**Enhanced Fir Filter Design Using Hybrid Adders and Multipliers**" submitted by **PALURU THARUN (A21126512164)** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Electronics & Communication Engineering** of Anil Neerukonda Institute of Technology and Sciences, Andhra University, Visakhapatnam, is a record of Bonafide work carried out under my guidance and supervision.

| **Project Guide** | **Head of the Department** | **External Examiner** |
|---|---|---|
| **Mr. N. Srinivasa Naidu** | **Dr. B. Jagadeesh** | |
| **Assistant Professor** | **Professor** | |
| Department of E.C.E | Department of E.C.E | |
| ANITS | ANITS | |

# ACKNOWLEDGEMENT

We would like to express our deep gratitude to our project guide **Mr. N. Srinivasa Naidu,** Assistant Professor, Department of Electronics and Communications Engineering, ANITS, for his guidance with unsurpassed knowledge and immense encouragement. We are grateful to **Dr. B. Jagadeesh**, Head of the Department, Electronics and Communication Engineering, for providing us with the required facilities for the completion of the project work.

We are very much thankful to the **Principal and Management, ANITS,** Sangivalasa, for their encouragement and cooperation to carry out this work.

We express our thanks to all **teaching faculty** of the Department of ECE**,** whose suggestions during reviews helped us in the accomplishments of our project. We would like to thank all **non–teaching staff** of the Department of ECE, ANITS for providing great assistance in the accomplishment of our project.

We would like to thank our parents, friends, and classmates for their encouragement throughout our project period. Last but not least, we thank everyone for supporting us directly or indirectly in completing this project successfully.

**PROJECT STUDENT**

PALURU THARUN

(A21126512164)

# ABSTRACT

Finite Impulse Response (FIR) filters find extensive usage in digital signal processing applications such as audio signal enhancement, image processing, analysis of biomedical signals, and communication systems. FIR filters have their benefits such as linear phase response and inherent stability but are constrained by high power requirements and an added propagation delay with conventional approaches using inefficient arithmetic building blocks.

This work overcomes these limitations through the design of a more efficient FIR filter architecture based on enhancing arithmetic performance. The central innovation is the creation of hybrid arithmetic units: a 32-bit hybrid adder and a 16-bit hybrid multiplier designed to reduce power-delay product (PDP) with high-speed capabilities. The hybrid adder is built by combining Ling and Sparse Kogge-Stone (SKSA) adder topologies (LKSA), taking advantage of their complementary properties to minimize critical path delay and switching activity. The hybrid multiplier utilizes a 4-chain tree addition method, summing up partial products in an efficient manner with serially configured hybrid adders. These specialized arithmetic blocks were simulated with Verilog HDL and synthesized with the Xilinx Vivado 2024.1 toolchain. All simulations and synthesis processes were carried out targeting the Artix-7 FPGA (Part: xc7a50tcpg236-1), a popular mid-range FPGA from AMD Xilinx.

Comprehensive synthesis results indicate that the new 32-bit LKSA hybrid adder offers an improvement of 45.64% in PDP over the conventional 32-bit Ling adder and 43.82% over the SKSA. Likewise, the new 16-bit hybrid multiplier performs better than the conventional 16-bit Vedic multiplier by 41.53% in PDP, as well as other configurations such as Wallace and Dadda, with respect to power and timing. When implemented within a 16-tap FIR filter structure, the overall system realizes an improvement 85.87% in PDP and 83.82% power savings, and a 12.72% propagation delay reduction over a traditional FIR filter implemented by direct arithmetic.

In conclusion, the project offers a high-performance and scalable FIR filter design based on hybrid arithmetic units, with improvements over traditional methods in terms of speed and power consumption.

# CONTENTS

# List Of Figures

# List Of Tables

# List Of Abbreviations

**FIR:** Finite Impulse Response

**DSP:** Digital Signal Processing

**RCA:** Ripple Carry Adder

**CSA:** Carry Save Adder

**CLA:** Carry Look Ahead Adder

**KSA:** Kogge-Stone Adder

**LA:** Ling Adder

**BKA:** Brent-Kung Adder

**SKSA**: Sparse Kogge-Stone Adder

**LSKSA:** Ling Sparse Kogge Stone Hybrid Adder

**PDP:** Power Delay Product

**VLSI:** Very-Large-Scale Integration

**ECG:** Electrocardiogram

**HDL**: Hardware Description Language

**Verilog**: Verilog HDL

**D-ff:** D-type Flip-Flop

**Vivado:** Xilinx Vivado Design Suite

**VCD:** Value Change Dump

**SAIF:** Switching Activity Interchange Format

**Dadda:** Dadda Multiplier

**Karatsuba:** Karatsuba Multiplier

**Wallace**: Wallace Tree Multiplier

**Vedic:** Vedic Multiplier

**LUT:** Look-Up Table

**WNS:** Worst Negative Slack

**TNS:** Total Negative Slack

**FPGA:** Field-programmable gate Array

# Chapter 1

# Introduction

## 1.1 Background

Digital Signal Processing (DSP) plays a critical role in modern applications, such as telecommunications, audio processing, biomedical engineering, and image enhancement. Among the fundamental building blocks of DSP systems, Finite Impulse Response (FIR) filters are widely preferred due to their inherent stability, linear phase response, and deterministic behaviour. These characteristics make FIR filters essential in applications where signal accuracy and consistency are paramount.

An FIR filter processes input signals by computing a weighted sum of current and past input samples, eliminating feedback loops and ensuring a well-controlled impulse response. However, the efficiency of an FIR filter is highly dependent on the underlying arithmetic operations, primarily addition and multiplication, which significantly impact speed, power consumption, and silicon area utilization.

## 1.2 Problem Statement

Traditional FIR filter architectures rely on conventional adders and multipliers, such as Ripple Carry Adders (RCA), Carry Look-Ahead Adders (CLA), and Wallace Tree multipliers. While effective, these designs suffer from the following drawbacks:

- High Power Consumption: Conventional arithmetic units introduce excessive switching activity, leading to increased dynamic power dissipation.
- Increased Propagation Delay: The sequential nature of traditional adders and multipliers results in longer computation times, restricting the filter's real-time processing capabilities.
- Large Area Utilization: Complex multiplier circuits occupy significant chip area, making integration in power-sensitive embedded systems difficult.

With the increasing demand for high-speed, low-power, and compact DSP implementations, conventional FIR filter designs fail to meet modern performance benchmarks. This necessitates the development of optimized arithmetic architectures that enhance computational efficiency without compromising accuracy.

## 1.3 Objectives

This research aims to design and implement an enhanced FIR filter architecture that leverages hybrid adder and multiplier structures to optimize performance in terms of power, delay, and area. The key objectives include:

- Develop a 16-Tap FIR Filter that balances complexity and performance for DSP applications.
- Implement a Hybrid Adder Design, integrating a Sparse Kogge-Stone Adder (SKSA) and a Ling Adder (LA) to achieve optimized speed and power efficiency.
- Design an Optimized Hybrid Multiplier using a 4-chain tree addition method for faster multiplication while reducing power consumption.
- Evaluate Performance Metrics by analyzing power consumption, propagation delay, and silicon area using synthesis tools.
- Compare with Conventional Architectures to quantify improvements over traditional FIR filter implementations.

## 1.4 Project Overview

The project encompasses the following:

- **Literature Review:** Examination of existing FIR filter architectures, conventional arithmetic units, and hybrid computing techniques.
- **Proposed Design:** Development of hybrid adder and multiplier architectures tailored for FIR filter applications.
- **Implementation:** Verilog-based hardware description of the proposed FIR filter, synthesized and tested using Xilinx Vivado.
- **Performance Evaluation:** Measurement and comparison of power, delay, and area metrics under both timing-constrained and non-timing-constrained conditions.

By addressing the limitations of conventional FIR filter designs, this research contributes to the development of high-performance, energy-efficient FIR filters suitable for next-generation DSP systems.

## 1.5 Methodology

The methodology employs a systematic, multi-stage approach to design and optimize the proposed FIR filter architecture. The methodology consists of three primary stages:

### 1.5.1 Hybrid Adder Design

The adder design phase involved comprehensive investigation of parallel prefix adder architectures and their hybrid configurations:

**Comparative Analysis**: Detailed performance comparison of various adder topologies, including Brent-Kung, Ling, and Sparse Kogge-Stone across different bit widths (16 and 32 bits).

**Hybrid Configuration Development**: Creation of novel hybrid designs that strategically apply different adder structures to varying bit positions within the same arithmetic unit. For example, using Kogge-Stone for the least significant bits to optimize critical paths while employing Brent-Kung for higher-order bits to reduce wiring complexity.

### 1.5.2 Hybrid Multiplier Design

- Exploration of multiplier architectures such as Booth encoding, Wallace Tree, and Vedic multiplication, integrated with hybrid adders.
- Trade-off analysis between computational complexity and performance to determine the most efficient multiplier design.
- **Integration with Hybrid Adders***:* Strategic mapping of the final addition stages to the previously developed hybrid adder architectures, with careful consideration of timing characteristics to optimize the critical path through the complete multiplication operation.

### 1.6 FIR Filter Optimization

The filter optimization phase focused on integrating the developed arithmetic (Hybrid Adder and Hybrid Multiplier) components into an optimized FIR filter architecture namely, the Hybrid Adder (LING16 + SKSA16 with enable logic) and the Hybrid Multiplier (4-chain tree structure using 32-bit LKSA) into a 16-tap FIR filter architecture.

Traditional FIR filters rely heavily on standard arithmetic operators (+ and *) for tap multiplications and summations, resulting in higher propagation delay and power dissipation, particularly in high-tap implementations. To address this, the proposed design replaces the

conventional arithmetic units with custom-designed hardware-efficient hybrid structures that reduce switching activity, improve timing, and optimize area utilization.

The **optimized FIR filter architecture** includes the following key improvements:

- Hybrid Multipliers are used to perform coefficient multiplications for all 16 taps. These multipliers leverage the speed and parallelism of 4-chain tree addition using hybrid LKSA32 adders.
- The adder tree structure for summing partial products is organized hierarchically, reducing the critical path delay by minimizing the depth of carry propagation.
- Enable-controlled delay elements are introduced between stages to control data flow and reduce unnecessary switching, significantly lowering dynamic power consumption.

# Chapter 2

# Literature Review

## 2.1 FIR Filters in Digital Signal Processing

Finite Impulse Response (FIR) filters are widely used in digital signal processing (DSP) due to their inherent stability and linear-phase characteristics. Unlike Infinite Impulse Response (IIR) filters, FIR filters operate non-recursively, ensuring that the impulse response eventually settles to zero. This makes them ideal for applications such as audio processing, biomedical signal analysis, and telecommunications. However, the overall performance of FIR filters depends heavily on the efficiency of the underlying arithmetic units, particularly the adders and multipliers used to perform the convolution operations.

## 2.2 Adder Architectures in FIR Filters

Efficient addition is critical to achieving high-speed and low-power FIR filters. Conventional adders like the Ripple Carry Adder (RCA) suffer from high propagation delays due to sequential carry propagation. To overcome these challenges, several advanced adder architectures have been developed.

### 2.2.1 Kogge-Stone Adder (KSA)

The Kogge-Stone Adder is a parallel-prefix adder known for its minimal propagation delay due to extensive parallelism. However, its complexity and extensive routing result in higher area and power consumption.

### 2.2.2 Sparse Kogge-Stone Adder (SKSA)

The Sparse Kogge-Stone Adder (SKSA) modifies the full KSA by selectively generating carries, reducing both interconnect complexity and power consumption while still offering fast carry propagation.

### 2.2.3 Ling Adder

The Ling Adder optimizes the carry computation by transforming the conventional generate (G) and propagate (P) signals into a simpler form, thereby reducing the logic complexity. This results in lower power consumption without significantly compromising speed.

### 2.2.4 Brent-Kung Adder (BKA)

The Brent-Kung Adder (BKA) employs a hierarchical approach to compute carries, reducing logic depth and power consumption compared to full KSA implementations. Although its latency can be slightly higher than SKSA, its power efficiency makes it an important design reference.

### 2.2.5 Proposed Hybrid Adder Approach

Our proposed design integrates the advantages of the Sparse Kogge-Stone and Ling adders. The 32-bit addition is partitioned into two 16-bit segments: the upper 16 bits are processed using SKSA for rapid carry propagation, while the lower 16 bits are computed with the Ling adder to minimize power consumption. Additionally, the Brent-Kung Adder is referenced for its power efficiency. This hybrid adder architecture significantly reduces the Power Delay Product (PDP) compared to conventional designs.

## 2.3 Multiplier Architectures in FIR Filters

Multiplication is the most computationally intensive operation in FIR filters, and its efficiency critically affects overall filter performance. Conventional multipliers, such as Array, Booth, Wallace Tree, Dadda, and Vedic multipliers, each have their advantages and drawbacks:

- **Booth Multiplier:** Utilizes recoding to reduce the number of partial products; however, it introduces additional overhead in the pre-processing stage.
- **Wallace Tree Multiplier:** It employs a parallel tree structure to reduce partial products quickly, but it often requires complex wiring and consumes more power.
- **Dadda Multiplier:** Similar to Wallace Tree multipliers, but optimized for area efficiency; it still faces challenges in power consumption and delay.
- **Vedic Multiplier:** Based on ancient Vedic mathematics, these multipliers offer moderate improvements in power and speed but may not meet the stringent demands of high-speed DSP systems.

### 2.3.1 Proposed Hybrid Multiplier Design

The proposed hybrid multiplier builds upon the optimized hybrid adder proposed. Instead of using traditional reduction methods (such as Wallace or Dadda trees), our design employs a 4-chain tree addition technique. Here, 16 partial products are divided into four parallel chains, with each chain being summed using the hybrid adder (integrating SKSA and Ling techniques). The intermediate sums are then combined hierarchically to produce the final product. This method not only reduces the propagation delay and power consumption but also achieves a significant improvement in the overall Power Delay Product (PDP) compared to conventional 16-bit Vedic multipliers.

Table 2.1: Comparison of Conventional Adders Architecture

| **Architecture** | **Delay** | **Power Consumption** | **Area Utilization** | **Comments** |
|---|---|---|---|---|
| Brent-Kung Adder (BKA) | Low | Low | Small | Efficient carry computation with reduced area and power consumption; slightly higher delay than KSA. |
| Kogge-Stone Adder (KSA) | Low | High | Large | Fast carry propagation; high area and power due to extensive routing. |
| Ling Adder | Low | Medium | Moderate | Simplifies carry logic, reducing power consumption while maintaining speed. |
| Sparse Kogge-Stone Adder (SKA) | Low | Medium | Moderate | Balances speed and area; reduces wiring complexity compared to KSA. |

Table 2.2: Comparison of Conventional Multipliers Architecture

| **Architecture** | **Delay** | **Power Consumption** | **Area Utilization** | **Comments** |
|---|---|---|---|---|
| Wallace Tree Multiplier | Low | High | Large | Fast parallel reduction of partial products; increased wiring complexity. |
| Dadda Multiplier | Low | Medium | Moderate | Similar to Wallace tree but optimized for area efficiency. |
| Vedic Multiplier | Medium-Low | Medium-Low | Moderate | Utilizes ancient algorithms for efficient multiplication; moderate improvements. |

## 2.4 Introduction to FIR Filters

Finite Impulse Response (FIR) filters are a class of digital filters characterized by a finite-duration impulse response. Unlike Infinite Impulse Response (IIR) filters, FIR filters do not use feedback, ensuring absolute stability and a linear-phase response. The fundamental operation of an FIR filter is based on the convolution of the input signal with the filter's impulse response.

### 2.4.1 Mathematical Representation

The output of an FIR filter is computed as:

$$y(n) = \sum_{k=0}^{N-1} h(k)\ x(n-k)$$

where:

- y(n) is the output signal,
- x(n) is the input signal,
- h(k) represents the filter coefficients,
- N is the filter order.

This equation illustrates that an FIR filter applies a weighted sum of the most recent N input samples to generate an output.

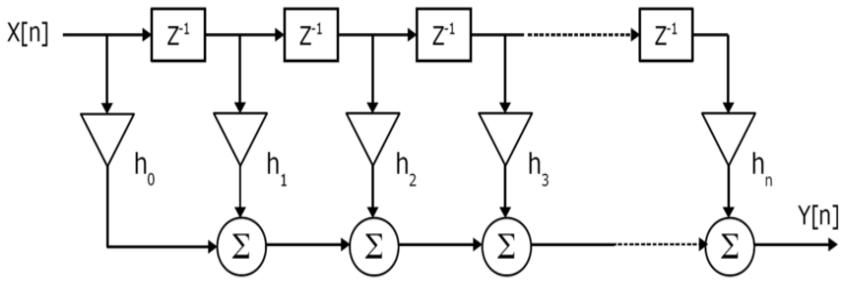



Figure 1: Logical Structure of FIR Filter

The convolution operation in FIR filters requires extensive use of multiplications and additions, making efficient arithmetic unit design crucial for high-performance implementations.

## 2.5 Types of FIR Filters

### 2.5.1 Classification by Frequency Response

- **Low Pass Filters**

Lowpass FIR filters allow signals below a cutoff frequency to pass through while attenuating higher frequencies. These filters are commonly used for smoothing signals, removing high-frequency noise, and down-sampling operations. The impulse response of an ideal lowpass filter is a sinc function, though practical implementations use windowed approximations.

- **High Pass Filters**

High-pass FIR filters pass signals above a specified frequency while attenuating lower frequencies. They are useful for edge detection in image processing, audio treble enhancement, and removing DC offsets or low-frequency drift. A high-pass filter can be derived from a low-pass filter by spectral inversion.

- **Bandpass Filters**

Bandpass FIR filters allow signals within a specific frequency band to pass while attenuating frequencies outside this band. These filters are essential in communications for channel selection, audio equalization, and isolating specific frequency components in complex signals. They can be created by cascading high-pass and low-pass filters or through direct design methods.

- **Bandstop (Notch) Filters**

Bandstop FIR filters attenuate signals within a specific frequency band while allowing others to pass. They are commonly used to remove power line interference (50/60 Hz), suppress specific noise frequencies, or eliminate unwanted resonances. Notch filters are specialized bandstop filters with a very narrow stopband.

- **Differentiators**

Differentiator FIR filters approximate the derivative of the input signal in the time domain. Their frequency response is proportional to frequency, making them useful for edge detection, pitch detection, and waveform analysis applications.

- **Hilbert Transformers**

Hilbert transformer FIR filters shift the phase of all frequency components by 90 degrees, creating the analytic representation of a signal. They are crucial for single-sideband modulation, envelope detection, and instantaneous frequency analysis in communication systems.

### 2.5.2 Classification by Symmetry

- **Linear-Phase FIR Filters**

Linear-phase filters have coefficient symmetry that ensures constant group delay across all frequencies, preserving the shape of the filtered signal. They are further classified into:

  - **Type I**: Even filter order with even symmetry (h[n] = h[N-1-n])
  - **Type II**: Odd filter order with even symmetry (h[n] = h[N-1-n])
  - **Type III**: Even filter order with odd symmetry (h[n] = -h[N-1-n])
  - **Type IV**: Odd filter order with odd symmetry (h[n] = -h[N-1-n])

- **Minimum-Phase FIR Filters**

These filters concentrate the impulse response energy near the beginning, minimizing the group delay. While they lack linear phase, they can achieve a specified magnitude response with fewer coefficients than linear-phase filters, making them valuable for applications with strict latency constraints.

- **Maximum-Phase FIR Filters**

The energy of maximum-phase filter impulse responses is concentrated toward the end. These are less common but find use in specialized applications like channel equalization.

## 2.6 Implementation Structures

- **Direct Form**

The most straightforward implementation directly implements the convolution sum. It's conceptually simple but may not be computationally efficient for higher-order filters.

- **Transposed Form**

A mathematically equivalent structure that can offer implementation advantages, particularly for pipelined hardware designs.

## 2.7 Key Features of FIR Filters

- **Linear Phase Response**: Ensures no phase distortion, which is critical in communication and audio processing applications.
- **Absolute Stability**: Unlike IIR filters, FIR filters do not rely on feedback, making them inherently stable.
- **Higher Computational Complexity**: Requires more computations due to the absence of recursive structures.
- **Efficient Hardware Implementation**: Well-suited for FPGA and ASIC implementations due to their simple structure.

## 2.8 Comparison of FIR and IIR Filters

Table 2.3: FIR vs IIR Filters

| Feature | FIR Filter | IIR Filter |
|---|---|---|
| **Stability** | Always stable | Can be unstable due to feedback |
| **Phase Response** | Linear phase | Non-linear phase |
| **Computational Complexity** | Higher order required | Lower order |
| **Implementation** | Uses only multipliers and adders | Uses recursive structures |
| **Memory Requirement** | Higher | Lower |
| **Impulse Response** | Finite duration | Infinite duration |

## 2.9 Importance of FIR Filters in DSP Applications

FIR (Finite Impulse Response) filters play a crucial role in modern digital signal processing (DSP) applications across numerous industries. Their importance stems from several key characteristics that make them indispensable in signal processing systems:

### 2.9.1 Guaranteed Stability

FIR filters are inherently stable due to their non-recursive structure. Since they don't use feedback, they cannot become unstable regardless of the coefficient values. This guaranteed stability is critical in mission-critical applications like medical devices, aerospace systems, and telecommunications, where system failure could have serious consequences.

### 2.9.2 Linear Phase Response

One of the most valuable properties of FIR filters is their ability to provide perfect linear phase response through symmetric coefficient design. This characteristic ensures that all frequency components experience the same time delay, preserving the shape of the filtered signal. This is essential in:

- **Image processing**: Prevents spatial distortion
- **Audio processing**: Maintains temporal relationships between harmonics
- **Data communications**: Preserves pulse shapes and timing for reliable data recovery
- **Medical signal analysis**: Ensures accurate representation of physiological waveforms

### 2.9.3 Precise Frequency Response Control

FIR filters allow designers to control frequency response characteristics with minimal constraints. This flexibility enables:

- Implementation of arbitrary magnitude responses
- Sharp transition bands when needed
- Precise control over passband ripple and stopband attenuation
- Custom-shaped responses for specialized applications

# Chapter 3

# Adders Overview

## 3.1 Introduction

Adders are fundamental components in digital arithmetic circuits, widely used in digital signal processing (DSP), cryptographic applications, microprocessors, and arithmetic logic units (ALUs). The efficiency of an adder depends on three key parameters:

- **Speed (Propagation Delay):** Determines how fast the adder produces the sum.
- **Power Consumption:** Essential for energy-efficient and battery-operated applications.
- **Area Utilization:** Defines how much silicon real estate the adder occupies.

Among different adder architectures like Ripple Carry Adder (RCA), Carry-Select Adder (CSA), parallel-prefix adders have emerged as a high-performance solution due to their ability to compute carry signals efficiently. This chapter discusses key parallel-prefix adder architectures: Brent-Kung (BKA), Kogge-Stone (KSA), Ling, and Sparse Kogge-Stone (SKSA), and evaluates their suitability for 16-bit and 32-bit implementations in high-speed FIR filters.

## 3.2 Parallel Prefix Adders: An Overview

Parallel-prefix adders improve carry computation using tree-based structures, significantly reducing the delay compared to conventional ripple-carry and carry-lookahead adders. The operation of a parallel-prefix adder consists of three key stages:

### (a) Pre-Processing Stage

In this stage, the Generate (G) and Propagate (P) signals are computed for each bit position as follows:

$$G_i = A_i \cdot B_i$$

$$P_i = A_i \oplus B_i$$

### (b) Carry Generation Using Prefix Tree

Parallel-prefix networks use Black and Gray cells to propagate and merge carries efficiently. The fundamental recursive equations governing prefix computation are:

$$G_i^{(j)} = G_i^{(j-1)} + \left(P_i^{(j-1)} \cdot G_{i-2^{j-1}}^{(j-1)}\right)$$

$$P_i^{(j)} = P_i^{(j-1)} \cdot P_{i-2^{j-1}}^{(j-1)}$$

**(c) Sum Computation**

The final sum bits are computed as:

$$S_i = P_i \oplus C_{i-1}$$

Where;

A, B are inputs,

G is Generate and P is Propagate,

i and j represent bit positions,

S is Sum and $C_{i-1}$ is the final carry obtained from the prefix network.


Figure 2: Parallel Prefix Adder Architecture

## 3.3 Detailed Analysis of Selected Adder Architectures

### 3.3.1 Brent-Kung Adder (BKA)

The Brent-Kung Adder (BKA) is designed to optimize fan out and interconnect complexity while maintaining efficient carry computation.

**Architectural Insights:**

- The prefix network follows a logarithmic tree structure to compute carry signals.
- Uses a minimum number of black and grey cells, making it area-efficient.
- Has $2\log_2 N - 1$ logic levels for carry computation.

**Working Principle:**

- Generates local carries first, followed by a hierarchical merging process that propagates carry information.

The Brent-Kung adder uses a parallel prefix approach with the following key equations:

Generate and Propagate Signals:

$$g_i = a_i \cdot b_i$$

$$p_i = a_i \oplus b_i$$

Group Generate and Propagate:

$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$

$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

Carry calculation:

$$c_{i+1} = G_{i:0}$$

Sum calculation:

$$s_i = p_i \oplus c_i$$

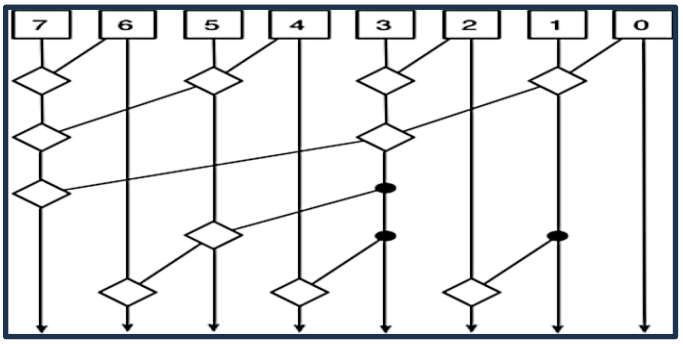



Figure 3: Architecture of 8-bit BKA Adder

### 3.3.2 Kogge-Stone Adder (KSA)

The Kogge-Stone Adder (KSA) is the fastest parallel-prefix adder and is widely used in high-performance computing. Kogge Stone Adder (KSA) is a parallel prefix form carry look ahead adder (CLA). it is widely considered the fastest adder and is widely used in the industry for high-performance arithmetic circuits. The architecture of KSA is quite similar to CLA.

**Architectural Insights:**

- Employs maximum parallelism in carry propagation.
- Features a dense interconnect network that ensures minimal delay.
- Requires $\log_2 N$ prefix computation stages.

**Working Principle:**

- Every bit position computes its carry independently using a full carry computation tree.
- Uses an extensive number of black and grey cells, leading to high wiring complexity.

Kogge-Stone adder employs maximum parallelism with these equations:

Generate and Propagate Signals:

$$g_i = a_i \cdot b_i$$

$$p_i = a_i \oplus b_i$$

Group Generate and Propagate:

$$G_{i:j} = G_{i:j-1} + P_{i:j-1} \cdot G_{i-2^{j-1}:j-1}$$

$$P_{i:j} = P_{i:j-1} \cdot P_{i-2^{j-1}:j-1}$$

Carry calculation:

$$c_{i+1} = G_{i:0}$$

Sum calculation:

$$s_i = p_i \oplus c_i$$

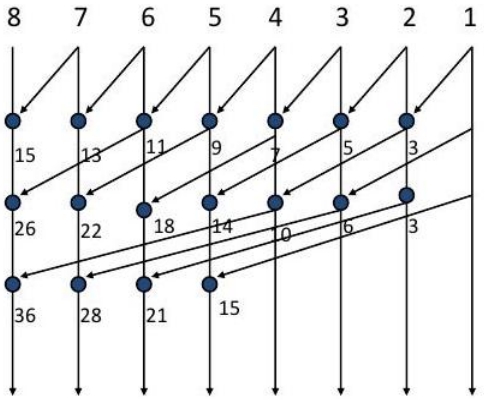



Figure 4: Architecture of 8-bit KSA Adder

### 3.3.3 Ling Adder

The Ling Adder is an optimized parallel-prefix adder that modifies the generate and propagate functions to improve power efficiency. It is an advanced variation of the carry-lookahead adder designed to enhance the speed of binary addition while minimizing hardware complexity. It achieves this by efficiently generating intermediate signals that simplify the computation of carry bits. The Ling Adder is known for its ability to optimize the critical path delay, making it a valuable component in high-speed arithmetic circuits and performance-sensitive applications.

**Architectural Insights:**

- Reduces complexity in carry computation by reordering the traditional carry-generation logic.
- Uses fewer gates than KSA, leading to reduced power consumption

1. Intermediate Signal Generation: The Ling Adder creates intermediate signals to simplify carry computation, leading to faster addition.
2. Efficient Carry Propagation: It uses generate (g) and propagate (p) signals to optimize carry propagation, reducing the critical path delay.
3. Optimized Area and Delay: By reducing the number of logic gates, the Ling Adder achieves a balance between speed and resource usage, making it ideal for high-performance systems

**Working Principle:**

The Ling adder is a variation of the parallel prefix adder that uses the same basic Propagate and Generate signals as conventional adders, but it modifies the carry computation using a pseudo-generate signal. This modification reduces logic complexity in the prefix network and enables faster carry generation.

- **Ling Adder Equations:**

The Ling adder is an optimized parallel prefix adder that modifies the carry propagation equations to improve efficiency.

Generate and Propagate signals:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \cdot B_i$$

Pseudo-generate(H) definition:

$$H_i = G_i + P_i \cdot G_{i-1}$$

Carry calculation:

$$C_i = H_{i-1}$$

This small change in the carry logic allows faster and simpler prefix tree construction, especially beneficial in large-bit adders. This transformation minimizes circuit complexity and reduces dynamic power.

Sum calculation:

$$S_i = P_i \oplus H_{i-1}$$

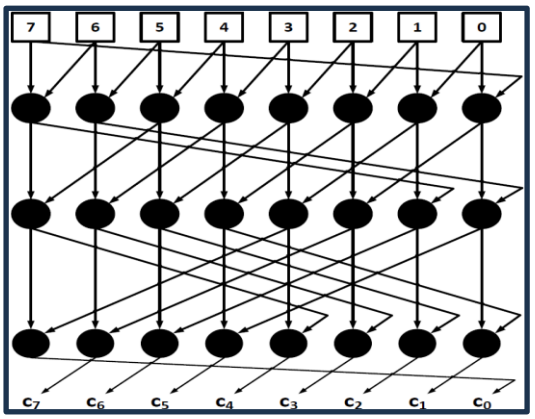



Figure 5: Architecture of 8-bit LING Adder

### 3.3.4 Sparse Kogge-Stone Adder (SKSA)

The Sparse Kogge-Stone Adder (SKSA) is a hybrid prefix adder that balances speed and area efficiency. The sparse Kogge-Stone adder is a subtype of Kogge-Stone adder that uses fewer black cells and grey cells.

Strategically eliminates some of the prefix computation nodes and wires. It introduces a sparsity factor that determines which levels have reduced nodes.

**Architectural Insights:**

- Similar to KSA but computes carry at specific intervals instead of every bit position.
- Reducing interconnect complexity, making it more scalable.

**Working Principle:**

- Instead of full prefix computation, carry propagation is skipped at specific bit intervals:
- This reduces the number of black and grey cells, improving power and area efficiency.

Generate and Propagate Signals:

$$P_i = A_i \oplus B_i$$

$$G_i = A_i \cdot B_i$$

*Sparsity factor k (typically k=2 or k=4)*

*Group Generate and Propagate are computed only for every k-th bit:*

$$G_{i:j} = G_i + P_i \cdot G_{i-1:j} \quad \text{(for selected i)}$$

$$P_{i:j} = P_i \cdot P_{i-1:j} \quad \text{(for selected i)}$$

Ripple within each k-bit group:

$$C_i = \text{From prefix network or local logic}$$

$$c_{i+1} = g_i + p_i \cdot c_i \quad \text{for intermediate carries}$$

Sum calculation:

$$S_i = P_i \oplus C_i$$

## 3.4 Comparison of 16-bit vs. 32-bit Implementation Results

Table 3.1: 16-bit Conventional Adders Results

| Adder Name | Dynamic power (W) | Static power (W) | Total Power (W) | Delay (ns) | Power delay product (W. ns) | Slice logic | LUT logic |
|---|---|---|---|---|---|---|---|
| BKA | 11.019 | 0.158 | 11.178 | 12.196 | 136.293 | 8 | 26 |
| KSA | 11.326 | 0.164 | 11.490 | 12.081 | 138.810 | 14 | 44 |
| LING | 9.616 | 0.135 | 9.751 | 14.402 | 140.433 | 8 | 29 |
| SKSA | 10.910 | 0.156 | 11.066 | 16.095 | 178.107 | 7 | 17 |

Table 3.2: 32-bit Conventional Adders Results

| Adder Name | Dynamic Power (W) | Static power (W) | Total Power (W) | Delay (ns) | Power delay product (W. ns) | Slice logic | LUT logic |
|---|---|---|---|---|---|---|---|
| BKA | 22.507 | 0.485 | 22.991 | 22.469 | 516.584 | 24 | 64 |
| KSA | 23.553 | 0.485 | 24.037 | 17.409 | 418.460 | 41 | 134 |
| LING | 22.257 | 0.485 | 22.742 | 25.407 | 577.805 | 23 | 55 |
| SKSA | 22.259 | 0.458 | 22.744 | 24.585 | 559.161 | 23 | 55 |

## 3.5 Synthesis Reports of Adders

### 3.5.1 Reports of 16-bit Conventional Adders

*3.5.1.1 BKA 16-bit:*

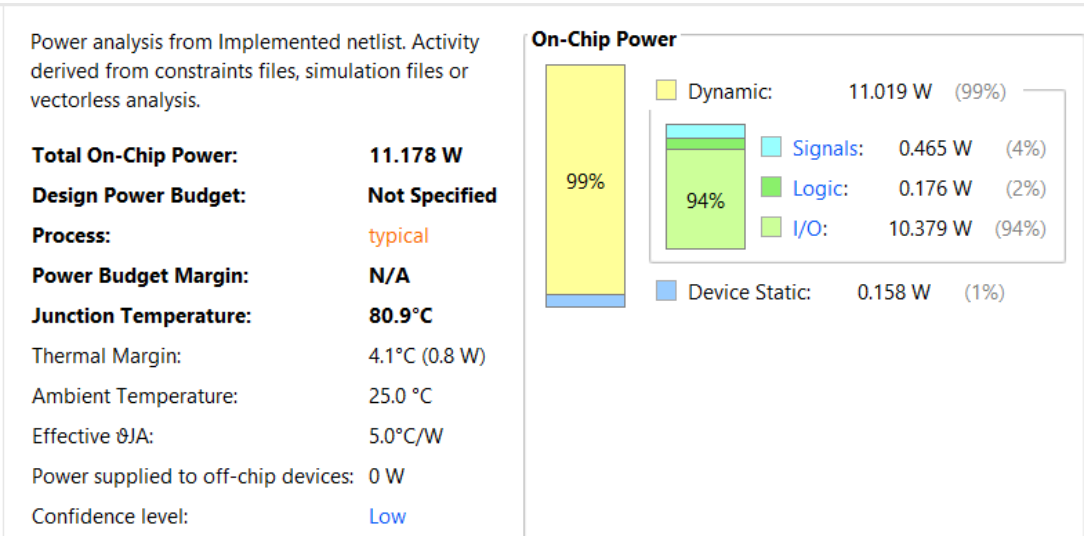



Figure 6: Bka_16-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 7 | 4 | a[0] | sum[15] | 12.196 | 4.409 | 7.787 | ∞ | input port clock |
| Path 2 | ∞ | 7 | 4 | a[0] | sum[10] | 12.029 | 4.871 | 7.158 | ∞ | input port clock |
| Path 3 | ∞ | 7 | 4 | a[0] | sum[11] | 11.951 | 4.645 | 7.307 | ∞ | input port clock |
| Path 4 | ∞ | 7 | 4 | a[0] | sum[14] | 11.914 | 4.407 | 7.506 | ∞ | input port clock |
| Path 5 | ∞ | 7 | 4 | a[0] | sum[9] | 11.790 | 4.629 | 7.161 | ∞ | input port clock |
| Path 6 | ∞ | 6 | 4 | a[0] | sum[13] | 11.687 | 4.515 | 7.172 | ∞ | input port clock |
| Path 7 | ∞ | 6 | 4 | a[0] | sum[12] | 11.415 | 4.277 | 7.138 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 4 | a[0] | cout | 11.415 | 4.410 | 7.005 | ∞ | input port clock |
| Path 9 | ∞ | 6 | 4 | a[0] | sum[6] | 11.314 | 4.750 | 6.564 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 4 | a[0] | sum[7] | 11.298 | 4.727 | 6.571 | ∞ | input port clock |

Figure 7: Bka_16-Timing

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 8 | 0 | 0 | 8150 | 0.10 |
| SLICEL | 8 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 26 | 0 | 0 | 32600 | 0.08 |
| using O5 output only | 0 | | | | |
| using O6 output only | 12 | | | | |
| using O5 and O6 | 14 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 8: Bka_16-Area

### *3.5.1.2 KSA 16-bit:*

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **11.49 W** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **82.4°C** |
| Thermal Margin: | 2.6°C (0.5 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

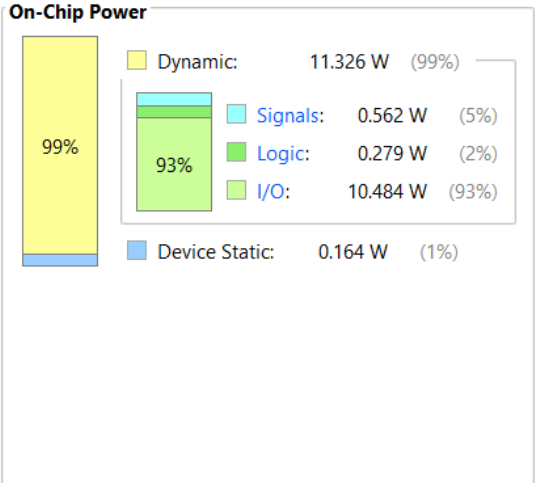



Figure 9: Ksa_16-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 6 | 5 | A[1] | Cout | 12.081 | 4.512 | 7.569 | ∞ | input port clock |
| Path 2 | ∞ | 6 | 5 | A[1] | Sum[12] | 11.846 | 4.504 | 7.343 | ∞ | input port clock |
| Path 3 | ∞ | 6 | 5 | A[5] | Sum[13] | 11.254 | 4.279 | 6.975 | ∞ | input port clock |
| Path 4 | ∞ | 6 | 5 | A[1] | Sum[10] | 11.193 | 4.286 | 6.907 | ∞ | input port clock |
| Path 5 | ∞ | 6 | 6 | A[0] | Sum[15] | 10.949 | 4.055 | 6.894 | ∞ | input port clock |
| Path 6 | ∞ | 6 | 6 | A[0] | Sum[11] | 10.924 | 4.061 | 6.864 | ∞ | input port clock |
| Path 7 | ∞ | 6 | 5 | A[1] | Sum[14] | 10.918 | 4.048 | 6.870 | ∞ | input port clock |
| Path 8 | ∞ | 5 | 5 | A[1] | Sum[8] | 10.660 | 4.382 | 6.279 | ∞ | input port clock |
| Path 9 | ∞ | 6 | 6 | A[0] | Sum[9] | 10.557 | 4.045 | 6.512 | ∞ | input port clock |
| Path 10 | ∞ | 5 | 5 | A[1] | Sum[4] | 10.392 | 4.382 | 6.010 | ∞ | input port clock |

Figure 10: Ksa_16-Timing

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 14 | 0 | 0 | 8150 | 0.17 |
| SLICEL | 14 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 44 | 0 | 0 | 32600 | 0.13 |
| using O5 output only | 0 | | | | |
| using O6 output only | 22 | | | | |
| using O5 and O6 | 22 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 11: Ksa_16-Area

### *3.5.1.3 LING 16-bit:*

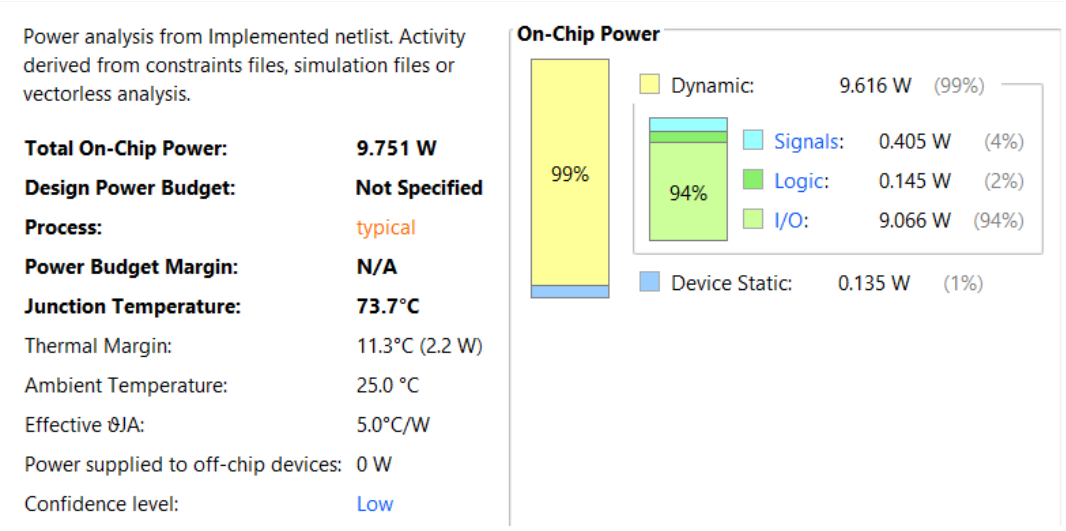



Figure 12: Ling_16-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 10 | 3 | A[1] | Sum[14] | 14.402 | 4.544 | 9.858 | ∞ | input port clock |
| Path 2 | ∞ | 10 | 3 | A[1] | Sum[15] | 14.137 | 4.545 | 9.592 | ∞ | input port clock |
| Path 3 | ∞ | 9 | 3 | A[1] | Sum[12] | 13.477 | 4.414 | 9.063 | ∞ | input port clock |
| Path 4 | ∞ | 9 | 3 | A[1] | Sum[13] | 13.368 | 4.415 | 8.953 | ∞ | input port clock |
| Path 5 | ∞ | 8 | 3 | A[1] | Sum[10] | 12.542 | 4.298 | 8.244 | ∞ | input port clock |
| Path 6 | ∞ | 8 | 3 | A[1] | Sum[11] | 12.436 | 4.303 | 8.133 | ∞ | input port clock |
| Path 7 | ∞ | 10 | 3 | A[1] | CarryOut | 12.005 | 4.546 | 7.458 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 3 | A[1] | Sum[8] | 11.593 | 4.168 | 7.425 | ∞ | input port clock |
| Path 9 | ∞ | 7 | 3 | A[1] | Sum[9] | 11.477 | 4.163 | 7.314 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 3 | A[1] | Sum[6] | 11.025 | 4.049 | 6.976 | ∞ | input port clock |

Figure 13: Ling_16-Timing

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 8 | 0 | 0 | 8150 | 0.10 |
| SLICEL | 8 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 29 | 0 | 0 | 32600 | 0.09 |
| using O5 output only | 0 | | | | |
| using O6 output only | 27 | | | | |
| using O5 and O6 | 2 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 14: Ling_16-Area

### *3.5.1.4 SKSA 16-bit:*

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **11.066 W** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **80.3°C** |
| Thermal Margin: | 4.7°C (1.0 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

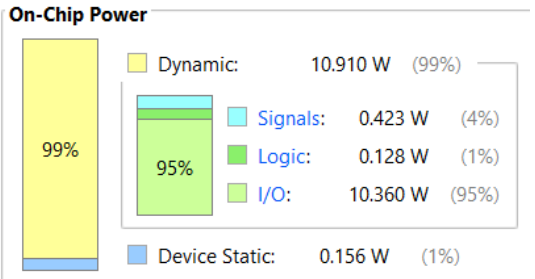



Figure 15: Sksa_16-Power

**Unconstrained Paths - NONE - NONE - Setup**

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 10 | 4 | a[0] | sum[14] | 16.095 | 6.104 | 9.990 | ∞ | input port clock |
| Path 2 | ∞ | 10 | 4 | a[0] | cout | 14.967 | 5.869 | 9.099 | ∞ | input port clock |
| Path 3 | ∞ | 10 | 4 | a[0] | sum[15] | 14.705 | 5.640 | 9.065 | ∞ | input port clock |
| Path 4 | ∞ | 9 | 4 | a[0] | sum[13] | 14.676 | 5.510 | 9.167 | ∞ | input port clock |
| Path 5 | ∞ | 9 | 4 | a[0] | sum[12] | 14.289 | 5.508 | 8.780 | ∞ | input port clock |
| Path 6 | ∞ | 8 | 3 | a[0] | sum[10] | 13.368 | 5.196 | 8.172 | ∞ | input port clock |
| Path 7 | ∞ | 8 | 3 | a[0] | sum[11] | 13.357 | 5.202 | 8.156 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 3 | a[0] | sum[9] | 12.897 | 5.292 | 7.605 | ∞ | input port clock |
| Path 9 | ∞ | 7 | 3 | a[0] | sum[8] | 12.394 | 5.066 | 7.328 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 3 | a[0] | sum[7] | 11.420 | 4.694 | 6.726 | ∞ | input port clock |

Figure 16: Sksa_16-Timing

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 7 | 0 | 0 | 8150 | 0.09 |
| SLICEL | 7 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 17 | 0 | 0 | 32600 | 0.05 |
| using O5 output only | 0 | | | | |
| using O6 output only | 8 | | | | |
| using O5 and O6 | 9 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 17: Sksa_16-Area

## 3.5.2 Reports of 32-bit Conventional Adders

### *3.5.2.1 BKA 32-bit:*

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **22.991 W (Junction temp exceeded!)** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **125.0°C** |
| Thermal Margin: | -54.9°C (-10.6 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

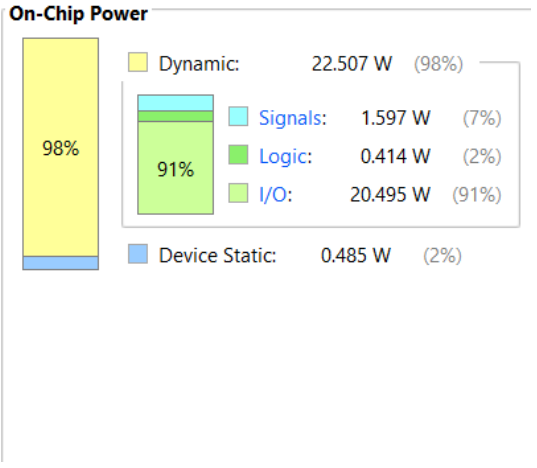



Figure 18: Bka_32-Power

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 12 | 5 | cin | sum[29] | 22.469 | 5.708 | 16.760 | ∞ | input port clock |
| Path 2 | ∞ | 12 | 5 | cin | sum[30] | 21.913 | 5.471 | 16.442 | ∞ | input port clock |
| Path 3 | ∞ | 11 | 5 | cin | sum[27] | 21.767 | 5.794 | 15.972 | ∞ | input port clock |
| Path 4 | ∞ | 11 | 5 | cin | sum[26] | 21.603 | 6.050 | 15.553 | ∞ | input port clock |
| Path 5 | ∞ | 11 | 5 | cin | sum[25] | 21.544 | 5.801 | 15.742 | ∞ | input port clock |
| Path 6 | ∞ | 12 | 5 | cin | sum[28] | 21.400 | 5.464 | 15.936 | ∞ | input port clock |
| Path 7 | ∞ | 12 | 5 | cin | sum[31] | 21.151 | 5.236 | 15.915 | ∞ | input port clock |
| Path 8 | ∞ | 9 | 5 | cin | sum[20] | 19.855 | 5.331 | 14.524 | ∞ | input port clock |
| Path 9 | ∞ | 10 | 5 | cin | sum[23] | 19.764 | 5.430 | 14.334 | ∞ | input port clock |
| Path 10 | ∞ | 10 | 5 | cin | sum[24] | 19.527 | 5.445 | 14.082 | ∞ | input port clock |

Figure 19: Bka_32-Timing

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 24 | 0 | 0 | 8150 | 0.29 |
| SLICEL | 24 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 64 | 0 | 0 | 32600 | 0.20 |
| using O5 output only | 0 | | | | |
| using O6 output only | 32 | | | | |
| using O5 and O6 | 32 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 20: Bka_32-Area

### *3.5.2.2 KSA 32-bit:*

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **24.037 W (Junction temp exceeded!)** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **125.0°C** |
| Thermal Margin: | -60.2°C (-11.7 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

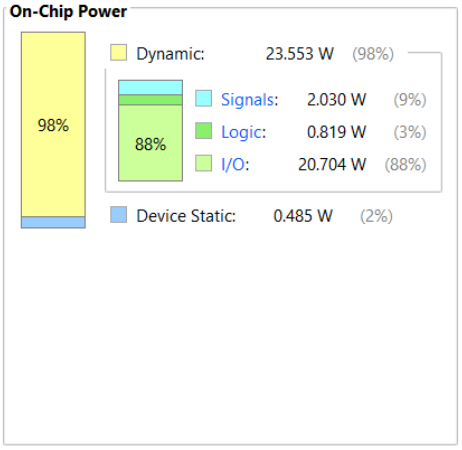



Figure 21: Ksa_32-Power

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 6 | 5 | cin | sum[28] | 17.409 | 4.026 | 13.384 | ∞ | input port clock |
| Path 2 | ∞ | 6 | 7 | b[1] | sum[29] | 16.788 | 4.301 | 12.488 | ∞ | input port clock |
| Path 3 | ∞ | 6 | 7 | b[1] | sum[23] | 16.549 | 4.273 | 12.276 | ∞ | input port clock |
| Path 4 | ∞ | 6 | 5 | cin | sum[20] | 16.488 | 4.029 | 12.459 | ∞ | input port clock |
| Path 5 | ∞ | 6 | 7 | b[1] | sum[31] | 16.256 | 4.284 | 11.972 | ∞ | input port clock |
| Path 6 | ∞ | 6 | 7 | b[25] | sum[27] | 16.081 | 4.272 | 11.809 | ∞ | input port clock |
| Path 7 | ∞ | 6 | 7 | b[1] | sum[25] | 16.080 | 4.292 | 11.788 | ∞ | input port clock |
| Path 8 | ∞ | 6 | 5 | cin | cout | 16.066 | 4.044 | 12.022 | ∞ | input port clock |
| Path 9 | ∞ | 6 | 10 | b[8] | sum[21] | 16.053 | 4.530 | 11.524 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 10 | b[8] | sum[22] | 15.786 | 4.266 | 11.521 | ∞ | input port clock |

Figure 22: Ksa_32-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 41 | 0 | 0 | 8150 | 0.50 |
| SLICEL | 30 | 0 | | | |
| SLICEM | 11 | 0 | | | |
| LUT as Logic | 134 | 0 | 0 | 32600 | 0.41 |
| using O5 output only | 0 | | | | |
| using O6 output only | 69 | | | | |
| using O5 and O6 | 65 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 23: Ksa_32-Area

### *3.5.2.3 LING 32-bit:*

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **22.742 W (Junction temp exceeded!)** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **125.0°C** |
| Thermal Margin: | -53.7°C (-10.4 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

Launch Power Constraint Advisor to find and fix invalid switching activity

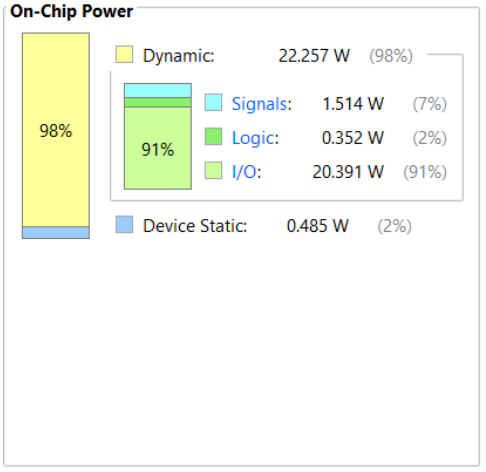
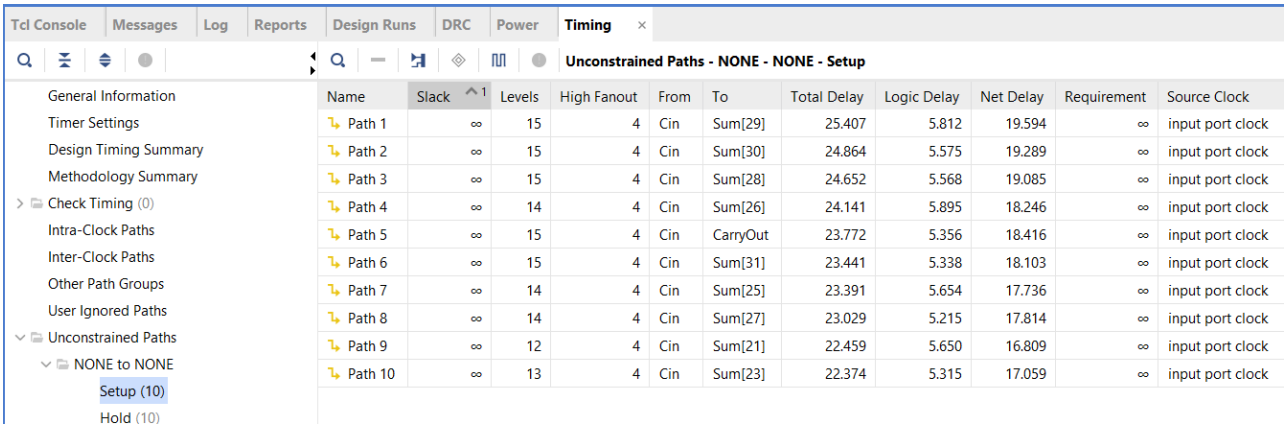



Figure 24: Ling_32-Power




| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 15 | 4 | Cin | Sum[29] | 25.407 | 5.812 | 19.594 | ∞ | input port clock |
| Path 2 | ∞ | 15 | 4 | Cin | Sum[30] | 24.864 | 5.575 | 19.289 | ∞ | input port clock |
| Path 3 | ∞ | 15 | 4 | Cin | Sum[28] | 24.652 | 5.568 | 19.085 | ∞ | input port clock |
| Path 4 | ∞ | 14 | 4 | Cin | Sum[26] | 24.141 | 5.895 | 18.246 | ∞ | input port clock |
| Path 5 | ∞ | 15 | 4 | Cin | CarryOut | 23.772 | 5.356 | 18.416 | ∞ | input port clock |
| Path 6 | ∞ | 15 | 4 | Cin | Sum[31] | 23.441 | 5.338 | 18.103 | ∞ | input port clock |
| Path 7 | ∞ | 14 | 4 | Cin | Sum[25] | 23.391 | 5.654 | 17.736 | ∞ | input port clock |
| Path 8 | ∞ | 14 | 4 | Cin | Sum[27] | 23.029 | 5.215 | 17.814 | ∞ | input port clock |
| Path 9 | ∞ | 12 | 4 | Cin | Sum[21] | 22.459 | 5.650 | 16.809 | ∞ | input port clock |
| Path 10 | ∞ | 13 | 4 | Cin | Sum[23] | 22.374 | 5.315 | 17.059 | ∞ | input port clock |

Figure 25: Ling_32-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 23 | 0 | 0 | 8150 | 0.28 |
| SLICEL | 23 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 55 | 0 | 0 | 32600 | 0.17 |
| using O5 output only | 0 | | | | |
| using O6 output only | 31 | | | | |
| using O5 and O6 | 24 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 26: Ling_32-Area

### *3.5.2.4 SKSA 32-bit:*

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **22.744 W (Junction temp exceeded!)** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **125.0°C** |
| Thermal Margin: | -53.7°C (-10.4 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

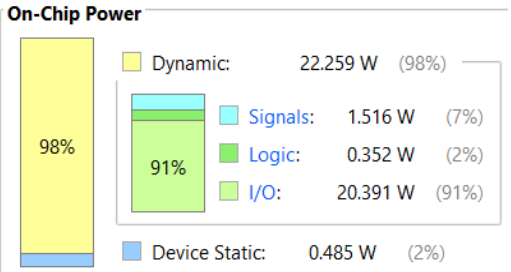



Figure 27: Sksa_32-Power

**Unconstrained Paths - NONE - NONE - Setup**

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 15 | 4 | cin | sum[29] | 24.585 | 5.810 | 18.774 | ∞ | input port clock |
| Path 2 | ∞ | 15 | 4 | cin | sum[30] | 24.042 | 5.573 | 18.469 | ∞ | input port clock |
| Path 3 | ∞ | 14 | 4 | cin | sum[26] | 24.019 | 5.929 | 18.090 | ∞ | input port clock |
| Path 4 | ∞ | 15 | 4 | cin | sum[28] | 23.830 | 5.566 | 18.264 | ∞ | input port clock |
| Path 5 | ∞ | 15 | 4 | cin | cout | 23.777 | 5.356 | 18.421 | ∞ | input port clock |
| Path 6 | ∞ | 15 | 4 | cin | sum[31] | 23.446 | 5.338 | 18.108 | ∞ | input port clock |
| Path 7 | ∞ | 14 | 4 | cin | sum[25] | 23.269 | 5.688 | 17.581 | ∞ | input port clock |
| Path 8 | ∞ | 14 | 4 | cin | sum[27] | 22.209 | 5.215 | 16.994 | ∞ | input port clock |
| Path 9 | ∞ | 13 | 4 | cin | sum[24] | 22.038 | 5.330 | 16.708 | ∞ | input port clock |
| Path 10 | ∞ | 12 | 4 | cin | sum[21] | 21.960 | 5.652 | 16.308 | ∞ | input port clock |

Figure 28: Sksa_32-Timing

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 23 | 0 | 0 | 8150 | 0.28 |
| SLICEL | 23 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 55 | 0 | 0 | 32600 | 0.17 |
| using O5 output only | 0 | | | | |
| using O6 output only | 31 | | | | |
| using O5 and O6 | 24 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 29: Sksa_32-Area

# Chapter 4

# Hybrid Adders

## 4.1 Introduction

As FIR filters demand optimized arithmetic units for efficient computation, conventional adders often fall short in meeting the low-power and high-speed requirements of modern VLSI systems. To address this, hybrid adder architectures were developed by combining the strengths of various parallel-prefix adders. These hybrid structures aim to achieve a balance between power consumption, area, and delay, thereby enhancing the overall performance of FIR filter implementations.

This chapter explores both initial hybrid designs without enable control and further enhanced versions incorporating enable signal-based power gating techniques. The technical implementation, synthesis results, and rationale behind the final proposed design are discussed in detail.

## 4.2 Initial Hybrid Adder Designs

To begin the design space exploration, 16-bit hybrid adders were created by combining two different 8-bit conventional adders. The objective was to assess the performance improvement from hybridizing adders known for different trade-offs in area, delay, and power. Each design was synthesized using Xilinx Vivado to analyze performance metrics.

### 4.2.1 Hybrid Adder: LING8 + BKA8

- **Design Rationale**: Combines the power-efficient Ling adder with the area-optimized Brent-Kung adder.
- **Architecture**: The lower 8 bits are processed using the Ling adder, while the upper 8 bits use the Brent-Kung adder.
- **Advantages**: Expected moderate delay and low power.

### 4.2.2 Hybrid Adder: LING8 + KSA8

- **Design Rationale**: Combines power efficiency with high-speed Kogge-Stone logic.
- **Architecture**: Ling adder for LSBs and KSA for MSBs.
- **Observation**: While the delay improved, the power and area slightly increased.

### 4.2.3 Hybrid Adder: SKSA8 + BKA8

- **Design Rationale**: Balances speed (SKSA) and area (BKA).
- **Architecture**: SKSA for the lower half and BKA for the upper half.
- **Result**: Moderate performance across all metrics but didn't meet low-power requirements.

### 4.2.4 Hybrid Adder: LING8 + SKSA8

- **Design Rationale**: Targets a balance between low power and fast computation.
- **Architecture**: Ling adder for lower 8 bits and SKSA for upper 8 bits.
- **Observation**: Performed better among the four initial hybrids.

Despite moderate improvements, none of the above designs significantly outperformed conventional adders in all metrics. Hence required further optimization.

## 4.3 Power Optimization with Enable Signal Integration

To further improve power efficiency, the concept of enable-controlled operation was introduced. The Adder Enable Signal allows the adder logic to be activated only when needed, significantly reducing unnecessary dynamic switching activity.

### 4.3.1 Implementation of Enable Signal

Operand isolation or Power gating is employed in the proposed hybrid adder architectures to reduce unnecessary dynamic power dissipation. This technique introduces an enable control signal that acts as a gating mechanism for operand inputs. When the adder is not required to perform computation, the enable signal disables the propagation of input transitions through the functional unit, thereby suppressing switching activity and conserving power. In the proposed design, operand isolation is implemented by masking the inputs to the hybrid adder modules using logic gates (e.g., AND gates or multiplexers). The operand signals are passed to the adder only when enable = 1. When enable = 0, the inputs are forced to a fixed logic level (typically zero), and the internal nodes remain static. It is a part of the spurious-power suppression technique. This technique is particularly beneficial in architectures such as FIR filters where operations occur conditionally or sequentially. Operand isolation complements other power-saving techniques by targeting fine-grained switching activity without affecting the clock distribution or introducing latency.

- **Approach Used**: A conditional dataflow mechanism where the output is computed only when enable = 1. If enable = 0, the adder forces the output to zero, thereby avoiding switching.
- **Code-Level**: This is implemented via a high-level control signal (enable) integrated within always blocks in Verilog.

```
always @(*)
    begin
     if (enable)
        sum = A + B;
     else
        sum = 0;
end
```

- **Power Advantage**: Significantly reduces dynamic power, especially in pipelined FIR filter architectures where not all adders are used simultaneously.

## 4.4 Hybrid Adders 16-bit with Enable

After evaluating the enable signal mechanism, it was integrated into previously explored hybrid designs. The architecture was kept the same, but control logic was introduced around the core adder units.

Each input of 16 bits is divided into two 8-bit inputs to achieve the hybrid addition by combining different architectures to add the MSBs and LSBs of the input separately. Output is evaluated only if enable signal is high else output is forced to zero.

### 4.4.1 Hybrid Adder: LING + BKA with Enable

- **Architecture**: Lower 8 bits - Ling, Upper 8 bits - BKA.
- **Observation**: Improved power profile with reduced unnecessary toggling.

**Synthesis report**:

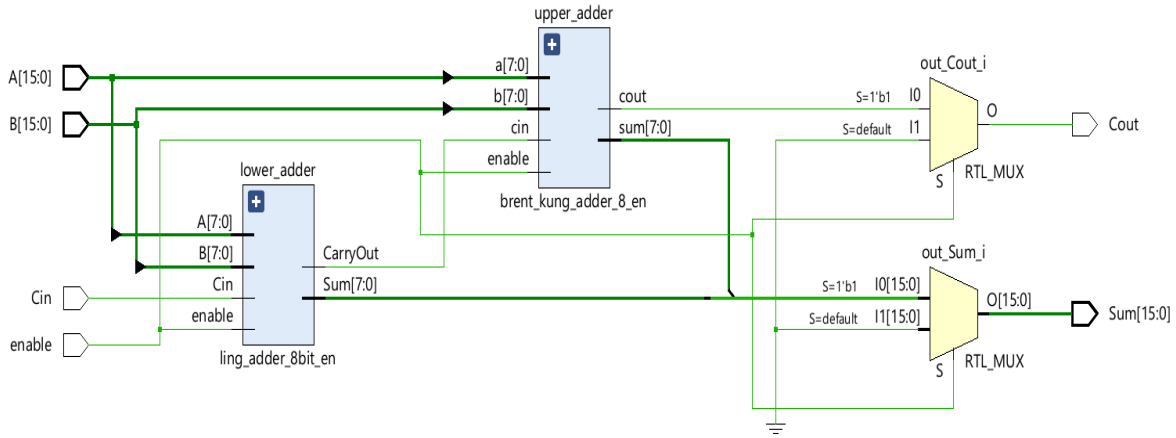



Figure 30: Hybrid_Adr16_Ling8+Bka8-Schematic

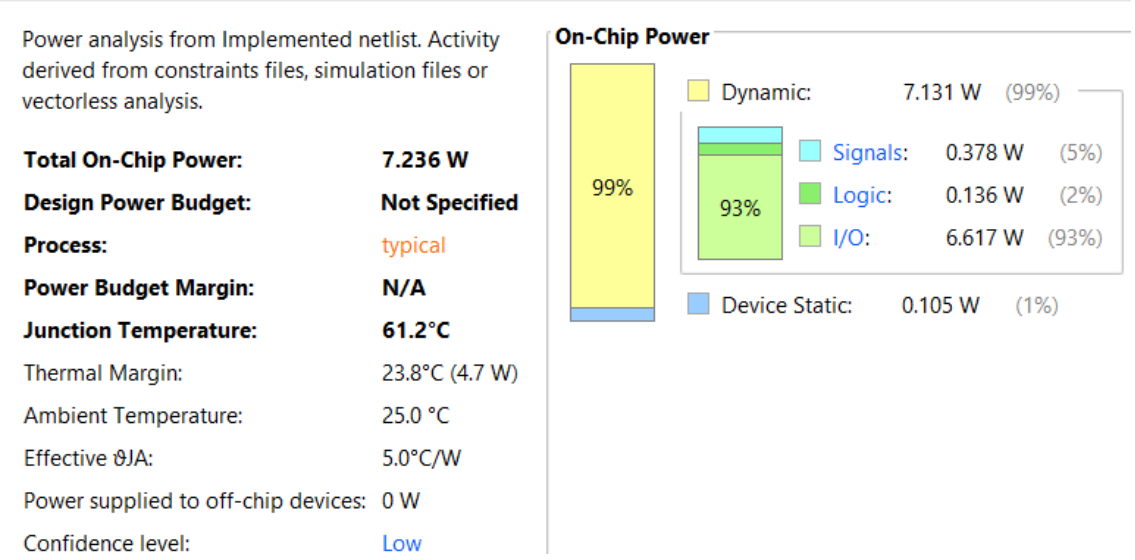



Figure 31: Hybrid_Adr16_Ling8+Bka8-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 9 | 4 | A[1] | Cout | 14.120 | 4.410 | 9.710 | ∞ | input port clock |
| Path 2 | ∞ | 9 | 4 | A[1] | Sum[13] | 13.190 | 4.420 | 8.770 | ∞ | input port clock |
| Path 3 | ∞ | 9 | 4 | A[1] | Sum[12] | 13.050 | 4.415 | 8.634 | ∞ | input port clock |
| Path 4 | ∞ | 9 | 4 | A[1] | Sum[14] | 13.044 | 4.421 | 8.622 | ∞ | input port clock |
| Path 5 | ∞ | 9 | 4 | A[1] | Sum[15] | 12.869 | 4.422 | 8.446 | ∞ | input port clock |
| Path 6 | ∞ | 8 | 4 | A[1] | Sum[10] | 12.269 | 4.303 | 7.966 | ∞ | input port clock |
| Path 7 | ∞ | 8 | 4 | A[1] | Sum[11] | 11.849 | 4.290 | 7.559 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 4 | A[1] | Sum[8] | 11.588 | 4.163 | 7.425 | ∞ | input port clock |
| Path 9 | ∞ | 7 | 4 | A[1] | Sum[9] | 11.385 | 4.174 | 7.211 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 4 | A[1] | Sum[7] | 10.360 | 4.044 | 6.317 | ∞ | input port clock |

Figure 32: Hybrid_Adr16_Ling8+Bka8-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 9 | 0 | 0 | 8150 | 0.11 |
| SLICEL | 9 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 30 | 0 | 0 | 32600 | 0.09 |
| using O5 output only | 0 | | | | |
| using O6 output only | 27 | | | | |
| using O5 and O6 | 3 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 33: Hybrid_Adr16_Ling8+Bka8-Area

### 4.4.2 Hybrid Adder: LING + KSA with Enable

- **Architecture**: Combination of Ling and KSA, both gated with enable.
- **Observation**: High performance with better delay than the BKA variant.

**Synthesis report**:

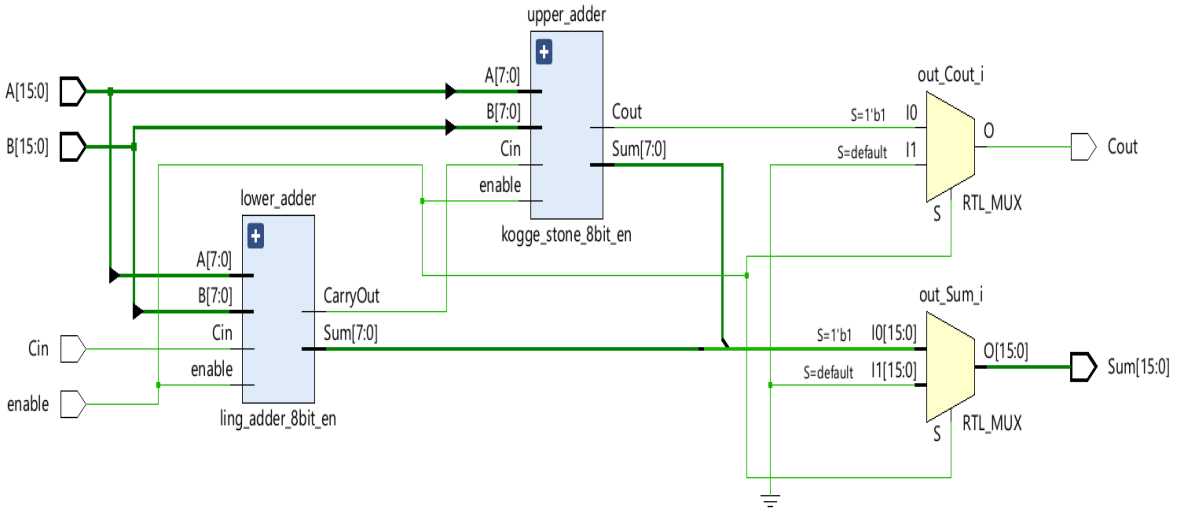



Figure 34: Hybrid_Adr16_Ling8_Ksa8-Schematic

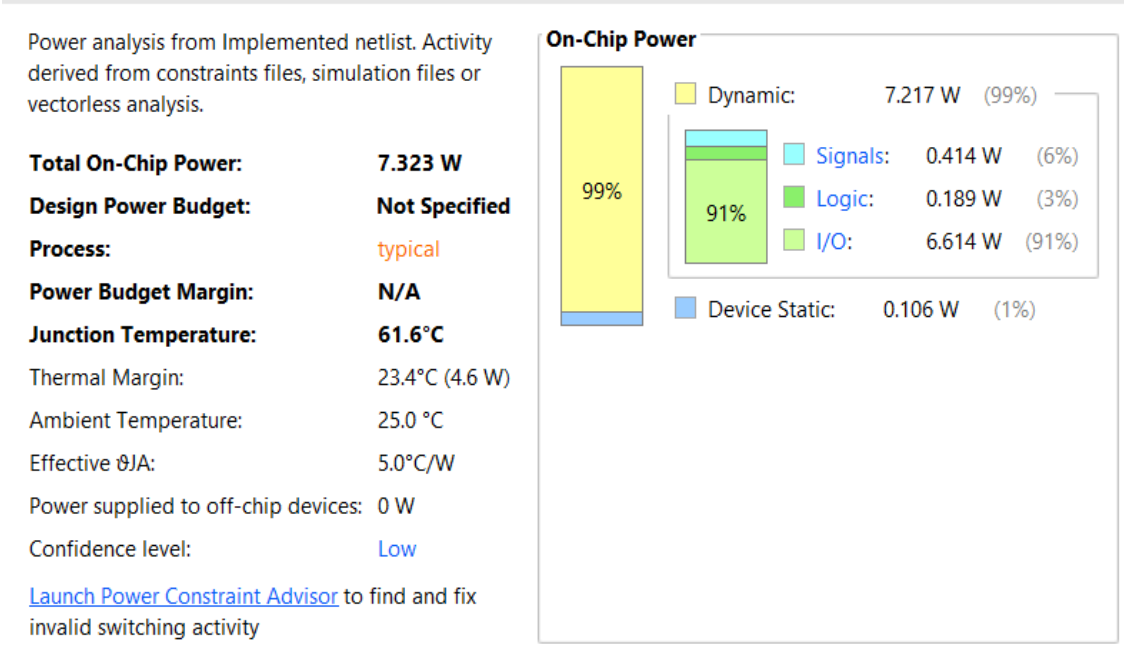



Figure 35: Hybrid_Adr16_Ling8_Ksa8-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 7 | 4 | A[0] | Cout | 13.639 | 4.167 | 9.472 | ∞ | input port clock |
| Path 2 | ∞ | 7 | 4 | A[0] | Sum[15] | 11.957 | 4.180 | 7.777 | ∞ | input port clock |
| Path 3 | ∞ | 7 | 4 | A[0] | Sum[12] | 11.528 | 4.401 | 7.127 | ∞ | input port clock |
| Path 4 | ∞ | 7 | 4 | A[0] | Sum[11] | 11.414 | 4.171 | 7.243 | ∞ | input port clock |
| Path 5 | ∞ | 7 | 4 | A[0] | Sum[14] | 11.295 | 4.179 | 7.116 | ∞ | input port clock |
| Path 6 | ∞ | 7 | 4 | A[0] | Sum[13] | 11.272 | 4.177 | 7.095 | ∞ | input port clock |
| Path 7 | ∞ | 7 | 4 | A[0] | Sum[9] | 11.156 | 4.179 | 6.977 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 4 | A[0] | Sum[10] | 10.864 | 4.185 | 6.680 | ∞ | input port clock |
| Path 9 | ∞ | 7 | 4 | A[0] | Sum[8] | 10.778 | 4.169 | 6.609 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 4 | A[0] | Sum[6] | 10.031 | 4.031 | 6.000 | ∞ | input port clock |

Figure 36: Hybrid_Adr16_Ling8_Ksa8-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 12 | 0 | 0 | 8150 | 0.15 |
| SLICEL | 11 | 0 | | | |
| SLICEM | 1 | 0 | | | |
| LUT as Logic | 38 | 0 | 0 | 32600 | 0.12 |
| using O5 output only | 0 | | | | |
| using O6 output only | 28 | | | | |
| using O5 and O6 | 10 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 37: Hybrid_Adr16_Ling8_Ksa8-Area

### 4.4.3 Hybrid Adder: SKSA + BKA with Enable

- **Architecture**: SKSA for lower bits and BKA for upper bits.
- **Result**: Balanced performance, reduced power due to enable control.

**Synthesis report:**

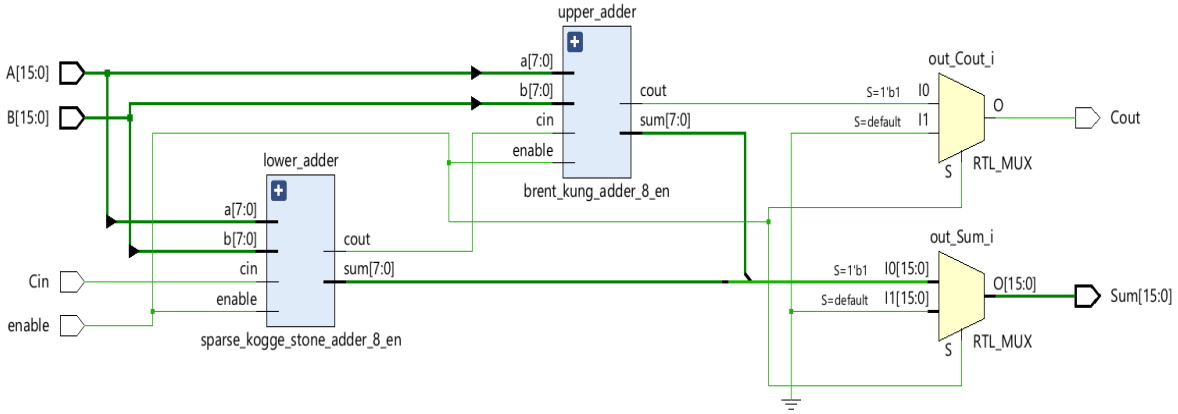



Figure 38: Hybrid_Adr16_Sksa8_Bka8-Schematic

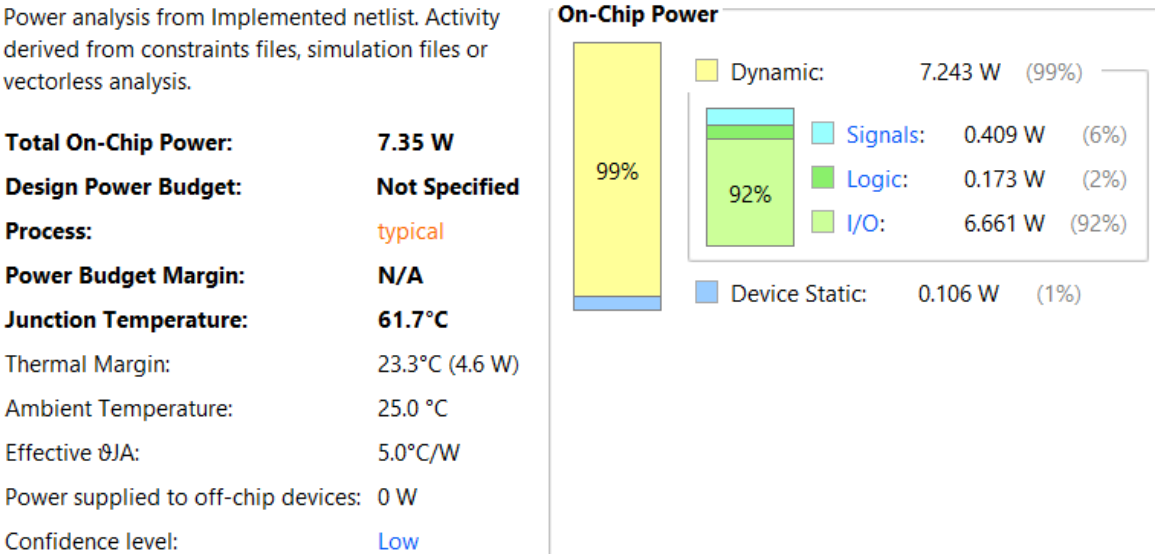



Figure 39: Hybrid_Adr16_Sksa8_Bka8-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 7 | 5 | A[4] | Cout | 14.208 | 4.411 | 9.797 | ∞ | input port clock |
| Path 2 | ∞ | 7 | 5 | A[4] | Sum[12] | 12.052 | 4.417 | 7.635 | ∞ | input port clock |
| Path 3 | ∞ | 7 | 5 | A[4] | Sum[14] | 12.048 | 4.423 | 7.625 | ∞ | input port clock |
| Path 4 | ∞ | 7 | 5 | A[4] | Sum[15] | 12.044 | 4.424 | 7.620 | ∞ | input port clock |
| Path 5 | ∞ | 7 | 5 | A[4] | Sum[11] | 12.022 | 4.416 | 7.607 | ∞ | input port clock |
| Path 6 | ∞ | 7 | 5 | A[4] | Sum[13] | 11.962 | 4.422 | 7.540 | ∞ | input port clock |
| Path 7 | ∞ | 7 | 5 | A[4] | Sum[10] | 11.500 | 4.429 | 7.072 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 4 | A[0] | Sum[8] | 10.961 | 4.169 | 6.792 | ∞ | input port clock |
| Path 9 | ∞ | 7 | 4 | A[0] | Sum[9] | 10.499 | 4.179 | 6.320 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 4 | A[0] | Sum[6] | 9.921 | 4.031 | 5.890 | ∞ | input port clock |

Figure 40: Hybrid_Adr16_Sksa8_Bka8-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 12 | 0 | 0 | 8150 | 0.15 |
| SLICEL | 12 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 36 | 0 | 0 | 32600 | 0.11 |
| using O5 output only | 0 | | | | |
| using O6 output only | 27 | | | | |
| using O5 and O6 | 9 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 41: Hybrid_Adr16_Sksa8_Bka8-Area

### 4.4.4 Hybrid Adder: LING + SKSA with Enable (Proposed)

- **Architecture**: Combines the low-power Ling and fast SKSA.
- **Outcome**: Achieved the lowest power-delay product among all 16-bit hybrids. Selected as the final proposed design for both 16-bit and 32-bit architectures.

**Synthesis Report:**

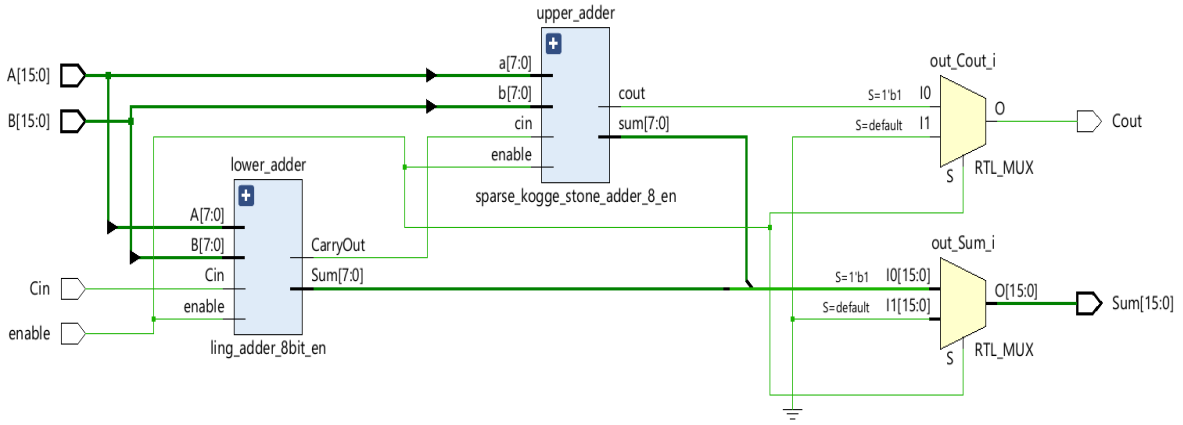



Figure 42: Hybrid_Adr16_Ling8_Sksa8-Scchematic

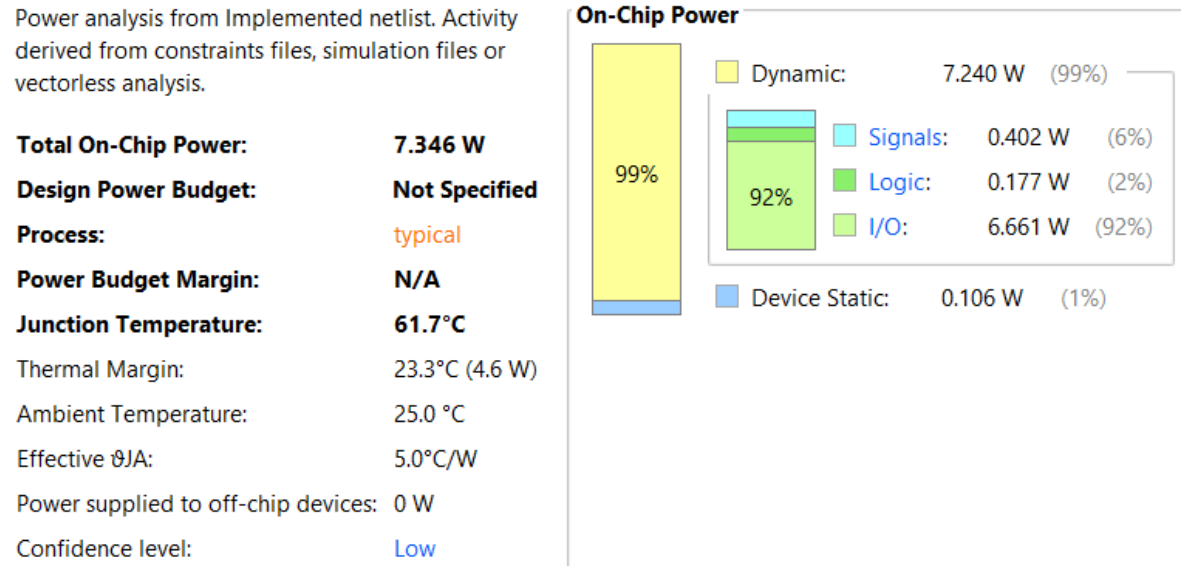



Figure 43: Hybrid_Adr16_Ling8_Sksa8-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 7 | 4 | Cin | Cout | 13.974 | 4.156 | 9.818 | ∞ | input port clock |
| Path 2 | ∞ | 7 | 4 | Cin | Sum[15] | 11.919 | 4.169 | 7.750 | ∞ | input port clock |
| Path 3 | ∞ | 7 | 4 | Cin | Sum[13] | 11.694 | 4.166 | 7.527 | ∞ | input port clock |
| Path 4 | ∞ | 7 | 4 | Cin | Sum[11] | 11.677 | 4.160 | 7.516 | ∞ | input port clock |
| Path 5 | ∞ | 7 | 4 | Cin | Sum[12] | 11.469 | 4.162 | 7.307 | ∞ | input port clock |
| Path 6 | ∞ | 7 | 4 | Cin | Sum[14] | 11.432 | 4.168 | 7.264 | ∞ | input port clock |
| Path 7 | ∞ | 7 | 4 | Cin | Sum[10] | 11.149 | 4.174 | 6.976 | ∞ | input port clock |
| Path 8 | ∞ | 7 | 4 | Cin | Sum[8] | 10.746 | 4.158 | 6.588 | ∞ | input port clock |
| Path 9 | ∞ | 7 | 4 | Cin | Sum[9] | 10.296 | 4.168 | 6.128 | ∞ | input port clock |
| Path 10 | ∞ | 6 | 4 | Cin | Sum[6] | 9.966 | 4.020 | 5.946 | ∞ | input port clock |

Figure 44: Hybrid_Adr16_Ling8_Sksa8-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 10 | 0 | 0 | 8150 | 0.12 |
| SLICEL | 10 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 37 | 0 | 0 | 32600 | 0.11 |
| using O5 output only | 0 | | | | |
| using O6 output only | 29 | | | | |
| using O5 and O6 | 8 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 45: Hybrid_Adr16_Ling8_Sksa8-Area

## 4.5 Hybrid Adders 32-bit with Enable

The enable-controlled hybrid strategy was extended to 32-bit implementations. These were built using either:

- Two 16-bit hybrid blocks, or four 8-bit conventional adder blocks, depending on the configuration.

### 4.5.1 Hybrid Adder: LING + BKA with Enable

- **Structure**: Two 16-bit blocks (Ling16 + BKA16) integrated.

### Synthesis Report:

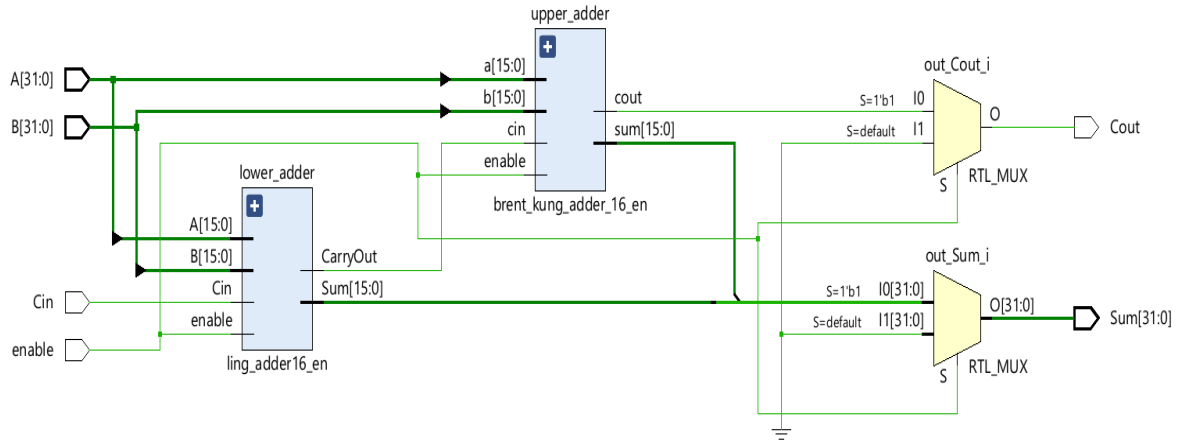



Figure 46: Hybrid_Adr32_Ling16_Bka16-Schematic

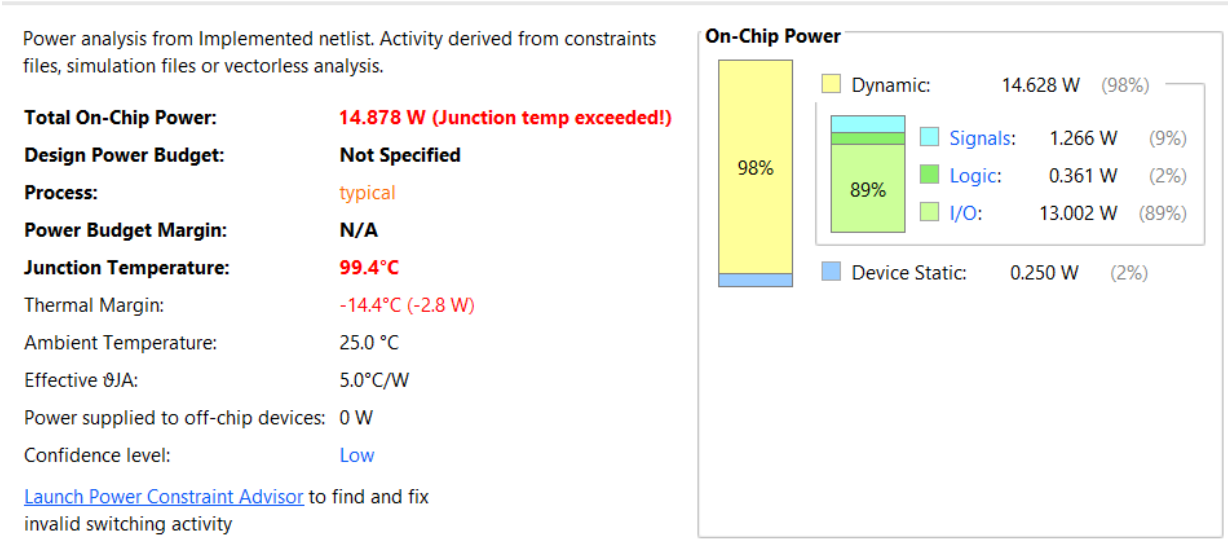



Figure 47: Hybrid_Adr32_Ling16_Bka16-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 13 | 4 | Cin | Sum[31] | 22.121 | 4.912 | 17.209 | ∞ | input port clock |
| Path 2 | ∞ | 13 | 5 | Cin | Sum[29] | 21.841 | 4.910 | 16.931 | ∞ | input port clock |
| Path 3 | ∞ | 13 | 5 | Cin | Cout | 21.779 | 4.906 | 16.872 | ∞ | input port clock |
| Path 4 | ∞ | 13 | 5 | Cin | Sum[30] | 21.574 | 4.894 | 16.680 | ∞ | input port clock |
| Path 5 | ∞ | 13 | 5 | Cin | Sum[28] | 21.369 | 4.894 | 16.476 | ∞ | input port clock |
| Path 6 | ∞ | 13 | 4 | Cin | Sum[22] | 21.335 | 4.894 | 16.441 | ∞ | input port clock |
| Path 7 | ∞ | 13 | 5 | Cin | Sum[26] | 21.154 | 4.915 | 16.239 | ∞ | input port clock |
| Path 8 | ∞ | 12 | 4 | Cin | Sum[21] | 20.758 | 4.770 | 15.988 | ∞ | input port clock |
| Path 9 | ∞ | 13 | 5 | Cin | Sum[27] | 20.588 | 4.895 | 15.693 | ∞ | input port clock |
| Path 10 | ∞ | 12 | 4 | Cin | Sum[20] | 20.549 | 5.007 | 15.541 | ∞ | input port clock |

Figure 48: Hybrid_Adr32_Ling16_Bka16-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 26 | 0 | 0 | 8150 | 0.32 |
| SLICEL | 24 | 0 | | | |
| SLICEM | 2 | 0 | | | |
| LUT as Logic | 72 | 0 | 0 | 32600 | 0.22 |
| using O5 output only | 0 | | | | |
| using O6 output only | 56 | | | | |
| using O5 and O6 | 16 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 49: Hybrid_Adr32_Ling16_Bka16-Area

### 4.5.2 Hybrid Adder: LING + KSA with Enable

- **Structure**: Two 16-bit blocks combining Ling_16 and KSA_16 to form 32-bit hybrid adder.

**Synthesis Report:**

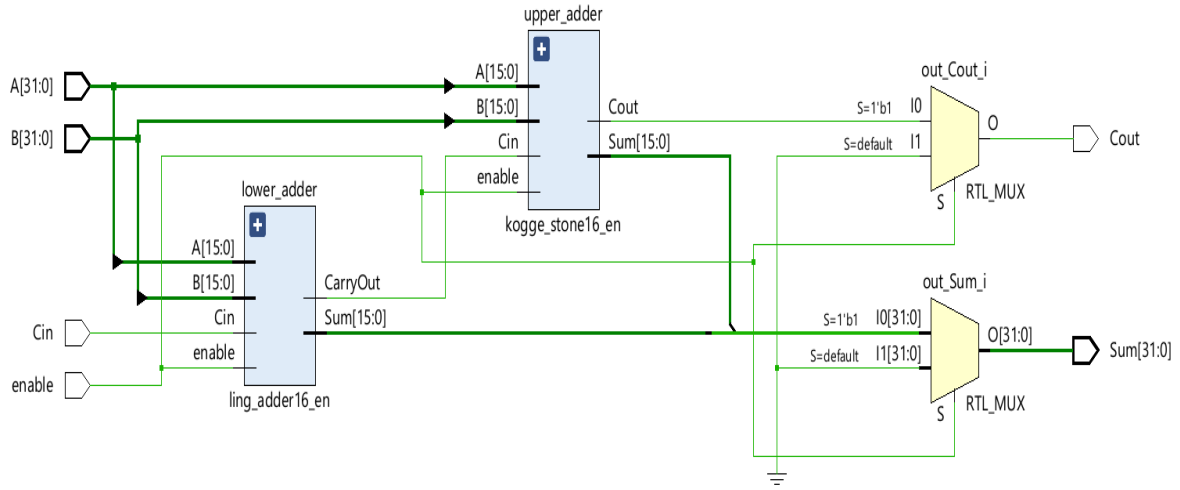



Figure 50: Hybrid_Adr32_Ling16_Ksa16-Schematic

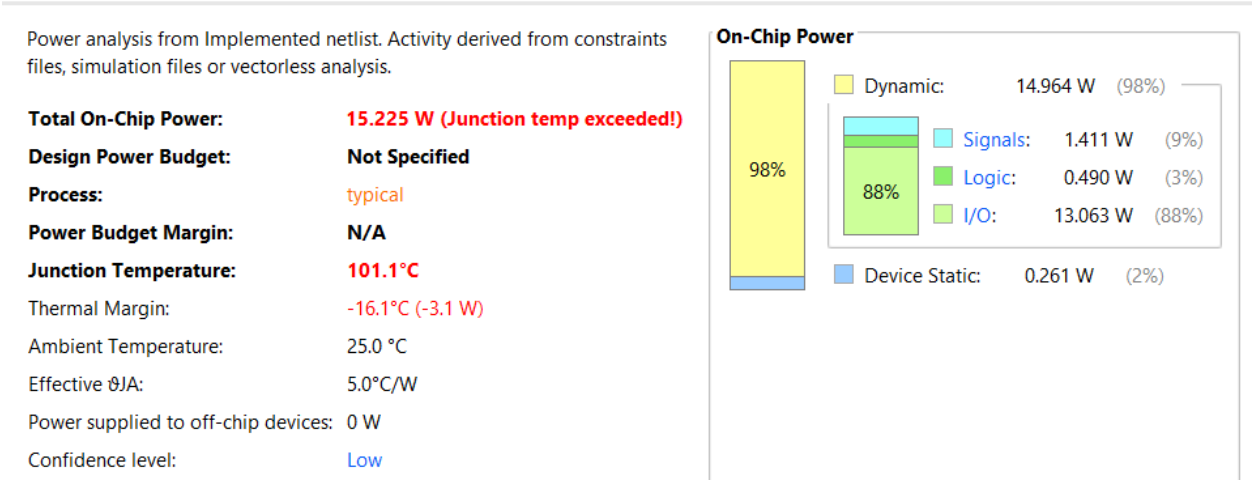



Figure 51: Hybrid_Adr32_Ling16_Ksa16-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 11 | 11 | Cin | Sum[29] | 21.822 | 4.662 | 17.160 | ∞ | input port clock |
| Path 2 | ∞ | 11 | 11 | Cin | Sum[23] | 21.753 | 4.635 | 17.118 | ∞ | input port clock |
| Path 3 | ∞ | 11 | 11 | Cin | Cout | 21.643 | 4.658 | 16.985 | ∞ | input port clock |
| Path 4 | ∞ | 11 | 11 | Cin | Sum[24] | 21.632 | 4.650 | 16.982 | ∞ | input port clock |
| Path 5 | ∞ | 11 | 11 | Cin | Sum[22] | 21.596 | 4.646 | 16.951 | ∞ | input port clock |
| Path 6 | ∞ | 11 | 11 | Cin | Sum[31] | 21.516 | 4.664 | 16.852 | ∞ | input port clock |
| Path 7 | ∞ | 11 | 11 | Cin | Sum[27] | 21.348 | 4.647 | 16.701 | ∞ | input port clock |
| Path 8 | ∞ | 11 | 11 | Cin | Sum[30] | 21.242 | 4.646 | 16.596 | ∞ | input port clock |
| Path 9 | ∞ | 11 | 11 | Cin | Sum[21] | 21.204 | 4.876 | 16.329 | ∞ | input port clock |
| Path 10 | ∞ | 11 | 11 | Cin | Sum[26] | 21.153 | 4.667 | 16.486 | ∞ | input port clock |

Figure 52: Hybrid_Adr32_Ling16_Ksa16-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 30 | 0 | 0 | 8150 | 0.37 |
| SLICEL | 25 | 0 | | | |
| SLICEM | 5 | 0 | | | |
| LUT as Logic | 95 | 0 | 0 | 32600 | 0.29 |
| using O5 output only | 0 | | | | |
| using O6 output only | 71 | | | | |
| using O5 and O6 | 24 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 53: Hybrid_Adr32_Ling16_Ksa16-Area

### 4.5.3 Hybrid Adder: BKA + SKSA with Enable

- **Structure**: using four 8-bit adders:
  - Lower 8 bits (bits [7:0]) using Brent-Kung 8-bit adder with enable.
  - Next 8 bits (bits [15:8]) using Sparse Kogge Stone 8-bit adder with enable.
  - Next 8 bits (bits [23:16]) using another Brent-Kung 8-bit adder with enable.
  - Upper 8 bits (bits [31:24]) using another Sparse Kogge Stone 8-bit adder with enable.

**Synthesis Report:**

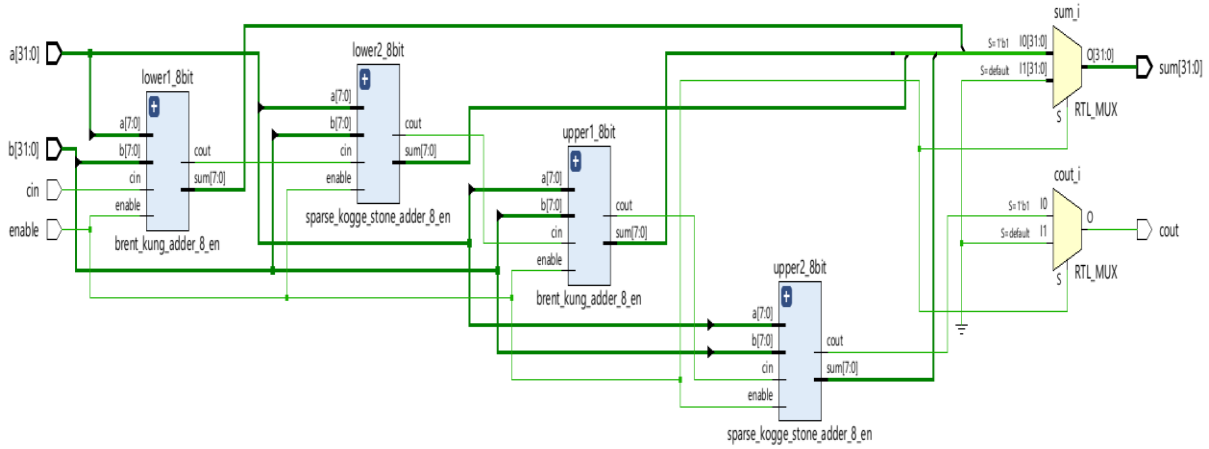



Figure 54: Hybrid_Adr32_(Bka8+Sksa8)*2-Schematic

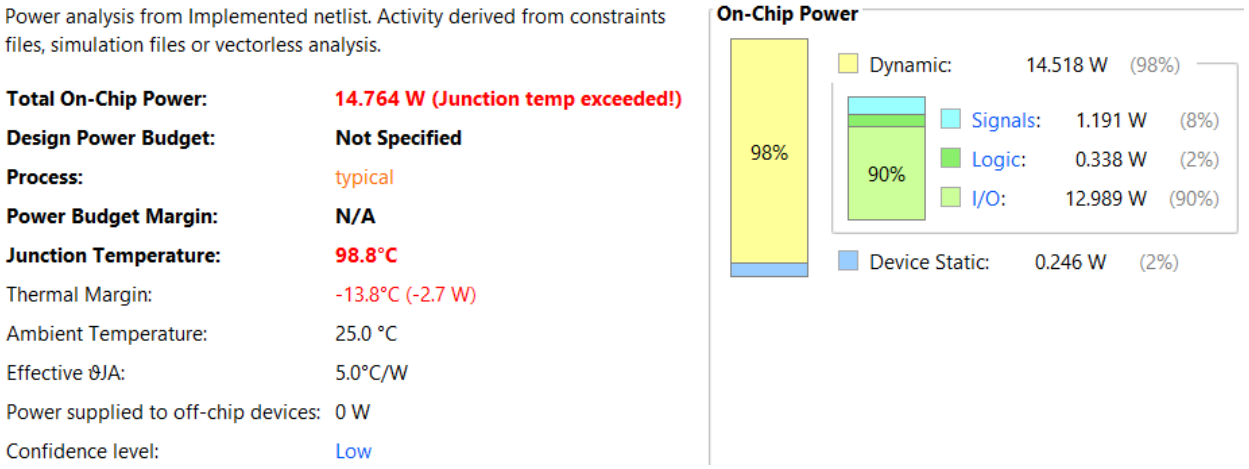



Figure 55: Hybrid_Adr32_(Bka8+Sksa8)*2-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 13 | 39 | enable | cout | 24.430 | 5.150 | 19.281 | ∞ | input port clock |
| Path 2 | ∞ | 13 | 39 | enable | sum[30] | 24.059 | 5.137 | 18.922 | ∞ | input port clock |
| Path 3 | ∞ | 13 | 39 | enable | sum[31] | 23.853 | 5.155 | 18.698 | ∞ | input port clock |
| Path 4 | ∞ | 12 | 4 | cin | sum[22] | 22.750 | 5.000 | 17.750 | ∞ | input port clock |
| Path 5 | ∞ | 12 | 39 | enable | sum[28] | 22.449 | 5.013 | 17.437 | ∞ | input port clock |
| Path 6 | ∞ | 12 | 39 | enable | sum[29] | 22.113 | 5.030 | 17.084 | ∞ | input port clock |
| Path 7 | ∞ | 11 | 4 | cin | sum[21] | 22.064 | 4.876 | 17.189 | ∞ | input port clock |
| Path 8 | ∞ | 11 | 39 | enable | sum[26] | 21.950 | 4.910 | 17.039 | ∞ | input port clock |
| Path 9 | ∞ | 12 | 4 | cin | sum[23] | 21.762 | 4.989 | 16.773 | ∞ | input port clock |
| Path 10 | ∞ | 11 | 4 | cin | sum[20] | 21.132 | 4.879 | 16.253 | ∞ | input port clock |

Figure 56: Hybrid_Adr32_(Bka8+Sksa8)*2-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 26 | 0 | 0 | 8150 | 0.32 |
| SLICEL | 25 | 0 | | | |
| SLICEM | 1 | 0 | | | |
| LUT as Logic | 71 | 0 | 0 | 32600 | 0.22 |
| using O5 output only | 0 | | | | |
| using O6 output only | 60 | | | | |
| using O5 and O6 | 11 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 57: Hybrid_Adr32_(Bka8+Sksa8)*2-Area

### 4.5.4 Hybrid Adder: LING + BKA+ KSA + SKSA with Enable

- **Structure**: 8-bit Ling, followed by two stages of 12-bit and 12-bit KSA blocks.
  - Instantiate four different 8-bit adder blocks (each with an enable input)
  - Block 0: Use Ling 8-bit Adder with Enable. (bits [7:0])
  - Block 1: Use Brent-Kung 8-bit Adder with Enable. (bits [15:8])
  - Block 2: Use Full Kogge-Stone 8-bit Adder with Enable. (bits [23:16])

- Block 3: Use Sparse Kogge-Stone 8-bit Adder with Enable. (bits [31:24])

## Synthesis Report

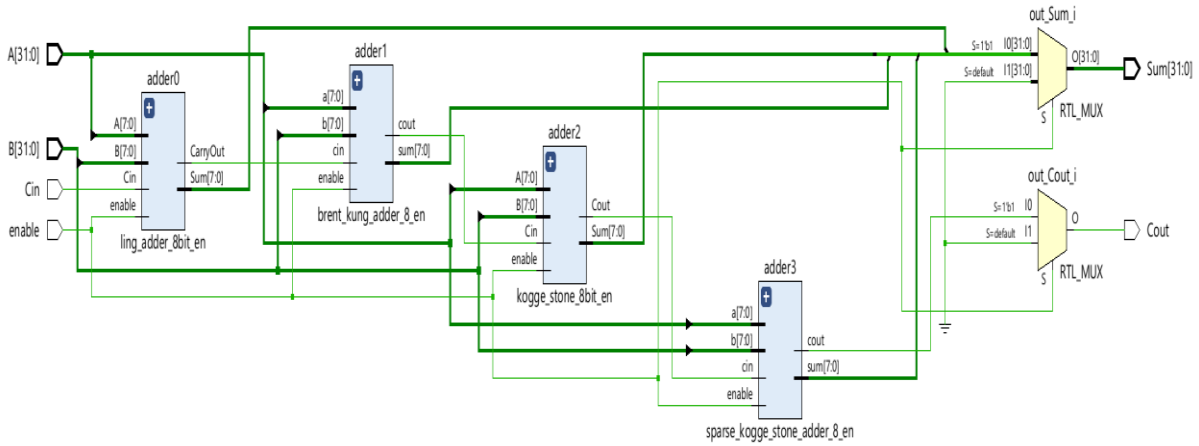



Figure 58: Hybrid_Adr32_Ling8_Bka8_Ksa8_Sksa8-Schematic

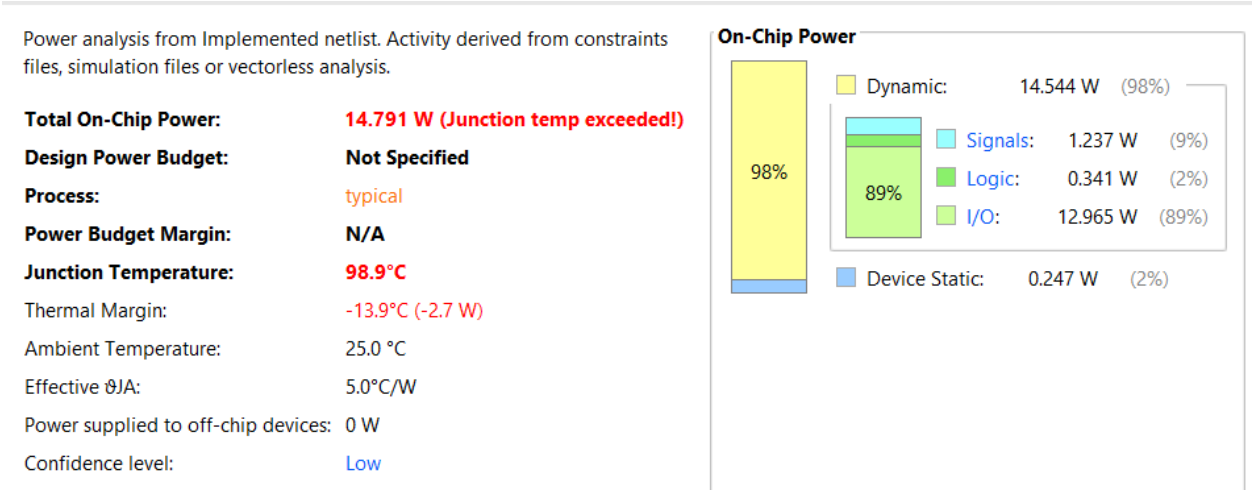



Figure 59: Hybrid_Adr32_Ling8_Bka8_Ksa8_Sksa8-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack ^1 | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 11 | 40 | enable | Sum[23] | 22.412 | 5.104 | 17.308 | ∞ | input port clock |
| Path 2 | ∞ | 11 | 40 | enable | Sum[22] | 22.206 | 5.115 | 17.091 | ∞ | input port clock |
| Path 3 | ∞ | 10 | 40 | enable | Sum[28] | 22.187 | 4.763 | 17.424 | ∞ | input port clock |
| Path 4 | ∞ | 10 | 40 | enable | Sum[29] | 21.899 | 4.780 | 17.120 | ∞ | input port clock |
| Path 5 | ∞ | 11 | 40 | enable | Sum[31] | 21.858 | 5.135 | 16.723 | ∞ | input port clock |
| Path 6 | ∞ | 9 | 40 | enable | Sum[26] | 21.459 | 4.660 | 16.799 | ∞ | input port clock |
| Path 7 | ∞ | 9 | 40 | enable | Sum[27] | 21.223 | 4.640 | 16.582 | ∞ | input port clock |
| Path 8 | ∞ | 10 | 40 | enable | Sum[20] | 21.043 | 4.994 | 16.049 | ∞ | input port clock |
| Path 9 | ∞ | 10 | 40 | enable | Cout | 21.004 | 4.776 | 16.229 | ∞ | input port clock |
| Path 10 | ∞ | 10 | 40 | enable | Sum[30] | 20.853 | 4.993 | 15.860 | ∞ | input port clock |

Figure 60: Hybrid_Adr32_Ling8_Bka8_Ksa8_Sksa8-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 27 | 0 | 0 | 8150 | 0.33 |
| SLICEL | 21 | 0 | | | |
| SLICEM | 6 | 0 | | | |
| LUT as Logic | 70 | 0 | 0 | 32600 | 0.21 |
| using O5 output only | 0 | | | | |
| using O6 output only | 55 | | | | |
| using O5 and O6 | 15 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 61: Hybrid_Adr32_Ling8_Bka8_Ksa8_Sksa8-Area

### 4.5.5 Hybrid Adder: LING + SKSA with Enable (Proposed)

- **Architecture**: Two 16-bit blocks of Ling and SKSA with enable.

  Split the 32-bit operands into lower and upper 16-bit halves.

  Lower 16-bit adder using a Ling architecture – bits [15:0].

  Upper 16-bit adder using a Sparse Kogge Stone Adder - bit [31:16].
- **Observation**: It outperformed all other configurations in power-delay-product (PDP).

**Synthesis Report:**

**Proposed 32-Bit Hybrid Adder**

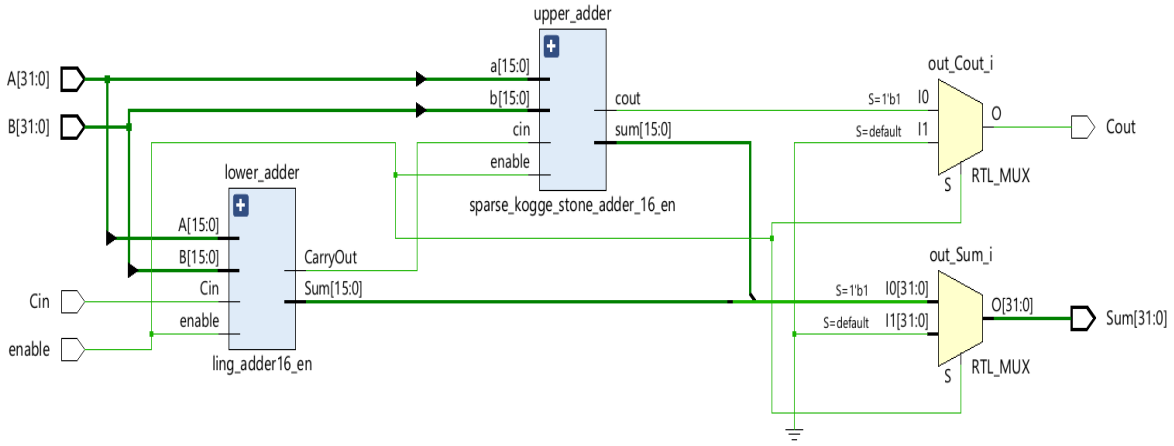



Figure 62: Hybrid_Adr32_Ling16_Sksa16-Schematic

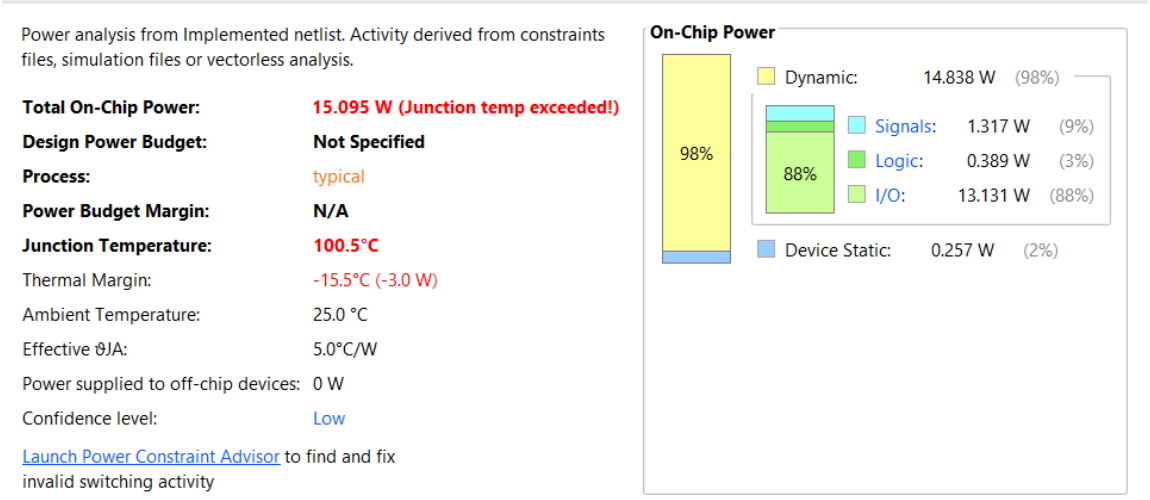



Figure 63: Hybrid_Adr32_Ling16_Sksa16-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 11 | 4 | Cin | Cout | 20.807 | 4.658 | 16.148 | ∞ | input port clock |
| Path 2 | ∞ | 11 | 5 | Cin | Sum[28] | 20.683 | 4.646 | 16.038 | ∞ | input port clock |
| Path 3 | ∞ | 11 | 5 | Cin | Sum[30] | 20.188 | 4.646 | 15.542 | ∞ | input port clock |
| Path 4 | ∞ | 11 | 5 | Cin | Sum[22] | 20.126 | 4.646 | 15.480 | ∞ | input port clock |
| Path 5 | ∞ | 11 | 5 | Cin | Sum[26] | 20.122 | 4.667 | 15.455 | ∞ | input port clock |
| Path 6 | ∞ | 11 | 5 | Cin | Sum[25] | 20.074 | 4.654 | 15.420 | ∞ | input port clock |
| Path 7 | ∞ | 11 | 5 | Cin | Sum[31] | 19.960 | 4.664 | 15.297 | ∞ | input port clock |
| Path 8 | ∞ | 10 | 5 | Cin | Sum[21] | 19.858 | 4.522 | 15.337 | ∞ | input port clock |
| Path 9 | ∞ | 11 | 5 | Cin | Sum[23] | 19.844 | 4.635 | 15.210 | ∞ | input port clock |
| Path 10 | ∞ | 11 | 5 | Cin | Sum[24] | 19.730 | 4.650 | 15.079 | ∞ | input port clock |

Figure 64: Hybrid_Adr32_Ling16_Sksa16-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 31 | 0 | 0 | 8150 | 0.38 |
| SLICEL | 31 | 0 | | | |
| SLICEM | 0 | 0 | | | |
| LUT as Logic | 80 | 0 | 0 | 32600 | 0.25 |
| using O5 output only | 0 | | | | |
| using O6 output only | 63 | | | | |
| using O5 and O6 | 17 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 65: Hybrid_Adr32_Ling16_Sksa16-Area

## 4.6 Summary

A comprehensive comparison of all 16-bit and 32-bit hybrid adder architectures with enable was performed using Xilinx Vivado. Hybrid adder is proposed based on the trade off values of the metrics evaluated include:

Total Power, Delay (ns), Power-Delay Product, Slice Logic Used, LUT Utilization.

Table 4.1: 16-bit Hybrid Adders Results

| Adder Name | Total Power (W) | Delay (ns) | Power delay product (W.ns) | Slice | LUT as logic |
|---|---|---|---|---|---|
| LING+BKA | 7.236 | 14.120 | 102.172 | 9 | 30 |
| LING+KSA | 7.323 | 13.639 | 99.878 | 12 | 38 |
| SKSA+BKA | 7.350 | 14.208 | 104.428 | 12 | 36 |
| LING+SKSA (Proposed 16-Bit Hybrid Adder) | 7.346 | 13.974 | 102.653 | 9 | 30 |

Table 4.2: 32-bit Hybrid Adders Results

| Adder Name | Total Power (W) | Delay (ns) | Power delay product (W.ns) | Slice | LUT logic |
|---|---|---|---|---|---|
| LING+BKA | 14.878 | 22.121 | 329.116 | 26 | 72 |
| LING+KSA | 15.225 | 21.822 | 332.239 | 30 | 95 |
| BKA8+SKSA8+ BKA8+SKSA8 | 14.764 | 24.430 | 360.684 | 26 | 71 |
| LING8+BKA8+ KSA8+SKSA8 | 14.791 | 22.412 | 331.495 | 21 | 70 |
| LING+SKSA (Proposed 32-Bit Hybrid Adder) | 15.095 | 20.807 | 314.081 | 31 | 80 |

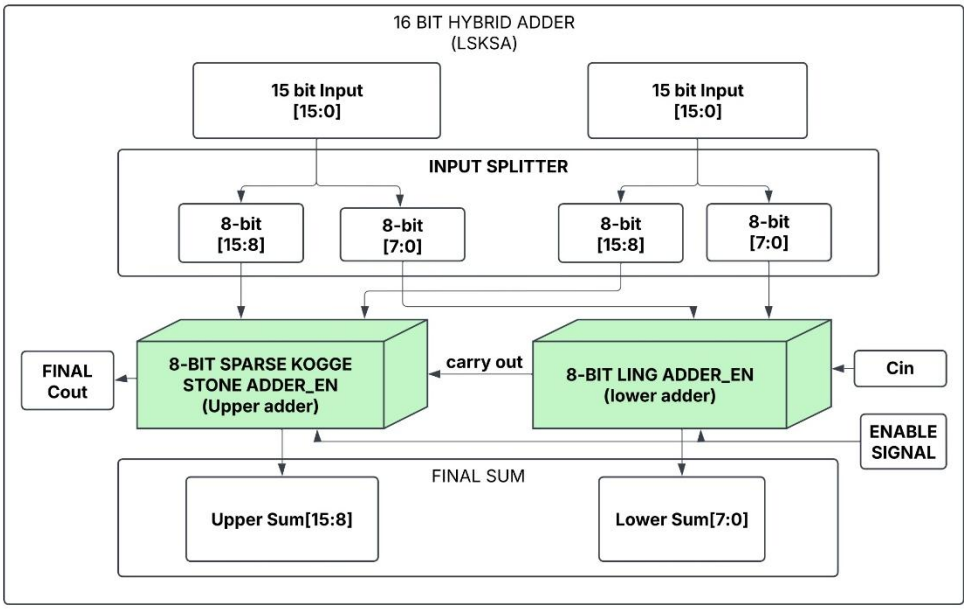



Figure 66: Block Diagram of Proposed 16-bit Hybrid Adder

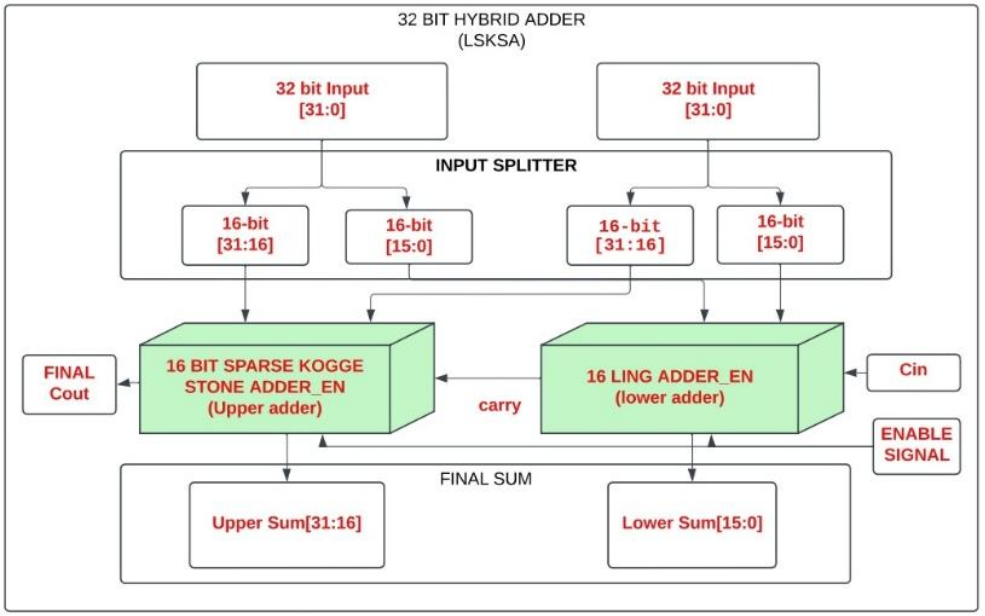



Figure 67: Block Diagram of Proposed 32-bit Hybrid Adder

# Chapter 5

# Multipliers Overview

## 5.1 Introduction

In modern digital signal processing and embedded systems, multipliers play a pivotal role, particularly in computationally intensive tasks such as convolution, filtering, and matrix multiplication. Performance metrics like power consumption, delay, and silicon area are critical in evaluating the efficiency of multipliers. This chapter focuses on conventional multiplier architectures, providing a comparative analysis to establish a baseline for performance evaluation in subsequent hybrid implementations. Today, multipliers are extensively employed in areas such as digital signal analysis. In order to make multipliers suitable for various high speed, low power, and compact VLSI implementations, many researchers have tried and are still trying to creation of a multiplier which satisfy one of the two design targets listed as: excessive pace, minimal power use, consistency for the design and thus fewer space, or perhaps all of these goals within a single a multiplier.

## 5.2 Conventional Multiplier Architectures

Conventional multiplier designs are widely studied and form the foundation for multiplication operations in digital systems. Each architecture is characterized by unique design principles and trade-offs. Convention Multipliers Implemented for reference are:

### 5.2.1 Dadda Multiplier

This design employs a column compression technique that reduces the number of partial product rows efficiently before the final addition. It is recognized for high-speed performance but tends to consume more power due to increased circuit complexity. The Dadda multiplier is an optimized version of the Wallace tree multiplier, designed to reduce the number of required carry-save adders while minimizing propagation delay. The key innovation in the Dadda multiplier is its strategic approach to compressing partial products before executing the final addition. It achieves this by following a three-step process:

1. Generation of partial products: Similar to the array multiplier, the Dadda multiplier uses AND gates to create the partial products.
2. Reduction of partial products: Unlike the Wallace Tree approach, Dadda multipliers use a specific column-wise reduction strategy, where the number of bits per column is gradually reduced to achieve minimal delay.
3. Final addition: The reduced partial products are summed using a conventional adder such as a carry-lookahead adder (CLA) or ripple carry adder (RCA).

The Dadda multiplier achieves faster computation compared to the array multiplier due to its efficient reduction technique, though it comes at the cost of increased circuit complexity.

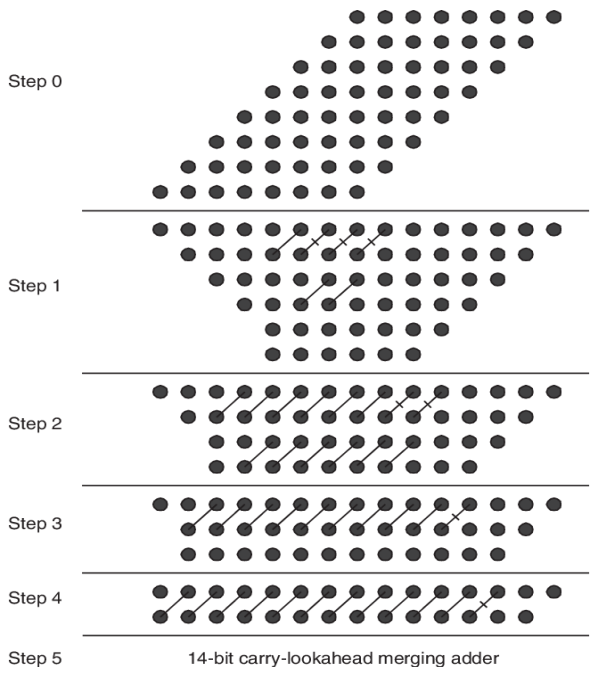



Figure 68: Conventional Dadda Multiplier 8-bit

## Synthesis Report:

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **37.424 W (Junction temp exceeded!)** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **125.0°C** |
| Thermal Margin: | -127.1°C (-25.1 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

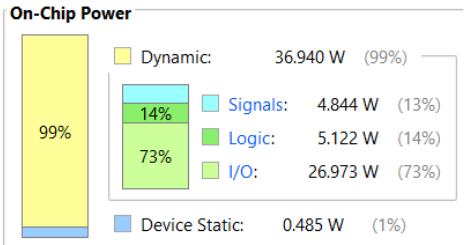



Figure 69: 16-Bit Dadda Multiplier-Power

**Unconstrained Paths - NONE - NONE - Setup**

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 17 | 45 | B[2] | Y[29] | 25.773 | 7.011 | 18.762 | ∞ | input port clock |
| Path 2 | ∞ | 17 | 45 | B[2] | Y[30] | 25.382 | 6.783 | 18.599 | ∞ | input port clock |
| Path 3 | ∞ | 17 | 45 | B[2] | Y[28] | 25.291 | 6.795 | 18.497 | ∞ | input port clock |
| Path 4 | ∞ | 17 | 45 | B[2] | Y[31] | 25.188 | 6.788 | 18.400 | ∞ | input port clock |
| Path 5 | ∞ | 16 | 45 | B[2] | Y[25] | 24.688 | 6.887 | 17.800 | ∞ | input port clock |
| Path 6 | ∞ | 16 | 45 | B[2] | Y[27] | 24.472 | 6.665 | 17.806 | ∞ | input port clock |
| Path 7 | ∞ | 16 | 45 | B[2] | Y[26] | 24.452 | 6.655 | 17.798 | ∞ | input port clock |
| Path 8 | ∞ | 16 | 45 | B[2] | Y[24] | 24.090 | 6.641 | 17.449 | ∞ | input port clock |
| Path 9 | ∞ | 15 | 45 | B[2] | Y[23] | 23.108 | 6.313 | 16.795 | ∞ | input port clock |
| Path 10 | ∞ | 15 | 45 | B[2] | Y[22] | 23.069 | 6.293 | 16.775 | ∞ | input port clock |

Figure 70: 16-Bit Dadda Multiplier-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 102 | 0 | 0 | 8150 | 1.25 |
| SLICEL | 71 | 0 | | | |
| SLICEM | 31 | 0 | | | |
| LUT as Logic | 350 | 0 | 0 | 32600 | 1.07 |
| using O5 output only | 0 | | | | |
| using O6 output only | 270 | | | | |
| using O5 and O6 | 80 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 71: 16-Bit Dadda Multiplier-Area

### 5.2.2 Karatsuba Multiplier:

Based on the divide-and-conquer principle, this architecture recursively splits input operands to reduce the number of elementary multiplications. It is efficient in reducing area and exhibits moderate speed, though it becomes computationally intensive at higher bit widths. The Karatsuba algorithm is a recursive multiplication technique that reduces the number of required multiplications by breaking operands into smaller components. It is particularly efficient for large-number multiplications, where direct computation would be expensive. The recursive nature of this method allows faster computation, especially for high-bit-width operations. However, it involves additional overhead in managing the recursion, making it less efficient for small-bit-width multiplications.

### Synthesis Report:

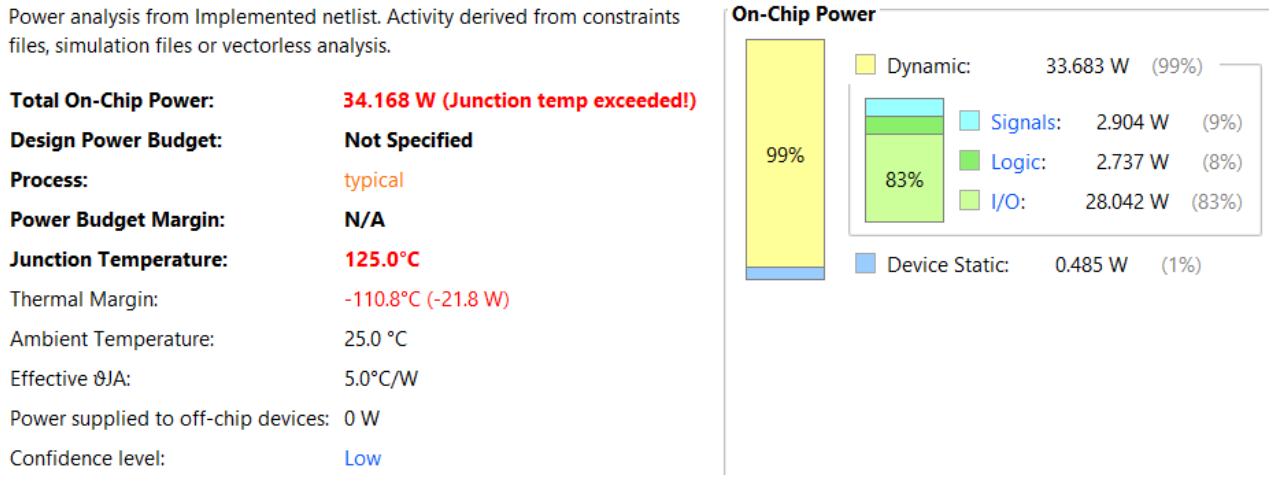



Figure 72: 16-Bit Karatsuba Multiplier-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 20 | 56 | a[7] | out[31] | 21.154 | 9.376 | 11.778 | ∞ | input port clock |
| Path 2 | ∞ | 19 | 56 | a[7] | out[30] | 21.104 | 9.355 | 11.749 | ∞ | input port clock |
| Path 3 | ∞ | 19 | 56 | a[7] | out[29] | 21.012 | 9.276 | 11.736 | ∞ | input port clock |
| Path 4 | ∞ | 19 | 56 | a[7] | out[28] | 20.980 | 9.385 | 11.595 | ∞ | input port clock |
| Path 5 | ∞ | 19 | 56 | a[7] | out[27] | 20.867 | 9.263 | 11.603 | ∞ | input port clock |
| Path 6 | ∞ | 17 | 56 | a[7] | out[26] | 20.794 | 8.895 | 11.899 | ∞ | input port clock |
| Path 7 | ∞ | 17 | 56 | a[7] | out[24] | 20.745 | 8.899 | 11.846 | ∞ | input port clock |
| Path 8 | ∞ | 17 | 56 | a[7] | out[25] | 20.718 | 8.821 | 11.897 | ∞ | input port clock |
| Path 9 | ∞ | 17 | 56 | a[7] | out[23] | 20.689 | 8.807 | 11.882 | ∞ | input port clock |
| Path 10 | ∞ | 16 | 56 | a[7] | out[22] | 20.277 | 8.420 | 11.858 | ∞ | input port clock |

Figure 73: 16-Bit Karatsuba Multiplier-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 149 | 0 | 0 | 8150 | 1.83 |
| SLICEL | 102 | 0 | | | |
| SLICEM | 47 | 0 | | | |
| LUT as Logic | 465 | 0 | 0 | 32600 | 1.43 |
| using O5 output only | 0 | | | | |
| using O6 output only | 439 | | | | |
| using O5 and O6 | 26 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 74: 16-Bit Karatsuba Multiplier-Area

### 5.2.3 Wallace Tree Multiplier:

This multiplier uses a fast reduction tree to compress partial products quickly. It is one of the fastest conventional designs but involves complex wiring and control logic, increasing design overhead. A Wallace tree is a very effective hardware method for implementing an electronic circuit that multiplies two different numbers. This was invented in 1964 by Chris Wallace, an Australian computer scientist**.**

The Wallace tree has following stages:

1. To get n 2 outcomes, divide each bit of one argument by each bit of the other. The wires' weights vary according on where the multiplied bits are located.
2. Layers of half and full adders limit the number of partial products to two.
3. With a standard adder, divide the wires into two integers.

The second step is carried out as shown below. Add the next layer if there are three or more wires of the same weight.

• Any four identically weighed wires can be combined into a full adder. The outcome will be a heavier output wire and an output wire with the identical weight as each of the four input wires.

• If there are still two identical wires, feed them into a half-adder.

•Connect the final wire to the following layer if only one is left.

The Wallace tree benefits from having only reduction layers with an O (1) propagation delay.

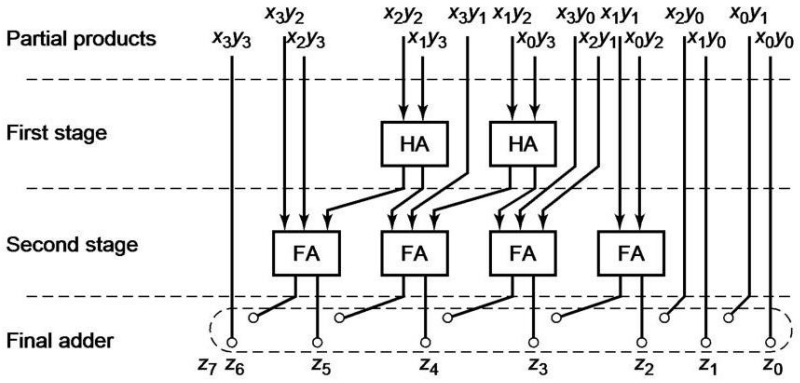



Figure 75: Wallace Tree Multiplier

## Synthesis Report:

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **39.189 W (Junction temp exceeded!)** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **125.0°C** |
| Thermal Margin: | -135.9°C (-26.8 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

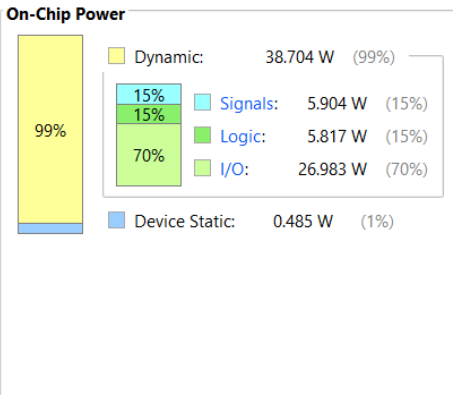



Figure 76: 16-Bit Wallace Tree Multiplier-Power

**Unconstrained Paths - NONE - NONE - Setup**

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 16 | 33 | b[8] | p[27] | 23.546 | 5.993 | 17.553 | ∞ | input port clock |
| Path 2 | ∞ | 17 | 33 | b[8] | p[30] | 23.502 | 5.881 | 17.621 | ∞ | input port clock |
| Path 3 | ∞ | 17 | 33 | b[8] | p[31] | 23.274 | 5.886 | 17.388 | ∞ | input port clock |
| Path 4 | ∞ | 15 | 33 | b[8] | p[25] | 22.510 | 5.863 | 16.647 | ∞ | input port clock |
| Path 5 | ∞ | 15 | 33 | b[8] | p[26] | 22.447 | 5.859 | 16.588 | ∞ | input port clock |
| Path 6 | ∞ | 16 | 33 | b[8] | p[29] | 22.266 | 5.755 | 16.511 | ∞ | input port clock |
| Path 7 | ∞ | 16 | 33 | b[8] | p[28] | 22.260 | 5.769 | 16.491 | ∞ | input port clock |
| Path 8 | ∞ | 14 | 33 | b[8] | p[24] | 21.406 | 5.491 | 15.915 | ∞ | input port clock |
| Path 9 | ∞ | 14 | 33 | b[8] | p[23] | 21.127 | 5.515 | 15.612 | ∞ | input port clock |
| Path 10 | ∞ | 13 | 33 | b[8] | p[22] | 19.932 | 5.143 | 14.789 | ∞ | input port clock |

Figure 77: 16-Bit Wallace Tree Multiplier-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 110 | 0 | 0 | 8150 | 1.35 |
| SLICEL | 69 | 0 | | | |
| SLICEM | 41 | 0 | | | |
| LUT as Logic | 387 | 0 | 0 | 32600 | 1.19 |
| using O5 output only | 0 | | | | |
| using O6 output only | 307 | | | | |
| using O5 and O6 | 80 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 78: 16-Bit Wallace Tree Multiplier-Area

### 5.2.4 Vedic Multiplier

Inspired by ancient Indian mathematics, particularly the Urdhva Tiryakbhyam (vertically and crosswise) algorithm, this design reduces the number of computational steps required for multiplication. It is compact and highly efficient in both speed and power, making it suitable for low-power applications.

The core mathematical principle behind this approach is:

$$P = (A\text{high} \times B\text{high}) \times 2^{2m} + ((A\text{high} \times B\text{low}) + (A\text{low} \times B\text{high})) \times 2^{m} + (A\text{low} \times B\text{low})$$

Where Ahigh and Alow represent the higher and lower bits of the operand A, respectively, and the same applies to operand B. This method ensures low power consumption and fast computation, making it ideal for applications requiring rapid arithmetic operations. However, scalability issues arise for large-bit-width multiplications, limiting its efficiency in certain cases.

## Synthesis Report:

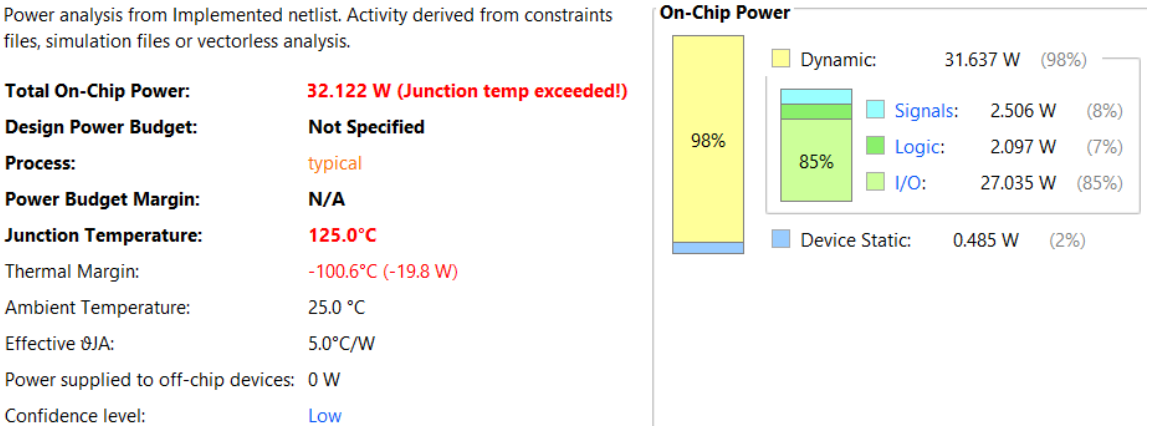



Figure 79: 16-Bit Vedic Multiplier-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 18 | 28 | b[0] | c[31] | 19.527 | 8.300 | 11.227 | ∞ | input port clock |
| Path 2 | ∞ | 18 | 28 | b[0] | c[29] | 19.500 | 8.312 | 11.188 | ∞ | input port clock |
| Path 3 | ∞ | 18 | 28 | b[0] | c[30] | 19.294 | 8.217 | 11.077 | ∞ | input port clock |
| Path 4 | ∞ | 18 | 28 | b[0] | c[28] | 19.258 | 8.209 | 11.049 | ∞ | input port clock |
| Path 5 | ∞ | 17 | 28 | b[0] | c[25] | 19.252 | 8.199 | 11.052 | ∞ | input port clock |
| Path 6 | ∞ | 17 | 28 | b[0] | c[27] | 19.243 | 8.187 | 11.055 | ∞ | input port clock |
| Path 7 | ∞ | 17 | 28 | b[0] | c[26] | 19.139 | 8.099 | 11.040 | ∞ | input port clock |
| Path 8 | ∞ | 17 | 28 | b[0] | c[24] | 19.127 | 8.065 | 11.061 | ∞ | input port clock |
| Path 9 | ∞ | 16 | 28 | b[0] | c[23] | 19.123 | 8.070 | 11.053 | ∞ | input port clock |
| Path 10 | ∞ | 16 | 28 | b[0] | c[21] | 19.098 | 8.083 | 11.015 | ∞ | input port clock |

Figure 80: 16-Bit Vedic Multiplier-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 109 | 0 | 0 | 8150 | 1.34 |
| SLICEL | 70 | 0 | | | |
| SLICEM | 39 | 0 | | | |
| LUT as Logic | 334 | 0 | 0 | 32600 | 1.02 |
| using O5 output only | 0 | | | | |
| using O6 output only | 309 | | | | |
| using O5 and O6 | 25 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 81: 16-Bit Vedic Multiplier-Area

## 5.3 Synthesis and Implementation Report

Table 5.1: 16-bit Conventional Multiples Results

| Design | Total Power (W) | Delay (ns) | PDP (W·ns) | Slice Logic | LUT Logic |
|---|---|---|---|---|---|
| Dadda | 37.424 | 25.773 | 964.528 | 102 | 350 |
| Karatsuba | 34.168 | 21.154 | 722.789 | 149 | 465 |
| Wallace | 39.189 | 23.546 | 922.744 | 110 | 387 |
| Vedic | 32.122 | 19.527 | 627.246 | 109 | 334 |

# Chapter 6

# Hybrid Multiplier Architectures

## 6.1 Introduction

Hybrid multiplier architectures aim to enhance speed, power efficiency, and area optimization by combining multiple adder types to improve partial product accumulation. Traditional multiplier architectures often face a trade-off between delay and power, particularly in digital signal processing (DSP) systems. The designs proposed in this work utilize advanced hybrid adders composed of optimized LING16 and SKSA16 structures to strike a balance between fast carry propagation and energy efficiency. Earlier designs used a combination of LING8 + SKSA8 + LING8 + SKSA8, denoted as the "all 4 adders" configuration, which was later improved into the final LSKSA hybrid adder.

The Partial Product Adder Multiplier is a specialized multiplication method that efficiently computes the product of two binary numbers by leveraging the concept of partial products. This multiplier breaks down the multiplication process into simpler, manageable parts by generating partial products for each bit of the multiplier and then summing these products to obtain the final result. In this approach, the Partial Product Adder Multiplier uses combinational logic to create a series of partial products based on the bits of the multiplicand and multiplier. Each bit of the multiplier triggers an AND operation with the multiplicand, producing a set of partial products. These partial products are then added together using an adder tree or other summation techniques to yield the final product. This method enhances computational efficiency by organizing the multiplication into a structured process that simplifies hardware implementation. The Partial Product Adder Multiplier is particularly effective in digital circuit design, where it serves as a foundational component in applications such as digital signal processing, embedded systems, and high-performance computing environments. By optimizing the handling of binary multiplications, the Partial Product Adder Multiplier provides a reliable solution that balances speed, efficiency, and accuracy, making it a vital tool in modern digital arithmetic.

## 6.2 Hybrid Adder Composition

The final proposed hybrid adder architecture is a 32-bit unit composed of:

- LING16 block: High-speed carry generation using prefix computation.
- SKSA16 block: A Sparse Kogge-Stone Adder optimized for reduced logic and energy consumption.

Initial versions used a modular sequence of:

- LING8 + SKSA8 + LING8 + SKSA8, forming four internal 8-bit stages, collectively referred to as "all 4 adders."

These modular designs were integrated across various multiplier architectures to explore different partial product reduction techniques.

## 6.3 Partial Product Generation

For all hybrid multiplier variants, 16 partial products are generated from 16-bit multiplicand and multiplier inputs using bitwise AND operations.

- These partial products are generated using AND gates between bits of inputs A and B.

  PP0 = A & {16{B[0]}}, PP1 = (A & {16{B[1]}}) << 1, ..... PP15 = (A & {16{B[15]}}) << 15.

  For example, let's multiply A (Multiplicand) = 1011 (11 in decimal) and B (Multiplier) = 1101 (13 in decimal):

```
        1011        (Multiplicand – A)
      × 1101        (Multiplier – B)
-------------------------------------------------------------------------
        1011        (B [0] × A)
    +   00000       (B [1] × A, shifted 1 place)
    +  101100       (B [2] × A, shifted 2 places)
    + 1011000       (B [3] × A, shifted 3 places)
 ------------------------------------------------------------------------
    10001111        (Final Product - 143 in decimal)
```

- The result is a set of 16 aligned partial binary values each of 32-bit size that need to be summed to produce the final product.

These are then summed using 32-bit hybrid adders based on the selected architectural scheme.

The following techniques detail how these partial products are added:

## 6.4 Architectural Scheme for Addition of Partial Products:

### 6.4.1 Parallel Addition with 32-bit Hybrid Adders (All 4 Adders)

- **Architecture**: Multi-stage, highly parallel addition using 32-bit hybrid adders composed of LING8 + SKSA8 + LING8 + SKSA8.
- **Stages**:
    - **Stage 1**: 7 hybrid adders process 14 partial products in pairs.
    - **Stage 2**: 4 adders combine the results of stage 1.
    - **Stage 3**: 2 adders further reduce the intermediate results.
    - **Stage 4**: Final summation using one hybrid adder.
- **Characteristics**: High speed due to parallelism but with increased area and switching activity.

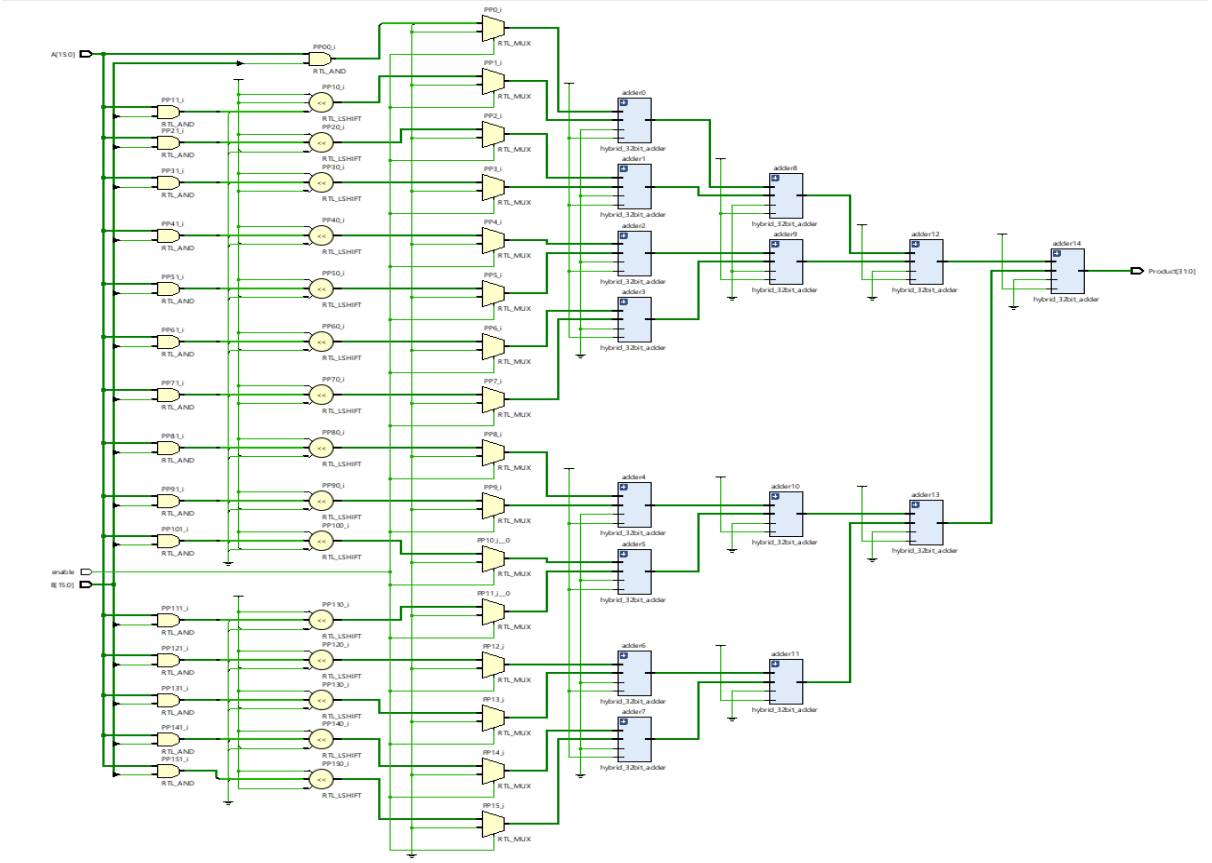


Figure 82: Schematic Of 16-Bit Hybrid Multiplier with Parallel Addition Architecture

### 6.4.2 Serial Addition with 32-bit Hybrid Adders (All 4 Adders)

- **Architecture**: Ripple-based, sequential addition using 15 hybrid adders.
- **Process**:
    - Adders operate in series with carry ripple.
    - Each output becomes the input to the next.
- **Advantages**: Low hardware usage and power.
- **Disadvantages**: Increased delay due to dependency.

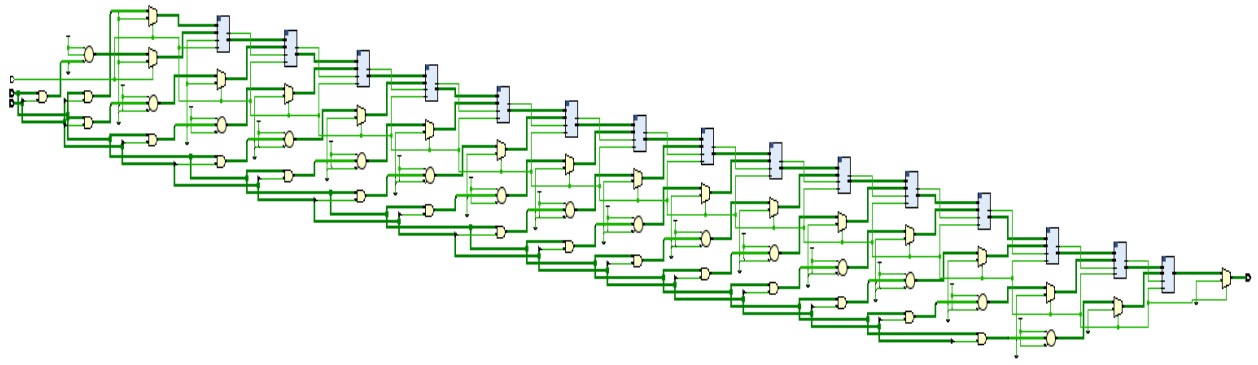


Figure 83: Schematic Of 16-Bit Hybrid Multiplier with Serial Ripple Addition Architecture

### 6.4.3 Two-Chain Tree Addition with 32-bit Hybrid Adders (All 4 Adders)

- **Architecture**: Partial products are split into two independent serial chains.
- **Process**:
    - **Chain 1**: Adds PP0 to PP7.
    - **Chain 2**: Adds PP8 to PP15.
    - The results are summed at stage 2.
- **Balance**: Offers a compromise between delay and hardware complexity.

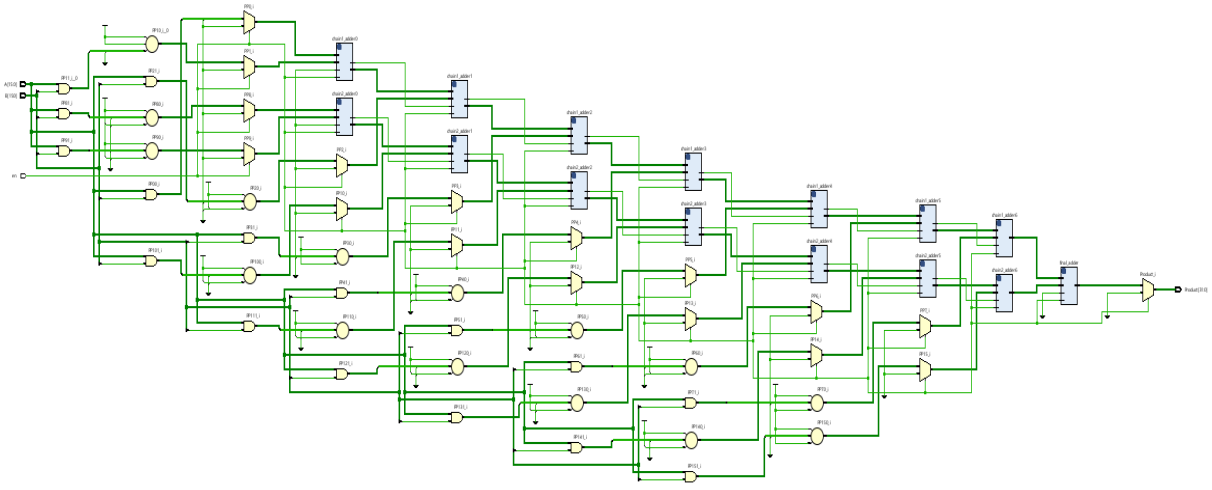


Figure 84: Schematic Of 16-Bit Hybrid Multiplier with 2-Chain Addition Architecture

### 6.4.4 Four-Chain Tree Addition with 32-bit Hybrid Adders (All 4 Adders)

- **Architecture**: Partial products are split into four parallel serial chains.
- **Stage 1**:
    - **Chain 1**: PP0 to PP3
    - **Chain 2**: PP4 to PP7
    - **Chain 3**: PP8 to PP11
    - **Chain 4**: PP12 to PP15
- **Stage 2**: Results combined in pairs using two adders.
- **Stage 3**: Final addition.
- **Benefit**: Reduced delay and balanced logic.

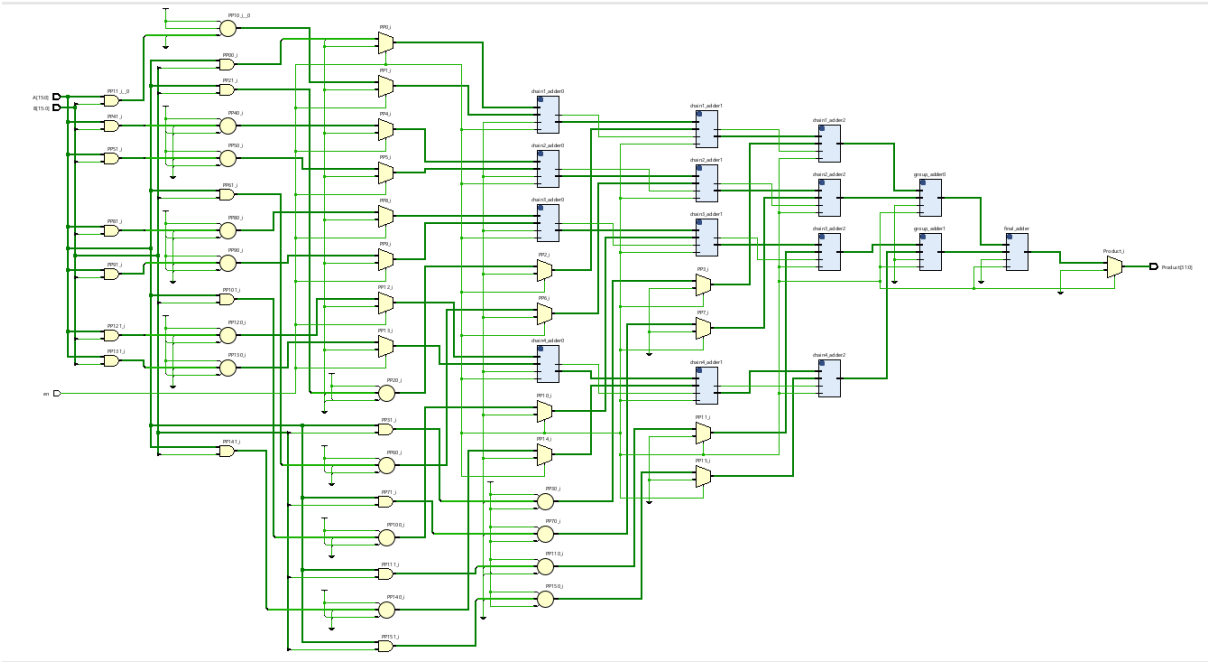


Figure 85: Schematic Of 16-Bit Hybrid Multiplier with 4-Chain Addition Architecture

### 6.4.5 Proposed Design: FOUR-Chain Tree Addition with Proposed Hybrid Adder (Hyd_4chain_LSKSA)

- **Final Architecture**: 4-chain tree structure with the final 32-bit hybrid adder composed of LING16 + SKSA16.

- **Execution**:
    - Four chains add partial products serially.
    - Intermediate results are combined in two stages.
- **Stage 1**:
    - **Chain 1**: PP0 to PP3
    - **Chain 2**: PP4 to PP7
    - **Chain 3**: PP8 to PP11
    - **Chain 4**: PP12 to PP15
- **Stage 2**: Results combined in pairs using two adders.
- **Stage 3**: Final addition.

Adder used for the addition of partial products at each stage is the proposed 32-bit hybrid with LING16+SKSA16

- **Advantages**:
    - Best PDP among all variants.
    - Optimised for low power and delay.
    - Suitable for real-time DSP applications like FIR filters.

**Synthesis Reports: Proposed 16-Bit Hybrid Multiplier**

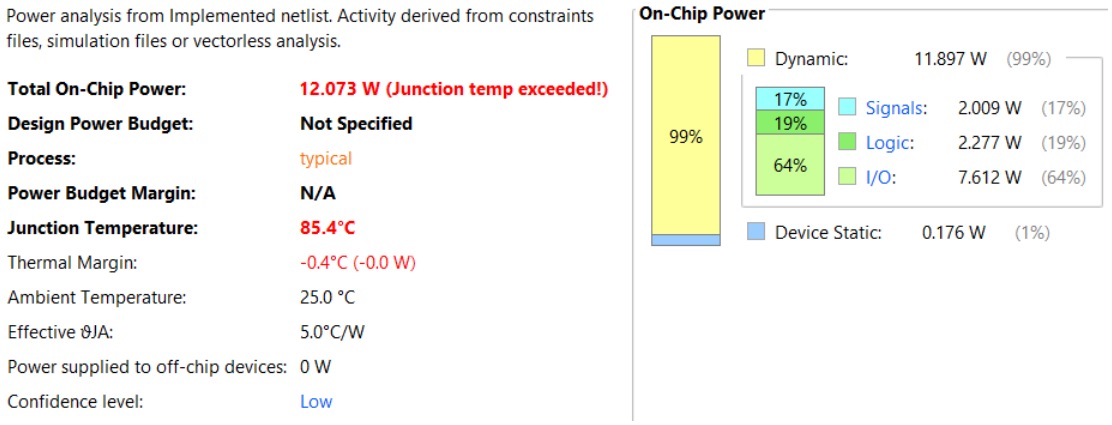



Figure 86: Proposed 16-Bit Multiplier Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 19 | 615 | en | Product[29] | 30.375 | 6.587 | 23.789 | ∞ | input port clock |
| Path 2 | ∞ | 19 | 615 | en | Product[30] | 30.163 | 6.361 | 23.802 | ∞ | input port clock |
| Path 3 | ∞ | 19 | 615 | en | Product[28] | 30.101 | 6.355 | 23.746 | ∞ | input port clock |
| Path 4 | ∞ | 19 | 615 | en | Product[27] | 29.886 | 6.598 | 23.288 | ∞ | input port clock |
| Path 5 | ∞ | 18 | 615 | en | Product[25] | 29.050 | 6.480 | 22.570 | ∞ | input port clock |
| Path 6 | ∞ | 18 | 615 | en | Product[24] | 28.938 | 6.233 | 22.705 | ∞ | input port clock |
| Path 7 | ∞ | 18 | 615 | en | Product[26] | 28.803 | 6.239 | 22.565 | ∞ | input port clock |
| Path 8 | ∞ | 19 | 615 | en | Product[31] | 28.524 | 6.127 | 22.397 | ∞ | input port clock |
| Path 9 | ∞ | 17 | 615 | en | Product[23] | 27.182 | 5.855 | 21.327 | ∞ | input port clock |
| Path 10 | ∞ | 17 | 615 | en | Product[22] | 26.995 | 5.878 | 21.117 | ∞ | input port clock |

Figure 87: Proposed 16-Bit Multiplier Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 158 | 0 | 0 | 8150 | 1.94 |
| SLICEL | 91 | 0 | | | |
| SLICEM | 67 | 0 | | | |
| LUT as Logic | 529 | 0 | 0 | 32600 | 1.62 |
| using O5 output only | 0 | | | | |
| using O6 output only | 348 | | | | |
| using O5 and O6 | 181 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 0 | 0 | 0 | 65200 | 0.00 |
| Register driven from within the Slice | 0 | | | | |
| Register driven from outside the Slice | 0 | | | | |
| Unique Control Sets | 0 | | 0 | 8150 | 0.00 |

Figure 88: Proposed 16-Bit Multiplier Area

## 6.5 Observations

Table 6.1:16-Bit Hybrid Multiplier Results

| MULTIPLIER | Total Power (W) | Delay (ns) | Power Delay Product (W.ns) | Slice | LUT as logic |
|---|---|---|---|---|---|
| Hybrid multiplier_Parallel addition_ LING8+SKSA8+LING8+SKSA8 | 33.827 | 25.573 | 865.057 | 183 | 644 |
| Hybrid multiplier_Serial addition_ LING8+SKSA8+LING8+SKSA8 | 13.898 | 42.897 | 596.182 | 231 | 778 |
| Hybrid multiplier_2 chain tree addition_ LING8+SKSA8+LING8+SKSA8 | 13.555 | 31.761 | 430.520 | 224 | 780 |
| Hybrid multiplier_4 chain tree addition_ LING8+SKSA8+LING8+SKSA8 | 13.355 | 29.783 | 397.751 | 216 | 743 |
| Hybrid multiplier_4 chain tree addition_LKSA | 13.118 | 28.400 | 372.551 | 228 | 774 |
| Hybrid multiplier_4 chain tree addition_LBKA | 32.572 | 25.490 | 830.260 | 156 | 543 |
| **Hybrid multiplier_4 chain tree addition_LSKSA (Proposed)** | **12.073** | **30.375** | **366.717** | **158** | **529** |

4-chain tree with LSKSA (Proposed) achieved the best Power Delay Product, making it ideal for low-power applications.

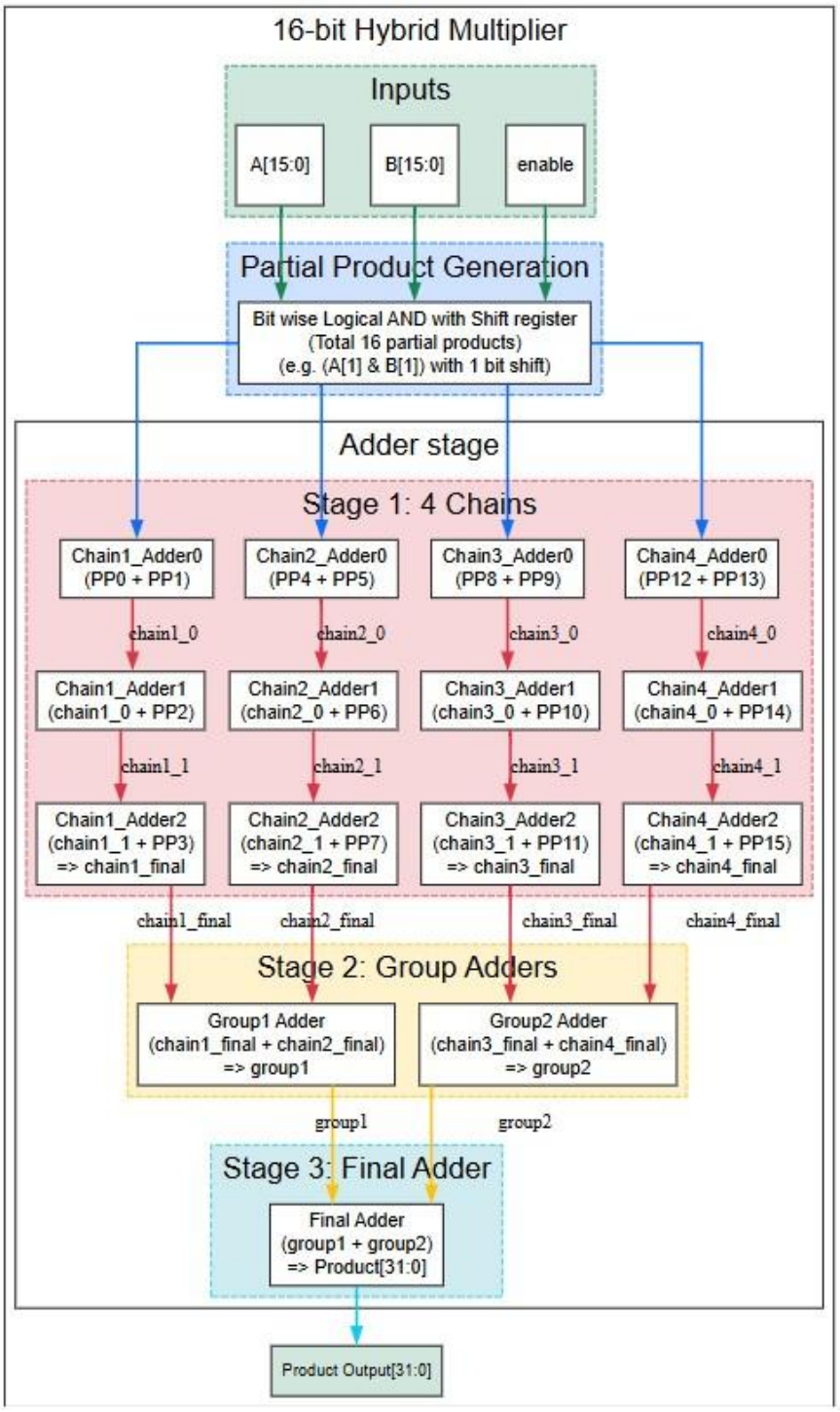



Figure 89: Block Diagram of Proposed 16-bit Hybrid Multiplier

# Chapter 7

# Delay Unit Design

## 7.1 Introduction

In digital FIR filter architectures, delay units are fundamental components that manage the propagation of input data samples through the filter taps. They ensure each tap processes a time-shifted version of the input, which is essential for accurate convolution-based filtering. This chapter outlines two types of delay units utilized in this project: a conventional version without an enable signal and an enhanced version integrated with an enable control. The enhanced version plays a key role in power-efficient FIR filter design.

## 7.2 Conventional Delay Unit

The conventional delay unit, used in all the baseline FIR filters developed in this project, operates based on a basic D flip-flop mechanism. This design captures and forwards the input data to the next stage at every positive clock edge, regardless of whether new data needs to be processed.

**Operational Characteristics:**

- **Clock-driven propagation**: Data is latched and passed forward on each clock cycle.
- **Reset feature**: During reset, the stored data is cleared to zero.
- **No conditional gating**: The unit updates continuously, even during idle input periods.

**Implications in FIR Filters:**

This design is simple and easy to implement, making it a common choice in traditional FIR architectures. However, it lacks data flow control and, therefore, suffers from higher dynamic power consumption, especially in cases where new data samples are infrequent. Since the data path remains active throughout, unnecessary toggling can occur, resulting in less efficient power usage.

## 7.3 Enhanced Delay Unit (With Enable Signal)

To overcome the inefficiencies of continuous data propagation, an enhanced delay unit was introduced in the **proposed FIR filter**. This version includes an **enable signal**, allowing the

unit to update its output only when necessary. The inclusion of this control logic makes the delay element dynamic and data-aware.

**Operational Characteristics:**

- **Enable-based propagation**: The unit updates the stored data only when the enable signal is asserted.
- **Reset integration**: Similar to the conventional unit, it resets the stored value when the reset signal is activated.
- **Gating logic**: When the enable signal is low, the unit holds the previous value and avoids toggling, thereby reducing power consumption.

  The internal behaviour of the delay unit can be summarised as

## 7.4 Impact on FIR Filter Design:

The enable-controlled delay unit provides a more efficient means of handling data transitions. It is particularly useful in real-time systems or low-power applications where data samples may not arrive continuously. By incorporating conditional logic, this delay unit effectively eliminates unnecessary switching activity and extends power savings across the filter chain. In the proposed enhanced FIR filter, this delay unit plays a critical role in managing data flow between taps, ensuring that each stage processes only valid and required samples. This selective propagation leads to a substantial reduction in dynamic power usage without sacrificing performance or accuracy.

Table 7.1: Conventional vs Enabled Delay Units

| Feature | Without Enable | With Enable (Proposed) |
|---|---|---|
| **Data Update Behaviour** | Always updates on clock | Updates only when enabled |
| **Power Consumption** | Higher (uncontrolled toggling) | Lower (data-aware gating) |
| **Design Simplicity** | Simple | Slightly complex |
| **Control Signal Support** | None | Requires additional control |
| **Usage** | All baseline FIR filters | Only in proposed FIR filter |

# Chapter 8

# Design of Enhanced FIR Filter

## 8.1 Introduction

Finite Impulse Response (FIR) filters are essential components in digital signal processing systems due to their inherent stability and linear phase characteristics. This chapter presents the design and implementation of various FIR filter architectures developed as part of this project. The focus is on leveraging hybrid arithmetic units and architectural optimizations to achieve improvements in speed, power, and area.

A 16-tap FIR filter with symmetric coefficients was used as the base model. The coefficients, optimized for ECG low-pass filtering, are as follows (hexadecimal format):

**Coefficients:**

```
h[0]  = 16'h0003;  h[1]  = 16'h0007;  h[2]  = 16'h000C;  h[3]  = 16'h0012;
h[4]  = 16'h0019;  h[5]  = 16'h0020;  h[6]  = 16'h0026;  h[7]  = 16'h002A;
h[8]  = 16'h002A;  h[9]  = 16'h0026;  h[10] = 16'h0020;  h[11] = 16'h0019;
h[12] = 16'h0012;  h[13] = 16'h000C;  h[14] = 16'h0007;  h[15] = 16'h0003;
```

The symmetry of the coefficients allows for paired computation and contributes to hardware efficiency.

## 8.2 FIR Filter Using Direct Arithmetic

The **Direct Arithmetic** design employs Verilog's built-in * and + operators for multiplications and additions. This means each output sample is generated directly using:

$$y[n] = b0*x[n] + b1*x[n-1] + ... + b15*x[n-15]$$

- **Multiplications**: Direct coefficient-to-sample multiplication using synthesizable *.
- **Additions**: Partial products are summed sequentially as they appear in the equation.

This design is easy to implement and synthesize but is less optimized for performance and power. The adder structure is inferred automatically and lacks any architectural enhancement for timing or resource optimization.

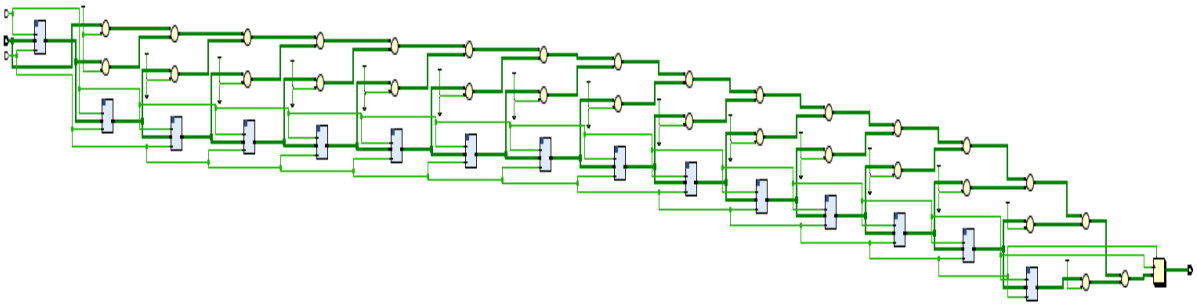


Figure 90: Schematic of 16-tap Conventional FIR Filter

## 8.3 FIR Filter with Parallel Addition

In this design, all multiplications are carried out simultaneously. The resulting partial products are also summed in parallel, similar to the parallel addition structure used in your hybrid multipliers.

- **Multipliers**: Operate in parallel.
- **Adders**: Organized in a fully parallel structure using multiple hybrid adder blocks.
- **Performance**: Extremely fast due to concurrent computation.
- **Drawback**: High area and power usage due to full parallelism.

This method prioritizes speed and is suitable when real-time high-throughput operation is necessary.

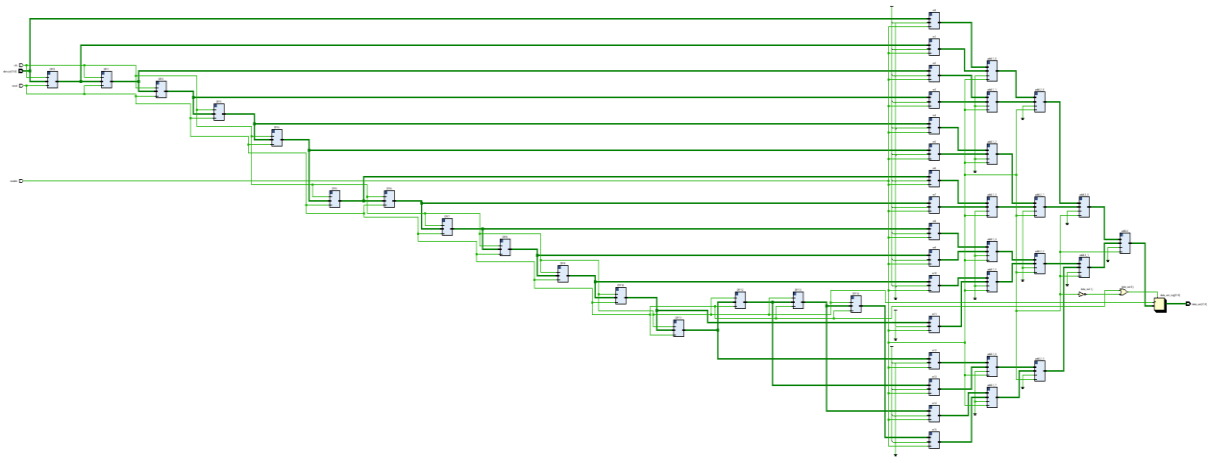


Figure 91: Schematic of 16-tap FIR Filter with Parallel Addition Architecture

## 8.4 FIR Filter with Serial Addition

Here, all the input samples are multiplied with their corresponding coefficients in parallel. However, instead of summing the products in parallel, a single hybrid adder (or a small number of them) is reused in multiple clock cycles to accumulate the result.

- **Operation**: Sequential accumulation of partial products.
- **Efficiency**: Area and power consumption are minimized.
- **Limitation**: Higher latency due to serialized addition steps.

This design mimics the serial hybrid multiplier and is ideal for applications prioritizing low power and minimal hardware usage over speed.

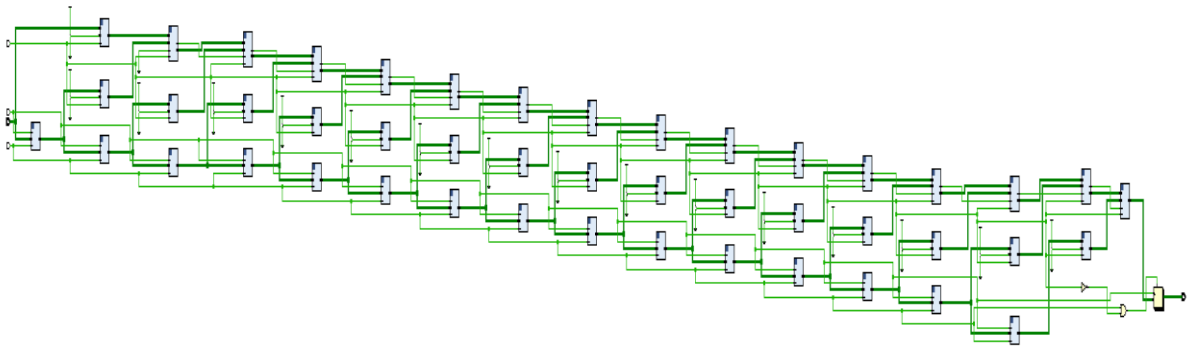


Figure 92: Schematic of 16-tap FIR Filter with Serial Addition Architecture

## 8.5 FIR Filter with Tree-Based Addition

This version uses a tree structure to sum the 16 partial products obtained from parallel multiplications. The addition tree uses either 2-chain or 4-chain hybrid adder structures inspired by your hybrid multiplier trees.

- **Stages**:
    - **Stage 1**: Four sets of 4 multipliers feed into serial adder chains.
    - **Stage 2**: Pairs of results from Stage 1 are combined.
    - **Stage 3**: Final summation produces the output.
- **Performance**: Offers a balance between area, power, and speed.

The tree-based method improves timing performance significantly over serial architectures while consuming fewer resources than fully parallel designs.

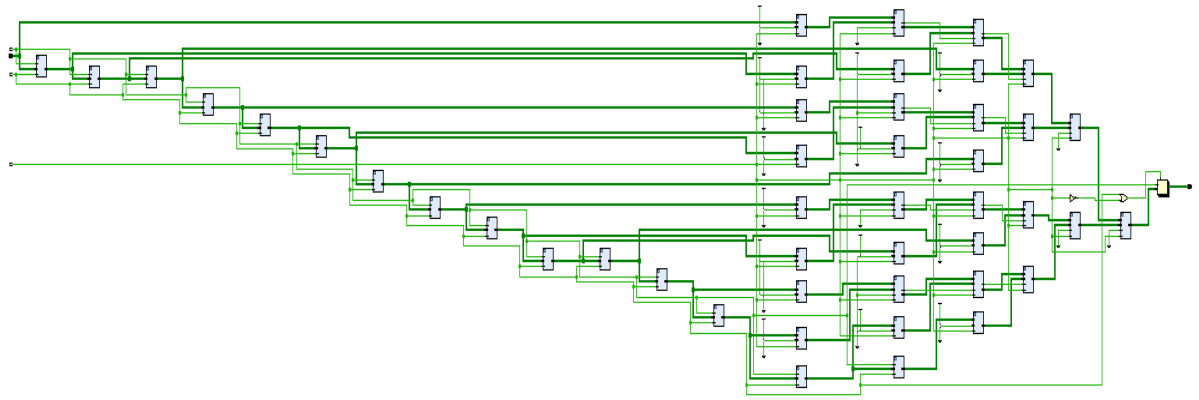


Figure 93: Schematic of 16-tap FIR Filter with Four-Chain Tree Addition Architecture

## 8.6 Enhanced FIR Filter with Enable-Controlled Delay

This is the proposed architecture that extends the tree-based addition structure with enable-controlled delay element**s**. These enable signals control when the D flip-flops (used as delay elements) propagate data, which minimizes unnecessary switching activity.

- **Multipliers**: Implemented with hybrid multipliers using 4-chain tree adder logic.
- **Delay Units**: Enable-controlled D flip-flops reduce dynamic power.
- **Adder Tree**: Three-level tree as described in Section 10.5.
- **Benefits**:
    - Dynamic power savings due to controlled switching.
    - Meets tight timing constraints while minimizing energy consumption.

This architecture is particularly suitable for applications like ECG signal filtering in wearable devices, where both low power and real-time processing are critical.

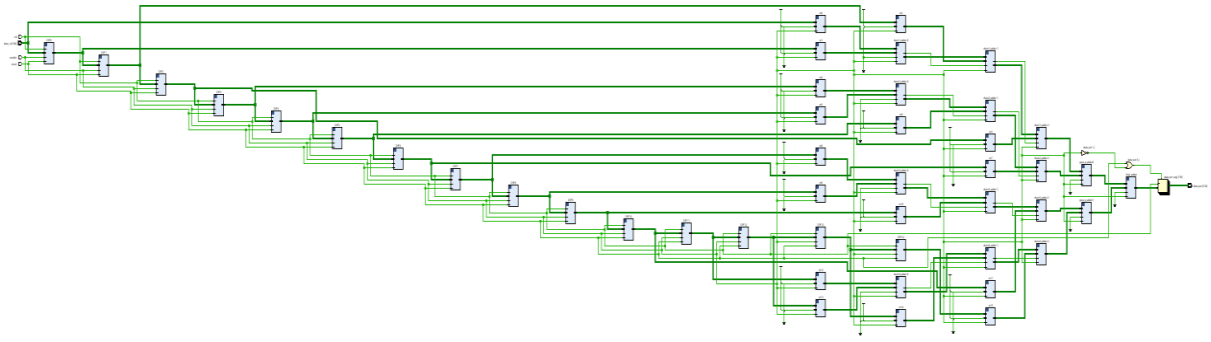


Figure 94: Schematic of Proposed Enhanced 16-tap FIR Filter

## Synthesis Report: Enhanced Fir Filter

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

| | |
|---|---|
| **Total On-Chip Power:** | **5.957 W** |
| **Design Power Budget:** | **Not Specified** |
| **Process:** | typical |
| **Power Budget Margin:** | **N/A** |
| **Junction Temperature:** | **54.8°C** |
| Thermal Margin: | 30.2°C (6.0 W) |
| Ambient Temperature: | 25.0 °C |
| Effective ϑJA: | 5.0°C/W |
| Power supplied to off-chip devices: | 0 W |
| Confidence level: | Low |

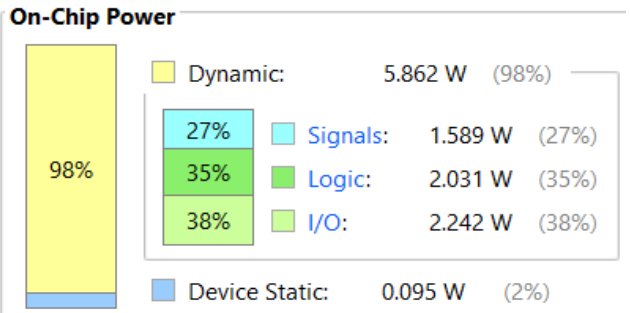



Figure 95: Enhanced Fir Filter-Power

Unconstrained Paths - NONE - NONE - Setup

| Name | Slack | Levels | High Fanout | From | To | Total Delay | Logic Delay | Net Delay | Requirement | Source Clock |
|---|---|---|---|---|---|---|---|---|---|---|
| Path 1 | ∞ | 26 | 1056 | enable | data_out_reg[24]/D | 30.928 | 6.984 | 23.944 | ∞ | input port clock |
| Path 2 | ∞ | 26 | 1056 | enable | data_out_reg[23]/D | 30.900 | 6.956 | 23.944 | ∞ | input port clock |
| Path 3 | ∞ | 25 | 1056 | enable | data_out_reg[22]/D | 29.807 | 6.832 | 22.975 | ∞ | input port clock |
| Path 4 | ∞ | 25 | 1056 | enable | data_out_reg[21]/D | 29.614 | 6.832 | 22.781 | ∞ | input port clock |
| Path 5 | ∞ | 24 | 1056 | enable | data_out_reg[20]/D | 29.485 | 6.708 | 22.777 | ∞ | input port clock |
| Path 6 | ∞ | 23 | 1056 | enable | data_out_reg[19]/D | 28.594 | 6.356 | 22.238 | ∞ | input port clock |
| Path 7 | ∞ | 22 | 1056 | enable | data_out_reg[18]/D | 27.592 | 6.004 | 21.588 | ∞ | input port clock |
| Path 8 | ∞ | 21 | 1056 | enable | data_out_reg[17]/D | 26.623 | 5.644 | 20.979 | ∞ | input port clock |
| Path 9 | ∞ | 20 | 1056 | enable | data_out_reg[16]/D | 25.734 | 5.292 | 20.442 | ∞ | input port clock |
| Path 10 | ∞ | 19 | 1056 | enable | data_out_reg[15]/D | 24.264 | 5.168 | 19.096 | ∞ | input port clock |

Figure 96: Enhanced Fir Filter-Timing

2. Slice Logic Distribution

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 267 | 0 | 0 | 8150 | 3.28 |
| SLICEL | 160 | 0 | | | |
| SLICEM | 107 | 0 | | | |
| LUT as Logic | 903 | 0 | 0 | 32600 | 2.77 |
| using O5 output only | 0 | | | | |
| using O6 output only | 578 | | | | |
| using O5 and O6 | 325 | | | | |
| LUT as Memory | 0 | 0 | 0 | 9600 | 0.00 |
| LUT as Distributed RAM | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| LUT as Shift Register | 0 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 0 | | | | |
| Slice Registers | 265 | 0 | 0 | 65200 | 0.41 |
| Register driven from within the Slice | 168 | | | | |
| Register driven from outside the Slice | 97 | | | | |
| LUT in front of the register is unused | 10 | | | | |
| LUT in front of the register is used | 87 | | | | |
| Unique Control Sets | 2 | | 0 | 8150 | 0.02 |

Figure 97: Enhanced Fir Filter-Area

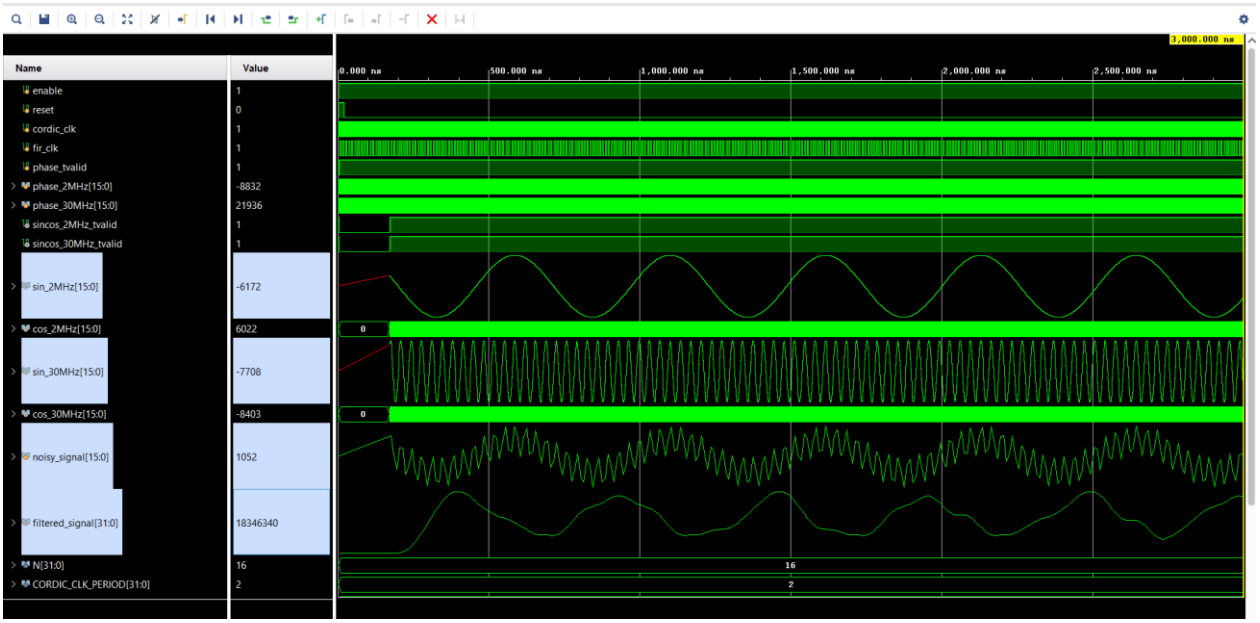



Figure 98: Enhanced Fir Filter Lowpass Analog Output

## 8.7 FIR Architectures with Timing Constraints

To assess performance under real-time operation, both the direct and enhanced architectures were evaluated with specific timing constraints:

### 8.7.1 Direct Arithmetic at 28.571 MHz (35 ns Period)

- **Clock Constraint**: create_clock -period 35.0
- **Result**: The direct arithmetic design met the constraint with ease, due to its straightforward implementation.
- **Insight**: No special logic for timing optimization, but adequate for moderate-speed applications.

**Synthesis Report:**

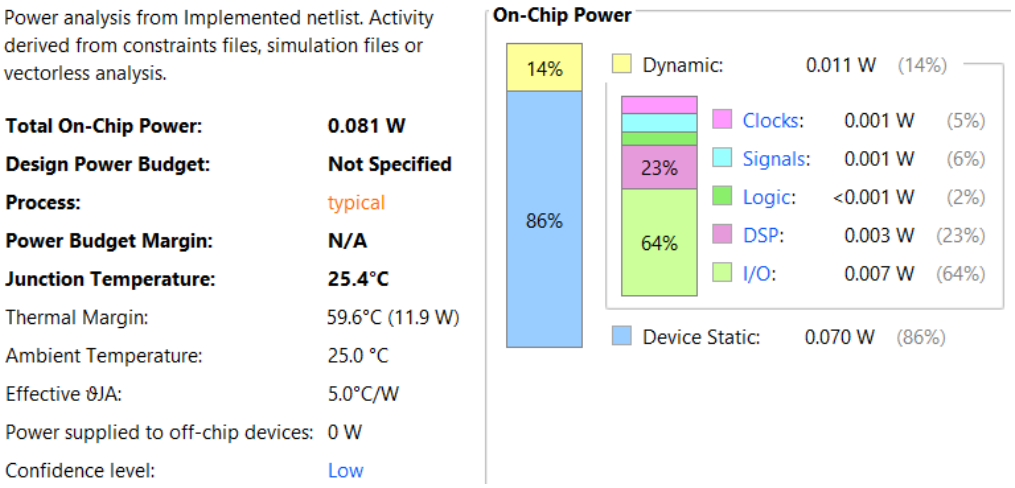



Figure 99: Conventional Fir Filter Power with Timing Constraints

```
Timing Report

Slack (MET) :             1.709ns  (required time - arrival time)
  Source:                 Mul15/CLK
                            (rising edge-triggered cell DSP48E1 clocked by clk  {rise@0.000ns fall@17.500ns period=35.000ns})
  Destination:            data_out_reg[25]/D
                            (rising edge-triggered cell FDRE clocked by clk  {rise@0.000ns fall@17.500ns period=35.000ns})
  Path Group:             clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            35.000ns  (clk rise@35.000ns - clk rise@0.000ns)
  Data Path Delay:        33.177ns  (logic 28.611ns (86.238%)  route 4.566ns (13.762%))
  Logic Levels:           30  (CARRY4=17 DSP48E1=11 LUT2=2)
  Clock Path Skew:        -0.141ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.203ns = ( 39.203 - 35.000 )
    Source Clock Delay      (SCD):    4.665ns
    Clock Pessimism Removal (CPR):    0.322ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns
```

Figure 100: Conventional Fir Filter Timing Report with Timing Constraints

### 8.7.2 Enhanced FIR Filter at 33.333 MHz (30 ns Period)

- **Clock Constraint**: create_clock -period 30.0
- **Result**: The enhanced design successfully closed timing under this tighter constraint.
- **Advantage**: The enable-controlled delay and structured adder tree ensured minimal critical path delays and reduced power.

### Synthesis Report:

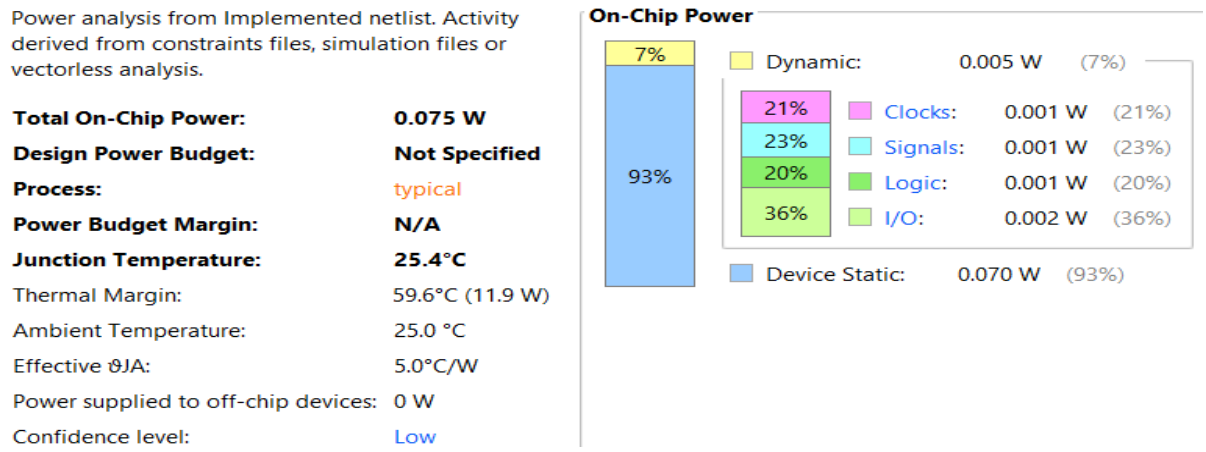



Figure 101: Enhanced Fir Filter Power with Timing Constraint

```
Timing Report

Slack (MET) :             10.044ns  (required time - arrival time)
  Source:                 data_in[1]
                            (input port clocked by clk  {rise@0.000ns fall@15.000ns period=30.000ns})
  Destination:            data_out_reg[22]/D
                            (rising edge-triggered cell FDRE clocked by clk  {rise@0.000ns fall@15.000ns period=30.000ns})
  Path Group:             clk
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            30.000ns  (clk rise@30.000ns - clk rise@0.000ns)
  Data Path Delay:        22.065ns  (logic 4.938ns (22.379%)  route 17.127ns (77.621%))
  Logic Levels:           26  (IBUF=1 LUT3=9 LUT4=8 LUT5=4 LUT6=4)
  Input Delay:            2.500ns
  Clock Path Skew:        4.567ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    4.567ns = ( 34.567 - 30.000 )
    Source Clock Delay      (SCD):    0.000ns
    Clock Pessimism Removal (CPR):    0.000ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
```

Figure 102: Enhanced Fir Filter Timing Report with Timing Constraints

## 8.8 Summary

Each FIR filter design represents a trade-off among speed, area, and power:

The Enhanced FIR Filter with tree-based addition and enable-controlled delay achieved the best overall performance, making it the most effective design for low-power, high-speed applications.

Table 8.1: 16 Tap Fir Filter Results

| **Type** | **16tap FIR Filter** | **Total Power (w)** | **Delay (ns)** | **Power Delay Product (W.ns)** | **Slice** | **LUT as logic** | **Slice Registers** |
|---|---|---|---|---|---|---|---|
| Without timing constraints | Fir Direct arithmetic (conventional) | 36.929 | 35.436 | 1308.616 | 78 | 104 | 234 |
| | Fir_parallel_addition | 26.076 | 30.127 | 785.591 | 269 | 883 | 265 |
| | Fir_serial_addition | 23.806 | 38.870 | 925.339 | 285 | 970 | 267 |
| | Fir_tree_addition | 23.316 | 31.102 | 725.174 | 279 | 913 | 265 |
| | **Enhanced FIR filter tree addition** | **5.975** | **30.928** | **184.794** | **267** | **903** | **265** |
| With timing constraints | FIR Direct arithmetic with timing constraints 28.571 MHz (35ns period) | 0.081 | 33.177 | 2.687 | 75 | 104 | 234 |
| | Enhanced FIR filter With timing constraints 33.333 MHz (30nsperiod) | 0.075 | 22.065 | 1.654 | 201 | 921 | 265 |

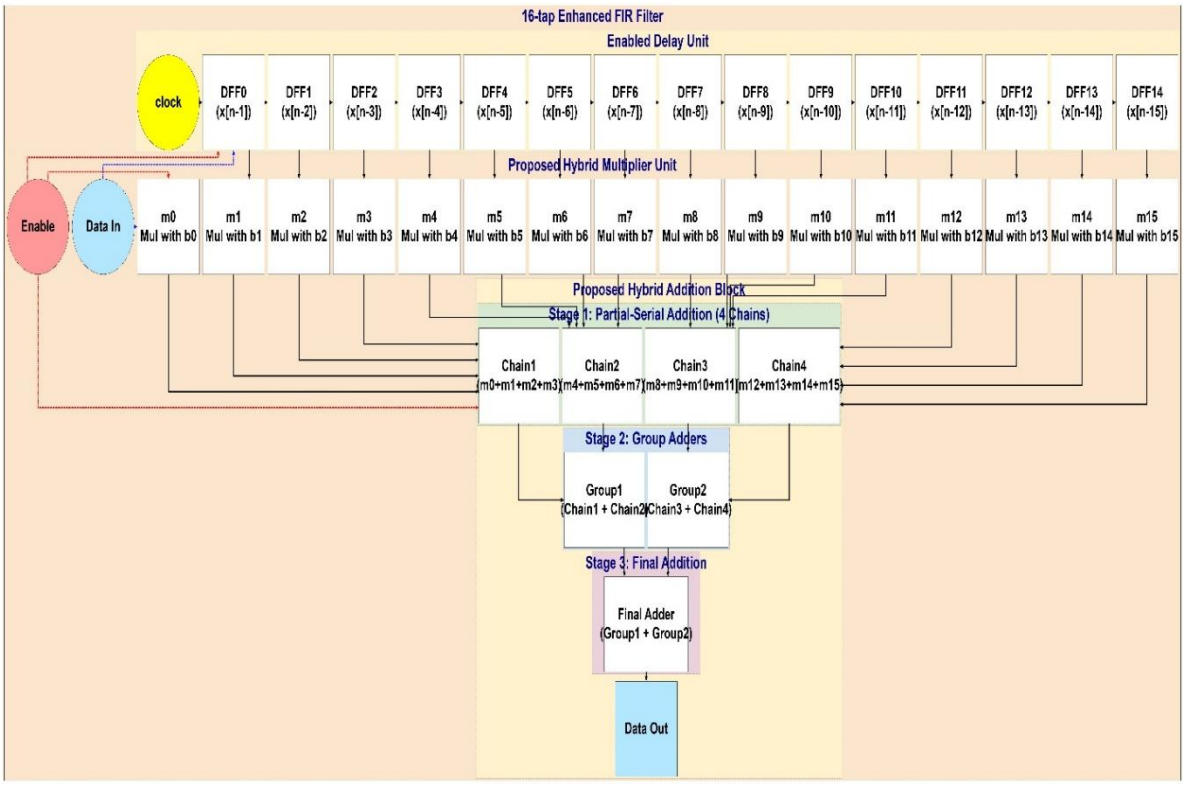



Figure 103: Block Diagram of Proposed 16 tap Enhanced FIR filter

# Chapter 9

# Introduction To Verilog

## 9.1 Definition:

The Hardware Description Language (HDL) Verilog is used for describing digital systems such as flip-flops, microprocessors, network switches, or other electronic components. Verilog was developed to streamline the design process and improve the robustness and flexibility of HDL based development. It empowers engineers to define the desired behaviour of digital circuits, which can then be automatically converted into real-world hardware components like combinational gates and sequential logic blocks. Verilog has become one of the most widely used HDLs in the semiconductor industry. Like other HDLs, Verilog supports both bottom-up and top-down design approaches:

- **Bottom-Up Design:**
  Traditionally, designs are built from the ground up using logic gates at the gate level. This allows for fundamental and structured design development.
- **Top-Down Design:**
  Offers advantages such as early testing, easier technology migration, and more organized system-level design.

## 9.2 Uses of Verilog:

Verilog provides a high level of abstraction, which simplifies hardware design by allowing engineers to focus on functionality rather than transistor-level details. For instance, designing a D flip-flop in Verilog does not require precise control of transistor placements or timing details like rise/fall times. Instead, Verilog allows designers to describe behaviourally what the circuit should do, and synthesis tools handle the physical implementation.

## 9.3 Features of Verilog:

- Case-sensitive language
- Keywords are lowercase
- Syntax derived mostly from the C programming language
- Model’s digital systems at algorithmic, RTL, gate, and switch levels

- No concept of packages in traditional Verilog
- Supports powerful simulation features like TEXTIO, PLI, and UDPs

## 9.4 Verilog Modeling Styles

Verilog supports several Modeling styles, each suited for different design abstraction levels and purposes:

**1. Behavioural Modeling**

- Uses procedural blocks (always, initial)
- Describes functionality without specifying hardware structure
- Uses constructs like if-else, case statements, loops
- Good for complex algorithms and high-level system behaviour

**2. Dataflow Modeling**

- Uses continuous assignments (assign)
- Describes connections between signals
- Models signal flow through expressions
- Ideal for combinational logic

**3. Structural Modeling**

- Describes design as interconnected components (gates, modules)
- Uses module instantiation and port connections
- Closely resembles the actual hardware implementation
- Good for bottom-up design approaches

**4. Gate-Level Modeling**

- Uses built-in primitive gates (AND, OR, NOT)
- Very low level of abstraction
- Explicitly describes circuit in terms of logic gates
- Useful for technology-specific implementations

**5. Switch-Level Modeling**

- Models at transistor switch level
- Uses MOS switches (nmos, pmos, cmos)
- Less commonly used in modern design flows
- Valuable for custom circuit design

# Chapter 10

# Xilinx Vivado 2024.1

## 10.1 Introduction

Xilinx Vivado is a modern FPGA design suite developed by Xilinx, replacing the legacy ISE tool. It supports a comprehensive FPGA development flow including design entry, synthesis, implementation, simulation, and device programming. Vivado 2024.1 introduces enhanced performance, an improved user interface, and more efficient handling of IP cores, making it suitable for high-performance and low-power designs. In this chapter, the features, GUI components, and workflow of Vivado 2024.1 are described, demonstrating how it was used to implement and analyze the proposed digital designs in this project.

## 10.2 Vivado Design Flow Overview

The Vivado design methodology follows a structured sequence of stages, ensuring that each component of the digital system is verified and optimized:

- RTL design entry (Verilog or VHDL)
- Functional simulation (optional, pre-synthesis)
- Synthesis of the RTL design
- Implementation (placement and routing)
- Bitstream generation
- Device programming and hardware debugging

This flow ensures that the design is fully validated before hardware deployment.

## 10.3 Vivado GUI Description

Vivado 2024.1 features an integrated development environment with the following major panels:

- **Start Page**: Allows users to create or open projects, and access documentation and recent activity.
- **Flow Navigator**: Located on the left, it provides quick access to steps like Simulation, Synthesis, Implementation, and Programming.
- **Project Manager**: Manages design sources, constraints, simulation files, and IPs.

- **Diagram Window**: Used for block design and schematic views.
- **Tcl Console**: Enables users to interact with Vivado using scripting commands.
- **Messages, Log, and Reports Tabs**: Provide detailed feedback during each design phase.

## 10.4 Project Creation in Vivado 2024.1

Creating a project in Vivado involves:

1. Selecting “Create New Project” on the start page.
2. Assigning a project name and selecting a directory.
3. Choosing the project type (usually an RTL Project).
4. Adding design source files and simulation sources.
5. Associating a constraints file (*.xdc).
6. Selecting the target FPGA device or evaluation board.

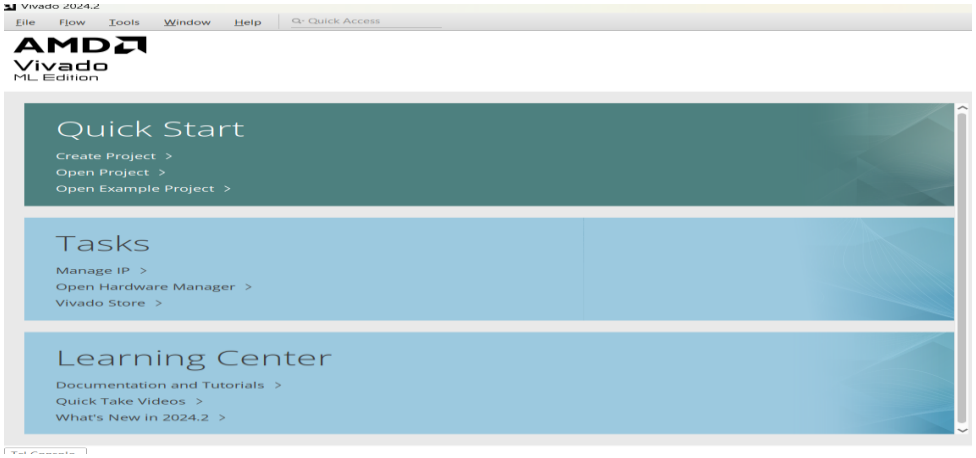



Figure 104: Creating the project file

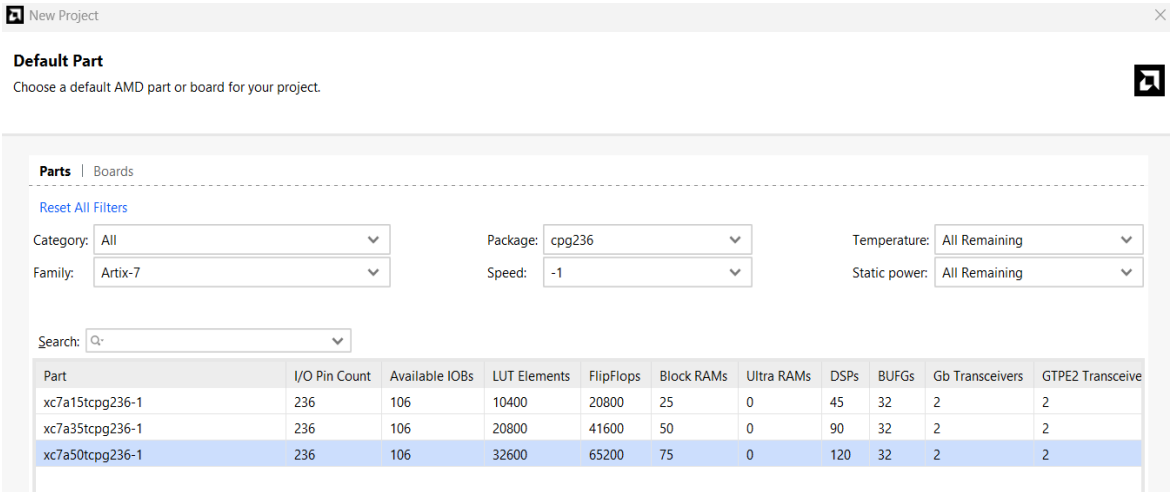



Figure 105: Board used in this Project (Artix-7: cpg236)

## 10.5 Adding Source Files and Constraints

Design files such as Verilog modules are added via the "Add Sources" interface. The tool supports both new file creation and importing existing files. Constraints files (*.xdc) are used to define pin locations, clocks, and timing constraints. These constraints are crucial for real-world deployment on FPGA boards.

## 10.6 Simulation and Synthesis

Vivado 2024.1 includes a robust simulation environment using the XSIM engine. In this project, simulation was extensively used to verify the correctness of all individual modules and top-level designs such as:

- **Adder architectures** (LING, KSA, BKA, SKSA, and hybrid variants)
- **Multiplier modules** (Vedic, Dadda, Wallace, Karatsuba, 2-chain, 4-chain hybrids)
- **FIR filter architectures** (Direct, Parallel, Serial, Tree-based, Enhanced)

**Testbenches** were written for each module to validate functionality against expected outputs using various stimulus conditions. During simulation:

- Waveform traces were analysed using the XSIM waveform viewer.
- Internal signals like carry, sum, partial products, and delayed inputs were tracked for correctness.
- Both functional simulation and timing simulation (post-synthesis) were performed.

**Synthesis Overview**

Synthesis converts the HDL design into a gate-level netlist targeting the selected FPGA family (e.g., Artix-7). In Vivado, this process includes:

- **Parsing** and analyzing Verilog code
- **Optimizing** combinational and sequential logic
- **Inferring** hardware blocks like adders, multipliers, and registers
- **Mapping** logic to LUTs, flip-flops (FFs), DSP slices, and block RAMs

Key synthesis reports generated include:

- **Utilization report**:
    - Number of LUTs, FFs, DSPs, and IOs used
    - Hybrid designs typically consume more LUTs due to their custom architectures, but optimized versions (like LSKSA) reduce this overhead.

- **Timing summary**:
    - Worst Negative Slack (WNS)
    - Total Negative Slack (TNS)
    - Critical path delay: For enhanced FIR filters, the target delay was met under the 30ns clock period constraint.
- **Power estimation**:
    - Dynamic and static power breakdown
    - Enable-controlled designs showed lower switching activity and reduced power
- **Clock report**:
    - Confirms that input/output delays, setup, and hold timing met the defined constraints:
      create_clock -name clk -period 30.0 [get_ports clk]
      set_input_delay -clock clk -max 2.5 ...

The synthesis results validated that all modules, including the enhanced FIR filter, successfully met the design and timing requirements.

## 10.7 Design Analysis Tools

Vivado offers multiple tools for in-depth design analysis:

- **Waveform Viewer**: For visualizing simulation signals.
- **Netlist and Schematic Viewer**: Displays logic connections and hierarchies.
- **Timing Report**: Shows critical paths and slack times.
- **Utilization Summary**: Reports how much of each resource is consumed.
- **Power Analyzer**: Estimates static and dynamic power consumption.
- **ILA (Integrated Logic Analyzer)**: A debugging tool that can probe internal signals in real-time.

## 10.8 Application in This Project

In this project, Vivado 2024.1 was used for:

- Writing and managing Verilog code for FIR filters, adders, and multipliers.
- Simulating and validating functional behaviour using testbenches.
- Synthesizing and implementing the proposed designs.
- Extracting reports on area, power, and timing for performance comparison.

# Chapter 11

# Conclusion

## 11.1 Future Scope

The enhanced hybrid FIR filter design presented in this work lays the foundation for several promising research directions. Key areas for future development include:

- **Accurate Power Profiling**: To complement the current estimates, future implementations should incorporate complete pin constraints and switching activity data (.saif or .vcd) to enable precise dynamic power analysis and real-world energy profiling.
- **Application-Specific Filter Customization**: The symmetric coefficient set can be further tuned or dynamically reconfigured to suit various real-time biomedical, audio, or image-processing applications.
- **Scalability and Bit-Width Extension**: The proposed 32-bit hybrid adder and 4-chain multiplier structures can be extended to higher word lengths, enabling broader applicability in high-performance DSP systems.
- **FPGA Deployment**: Hardware-level deployment and validation on FPGA platforms with proper IO configuration and real-time signal acquisition will facilitate system-level benchmarking and optimization.

**Disclaimer on Power Analysis:** The power estimation results presented in this work were obtained using Xilinx Vivado's built-in power analysis tools without incorporating complete pin constraints and switching activity (.saif or .vcd) files. As such, the reported values reflect only static and estimated dynamic power based on default toggle rates. For accurate real-time power profiling, further analysis using full post-implementation simulation and switching activity capture is recommended as future work.

## 11.2 Conclusion

The project titled "Enhanced FIR Filter Design Using Hybrid Adders and Multipliers" successfully proposed and validated an optimized FIR filter architecture leveraging advanced hybrid arithmetic units. The design introduced a 32-bit hybrid adder combining LING16 and SKSA16 with enable logic for power gating, and a hybrid multiplier utilizing a 4-chain tree

based partial product addition structure built with hybrid adders. These innovations significantly improved the performance of the FIR filter in terms of power consumption, propagation delay, and power-delay product (PDP). Comprehensive synthesis results demonstrated the effectiveness of the proposed architecture at both the individual component level and the overall FIR system level. While these improvements introduced a modest area overhead, the overall trade-off favoured substantial gains in speed and energy efficiency.

summary of the performance improvements is as follows:

**A. Component-Level Improvements**

- **16-bit Hybrid Adder vs. Ling Adder**
  - Power Reduction: 24.66%
  - Delay Reduction: 2.97%
  - PDP Improvement: 26.90%
- **32-bit Hybrid Adder vs. Ling Adder**
  - Power Reduction: 33.63%
  - Delay Reduction: 18.60%
  - PDP Improvement: 45.64%
- **32-bit Hybrid Adder vs. Sparse Kogge–Stone Adder (SKSA)**
  - Power Reduction: 33.63%
  - Delay Reduction: 15.36%
  - PDP Improvement: 43.82%
- **16-bit Hybrid Multiplier vs. Conventional Multipliers**
  - Power Reduction (Vedic): 62.42%
  - PDP Improvement (Vedic): 41.54%
  - Power Reduction (Wallace Tree): 69.19%
  - PDP Improvement (Wallace Tree): 60.26%

**B. System-Level Improvements (16-Tap FIR Filter)**

- Compared to a FIR filter using direct arithmetic operators without any constraints:
  - Power Consumption Reduced by: 83.82%
  - Delay Reduced by: 12.72%
  - Power-Delay Product Improved by: 85.87%

Table 11.1: Performance Improvements in Proposed Designs

| Component | Metric | Conventional Design | Proposed Hybrid Design Improvement |
|---|---|---|---|
| **16-bit Adder** | Power Consumption | Ling adder | 24.66% reduction |
| | Delay | Ling adder | 2.97% reduction |
| | Power Delay product | Ling adder | 26.90% reduction |
| **32-bit Adder** | Power Consumption | Ling adder | 33.63% reduction |
| | Delay | Ling adder | 18.6% reduction |
| | Power Delay product | Ling adder | 45.64% reduction |
| | Power Consumption | Sparse Kogge–Stone adder | 33.63% reduction |
| | Delay | Sparse Kogge–Stone adder | 15.36% reduction |
| | Power Delay product | Sparse Kogge–Stone adder | 43.82% reduction |
| **16-bit Multiplier** | Power Consumption | Vedic Multiplier | 62.415% reduction |
| | Power Delay product | Vedic Multiplier | 41.535% reduction |
| | Power Consumption | Wallace Tree Multiplier | 69.192% reduction |
| | Power Delay product | Wallace Tree Multiplier | 60.257% reduction |
| **16 tap FIR Filter** Without constraints | Power Consumption | FIR Filter with Arithmetic Operators | 83.82% reduction |
| | Delay | FIR Filter with Arithmetic Operators | 12.721% reduction |
| | Power Delay Product | FIR Filter with Arithmetic Operators | 85.87% reduction |

## Graphical Representation of Proposed Designs Results

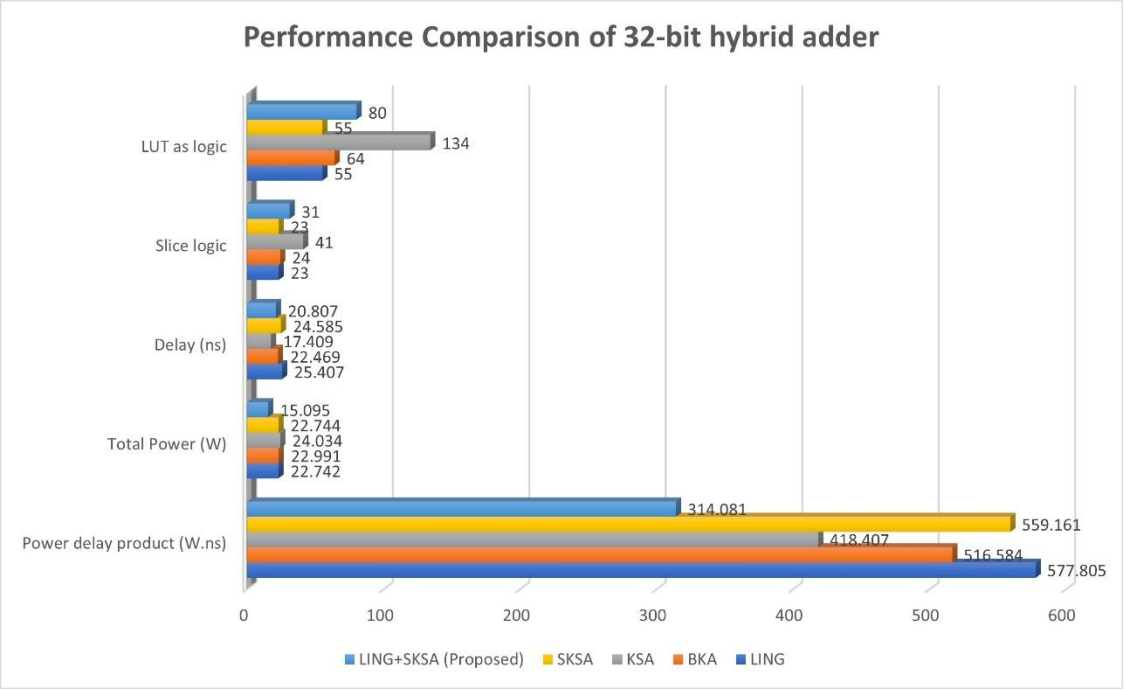



Figure 106: Performance Comparison of 32-bit hybrid adder

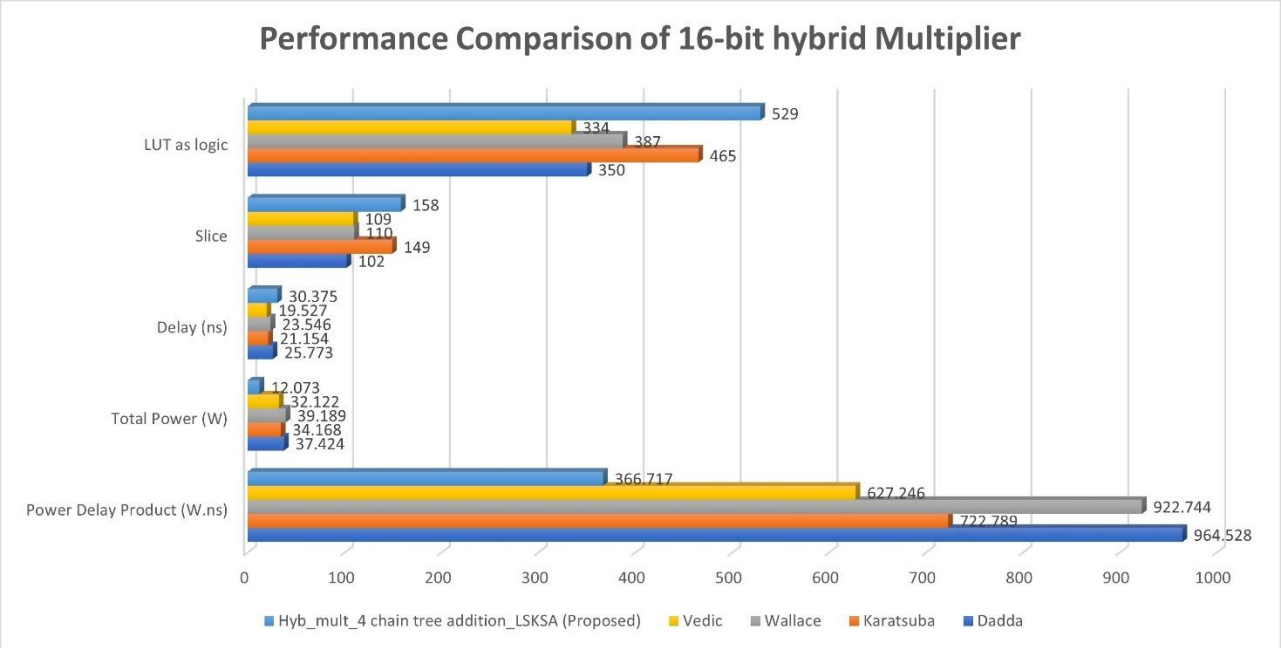



Figure 107: Performance Comparison of 16-bit hybrid Multiplier

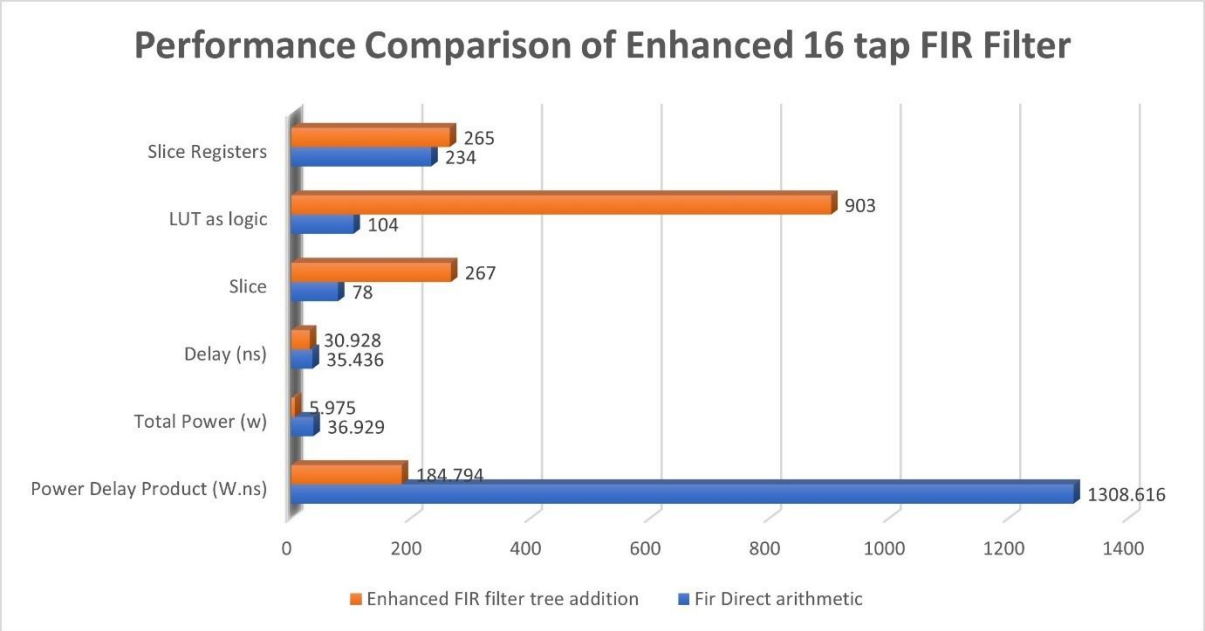



Figure 108: Performance Comparison of Enhanced 16 tap FIR Filter

In conclusion, the proposed FIR filter with hybrid Adder and Multiplier architecture offers a robust, energy-efficient, and high-performance alternative to traditional designs, with flexibility for adaptation to a wide range of signal processing tasks. The architectural methodology adopted combining the strengths of different parallel prefix adders presents a unique, modular, and reusable arithmetic framework for future arithmetic-intensive hardware systems.

# References

[1] J. F. Sayed, B. H. Hasan, B. Muntasir, M. Hasan and F. Arifin, "Design and Evaluation of a FIR Filter Using Hybrid Adders and Vedic Multipliers," 2021 2nd International Conference on Robotics, Electrical and Signal Processing Techniques (ICREST), DHAKA, Bangladesh, 2021, pp. 748-752, Doi: 10.1109/ICREST51555.2021.9331063.

[2] R. R, B. Biju, A. Joy, V. Ganeshan and S. R, "Design and Performance Analysis of Various 32-bit Hybrid Adders using Verilog," 2022 IEEE International Conference on Signal Processing, Informatics, Communication and Energy Systems (SPICES), THIRUVANANTHAPURAM, India, 2022, pp. 105-112, doi:10.1109/SPICES52834.2022.9774131.

[3] S. N. Leela, D. Keerthi Chandrika, K. Swetha, D. G. Kalali and G. Shanthi, "A Novel Design of High Speed Multiplier Using Hybrid Adder Technique," 2024 3rd International Conference for Innovation in Technology (INOCON), Bangalore, India, 2024, pp. 1-5, doi: 10.1109/INOCON60754.2024.10512237.

[4] V. M B, P. Akkamanchi and V. Uttarkar, "Low Power High Speed Brent Kung Adder Using SPST," 2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT), Bangalore, India, 2022, pp. 1-6, doi: 10.1109/GCAT55367.2022.9971848.

[5] T. Shanmugaraja. and N. Kathikeyan, "Power Effective Multiply Accumulation Configuration For Low Power Applications Using Modified Parallel Prefix Adders," 2023 International Conference on Applied Intelligence and Sustainable Computing (ICAISC), Dharwad, India, 2023, pp. 1-7, doi: 10.1109/ICAISC58445.2023.10200142.

[6] N. Shang et al., "A 32-Bit Ripple-Ling Hybrid Carry Adder," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 71, no. 6, pp. 2709-2722, June 2024, doi: 10.1109/TCSI.2024.3352139.

[7] S. Arish and R. K. Sharma, "An efficient binary multiplier design for high speed applications using Karatsuba algorithm and Urdhva Tiryagbhyam algorithm," 2015 Global Conference on Communication Technologies (GCCT), Thuckalay, India, 2015, pp. 192-196, doi: 10.1109/GCCT.2015.7342650.

[8] K. Gunasekaran and M. Manikandan, "Low power and area efficient reconfigurable FIR filter implementation in FPGA," 2013 International Conference on Current Trends in Engineering and Technology (ICCTET), Coimbatore, India, 2013, pp. 300-303, doi:10.1109/ICCTET.2013.6675970.

[9] S. S. Pujari, P. P. Muduli, A. Panda, R. Badhai, S. Nayak and Y. Sahoo, "Design & implementation of FIR filters using on-board ADC DAC & FPGA," International Conference

on Information Communication and Embedded Systems (ICICES2014), Chennai, India, 2014, pp. 1-6, doi: 10.1109/ICICES.2014.7033995.

[10] Firmansyah, Iman & Yamaguchi, Yoshiki. (2022). Real-Time FPGA Implementation of FIR Filter Using OpenCL Design. Journal of Signal Processing Systems. 94. 1-13. 10.1007/s11265-021-01723-6.

[11] M. A. B. M, "An Efficient FPGA Implementation of Wiener Filter," 2024 IEEE 8th International Conference on Information and Communication Technology (CICT), Prayagraj UP, India, 2024, pp. 1-6, doi: 10.1109/CICT64037.2024.10899577.

[12] K. Debnath, S. Dhabal and P. Venkateswaran, "Design of High-Pass FIR Filter using Arithmetic Optimization Algorithm and its FPGA Implementation," 2022 IEEE Region 10 Symposium (TENSYMP), Mumbai, India, 2022, pp. 1-6, doi: 10.1109/TENSYMP54529.2022.9864333.

[13] B. Kishor and K. Raj, "Design and Implementation of High Throughput and Efficient FIR Filter Architectures Using Parallel Processing," 2024 International Conference on Recent Innovation in Smart and Sustainable Technology (ICRISST), Bengaluru, India, 2024, pp. 1-6, doi: 10.1109/ICRISST59181.2024.10921940.