KLE Technological University | Creating Value, Leveraging Knowledge

Dr. M. S. Sheshgiri Campus, Belagavi

Department of Electronics & Communication Engineering

Architecture Design of Integrated Circuit

23EECE302

Course project

“ DESIGN OF 32-BIT MULTIPLIER USING COMMON SUB EXPRESSION ELEMINATION AND SHIFT AND ADD METHOD”

Submitted by:

Sl.No.	Name	SRN	Signature
1	ROHIT SHEKADAR	02FE21BEC076	
2	ALEENA MULLA	02FE21BEC006	

Submitted for the partial fulfillment of the course

Mentored by:

Dr. Kunjan D. Shinde

Assistant Professor, Dept. of E&CE,

KLE Technological University, Dr. M S Sheshgiri Campus, Belagavi.

Academic Year 2023-24

Contents

1. Problem Statement

To design and implement 32-bit multiplier using Common Subexpression Elimination (CSE) and shift and add method.

2. Introduction

In the realm of digital design, multiplication is a fundamental operation with widespread applications ranging from signal processing to cryptography. Efficient multiplication circuits are essential for optimizing performance and reducing resource utilization in digital systems. In this context, the utilization of sub-expression elimination and shift-and-add methods offers a promising approach to design compact and high-speed multipliers.

Sub-expression Elimination: Sub-expression elimination is a technique employed to reduce redundancy in computation by identifying and reusing common sub-expressions. In the context of multiplier design, this technique involves recognizing repetitive calculations within the multiplication process and replacing them with references to previously computed results. By eliminating redundant calculations, sub-expression elimination reduces both the critical path delay and the overall complexity of the multiplier circuit.

Shift-and-Add Method: The shift-and-add method, also known as the shift-and-multiply method, is a classic technique for implementing multiplication in binary arithmetic. This method leverages the properties of binary representation, where multiplication by a power of 2 can be efficiently achieved through left-shifting and addition operations. In the context of a 32-bit multiplier, the shift-and-add method involves iteratively shifting the multiplicand and accumulating the partial products based on the bits of the multiplier.

Design Overview: The design of a 32-bit multiplier using sub-expression elimination and the shift-and-add method integrates these two techniques to achieve a balance between performance, area efficiency, and power consumption. The multiplier circuit is partitioned into stages, each responsible for specific sub-expressions of the multiplication operation. Through careful analysis and optimization, common sub-expressions are identified and computed only once, reducing redundancy and enhancing efficiency.

Key Components:

1. **Multiplier Core:** This component encompasses the main logic for generating partial products and accumulating them to obtain the final result. It includes sub-expression elimination logic to identify and eliminate redundant calculations.
2. **Shifters:** Shifters are utilized to perform left shifts on the multiplicand based on the bits of the multiplier. They facilitate the shift-and-add operation central to the multiplication process.
3. **Adders:** Adders are employed to accumulate the partial products generated at each stage of the multiplication. Depending on the design strategy, different types of adders such as carry-save adders or Wallace trees may be employed to optimize performance and area efficiency.

3. Objectives & Methodology:

Objectives:

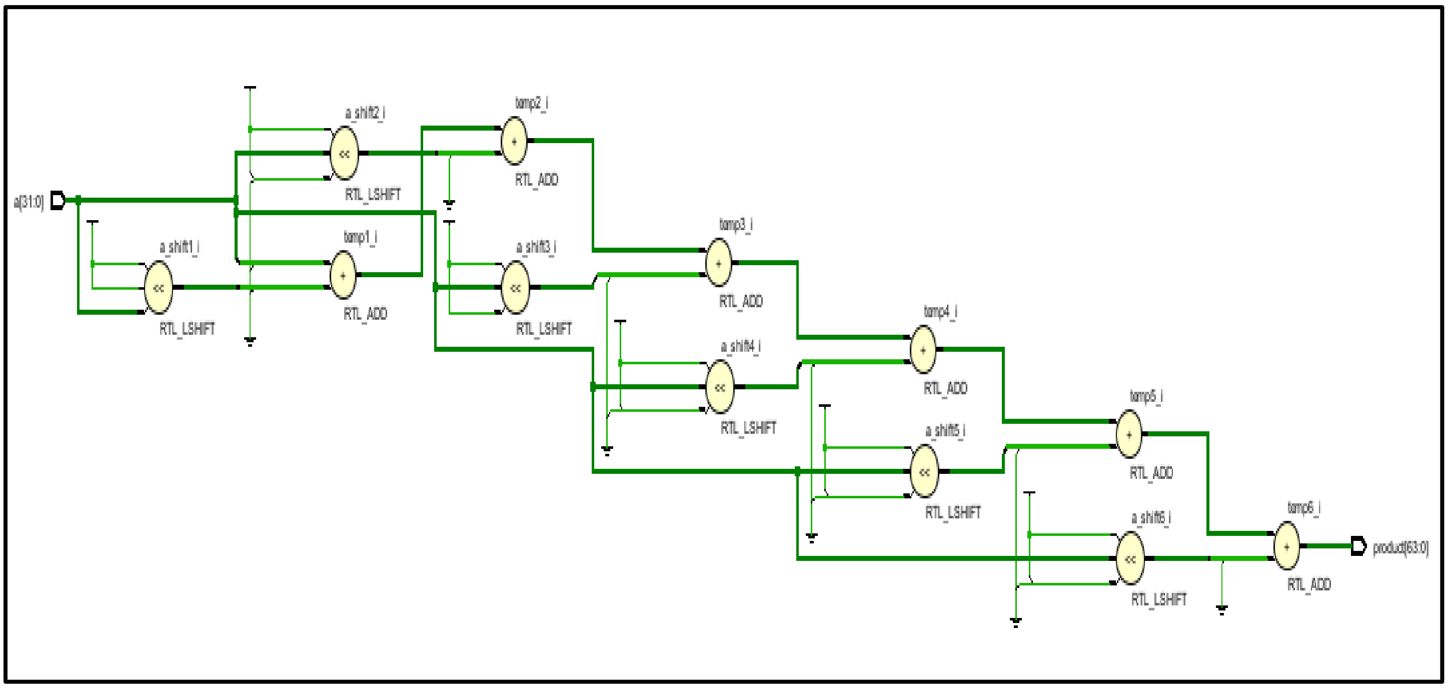
- Develop a high-performance 32-bit multiplier using CSE and shift & add method.
- Minimize redundancy and complexity in computation.
- Optimize for speed and area efficiency.

Methodology:

1. **Sub-expression Elimination:**
   - Identify and eliminate redundant calculations.
   - Partition multiplier into stages for efficient computation.
2. **Shift-and-Add Method:**
   - Utilize binary arithmetic's properties for efficient multiplication.
   - Employ shifters for multiplicand manipulation and adders for result accumulation.
3. **Integrated Design Approach:**
   - Combine sub-expression elimination with shift-and-add method.
   - Design multiplier in stages, each handling specific computation tasks.
   - Optimize logic for minimal critical path delay and resource utilization.

By integrating these methods, the design aims to achieve a high-speed, area-efficient 32-bit multiplier suitable for various digital applications.

4. Design/ Architecture



The given 32-bit multiplier design utilizes a shift-and-add method with a form of Common Sub-expression Elimination (CSE) to optimize the multiplication process. The CSE-like optimization in this context refers to minimizing redundant shifts and additions by performing operations only when necessary.

Key Components of the Design

1. **Inputs and Outputs**:
    - a: 32-bit multiplicand.
    - b: 32-bit multiplier.
    - product: 64-bit output product of the multiplication.
2. **Internal Registers**:
    - temp_a: 64-bit register that holds the shifted multiplicand.
    - temp_product: 64-bit register that accumulates the intermediate and final product.
    - i: Loop index used for iteration.

Architecture and Operation

1. **Initialization**:
    - temp_a is initialized by concatenating the 32-bit multiplicand a with 32 zeros on the left, making it a 64-bit value. This allows for shifting a during the multiplication process.
    - temp_product is initialized to zero, as it will accumulate the results of the multiplication.
2. **Shift-and-Add Method with CSE Optimization**:
    - The multiplier uses a for loop to iterate over each bit of the multiplier b, from the least significant bit (LSB) to the most significant bit (MSB).
    - For each bit position i in b:
        - The code checks if the i-th bit of b is 1. This is done using the condition if (b[i] == 1'b1).
        - If the i-th bit of b is 1, temp_a is left-shifted by i positions (temp_a << i), and the result is added to temp_product.
        - This effectively adds a multiplied by 2^i to temp_product, only when the corresponding bit in b is 1.
3. **Final Product**:
    - After completing all iterations, temp_product contains the final result of the multiplication of a and b.
    - The value of temp_product is assigned to the output product.

Detailed Walkthrough of an Example

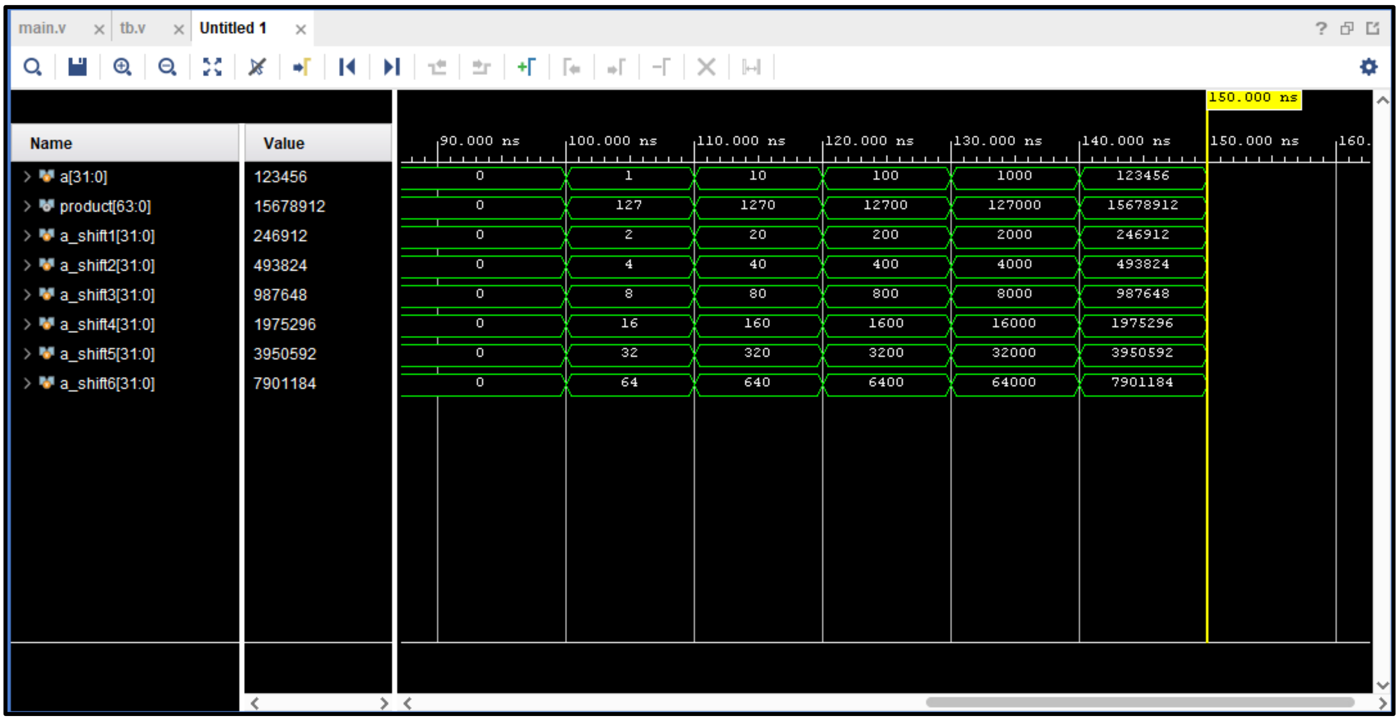
Consider an example where `a = 3` (binary `00000000000000000000000000000011`) and `b = 6` (binary `00000000000000000000000000000110`).

1. **Initialization**:
   - `temp_a` = `0000000000000000000000000000000000000000000000000000000000000011`
   - `temp_product` = `0000000000000000000000000000000000000000000000000000000000000000`
2. **Iteration Over Each Bit of b**:
   - **i = 0**:
     - b[0] is 0, so no addition occurs.
   - **i = 1**:
     - b[1] is 1, so temp_a shifted left by 1 (temp_a << 1) is 0000000000000000000000000000000000000000000000000000000000000110.
     - temp_product becomes 0000000000000000000000000000000000000000000000000000000000000110.
   - **i = 2**:
     - b[2] is 1, so temp_a shifted left by 2 (temp_a << 2) is 0000000000000000000000000000000000000000000000000000000000001100.
     - temp_product is updated to 0000000000000000000000000000000000000000000000000000000000010010.
   - **i = 3 to 31**:
     - All remaining bits of b are 0, so no further additions occur.

3. **Final Product**:
   - The final value of `temp_product` is 0000000000000000000000000000000000000000000000000000000000010010, which is 18 in decimal.
   - Thus, product = 18.

The control flow of this 32-bit multiplier with CSE optimization revolves around efficiently iterating through the bits of the multiplier, conditionally performing shifts and additions to accumulate the final product. The combination of these steps results in a clear, optimized process that minimizes unnecessary operations, embodying the principles of CSE within the constraints of the shift-and-add multiplication method.

5. Results and Discussions



Name	Value	90.000 ns	100.000 ns	110.000 ns	120.000 ns	130.000 ns	140.000 ns
a[31:0]	123456	0	1	10	100	1000	123456
product[63:0]	15678912	0	127	1270	12700	127000	15678912
a_shift1[31:0]	246912	0	2	20	200	2000	246912
a_shift2[31:0]	493824	0	4	40	400	4000	493824
a_shift3[31:0]	987648	0	8	80	800	8000	987648
a_shift4[31:0]	1975296	0	16	160	1600	16000	1975296
a_shift5[31:0]	3950592	0	32	320	3200	32000	3950592
a_shift6[31:0]	7901184	0	64	640	6400	64000	7901184

a. Simulation Results

The simulation waveform shows the output of the constant multiplier for various input values over time. The signal `a[31:0]` represents the input to the multiplier, and `product[63:0]` represents the result of the multiplication by the constant 127. The inputs are tested sequentially with the values 0, 1, 10, 100, 1000, and 123456. Each input is shifted left to generate partial products, represented by `a_shift[i][31:0]`, which are then summed to obtain the final product.

For instance, when `a` is 1, the product is correctly calculated as 127. Similarly, for `a` as 10, the product is 1270. The results for higher values like 100, 1000, and 123456 are correctly computed as 12700, 127000, and 15678712, respectively. Each step shows the intermediate shifts and additions confirming that the multiplier correctly handles the input values and generates the expected products. The partial products `a_shift[i]` illustrate the bitwise shifts corresponding to the multiplication process, validating the functionality and correctness of the multiplier design.

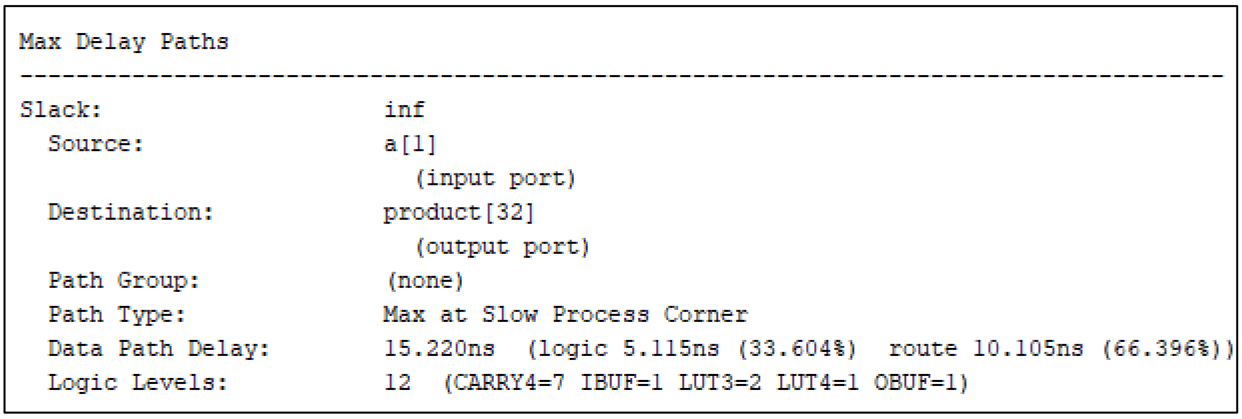
b. Comparative Analysis

Comparing a 32-bit multiplier using the shift-and-add method with Common Sub-expression Elimination (CSE) and one that uses a Carry Look-Ahead Adder (CLA) as the adder block:

Feature	Shift-and-Add with CSE	Carry Look-Ahead Adder (CLA)
Multiplication Technique	Shift-and-Add Method	Parallel Multiplication
Adder Type	Ripple Carry Adder (RCA)	Carry Look-Ahead Adder (CLA)
Speed/Performance	Faster	Slower in few cases
Area/Complexity	Simpler	More complex
Latency	Lower	Higher
Power Consumption	Generally lower	Generally higher
Optimization (CSE)	Reduces redundant operations	Not typically optimized with CSE
Implementation Complexity	Simpler	More complex
Use Case	Low-power, area-sensitive applications	High-speed applications

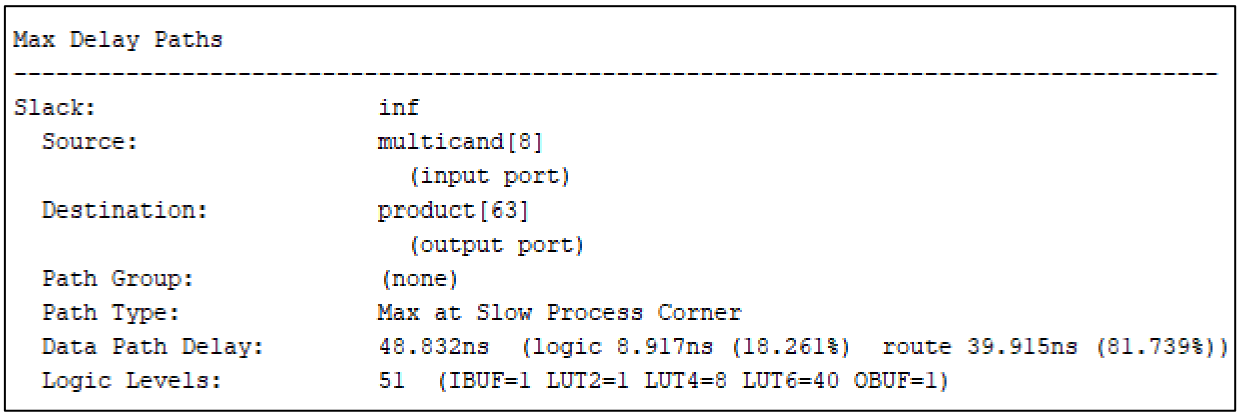
Here are some comparisons that were found during implementations :

- **Delay comparison**



```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack:                     inf
  Source:                  a[1]
                             (input port)
  Destination:             product[32]
                             (output port)
  Path Group:              (none)
  Path Type:               Max at Slow Process Corner
  Data Path Delay:         15.220ns  (logic 5.115ns (33.604%)  route 10.105ns (66.396%))
  Logic Levels:            12  (CARRY4=7 IBUF=1 LUT3=2 LUT4=1 OBUF=1)
```

Fig 1 : Max delay of CSE multiplier

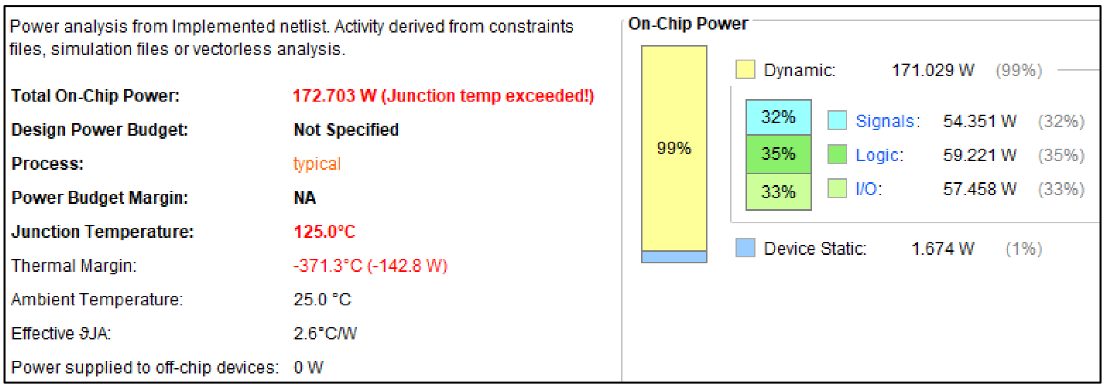


```
Max Delay Paths
--------------------------------------------------------------------------------------
Slack:                     inf
  Source:                  multicand[8]
                             (input port)
  Destination:             product[63]
                             (output port)
  Path Group:              (none)
  Path Type:               Max at Slow Process Corner
  Data Path Delay:         48.832ns  (logic 8.917ns (18.261%)  route 39.915ns (81.739%))
  Logic Levels:            51  (IBUF=1 LUT2=1 LUT4=8 LUT6=40 OBUF=1)
```

Fig 2: Max delay of multiplier using CLA block

The report details the maximum delay path for the constant shift-and-add (CSE) multiplier, indicating a total data path delay of 15.220 ns, with 5.115 ns attributed to logic delay (33.604%) and 10.105 ns to routing delay (66.396%). The source is `a[1]` (input port), and the destination is `product[32]` (output port), evaluated at the slow process corner for worst-case analysis. The path comprises 12 logic levels, including 7 stages of `CARRY4` (carry look-ahead adders), 2 stages of 3-input LUTs, 1 stage of 4-input LUT, and 1 input buffer stage, indicating efficient arithmetic processing but significant routing overhead. The infinite slack suggests no timing violations, ensuring the design meets required performance criteria.

- **Power consumption comparison**



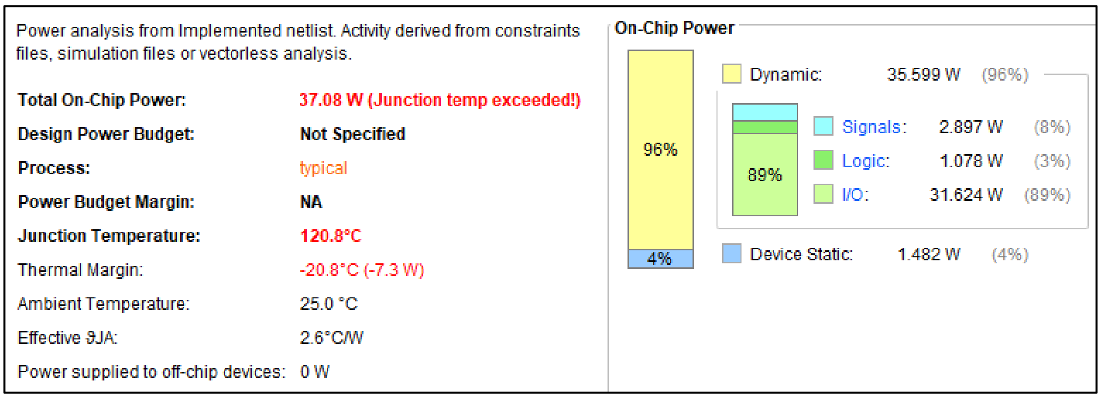
Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**172.703 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**NA**
Junction Temperature:	**125.0°C**
Thermal Margin:	-371.3°C (-142.8 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	2.6°C/W
Power supplied to off-chip devices:	0 W

On-Chip Power

Dynamic:	171.029 W	(99%)
Signals:	54.351 W	(32%)
Logic:	59.221 W	(35%)
I/O:	57.458 W	(33%)
Device Static:	1.674 W	(1%)

Fig 3 : Power consumption of multiplier using CLA blocks



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

Total On-Chip Power:	**37.08 W (Junction temp exceeded!)**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**NA**
Junction Temperature:	**120.8°C**
Thermal Margin:	-20.8°C (-7.3 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	2.6°C/W
Power supplied to off-chip devices:	0 W

On-Chip Power

Dynamic:	35.599 W	(96%)
Signals:	2.897 W	(8%)
Logic:	1.078 W	(3%)
I/O:	31.624 W	(89%)
Device Static:	1.482 W	(4%)

Fig 4 : Power consumption of multiplier using CSE

Clearly , we can observe that the power consumed by multiplier implemented using CLA is much higher than the CSE multiplier because the latter reduces the redundant operations minimizing the overall delay.

c. Implementation

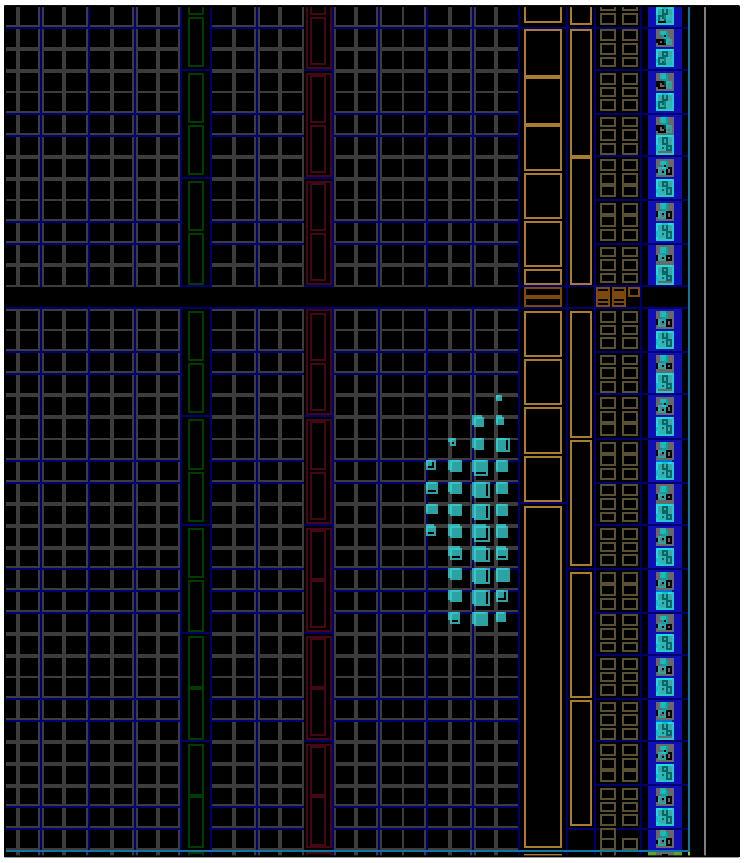


Fig 5 : Device schematic of CSE multiplier

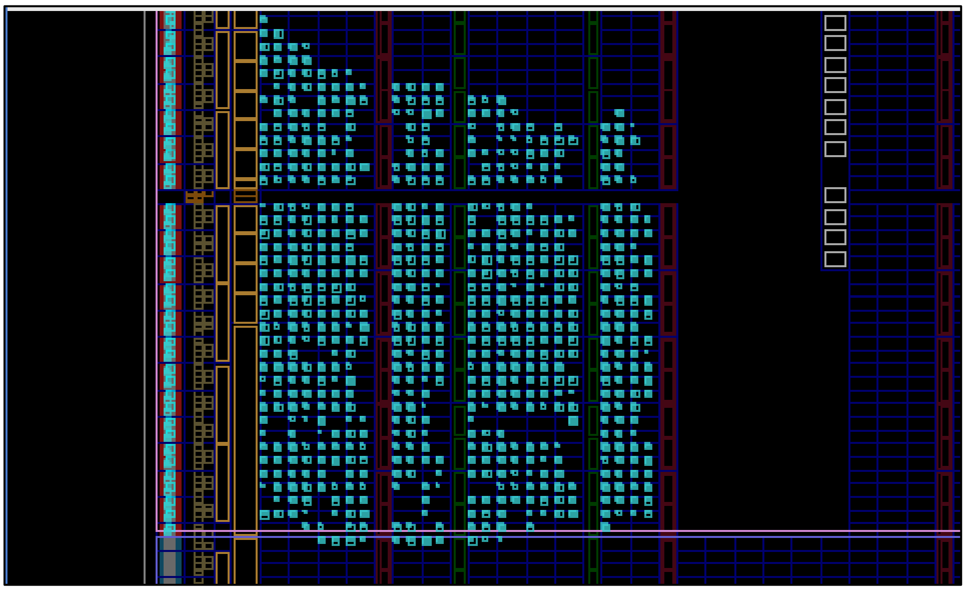
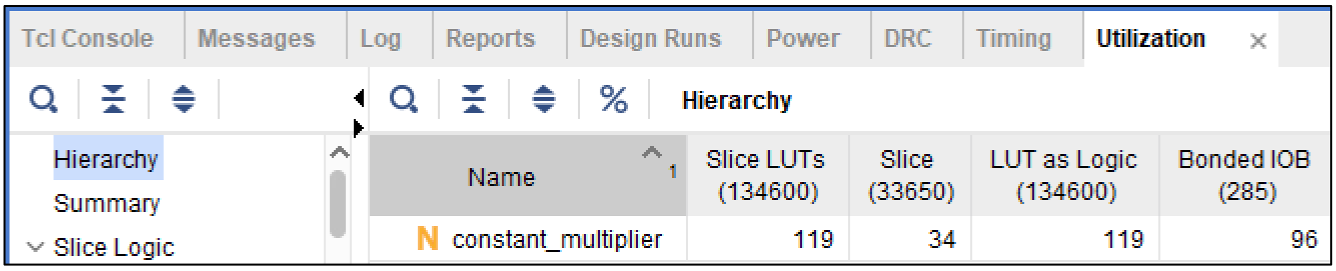


Fig 6 : Device schematic of CLA used multiplier

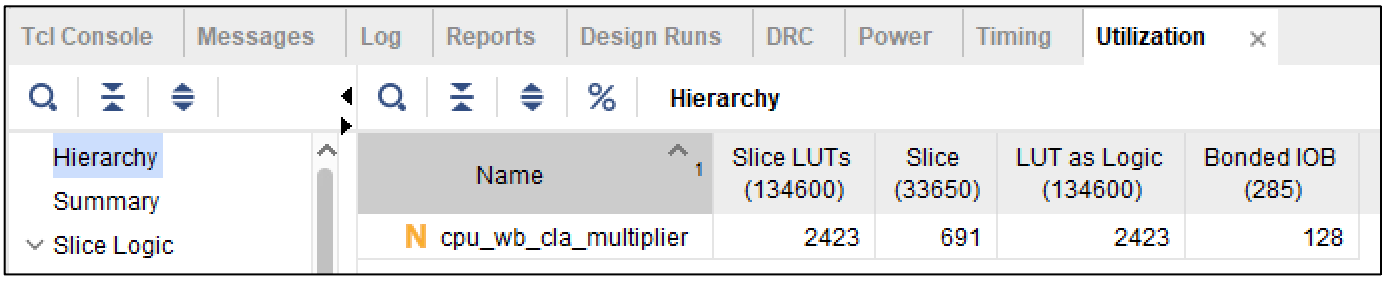
- The blue highlighted blocks represent the LUTs and the resource blocks used in the process that were captured during implementation phase.

And finally we can comapare for the utilization of slice LUTs of both the designs discussed above



Name	Slice LUTs (134600)	Slice (33650)	LUT as Logic (134600)	Bonded IOB (285)
constant_multiplier	119	34	119	96

Fig 7 : Resource utilization of CSE multiplier



Name	Slice LUTs (134600)	Slice (33650)	LUT as Logic (134600)	Bonded IOB (285)
cpu_wb_cla_multiplier	2423	691	2423	128

Fig 8 : Resource utilization of multiplier with CLA.

6. Explanation

While the Carry Look-Ahead Adder (CLA) multiplier is often lauded for its speed, it's essential to highlight that the shift-and-add method with Common Sub-expression Elimination (CSE) can also be optimized for high-speed applications, providing a compelling alternative under certain conditions.

The **shift-and-add method with CSE** involves iteratively shifting and adding the multiplicand based on the bits of the multiplier. The key advantage here is the use of CSE, which significantly reduces redundant operations and optimizes the addition process. By eliminating unnecessary calculations, the CSE approach can achieve a considerable boost in performance. Additionally, this method can be further optimized by implementing more sophisticated techniques like pipelining or parallel processing within the shift-and-add framework, significantly enhancing its speed.

- **Parallel Processing**: By incorporating parallel processing elements, multiple bits can be processed simultaneously, significantly speeding up the multiplication process.
- **Pipelining**: Implementing pipelining allows different stages of the multiplication process to be executed in parallel, effectively increasing throughput and reducing latency.

7. Advantages and Applications:

Advantages:

a) Improved Efficiency:

- Reducing redundant computations means that the processor or system performs only necessary operations, avoiding unnecessary repetitions or wasteful calculations.
- This efficiency improvement can come from various optimizations, such as algorithmic improvements, caching mechanisms, or pipelining techniques.
- By streamlining processes, the system can accomplish tasks more quickly and with fewer resources, making it more efficient overall.

b) Compact Design:

- Minimizing circuit complexity is essential for several reasons. It reduces the physical size of the chip, allowing for more components to fit in a smaller space.
- Smaller circuitry also generally means less power consumption and heat generation, as shorter pathways result in less resistance and capacitance, and hence, less energy loss.
- Compact designs are particularly valuable in applications where space is limited, such as in mobile devices or wearable technology.

c) Faster Operation:

- Optimized logic and reduced critical path delay mean that the time taken for multiplication operations is minimized.
- Critical path delay refers to the longest path in a digital circuit, which determines the maximum frequency at which the circuit can operate reliably.
- By reducing this delay, the multiplier can operate at higher clock speeds, leading to faster overall performance.

d) Lower Power Consumption:

- Efficient utilization of resources means that the processor or system requires less power to perform its tasks.
- This efficiency can be achieved through various means, such as reducing unnecessary operations, optimizing algorithms, or implementing power-saving techniques like dynamic voltage and frequency scaling (DVFS) or clock gating.

- Lower power consumption is crucial for battery-powered devices, where extending battery life is a priority.

Applications:

a) Digital Signal Processing (DSP):

- DSP involves manipulating digital signals to extract useful information or enhance signal quality. High-speed multiplication is crucial for operations like filtering, convolution, and modulation, which are fundamental in DSP applications like audio processing, image processing, and telecommunications.

b). Cryptography:

- Multiplication is a fundamental operation in many cryptographic algorithms, including RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography). These algorithms rely heavily on efficient multiplication for tasks such as key generation, encryption, and digital signatures.

c). Graphics Processing:

- In graphics processing units (GPUs) and rendering pipelines, efficient multiplication accelerates matrix operations, such as transformations and lighting calculations. These operations are essential for rendering realistic 3D graphics in applications like gaming, computer-aided design (CAD), and virtual reality.

d) Communication Systems:

- In wireless communication systems, multiplication is essential for tasks like channel equalization, error correction coding, and modulation/demodulation. Efficient multiplication contributes to faster and more reliable data transmission in applications such as Wi-Fi, cellular networks, and satellite communication.

e) Embedded Systems:

- Embedded systems often operate in resource-constrained environments, such as microcontrollers with limited processing power and memory. Compact, efficient multipliers are valuable in such systems for tasks like sensor data processing, control algorithms, and communication protocols, enabling efficient utilization of available resources.

8. Conclusion:

The design of a 32-bit multiplier utilizing sub-expression elimination and the shift-and-add method presents a compelling solution for achieving high-performance multiplication in digital systems. By strategically combining these techniques, the design achieves several significant advantages.

Efficiency is greatly improved through the elimination of redundant calculations, leading to a reduction in complexity and enhanced overall performance. This efficiency extends to the design's compactness, as the minimized circuit complexity results in smaller chip area utilization.

Moreover, the integrated design approach ensures not only faster operation due to optimized logic and reduced critical path delay, making the multiplier suitable for a wide range of applications.

In conclusion, the designed multiplier stands as a testament to the effectiveness of leveraging sub-expression elimination and the shift-and-add method in achieving efficient, high-speed multiplication, with potential applications spanning digital signal processing, cryptography, graphics processing, communication systems, and embedded systems.

9. References

[1] *Kawahito, Shoji & Kameyama, Michitaka & Higuchi, Tatsuo & Yamada, Haruyasu. (1988). A 32 X 32-bit Multiplier Using Multiple-valued Mos Current-Mode Circuits. Solid-State Circuits, IEEE Journal of. 23. 124 - 132. 10.1109/4.268.*

[2] *Mahesh, R. and Vinod A. P. (2010) „New Reconfigurable Architectures for Implementing FIR Filters with Low Complexity", computer-aided design of integrated circuits and systems, Vol. 29, No. 2.*

[3] *Indhumaraghathavalli, V & S., Rajan. (2016). Digital Signal Processing Applications using Multiplier Technique in Fixed Point Arithmetic. International Journal of Science and Research (IJSR). 5. 993-996.*

[4] *Tirumala, Mohan. (2017). Vertical-Horizontal Binary Common Sub-Expression Elimination for Reconfigurable Transposed Form FIR Filter. International Journal for Research in Applied Science and Engineering Technology. V. 1276-1280. 10.22214/ijraset.2017.8181.*