Lab Report: 25-bit Adder and Multiplier Design

6338216 Senquan Zhang

6417108 Zhihao Fu

6433812 Bernardo Capoferri

Content

1. Introduction

This report is about the workflow of our design and evaluation of three 25-bit adders and three 25-bit multipliers. In the following chapters.

2. Metric selection and algorithm theories

In this chapter, the metrics for our adders and multipliers selections will be declared. The theories of the algorithms we choose will be introduced.

2.1 Multilevel Carry Skip Adder

2.1.1 Multilevel Carry Skip Metric

In this work, the adder architecture is evaluated in terms of performance (delay) and energy consumption. To capture the trade-off between these two metrics, we use the Energy–Delay Product (EDP), defined as:

$$\mathrm{EDP} = \mathrm{Energy} \times \mathrm{Delay}$$

$$\mathrm{Delay} = \mathrm{T_{req}} - \mathrm{WNS}$$

The multilevel carry-skip adder was selected because it provides a balanced trade-off between computation speed and energy efficiency. Compared to a pure ripple-carry adder, it reduces the worst-case carry

propagation delay by allowing selected carry signals to bypass multiple blocks. At the same time, it avoids the large power and area overhead associated with full carry-lookahead or prefix adders. As a result, the multilevel carry-skip architecture achieves a lower Energy–Delay Product (EDP) for medium word lengths, making it a suitable choice for balanced performance-energy optimization.

2.1.2 Multilevel Carry Skip Theory

This design accelerates addition by combining local ripple-carry computation with block-level and group-level carry skipping. This reduces the worst-case propagation delay compared to a pure ripple-carry adder.

1. 5-Bit Basic Block

The smallest computational unit is a 5-bit ripple-carry adder. Each block performs two functions:

1. Local sum and carry computation using ripple-carry.
2. Block propagate detection, where each bit generates a propagate signal

$$p_i = A_i \oplus B_i$$

and the block propagate is

$$P_{\text{block}} = \bigwedge_{i=0}^{4} p_i$$

If $P_{\text{block}} = 1$, the block will not block the incoming carry.

2. Intermediate Structure: Block Groups and Skip Logic Multiple 5-bit blocks are grouped to form larger adders:

- Group0 (10 bits): composed of 2 blocks
- Group1 (15 bits): composed of 3 blocks

Within each group, two levels of skip logic are applied:

(a) Block-Level Skip Logic (Level-1)

Between adjacent 5-bit blocks, the carry-out is selected as:

$$c_{k+1} = \begin{cases} c_k, & \text{if } P_{\text{block}} = 1 \\ \text{cout}_{\text{ripple}}, & \text{otherwise} \end{cases}$$

This allows the carry to bypass intermediate ripple computation when possible.

(b) Group-Level Skip Logic (Level-2)

Each group computes a group propagate signal:

$$P_{\text{group}} = \wedge P_{\text{block}}$$

The group carry-out is then selected as:

$$C_{\text{out}} = \begin{cases} C_{\text{in}}, & \text{if } P_{\text{group}} = 1 \\ \text{cout}_{\text{ripple}}, & \text{otherwise} \end{cases}$$

This enables carry skipping across an entire 10-bit or 15-bit region.

3. Top-Level 25-Bit Adder Integration

The final 25-bit adder is composed as:

- Lower 10 bits → Group0 (two 5-bit blocks)
- Upper 15 bits → Group1 (three 5-bit blocks)

The top-level input carry is fixed to zero, and the group carry-out of Group0 feeds the input carry of Group1.

Finally, signed overflow is detected using the standard two's-complement rule:

$$\text{overflow} = (A_{24} \wedge B_{24} \wedge \neg S_{24}) \vee (\neg A_{24} \wedge \neg B_{24} \wedge S_{24})$$

2.2 Kogge-Stone Adder

2.2.1 KSA Metric

In this design, we aimed to find an algorithm that achieves maximum speed and offers an outlined advantage in delay performance. Therefore, during the algorithm selection stage, we adopted a single evaluation metric: minimizing delay.

Through literature review and comparison, we found that the Kogge–Stone Adder (KSA) uses a parallel prefix carry-propagation structure, and has the fastest possible implementation of a parallel prefix computation if only two-input blocks are allowed. With the delay

of only $log_2 k$ levels.

As a result, we selected the Kogge–Stone algorithm as the implementation for this design, where delay is the only metric considered.

2.2.2 KSA theory

A 25-bit Kogge–Stone Adder (KSA) is a parallel-prefix adder that

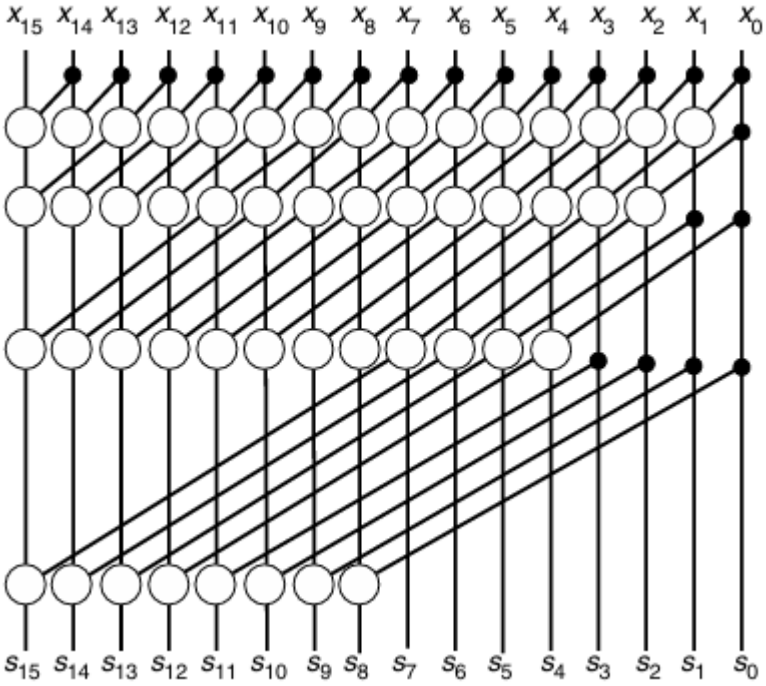


Figure 1 Kogge–Stone parallel prefix graph for 16 inputs

accelerates addition by computing carries in a highly parallel. Instead of letting the carry ripple through all 25 bits, KSA forms carry information over progressively larger bit ranges, which reduces the carry-computation depth to approximately $\lceil \log_2 25 \rceil = 5$ logic levels.

(1) Generate and Propagate signals.

For each bit $i \in [0,24]$, KSA defines:

$$g_i = A_i \wedge B_i, p_i = A_i \oplus B_i$$

where g_i indicates that bit i produces a carry regardless of the input carry, and p_i indicates whether bit i propagates an incoming carry.

(2) Prefix combination.

KSA uses an associative prefix operator to combine adjacent ranges of bits. For a "high" range and a "low" range, the combined pair is:

$$G = G_{\text{high}} \vee \left(P_{\text{high}} \wedge G_{\text{low}}\right), P = P_{\text{high}} \wedge P_{\text{low}}$$

This allows carry behavior to be summarized over larger intervals $\left(G_{i:j}, P_{i:j}\right)$, where $G_{i:j}$indicates whether the block from bit j to bit i generates a carry, and $P_{i:j}$ indicates whether it propagates a carry.

(3) Logarithmic-depth carry tree.

For 25 bits, the prefix network expands the span by powers of two (1, 2, 4, 8, 16). After five stages, each bit position $i$has the group-generate term $G_{i:0}$, which determines the carry into bit $i+1$when the external carry-in is zero:

$$C_0 = 0, C_{i+1} = G_{i:0}$$

(4) Sum computation.

Once carries are available, the sum bits are produced as:

$$S_i = p_i \oplus C_i, i = 0, \ldots, 24$$

In summary, the 25-bit KSA achieves low carry delay by using a five-level parallel-prefix carry tree, which is its main advantage for high-speed addition.

2.3 Hybrid Adder

2.3.1 Hybrid Adder Metric

In this adder we strive to achieve the best performance per area, the metric to be used will be the Worst Negative Slack (WNS), total Look Up Tables used (LUTs) and number of registers used. To evaluate the final design, we will calculate the product of the area (LUTs + registers) to the timing results, giving us the Area Delay Product (ADP) giving us the following formulas:

$$Delay = Treq - WNS$$

$$Area = LUTs + registers$$

$$ADP = Delay\ x\ Area$$

2.3.2 Hybrid Adder theory

For the adder a hybrid between the Carry-Lookahead / Carry-Select methods were chosen as it gave the best performance per logic gate utilized.

While a Carry-Select adder (CSA) by itself is quite extensive as it

must calculate the results for the carry being either 0 or 1, by hybridizing with a Carry-Lookahead its possible to eliminate long carry chains while maintaining the area to a reasonable size.

2.4 Radix-4 Booth with Wallace tree and KSA

2.4.1 Radix-4 Booth with Wallace tree and KSA Metric

In this multiplier design, the objective is to minimize propagation delay. Under this metric, we prioritize solutions that minimize the number of partial-product rows, reduce the number of sequential accumulation steps, and use a fast final carry-propagation stage.

Based on propagation-delay comparisons of multiplier structures, we selected a Radix-4 Booth + Wallace reduction tree + Kogge–Stone Adder (KSA) implementation. Radix-4 Booth recoding reduces the number of partial-product rows, while the Wallace reduction tree ，which in the book was declared as fastest possible design, provides a low-depth, highly parallel accumulation of these rows into two final numbers，sum and carry. These two numbers are then added using a Kogge–Stone adder, which we previously selected for its low delay metric.

As a result, the overall multiplier delay is dominated by a small number of parallel reduction layers plus a fast final adder stage, which matches our single design metric of minimizing delay.

2.4.2 Radix-4 Booth with Wallace tree and KSA theory

(1) Radix-4 Booth recording

A radix-4 Modified Booth recoding multiplier reduces multiplication delay by decreasing the number of partial-product rows. Instead of generating one partial product per multiplier bit, radix-4 recoding processes the multiplier in consecutive 3-bit segments overlapping in

1 bit and produces signed digits. Thus, for k-bit binary numbers in 2's-complement format, the Booth-encoded radix-4 version will have ⌈k/2⌉digits.

x_{i+1}	x_i	x_{i-1}	y_{i+1}	y_i	$z_{i/2}$	Explanation
0	0	0	0	0	0	No string of 1s in sight
0	0	1	0	1	1	End of a string of 1s in x
0	1	0	1	−1	1	Isolated 1 in x
0	1	1	1	0	2	End of a string of 1s in x
1	0	0	−1	0	−2	Beginning of a String of 1s in x
1	0	1	−1	1	−1	End one string, begin new string
1	1	0	0	−1	−1	Beginning of a string of 1s in x
1	1	1	0	0	0	Continuation of string of 1s in x

Figure 2 Radix-4 Booth's recoding yielding

In our design, the multiplier operand B is recoded from 25 bits into 13 Booth digits. Each overlapping 3-bit group is mapped to a signed recode digit.

```
case ({y2,y1,y0})
    3'b000, 3'b111: booth4_code =  0;
    3'b001, 3'b010: booth4_code =  1;
    3'b011:         booth4_code =  2;
    3'b100:         booth4_code = -2;
    3'b101, 3'b110: booth4_code = -1;
    default:        booth4_code =  0;
endcase
```

Figure 3 Radix-4 Booth code map

For each group k, a partial product is formed from the multiplicand A using only the multiples $\{0, \pm A, \pm 2A\}$:

$$PPk = (booth4\_code_k \cdot A) \ll (2k)$$

The '$\ll (2k)$' is used to align the partial product to the correct weight. After recording, the number of partial product rows is reduced to 13, compared to 25 rows in a design without Radix-4 Booth recoding. As a result, fewer partial product rows are presented to the reduction tree, which reduces the amount of accumulation and compression required. This helps lower the overall propagation delay.

(2) Wallace reduction tree

Wallace reduction tree is a kind of full-tree multipliers architectures where all k multiples of the multiplicand are produced at once, and a k-input carry-save adder (CSA) tree is used to reduce them to two operands for the final addition. Following this concept, after Radix-4 Booth recoding, our multiplier produces 13 aligned partial-product rows, which must be summed to obtain the final product. A Wallace reduction tree is adopted to minimize

propagation delay by reducing the number of rows in a highly parallel manner, rather than accumulating them sequentially.

Wallace's strategy for building CSA trees is to combine the partial-product bits at the earliest opportunity. The Wallace tree is built from 3:2 compressors. In each compression step, three partial-product rows are combined into two rows: a sum row and a carry row. The carry row represents carries that are shifted one bit to a higher position, which helps keep the reduction depth small and avoids unnecessary sequential accumulation.

In our design, the 13 Booth recoded partial product rows are compressed through multiple stages until only two final rows remain, denoted as S_row and C_row. These two rows are then resolved by a single final adder. The Wallace reduction tree reduces the overall delay by avoiding long carry-propagation during intermediate summations.

2.5 Radix-4 Booth with Row-wise Binary Adder Tree

2.5.1 Radix-4 Booth with Row-wise Binary Adder Tree Metrics

In this work, the multiplier architecture is evaluated in terms of performance (delay) and energy consumption. To capture the trade-off between these two metrics, we use the Energy–Delay Product (EDP), defined as:

$$\mathrm{EDP} = \mathrm{Energy} \times \mathrm{Delay}$$

$$\mathrm{Delay} = \mathrm{T_{req}} - \mathrm{WNS}$$

The Radix-4 Booth with Row-wise Binary Adder was selected. This structure provides a practical delay–energy trade-off: the binary adder tree reduces accumulation delay from linear to logarithmic complexity, while avoiding the high hardware and switching overhead of fully CSA-based Wallace or Dadda trees. As a result, the design achieves moderate delay with lower energy consumption.

2.5.2 Radix-4 Booth with Row-wise Binary Adder Tree Theory

The theory of Radix-4 Booth can be seen in 2.4.

After radix-4 Modified Booth recoding, the multiplier generates 13 aligned partial-product rows, each already shifted to its correct bit position. If these 13 rows were accumulated sequentially with a single adder, the critical path would contain 12 large additions, leading to a long propagation delay. To improve performance, the design adopts a tree-style reduction structure, where multiple partial products are summed in parallel over several levels.

In this implementation, all partial products have a uniform width of 50 bits and are stored in the array $\mathrm{pp}[0..12]$. The reduction proceeds in several stages:

- **First level (13 to 7 rows):**

The partial products are added in pairs:

$$\begin{aligned} s_0 &= \mathrm{pp}_0 + \mathrm{pp}_1, \\ s_1 &= \mathrm{pp}_2 + \mathrm{pp}_3, \\ s_2 &= \mathrm{pp}_4 + \mathrm{pp}_5, \\ s_3 &= \mathrm{pp}_6 + \mathrm{pp}_7, \\ s_4 &= \mathrm{pp}_8 + \mathrm{pp}_9, \\ s_5 &= \mathrm{pp}_{10} + \mathrm{pp}_{11}, \end{aligned}$$

while $\mathrm{pp}_{12}$is carried to the next level unchanged. This reduces 13 rows to 7 rows.

- **Second level (7 to 4 rows):**

The intermediate sums are further combined:

$$\begin{aligned} t_0 &= s_0 + s_1, \\ t_1 &= s_2 + s_3, \\ t_2 &= s_4 + s_5, \end{aligned}$$

together with the remaining pp_{12}. At this point, the number of rows is reduced to 4.

- **Third level (4 to 2 rows):**

The four rows are compressed into two:

$$\begin{aligned} u_0 &= t_0 + t_1, \\ u_1 &= t_2 + \mathrm{pp}_{12} \end{aligned}$$

- **Final level (2 to 1 row):**

Finally, the two remaining rows are added to obtain the 50-bit product:

$$\text{prod} = u_0 + u_1.$$

Conceptually, each "+" corresponds to a 50-bit signed binary adder. Although the RTL description uses standard two-operand additions, the structure of the reduction is tree-like: the number of operand rows is roughly halved at each level. This leads to a computation depth that grows approximately with $\log_2(N)$ rather than linearly with N, where N is the number of original partial products. As a result, the critical path delay of the accumulation stage is significantly reduced compared to a simple linear adder chain, while fully preserving arithmetic correctness.

2.6 Radix-4 Booth with Dadda tree

2.6.1 Radix-4 Booth with Dadda tree Metric

For this design, the desired result was the product of the delay times the area, with a lower value for both being desired. For this purpose, a multiplier design utilizing a Radix-4 Booth encoding, Dadda tree with a hybrid adder was chosen. The Booth encoding was chosen due to it shrinking the number of partial row products, while the Dadda design was chosen due to it reducing the complexity of the CSA trees most of the time relative to Wallace. The Hybrid adder was chosen due to the same reasons as given in section 2.3 of this paper.

2.6.2 Radix-4 Booth with Dadda tree Theory

As given in Behrooz Parhami's book, Dadda's method prioritizes combining the partial products as late as possible, by reducing the number of operands to the next lower value, at the same time as keeping the critical path of the CSA tree to a minimum, this often leads to a reduction in the height of the tree as well as the numbers of Full Adders and Half-adders when compared to Wallce's method.

3. Results

3.1 Time constraints

The time constraints for the design, the clock period set in the TimeConstraints.xdc file, are recorded as follows:

• Clock Period:

Adders: 2 ns (0.5 GHz frequency)

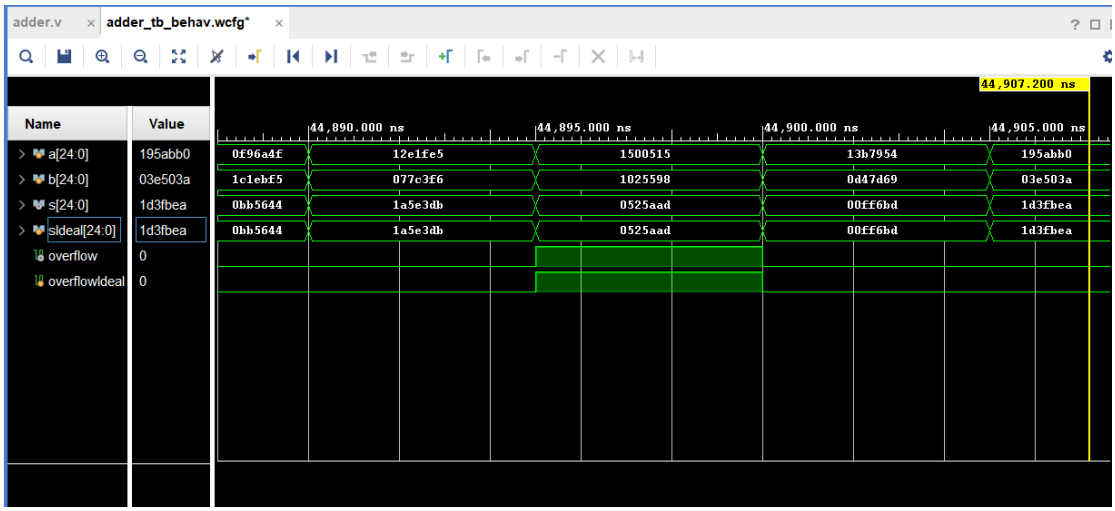
Radix-4 Booth with Wallace tree and KSA: 3.4 ns (294.118 MHz frequency)

Radix-4 Booth with Radix-4 Booth with Row-wise Binary Adder Tree: 3.7 ns 270.772 MHz frequency)

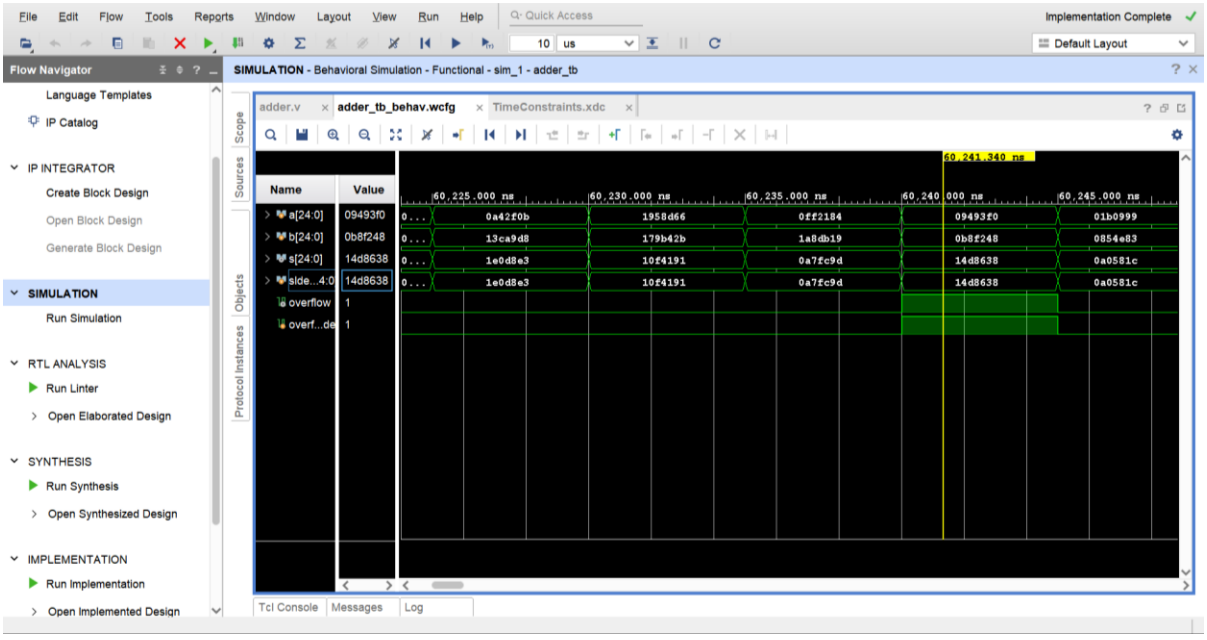
Radix-4 Booth with Dadda tree: 4.1 ns 243.902 MHz.

3.2 Testbench verification

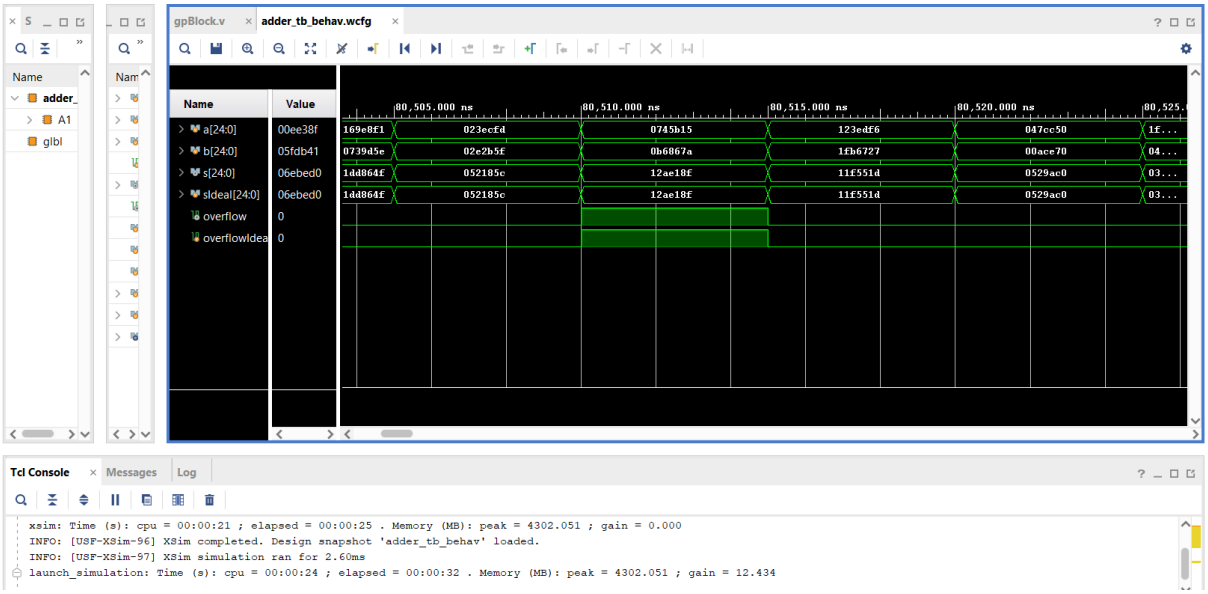
3.2.1 Multilevel Carry Skip Adder



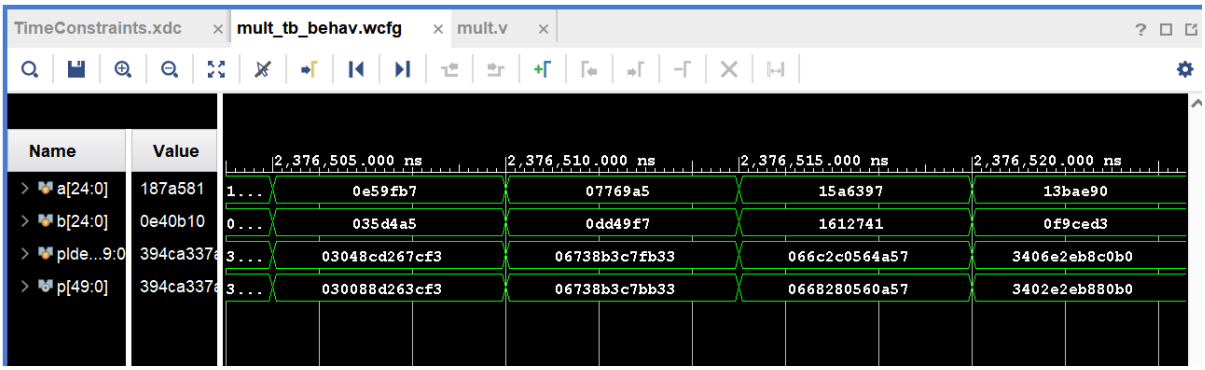
3.2.2 Kogge-stone adder



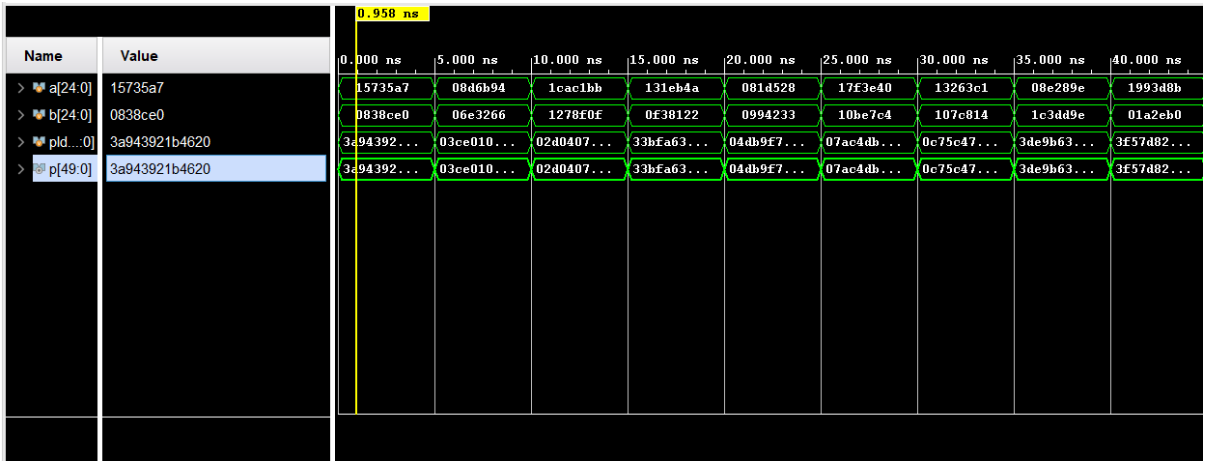
3.2.3 Hybrid CLA/CSLA Adder



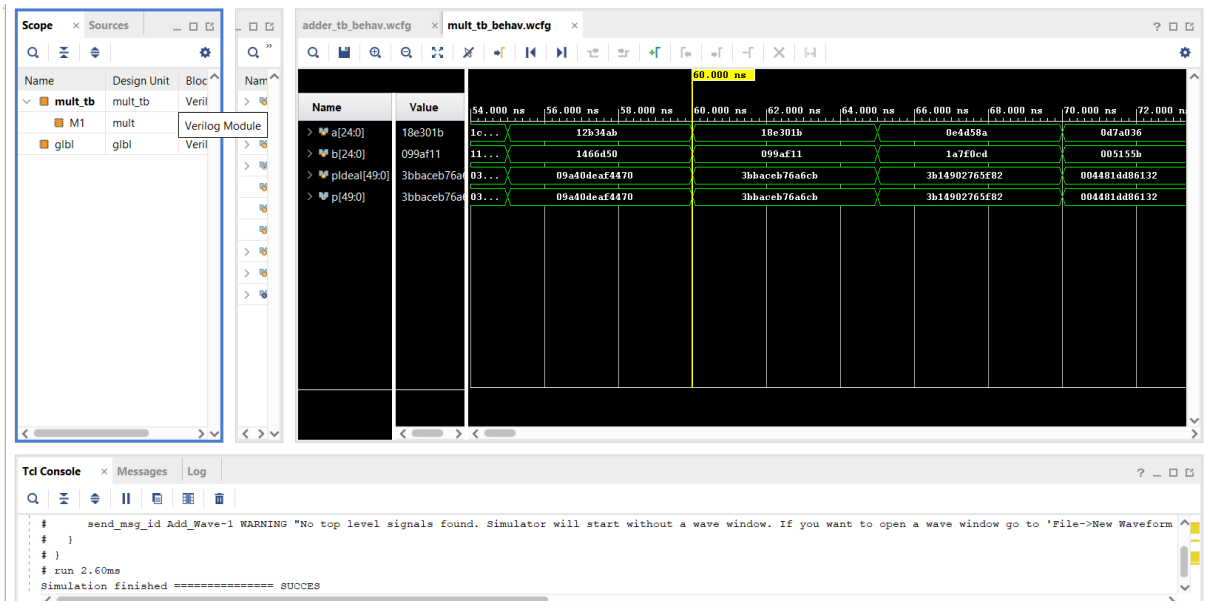
3.2.4 Radix-4 Booth with Wallace tree and KSA



3.2.5 Radix-4 Booth with Row-wise Binary Adder Tree

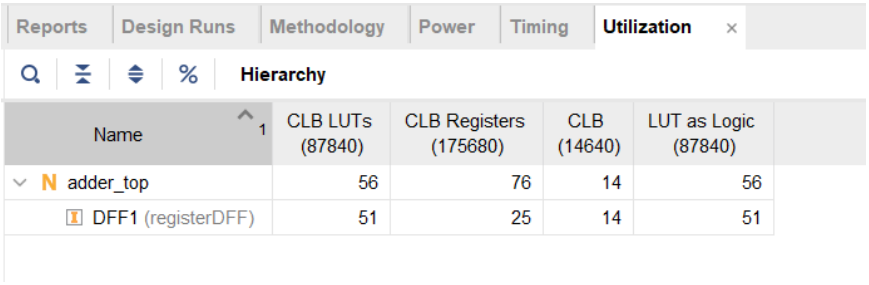


3.2.6 Radix-4 Booth with Dadda tree and Hybrid Adder



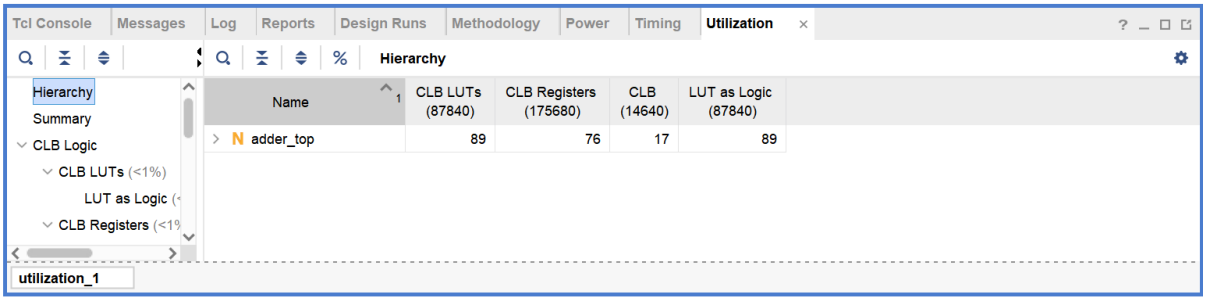
3.3 Area

3.3.1 Multilevel Carry Skip Adder



Name	CLB LUTs (87840)	CLB Registers (175680)	CLB (14640)	LUT as Logic (87840)
adder_top	56	76	14	56
DFF1 (registerDFF)	51	25	14	51

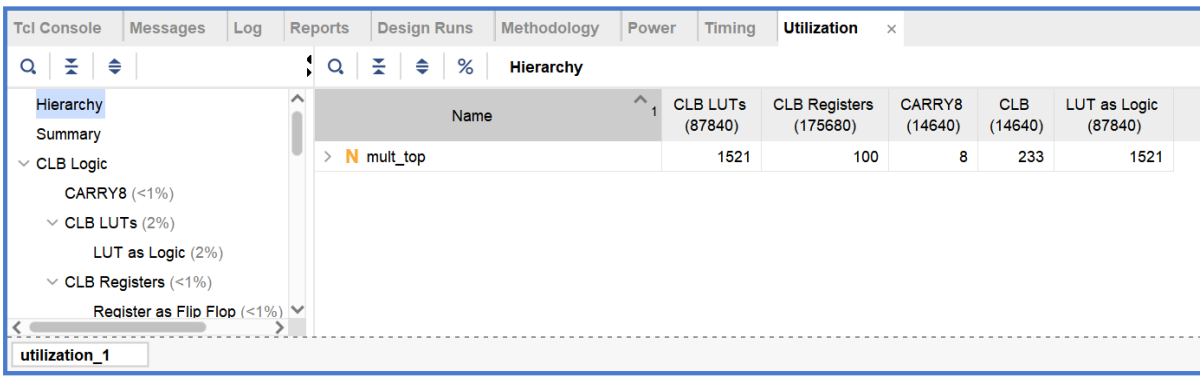
3.3.2 Kogge-stone adder



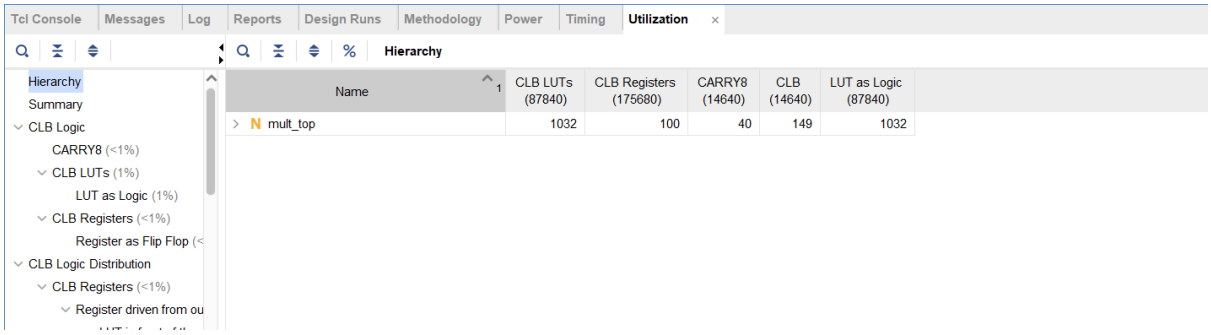
3.3.3 Hybrid CLA/CSLA Adder

Name	CLB LUTs (87840)	CLB Registers (175680)	CLB (14640)	LUT as Logic (87840)
adder_top	64	76	13	64
DFF1 (registerDFF)	59	25	13	59

3.3.4 Radix-4 Booth with Wallace tree and KSA



3.3.5 Radix-4 Booth with Row-wise Binary Adder Tree



3.3.6 Radix-4 Booth with Dadda tree and Hybrid Adder

Name	CLB LUTs (87840)	CLB Registers (175680)	CARRY8 (14640)	CLB (14640)	LUT as Logic (87840)
mult_top	1567	102	172	227	1567
BOOTH (booth_encoder)	1567	0	172	223	1567

3.4 Delay

3.4.1 Multilevel Carry Skip Adder

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.402 ns	Worst Hold Slack (WHS):	0.075 ns	Worst Pulse Width Slack (WPWS):	0.225 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	26	Total Number of Endpoints:	26	Total Number of Endpoints:	76

All user specified timing constraints are met.

3.4.2 Kogge-stone adder

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.466 ns	Worst Hold Slack (WHS):	0.066 ns	Worst Pulse Width Slack (WPWS):	0.225 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	26	Total Number of Endpoints:	26	Total Number of Endpoints:	76

All user specified timing constraints are met.

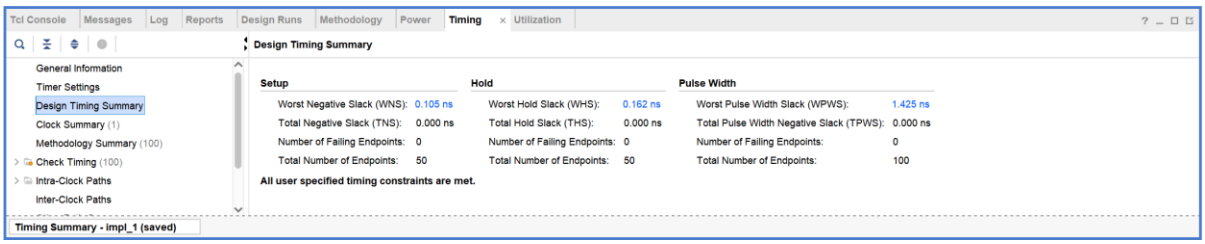
3.4.3 Hybrid CLA/CSLA Adder

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.451 ns	Worst Hold Slack (WHS):	0.055 ns	Worst Pulse Width Slack (WPWS):	0.225 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	26	Total Number of Endpoints:	26	Total Number of Endpoints:	76

All user specified timing constraints are met.

3.4.4 Radix-4 Booth with Wallace tree and KSA



3.4.5 Radix-4 Booth with Row-wise Binary Adder Tree

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.115 ns	Worst Hold Slack (WHS):	0.166 ns	Worst Pulse Width Slack (WPWS):	0.975 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	50	Total Number of Endpoints:	50	Total Number of Endpoints:	100

All user specified timing constraints are met.

3.4.6 Radix-4 Booth with Dadda tree and Hybrid Adder

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.449 ns	Worst Hold Slack (WHS):	0.423 ns	Worst Pulse Width Slack (WPWS):	1.775 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	50	Total Number of Endpoints:	50	Total Number of Endpoints:	102

All user specified timing constraints are met.

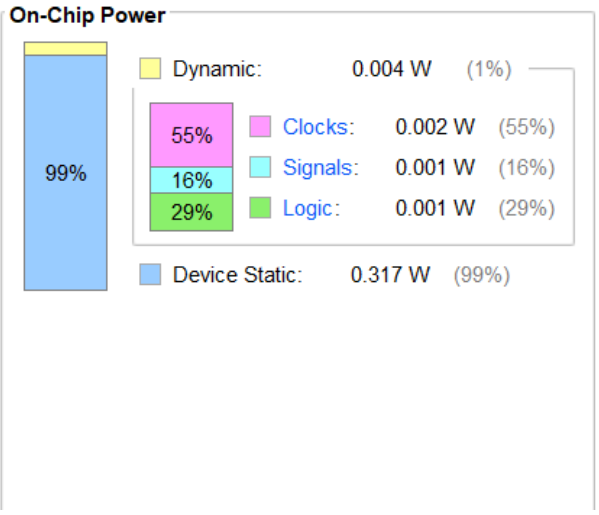
3.5 Power

3.5.1 Multilevel Carry Skip Adder

Summary

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

Total On-Chip Power:	**0.321 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	74.6°C (53.7 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

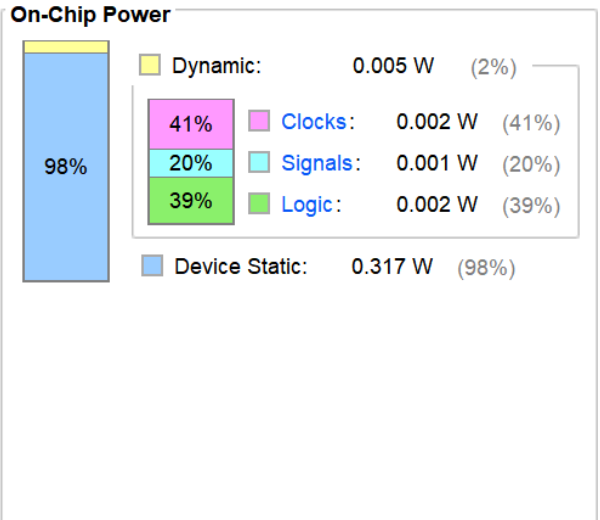


3.5.2 Kogge-stone adder

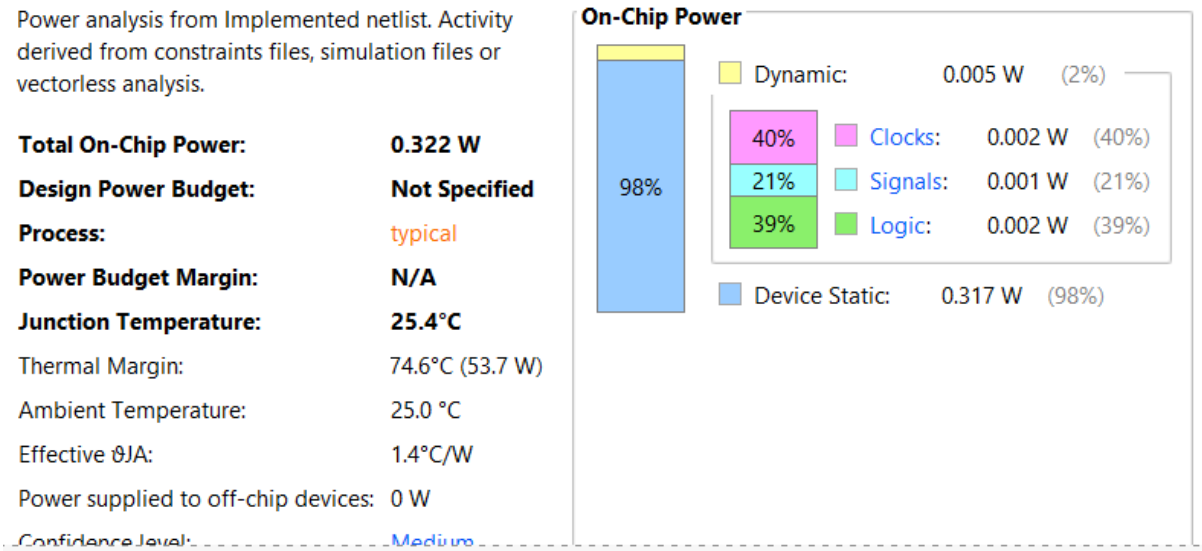
Summary

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

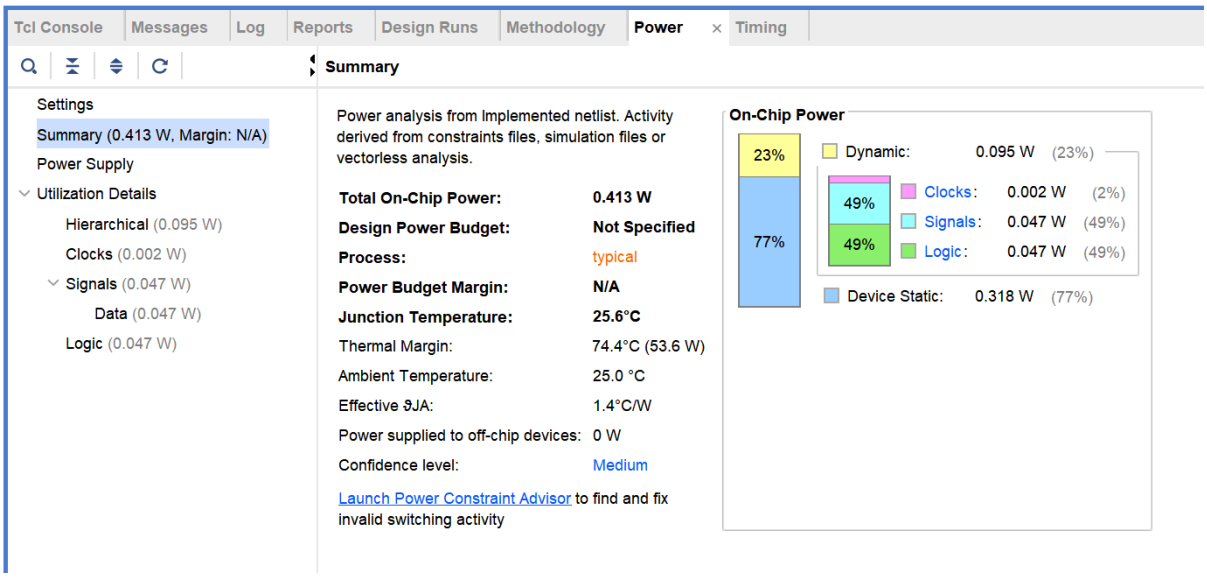
Total On-Chip Power:	**0.322 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.4°C**
Thermal Margin:	74.6°C (53.7 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium



3.5.3 Hybrid CLA/CSLA Adder



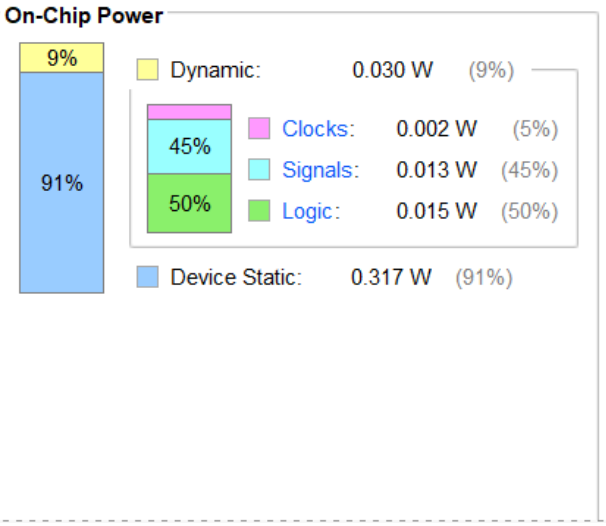
3.5.4 Radix-4 Booth with Wallace tree and KSA



3.5.5 Radix-4 Booth with Row-wise Binary Adder Tree

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

Total On-Chip Power:	**0.347 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	74.5°C (53.6 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

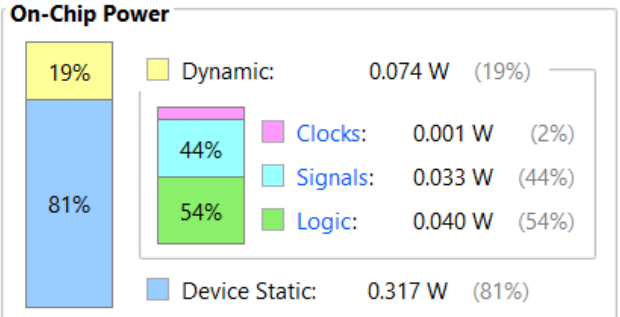


3.5.6 Radix-4 Booth with Dadda tree and Hybrid Adder

Summary

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

Total On-Chip Power:	**0.392 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	74.5°C (53.6 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium



4 Discussion

4.1 Interpretation of the results

Area: #CLB LUTs+#CLB Registers

Delay: CLK-WNS

EDP: Energy × Delay

ADP: Area × Delay

Design	CLK	WNS(ns)	Energy	Area	Delay (ns)	EDP	ADP
Multilevel Carry Skip Adder	2ns	0.402	0.321W	132	1.598	0.513	210.936
Kogge-stone adder	2ns	0.466	0.322W	165	1.534	0.494	253.11
Hybrid CLA/CSLA Adder	2ns	0.451	0.322W	140	1.549	0.499	216.86
Booth with Wallace tree and KSA	3.4ns	0.105	0.431W	1621	3.295	1.420	5341.195
Booth with Row-wise Binary Adder Tree	3.7ns	0.115	0.347W	1132	3.585	1.244	4058.22
Booth with Dadda tree and Hybrid Adder	4.1ns	0.449	0.392W	1679	3.651	1.431	6130.029

In the sample project, the structure of the adder and the multiplier was simply defined as ‘sum = a + b’ and ‘prod = a* b’, which caused Vivado to automatically optimize the various metrics of the adder and multiplier to their best. Therefore, our design did not demonstrate any better performance than the example in any aspect.

As we designed our adder and multiplier using three metrics, minimizing delay, EDP and ADP. When comparing to each other’s adder and multiplier design, the metrics are mostly achieved. In particular, the designs optimized ‘minimizing delay’ consistently have the lowest propagation delay. There is an issue with the EDP metric in the adder designs. The Multilevel Carry-Skip Adder was intended to

achieve a lower EDP, but it ended up with the highest value. A possible reason is that the power differences among the adders are not significant, which might warrant a more thorough investigation of the power simulation process. Although the Multilevel Carry-Skip Adder has the lowest power, its propagation delay is significantly larger than that of the other two designs, which dominates the EDP. Therefore, we conclude that this design succeeded in reducing power. The other problem occurs on the ADP metric in the multiplier, whose value is larger than both its competitors, this might be an indication of poor optimization of the design, since the Hybrid adder used was already larger and slower than the native adder of Vivado which was used in the Row-wise multiplier.

4.2 Possible justification for choosing 25-bit operands.

1. Using a 25-bit width avoids power- of -two sizes, make the designs correctly handle the boundary conditions and evaluate the generality.
2. Using 25-bit can have a better compatibility with existing hardware modules and data formats. Many DSP pipelines use a 24-bit data field with one sign bit (25 bits in total). Therefore, selecting 25-bit operands enables seamless integration with such devices without additional data truncation, re-alignment or conversion overhead.