M-Extension

instr	FMT	opcode	funct3	funct7
mul	R	0110011	0x0	0x01
mulh	R	0110011	0x1	0x01
mulhsu	R	0110011	0x2	0x01
mulhu	R	0110011	0x3	0x01
div	R	0110011	0x4	0x01
divu	R	0110011	0x5	0x01
rem	R	0110011	0x6	0x01
remu	R	0110011	0x7	0x01
mulw	R	0111011	0x0	0x01
divw	R	0111011	0x4	0x01
divuw	R	0111011	0x5	0x01
remw	R	0111011	0x6	0x01
remuw	R	0111011	0x7	0x01

Multiplication

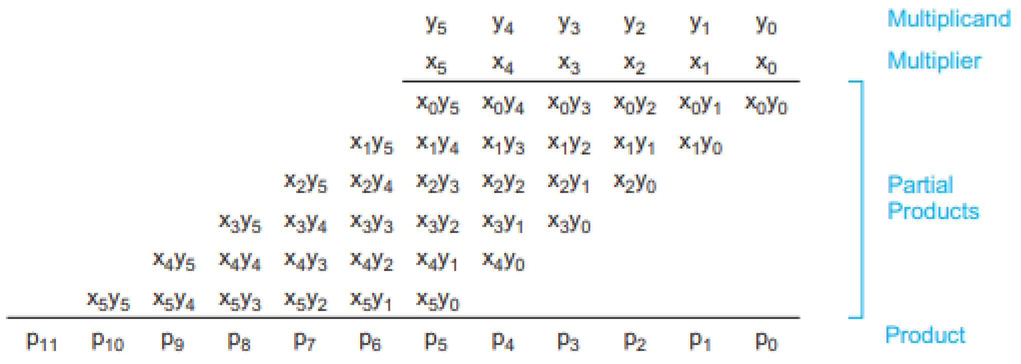
Preliminaries

One of the challenges in VLSI design is to design efficient and fast multipliers, which are the core components of many arithmetic operations.

In general, an N × N multiplier multiplies two N-bit numbers and produces a 2N-bit result.

Signed and unsigned multiplication differ. For example, consider 0xFE × 0xFD. If these 8-bit numbers are interpreted as signed integers, they represent −2 and −3, so the 16-bit product is 0x0006. If these numbers are interpreted as unsigned integers, the 16-bit product is 0xFB06. Notice that, in either case, the least significant byte is 0x06.

M × N-bit multiplication P = Y × X can be viewed as forming N partial products of M bits each, and then summing the appropriately shifted partial products to produce an M + N-bit result P.


Multiplicand
Multiplier
Partial Products
Product
y_5 y_4 y_3 y_2 y_1 y_0
x_5 x_4 x_3 x_2 x_1 x_0
x_0y_5 x_0y_4 x_0y_3 x_0y_2 x_0y_1 x_0y_0
x_1y_5 x_1y_4 x_1y_3 x_1y_2 x_1y_1 x_1y_0
x_2y_5 x_2y_4 x_2y_3 x_2y_2 x_2y_1 x_2y_0
x_3y_5 x_3y_4 x_3y_3 x_3y_2 x_3y_1 x_3y_0
x_4y_5 x_4y_4 x_4y_3 x_4y_2 x_4y_1 x_4y_0
x_5y_5 x_5y_4 x_5y_3 x_5y_2 x_5y_1 x_5y_0
p_{11} p_{10} p_9 p_8 p_7 p_6 p_5 p_4 p_3 p_2 p_1 p_0

There are a number of techniques that can be used to perform multiplication. In general, the choice is based upon factors such as latency, throughput, energy, area, and design complexity.

- Use an M + 1-bit carry-propagate adder (CPA) to add the first two partial products, then another CPA to add the third partial product to the running sum, and so forth. Such an approach requires N – 1 CPAs and is slow.
- Use some sort of array or tree of full adders to sum the partial products (More efficient parallel approach).

The number of partial products to sum can be reduced using **Booth encoding** and the number of logic levels required to perform the summation can be reduced with **Wallace trees.**

Radix 2^r multipliers produce N/r partial products, each of which depend on r bits of the multiplier. For example a 64-bit x 64-bit radix-16 multiplier produces 64/4=16 partial products. Each one depends on 4 bits of the multiplier.

Modified Booth encoding allows higher radix parallel operation without generating the hard multiples by instead using negative partial products.

Proposed Algorithm

To improve the speed performance of multiplication, the number of partial products have been reduced by using Radix-16 Booth Algorithm and for reducing the delay of summation of partial products Wallace Tree Structure has been used. These partial products are summed using a compressor in the structure of Wallace Tree. We used eight 4-2 compressors arranged in 4 levels. CLA has been used for final results where CLA indicates carry look ahead adder that ahead carry of compressor.

Let’s discuss an example of a radix-4 booth encoder for 6 x 6-bit signed multiplication:

A radix-4 multiplier produces N/2 partial products. Each partial product is 0, Y, 2Y, or 3Y, depending on a pair of bits of X. Computing 2Y is a simple shift, but 3Y is a hard multiple requiring a slow carry propagate addition of Y + 2Y before partial product generation begins.

Modified Booth encoding [MacSorley61] allows higher radix parallel operation without generating the hard 3Y multiple by instead using negative partial products. Observe that 3Y = 4Y – Y and 2Y = 4Y – 2Y.

Table below shows how the partial products are selected, based on bits of the multiplier. Negative partial products are generated by taking the two’s complement of the multiplicand (possibly left-shifted by one column for –2Y).

x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	x_{2i+1}	x_{2i}	x_{2i-1}	PP_i
0	0	0	0	1	0	0	-2Y
0	0	1	Y	1	0	1	-Y
0	1	0	Y	1	1	0	-Y
0	1	1	2Y	1	1	1	-0

EX.1:

-25 x -25 = 1100111 x 1100111 >> where: 1 is a sign bit.

The multiplier X is appended by a zero @LSB and sign extended @ MSB

X = 11100111.0

The PPs are extracted as shown:

```
X = 11100111.0     >> 110     >> -Y (2's comp of Y)          >> PP0 = 000000011001
X = 11100111.0     >> 011     >> 2Y (Y << 2)                 >> PP1 = 111100111000
X = 11100111.0     >> 100     >> -2Y (2's comp of 2Y)        >> PP2 = 001100100000
X = 11100111.0     >> 111     >> -0                          >> PP3 = 000000000000
                                                             Result  =  001001110001
```

EX.2:

39 x -25 = 0100111 x 1100111 >> where: 1, 0 are sign bits.

The multiplier X is appended by a zero @LSB and sign extended @ MSB

X = 11100111.0

The PPs are extracted as shown:

```
X = 11100111.0     >> 110     >> -Y (2's comp of Y)          >> PP0 = 000001011001
X = 11100111.0     >> 011     >> 2Y (Y << 2)                 >> PP1 = 000100111000
X = 11100111.0     >> 100     >> -2Y (2's comp of 2Y)        >> PP2 = 101100100000
X = 11100111.0     >> 111     >> -0                          >> PP3 = 000000000000
                                                             Result  =  110000110001
```

EX.3:

-25 x 39 = 1100111 x 0100111 >> where: 1, 0 are sign bits.

The multiplier X is appended by a zero @LSB and sign extended @ MSB

X = 00100111.0

The PPs are extracted as shown:

```
X = 00100111.0     >> 110     >> -Y (2's comp of Y)          >> PP0 = 000000011001
X = 00100111.0     >> 011     >> 2Y (Y << 2)                 >> PP1 = 001100111000
X = 00100111.0     >> 100     >> -2Y (2's comp of 2Y)        >> PP2 = 101100100000
X = 00100111.0     >> 001     >> Y                           >> PP3 = 100111000000
                                                             Result  =  110000110001
```

EX.4:

39 x 39 = 0100111 x 0100111 >> where: 0 is a sign bit.

The multiplier X is appended by a zero @LSB and sign extended @ MSB

X = 00100111.0

The PPs are extracted as shown:

```
X = 00100111.0     >> 110     >> -Y (2's comp of Y)          >> PP0 = 000001011001
X = 00100111.0     >> 011     >> 2Y (Y << 2)                 >> PP1 = 000100111000
X = 00100111.0     >> 100     >> -2Y (2's comp of 2Y)        >> PP2 = 101100100000
X = 00100111.0     >> 001     >> Y                           >> PP3 = 100111000000
                                                             Result  =  010111110001
```

- ## Radix-16 Booth Encoder for 64 x 64-bit Multiplication

In the proposed modified Booth Algorithm, we perform signed multiplication. Multiplier X and Multiplicand Y are appended with a sign bit to specify whether it is a signed or unsigned operand.

For multiplier X, a zero is appended at LSB and sign-extended by 3 bits at MSB then it has been divided into groups of overlapped 5-bits as shown in figure 2.

Each group of 5-bits have been operationed according to modified Booth Algorithm for generation of partial products ±0, ±Y, ±2Y, ±3Y, ±4Y, ±5Y, ±6Y, ±7Y, ±8Y as shown in table 2. The multiples 2Y, 4Y and 8Y can be obtained by simple shifting from Y. The multiples 3Y, 5Y, 6Y and 7Y are the complex multiples of Y. Because 6Y can be obtained by shifting from 3Y, 3Y, 5Y and 7Y are the only complex multiples that need to be sub-generated by 5-bit short adder.

Number of partial products for N x N-bit signed multiplication is $\lceil \frac{N}{r} \rceil = \lceil \frac{64+1}{log_2 16} \rceil = 17\ PP.$

Each PP is appended with zeros at LSB and sign-extended to be of length 2xN.

To perform unsigned multiplication, we append a zero at MSB of the operand to be unsigned and then perform signed multiplication on 65-bit data width as shown in the following piece of code:

```
case (i_mul_in_control)
          MUL:
            begin
              o_mul_in_multiplicand = {i_mul_in_srcA[XLEN-1], i_mul_in_srcA};
              o_mul_in_multiplier = {i_mul_in_srcB[XLEN-1], i_mul_in_srcB};
            end
          MULH:
            begin
              o_mul_in_multiplicand = {i_mul_in_srcA[XLEN-1], i_mul_in_srcA};
              o_mul_in_multiplier = {i_mul_in_srcB[XLEN-1], i_mul_in_srcB};
            end
          MULHSU:
            begin
              o_mul_in_multiplicand = {i_mul_in_srcA[XLEN-1], i_mul_in_srcA};
              o_mul_in_multiplier = {1'b0, i_mul_in_srcB};
            end
          MULHU:
            begin
              o_mul_in_multiplicand = {1'b0, i_mul_in_srcA};
              o_mul_in_multiplier = {1'b0, i_mul_in_srcB};
            end
          default:
            begin
              o_mul_in_multiplicand = {i_mul_in_srcA[XLEN-1], i_mul_in_srcA};
              o_mul_in_multiplier = {i_mul_in_srcB[XLEN-1], i_mul_in_srcB};
            end
        endcase
```

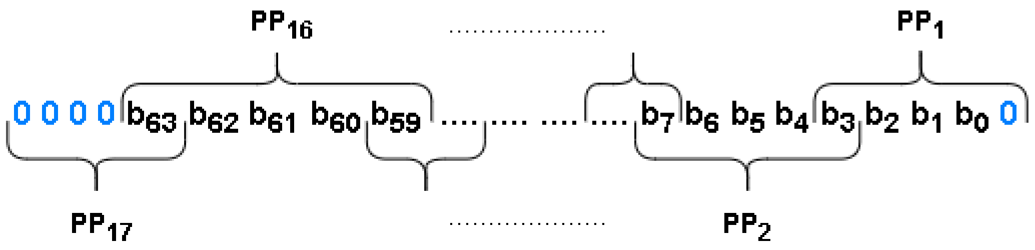

PP16
PP1
0 0 0 0 b63 b62 b61 b60 b59 .... b7 b6 b5 b4 b3 b2 b1 b0 0
PP17
PP2

Figure 2.

x_{4i+3}	x_{4i+2}	x_{4i+1}	x_{4i}	x_{4i-1}	PP_i	x_{4i+3}	x_{4i+2}	x_{4i+1}	x_{4i}	x_{4i-1}	PP_i
0	0	0	0	0	0	1	0	0	0	0	-8Y
0	0	0	0	1	Y	1	0	0	0	1	-7Y
0	0	0	1	0	Y	1	0	0	1	0	-7Y
0	0	0	1	1	2Y	1	0	0	1	1	-6Y
0	0	1	0	0	2Y	1	0	1	0	0	-6Y
0	0	1	0	1	3Y	1	0	1	0	1	-5Y
0	0	1	1	0	3Y	1	0	1	1	0	-5Y
0	0	1	1	1	4Y	1	0	1	1	1	-4Y
0	1	0	0	0	4Y	1	1	0	0	0	-4Y
0	1	0	0	1	5Y	1	1	0	0	1	-3Y
0	1	0	1	0	5Y	1	1	0	1	0	-3Y
0	1	0	1	1	6Y	1	1	0	1	1	-2Y
0	1	1	0	0	6Y	1	1	1	0	0	-2Y
0	1	1	0	1	7Y	1	1	1	0	1	-Y
0	1	1	1	0	7Y	1	1	1	1	0	-Y
0	1	1	1	1	8Y	1	1	1	1	1	-0

Table 2.

- 4-Levels Wallace tree using 4-2 Compressor

To accelerate the speed of the whole adder array, the partial products need to be compressed. 4-2 compressor is the most frequently used component during the compressing of partial products. Its structure is shown in figure 3.

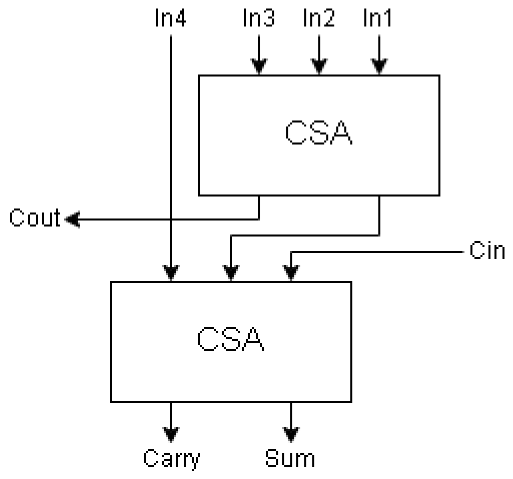


Figure 3.

There are 4 partial product input signals to the 4-2 compressor. They are In1:In4, Cin is the carry input of the adjacent compressor; sum is pseudo-sum; Carry and Cout are carry output, they have the same weights. The independence of input carry and output carry can insure the partial product add separately at the same time.

The four inputs In1, In2, In3, and In4, and the output sum have the same weight. The output carry is weighted one binary bit order higher.

By Radix-16 Booth encoding the 64 × 64-b Parallel multiplier, 17 partial products are obtained. After the compress of the 4-2 compressor, the compressed tree is shown in figure 4.

CLA has been used for final results where CLA indicates carry look ahead adder that ahead carry of compressor.

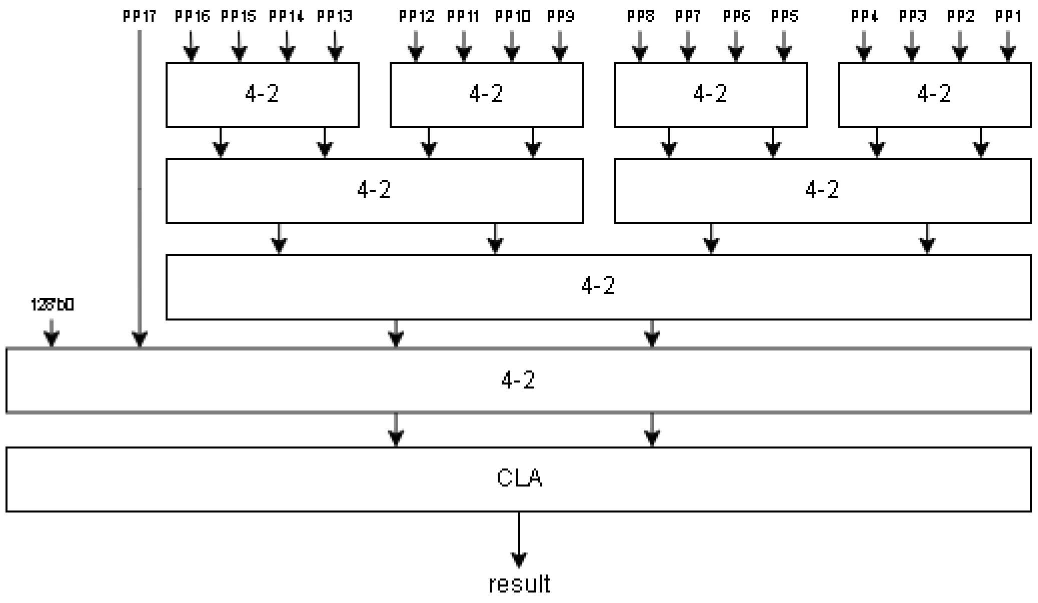


Figure 4.

At gate level, high input compressors are anatomized into XOR gates and carry generators normally implemented by multiplexers (MUX). Therefore, different designs can be classified based on the critical path delay in terms of the number of primitive gates.

The straightforward implementation of a 4-2 compressor of figure 3 has a long critical path delay of 4T. Where T is the delay of one XOR gate.

A 4-2 compressor flattened and optimized at gate level to reduce the critical path delay is shown in figure 5. It is used as a benchmark for evaluating the performance of other low voltage and low-power 4-2 compressor circuits. [3]

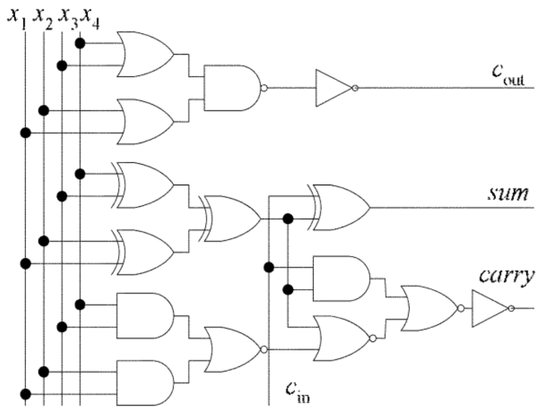


Figure 5.

Carry Look Ahead Adder

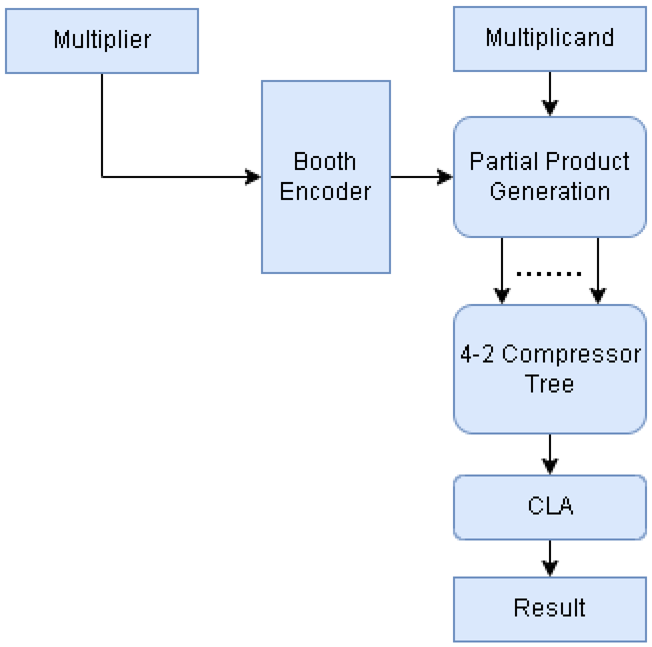
CLA has changed the generate method of serial carry that the high-order carry depends on the adjacent low-order.
Once data A, B involved in the additive arrived, the carry output of each data-bit generated after a series of logical calculations at the same time. The carry and standard sum of each data-bit are unrelated to low-order carry. Therefore, the problem of successive carry is solved, and adding time is not related to the number of digits.

$C_{out} = AB + (A \oplus B)C_{in}$
Let $G = AB$ where G is Carry Generated & $P = (A \oplus B)$ where P is Carry Propagated.
Then $C_{out} = G + PC_{in}$ and can be generalized as $C_i = G_i + P_i C_{i-1}$ where i is an integer.
For 4-bit CLA:
At $i = 0 : C_0 = G_0 + P_0 C_{-1}$ where C_{-1} is C_{in}
At $i = 1 : C_1 = G_1 + P_1 C_0 = G_1 + (G_0 + P_0 C_{-1})P_1$
At $i = 2 : C_2 = G_2 + P_2 C_1 = G_2 + (G_1 + (G_0 + P_0 C_{-1})P_1)P_2$
At $i = 3 : C_3 = G_3 + P_3 C_2 = G_3 + (G_2 + (G_1 + (G_0 + P_0 C_{-1})P_1)P_2)P_3$
Therefore carry-out depends only on current inputs and initial carry-in.

A	B	Cin	Cout
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Steps

1. Zero-extend the multiplicand X to be N+1.
2. Append a zero at LSB and 4 zeros at MSB of the multiplier.
3. Divide the multiplier into groups each of overlapped 5-bits (r+1) as shown in figure 2.
4. Use each group to generate a partial product according to table 1.
5. Compress each 4 partial products using a 4-2 compressor.
6. Use CLA to obtain the final result.



Booth Encoded Compressor Tree Multiplier

Why Radix-16 MBE with Compressor-tree?

From [2]:

- Array multipliers have the most delay and a large power consumption.
- Wallace tree multiplier itself is not that much low area circuit, it can be effectively used along with other designs to increase the performance of the multipliers.
- The modified booth encoder (MBE) has been proven to be one the best multipliers as its power consumption and delay are comparatively less and can be incorporated into another multiplier easily.
- The conventional Vedic multiplier does not give that much improvement in terms of speed, area or power consumption.
- Booth encoded Wallace tree multiplier turns out to be the fastest and it can be optimized effectively to give better performance.
- Table 2 shows the delay comparison of various 32 bit multipliers in nanoseconds.

Array multiplier	72.986
Wallace multiplier	53.198
Bypassing multiplier	62.520
Booth multiplier	55.700
Optimized Vedic multiplier	31.834

Booth Wallace multiplier (radix 2 with CLA)	28.600

Table 2.

From [1]:

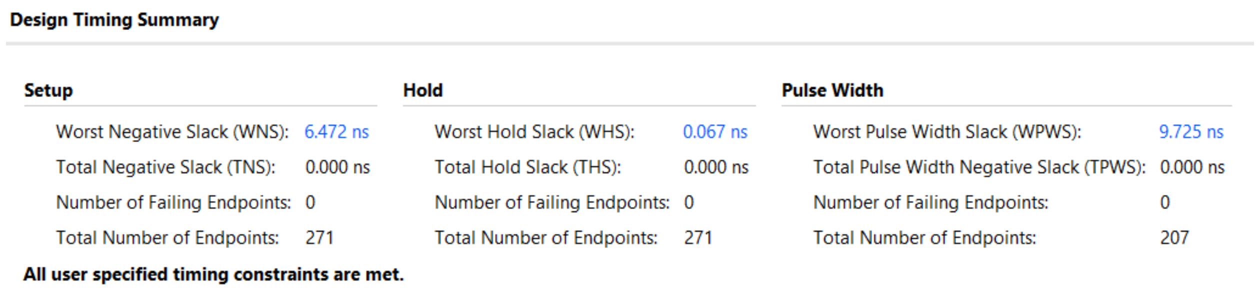
- Table 3 shows the delay comparison of different radix (base) 64-bit multipliers in nanoseconds.

Multiplier	Delay (ns)	Slices used
64x64-bit multiplier using radix-4 and array structure	28.96	14271/69120 (20%)
High speed parallel 32x32-b Multiplier using radix-16 booth encode	7.55	182/2352 (7%)
Signed 64x64 bit multiplier using radix-16 Booth Algorithm	1.8	3347/69120 (3%)
Signed 64x64-bit multiplier using radix-32 Booth Algorithm	1.4	3677/69120 (5.31%)

Table 3.

Our FPGA Implementation Reports

Timing Report:



Design Timing Summary

Setup

Worst Negative Slack (WNS):	6.472 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	271

Hold

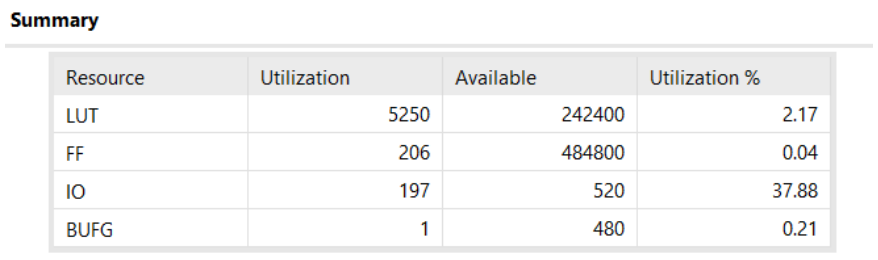
Worst Hold Slack (WHS):	0.067 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	271

Pulse Width

Worst Pulse Width Slack (WPWS):	9.725 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	207

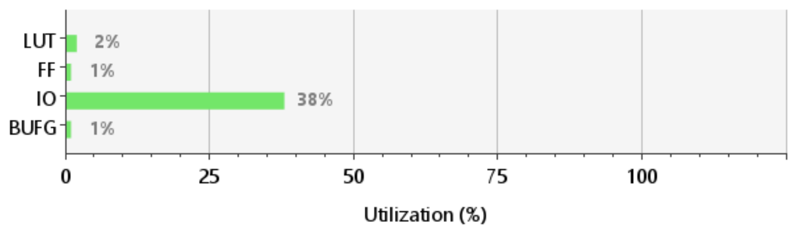
All user specified timing constraints are met.

Utilization Report:

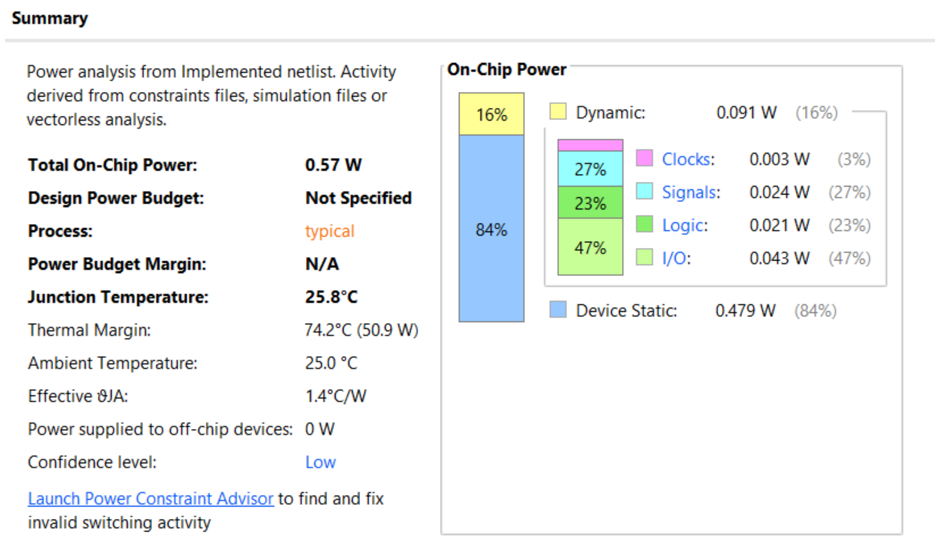


Summary

Resource	Utilization	Available	Utilization %
LUT	5250	242400	2.17
FF	206	484800	0.04
IO	197	520	37.88
BUFG	1	480	0.21



Power Report:



Summary

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

Total On-Chip Power:	**0.57 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.8°C**
Thermal Margin:	74.2°C (50.9 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1.4°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.091 W	(16%)	
Clocks:	0.003 W	(3%)	
Signals:	0.024 W	(27%)	
Logic:	0.021 W	(23%)	
I/O:	0.043 W	(47%)	
Device Static:	0.479 W	(84%)	

References

1. M. Bansal, S. Nakhate and A. Somkuwar, "High Performance Pipelined Signed 64x64-Bit Multiplier Using Radix-32 Modified Booth Algorithm and Wallace Structure," *2011 International Conference on Computational Intelligence and Communication Networks*, Gwalior, India, 2011, pp. 411-415, doi: 10.1109/CICN.2011.86.
2. K. N. Singh and H. Tarunkumar, "A review on various multipliers designs in VLSI," *2015 Annual IEEE India Conference (INDICON)*, New Delhi, India, 2015, pp. 1-4, doi: 10.1109/INDICON.2015.7443420.
3. D. Radhakrishnan and A. P. Preethy, "Low power CMOS pass logic 4-2 compressor for high-speed multiplication," *Proceedings of the 43rd IEEE Midwest Symposium on Circuits and Systems (Cat.No.CH37144)*, Lansing, MI, USA, 2000, pp. 1296-1298 vol.3, doi: 10.1109/MWSCAS.2000.951453.
4. N. Weste and D. Harris, CMOS VLSI Design: A Circuits and Systems Perspective, 5th ed. Pearson, 2011.